A Digital Media Similarity Measure for Triage of Digital Forensic Evidence

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Myeong Lyel Lim
Master of Science
The Pennsylvania State University, 1992
Bachelor in Economics
Korea Univsersity, 1984

Director: Dr. James H. Jones, Jr., Professor
Department of Information Technology

Fall Semester 2020
George Mason University
Fairfax, VA

# Dedication

First, I dedicate this dissertation to my parents. I would not be the person I am today if it was not for them. Second, I dedicate this to my wife, Miyoung Lee, for her unfaltering support and patience while I was avoiding completing this thesis. Third, I dedicate this to my two sons, Jongoh and Jongyeop. I hope they understand one day why dad spent lots of his time on the computer at the old age of his life.

# Acknowledgments

I would like to thank the following people who made this possible: My advisor at George Mason University, Dr. Jones, whose insight and knowledge into the subject matter steered me through this research.

My wife, Miyoung Lee, whose support was essential for me to complete this Doctoral program.

I would like to thank my son Jongyeop for proofreading this dissertation during its compilation.

Finally, I would like to thank my co-worker at Fannie Mae, Dr. Andrei Chetcheprov, who reviewed my paper with good comments.

# Table of Contents

# List of Tables

# List of Figures

# Abstract

A DIGITAL MEDIA SIMILARITY MEASURE FOR TRIAGE OF DIGITAL FORENSIC EVIDENCE

Myeong Lyel Lim, PhD

George Mason University, 2020

Dissertation Director: Dr. James H. Jones, Jr.

As the volume of potential digital evidence increases, digital forensics investigators are challenged to find the best allocation of their limited resources. While automation will continue to partially mitigate this problem, the preliminary question of which media should be examined by human or machine remains largely unsolved.

Prior work has established various methods to assess digital media similarity which may aid in prioritization decisions. Similarity measures may also be used to establish links between media, and by extension, links between the individuals or organizations associated with that media. Existing similarity measures, however, have high computational costs which can delay identification of digital media warranting immediate attention or render link establishment across large collections of data impractical.

In this work, I propose, develop, and validate a methodology for assessing digital media similarity to assist with digital media triage decisions. The application of my work is predicated on the idea that unexamined media is likely to be relevant and interesting to an investigator if this unexamined media is similar to other media previously determined to be interesting and relevant. My methodology builds on prior work using sector hashing and the Jaccard index similarity measure. I combine these methods in a novel way and

demonstrate the accuracy of my method against a test set of hard disk images with known ground truth. My method is called Jaccard Index with Normalized Frequency (**JINF**) and calculates the similarity measure between two disk images by normalizing the frequency of the distinct sectors.

I also developed and tested two extensions to improve performance. The first extension randomly samples sectors from digital media under examination and applies a modified **JINF** method. I demonstrate that the JINF disk similarity measure remains useful with sampling rates as low as 5%. The second extension takes advantage of parallel processing. The method distributes the computation across multiple processors after partitioning the digital media, then it combines the results into an overall similarity measure which preserves the accuracy of the original method on a single processor. Experimental results provided as much as a 51% reduction in processing time.

My work goes beyond interesting file and file fragment matching; rather, I assess the overall similarity of digital media to identify systems which might share applications and user content, and hence be related, even if some common files of interest are encrypted, deleted, or otherwise not available. In addition to triage decisions, digital media similarity may be used to infer links and associations between the disparate entities owning or using the respective digital devices.

# Chapter 1: Introduction

Digital forensics analysts, examiners, and investigators process and extract evidence from digital sources and media, often but not exclusively to support the investigation of a crime. Digital evidence is both fragile and volatile. It requires the attention of a trained specialist to ensure that content of evidentiary value can be effectively isolated and extracted in a forensically sound manner that will bear the scrutiny of a court of law. One of the roles of the digital forensic analyst is to find supporting evidence by recovering data such as files, e-mails, and photos from computer hard drives and other data storage and processing devices such as cell phones, flash drives, the RAM of such devices, and network flows. The popularity and increasing use of cloud computing has expanded the scope of storage to multiple geographically dispersed machines, and systems such as game consoles, IoT devices, and embedded systems are increasing proving useful in digital forensic investigations. As more digital data is created and digital storage systems grow in size, forensic analysts are overwhelmed by the sheer volume of data to be analyzed, and backlogs in digital forensic labs are common. More than 15,000 digital devices and storage media were previewed and six petabytes of data were processed by the FBI in the year 2017, according to the FBI Regional Computer Forensics Laboratory report, and several local offices of the FBI Regional Computer Forensics Laboratories set reducing backlog as an explicit goal [1]. In 2018, the Digital Forensics unit of the DHS Cyber Crime Center alone processed seven petabytes of data [1]. Even with automated tools like EnCase [2], FTK, and Autopsy, additional human review time is required before the drive is fully analyzed. With limited time, forensic analysts must pick and choose which digital media to review among the many available, making media triage a necessity. While triage tools exist for explicit tasks like finding substrings of

[1] Private communication with a researcher doing work with the Cyber Crime Center.

interest or specific files, I propose a more general purpose triage method based on a similarity measure between arbitrary size content and a labeled collection of digital media images. For example, a hard disk image which shows high similarity against a cluster of previously collected and labeled disk images of interest can be prioritized for further analysis, and this similarity may be used to infer relationships between entities as well as serving as the basis for examination of additional media.

On top of introducing a novel method for digital media similarity measurement, I developed methods to improve the performance of the basic algorithm by random sampling and splitting drives for parallel computation.

## 1.1 Problem statement

Digital forensic examiners seek to prioritize data sources to be analyzed given limited time, human resources, and computing resources. Manual and forensic-tool-based analyses may take many hours to complete for each data source. Prior work has established various computational methods to assess digital media similarity which may aid in prioritization decisions, e.g., a hard drive similar to one used by a known criminal or terrorist organization may be more likely to contain probative data or actionable intelligence.

Existing forensic tools and human analysis may take many hours to complete for each data source. Analysts often do not have information upon which to make a decision about which media to work on first, something that can only be determined by spending valuable time and resources on each candidate source. The lack of efficient tools and lack of knowledge about potential evidence on a device causes inefficiencies and can lead to critical deadlines being missed or cause delays in the dissemination of actionable intelligence.

Forensic analysis work is often time-sensitive and investigators have limited time and resources, so early triage and prioritization of digital evidence is necessary.

I propose a digital media similarity measure based on sector hashing and a variant of the Jaccard index to help address this problem. This measure will improve the efficiency and accuracy of forensic analysts by quickly and accurately measuring the similarity of

unexamined digital media to other images which are known to be interesting and relevant. A similar image is more likely to contain evidence of interest and may be used to discover previously unknown links between entities.

## 1.2 Use cases

The two use cases I consider are:

1. I've got a bunch of digital media. Are any of these items similar to media I've seen before and which of these do I care about?
2. I've got a bunch of digital media and I know where it came from. Are any of these items similar to each other? Can such similarity tell me anything about connections between people or devices?

Breitinger et al. [3] states that resemblance (R) and containment (C) are two common types of similarity queries and a similarity identification algorithm should be able to handle one of the following four cases: Object similarity detection (R), Cross correlation (R), Embedded object detection (C), and Fragment detection (C). My use cases are similarity detection and cross correlation between hard disk images. My method does not require two digital evidence sources to be of the same size. I consider use cases from a forensics shop, such as one that might be used by law enforcement or national security organizations. In either case, the analysts may or may not know the source of each evidentiary item and its relationship to a current investigation.

## 1.3 Chapter outline

The following chapters discuss several of the design issues for implementing sector hashes for forensic triage analysis. Chapter 2 provides technical background such as Bloom filters, hashing, file formats, and similarity measures. Chapter 3 discusses prior work about similarity measure algorithms. Chapter 4 introduces Jaccard index with frequency (**JIWF**) and Jaccard index with normalized frequency (**JINF**) methods. **JIWF** is based on Jaccard

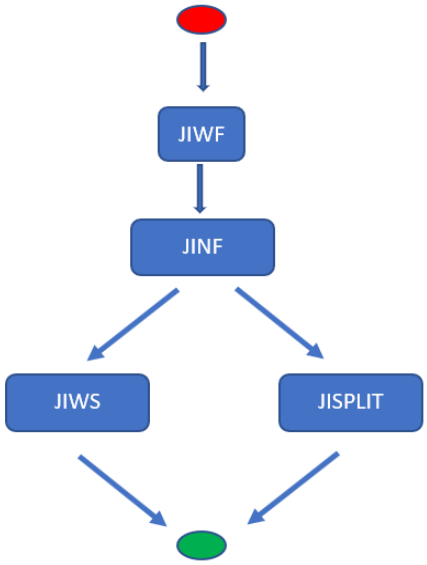


Figure 1.1: Flow of the research.

index and sector hashing. **JINF** is designed to resolve the issue of **JIWF** coming from the size difference between two disk images. **JINF** is the primary contribution of my research, and the two extensions are based on this.

Chapter 5 discusses the validation of both **JIWF** and **JINF** using a set of disk images with known similarity. Comparison between these two methods is also discussed in this chapter. Chapter 6 presents my extension called the Jaccard Index with Sample (**JIWS**). It is modified from **JINF**, which changes linearly with sample rate and also depends on the number of common hashes between two images. Chapter 7 introduces a parallel computing environment (ARGO) at George Mason University. The ARGO platform was used to implement the parallel computation method and experiments. Chapter 8 presents my other extension called the Jaccard Index with Split (**JISPLIT**) which takes advantage of the parallel processing to reduce the overall processing time. Input media is partitioned, distributed to each processing unit of the parallel platform, and the partial results are then combined in one processing node. Chapter 9 concludes the dissertation, discusses the limitations of current implementations, and presents future work.

As shown in Figure 1.1, **JINF**, which is improved from **JIWF**, is the fundamental method in my dissertation. The extensions **JIWS** and **JISPLIT** are independently developed but based on **JINF**.

The major contributions of this thesis are as follows: (1) A novel sector hash based similarity measure between two disk images (**JINF**). (2) Two extensions of **JINF** to improve performance: sampling (**JIWS**) and splitting (**JISPLIT**).

In conclusion, these methods will provide forensic analysts a fast media triage and prioritization capability, as well as a means to identify previously unknown associations between digital media and the entities using the devices containing that media.

# Chapter 2: Technical Background

This section introduces technical background information about cryptographic hash functions, file hashing, and sector hashing. In the forensic area, cryptographic hash functions and sector hashes are used to identify evidence in traditional digital evidence processing. In this section, both methods are explained to help with understanding the remaining sections. The advantages and limitations of these methods are also discussed.

I also discuss well-known similarity measures, most of which measure similarity as a distance as well as a Jaccard index.

A cryptographic hash function is a hash function that tries to guarantee a number of security properties such that it is hard to find collisions or pre-images, and that the output appears random. It computes a fixed size output for an arbitrary length input, where changes in the input have unpredictable and equally significant effects on the output regardless of the scope or nature of the input change. **Sector hashing** is the process of computing hashes for data stored in digital media sectors, and **File hashing** is the process of computing hashes for data stored in digital media files. One advantage of sector hashing vs. file hashing is that sector hashing does not require file system interpretation, nor does it require the entire file to be present for the presence of common data to be inferred. In general, finding more content from a file increases the likelihood that the full file was, in fact, previously present and facilitates analysis, but recovering whole files is not necessary to conduct triage and useful analysis. In my work, matched sectors may be the result of deleted and partially overwritten data as well as other activity (temporary files, swap space files, etc.).

## 2.1 File hashing

The hard drive may be the first thing that any forensic analyst will investigate. The traditional hard drive forensic tool reads the entire contents of a hard drive and tries to locate the data that would be useful to the investigator. These tools usually use the file system to go through the directory listing and identify files. This method may take a long time, especially when there are stacks of drives to sort through. Forensic analysts are interested in finding evidence from seized hard disk images. For example, a security officer in a company might examine the computer of a suspicious employee for unauthorized access to company classified documents. A law enforcement agency (LEA) is any government agency responsible for the enforcement of the laws. Officers in these agencies might search a home computer of a suspect for illegal child sexual abuse images. Network analysts might try to reconstruct the network streams to identify that any malware is downloaded. As in these examples and many other situations , the evidence is commonly a file that is a prohibited image, or video files, and the contents of these files are known. These files are called target files and forensic analysts try to identify them to prove their theory. A digital signature is a mathematical technique used to validate the authenticity and integrity of a message, software or digital document, which is often referred to as a file. Additionally, forensic analysts use cryptographic hash functions to create digital signatures of target files. These hash values are used to identify the file of known contents to which same hash functions are applied and resulted hash values are stored to be matched. If the hash value of the target file and the hash value of the file of interest match, this strongly indicates that the file in question resides on that drive. Since it is highly unlikely that a cryptographic hash function will generate the same hash value for two distinct files, known as strong collision resistance, forensic analysts can have confidence in their claim. Finding key data that can be used as evidence in criminal court cases is not an easy task for forensic analysts. They identify files by computing hash values using MD5 or SHA1 hash algorithm and then look for a database with known hash values. Most popular forensic packages have built-in support for file identification by using hash values. The National Software Reference Library (NSRL)

Reference Data Set (RDS) is one of the well known databases.

**Limitations of file hashing**

Files hashes have inherent limitations in identifying the known contents. By changing a single bit of a file, the computed hash value of a modified file can be totally different. Malicious hackers, malware producers, pornographers and other authors of files whose contents they do not want to be detected can change a comma to a period, add one space or append few random bytes to evade detection. If the file is damaged or some section of the file is deleted or some erroneous bytes are inserted by a network error, the file hash method is not useful.

## 2.2 Sector hashing

An alternative method to file hashing is sector hashing. This method looks at the hash values of chunks of files instead of the entire file as a means to identify target files from digital storage. Garfinkel et al. [4] used blocks, which are broken up pieces of file into 4096 bytes, to identify the target files. 4096-byte sectors are hashed, and the hash value is compared to block hash values of the target files. They listed the situations that need this approach:

1. File systems and files may not be recoverable due to damage, media failure, partial overwriting, or the use of an unknown file system.

2. There may be insufficient time to read the entire file system, or a need to process data in parallel.

3. File contents may be encrypted.

4. The tree structure of file systems makes it hard to parallelize many types of forensic operations.

Sector hash does not depend on the underlying file structure. To identify the target files in a large disk drives, cryptographic hashes on sectors of the disk can be used instead

of whole files. Most of the existing file systems align the starting point of the file with the beginning of a disk sector. Thus, the first sector of a large file is stored in one disk sector and the second sector is stored on anther disk sector, which is typically the adjacent disk sector, and so on. Hence, it does not have to seek back and forth to follow the file system pointers. It can read sectors sequentially off the hard disk by reducing seek time. Sector hashing is prone to produce high false positives compared to traditional file hashing because of the common data block that can be present in many files. Parallelization can be achieved by splitting the hard disk image into several chunks and processing each chunk simultaneously with different CPUs. As mentioned before, the entire file does not have to be present to be analyzed and to be matched on a hard disk drive. For example, when large video files are removed and a few sectors are reused by the operating system for other purposes, most of the video is still survived on the hard drive. A file recovery tool is applied and recovers as much as it can. If we use the file hashes, the recovered segments of the video will not appear in a database of file hashes. But if the sector hashing method is used for this recovered video, some of the hash values of the block might match the block hash values of the target video file. For in-depth analysis, looking for target content at the file-level is necessary and useful but it is not ideal for effective triage. It takes time for tools to parse the file system, and it is not effective if the program carves the files based on headers and footers. The program has to know the information of the target media such as the operation system versions etc. In general, file carving tools yield false positives also. Sector hashing can be combined with random sampling to speed up the decision process of identifying whether or not a target file is on a device. This random sampling uses the same methodology of sector hashing but instead of reading all the data, only small portions of data is randomly read and it gives results with high probability.

**Limitations of sector hashing**

For sector hashing method to be successful, the file has to be sector aligned on the hard disk. Block hashes are generated from the beginning of a file. If the file is not sector aligned, the sectors in a database cannot match the file blocks on the target media. As a solution,

we may pre-compute multiple hashes for each sector to prepare for the non-sector aligned blocks. However, this method is not practical because we need lots of storage to save the extra hashes. Similarity digest by Roussev [5] is suggested as another alternative which does not use sector hashing at all. This method takes lots of processing time and needs space. Most file systems are sector aligned. Examples include FAT, NTFS, Ext4, and ZFS. The future file system for Linux, B-tree File System, also uses sectors. However, if a file in a B-tree File System is smaller than 4 KiB, then it is packed in the leaf nodes of the B-tree file system structure and it is not sector aligned. In general, though we cannot identify small files that are less than sector size, most modern files these days are larger than 512 B and 4 KiB, which is becoming the standard sector size.

Sector hashing's real limitation is that it cannot do anything about encrypted file systems. Bitlocker for NTFS and ReFS and FileVault2 for HFS+ encrypt data when it is written to the storage device and decrypt when it is read back. Since a different key is used when data is encrypted, the resulted encrypted data will be different on different storage mediums every time.

## 2.3 Sector hashing on various file systems

To take advantage of sector hashing, which does not require knowing the underlying file system, the alignment of file data has to be on sector boundaries. Young and Foster [6] explained the various file systems from the point of sector hashing. They distinguished the terms: sector and block. Sector refers to a certain amount of data extracted from disk image. Block means a chunk of data extracted from file or file systems. In this thesis, this distinction is assumed.

#### FAT (File Allocation Table)

FAT was introduced by Microsoft with DOS. We can see this file system in Secure Digital (SD) cards, thumb drives, floppy disks and external disks. It is the standard file system for digital cameras also. This file system has improved as the capability of disk drives extended.

Now there are three major FAT variants: FAT12, FAT16 and FAT32. All three variants are block-aligned data. FAT is supported by windows NT and uses the file allocation table (FAT), which is a table that is stored at the top of the volume. Two copies of FAT are maintained to protect the volume. The boot files can be located because the root directory and the FAT tables are stored in a specific location. In 2006, exFAT (extended FAT) was developed by Microsoft to resolve the FAT's limitation of file size and performance. It can support file sizes bigger than 4 GB which are allowed by FAT32, though the security features of NTFS are lessened.

**NTFS (New Technology File System)**

This file system is the default one for the modern Windows system. The system drive is formatted with NTFS when you install Windows. It was shown in the consumer version of Windows XP. It has several features that FAT32 or exFAT do not have: file permissions for security, encryption, disk quota limits, shadow copies for backups, hard links, and others. NTFS was introduced with the following goals on mind:

- Reliability
- A platform for added functionality
- Support POSIX requirements
- Removal of the limitations of the FAT and HPFS file systems

The MFT (Master File Table) is used in NTFS and every file and directory has an entry in the MFT. For large files, block-align is done in this file system. For files smaller than 1024 bytes, block-align cannot be done since whole contents of a small file can be stored in the MFT. When a 4-KB pysical sector was introduced, the operating systems before Windows 8 and and Server 2012 could not support NTFS. An *emulation* mode was adopted, but it can make file system clusters to go beyond the sector boundaries.

### Ext4

Most Linux systems and later versions of Android adopt this file system. Ext4 uses extents and Ext3 employs a block pointer system. Both of them are block-aligned. The file is always aligned with the underlying media.

### ZFS (Z file system)

Traditionally file systems were built over a single physical device. A volume manager was designed to refer multiple devices, but this added some complexity and limited certain file system improvement. In ZFS, the volume manager was removed. They use a storage pool to place devices. File systems can be created on top of this storage pool and it works as an arbitrary data storage so that they do not depend on individual devices. Any file systems in the pool can share the storage space. File systems can grow automatically within the pool and any file system in the pool can use additional space immediately.

In ZFS, blocks are aligned with the storage media. Sector hashing can be used without any issue.

### BTRFS (B-tree file system)

BTRFS is an open source file system which was introduced in 2007. It is adopted in some versions of Linux as a default file system. An extent is a contiguous storage area saved for a file in a file system. Since BRTFS is based on copy-on-write, snapshots and clones are implemented efficiently. Rodeh et al. [7] listed its main features:

- CRCs maintained for all metadata and data
- Efficient writeable snapshots, clones as first class citizens
- Multi-device support
- Online resize and defragmentation
- Compression

- Efficient storage for small files
- SSD optimizations and TRIM support

BTRFS uses extents of blocks to create large files. For small files, they can be packed into the leaf block of the BTRFS. Thus for files smaller than 4KB, it might be sector-aligned.

### AFF4 (Advanced Forensic Framework 4)

Cohen et al. [8] developed AFF4 to address limitation of the traditional forensic file formats. A key feature of AFF4 is address space virtualization. A virtual bitstream is specified by a Map stream. Maps can refer to arbitrary sources of bytes. Map uses the globally unique referencing scheme which refers to objects in the AFF4 system.

Cohen et al. [9] introduced hash-based disk imaging using AFF4. AFF4 can divide the disk into several byte ranges and save each run separately in an AFF4 volume. The image map is created and is used to reconstitute the disk image at a later stage. An AFF4 volume contains several number of byte streams with a URL of the form aff4://encoded_hash. There can be one or more AFF4 volumes and a single AFF4 map object can represent the hard disk image.

Current implementations of this dissertation do not have an interface to incorporate the AFF4, which will be a future research goal.

### Encrypted File systems

Some applications encrypt a file and move it to a different system. In this case, the encrypted section of the data remains the same on a transferred system. As long as the encrypted data is not encrypted again, the sector hashing method can be used to verify the evidence of the existence of the encrypted file. However, the encrypted file system behaves differently. In most encrypted file systems, each drive is encrypted with a different key. This means that identical data can be hashed into different hash values. This problem may be avoided by reading the data after the decryption key is loaded.

## 2.4 Similarity measures

A similarity measure is a quantified characterization of how alike two data objects are. This measure is sometimes represented by a distance. If this distance is small, there is a high degree of similarity. Conversely, a large distance means there is a low degree of similarity.

Applying similarity measures is subjective and highly dependent on the domain and application. For example, two fruits are similar or not because of color, size, or taste. Care should be taken when calculating distance across dimensions/features that are unrelated. The relative values of each element must be normalized, or one feature could end up dominating the distance calculation. Similarity is usually measured in the range 0 to 1 [0,1].

Two main considerations about similarity:

$$Similarity = 1 \; if X = Y$$

$$Similarity = 0 \; if X \neq Y$$

$$Where \; X \; and \; Y \; are \; two \; objects.$$

### 2.4.1 Euclidean distance

*Euclidean distance* is the most common distance-based similarity measure. In most cases when people talk about distance, they are referring to Euclidean distance. When data is dense or continuous, this is the best proximity measure.

The Euclidean distance between two points is the length of the path connecting them. The Pythagorean theorem gives this distance between two points.

In general, for an n-dimensional space, the distance between $\mathbf{p}$ and $\mathbf{q}$ is :

$$Euclidean \; distance \; (\mathbf{p}, \mathbf{q}) = \sqrt{\sum_{i=1}^{N} (x_i - y_i)^2} \tag{2.1}$$

where $\mathbf{p} = (x_1, x_2, \ldots, x_n)$ and $\mathbf{q} = (y_1, y_2, \ldots, y_n)$

### 2.4.2 Manhattan distance

*Manhattan distance* is a metric in which the distance between two points is the sum of the absolute differences of their Cartesian coordinates. A simple way of saying this is the total sum of the difference between the x-coordinates and y-coordinates. Manhattan Distance is used when we measure the distance between two data points in a grid-like path. To find the Manhattan distance between A and B, we have to sum up the absolute x-axis and y–axis variation. It means we have to find how these two points, A and B, are varying in X-axis and Y-axis. A more mathematical way of saying Manhattan distance is that it is the distance between two points measured along the axes at right angles.

In a plane with **p** at (x1, y1) and **q** at (x2, y2):

$$Manhattan\ distance\ (\mathbf{p}, \mathbf{q}) = |x1 - x2| + |y1 - y2| \tag{2.2}$$

This Manhattan distance metric is also known as Manhattan length, rectilinear distance, L1 distance or L1 norm, Minkowski's L1 distance, taxi-cab metric, or city block distance. Manhattan distance can be generalized like Euclidean distance.

### 2.4.3 Minkowski distance

The *Minkowski distance* is a generalized metric form of Euclidean distance and Manhattan distance.

The *Minkowski distance* of order $\lambda$ (where $\lambda$ is an integer) between two points

$$p = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n \ and \ q = (y_1, y_2, \ldots, y_n) \in \mathbb{R}^n$$

is defined as:

$$D\left(X, Y\right) = \left(\sum_{i=1}^{n} |x_i - y_i|^{\lambda}\right)^{\frac{1}{\lambda}} \tag{2.3}$$

*Minkowski distance* is typically used with $\lambda$ being 1 or 2, which corresponds to the *Manhattan distance* (2.2) and the *Euclidean distance* (2.1), respectively. In the limiting case of $\lambda$ reaching infinity, we obtain the *Chebyshev distance*:

$$\lim_{\lambda \to \infty} \left( \sum_{i=1}^{n} |x_i - y_i|^{\lambda} \right)^{\frac{1}{\lambda}} = \max_{i=1}^{n} |x_i - y_i|. \tag{2.4}$$

**Synonyms of Minkowski**: Different names for the *Minkowski distance* or *Minkowski metric* arise from the order:

$\lambda = 1$ is the *Manhattan distance*. Synonyms are L1-Norm, Taxicab or City-Block distance. For two vectors of ranked ordinal variables, the Manhattan distance is sometimes called Foot-ruler distance.

$\lambda = 2$ is the *Euclidean distance*. Synonyms are L2-Norm or Ruler distance. For two vectors of ranked ordinal variables, the *Euclidean distance* is sometimes called Spear-man distance.

$\lambda = \infty$ is the *Chebyshev distance*. Synonyms are Lmax-Norm or Chessboard distance.

### 2.4.4 Edit distance

Edit distance is a way of measuring the similarity between two strings. There are several different definitions of edit distance. Levenshtein distance is used interchangeably with Edit distance since it is the most common way of measuring edit distance [10]. It measures the minimal number of operation to match two string by using three operations:

- insertions
- deletions
- substitutions

insertions, deletions or substitutions

The search problem is called "string matching with k differences" in the literature.

Hamming distance is used to compare two binary strings. It allows substitution only and applies to strings of the same length. It is often used in the data communication field for error detection or error correction.

I looked at edit distance to devise an algorithm for my research: disk similarity. The contents of each disk might be considered a long string. Edit distance can be applied to two long strings. This approach fails in several aspects. When two disks to be compared do not have the same size, edit distance is not meaningful. If two disks have the same contents, but the location of the contents, (for example, sectors), are different, edit distance will return very low similarity values. However, my algorithm will return a higher similarity value since it does not depend on sector location.

### 2.4.5 Cosine similarity

The *cosine similarity* metric finds the normalized dot product of two attributes. By determining the *cosine similarity*, we effectively try to find the cosine of the angle between the two objects. The cosine of 0° is 1, and it is less than 1 for any other angle.

It is thus a judgement of orientation and not magnitude: two vectors with the same orientation have a *cosine similarity* of 1, two vectors at 90° have a similarity of 0, and two vectors diametrically opposed have a similarity of -1, independent of their magnitude.

*Cosine similarity* is particularly used in positive space, where the outcome is neatly bounded in [0,1]. One of the reasons for the popularity of *cosine similarity* is that it is very efficient to evaluate, especially for sparse vectors.

### 2.4.6 Jaccard index (JI)

Jaccard index (**JI**), also known as the **Jaccard similarity coefficient**, is a simple and widely used similarity measure [11] applied to arbitrary sets of data. It measures similarity between finite sample sets and is computed as follows:

$$JI(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

$$0 \leq JI(A, B) \leq 1$$

**JI** is defined as the magnitude of the intersection divided by the magnitude of the union of two sets, A and B. In my work, the sets A and B consist of sector hashes from digital media of potential interest.

Jaccard index can also be a measurement of asymmetric information on binary variables.

$$JI_ij = \frac{p}{p+q+r}$$

where $p$ is the number of variables that are positive for both objects.

$q$ is the number of variables that are positive for the $i^{th}$ object.

$r$ is the number of variables that are positive for the $j^{th}$ object.

**Weighted Jaccard index**

If x =($x_1$, $x_2$, ..., $x_n$) and y =($y_1$, $y_2$, ..., $y_n$) are two vectors with all real $x_i$, $y_i \geq 0$, then their weighted Jaccard index ($JI_w$) is defined as

$$JI_w = \frac{\sum_{i=1}^{n} Min(x_i, y_i)}{\sum_{i=1}^{n} Max(x_i, y_i)}$$

It is also known as Ruzicka[1]similarity. In my research, I adopted the weighted Jaccard index for the calculation of **JINF**.

## 2.5 Entropy

Entropy or information entropy is the expected value or quantity in information theory. It can be used to describe the average level of information for any random variable. Claude Shannon introduced this concept of information entropy in his paper in 1948 [13].

Given a random variable X, the entropy H(X) is as follows:

[1]This Ruzicka work [12] appears to be relevant because several research papers cite this, but I was unable to locate a copy of the original paper to confirm.

$$H(X) = -\sum_{i}^{N} P_X(x_i) \log_b P_X(x_i)$$

where N is number of different outcomes and $P_X(x_i)$ is the probability that outcome $x_i$ can happen.

Data communication has three elements: data source, channel for communication and a receiver. In his theory, the main issue of the communication is that the receiver has to identify data that is based on the signal from the source through the channel. The entropy can be regarded as an absolute mathematical limit of the lossless compression of data from the data source on a noiseless channel. Entropy can be used as a measure of the relative randomness of a data or the information density.

Entropy is loosely defined as the relative randomness of a given data unit. It can be used as a measure of compression state of a data unit. ASCII text files and Bitmap Image files can be compressed more than encrypted data. Thus, ASCII text has a lower entropy value than encrypted data.

Hall and Davis [14] presented a method where the entropy and compressibility of file fragments are used for identifying file type by using the sliding window technique.

## 2.6 Bloom filter

A Bloom filter is a compact data structure that can answer a membership query with memory-efficiency and speed. It is a probabilistic representation of a set to support the query: *is this element in a set?*. It can tell if the element definitely is *not* in the set, and it can tell if the element is *possibly* in the set.

A Bloom filter is implemented with a Bit Vector of size m, which is an array of bits. Initially, all bits are set to 0. For the membership query, k different hash functions are used. Each of the k different hash functions map or hash the element to one of the array bits. These hash functions are expected to assign the elements to the bit vector with the

uniform random distribution. The number of hash functions are related to the error ratio and is typically a small constant. The size of the bit vector depends on the number of hash functions, k, and the size of the set. Figure 2.1 shows a Bloom filter with 3 hash functions and bit array of size 9.

When an element is added, each of the k hash functions is evaluated and finds the position in the bit array. The computed position in the bit array is set to 1. It is also possible that two or more hash functions point to the same position in the bit array. To insert a new element, X, in a Bloom filter, we calculate h1(X), h2(X) and h3(X), which will return 2nd, 4th and 5th position respectively. These bits are set to 1 as shown in Figure 2.1. For the membership test of an element, the process is the same as the addition process except it does not set the bit to 1. Instead, it checks the calculated position to see if the bit in that position is set to 1 or not. If any of the bits that the hash functions calculated equal 0, then the queried element is definitely not in the set. If all of the bits checked are 1, then the element is probably in the set. It can have a false positive result.

It is not possible to remove an element from a Bloom filter because we can not tell if a bit is set by more than one element. For example, in Figure 2.1, to remove X from a Bloom filter, we cannot reset the second bit from 1 to 0 since it was set by X and Z as well.

Though a Bloom filter has a problem of false positive results, it has a space advantage compared to other data structures, such as binary search trees, hash tables, and linked lists. Most of them need space to store the data items, while a Bloom filter does not store the data.

A Bloom filter is used in various settings such as Web cache sharing [15], a compact representation of a differential file [16, 17], and Free text searching [18] where the set of words in a text is represented by a Bloom filter.

