Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

Clustering Network Security Data for Efficient Human Analysis

Bachelor thesis

Author:	Jan Svoboda
Thesis supervisor:	doc. Mgr. Viliam Lisý, MSc., Ph.D.
Submission:	May 2024

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Svoboda Jan**	Personal ID number:	**498997**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Cybernetics**		
Study program:	**Open Informatics**		
Specialisation:	**Artificial Intelligence and Computer Science**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Clustering Network Security Data for Efficient Human Analysis

Bachelor’s thesis title in Czech:

Shlukování dat ze síťové bezpečnosti pro efektivní lidskou analýzu

Guidelines:

Network security analysts must process large amounts of data. They often use various domain-specific hand-crafted clustering methods to work with larger interrelated sets of samples, rather than the individual samples. The goal of this thesis is to analyze this practice from more rigorous machine learning viewpoint.
The student will:
• review existing literature on the use of clustering for reducing workload of network security analysts;
• propose a metric usable to compare the quality of clusterings constructed for this purpose;
• implement a method that will cluster network security samples in order to (not necessary directly) optimize this metric;
• find a suitable dataset of network security data, for example accessed URLs;
• compare the implemented clustering method to baselines w.r.t. the proposed metric;
• qualitatively assess the usefulness of the created clustering for manual analysis.

Bibliography / sources:

[1] Zhong, S., Khoshgoftaar, T. M., & Seliya, N. (2007). Clustering-based network intrusion detection. International Journal of reliability, Quality and safety Engineering, 14(02), 169-187.
[2] Xu, K., Zhang, Z. L., & Bhattacharyya, S. (2008). Internet traffic behavior profiling for network security monitoring. IEEE/ACM Transactions On Networking, 16(6), 1241-1252.
[3] Wang, J., Yang, L., Wu, J., & Abawajy, J. H. (2017, May). Clustering analysis for malicious network traffic. In 2017 IEEE International Conference on Communications (ICC) (pp. 1-6). IEEE.
[4] Gu, G., Perdisci, R., Zhang, J., & Lee, W. (2008). Botminer: Clustering analysis of network traffic for protocol-and structure-independent botnet detection.
[5] Rentea, R., & Oprişa, C. (2021, October). Fast clustering for massive collections of malicious URLs. In 2021 IEEE 17th International Conference on Intelligent Computer Communication and Processing (ICCP) (pp. 11-18). IEEE.

Name and workplace of bachelor’s thesis supervisor:

doc. Mgr. Viliam Lisý, MSc., Ph.D. Artificial Intelligence Center FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **10.01.2024** Deadline for bachelor thesis submission: **24.05.2024**

Assignment valid until: **21.09.2025**

doc. Mgr. Viliam Lisý, MSc., Ph.D.	prof. Dr. Ing. Jan Kybic	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature

Author statement

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, 17.5.2024 ....

Jan Svoboda

Poděkování

Děkuji panu doc. Viliamu Lisému, Ph.D. za vedení mé bakalářské práce a za podnětné návrhy, které ji obohatily. Děkuji rodičům a přátelům za podporu.

Jan Svoboda

Název práce:
Shlukování dat ze síťové bezpečnosti pro efektivní lidskou analýzu

Autor: Jan Svoboda

Studijní program: Otevřená Informatika
Obor: Základy umělé inteligence a počítačových věd
Druh práce: Bakalářská práce

Vedoucí práce: doc. Mgr. Viliam Lisý, MSc., Ph.D.
Artificial Intelligence Center FEE

Abstrakt: Práce zkoumá využití shlukování při manuální analýze dat z oblasti síťové bezpečnosti. Výzkumníci často používají ručně vytvořené shlukovací metody k ruční analýze skupin příbuzných vzorků. Navrhujeme metriku pro měření toho, jak dobře se clusterovací algorithmus hodí k podobnému využití. Metrika je prezentována v podobě evaluačního protokolu, modelovaného podle situace, kdy má analytik za úkol ručně klasifikovat čerstvě shromážděná data pro použití na trénování klasifikátoru. Vybrané implementace částí evaluačního protokolu jsou důkladně popsány. Následně je testováno několik algoritmů, které jsou laděny tak, aby maximalizovaly navrhovanou metriku. Nejlépe fungující shlukování je použito na nesouvisejícím datasetu a shluky jsou analyzovány za účelem posouzení úspěšnosti navržené metriky.

Klíčová slova: shlukování, síťová bezpečnost, phishing, škodlivá URL, efektivní lidská analýza , ansámblový klasifikátor

Title:
Clustering Network Security Data for Efficient Human Analysis

Author: Jan Svoboda

Abstract: The thesis examines the use of clustering in the manual analysis of data from network security. Researchers often use hand-crafted clustering techniques to analyze sets of related samples manually. We propose a metric to measure how well a clustering algorithm is suited for similar uses. The metric is presented as an evaluation protocol modeled after a situation where an analyst is tasked to manually label recently gathered data for use in classifier training. The selected implementations of each evaluation protocol part are thoroughly described. Several algorithms are then tested and fine-tuned to maximize the proposed metric. The best-performing clustering is used on an unrelated dataset, and the clusters are analyzed to assess the success of the proposed metric.

Key words: cluster analysis, network security, phishing, malicious URL, efficient human analysis, ensemble classifier

Contents

Chapter 1

Introduction

Hackers can use websites to compromise a user's security by gaining access details (phishing, session stealing), implanting malware into 3^{rd} party websites, posing as trusted entities to commit fraud or otherwise misleading users. Many of these attacks use URLs as the attack vector, presented to the end-user with the intent to be opened. Correctly classifying URLs (and predicting threats) helps defend from these attacks. Criminals constantly develop new approaches to escaping detection or exploiting vulnerabilities, making malware detection harder. The phenomenon of older data and models being obsolete because of changing trends is called **concept drift** [1].

Because of the severe concept drift present in cybersecurity, using an older dataset (more than several months old) for classifier training leads to mediocre results. The outdated training dataset may not contain information about novel exploits and obfuscation methods. We may need to gather recent data to have an up-to-date training dataset. More work is needed to obtain the correct labels for supervised learning, since the data is usually unknown.
An **analyst** can label a single data instance by examining its behavior and assigning class accordingly. Large volumes of data make this process extremely time-consuming. We can instead cluster the data, then classify the whole cluster at once by examining the common traits of the data [2].

Goal of the thesis In this thesis, we analyze the practice of using clustering to decrease the workload of analysts from a systematic standpoint. The goal is to find a metric to assert clustering's usability for this purpose and then find an algorithm that maximizes this metric on selected data from network security.

Proposed metric Our metric is based on the described situation. Let us imagine that we have gathered very recent data, but we need to label some of it ourselves with the help of clustering algorithms. First, the data is clustered. *Some* clusters are shown to the expert for manual analysis. The labeled subset of the data is used to train a classifier.

The classifier is then tested, and the resulting metrics (accuracy, F1-measure) are treated as an indication of the clustering's success, acting as our metric. This metric does not directly capture "usefulness" or "goodness" but rather the usability of the labeled data in a classification setting. We call the whole process the "evaluation protocol."

Modeling the situation Expert analyst attention is a scarce resource, so only a limited number of clusters are presented for labeling. This requirement poses an additional challenge in selecting the best clusters to maximize the proposed metric. One explored selection method uses an out-of-date dataset to gain information about clusters. The dataset would perform poorly if we used it for training directly. We can, however, gain some information about clusters by using an ensemble classifier in a specific manner. We use classification confidence in addition to predicted classes to estimate the label of a cluster.

Thesis structure We review relevant literature and research viable models to use during each step of the evaluation protocol in Chapters 2 and 3. In Chapter 4, we discuss our approach to hyper-parameter tuning and show obtained results. Relevant results are the best performing clustering algorithm and qualitative analysis of extracted clusters from previously unseen dataset. We interpret the results and conclude the thesis in Chapters 5 and 6.

Choice of data We investigate URL addresses as a specific type of network data. Paper [3] clusters malicious URLs from "open source threat intelligence feeds" to discover hidden structures in the data. In [4], the authors use lexical features to train classifiers on URL data.
Other types of network data that were considered for the thesis are captured network traffic [2] [5] [6], stored **cross-site scripting** code [7] and **DNS** queries [8].
URLs (as short text data) are more easily interpretable than other mentioned data types, and even non-experts can immediately see similarities between clustered data. In addition, it is easy to find practical and relatable uses for efficient malicious URL detection in web browsers and email clients, preventing users from clicking on malicious links.

Scope of AI use in the thesis We want to disclose all use of AI-powered applications in this thesis. ChatGPT was used in the very early stages to discuss broad ideas and during implementation to generate code for basic tasks like data visualization and loading files. During the final edit, Grammarly was used to correct grammar and clear some formulations.

Chapter 2

Preliminaries

2.1 Clustering

Clustering refers to a diverse group of algorithms for data analysis. The data is organized into groups based on some criterion, usually expressing a similarity between instances. Data in the same group (cluster) tend to be more related to each other than to data in other clusters [9].
Most of the algorithms operate on metric space called **feature space** ($\mathbb{R}^D$), where each data point is represented by a vector ($\mathbf{x} \in \mathbb{R}^D$). In this case, more "related" points are closer to each other as judged by distance function (metric).

This section explains used algorithms and discusses some interesting uses, particularly in classification pipelines. An overview of the algorithms mentioned can be found in [10] and [11].

Terminology **Centroid-based** clustering measures similarity by distance to a center of each cluster. **Density-based** defines clusters as areas of high point density in the feature space. **Distribution-based** algorithms fit probability distributions into the data to maximize the likelihood. **Connectivity-based** models build connections between points based on distance. This nomenclature is particularly useful, highlighting the major differences between used algorithms.

2.1.1 Usage of clustering

Clustering algorithms are considered **unsupervised** because they do not need labeled data to work. Clustering is used for diverse tasks in exploratory analysis, visualization, feature reduction, and classification pipelines.

[12] use clustering for feature reduction. Clustering-based color quantization aids compression algorithms [13][14]. In psychology, hierarchical clustering (in 2.1.5) is used to find "meaningful subgroups" of cases [15][16]. Biologists use clustering to analyze "DNA microarray" data [17]. Density-based clustering can detect outliers, which is used in [18] for meteorological research.

2.1.2 Using clustering for classification

Clustering algorithms can be utilised for **classification** in different ways. A straightforward approach is to find clusters of training data and then use the known labels of the data to assign a class to each cluster (for example, by majority vote). New, unseen data is classified as the cluster it would belong to or the closest in the feature space. This is a **supervised** usage of clustering, which was investigated by the author in a semestral project before writing this thesis.

Another usage in classification, more relevant for this thesis, is **unsupervised**. First, the unlabeled data is clustered. Then, the individual clusters are classified (by one of different means). This is useful in many situations where the data labels are hard to obtain and supervised techniques cannot be used.
In the situation we are modeling, a cybersecurity analyst examines data from each cluster to label it. This is faster than classifying every data independently. [19] propose a clustering algorithm specifically for URL data that produces a set of centroids of each cluster. In their model, they choose those centroids for analysts to examine. The proposed algorithm in [19] ensures that every data in the cluster is similar to some chosen centroids. In this case, the chosen data does represent all of the data in a cluster, to some extent.

We can assign the class to a cluster by mechanisms other than manual labeling. For example, matching a cluster to a previously known malicious sample and processing the whole cluster as malicious as in [5]. The authors of this paper implant known malicious traffic into the unlabeled data and then observe to which cluster the particular data belongs. The authors of [2] propose a "self-labeling heuristics" to automatically assign labels to data based on position inside a cluster. This paper has inspired our initial approaches for selecting clusters for manual classification.

2.1.3 Centroid based clustering

Centroids are vectors from feature space, representing the cluster centers. These centers are updated iteratively to minimize an objective function. Better solutions can be found by re-running the algorithm with different initial conditions.

K-means

A prime example is algorithm K-means [10][20], a centroid-based iterative algorithm with **strict partitioning** (one data is in exactly one cluster). **K** in the name refers to the number of clusters this algorithm produces, which is set beforehand as one of the parameters.

K-means uses L2 norm as metric to evaluate distance between points to find set of centroids C^* (2.1) and centroid assignment L^* (2.2) that minimize the objective function known as **Inertia** (2.4) (referred to as **Sum of Square Error** criterion).
The approximate solution is found by iteration (2.1.3). K-means converges to a local minimum in a finite number of steps. For every set of centroids, the feature space is divided into convex subsets of points to which a particular centroid is the closest.
Centroid assignment (2.2) can be thought of as a function $L : T \to C$, assigning $c_j \in C$ to

every $x \in T$, where T denotes the training set (2.3). l_i is then $c \in C$ closest to x_i.

K-means problem formulation

$$C = \{c_j\}_{j=1}^{K}, \quad c_j \in \mathbb{R}^D \tag{2.1}$$
$$L = \{l_i\}_{i=1}^{N}, \quad l_i \in C, \quad l_i = \arg\min_{c \in C} ||x_i - c||_2 \tag{2.2}$$
$$T = \{x_i\}_{i=1}^{N}, \quad x_i \in \mathbb{R}^D \tag{2.3}$$

$$I_{C,L} = \sum_{i=1}^{N} ||x_i - l_i||_2^2 \tag{2.4}$$
$$C^*, L^* = \arg \min_{all\ C,L} I_{C,L} \tag{2.5}$$

K-means iteration steps

1) Initialise C_0.
2) $\forall x_i \in T : l_i = \arg\min_{c \in C} ||x_i - c||$.
3) Calculate C_t as the average of points assigned to each cluster.
4) If $C_t \neq C_{t-1}$, goto step 2, otherwise end.

In every iteration of the K-means algorithm, $N * K$ distance computations are made. Worst case complexity over it iterations is $O(it \cdot N \cdot K)$ [21] .

Initialisation In step 1 of the K-means iteration above, there is freedom in choosing initial centroid positions. A commonly used algorithm, which we use exclusively in this thesis, **K-means++**, chooses the first center uniformly from T, then chooses the next centers with probability proportional to d_i^2 where

$$d_i = \min_{c \in C} ||x_i - c||_2 \quad while\ |C| \leq K \tag{2.6}$$

Other standard methods are **random** and **Forgy** [22].

Modifications There are ways to improve the K-means algorithm in convergence, computation complexity, memory requirements, or escaping local minima [21].
Notable modification is **minibatch** K-means, which uses small randomly sampled batches of data with diminishing step size to an approximate solution. This can be seen as a minibatch gradient descent. It is shown to perform better on some datasets (in terms of inertia I) and is generally faster on large-scale data [23] [24]. A minibatch version of K-means solves a problem with limited memory since the batches can be loaded sequentially. Many algorithms can be seen as extensions of basic K-means (**K-medians**, **K-medoids**, **Fuzzy C-means** [21]), but these are not explored in the thesis.

2.1.4 Density based clustering

Unlike K-means, density-based algorithms [25] can find arbitrarily shaped clusters because they separate denser regions of feature space from less dense **border** regions. Points in those sparse regions are labeled as outliers.

DBSCAN

DBSCAN is short for "density-based spatial clustering of applications with noise. "DBSCAN requires setting radius ϵ to find "dense" areas around data [26]. The algorithm is parameterised by the choice of ϵ and minimum number of points in a neighborhood (min_pts).

We can see similarities between DBSCAN and density estimation via Parzen window method with uniform kernel. Both methods approximate density by counting number of datapoints inside a symmetric neighborhood of each point. DBSCAN algorithm does not approximate density through the whole feature space but only finds **core points** that have more than min_pts points in their ϵ neighborhood. Core points in each other's ϵ neighborhoods are connected into a cluster. In the next step, every point *directly* reachable from this cluster **core** is added.

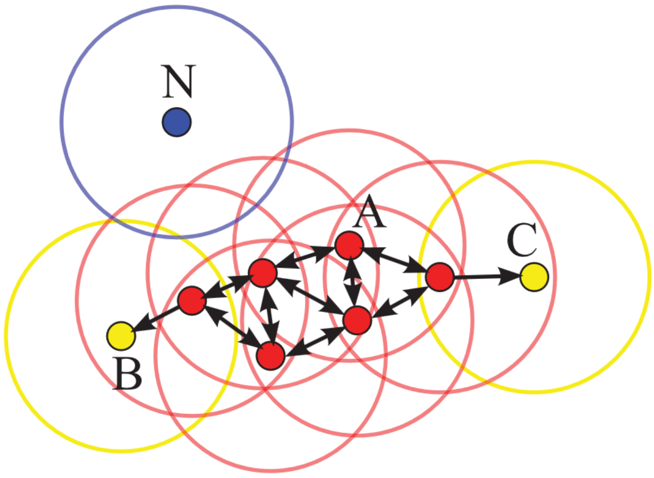


Figure 2.1: DBSCAN scheme from [26]

In Figure 2.1, we see a schema of the algorithm working on a small sample. This image was taken from [26]. In this case, min_pts is set to 2, and the ϵ parameter is not specified. The neighborhood of every point is denoted by a colored circle around it. Core points are colored red. Both points B and C are density reachable from A. N is not reachable from the core. B and C are called **border points** since they do not have enough neighbors to become core points but are connected to a cluster core. N is an **outlier** and is not added to the cluster.
Object 1 is **density reachable** from 2 if a chain of core points connecting 1 and 2 exists (including the starting point). C is density reachable from A, but not A from C since C is not a core point.

The algorithm's worst-case complexity is $O(n^2)$ [26]. The algorithm can be sped up by using kd-trees, ball-trees, and other indexing structures, improving the *average* complexity to $n \cdot \log n$ [26].

While the core detection is always deterministic, **contested** border points (density reachable from more cores) are added to different clusters depending on implementation. Used implementation (sklearn [27]) connects the border point to the first legible cluster it finds and is therefore determined by point ordering. DBSCAN is very sensitive to the choice of parameters, which are hard to set [26], which we have also observed.

Other density-based algorithms often use DBSCAN as the base model to improve. **HDBSCAN** is a high-density version of DBSCAN. This algorithm does not add border points to the cluster cores and can extract clusters of varying densities while being efficient in high-dimensional space. **OPTICS** uses reachability between points to find clusters and can detect clusters of varying densities. It uses the same notion of core points and reachability.

OPTICS

OPTICS, short for "ordering points to identify the clustering structure," can be seen as a generalized DBSCAN algorithm that relaxes the requirements for ϵ. OPTICS allows ϵ to be in a range of values instead of being set precisely. The idea is that not one "global" ϵ can extract all underlying structures from the data, and different densities may need to be found in different regions of feature space [28]. Instead of generating clustering information for one instance of DBSCAN, OPTICS simultaneously extracts this information for all ϵ below max_ϵ.

