Content-Based Video Retrieval for Pattern Matching Video Clips

Adam Jaamour

Bachelor of Science in Computer Science with Honours
The University of Bath
May 2019

Content-Based Video Retrieval for Pattern Matching Video Clips

Submitted by: Adam Jaamour

COPYRIGHT

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed: 

Abstract

This project presents the design concepts and implementation steps of a content-based retrieval system for videos. Nowadays, unstructured data grows at exponential rates, and content-based retrieval systems can help improve the problem. Most of this unorganised data originating from social networks exists in the form of videos, which is why the task of retrieving videos from large databases is an important one.

The project was originally inspired by the famous music-matching mobile application *Shazam*, with the aim to create a similar system for matching movies in order to address the previously mentioned issue. However, an application of this scope has natural limitations due to the colossal size that a database of movies would occupy and the legal issues of employing copyrighted movies for an application. Therefore, this dissertation aims to create a prototype version of the system and later explore potential improvements to overcome these limitations.

Ultimately, a functional system was built by combining multiple methods into one pipeline and tested with a database of 50 short videos along with various videos recorded through mobile phones, resulting in correct matches reaching accuracies of 93%. To increase the realism of the tests, the recorded queries replicated videos of poor quality with shaking hand motions and inadequate framing to imitate what user-recorded videos would look like, which the system managed to cope with at the cost of some accuracy. The results were then compared to an online experiment conducted to establish ground truth, which required participants to play the role of the video matching system. To complete the pipeline, a feature-length movie was used to test how it could be condensed into one still per shot.

The code developed for this dissertation can be found online at the following URL: `https://github.com/Adamouization/Content-Based-Video-Retrieval-Code`.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to start by thanking my project supervisor, Dr. Yong-Liang Yang, for his precious help and guidance during this project, and for his lightning fast replies to my numerous emails.

I would also like to thank all my family and friends, and most importantly my mother, father, and sister, for all their love and support throughout my life and studies.

Chapter 1

Introduction

1.1 Motivation

During the past decade, the amount of data in the world has grown at exponential rates and is currently showing no signs of slowing down. According to infographics released by IBM, already 2.7 Zb[1] of data existed in the world in 2012 (Karr, 2012), enough to fill up almost 40 billion 64 Gb iPhones. This number has since then risen to 8 Zb in 2015 and is expected to reach a shocking yearly production of 35 Zb by 2020 (Deutscher, 2012) (Karr, 2012). 85% of all the world data is considered to be unstructured data (Blumberg and Atre, 2003), which is mainly made up of multimedia in the form of images and videos.

A fundamental factor in this growth is the rise of mobile devices usage and social media. Indeed, the amount of mobile-dependant users has grown at impressive rates, with now 95% of the United States citizens owning a mobile phone (Fanning, Mullen and McAuley, 2012). People tend to use their mobile devices for most day-to-day activities, with a majority of this time spent on social media services such as YouTube or Facebook. These social networks are the primary cause for the exponential growth of unstructured data mentioned earlier, with an average of over 300 hours of new video content continuously uploaded to YouTube every minute. The combination of the usage of mobile devices and the constant generation of data on social media are the main contributors in today's flood of unstructured data.

Various systems, including commercialised mobile applications, already exist to help organise this unstructured data and render it useful and accessible. These systems cover specific types of unstructured data such as music, images or CCTV security footage for example. However, none tackle the problem of unstructured data concerning long videos such as movies. Indeed, with

[1] Zettabytes. 1 Zb = 1 trillion Gb.

5,626,984 movies released as of December 2018 (IMDb, 2018), a lot of unstructured data is generated around these movies, e.g. in the form of short mobile recordings or uploaded copies circulating on the web. Therefore, the goal of this project is to tackle this problem by creating a retrieval system targeting long videos that could contribute to improving this ongoing problem.

1.2 Problem Description

Content-based video retrieval, or CBVR, is a computer vision task aimed at solving the problem of searching large databases of videos, where "content" corresponds to visual information from a video such as colours, shapes or motion that can be extracted and used to retrieve the desired video from an extensive database efficiently. The main difficulty within content-based video retrieval systems lies with the database of videos itself. As mentioned earlier, the videos in a database correspond to unstructured data, meaning that only the video data[2] and its metadata[3] are stored in the database. The computations involved in querying a database of videos using only the metadata available and not the visual information from a video itself would be too expensive and slow to compute, but most importantly, highly inefficient (Patel and Meshram, 2012). Therefore, analysing the visual content from videos is a necessity to build an adequate CBVR system. Another difficulty involves the large size of the database itself. Indeed, more complications arise from databases populated with a large number of videos, especially when these are long (e.g. feature-length films). Solutions to those issues hence require efficient and resourceful pattern matching algorithms.

Several visual search techniques exist for querying databases of unstructured data containing videos or images. These techniques can be classified based on the type of query and on the type of database used, as shown in table 1.1. The most common forms of visual search consist in querying a database of images either with an image (I2I) or with a video (V2I), as depicted in section 1.3 where similar existing systems are mentioned. Another less common variant consists of querying a database of images with a video query (V2I) (Araujo and Girod, 2018). However, this dissertation will solely focus on querying a database of videos with a video query (V2V). Because algorithms from other variants of visual search can be relevant to V2V, existing solutions for these variants will also be explored to find ways potential techniques that could be implemented with this project.

[2] Video data includes the video frames and the audio.

[3] Examples of metadata related to a video file includes captions, file name, file type, video length, file size, ...

	Database of Images	**Database of Videos**
Image Query	*I2I*	*I2V*
Video Query	*V2I*	*V2V*

Table 1.1: The four types of content-based retrieval systems involving images and videos, classified by the type of query used and by the type of database being searched.

To pattern match the query video to a video in the database, key visual elements from the query video, called *"features"*, are extracted and compared to the same extracted features from videos in the database to find similarities between them. These features can include elements such as colours, object shapes, their motion or their shapes (Patel and Meshram, 2012), as well colour distributions, colour layouts or textures (Petković, 2000), or other features unique to movies such as on-screen text or audio components, e.g. soundtracks, dialogues or sound effects. A broad spectrum of systems already exists to efficiently retrieve information from large databases, which are discussed in the section below.

1.3 Related Systems & Their Applications

Visual search technology has an endless amount of applications in fields such as education, navigation, health, security and utilities. Some visual search systems have already found their way to commercial applications. For example, YouTube's Content ID system uses visual search, and more precisely V2V, to detect copyright infringements on user-uploaded videos by automatically comparing an uploaded video with a database of protected videos, allowing them to take action against videos infringing copyrights (Lars, 2012). Another example is Google Lens[4], a mobile application that recognises the environment through a mobile device's camera in order to relay information about objects of interest. For instance, if pointing the camera at a landmark, information such as historical facts or opening hours about that location will be displayed on the screen, or if pointing it at a WiFi router's label, the mobile device will automatically connect to the network (Villas-Boas, 2017). Other applications include A9's Amazon Flow[5] that uses deep-learning based computer vision to power Amazon's search services, as well as Shazam[6], a mobile application that matches short user-recorded sounds to a piece of music. Shazam is also the main inspiration behind this project, with the initial goal of creating a similar system to match movies.

[4]Google Lens: `https://lens.google.com/`
[5]A9 Visual Search: `https://www.a9.com/what-we-do/visual-search.html`
[6]Shazam: `https://www.shazam.com/gb/company`

Aside from the commercialised systems stated in the previous paragraph, many visual search applications are yet to be implemented for large-scale use. For example, a system allowing companies to detect all appearances of their logos in live television broadcasts, or a system enabling students to find a section of a recorded lecture by using a lecture slide as an input query are possibilities that are already being explored (Araujo and Girod, 2018). Recently, engaging research in visual search systems has gained momentum, notably with the annual TRECVID[7] conference (Awad, Butt, Curtis, Lee, Fiscus, Godil, Joy, Delgado, Smeaton, Graham, Kraaij, Quénot, Magalhaes, Semedo and Blasi, 2018) hosted by the NIST[8]. This conference's goal is to organise workshops that focus on information retrieval research, with an emphasis on content-based retrieval systems for digital videos (Smeaton, Over and Kraaij, 2006).

1.4 Project Aims

The aim of the project is therefore to explore the possible implementation of a prototype CBVR algorithm with an orientation towards matching mobile-recorded queries to movies. An ideal commercial system for mobile devices would work with large databases of movies and would allow users to directly use their mobile phone to point their camera to a screen displaying a movie, which would, in turn, tell them which movie is being played along with additional information[9], as imagined in Figure 1.1. However, this is not a realistic target due to the time constraints set by this project. Thus, this project will aim to achieve the following:

- Investigate the different feature extraction and comparison methods that can be used to implement an operational CBVR system.
- Explore how the combination of these different methods can be used to design a unique functional system that could eventually work with movies.
- Develop a system that can be tested with a vast array of queries and a database.
- Evaluate the performance of the system with a reasonable testing database size and realistic queries, and eventually with feature-length movies.
- Discuss plans for future work and the limitations that such a system would have if it were to be transformed into a marketable mobile application.

[7] Text REtrieval Conference Video Retrieval Evaluation

[8] National Institute of Standards and Technology

[9] Additional information such as cast, crew, ratings, runtime and synopsis could be retrieved from IMDb (Internet Movie Database)

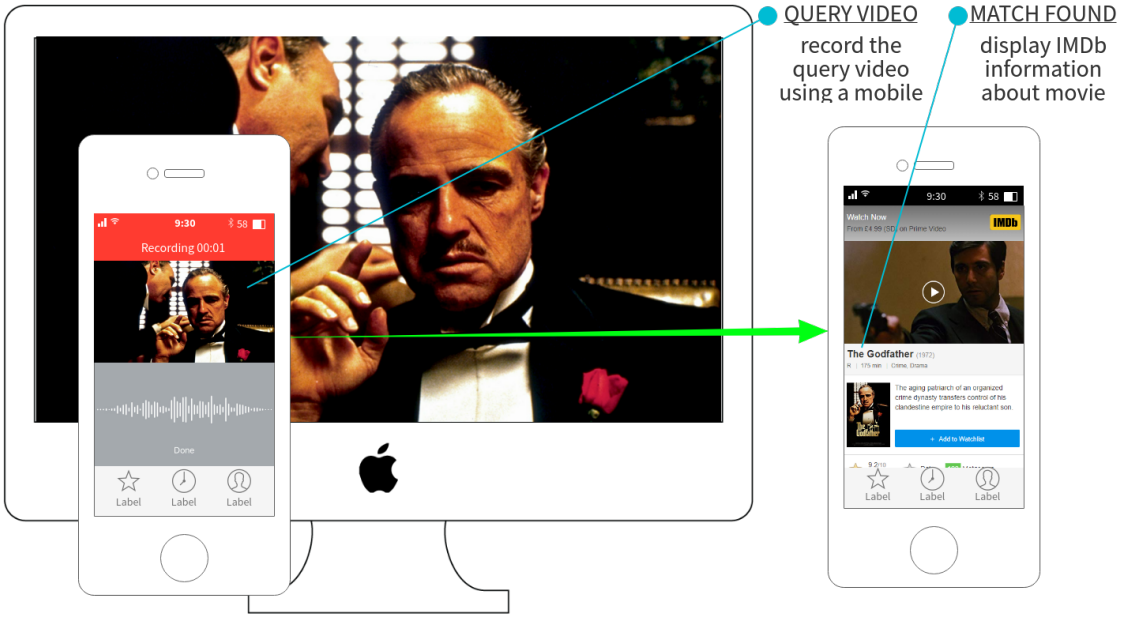

Figure 1.1: Wireframe imagining the basic high-level concept of an ideal commercial CBVR mobile application for matching movies by using mobile devices.

1.5 Report Structure

- **Introduction**
  An overview of the motivation, the problem description and the existing applications of content-based retrieval systems, followed by the goals this project aims to achieve.

- **Literature & Technology Survey**
  An extensive study of the literature surrounding the topics required to implement the system, including content-based retrieval concepts, visual content extraction and structural video representations.

- **Requirements**
  A detailed listing of the different requirements needed to design and implement the system based on the literature.

- **Design**
  A high-level exploration of potential solutions to meet the previously established requirements and formulate a final solution to implement.

- **Implementation**
  A comprehensive examination of the different steps followed to build the system from conception to a functional prototype.

- **Testing & Evaluation**
  A review of the different tests conducted and their results to assess the efficiency of the system and its quality as a whole.

- **Conclusions**
  A summary of the accomplished initial project aims, the system's limitations and plans for future work, and a final reflection on the project as a whole.

Chapter 2

Literature & Technology Survey

A full spectrum of different techniques can potentially be used to build a content-based video retrieval system. Grasping the theory behind these various techniques is primordial to establish the requirements and conceptualise the design in the next chapters. This chapter will thus study the literature surrounding three crucial areas that are necessary to understand what is required for a content-based video retrieval system targeting long videos.

The first section will address general content-based retrieval systems concepts, retracing the evolution of these visual search systems for videos and addressing the challenges posed by targeting a content-based retrieval system for videos. The second section will focus on extracting visual content from videos, starting with the different types of static features, followed by dynamic features, and ending with models used for pattern matching. Finally, the third section targets how the structure of videos can be used to segment and densely represent them in order to optimise the system.

2.1 Content-Based Retrieval System Concepts

Content-based retrieval is a type of visual search technique where large databases of either images or videos are queried to find the closest match to an input query. Although this project focuses on content-based video retrieval, also referred to as CBVR[1], image retrieval techniques (CBIR[2]) will be discussed as well due to their relevance in video retrieval.

This section will first review the different video retrieval methods and their

[1] Content-Based Video Retrieval
[2] Content-Based Image Retrieval

evolution, starting from text-based retrieval to content-based retrieval, before addressing the various challenges that exist in video retrieval, such as the difficulty caused by the temporal aspect of videos compared to images.

2.1.1 Video Retrieval Methods

Drastic advances in video capturing technology have caused significant amounts of unstructured data in the form of videos to be produced in recent years. This has lead to a high-demand to develop new efficient solutions for processing this data, with video retrieval being the answer.

Throughout the years, video retrieval has improved in parallel with the breakthroughs in video recording devices. Early video retrieval techniques used a text-based approach where the system accepted text input to search the database of videos (Lai and Yang, 2015), as seen in Figure 2.1.*a*. For example, the user would input the string query *"De Niro"*, which would return all movies in which Robert De Niro starred; or *"Coppola"* to find all movies directed by Francis Ford Coppola. Unique textual aspects of videos such as movie credits or sports scores were often analysed using Optical Character Recognition algorithms (Li and Doermann, 2002). The query text was then compared to a video file's content, such as colours, shapes, texture, luminance or objects, or the file's metadata[3], such as the video title, author, date, content description, commentaries, captions or keywords (Li and Doermann, 2002) (Feng, Cao, Bao, Bao, Zhang, Lin and Yun, 2011) (Patel and Meshram, 2012). However, these techniques were highly inefficient compared to content-based techniques as they often relied on manually noted annotations and textual descriptions to find similarities used to match the query video to a video in the database. Most importantly, they did not make use of the actual visual content that describes a video.

Content-based retrieval techniques quickly replaced text-based retrieval techniques towards the end of the 20th century by making use of the visual content to compute similarities between videos (Lai and Yang, 2015). Instead of accepting a text string, the system accepts a video as input to extract its visual contents, as shown in Figure 2.1.*b*. According to Petković et al. (Petković, 2000), this visual content can be broken down into three different categories:

- *Raw data*, which corresponds to individual raw video frames and the video file's attributes such as the frame rate per seconds, the number of bits per pixel or the colour model used.

[3]The data associated to a video file

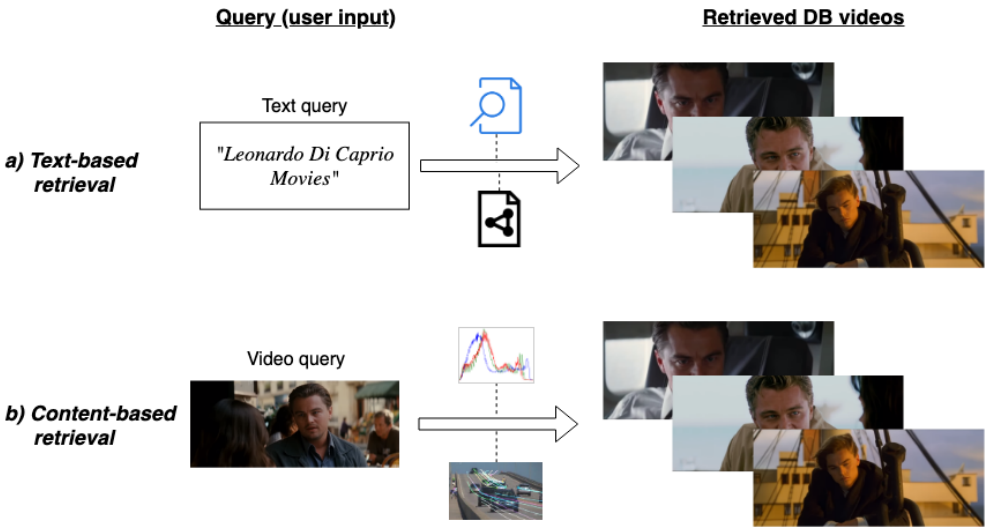

Figure 2.1: Illustrations of a text-based content-retrieval system *(a)* and a content-based video retrieval system *(b)*.

- *Low-level visual content* consists of the visual features that describe a video. This content includes colours, shapes, textures and motion. Low-level visual content can be extracted into static features such as histograms (see Section 2.2.1) or into dynamic features such as objects or motion (see Section 2.2.2) using a wide variety of existing techniques. Once the content has been extracted, it can be used to compute similarities between videos and pattern match them (Lai and Yang, 2015).

- *Semantic content* contains the high-level concepts that are present in a video. These high-level concepts can be described as objects or events using the features. To extract semantic content from a video, a grammar of rules for objects must be provided. An example of an object rule could be "if the shape is round, the colour is orange, and the object is moving, then that object is a basketball".

In comparison to raw data, low-level visual content provides the most relevant visual information that can be extracted from a video for a CBVR system. Semantic content extraction adds an additional layer of complexity compared to low-level visual content extraction as it requires domain knowledge and user interaction (Petković, 2000). Therefore, this project will focus on using low-level visual content to extract information about the video and compute the similarities between the query video and database videos. It is important to note that this project's goal differs from traditional CBVR systems where a list of videos is returned (see in Figure 2.1.*b*), as it must return a specific video that matches the most the query video. To improve the pattern match-

ing accuracy phase, raw data (e.g. audio) and metadata (e.g. captions) may be used to improve the pattern matching accuracy (Patel and Meshram, 2012).