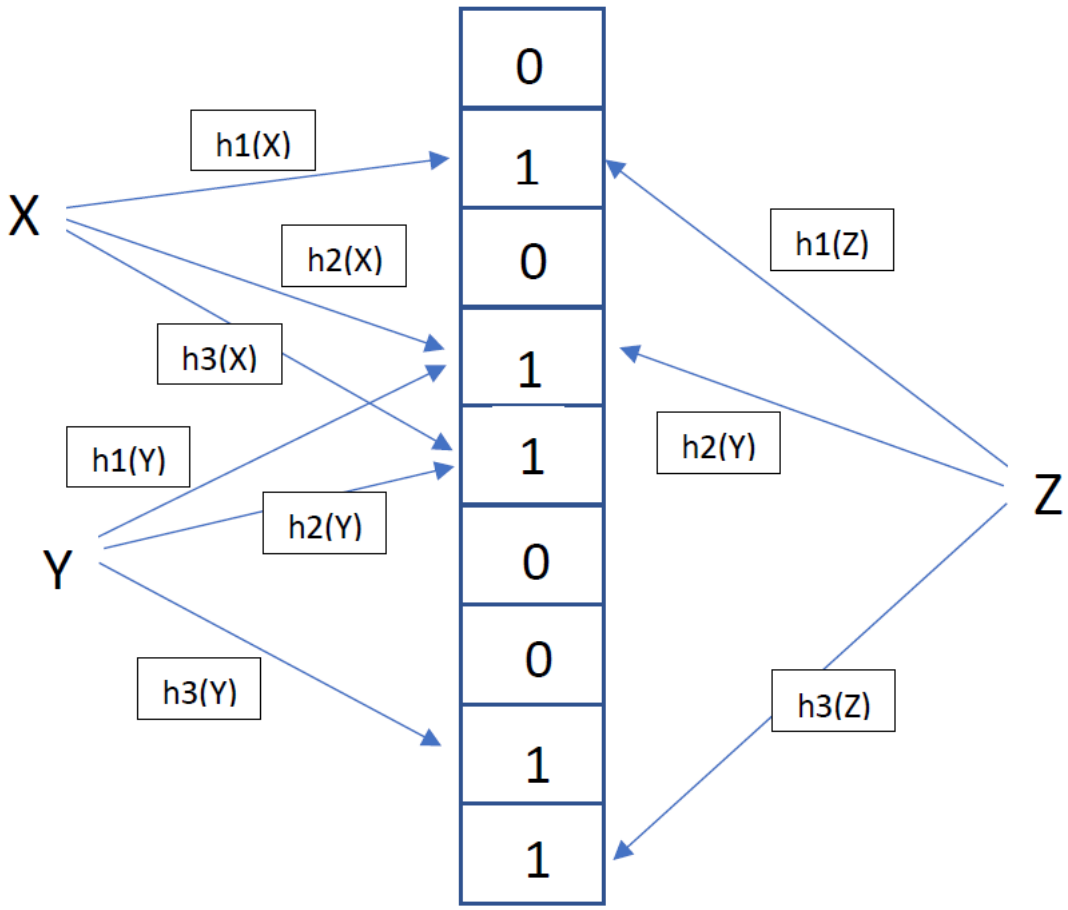


Figure 2.1: Bloom filter. k = 3, m = 9.

## 2.7 Hashing methods

A hash value $h$ is generated by a function H of the form

$$h = H(M)$$

where M is a variable-length message and $H(M)$ is the fixed-length hash value. The purpose of a hash function [19] is to produce a "fingerprint" of a file, message, or other block of data. To be useful for message authentication, a hash function H must have the following properties:

1. $H(x)$ can be applied to a block of data of any size.
2. $H(x)$ produces a fixed-length output.
3. $H(x)$ is relatively easy to compute for any given x, making both hardware and software implementations practical.
4. For any given value $h$, it is computationally infeasible to find $x$ such that $H(x) = h$. This is sometimes referred to in the literature as the one-way property.
5. For any given block $x$, it is computationally infeasible to find $y \neq x$ such that $H(y) = H(x)$. This is sometimes referred to as weak collision resistance.
6. It is computationally infeasible to find any pair $(x, y)$ such that $H(x) = H(y)$. This is sometimes referred to as strong collision resistance.

There are several hashing methods such as MD5, SHA-1, CRC32 and CRC64.

Dandass et al. [20] shows their research results from the analysis of various hashing methods such as MD5, SHA1, CRC32 and CRC64 by extracting over 500 million sectors from a large number of files.

In my research, I use MD5 among many hashing methods for efficiency. Though MD5 does not have the property of collision resistance, I do not have to worry about this known weaknesses for the implementation of my algorithm.

I introduce MD5 and SHA-1 in this section.

### 2.7.1 MD5

The MD5 hashing algorithm was introduced by Rivest in 1992 for digital signature applications [21]. MD5 works with 512-bit blocks and each block is broken down into 16 words which is 32 bits each. The output of MD5 is a 128 bit (16-byte) hash. To make sure that the input can be divided by 512, padding the input is applied. It was originally intended to be used in authentication of digital signatures. However, it is no longer classified as a reliable method for cryptographic checksum. Though there are proven attacks to MD5, it is still an effective hashing method for my purposes since it is fast and attacks are not relevant for my use cases.

### 2.7.2 SHA

SHA-1 was introduced in 1995 and was updated as FIPS PUB 180-2 [22] in 2002. The output of SHA-1 is a 160-bit (20-byte) hash. Like the MD5 algorithm, SHA-1 also adopts the padding method to input data in order to make sure that input data is divisible by 512. Some weaknesses of this algorithm have also been reported, but SHA-1 remains suitable for our purposes for the same reasons that MD5 is. SHA-2 was subsequently developed and released in 2001 to address limitations and attacks against SHA-1, and my methods could easily be adapted to use SHA-2 or any future hashing algorithm.

# Chapter 3: Previous Work

This chapter focuses on closely related similarity measure work. Some of them inspired my research method and others provided valuable guidance for future research.

## 3.1 Forensic feature extraction and cross-drive analysis

Garfinkel [23] identified an issue with typical forensic analysis. The problem was that the hard disk image did not correlate to other images. They listed three problems: (1) improper prioritization, (2) lost opportunities for data correlation, and (3) improper emphasis on document recovery. They tried to address these problems by introducing Forensic Feature Extraction (FFE) and Cross-Drive Analysis (CDA). Various lexigraphic techniques are applied for extracting information from hundreds of hard disk images.

### 3.1.1 Pseudo-unique identifier

Many forensic tools and algorithms use string search as their basis. These strings can be user-specified regular expressions that match features such as email addresses, telephone numbers, social security numbers, credit card numbers, network IP addresses, and other kinds of information that might be pseudo-unique identifiers [4, 24–26]. "A pseudo-unique identifier is an identifier that has sufficient entropy such that within a given corpus it is highly unlikely that the identifier will be repeated by chance [23]." Email Message-ID is a typical example of pseudo-unique identifier. Having the same Message-ID on two different machines strongly implies that the identifier is transferred from one machine to another machine. They have found that the following properties exist in a good pseudo-unique identifier.

1. They are long enough so that collisions are unlikely to occur by chance.

2. They can be recognized using a regular expression and do not require parsing or semantic analysis.

3. They do not change over time.

4. They can be correlated with specific documents, people or organizations.

Since all identifiers in a particular class of identifiers do not have to be pseudo-unique, any forensic tool which utilizes pseudo-unique identifiers needs to know how to distinguish between ubiquitous identifiers and truly pseudo-unique identifiers.

### 3.1.2 Feature extractors

They have built a program called *feature extractors* which can scan a hard disk image, extract pseudo-unique features and store the output in a file. This program can extract the following features:

1. An email address.

2. An email Message-ID.

3. An email Subject.

4. A Date from which data and time stamps can be extracted in a variety of forms.

5. A cookie which distinguishes from Set-Cookie:header in web page cache files.

6. A US social security number.

7. A Credit card number.

*Feature extractors* run over the extracted string files and write their results to feature files. These extracted features are then applied to a multi-drive corpus to identify associations between different drives. They used those features from *feature extractors* for disk image attribution and also built a tool which scans disk images to report the potential information that should have been destroyed under the Fair and Accurate Credit Transactions Act. For

rapid ownership determination, they created a histogram of the email addresses on the disk images which are obtained from *feature extractors*. The most frequent email address on that drive is likely the address of the owner of that machine.

### 3.1.3 Cross-drive analysis (CDA)

Cross-drive analysis is the term that they have coined to describe their method used in forensic analysis for a data set that spans multiple hard drives. Their theory behind CDA is that data acquired from multiple hard disks can improve the analysis of a drive in both situations: multiple drives are related to a given drive and they are not related. They created a stop list for features that can be ignored in forensic analysis because they are ubiquitous. They can prioritize the drives by the number of extracted features and this process is named *hot drive identification*. For the second order CDA, the CDA developers raise a different question: Which drives in the corpus have the largest number of features in common? To answer this question, they implemented a program called Multi Drive Correlator (MDC). The MDC is a function whose input is a set of drive images with a feature to be correlated, and whose output is a list of (feature, drive-list) tuples. This program reads multiple feature files and generates a report. This report shows the number of drives on which each feature was identified, the total number of times that feature was identified on all drives, and a list of drives on which that feature occurs. Once the correlation list is generated, a report is generated to tell which drives are most highly correlated. They used three weighting functions for the purpose of scoring the correlation between each pair of drives.

Let:

D = number of drives

F = number of extracted features

$d_0 ... d_D$ = drives in corpus

$f_0 ... F_F$ = Extracted feature

$$FP(f_n, d_n) = \begin{cases} 0 \ f_n \texttt{ not present on } d_n \\ 1 \ f_n \texttt{ present on } d_n \end{cases}$$

$S_1$ is to add up the numbers of features that two drivers have in common:

$$S_1(d_1, d_2) = \sum_{i=0}^{F} FP(f_n, d_1) FP(f_n, d_2)$$

$S_2$ is to discount features by the number of drives on which they appear, which makes correlations resulting from pseudo-unique features more important than correlations based on ubiquitous features:

$$DC(f) = \sum_{i=0}^{D} FP(f, d_n) = \#\texttt{ of drives with feature } f$$

$$S_2(d_1, d_2) = \sum_{i=0}^{F} \frac{FP(f_n, d_1) FP(f_n, d_2)}{DC(f_n)}$$

$S_3$ accounts for the case that rare features that are present in high concentrates on drive $d_1$ and/or $d_2$ need to increase the weight:

$$FC(f, d) = \texttt{count of feature } f \texttt{ on drive } d$$

$$S_3(d_1, d_2) = \sum_{i=0}^{F} \frac{FC(f_n, d_1) FC(f_n, d_2)}{DC(f_n)}$$

They applied an MDC using the social security number and also credit card number extracted. By using this method they were able to find their own clerical mistake with the help of metadata that they recorded. They pointed out that a more accurate scoring method

is needed between drive pairs and to cluster drives. They also noted that better *feature extractors* are also necessary.

## 3.2 Bulk extractor

Beverly et al. [27] extended this work using Ethernet MAC (Media Access Control) addresses extracted from validated IP packets. A MAC address is a hardware identification number that identifies each device uniquely on a network. Ethernet Media Access Control(MAC) addresses can be important in forensic analysis. Every network card such as an Ethernet card or Wi-Fi card is assigned with this unique number and cannot be changed. Millions of networkable devices exist and each device requires a unique MAC address. Since there must be a vast range of possible addresses, MAC addresses consist of six two-digit hexadecimal numbers which are separated by colons. For example, an Ethernet card may have a MAC address of 00:0e:83:b2:c0:3e. The first three octets of MAC are assigned to specific manufacturers and can be used to derive some information about the device which uses this specific MAC address. War driving, also called access point mapping, is the act of exploiting and locating connections to wireless local area networks while driving around buildings in a city or elsewhere. For war driving, you need a computer, a vehicle, a wireless Ethernet card which is set to work in promiscuous mode, and an antenna which can be mounted on the vehicle, maybe on top of the vehicle or inside the vehicle. A wireless LAN may have a range which goes beyond a building and a user who is outside of that building may be able to intrude into the network to get a free Internet connection. The MAC addresses of wireless routers can be correlated with the database of war driving. The fact that a device has been associated with a physical network can be used to derive information about membership in an organization or a physical location. They expanded the scope of network forensics to encompass the findings of long terminated network data on fixed media and in memory. They created the ground truth dataset for verifying their network carver. For collecting ground truth data, all data on a test machine was securely erased and a virgin copy of the operating system was installed. They transferred the four files of various sizes

from three different systems by using HTTP or SCP. All the packets were captured when they enter or leave the system by using a promiscuous recorder. This ground truth dataset was used to develop their carving strategy. For their network carver, this dataset and a method to generalize discriminatory patterns were needed.

To achieve better heuristics, they wrote software that scans for the byte patterns of the known IP addresses and calculates the frequency of n-gram byte patterns which exist before and after the addresses at different offsets within a specified window. Hibernation files were processed with this new carver, which uses these signatures and all the IP address that were found were tabulated. On the basis of the signature analysis, a new software package was created for *bulk_extractor*, which is an open source tool for forensics. The following data structures were carved with this tool:

- IP Packets
- Socket Structures
- Windows
- Ethernet

They also implemented various technique of validating IP addresses to reduce the false positive rate.

1. Checksum: A checksum value over the IP header in an IP packet is provided as a means of self-validation. They examined that a correct checksum exists for all IP packets that were discovered.

2. Filtering: Some IP addresses are reserved, invalid, or not likely to be exist. They found bogus IP address by using this filtering method.

3. Frequency analysis: A histogram were created with the IP address that were discovered during the process of carving data structures. More frequent IP addresses are likely to be the outcome of actual system activity.

4. Correlation between modalities: Correlation using various technologies can give confidence in inferring valuable information. For example, IP addresses from two data structures hold a greater likelihood of coming from prior traffic than IP addresses from a single data structure.

They did a CDA with MAC addresses using the method introduced in [23]. They treated both the MAC addresses and disk images as nodes and an address on a hard disk image as a link in a graph. From the partitioned graph, they were able to create distinct clusters from the collection of disk images. For future work, they want to extend their work to other platforms such as smartphones which are becoming more complex and sophisticated. One of the strong points of my methods is that it does not depend on platforms.

## 3.3 Distinct sector hashes for target file detection

Young, Foster, Garfinkel, and Fairbanks [6] introduced a file agnostic approach leveraging the speed of hashing. They use sector hashes instead of file hashes. They compared blocks, which are fixed sized file fragments, to a large data set of sector hashes, and they considered individual sectors and collections of contiguous sectors (blocks or clusters). They identified the file with sector hashing by checking if there is a distinct file block or blocks. They defined a *distinct* block as "one that does not occur anywhere more than once except as a block in a copy of the original file". Their method is based on two hypotheses:

- If a block of data from a file is distinct, then a copy of that block found on a data storage device is evidence that the file is or was present.
- If the blocks of that file are shown to be distinct with respect to a large and representative corpus, then those blocks can be treated as if they are universally distinct.

They suggested that it will be more accurate and faster for the analysis of any digital media if they use a database of hash values computed from fixed sized blocks of data. Large corpora - Govdocs [28] and the NSRL RDS [29] - were used to populate a database of hashes. Three types of sectors, *singleton*, *paired*, and *common* sectors, were analyzed to

understand the root causes of non-distinct blocks. When they counted the number of blocks in large corpora, if a block occurred once or twice, they were called *singleton* and *paired*, respectively. If the count was greater than two, it was called *common* blocks. Their logic behind counting the frequency of the sector is that if *paired* and *common* are rare, then *singleton* blocks are universally distinct. Also by looking at the context of those blocks, they tried to analyze the root causes of occurring more than once. They used a 512 byte block and a 4 KB block to analyze since modern hard drives use 4 KB sectors instead of 512 byte sectors. The most *common* block was all NUL(0x00) bytes. Also, a data structure which is used internally in Adobe PDF occurred more than 200,000 times in the GovDocs corpus. Microsoft Office had several *common* blocks in its internal data structure. The different versions of the same malware had *common* blocks at the same offset of the files.

The major finding for *common* sectors was that the same block existed in many files due to malware code reuse and common file container formats. For the goal of field deployment on a laptop, they suggested sampling of sectors instead of processing all media sectors. Several database implementations were considered and a Bloom filter front end was implemented to speed up generic query time [30].

They analyzed several file systems to demonstrate the generality of the approach. The authors noted that file encryption and encrypted file systems present a problem since the same data of interest will be stored differently when encrypted.

## 3.4 Impact of common blocks on similarity

Moia et al. [31] assessed the impact of common blocks on similarity. They showed how common data can be identified and how those are spread over various file types and their frequencies. They also found out that this common data is often generated by the application and not by the user. Their experiment shows that by removing common data, they were able to reduce about 87% of the number of matches.

## 3.5 Contraband file detection

Penrose et al. [32] used a Bloom filter for initial fast contraband file detection. A Bloom filter reduces the size of a database (hashes in this case) by an order of magnitude, but incurs a small false positive rate in return. The authors implemented a larger Bloom filter to provide faster access, achieving 99% accuracy while scanning for contraband files in minutes using a test data set.

## 3.6 Djb2

Roussev et al. [33] introduced an approach which calculates similarity scores by using the *djb2* hash function. Two kinds of hash functions are commonly used: polynomial hash, and cryptographic hash. Polynomial hash is geared for speed and the main function of cryptographic hash is for collision-resistance. *Djb2* is a simple polynomial hash and it was introduced by Dan Bernstein [34]. These scores obtained by *djb2* hash are used to find similar files.

The same authors show an approach for prallelizing forensics tools on GPUs [35].

## 3.7 Proactive object fingerprinting and storage (PROOFS)

Shields et al. [36] presented Proactive Object Fingerprinting and Storage (PROOFS) which is a forensic acquisition and monitoring system named over a network. It is a continuous forensic evidence collection system for file system forensics. They create and store signatures for files that are modified or changed within a network. After creating a dictionary of statistically important terms, text fingerprints are generated which will be the part of a signature. IDF (inverse document frequency) is used as a selection criteria for terms. IDF is the log of the sum of all the documents divided by the number of documents which contain the term. The fingerprints of the source files are compared to all other fingerprints using cosine similarity matching. PROOF can identify similar files even if the format of the files

are changed.

## 3.8 Sector signature analysis

File systems usually store files in clusters that are made up of contiguous sectors on disks. An example can be seen in Windows XP's file system, NTFS. In this system's default configuration, it distributes file space in continuous increments of eight contiguous sectors. A sector is 512 bytes, which means one cluster is 4096 bytes. Consecutive clusters in a file will not always be stored contiguously. When the file system has to distribute a cluster to a file, the free space is used up. In other words, the clusters that other files are not using. Multiple cycles of adding and removing files will cause the free space of the drive to be fragmented. This fragmentation will, in turn, cause new files to be scattered over non-contiguous clusters.

Whenever a file is deleted from the disk, the file's contents are not actually zeroized. Instead, the file system will usually set the file as available for reuse. This means that, barring a situation in which the user overwrites the file with another file, a deleted file can be recovered. Even in a situation where the deleted file's cluster is occupied by another file that has overwritten the cluster partially, the fragment data leftover in the cluster can be analyzed. This analysis is known as the sector signature analysis technique. This is done by looking at the file system's data structures. If the data structures responsible for mapping the file contents onto disk clusters have not been overwritten, a deleted file can be recovered with relative ease. If the metadata is overwritten, however, recovery is much less likely. The sector signature matching technique can still be used on sectors that have not been overwritten with new files.

Small files approximately under 900 bytes are stored by NTFS into the master file table (MFT). Metadata and file directory information is stored in MFT. Thus, the files would not align to disk sector boundaries. Files of interest are usually larger than what the NTFS stores and as such are stored in clusters not in the MFT. This means that these larger files will definitely align on 512-byte sector boundaries.

## 3.9 Hash-based carving

Garfinkel et al. [37] presented a method based on hash values of blocks instead of whole-file hashing, and the basic idea of my similarity measure method comes from this block hashing method.

They implemented rules to identify non-probative blocks [37]:

- *The ramp test:* There is a common block pattern where 32-bit numbers increase monotonically. They implemented a simple test which returns True if half of the bytes in a buffer matches the increasing pattern (ramp).
- *The white space test:* If a block contains the three quarters or more whitespace, they treat it as non-probative.
- *The 4-byte histogram test:* If any single 4-gram exists more than 256 times in a sector, it is removed. This pattern appears in Apple QuickTime and Microsoft file formats.

On top of these three rules, they also did the entropy test. After experimenting their rules and entropy test, they conclude that a rule-based method works better than the entropy method.

They presented the HASH-SETS algorithm that can identify the existence of files, and the HASH-RUN algorithm which reassembles files using a database of file block hashes. A fixed block size (e.g., 4 KiB) may become a problem due to file system alignment. They overcame this issue by hashing overlapping blocks with a 4 KiB sliding window over the entire drive, moving the window one sector at a time.

They presented real-world experience to use hash-based carving. *Hash-based carving* is an alternative approach that relies on comparing hashes of physical sectors of the media to a database of hashes created by hashing every block of the target files [37, 38]. Hash-based carving is the process of identifying a target file on a media to be searched by hashing blocks from both the target file and the media and checking for hash matches. Those blocks have to be the same size. The primary obstacle to implement hash-based carving is a high demand

of the block hash database. Conventional databases cannot meet performance requirements, which require lots of hash lookups per second. They developed *hashdb* [39] for this end and used this specially designed database to implements HASH-SETS which can determine the existence of the target files on media being searched. They found out that a major hurdle to hash-based carving was to resolve chance matches that happen between individual blocks on the media being searched and individual blocks on the database. These happened because identical blocks exist in many different files not because the target files once existed on the media being searched. They used the MD5 hash algorithm because it has fast calculation capability and the fact that MD5 is not collision resistant [40] was not an issue since they used MD5 to search for the known content. In my research, I used MD5 for the same efficiency reason and even if there are couple collisions, which is extremely unfeasible, the impact on the result of the methods presented in this dissertation is negligible. They used the term *probative* instead of *distinct* to describe data which has a high probability that a whole content of the file was once present. The reason they change the terminology is that many blocks which seem to be distinct in a small data sets are not distinct when a larger file corpus is investigated. They listed several scenarios that hash matches can happen:

- A copy of the target file is present on the media being searched.
- A copy of the target file was deleted and overwritten partially.
- On search media, a file which shares several sectors with the target file exists.
- A target file is embedded in a larger encompassing file.

A good hash-based carving method is expected to address all these scenarios simultaneously. Their hash-based carving has four steps:

- Database building.
- Media scanning.
- Candidate selection

- Target assembly.

They identified the *common blocks*, which are present in many different target files. The most common block consists of all NULLs, which is also deleted in my method in the process of whitelist removal. The second most common block is a block consisting of 32-bit numbers which increase monotonically. I did not incorporate this type of common block in whitelist removal. Common block analysis is an area to be further researched for whiltelist removal process in my research. For their experiment setup, they used M57-Patents datasets [41].

## 3.10 M57-Patents datasets

One of the problems in forensic research is that we do not have proper, real data suitable for research purposes because information can be confidential, sensitive, or proprietary. As a result, many researchers and teachers in digital forensics fields spend a good amount of their time preparing forensic materials such as dumps of memory, disk images, and network packet dumps. However, achieved data is not realistic enough and also often unnecessarily complex. Creating test data which resolves these problems is very difficult. Hence Woods et al. [41] created realistic forensic datasets, known as *M57-Patents*, to support cyber security and digital forensics researcher. It is a multi-modal corpus which includes memory and hard drive images, network packet captures and images from USB drives and cellphones.

A scripted scenario is prepared for creating this corpus. It is simpler than real world data but complex enough to be used by the forensics researchers and students. With this corpora, forensic tools can be verified and data preparation time is reduced. This corpus can be redistributed without worrying about non-public information and licenses. I used this corpus for my research as a known truth to validate the algorithm. The scenario uses a fictitious company called M57 whose business is in patent prior art searches. All the actions of the employees are recorded on a private network.

This scenario records activities from a 17-day period in November and December, 2009. The fictitious company, M57, has four employees: a CEO (Pat), two patent researcher

(Charlie and Jo) and one IT administrator (Terry). They imaged the daily activities of each employee except for holidays and weekends. Three workstations which belong to Pat, Charlie and Jo have installed Windows XP and on Terry's machine, Windows Vista Business 32-bit was installed.

This corpus can be used for disk and network forensic exercises by teachers, students, and researchers since the memory, hard drives, thumb drives, and network traffic for each employee were captured, and hard disks were imaged daily. To address the Microsoft licensing issue and provide as complete a disk image as possible, libraries and executables proprietary to Microsoft were disabled before the corpus was public distributed.

## 3.11 Random sampling and sector hashing

Taguchi [42] experimented with different sample sizes by using random sampling and sector hashing for drive triage. Given a drive, the goal was to give analysts information about the worthiness of continuing further investigation. If a block hash value of target data is found in a database, then it is very probable that the target file is contained in the drive. If no hashes are found with sampling, a confidence level is computed indicating the likelihood that the target data is not on the media. Taguchi tried to find the target data, which is defined as "any known files that the user is interested in findings". By taking equally sized chunks of a file, target data are organized into target blocks. If the target data is not present, the target blocks are either missed while sampling or non-existent. Instead of proving that target data do not exist, the probability that they are missed while sampling is calculated. *Confidence* is defined as one minus this probability. Sample sizes were calculated with two parameters: the size of target data and a *confidence* level. They found out that only 1.4% of a 1 TB drive is needed for 99% confidence and 10 MiB target data.

## 3.12 Fuzzy hashing

Fuzzy hashing (FH) is also called *context triggered piecewise hashing* (CTPH). Traditional hashes can determine if two object are the same or not. Instead, CTPH often gives an answer with a score percentage between 0 and 100. The higher the score is , the more similar two objects are. It can be conceived as a combination of cryptographic hashes, piecewise hashes, and rolling hashes.

Breitinger et al. [3] provided a definition and terminology for approximate matching (a.k.a. similarity hashing or fuzzy hashing). They divided approximate matching methods into three main categories: Bytewise, Syntactic and Semantic matching. They described the Bytewise matching method by saying that it " ... relies only on the sequences of bytes that make up a digital object, without reference to any structures within the data stream, or to any meaning the byte stream may have when appropriately interpreted." My method in this dissertation can be regarded as one of the Bytewise matching methods since it does not rely on any internal structure of the hard drive and does not give any meaning of the byte stream.

Bjelland et al. [43] present common scenarios where approximate matching can be applied: *Search*, *Streaming* and *Clustering*. In search mode, the data space is large compared to streaming mode. In clustering mode, input and data space is the same. For large sets, approximate matching is impractical due to its high latency.

Moia et al. [44] present steps to develop new approximate matching functions. Approximate matching functions are used because of limitations on cryptographic hash functions, which cannot detect non-identical but similar data. They present a list of requirements for forensic investigation, evaluation approaches, and design problems that these functions face in the development of a new approximate matching function.

### 3.12.1 Spamsum

Tridgell [45] presented *context triggered piecewise hashing* to find updates of files, implemented as the *spamsum* program. It identifies email messages which are similar to already

known spam mails.

### 3.12.2 Ssdeep

Kornblum [46] developed *ssdeep* in 2006. It is a program for calculating and matching *context triggered piecewise hashing* values. It is based on the *spamsum* program previously developed by Trigdell [45]. It identifies the contexts in a binary data object and in between contexts, while chunks are hashed. The sequence of hashes are saved for finding similarity between two objects. The context is not a fixed size, instead its size is decided by the contents of the object. In *ssdeep*, the sequence of hashes is a string sequence and each character corresponds to a chunk of the binary data object. To compare two objects, it uses a weighted version of *edit distance*.

Allison defined the *edit distance* [47]:

> The edit distance of two strings, s1 and s2, is defined as the minimum number of point mutations required to change s1 into s2, where a point mutation is one of the following:
>
> - Changing a letter,
> - Inserting a letter or
> - Deleting a letter

*Ssdeep* is more appropriate for relatively small objects that are similar sizes. It is vulnerable to a basic attack which inserts trigger sequences at the beginning of the file by exploiting the fact that an *ssdeep* signature value can have, at most, 64 characters [48]. The attack randomly generates trigger sequences and inserts them at the beginning of the file. The order is also important in the sequence generated by *Ssdeep*. My method does not depend on the order.

### 3.12.3 Sdhash

Roussev et al. [5, 49, 50] introduced the Similarity Digests Hashing method with a program called *sdhash* in 2010. It finds the features from a neighborhood that has the lowest probability of being encountered by chance. Each of the selected features, which is a 64 byte sequence, is hashed and put into a Bloom filter. When a filter reaches its full capacity, a new filter is generated. Thus, a similarity digest is a collection of a sequence of Bloom filters. A normalized entropy measure between digest are calculated and similarity score is returned. Sdhash compares the files using Hamming distance.

### 3.12.4 Mrsh-v2

Breitinger et al. [51] proposed *mrsh-v2*, which is based on *MRS* hash [33] and Context triggered piecewise hashing (CTPH). This algorithm uses a sequence of Bloom filters for fast comparison instead of a Base64 encoded fingerprint. It divides an input into chunks using a rolling hash. Each chunk is hashed and inserted into a Bloom filter. Like *sdhash*, *mrsh-v2* will have a variable length fingerprint, where it tries to have 0.5% of the input length. It has two modes: fragment detection and file similarity.

### 3.12.5 TLSH

Oliver et al.[52] presented TLSH, which is a locality sensitive hash. It populates an array of bucket counts by processing an input byte sequence using a sliding window. Quartile points are calculated from the array and then the digest header and digest body are constructed. Digest header values are based on the quartile points, the length of the file, and a checksum. The digest body consists of a sequence of bit pairs determined by each bucket's value in relation to the quartile points. TLSH assigns a distance score between two digests, which is a summed-up distance between the digest headers and the digest bodies. An approximate Hamming distance is used to calculate the distance between the two digest bodies. The distance between two digest headers is based on file lengths and quartile ratios. According to the experiments done in this paper [53], TLSH seems to be more robust to random

adversarial manipulations than *ssdeep* and *sdhash*.

# Chapter 4: Methodology

The digital media similarity measure I propose uses sector hashes as the input to a modified Jaccard index calculation, where the modifications are (1) a whitelist to remove low-entropy sectors, (2) a frequency weight to reflect content uniqueness, and (3) a normalization to account for differences in media size.

My proposed method is based on a modified Jaccard index similarity measure computed over digital media sector hashes. I compute similarity based on sectors present in different digital media, after adjusting for known common sectors (e.g., OS, low entropy) and weighting for sector frequency. I use a sector size of 512-bytes regardless of the media's actual sector size so that fragments will match across devices with different sector sizes, which I assume to be 512 bytes or 4096 bytes in most cases. I assume that files are stored on sector (cluster) boundaries, which is generally accepted to be true in the digital forensics domain. I note that sector hashing and other content-specific techniques, including my method, will not match the same data encrypted with different keys.

I introduce the foundational Jaccard index calculation over sector hashes in section 4.1.1. A whitelist is evaluated to remove low-entropy (non-discriminatory) sectors in sections 4.1.2 and 4.1.3. Sections 4.2 and 4.3 discuss the computation of this basic similarity for a single comparison drive and a set of comparison drives, respectively. In section 4.4, I modify this calculation to account for hash frequency, i.e., sector content uniqueness, and in section 4.5, I normalize my similarity measure calculation to account for differences in digital media sizes. This normalized similarity measure is named the Jaccard index with normalized frequency (**JINF**) and is the basic method for other improved methods.

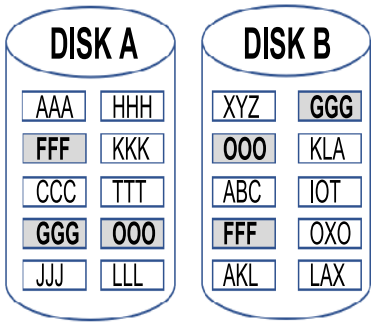


Figure 4.1: Two hard disks with three common sectors.

## 4.1 Jaccard index in sector hashing

In my work, the Jaccard index can be understood as the number of common sectors from two sources (e.g., hard disks A and B), divided by the number of all sectors from the two sources minus the number of common sectors in the two sources.

As shown in Figure 4.1, hard disks A and B have 10 sectors each, and the values in each square represent the hash values of each sector. Both have three of the same sectors. In this scenario, the Jaccard index will be **0.1765** : $3/(10 + 10 - 3)$.

In the sections that follow, I present the basis for my refinements to the Jaccard index in the context of digital media image similarity.