Instead of extracting DBSCAN results for one ϵ, the algorithm encodes density into reachability distances, which is used to judge cluster membership. OPTICS works by ordering the density-reachable points by the **reachability distance** (eq. 2.7) from a previously processed point. Clusters are then extracted from this ordering.

$$reach\_dist(A, B) = \max\{core\_distance(A), distance(A, B)\} \tag{2.7}$$

$$core\_distance(A) = distance\ to\ min\_pts^{th}\ nearest\ neighbor\ of\ A \tag{2.8}$$

This distance can be interpreted as ϵ_i that would make B directly reachable from A if A was a core point (cannot be below $core\_distance(A)$). Note that the processing of points starts from points with the smallest reachability distance between them.
Ordered points and their reachability distance can be encoded into **reachability plot**, sometimes called **OPTICS plot** with ordered points on the x-axis and reachability distance on the y-axis.
"Valleys" of this plot signify denser areas extracted as clusters. Steeper areas show a significant drop in density. Valleys are then extracted from the plot to find the clusters. This can be seen in Figure 2.2, where the extracted clusters are colored.

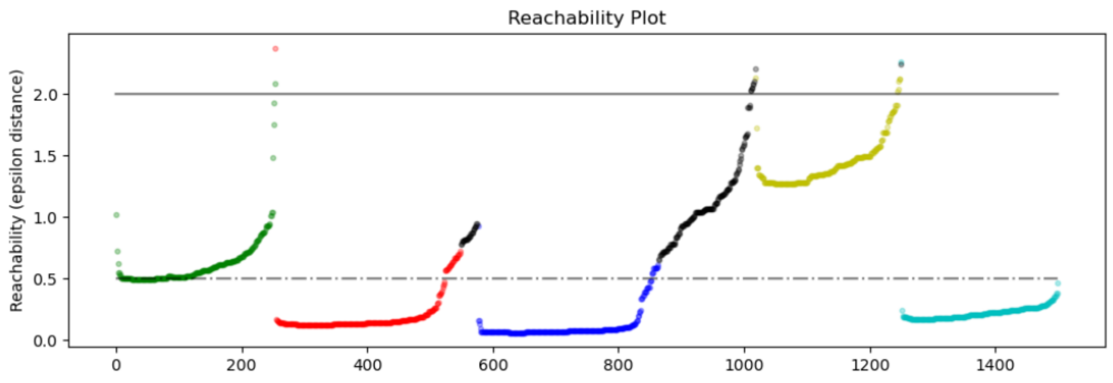


Figure 2.2: OPTICS reachability plot taken from [29].

Two ways to extract the clusters are used. **First**, the original paper [28] proposes a DBSCAN-like cluster extraction technique, which "cuts" the graph on the y-axis at ϵ and finds an uninterrupted range of points, which is extracted as a cluster. This can speed up repeated DBSCAN extractions but is not the intended use of OPTICS, according to [28]. The **second** extraction technique automatically recognizes clusters delimited by **steep** areas of the plot. In Figure 2.2, clusters were extracted by this automatic method, with outliers colored black. The yellow cluster has a significantly lower density (judged by the reachability distances between ordered points) but is still extracted as a cluster. [30] propose a predecessor correction algorithm, used by default in the used sklearn implementation. Without it, points from the steeper areas of the OPTICS plot are sometimes incorrectly added to an unrelated cluster. The authors also show that while OPTICS has complexity of $O(n^2)$, with indexing structures like kd-trees, *average* complexity can be closer to $n \cdot \log n$, similarly to DBSCAN [26].

2.1.5 Hierarchical clustering

Previous techniques could not differentiate finer structures inside clusters (sub-clusters and their sub-clusters). **Hierarchical clustering** (also called **connectivity based**) works either from bottom-up (**agglomerative**) or top-down (**divisive**)[20][11]. Hierarchical algorithms create connections between clusters based on their distance from each other. A produced binary tree can be visualized as a **dendrogram** with individual points in the leaves and nodes representing sub-clusters. The number of clusters extracted is determined by the depth of the tree.

Agglomerative clustering

Basic agglomerative clustering (bottom-up) works on a simple principle, as seen in the algorithm steps below. Two closest clusters are joined, and the new node is added to the **binary merge tree**. The distance between 2 clusters is called a **linkage distance**.

The agglomerative algorithm consists of the following steps:

1) Every point is a cluster (leaf).
2) Find 2 clusters with the smallest linkage distance
3) New cluster is the conjunction of those 2 clusters
4) Create a new node in the merge tree above the two children
5) repeat from 2) until all points are in 1 cluster

In Figure 2.3, [31] have clustered car models by numerical features pertaining to the technical details of the cars (e.g. power, number of cylinders). The height on the y-axis is the connection length between the clusters as the clusters were merged in the feature space.

Linkages Linkage distance LD can be seen as a measure of distance between clusters. Simplest LD is called **single-linkage** and is just the minimum of (arbitrary) distance d between all pairings of points, such that the points are from different clusters. To illustrate how linkage typically works, in eq. 2.9 we can see single linkage distance between clusters A and B, using distance function d.

$$single(A,B) = \min d(x,y), \quad x \in A, \quad y \in B \tag{2.9}$$

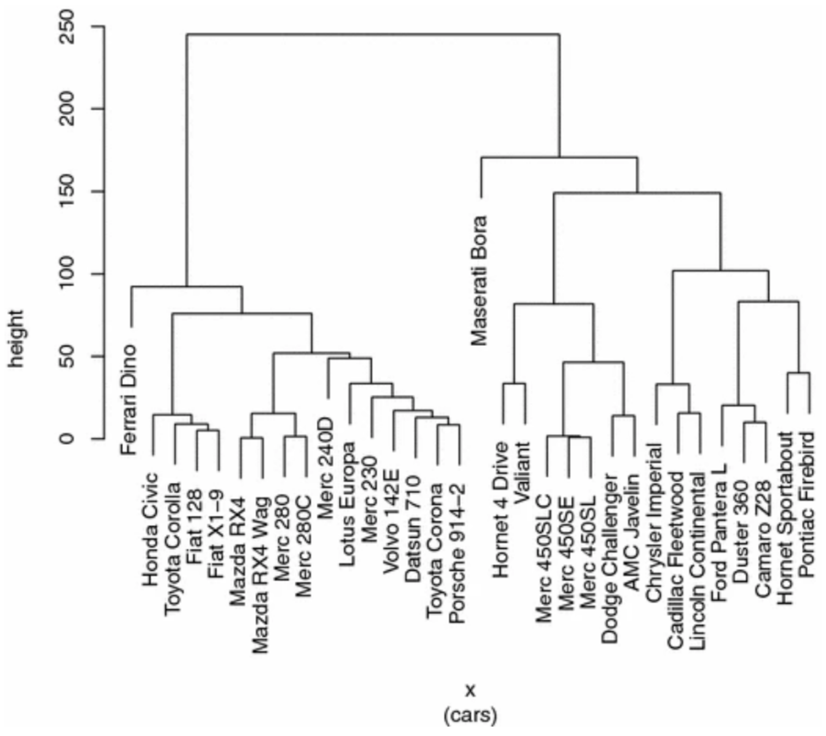


Figure 2.3: Dendrogram formed by clustering car models. Taken from [31]

Complete linkage takes *maximum* of distances between pairs instead of minimum. This prevents a known problem with a single linkage, where two well-defined clusters can be connected by a chain of very close points while being very far away in the feature space. **Average linkage** finds the distance between clusters as averages between all possible pairs. The algorithm is versatile because it can use different distance functions and linkages almost unchanged. **Centroid linkage** measures the distance between a centroid, which is an average of the cluster's points. **Ward linkage** computes the increase in inertia (eq. 2.4 in 2.1.3) if two clusters were merged and chooses the smallest increase.

At the end, the dendrogram is cut to obtain a specified number of clusters. The OPTIC's complexity is $O(n^2)$, but since the algorithm is very similar to many graph-based algorithms, the computation is well understood, and many efficient (or parallel) implementations exist.
Agglomerative algorithms can be used on data preprocessed by other clustering algorithms. BIRCH uses an algorithm to compress the dataset into smaller amount of representative samples, which are then clustered by the agglomerative algorithm to obtain a set number of larger clusters.

BIRCH

BIRCH, short for "balanced iterative reducing and clustering using hierarchies", uses agglomerative clustering in one of its phases in addition to a clever tree-building algorithm. The tree-building algorithm uses a reduced cluster representation, known as **clustering feature** (CF). A cluster's CF is a triple consisting of the number of points, the sum of points, and the sum of squares of points belonging to the cluster (in eq. 2.10, where CF of cluster A is described). When clusters are merged, CFs are added to describe the new cluster.

$$CF(A) = (N, \mathbf{L}, S), \quad N = |A|, \quad \mathbf{L} = \sum \mathbf{x}, \quad S = \sum \mathbf{x}^T \mathbf{x}, \quad \mathbf{x} \in A \tag{2.10}$$

CF tree is a balanced tree described by **branch factor** B and **clustering threshold** T. Every node is a cluster of its own characterized by a CF record. Leaves are filled with the actual data. Nodes of this tree have at most B children. This tree is built dynamically by adding new data to the appropriate clusters and updating CF recursively. CF is used during the construction of the tree to recursively find the closest child node based on a distance function and to resolve balancing issues caused by splitting a leaf node. When the data is placed into a leaf, a change in the CF is propagated up the tree. All nodes in the leaf have to "fit" inside the clustering threshold diameter T; otherwise, the leaf will be split. CF tree is a type of B+ tree. This explanation is loosely interpreted from the original article proposing this method [32].

Because of the requirement for a limited number of sub-clusters/child nodes, one actual cluster might be split between multiple leaves. This is remedied by the final agglomerative clustering, which can merge similar clusters.
BIRCH linearly scans the data and then adds it to the CF tree, making it fast on larger datasets with a complexity of $O(n)$. The (optional) global clustering step has the $O(n^2)$ complexity since agglomerative methods are usually used.

2.2 URL

Uniform resource locator (URL) is a string used as a reference for a resource inside a computer network and a primary mechanism for accessing websites. The structure of a URL is defined in [33]. The URL begins with a string specifying **scheme**, for example **’http:’**, **’https:’**, **’ftp:’** or **’mailto:’**.
The scheme is the type of command/protocol that is used to communicate with the respective source/service and dictates a specific structure to the URL. Only http and https schemes denote the respective communication protocols [34] are used in our dataset. These protocols can be omitted when using a browser since the http or https connection is implicitly established. Both protocols are built on TCP protocol, the difference being that https encrypts the communication between the server and the computer. The structure of **httpurl** (a specific URL type to fetch documents from the internet) is shown in 2.11. From this point on, we only talk about hhtpurl when referencing URLs.

$$httpurl = http : //hostname : port/path?parameters\#fragment \tag{2.11}$$

Hostname in (2.11) can be subdivided into domains, which are strings separated by ’.’. Domains are used to find the website’s IP by querying a DNS server. The subdomain most on the right-hand side is called a **top-level domain** (TLD). Most URLs do not specify a port unless connecting to an IP address directly.
The next part of the URL after hostname (and optionally port) separated by ’/’ and before ’?’ is called **path** and consists of "sub-paths" divided by ’/’. The path is used for navigation inside the server to which the browser is connecting. Websites have a tree-like directory structure, which can be abused by some exploit techniques (local file inclusion).
Query is part of the URL after ’?’, with the individual **parameters** delimited by ’&.’ Query is used to send additional information to the website, like filters, pagination, and

search queries (e.g., on e-shops and when browsing content). Fragment (after '#') is a part of the URL used for navigating the website structure and is not used to communicate with the web server.

Escape is a combination of '% hex hex' where hex is a hexadecimal number . When using an escaped URL, the string is decoded to a UTF-8 string in the browser. For example, '%21' would be decoded to '!'. Escaping URLs is sometimes used to obfuscate the address to make it harder to read by a human [4]. A more legitimate use is to encode Japanese, Chinese, or Arabic characters into URL-compliant form. Therefore, all uses of escape cannot be seen as malicious.

2.3 Cyber threats using URL

There are many types of threats using URLs, that we need to guard against. Criminals use the web as an environment to spread malware and spyware of various purposes and types, trick people to steal their money, and get access to sensitive data or resources.

Downloading Malware Hackers can embed malicious links in legitimate websites (like forums and comments) to lead users to malicious content. If a website downloads malware to the user's device without consent, it is known as **drive-by download** [35][36]. A way to do this is to get the browser to open a fabricated URL by **redirecting** the user from the compromised websites. Downloaded executables range in effects. Significant categories are **viruses, worms, Trojans, ransomware, keyloggers** with many more sub-categories existing [37]. Another way to use URL is to **push** the compromised link to the user via email and search engines to redirect user traffic to hacker-owned websites.

Injections **Code injection** exploits use URLs and associated path (2.11) to inject code into the execution of otherwise safe action in the browser. When performing **cross-site-scripting** (XSS), attackers design scripts that are executed when fetching the website in the browser. **cross site request forgery** (XSRF) changes parameters in the URL to make the page do actions unfavorable for the user (send money or sensitive information through email, etc.). Other injection mechanisms include embedding code into webpages, which is run when visited (stored XSS). Additional description of XSS attacks in [7].

Scams **Scam** is a type of cybercrime that does not exploit vulnerabilities and programming tricks to harm but uses manipulation, **social engineering**, blackmailing, and clever tricks to commit fraud and damage users. A lot of **scam** websites try to appear legitimate to fool users into sending money, entrust them with credit card information and other sensitive data, or trick them into downloading and running malware. There are many types of scams; some common are **phishing, Nigerian prince scam, romance scam, lottery scam, sextortion (sex extortion), false giveaways, technical support scam**. Some approaches use URLs to redirect users to fake websites or use misleading links. **Phishing** is a specific type of scam, "directing users to fraudulent websites" [38] and posing as a trusted entity. One way to appear legitimate is to have a hostname similar to a known website (and the same visuals). This is known as **Cyberquatting**, when the malicious website has a name very similar to a known company or vendor, usually abusing similarity between some ASCII characters (like a and α) or typos that users often make (**Typosquatting**).

Botnets and Cryptominers **Botnet** is a set of "zombie" computers infected by **trojan horse** malware that appears to function normally but covertly follows orders from the hacker, who transmits commands through **command and conquer** server (C&C) [39]. Some of the C&Cs communicate through webservers (called "rendezvous points") by sending the computer a list of commands. The rendezvous points are reached by URL, generated by **DGA** (**domain generation algorithm**) [40]. Domain generation makes it hard to filter out and shut down all servers since new ones can be created quickly. Botnets can cause **DDoS** attacks, send spam messages, covertly mine cryptocurrencies (**cryptominers**) and aid in other illegal activities[5].

URL Obfuscation **Obfuscating** means changing the URL (or code) structure to circumvent filters, blacklists, and escape detection by the authorities and antivirus software vendors. A common (and easy to detect) way is to **encode** parts of the URL to different charset or number "base" (like hexadecimal or base 64). Another popular method is to encode some characters as escaped (section URL -2.2), writing it as $\%$ *hex hex*. URL can be encoded multiple times to further hide the intent and evade detection. Escaped addresses cannot be easily read by the user and require further work from the filtering software. [41] identified four prominent obfuscation types listed below.

Type I: Obfuscating the Host with an IP address.

Type II: Obfuscating the Host with another Domain.

Type III: Obfuscating with large host names.

Type IV: Domain unknown or misspelled.

[4] distinguishes obfuscating the hostname (using IP addresses or domain prefixes) and obfuscating the path of the URL (escaping, encoding, and redirecting).

2.4 Machine learning models

Clustering is considered to be **unsupervised** learning since the algorithm does not use the class of the training data. **Supervised** methods rely on a fully labeled dataset to learn discriminative functions. Examples of commonly used models are **logistic regression**, **K-NN**, **random forest**, **gradient boosted trees**, **SVM** [20] and **deep neural networks** [42]. There are other useful methods besides ML classifiers to be used on network data, like **blacklisting**.

2.4.1 Neural networks

A "stronger" classifier we decided on is an **artificial neural network** (ANN or just NN). Derivations and in-depth explanations are found in [42]. A neural network is composed of simple units called **neurons** that compute an affine transformation of input on which a non-linear function is applied. Historically, *tanh* or *sigmoid* (eq. 2.16) were used, while more recently $ReLU$(2.13) and its variants (Leaky ReLU, Parametric ReLU) are the norm. Neurons are organized into layers, in which all units take the same input at one computation pass. Linear layers are realized by matrix multiplication, where matrix **W** holds the weights of each neuron as columns.

$$y = f(\mathbf{W}^T\mathbf{x} + b), \quad f = \text{activation function} \tag{2.12}$$

For classification tasks, the very last layer of neurons has as many neurons as there are classes, and the very first as the number of input features. In Figure 2.4, a net with three input dimensions and two classes is visualized. The layers in-between are called **hidden layers** and are omitted from the visualization since there can be multiple or none with arbitrary width. The last nonlinearity is a **softmax** layer (2.14), which applies a softmax function producing a normalized vector y_i for every input, containing distribution over all classes. The predicted class is $\arg\max y_i$ for individual input data. The variant of ANN

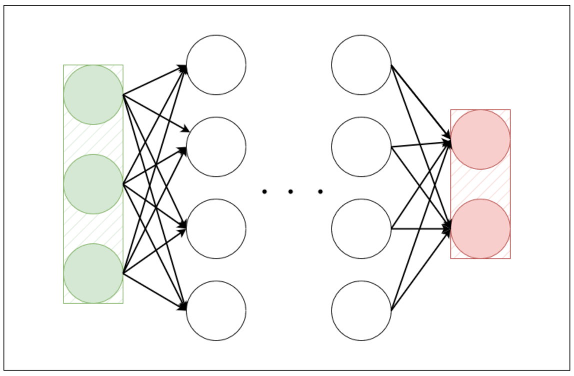


Figure 2.4: NN scheme with neurons and layers connected from left to right

we are using has several layers of neurons and uses the ReLU activation function. This architecture is called **feed-forward neural network**.

$$ReLU(x) = \max\{0, x\} \tag{2.13}$$

$$y_i = softmax(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{C} e^{x_j}} \quad \mathbf{x} \in \mathbb{R}^C \tag{2.14}$$

Feed-forward network could be thought of as many simpler functions composed together applied on the input. The learning process is derived from this fact. **Gradient descent** is a first-order optimization method that is used to find local minima of *differentiable* function F by performing iteration (eq. 2.15). α in eq. (2.15) denotes the step size. The derivative of ReLU at 0 is commonly defined as 0 or 1 to make the function differentiable [42].

$$x_{t+1} = x_t - \alpha_t \nabla F(x_t) \tag{2.15}$$

In usual NNs, the gradient is computed for **batch** of data at a time, and weights are updated afterward. Learning on the whole training set separated into batches is called **epoch**. When computing gradient, application of **chain rule** is used to find derivative for weights in a specific neuron. The error "propagates" backward from the end of the network and updates each layer's weights accordingly. This is called **back-propagation**. **Loss function** is used to measure the error of prediction against a true class of the data with mean square error (**MSE**) and **cross-entropy** being commonly used. NN performance can be improved by using different step size schedulers, optimizers (changing the exact direction of gradient descent), using momentum, changing the network depth and width, or starting with different step sizes. Sophisticated weight initialization is often performed to "kick-start" the learning process.