2.1.2 Temporal Aspects of Videos

Temporal Structure of a Video

The most important difference between content-based image retrieval and video retrieval lies within the temporal aspect of the video. Naturally, the temporal aspect of a video clip stores supplementary information about the content, including dynamic low-level visual content, e.g. an object's motion, and semantic content, e.g. actions and events. A video's temporal structure can be subdivided into three units, as shown in Figure 2.2 (Araujo and Girod, 2018):

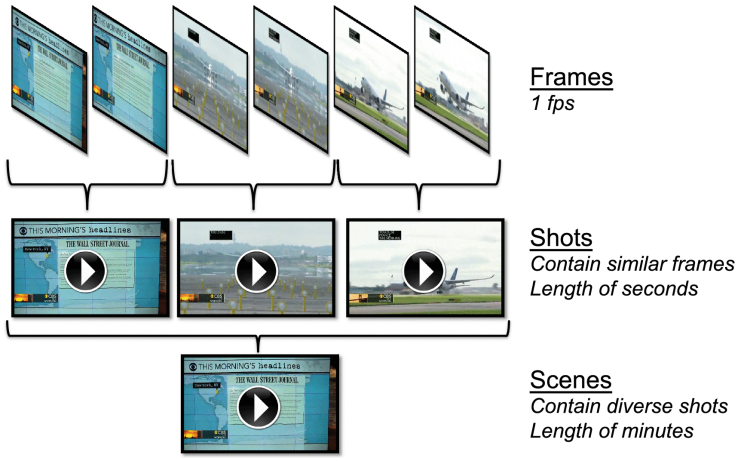

Figure 2.2: Temporal structure of videos, including the different terms used to describe temporal video units. Figure courtesy of A. Araujo et al.

- *Frames* correspond to the smallest temporal unit of a video file. A single segment of a video is referred to as a frame. Frames are also used to describe the frame rate (the frequency at which consecutive stills appear on a screen every second), e.g. "24 fps" corresponds to a video made up of 24 stills per second.
- *Shots* are grouped sequences of visually similar frames. They are usually described in seconds.
- *Scenes* are collections of shots which are related based on the action and objects present in the shot, thus giving them a semantic aspect. The length of a scene is generally calculated in minutes rather than seconds.

Because this project will explore possible solutions to create a CBVR system with feature-length movies in mind, a fourth video temporal structure category relevant to this project can be added to Araujo et al.'s initial list:

- *Movies* can be described as a large group of scenes that are used to tell a story. Movie durations commonly range from one to three hours.

Challenges of Temporality

Multiple challenges arise when dealing with CBVR systems in contrast to CBIR systems as the videos' temporal aspect adds a new dimension of complexity when extracting visual information. While the low-level visual content describing images remains mostly the same for videos, some new information that did not exist in images can be extracted, such motion. As mentioned previously, videos are made up of frames, which make up shots when a combination of similar frames are played in succession. This means that videos carry information about motion, such as the trajectory of objects. This introduces unique challenges to the algorithms used to extract motion.

Because videos are made up of numerous stills, usually around 24 frames per second (Brownlow, 1980), two consecutive frames are near-identical. The pixels describing an object in one frame will remain the same in the next frame, except for the edge pixels perpendicular to the motion's trajectory (Bradski and Kaehler, 2008).

Figure 2.3 shows six frames from Forrest Gump's famous running shot, which lasts forty-four seconds, making up a total of 1056 frames. Each frame in the figure was captured with ten-seconds intervals, meaning 240 frames separate each still. In the first three frames, the group of people running in the background barely moves in the space of twenty seconds. The pixels that describe the group in the shot remain mostly unchanged for all of the frames between the three samples, equivalent to 480 frames, with a few additional pixels describing the group as it advances towards the camera. The same can be said about the red cap in the last two frames. Most of the pixels making up the cap in the fifth frame remain the same in the sixth frame. This example perfectly betrays the reason why analysing a video frame by frame would be extremely inefficient when it comes to a CBVR system. Due to the similarities between consecutive frames, these should be aggregated (Araujo and Girod, 2018) to describe a shot by using a selection of frames, such as taking the six frames in Figure 2.3 to describe the entire 44 seconds of video, rather than keeping the original 1056 frames to describe it.

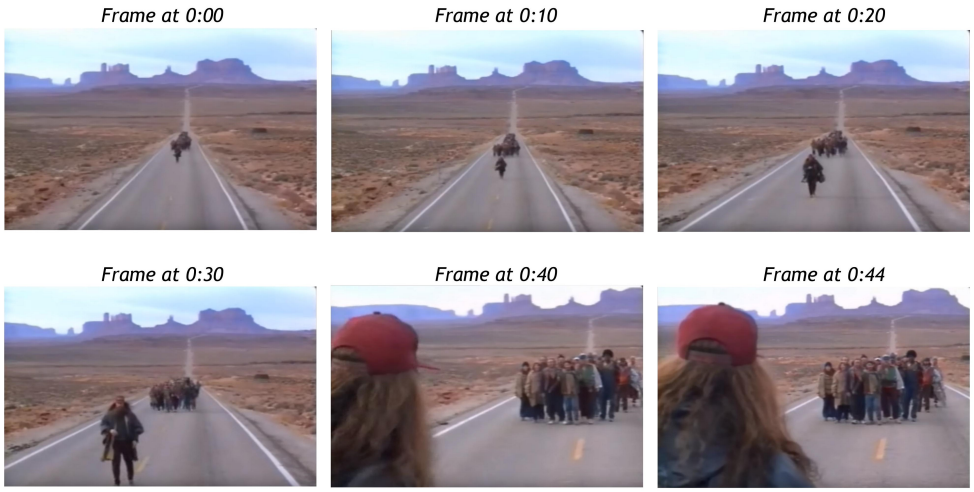

Figure 2.3: Frames from the famous running scene in Forrest Gump extracted at intervals of 10 seconds. Video frames courtesy of *"Forrest Gump long run scene"* YouTube video available online: `https://youtu.be/QgnJ8GpsBG8?t=325`.

2.2 Visual Content Extraction

Extracting the visual content from a video allows this content to be used to describe videos and compute similarities between them. This visual content is extracted from the aforementioned low-level visual content (see Section 2.1.1) (Petković, 2000) and stored in the form features, also referred to as visual descriptors. The term "features" is very broad and can be used to describe many different visual aspects in an image or a video, ranging from colours, shapes and textures to points, edges, objects and motion.

These features can be divided into two categories: static features and dynamic features (Petković, 2000). This section will first survey examples of static visual descriptors and methods to extract them from videos, and will then focus on examples and methods of extracting dynamic visual descriptors from videos, before ending on a brief overview of efficient learning models to pattern match images and videos using these features.

2.2.1 Static Features

Methods to extract static features operate on stills, which can correspond to individual video frames, thumbnails or keyframes (see Section 2.3). This means that traditional image techniques can be applied to those stills (Hu, Xie, Li, Zeng and Maybank, 2011). They are organised in three different categories: colour-based features, texture-based features and shape-based features, which can be visualised in Figure 2.4.
t

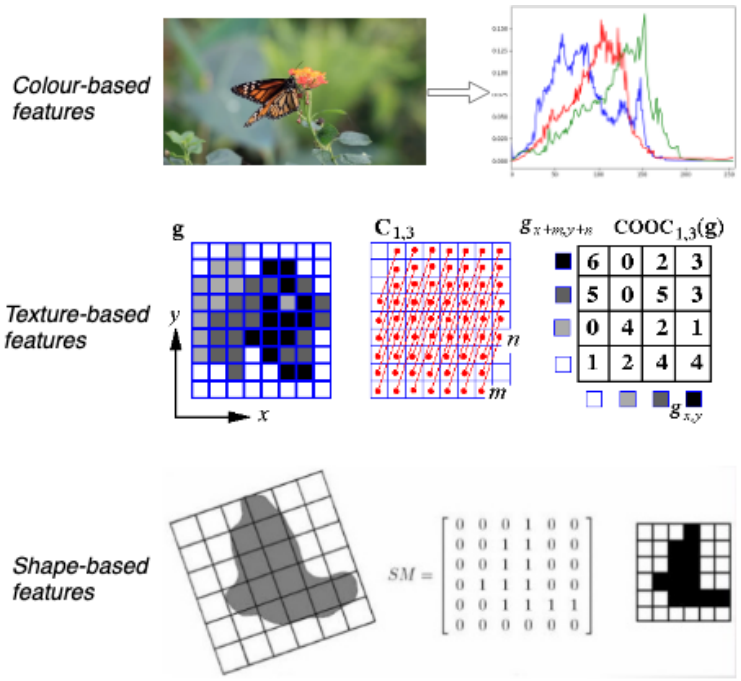

Figure 2.4: Example of static features, including colour-based features (with an RGB colour histograms of 256 bins for colour-based features), texture-based features (with an co-occurrence matrix showing the frequency of pairs of pixels) and shape-based features (with a shape matrix descriptor).

Colour-based Features

The primary colour-based features model are colour histograms. In general, a histogram consists of counts of some underlying data that is organised into predetermined bins to a statistical representation of the distribution of that data. Figure 2.5 depicts an example of a histogram where a collection of points is organised into specific pre-defined bins based on their location relative to a vertical grid.

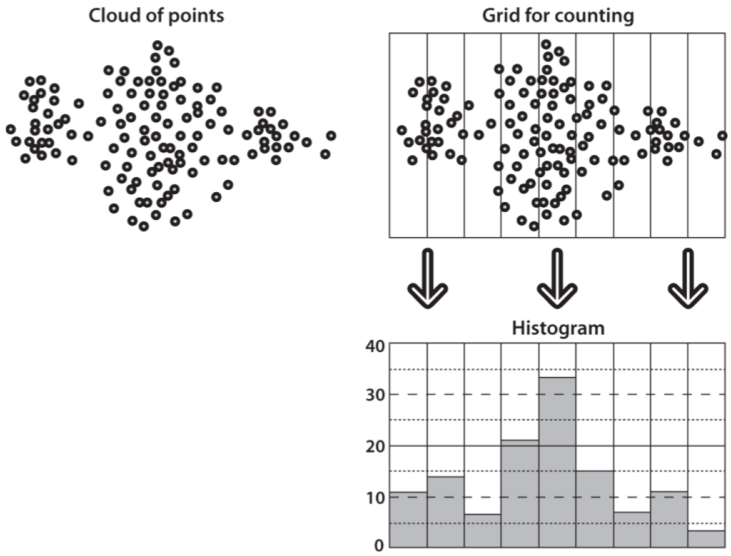

Figure 2.5: Example of a histogram counting the location of points relative to a vertical grid. Image courtesy of Bradski and Kaehler.

In the case of a colour histogram, the underlying data that the histogram is representing is the distribution of the colour pixels throughout an image or a video frame. Different kinds of colour histograms exist as they depend on the chosen colour space, which includes greyscale[4], RGB[5], HSV[6] or HSL[7] colour spaces to name a few. These vary based on the applications of the colour histograms.

Typically, for an RGB colour histogram, 256 bins are used to accurately represent all the possible values that the pixels can take (ranging from 0 to 255) for each of the three RGB channels, which are then plotted as three individual graphs. Choosing the right range for the histogram's bins is crucial to represent the distribution efficiently. If the range of pixels that defines the bins is too wide (there are less overall bins), then the histogram's distribution would be too coarse-grained, and the general structure of the histogram would be lost, as pointed out by the left-hand side of Figure 2.6. On the other hand, if the bins' range is too narrow (there are more overall bins), then the histogram's distribution would not be represented accurately, and there would be many spiky cells, as betrayed in the right-hand side of Figure 2.6 (Bradski and Kaehler, 2008).

[4]Black & White, single channel
[5]Red Green Blue, three channels
[6]Hue Saturation Value
[7]Hue Saturation Light

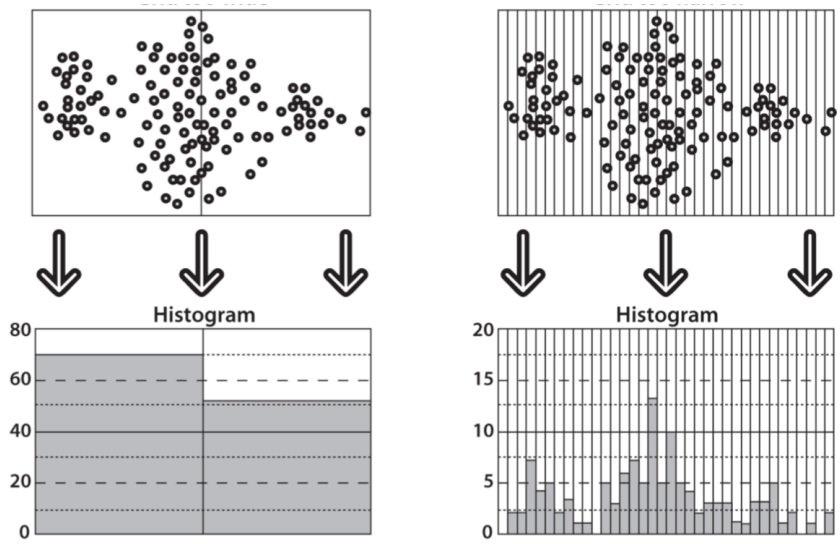

Figure 2.6: If the range of the bins is too large, then the distribution is coarse (left). If the range of the bins is too small, then the distribution is not accurately represented and spikes cells appear (right). Image courtesy of Bradski and Kaehler.

One of the inaccuracies with colour histograms lies within the scope of the distribution. If the histogram represents the global distribution of all the pixels in the still, then two images might have very similar histograms (Petković, 2000). For example, a histogram containing 60% white pixels and 40% blue pixels could either describe both a blue sky with white clouds or a snowy landscape with a blue sky. Despite both histograms being useful colour-based features, the actual result will still be poor when it used for matching the histograms. A solution consists in segmenting the still into multiple local images, and extract the local colour-based features for each segment. For example, the still could be partitioned into a 5x5 grid, and a colour histogram could then be computed for each grid (Yan and Hauptmann, 2007). This would enable colour-based features to represent specific regions of the still rather than globally describing an image. However, the same problem mentioned earlier could occur if the frame is segmented into too many regions, causing the overall histogram to be coarse.

Other types of colour-based features can be extracted from images and videos such as colour moments and colour correlograms (Huang, Kumar, Mitra, Zhu and Zabih, 1997). However, colour-based features have their limitations as they cannot describe textures and shapes, thus rendering them inefficient in certain applications (Hu et al., 2011).

Texture-based Features

Texture-based features are another type of static features, often used to detect patterns in images. Textures in images correspond to similar patterns over an area. Based on this definition, an image can be considered as a mosaic of similar patterns. Therefore, texture-based features aim to outline the areas of an image with similar patterns, which can then be used to query a database and retrieve content with similar patterns (Manjunath and Ma, 1996). An example of image areas with similar patterns that could potentially be extracted as texture-based features can be found in Figure 2.7.

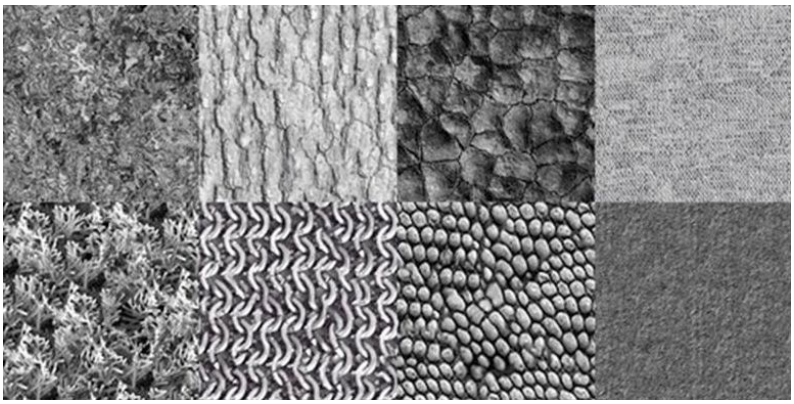

Figure 2.7: Examples of potential areas with similar patterns that could be extracted as texture features.

Texture-based features are often used in parallel with colour-based features. The aforesaid problem from Section 2.2.1, where two different objects might share a similar histogram, can be solved by using texture-based features to differentiate them, e.g. green tree leaves and green grass. These can be discerned by using a variety of features such as Tamura features, which extract information including coarseness, contrast, directionality, linelikeness, regularity and roughness of the objects (Amir, Berg, Chang, Hsu, Iyengar, Lin, Naphade, Natsev, Neti, Nock et al., 2003) or Gabor filters, which filter the frequency content in specific directions to detect similar patterns in localised regions (Manjunath and Ma, 1996).

Shape-based Features

Shape-based features are used to describe the overall shape of objects present in the image. The most common approach consists in computing an Edge Histogram Descriptor, which consists in detecting the edges present in the image using edge detectors and then plot their spatial distribution in a histogram by counting the number of pixels for each edge (Hauptmann, Baron, Chen, Christel, Duygulu, Huang, Jin, Lin, Ng and Moraveji, 2004). Edges can be detected

by applying Sobel operators through convolution, which are filters that detect horizontal and vertical edges, or by using more advanced algorithms such as the Canny edge detector.

These shape-based features have many desired properties, such as identifiability, translation/rotation/scale invariance and noise resistance, but are thus harder to extract and require more computing power than colour-based features and texture-based features (Park, 2011).

2.2.2 Dynamic Features

In contrast to static features, which can be extracted from individual video frames, dynamic features require the continuity between consecutive frames to extract relevant visual descriptors, making use of the temporal aspect of the video mentioned in section 2.1.2). These features, which both rely on features that can be tracked across frames, can be divided into two subcategories: object features and motion features. However, before reviewing the techniques used to extract these features, it is crucial to specify what defines a good visual feature that can be tracked.

Points of Interest

Figure 2.8 shows an image with coloured windows used to make the difference between poor and good potential features that could be used for object and motion features:

- *Flat surfaces* are portrayed in blue in Figure 2.8. These blue windows are spread over large areas of the image, meaning it is difficult to find their specific location. Moving the blue window along the image in any direction will result in the same visual content being represented in the window. Therefore, these flat regions are the worst structures as they do not contain any useful information.
- *Edges* are characterised in green in Figure 2.8. These are more informative than flat surfaces as they can be more accurately localised, but pinpointing an exact location is still hard as the patch can be moved in the direction parallel to the edge. Moving the green window along the edge will again result in the same visual content being represented in the window. Edges are efficient to detect object boundaries, but not for tracking specific points.
- *Corner* are characterised in red in Figure 2.8. These are the most descriptive points as they are often unique and can be precisely located in an image. Moving the red window in any direction will cause it to look different. Corners are therefore the ideal candidate for features used in object matching and tracking.

Figure 2.8: An image of the Palace of Monaco with coloured windows representing poor features in blue (flat), edges in green, and good features in red (corners).

Once expressive and unique descriptors like corners are detected, they can be used to extract object features and motion features, and to compute similarities between the query video and the videos in the database. Many different algorithms exist to find robust features and interest points. For example, Sobel operators can be used to detect edges, and Shi and Tomasi algorithms and Harris operators can be used to detect corners (Bradski and Kaehler, 2008). Combinations of both can be used, by detecting edges to facilitate corner detection.