### 4.1.1 Similarity measure with Jaccard index

In this section, I present the general approach for computing the Jaccard index between two hard disks. Given two hard disks, each one is divided by sectors. For example, if we have a 1 Terabyte ($2^{40}$ bytes) hard disk and the sector size is 512 bytes, then $2^{31}$ sectors exist. If both hard disks are exactly same, for example one hard disk is an image copied

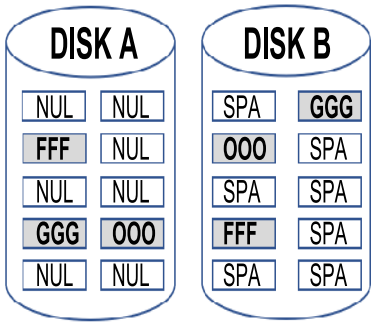


Figure 4.2: Display of whitelist sectors.

from another hard disk, then the Jaccard index will be ***1 (one)***, indicating high (perfect) similarity level between these two hard disks. If two hard disks have no common sectors at all, then the Jaccard index will be zero, i.e., no similarity. To calculate the Jaccard index **JI**(A, B), where $A$ is the set of all the sectors from the first drive and $B$ is the set of all sectors from the second drive, I replace $A$ with the set of all hash values which correspond to each sector in the first drive. Likewise, $B$ is replaced with the set of all hash values from the second drive. I applied the method in the previous section.

### 4.1.2 The impact of a whitelist

As seen in Figure 4.2, the seven non-matching sectors from drive A contain only NULL bytes and the seven non-matching sectors from drive B contain only SPACE characters. For convenience, assume in Figure 4.2 that the hash value of a NULL byte sector is NUL and the hash value of a SPACE sector is SPA. If I use the general method of Jaccard index, the Jaccard index value will be the same as Figure 4.1. However, the similarity level in the Figure 4.2 case should be higher than that of the case in Figure 4.1 because all the

meaningful sectors match in 4.2 (the NULL and SPACE sectors are not meaningful). In fact, the similarity level should be the highest, i.e., **1**. If I consider this situation in the calculation of the Jaccard index, I can get a more realistic Jaccard index value. In practice, it is better to report that these two drives match exactly instead of giving low similarity level based on the unadjusted Jaccard index, which would be 0.1765. I can achieve this goal by removing these NULL byte sectors or SPACE byte sectors from the sets being compared. Once removed from both drives, the total number of sectors are now three each. And all three sectors from both drives match exactly, hence I get **1** as the Jaccard index value. These NULL and SPACE byte sectors are members of my whitelist, which are removed prior to Jaccard index calculation.

### 4.1.3 Whitelist candidates

In addition to NULL byte and SPACE byte sectors, other candidates for the whitelist are low entropy sectors and sectors from a clean operating system (OS) install. If I install an operating system on a clean machine, any sectors written during the installation of the operating system will not contribute to a similarity measure and are good candidates for the whitelist. I collected the sector hashes of the OS on my test drives and saved them in a database for pre-filtering (exclusion) purposes. In practice, I could keep an inventory of OS sector hashes and update when new versions are released. Though the publicly available Known File Filter (KFF) method can be used, I applied simple filtering method since my test drives are M57-Patent scenario.

## 4.2 Similarity measure between two drives

Computing a similarity measure between two drives is straightforward but relies on the construction of a good whitelist. Once all the sectors are filtered with the whitelist, I can calculate the Jaccard index between two data sources (e.g., hard drives) as shown later in this chapter. In this dissertation, I use the term *target drive* for the drive which will be triaged using a similarity measure against a known and established *source drive*. For

Table 4.1: Jaccard index comparison chart of cluster drives.

| | $d_1$ | $d_2$ | .... | $d_k$ |
|---|---|---|---|---|
| $d_1$ | 1 | JI(1,2) | ... | JI(1,k) |
| $d_2$ | JI(2,1) | 1 | | JI(2,k) |
| ... | | | 1 | |
| $d_k$ | JI(k,1) | JI(k,2) | | 1 |
| Note: JI(i,j) = JI(j,i) | | | | |

example, a collection of hard drives from a known criminal organization are *source* drives. If I subsequently obtain a drive and want to know if it is similar to any of the criminal organization drives, then this new drive is a *target* drive.

## 4.3 Similarity measure against a cluster of drives

As a triage tool, we are often interested in the similarity of one target (hard drive) against different groups of sources (e.g., multiple hard drives), where these groups might represent different priorities, levels of interest, or specific staff or organizations. The drives in a cluster might be hard disks confiscated from terrorist groups or collections of hard disks that contain some malicious programs that are of interest. In other words, we have multiple different groups of source media. Once the whitelist is constructed and saved in a database, sectors of target drives and clusters of interest can be scanned and ignored based on this database, which will improve the speed and accuracy of the similarity measure calculation.

Given an image of interest and a cluster of labeled drives, one approach to assess the image of interest is to first calculate the Jaccard index between any two drives in the cluster. If we have k number of drives in the cluster, Table 4.1 shows the comparison chart among them.

Next, the image of interest can be compared against each drive in the cluster and the Jaccard index can be calculated, resulting in the comparison chart of Table 4.2. The last row ($d_t$) in Table 4.2 can be compared against the rows ($d_i$). From this table, I can give a statistically meaningful assessment by looking at the values from these two sets.

Table 4.2: Jaccard index comparison chart of target drive against cluster drives.

| | $d_1$ | $d_2$ | .... | $d_k$ |
|---|---|---|---|---|
| $d_1$ | 1 | JI(1,2) | ... | JI(1,k) |
| $d_2$ | JI(2,1) | 1 | | JI(2,k) |
| ... | | | 1 | |
| $d_k$ | JI(k,1) | JI(k,2) | | 1 |
| $d_t$ | JI(t,1) | JI(t,2) | | JI(t,k) |

Another approach is to create an imaginary drive containing all the sectors of the cluster drives. This imaginary drive is just the union of all sectors from drives in a cluster. If there are any sectors which are shared by multiple drives, they are each counted only once. I create a Jaccard index between the target sectors and this imaginary drive, resulting in the comparison chart of Table 4.3.

Table 4.3: Jaccard index comparison chart of target drive against imaginary drive.

| | Imaginary drive I |
|---|---|
| $d_1$ | JI(1,I) |
| $d_2$ | JI(2,I) |
| ... | |
| $d_k$ | JI(k,I) |
| $d_t$ | JI(t,I) |

## 4.4 Similarity measure with frequency

To make the analysis simple, I will use an imaginary drive **I** which will be used to simulate the source drive as shown in Table 4.5 and two target drives: $t_A$, and $t_F$. Imaginary drive **I** is created by combining all three drives: c1, c2 and c3 in the cluster as shown in Table 4.4. Let us assume that both target drives, $t_A$ and $t_F$, each have a single but different sector, A and F, respectively. Sector A in an imaginary drive is shared by every source drive in a cluster and sector F comes from only one cluster drive, **c2**. In this case, intuitively I can feel that the $t_A$ drive has more similarity than $t_F$ in the cluster comparison since sector A is

Table 4.4: Sector display of cluster and target drive.

Cluster drives.

| C1 | C2 | C3 |
|---|---|---|
| A | A | A |
| B | B | C |
| C | E | E |
| D | F | G |

Target drives.

| t1 | t2 |
|---|---|
| A | C |
| E | D |
| F | F |
| G | G |
| H | H |

Table 4.5: Sector frequency display of Imaginary drive and target drives.

Frequency display of I.

| Imaginary Drive I | |
|---|---|
| A | 3 |
| B | 2 |
| C | 1 |
| D | 1 |
| E | 2 |
| F | 1 |
| G | 1 |
| Total | 11 |

Frequency display of Target drives.

| t1 | | t2 | |
|---|---|---|---|
| A | 3 | C | 1 |
| E | 2 | D | 1 |
| F | 1 | F | 1 |
| G | 1 | G | 1 |
| H | 1 | H | 1 |
| Total 8 | | Total 5 | |

contained in every cluster drive. To prove this intuition with calculation, the frequency of the individual sector is used as an adjusting factor in the Jaccard index calculation. Table 4.4 shows the sectors in each drive (c1, c2, c3) of a cluster and the sectors in each target drive (t1, t2). Table 4.5 shows the frequency of the sectors in the Imaginary drive and each target drive (t1, t2).

The standard Jaccard index for both **JI**(t1, I) and **JI**(t2, I) will be 4/8 (0.5).

$$\mathbf{JI}(t1, I) = \frac{A, E, F, G}{(A, E, F, G, H) + (A, B, C, D, E, F, G) - (A, E, F, G)} \tag{4.1}$$

$$\mathbf{JI}(t2, I) = \frac{C, D, F, G}{(C, D, F, G, H) + (A, B, C, D, E, F, G) - (C, D, F, G)} \tag{4.2}$$

However, t1 is more similar than t2 since t1 has sector A, which is shared by all cluster

drives (c1, c2 and c3) and it has sector E which is shared by two drives (c2 and c3). A new modified Jaccard index with frequency (**JIWF**) between two drives (d1, d2) is introduced here.

$$JIWF(d1, d2) = \frac{number\ of\ common\ sectors\ with\ frequency\ from\ d1\ and\ d2}{number\ of\ all\ sectors\ with\ frequency\ from\ d1\ and\ d2} \tag{4.3}$$

When **JIWF** is calculated, if a sector is shared by $n$ number of drives, that sector is counted $n$ times. The numerator of **JIWF**(t1, I) is 7 because there are four common sectors (A, E, F, and G) and A is counted 3 times and E is counted 2 times. The denominator of **JIWF**(t1, I) is 12 because there are a total of 8 sectors (A, B, C, D, E, F, G, H) and A is counted 3 times and B and E are counted 2 times respectively. Now **JIWF**(t1, I) is $\frac{7}{12}(0.583)$ and **JIWF**(t2, I) is $\frac{4}{12}(0.333)$. This new method, **JIWF**, shows that t1 is more similar than t2 since it incorporates frequency in the calculation of Jaccard index.

## 4.5 Similarity measure with normalized frequency

So far, the assumption that I used in the experiment was that the sizes of the target and cluster drives are similar. If the target drive is a thumb drive and the cluster drive is a drive of several terabytes, then I ask:

- Can we use the same method we suggested here in the previous sections?
- If we can use the same method, how do we interpret the value of similarity measure?
- If we cannot use the same method, how can we approach this issue?

To answer the above questions, a new approach is proposed. For the standard calculation of Jaccard Index, intersection and union are used. For the new approach, I come up with a modified definition of Intersection and Union: $Intersection^*$ ($I^*$) and $Union^*$ ($U^*$). Given two numbers, the Intersection* and Union* of these two numbers are defined below.

Table 4.6: Hash value and frequency of source drive.

| Hash value | Frequency | Normalized Frequency |
|---|---|---|
| A | 5 | 0.3333 |
| B | 4 | 0.2667 |
| C | 3 | 0.2 |
| D | 2 | 0.1333 |
| E | 1 | 0.0667 |
| total | 15 | 1 |

Table 4.7: Hash value and frequency of target drive: target 1.

| Hash value | Frequency | Normalized Frequency |
|---|---|---|
| A | 1 | 0.0667 |
| B | 2 | 0.1333 |
| C | 3 | 0.2 |
| D | 4 | 0.2667 |
| E | 5 | 0.3333 |
| total | 15 | 1 |

$$Intersection^*(N1, N2) = Min(|N1|, |N2|)$$
$$Union^*(N1, N2) = Max(|N1|, |N2|)$$

where N1, N2 are normalized frequencies.

I also use the normalized frequency. Given a frequency, C, and total frequency, T, The normalized frequency of C: $\mathbf{N = C/T}$. This normalization is introduced to take care of the size difference between target and cluster drives. In this method, T is the total number of sectors in a drive and C is the frequency of a certain sector hash value. For each hash value, I can calculate its normalized frequency.

Suppose we have a simplified source and target drive as shown in Table 4.6 and Table 4.7 respectively.

To calculate the Jaccard index, I compute the two normalized values for each hash value: one from the source and the other from the target. The following Table 4.8 shows the $Intersection^*$ and $Union^*$ of the two normalized values.

Table 4.8: $Intersection^*$ and $Union^*$ of two normalized frequency values.

| Hash value | Normalized frequency of source | Normalized frequency of target | $Intersection^*$ | $Union^*$ |
|---|---|---|---|---|
| A | 0.3333 | 0.0667 | 0.0667 | 0.3333 |
| B | 0.2667 | 0.1333 | 0.1333 | 0.2667 |
| C | 0.2 | 0.2 | 0.2 | 0.2 |
| D | 0.1333 | 0.2667 | 0.1333 | 0.2667 |
| E | 0.0667 | 0.3333 | 0.0667 | 0.3333 |
| total | | | 0.6 | 1.4 |
| | | | **JINF =** | **0.4286** |

When I add all the $Intersection^*$ values of each hash, the total is 0.6. Likewise, the sum of all $Union^*$ values is 1.4. Here I define the Jaccard Index with Normalized Frequency **(JINF)** of two drives: S and T.

$$\textbf{JINF}(S,T) = \frac{sum\ of\ all\ Intersection^*(S,T)}{sum\ of\ all\ Union^*(S,T)} \tag{4.4}$$

In this example, the **JINF** value will be $0.6/1.4 = 0.4286$. As a note, the **JINF** value of the two same drives will be 1.0 with this approach since $Intersection^*$and $Union^*$ values of normalized frequency of each sector hash are the same.

## 4.6 Effectiveness of JINF

Tables 4.9 and 4.10 demonstrate how the **JINF** value is changing when the frequency of sector hash A increases by one. In these tables, the normalized frequency of each drive is hidden. As the frequency of the first block of the target drive, **A**, moves toward the frequency of the same sector hash, **A**, of the source drive, the similarity level should be increased. We may consider each case as a new target drive to be checked against the source drive. In each case, we see that the **JINF** value showing similarity level increases when the frequency of sector hash **A** increases. We also note that the total number of blocks increases by one as we increase the frequency of sector hash **A** by one. This increase of total number of blocks will bring down the similarity level since the portion of each block

Table 4.9: **JINF** comparison: target 2, target 3 and target 4.

| | target 2 | | | target 3 | | | target 4 | | |
|---|---|---|---|---|---|---|---|---|---|
| Hash | Frq | $I^*$ | $U^*$ | Frq | $I^*$ | $U^*$ | Frq | $I^*$ | $U^*$ |
| A | **2** | 0.125 | 0.3333 | **3** | 0.1764 | 0.3333 | **4** | 0.2222 | 0.3333 |
| B | 2 | 0.125 | 0.2667 | 2 | 0.1176 | 0.2667 | 2 | 0.1111 | 0.2667 |
| C | 3 | 0.1875 | 0.2 | 3 | 0.1764 | 0.2 | 3 | 0.1667 | 0.2 |
| D | 4 | 0.1333 | 0.25 | 4 | 0.1333 | 0.2352 | 4 | 0.1333 | 0.2222 |
| E | 5 | 0.0667 | 0.3125 | 5 | 0.0666 | 0.2941 | 5 | 0.0667 | 0.2778 |
| Sum | 16 | 0.6375 | 1.3625 | 17 | 0.6705 | 1.3294 | 18 | 0.7 | 1.3 |
| **JINF** | 0.4678 | | | 0.5044 | | | 0.5384 | | |

against the total number of blocks decreases. However, the positive effect of increasing the frequency of sector hash **A** is greater than the negative effect of increasing the total number of blocks. Similarity level increases when we add the frequency of the sector hash **A**. For example, let us create a new target drive 6 from target drive 5. If we reduce the frequency of sector hash **E** from 5 to 4 in target drive 6, we get 0.6071 as the **JINF** value of target 6. This is higher than the **JINF** value of target 5, which is 0.5702. The reason is that the total number of blocks of target drive 6 is now closer to the value of the source drive and has less negative impact to the calculation of the **JINF** compared to target 5.

To show how well this method copes with the size difference of the source and target drives, I created a target drive 7. The frequency of each block is copied from target drive 1 shown in Table 4.7 and multiplied by 10. The **JINF** value of both drives, target 1 and target 7, are the same because the normalized frequency of each hash block is the same for both drives. Therefore, this method does not depend on the size of the drives to be measured.

So far, example tables contain the same hashes with different frequency. For example, if the source has an uncommon hash, **F**, that the target 1 does not have, we can expect that this **JINF**(new source, target 1) will be smaller than the other **JINF**(source, target 1), which is shown in Table 4.8. In the new source, the normalized frequencies of each hash will be smaller than those of the source because of increased total sectors, which is shown in Table 4.11. $Intersection^*$ of the uncommon hash **F** will be 0. This will decreases the

Table 4.10: **JINF** comparison: target 5, target 6 and target 7.

| | target 5 | | | target 6 | | | target 7 | | |
|---|---|---|---|---|---|---|---|---|---|
| Hash | Frq | $I^*$ | $U^*$ | Frq | $I^*$ | $U^*$ | Frq | $I^*$ | $U^*$ |
| A | **5** | 0.2631 | 0.3333 | **5** | 0.2778 | 0.3333 | **10** | 0.0667 | 0.3333 |
| B | 2 | 0.1052 | 0.2667 | 2 | 0.1111 | 0.2667 | 20 | 0.1333 | 0.2667 |
| C | 3 | 0.1578 | 0.2 | 3 | 0.1667 | 0.2 | 30 | 0.2 | 0.2 |
| D | 4 | 0.1333 | 0.2106 | 4 | 0.1333 | 0.2222 | 40 | 0.1333 | 0.2667 |
| E | 5 | 0.0667 | 0.2631 | 4 | 0.0667 | 0.2222 | 50 | 0.0667 | 0.3333 |
| Sum | 19 | 0.7263 | 1.2736 | 18 | 0.7556 | 1.2444 | 150 | 0.6 | 1.4 |
| **JINF** | 0.5702 | | | 0.6071 | | | 0.4286 | | |

Table 4.11: Hash value and frequency of new source drive.

| Hash value | Frequency | Normalized Frequency |
|---|---|---|
| A | 5 | 0.25 |
| B | 4 | 0.2 |
| C | 3 | 0.15 |
| D | 2 | 0.1 |
| E | 1 | 0.05 |
| F | 5 | 0.25 |
| total | 20 | 1 |

sum of $Intersection^*$ from 0.6 to 0.5. However, $Union^*$ of **F** has a positive value, which will increase sum of $Union^*$ from 1.4 to 1.5. The final result of **JINF**(new source, target 1) is 0.3333 and it is smaller than **JINF**(source, target 1), which is 0.4286.

Table 4.12 shows the calculation of **JINF** values when the new source image has an uncommon hash, **F**.

Table 4.12: $Intersection^*$ and $Union^*$ between new source and target 1.

| Hash value | Normalized Frequency of new source | Normalized Frequency of target | $Intersection^*$ | $Union^*$ |
|---|---|---|---|---|
| A | 0.25 | 0.0667 | 0.0667 | 0.25 |
| B | 0.2 | 0.1333 | 0.1333 | 0.2 |
| C | 0.15 | 0.2 | 0.15 | 0.2 |
| D | 0.1 | 0.2667 | 0.1 | 0.2667 |
| E | 0.05 | 0.3333 | 0.05 | 0.3333 |
| **F** | 0.25 | 0.0 | 0.0 | 0.25 |
| total | | | 0.5 | 1.5 |
| | | | **JINF =** | **0.3333** |

## 4.7 Conclusion

**JINF** is a digital media similarity measure based on digital media sector content comparisons and a variant of the Jaccard index. The modified Jaccard Index calculation adjusts set intersection and union counts based on the underlying frequency of the set members, which are hash values and represent the contents of digital media sectors in my work. My normalization methods allow **JINF** to apply to digital media of different sizes. The method is also independent of file system information and form factor, so applies across operating systems, file systems, and different physical devices.

# Chapter 5: Results and Validation of JIWF and JINF

I validated both proposed similarity measures (**JIWF** and **JINF**) introduced in the previous chapter using a set of disk images with known similarity. Comparison between these two methods is discussed in this chapter. The M57 Patents Scenario dataset [28] consists of 68 hard disk images. These images are taken from four distinct systems (named Pat, Terry, Jo, and Charlie) over a 25-day period (each system was imaged 17 times during the 25-day experiment).

For my purposes, each of the four systems represents a similar set of images. This is due to the fact that they are the same source systems, with the only differences being due to the normal use incurred by the experiment. I build sets of similar images using a subset of one user's images, then compare one of that user's other images to the set. I would expect high similarity, which in fact is what I found. My validation was intended as a preliminary confirmation that the proposed measure is computationally correct and not as a scalability test; additional testing is planned against the Real Data Corpus, a data set of thousands of disparate media sources with no ground truth.

## 5.1 Initial validation of the method

For the initial validation of the methods described in section 4, sequential snapshots of a single drive were used. On a clean drive, the following sequence of actions are performed after the Windows operating system is installed:

- (a) an application is *installed*,
- (b) an application is *opened*,
- (c) an application is *closed*,

- (d) an application is *uninstalled*, and
- (e) system is *rebooted*.

The above procedure is repeated for three clean drives with three different applications: Wireshark, Firefox, and Safari. Let us call these generic drives as $WS$, $FF$ and $SA$. After each step in the above sequence, snapshots of the hard disk image are saved for each drive. They are named as $WS_a$, $WS_b$, $WS_c$, $WS_d$ and $WS_e$ for drive $WS$. For example, $WS_c$ is the snapshot of drive WS after Wireshark is closed, which is action **(c)** in the above sequence. Likewise, I name the snapshot of the drive for Firefox and Safari as $FF_x$ and $SA_x$ respectively, where $x$ is one of $\{a, b, c, d, e\}$, and here I say that every $FF_x$ are in the same category. For example, $WS_a$ and $WS_b$ are in the same category with $WS_c$, but not in the same category with any snapshots of $FF$ drive.

## 5.2 Whitelist sector removal

As described above, sectors collected after Windows O/S installation and action (a) correspond to whitelist candidates. Therefore, sectors from snapshots $WS_a$, $FF_a$ and $SA_a$ were added to the whitelist database. The elimination of these sectors from consideration reduces the computational effort and enhances the accuracy of the Jaccard index.

## 5.3 Jaccard index with frequency (JIWF)

For this research, I used the hashdb [37] tool. To compute the Jaccard Index between two drives, A and B, first I remove all the sectors of the Windows OS from each drive and then construct a hashdb for each slimmed drive: h-A and h-B. An intersection of hashdbs, say Int(h-A, h-B), is constructed by the hashdb command **intersect hash** (h-A, h-B) and union of hashdbs, Un(h-A, h-B), is done by **add_multiple** (h-A, h-B). I also used the result of the hashdb *size* command as input to the Jaccard index calculation. The *size* command returns **hash data store** and **hash store** values. I use the **hash store** value as the input

Table 5.1: Jaccard indices between snapshots in the same category.

| | $WS_b$ | $WS_c$ | $WS_d$ |
|---|---|---|---|
| $WS_b$ | | 0.08858 | 0.0339 |
| $WS_c$ | 0.08858 | | 0.4419 |
| $WS_d$ | 0.0339 | 0.4419 | |
| $WS_e$ | 0.0193 | 0.2756 | 0.6160 |

to the Jaccard index calculation.

$$TheJaccardIndex(A,B) = \frac{hash\ store\ value\ of\ Int(h\text{-}A, h\text{-}B)}{hash\ store\ value\ of\ Un(h\text{-}A, h\text{-}B)}$$

The above formula is used to approximate the Jaccard index with frequency, $JIWF$. To create an imaginary drive $I(h\text{-}A, h\text{-}B, h\text{-}C)$ from three hard disk images: A, B, and C, the hashdb **add_multipe** command is used.

$$I(h\text{-}A, h\text{-}B, h\text{-}C) = add\_multiple(h\text{-}A, h\text{-}B, h\text{-}C)$$

Next, all the **JI** values between each snapshot were calculated: **JI**$(WS_b, WS_c)$, **JI**$(WS_b, WS_d)$, **JI**$(WS_b, WS_e)$, **JI**$(WS_c, WS_d)$, **JI**$(WS_c, WS_e)$ and **JI**$(WS_d, WS_e)$. Jaccard indices between $WS_b$ and other instances becomes smaller after each sequence of actions as shown in Table 5.1. This is expected since a drive with an installed application and a drive with the same application uninstalled are less similar. Once the system is rebooted after uninstallation, previously matching sectors are overwritten and no longer match. A similar trend was observed between $WS_c$ and other instances.

The Jaccard indices of any two drives from different categories were very low compared to Jaccard indices of any two drives within the same category, as shown in Table 5.2

When I have not calculated the Jaccard indices between pairs, I left the corresponding cell as empty in Table 5.1 and Table 5.2.

Table 5.2: Jaccard indices comparison between snapshots.

| | $FF_c$ | $FF_d$ |
|---|---|---|
| $FF_c$ | | 0.5478 |
| $FF_d$ | 0.5478 | |
| $FF_e$ | 0.1694 | 0.2807 |
| $WS_c$ | 0.00049 | 0.00042 |
| $WS_d$ | 0.00227 | 0.00193 |
| $WS_e$ | 0 | 0.00451 |

## 5.4 Similarity measure between a target drive and a cluster of drives

In the previous section, I used simple drive images to validate my methodology. A single application was installed on top of the Windows 7 OS, and I took snapshots after each action. For this next test, I use realistic data sets created by a project funded by the National Science Foundation. One of these data sets is the **2009-M57-Patents scenario**[54]. This scenario involves a small company called M57, which was engaged in prior art searches for patents. The employees' actions in the fictitious patents research firm are recorded on a private network. The 2009-M57-Patents scenario consists of activities for a 17 day period in the months of November and December, 2009. The company has four employees: Pat (CEO), Charlie (patent researcher), Jo (patent researcher), and Terry (IT administrator). The hard drive of each employees' workstation was imaged daily, except for weekends and holidays. Three workstations have windows XP installed, and Windows Vista Business 32-bit was installed on Terry's workstation.

From each drive, I created a corresponding hashdb instance. I also created a hashdb isntance for a clean hard drive, which has only the Windows OS (XP and 7). I used the hashdb **subtract** command to remove Windows OS sectors from each employees' hashdb. I call these *slimmed* hashdbs. From the list of slimmed hashdbs, I created an imaginary drive by choosing five random slimmed hashdbs.

I created I_S_Charlie.hdb, an Imaginary hashdb from Nov-11, Nov-20, Nov-30, Dec-04

and Dec-10 hashdbs of slimmed Charlie hashdbs. This imaginary drive simulates the cluster of drives that I want to find similarity with when compared to a target drive. For each target drive hashdb, I generated the Jaccard index against I_S_Charlie.hdb. If a target drive and cluster drives are from different categories, I expect that the similarity measure will be much lower.

5.1 shows some results of this scenario. I again used the same method which was explained in the previous section.

The hashdb *size* command returns the number of entries in the LMDB database [55]. It returns two values: **hash data store** value and **hash store** value. The LMDB Hash Store is a highly compressed optimized store of all the block hashes in the database. When scanning for a hash, if it is not in this store, then it is not in the database. Because of the degree of optimization, there can be false positives. To compensate, when a hash is found in the LMDB Hash Store, hashdb reads the LMDB Hash Data Store to be sure the hash actually exists. The LMDB Hash Data Store is a multi-map store of all hashes and their associated data and source information [56].

I created an Imaginary cluster drive for each employee and calculated the Jaccard Index. Figure 5.1 shows the Jaccard index between the Charlie Imaginary cluster drive and each employee. As days go by, the Jaccard Index of Charlie's daily hard drive shows higher values compared to other employees. The hard disk images which were used to generate Charlie's imaginary drive are marked with @ in Figure 5.1. For the other three employees, I see a similar pattern, as shown in Figures 5.2, 5.3, and 5.4. For Terry, the absolute Jaccard Index is lower compared to other employees because the size of Terry's daily hard drive is bigger than other employee's drive. Except for the first few Jo images, all indicated the correct result.

## 5.5 Jaccard Index with normalized frequency (JINF)

In the previous section, I introduced the Jaccard index with frequency method (**JIWF**). I applied this method to the 2009-M57-Patents scenario. After building hashdb for each

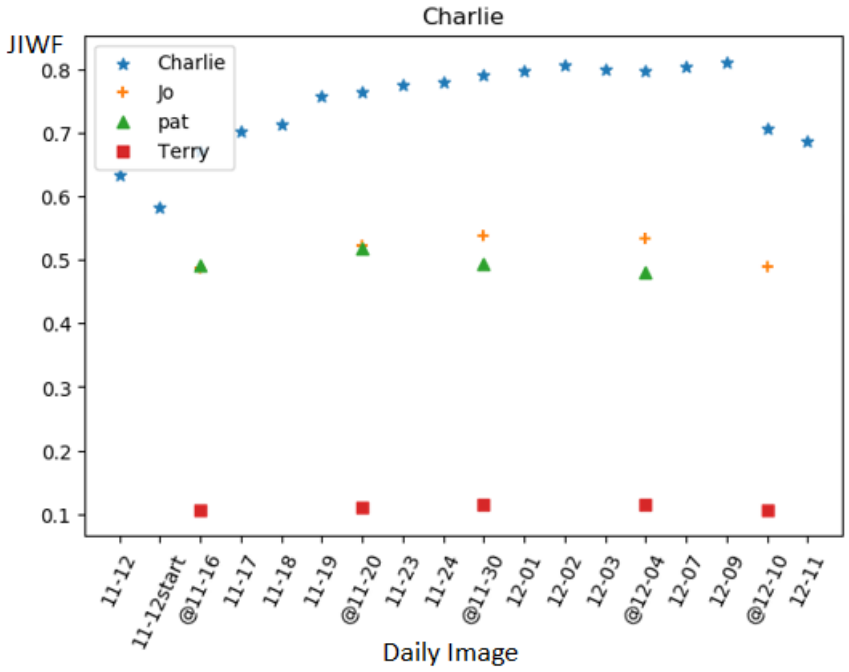


Figure 5.1: Jaccard index frequency of Charlie.

day of each employee, the hashdb **subtract** command was used to remove OS blocks from each hashdb. I also created one imaginary source drive for each employee by combining 5 random hashdbs from that employee's own daily drive image list. The hard disk images which were used to generate Charlie's imaginary drive are marked with @ in Figure5.5. Similarly, randomly chosen daily images are marked with @ for Jo, Pat and Terry in Figure 5.6, 5.7, and 5.8 respectively. Each figure shows the Jaccard index value comparing four imaginary drives and target drives for each employee. For example, Figure 5.5 for Charlie compares four imaginary source drives with Charlie's daily target drive images. The x-axis shows the list of target drives from Charlie and the Y-axis shows the Jaccard index value. The asterisk, (*), shows the Jaccard index value against Charlie's Imaginary drive. As you can see, all four employee's target drives show a higher Jaccard index value when they are checked against their own imaginary source drives. As a common behavior, the Jaccard index value of all four employees against their own imaginary disks increases steadily for the

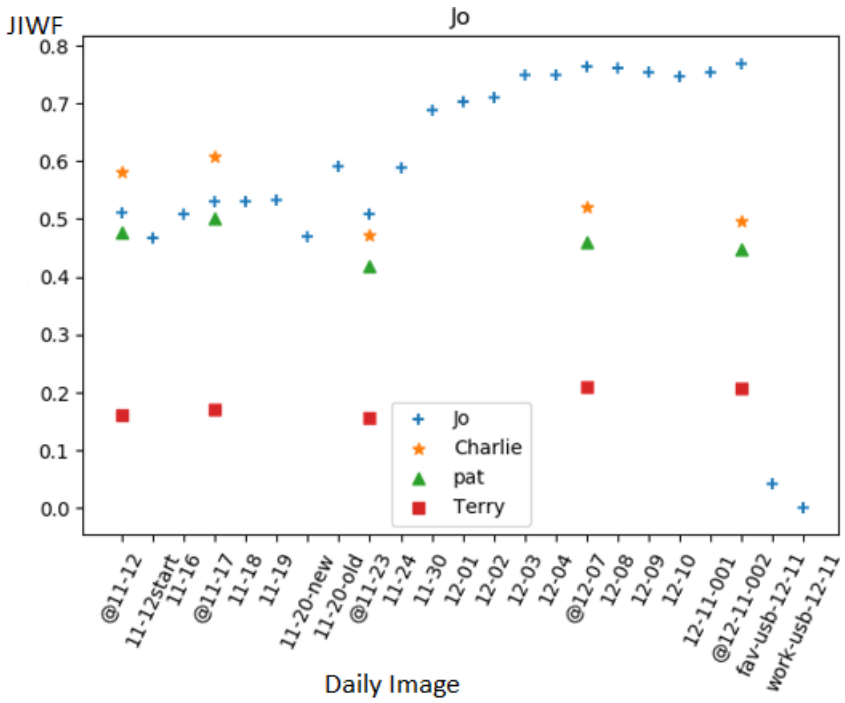


Figure 5.2: Jaccard index frequency of Jo.

first 10 days, stabilizes, and then drops during the last few days. This can be explained. As time passes, the similarity level between each day and the imaginary disk increases, and once it reaches a certain usage level, the similarity level between two days has little distinction. The last few days, the similarity level drops because files are deleted and sector contents overwritten, leaving fewer sectors to match the static imaginary drive. The similarity level between the USBs of Jo and the imaginary hard disk is almost zero, which I expected. In this test, a total of 305 hard disk images were compared to 4 imaginary disk images. Except the first five of Jo's images (see Figure 5.6), all showed the correct results, which is more than 98% accuracy. Those five incorrect results are from images taken chronologically prior to images forming the cluster set, and even so are still near the top candidates on the same day.