Dropout NNs have a problem with **overfitting**. This presents as a network producing good results on train data but poor results on new, unseen data. The network is **overtrained**, as it has learned weights for flawless classification of training data but generalizes poorly. A special layer, **dropout**, randomly masks each connection between two layers with probability $p \in \langle 0, 1\rangle$ given as input (visualization in Figure 2.5) to prevent overfitting [43]. This layer is turned off for inference, and every connection is used to classify new instances. Technically, dropping random connections is called **dropconnect** while dropout originally referred to masking whole neurons, but dropconnect is a generalization of dropout and is used in literature interchangeably [44].

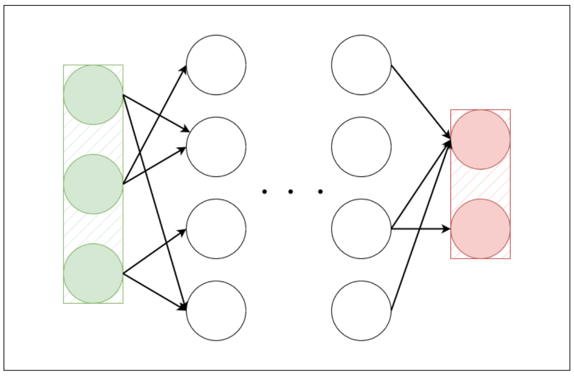


Figure 2.5: NN with some connections dropped ($p = 0.5$)

Convolutional neural networks (CNN) are used for classifying images, time series, and other data with local spatial or temporal structure. [45] show how CNN can classify URLs directly without training the embedding first. CNNs were briefly considered but are not used in the thesis.

2.4.2 Logistic regression

In order to have a context for classification results during the evaluation of individual clustering algorithms, we will use another classifier. **Logistic regression** (logreg) is a **linear** classifier and, therefore, has a lower discrimination capability than NN. Logistic regression can be considered ANN without any hidden layers (just input and last layer) with softmax activation function, minimizing cross-entropy loss (corresponding to maximizing likelihood). Since a single matrix of weights is updated, gradient computation is faster than in NNs.
This is not the only interpretation of logistic regression. The binary classifier can be derived from observing the likelihood ratio in Bayesian classification tasks and then generalized to the multiclass settings. The binary classifier used a sigmoid function as nonlinearity (eq. 2.16).

$$\sigma(x) = (1 + e^{-x})^{-1} \tag{2.16}$$

2.5 Features and data representation

To be able to learn various discriminative methods, the data must be assigned an interpretation in the feature space, $\mathbf{x} \in \mathbb{R}^D$. Individual dimensions coincide with the features chosen. Initially, we can use the knowledge of the data and context to "engineer" the features ourselves. For example, one could assume that the URL length is important

for classification and choose it as a feature (it is then computed for every address).
Representation learning (or **embedding**), means using models and algorithms to find a representation of data in feature space that preserves similarity [46]. It is an improvement over hand-engineering features because no domain knowledge is needed to use these models. A common use is to assign vectors to strings (URLs). Words with similar meanings should be encoded as similar vectors. This section describes models for feature extraction used in this thesis. We have decided to use only **static** and **lexical** features.

Static features are obtained only from the actual data (from the text of the URL) without executing the associated content [47]. **Dynamic** features are then collected from investigating the effects of the content (executables in sandbox environments, visiting the URL address, and capturing the traffic). Features derived from text by examining characters and words are called **lexical** features.

2.5.1 Extracting features from text

In the context of this section, "token," "term," and "sub-string" are synonyms for a smaller part of the document (URL). One could choose words or n-grams (length n substrings) as tokens. Authors use purely lexical features for URL classification in [48] and [4]. [49] uses a mix of lexical and "host-based" features (IP address, WHOIS, DNS records, etc.). [50] use TD-IDF features, as do we in this thesis.

TF-IDF The primarily used model, **TF-IDF** (short for Term Frequency–Inverse Document Frequency), uses information about a number of different documents in T that contain the term t to calculate the vector representation [51]. First, the training corpus is split into tokens. Occurrences of each token are counted. The tokens are filtered by having above a set *minimum* and below a set *maximum* occurrences in the whole corpus to find only the most relevant tokens. Very common tokens, like 'www.' do not add much information. This is also true for very uncommon tokens, which occur only a handful of times. Every feature of the final representation corresponds to a token.
When calculating the features of a specific URL, it is split into tokens. Then $\mathit{tfidf}(t, d)$ values are calculated for every token with a corresponding feature in the representation. 0 is returned in every feature whose corresponding token is not in the URL.

Term Frequency (tf), is a measure of how many times a token has occurred in d, normalized by the sum of all frequencies, as seen in (2.17).

$$\mathit{tf}(t, d) = \frac{f_d(t)}{\sum_{t' \in d} f_d(t')} \tag{2.17}$$

$$f_d(t) = \text{Occurrences of t in d}$$

Document Frequency (df), calculated over whole T, is number of $d \in T$, that contain t, divided by the count of all documents as in (2.18).

$$\mathit{df}(t) = \frac{|\{d \in T : t \in d\}|}{|T|} \tag{2.18}$$

Quotient $tfidf$, for a term and document, is calculated from tf and df in (2.19).

$$\mathit{tfidf}(t, d) = \mathit{tf}(t, d) \cdot \log(\mathit{df}(t)^{-1}) \tag{2.19}$$

High $tfidf(t, d)$ signifies having a high number of occurrences of t in d while being less common in T overall. In (2.21), all three-word tokens were assigned a feature in the representation. Hence, there are three non-zero items in $\mathbf{v}$. For this example, the "TF-IDF vectorizer" trained on a corpus of 30k URLs produced 60k features.

$$d = 'www.youtube.com' \tag{2.20}$$

$$\mathbf{v} = [0, .., 0.87, .., 0.43, 0.23, .., 0] \tag{2.21}$$

Note that TF-IDF does not preserve the relationship between synonyms from documents to the final representation. TF-IDF and similar models (e.g. BOW) disregard the order of the tokens and only consider the frequency of their occurrences.

Dimensionality reduction When dealing with high-dimensional data (for example, 60k dimensions as a result of TF-IDF), a **dimensionality reduction** technique can be used[52]. These methods substitute the points with a lower dimensional approximation, which best preserves spatial relationships. Linear methods find a projection to a linear subspace of lower dimension.
A standard linear method is **PCA** (Principal Component Analysis). PCA can be understood as an approximation of a data matrix with a matrix of lower rank L. "Principal components" are the directions in which the variance of data is maximal and, in turn, best preserve the information. It can be shown that these directions coincide with eigenvectors belonging to the largest eigenvalues of the matrix containing the data. This means that the projection calculated is different for a different data matrix. Eigenvectors are calculated by **Singular Value Decomposition** or **eigendecomposition**. A more in-depth explanation can be found in [20]. PCA can be used to transform data into two dimensions for visualization. Note that this method produces very distorted results. Non-linear techniques, like t-SNE and UMAP [53], can be used for nicer visualizations.

2.6 Classification metrics

We need to be able to compare different classifiers to each other. Usual multiclass classification metrics [54] are based on counting differences between predicted and real classes of the data. There are other metrics specifically for clustering analysis (internal and external), like **rand index**, **silhouette coefficient**, and **purity**. Comprehensive review of clustering metrics can be found in [11]. We are, however, interested in metrics to asses classification results in the proposed evaluation protocol and not for clustering itself. There is no "best" way to measure a classifier's performance, and we need to use multiple metrics to capture different information.

Confusion matrix The classification results are usually structured into a **confusion matrix**. For two classes (A and B), this can be understood as a $2 \cdot 2$ table, as seen below in table 2.22, where classification is taken from the point of class A. **TP** (True Positive) counts instances of A predicted as A. **TN** (True Negative) are instances of class B, classified as B. These are not errors.
FP (False Positive) is the amount of B predicted as A and **FN** (False Negative), if the amount of A, predicted as B. General confusion matrix M (where $M_{ij},\ i, j \leq C$ is the number of class i classified as j) can be calculated for any number of classes C, but can be "collapsed" into this $2 \cdot 2$ matrix by counting all classes except A as a class "other."

Detecting A	Predicted class		
		A	B
Actual class	A	TP	FN
	B	FP	TN

(2.22)

Precision & Recall From the confusion matrix, we can calculate **Precision**, which is the proportion of data predicted as A, that is actually A (eq. (2.23)). **Recall** refers to the proportion of data from class A, which was actually classified as A, as calculated in equation (2.24). For example, $precision = 0.95$ and $recall = 0,42$ means that 42%

$$Precision_A = \frac{TP}{TP + FP} \tag{2.23}$$

$$Recall_A = \frac{TP}{TP + FN} \tag{2.24}$$

Accuracy and F-Measure **Accuracy** measures a proportion of data that is classified correctly in equation (2.25). **F1-measure**, also called F-score, is a (weighted) harmonic mean of Precision and Recall. "F1" refers to the weighing factor between Precision and Recall being 1. The higher weight would favor precision over recall. We will only consider the F1-measure. In equation (2.26), there are two alternative ways to calculate F1-Measure.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.25}$$

$$F_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} = \frac{2 \cdot TP}{2 \cdot TP + FP + FN} \tag{2.26}$$

2.7 Data normalization

Many ML algorithms (including clustering) benefit greatly from data normalization [55]. Normalization scales features and vector representations to be similarly spaced, which primes ML methods for better convergence and standardizes the run of the algorithm. For example, not normalized data can have one feature in $\langle 0, 1\rangle$ and another in $\langle -255, 255\rangle$, affecting the result of some methods. Formulations of the methods mentioned are in [56].

Min-Max Normalization Min-max re-scales the features to interval $\langle 0, 1\rangle$ based on each dimension's highest and lowest values.

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)} \tag{2.27}$$

Zero Mean Normalization This method standardizes each feature to have mean $\mu = 0$ and $\sigma = 1$.

$$x' = \frac{x - \mu}{\sigma} \qquad \mu = \frac{1}{N}\sum_{i=0}^{N} x_i \tag{2.28}$$

Unit Length Normalization One of the simplest methods projects every vector in feature space onto a unit sphere (scales the vector to a unit length). This is done by dividing each vector by its Euclidean length (L2 norm). Works only for $\mathbf{x} \neq \vec{0}$.

$$x' = \frac{x}{||x||} \tag{2.29}$$

2.8 Classification uncertainty and concept drift

Several problems are encountered when using a classifier on data on which it was not trained. A very strong concept drift is present in many cybersecurity domains, meaning that models must be updated and retrained often to produce usable results. Concept drift formally refers to the change in the relation between input and output variables of a model [1]. Changing posterior probability distribution over the feature space $p(k|x)$ invalidates ML models that do not account for this shift.
This change could be caused by new exploits, changing trends, and evolving obfuscation. This translates to changing the prior distribution of classes $p(k)$ (proportions of malware types seen) as well as class likelihood $p(x|k)$ (how a specific exploit acts) or what type of data comes to the input (change in $p(x)$) in Bayes' equation 2.30 [20]. Non-Bayes classifiers (NNs, logreg) do not work with posterior distributions directly but are built on the principles of Bayesian decision theory and are influenced by the concept drift.

$$p(k|x) = \frac{p(x|k) \cdot p(k)}{p(x)}, \quad \text{class k}, \quad x \in \mathbb{R}^D \tag{2.30}$$

The discrimination ability of many ML models is best in the regions of feature space, where the training data is dense and provides many opportunities to learn. New data from sparse regions have low prediction accuracy because there are few similar samples. This is called **out-of-distribution** data. Some detection mechanism, like in [57] is needed to asses when a prediction is unreliable, since usual classifiers have no notion of uncertainty built into them. For example, neural networks give a prediction as a distribution over the classes, from which arg max is taken as the most likely class (explained in 2.4.1). Even for entirely unfamiliar data, the network gives a prediction as usual without detecting any problem. What is more, neural nets are known to be **overconfident** [58] and produce "strong" results (vouching strongly for one class) even when uncertain and on out-of-distribution data. In the presence of concept drift, new samples are more out-of-distribution for the classifier trained on older data.

NN calibration Confidence estimation is tied to Bayesian decision-making and minimizing overall risk. When the neural network outputs 0.2 for malware, we would like the input to be malware in 20 % of cases. If the classifier predicts the certainty of its output, we can discard less certain results to our benefit. While the output of NNs could be interpreted as the posterior probability $p(k|x)$, it is rarely a good approximation. Additional **calibration** techniques can be used on NNs to approximate the posterior better [59] [58].
Note that the type of uncertainty we are facing because of the concept drift (particularly change in $p(x)$) is called **epistemic** (systematic) and arises from classifier training on incomplete data. Another type of uncertainty, **aleatoric**, is caused by inherent randomness in the training set and cannot be reduced [60]. Both of the types are usually present at once.

For this thesis, only estimating NN output confidence is sufficient; we do not use calibration or mitigate the concept drift. We want to use a classifier trained on an older dataset to asses new data and gain at least some information that can be used when choosing clusters. We can use **ensemble classifier** to asses prediction confidence. We show two types of explored NN ensembles ("normal" NN ensemble and Monte Carlo dropout ensemble). We reviewed literature on classification uncertainty and ensemble classifiers [61] [62] [63].

Ensemble classifier **Ensemble classifier** is a model, that combines output from multiple **weak** classifiers. Weak does not mean badly performing in this context. Two main approaches for training ensembles are **bagging** (e.g. random forest) and **boosting** (e.g. adaboost). Bagging draws a random subset of training data (with replacement) and trains the classifier on this subset. Boosting sequentially picks or trains new members based on the re-weighted training set, prioritizing hard-to-classify samples in subsequent iterations. Weak classifiers are sometimes **weighted** to achieve better accuracy, as the resulting prediction combines carefully chosen contributions from each classifier.

2.8.1 NN ensemble

The first relevant model is a "simple" ensemble of neural networks with the same weights. An ensemble of NNs is called **deep ensemble**. Precise implementation of our NNs and ensembles are discussed in Chapter 3. Our **weak** classifiers are trained on random subsets of data, which effectively constitutes a bagging ensemble. [61] suggests that no additional randomization is needed because of NN training's inherent randomness. Prediction confidence is estimated from the models' " agreement " amount. If the classifiers are in agreement with each other, the prediction is very certain. There is a certain freedom to interpret what agreement means.

General ensemble For a "general" ensemble, the output type of individual classifiers can be dissimilar. Some classifiers output distribution over classes (NNs), while others only the predicted class (K-NN, SVM). In this case, the ensemble might agree if all (most) of the classifiers predict the same class (majority vote). This can be generalized to weighted vote [63].

Aggregating NN output Since we are using only NNs in our ensemble, we can combine the distribution from individual classifiers in a more sophisticated way [61] (eq. 2.31) as the class with the highest mean of NN outputs (this is known as **model averaging**). h_j is an output of j^{th} neural network.

$$prediction(x_i) = \arg\max \frac{1}{N}\sum_{j=1}^{N} h_j(x_i) \tag{2.31}$$

The mean and variance of the NN outputs are calculated. Variance is then interpreted as uncertainty. If all NNs have similar outputs for the predicted class, the networks agree more on the classification.

$$\mu_i = \frac{1}{N}\sum_{j=1}^{N} h_j(x_i) \tag{2.32}$$

$$\sigma_i^2 = \frac{1}{N}\sum_{j=1}^{N}(h_j(x_i) - \mu_i)^2 \tag{2.33}$$

The certainty/confidence of the prediction is calculated as (1 - variance on predicted class), as in 2.34. Note that [] denotes an indexing operator in equation 2.34.

$$certainty_i = 1 - \sigma_i^2[prediction(x_i)] \tag{2.34}$$

Our ensemble produces a finer distinction of (un)certainty levels since the output is a real $x \in \langle 0, 1 \rangle$. We can then set threshold δ and filter every point with $certainty \geq \delta$.

2.8.2 Monte Carlo dropout ensemble

Training the traditional ensemble takes a lot of time and requires much memory, especially when simultaneously holding many neural networks in memory. **Monte Carlo dropout** is a scalable way to approximate ensemble by using dropout in one neural network during inference instead of many different neural net instances. When training NN with dropout (in section 2.4.1), the resulting network can be interpreted as an ensemble of sparse networks trained all at once and deciding unison [43]. Usually, dropout is "turned off" for inference, and all neurons (connections) are active. If we drop the connections out for prediction, this group of thinner networks can approximate an ensemble. Many predictions can be made with no additional training time and averaged, as in the ensemble mentioned before (2.34). [64] grounds the MC dropout approach in theory as an approximate Bayesian inference in Gaussian processes and extensively studies this type of ensemble.

Chapter 3

Methodology

3.1 Overview

This thesis aims to study how clustering methods can ease the workload of cybersecurity analysts in the context of network security. We are to define a metric to systematically assess the clustering quality for this purpose and then find a clustering that maximizes the proposed metric. We use the classification of URL addresses as an appropriate context for the thesis, as the data is interpretable, easy to work with, and illustrative since most people are exposed to URL addresses daily.

Chapter content We first describe the setting of the problem, from which we derive our evaluation protocol. We then describe the proposed metric and explain the idea behind it. Afterward, all individual steps of the evaluation pipeline are analyzed and modeled.

Setting Hackers use URL to harm users in many ways (in 2.3). We need labeled data for training supervised models to correctly classify the threats, but where do we get it? Using an older dataset is possible, but we get mediocre results because of the strong concept drift inherent to cybersecurity. A perpetual arms race between hackers and security solution vendors results in swiftly evolving threats. Labeled data older than several months is partially obsolete as it does not contain information about new threats. We want to use very **recent** data for training to be able to spot novel exploits.
When we gather new data (honeypots, traffic monitoring, etc.), additional work is needed to obtain correct labels, because the data is usually unknown. Analysts can individually examine and label every datapoint using their domain knowledge and experience. This is time-inefficient and requires considerable resources, especially for large data streams.