Object Features

Object features correspond to objects that are detected using the colour, texture and size of image regions. Some of the most common objects usually detected in videos are faces, as many CBVR systems use them to compute similarities between videos (Sivic, Everingham and Zisserman, 2005). However, extracting object features is time-consuming and expensive in terms of required processing power, which is why CBVR algorithms either focus on detecting specific sets of objects rather than general objects that may be present anywhere in an image; or focus on extracting static features.

Motion Features

Motion features are a unique characteristic of videos that are absent from images. All of the above-mentioned features can be extracted from images, apart from motion features, which are unique to videos. These can originate from two sources: either background camera movement or foreground object movement. Motion features can, therefore, be classified into two categories: camera-based motion features and object-based motion features. *Camera-based motion features* include movements such as camera zooms, pannings and tiltings. *Object-based motion features* are more interesting than the former since they can describe key objects in the shot, which can be further classified into statistics-based and trajectory-based features (Hu et al., 2011):

- *Statistics-based* motion features are extracted to model the local and global distributions of motion points in the video. For example, a causal Gibbs model can be used to represent the spatiotemporal distribution of local motion measurements in a shot once the trajectory-based motion features have been used to prune undesired camera motions (Fablet, Bouthemy and Perez, 2002). This type of motion features is very cheap to extract but lacks depth as they cannot represent object actions and object relationships accurately.

- *Trajectory-based* motion features are extracted by representing object trajectories in a shot. Despite the long list of algorithms that exist to obtain these features, this section will focus on differentiating sparse and dense techniques with one of the most popular methods: optical flow. Optical flow can be defined as the relative motion between the visual content in a shot and the camera (Bradski and Kaehler, 2008):

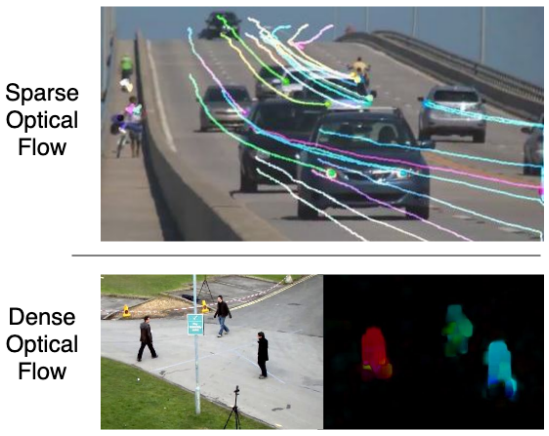

Figure 2.9: Visualisation of sparse and dense optical flow algorithms on different shots.

- In *sparse optical flow*, only a few pixels from the frames in a shot are used. Features describing objects, such as corners, are used to track an object's motion across the shot. However, these only give information about where certain parts of the object are going, which are then used to estimate the direction the overall object is moving towards. Sparse optical flows. For example, Figure 2.9 indicates the overall motion of a car by tracking only a few pixels from it.
- On the other hand, *dense optical flow* calculates the optical flow for every pixel in the frame. At each pixel of the frame, the direction and magnitude of that specific location are represented by an arrow, which can variate in direction and length (see Figure 2.10).

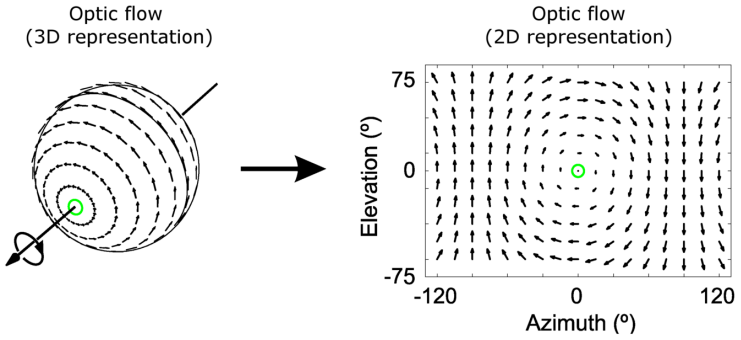

Figure 2.10: A 3D and a 2D representation of dense optical flow. Each arrow represents the direction and magnitude of a specific pixel location, which can variate in direction and length.

Object-based motion features have many applications outside of CBVR systems. One of these applications is calculating the optical flow of a shot to prune camera movements caused by unstable recording, e.g. shaking hands movement (Wang, Oneata, Verbeek and Schmid, 2016). Compared to statistics-based features, trajectory-based features can be used to describe object actions, but they rely on multiple challenging tasks to function efficiently such as accurate object tracking and automatic trajectory recording. Finally, object-based motion features can also be classified in a third category: objects' relationship-based motion features. However, these are not described in this literature review as they make use of the objects' symbolic representations, which is not relevant to this topic.

2.2.3 Models for Pattern Matching

So far, the simplest model that can be used for pattern matching has been introduced in Section 2.2.1, which is the histogram model. Distributions of

features are represented in the form of histograms and used to calculate the similarities between each other by using simple distance measuring algorithms adopting nearest neighbour approaches. More advanced and efficient models are briefly reviewed in this section, ranging from Bags-of-Visual-Words models before ending with a quick word on deep learning models.

Bag-of-Visual-Words

The idea behind the BoVW (Bag-of-Visual-Words) model originates from the BoW (Bag-of-Words) model for text document analysis. Indeed, text documents can be represented as a bag of important keywords; excluding uninformative words such as "the" or "it". The BoW of the document is built by measuring the frequency of each keyword in the document. This BoW then allows the type of document to be predicted, e.g. a document with a high frequency of words such as "racket", "serve" and "ball" is likely to be a document about tennis. The opposite can be applied as well, where the type of the document can predict the frequency of keywords.

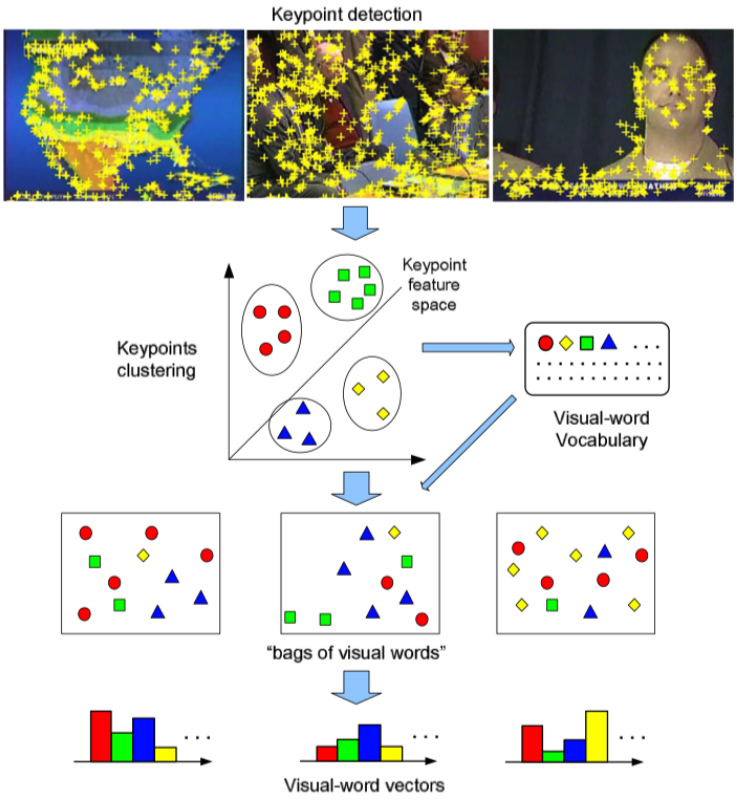

Figure 2.11: Visualisation of the generation of a BoVW used to represent images with histograms counting the occurences of each visual word. Image courtesy of Yang,Jiang, Hauptmann and Ngo (2007).

This concept of BoW can be applied to images in the form of BoVW. Rather than describing actual words, the "visual words" correspond to features extracted from the image (the same features mentioned in Section 2.2.1 and Section 2.2.2), which are then clustered using algorithms such as k-means clustering or mean-shift. Once the features are grouped, centroids (the middle of a cluster) can be extracted to represent a visual word (Yang, Jiang, Hauptmann and Ngo, 2007), as shown in Figure 2.11.

With the BoVW now built, the image can be represented by a histogram counting the number of occurrences of each code word in the image, in a similar fashion that the frequency of keywords is counted for a text document. However, a large number of visual words need to be defined to avoid losing information when representing the image with the histogram; if there are too few visual words, then the histogram will be too coarse. With a dictionary of visual words, a classifier can be trained by ingesting multiple images, or video frames, and generating the histogram of occurrences of visual words in each image. Once this step is done, a new training image can be used as input to generate its histogram, which can then be compared to each previously trained image to find the closest match (Wang et al., 2016).

Deep Learning Systems

The conventional models stated earlier are naturally limited by their inability to process raw data. Indeed, raw data (e.g. the pixels in an image) cannot be directly processed by machine learning systems, including the aforementioned ones. Sophisticated feature extractors need to be designed to transform the raw data into suitable feature vector representations. Only then can learning systems such as classifiers ingest the data pattern match it (LeCun, Bengio and Hinton, 2015).

Deep learning models are much more efficient as they are based on representation learning methods, which allows raw data to be directly used. Representation learning methods enable deep learning systems to automatically discover and learn the features through the use of multiple levels of representation that transform the data as it goes down the layers, starting with the raw data that is transformed into more and more abstract representations. For example, a deep learning system using an image (a matrix of pixels) as input would perhaps detect edge features in the first layer, followed by detecting the arrangement of these edges in the second layer, before combining them to assemble motifs in the third layer and eventually detect objects in deeper layers. All of these layers that learn how to extract features are not influenced by humans, they are learned based on the data (LeCun et al., 2015).

These deep learning models, with the most popular ones being neural networks, have much potential. Due to their nature that requires little manual engineering, they work very efficiently when dealing with large amounts of data, which fits perfectly with content-based retrieval systems.

2.3 Structural Video Representations

The database of videos can be pre-processed to optimise the visual content extraction and pattern matching phases. This section covers different techniques to represent long videos in dense formats, with an emphasis on feature-length movies. As stated in Section 2.1.2 on the temporal aspects of videos, movies can be defined as a logically ordered collection of scenes, which contain multiple shots, all made up of individual frames. Shots are therefore the logical fundamental unit to organise and segment a video into (Hu et al., 2011). Preprocessing the database of movies by segmenting it into a list of shots and representing each shot with a single keyframe is an effective solution that will exponentially improve a CBVR system's efficiency. However, a limit must be set on the amount of data that is segmented to balance the efficiency and the accuracy of the CBVR system.

2.3.1 Temporal Movie Segmentation

The frames that make up a shot usually show strong content correlation. This means that features extracted from one frame will be remarkably similar in another frame from the same shot. Detecting shot boundaries would allow the movie to be segmented and organised by shots, as depicted in Figure 2.12.

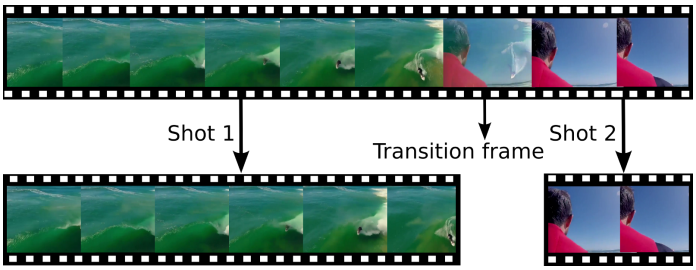

Figure 2.12: Shot boundary detection example of a video scene made up of two shots with a gradual transition between the two shots. Figure courtesy of Michael Gygli available online at: `https://medium.com/gifs-ai/ridiculously-fast-shot-boundary-detection-with-fully-convolutional-neural-networks-da9d8c73e86c`

These boundaries are defined by the type of transition between two different shots, which can either be defined as a quick cut when the transition is direct, or as gradual when it is a dissolve or a fade in/out transition that runs

over multiple frames (Yuan, Wang, Xiao, Zheng, Li, Lin and Zhang, 2007), as shown in Figure 2.13.

Figure 2.13: . Visual examples of a quick cut, a dissolve cut and a fading cut. Frames courtesy of Koprinska & Carrato (2001) "Temporal video segmentation: A survey" available online at: `https://www.sciencedirect.com/science/article/pii/S0923596500000114`

The goal of a shot boundary detection algorithm is to detect the discontinuity between consecutive frames. Also know as temporal video segmentation, this process requires three steps:

1. Feature extraction
2. Similarity measurements
3. Shot boundary detection

Feature Extraction

The first step in video segmentation consists in extracting features. These features include all the diverse types mentioned in Section 2.2, ranging from static features such as colour-based features (e.g. colour histograms) (Hoi, Wong and Lyu, 2006) to dynamic features such as corner points and SIFT[8]. Although colour histograms are simple to compute and work well with most shots as long as there is at most minor camera movement, they are inefficient when the shots contain major camera movements (Hu et al., 2011). For example, if in the shot the camera moves from the inside a house towards the outside by going through the window, the frames retrieved from the beginning of the shot will be extremely different to frames from the end of the shot.

[8]Scale Invariant Feature Transform

Dynamic features are therefore more efficient for shot boundary detection than histograms due to their robustness. On the one hand, edge features are more vigorous than histograms when dealing with significant camera movement and can handle changes in luminosity. On the other hand, corner and motion features can additionally handle camera motion and the impact of objects in the shot such rapid motion, e.g. people walking in front of the camera. However, these dynamic features are more complicated to extract and do not always outperform simple features like colour histograms. Due to their simplicity, colour histograms remain the most common feature extraction technique for shot boundary detection (Yuan et al., 2007).

Similarity Measurements

The second step in video segmentation is to use these extracted features to compute the similarities between frames. Many metrics exist to compute the similarities between two the extracted features. The most basic method consists in computing the Euclidean distance or the absolute distance $g(n, n+k)$ between a pair of frames (Janwe and Bhoyar, 2013, p.476), as shown in Equation 2.1, where n and $n+k$ represent the two frames being compared, and $I_n(x, y)$ the intensity level of frame n at pixels locations (x, y):

$$g(n, n+k) = \sum_{x,y} |I_n(x, y) - I_{n+k}(x, y)| \tag{2.1}$$

More advanced distance metrics can be used based on the type of feature that was extracted. For instance, intersection and chi-square distances can be used to compute the distance between the histograms of two frames (Camara-Chavez, Precioso, Cord, Phillip-Foliguet and Araujo, 2007, p.197). During the 2006 TRECVID conference, *Hoi et al.*, used grey scale histograms for their extracted features and calculated the colour differences between frames using the EMD[9]. EMD is used to calculate the distance between two probability distributions, which are represented by the two histograms that represent each frame in (Hoi et al., 2006, p.2)'s case. These similarities can be measured using two techniques: pair-wise similarities and window similarities.

The first technique rests on using pairs of frames to compute the similarities between them. The similarities between two consecutive frames I_1 and I_2 are compared, before comparing the pair of frames I_2 and I_3, and so forth until frames I_{n-1} and I_n are compared. This straightforward technique precisely detects rough changes between the visual content in a shot (e.g. quick cuts between two shots) but is more sensitive to intensity changes caused by noise

[9]Earth Mover's Distance

and disturbances such as camera motion and objects, leading to a high level of wrong detections (Janwe and Bhoyar, 2013).

The second technique is the window-based similarity measure that calculates the similarities between multiple frames specified within a range (Cernekova, Pitas and Nikou, 2006). This helps counter the effects of noise and disturbances captured by the pair-based approach but is more expensive to compute, therefore less used (Hu et al., 2011).

Detection

The final step in segmenting a video is the detection of the frames to cut, which is achieved by using the measured similarities between the frames. Two methods exist for this step: a threshold-based approach and a statistical learning-based approach.

The *threshold-based approach* compares the measured similarities between a pair of frames with a threshold. Whenever the similarity measure is smaller than the threshold, a shot boundary is detected, and the shot can be cut. Figure 2.14 betrays an example of a threshold-based approach for different types of transitions, with a correct detection (A), a missed detection (B) and a misinterpreted detection (C).

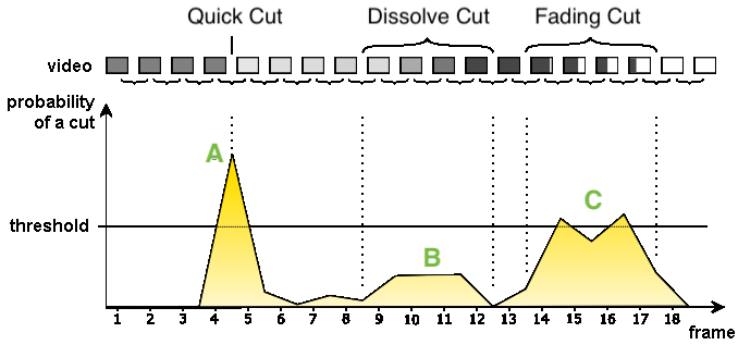

Figure 2.14: Threshold-based approach for shot boundary detection. In this example, the shot boundary is detected at A, it is missed at B, and it is misinterpreted at C as two cuts rather than a single one.

Either global thresholds, adaptive thresholds, or combinations of both can be used (Cernekova et al., 2006) (Hu et al., 2011):

- *Global thresholds* are set empirically and predefined for the entire video. They are therefore inefficient when trying to detect local variations as these are not integrated into the initial global threshold calculations.

- *Adaptive thresholds* are local thresholds that are continuously calculated for windows of frames. As the window slides across the video, the threshold is re-estimated accordingly to account for local variations. These are more accurate than global thresholds but are harder to estimate and require some knowledge about the video to set variables such as the window's size.

- *Combination of global and adaptive thresholds* are adaptive thresholds that take into account the values of global thresholds when estimating the frames within the window. Different thresholds can be determined for different types of transitions such as cut and dissolve transitions. However, these add increased complexity to the threshold estimation.

The *statistical learning-based approach* is a two-class classification task where each frame is classified either as a "shot change" or as a "no shot change" based on the extracted features and the similarity measurements between those features. In more complex systems, additional classes can be added by adding a classification of quick and gradual transitions for example. Two types of classifiers can be used for this approach:

- *Supervised learning-based classifiers*, such as the Support Vector Machine (Camara-Chavez et al., 2007) and Adaboost (Zhao and Cai, 2006, p.617). These supervised learning approaches have multiple advantages compared to threshold-based approaches as they do not require thresholds to be set and many different types of features can be combined in the feature extraction step to increase the classifier's accuracy. However, this approach requires more expertise as the classifier relies on a well-trained data set to remain accurate (Hu et al., 2011).

- *Unsupervised learning-based classifiers* are divided into frame similarity-based and frame-based algorithms. *Frame similarity-based classifiers* groups the similarity measurements between a pair of frames into two clusters. The first cluster holds the similarity measurements with low values, which correspond to the "shot change" class, while the second cluster holds the similarity measurements with high values, which correspond to the "no shot change" class. K-Means and Fuzzy K-Means clustering algorithms can be used for frame similarity-based classifiers (Lo and Wang, 2001). Alternatively, *frame-based classifiers* consider each shot as a cluster of frames with similar features. Clustering ensembles can be used to classify frames into their respective shots (Chang, Lee, Hong and Archibald, 2007). This approach is advantageous over supervised learning-based classifiers as no prior training is required for, but the logical temporal order of the shots in scenes is not preserved.