From the given daily 40-gigabyte hard disk image, it took 90 minutes on average to obtain a **JINF** value onto a personal computer with an Intel(R) Core(TM) i7 @ 2.30GHz

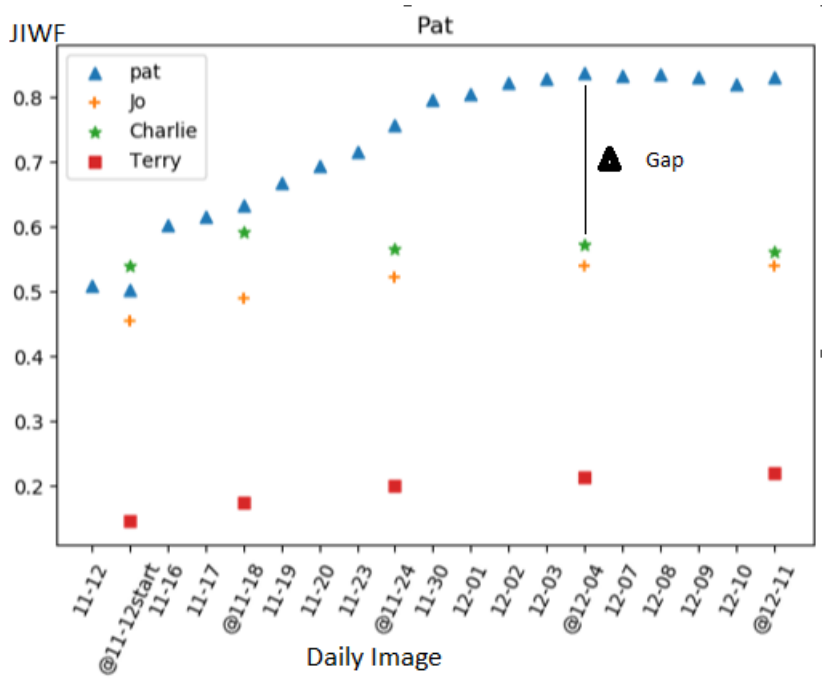


Figure 5.3: Jaccard index frequency of Pat.

CPU and a 2-terabyte SSD memory.

## 5.6 Comparison of JIWF and JINF

In the previous section, I showed the results of applying these two methods, **JIWF** and **JINF**, to the same M57 data set. Both provide accurate results and would support triage decisions. But the Jaccard index with normalized frequency shows a greater gap between the correct results and the next highest similarity score. Consider both of Pat's graphs on the date of Dec 4th in Figure 5.3 and 5.7. Both graphs show the calculated Jaccard index of every drive in M57 against Pat's imaginary drive. Hence, I expect a higher similarity measure for any of Pat's drives. In Figure 5.7, where Jaccard index of normalized frequency is applied, Pat's hard drive has a similarity measure of 0.7, while Charlie and Jo's drives show a 0.3 similarity measure. On the other hand, in Figure 5.3, where the non-normalized method is applied, Pat's drive has 0.8 and Charlie and Jo's drives have 0.55 as the similarity

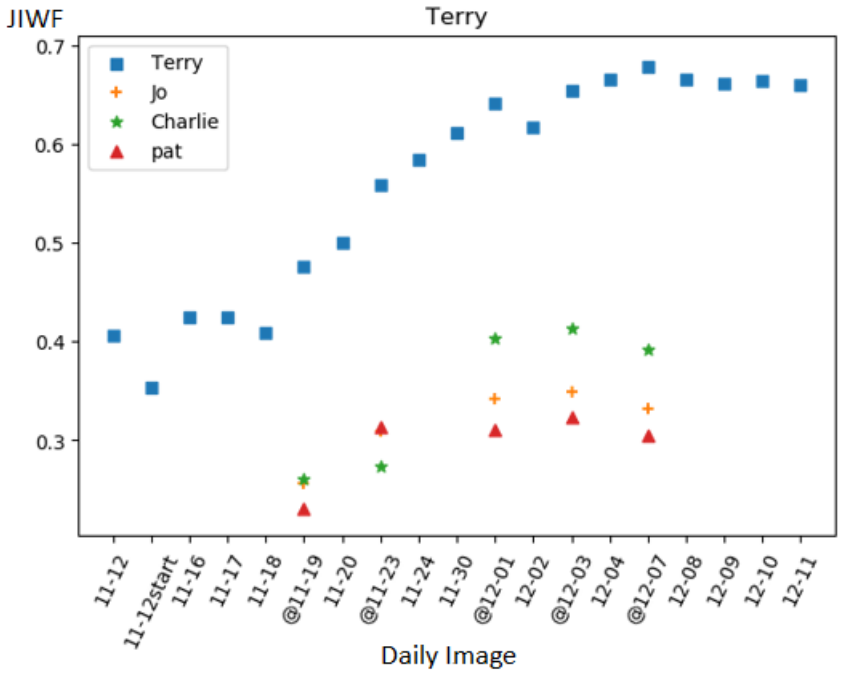


Figure 5.4: Jaccard index frequency of Terry.

measure. The gap 0.4 (0.7-0.3) when using the normalized frequency method is bigger than 0.25 (0.8-0.55) for the non-normalized frequency method. For all four employees I can observe similar behavior: a bigger gap for the normalized method, as shown in Table 5.3. As a side note, for Jaccard index with normalized frequency method, when I eliminated high frequency sector hashes in the calculations, the resulting similarity measure was greater. As I gradually lower the cut-off value of high frequency sector hashes from 5000 to 50, the similarity value increases accordingly, which I expected. In other words, lowering the sector hash frequency cutoff reduces "noise" hashes and concentrates the calculation on the most significant matches.

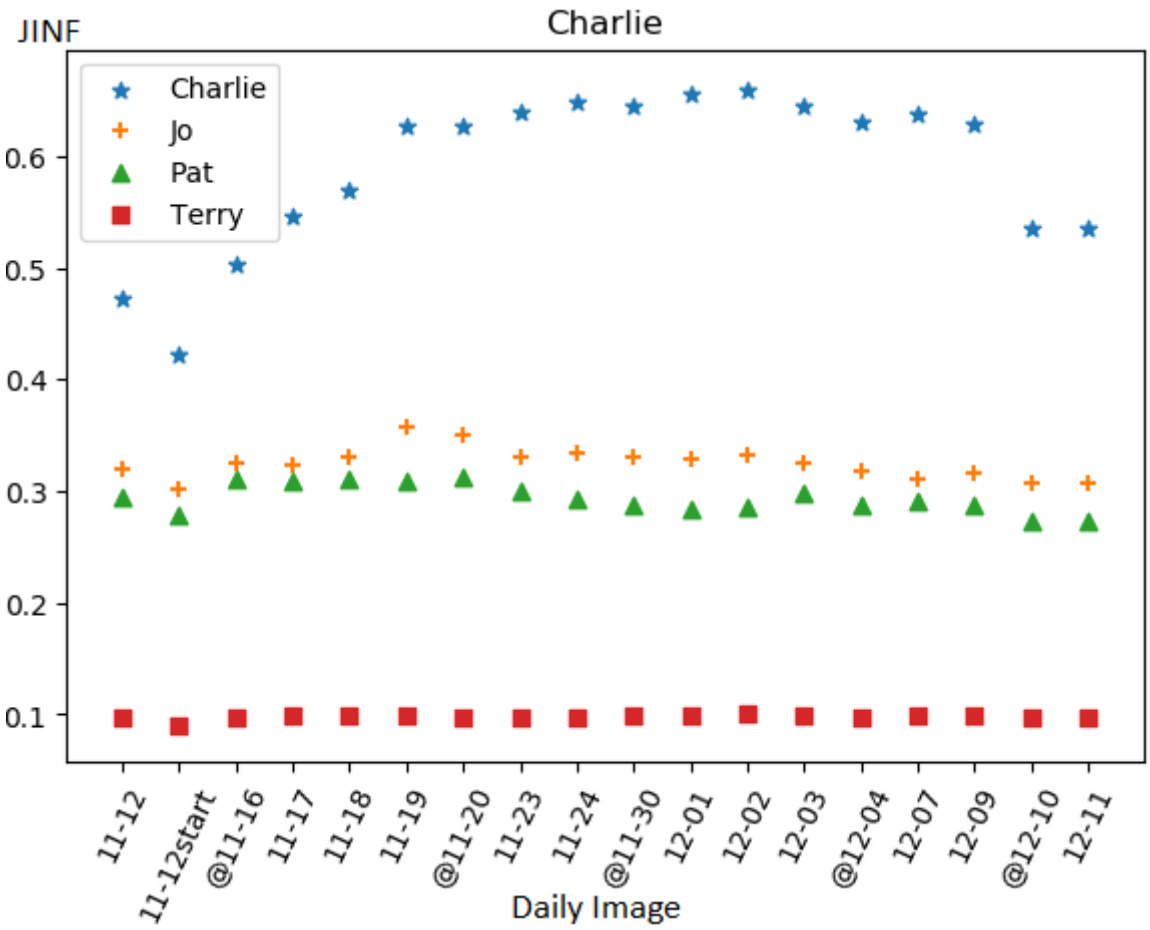


Figure 5.5: Jaccard index normalized frequency of Charlie.

Table 5.3: Performance comparison between **JIWF** and **JINF**.

| | JIWF | JINF |
|---|---|---|
| Charlie | 0.229789 | 0.265333 |
| Jo | 0.085746 | 0.139794 |
| Pat | 0.146044 | 0.279771 |
| Terry | 0.244807 | 0.283844 |
| Avg | 0.176597 | 0.242186 |

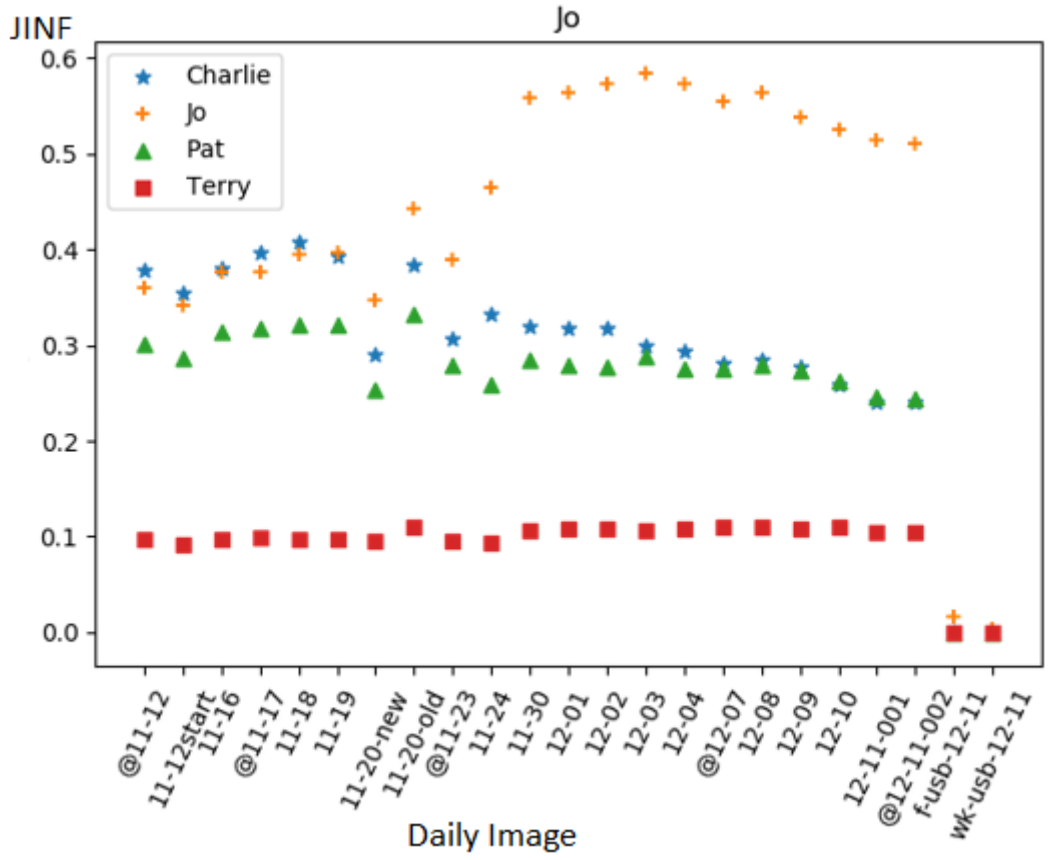


Figure 5.6: Jaccard index normalized frequency of Jo.

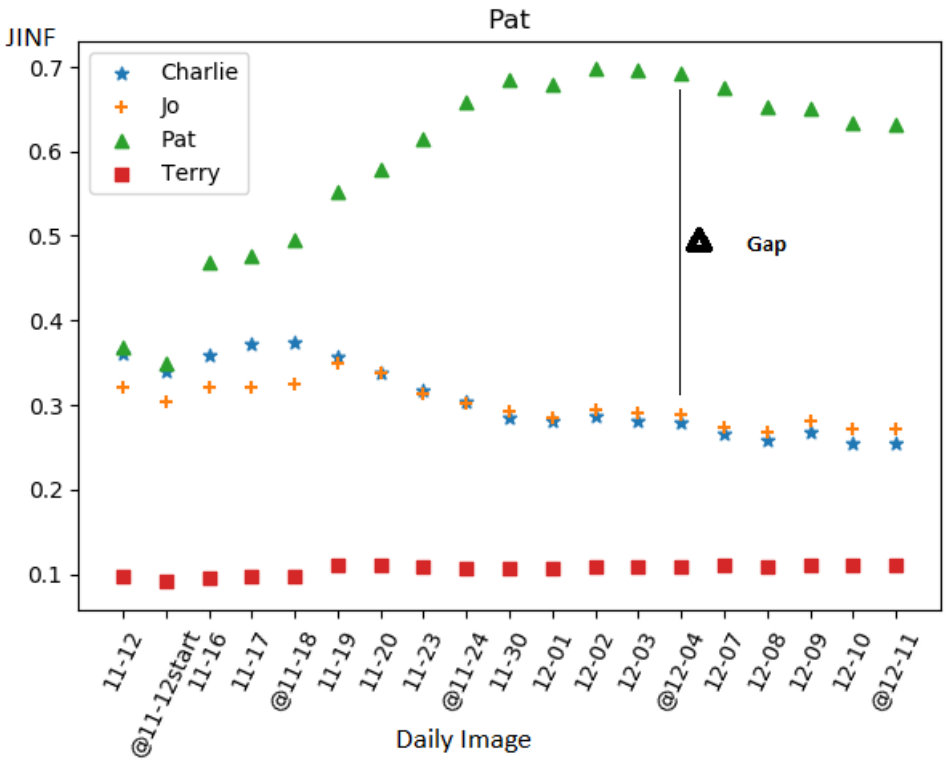


Figure 5.7: Jaccard index normalized frequency of Pat.

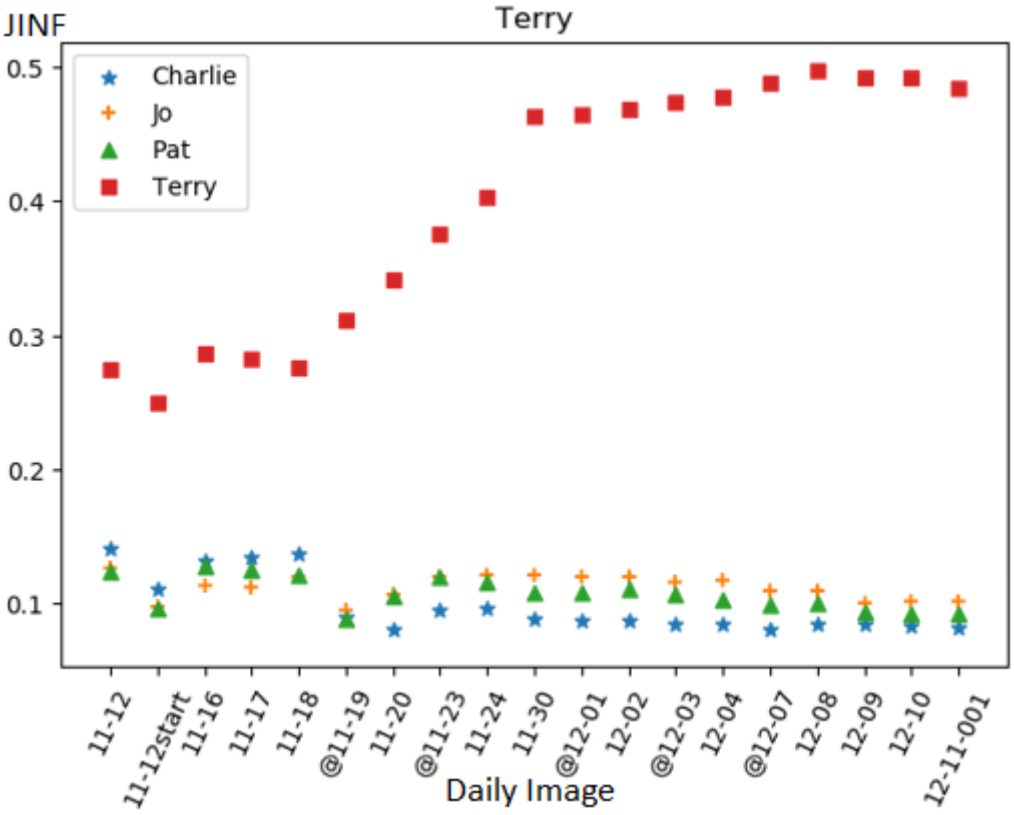


Figure 5.8: Jaccard index normalized frequency of Terry.

# Chapter 6: Jaccard Index with Sampling (JIWS)

In this chapter, I introduce a modified **JINF** which can be used for sampled images. The computation time it takes to calculate similarity is a critical factor for field analysts, especially whose purpose is to pre-triage of the huge list of evidence images. To reduce the processing time to calculate the **JINF** between *Source* and *Target* images, I created several sampled images from the *Target* image by varying the sampling rate.

## 6.1 How to create sampled images

To create a sampled image, I choose a random sector by using a random number generator. Once a sector is selected, I check if this is a Null byte sector or an OS sector . If it is a Null byte sector or an OS sector, it is discarded and I choose another one. This step is repeated until the decided sampling rate is achieved.

I sampled from all the daily images of 4 employees of M57 with 7 different sampling rates. Each daily image is slimmed by removing white sectors such as OS sectors or Null byte sectors as a pre-processing step, known as whitelist removal. The sampling rates were 5, 10, 20 , 30, 40, 50 and 75 percent. $TRGT_{10}$ represents the sampled image from a slimmed *Target* image with a 10 percent sampling rate.

## 6.2 Processing time with sampled images

It takes a few minutes for a 10 GB hard drive and 25 minutes for a 40 GB hard drive to extract sampled images from whole images. Once sampled images are created, JINF method is applied to them. The **JINF** calculation time between two images of 10 GB takes 16 minutes. The average **JINF** calculation time for each sample rate takes an average of 8,

9, 10, 11, 12, 13 and 14 minutes for the sample rates of 5, 10, 20, 30, 40, 50 and 75 percent, respectively, onto a personal computer with an Intel(R) Core(TM) i7 @ 2.30GHz CPU and a 2-terabyte SSD memory.

## 6.3 Analysis of JINF with sampled images

We observe how the **JINF** between *Source* and sampled images from *Target* differs from the **JINF** between *Source* and *Target*.

Each sampled image is compared to the daily images. As a guide, the **JINF** between full images are calculated also. I re-wrote the algorithm with Python and the *groupby* function call of the *Pandas* module to simplify the code and improve efficiency. If I get a similar value for the **JINF** between *Source* and sampled images, then sampling from *Target* is worth doing to save time. If the **JINF** works even with a sampled image of low sampling rate, this is better since I can save time in creating a sampled image and processing **JINF**. Unfortunately, this is not the case.

In the previous calculation of **JINF**, I overcame the low value of similarity, which came from the size difference of source and target images by normalizing the hash count. However, I notice that the low sample rate generates low **JINF** not because of small image size, but because of the small number of common hash counts between source and target images.

## 6.4 Approach to solve the issue of JINF with sampled images

If the sampling rate is doubled, then **JINF** also increases accordingly. Therefore, to be more reasonable in deciding similarity, I conclude that to decide the similarity, the set of sectors collected from the sampled image should be weighted more than other sectors from the source. It is not fair that the not-chosen sectors in sampling are treated the same as the chosen sectors. Some kind of normalization of JINF method was needed to apply **JINF** to sampled images. Thus, I came up with a new novel method called the Jaccard Index with Sample (**JIWS**). This approach is based on **JINF** which changes linearly with sample rate

and also depends on the number of common hashes between two images.

When I design my heuristics, **JIWS**, I have two goals in mind. Since I already validated **JINF**, my first goal is to have heuristics behave similarly to **JINF**. Meaning, given two target drives, T1 and T2, if **JINF**(Source, T1) is higher than **JINF**(Source, T2), **JIWF**(Source, T1) should be higher than **JIWS**(Source, T2). Finally, my second goal is that heuristics should not be impacted by the sample size.

## 6.5 JIWS formula

For the sampled images, the new approach, which is **JIWS** between *Source* and *Target*(sample image), is presented here.

$$\textbf{JIWS} = CommonRatio * [\textbf{JINF} + CommonRatio * (1 - \textbf{JINF})] \tag{6.1}$$

$$CommonRatio(Source, Target) = \frac{unique\ hash\ count\ of\ common}{total\ unique\ hash\ count\ of\ Target} \tag{6.2}$$

The *CommonRatio* is defined in the formula (6.2) for the common hashes between two images. The denominator count is selected from *Target*, which is assumed to be smaller than *Source*. If the sampled image is bigger than *Source*, I use the count from *Source* image in the denominator.

## 6.6 JIWS with an example

I will explain **JIWS** with an example. As seen in Table 6.1 , *Source* and *Target* have 15 total sectors. Each has 11 unique sectors with different hash frequencies as shown in Table 6.2.

Table 6.3 shows the 33 percent sample image, $TRGT_{33}$, from *Target* image and its frequency of each hash. We will calculate **JIWS**(*Source*, $TRGT_{33}$) in this section.

**JINF**(*Source*, *Target*) and **JINF**(*Source*, $TRGT_{33}$) are calculated in Table 6.4. Their

Table 6.1: Layout of Source and Target.

Source layout.

| Sector number | Hash |
|---|---|
| 1 | K |
| 2 | B |
| 3 | C |
| 4 | I |
| 5 | J |
| 6 | K |
| 7 | G |
| 8 | H |
| 9 | I |
| 10 | D |
| 11 | J |
| 12 | E |
| 13 | A |
| 14 | K |
| 15 | F |

Target layout.

| Sector number | Hash |
|---|---|
| 1 | A |
| 2 | Z |
| 3 | Q |
| 4 | A |
| 5 | B |
| 6 | C |
| 7 | D |
| 8 | E |
| 9 | F |
| 10 | G |
| 11 | Y |
| 12 | Q |
| 13 | Y |
| 14 | Z |
| 15 | Z |

Table 6.2: Hash count of *Source* and *Target*.

Hash count of *Source*.

| Hash | Count |
|---|---|
| **A** | 1 |
| **B** | 1 |
| **C** | 1 |
| **D** | 1 |
| **E** | 1 |
| **F** | 1 |
| **G** | 1 |
| *H* | 1 |
| *I* | 2 |
| *J* | 2 |
| *K* | 3 |

Hash count of *Target*.

| Hash | Count |
|---|---|
| **A** | 2 |
| **B** | 1 |
| **C** | 1 |
| **D** | 1 |
| **E** | 1 |
| **F** | 1 |
| **G** | 1 |
| *Q* | 2 |
| *Y* | 2 |
| *Z* | 3 |

values are 0.3043 and 0.1304 respectively. As I discussed in section 6.3, **JINF**(*Source*, $TRGT_{33}$) is proportionally smaller than **JINF**(*Source*, *Target*).

Table 6.3: $TRGT_{33}$ layout and hash count.

$TRGT_{33}$ layout.

| Offset of Target | Hash |
|---|---|
| 1 | **A** |
| 3 | *Q* |
| 6 | **C** |
| 9 | **F** |
| 12 | *Q* |
| 15 | *Z* |

Hash count of $TRGT_{33}$.

| Hash | Count |
|---|---|
| **A** | 1 |
| *Q* | 2 |
| **C** | 1 |
| **F** | 1 |
| *Z* | 1 |

Table 6.4: JINF(*Source*, *Target*) and JINF(*Source*, $TRGT_{33}$).

| *Source*, *Target* | | | *Source*, $TRGT_{33}$ | | |
|---|---|---|---|---|---|
| Normalized Hash Count | Int* | Union* | Normalized Hash Count | Int* | Union* |
| A: (1/15, 2/15) | 1/15 | 2/15 | A: (1/15, 1/6) | 1/15 | 1/6 |
| B: (1/15, 1/15) | 1/15 | 1/15 | B: (1/15, 0) | 0 | 1/15 |
| C: (1/15, 1/15) | 1/15 | 1/15 | C: (1/15, 1/6) | 1/15 | 1/6 |
| D: (1/15, 1/15) | 1/15 | 1/15 | D: (1/15, 0) | 0 | 1/15 |
| E: (1/15, 1/15) | 1/15 | 1/15 | E: (1/15, 0) | 0 | 1/15 |
| F: (1/15, 1/15) | 1/15 | 1/15 | F: (1/15, 1/6) | 1/15 | 1/6 |
| G: (1/15, 1/15) | 1/15 | 1/15 | G: (1/15, 0) | 0 | 1/15 |
| H: (1/15, 0/15) | 0 | 1/15 | H: (1/15, 0) | 0 | 1/15 |
| I: (2/15, 0/15) | 0 | 2/15 | I: (1/15, 0) | 0 | 1/15 |
| J: (2/15, 0/15) | 0 | 2/15 | J: (1/15, 0) | 0 | 1/15 |
| K: (3/15, 0/15) | 0 | 3/15 | K: (1/15, 0) | 0 | 1/15 |
| Q: (0, 2/15) | 0 | 2/15 | Q: (0, 2/6) | 0 | 2/6 |
| Y: (0, 2/15) | 0 | 2/15 | Y: (0, 0) | 0 | 0 |
| Z: (0, 3/15) | 0 | 3/15 | Z: (0, 1/6) | 0 | 1/6 |
| Sum | 7/15 | 23/15 | Sum | 3/15 | 23/15 |
| **JINF**(*Source*, *Target*) = | $\frac{(7/15)}{(23/15)}$ | = **0.3043** | **JINF**(*Source*, $TRGT_{33}$) = | $\frac{(3/15)}{(24/15)}$ | = **0.1304** |

Detailed calculation of **JINF**(*Source*, $TRGT_{33}$), which is in Table 6.4, is explained below.

$$Total(Drive) = total\ hash\ count\ of\ Drive$$

$$Union^{*}ofHash_i = max(\frac{Hash_i\ count\ of\ Source}{Total(Source)}, \frac{Hash_i\ count\ of\ TRGT_{33}}{Total(TRGT_{33})})$$

$$Intersect^{*}ofHash_i = min(\frac{Hash_i\ count\ of\ Source}{Total(Source)}, \frac{Hash_i\ count\ of\ TRGT_{33}}{Total(TRGT_{33})})$$

$$\textbf{JINF}(Source, TRGT_{33}) = \frac{Sum\ of\ all\ Intersect^{*}ofHash_i}{Sum\ of\ all\ Union^{*}\ of\ Hash_i} = \frac{\frac{3}{15}}{\frac{23}{15}} = 0.1304$$

The *CommonRatio* is calculated in the formula (6.3) for the common hashes between *Source* and $TRGT_{33}$. It does not consider the frequency of each hash unlike **JINF** calculation. The denominator count is selected from sample image, $TRGT_{33}$. Since *Source* and $TRGT_{33}$ share hashes **A**, **C** and **F**, the nominator value is 3 and the denominator values is 5 for total 5 unique hashes.

The calculation of **JIWS**(*Source*, $TRGT_{33}$) is shown in equation 6.4 by applying formula 6.1. We can observe that **JIWS**(*Source*, $TRGT_{33}$), 0.3913, is normalized when we compared the **JINF**(*Source*, $TRGT_{33}$)) value which is 0.1304. And this value is similar with **JINF**(*Source*, *Target*), which is 0.3043.

$$CommonRatio(Source, TRGT_{33}) = \frac{unique\ hash\ count\ of\ common}{total\ unique\ hash\ count\ of\ TRGT_{33}} = \frac{3}{5} = 0.6 \qquad (6.3)$$

$$\textbf{JIWS}(Source, TRGT_{33}) = 0.6 * [0.1304 + 0.6 * (1 - 0.1304)] = 0.3913 \qquad (6.4)$$

Table 6.5: Histogram analysis: low frequency hashes of c12-09.

| Freq | Count | Total | Freq | Count | Total | Freq | Count | Total |
|---|---|---|---|---|---|---|---|---|
| 1 | 5904230 | 5904230 | 2 | 1049561 | 2099122 | 3 | 428063 | 1284189 |
| 4 | 151555 | 606220 | 5 | 66062 | 330310 | 6 | 16567 | 99402 |
| 7 | 19434 | 136038 | 8 | 7540 | 60320 | 9 | 7988 | 71892 |
| 10 | 14555 | 145550 | 11 | 3094 | 34034 | 12 | 1533 | 18396 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| | | | | | | | | Total = 7708412 |

## 6.7 Observation of *CommonRatio*

Table 6.5 shows a histogram analysis of Charlie's December 9th image. As we can see for this image (c12-09), the ratio of unique hash counts (5904230) is 77 percent of the total hashes (7708412). Most images have similar statistics in histogram analysis.

As we can observe in the Table 6.8, the *CommonRatio* among samples are very similar and the unique hash count of sample images are 40 to 85 percent of the total hash count of the most sample images. Therefore, I arrange the *Source* and *Target* images in Table 6.1 based on these observed statistics.

## 6.8 Reasoning behind the JIWS formula

Though **JIWS** formula is heuristics, but it came from the observation that **JINF** changes proportionally to the size of sample image, and *commonRatio* remains near constant among the sample images from the same source. The *commonRatio* increases if the **JINF** value increases, which is expected because a high **JINF** means that both images are more similar. For normalization of Jaccard Index, (1-**JINF**) and **JINF** in the **JIWS** formula (6.1) work inversely. If we have a high **JINF** value, a small portion of *commonRatio* will be added with **JINF** and then this added value is multiplied to *commonRatio*. If **JINF** is low, then a high portion of *commonRatio* is added to **JINF**, and this sum is multiplied by *commonRatio*.

Since *Source* and *Target* share hashes **A**, **B**, **C**, **D**, **E**, **F** and **G**, the *CommonRatio* is

calculated.

$$CommonRatio(Source, Target) = \frac{unique\ hash\ count\ of\ common}{total\ unique\ hash\ count\ of\ Target} = \frac{7}{11} = 0.6364 \qquad (6.5)$$

$$\textbf{JIWS}(Source, Target) = 0.6364 * [0.3043 + 0.6364 * (1 - 0.3043)] = 0.4754$$

## 6.9 JINF is bounded by *CommonRatio*

In the **JINF** calculation, **JINF** is bounded by *CommonRatio*, meaning the value of **JINF** can be at most *CommonRatio*. I proved this to be lemma 1.