We can cluster the data and then classify whole clusters by examining common characteristics and how the samples behave, similar to labeling single instances. This assumes that the data is similar enough to assign the same class. For this purpose, we want to create coherent clusters that include maximum information to gain the most value from the clustering step.

We want to asses clustering's usefulness for manual analysis. This is quite an abstract quality, which is hard to measure objectively. Instead, we propose a sort of **evaluation protocol** (pipeline), from which a metric is derived later. The evaluation protocol is modelled after the described situation, where an analyst labels clusters of unknown data, in order to train a classifier.

Modeling the evaluation pipeline We can see a scheme of the evaluation protocol in Figure 3.1. This pipeline closely models the presented situation. In **part 2**, we cluster the data and select N clusters to be analyzed by an expert. In **part 5**, a **simulated** security expert infers labels for the clusters. We use the ground truth labels to simulate labeling by human. In **part 7**, we train an "end classifier" using the labeled data. We test this classifier on a test portion of the dataset, which is split from the training set in **part 1**. The whole pipeline is wrapped in a cross-validation loop to prevent overfitting when tuning hyper-parameters. We use two end classifiers (neural network and logistic regression) to measure performance.

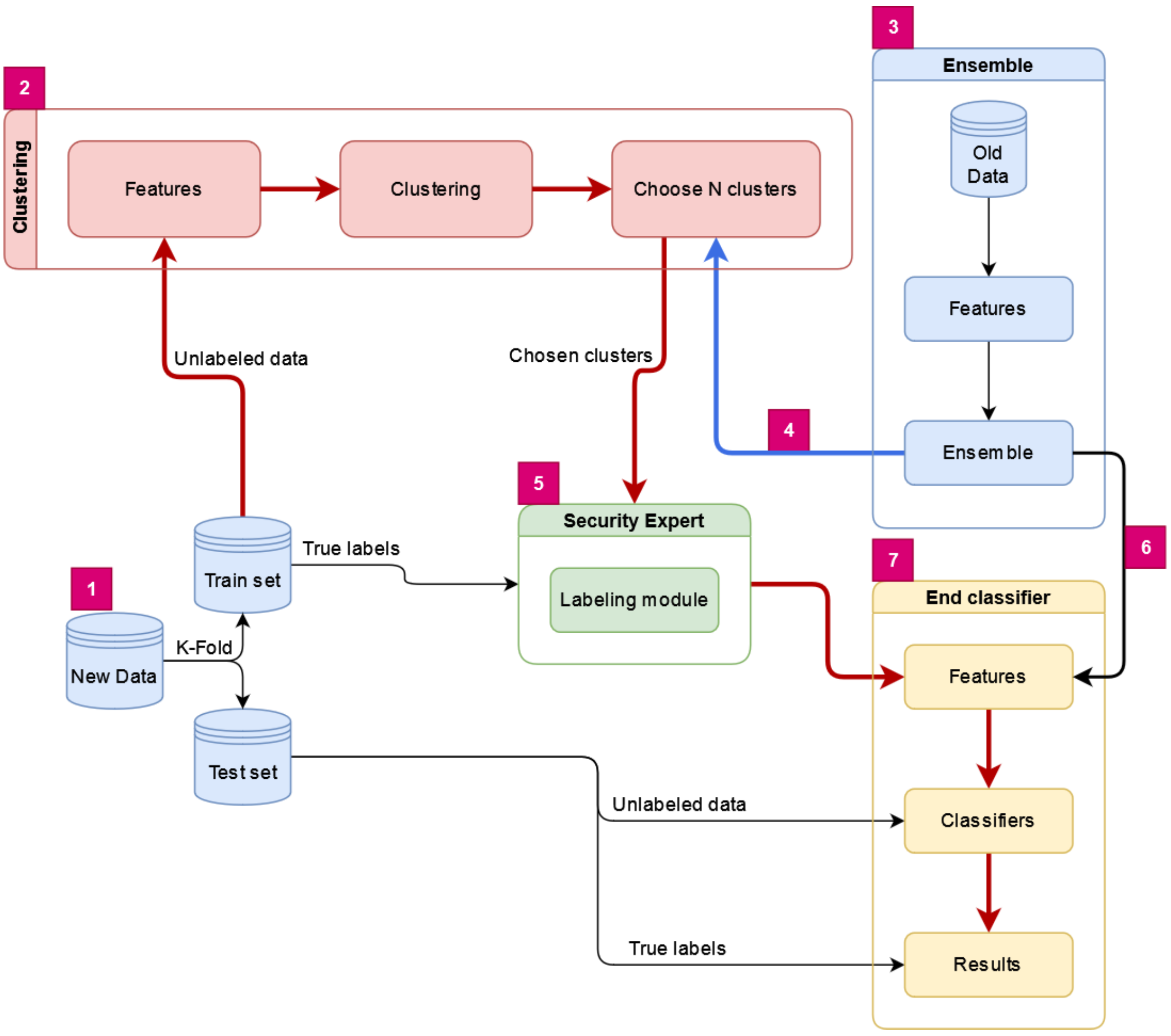


Figure 3.1: The evaluation protocol scheme

Selecting clusters for analysis We want to gain maximal value from the human work involved in the labeling. Expert analyst's attention is expensive compared to computation. We want to automatically select the best clusters to be manually labeled by an analyst in exchange for computational overhead. We explore several cluster selection heuristics in 3.5.

One of the discussed solutions uses an obsolete dataset to inform the cluster choice. Using this dataset directly would lead to bad results, since the new dataset includes many

out-of-distribution data, which would be incorrectly classified. Concept drift, out-of-distribution data, and classification certainty estimation are discussed in 2.34. Instead, we can use the old dataset to predict cluster labels. We do not need exact prediction to gain *some* information about the clusters. This way, we can still use available but older (labeled) datasets to help us in a certain way.

We now explain the rest of Figure 3.1. We train an **ensemble classifier** in **part 3** of the evaluation pipeline. The ensemble classifies the training data, estimating **prediction certainty** in the process. We use the certainty scores and predictions in **part 4** to estimate cluster classes. This information is used to select clusters to be analyzed. In **part 6**, we *can* use very certainly classified data directly to train the end classifier.
Note, that other cluster selection heuristics that do not require additional labeled data and training an ensemble have been tested (selecting highest entropy and smallest clusters).

Summary of the evaluation pipeline, as seen in Figure 3.1:

1) New dataset is split to train/test set (in each fold of cross-validation)
2) The unlabeled data is clustered
3) Ensemble is trained on the old data and used on the train set, estimating confidence
4) Clusters are chosen for further evaluation
  ↪Classes and confidences from step 3 may be used
5) Expert (analyst) module classifies the clusters using the true labels of the train set
6) The most certainly classified data (above a threshold) *can* be used directly for training.
7) Classifiers are trained on the subset of training data labeled by the analyst.
  ↪ Classification metrics are computed on the test set.

Proposed metric The proposed **metric of clustering quality** is the accuracy and F1-measure obtained on the test set by the end classifiers in our evaluation protocol. The idea is that if a clustering performs well in this modeled situation, then some qualities (otherwise hard to pinpoint) make the clustering well-suited for similar uses. We would like the best found clustering to produce coherent clusters from a different dataset, in order to confirm, that the metric works. For this purpose, we have gathered the dataset C, explained in 3.2.3.

Similar work We have reviewed the literature on using clustering in classification pipelines to help manual classification (more in 2.1.2). The authors of [19] propose a clustering algorithm that produces several centroids of each cluster to be submitted to a cybersecurity expert to analyze further. The quantitative analysis in similar papers focuses mostly on the quality of the clusters (using internal criteria like silhouette coefficient) and coverage of the dataset (how much of the dataset can be classified in as few clusters as possible) instead of directly measuring the accuracy of inferred labels against known ground truth.

[2] compare several algorithms in terms of resulting cluster purity (fraction of the majority class). This could be used alongside other metrics like dataset coverage. We think it is crucial to consider the amount of work by the analyst, which is not accounted for by [2] or [19]. We do this by limiting the number of analyzed clusters. We maximize the metric while requiring the same amount of work from the analyst. In our experience, when we maximize cluster purity alone, the extracted clusters are often very small but highly homogeneous. When we try to maximize the coverage of the dataset (in a set number of

clusters), the preferred clusters are larger and, therefore, less more dissimilar. Maximising coverage often leads to lower cluster purity, because the extracted clusters are larger. Our proposed metric delegates the problem of balancing the coverage and purity of the clusters to the end classifier.

Active learning We want to highlight a particular similarity of the evaluation protocol to **active learning** pipelines. Active learning models choose, which data is best for their training, usually our of a huge set of unlabeled data. This data is then given a label by an outside entity (called the **oracle**) and used for training. After every training step, the model chooses new data to be classified by the oracle. The model can choose (queue) data based on different criteria. If a good choice of data for training is applied, the model needs much less labeled data for training. Overview of active learning can be found in [65].

We do not study active learning in the thesis since we pick all clusters simultaneously. If we were to select a cluster, present it to an analyst for labeling, and then pick another cluster based on the information provided by previously labeled data, we would consider our model an active learner. We decided to select all clusters at once and not sequentially since it is less complex to explain and implement. It allows us to experiment with different selection functions that do not fit into the active learning model (taking the highest entropy and smallest clusters).

3.2 Data

We use several URL datasets in different parts of the evaluation protocol. **Dataset A** is used in place of new and unseen data. We have compiled this dataset ourselves. **Dataset B** is used to train the ensemble classifier. An older, publicly available dataset from Kaggle [66] is used. In the situation we are modeling, using the older dataset to directly train a classifier leads to bad results, which is confirmed in Chapter 4 by one of our baselines (Table 4.1). The best clustering is tested on **dataset C**, consisting of unlabeled URLs, which we extracted from proxy-logs obtained from [67]. This is done to asses the resulting clustering on different data, than which it was fine-tuned on.

Dataset	Size	Balance of classes
A	110k	balanced
B	554.7k	in Table 3.3
C	76k	unknown

Table 3.1: Dataset overview

Class descriptions There are four classes in datasets A and B. Explanation is loosely taken from [68]. **Benign** URLs lead to trustworthy, legitimate websites, that don't pose a threat to the user. **Phishing** URLs lead to phishing websites, that try to appear legitimate to fool users. **Malware** contains URLs that lead to hacker compromised websites injecting and downloading malware. **Defacement** URLs are from legitimate websites hosting fraudulent or hidden URLs that lead to malicious webpages. This class is only (initially) present in dataset B, and was filtered out for our purposes.

3.2.1 Dataset A - Recent dataset

Dataset A consists of 110k **recently** gathered URLs from **benign**, **malware** and **phishing** classes, which we sampled from publicly available sources. The human analysis in our evaluation protocol is performed on this dataset. We tried to gather other classes, like **cross-site-scripting** (XSS), **SQL injection** (SQLi) (in 2.3) and **defacement**, but we could not find any sufficiently recent data. XSS and SQLi are found in stored scripts, (studied in [7]), but we could not find any sources of URL based versions. There was a strong emphasis on obtaining very recent data for this dataset.

Defacement class Defacement class is of special interest to us, since it is contained in dataset B also used in this thesis. We were unable to obtain sufficiently recent samples of defacement URLs when compiling dataset A. In the original source of the defacement data for dataset B, ISCX-URL2016 [68], the authors use a specialised web-crawler to explore some previously known defaced websites (described in [4]). We do not have sufficient resources to gather our own data in similar manner. We decided to filter out the defacement data from dataset B for compatibility of the datasets in our pipeline. We are aware, that this might have decreased the diversity of dataset B. We think filtering the data out was an acceptable solution, since we only use this dataset in training the ensemble and we do not analyze it directly via clustering.

Sources Sources of all included classes can be found in Table 3.2. We gathered our **malware** data from URLhaus [69], a public repository, where users can add URLs that distribute malicious binaries. 90 day old or still active malware captures were downloaded on 24.3.2024. **Phishing** data was procured from a git repository of user **mitchellkrogza** [70], which searches for new phishing links and periodically confirms the websites are still active. We considered using Phishtank [71], but it currently does not accept new users, preventing download without an API key. We had problem with obtaining **benign data**. There is usually no need to hold databases full of safe links, especially full of private searches. The most current dataset we found is [72] from 2020. We considered using public webcrawl data (e.g. CommonCrawl). Similar repositories are extremely large and using Google Safe Browsing APi or VirusTotal to confirm the data is benign is advisable. We decided against this approach, since we have limited hardware resources.

Class	Source	Amount	Published	Note
Phishing	phishing.database [70]	37k	27.2.2024	Active phishing URLs
Malware	URLhaus [69]	48k	24.3.2024	Last 90 days & active
Benign	Webpage dataset [72]	1,1 million	2020	

Table 3.2: Sources for dataset A

Balancing the dataset We initially balanced our dataset 80/10/10 in favour of benign data, since real-world cybersecurity data is highly unbalanced. Because of this balance, most selected clusters were benign which resulted in near perfect results on benign test instances and false negatives on other classes. When classifying this unbalanced dataset, results above 90 % accuracy were usual. Most of the data was classified as benign, with around 60 % accuracy on non-benign classes. Consequences of unbalanced dataset is studied by [73], who describe similar effects. We decided to use the same data, but **balance** the amount of individual classes in our final dataset with **110k** items. The data from individual classes were sampled randomly from gathered data.

3.2.2 Dataset B - Older dataset

The second used dataset is "Malicious URLs dataset" obtained from Kaggle [66]. In the original dataset, 651 191 URLs are separated to classes **malware**, **phishing**, **benign** and **defacement**.

The Kaggle dataset was published in 2021, but majority of the data is considerably older. The dataset is built from several publicly available datasets. The base of is taken from dataset "**ISCX-URL2016**" [68], released by Canadian Institute for Cybersecurity in 2016. Spam URLs from ISCX-URL2016 were discarded.
More phishing URLs were taken from "**phishtank**" [71] and "**phishstorm**" [74]. We assume the phishtank data was collected when the Kaggle dataset was made public in 2021, since no information is given. Phishstorm was made public in 2014 as per the original publication [75].
More benign data was taken from Github repository "Using-machine-learning-to-detect-malicious-URLs" of user **faizann24** [76] released in 2017. Malware data amount was increased by "**Malware domain blacklist dataset**", which is currently (April 2024) not available. We did not find the year the data was gathered originally.

The data is gathered between years 2014 and 2021. We conclude, that the data is sufficient for our use as an out-of-date dataset, at the time of working on this thesis in early 2024. Class benign in dataset A was gathered in 2020, while in dataset B, this class was gathered before 2016, even though the dataset B was made public in 2021.

The dataset is disbalance in favour of benign data. Additionally, defacement data was filtered out as described previously (3.2.1. Precise amounts of individual classes after filtering out 95k of defacement data listed in Table 3.3.

Class	Quantity [10^3]	Percentage of whole
benign	428,1	77.17 %
malware	32,5	5.68 %
phishing	94,1	16.97 %

Table 3.3: Amount of data from each class (in thousands)

3.2.3 Dataset C - Raw dataset

We have extracted unfiltered URLs from publicly available server logs. The data was "recovered from public FTP servers in Syria over a period of six weeks in late 2011", but was eventually leaked and is now public domain (according to the source website [67]). Syrian internet providers heavily censored traffic at the time. [77] are interested in the censorship strategies used on the data from this leak.

We are more interested in cybersecurity phenomena in this dataset, which we do not examine before clustering to make sure we do not unknowingly use any knowledge of this data when tuning the parameters on dataset A. This dataset is unlabeled. We processed the logs by extracting the requested URL and *removing duplicates*. The requested URL was assembled from URL scheme, hostname, path and query in different columns of the log record. We think filtering unique addresses was necessary. Multiple datapoints with exactly same features would affect the clustering algorithms in different ways, which is not the focus of this thesis.

From one of the blocks downloadable at [67], **76k** unique URLs were extracted. We consider this enough for our purposes. We extract all URLs, even from the censored websites (several hundred URLs). We are aware of the sensitive nature of the data, and we avoid directly showing addresses with personal information (username, email, password) in the URL. Client IP is redacted from the data.

We chose this exact dataset, because we value unchanged and unfiltered data for the final analysis. The data has been gathered at one place and one time, meaning it is as close to real-world traffic as we could get.

3.3 Features

We have to first find suitable representation of the URLs to use the clustering on. Papers on the topic of URL classification suggest combination of **static** and **dynamic** features. Some show geographical, DNS and IP related [49] features or features derived from links between websites [78]. We use only **static**, **lexical** features derived form the text of the URL [48], as opposed to dynamically gathered features obtained by executing the content hidden behind the URL. More about feature extraction in 2.5.

When using static features, no further evaluation of the website is needed. The model is simpler and can be trained quickly. Additionally, many malicious websites are taken offline rapidly after being reported to the ISPs and hosting services, which disallows gathering additional data from the websites. We use a set of hand-picked features with combination of **PCA**, **TF-IDF** [50] and **normalisation** techniques to obtain sufficient vector representation of the data. Details of the models are in 2.5. An alternative to TF-IDF is to use pretrained embedding like **word2vec** [79] and **BERT** [80]. We decided to use simpler models that can be trained locally and are easier to interpret.

Feature extraction overview TF-IDF followed by PCA is used in parallel to hand picked features (in Table 3.4). These vectors are appended and normalised by a chosen method afterward. The process is visualised in figure 3.2. We can optionally not use some of these models. Is possible to use just a picked feature set or TF-IDF. PCA can optionally be omitted, but the resulting vectors are usually very large and sparse.

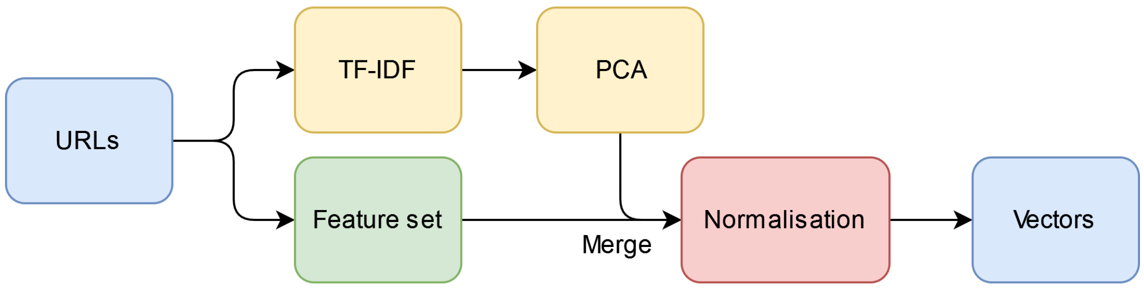


Figure 3.2: Architecture of Feature Extraction module

Note, that same feature set is used before the end classifiers. We directly use the model, which was trained on whole training corpus. This was done to decrease the amount of free parameters. If we decoupled the 2 sets of features, changing amount of data coming from analyst would make the tuning harder and results less controllable.