All the aforementioned techniques in this section that are used in the three steps required in the process of segmenting a video can be applied to feature-

length movies but require some improvements to work efficiently with these large videos. For instance, (Hanjalic, Lagendijk and Biemond, 1999) suggests considering the semantic aspects of movies, since people relate to the story, such as a marking dialogue, when they remember a movie.

2.3.2 Key Frame and Thumbnail Extraction

Frames from the same shot usually describe the same visual content, meaning there is much redundancy amongst the frames that make up a shot. It is, therefore, more efficient to deal with a single frame that represents the entire shot or the entire video rather than dealing with an entire video.

Key Frames

Once shot boundaries have been detected and the video has been divided into a series of shots, a single frame, named the "keyframe", can be selected to represent the entire shot. (Heo, Lee and Jung, 2016) suggests only considering keyframes in videos to operate on. These key frames would be used for all the sections mentioned in this Literature Survey, including the visual content extraction phases, the pattern matching phases, and database videos pre-processing phase. A wide variety of approaches exist to extract a key frame from a video.

The most simple and popular approach is sequential comparison between frames where frames are sequentially compared until one frame very different to the previous one is found, at which point that frame is set as a new keyframe (Heo et al., 2016). More advanced algorithms can be used such as the global comparison between frames and reference frames, but rely on the same feature comparison concept. Frames are compared using the features mentioned in Section 2.3.1.1 e.g. histograms, and similarities are calculated using the same distance functions mentioned in Section 2.3.1.2 e.g. chi-square distance (Hu et al., 2011).

To illustrate the advantage of using keyframes, a three-seconds long shot recorded at 30 fps[10] of a ball rolling on the ground can be used (90 frames in total). Analysing all the frames individually as stills would be highly inefficient. However, selecting keyframes to work on, as depicted in Figure 2.15 where a single frame is chosen for each second, would mean that three keyframes can be used for feature extraction and pattern matching instead of using all of the 90 frames that make up the video.

[10]Frames Per Second

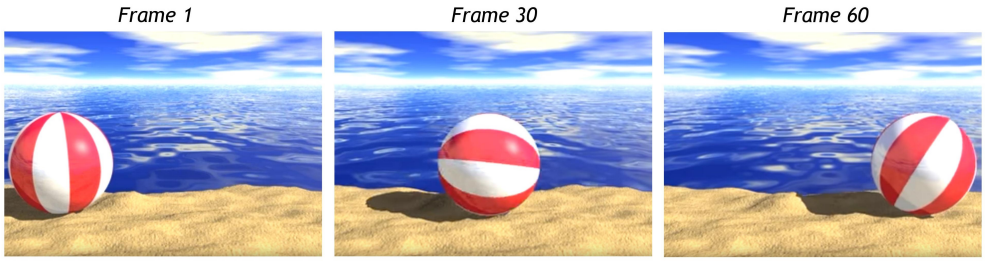

Figure 2.15: Example of frames to sample for low-visual content analysis. The first frame for each second (one frame every thirty seconds) is retrieved for a 30 fps 3-second video of a ball rolling from the left-hand side of the screen to the right-hand side. Video frames courtesy of *"How to Animate a Rolling Ball"* YouTube video available online: `https://youtu.be/cgbLAreElNI?t=130`.

Thumbnails for Initial Shortlisting

Thumbnails can be generated for each video in a database, which are stored as additional data along with the original video file (Okabe, Dobashi and Anjyo, 2018). In a CBVR system, the thumbnails for the query video and the database videos would be generated using the same algorithm in order to create similar outputs. This technique can be used in parallel to (Araujo and Girod, 2018), who states that an initial shortlist of potentially matching videos can be generated before the main pattern matching phase. This shortlist can be created by computing the similarities between the query video's thumbnail and the thumbnails of the database videos.

Many advantages can be gained from this small initial step which could have a meaningful impact on the system's overall speed and efficiency performance. Indeed, database videos that share no similarities to the query video will not be considered at all during the main pattern matching phase, e.g. if the query video corresponds to a colourful sunset, then database videos of cloudy environments will be immediately filtered out as it is unlikely that they will match with the query video in the main pattern matching phase. This step diminishes the number of videos to compare to the query.

This initial step of generating thumbnails for the query video and each video in the database is extremely speedy as it only uses a single frame that describes the entire video. Therefore the entire process will not be noticeably slowed down. However, this concept could only be applied to relatively small databases where the videos contain a single scene with multiple similar shots. Indeed, shots can be characterised by a single still as they are made up of frames with similar visual content, whereas scenes can be made up of shots that change a lot and describe different visual content. Using thumbnails to

shortlist a database of feature-length movies would greatly diminish the accuracy of the system due to the high number of scenes that make up a movie that cannot be resumed to a single thumbnail. The database videos' thumbnails will have already been generated during the database's pre-processing phase, which only occurs a single time. A possible solution could be to create a thumbnail for each shot in a scene and store each thumbnail in a list. For example, if a scene contains six different shots, then six thumbnails will be generated to describe that scene.

2.4 Chapter Summary

The literature in this chapter covers the main theoretical and technological aspects required to understand the nature of content-based retrieval systems for videos, how visual content is extracted from these videos and used, and how the temporal structure of videos can be employed for optimisation. With the knowledge gained from the literature coupled with the problems identified in Section 1.2, a set of requirements of what the system needs to achieve can be formulated in the next chapter.

Chapter 3

Requirements

This chapter establishes the requirements of the system. These requirements were devised in response to the problems mentioned in Section 1.2 of Chapter 1 and builds upon the information provided by the Literature and Technology Survey in Chapter 2.

The following requirements are divided between non-functional and functional sections, which are further split into more specific sections. They are prioritised using the words *must* (Priority 1), *should* (Priority 2) and *may* (Priority 3); where *must* indicates a crucial requirement to the system, *should* an optimal requirement and *may* a beneficial requirement.

3.1 Functional Requirements

3.1.1 System Requirements

To establish system requirements, the system can be broken down into three phases. Based on the models described in the Literature & Technology Survey in Section 2.2.1 and Section 2.2.3, and taking into consideration the time constraints and scope of the project, a training/testing approach is adopted. For each database video, features are extracted and stored in the first phase before being compared to a query's features in the second phase. A third phase can be added based on the video processing techniques stated in Section 2.3.

Off-line Feature Extraction Phase

These requirements concern the off-line feature extraction phase where the system learns by ingesting static unsupervised data corresponding to the database videos. In this phase, features are extracted from each video to generate feature vectors which are stored in files for the next phase.

F1 **The system must run this phase a single time without requiring to be executed again for the next phase**.
The features stored in files must be re-used in the next phase rather than being generated every time. The system **must** re-ingest the data whenever a change is carried out in the database, e.g. a video has been added, or a video has been removed.
Priority 1.

F2 **Simple features that are proven to work efficiently with videos must be chosen to be extracted.**
Due to the nature of the project, a line must be drawn between the complexity of the feature being extracted to represent the videos and the efficiency of working with such a feature. For example, if the feature is very complicated to extract and use, but does not offer many more advantages than a simpler feature to extract and use, then the more straightforward feature should be used.
Priority 1.

F3 **The system must store the extracted features it used for learning in files for future re-use in the on-line retrieval phase.**
Files can be saved either in plain text "*.txt*" file format to be human-readable, or in other file formats such as binary "*.dat*" files.
Priority 1.

F4 **The files containing the training data must be able to be quickly loaded into memory during the matching phase.**
To minimise additional time spent on I/O operations, the files should be loaded quickly using efficient functions designed for this type of operation. If necessary, an external third-party library may be used to provided functions for quicker I/O operations.
Priority 1.

F5 **The system's interface should specify that it is ingesting and extracting features from the database videos.**
It can do so by either displaying a loading spinner or by iteratively printing text specifying which video is being currently processed.
Priority 2.

On-line Retrieval Phase

These requirements concern the retrieval of the system, where the same extracted features from the previous phase are extracted from the query video and compared to the stored features from each database video by calculating the distance between these features.

F6 **The system must accept a pre-recorded video as input.**
This pre-recorded video will serve as the query video.
Priority 1.

F7 **The system must crop the recorded video to select a "region of interest".**
The system must ignore the pixels outside of the selected area of the clip when extracting its features.
Priority 1.

F8 **The system must use well-known distance metrics to calculate the distances between the query's feature vectors and the database videos' feature vectors.**
Hundreds of metrics exist, but some are more efficient for the task at hand. A good indication that a distance metric is useful for the task at hand is its availability for popular programming languages and their mainstream third-party libraries.
Priority 1.

F9 **The system must display the results once the classification has been completed.**
The retrieved video **must** be emphasised over the other videos to clearly point out that the result has been found. Additional results such as individual distances between the query and each video can be displayed as well, e.g. tables.
Priority 1.

F10 **The system must be evaluated using reliable performance measurements.**
The accuracy of the system should be calculated for different queries.
Priority 1.

F11 **The system should stabilise the query video before extracting its features.**
The stabilised video will diminish the effects of camera movement and increase the similarities between the video's extracted features and the database videos' features.
Priority 2.

F12 **The system may be able to directly record a query video.**
Processing the query video while it is being recorded would drastically improve the system's runtime in this phase rather than waiting for the entire video to be recorded and only processing it after the recording is finished.
Priority 3.

F13 **The system may find a match to the query in less than 10 seconds.**
Commercial applications usually take around 10 seconds to produce a match to the query. Therefore, this project should take roughly the same amount of time before producing an output.
Priority 3.

Database Pre-processing

These requirements concern the database pre-processing phase of the system. This phase concerns the operations to carry out the database videos to improve the off-line feature extraction and on-line retrieval phases.

F14 **Long videos, e.g. feature-length movies, must be processed for shot boundary detection to segment them into scenes.**
A single frame must be used to represent each shot of a long video. However, this is not required if the video is short enough to be a shot by itself.
Priority 1.

F15 **The same features from the off-line feature extraction and on-line retrieval phases should be used for the shot boundary detection algorithm.**
This would ensure cohesion between different phases of the program and allow code to be re-used. Priority 2.

General Requirements

F16 **The MVP[1] must at least be a Command Line Interface that accepts command line inputs to run different phases of the program and enable/disable options.**
A simple CLI is the minimum interface required to run the system with different arguments and to display results.
Priority 1.

F17 **The system must be able to run on a machine equipped with a terminal.**
Any machine running Windows, Mac or Linux distributions should be able to run the system with the correct project dependencies installed.
Priority 1.

F18 **The system should include a "README" file with instructions on how to run the system.**
The "README" file allows the system to be accessible to other users

[1] Minimum Viable Product

with the appropriate background knowledge to install the pre-required dependencies and run the system successfully.
Priority 2.

F19 **The interface may be a graphical interface.**
For example, it could be a simple HTML web page or a native UI desktop application (which will depend on the chosen programming language). This would allow for information to be more clearly displayed on the screen.
Priority 3.

3.1.2 Code Design Requirements

F20 **The system's code must be source controlled using Git and backed up on GitHub in a private repository.**
This will enable the code to be safely backed and offer a timeline of the system's development phase.
Priority 1.

F21 **The system should follow coding style guidelines set for the programming language of choice.**
This will ensure that the code will be written at the highest standard by following professional practices and will be more easily readable by industry experts.
Priority 2.

F22 **The system should use third-party libraries for complex aspects of the software to avoid rewriting complicated code that already exists and is easily accessible.**
Rewriting code that already exists could be a waste of time for a project with such as shirt time-frame. Therefore, using third-party libraries will mean more time can be spent designing and implementing high-level concepts of the system.
Priority 2.

F23 **The system may be covered with unit tests and integration tests.**
To ensure that different core aspects of the system work as expected, unit tests can be written around these sections. Additionally, multiple core aspects can be tested as a whole with integration tests.
Priority 3.

3.1.3 Data Requirements

F24 **The videos used in the database must not violate any copyright laws.**
They must have licenses allowing them to be re-used for personal and commercial purposes at no cost.
Priority 1.

F25 **The duration of the videos should range from 7 to 14 seconds.**
Working with short videos is enough to prove that the system pipeline works as intended since a considerable number of them can be used to populate the database, whereas only a handful of videos could be efficiently used if feature-length movies were used instead. 7 seconds is enough to ensure each video has at least 210 frames[2] to analyse.
Priority 2.

F26 **There should be a minimum of 50 unique videos in the database.**
Because short videos rather than long ones are used to test the system pipeline, there should be a considerable amount of them to challenge the system.
Priority 2.

F27 **The system pipeline may be tested with a few feature-length movies as database videos.**
The main goal of the project is to build the system pipeline, which can be tested with a database of many short videos rather than a database of a few long videos. To further explore the efficiency of the system, feature-length movies could be used to report the results when attempting to match them to a short query.
Priority 3.

3.2 Non-Functional Requirements

NF1 **The latest technologies must be used to build the system.**
The latest versions of programming languages and libraries must be used to ensure that the system takes advantage of up-to-date technology.
Priority 1.

NF2 **The system should not be built with external users in mind.**
It should only be run by the system's creator or users with a background in computer science who could understand how to run it using the README file provided.
Priority 2.

[2] Assuming videos with a frame rate of 30 fps.

NF3 **The interface should make the results easy to read.**
This could be achieved with the use of large fonts, colours and bold text. The text used to discern the main results from less important output should be emphasized.

NF4 **The system may run smoothly and uninterrupted.**
Priority 3.

NF5 **The system may not crash or exit unexpectedly with no error message.**
Priority 3.

3.3 Summary

This chapter has summarised the requirements needed to design and implement a successful system based on the problems formulated in Chapter 1 Section 1.2 and the Literature and Technology Survey from Chapter 2. Now that the requirements have been established, different potential solutions will be explored in the next chapter, including the benefits and disadvantages of each method, before one global solution can be chosen and implemented.

Chapter 4

Design

With the requirements necessary to build the system now formulated, potential solutions to fulfil the list of requirements from Chapter 3 for each aspect of the system can now be analysed before choosing a final solution.

Throughout this chapter, the requirements established from the previous chapter referred as "F" stand for functional requirements, while requirements referred as "NF" stand for non-functional requirements.

The first section of this chapter will focus on the programming language that will be used to build the system. Next, the second section explores different solutions for the system's pipeline, starting with the off-line feature extraction phase, followed by the on-line retrieval phase, and ending with the database pre-processing phase. Finally, the last section will focus on general design decisions such as the type of interface, the type of file to store the features in and the type of videos to populate the database with. These two sections are used to conclude the chapter with a detailed outline of the final chosen solutions for each aspect of the system, which are all used to eventually bring the system to life in the next chapter.

4.1 Programming Language

The programming language is one of the essential aspects when it comes to designing and later implementing the system as it is the medium used to transform the system from design to implementation. However, choosing between 250 different programming languages (TIOBE, 2019) can be a tricky exercise, which is why multiple views have to be evaluated before choosing a programming language. Various criteria are considered to justify the choice of the programming languages, including the availability of computer vision-related functions, the runtime speed, the readability and the personal preference.

The first element to consider when choosing a programming language for this type of project is the availability of functions for video manipulation and computer vision-related operations. Indeed, using high-level functions to, e.g. read videos or generate histograms, is primordial to avoid spending time manually implementing each of these functionalities and to instead devote more time on implementing high-level concepts to implement a functional pipeline. Some languages, such as MATLAB, allow for efficient manipulation of images/videos and offer native collections of computer vision functions without the need of using libraries. For other programming languages such as Python and Java, this is not the case. The most popular library in this field is OpenCV[1], which offers functions for real-time computer vision applications. This library supports all the main operating systems (Linux, macOS, Windows, etc.) and although it was mainly designed for C/C++, it contains bindings for the main general-purpose programming languages such as Python, C++, Java, C#, Javascript and Haskell.

Among the previously mentioned programming languages, C and C++ are the fastest languages, according to *Benchmark Games*[2], followed by Java, MATLAB and Python. However, Python can be extended with languages such as C/C++ with ease. This means that coupling Python with the OpenCV library that is initially written in C++ allows the Python code to execute as fast as the original C++ code since it is the actual OpenCV C++ code that is running in the background. Furthermore, Python has an extensive set of third-party libraries that add powerful functionalities to the language, such as NumPy[3], SciPy[4] and Matplotlib[5], which all add mathematical and scientific functions as efficient as MATLAB's native functions. Finally, of all the specified programming languages, Python is the favoured one in terms of personal preference, familiarity, and experience. Therefore, the relatively slower execution speeds of Python being outweighed when using OpenCV functions, coupled with the personal preference for Python, make Python the obvious programming language choice for this project.

For a complete review of the main pros and cons considered between when deciding between Python, C++, Java and MATLAB, refer to Appendix A.

[1] OpenCV, `https://opencv.org/`

[2] Benchmark Games: `https://benchmarksgame-team.pages.debian.net/benchmarksgame/`

[3] NumPy: `https://www.numpy.org/`

[4] SciPy: `https://www.scipy.org/scipylib/index.html`

[5] Matplotlib: `https://matplotlib.org/`

4.2 Pipeline Design Analysis

To chose a global design solution for the system, it is easier to break down the system's pipeline into different phases and review the numerous potential designs for each of these phases separately, as shown in the concise diagram in Figure 4.1:

- **Off-line Feature Extraction Phase**. This phase corresponds to the "training" phase of the system, where features are extracted from each video in the database and then stored in data files to be later used in the on-line retrieval phase.
- **On-line Retrieval Phase**. This phase is affiliated to the "testing" phase of the system, where a single video query is matched to one of the database videos by extracting the same features from the previous phase and comparing them to the stored features.
- **Database Pre-processing Phase**. This is an optional phase where the database videos are processed before the off-line feature extraction phase to improve the accuracy and speed of the database videos feature extraction.

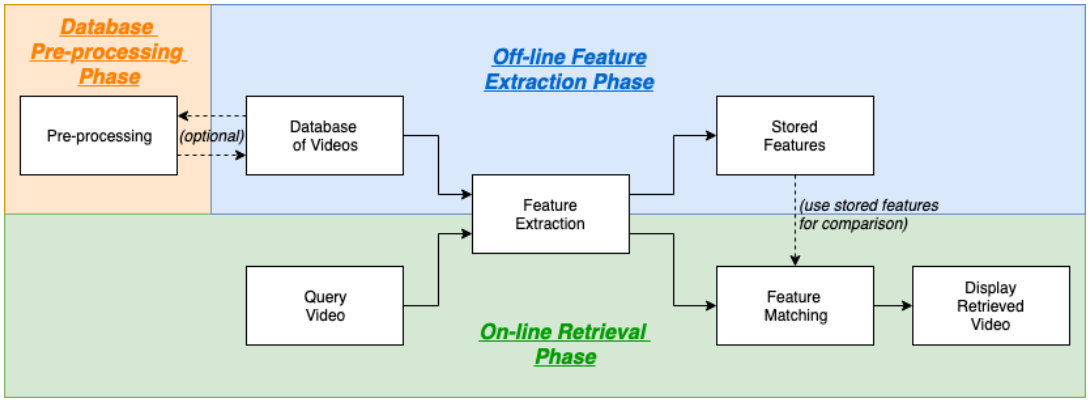

Figure 4.1: Basic CBVR system diagram depicting a high-level pipeline separated into three phases.