**Lemma 1.** ***JINF*** *cannot be greater than CommonRatio*

*Proof.* I will prove this with the example as shown in Table 6.6. Both images, S and T, have common hashes, c1, c2, ... and cn. Non-common hashes of image S are s1, s2, ... and sr. And t1, t2 .. tr represent the non-common hashes of image T respectively. For better understanding, the template for calculating the **JINF** of these two images is shown in Table 6.7. Let Cn represent the total count of common sectors, while Sn and Tn represent the total hash count of image S and image T. The total hash count of T is equal or less than that of image S. For simplicity, let's assume that there is only one occurrence for each unique hash value. I will relax this assumption later.

$$Sum\ of\ Intersect* \ = \ (0 + \frac{Cn}{Sn} + 0)$$

$$Sum\ of\ Union* \ = \ (\frac{Sr}{Sn} + \frac{Cn}{Tn} + \frac{Tr}{Tn}).$$

$$\textbf{JINF}(S,T) = \frac{\frac{Cn}{Sn}}{\frac{Sr}{Sn} + \frac{Cn+Tr}{Tn}} = \frac{\frac{Cn}{Sn}}{\frac{Sr}{Sn} + \frac{Tn}{Tn}} = \frac{\frac{Cn}{Sn}}{\frac{Sr}{Sn} + 1} = \frac{\frac{Cn}{Sn}}{\frac{Sr+Sn}{Sn}} = \frac{Cn}{Sr+Sn}$$

$$CommonRatio(S,T) = \frac{Cn}{Tn}$$

$$Since\ Tn \leq Sn,\ Tn \leq (Sr + Sn).\ \ Therefore,\ \frac{Cn}{Tn} \geq \frac{Cn}{Sr+Sn}$$

**JINF** equals *commonRatio* only when Sr is 0. If I relax the assumption that each hash has only one occurrence, the nominator part of **JINF** becomes smaller. Hence the proof is still valid.

Table 6.6: Two Images to Compare.

| S | | T |
|---|---|---|
| s1 | | c1 |
| s2 | | c2 |
| ... | | c3 |
| sr | | ... |
| c1 | | ... |
| c2 | | cn |
| ... | | t1 |
| ... | | t2 |
| cn | | ... |
| | | tr |

Table 6.7: Template for **JINF**(S, T) calculation.

| S | T | Normalized (Min, Max) |
|---|---|---|
| Sr | | (0, Sr/Sn) |
| Cn | Cn | (Cn/Sn, Cn/Tn) |
| | Tr | (0, Tr/Tn) |

□

## 6.10 Further analysis of JIWS

The similarity value of **JINF**(*Source*, *Target*) and **JIWS**(*Source*, *Target*) is 0.3043 and 0.4754, respectively. However, the **JIWS** (*Source*, $TRGT_{33}$) value of 0.3913 is a better result than the **JINF** (*Source*, $TRGT_{33}$) value of 0.1304 for a sample image, $TRGT_{33}$, because the the gap ratio of **JIWS** is less than the one of **JINF**. My method explains this well when I measure the similarity between the source and the sample image from the source itself. The ideal similarity measuring method should return 1 as a perfect match. My **JINF** method returns 1 when two same images are compared. Let's say $SRC_{10}$ is the 10 percent sample image of the source, *Source*. The subscript value 10 in $SRC_{10}$ represents the sample ratio. The **JINF**(*Source*, $SRC_{10}$) value will be around 0.1.

However, the common ratio between $SRC_{10}$ and *Source* should be 1.0 since all the sectors of $SRC_{10}$ are from *Source*. Since *CommonRatio* is 1.0, **JIWS** becomes 1.0 *[jinf + 1.0 * (1-jinf) ] = 1.0. **JIWS** values for $SRC_5$, $SRC_{10}$, $SRC_{20}$, $SRC_{30}$, $SRC_{40}$, $SRC_{50}$ and $SRC_{75}$ with *Source* are all 1.0.

## 6.11 Result and validation

Table 6.8 shows **JINF**, *CommonRatio* (cr), and **JIWS** values between a December 3rd whole image without OS sectors of Charlie and sample images from a couple daily images of Charlie.

Figure 6.1 shows the **JIWS** between daily images of Charlie and Charlie's 7 samples of the December 3rd image. Figure 6.2 shows the enlarged peak view of Figure 6.1. Each dot under the top solid line represents **JIWS** values of sampled images. The highest solid line represents the **JIWS** between daily images of Charlie and December 3rd whole image without OS sectors of Charlie. The middle connected dotted line shows the **JINF** between daily images of Charlie and December 3rd whole image without OS sectors of Charlie. The bottom solid line shows the **JIWS** between daily images of Charlie and December 3rd whole image without OS sectors of Jo. Sampled images of Charlie reveal the higher value

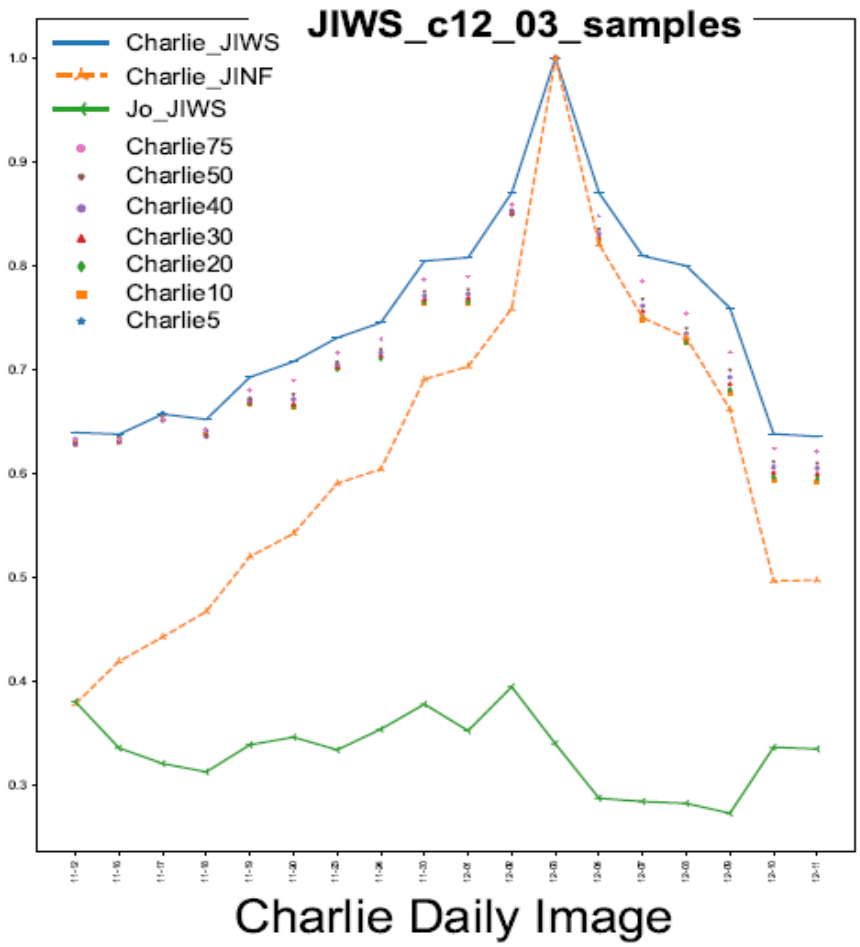


Figure 6.1: **JIWS** and **JINF** between 7 samples of Charlie 12-03 and daily images of Charlie.

Table 6.8: **JIWS** C12-03 with daily sampled images.

| Day | | 0.05 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.75 | 1 |
|---|---|---|---|---|---|---|---|---|---|
| C11-12 | jinf | 0.0498 | 0.0777 | 0.1263 | 0.1685 | 0.2052 | 0.2382 | 0.3103 | 0.3785 |
| | cr | 0.7907 | 0.7857 | 0.7790 | 0.7739 | 0.7697 | 0.7661 | 0.7593 | 0.7546 |
| | jiws | 0.6334 | 0.6303 | 0.6286 | 0.6284 | 0.6289 | 0.6296 | 0.6332 | 0.63951 |
| C11-23 | jinf | 0.0719 | 0.1152 | 0.1869 | 0.2482 | 0.2997 | 0.3503 | 0.4637 | 0.5907 |
| | cr | 0.8317 | 0.8276 | 0.8205 | 0.8159 | 0.8118 | 0.8082 | 0.8017 | 0.7970 |
| | jiws | 0.7019 | 0.7014 | 0.7008 | 0.7029 | 0.7049 | 0.7075 | 0.7164 | 0.7308 |
| C12-02 | jinf | 0.0841 | 0.1321 | 0.2152 | 0.2871 | 0.3527 | 0.4168 | 0.5697 | 0.7588 |
| | cr | 0.9195 | 0.9166 | 0.9125 | 0.9094 | 0.9068 | 0.9043 | 0.8985 | 0.8934 |
| | jiws | 0.8518 | 0.8502 | 0.8498 | 0.8507 | 0.8521 | 0.8539 | 0.8593 | 0.8704 |
| C12-03 | jinf | 0.0922 | 0.1473 | 0.2443 | 0.3315 | 0.4110 | 0.4931 | 0.6899 | 1 |
| | cr | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | jiws | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| C12-10 | jinf | 0.0738 | 0.1120 | 0.1748 | 0.2291 | 0.2740 | 0.3151 | 0.4040 | 0.4967 |
| | cr | 0.764 | 0.7570 | 0.7511 | 0.7470 | 0.7444 | 0.7424 | 0.7386 | 0.7360 |
| | jiws | 0.5970 | 0.5932 | 0.5968 | 0.6013 | 0.6062 | 0.6114 | 0.6236 | 0.6383 |

for **JIWS** than for the whole image of Jo of the same day.

Figure 6.3, 6.4, 6.5, 6.6, 6.7 and 6.8 show the **JIWS** between daily images of Charlie and two sample images of Nov-11, Nov-18, Nov-24, Dec-03, Dec-08 and Dec-11 of Charlie. As a guide, the top solid line of every graph shows the **JIWS** between daily images of Charlie and non-sampled images of the listed days of Charlie. The *jiws_C11_12_05* line in the legend represents the result of **JIWS** with Charlie(C) 5% image on November 12th. The bottom solid line shows the **JIWS** between daily images of Charlie and non-sampled image from Dec 3rd image of Jo, which I choose as a known non-related image.

## 6.12 Conclusion

As we can see, most **JIWS** values with only a 5-percent sampled image of the *Target* image follow the trajectory of **JIWS**(*Source*, *Target*). They distinguish with not-related images and perform better than **JINF**. The sample images of Charlie on Nov-12, which is the first day the M57 experiment, shows low performance. **JIWS** values between the sampled images from the first day of Charlie and images after two weeks of use from Charlie, and **JIWS** values between Dec 3rd image of Jo and images after two weeks of use from Charlie

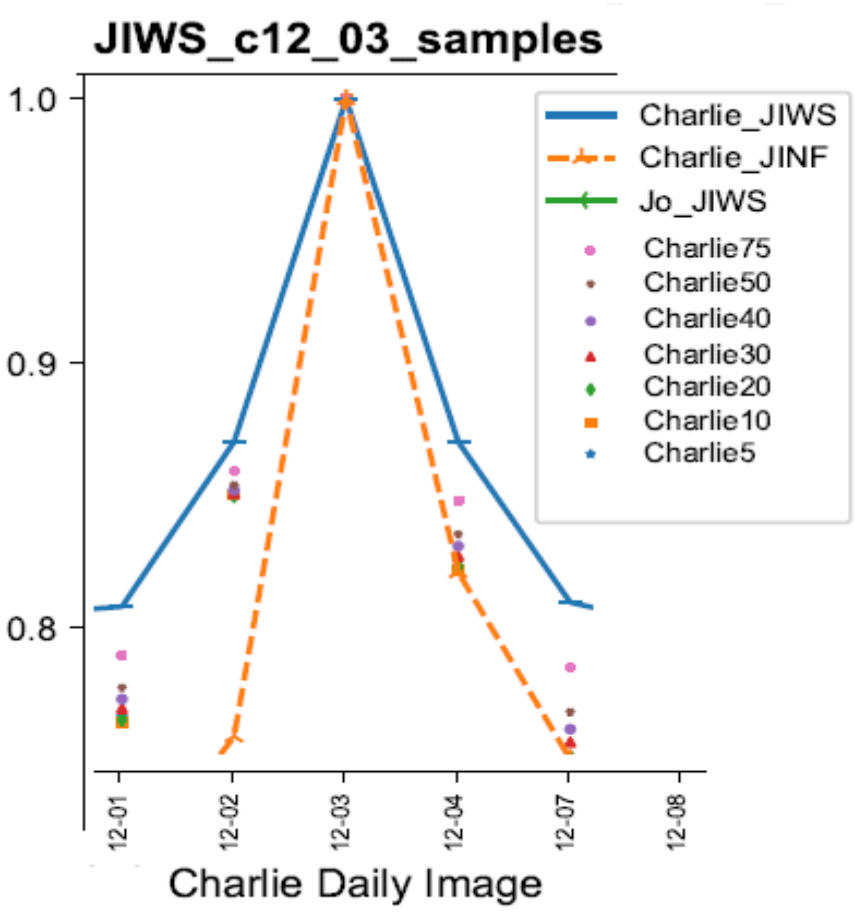


Figure 6.2: Enlarged peak view of Figure 6.1.

are less distinguishable. I believe that this is because the sampled images of the first day do not have much activity to distinguish themselves from other images.

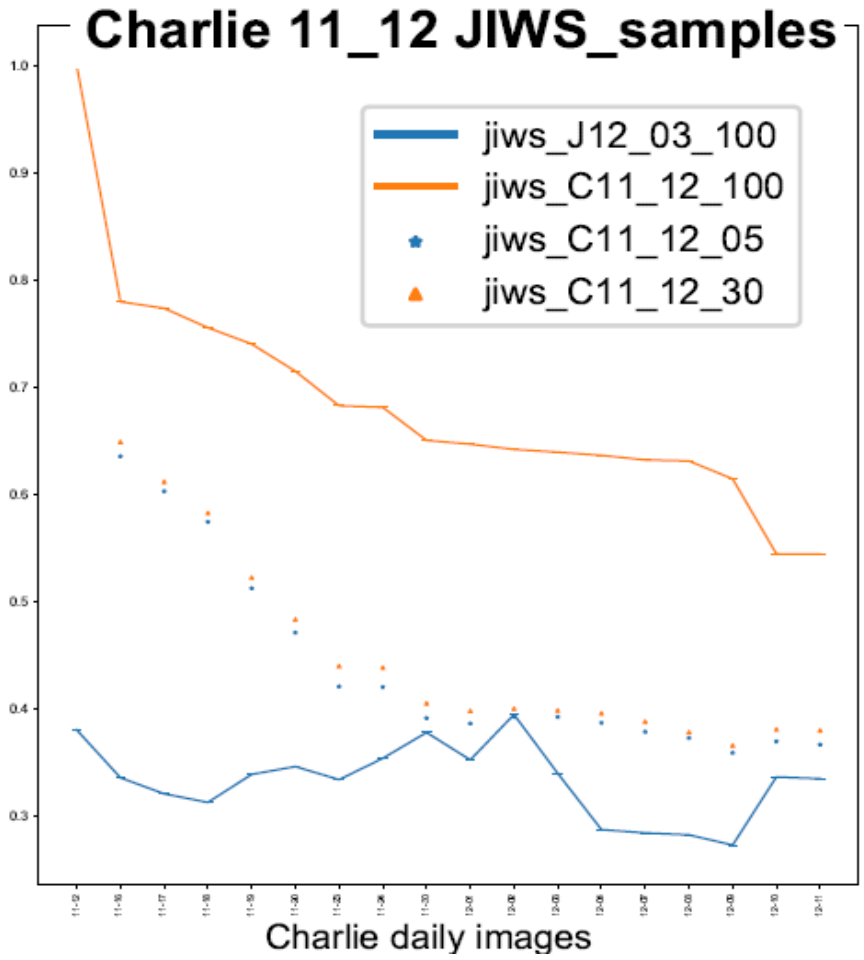


Figure 6.3: **JIWS** between 2 samples of Charlie 11-12 and daily images of Charlie.

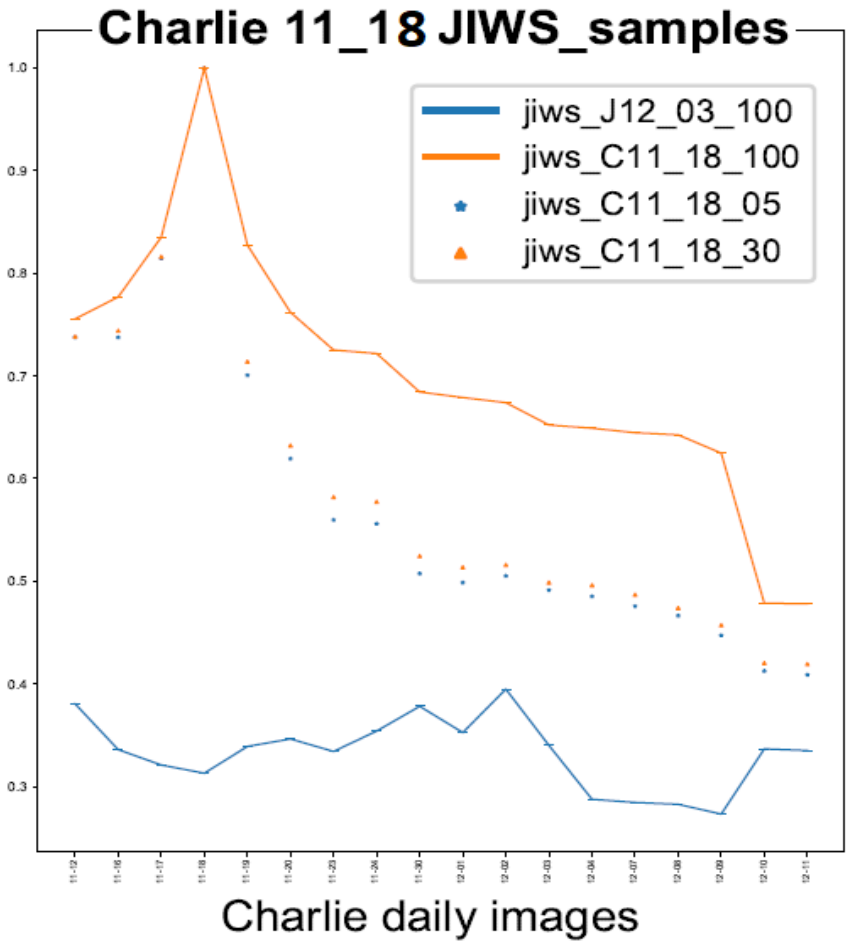


Figure 6.4: **JIWS** between 2 samples of Charlie 11-24 and daily images of Charlie.

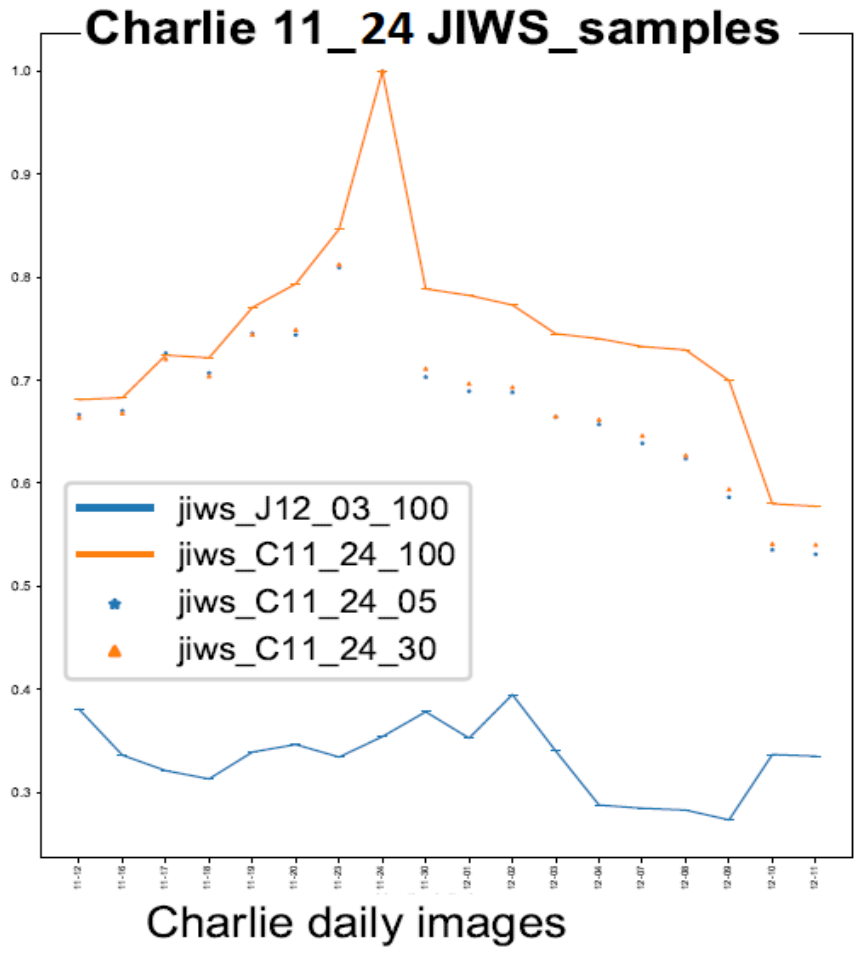


Figure 6.5: **JIWS** between 2 samples of Charlie 11-24 and daily images of Charlie.

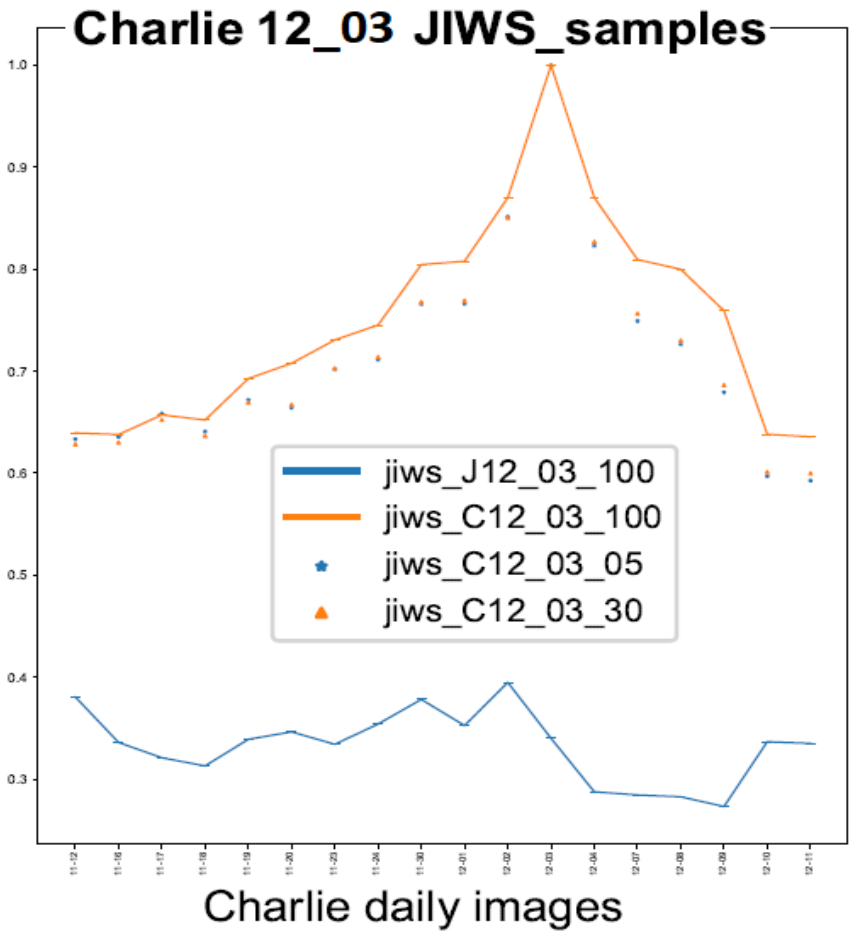


Figure 6.6: **JIWS** between 2 samples of Charlie 12-03 and daily images of Charlie.

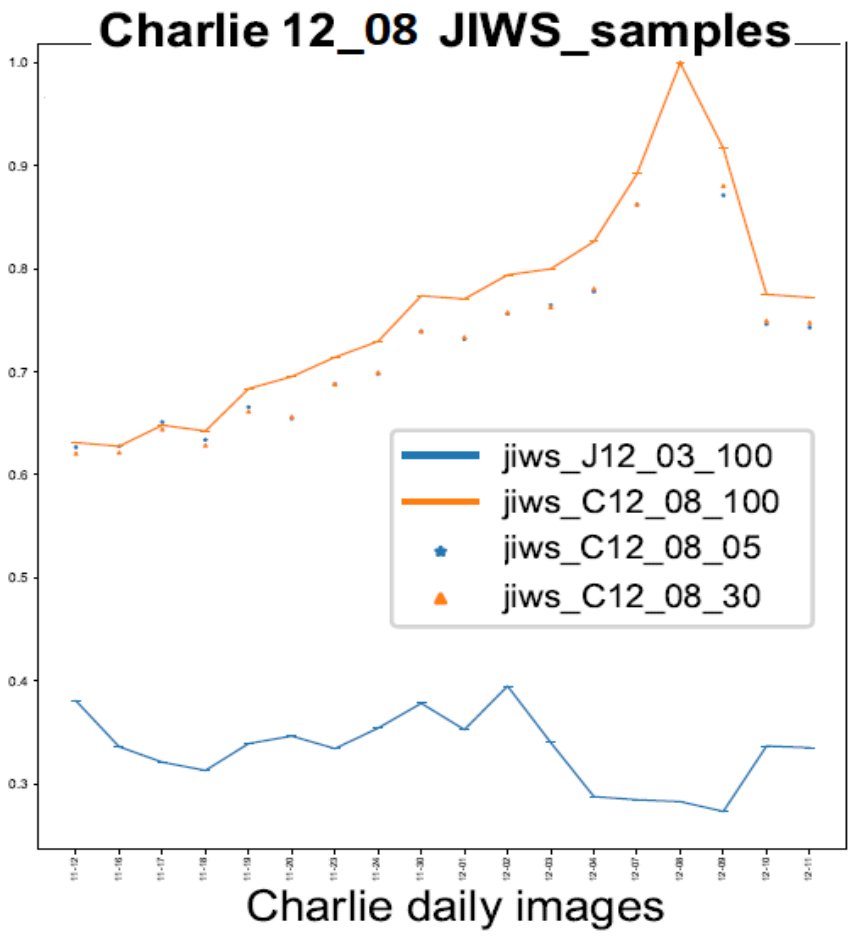


Figure 6.7: **JIWS** between 2 samples of Charlie 12-08 and daily images of Charlie.

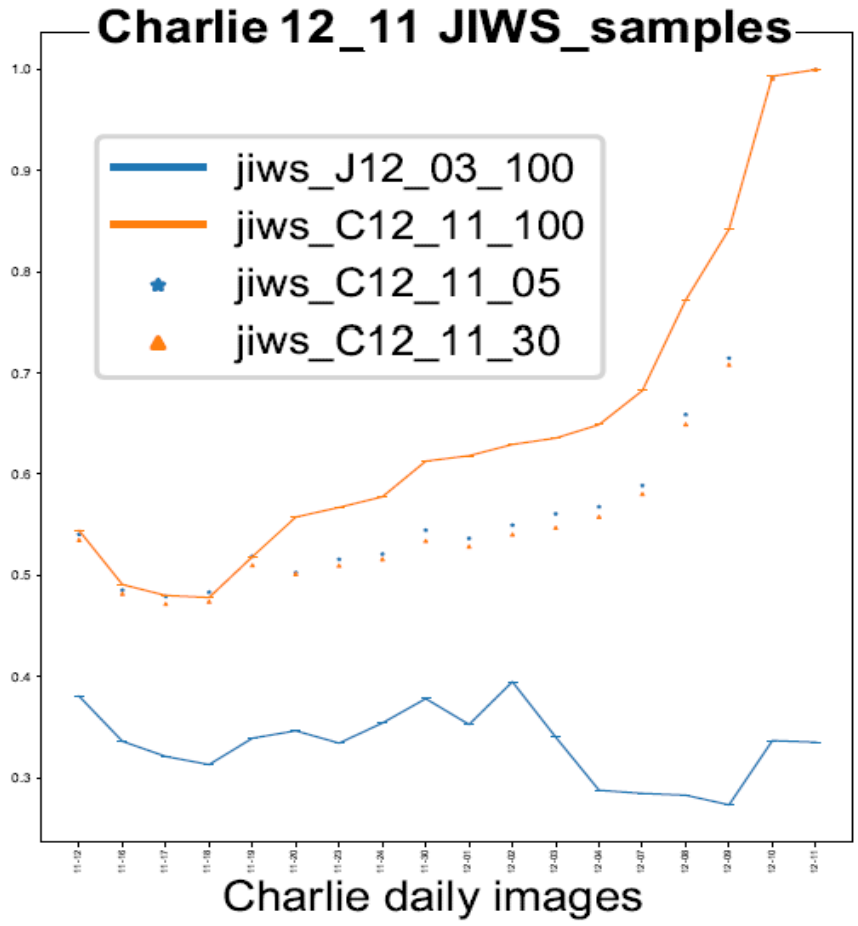


Figure 6.8: **JIWS** between 2 samples of Charlie 12-11 and daily images of Charlie.

# Chapter 7: Parallel Computing Experiments

In this chapter, I introduce the ARGO parallel processing platform at George Mason University, and show how much I can gain in performance by using ARGO.

## 7.1 ARGO overview

The ARGO Cluster is a batch computing resource available to all faculty and their students at George Mason University. It was assembled by Dell technicians during the week of March 11th, 2013 and consists of the following hardware:

- 2 Login nodes
  - Dell PowerEdge R720, with Intel Xeon E5-2670
  - 16 core CPUs, 1 TB hard drive, 64 GB RAM
- 2 "Fat" nodes
  - 2 Dell PowerEdge R815, with quad AMD Opteron 6276
  - 64 core CPUs, 1 TB hard drive, 512 GB RAM
- 71 Dell Compute Nodes
  - Dual Intel Xeon CPUs
  - 16 to 28 cores, 64 GB to 1.5 TB RAM
- 10 GPU nodes
  - 4 nodes with NVIDIA K80 GPUs (nodes 40,50,55 and 56).
    - 12 GB onboard memory per GPU

    - 24 core CPUs
    - 128 to 512 GB RAM per node
  - 6 nodes with NVIDIA V100 GPUs (nodes 76-81).
    - 32 GB onboard memory per GPU
    - 2 nodes have NVLink for fast GPU-to-GPU communication (nodes 77 and 79)
    - 28 core CPUs
    - 750 GB to 1.5 TB RAM per node
- Dell NSS PowerEdge R620 with 150 TB storage space.
- Interconnect networks are InfiniBand (for message passing), Gigabit Ethernet (for I/O).
- 20 Gbit (bonded) external connections for each head node.

## 7.2 Simple linux utility for resource management

The Simple Linux Utility for Resource Management (*Slurm*) is an open source, fault-tolerant, and highly scalable cluster management and job scheduling system for large and small Linux clusters. It has similarities with the *Sun Grid Engine Scheduler* and the *Univa Grid Engine Scheduler*, which were used previously on *ARGO*. It also shares similarities with the open source resource manager known as *TORQUE*. The *Slurm* batch-queueing system provides the mechanism by which all jobs are submitted to the *ARGO* Cluster and are scheduled to run on the compute nodes. If we want to get the Jaccard index between 19 daily images of Charlie with 21 daily images of Pat, we assign 19 nodes in the *Slurm* batch file and the same python code is executed at each node.

In reality, a job is killed by timeout or when it is short of memory. Instead of being killed, it can be delayed by priority.