Hand-picked features We chose 11 features to be calculated directly. We think these parameters are important when labeling URLs based on our experience with URL based threats. URLs are stripped of 'http://', 'https://' and 'www.' in front, since these are extremely common and we think do not provide additional information because of that. In Table 3.4, we can see the individual features with explanations. We split the URL into host:port and the rest (called path-string) to construct features 3,8 and 2,7. Other features are calculated from the whole URL. Explanation of URL structure in 2.11. We were influenced by [48], [78] and [49] when modeling these features.

Index	Feature Description	Value range
1	Length of URL	$\mathbb{N}^+$
2	Number of divisions in path-string	$\mathbb{N}^+$
3	Number of divisions in host-port	$\mathbb{N}^+$
4	Amount of unique characters in URL	$\mathbb{N}^+$
5	Count of '%' in URL	$\mathbb{N}^+$
6	Count of '&' in URL	$\mathbb{N}^+$
7	Avg. length of path-string subdivision	$\mathbb{R}^+$
8	Avg. length of subdomain	$\mathbb{R}^+$
9	Count of '/' in URL	$\mathbb{N}^+$
10	Count of '_' in URL	$\mathbb{N}^+$
11	Count of '-' in URL	$\mathbb{N}^+$

Table 3.4: Hand picked feature set

TF-IDF Vectorizer TF-IDF was selected over simpler BOW model [81] for generating lexical features. TF-IDF produces sparse vectors, based on string occurence frequency. The strings can be generated as n-grams of arbitrary length or sub-strings split by URL delimiters ('.,;/&=' etc.). TF-IDF module produces a vector of features for each preprocessed URL. The principles of TF-IDF are explained in 2.19. In Table 3.5, settings relevant to this model are listed.

Parameter	Description	Format
Min *df*	Minimal number of documents term appears in.	$\mathbb{N}^+$
Max *df*	Maximal document frequency for filtering terms.	$\langle 0, 1 \rangle$
Tokeniser	Word, n-gram or n-gram within words; How to split terms from URL	Choice
N-gram range	Range of substring lengths for tokeniser, when N-gram is chosen	From-To

Table 3.5: Parameters of TF-IDF module

PCA After producing a huge number of features (up to tens of thousands) using TF-IDF model, PCA is used to reduce the dimension of the representation (described in 2.5.1) of the resulting vectors. PCA is very fast compared to non-linear techniques (t-SNE, UMAP [53]) and works on unlabeled data (unlike for example LDA). Only tunable parameter for PCA is the *output dimension* ($\leq$ *input dimension*).

Normalisation We have decided to use three normalisation algorithms. The methods are explained in 2.7. **Min-Max** norm. scales each dimension to a preset range of values (default $\langle 0, 1 \rangle$) based on maximum and minimum value. **Zero mean** norm. scales the data in every dimension to have $\mu = 0,\ \sigma = 1$. **Unit length** norm. scales each datapoint independently by normalising its length to 1.

3.4 Clustering algorithms

Usage, characteristics and mechanisms of clustering algorithms are explained in 2.1. We have chosen clustering algorithms from **centroid-based** (minibatch K-means), **density-based** (DBSCANB, OPTICS) and **connectivity-based** (BIRCH, agglomerative clustering) algorithms. We feel, that diverse algorithms based on different principles provide the most interesting comparison.

The authors of [11] use more comprehensive division of clustering methods (than we do in 2.1.) The paper also explains some more advanced algorithms, that we do not cover in this thesis, like genetic algorithms, swarm intelligence based algorithms, EM algorithm, self organising maps, affinity propagation and subspace clustering. We think that testing the most complex clustering algorithm (or more algorithms) is not the focus of this thesis. We are rather interested in showing how the evaluation loop works in practice and whether the resulting clustering works on unseen data. In this Section, tunable parameters of the clustering algorithms are summarised.

Minibatch K-means We use **minibatch K-means** with **K-means++** initialisation instead of "vanilla" K-means. These improvements algorithm are very common and result in faster runtime, faster convergence and better value of local minima. Details of K-means and it's minibatch version in 2.1.3. The size of the minibatch used is default in sklearn (1024). Parameters are summarised in Table 3.6.

Parameter	Description
n	Number of clusters
Inits	Number of re-runs of the algorithms

Table 3.6: Parameters of minibatch K-means

DBSCAN DBSCAN is only parametrised by the minimal density of core areas, set by ϵ and *min_pts*. Details of density based algorithms, particularly DBSCAN and OPTICS can be found in 2.1.4. DBSCAN parameters are in Table 3.7.

Parameter	Description
ϵ	Radius of neighborhood
min_pts	Minimum points in neighborhood

Table 3.7: Parameters of DBSCAN algorithm

OPTICS OPTICS is similar to DBSCAN, but uses additional parameters. *max_eps* denotes the maximum neighborhood size considered when ordering the data by reachability (explained in 2.1.4). We used the default method of extracting clusters, which separates clusters based on steepness of reachability graph, as explained in 2.1.4. OPTICS parameters are in Table 3.8.

Parameter	Description
ϵ	Radius of neighborhood
min_pts	Minimum points in neighborhood
max_eps	Maximal considered ϵ

Table 3.8: Parameters of OPTICS

BIRCH BIRCH algorithm (explained in 2.1.5) is parameterised by branching factor B and distance threshold T, which parameterise the CF tree built from the data, from which *n_clusters* leaves are cut. Global agglomerative step of the implementation is done using Ward linkage. Parameter summary is in table 3.9.

Parameter	Description
B	Num. of child nodes in the tree
T	Maximal radius of cluster
n	Number of clusters

Table 3.9: Parameters of BIRCH

Agglomerative clustering Agglomerative clustering (explained in 2.1.5) is used in the thesis as a commonly used hierarchical algorithm. We use a memory efficient implementation, which could only use **single**, **Ward**, **median** and **centroid** linkages, since for the other linkages, memory efficient algorithms are not known. Linkages are explained in 2.1.5. Only parameter is n = number of clusters.

3.5 Cluster selection

We are allowed to present only a set number of clusters to the analyst, which we have to select from all extracted clusters. We *can* use confidence and predictions produced by an ensemble classifier (expanded on in 3.8, 2.8) to inform the choice of clusters. Alternatively, we sort clusters by size or by entropy of the data, then choose a portion of the clusters.

Size of clusters The simplest heuristic for choosing clusters takes N smallest clusters. The idea is, that the smaller clusters are often very **pure**. In [82], the authors automatically label the smallest clusters as malicious, since the benign data extremely outweighs the anomalous data and tends to cluster together, forming large clusters. Our dataset is balanced in comparison. We still consider this a good option because small clusters tend to be pure. [2] conclude, that this heuristic is not reliable, since some of the smallest clusters were observed to be benign and vice-versa. In our system, the clusters are labeled by the analyst, which effectively solves this problem by assigning the correct label.

Entropy of clusters Second heuristic used takes N clusters with highest URl entropy. We compute this by counting all **word** tokens in a cluster, **normalising** this count to obtain distribution and then computing **Shannon entropy** of this distribution (3.1).

$$H(\chi) = -\sum_{x\in\chi} p(x)\log_2 p(x) \tag{3.1}$$

Formula for calculating entropy H is in equation 3.1. χ is the distribution over tokens found in the cluster, and sums to 1.

Clusters with high entropy contain more information, but could be less pure and pose a problem for analyst.

3.5.1 Choosing clusters using labels and confidence estimates from ensemble

Heuristics in this subsection use the information obtained from ensemble classifier to choose clusters in a more sophisticated manner. The selected ensemble is explained in 3.8.

Predicting class of a cluster We use the predicted classes of the data with the estimated prediction confidences to find a label for the cluster. In eq. 3.2, $\mathbf{class(x_i)}$ is a one-hot encoding of the class predicted for instance x_i. We use confidence of the prediction as a weight. $\mathbf{pred(C)}$ is a distribution over classes. The most certain predictions have a higher weight than uncertain ones, "moving" the whole cluster more towards the class with certain predictions.

$$\mathbf{pred(C)} = normalise(\sum_{x_i\in C} \mathbf{class(x_i)} * confidence(x_i)) \tag{3.2}$$

Cluster selection We use information from eq. 3.2 to sequentially select clusters. We want the classes of selected data to reflect a set class distribution (e.g. 80/10/10 % of benign, malware, phishing). In each step, we calculate, which class we need to add the most of, then select a cluster with the highest proportion of this class.

We begin with the target class *balance*, which we want to be achieved in the selected datapoints. *current_frac* and *points_chosen* help us keep track of how many datapoints from each class were already selected. In **step 2**, we calculate the *needed* fraction of each class. In **step 3**, we choose a cluster, which has a highest predicted proportion of the most needed class. We update *points_chosen* by adding the size of the cluster to the class we were picking as most needed. We do not use the label from the analyst, but we count the whole cluster as the class we were trying to select. Our approach works well, if we have can select between many pure clusters.

Steps of the proposed algorithm:

1) Initialise *current_frac*, *points_chosen* as zero vectors
2) *needed* = *balance* - *current_frac*
3) Choose cluster with highest *pred(C)* for the class with highest *needed*
4) Adjust *points_chosen* for the needed class and recalculate *current_frac*
5) If need more clusters: goto step 2

Problems of our approach The*current_frac* information at the end of the algorithm does not need to resemble the real distribution of the classes in the selected clusters, because we do not have information from the analyst. If we used information from the analyst to update *points_chosen*, then the pipeline could be considered an active learning model.
Note, that there are many different ways to use the predicted classes from ensemble classifier. We could weigh the prediction certainty by different functions to make high certainty more advantageous. We could use the predicted classes to estimate cluster purity and take the purest clusters, regardless of the class.

How to choose the balance of the clusters?

Next, we needed to asses, how we want the algorithm to balance the cluster choice. We propose two different distributions: **evenly balanced**, and **80/0/20** (benign, malware, phishing).

Evenly balanced We think it is beneficial for the input to the end classifier to reflect there balance of the test data.It is often not possible to know a class balance in a completely unknown dataset. It is a good starting point to select balanced amount of data from each class, if we do not know anything else. The real balance is often changed, when some of the clusters are rejected by the expert module.

Balanced towards benign We have observed tendency of some clustering algorithms to produce a majority of phishing clusters. DBSCAN and agglomerative clustering often resulted in most (80 %) clusters being almost purely phishing data. This knowledge can be used to balance the dataset. The data, that is classified from the ensemble with certainty above 99 % is mostly labeled as malware. We can choose a balance of 80/0/20 of benign, malware and phishing data to select with the hope, that phishing clusters will "overpower" the arbitrary disbalance towards benign, while we add malware samples obtained as the most confidently classified data from the ensemble classifier.

3.6 Expert module

We decided to use an algorithm to model evaluating clusters by a cybersecurity expert, because it would be time-consuming to manually label clusters during every iteration of the loop. Additionally, the authors own experience with malicious URLs might be insufficient, and could lead to errors and inconsistencies in labeling.

Labeling process We decided to use the knowledge of the real classes (ground truth) to label each cluster. We count how many datapoints of each class are in a cluster, and label the whole cluster as the most represented class. **Purity** is the fraction of the most represented class. The expert can **reject** the cluster, when the purity is below a specified **threshold** ξ. We limit the number of clusters presented to the expert.

Amount of clusters labeled We decided to model the number of processed clusters as **N=150**. If labeling a single cluster took around a minute, labeling all 150 would be around 2.5 hours, which is reasonable. Optimally, we want to label tens or hundreds URLs at once, not just a few at a time, to benefit from the clustering step.

Setting the threshold We Initially fixed the threshold at $\xi = \mathbf{99}\%$. The intention was, that only the most homogeneous clusters can be labeled by the analyst precisely, and labeling other clusters would lead to imprecise labels. We examined more closely the choice of threshold, since we had concerns, that cluster with high entropy would be hard to label by the human expert and our model would need to be updated to better model the manual analysis.

Based on manually inspecting the clusters, we decided to *relax* the threshold to reject less clusters. This was done, because we could find phenomena and label even more impure clusters (than with $\xi = \mathbf{99}\%$). We would not reject many of the impure clusters. When presented with similar clusters with around 99% purity, we could not find a difference, but our model had rejected one of the clusters. We allowed more relaxed labeling, which rejects less clusters. With good choice of features and clustering parameters, the similarity of the data within the cluster is inherently high and we mostly do not need to be concerned with homogeneity. New threshold was set as $\xi = \mathbf{90}\%$.

3.7 End classifiers and used metrics

We use a classifiers at the end of our evaluation pipeline to simulate using the labeled data for classification. The models are then used to classify test data, giving the final classification metrics. This is done in every fold of cross-validation. We use **neural network** (NN) and **logistic regression** (logreg), but we believe that SVM, K-NN and decision trees or tree ensembles would be sufficient for the project. We benefit from using NN instead of the other options, because the implemented NN infrastructure is used in the ensemble classifiers (in 3.8).

Considered classifiers **SVM**s work well with smaller number of input data (that some clusterings produce) and generalise well. A concert with SVM is bad scaling with multiple classes (realised using one-vs-all scheme). **Decision trees** offer explainability, because the applied decision rules can be extracted. On our hardware, K-NN took an extraordinary amount of time during inference, even when using kd-trees.

Used classifiers are explained in 2.4 (focusing on NNs). We use comparatively shallow feed-forward neural network with dropout. Dropout parameters and used architecture is in Figure 3.3). We have tested CNN on the same features (convolution can be seen as sparse linear layer sharing weights for each set of connections), to no considerable improvement in accuracy. We train the network for **90 epochs**, since longer training did not lead to improvement. **Batch size** chosen is **512**, as a compromise between learning faster with large batches and better convergence with smaller ones [83]. We use **ADAM** optimiser and **cosine annealing** scheduler. The initial **learning rate** is fixed at **0.05**. We have considered using **warm restart** method for updating step size, but it did not lead to improvement.

Overfitting *Note*, that neural networks tend to overfit by training too long on smaller train set. This happens a lot in our pipeline, when clustering produces very small clusters or when the expert module rejects a considerable portion of clusters. We have implemented the neural network with several regularisation mechanisms, namely weight decay and dropout to mitigate this problem. We do *not* change number of epochs based on amount of training data.

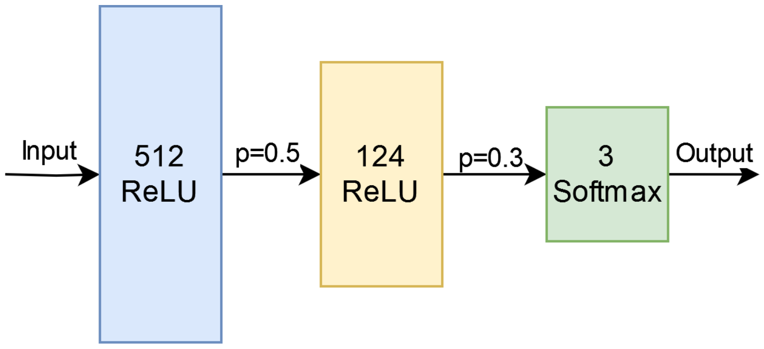


Figure 3.3: Used NN architecture. Layer scheme with dropout parameter inbetween.

Metrics The metrics used to asses classifier performance are **accuracy** and weighted **F1-measure** (in 2.6). F1-measure is a good compromise between recall and precision, which we find individually useful. It is not feasible to take into account precision, recall and accuracy of all 3 classes during every run. Hence the use of (weighted) F1 as a compound metric, together with accuracy. Since we eventually decided to use a balanced dataset, F1 strongly coincides with accuracy. We do not show F1-measure most of the time, since it copies the improvements in accuracy well. The metrics are averaged over runs of cross-validation. Standard deviation is an important indication of algorithms robustness and reliability.

3.8 Ensemble classifier

Theoretical basics needed for this Section are explained in 2.8. We want to use older, available dataset of the same type to *automatically* gather some information about the unseen data. Classifiers used on **out-of-distribution** data give unreliable results, but we still need some information. We could either use an out-of-distribution detector [57], calibrate our NN [59] or somehow estimate the classification certainty of a classifier [62] [61]. We have decided, that estimating classification confidence is sufficient for this project. We decided to use an **ensemble classifier** trained on dataset B to classify dataset A and estimate the prediction certainty in the process. We feel, that use of ensembles is more explainable end easier to interpret and implement than other methods mentioned.

We have investigated several types of ensemble classifiers for this purpose. First, a general ensemble built from different classifiers was implemented. The confidence of the classification was calculated as the fraction of classifiers which voted for the final class. This was a rough scale and finer distinction between levels of confidence was needed. This ensemble lead to very bad accuracy of the inferred labels.

3.8.1 NN ensemble and MC dropout comparison

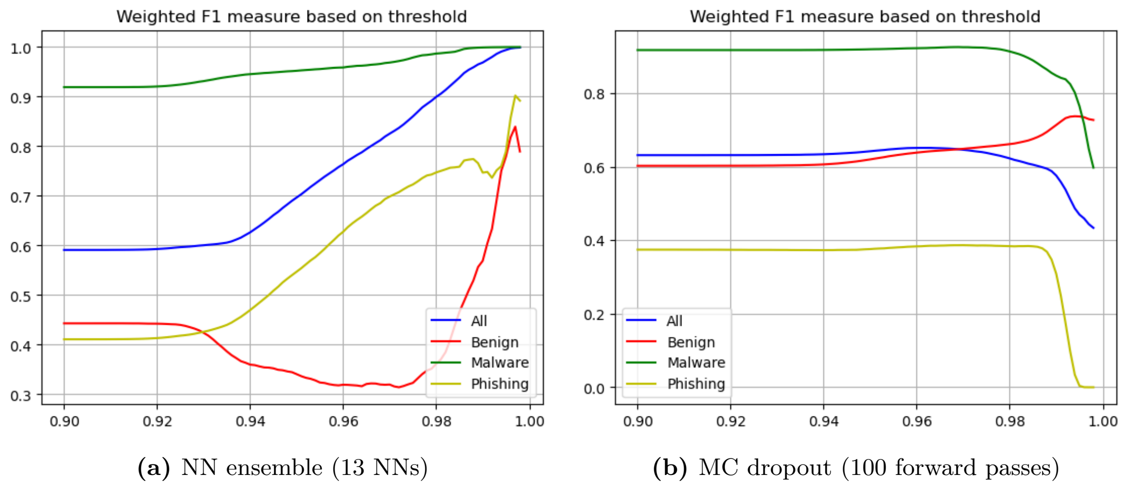
The two main considered models, NN ensemble and MC dropout are explained in 2.8. NN ensemble uses many independent NNs, while MC dropout uses one NN with dropout enabled for inference to approximate an ensemble. During data classification, the class with the highest mean over all NN outputs if taken as the prediction. The variance of NN outputs is interpreted as uncertainty.