4.2.1 Off-line Feature Extraction Phase

Retracing Section 2.2 of the Literature & Technology Survey, the first distinction between the possible types of features that could be extracted from the videos is static and dynamic features. Because static features extraction is a much less challenging task to code and to compute than dynamic feature extraction, while providing almost as much information as dynamic features

and being cheaper to compute (Hu et al., 2011), static features are therefore the logical choice for this project. However, there are many different choices of static features to extract, including colour-based, texture-based and shape-based features. Texture and shape-based features are more suitable for images than videos, while colour-based videos are suitable for both. Therefore, colour-based features are the features used to describe the videos, fulfilling requirement F2.

The most efficient type of model for representing colour-based features is histograms, which are described in Section 2.2.1, Chapter 2. Different histograms exist for a variety of colour spaces depending on the application, with 2D histograms for *greyscale* and *RGB* colour spaces, and 3D histograms for the *HSV* colour space. Because these are the three most used colour models for describing videos and their frames, and the most accessible ones based on the third-party libraries available for Python, they are the three histograms models chosen for colour feature extraction.

To maximise speed, a compact signature should be generated for each video. The inspiration of representing a video in a unique and compact signature originates from (Araujo and Girod, 2018) who uses fisher vectors to construct compact signatures for the database videos and the query video. This project applies Araujo's idea of creating such a signature for the database videos and the query video to histograms, where histograms are generated for multiple frames of video and are then averaged into a single histogram for each model. Once generated, the feature vectors are then stored in files in order to be easily re-used in the on-line retrieval phase rather than being re-calculated in this phase every time. The design choices for storing the compact signatures, also known as "feature vectors", are explored in more detail in Section 4.3.2.

4.2.2 On-line Retrieval Phase

Similarity Measurement

Histograms are discrete distributions of the extracted colour-based features. Therefore, the distance between two histograms can be used to establish how close they are to each other. A long array of metrics exists to calculate the distance between two discrete distributions. Therefore, the most common distance metrics can be used to perform the measurements. To determine which distance metrics to use, their availability in popular third-party libraries is considered. With Python as the chosen programming language and the third-party libraries to use already decided, the metrics available can be explored from the OpenCV and SciPy libraries.

OpenCV, the Python computer vision library, offers multiple metrics to compare histograms, including the *Correlation*, *Chi-Square*, *Intersection*, *Hellinger Distance* using the Bhattacharyya coefficient, an *alternative Chi-Square* and the *Kullback-Leibler Divergence* (OpenCV, 2015). All of the mentioned metrics can be used in the system, apart from the alternative Chi-Square distance and the KL Divergence. In the first hand, the alternative Chi-Square distance is more efficient when working with texture-based features rather than colour-based ones (Puzicha, Hofmann and Buhmann, 1997). In the second hand, the Kullback-Leibler Divergence is more aimed at calculating the loss of information between two different distributions rather than calculating the actual distance between them (Kurt, 2017). An additional advantage of using the OpenCV library to compare histograms, which is already mentioned in Section 4.1, is the fact that the original C++ code is being executed to calculate the comparison itself, meaning that using OpenCV to compare histograms is the quickest option available, even quicker than writing custom distance functions directly in Python.

SciPy, the Python library for mathematical/scientific/engineering functions, offers two more advanced statistical distance metrics to compare discrete distributions, including the Wassertein Distance[6] and the Energy Distance, which are both perfectly suitable for comparing histograms.

Using these metrics from the OpenCV and SciPy libraries satisfies requirement F22 of using libraries to avoid re-writing code that would be less efficient than the libraries'. Based on the availability of distance metrics in OpenCV and SciPy, the following distance metrics will be used when implementing the system, fulfilling requirement F8:

- OpenCV distance comparison metrics:
  - Correlation
  - Intersection
  - Chi-Square (first version only)
  - Hellinger (using the Bhattacharyya coefficient)
- SciPy statistical distances:
  - Earth's Mover Distance (Wassertein Distance)
  - Energy Distance

Smaller results[7] between two histograms using the aforementioned metrics translates to similarities between those histograms. Once the distance between

[6]The Wassertein Distance is also referred to as the Earth Mover's Distance.

[7]or larger results, depending on the metric

the query video and each database video has been calculated, the smallest or largest distance is used to find the best match. The number of correct matches made, also known as *true positives*, and the number of incorrect matches, also known as *false positives*, can be used to evaluate the system's accuracy (see requirement F10).

A few metrics are more complex than others, which, coupled with the colour model used, means that some distances have more importance than others. Therefore, the importance of distances depends on the colour space used. Following this logic, grey scale histograms have low importance as they only use one channel, RGB histograms have medium importance as they use three channels, and HSV histograms have the highest importance as they describe the colours in a frame with more detail than RGB.

Query Video Pre-processing

Drastic improvements in accuracy can be made by pre-processing the query video in the on-line retrieval phase. The query video corresponds to the video that a user records with a mobile-device and uses as input in the system in order to be matched with one of the database videos. A commercial application would allow the user to directly record a video from the application, which would, in turn, be used to extract its features and generate its feature vectors before comparing them to the feature vectors of the database videos. Based on current commercial computer vision systems, e.g. Shazam for music recognition, the system could even start processing the video as it is being written rather than waiting for it to be saved. However, these are design aspects that would be required if the system were to be released to the public on a large scale. This is not the case since the goal is to develop the matching algorithm and complete a functional pipeline, meaning the query video may be a recording of one of the database videos used as an input once the program starts running in the on-line retrieval phase. To simulate a realistic query and a potential commercial system, the video should be pre-recorded through a mobile device, fulfilling requirement F6, rather than being recorded directly through a mobile application, as pointed out by the low-priority requirement F12.

Recording the query through a mobile device's camera gives rise to multiple challenges. First, there is potential for large visual differences between the query clip and the database video due to the capture conditions (Liu, Mei and Zhang, 2014) (Wang et al., 2016). Some unfavourable capture conditions include:

- Undesired camera movements due to unstable recording, e.g. unstable recording, shaking hands.

- Low-quality query due to poor user recording, e.g. videos recorded at a distance (down-scaled) and skewed. Low-quality can also be caused by environmental conditions, e.g. lighting and reflections.
- Video noise because of the camera sensor.
- Decoding artefacts caused by various file compression.

These low-quality conditions add difficulty to the on-line retrieval phase and must therefore be filtered out by pre-processing the query video before computing its feature vector. Undesired camera movement due to shaking hands can be fixed by applying video stabilisation (requirement F11), while low-quality user recording can be solved by selecting a ROI[8] (requirement F7). Nonetheless, selecting an ROI can be tricky as it relies on the capture conditions to maximise the matching accuracy. Four different types of query conditions are depicted in Figure 4.2:

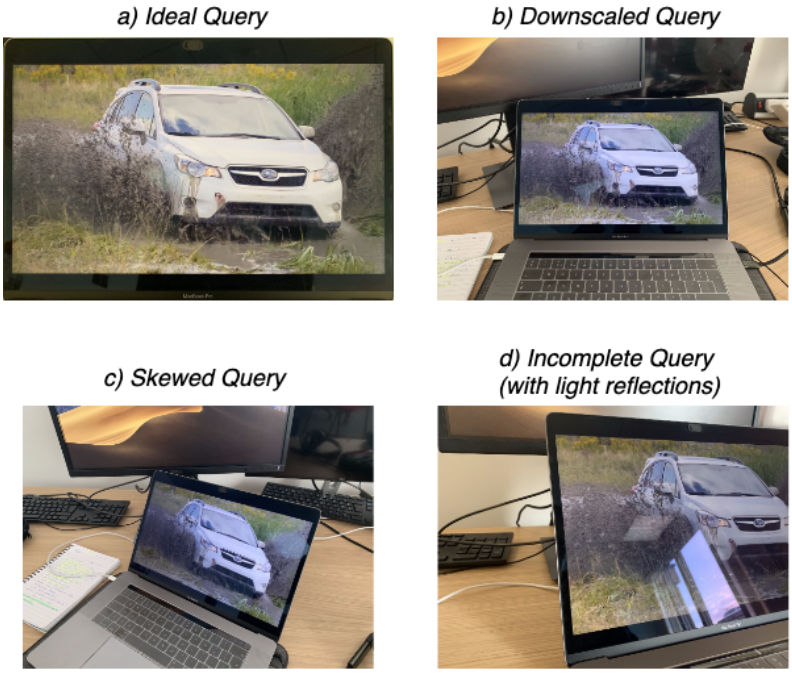

Figure 4.2: Different types of video queries, emphasising the contrast between what an idea query and poor quality queries would look like. All of these queries are often coupled with minor camera movement due to shaking hands.

- a) Ideal Query: allows for an accurate ROI selection.

[8] Region of Interest

- b) Down-scaled Query: requires more complex ROI selection as it is harder to detect the edges of the screen automatically. There is also a loss of information as an important section of the frame is excluded.

- c) Skewed Query: requires even more complex ROI selection and results in more loss of information.

- d) Incomplete Query: Greatly diminishes the system accuracy since part of the recorded video is missing.

The system should never expect ideal queries and always expect down-scaled and skewed queries as input. Ideal queries are rare as they depend on competent users with the knowledge of what defines a good video query. However, incomplete queries as shown in Figure 4.2d should not be accounted for as it is near-impossible to match such a query to a database video accurately as part of the screen is missing. Therefore, based on this analysis, the assumption that automatically selecting the ROI is a very complex task on its own can be made. Indeed, automatically detecting the edges of different screens to select the video being played as the ROI could grow to be even more intricate than the CBVR system itself. As a result, the ROI should be manually selected by choosing points on the query video to draw a contour around the screen.

4.2.3 Database Pre-Processing Phase

The database pre-processing phase aims to modify the database videos to improve the efficiency and speed of the feature vector generation. To achieve this, a shot boundary detection algorithm can be applied to each database video to extract a single frame per shot. This would cut down the size of a video to a tiny fraction of the original frames (ideally, one frame per shot), translating into fewer frames to be analysed when generating the feature vectors. The global threshold approach, which is the simplest and most efficient one (see Section 2.3.1, can be used to extract those frames. The distance metrics mentioned in Section 4.2.2 can be used to compare consecutive frames' histograms and detect shot boundaries when the distance between two histograms is greater than the global threshold.

4.3 General Project Design

Now that the design choices regarding the system pipeline have been made, general system choices such as the interface, the feature storage and the database videos can be inspected.

4.3.1 Interface

With Python being the chosen programming language to implement the system and the type of query videos being decided, the next decision is the type of interface that is used to present the results. Although it was specified in Chapter 1 that the system is meant to target mobile devices, creating a mobile application is a waste of resources for a project with such a narrow time frame. Although Python offers many choices of creating native GUIs using frameworks such as Tkinter, PyQt or WxPython, or creating HTML websites, building an interactive graphical interface would take as much time as developing the entire system pipeline. Indeed, the primary goal of the project is to complete the different phases of the pipeline that process and match a query video to a database video. Spending time on designing a GUI[9] for a mobile device is unnecessary as the objective is not to create a commercial application that can be tested with users, as established by requirement NF2, but to develop the matching algorithm that would work in the background of such an application.

The only pieces of information that need to be displayed are the following:

- Progress of current phase, e.g. during the off-line feature extraction phase, specify how many videos have been processed and how many are left (see requirement F5). Visual cues, such as spinners, to can be used to indicate that work is being carried out by the program.
- Histograms: display the generated histograms for each video.
- Manual Cropping: display the video to have the user manually select the region of interest. Show visual indicators of the ROI being selected.
- Results (see requirement F9):
  - The distance measurements between the query video and each database videos, while specifying which distance metric is used. Elegant output methods such as tables could be used to display the results to satisfy requirement NF3.
  - The final result, which should correspond to a thumbnail of the matched video.
  - Additional data to measure the efficiency, e.g. the runtime and the accuracy (comparison of the number of true positives and the number of false positives)

Most of the information previously mentioned can be displayed directly in the console, such as the current phase progress, the distance measurements

[9] Graphical User Interface

and the efficiency results. The only graphical requirements are for the manual query cropping, the histograms and the final result output. The OpenCV GUI functions can be used to display the video in a new window and to manually draw the contour of the region of interest as automatically detecting the edges of the screen in the query video would be a very complicated task on its own. Regarding the histograms and the final result, it is crucial to show visual representations of the histograms and the matched video as it was noticed that during the demonstration of progress in February, only printing the title of the matched video did not provide a sense of accomplishment. Displaying the histograms and the matched video result can be done through Matplotlib, which is a library used to display plots (histograms) and images (thumbnail of the matched video). The combination of a CLI-based application supported by a few graphical interfaces meet requirement F16, while requirement F19 will not be fulfilled due to time constraints as it is a low-priority requirement.

4.3.2 Feature Storing File Type

The features extracted during the off-line feature extraction phase (see Section 4.2.1) and used during the on-line retrieval phase (see Section 4.2.2) can either be stored in plain text files or binary files. Due to the small amount of spread data (three histograms for each video) and the advantages of being able to view the data being written to the text files for debugging purposes, plain text files are the chosen file types for storing the features extracted from the database videos, satisfying requirement F3. Additionally, with Python as the chosen programming language, optimal functions from the NumPy library can be used for quick I/O operations when saving the data in plain text files, fulfilling requirement F4. For a complete review of the pros and cons between text files and binary files, see Appendix B.

4.3.3 Database videos

Although the goal of the project is to eventually work with feature-length movies, the system will mainly be tested with short videos ranging between 7 and 14 seconds (see requirement F25). Indeed, using feature-length movies is not necessary to test the different stages of the system pipeline, and would not meet requirement F24 since they are copyrighted and not re-usable. A single feature-length movie can be used to test the database pre-processing phase rather than the entire pipeline (see requirement F27), while the short videos can be used to test the combination of the off-line feature extraction and on-line retrieval phases.

4.4 Chosen Solution

1. **Programming Language**:
   (a) Python 3.7[10]
   (b) Third-party libraries:
      i. OpenCV
      ii. NumPy
      iii. SciPy
      iv. Matplotlib
2. **Off-line Feature Extraction Phase**:
   (a) **Types of feature**: static colour-based features in the form of colour histograms.
   (b) **Types of Histograms Models**:
      i. Greyscale
      ii. RGB
      iii. HSV
   (c) **Feature Vector (Compact Signature)**: Averaged histogram (for each of the three histogram models)
3. **On-line Retrieval**:
   (a) **Distance metrics**:
      i. Correlation
      ii. Intersection
      iii. Chi-Square Distance
      iv. Hellinger Distance
      v. Wassertein Distance
      vi. Energy Distance
   (b) **Query Video Pre-Processing**:
      i. Video stabilisation
      ii. ROI selection: manual
4. **Database pre-processing**: shot boundary detection algorithm using global threshold approach
5. **Interface**: Mainly command line for printing data, along with OpenCV GUI for the query video cropping and Matplotlib for displaying the histograms and results.

[10] Version 3.7 is the latest stable release of Python, satisfying NF1.

6. **Feature Storing File Type**: Plain Text (".txt") files.
7. **Database Videos**: Short license-free videos ranging between 7 and 14 seconds each.

4.5 Summary

With the main design aspects of the system pipeline phases laid out, along with an outline of the various system components such as the query video pre-processing operations, programming language, type of interface, type of storage feature file and type of videos used to populate the database, the system can finally be implemented in the next chapter.

Chapter 5

Implementation

This chapter presents an overview of the different steps taken to fully implement a functional system by exploring the entire system pipeline. The chapter begins with the first stage of the pipeline, the off-line colour-based feature extraction phase, where histograms are extracted and averaged to create the feature vectors for each database video before being saved in files. Next, the second stage of the pipeline, the on-line retrieval phase, is covered, explaining how the query video is matched to one of the database videos using the compact histograms previously generated. Finally, the final optional stage of the pipeline is explored, the database pre-processing phase, explaining how videos can be segmented into denser videos. These three phases of the pipeline culminate with the flowchart diagram representing the entire system. Figures, mathematical formulas and code snippets are used to assist with examples and more precise visualisations. To round off the chapter, the approach used to gradually test the system during development is detailed, along with a coverage of the code structure of the project.

5.1 Off-line Colour-Based Feature Extraction

This section details the different steps that go towards extracting colour features from the database videos to generate histograms and creating the feature vectors used to represent each video. The first step consists in establishing a selection of frames to process, followed by generating histograms for each of those selected frames. Once the histograms are generated, they are averaged into a single histogram before being normalised and stored into a plain text file.

Figure 5.1 demonstrates a visual example of inner pipeline of this phase with RGB histograms only. This phase is repeated for greyscale and HSV histograms as well). The function controlling the execution process of this phase can be found in Appendix H.1.2.

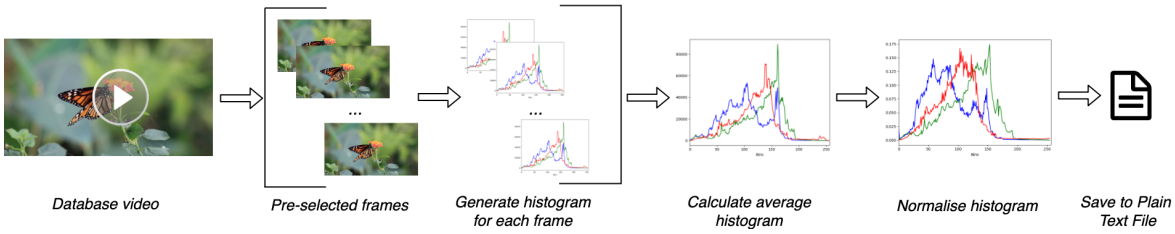

Figure 5.1: Visualisation of the pipeline of the off-line colour-based feature extraction phase with RGB histograms only.

This phase has to be repeated for every video in the database. For consistency throughout the various examples that are illustrated in this section, the *butterfly* video is used to provide examples of the multiple off-line colour-based feature extraction steps.

5.1.1 Histogram Generation

As described in Section 2.2.1, a histogram represents the distribution of the pixels in a single frame. Different types of histograms can be generated based on the colour model used. This section covers how three types of histograms are constructed based on the colour model used, starting with greyscale histograms, followed by RGB histograms and ending with HSV histograms. Supporting code listings covering the generation of these three histograms can be found in the appendix, including the greyscale histograms in Appendix H.2.2, the RGB histograms in Appendix H.2.3, the HSV histograms in Appendix H.2.4 and a high-level module overview of the *HistogramGenerator* class and its functions in Appendix H.2.1.