After all the jobs at every node are finished with or without success, the *Slurm* batch job is executed again and it picks the leftover comparison at each node and continues until it is completed or killed. We can repeat this steps until all the jobs are completed successfully.

## 7.3 Performance using ARGO

Each employee of M57 has an average of 20 daily images. For each day, 7 sampled images are created. The total sample images of 4 employees are close to 600 and these sample images are to be compared to 80 daily images, which requires 48000 calculations. In a single computer environment, these calculations will take forever. Therefore, I used the Slurm batch command in the ARGO parallel processing platform.

When a job starts at each node in ARGO, it recognizes its own node id and chooses the corresponding day among Charlie's 19 daily images. The selected daily image of Charlie will be used a source image and the target list is built from Pat's daily images. For example, if node 10 chooses the Dec 1st image of Charlie as the source, $JINF_1(C12-01, P11-12), JINF_2(C12-01, P11-16), JINF_3(C12-01, P11-17)...$ *and* $JINF_{21}(C12-01, P12-10)$ are calculated at node 10.

At every 19 nodes, 21 **JINF** are calculated. It takes an average of 20 minutes to calculate the **JINF** between whole images. To finish a total of 399 **JINF** calculations, it takes around 7 hours in the *ARGO* platform. Otherwise, it will take 133 hours in a single machine.

# Chapter 8: Jaccard Index with Split (JISPLIT)

In this chapter, I propose, develop, and test a method to parallelize the computation of an existing digital media similarity measure called Jaccard Index with Normalized Frequency (**JINF**) [57] and reduce the time to obtain a similarity measure. The proposed method partitions the digital media under examination and distributes the computation across multiple processors or systems, then combines the results into an overall similarity measure which preserves the accuracy of the original method (**JINF**) when run on a single processor.

## 8.1 Overview of JISPLIT

To speed up computation as much as possible, I tried to split the target files into a number of smaller files and tried to get the similarity measure between *source* images and small target files. The proposed parallelization method splits the digital media of interest into $N$ equal partitions. Each partition is then distributed to a separate processor for computation of the necessary precursor values for the **JINF** similarity measure.

This precursor operation is the most computationally intensive part of the **JINF** method, as it requires operations over each sector of the media being compared. These precursor values are then passed back to the central processing node, which aggregates the results and computes the final similarity measure.

## 8.2 Parallel processing environment of JISPLIT

In order to distribute computing process to other cores on a laptop, the Pool class in the Python multiprocessing library is normally used. Although this approach also works on a cluster, it does not utilize the available computing power fully. For ARGO, up to 64 cores

Table 8.1: Layout of the split files: *T1* and *T2*.

| T1 | | T2 | |
|---|---|---|---|
| 1 | A | 1 | E |
| 2 | Z | 2 | F |
| 3 | Q | 3 | G |
| 4 | A | 4 | Y |
| 5 | B | 5 | Q |
| 6 | C | 6 | Y |
| 7 | D | 7 | Z |
| | | 8 | Z |

are assigned to a job. I used a pool-like class, *mpi4py* library, which is very similar to the one in the multiprocessing library. I set up an environment by using mpi4py and took advantage of a larger number of cores. A Python Virtual Environment was created for the job and Python modules are installed in the virtual environment. The MPIPoolExecutor.map() function actually handles all the complexity of coordinating communications with workers, farming out the different tasks, and then collecting the results.

## 8.3 Intuitive heuristics and its analysis

Let's say *Target* is split into 10 small files, namely $TRGT_{sp_1}$,$TRGT_{sp_2}$,$TRGT_{sp_3}$, ... $TRGT_{sp_i}$ ... and $TRGT_{sp_{10}}$. Each $TRGT_{sp_i}$ can be regarded as an individual image for the calculation. Initially, I divide the target media to roughly equal size and distribute the calculation of **JINF**(*Source*, $TRGT_{sp_i}$) to each core of *ARGO*. I expected that sum of all **JINF**(Source, $TRGT_{sp_i}$) values matches **JINF**(*Source*,*Target*) value.

Preliminary experiments established that the sum result of the **JINF** values from the split files does not match the **JINF**(Source, Target), which is 0.3043, as shown in Table 6.4.

I analyzed the reason. I will explain the reason using the same *Source* and *Target* in Table 6.1 as an example. I split *Target* into two split target files *T1* and *T2* as in Table 8.1. Hash frequencies of the two split files are shown in Table 8.2.

Table 8.2: Hash frequency of *T1* and *T2*.

| T1 | | T2 | |
|---|---|---|---|
| Hash | Frequency | Hash | Frequency |
| A | 2 | E | 1 |
| B | 1 | F | 1 |
| C | 1 | G | 1 |
| D | 1 | Q | 1 |
| Q | 1 | Y | 2 |
| Z | 1 | Z | 2 |

Now I will calculate **JINF**(*Source*, *T1*) and **JINF**(*Source*, *T2*), as shown in Table 8.3 and Table 8.4

The total sum of these two **JINF** values will be 0.22222 + 0.15 = 0.37222. But the **JINF**(*Source*, *Target*) is 0.3043, as shown in Table 6.4. This discrepancy happens because Hash Q and Hash Z are split into two different split files. When a hash is split into more than one split target file, the *Union** value of that hash cannot have the same *Maximum* value that it does in the non-split JINF calculations. If I sum *Union** from all pairs of (*Source*, $TRGT_{sp_i}$), the *Union** value of the split hash is always less than the maximum *Union** value between (*Source*, *Target*), unless I split the file intelligently. What I want is to have the same maximum value (23/15) as in Table 6.4 even after I sum *Union** from all pairs of (*Source*, $TRGT_{sp_i}$).

However, if I sum the *Int** values from all pairs of (*Source*, $TRGT_{sp_i}$), the result will be the same as the sum of *Int** between *Source* and *Target*.

## 8.4 Solution for intuitive heuristics

To handle this issue, I have to do pre-processing and post-processing before and after using the ARGO platform.

For the pre-processing step, I apply the *panda groupby* operation on the hash field, which sorts the hash. This makes sure that when a split is done, the same hash has to be put in

Table 8.3: **JINF**(*Source*, *T1*).

| Normalized Hash Count (*Source*, *T1*) | Int* | Union* |
|---|---|---|
| A: (1/15, 2/15) | 1/15 | 2/15 |
| B: (1/15, 1/15) | 1/15 | 1/15 |
| C: (1/15, 1/15) | 1/15 | 1/15 |
| D: (1/15, 1/15) | 1/15 | 1/15 |
| E: (1/15, 0) | 0 | 1/15 |
| F: (1/15, 0) | 0 | 1/15 |
| G: (1/15, 0) | 0 | 1/15 |
| H: (1/15, 0) | 0 | 1/15 |
| I: (2/15, 0) | 0 | 2/15 |
| J: (2/15, 0) | 0 | 2/15 |
| K: (3/15, 0) | 0 | 3/15 |
| Q: (0, 1/15) | 0 | **1/15** |
| Y: (0, 0) | 0 | 0 |
| Z: (0, 1/15) | 0 | **1/15** |
| Sum | 4/15 | 18/15 |
| **JINF**(*Source*, *T1*) = | $\frac{(4/15)}{(18/15)}$ | = **0.2222** |

Table 8.4: **JINF**(*Source*, *T2*).

| Normalized Hash Count (*Source*, *T2*) | Int* | Union* |
|---|---|---|
| A: (1/15, 0) | 0 | 1/15 |
| B: (1/15, 0) | 0 | 1/15 |
| C: (1/15, 0) | 0 | 1/15 |
| D: (1/15, 0) | 0 | 1/15 |
| E: (1/15, 1/15) | 1/15 | 1/15 |
| F: (1/15, 1/15) | 1/15 | 1/15 |
| G: (1/15, 1/15) | 1/15 | 1/15 |
| H: (1/15, 0) | 0 | 1/15 |
| I: (2/15, 0) | 0 | 2/15 |
| J: (2/15, 0) | 0 | 2/15 |
| K: (3/15, 0) | 0 | 3/15 |
| Q: (0, 1/15) | 0 | **1/15** |
| Y: (0, 2/15) | 0 | 2/15 |
| Z: (0, 2/15) | 0 | **2/15** |
| Sum | 4/15 | 18/15 |
| **JINF**(*Source*, *T2*) = | $\frac{(3/15)}{(20/15)}$ | = **0.15** |

the same file. In the previous example, hash values Q and Z had to be either *T1* or *T2*, but not in both. Once the ARGO platform finishes the parallel processing for each pair of (source, $trgt_{split_i}$), each ARGO node will return a 2-tuple:

$$(value_i, \{(h1, NF(h1)), (h2, NF(h2)), ..., (hk, NF(hk))\}_i)$$

where NF(h) is the normalized frequency of a hash, $h$, and k is the total number of unique hashes assigned to each ARGO node i.

The first element in the tuple is the *sum of Int** and the second element is the *Union** column, which is the last column as shown in Table 8.3 and 8.4.

One ARGO working node will be assigned for each pair of **JINF**(*Source*, $TRGT_{sp_i}$) calculation. Note that each working node does not complete the normal JINF calculation. Instead, they return an intermediate result as a 2-tuple.

$$\textbf{JISPLIT}(Source, Target) = \frac{final\ sum\ of\ Int*}{final\ sum\ of\ Union*} \tag{8.1}$$

$$final\ sum\ of\ Int* = \sum_{i=1}^{N} sum\ of\ Int^*\ from\ each\ ARGO\ node_i. \tag{8.2}$$

$$final\ sum\ of\ Union* = \sum_{j=1}^{H} Union*_j\ \ row\ from\ \textbf{Final Union* column} \tag{8.3}$$

Where N is the number of split files and H is the number of unique hashes.

I add up all *the sum of Int** and use this value as the numerator for the final **JISPLIT**(*Source*, *Target*) calculation, which is straightforward.

For the denominator of the final **JINF** calculation, I construct the **Final Union* column** using all of the *Union** columns returned from each ARGO node.

The normalized frequency of each hash value, $h$, of the **Final Union* column**, is the

Table 8.5: Hash frequency and split files of Target.

| Hash | Freq | Split file |
|---|---|---|
| A | 2 | $T_a$ |
| B | 1 | $T_a$ |
| C | 1 | $T_a$ |
| D | 1 | $T_a$ |
| E | 1 | $T_a$ |
| F | 1 | $T_a$ |
| G | 1 | $T_b$ |
| Q | 2 | $T_b$ |
| Y | 2 | $T_b$ |
| Z | 3 | $T_b$ |

**Max**$(NF(h)_1, NF(h)_2, ..., NF(h)_H)$ where $NF(h)_i$ is the normalized frequency of a hash, $h$, from ARGO node i.

I sum up all the rows of the **Final Union* column** and use it as the denominator of the final **JISPLIT**(*Source*, *Target*).

## 8.5 Example of JISPLIT

We can take a look at the example for better understanding. I split the target into two files: $T_a$ and $T_b$ after applying the *groupby* operation on the hash field of the Target. Table 8.5 shows the hash frequency and which hash belongs to which split file.

Let's assume the working $node_a$ of ARGO calculates **JINF**(*Source*, $T_a$) and the working $node_b$ calculates **JINF**(*Source*, $T_b$).

$Node_a$ will return 6/15 as the *sum of Int** and the *Union** column, which is the last column of Table 8.6. $Node_b$ will return 1/15 as the *sum of Int** and the *Union** column which is the last column of Table 8.7. The returned 2-tuple from $Node_a$ will be (6/15, {(A, 2/15), (B, 1/15), (C, 1/15), ... (K, 3/15))}). As a post processing, the nominator will be the sum of (6/15, 1/15) which is 7/15. The **Final Union* column** is constructed as shown in Table 8.8 by selecting the $Max$ of two columns returned by each working node.

The denominator value is the sum of the **Final Union* column**, which is 23/15.

Table 8.6: **JINF**(*Source*, $T_a$).

| Normalized Hash Count (*Source*, $T_a$) | Int* | Union* |
|---|---|---|
| A: (1/15, 2/15) | 1/15 | **2/15** |
| B: (1/15, 1/15) | 1/15 | **1/15** |
| C: (1/15, 1/15) | 1/15 | **1/15** |
| D: (1/15, 1/15) | 1/15 | **1/15** |
| E: (1/15, 1/15) | 1/15 | **1/15** |
| F: (1/15, 1/15) | 1/15 | **1/15** |
| G: (1/15, 0) | 0 | **1/15** |
| H: (1/15, 0) | 0 | **1/15** |
| I: (2/15, 0) | 0 | **2/15** |
| J: (2/15, 0) | 0 | **2/15** |
| K: (3/15, 0) | 0 | **3/15** |
| Q: (0, 0) | 0 | **0** |
| Y: (0, 0) | 0 | **0** |
| Z: (0, 0) | 0 | **0** |
| Sum | **6/15** | |

Table 8.7: **JINF**(*Source*, $T_b$).

| Normalized Hash Count (*Source*, $T_b$) | Int* | Union* |
|---|---|---|
| A: (1/15, 0) | 0 | **1/15** |
| B: (1/15, 0) | 0 | **1/15** |
| C: (1/15, 0) | 0 | **1/15** |
| D: (1/15, 0) | 0 | **1/15** |
| E: (1/15, 0) | 0 | **1/15** |
| F: (1/15, 0) | 0 | **1/15** |
| G: (1/15, 1/15) | 1/15 | **1/15** |
| H: (1/15, 0) | 0 | **1/15** |
| I: (2/15, 0) | 0 | **2/15** |
| J: (2/15, 0) | 0 | **2/15** |
| K: (3/15, 0) | 0 | **3/15** |
| Q: (0, 2/15) | 0 | **2/15** |
| Y: (0, 2/15) | 0 | **2/15** |
| Z: (0, 3/15) | 0 | **3/15** |
| Sum | **1/15** | |

Table 8.8: Final Union* Column.

| Hash | Union* of $T_a$ | Union* of $T_b$ | **Final Union* column** |
|---|---|---|---|
| A | **2/15** | **1/15** | Max(2/15, 1/15) = **2/15** |
| B | 1/15 | 1/15 | 1/15 |
| C | 1/15 | 1/15 | 1/15 |
| D | 1/15 | 1/15 | 1/15 |
| E | 1/15 | 1/15 | 1/15 |
| F | 1/15 | 1/15 | 1/15 |
| G | 1/15 | 1/15 | 1/15 |
| H | 1/15 | 1/15 | 1/15 |
| I | 2/15 | 2/15 | 2/15 |
| J | 2/15 | 2/15 | 2/15 |
| K | 3/15 | 3/15 | 3/15 |
| Q | 0 | 2/15 | 2/15 |
| Y | 0 | 2/15 | 2/15 |
| Z | 0 | 3/15 | 3/15 |
| Sum | 1/15 | 22/15 | **23/15** |

**JISPLIT** will be $\frac{7/15}{23/15} = 0.3043$ which matches the **JINF**(*Source*, *Target*) value as shown in Table 6.4.

## 8.6 Splitting both source and target

So far I have split the target file only. I can also split the source file. Table 8.9 also shows how I split *Source*. I apply the same pre-processing step as I explained previously.

I need four working nodes of ARGO since there are 4 pairs of split file combination: ($S_a$, $T_a$), ($S_a$, $T_b$), ($S_b$, $T_a$) and ($S_b$, $T_b$). Table 8.10, 8.11, 8.12 and 8.13 show the **JINF** calculation for each combination.

Table 8.14 shows the return value of *Int** from each working node. The sum 7/15 will be used as a nominator of **JISPLIT**(*Source*, *Target*) calculation.

Table 8.15 shows the return *Union** column from each working node. The sum 23/15 will be used as a denominator of **JISPLIT**(*Source*, *Target*) calculation. The final **JISPLIT** result, found by splitting both *Source* and *Target*, matches the **JINF**(*Source*, *Target*) value in Table **??** as well.

Table 8.9: Hash frequency and split files of *Source*.

| Hash | Frequency | split file |
|---|---|---|
| A | 1 | $S_a$ |
| B | 1 | $S_a$ |
| C | 1 | $S_a$ |
| D | 1 | $S_a$ |
| E | 1 | $S_a$ |
| F | 1 | $S_b$ |
| G | 1 | $S_b$ |
| H | 1 | $S_b$ |
| I | 2 | $S_b$ |
| J | 2 | $S_b$ |
| K | 3 | $S_b$ |

Table 8.10: JINF($S_a$, $T_a$).

| Normalized hash frequency ($S_a$, $T_a$) | Int* | Union* |
|---|---|---|
| A: (1/15, 2/15) | 1/15 | **2/15** |
| B: (1/15, 1/15) | 1/15 | **1/15** |
| C: (1/15, 1/15) | 1/15 | **1/15** |
| D: (1/15, 1/15) | 1/15 | **1/15** |
| E: (1/15, 1/15) | 1/15 | **1/15** |
| F: (0, 1/15) | 0 | **1/15** |
| Sum | **5/15** | |

Table 8.11: **JINF**($S_b$, $T_a$).

| Normalized hash frequency ($S_b$, $T_a$) | Int* | Union* |
|---|---|---|
| A: (0, 2/15) | 0 | **2/15** |
| B: (0, 1/15) | 0 | **1/15** |
| C: (0, 1/15) | 0 | **1/15** |
| D: (0, 1/15) | 0 | **1/15** |
| E: (0, 1/15) | 0 | **1/15** |
| F: (1/15, 1/15) | 1/15 | **1/15** |
| G: (1/15, 0) | 0 | **1/15** |
| H: (1/15, 0) | 0 | **1/15** |
| I: (2/15, 0) | 0 | **2/15** |
| J: (2/15, 0) | 0 | **2/15** |
| K: (3/15, 0) | 0 | **3/15** |
| Sum | **1/15** | |

Table 8.12: **JINF**($S_a$, $T_b$).

| Normalized hash frequency ($S_a$, $T_b$) | Int* | Union* |
|---|---|---|
| A: (1/15, 0) | 0 | **1/15** |
| B: (1/15, 0) | 0 | **1/15** |
| C: (1/15, 0) | 0 | **1/15** |
| D: (1/15, 0) | 0 | **1/15** |
| E: (1/15, 0) | 0 | **1/15** |
| G: (0, 1/15) | 0 | **1/15** |
| Q: (0, 2/15) | 0 | **2/15** |
| Y: (0, 2/15) | 0 | **2/15** |
| Z: (0, 3/15) | 0 | **3/15** |
| Sum | **0** | |

Table 8.13: **JINF**($S_b$, $T_b$).

| Normalized hash frequency ($S_b$, $T_b$) | Int* | Union* |
|---|---|---|
| F: (1/15, 0) | 0 | **1/15** |
| G: (1/15, 1/15) | 1/15 | **1/15** |
| H: (1/15, 0) | 0 | **1/15** |
| I: (2/15, 0) | 0 | **2/15** |
| J: (2/15, 0) | 0 | **2/15** |
| K: (3/15, 0) | 0 | **3/15** |
| Q: (0, 2/15) | 0 | **2/15** |
| Y: (0, 2/15) | 0 | **2/15** |
| Z: (0, 3/15) | 0 | **3/15** |
| Sum | **1/15** | |

Table 8.14: Final Int* from 4 working ARGO nodes.

| Pair of split files | Int* |
|---|---|
| ($S_a$, $T_a$) | 5/15 |
| ($S_b$, $T_a$) | 1/15 |
| ($S_a$, $T_b$) | 0/15 |
| ($S_b$, $T_b$) | 1/15 |
| Sum | **7/15** |

Table 8.15: Final Union* Column from 4 working ARGO nodes.

| Hash | U* ($S_a$ $T_a$) | U* ($S_b$ $T_a$) | U* ($S_a$ $T_b$) | U* ($S_b$ $T_b$) | **Final U* column** |
|---|---|---|---|---|---|
| A | 2/15 | 2/15 | 1/15 | N/A | 2/15 |
| B | 1/15 | 1/15 | 1/15 | N/A | 1/15 |
| C | 1/15 | 1/15 | 1/15 | N/A | 1/15 |
| D | 1/15 | 1/15 | 1/15 | N/A | 1/15 |
| E | 1/15 | 1/15 | 1/15 | N/A | 1/15 |
| F | 1/15 | 1/15 | 1/15 | 1/15 | 1/15 |
| G | N/A | 1/15 | 1/15 | 1/15 | 1/15 |
| H | N/A | 1/15 | 1/15 | 1/15 | 1/15 |
| I | N/A | 2/15 | 2/15 | 2/15 | 2/15 |
| J | N/A | 2/15 | 2/15 | 2/15 | 2/15 |
| K | N/A | 3/15 | 3/15 | 3/15 | 3/15 |
| Q | N/A | N/A | 2/15 | 2/15 | 2/15 |
| Y | N/A | N/A | 2/15 | 2/15 | 2/15 |
| Z | N/A | N/A | 3/15 | 3/15 | 3/15 |
| Sum | | | | | **23/15** |

## 8.7 Performance of JISPLIT

I measured the performance of **JISPLIT** in ARGO. For a 10 GB source file and a 10 GB target file, the processing time improved around 15 percent, from 13 minutes to 11 minutes. When I increased the number of Split files to more than a certain number, which depends on the original size of pre-split file, it took more time compared to calculating **JINF** without split. I believe that this is because of the pre- and post-processing overhead, which includes multiple file access operations. For a 10 GB source file and a 40 GB target file, **JISPLIT** shortens the processing time by about half (51% reduction), as shown in Table 8.16. To calculate the **JINF** between two huge images in my implementation is computationally intensive, especially with respect to the system memory. By partitioning the source and target media into smaller sized files and by distributing the computation, I can reduce the overall time to compute a similarity measure.

Even a single processor system can take advantage of my splitting method by calculating piece-by-piece and then post-processing the piecemeal results without using the parallel platform. By splitting the source and target to a manageable size and by giving it to a

Table 8.16: **JISPLIT** performance on ARGO platform.

| No of target split files | Processing time |
|---|---|
| 1 | 22 min 15 sec |
| 3 | 12 min 41 sec |
| 6 | 10 min 49 sec |
| 9 | 11 min 57 sec |
| 12 | 12 min 29 sec |
| source: 10 GB, target: 40 GB | |

process such as ARGO working node, which does not require high-end specification, I can improve the performance. The optimal number of splits for a certain sized hard disk image is an area to be further researched. It may depends not only the size but also on the hash layout.

## 8.8 Conclusion

In this chapter I showed how I can utilize parallel processing by splitting the images of interest. And I proved that the combined partial **JINF** results from each processor matches the single **JINF** value without splitting.

Experimental results provided as much as a 51% reduction in processing time. Processing time reductions varied based on the number of splits chosen and the specific digital media under examination, suggesting potential optimization strategies which I identified as future work.

# Chapter 9: Conclusions, Limitations, and Future Work

## Conclusions

Early triage and prioritization of digital evidence is an important task for the forensics analysts. The main goal of my research is to compute digital media image similarity to enable efficient triage and association discovery.

Most of the existing similarity measures work at the file level or the object level, while my approach works at the sector level and hence is robust in the face of deleted and partially overwritten data.

**JINF** is the first similarity method which compares hard disk images and provides more than 98% accuracy when it was validated with 400 image pairs.

I provide a novel algorithm for computing digital media similarity, **JINF**, and two extensions which can improve performance: **JIWS** and **JISPLIT**. My digital media similarity measure is based on a modified Jaccard index using sector hash values. My method has three key differences when compared to a basic Jaccard index calculation, specifically the exclusion of a white list of low-entropy sectors, a hash frequency weight to account for content uniqueness **(JIWF)**, and a normalization factor to allow for the accurate comparison of media of different sizes **(JINF)**. I validated the methods using disk images with known similarity, and I confirmed the highest accuracy and discrimination using the full **JINF** method.

The first performance extension, **JIWS**, samples media sectors instead of processing the whole image. Even with as little as 5% of the whole image, this method provided satisfactory results. To improve performance another way, I utilized parallel processing in my **JISPLIT** method. By partitioning the input media intelligently, and making necessary adaptations to re-combine the distributed results, I was able to reduce the computation time by up to 51%. **JIWS** can also utilize **JISPLIT** for more combined performance improvement.

I conclude that sector content comparisons, when appropriately computed, can provide an accurate and fast measure of digital media similarity, which in turn can assist with digital image triage decisions and link discovery across sources and entities.

**Limitations**

My approach may be vulnerable to a determined adversary willing to:

- Selectively delete and overwrite content in common with another digital device
- Plant false fragments to mislead the algorithm and examiner
- Wipe digital media at a low level
- Encrypt different media with unique keys

While these actions would severely limit the effectiveness of my approach, the first two require a considerable investment of time and skill on the adversary's part, and may be detectable after the fact. The last two items are effective but would also be obvious to an investigator, and so are more likely to lead to no result rather then false results. It should also be noted that my method will work on compressed and encrypted files, as long as the compression methods or encryption keys are the same across systems, and my method will work on damaged media or partially recovered content, as I do not require file system information.

**Future work**

Future research will expand the size of the validation test set, will confirm the utility of the normalization factor, and will evaluate the speed and performance tradeoff of statistical sampling. Practical and optimal partitioning schemes for the **JISPLIT** method will also be investigated.

As digital forensics data volumes and velocities continue to grow, future work may leverage and apply tools and methods from Big Data research to digital forensics. For example, *Hadoop* is an open-source software framework used for storing and processing Big Data in a distributed manner on large clusters of commodity hardware. *Hadoop* was developed and based on the paper written by *Google* on the MapReduce system, and it

applies concepts of functional programming. Though I have not implemented the **JINF**, **JIWS**, and **JISPLIT** methods in Big Data frameworks such as *Hadoop* or *Spark*, these methods are suitable for Big Data approaches.

Other future work will give more weight to key sectors, where the criteria for *key* remains to be determined, as well as incorporating the relative positions of matching sectors. Future work will also consider using matching sectors to direct analysts to specific files or remnants on a piece of digital media.

Moia et al. [31] researched common data which repeats over many files. Common data which repeats over many files is less relevant from a forensic analyst perspective. When similarity methods are applied, these data should be avoided. Common data can be generated by system processes, applications, or user actions. Those which are created by the user are typically more meaningful to forensic analysts. It would be valuable information to know the source and characteristics of these common sectors, and future work will investigate how such knowledge might be incorporated into the similarity measures. Reducing less meaningful common blocks also would improve the efficiency and accuracy of any similarity algorithm.

In my methods, OS sectors which are created by a clean OS install and sequence of NULL bytes are used as a whitelist. These sectors are removed as a pre-processing step before any similarity method. A deeper analysis of whitelisting may improve the algorithm's accuracy and efficiency.

# Appendix A: Data conversion

```
#####################################################
# This appendix chapter explains how to convert
# M57 data to vmdk files or hash files.
#
# Author: Myeong L. Lim
#         mlim4@masonlive.gmu.edu
#         Nov, 2020
#####################################################
```

M57 daily images are located at:

*http://downloads.digitalcorpora.org/corpora/scenarios/2009-m57-patents/drives-redacted/*

Since the file format of these images are E01, I used *FTK Imager Lite* to convert E01 files to *img* files.

From *FTK Imager Lite*, I add an E01 file to the evidence tree by choosing *Add Evidence Item* with *Image File* option. After right-clicking the evidence item, I choose *Export Disk Image* and click *Add* Button. After selecting the *Raw(dd)* option, type the case file information. On the *Select Image Destination* page, set *Image Fragment Size(MB)* as 0.

To convert *E01* file to *vmdk* file, I used the following command:

```
$ qemu-img convert -O vmdk file.E01 file.vmdk
```

To convert *img* file to a sector hash file, I used the following command:

```
$ md5deep -p 512 file.img > md5hash.txt
```

# Appendix B: ARGO Slurm batch file for SPLIT batch job

```
#####################################################
# This file is the Slurm batch file, which is used for
# splitting source and target.
# inputSplit.txt contains the list of source and target files.
# Detailed information about ARGO can be found from:
#    http://wiki.orc.gmu.edu/index.php/About_ARGO
#
# Author: Myeong L. Lim
#         mlim4@masonlive.gmu.edu
#         Nov, 2020
#####################################################


#!/bin/sh

## Give your job a name to distinguish it from other jobs you run.
#SBATCH --job-name=mySplit

## General partitions: all-HiPri, bigmem-HiPri   --   (12 hour limit)
##                     all-LoPri, bigmem-LoPri, gpuq  (5 days limit)
## Restricted: CDS_q, CS_q, STATS_q, HH_q, GA_q, ES_q, COS_q  (10 day limit)
#SBATCH --partition=all-HiPri

## Separate output and error messages into 2 files.
## NOTE: %u=userID, %x=jobName, %N=nodeID, %j=jobID, %A=arrayID, %a=arrayTaskID
```

```
#SBATCH --output=/scratch/%u/%x/%x-%N-%j.out  # Output file
#SBATCH --error=/scratch/%u/%x/%x-%N-%j.err   # Error file

##SBATCH --output=/scratch/%u/%x-%N-%j.out  # Output file
##SBATCH --error=/scratch/%u/%x-%N-%j.err   # Error file

## Slurm can send you updates via email
#SBATCH --mail-type=BEGIN,END,FAIL         # ALL,NONE,BEGIN,END,FAIL,REQUEUE,..
#SBATCH --mail-user=mlim4@gmu.edu     # Put your GMU email address here

## Specify how much memory your job needs. (2G is the default)
#SBATCH --mem=25G        # Total memory needed per task (units: K,M,G,T)

## Specify how much time your job needs. (default: see partition above)
##SBATCH --time=0-11:50  # Total time needed for job: Days-Hours:Minutes
#SBATCH --time=0-02:00

#SBATCH --ntasks=51   # 50 workers, 1 manager
## C:19,  J:21,  P:18,  T:20
## C_SNo:126, P_SNo : 119
####SBATCH --array=1-497

## Load the relevant modules needed for the job
module load openmpi/4.0.0
module load python/3.7.4
source ~/MPIpool/bin/activate
```

```
## Run your program or script

mpirun -np $SLURM_NTASKS python -m mpi4py.futures mySplit.py X inputSplit.txt
```

## Appendix C: Slurm batch file input: *inputSplit.txt*

```
#####################################################
# inputSplit.txt
#-------------------------------
# This file is used as an input file for Slurm batch.
# Each csv file from SRC and TRGT will be made as a pair
# and then sent to ARGO working node.
#
# Author: Myeong L. Lim
#         mlim4@masonlive.gmu.edu
#         Nov, 2020
#####################################################
SRC
/projects/jjones/mlim4/M57/CSVFrImageNoOS/C/c11_12_NoOS.csv
/projects/jjones/mlim4/M57/CSVFrImageNoOS/C/c11_16_NoOS.csv
/projects/jjones/mlim4/M57/CSVFrImageNoOS/C/c11_17_NoOS.csv
/projects/jjones/mlim4/M57/CSVFrImageNoOS/C/c11_18_NoOS.csv
/projects/jjones/mlim4/M57/CSVFrImageNoOS/C/c11_19_NoOS.csv
/projects/jjones/mlim4/M57/CSVFrImageNoOS/C/c11_20_NoOS.csv
/projects/jjones/mlim4/M57/CSVFrImageNoOS/C/c11_23_NoOS.csv
TRGT
/projects/jjones/mlim4/M57/CSVFrImageNoOS/P/p11_12_NoOS.csv
/projects/jjones/mlim4/M57/CSVFrImageNoOS/P/p11_16_NoOS.csv
/projects/jjones/mlim4/M57/CSVFrImageNoOS/P/p11_17_NoOS.csv
/projects/jjones/mlim4/M57/CSVFrImageNoOS/P/p11_18_NoOS.csv
/projects/jjones/mlim4/M57/CSVFrImageNoOS/P/p11_19_NoOS.csv
/projects/jjones/mlim4/M57/CSVFrImageNoOS/P/p11_20_NoOS.csv
```