The authors of [84] compare empirically these two models for estimating **epistemic uncertainty** in the context of computer vision. The authors conclude, that the traditional ensemble outperforms MC dropout in "reliability and practical usefulness". The lacking performance of MC dropout is attributed to lesser variability between individual predictions. We did come to the same conclusion when deploying these two models in our task. A drawback to traditional ensemble is it's training time, while MC dropout is trained comparatively quickly and can be scaled during inference.

Architecture and feature extraction When training both ensembles, features of URLs were extracted using the same model as in the main loop (TF-IDF with PCA mixed with hand picked features). Since more data was used for every NN in MC dropout and deep ensemble, higher *min_df* of TF-IDF model was set to filter out rare tokens. Both ensembles use the same architecture as the end classifier NN (Figure 3.3). Decoupling the ensemble architectures might yield better results, but we simplified this problem and compared similar ensembles.

Ensemble architectures The NN ensemble is built from 13 neural nets. Each NN was trained on 250k data randomly sampled from dataset B. For MC dropout, one NN was trained on 400k samples from dataset B, with dropout set as $p = 0.5$ between all layers for 150 epochs. Multiple forward passes with dropout enable are then realised. The results are processed as in the regular ensemble (eq. 2.31 in 2.8.1)

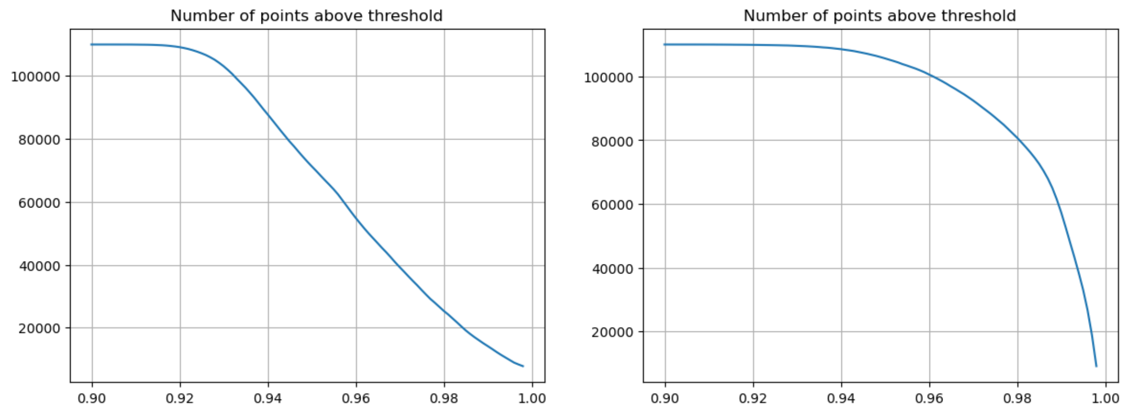
Results on dataset A For Figure 3.4, we trained both ensembles on the dataset B to predict classes of dataset A. We then filter points, whose classification confidence as above a certain *threshold*. In Figure 3.4, we plot F1-measure of each class, but only on points with prediction confidence above the *threshold* on x-axis.



(a) NN ensemble (13 NNs) **(b)** MC dropout (100 forward passes)

Figure 3.4: F1 of each class on points above a confidence threshold for both types of ensemble

We can see that with higher confidence threshold, F1-measure improves for the usual NN ensemble. MC dropout F1-measure in Figure 3.4b quickly worsens when taking only the most certain classification results. Tuning the number of predictions from MC dropout (forward passes) did not seem to remedy this problem. From our experience the malware URLs are classified with more certainty, probably because the distribution of malware class did not "drift" as much between the datasets as for the other classes. This manifests as high F1-measure on class 'Malware' in both graphs in Figure 3.4.



(a) NN ensemble (13 NNs) **(b)** MC dropout (100 forward passes)

Figure 3.5: Number of instances with confidence above threshold (dataset of 110k)

In the Figure 3.5, we see how many points are predicted with confidence above a certain threshold (on the x-axis). NN ensemble is less confident and the number of data above threshold starts to "fall of" earlier. We see that only about 25k predictions are more confident than 98%. MC dropout is very confident in its prediction; there is 80k data classified more confidently than 98%. MC dropout is one NN with some weights masked, so it makes sense, that there is less variance between forward passes.
Note that in Figures 3.4 and 3.5, the x-axis starts at 0.9, since there are no points with confidence less than 90% and all of the graphs are constant below this value.

Usage of the ensemble in our pipeline Based on the results presented in this Section and conclusion by [84], **we have decided to use the "normal" NN ensemble** rather than MC dropout. The ensemble is applied on dataset A. Every network and its respective feature extraction class instance is used independently. Then the results from NNs are processed. For every URL, the ensemble produces **class** and **confidence** (= certainty = 1-uncertainty). We use these results two-fold. **First**, the classes and confidences for training data are passed to the function choosing clusters for the expert to classify (described before in 3.5.1). **Second** use is to supply the most certain data to the end classifier directly, with a hope, that very certain predictions are correct. The data with certainty above 0.99 is predominantly classified as malware. An idea for one of the cluster selection functions is to choose benign and phishing clusters in order to have somewhat balanced dataset, while using the data with the most certainty (mostly predicted as malware).

3.9 Implementation notes

The evaluation protocol was implemented in python programming languages. TF-IDF, PCA, clustering algorithms and k-fold implementation were taken from **sklearn** [27]. Memory efficient agglomerative algorithm was used from **fastcluster** library. The focus of the implementation was fast iteration of the entire process to make the parameter tuning time efficient. NN implementation was done in **pytorch** using **CUDA**, since we have access to a reasonably efficient NVIDIA GPU. Structure of the attached code repository with brief description of the files can be found in the Appendix A.

Hardware specifications The tuning process was run on a personal computer with 8 core Intel Core i5 processor, 8GB RAM and NVIDIA GTX 1650 graphics card. The bottleneck for the majority of algorithms is the small RAM. This was partially remedied by using memory efficient implementations. For some parameters and algorithms, the memory requirements were too high, therefore we were not able to search through all parameter ranges for all parameter. For example when using BIRCH, with small branching factor the CF tree is very deep and has to be held in memory at once.

Sklearn acceleration We used sklearn-intelex acceleration library [85],that patches some sklearn algorithms to take advantage of Intel CPU to decrease runtime. Namely K-means, DBSCAN, PCA and logreg patches are available. The performance increase was tested using a benchmark repository [86]. We have achieved speed-up from 3 to 7 times using different methods.

Chapter 4

Results

In this Chapter, we show baseline results to give a context to the rest of the Chapter, discuss the tuning process and find the clustering that maximizes our metric. We first fix the parameters of feature extraction and choice of clusters to make the initial tuning easier. Two best performing algorithms are selected for further tuning (now tuning all available parameters). The best performing clustering with fine-tuned parameters is used on dataset C to form clusters, which are then qualitatively analysed.

Tuning process By tuning parameters mean iterating over the parameter space to find the best performing combination. Grid search is too resource intensive, since evaluating *one* set of parameters with **5-fold cross-validation** takes from 10 to 50 minutes on our hardware. We iterate over one parameter at a time, then choose the best value. Free parameters are tuned one at a time, until there is no improvement. We usually use a rough step first, and then search the values near the maximal value for greater efficiency. In Figure 4.1, we see a step of 0.05 when iterating on parameter ϵ. Next, we would use smaller step (e.g. 0.02) around maximum. During the whole tuning process, we used 5-fold cross-validation.

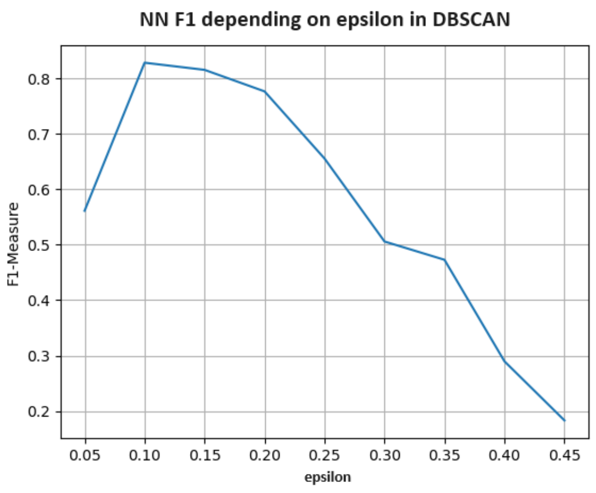


Figure 4.1: Resulting NN F1-measure, when iterating ϵ of DBSCAN algorithm. 0.05 to 0.45 with step of 0.05

Tuned parameters In Chapter 3, list tunable parameters of used models. There are some parameters, which we did not conisder tuning, for example distance metric. Used metric is an important parameter, so why do we not tune it further? We intentionally omit some parameters completely, because the whole model is very complex as is. Each metric would need to have parameters tuned individually, and the amount of work needed would only increase, since there is a great number of possible metrics that we could have considered (Mahalanobis', Minkowski, Chebishev distances, cosine similarity etc.).

We are rather interested in how our evaluation protocol works in practice, how different clustering's properties influence the retrieved clusters and general methodology of tuning the pipeline. Our goal in this Chapter is to illustrate, how we approach the challenge of finding a well performing algorithm with it's parameters.

4.1 Baseline results

In order to put the achieved accuracy in perspective, we show baseline results, which make sense in the modeled situation. We trained the NN (same architecture as the end classifier, Fig. 3.3) on datasets A and B, then tested it on the portion of dataset A not used for training.
To be able to use the neural networks directly, we need to extract features from the training URL sets. Based on our previous experiments, TF-IDF with PCA (output dimension=30), standard-scaler normalisation and precalculated features (expanded in 3.3) perform well. We only adjusted min_df of TF-IDF model based on train set size, to be seen in Table 4.1. We performed **5-fold cross-validation** when training the NN on dataset A. When training on dataset B, we chose 400k datapoints randomly **5 times**, and tested it on whole dataset A.

Using dataset B We claim, that an out-of-date dataset leads to mediocre results when used for classification of a more recent dataset, because of strong concept drift. We trained our NN (in 3.3) in 150 epochs on 400k randomly drawn URLs from dataset B, then predicted classes of dataset A. We'd like the NN trained on the clustered data to perform better than this. If not, it would be better to just use the out-of-date dataset to train a classifier and not use clustering at all.

Using dataset A We then trained the NN on 80% of dataset A, to simulate having correct labels from the current dataset. This is the optimal result, that we would like to get close to with our clustering method. We do not expect to achieve better result than this, since in the very best case, the clusters are 100% pure and cover the whole dataset, which is then labelled by the expert exactly the same as the ground truth. For many uses in cybersecurity, 94 % accuracy is very low and the used NN would need more tuning to obtain viable results.

Trained on	min_df	Epochs	Train size	Test size	**Accuracy**[%]	**F1-measure**
Dataset B	1500	150	400k	110k	70 ± 0.3	0.7 ± 0.01
Dataset A	400	90	88k	22k	94.4 ± 0.1	0.94 ± 0.00

Table 4.1: Baseline results training NN directly on datasets. Predicting labels of dataset A. Mean and deviation over 5 runs.

4.2 Parameter tuning and initial clustering comparison

Our evaluation loop has many free parameters. Some, like NN and ensemble parameters were set beforehand and stay the same during the tuning process. Architecture and settings of used NN is explained in 3.7, and in Figure 3.3. Settings for modeling expert analyst are expanded on in 3.6. Ensemble arcitecture and training is explained in 3.8.

4.2.1 Fixed feature extraction parameters

We mainly investigate parameters related to feature extraction, clustering algorithms and choice of clusters. We first find a good set of parameters for feature extraction, so we can perform initial tuning of the algorithms with fewer free parameters.
Clustering in general suffers from the **curse of dimensionality**, which is a general decrease in performance caused by high dimensional features. TF-IDF followed by PCA, together with hand picked feature set performed the best during initial testing. For normalisation, **min-max** scaling and **zero-mean** scaling worked well, unlike **unit length** normalisation. We use zero-mean as default.
The have to decide between "**word**" and "**char**" tokens to be used in TF-IDF. We extract trigrams and compute features from those. Longer n-grams slow the model and require more memory (exponentially). We select **clusters with largest entropy** during this tuning process, since this method performed well during our initial testing. The **high information** contained in the individual clusters often outweighs the fact, that a large portion of clusters is rejected. Summary of the fixed parameters is in Table 4.2.

Parameter	Fixed value
TF-IDF min_df	100
TF-IDF max_df	0.05
PCA output dimension	35
Choice of clusters	Highest entropy

Table 4.2: Parameters fixed for the initial tuning

4.2.2 Clustering comparison

We want to have an initial comparison to choose 2 algorithms to focus more closely on.
We compiled best results obtained form each clustering method. Mean ± standard deviation of 5-fold cross-validation is in Table 4.3. We decided to take into account mainly NN results, since it performed better than logreg during most of the intial tests.

Algorithm	Parameters	**Accuracy**[%]	**F1-measure**
minibatch K-means	$n = 300$, 150 runs	74.1 ± 5.6	0.72 ± 0.08
DBSCAN	$\epsilon = 0.1$, $min\_pts = 16$	77.6 ± 3.8	0.77 ± 0.04
OPTICS	$\epsilon = 0.51$, $max\_eps = 1.7$ $min\_pts = 12$	68.2 ± 6.4	0.66 ± 0.08
BIRCH	$B = 50$, $T = 1.6$, $n = 500$	71.9 ± 7.1	0.68 ± 0.12
Agglomerative	Ward linkage, $n = 250$	69.5 ± 7.2	0.64 ± 0.10

Table 4.3: NN results with standard deviation (5-fold cross-val)

Some algorithms performance is unreliable, as judged by the standard deviation of results. When evaluating the same set of parameters, widely different outcomes were observed. When using BIRCH with best parameters, the lowest and highest accuracies were 61% and 82% (out of 5 runs in cross-val.). We think less reliable algorithms are hard to use in practice, when we do not have the end accuracy as an indication of success and we are required to use the extracted clusters.

Disadvantages of our approach We are aware of several problems with our tuning approach. We might not have been able to find a better performing set of parameters, since we greedily maximize one parameter at a time and not explore the whole parameter space.
There is a possibility, that some of the parameters fixed for the preliminary tuning (feature extraction, cluster selection criteria or even NN architecture) might naturally boost performance of some algorithms, while disadvantaging others.
Lastly, the results depend on the amount of input data. When we use more data, more features are produced by TF-IDF. This is why we changed *min_df* for ensemble classifiers and baseline results. The clusters are larger and sometimes more mixed, just because there is more samples of different data. This means we might need to fine-tune the

Observations about used algorithms We initially thought, that **OPTICS** will perform better than **DBSCAN**, because OPTICS can extract clusters with different densities. In our experiments, OPTICS lead to worse results overall (Table 4.3). This might be, because there are more parameters to tune and we might not been able to find the best hyper-parameter settings. **DBSCAN** is exceptionally reliant on setting of hyper-parameters. In Figure 4.1, we see results ranging from 20 to 80 % depending on setting of ϵ with all other parameters same. **Single-linkage** agglomerative clustering lead to poor result, because of the effect described in 2.1.5, where unrelated clusters are connected by a chain of points. Out of all algorithms tested, **BIRCH** has the highest spread of results and is therefore most unreliable.

4.3 Best algorithms

Based on the results in Table 4.3, we decided to explore **minibatch K-means** and **DBSCAN**. These two algorithms lead to best results with the smallest std. deviation. We show only resulting accuracy, since F1-measure closely copied accuracy of both classifiers, especially near the best performing parameters. We used NN accuracy for tuning, as previously.

Additionally to previously tuned parameters, we tried using different features (with and without: TF-IDF, PCA, hand-picked feature set), different normalization functions and all proposed criteria for cluster choice. Those are **smallest** clusters, **highest entropy** and **balanced** or **80/0/20** (with the most certainly classified data) using confidence estimates from ensemble classifier, as explained in 3.5.

4.3.1 DBSCAN

When using DBSCAN, the best performing set of feature extraction parameters usually produced around 120-155 clusters, which makes the cluster selection mostly irrelevant. When the clustering produced more than 150 clusters, selecting high entropy clusters yielded the best results (by a small margin). We tried to produce more clusters (higher *min_pts* or lower ϵ) to force the cluster selection to work better. This has lead to slightly worse results and was not used. In Table 4.4, the best performing set of parameters for DBSCAN is noted.

Parameter	Value
ϵ	0.1
min_pts	20
Normalisation	Min-max
PCA output dim.	20
TF-IDF min_df	130
TF-IDF max_df	0.05
Cluster choice	Highest entropy

Table 4.4: Best performing parameters when using DBSCAN

4.3.2 Minibatch K-means

In Table 4.5, we see best performing set of parameters for minibatch K-means.

Parameter	Value
n	320
Inits	150
Normalisation	Min-max
PCA output dim.	70
TF-IDF min_df	350
TF-IDF max_df	0.05
Cluster choice	Balanced

Table 4.5: Best performing parameters when using minibatch K-means

Because K-means can not produce outliers, the clusters were more mixed. K-means did work well with sequential cluster choice using ensemble results. Taking clusters with highest entropy lead to more than half of clusters being rejected. If we do not use PCA, K-means produces one large cluster, which is then rejected. Lastly, we did not see any improvement when doing more than about 150 re-initialisations.

4.3.3 Final comparison

In table 4.6 we see the final accuracies of both end classifiers for both fine-tuned algorithms (parameters in Tables 4.4, 4.5). We can see, that the best performing algorithm is minibatch K-means, with accuracy 87.6 %. These results are expected from our baselines. This final accuracy is better than 70% accuracy gained when training NN on the out-of-date dataset, but not as good as 94 % accuracy of using dataset A directly to train the network.

Algorithm	**NN Accuracy**[%]	**Logreg Accuracy**[%]
DBSCAN	85.1 ± 0.9	84.9 ± 0.9
minibatch K-means	87.6 ± 0.5	87.3 ± 0.4

Table 4.6: Best results obtained with two selected algorithms

Notes on final tuning Near the final set of parameters, std. deviation of the results is small. We think it is because our tuning process prioritised results with small result variance by design. If an algorithm performs unreliably, it tends to have lower average performance, since some of the runs have poor results, even though some runs may produce better results than more stable algorithms. We did not see any results better than 88% accuracy in the tuning process.

4.4 K-means on the raw dataset

The proposed metric is supposed to (indirectly) asses, how well clustering extracts "interesting" groups of data. We think it is beneficial extract and analyze clusters from a *different* dataset, than is used for parameter tuning. We use the best clustering on dataset C for this analysis. We do not expect a huge amount of malicious URLs, since an overwhelming majority of real world data is benign. We could not find any obviously malicious addresses after individually inspecting several thousand URLs. *Note*, that the individual models were not *trained* using dataset A, which was only used for parameter validation.

4.4.1 Selected clusters

We use minibatch K-means with parameters from Table 4.5. We select 150 clusters using an algorithm described in 3.5.1, which targets **balanced** amount of data from every class. We examined every one of the 150 clusters produced to see, if asses resulting cluster homogeneity and if there were any phenomena, that could be interesting from cybersecurity standpoint. We did not find any clusters to be malicious, but there were some suspicious clusters.

Cluster homogeneity

We found most of the clusters to be very homogeneous. Three addresses below were extracted a part of the same cluster (with 238 items). All of the addresses in this cluster look very similar.

```
http://u.goal.com/87200/87240hp2.jpg
http://u.goal.com/138000/138028_thumb.jpg
http://u.goal.com/117900/117996_thumb.jpg
```

From our experience, K-means extract very homogeneous clusters from denser areas of the feature space. This is a little counter intuitive, because there is much more data in those areas, which could get mixed into the clusters, but in reality, there is also more centroids "competing" for the datapoints. In sparser areas of the feature space, one centroids "takes" data from much larger area, which means the data is less similar. We observed similar behavior by examining 2D visualisations of the data. Sorting clusters by the largest distance between any pair of included point could be used as another criterion for cluster selection.

Below, three samples from several chosen clusters are separated by a newline. There is usually some similarity in structure, length and used keywords or domains. Size of the clusters is usually around 200 items

```
http://apir.webrep.avast.com/v1/rating/bearshare.com
http://apir.webrep.avast.com/v1/rating/dailygoodgames.net?623FAC
http://apir.webrep.avast.com/v1/rating/search.conduit.com?1F65135

http://s2.static.tnaflix.com/thumbs/d1/19_163085.jpg
http://s3.static.tnaflix.com/thumbs/f1/2_87286.jpg
http://s3.static.tnaflix.com/thumbs/31/17_38398.jpg

http://ads.trafficjunky.net/ads?zone_id=541&site_id=2&c=
http://ads.trafficjunky.net/ads?zone_id=67&site_id=2&cache=1312323606
http://ads.trafficjunky.net/ads?zone_id=67&site_id=2&cache=1312323546
```

There were some clusters, which were not as homogeneous. Below, four samples are chosen. There is some similarity between the addresses (string 'download' and 'ramadan' inside the URL). Similar clusters would mostly be rejected, because we could not decide on a label by inspecting just a handful of addresses. We think labeling the clusters with a common strings is context sensitive as we would label cluster with sites about Ramadan (especially from Syria) to be benign, unlike cluster with URLs including 'download'. We would say about half of clusters were very homogeneous.

```
http://download725.avast.com/iavs5x/jrog2-29d-29c.vpx
http://cache-download.real.com/free/windows/installer/
    upgradehelper/stub/v8/R61END/RealPlayer.exe
http://spiel.antamar.org/downloads/bilder/screens/simple1_klein.jpg
http://download833.avast.com/cgi-bin/iavs4stats.cgi

http://www.nokiamoon.com/vb/2pderamadan/header/hbg.jpg
http://i1.makcdn.com/images/Ramadan2011/SMS/sms-bg.jpg
http://www.sdeaf.net/vb/ramdan2011/header1.jpg
http://www.supersy.com/vb/w1a1-ramadan1431/body/w1a1-ramadan_05.jpg
```

Only few clusters were completely inconsistent (or we could not find any significant similarity). Samples of one such cluster below.

```
http://cdn.wibiya.com/Toolbars/dir_0810/Toolbar_810101/Loader_810101.js
http://www.silkengirl.com/wp-content/i/shay-laren-16-9747.jpg
http://secure.wlxrs.com/$live.controls.images/cp/RemoveContact.png
```

There were many (>30) clusters of pornographic websites (each hundreds of websites large). Either whole cluster consisting of the same website with different path, or different websites with keyword 'porn' in the name. Other very homogeneous clusters lead to news websites (sports and general news), YouTube, online games (Travian, Kings of Camelot) mail hosting services (my.mail.ru), instant messaging services (iloveim, Facebook), Google

and other frequently used websites. All clusters with hundreds of items, from 100 up to around 600 in the larger clusters.
In cluster below, different subdomains are *probably* used for load-balancing and navigating inside a larger website structure. Some websites use a version of 'ww9' instead of 'www', but we still think the usage is legitimate and not an obfuscation attempt.

```
http://dc17.arabsh.com/i/03101/g7xnl30nxhju.png
http://dc16.arabsh.com/i/03160/9gh3eh2pk8nd.swf
http://dc03.arabsh.com/i/00594/gh4jbndf5ems.gif
```

Suspicious clusters

There are some clusters, that were examined more carefully, since they looked like possible malicious addresses. After thorough examination, we confirmed that the suspicious addresses are benign via VirusTotal [87]. We base our analysis on our previous experience with malicious URLs gained when researching URL based threats(2.4) and examining the other datasets. We tried opening several URLs in a virtual machine, but most websites in the dataset are inactive, probably due to the age of the dataset. Other resources, like base-64 decoder, URL de-escape tool and similar were used during analysis, as well as plain "googling" of various domains.

Malware addresses often use **IP address** as a hostname as a form of **obfuscation**. Below, we can see several addresses from a cluster of 387 URLs, which all use IP in their hostname. Additionally these URLs have very similar path. When observing malicious addresses in dataset A, the URLs often lead to '.sh', '.exe', '.i' and 'mozi.m' (a Chinese botnet [88]) executables. All of the addresses in this cluster lead to '.jpg' files. We found several clusters of IP addresses, all leading to files of format '.gif and '.jpg', with keywords like 'image' and 'gallery'.

```
http://194.187.98.229/am/st/thumbs/099/F2UXj4dFM7.jpg
http://194.187.98.230/am/st/thumbs/910/FgNUvGOpa2.jpg
http://194.187.98.231/am/st/thumbs/678/gUILEOOgMm.jpg
```

Several clusters contained other URLs in the query, usually used for redirection, which can be used in malicious ways. The URL might look legitimate because of a well-known hostname, to then redirect user to a compromised website, by using address hidden in the query.
Below is a very long URL, which contains two URLs in its path. The referred website is legitimate. Redirection is often used to chain different websites together for advertisement tracking. Facebook URLs with redirection are often just on-site links posted by users, leading to outside sources like images and videos.

```
http://d1.openx.org/ajs.php?zoneid=58412&cb=74805325464&charset=utf-8&
loc=http%3A//www.manjam.com/play/history.aspx&referer=http%3A//www.
manjam.com/play/history.aspx
```

In this dataset, when there are these long URLs with redirection, it is usually some kind of content delivery website (e.g. `ad.media-servers.net`, `google-analytics.com`). Large clusters of URLs with many redirections formed, probably because the URL are very long. Redirection URLs are usually not accessed directly by users, but work in the background when fetching content. Some of the URLs use **iframe**, which is a html element allowing referencing other websites. This element can be used by hackers, for example by XSS, especially for redirection. Vast usage of iframe in 2011, when this dataset was gathered is probable. This is another reason these clusters were examined carefully. Example of a very long URL (probably from ad-tracking server) below. We tried decoding the URL as base 64 and de-escaping the characters, in case it was just obfuscated URL, but the results are unintelligible and probably not encoded.

```
http://ad.reachjunction.com/rw?title=&qs=iframe3%3FDt%2DDACAqHAD0DJ0AAAAA
AGbVJgAAAAAAAgAAAAAAAAAAAP8AAAAEAvj%2DKgAAAAAATBYdAAAAAAAd0zIAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAB9wxEAAAAAAAAICAwAAAAAAAAAA
AAAAAAAAAACXCYaFPwAAAAAAAAAAAAADA%2EIvnij8AAAAAAAAAAAAQCKtN5M%2EAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAADv4C%2E6dF2BCq2biwVbdEGVWqagbtC3kQHfcNXn
AAAAAA%3D%3D%2C%2Chttp%253A%252F%252Fdownload%2Em5zn%2Ecom%252F%2CB%253D10
%2526Z%253D0x0%2526%5Fsalt%253D2054558863%2526r%253D1%2526s%253D1845792%25
26y%253D29%2C17d5e582%2Dbd56%2D11e0%2Db08d%2D47cbf7dcc19f
```

Some URLs use escaped characters, but we haven't been able to find a cluster, where it is used for obfuscation. Escaping characters from UTF-8 that are not default to URL format (Japanese, Arabic, emoji etc.) is a legitimate use we have seen often. This sub-string from a query of one of the URLs: `eurl=http%3A%2F%2Fwww%2Epaltalk%2Ecom%2Fperl` can be decoded to `eurl=http://www.paltalk.com/per`.

Tens of clusters with very long URLs, full of escaped characters and redirection URLs were found. We think the length and complexity of path and query is major factor in grouping these together, as well as the exact same hostname. Short Facebook URLs were in different clusters than Facebook clusters with many redirection inside long queries, even though they have the same hostname and possibly path structure. This points to the feature extraction methods being able to distinguish different aspects of the URL, like length, structure, lexical similarities and not just clustering the URLs together because of one similar feature.

Chapter 5

Discussion

5.1 Result summary and interpretation

Quantitative results The achieved accuracy (87.6 %) of best-performing clustering (minibatch K-means) in the evaluation pipeline is consistent with introduced baselines (in Table 4.1). **First**, our method of clustering the data for labeling and then training a classifier results in a better accuracy than using out-of-date datasets to train a classifier (70 %). This is a positive outcome since otherwise, it would be better not to use clustering methods at all and just train a classifier on a different dataset. **Second**, the resulting accuracy does not surpass 94 % accuracy, which we get by training the NN directly on dataset A. This is also not surprising since there is no better data to train on than the same, fully labeled dataset with the same distribution of classes. Notably, DBSCAN's best performance leads to around 85 % accuracy, without using any information gained from dataset B, just by picking high entropy clusters.

Using an ensemble classifier to estimate classification certainty proved helpful since it performed the best out of the cluster selection functions tested. We aimed for a balanced amount of data from all classes. Adding the most certainly classified data directly to train the classifier (and adjusting the balance accordingly) performed slightly worse.

Qualitative results We also wanted to know if there are "interesting" clusters extracted from dataset C, which was not used for parameter tuning. The clustering has been partially successful. Many of the clusters are homogeneous to the point that we can classify the cluster by inspecting a handful of items. Most of the less homogeneous clusters still have some internal similarity, like the same domain, shared sub-strings, and structure (examples are clusters with 'ramadan' and 'download' substrings).
We suspected that some selected clusters might contain cybersecurity threats, but no malicious cluster was confirmed by further examination. We observed vast usage of IP addresses in hostname, long queries with redirections, and escaped characters in the clusters. We used VirusTotal to confirm that the data was benign. Many phenomena present in datasets A and B (like different hostname obfuscations, base-64 encoding in query, and typosquatting) were not found in any clusters from dataset C.

5.2 Is the found clustering good?

The size of dataset C (76k) is *by chance* similar to the size of the portion used during K-fold (88k). The amount of training data does affect feature extraction algorithms and data scaling, which are crucial for clustering. Clustering algorithms are affected by the amount of input data by design since we need the context of other data to form clusters. In this thesis, we did not focus on accommodating different-sized datasets. Therefore, we do not know what would happen if dataset C had (much) more or fewer items. The resulting clusters are mostly coherent, and some interesting phenomena were found. We think the results are promising, but the analysis of dataset C is not enough to prove the usability of the clustering algorithm with certainty.

As for the quantitative analysis, note that the best set of results obtained by minibatch K-means apply only to dataset A, on which the clustering parameters were validated. We did not measure these metrics on datasets not used for validation since we do not know the labels of dataset C. If we were to deploy the resulting clustering with real analyst, the success of deployed classifiers may differ. Accuracy inside the evaluation loop is supposed to be taken as metric used for tuning and approximate measurement of usefulness (in similar situations), it is not the final performance on real world dataset with real human analyst, which is hard to asses, when the ground truth is not known.

We think our hyper-parameter search method has been successful (with the limitations of our hardware), but a more thorough method (like grid-search) might find better performing clustering algorithms. The found clustering is adequate for our purposes, based on the resulting accuracy in the evaluation pipeline and cluster extracted from dataset C.

5.3 Does the proposed metric work?

We think the proposed metric's success in an specific situation relies on properly tuning and modeling all different parts of the evaluation loop. By using "stronger" end classifier, we could obtain better results, without changing anything about the actual clustering algorithm. This means, that just the accuracy (and F1-measure) alone is not sufficient description of clustering's usefulness. We have to carefully consider what end classifier we use, class balance and sizes of all datasets involved and how the labeling process is modeled. There are many clustering algorithms, feature extraction models and cluster selection functions not considered, which may achieve better result than our best clustering.

Intuitively, if the individual parts of the evaluation loop correspond closely to the real use-case, then the resulting accuracy more closely reflects real performance when deployed "in the wild". We don't think the qualitative results are enough to prove, that the proposed method works, but we think the extracted clusters were interesting and mostly coherent (homogeneous) and we were partially successful, since this clustering method has been tuned without knowing anything about dataset C, or having a notion of what "good" clusters are, just by improving the accuracy of the end classifier.

5.4 Assignment fulfillment

We want to review individual points of the assignment to make sure the thesis meets the requirements for completion.

1: "*Review existing literature on the use of clustering for reducing workload of network security analyst.*" We go in depth into different ways how clustering is used in classification pipelines to help analysts in 2.1.2.

2: "*Propose a metric usable to compare the quality of clusterings constructed for this purpose.*" We proposed an evaluation protocol to asses how well a clustering performs, when the clusters are labeled by analyst and a classifier is trained on the labeled data. The metric in question is the accuracy and F1-measure of the end classifier(s).

3: "*Implement a method that will cluster network security samples in order to (not necessary directly) optimize this metric.*" We implemented the evaluation protocol, chose representative clustering algorithms and several cluster selection functions. We explain how we found the best clustering (w.r.t. the proposed metric) in Chapter 4.

4: "*Find a suitable dataset of network security data, for example accessed URLs.*" We use three datasets in the thesis. Current dataset (gathered from different sources), out-of-date dataset (from Kaggle) and an unfiltered real-world dataset (extracted from found server log data). The datasets and gathering process are described in 3.2.

5: "*Compare the implemented clustering method to baselines w.r.t. the proposed metric.*" We show several baselines, which make sense in the modeled situation. We also show results obtained on all tested clustering algorithms, not just the best results. This can give some idea on how much the accuracy varies between different sets of parameters and used algorithms.

6: "*Qualitatively assess the usefulness of the created clustering for manual analysis.*" We present qualitative and quantitative results in Chapter 4. Quantitative results pertain to best accuracy achieved on the validation dataset (current dataset). Qualitative analysis was conducted on the real-world data. We found some interesting clusters. Most of the clusters have some internal similarity. We discuss the results in this Chapter.

Chapter 6

Conclusion

In this Chapter, we conclude the thesis. We first summarize the goal, our approach, and results and then recommend future work and list the contributions of the thesis.

Summary The thesis aims to analyze the use of clustering to reduce the workload of network security analysts. We were tasked with finding a metric to assert clustering's usefulness for manual analysis. We chose to model a common situation where we need to train a classifier to be used on very recent data for which we do not have labels, but older, fully labeled datasets of the same type are available. Clustering is used to speed up manual classification by a human analyst. We model this situation as best as possible using an evaluation pipeline, from which the clustering usefulness metric is derived. We cluster the data and select the best clusters to be manually labeled by an analyst. A classifier is then trained on the labeled subset of the data. The accuracy of the classifier is used as a metric of clustering's usefulness. We explore our approach on selected URL datasets.

We analyzed every step of the evaluation pipeline to find viable models and implementations. Chosen clustering algorithms were then fine-tuned to maximize the performance in terms of the proposed metric. Finally, this algorithm was used on the different datasets, and the extracted clusters were analyzed to assert whether maximizing the proposed metric leads to good clusters.

A notable approach for cluster selection uses an ensemble classifier and an older dataset to infer classes of the new data, which are used (with estimated classification certainty) to predict cluster labels. Some clusters are then selected based on these labels for further analysis.

Results Clustering the recent dataset using minibatch K-means and then selecting clusters sequentially by using prediction and certainty estimates from an ensemble classifier leads to the best performance. The clustering method achieved 87.6 % accuracy on the test portion of the recent dataset using a neural network. This is better than using the older dataset directly for training, which achieved 70 % accuracy.

This clustering was then used on a dataset of 76k samples extracted from leaked server logs. We analyzed the clusters to find any interesting phenomena. There were no malicious clusters present, but this could have been caused by a lack of malicious data in the dataset. Most of the extracted clusters had high internal similarity, and we would describe them as homogeneous. Many could have been labeled by inspecting a handful of selected items, which is desirable. Very few clusters had no common features (these would have been rejected when presented for labeling). We think this is a success, but more results (on different-sized datasets with different class balances) are needed to prove that the metric works well in different settings. We cannot reject the possibility that the clustering performed well on the real-world dataset simply because of similar dataset sizes.

Conclusion The proposed metric leads to promising results, both in terms of resulting accuracy and extracted clusters. We think the metric captures some notion of clustering suitability for manual analysis based on extracted clusters, which were mostly coherent and could be labeled together. The best clustering has achieved an 87.6 % accuracy in our evaluation protocol, which means (with our datasets) it is better to cluster recent data and select some clusters for labeling (using the older dataset to gain some heuristic) than use an older dataset directly for training a classifier.