/projects/jjones/mlim4/M57/CSVFrImageNoOS/P/p11_23_NoOS.csv

/projects/jjones/mlim4/M57/CSVFrImageNoOS/P/p11_24_NoOS.csv

## Appendix D: Python code for *mySplit.py*

```
############################################################
# This program implements JISPLIT method.
#    JISPLIT (Source, Target)
# Source is split into (S1, S2 ... Sn)
# Target is split into (T1, T2 ... Tm)
#------------------------------------------
# It reads the split source and split target list:
#     (S1, S2..Sn) and (T1, T2..Tm).
# It makes a pair (Si, Tj) from the source and target list.
# Sends each pair to an ARGO working node.
# Collects the intermediate outputs from each
# working node and does the final calculation for JISPLIT.
# Core subroutine is in splitJaccardMain.py
#    getJaccardIndexBetweenSrcAndTrgtSplitFile()
#
# Author: Myeong L. Lim
#         mlim4@masonlive.gmu.edu
#         Nov, 2020
############################################################
# MPIpool.py

from mpi4py.futures import MPIPoolExecutor
import math
import textwrap
from datetime import datetime
```

```
import os
import sys
from pathlib import Path
import jaccardMain as jm
import splitJaccardMain as sp
import util as utl
import pandas as pd
from subprocess import call

#Md5Csv='/projects/jjones/mlim4/M57/Md5Csv/'
Md5Csv='/projects/jjones/mlim4/M57/CSVFrImageNoOS/'
SampleWithOS='/projects/jjones/mlim4/M57/SampleWithOS/'
SampleNoOS='/projects/jjones/mlim4/M57/SampleNoOS/'



#
# C_Days is a Dictionary : "SLURM_ARRAY_TASK_ID" will be the key
#
# Days dictionary
src_Days = []

targetList = []
targetSumDict = {} # target List element will be Key
splitToTrgtMap = {}



#====================================
#  input file format
```

```
#================================================
# SRC
# sfilename1
# sfilename2
# TRGT
# tfilename1
# tfilename3


#=======================================
#  Main()
#---------------------------
#=======================================
# python mySplit.py opt paramFile
#   opt = X
#============================================================


def Main():


   # set up Src_Days and targetSplitList
   if processCommandArgs() == False:
      return


   maxSplit = 15
   forceSplit = True
   print("src_Days\n")
   print(src_Days)
   print("targetList\n")
   for t in targetList:
```

```
   print(t)

if forceSplit :
  for src in src_Days:
     for trgt in targetList:
        if cleanSplitAndMinMaxFolders(src, trgt) == False:
           return

#  create split files from all target files
bResult, splitBaseDirList =
       createSplitFilesFromTargetList(maxSplit, forceSplit)

dumpSplitBaseDirList(splitBaseDirList)

if bResult == False:
   return

dumpTargetSumDict()

#---------------------------------------
# if you want to manually set up Src_Days and targetList
# set here
#  src_Days
#  targetList  :  Dictionary
#---------------------------------------

# split the target list into subranges
# subranges = [(1,10),(10,20)..]
```

```
#i in this program actual subrange will be :
# subranges = [(1,1),(2,2)..]

for src in src_Days:
  for trgtPath in splitBaseDirList:
    start = datetime.now()  # for calc
    trgtBaseDirName = os.path.basename(os.path.normpath(trgtPath))
    srcBaseName=  os.path.basename(src)
    splitKey =  srcBaseName + ', '+  trgtBaseDirName +'.csv'
    # split files are under trgtPath
    subranges = determine_subranges(src, trgtPath)
    dumpSubranges(subranges)

    utl.printTimeWithInfoString
      (" executor is calling ("+srcBaseName+', '+trgtBaseDirName +")" )
    executor = MPIPoolExecutor()
    JI_sets = executor.map(calc_JI_Split, subranges)
    executor.shutdown()

    utl.printTimeWithInfoString
      ("executor.map is done ... JI_sets :\n")
    jiwsSum = 0.0
    jinfSum = 0.0
    crSum = 0.0
    splitCount  = 0
    jiwsList=[]
    jinfList=[]
    crList=[]
```

```
 # primes = [p for plist in prime_sets for p in plist]
 flatten = [p for plist in JI_sets for p in plist]
 print('flatten')
 print(flatten)

 '''
 print('plist')
 for plist in JI_sets :
   print(plist)

 #print(JI_sets)
# x = next(JI_sets)
# print(x)
 '''
 for jiInfo in flatten:
    utl.printTimeWithInfoString
       (str(splitCount)+ ': jiInfo\n')

    idx = jiInfo.find('jiws=')
    idx1 = jiInfo[idx+6:].find(',')
    jiwsSplit =jiInfo[idx+6: idx+6 + idx1]
    print(jiwsSplit)
    jiwsList.append(jiwsSplit)
    jiwsSum = jiwsSum + float(jiwsSplit)

    idx = jiInfo.find('jinf=')
    idx1 = jiInfo[idx+6:].find(',')
```

```
    jinfSplit = jiInfo[idx+6: idx+6+idx1]
    jinfList.append(jinfSplit)
    jinfSum = jinfSum + float(jinfSplit)

    idx = jiInfo.find('comRatio=')
    idx1 = jiInfo[idx+10:].find(',')
    crSplit =jiInfo[idx+10: idx+10+idx1]
    crList.append(crSplit)
    crSum = crSum + float(crSplit)
    splitCount = splitCount + 1

jiSplitFinal = 'JISPLIT(' +splitKey +':'+str(jinfSum/splitCount)
  'JISPLIT('+splitKey +') jinfSum' + str(jinfSum) + ', jiwsAvg'
  + str(jiwsSum/splitCount)+': comRatioAvg'+str(crSum/splitCount)
jiwsListString= " ".join(str(x)+',' for x in jiwsList)
jinfListString= " ".join(str(x)+',' for x in jinfList)
crListString= " ".join(str(x)+',' for x in crList)
jiwsListFinal ='JIWS_LIST('+splitKey +')=[' + jiwsListString +']'
jiwsListFinal =jiwsListFinal.replace(',]', ']')
jinfListFinal ='JINF_LIST('+splitKey+')=[' + jinfListString +']'
jinfListFinal =jinfListFinal.replace(',]',']')
crListFinal ='COMRATIO_LIST('+splitKey+')=[' + crListString +']'
crListFinal = crListFinal.replace(',]', ']')
jm.saveFinalJaccardSplitIndexToFile(jiSplitFinal)
jm.saveFinalJaccardSplitIndexToFile(jiwsListFinal)
jm.saveFinalJaccardSplitIndexToFile(jinfListFinal)
jm.saveFinalJaccardSplitIndexToFile(crListFinal)
utl.printTimeWithInfoString(jiSplitFinal )
```

```
        utl.printTimeWithInfoString(jiwsListFinal )
        utl.printTimeWithInfoString(jinfListFinal )
        utl.printTimeWithInfoString(crListFinal )

        sp.readMinMaxFileAndGetJisplit(src, trgtPath)

        end = datetime.now()  # calc for one pair(src,target)
        hours, minutes, seconds = sp.calcDuration(end, start)

        timeString =
         "{:2.0f} hr, {:2.0f}min, {:3.1f}sec".format(hours,minutes,seconds)
        utl.printTimeWithInfoString( splitKey  + ':' + timeString)

#================================================
# def cleanSplitAndMinMaxFolders(srcfile, trgtfile)
#------------------------------------------------
def cleanSplitAndMinMaxFolders(srcfile, trgtfile):
     splitDirName = getSplitDir(trgtfile)
     #call("cd " + splitDirName +"; rm *", shell=True)
     if splitDirName == "" :
        utl.printTimeWithInfoString
          ("<<<ERROR>> check the directory for input file:" + trgtfile)
        return False
     list = listOfCSVFiles(splitDirName)
     if len(list) > 0 :
        utl.printTimeWithInfoString
          ("<<<ERROR>> remove files from splitDirName =" + splitDirName)
        return False
```

```
      minMaxDir = sp.getMinMaxDirNames(srcfile, trgtfile)
      list = listOfCSVFiles(minMaxDir)
      if len(list) > 0 :
         utl.printTimeWithInfoString
           ("<<<ERROR>> remove files from minMaxDir =" + minMaxDir)
         return False
      return True


#================================================
# def readSourceAndTargetList(inFullFile)
#------------------------------------------------
def readSourceAndTargetList(inFullFile):
   with open(inFullFile, 'r') as paramF:
      Lines = paramF.readlines()
      state = 'N' # none
      src_cnt = 1
      trgt_cnt = 1
      for line in Lines :
         if state =='N' and line.find('SRC')>= 0 :
            state = 'S'  # Source
            continue

         if state =='S' and  line.find('TRGT')< 0 :
            src_Days.append(line.rstrip('\n'))
            src_cnt = src_cnt + 1

         if state =='S' and line.find('TRGT')>= 0 :
            state = 'T'  # Trgt
```

```
            continue

         if state =='T' and line.find('EXCLUDE') < 0:
            #targetSplitList[trgt_cnt] = line.rstrip('\n')
            targetList.append(line.rstrip('\n'))
            trgt_cnt = trgt_cnt + 1

         if state =='T' and line.find('EXCLUDE') >= 0:
            state = 'X'  # Trgt
            continue

         if state =='X' and line.find('EXCLUDE') < 0:
            targExcludeList.append(line.rstrip('\n'))  # Exclude
            trgtExclude_cnt = trgtExclude_cnt + 1



#================================================
# def listOfCSVFiles(path)
#------------------------------------------------
def listOfCSVFiles(path):
  # r=root, d=directories, f = files
   files = []
   for r, d, f in os.walk(path):
      for file in f:
         if '.csv' in file:
            files.append(os.path.join(r, file))
   return files
```

```
#================================================
# def dictOfSplitFiles(path)
#------------------------------------------------
def dictOfSplitFiles(path):
  # r=root, d=directories, f = files
   dictSplit={}
   cnt =0
   fileCnt = 0
   singlefile =''
   for r, d, f in os.walk(path):
      for file in f:
         utl.printTimeWithInfoString
           ("dictOfSplitFiles . file found : " + file)
         if '_split' in file:
            dictSplit[cnt] = os.path.join(r, file)
            cnt = cnt + 1
            fileCnt =  fileCnt + 1
         else:
            singlefile = os.path.join(r, file)
            fileCnt =  fileCnt + 1

   if cnt == 0 and fileCnt == 1 :
           # check if single file without '_split' in the file name
      dictSplit[cnt] = singlefile
   else:
      utl.printTimeWithInfoString
        ("<<WARNING>> check the directory for split files:" + path)
```

```
   return dictSplit


def  dumpSplitDict(splitDict):
   print("===== dumpSplitDict ===== length =" + str(len(splitDict)))
   for k in splitDict.keys():
      print(str(k) +":" + splitDict[k])


#================================================
# def determine_subranges(src, trgtPath)
#------------------------------------------------
def determine_subranges(src, trgtPath):
    print("d_s:  trgtPath:" + trgtPath )
    trgtSum = targetSumDict[trgtPath]
    subranges = []
    splitDict = dictOfSplitFiles(trgtPath)
    print("d_s:  after dictOfSplitFiles" )
    dumpSplitDict(splitDict)
    for k in splitDict.keys():
       subranges.append( (src, [splitDict[k], trgtSum]))


    return( subranges )



#================================================
# def splitLengthIntoRanges(length, noSplit)
#------------------------------------------------
def splitLengthIntoRanges(length, noSplit):
    intervalList =[]
```

```
    interval = length // noSplit
    cummulatedInterval = 0
    for i in range(noSplit):
       intervalList.append(cummulatedInterval)
       cummulatedInterval =  cummulatedInterval + interval

    print(intervalList)
    return intervalList



#================================================
# def checkSplitFileExist(path)
#------------------------------------------------
def checkSplitFileExist(path):
   if len(listOfCSVFiles(path)) == 0:
     return False

   return True

#================================================
# def checkSplitFileExist(path)
#------------------------------------------------
#  Split the taget file and save them to destDir
#------------------------------------------------

def splitTarget(trgtFile, destDir, noSplit, force):
    colNames = ['hash', 'offset']
    df = pd.read_csv(trgtFile, names=colNames, header=None, sep=',')
```

```
trgtGrpDict = df.groupby(['hash']).groups
sumTrgt = df['hash'].count()

bFileExist =  checkSplitFileExist(destDir)
if force == True and bFileExist:
  utl.printTimeWithInfoString
    ("Split files already exist in " + destDir)
  utl.printTimeWithInfoString
    ("<WARNING> Either set 'forceSplit' to False
              or clean directory: {}".format(destDir))
  return -1
if force != True and checkSplitFileExist(destDir):
  utl.printTimeWithInfoString
    ("Using Split files which are already exist in " + destDir)
  return sumTrgt

intervalList = splitLengthIntoRanges(sumTrgt, noSplit)
print("===== ST:intervalList ======")
print(intervalList)

print("===== ST: noSplit =" + str(noSplit))
totalNoDicts = 0
splitDictList =[]
for i in range(noSplit):
    strt = intervalList[i]
    if i < noSplit-1 :
       end = intervalList[i + 1]
       curDict = dict(list(trgtGrpDict.items())[strt:end])
```

```
    if len(curDict.keys()) != 0 :
        splitDictList.append(curDict)
        totalNoDicts = totalNoDicts + 1
        print(str(i) + ":totalNoDict=" + str(totalNoDicts) +
          "  cond1(" + str(strt) + ":" +
          str(end) +") length=" + str(len(curDict)))
    else:
        print(str(i) + " Breaking out ......:totalNoDict=" +
          str(totalNoDicts) + "  cond1(" + str(strt) +
          ":" + str(end) +") length=" + str(len(curDict)))
        break
else:
    curDict = dict(list(trgtGrpDict.items())[strt:])
    if len(curDict.keys()) != 0 :
        splitDictList.append(curDict)
        totalNoDicts = totalNoDicts + 1
        print(str(i)+":* totalNoDict="+str(totalNoDicts)+
         " cond2("+str(strt)+":) length="+str(len(curDict)))
    else:
        print(str(i) + ": Breaking out   ** totalNoDict=" +
          str(totalNoDicts)+" cond2("+str(strt)+ ":) len="+
          str(len(curDict)))
        break

#print("ST: i=" + str(i) + " --curDict dump ---")
#print(curDict)
# no of dictionary can be made less than the noSplit
'''
```

```
        if len(curDict.keys()) != 0 :
           splitDictList.append(curDict)
        else:
           break
        '''

    print(" --ST: totalNoDicts  ---" + str(totalNoDicts))

    #dumpSplitDictList(splitDictList)

    start = datetime.now()  # for calc
    trgtBasename = os.path.basename(trgtFile)
    for i in range(totalNoDicts) :
       trgtBasename = trgtBasename.replace('.csv','')
       splitFName = trgtBasename + '_split_' + str(i) +'.csv'
       splitFileName = Path(destDir) / splitFName
       splitToTrgtMap[splitFileName]= trgtFile
       print(str(i)+": createSplitFile called with "+str(splitFileName))
       createSplitFile(splitFileName, splitDictList[i])

    return sumTrgt

#================================================
# def createSplitFile(splitFileName, dict)
#------------------------------------------------
def createSplitFile(splitFileName, dict):
    start = datetime.now()
    with open(splitFileName, 'w') as splitF:
```

```
        for key, val in dict.items():
            for v in val:
                splitF.write(key+','+ str(v) +'\n')

    end = datetime.now()
    hours, minutes, seconds = sp.calcDuration(end, start)
    timeString =
      "{:2.0f}hr, {:2.0f}min, {:3.1f}sec".format(hours, minutes, seconds)
    #timeString =
      str(hours)  +"hr " + str(minutes) +"min, " + str(seconds) + "sec"
    utl.printTimeWithInfoString
     ("Split file: "+str(splitFileName)+" created. Time : "+timeString)

    return

#================================================
# def getSplitDir(t)
#------------------------------------------------
def getSplitDir(t):
   print(t)
   baseDir = os.path.splitext(t)[0]
   if baseDir.find('CSVFrImageNoOS') > 0 :
      baseDir = baseDir.replace('CSVFrImageNoOS' ,'SPLIT')
      return baseDir
   else:
      utl.printTimeWithInfoString
        ("Error: File is not under CSVFrImageNoOS .")
      return ""
```

```
#================================================
# def createSplitFilesFromTargetList(maxSplit, force)
#------------------------------------------------
def createSplitFilesFromTargetList(maxSplit, force):
   # create SPLIT files under
   utl.printTimeWithInfoString(" Split files ...  starting.")
   splitBaseDirList = []
   for t in targetList:
      print(t)
      baseDir = getSplitDir(t)
      if baseDir == "" :
         return False, []
      utl.createDirIfNotExist(baseDir)
      sum = splitTarget(t, baseDir, maxSplit, force)
      if sum < 0 :
          utl.printTimeWithInfoString
            ("<<WARNING>> splitTarget returns sum LE 0. return False.")
          return False, []
      targetSumDict[baseDir] = sum
      utl.printTimeWithInfoString
        ("targetSumDict["+baseDir+"] =" + str(sum))
      splitBaseDirList.append(baseDir)
   utl.printTimeWithInfoString(" Split files ...   end.")

   #dumpTargetSumDict()
   return  True, splitBaseDirList

#================================================
```

```
# def dumpSplitDictList(splitDictList)
#-------------------------------------------------
def dumpSplitDictList(splitDictList):
   print("= splitDictList =length :" + str(len(splitDictList)))
   for e in splitDictList:
      print(e)



#=================================================
# def dumpSubranges(subranges)
#-------------------------------------------------
def dumpSubranges(subranges):
   print("=== subranges === length :" + str(len(subranges)))
   for e in subranges:
      print(e)

#=================================================
# def dumpTargetSumDict()
#-------------------------------------------------
def dumpTargetSumDict():
   print("=== targetSumDict === length :" + str(len(targetSumDict)))
   for e in targetSumDict:
      print(e)



#=================================================
# def dumpSplitBaseDirList(splitBaseDirList)
#-------------------------------------------------
```

```
def dumpSplitBaseDirList(splitBaseDirList):
   print("=== splitBaseDirList === length :" + str(len(splitBaseDirList)))
   for e in splitBaseDirList:
      print(e)


#================================================
# def calc_JI_Split(range_tuple)
#------------------------------------------------
# range tuple will have (src, split_target)
#----------------------------
def calc_JI_Split(range_tuple):
   f1, f2_list = range_tuple
   f2 = f2_list[0]
   trgtSum = f2_list[1]
   utl.printTimeWithInfoString
   (os.path.basename(f1) +','+os.path.basename(f2)+':begin calc_JI_Split')
   #jiInfoList =[]
   #jiSplitInfo = []
   #jiSplitInfo.append( calcOneJi(f1, f2))
   #jiInfoList.append( jiSplitInfo )

   '''

   if type(jiSplitInfo) is list:
      jiInfoString = " ".join(str(x) for x in jiSplitInfo)
      utl.printTimeWithInfoString
        ('jiSplitInfo(list):' + jiInfoString)
   else:
```

```
      utl.printTimeWithInfoString
        ('jiSplitInfo:' + jiSplitInfo)
   '''
   # utl.printTimeWithInfoString(  ".....ending calc_JI_Split " )
   return calcOneJi(f1, f2, trgtSum)

#================================================
# def processCommandArgs()
#------------------------------------------------
def processCommandArgs():
   if len(sys.argv)  != 3 :
      utl.printTimeWithInfoString
        ("Requires inputFile to read. usage:'mySplit.py X fName'")
      return False

   if sys.argv[1] == 'X': # we use Slurm
      if Path(sys.argv[2]).is_file():
          readSourceAndTargetList(sys.argv[2])
      else :
          utl.printTimeWithInfoString
            ( sys.argv[2] + "  is not a file or does not exist.")
          return False
   else :
      utl.printTimeWithInfoString
        ( "Override option 'X' or 'Y' is the only option.")
      return False

   utl.printTimeWithInfoString("argc={},argv={} {}".
```

```
          format(str(len(sys.argv)),sys.argv[1],sys.argv[2]))

   return True

#============================================
# def calcOneJi(f1, f2, totalF2)
# f1 and f2 are fullPath file name
#--------------------------------------------
def calcOneJi(f1, f2, totalF2):
  if utl.JI_SplitAlreadyCalculated
             (os.path.basename(f1), os.path.basename(f2)) :
    utl.printTimeWithInfoString
     (os.path.basename(f1)+','+os.path.basename(f2)+':already created')
    return 'Already Calculated'
  else:
    #totalF2 =  8498819
    #totalF2 =  15
# trgtName = splitToTrgtMap[f2]
# totalF2  = sumTrgtDict[trgtName]
#print('trgtname ='+trgtName+', sumFrDict='+str(sumTrgtDict[trgtName]))
    ji_Info=sp.getJaccardIndexBetweenSrcAndTrgtSplitFile(f1,f2,totalF2)

    utl.printTimeWithInfoString
      ( os.path.basename(f1) +',' + os.path.basename(f2) + ': ends')

      #  jm.dumpJaccardIndex()

  return ji_Info
```

```
if __name__ == '__main__':

    Main()
```

## Appendix E: Python code for *jaccardMain.py*

```
############################################################
# This program implements JINF method.
#    JINF (Source, Target).
#-------------------------------------------
# It can read several Source images
# and multiple Target images.
# (Source_1, Source_2 ... Source_k)
# (Target_1, Target_2 ... Target_n)
#-------------------------------------------
# It reads  the source and target lists.
# It makes a pair from source and target list.
# Sends each pair to an ARGO working node.
# Each working node individually calculates
# the JINF for each pair.
#
# Author: Myeong L. Lim
#         mlim4@masonlive.gmu.edu
#         Nov, 2020
############################################


import numpy as np
import pandas as pd
import ntpath
import time
from datetime import datetime
from time import gmtime, strftime
```

```
import util
import os

#================================================
# def MinMaxFind(n1, n2)
#------------------------------------------------
def MinMaxFind(n1, n2):
    if (n1 < n2):
        return n1, n2
    else:
        return n2, n1

#================================================
# def path_leaf(path)
#------------------------------------------------
def path_leaf(path):
    head, tail = ntpath.split(path)
    return tail or ntpath.basename(head)

jaccard = {}



#================================================
# def getSizeAdjustFactor(sum1, sum2)
#------------------------------------------------
def getSizeAdjustFactor(sum1, sum2) :
   if (sum1 > sum2):
       factor1 = 1.0
```

```
        factor2 = sum1 / sum2
    else:
        factor1 = sum2 / sum1
        factor2 = 1.0

    return factor1, factor2


#================================================
# def getJaccardIndexBetweenTwoWithSampleAndRatioAdjust
                  (srcfile, trgtFile, idxSlurm):
#   JIWS with sample
#------------------------------------------------
def getJaccardIndexBetweenTwoWithSampleAndRatioAdjust
                  (srcfile, trgtFile, idxSlurm):
    start = datetime.now()  # for calc
    startTimeForDisplay = time.strftime("%c")  # for display
    # Load spreadsheet
    colNames = ['hash', 'offset']
    src = pd.read_csv(srcfile, names=colNames, header=None, sep=',')
    trgt = pd.read_csv(trgtFile, names=colNames, header=None, sep=',')
    # df = pd.concat([src, trgt], names=['h1', 'h2'], axis=1)

    # print(src)
    # print (src['hash'])
    df2 = pd.concat([src['hash'], trgt['hash']], axis=1)

    df2.columns = ['h1', 'h2']
```

```
# print(df2.columns)
# print(df2)
# print(df2['h1'].count())
h1grp = df2.groupby(['h1']).groups
sumGrp1 = df2['h1'].count()
#print(" Group 1 done.")
h2grp = df2.groupby(['h2']).groups
sumGrp2 = df2['h2'].count()
#print(" Group 2 done.")
ratioDict1 = {}
ratioDict2 = {}

# get size adjust factor
factor1, factor2 = getSizeAdjustFactor(sumGrp1, sumGrp2)

total_1_factored = sumGrp1 * factor1
total_2_factored = sumGrp2 * factor2

cntD1_factored = {}
cntD2_factored = {}

#  ratioDict1[key]  is a list []
# after this
#     ratioDict1[key][0] will contain
#       (src count of unique hash) * factor1 / total_1_factored
# after next for loop,
#     ratioDict2[key][0] will contain
#       (trgt count of unique hash) * factor2 / total_2_factored
```

```
# eventually
#    ratioDict1[key][1] will contain
#       (trgt count of unique hash) * factor2  / total_2_factored
#    ratioDict1[key][0] and ratioDict1[key][1]
#            will be used to calc MinMax.
for key in h1grp.keys():
    cntD1_factored[key] = factor1 * len(h1grp[key])
    #print("h1 key ={}: **len {}".format(key, len(h1grp[key])))
    ratioDict1[key] = [cntD1_factored[key] / total_1_factored]
    # Entrophy

# print("==  Ratio Dict 1")
# print(ratioDict1)
commonUniqCnt = 0
commonTotalCntFactored1 = 0
commonTotalCntFactored2 = 0
for key in h2grp:
    cntD2_factored[key] = factor2 * len(h2grp[key])
    # print("h2 key ={}: *** len {}".format(key, len(h2grp[key])))
    # print("{}: {}".format(key, len(h2grp[key])))
    # dict2[key] = [len(h2grp[key])]
    # give the adjusted count of each hash
    ratioDict2[key] = [ cntD2_factored[key] / total_2_factored ]
    if key in ratioDict1.keys():  # common key
        commonUniqCnt = commonUniqCnt + 1
        commonTotalCntFactored1 =  commonTotalCntFactored1 +
                                      cntD1_factored[key]
        commonTotalCntFactored2 =  commonTotalCntFactored2 +
```

```
                                        cntD2_factored[key]
        # print("{} :: {}".format(key, len(dict1[key])))
        ratioDict1[key].append(ratioDict2[key][0]) #
    else: # src does not have key, so set it as 0
        ratioDict1[key] = [0, ratioDict2[key][0]]
# By now, rD1[h0] has only one element.
# src(h0 = 1, h1 =1) trgt(h1 = 2, h2 =2)
# rD1[h0] = [1]
# rD1[h1] = [1,2]   rD2[h1]=[2]
# rD1[h2] = [0,2]   rD2[h2]=[2]
# make rD1[h0]  = [1, 0] in the following loop by appending 0
#

minSum = 0
maxSum = 0
minSumEnt = 0
maxSumEnt = 0
for key in ratioDict1.keys():
    if len(ratioDict1[key]) == 1:
        # print(" h3 key {}: {}".format(key, ratioDict1[key]))
        ratioDict1[key].append(0)

        ratioDict2[key] = 0.0

    min, max = MinMaxFind(ratioDict1[key][0], ratioDict1[key][1])
    if key in h1grp.keys() and key in h2grp.keys() :
        minEnt, maxEnt =
           MinMaxFind(ratioDict1[key][0]/len(h1grp[key]),
```

```
                        ratioDict1[key][1]/len(h2grp[key]))
    elif   key in h1grp.keys() and key not in h2grp.keys() :
        minEnt, maxEnt =
           MinMaxFind(ratioDict1[key][0] / len(h1grp[key]), 0)
    elif key not in h1grp.keys() and key in h2grp.keys() :
        minEnt, maxEnt =
           MinMaxFind(0, ratioDict1[key][1]/len(h2grp[key]))

  # print("key={}(min, Max)=({}, {}),(minEnt, MaxEnt)=({}, {})".
  #            format( key, min, max,  minEnt, maxEnt))

    minSum = minSum + min
    maxSum = maxSum + max
    minSumEnt = minSumEnt + minEnt
    maxSumEnt = maxSumEnt + maxEnt
    # print("min =" + str(min) + ", max =" + str(max))
jinf = minSum / maxSum
jinfEnt = minSumEnt / maxSumEnt
# but ratioDict1 has all the keys from both(source and target)

srcUniqKeysSum = len(h1grp.keys())
trgtUniqKeysSum = len(h2grp.keys())
if (factor1 > factor2):
    commonRatio = commonUniqCnt / srcUniqKeysSum
    commonTotalRatioFactored=commonTotalCntFactored1/total_1_factored
    print("Common ratio from src")
else:
    commonRatio = commonUniqCnt / trgtUniqKeysSum
```

```
    commonTotalRatioFactored=commonTotalCntFactored2/total_2_factored
    print("Common ratio from trgt")

jaccardkey = (path_leaf(srcfile), path_leaf(trgtFile))

# print('commonCnt =' + str(commonUniqCnt) + ', srcUniqKeysSum =' +
# str(srcUniqKeysSum) + ', trgtUniqKeysSum =' + str(trgtUniqKeysSum))

end = datetime.now()
duration = end - start
duration_in_s = duration.total_seconds()
hours = divmod(duration_in_s, 3600)[0]
minutes = divmod(duration_in_s, 60)[0]
seconds = divmod(duration_in_s, 60)[1]
# strftime("%Y-%m-%d %H:%M:%S", gmtime())

jiws = commonRatio * (commonRatio * (1 - jinf) + jinf)
jiws_ENT = commonRatio * (commonRatio * (1 - jinfEnt) + jinfEnt)

sJiws = str('{:05.5f}'.format(jiws))
sJiws_ENT = str('{:05.5f}'.format(jiws_ENT))
sCommonRatio =  str('{:05.5f}'.format(commonRatio))
sJinf = str('{:05.5f}'.format(jinf))
sJinfEnt = str('{:05.5f}'.format(jinfEnt))
jiInfoString =
  'jiws= {},jiws_ENT= {},comRatio= {},jinf= {},jinfEnt = {}'.format(
              sJiws,  sJiws_ENT, sCommonRatio, sJinf, sJinfEnt)
minuteString = '{:3.0f}'.format(minutes)
```

```
secondString = '{:3.0f}'.format(seconds)
keyCountString = 'srcUniqueCount=' + str(srcUniqKeysSum) +
   ', trgtUniqueCount=' +  str(trgtUniqKeysSum) +
   ', comUniqueCount=' + str(commonUniqCnt)

sCommonTotalRatioFactored = '{:05.5f}'.format(commonTotalRatioFactored)
keyTotalCountString = 'srcTotalFactored='+str(int(total_1_factored))+
  ', trgtTotalFactored=' + str(int(total_2_factored)) +
   ', comTotalFactored_1=' + str( int(commonTotalCntFactored1)) +
    ', comTotalFactored_2=' + str(int(commonTotalCntFactored2)) +
    ', commonTotalRatio =' + sCommonTotalRatioFactored
jaccard[jaccardkey] = [jiInfoString]
jaccard[jaccardkey].append(keyCountString)
jaccard[jaccardkey].append(keyTotalCountString)
jaccard[jaccardkey].append('startTime =' + startTimeForDisplay)
jaccard[jaccardkey].append("{} min {} sec".
     format(minuteString, secondString))

#saveJaccardIndexToFile
if (idxSlurm == -1): #Split JI case
   saveJaccardSplitIndexToFile(jaccardkey)
else:
   saveJaccardIndexToFile(idxSlurm, jaccardkey)

#print(jaccard[jaccardkey])

return (str(jaccard[jaccardkey]))
```

```
#================================================
# def saveFinalJaccardSplitIndexToFile(finalResult)
#  final result of JISPLIT(src, target)
#  is saved
#------------------------------------------------
def saveFinalJaccardSplitIndexToFile(finalResult):
    jiInfoFile ='/scratch/mlim4/mySplit/OUTFILES/' +
                  'JI_SplitInfo.txt'
    with open(jiInfoFile, 'a+') as infofile:
       info = "{}".format(finalResult)
       infofile.write(info +'\n')


#================================================
# def saveJaccardSplitIndexToFile(jaccardkey)
#  individual result of JIWS(src, target_split_i)
#  is saved
#------------------------------------------------
def saveJaccardSplitIndexToFile(jaccardkey):
    jiInfoFile ='/scratch/mlim4/mySplit/OUTFILES/' +
                    'JI_SplitInfo.txt'
    with open(jiInfoFile, 'a+') as infofile:
       info = "{} :: {}".format(jaccardkey, jaccard[jaccardkey])
       infofile.write(info +'\n')


#================================================
# def getJaccardIndexBetweenTwo(srcfile, trgtFile)
#------------------------------------------------
def saveJaccardIndexToFile(idxSlurm, jaccardkey):
```

```
    jiInfoFile ='/scratch/mlim4/myJaccard/OUTFILES/' +
                  'JI_Info_' + str(idxSlurm) + '.txt'
    with open(jiInfoFile, 'a+') as infofile:
       info = "{} :: {}".format(jaccardkey, jaccard[jaccardkey])
       infofile.write(info +'\n')

#================================================
# def getJaccardIndexBetweenTwo(srcfile, trgtFile)
#------------------------------------------------
def getJaccardIndexBetweenTwo(srcfile, trgtFile):
    start = datetime.now()  # for calc
    startTimeForDisplay = time.strftime("%c")  # for display
    # Load spreadsheet
    colNames = ['hash', 'offset']
    src = pd.read_csv(srcfile, names=colNames, header=None, sep=',')
    trgt = pd.read_csv(trgtFile, names=colNames, header=None, sep=',')
    # df = pd.concat([src, trgt], names=['h1', 'h2'], axis=1)

    # print(src)

    # print (src['hash'])
    df2 = pd.concat([src['hash'], trgt['hash']], axis=1)

    df2.columns = ['h1', 'h2']
    # print(df2.columns)
    # print(df2)
    # print(df2['h1'].count())
    h1grp = df2.groupby(['h1']).groups
```

```
sumGrp1 = df2['h1'].count()
print(" Group 1 done.")
h2grp = df2.groupby(['h2']).groups
sumGrp2 = df2['h2'].count()
print(" Group 2 done.")
dict1 = {}
dict2 = {}
print("Total 1: " + str(sumGrp1) + ", Total 2: " + str(sumGrp2))

for key in h1grp:
    #print("h1 key ={}: **,,,,,len {}".format(key, len(h1grp[key])))
    dict1[key] = [len(h1grp[key]) / sumGrp1]

# print("==  Dict 1")
# print(dict1)

for key in h2grp:
    # print("h2 key ={}: **,,,,* len {}".format(key, len(h2grp[key])))
    # print("{}: {}".format(key, len(h2grp[key])))
    dict2[key] = [len(h2grp[key])]  # give the count of each hash

    if key in dict1:
        # print("{} :: {}".format(key, len(dict1[key])))
        dict1[key].append(len(h2grp[key]) / sumGrp2)
    else:
        dict1[key] = [len(h2grp[key]) / sumGrp2]

# At this, some will have only one element in the list of dict1.
```

```
# These are hash value which were not common.
# but dict1 will have all the keys from both : source and target.



minSum = 0
maxSum = 0
for key in dict1:
    if len(dict1[key]) == 1:
      #print(" h3 key {}: {}".format(key, dict1[key]))
      dict1[key].append(0)


    min, max = MinMaxFind(dict1[key][0], dict1[key][1])
    minSum = minSum + min
    maxSum = maxSum + max
    # print("min =" + str(min) + ", max =" + str(max))

jaccardIndex = minSum / maxSum
jaccardkey = (path_leaf(srcfile), path_leaf(trgtFile))
print("Jaccard Idx " + str(jaccardkey)+" : "+str(jaccardIndex))

end = datetime.now()
duration = end - start
duration_in_s = duration.total_seconds()
hours = divmod(duration_in_s, 3600)[0]
minutes = divmod(duration_in_s, 60)[0]
seconds = divmod(duration_in_s, 60)[1]
# strftime("%Y-%m-%d %H:%M:%S", gmtime())
```

```
    jaccard[jaccardkey] = [jaccardIndex]


    jaccard[jaccardkey].append(startTimeForDisplay)
    jaccard[jaccardkey].
        append((" process time:(mm, ss) ", minutes, seconds))


    return jaccardIndex

#=================================================
# def dumpJaccardIndex()
#-------------------------------------------------
def dumpJaccardIndex():
    for key in jaccard:
         print("{} :: {}".format(key, jaccard[key]))


#=================================================
# def saveJaccardIndexFullToFile(idxSlurm)
#-------------------------------------------------
# It compares only the whole image and save the JI
#-------------------------------------------------
def saveJaccardIndexFullToFile(idxSlurm):
   jiInfoFile ='/scratch/mlim4/myJaccard/OUTFILES/' +
        'JI_FullInfo_' + str(idxSlurm) + '.txt'
   with open(jiInfoFile, 'a+') as infofile:
       for key in jaccard:
          info = "{} :: {}".format(key, jaccard[key])
          infofile.write(info +'\n')
```

```
# for the quick test check up
srcfile = 'c:/Temp/JaccardData/source.csv'
trgtList = {'c:/Temp/JaccardData/target1.csv',
            'c:/Temp/JaccardData/target2.csv',
            'c:/temp/JaccardData/target3.csv',
            'c:/temp/JaccardData/target4.csv',
            'c:/temp/JaccardData/target5.csv',
            'c:/temp/JaccardData/target6.csv',
            'c:/temp/JaccardData/target7.csv'}

def Main():

    for trgtFile in sorted(trgtList):
       getJaccardIndexBetweenTwoWithRatioAdjust(srcfile, trgtFile)

    dumpJaccardIndex()

if __name__ == '__main__':
    Main()
```