We feel the need to point out that we decided to model manual analysis by using ground truth labels of the data to infer a class of the clusters. We found this to be a sufficient approximation of the process, but the resulting accuracy should be viewed in this context. The resulting accuracy might differ if we manually analyze the clusters or use another labeling algorithm.

Future work It is always possible to test other feature extraction techniques, clustering algorithms, or cluster selection techniques to achieve better accuracy in the evaluation protocol. If we were to deploy the resulting clustering algorithm in a commercial product, further experiments would be needed since countless algorithms can be combined and tested.

We think developing our cluster selection strategy into an active learning pipeline is a valid research area. Focusing on sequential cluster selection with the knowledge of labels from the analyst could lead to better accuracy of the end classifier. We could focus on improving the ensemble classifier prediction by weighing the estimated certainty (giving more advantage to very certain predictions) and using better or more of weak classifiers in order to predict the class of the clusters better. Lastly, more tests on different datasets should be done to confirm the validity of our method in different contexts.

Contribution We think formalization of the used methodology, from the idea behind the proposed metric to the selected implementation and parameter tuning, is desirable. Our metric is more of a framework in which different models can be tested to find the best-performing combination to deploy in similar situations. Additionally, we analyze every step of the evaluation protocol in the context of URL classification to identify viable models.

Great effort was made to analyze available data sources to gather our validation dataset. Recent, publicly available datasets of URL data are scarce, and an overview of available sources is convenient.

Bibliography

[1] João Gama et al. “A survey on concept drift adaptation”. In: *ACM computing surveys (CSUR)* 46.4 (2014), pp. 1–37.

[2] Shi Zhong, Taghi M Khoshgoftaar, and Naeem Seliya. “Clustering-based network intrusion detection”. In: *International Journal of reliability, Quality and safety Engineering* 14.02 (2007), pp. 169–187.

[3] Sampashree Nayak, Deepak Nadig, and Byrav Ramamurthy. “Analyzing malicious urls using a threat intelligence system”. In: *2019 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS)*. IEEE. 2019, pp. 1–4.

[4] Mohammad Saiful Islam Mamun et al. “Detecting malicious urls using lexical analysis”. In: *Network and System Security: 10th International Conference, NSS 2016, Taipei, Taiwan, September 28-30, 2016, Proceedings 10*. Springer. 2016, pp. 467–482.

[5] Guofei Gu et al. “Botminer: Clustering analysis of network traffic for protocol-and structure-independent botnet detection”. In: (2008).

[6] Kuai Xu, Zhi-Li Zhang, and Supratik Bhattacharyya. “Internet traffic behavior profiling for network security monitoring”. In: *IEEE/ACM Transactions On Networking* 16.6 (2008), pp. 1241–1252.

[7] Swaswati Goswami et al. “An Unsupervised Method for Detection of XSS Attack.” In: *Int. J. Netw. Secur.* 19.5 (2017), pp. 761–775.

[8] Tzy-Shiah Wang et al. “DBod: Clustering and detecting DGA-based botnets using DNS traffic analysis”. In: *Computers & Security* 64 (2017), pp. 1–15.

[9] Vladimir Estivill-Castro. “Why so many clustering algorithms: a position paper”. In: *ACM SIGKDD explorations newsletter* 4.1 (2002), pp. 65–75.

[10] Anil K Jain. “Data clustering: 50 years beyond K-means”. In: *Pattern recognition letters* 31.8 (2010), pp. 651–666.

[11] Absalom E Ezugwu et al. “A comprehensive survey of clustering algorithms: State-of-the-art machine learning applications, taxonomy, challenges, and future research prospects”. In: *Engineering Applications of Artificial Intelligence* 110 (2022), p. 104743.

[12] Rong-Fang Xu and Shie-Jue Lee. “Dimensionality reduction by feature clustering for regression problems”. In: *Information Sciences* 299 (2015), pp. 42–57.

[13] Mariusz Frackiewicz and Henryk Palus. “KM and KHM clustering techniques for colour image quantisation”. In: *Computational vision and medical image processing: recent trends* (2011), pp. 161–174.

[14] Laurent Hoeltgen, Pascal Peter, and Michael Breuß. “Clustering-based quantisation for PDE-based image compression”. In: *Signal, Image and Video Processing* 12 (2018), pp. 411–419.

[15] Odilia Yim and Kylee T Ramdeen. “Hierarchical cluster analysis: comparison of three linkage measures and application to psychological data”. In: *The quantitative methods for psychology* 11.1 (2015), pp. 8–21.

[16] Fred H Borgen and David C Barnett. "Applying cluster analysis in counseling psychology research." In: *Journal of counseling psychology* 34.4 (1987), p. 456.

[17] Jin Hwan Do and Dong-Kug Choi. "Clustering approaches to identifying gene expression patterns from DNA microarray data". In: *Molecules and cells* 25.2 (2008), pp. 279–288.

[18] Mete Çelik, Filiz Dadaşer-Çelik, and Ahmet Şakir Dokuz. "Anomaly detection in temperature data using DBSCAN algorithm". In: *2011 international symposium on innovations in intelligent systems and applications*. IEEE. 2011, pp. 91–95.

[19] Robert Rentea and Ciprian Oprişa. "Fast clustering for massive collections of malicious URLs". In: *2021 IEEE 17th International Conference on Intelligent Computer Communication and Processing (ICCP)*. IEEE. 2021, pp. 11–18.

[20] Richard O Duda, Peter E Hart, et al. *Pattern classification*. John Wiley & Sons, 2006.

[21] Abiodun M Ikotun et al. "K-means clustering algorithms: A comprehensive review, variants analysis, and advances in the era of big data". In: *Information Sciences* 622 (2023), pp. 178–210.

[22] José M Pena, Jose Antonio Lozano, and Pedro Larranaga. "An empirical comparison of four initialization methods for the k-means algorithm". In: *Pattern recognition letters* 20.10 (1999), pp. 1027–1040.

[23] Ali Feizollah et al. "Comparative study of k-means and mini batch k-means clustering algorithms in android malware detection using network traffic analysis". In: *2014 international symposium on biometrics and security technologies (ISBAST)*. IEEE. 2014, pp. 193–197.

[24] David Sculley. "Web-scale k-means clustering". In: *Proceedings of the 19th international conference on World wide web*. 2010, pp. 1177–1178.

[25] Hans-Peter Kriegel et al. "Density-based clustering". In: *Wiley interdisciplinary reviews: data mining and knowledge discovery* 1.3 (2011), pp. 231–240.

[26] Erich Schubert et al. "DBSCAN revisited, revisited: why and how you should (still) use DBSCAN". In: *ACM Transactions on Database Systems (TODS)* 42.3 (2017), pp. 1–21.

[27] *scikit-learn: machine learning in Python; scikit-learn 1.3.2 documentation — scikit-learn.org*. `https://scikit-learn.org/stable/index.html`. [Accessed 16-01-2024].

[28] Mihael Ankerst et al. "OPTICS: Ordering points to identify the clustering structure". In: *ACM Sigmod record* 28.2 (1999), pp. 49–60.

[29] https://scikit-learn.org/stable/auto$_e$*xamples/cluster/plot*$_o$*ptics.html*. [Accessed 18-01-2024].

[30] Erich Schubert and Michael Gertz. "Improving the Cluster Structure Extracted from OPTICS Plots." In: *LWDA*. 2018, pp. 318–329.

[31] Frank Nielsen and Frank Nielsen. "Hierarchical clustering". In: *Introduction to HPC with MPI for Data Science* (2016), pp. 195–211.

[32] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. "BIRCH: an efficient data clustering method for very large databases". In: *ACM sigmod record* 25.2 (1996), pp. 103–114.

[33] Tim Berners-Lee, Larry Masinter, and Mark McCahill. *Uniform resource locators (URL)*. Tech. rep. 1994.

[34] Roy Fielding. "Hypertext transfer protocol". In: *RFC 2616* (1999).

[35] Aditya K Sood and Sherali Zeadally. “Drive-by download attacks: A comparative study”. In: *It Professional* 18.5 (2016), pp. 18–25.

[36] Van Lam Le et al. “Anatomy of drive-by download attack”. In: *Proceedings of the Eleventh Australasian Information Security Conference-Volume 138*. 2013, pp. 49–58.

[37] Rabia Tahir. “A study on malware and malware detection techniques”. In: *International Journal of Education and Management Engineering* 8.2 (2018), p. 20.

[38] Rachna Dhamija, J Doug Tygar, and Marti Hearst. “Why phishing works”. In: *Proceedings of the SIGCHI conference on Human Factors in computing systems*. 2006, pp. 581–590.

[39] Maryam Feily, Alireza Shahrestani, and Sureswaran Ramadass. “A survey of botnet and botnet detection”. In: *2009 Third International Conference on Emerging Security Information, Systems and Technologies*. IEEE. 2009, pp. 268–273.

[40] Manos Antonakakis et al. “From {Throw-Away} Traffic to Bots: Detecting the Rise of {DGA-Based} Malware”. In: *21st USENIX Security Symposium (USENIX Security 12)*. 2012, pp. 491–506.

[41] Sujata Garera et al. “A framework for detection and measurement of phishing attacks”. In: *Proceedings of the 2007 ACM workshop on Recurring malcode*. 2007, pp. 1–8.

[42] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[43] Nitish Srivastava et al. “Dropout: a simple way to prevent neural networks from overfitting”. In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[44] Li Wan et al. “Regularization of neural networks using dropconnect”. In: *International conference on machine learning*. PMLR. 2013, pp. 1058–1066.

[45] Wei Wei et al. “Accurate and fast URL phishing detector: a convolutional neural network approach”. In: *Computer Networks* 178 (2020), p. 107275.

[46] Hung Le et al. “URLNet: Learning a URL representation with deep learning for malicious URL detection”. In: *arXiv preprint arXiv:1802.03162* (2018).

[47] Rakesh Verma and Avisha Das. “What’s in a url: Fast feature extraction and malicious url detection”. In: *Proceedings of the 3rd ACM on International Workshop on Security and Privacy Analytics*. 2017, pp. 55–63.

[48] Apoorva Joshi et al. “Using lexical features for malicious URL detection–a machine learning approach”. In: *arXiv preprint arXiv:1910.06277* (2019).

[49] Justin Ma et al. “Learning to detect malicious urls”. In: *ACM Transactions on Intelligent Systems and Technology (TIST)* 2.3 (2011), pp. 1–24.

[50] Jatin Acharya et al. “Detecting malware, malicious URLs and virus using machine learning and signature matching”. In: *2021 2nd International Conference for Emerging Technology (INCET)*. IEEE. 2021, pp. 1–5.

[51] Juan Ramos et al. “Using tf-idf to determine word relevance in document queries”. In: *Proceedings of the first instructional conference on machine learning*. Vol. 242. 1. New Jersey, USA. 2003, pp. 29–48.

[52] David G Underhill et al. “Enhancing text analysis via dimensionality reduction”. In: *2007 IEEE international conference on information reuse and integration*. IEEE. 2007, pp. 348–353.

[53] Leland McInnes, John Healy, and James Melville. "Umap: Uniform manifold approximation and projection for dimension reduction". In: *arXiv preprint arXiv:1802.03426* (2018).

[54] Margherita Grandini, Enrico Bagli, and Giorgio Visani. "Metrics for multi-class classification: an overview". In: *arXiv preprint arXiv:2008.05756* (2020).

[55] Md Manjurul Ahsan et al. "Effect of data scaling methods on machine learning algorithms and model performance". In: *Technologies* 9.3 (2021), p. 52.

[56] Jiawei Han, Jian Pei, and Hanghang Tong. *Data mining: concepts and techniques.* Morgan kaufmann, 2022.

[57] Terrance DeVries and Graham W Taylor. "Learning confidence for out-of-distribution detection in neural networks". In: *arXiv preprint arXiv:1802.04865* (2018).

[58] Hongxin Wei et al. "Mitigating neural network overconfidence with logit normalization". In: *International conference on machine learning.* PMLR. 2022, pp. 23631–23644.

[59] Chuan Guo et al. "On calibration of modern neural networks". In: *International conference on machine learning.* PMLR. 2017, pp. 1321–1330.

[60] Eyke Hüllermeier and Willem Waegeman. "Aleatoric and epistemic uncertainty in machine learning: An introduction to concepts and methods". In: *Machine learning* 110.3 (2021), pp. 457–506.

[61] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. "Simple and scalable predictive uncertainty estimation using deep ensembles". In: *Advances in neural information processing systems* 30 (2017).

[62] Tim Pearce et al. "Uncertainty in neural networks: Bayesian ensembling". In: *stat* 1050 (2018), p. 12.

[63] Michael Muhlbaier, Apostolos Topalis, and Robi Polikar. "Ensemble confidence estimates posterior probability". In: *International Workshop on Multiple Classifier Systems.* Springer. 2005, pp. 326–335.

[64] Yarin Gal and Zoubin Ghahramani. "Dropout as a bayesian approximation: Representing model uncertainty in deep learning". In: *international conference on machine learning.* PMLR. 2016, pp. 1050–1059.

[65] Burr Settles. "Active learning literature survey". In: (2009).

[66] Manu Siddhartha. *Malicious URLs dataset — kaggle.com.* `https://www.kaggle.com/datasets/sid321axn/malicious-urls-dataset`. [Accessed 16-01-2024].

[67] *So we herd u liek logz | Telecomix — bluesmote.com.* `http://bluesmote.com/`. [Accessed 25-04-2024].

[68] *URL 2016 | Datasets | Research | Canadian Institute for Cybersecurity | UNB — unb.ca.* `https://www.unb.ca/cic/datasets/url-2016.html`. [Accessed 16-01-2024].

[69] *URLhaus | API — urlhaus.abuse.ch.* `https://urlhaus.abuse.ch/api/#csv`. [Accessed 22-04-2024].

[70] *GitHub - mitchellkrogza/Phishing.Database: Phishing Domains, urls websites and threats database. We use the PyFunceble testing tool to validate the status of all known Phishing domains and provide stats to reveal how many unique domains used for Phishing are still active. — github.com.* `https://github.com/mitchellkrogza/Phishing.Database/tree/master`. [Accessed 22-04-2024].

[71] *PhishTank &gt; Developer Information — phishtank.com.* `https://www.phishtank.com/developer_info.php`. [Accessed 16-01-2024].

[72] AK Singh. "Malicious and benign webpages dataset". In: *Data in brief* 32 (2020), p. 106304.

[73] Marek Pawlicki et al. "On the impact of network data balancing in cybersecurity applications". In: *Computational Science–ICCS 2020: 20th International Conference, Amsterdam, The Netherlands, June 3–5, 2020, Proceedings, Part IV 20*. Springer. 2020, pp. 196–210.

[74] *PhishStorm - phishing / legitimate URL dataset — research.aalto.fi*. `https://research.aalto.fi/en/datasets/phishstorm-phishing-legitimate-url-dataset`. [Accessed 16-01-2024].

[75] Samuel Marchal et al. "PhishStorm: Detecting phishing with streaming analytics". In: *IEEE Transactions on Network and Service Management* 11.4 (2014), pp. 458–471.

[76] *Using-machine-learning-to-detect-malicious-URLs/data at master · faizann24/Using-machine-learning-to-detect-malicious-URLs — github.com*. `https://github.com/faizann24/Using-machine-learning-to-detect-malicious-URLs/tree/master/data`. [Accessed 16-01-2024].

[77] Abdelberi Chaabane et al. "Censorship in the wild: Analyzing Internet filtering in Syria". In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. 2014, pp. 285–298.

[78] Hyunsang Choi, Bin B Zhu, and Heejo Lee. "Detecting malicious web links and identifying their attack types". In: *2nd USENIX Conference on Web Application Development (WebApps 11)*. 2011.

[79] Tomas Mikolov et al. "Distributed representations of words and phrases and their compositionality". In: *Advances in neural information processing systems* 26 (2013).

[80] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[81] Wisam A Qader, Musa M Ameen, and Bilal I Ahmed. "An overview of bag of words; importance, implementation, applications, and challenges". In: *2019 international engineering conference (IEC)*. IEEE. 2019, pp. 200–204.

[82] Leonid Portnoy. "Intrusion detection with unlabeled data using clustering". PhD thesis. Columbia University, 2000.

[83] Ibrahem Kandel and Mauro Castelli. "The effect of batch size on the generalizability of the convolutional neural networks on a histopathology dataset". In: *ICT express* 6.4 (2020), pp. 312–315.

[84] Fredrik K Gustafsson, Martin Danelljan, and Thomas B Schon. "Evaluating scalable bayesian deep learning methods for robust computer vision". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops*. 2020, pp. 318–319.

[85] *scikit-learn-intelex — pypi.org*. `https://pypi.org/project/scikit-learn-intelex/`. [Accessed 28-04-2024].

[86] *GitHub - IntelPython/scikit-learn_bench*. `https://github.com/IntelPython/scikit-learn_bench`. [Accessed 28-04-2024].

[87] *VirusTotal — virustotal.com*. `https://www.virustotal.com/gui/url/`. [Accessed 07-05-2024].

[88] Teng-Fei Tu et al. "A comprehensive study of Mozi botnet". In: *International Journal of Intelligent Systems* 37.10 (2022), pp. 6877–6908.

[89] Jan Svoboda. *GitLab — gitlab.fel.cvut.cz.* `https://gitlab.fel.cvut.cz/svobo114/bp_clustering`. [Accessed 16-05-2024]. 2024.

Appendix A

Code structure

This appendix shows files and structure of the provided .zip file containing the python code used during the thesis, generated by Windows `tree \f` command.

Directory structure

```
ensembles.ipynb
final.ipynb
iterating.ipynb

/classes
    ensemble.py
    expert.py
    fe.py
    km.py
    loader.py
    neural_net.py
    record_class.py
    visuals.py

/results
    25.txt
    5.txt
    60.txt

/scripts
    features.py
    functions.py
    prints.py
```

Files `/results` contain samples of the clusters extracted from the dataset C. Number in the filename means how many samples are shown for each cluster, 5 being the most concise.

`ensemble.ipynb` jupyter notebook contains code used for testing the selected ensemble classifier implementations. `final.ipynb` is used for clustering dataset C using the best performing clustering algorithm and inspect the results. `iterating.ipynb` is the main notebook, in which the code for fine tuning the clustering parameters is written.

`/classes` and `/scripts` contain implementation of various models and standalone functions used in the jupyter notebooks.

The attachment is missing several important files, like all three datasets, saved configurations and images, and conda environment file. The main purpose of the attached code is showcasing our work and not for reproduction. The other files can be found at [89], stored in the authors git repository. The git repository is not intended as the main code documentation.