## Appendix F: Python code for *splitJaccardMain.py*

```
############################################################
# This file contains the main routine for JISPLIT.
# From mySplit.py, this function is called for JISPLIT.
#
#      getJaccardIndexBetweenSrcAndTrgtSplitFile()
#
# Author: Myeong L. Lim
#         mlim4@masonlive.gmu.edu
#         Nov, 2020
############################################################

import numpy as np
import pandas as pd
import ntpath
import time
from datetime import datetime
from time import gmtime, strftime
import util as utl
import os

#================================================
# def MinMaxFind(n1, n2)
#------------------------------------------------
def MinMaxFind(n1, n2):
    if (n1 < n2):
        return n1, n2
```

```
    else:
        return n2, n1

#================================================
# def path_leaf(path)
#------------------------------------------------
def path_leaf(path):
    head, tail = ntpath.split(path)
    return tail or ntpath.basename(head)

jaccard = {}

#================================================
# def getSizeAdjustFactor(sum1, sum2)
#------------------------------------------------
def getSizeAdjustFactor(sum1, sum2) :
   if (sum1 > sum2):
       factor1 = 1.0
       factor2 = sum1 / sum2
   else:
       factor1 = sum2 / sum1
       factor2 = 1.0

   return factor1, factor2

#================================================
# def dumpH1grp(h1grp):
#------------------------------------------------
```

```
def dumpH1grp(h1grp):
   print("== h1grp ===")
   print(h1grp)
   for k in h1grp.keys():
     print(k + ':' + str(len(h1grp[k])))


#================================================
# def dumpH2grp(h2grp):
#------------------------------------------------
def dumpH2grp(h2grp):
   print("== h2grp ===")
   print(h2grp)
   for k in h2grp.keys():
     print(k + ':' + str(len(h2grp[k])))


#================================================
# def dumpCntD1(cntD1)
#------------------------------------------------
def dumpCntD1(cntD1):
   print('==== dumpCntD1 === length:' + str(len(cntD1.keys())))
   for k in cntD1.keys():
      print (k +':')
      print (cntD1[k])


#================================================
# def dumpCntD2(cntD2)
#------------------------------------------------
def dumpCntD2(cntD2):
```

```
   print('==== dumpCntD2 === length:' + str(len(cntD2.keys())))
   for k in cntD2.keys():
      print (k +':')
      print (cntD2[k])

#================================================
# def getJaccardIndexBetweenSrcAndTrgtSplitFile
                (srcfile, trgtFile, totalSizeOfTrgt):
#------------------------------------------------
#   JISPLIT core code
#------------------------------------------------
# totalSizeOfTrgt: total count of trgt file before split
#----------------------------------------------------------------
def getJaccardIndexBetweenSrcAndTrgtSplitFile
                  (srcfile, trgtFile, totalSizeOfTrgt):
    start = datetime.now()  # for calc
    startTimeForDisplay = time.strftime("%c")  # for display
    # Load spreadsheet
    colNames = ['hash', 'offset']
    src = pd.read_csv(srcfile, names=colNames, header=None, sep=',')
    trgt = pd.read_csv(trgtFile, names=colNames, header=None, sep=',')
    # df = pd.concat([src, trgt], names=['h1', 'h2'], axis=1)

    # print(src)
    # print (src['hash'])
    df2 = pd.concat([src['hash'], trgt['hash']], axis=1)

    df2.columns = ['h1', 'h2']
```

```
h1grp = df2.groupby(['h1']).groups
sumGrp1 = df2['h1'].count()
#print(" Group 1 done.")
h2grp = df2.groupby(['h2']).groups
# sumGrp2 will not be used in SPLIT
sumGrp2 = df2['h2'].count()
#print(" Group 2 done.")

#dumpH1grp(h1grp)

#dumpH2grp(h2grp)

cntD1 = {}
cntD2 = {}

commonUniqCnt = 0
for key in h1grp:
    cntD1[key] = [len(h1grp[key]) / sumGrp1]

#dumpCntD1(cntD1)
for key in h2grp:
    normalizedVal = len(h2grp[key]) / totalSizeOfTrgt
    cntD2[key] = [normalizedVal]

    if key in cntD1:
        commonUniqCnt = commonUniqCnt + 1
        # print("{} :: {}".format(key, len(cntD1[key])))
```

```
        cntD1[key].append(normalizedVal)
    else:
        cntD1[key] = [normalizedVal]

#dumpCntD1(cntD1)
#dumpCntD2(cntD2)
# At this, some will have only one element in the list of cntD1
#    will have all the keys from both : source and target.


minMaxDict = {}
minSum = 0
maxSum = 0
for key in cntD1:
    if len(cntD1[key]) == 1:
      cntD1[key].append(0)

    min, max = MinMaxFind(cntD1[key][0], cntD1[key][1])
    minMaxDict[key]=[min, max]
    minSum = minSum + min
    maxSum = maxSum + max
    #print("min =" + str(min) + ", max =" + str(max))
#print("Final cntD1 contents...")
#dumpCntD1(cntD1)

jinf = minSum / maxSum
jaccardkey = (path_leaf(srcfile), path_leaf(trgtFile))
print("JINF " + str(jaccardkey) + " : " + str(jinf))
```

```
end = datetime.now()
hours, minutes, seconds = calcDuration(end, start)

srcUniqKeysSum = len(h1grp.keys())
trgtUniqKeysSum = len(h2grp.keys())

commonRatio = commonUniqCnt / trgtUniqKeysSum
jiws = commonRatio * (commonRatio * (1 - jinf) + jinf)

#print("before formatting")



keyCountString = ', srcUniqueCount=' + str(srcUniqKeysSum) +
    ', trgtUniqueCount='+str(trgtUniqKeysSum) + ', comUniqueCount='+
     str(commonUniqCnt)
#print("after key count string")
jiInfoString='jiws= ' + str(jiws) + ', comRatio= '+str(commonRatio)+
         ', jinf= ' + str(jinf)

timeString = ', time took: '+str(hours)+'hrs '+str(minutes) +'min '+
                    str(seconds) + 'sec'
#print('after Jiinfo string ')
result = jiInfoString + keyCountString + timeString

#print('after Jaccrd time append format ')
saveJaccardSplitIndexToFile(result)
```

```
    srcBaseName = path_leaf(srcfile)
    srcBaseName = srcBaseName.replace('.csv','')

    minMaxFName = srcBaseName + '_' +  path_leaf(trgtFile)
    print("====== minMaxFName ====")
    print(minMaxFName)

    print('getJaccard bef getMinMax  srcfile'+srcfile +': '+trgtFile)
    minMaxDir = getMinMaxDirNames(srcfile, trgtFile)
    print('get JI minMaxDir='+minMaxDir+', minMaxFName='+minMaxFName)
    utl.createDirIfNotExist(minMaxDir)
    saveMinMaxDict(minMaxDir, minMaxFName, minMaxDict)

    # return List

    #print(" === result===")
    #print(result)

    resultList =[]
    resultList.append(result)
    #print(" === resultList===")
    #print(resultList)

    return resultList

#================================================
# def calcDuration(end, start)
#------------------------------------------------
```

```
def calcDuration(end, start):
    duration = end - start
    duration_in_s = duration.total_seconds()
    hours = divmod(duration_in_s, 3600)[0]
    minutes = divmod(duration_in_s, 60)[0]
    seconds = divmod(duration_in_s, 60)[1]

    return hours, minutes, seconds

#================================================
# def getMinMaxDirNames(srcfile, trgtFile)
#------------------------------------------------
def getMinMaxDirNames(srcfile, trgtFile):
    srcBaseName = path_leaf(srcfile)
    srcBaseName = srcBaseName.replace('.csv','')
    trgtBaseName = path_leaf(trgtFile)
    trgtBaseName = trgtBaseName.replace('.csv','')

    minMaxFName = srcBaseName + '_' + trgtBaseName
    idx = minMaxFName.find('_split')
    if idx >= 0 :
       minMaxDirBasename = minMaxFName[:idx]
    else:
       minMaxDirBasename = minMaxFName

    minMaxDir ='/scratch/mlim4/mySplit/OUTFILES/' +
                                  minMaxDirBasename
```

```
    return minMaxDir


#================================================
# def dumpFinalDict(dict)
#------------------------------------------------
def dumpFinalDict(dict):
   print(" ==== dumpFinalDict ===")


   for k in dict.keys():
     print (k+':')
     print (dict[k])


#================================================
# def readMinMaxFileAndGetJisplit(srcfile, trgtfile)
#------------------------------------------------
def readMinMaxFileAndGetJisplit(srcfile, trgtfile):
   print('readMinMax src='+srcfile+': trgt='+trgtfile)
   minMaxDir = getMinMaxDirNames(srcfile, trgtfile)
   # use only minMaxDir
   print("rdMinMax: minMaxDir=" + minMaxDir )
   colNames = ['hash', 'min', 'max']
   list = listOfDictCSVFiles(minMaxDir)
   print
     ("listOfDictCSVFiles(minMaxDir) len="+str(len(list)))
   print(list)
   cnt =0
   for f in list:
      print('f=' + f)
```

```
      if cnt == 0:
         finalDict = makeDictFromFile(f)
         #dumpFinalDict(finalDict)
      else:
         midDict = makeDictFromFile(f)
         adjustFinalDict(finalDict, midDict)

      cnt = cnt + 1

   maxSum = 0
   for key in finalDict:
      if key != 'minSum' :
         maxSum = maxSum + finalDict[key][1]

   print('minSum = ' + str(finalDict['minSum']) +
                        ', maxSum=' + str(maxSum))
   print('jinf =' + str(finalDict['minSum'] /  maxSum))

#================================================
# def adjustFinalDict(finalDict, dict)
#------------------------------------------------
def adjustFinalDict(finalDict, dict):
   for key in dict.keys():
      if key not in finalDict.keys():
         finalDict[key] = dict[key]

   for key in finalDict.keys():
      if key  != 'minSum' and  key in dict.keys():
```

```
            if finalDict[key][1] < dict[key][1]:
               finalDict[key][1] = dict[key][1]


   finalDict['minSum'] =
       finalDict['minSum']+dict['minSum']

#================================================
# def makeDictFromFile(f)
#------------------------------------------------
def makeDictFromFile(f):
   d = {}
   with open(f, 'r')as  fp:
      minSum = 0.0
      for row in fp:
         hash, min, max = row.split(',')
         d[hash]=[float(min), float(max)]
         minSum = minSum + float(min)

   d['minSum'] = minSum
   return d

#================================================
# def listOfDictCSVFiles(path)
#------------------------------------------------
def listOfDictCSVFiles(path):
    # r=root, d=directories, f = files
    files = []
```

```
    for r, d, f in os.walk(path):
       for file in f:
          if 'wkusb' in file:
              continue
          if 'start' in file:
              continue
          if '.csv' in file:
             files.append(os.path.join(r, file))
    return files

#================================================
# def saveMinMaxDict(dir, fname, minMaxDict)
#------------------------------------------------
def saveMinMaxDict(dir, fname, minMaxDict):
    print("saveMinMax: dir=" + dir +", fname="+fname)

    jiInfoFile = dir + '/' + fname

    with open(jiInfoFile, 'w') as minMaxF:
       for k in minMaxDict.keys():
          minMaxF.write(k +','+str(minMaxDict[k][0])+
             ','+ str(minMaxDict[k][1]) +'\n')

#================================================
# def saveFinalJaccardSplitIndexToFile(finalResult)
#  final result of JISPLIT(src, target)
#  is saved
#------------------------------------------------
```

```
def saveFinalJaccardSplitIndexToFile(finalResult):
    jiInfoFile ='/scratch/mlim4/mySplit/OUTFILES/' +
                              'JI_SplitInfo.txt'
    with open(jiInfoFile, 'a+') as infofile:
       info = "{}".format(finalResult)
       infofile.write(info +'\n')


#================================================
# def saveJaccardSplitIndexToFile(resultString)
#  individual result of JIWS(src, target_split_i)
#  is saved
#------------------------------------------------
def saveJaccardSplitIndexToFile(resultString):
    jiInfoFile ='/scratch/mlim4/mySplit/OUTFILES/' +
                        'JI_SplitInfo.txt'
    with open(jiInfoFile, 'a+') as infofile:
       infofile.write(resultString +'\n')


#================================================
# def dumpJaccardIndex():
#------------------------------------------------
def dumpJaccardIndex():
    for key in jaccard:
         print("{} :: {}".format(key, jaccard[key]))


if __name__ == '__main__':
    Main()
```

# Appendix G: Python code for *util.py*

```
#####################################################
# This file contains utility routines for
# JINF and JISPLIT calculations.
#
# Author: Myeong L. Lim
#         mlim4@masonlive.gmu.edu
#         Nov, 2020
#####################################################



import os
import time
from datetime import datetime
from pathlib import Path


#================================================
# def createDirIfNotExist(dirName)
#------------------------------------------------
def createDirIfNotExist(dirName):
   if not os.path.exists(dirName):
       os.mkdir(dirName)
       print("Directory " , dirName ,  " Created ")
   else:
       print("Directory " , dirName ,  " already exists")


#================================================
```

```
# def listOfCSVFullPathFiles(path)
#-------------------------------------------------
def listOfCSVFullPathFiles(path):
    # r=root, d=directories, f = files
    files = []
    for r, d, f in os.walk(path):
       for file in f:
          if 'wkusb' in file:
              continue
          if 'start' in file:
              continue
          if '.csv' in file:
             files.append(os.path.join(r, file))
    return sorted(files)


#=================================================
# def listOfCSVBaseFiles(path)
#-------------------------------------------------
def listOfCSVBaseFiles(path):
    # r=root, d=directories, f = files
    files = []
    for r, d, f in os.walk(path):
       for file in f:
          if 'wkusb' in file:
              continue
          if 'start' in file:
              continue
          if '.csv' in file:
```

```
            #files.append(os.path.join(r, file))
            files.append(file)
    return sorted(files)

#================================================
# def listOfTextFiles(path)
#------------------------------------------------
def listOfTextFiles(path):
   # r=root, d=directories, f = files
   files = []
   for r, d, f in os.walk(path):
      for file in f:
         if '.txt' in file:
            files.append(os.path.join(r, file))
   return files

#================================================
# def JIAlreadyCalculated (f1, f2)
#------------------------------------------------
def JIAlreadyCalculated (f1, f2):
    OUTFILES_DIR = '/scratch/mlim4/myJaccard/OUTFILES/'
    jiInfoFileList = listOfTextFiles(OUTFILES_DIR)
    pattern1 = "('" + f1 + "', '"  + f2 + "')"
    pattern2 = "('" + f2 + "', '"  + f1 + "')"

    for f in jiInfoFileList :
      with open(f, 'r') as infofile:
         data= infofile.read()
```

```
          if data.find(pattern1) >= 0 or data.find(pattern2) >= 0 :
             return True

    print(pattern1 + ' is not calculated')
    print(pattern2 + ' is not calculated')
    return False

#================================================
# def getSize(fileobject):
#------------------------------------------------
def getSize(fileobject):
    fileobject.seek(0,2) # move the cursor to the end of the file
    size = fileobject.tell()
    return size

#================================================
# def printTimeWithInfoString(infoString) :
#------------------------------------------------
def printTimeWithInfoString(infoString) :
    timeForDisplay = time.strftime("%c")
    print(timeForDisplay + ': ' + infoString)

#================================================
# def creatDirIfNotExist(dirName):
#------------------------------------------------
def creatDirIfNotExist(dirName):
   if not os.path.exists(dirName):
       os.mkdir(dirName)
```

```
        print("Directory " , dirName ,  " Created ")
    else:
        print("Directory " , dirName ,  " already exists")

#================================================
# def getJI_FromJi_Info_Files (f1, f2 , option)
#------------------------------------------------
def getJI_FromJi_Info_Files (f1, f2 , option):
  OUTFILES_DIR = '/scratch/mlim4/myJaccard/OUTFILES/'
  jiInfoFileList = listOfTextFiles(OUTFILES_DIR)
  pattern1 = "('" + f1 + "', '"  + f2 + "')"
  pattern2 = "('" + f2 + "', '"  + f1 + "')"

  if option == 'jinf' :
     jiString = "jinf="
     addIndex = 6
  elif option == 'jiws' :
     jiString = 'jiws='
     addIndex = 6
  elif option == 'cr' :
     jiString = 'comRatio'
     addIndex = 9
  elif option == 'time' :
     jiString = 'startTime'
     addIndex = 10

  found = False
  for f in jiInfoFileList :
```

```
  with open(f, 'r') as infofile:
     data= infofile.read()
     p1_index = data.find(pattern1)
     p2_index = data.find(pattern2)
     if p1_index >= 0 :
        infofile.seek(p1_index)
        jiLine = infofile.read(420)
        jiLineString = jiLine
        index = jiLine.find(jiString) + addIndex
        jiVal= jiLine[index:index+6]
        jiOptString = jiLine[index:index+26]
        found = True
        print(jiLineString)
        break
     if p2_index >= 0 :
        infofile.seek(p2_index)
        jiLine = infofile.read(420)
        jiLineString = jiLine
        index = jiLine.find(jiString) + addIndex
        jiVal= jiLine[index:index+6]
        jiOptString = jiLine[index:index+26]
        found = True
        print(jiLineString)
        break



if found :
   if option == 'cr' or option == 'time' :
```

```
      return jiOptString
   else :
      return jiVal
else :
   return "Not found"
```

# Bibliography

# Bibliography


[1] (2017) Regional computer forensics laboratory annual report for fiscal year 2017. [Online]. Available: https://www.rcfl.gov/file-repository/09-rcfl-annual-2017-190130-print-1.pdf/view

[2] S. Bunting and W. Wei, *EnCase Computer Forensics: The Official EnCE: EnCase Certified Examiner Study Guide (1st ed.). Indianapolis, IN:.* Indianapolis, IN: Wiley Publishing, 2006.

[3] F. Breitinger, B. Guttman, M. McCarrin, V. Roussev, and D. White, "Approximate matching: Definition and terminology," *(Special Publication 800-168). National Institute of Standards and Technologies*, 2014. [Online]. Available: http://dx.doi.org/10.6028/NIST.SP.800-168

[4] S. Garfinkel, V. Roussev, A. Nelson, and D. White, "Using purpose-built functions and block hashes to enable small block and sub-file forensics," *Digital Investigation*, vol. 7, pp. S13–S23, Aug. 2010.

[5] V. Roussev, "Data fingerprinting with similarity digests," in *In IFIP Int. Conf. on Digital Forensics*. Berlin, Heidelberg: Springer, 2010, pp. 207–226.

[6] J. Young, K. Foster, S. Garfinkel, and K. Fairbanks, "Distinct sector hashes for target file detection," *Computer*, vol. 45, pp. 28–35, Dec. 2012.

[7] O. Rodeh, J. Bacik, and C. Mason, "Btrfs: The linux b-tree filesystem," *ACM Transactions on Storage(TOS)*, p. 9(3):9, 2013.

[8] M. Cohen, S. Garfinkel, and B. Schatz, "Extending the advanced forensic format to accommodate multiple data sources, logical evidence, arbitrary information and forensic workflow," *Digital Investigation*, vol. 6, p. S57–S68, 2009.

[9] M. Cohen and B. Schatz, "Hash based disk imaging using aff4," *Digital Investigation*, vol. 7, pp. S121–S128, 2010.

[10] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals." *In Soviet physics doklady*, vol. 10(4), pp. 707–710, February 1966.

[11] R. Real and J. Vargas, "The probability basis of jaccard index of similarity," *Systematic Biology*, vol. 45, pp. 380–385, Sep. 1996.

[12] M. Ružička, "Anwendung mathematisch-statistischer methoden in der geobotanik," *(Synthetische Bearbeitung von Aufnahmen) Biológia, Bratislava,*, vol. 13, p. 647–661, 1958.

[13] C. E. Shannon, "A mathematical theory of communication." *The Bell System Technical Journal.*, vol. 27, no. 3, p. 379–423, July 1948.

[14] G. Hall and W. Davis, "Sliding window measurement for file type identification." *IEEE information assurance workshop.*, 2006.

[15] L. Fan, P. Cao, J. Almeida, and A. Broder, "A scalable wide-area web cache sharing protocol," *In Proceeding of SIGCOMM*, 1998.

[16] L. L. Gremilion, "Designing a bloom filter for differential file access," *Communications of the ACM*, vol. 25, pp. 600–604, 1982.

[17] J. K. Mullin, "A second look at bloom filters," *Commun. ACM*, vol. 26, no. 8, pp. 570–571, 1983.

[18] M. V. Ramakrishna, "Practical performance of bloom filters and parallel free-text searching," *Communications of the ACM*, vol. 32, no. 10, pp. 1237–1239, October 1989.

[19] W. Stallings and L. Brown, *Computer Security: Principles and Practice.* Pearson, 2011.

[20] Y. S. Dandass, N. J. Necaise, and S. R. Thomas, "An empirical analysis of disk sector hashes for data carving." *Journal of Digital Forensic Practice*, vol. 2, no. 2, 2008.

[21] R. L. Rivest, "The md5 message digest algorithm." *Internet RFC 131, 1992.*, 1992. [Online]. Available: http://people.csail.mit.edu/rivest/Rivest-MD5.txt

[22] NIST-ITL, "Secure hash standard (shs)." 2002. [Online]. Available: https://csrc.nist.gov/publications/fipsfips180-4.

[23] S. Garfinkel, "Forensic feature extraction and cross-drive analysis," *The Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS)*, vol. 3, Sep. 2006.

[24] H. Parsonage. (2009) Computer forensics case assessment and triage. [Online]. Available: http://computerforensics.parsonage.co.uk/triage/ComputerForensicsCaseAssessmentAndTriageDiscu

[25] R. Walls, E. Learned-Miller, and B. Levine, "Forensic triage for mobile phones with dec0de," *Proceedings of the 20th USENIX security symposium*, 2011.

[26] S. Garfinkel, "Digital media triage with bulk data analysis and bulk_extractor," *Computers and Security*, vol. 32, pp. 56–72, 2013.

[27] R. Beverly, S. Garfinkel, and G. Cardwell, "Forensic carving of network packets and associated data structures," *Digital Investigation*, vol. 8, pp. S78–S89, 2011.

[28] S. Garfinkel, P. Farrell, V. Roussev, and G. Dinolt, "Bringing science to digital forensics with standardized forensic corpora," *digital investigation*, vol. 6, pp. S2–S11, 2009.

[29] (2014) National institute of standards and technology. national software reference library reference data set. [Online]. Available: http://www.nsrl.nist.gov

[30] B. Bloom, "Space/time tradeoffs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, pp. 422–426, 1970.

[31] V. H. G. Moia, F. Breitinger, and M. A. A. Henriques, "The impact of excluding common blocks for approximate matching." *Computers & Security*, vol. 89, February 2020.

[32] P. Penrose, W. Buchanan, and R. Macfarlane, "Fast contraband detection in large capacity disk drives," *Digital Investigation*, vol. 12(1), p. S22–S29, 2015.

[33] V. Roussev, G. G. Richard III, and L. Marziale, "Multiresolution similarity hashing." *Digital Investigation*, vol. 4, Supplement, pp. 105–113, September 2007.

[34] O. Yigit. [Online]. Available: http://www.cse.yorku.ca/∼oz/hash.html

[35] L. Marziale, G. G. Richard III, and V. Roussev, "Massive threading: Using gpus to increase the performance of digital forensics tools." *Digital Investigation*, vol. 4, Supplement, pp. 73–81, September 2007.

[36] C. Shields, O. Frieder, and M. Maloof, "A system for the proactive, continuous, and efficient collection of digital forensic evidence." *Digital Investigation*, vol. 8, Supplement, pp. S3–S13, August 2011.

[37] S. Garfinkel and M. McCarrin, "Hash-based carving: searching media for complete files and file fragments with sector hashing and hashdb," *Digital Investigation*, vol. 14, pp. S95–S105, 2015.

[38] S. Collange, M. Daumas, Y. S. Dandass, and D. Defour, "Using graphics processors for parallelizing hash-based data carving," *Proceedings of the 42nd Hawaii International Conference on System Sciences*, 2009.

[39] B. Allen. (2014) hashdb. [Online]. Available: https://github.com/simsong/hashdb.git

[40] X. Wang and H. Yu, "How to break md5 and other hash functions." *In: Cramer, R. (eds) Advances in Cryptology – EUROCRYPT 2005. EUROCRYPT 2005. Lecture Notes in Computer Science Springer, Berlin, Heidelberg*, vol. 3494, 2005.

[41] K. Woods, C. A. Lee, S. Garfinkel, D. Dittrich, A. Russell, and K. Kearton, "Creating realistic corpora for security and forensic education," *Proceedings of the Conference on Digital Forensics, Security and Law, Association of Digital Forensics, Security and Law*, p. 123, 2011.

[42] J. Taguchi, "Optimal sector sampling for drive triage," Master's thesis, Naval Postgraduate School, 2013.

[43] P. Bjelland, K. Franke, and A. Årnes, "Practical use of approximate hash based matching in digital investigations," *Digital Investigation*, vol. 11(1), p. s18–s26, 2014.

[44] V. H. G. Moia and M. A. A. Henriques, "A comparative analysis about similarity search strategies for digital forensics investigations," *In XXXV Simpósio Brasileiro de Telecomunicações e Processamento de Sinais (SBrT 2017), São Pedro, Brazil, sep*, 2017.

[45] A. Tridgell. (2002) Spamsum readme. [Online]. Available: http://samba.org/ftp/unpacked/junkcode/spamsum/README

[46] J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," *Digital Investigation*, vol. 3, pp. 91–97, 2006.

[47] L. Allison, "Dynamic programming algorithm (dpa) for editdistance." 1999. [Online]. Available: http://www.csse.monash.edu.au/∼lloyd/tildeAlgDS/Dynamic/Edit/

[48] F. Breitinger and H. Baier, "Performance issues about context-triggered piecewise hashing," *International Conference on Digital Forensics and Cyber Crime*, pp. 141–155, 2012.

[49] V. Roussev, Y. Chen, T. Bourg, and G. G. Richard, "md5bloom: Forensic filesystem hashing revisited," *Digital Investigation*, vol. 3, pp. 82–90, 2006.

[50] V. Roussev, "Building a better similarity trap with statistically improbable features," *In Proceedings of the 42nd Hawaii International Conference on System Sciences, IEEE*, 2009.

[51] F. Breitinger and H. Baier, "Similarity preserving hashing: eligible properties and a new algorithm mrsh-v2," *4th ICST conference on digital forensics and cyber crime (ICDF2C)*, 2012.

[52] J. Oliver, C. Cheng, and Y. Chen, "Tlsh - a locality sensitive hash," *4th Cybercrime and Trustworthy Computing Workshop, Sydney*, Nov. 2013. [Online]. Available: https://www.academia.edu/7833902/TLSH_-A_Locality_Sensitive_Hash

[53] J. Oliver, S. Forman, and C. Cheng, "Using randomization to attack similarity digests," *In International Conference on Applications and Techniques in Information Security, Springer*, pp. 199–210, 2014.

[54] (2017) m57-patents scenario. [Online]. Available: http://digitalcorpora.org/corpora/scenarios/m57-patents-scenario

[55] H. Chu. (2011) Lightning memory-mapped database manager. [Online]. Available: http://www.lmdb.tech/doc/

[56] (2017) hashdb 3.1.0 users manual. [Online]. Available: http://downloads.digitalcorpora.org/downloads/hashdb/hashdb_um.pdf

[57] M. Lim and J. Jones, "A digital media similarity measure for triage of digital forensic evidence." *IFIP International Conference on Digital Forensics*, pp. 111–135, January 2020.

# Curriculum Vitae

Myeong L. Lim received his Bachelor in Economics from Korea University, located in Seoul, South Korea, in 1984. He received his Master of Science in Computer Science from Pennsylvania State University in 1992. He was employed as a senior developer for 15 years at Fannie Mae. He started Techan Inc., a US government contractor, in 2020.