Greyscale Histogram

The first type of histogram that is generated is the greyscale histogram, also known as an intensity-based histogram. This type of histogram is used with frames that are first converted to a black & white colour space (see Figure 5.2), and thus has a single channel rather than the three traditional RGB channels found in colour images. The conversion is done using OpenCV's *cv2.cvtColor()* function along with the *cv2.COLOR_BGR2GRAY* attribute:

```
grey_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

Because the video frames are 8-bit images, the spectrum of the potential pixel values ranges between 256 possible values[1]. As a result, the histogram has 256 different bins from 0 to 255 to represent all the potential values that each pixel can take. The histogram itself is portrayed by Equation 5.1, where

[1] $2^8 = 256$

Figure 5.2: Conversion of a frame from the RGB colour space to greyscale (black & white).

$g \in [0, 255]$ and N_g represents the number of pixels that have an intensity equal to g. For example, $H(30)$ represents the number of pixels in the image with an intensity value of 30.

$$H(g) = N_g \tag{5.1}$$

The implementation of the greyscale histogram generation in the system is carried out through OpenCV's *calcHist()* method, with the result portrayed in Figure 5.3. The grey frame is passed to the function, along with the number of channels to use, which is a single one in this case ([0]), an empty mask *None* since the histogram is being calculated for the full frame, the number of bins [256], and the range of values $[0, 256]$:

```
histogram = cv2.calcHist([grey_frame], [0], None, [256], [0,256])
```

Figure 5.3: The generated greyscale histogram (not normalised).

These greyscale histograms are generated for multiple frames of the video, but not all the frames as this would be highly inefficient, as explained in Section 2.1.2. Indeed, rather than generating a histogram for each frame, which would be extremely inefficient and waste of computations, a selection of frames can be made in advance, which is then used for the histogram generation. Ideally, the frames extracted from the shot boundary detection algorithm later described in Section 5.3 would be used. However, because the range of the duration of the videos used is between 7 and 14 seconds and they are only made

of a single shot, the shot boundary detection algorithm would only extract one frame from the videos. Therefore, for the purpose of building a realistic system, one frame is extracted every second to build up the selection of frames, using the *_get_frames_to_process()* function, which can be found in Appendix H.2.10. For example, with the butterfly video that contains 301 frames and has a frame rate of 30 frames per second, 11 frames are selected to generate a greyscale histogram (frames *[1, 31, 61, ..., 271, 301]*).

Once the greyscale histograms are generated for each of these pre-selected frames, they are all stored away in a list, ready to be later averaged into a single histogram that will be used as the first feature vector to represent the entire video (see Section 5.1.2).

RGB Histogram

Now that greyscale histograms are generated for the video, RGB histograms are produced next. As described in 2.2.1, an RGB histogram represents the colour distribution of the pixels in a frame for three different channels: red, blue and green. The coloured frame is a combination of three colour channels, which can be split into three different stills, as rendered in Figure 5.4.

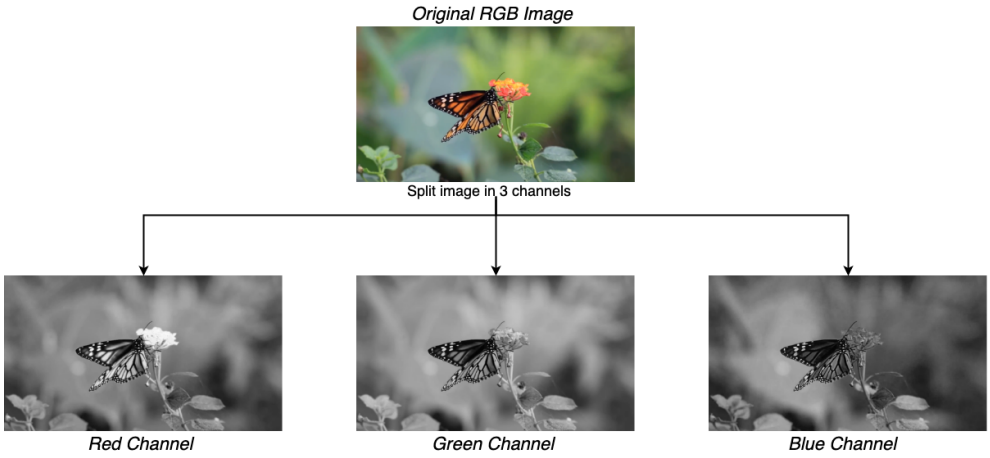

Figure 5.4: The three colour channels that make up an coloured RGB image.

The RGB histogram is similar to the previously mentioned greyscale histograms, except that there are three separate histograms rather than a single one. There is, therefore, a 2D histogram for each colour channel (one in red, one in blue and one in green), where the brightness levels of each pixel are represented for each of those three colour channels. Each of the stills resulting from the split depicted in Figure 5.4 will have their own histograms, which will form a complete RGB histogram when merged.

In contrast to the greyscale histograms, frames do not need to be converted to a different colour space when constructing RGB histograms as they are directly imported into three channels by OpenCV's *cv2.VideoCapture()* function. In a similar fashion to the greyscale histograms, 256 bins are used for the RGB histograms, representing the potential values that each pixel can take for each colour channel. The RGB histogram is a combination of three 2D histograms, which are illustrated by Equation 5.2, where $g_i \in [0, 255]$ and $N_g(i)$ represents the number of pixels that have an intensity equal to g for the colour channel i. For example, $H_{i=2}(60)$ represents the number of pixels in the third channel of the image (blue channel) with an intensity value of 60.

$$\sum_{i=0}^{2} H_i(g) = N_g[i] \tag{5.2}$$

To implement RGB histograms in the system, OpenCV's *calcHist()* is once again used. In an identical manner to the greyscale histogram, the frame is passed to the function along with the colour channel *[i]*, which are all placed inside a for-loop to generate a histogram for the three colour channels:

```
histogram = cv2.calcHist([frame], [i], None, [256], [0, 256])
```

At this point, the three individual histograms that have been generated for the three colour channels are merged into a single plot to build the RGB histogram, as depicted in Figure 5.5.

Figure 5.5: The combination of the generated histograms for the three colour channels to construct the RGB histogram (not normalised).

Now that RGB histograms have been generated for each of the pre-selected frames (one frame every second), they are all stored in a dictionary made up of three keys, one for each colour channel. Each key links to a list where the histograms are stored away before being averaged into a single histogram used to create the second feature vector.

HSV Histogram Generation

With the greyscale and RGB histograms now formed, the HSV histograms can finally be generated. HSV histograms make use of the HSV colour space, requiring the video frames to be converted from the original RGB colour space to the HSV colour space (see Figure 5.6). The conversion is done using OpenCV's *cv2.cvtColor()* function along with the *cv2.COLOR_BGR2HSV* attribute:

```
hsv_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
```

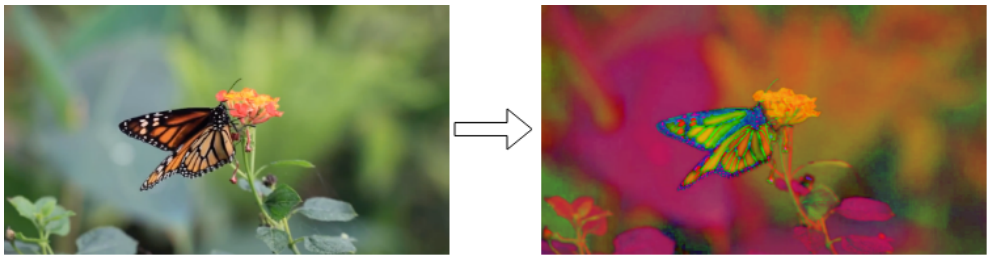

Figure 5.6: Conversion of a frame from the RGB colour space to the HSV colour space.

This type of histogram is more complex to generate as it is not a 2D histogram like the previous greyscale and RGB histograms, but a 3D histogram. Because they are represented in 3D rather than 2D due to the particularities of the HSV colour space, less bins can be used to plot the data as pixels fall into bins when they are in conjunction with the range of the three channels (the pixels values must be within the range of the hue AND the saturation AND the value). Therefore, to avoid creating a histogram that is neither too coarse nor too sparse, 8 bins are used for the hue channel, 12 bins are used for the saturation channel, and 3 bins are used for the value channel, resulting in a histogram of 288 values[2]. This histogram is approximately the same length as the 256 values previously used for the greyscale histograms and $3 * 256$ values for the RGB histograms. The histogram is described by Equation 5.3, where $N(h, s, v)$ represents the number of pixels that fall within the range of the bin with values h AND s AND v.

$$H(h, s, v) = N(h, s, v) \tag{5.3}$$

For example, a pixel with values $H(h = 31, s = 245, v = 200)$ will fall into a bin with[3]:

- a hue value between 22.5 and 45 (bin #2/8) AND
- a saturation value between 234.6 and 256 (bin #12/12) AND

[2] $8 * 12 * 3 = 288$

[3] Assuming ranges of 0-180 for Hue, 0-256 for Saturation, 0-256 for Value.

- a value intensity between 170.7 and 256 (bin #3/3).

The number of pixels in this bin will be $N(1, 11, 2)$ (0-based indexes).

Implementing the calculation of an HSV histogram in the system is again achieved through the use of OpenCV's *calcHist()* function, with the result shown in Figure 5.7. Similar arguments are passed such as the frame and an empty mask, but the arguments controlling the size of the histogram are different. In the code snippet below, *[0, 1, 2]* indicates that the histogram has three channels (H, S and V), *(8, 12, 3)* specifies the number of bins for each channel, and *[0, 180, 0, 256, 0, 256]* marks the ranges of these bins:

```
histogram = cv2.calcHist([hsv_frame], [0, 1, 2], None, (8, 12,
    ↪ 3), [0, 180, 0, 256, 0, 256])
```

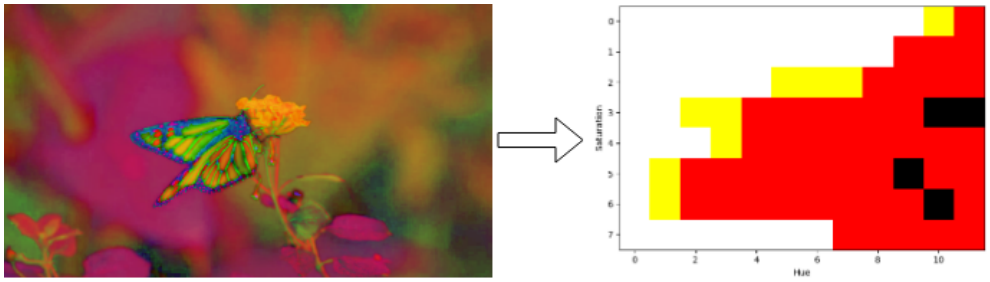

Figure 5.7: The generated 3D HSV histogram (not normalised).

To conclude this first step of the off-line colour-based extraction phase, HSV histograms are finally generated. Once all the HSV histograms are generated for each pre-selected frame, they are all stored in a list, like for the greyscale and RGB histograms. With the greyscale, RGB and HSV histograms now all generated for each pre-selected frame of the video; these can now be averaged to create one feature vector for each histogram, which is explained in the next section.

5.1.2 Video Feature Vector Generation

At this stage of the off-line colour-based extraction phase, multiple greyscale, RGB and HSV histograms have been generated for each pre-selected frame for a video. Each of these histograms can now be averaged into one single histogram per colour model for each video. This means that each database video will be represented by one averaged greyscale histogram, one averaged RGB histogram and one averaged HSV histogram, which will constitute the feature vectors of the video. This is achieved by looping through all the histograms' bins generated per video and calculating a new averaged value for each bin. For reminders, this process is repeated for every video in the database. Supporting code listings covering the generation of the feature vectors for the

three histogram models can be found in the appendix, including the greyscale histograms in Appendix H.2.5, the RGB histograms in Appendix H.2.6 and the HSV histograms in Appendix H.2.7.

For the **greyscale histograms**, the 256 bins of the N generated histograms are looped through and averaged (see Equation 5.4) to create a new mean value, where $H_n(i)$ corresponds to the number of pixels for the video's n^{th} histogram at pixel intensity i.

$$\sum_{n=0}^{N} \frac{\sum_{i=0}^{255} H_n(i)}{N} \tag{5.4}$$

The averaged value is then stored into the corresponding i_{th} bin of the new histogram. For example, if $N = 10$ (the video had ten pre-selected frames, so ten greyscale histograms were generated for that video), then for each of the 256 bins, the values of the ten histograms at that bin are averaged and the mean is stored in the new histogram.

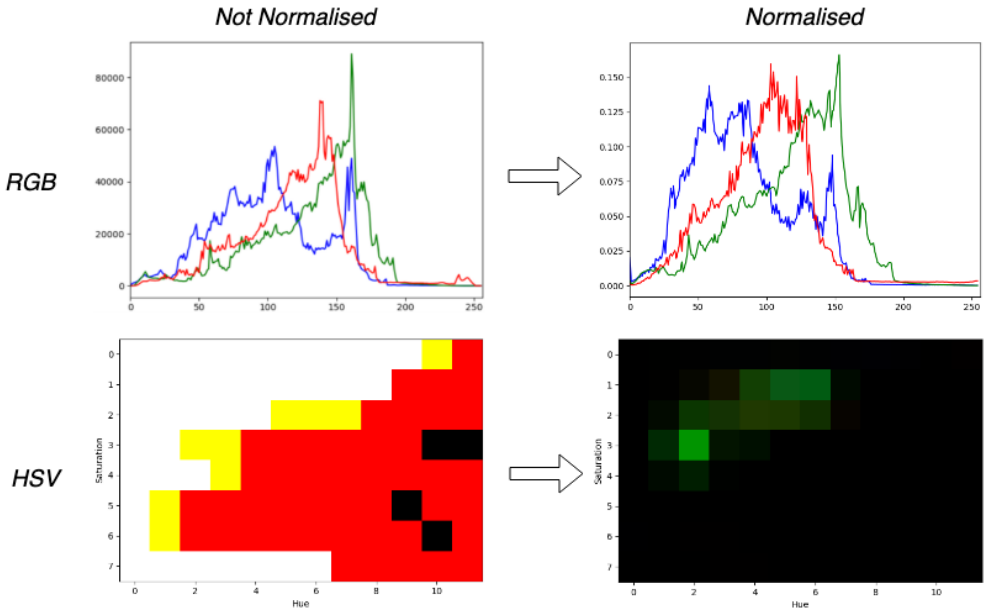

Figure 5.8: Normalising RGB and HSV histograms.

Once all the bins have been averaged and added to the new histogram, that averaged histogram is normalised to obtain scale invariance between the histograms of the different videos. Instead of representing the number of pixels that fall into a bin, a normalised histogram represents a relative percentage count of the number of pixels in each bin, as depicted in Figure 5.8. This is a necessary step in cases where the video frames have different sizes

(the larger frames to have more pixels than the smaller frames) as the histograms can be more accurately compared when they represent a relative percentage rather than an integer count. The normalisation is achieved through OpenCV's *cv2.normalize()* function:

```
histogram = cv2.normalize(histogram, histogram)
```

Finally, the averaged and normalised histogram values are stored in a plain text file as float values to maintain the precision, which is implemented using NumPy's *np.savetxt()* function:

```
np.savetxt("../histogram_data/{}/hist-{}".format(self.file_name,
    ↪ "gray"), avg_histogram, fmt='%f'))
```

With the first feature vector in the form of an averaged and normalised greyscale histogram now stored in a plain text file, the same steps are repeated for the RGB and HSV histogram, with a few differences in the loops. First, for the **RGB histograms**, identical steps to the greyscale histogram's are followed for the three colour channels rather than a single time (see Equation 5.5).

$$\sum_{h}^{(r,g,b)} (\sum_{n=0}^{N} \frac{\sum_{i=0}^{255} H_{h,n}(i)}{N_h}) \tag{5.5}$$

Second, for the **HSV histograms**, the histogram averaging is done by looping through each hue, saturation and value bin, since it is a 3D histogram and not a 2D histogram (see Equation 5.6).

$$\sum_{n=0}^{N} (\sum_{h} \sum_{s} \sum_{v} H_n(h, s, v)) \tag{5.6}$$

Finally, the *cv2.normalize()* and *np.savetxt()* functions are used to normalise and save the average RGB and HSV histograms to plain text files. Examples of the saved plain text files for the butterfly video can be found in Appendix D.1 for the greyscale histogram, in Appendix D.2 for the RGB histogram and in Appendix D.3 for the HSV histogram.

The final result of the off-line colour-based feature extraction phase is a set of three averaged histograms that represent an entire video, with the values saved in three separate plain text files, as depicted in Figure 5.9.

Thanks to an argument parsing function written at the start of the program (see Appendix H.1.1), there is a choice to display these histograms using Matplotlib while they are being generated based on a flag passed on program startup. If the *"showhists"* flag is passed, then all averaged histograms are

displayed before being saved in plain text files.

As it takes approximately three seconds to process a single video in order to generate and store the three averaged histograms, the entire phase can take a while depending on the number of videos in the database. Therefore, a spinner is employed to indicate that the system is performing computations, which is implemented using Py-Spin[4], a third-party library used to implement console-based spinners. The spinner, coupled with the histograms being displayed as they are being calculated, gives a visual indication that progress is being made.

With the colour-based features now saved in text files, they can be used multiple times in the on-line retrieval phase without needing to be regenerated, unless changes are made to the database videos, e.g. adding or removing videos.

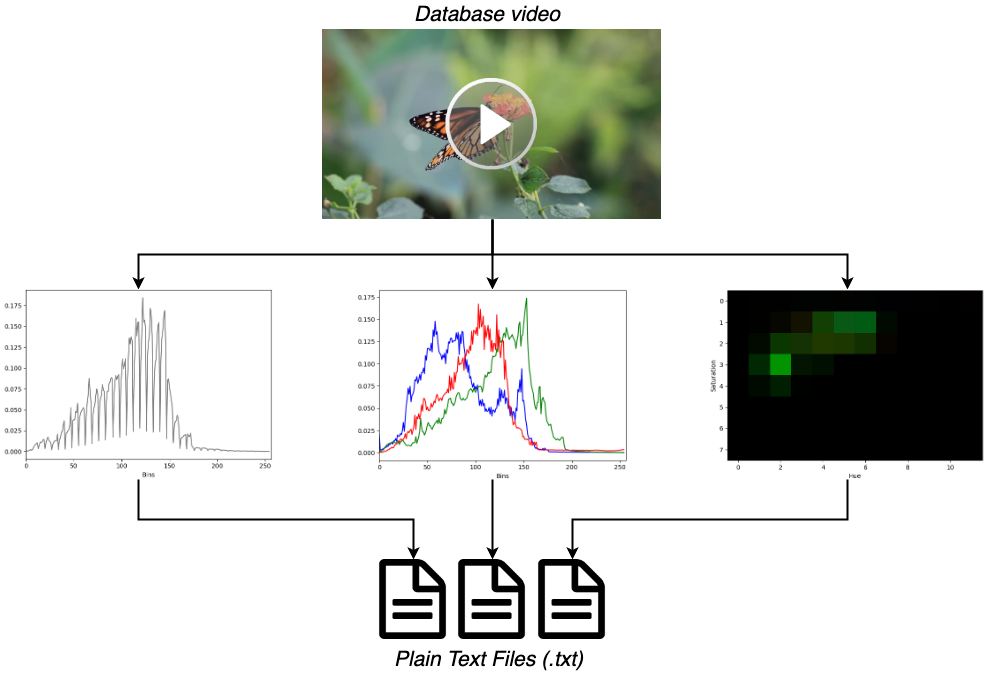

Figure 5.9: The three average histograms generated for each database video and saved in plain text files.

[4]Py-Spin: `https://github.com/lord63/py-spin`

5.2 On-line Retrieval

This section details the different steps that go towards matching a query video to one of the database videos using the colour-based features extracted in Section 5.1. The first step consists in processing the recorded query video to get rid of undesired artefacts such as poor framing (e.g. camera not properly positioned) and unwanted camera movements (e.g. shaky camera). Once the query is finished being processed, the same colour-based features previously extracted for all the database videos in the form of greyscale, RGB and HSV histograms are used to compute the similarities between the query video and each database video by measuring the distance between the histograms. The video with the highest score will be the match found by the system, which is displayed to the screen to show the final result. The initial function controlling the execution process of this phase can be found in Appendix H.1.3.

5.2.1 Query Video Pre-processing

Video Stabilisation

The first improvement that can be made to the query is video stabilisation. Video stabilisation is implemented through the VidStab[5] third-party library, which is a quick solution to prune query videos with shaky movements. Additionally, an option to extend the query's borders to maintain the original frame size is available to prevent losing information. The library is implemented in a custom class, which can be found in Appendix H.3.1.

A "yes/no" question is displayed on the console, giving a choice to stabilise the query video or not. If "yes" is chosen, the system first checks if a potential version of the video has already been previously stabilised and stabilises the query if it has not been stabilised already. Similarly to the previous phase, a spinner is used to indicate that the video is being stabilised as the process takes around twenty seconds for a ten second-long video. If "no" is chosen, then the system again checks for potential stabilised versions to use; otherwise, it uses the original version.

Region of Interest Selection

The second improvement that can be carried out to the query is selecting a Region Of Interest (ROI). Selecting an ROI involves cropping the query video to select only the area with the actual object of interest. In this case, this involves selecting the video playing on a display.

[5]Python Video Stabilisation: `https://github.com/AdamSpannbauer/python_video_stab`

ROI selection is achieved through OpenCV's GUI functionality, which is employed through multiple functions such as a function to display images using *cv2.imshow()*, one to draw on the displayed images using *cv2.rectangle()*, and one to record mouse events by writing a callback function named *click_and_crop()*:

```
cv2.setMouseCallback("Select ROI", self.click_and_crop)
```

These three functionalities are combined in the system to display the first frame of the query video and allow a manual selection of two points on the frame. The first point is selected when the left mouse button is pressed using *cv2.EVENT_LBUTTONDOWN*, and the second point is selected when the button is unpressed using *cv2.EVENT_LBUTTONUP*. With these two points, a rectangle can be drawn on the frame to indicate the area that was selected. If the rectangle is correctly drawn, the *'C'* key can be pressed to perform the cropping action; otherwise the *'R'* key can be pressed to erase the rectangle and attempt to re-draw it until a satisfactory ROI is selected. The point coordinates are then used to ignore all the pixels outside the drawn rectangle when generating the query video's histograms, as sketched in Figure 5.10. The full implementation of the ROI selection class can be found in Appendix H.3.2.

Figure 5.10: The process of manually selecting a ROI to crop the query video.

Once the video query has been processed for stability and ROI selection, the same feature vectors from the off-line extraction phase can be generated before being compared to the database videos' feature vectors.

5.2.2 Matching Query to a Database Video

Query Video Feature Extraction

In order to compare the query video to all the database videos, the same colour-based features described in Section 5.1 need first to be extracted and the feature vectors calculated. Three feature vectors are generated for the query video in the form of an averaged greyscale histogram, an averaged RGB histogram and an averaged HSV histogram. To avoid code duplication, the same functions written for the off-line colour-based feature extraction phase are re-used to generate the query video's histograms. This is achieved through a function parameter named *is_query*, which is a boolean that enters more sections of the function when it is true, such as selecting an ROI:

```
def generate_video_rgb_histogram(self, is_query=False,
    ↪ cur_ref_points=None):
```

Once these histograms are generated, they can be compared to the histograms of each database videos.

Distance Measurements

At this stage of the pipeline, all of the feature vectors for the database videos and the query video have now been created and can be used to compute the similarities between them. The similarity measurements are done by calculating the distance between each of the database videos' histograms with the query video's histograms. Four different distance metrics are used to compare the 2D greyscale and RGB histograms, while two distance metrics are used to calculate the similarities between the 3D HSV histograms. Supporting code listings covering the distance calculation between 2D and 3D histograms can be found in Appendix H.2.8.

The calculations between two 2D histograms are achieved through OpenCV's *cv2.compareHist()* function, where the two histograms to compare are passed along with the method of choice:

```
comparison = cv2.compareHist(query_hist, rgb_video_hist, method)
```

For the 3D HSV histograms, SciPy's statistical distances are used to calculate the distance between the different bins. Different functions are used between the 2D and 3D histograms as OpenCV's *cv2.compareHist()* function does not work with 3D arrays, whereas SciPy's statistical distances does. The six different methods used are detailed below, where $d(H_1, H_2)$ represents the distance between a histogram H_1 and a histogram H_2, and I represents the number of bins in the histograms.

1. Correlation:
   This is a metric that indicates the linear relationship between two histograms. The result of this metric ranges between $-1 < d(H_1, H_2) < 1$, where -1 represents a negative correlation, 0 an absence of correlation and 1 a high positive correlation. The closer the two histograms are to each other, the closer their correlation will be to 1. Equation 5.7 describes how this metric can be calculated.

$$d(H_1, H_2) = \frac{\sum_I (H_1(I) - \bar{H_1})(H_2(I) - \bar{H_2})}{\sqrt{\sum_I (H_1(I) - \bar{H_1})^2 (H_2(I) - \bar{H_2})^2}} \tag{5.7}$$

where $\bar{H}_k = \frac{1}{N}\sum_J H_k(J)$ and N is equal to the total number of bins in the histogram. This metric is available through OpenCV's *cv2.HISTCMP_CORREL* attribute:

```
comparison = cv2.compareHist(query_hist['grey'],
    ↪ dbvideo_hist_grey, cv2.HISTCMP_CORREL)
```

2. Intersection:
   This metric calculates the similarity between two histograms by superposing the two histograms and calculating how important the superposition is for each bin in the histogram. A value of 0 for a bin indicates that there is no overlap between the two histograms at that bin. The higher the intersection, the more the two histograms have in common, as portrayed by Equation 5.8.

$$d(H_1, H_2) = \sum_I min(H_1(I), H_2(I)) \tag{5.8}$$

   Intersection is implemented through OpenCV's *cv2.HISTCMP_INTERSECT* attribute:

```
comparison = cv2.compareHist(query_hist['grey'],
    ↪ dbvideo_hist_grey, cv2.HISTCMP_INTERSECT)
```

3. Chi-Square Distance:
   This metric uses the chi-square test $\chi^2 = \frac{(a-b)^2}{a}$ in the form of a distance function described by Equation 5.9. The lower the value, the more the data from H_1 fits the data from H_2 and vice versa, the higher the value, the less the data from H_1 fits the data from H_2.

$$d(H_1, H_2) = \sum_I \frac{(H_1(I) - H_2(I))^2}{H_1(I)} \tag{5.9}$$

   Therefore, the database video that will match the query the most is the one with the lowest chi-square distance. The chi-square distance is implemented in the code by using OpenCV's *cv2.HISTCMP_CHISQR*:

```
comparison = cv2.compareHist(query_hist['grey'],
    ↪ dbvideo_hist_grey, cv2.HISTCMP_CHISQR)
```

4. Hellinger Distance (using Bhattacharyya Coefficient):
   The Hellinger distance is used to quantify the difference between two discrete probability distributions (see Equation 5.10). The smaller the value, the closer the two histograms are to each other.

$$d(H_1, H_2) = \sqrt{1 - \frac{1}{\sqrt{\bar{H}_1 \bar{H}_2 N^2}} \sum_I \sqrt{H_1(I) \cdot H_2(I)}} \tag{5.10}$$

where $\bar{H_k} = \frac{1}{N} \sum_J H_k(J)$ and N is equal to the total number of bins in the histogram. The Hellinger metric is implemented in the code through OpenCV's *cv2.HISTCMP_HELLINGER*:

```
comparison = cv2.compareHist(query_hist['grey'],
    ↪ dbvideo_hist_grey, cv2.HISTCMP_HELLINGER)
```

5. Earth's Mover Distance (Wassertein Distance):
   The Earth's Mover Distance, also known as Wassertein Distance, is a metric used to measure the dissimilarity between two distributions. The EMD depicted in Equation 5.11 is calculated by measuring the least amount of moves required to transform one histogram into the other. Therefore, the smaller the EMD is, the more similar two histograms are to one another.

$$EMD(H_1, H_2) = \sum_I |EMD_I(H_1, H_2)| \tag{5.11}$$

   where:

   - $EMD_0 = 0$
   - $EMD_{I+1}(H_1, H_2) = H_1(I) + EMD_I - H_2(I)$

   The EMD between two histograms is implemented by looping through each slice of the 3D HSV histogram and using SciPy's *scipy.stats.wasserstein_distance()* function on each slice (the SciPy method named Wassertein Distance is a synonym for EMD, which is the term used in Computer Science):

```
comparison = 0
for h in range(0, self.bins[0]): # loop through hue bins
    for s in range(0, self.bins[1]): # loop through
    ↪ saturation bins
        comparison +=
    ↪ wasserstein_distance(query_histogram['hsv'][h][s],
    ↪ dbvideo_hsv_histogram[h][s])
```

6. Energy Distance:
   The Energy Distance is a metric used the cumulative distribution functions of two distributions to measure the distance between two histograms (see Equation 5.12). The smaller the distance is, the closer the two histograms are, with $d(H_1, H_2) = 0$ being identical histograms.

$$d(H_1, H_2) = \sqrt{2\mathbf{E}|X - Y| - \mathbf{E}|X - X'| - \mathbf{E}|Y - Y'|)} \tag{5.12}$$

   where: X and X' are independent random variables with the probability distribution H_1, and Y and Y' are independent random variables with the probability distribution H_2. The Energy Distance between two histograms is implemented by looping through each slice of the 3D HSV

histogram and using SciPy's *scipy.stats.energy_distance()* function on each slice:

```
comparison = 0
for h in range(0, self.bins[0]):  # loop through hue bins
    for s in range(0, self.bins[1]):  # loop through
    ↪ saturation bins
        comparison +=
    ↪ energy_distance(query_histogram['hsv'][h][s],
    ↪ dbvideo_hsv_histogram[h][s])
```

All of the distance metrics and histogram models are finally combined to produce the final result. For each histogram model distance metric used, scores indicating the similarities between each database video and the query video are calculated. A table is printed in the console to display the distance results for each metric and histogram model using the TerminalTables[6] third-party library. An example of the printed console table using the library can be found in Appendix C. Additionally, the data is also written to multiple CSV files to further analyse in tools such as Excel and in order to read the output data of the on-line retrieval phase after the console has been cleared or closed.

Of all the scores calculated for each model and distance metric, the one revealing which database video matches the query the most is used to increment a dictionary. The increment values rely on pre-defined weights assigned to each histogram model, which are detailed in Table 5.1:

Histogram Model	**Distance Measurement**	**Weight**
Greyscale	Correlation	1
	Intersection	1
	Chi-Square Distance	1
	Hellinger Distance (Bhattacharyya Coefficient)	1
RGB	Correlation	5
	Intersection	5
	Chi-Square Distance	5
	Hellinger Distance (Bhattacharyya Coefficient)	5
HSV	Earth's Mover Distance (Wassertein Distance)	10
	Energy Distance	10

Table 5.1: Weights used for the distances calculated based on the histogram model and the metric used. High importance is given to HSV results, medium importance to RGB results, and low importance to greyscale results.

[6] TerminalTables: `https://github.com/Robpol86/terminaltables`

- Greyscale histograms are the least descriptive model, with only one channel which cannot represent colours. The lowest weight worth 1 is therefore attributed to these.
- RGB histograms use three channels and can describe the colours in a frame. They are therefore attributed to a weight of 5.
- HSV histograms are the most complex histograms employed in the system as they use three channels to describe how human vision perceives colours, plotting the results in 3D histograms. The highest weight worth 10 is therefore attributed to these.

The dictionary is used at the end of the pipeline to determine which video was matched the most based on the number of occurrences in the dictionary. The number of occurrences is then converted into a probability of the videos in the dictionary to match the query video, which is plotted into a simple histogram using Matplotlib to visualise which video is matched the most, as shown in the left-hand side of Figure 5.11. The final result is displayed in the form of a figure that includes the first frames of the query video and the matched video, along with the runtime in seconds and the $accuracy = (\#\ True\ \ Positives)/(\#\ \ Matches\ \ Made)$ of the match, which is depicted in the right-hand side of Figure 5.11.

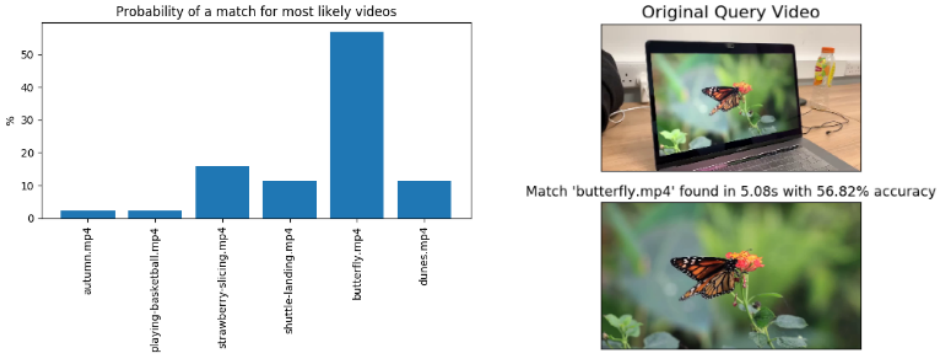

Figure 5.11: Example of the results at the end of the on-line retrieval phase. In the left-hand side is a histogram showing the probability in percentage % of the most likely videos to match the query. In the right-hand side is a figure presenting the final results, including the first frames of the query video and the match video along with the runtime and the accuracy of the match.

Once this figure with the final results is displayed, the program exits, marking the end of the pipeline. The on-line retrieval phase can be repeated any number of times using different query recordings of one of the database videos, without the need to run the off-line colour-based extraction phase again.

5.3 Database Pre-Processing

This section details the steps that go towards processing the database of videos with a shot boundary detection algorithm in order to segment a video. A short twelve-seconds long video made up of four shots is used for this phase of the system, which contains three different types of transitions between each shot, as depicted in Figure 5.12:

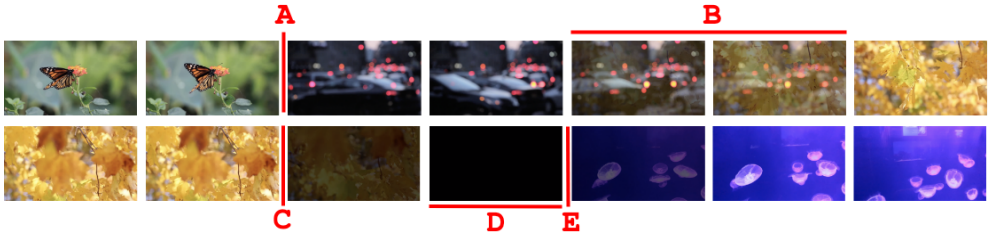

Figure 5.12: Video frames used for the shot boundary detection algorithm.

- Quick cut between the butterfly video and the traffic video,
- Dissolve cut between the traffic video and the autumn video,
- Fade to black cut between the autumn video and the jellyfish video.

The shot boundary detection algorithm is implemented using similar techniques described in the two previous phases. RGB histograms are generated for consecutive frames before being compared using distance metrics. The first metric used is the Kullback-Leibler Divergence, followed by the Intersection metric.

Kullback-Leibler Divergence

The Kullback-Leibler Divergence is normally used to calculate how much information is lost between two distributions, based on Equation 5.13, but it can nevertheless be used to detect when two consecutive histograms are too distant from one another.

$$d_{KL}(H_1||H_2) = \sum_I H_1(I) \cdot log(\frac{H_1(I)}{H_2(I)}) \tag{5.13}$$

Therefore, the RGB histograms of two consecutive frames are generated and compared using the KL Divergence metric for each colour channel in the his-

togram, which is adapted to the shot boundary detection algorithm in Equation 5.14.

$$detection \quad if \sum_{h}^{(r,g,b)} d_{KL}(H_1||H_2) > t \\ \sum_{h}^{(r,g,b)} (\sum_{I} H_{1,h}(I) \cdot log(\frac{H_{1,h}(I)}{H_{2,h}(I)})) > t \tag{5.14}$$

where t is the threshold. If the sum of the three distances is greater the global pre-defined threshold $t = 10$, then the two consecutive frames are different enough to be interpreted as a shot change, and the frame index is added to a list of detected shot boundaries. The global threshold of 10 was determined by running the code multiple times to determine a value that detects most of transitions mentioned earlier. This is implemented in the code using the same techniques mentioned in the previous techniques, employing the *cv2.calcHist()* function to generate the histograms for the three colour channels and *cv2.compareHist()* to compare the RGB histograms:

```
comparison_r = cv2.compareHist(prev_rgb_hist['r'][0],
    ↪ cur_rgb_hist['r'][0], cv2.HISTCMP_KL_DIV)
comparison_g = cv2.compareHist(prev_rgb_hist['g'][0],
    ↪ cur_rgb_hist['g'][0], cv2.HISTCMP_KL_DIV)
comparison_b = cv2.compareHist(prev_rgb_hist['b'][0],
    ↪ cur_rgb_hist['b'][0], cv2.HISTCMP_KL_DIV)
comparison = (comparison_b + comparison_g + comparison_r) / 3
```

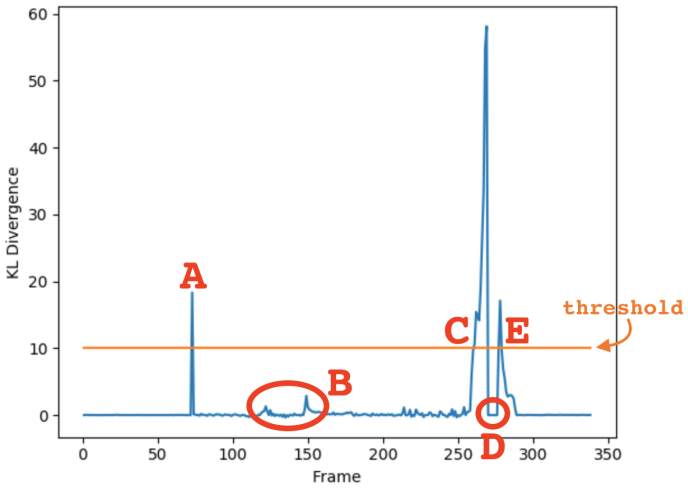

Figure 5.13: Example of the shot boundary detection algorithm on the test video from Figure 5.12 using the Kullback-Leibler Divergence to compare the RGB histograms between consecutive frames.

Once the video has been processed, all of the frames with a KL divergence superior to the threshold are marked as shot boundaries. The KL divergence between the consecutive frames from the test video is plotted using Matplotlib, with the results depicted in Figure 5.13.

This global threshold approach using KL Divergence to process consecutive frames detects hard cuts *(A)* and fading to black cuts *(C & E)*, but does not detect a dissolve cut *(B)*. It also detects the black frames in a fading to black transition as a new shot *(D)*.

Intersection

The Intersection metric previously explained in Section 5.2.2 with Equation 5.8 is also used to calculate the similarities between two consecutive frames in a video, as shown in Equation 5.15:

$$detection \quad if \sum_{h}^{(r,g,b)} d(H_1, H_2) < t \\ \sum_{h}^{(r,g,b)} (\sum_{I} min(H_{1,h}(I), H_{2,h}(I))) < t \tag{5.15}$$

where t is the threshold. In this case, if the sum of the three distances is smaller than then global threshold $t = 7$, then the two consecutive frames are not similar enough to be considered as the same shot, and the frame index is also added to a list of detected shot boundaries. Similarly to the previous technique, the global threshold of 7 was determined by running the code multiple times to find a value than could detect most transitions. *cv2.calcHist()* and *cv2.compareHist()* are once again used to generate and compare the RGB histograms. Once the video has been processed, all the frames with an intersection value smaller than the threshold are marked as shot boundaries, with the results depicted in Figure 5.14.

This global threshold approach using the Intersection metric to process consecutive frames detects all the transitions in the test video, including the hard cuts *(A)*, the dissolving cut *(B)* and the fading to black cuts *(C & E)*. However, it also detects the black frames in a fading to black transition as a new shot *(D)*.

Despite not being 100% accurate with the dissolve cut *(B)* not being detected and the black frames being detected as a new shot, the shot boundary detection algorithm is a satisfying method regardless of the metric being used (KL Divergence and Intersection both worked well with the test video). It is

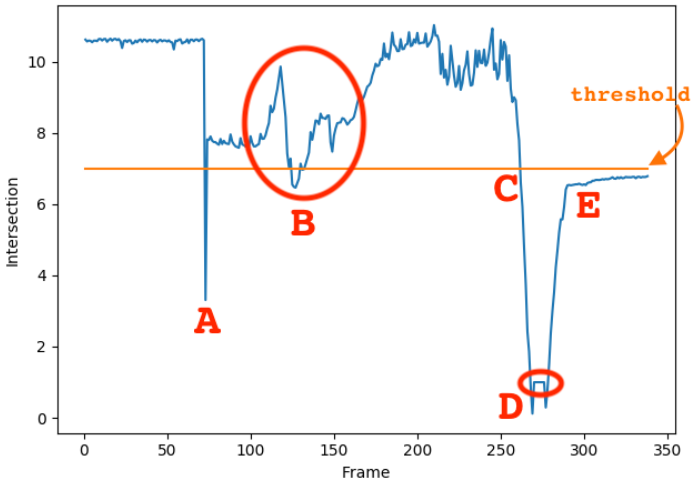

Figure 5.14: Example of the shot boundary detection algorithm on the test video from Figure 5.12 using the Intersection metric to compare the RGB histograms between consecutive frames.

especially efficient when using for long videos as the goal is to extract a fraction of the frames rather than generate an exhaustive list of all the shots and scenes in a video. As mentioned earlier in Section 5.1.1, this algorithm is not included in the actual pipeline flow as short videos made of single shots and ranging from 7 to 14 seconds are used. However, it is used when testing with longer videos comprised of multiple shots. Supporting code listings covering the code used for this phase can be found in the appendix, including the initial function controlling the execution process of this phase in Appendix H.1.4 and the shot boundary detection algorithm in Appendix H.2.9.

5.4 Complete System Pipeline Flowchart

Based on the detailed implementation of the system, a flowchart depicting the main aspects of the system can be found in Figure 5.16 towards the end of the chapter.

5.5 Development Testing

This project does not implement a traditional testing suite with unit tests, integration tests and acceptance tests (Priority 3 Requirement F19). Rather than spending time writing an extensive testing suite, which is not very useful considering the nature of the project, the program is repetitively tested before committing changes to GitHub. The test is carried out by using one of the database videos as the query video, which should always match that same

video with 100% accuracy (only true positives, no false positives) and no other potential matches listed (only one video in the histogram). An example of this test is portrayed in Figure 5.15. If an accuracy of 100% is not achieved, then the system is debugged to rectify mistakes and ensure an accuracy of 100% when using a database video as the query.

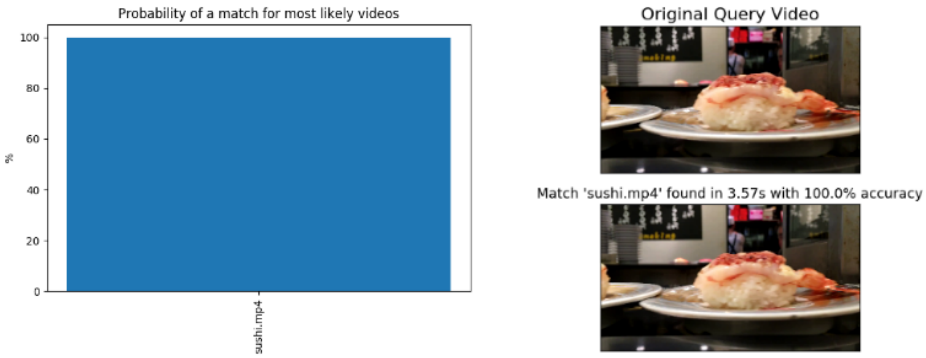

Figure 5.15: Testing that the system functions as expected before committing changes by using a database video as the input query.

5.6 Code Structure

This section covers the project structure of the implemented code, detailing what each directory and file represents. The code, along with the database and testing queries, can be found online on GitHub: `https://github.com/Adamouization/Content-Based-Video-Retrieval-Code`.

- *"app"* directory: contains all the Python code used to implement the system.
  - *"main.py"* file: the starting point of the program. Parses command line arguments to determine whether to run the off-line feature extraction phase, the on-line retrieval phase, or the database pre-processing phase.
  - *"histogram.py"* file: contains the *HistogramGenerator* class with functions to generate, average, store and compare greyscale, RGB and HSV histograms.
  - *"video_operations.py"* file: contains classes for video-related operations. The *ClickAndDrop* class is used for manually selecting the Region of Interest and the *VideoStabiliser* is used to stabilise the query video.

  - *"helpers.py"* file: contains multiple general-use functions such as retrieving file names, removing file extensions from filenames, displaying the final results in the form of plots or console outputs, parsing command line input and getting video information. The code for the various functions in this module can be found in Appendix H.4.
  - *"config.py"* file: contains global variables whose values are set by the command line arguments.
  - *"__init__.py"* file: default file required in any python package (cannot be deleted without causing errors).

- *"footage"* directory: where all the database of videos is located. A greyscale, RGB and HSV histogram is generated for each video in this directory. The query's histograms are then compared to each video's histograms from this directory.

- *"histogram_data"* directory: where the averaged greyscale, RGB and HSV histograms are stored as plain text files.

- *"recordings"* directory: where all the pre-recorded query videos used to test the system are placed.

- *"results"* directory: where general result-related files such as CSV and PNG files are placed.

- *"requirements.txt"* file: the different technologies and third-party libraries required to run the code. The file is used by the Pip[7] system to install everything in a single installation.

- *"README.md"* file: the instructions on how to setup and run the program locally.

5.7 Summary

This chapter detailed different steps that were followed to implement the different phases of the system. The off-line colour-based feature extraction phase was first looked at in detail, explaining how the greyscale, RGB and HSV histograms are generated for each database video before being averaged and saved into plain text files. The on-line retrieval phase was next dealt with, revealing the steps followed to pre-process the query video before matching it to one of the database videos through the use of distance metrics and displaying the final result. The database pre-processing phase was inspected last, describing how each database video can be segmented into a fraction of its original

[7] PyPi: `https://pypi.org/`

size by using a shot boundary detection algorithm. The chapter ended with a detailed diagram betraying the complete system's flowchart, followed by a list of the code's structure and ending on a brief note about testing. With the system completed and working as intended with all the requirements met, the next chapter will test and evaluate the accuracy, quality and scalability of the results for each phase of the system pipeline.

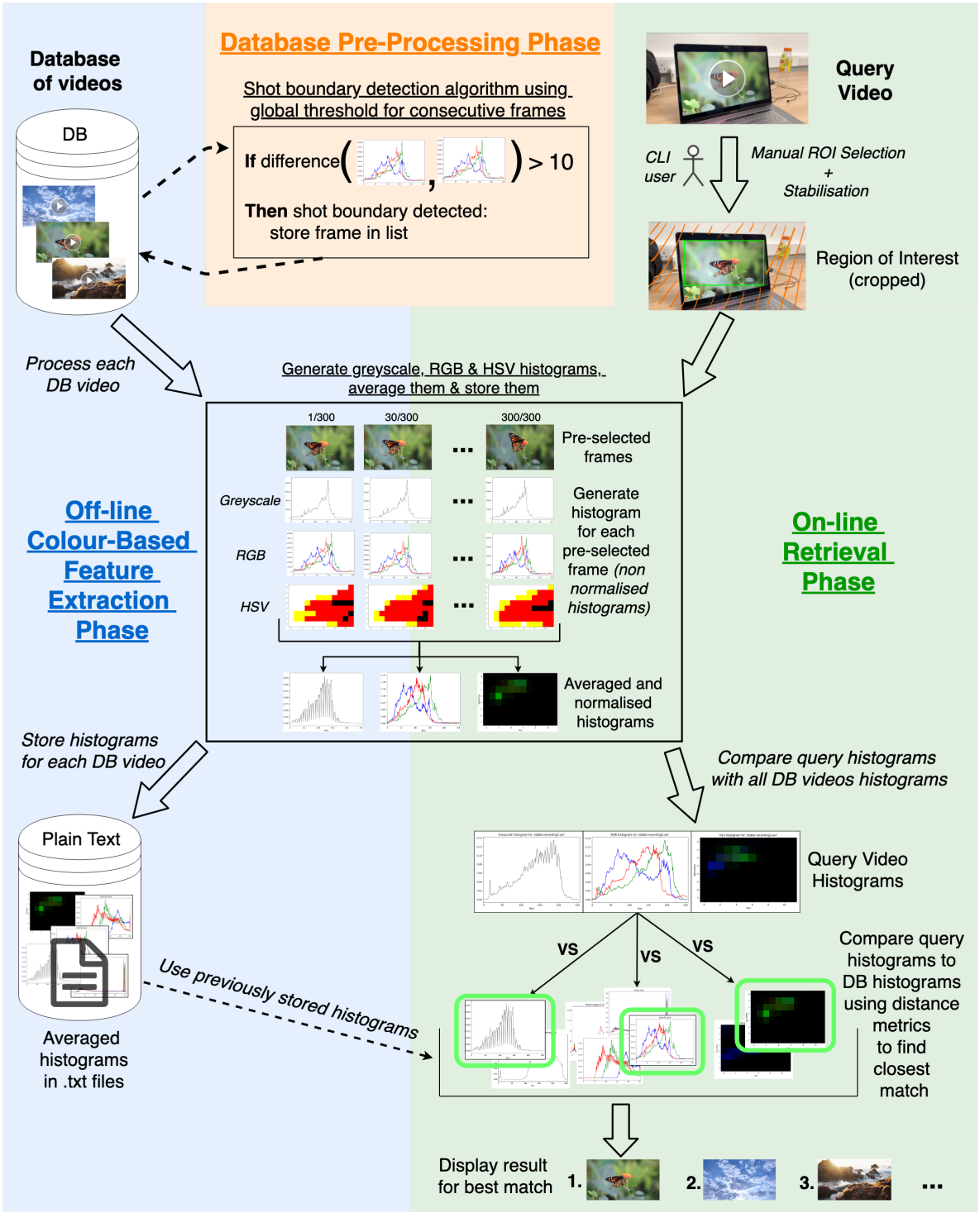

Figure 5.16: Complete flowchart of the system pipeline.

Chapter 6

Testing & Evaluation

This chapter aims to review how the system is tested in order to assess its quality as a whole. The three phases of the pipeline are analysed separately, with an emphasis on the on-line retrieval phase that produces the output of the system. The type of database videos and query videos used to test the system is clarified before diving into the evaluation of the results, starting with the on-line retrieval phase, followed by the off-line feature extraction phase and ending with the database pre-processing phase. These phases are analysed with the help of software evaluation tools, along with figures and graphs utilised for visual representations of the results. Finally, an online experiment is conducted to establish ground truth matching and compared to the system's result.

6.1 Testing Data

6.1.1 Database Videos

The system is tested using a database of 50 different videos, ranging from 7 to 14 seconds, as per requirements F25-F26. The database of videos is populated from scratch using free stock footage from *Pexels Video*[1]. The videos used to build the database are chosen from a rich library of videos encapsulating many different colours (e.g. bright, dark, warm, cold, colourful, etc.) and movements (e.g. still, motion, shaky, blurry, timelapses, etc.) to ensure diversification in the dataset. The database of videos used to test the system can be found in the *"footage"* directory in the provided code.

Employing existing databases of videos for CBVR tasks such as the TRECVID[2] dataset (Awad et al., 2018), which is the best dataset for CBVR-oriented tasks, would have been ideal for measuring the project's performance and for com-

[1]Pexels Video: `https://www.pexels.com/videos/`

[2]Text REtrieval Conference Video Retrieval Evaluation

paring it with existing solutions for this task. However, these datasets are not publicly available and hard to obtain. For example, most of the referenced proceedings in the Bibliography that present solutions to CBVR tasks were conducted for the TRECVID conferences, meaning that the datasets used to test the systems were provided to the participants for testing and evaluating results. However, the NIST[3] does not provide these databases for external research. Other databases of videos exist but target different computer vision tasks such as facial recognition or image retrieval rather than CBVR. Therefore, the previously mentioned custom database of 50 copyright-free videos is used to test this system.

6.1.2 Query Videos

Various query videos are recorded to test the on-line retrieval phase of the system. These queries are mobile recordings of one of the database videos. Different types of queries listed in Section 4.2.2 are used to test the limits of the system, including down-scaled (recording at a distance from the screen) and skewed queries (recording at an angle) with minor camera movement due to shaking hands. These conditions ensure the realism of the queries if the system were to be developed as a mobile application.

6.2 On-line Retrieval Phase Results Evaluation

The results are evaluated by counting the number of true positives and false positives for each video query used to test the system. A true positive corresponds to an outcome where the system matches the query to the correct database video, while a false positive is an outcome where the system matches the query to an incorrect database video. The number of true positives and false positives occurrences are counted for each histogram model and distance metric used per query. As detailed in the previous chapter at the end of Section 5.2.2, these are used to plot the probabilities of the most likely database video to match the query in the form of a percentage %. Additionally, other results are evaluated such as the runtime of the system, its scalability, and the comparison of the final results under different conditions.

6.2.1 Video Matching Performance

Accuracy Measurements

The system is first tested with down-scaled queries recorded at a straight angle to the screen. Shaky camera movement can be seen on the query videos, along

[3]National Institute of Standards and Technology, organiser of the annual TRECVID conference

with a considerable section of frame area not covering the actual video playing due to the distance to the screen. Despite these challenges, the queries still yield correct high-accuracy matches exceeding 50% of true positives, with some excellent results that exceed 90%. Figure 6.1 depicts two examples of queries that incorporate the specified challenges, with the first query retrieving the correct video with 93.18% accuracy, and the second one with 56.82%. The critical detail to notice in the left-hand side histograms is how the probability of the second closest match is much smaller than the probability of the closest match. Additionally, few alternative videos are marked as potential matches, with the first query only displaying two potential matches and the second query four potential matches.

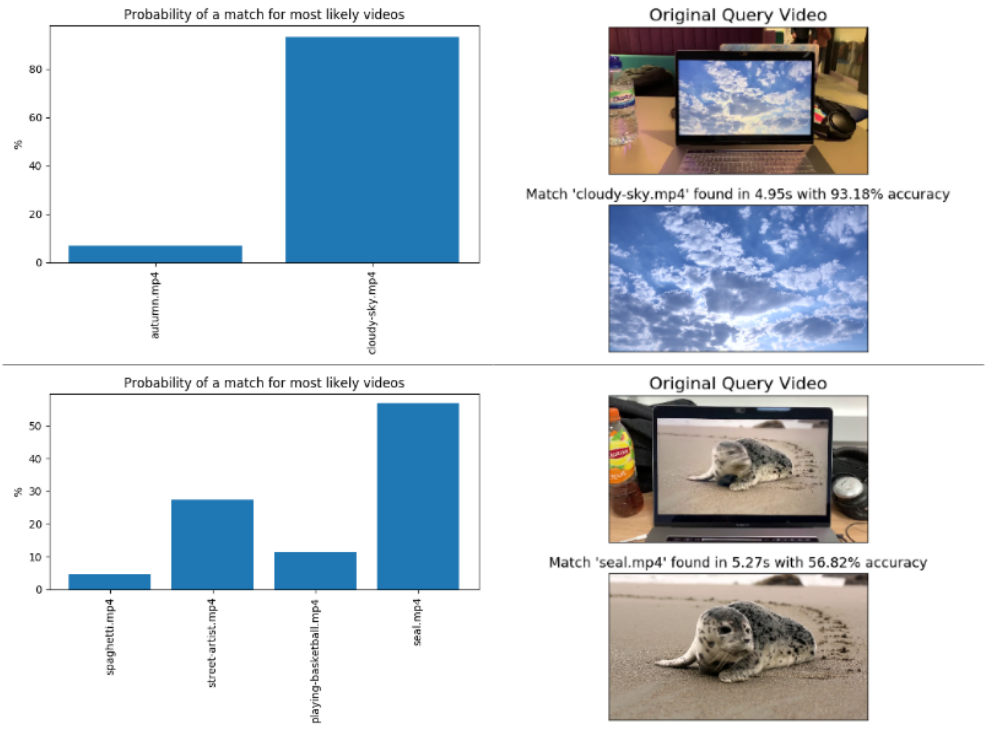

Figure 6.1: Examples of results using down-scaled queries (recorded at a distance).

Next, skewed queries recorded at an angle from the screen playing the database video are tested. These tested queries also include all of the challenges mentioned in the first test, such as down-scaling (recording at a distance) and unstable footage. Despite the increased challenge presented by the skewed query, the accuracy of the matches still exceed 50% and reach as high as 75% despite poor quality queries. Figure 6.2 portrays examples of skewed queries. The first query filmed from the left side of the screen is identified

as the correct match with 56.82% accuracy, while the second query filmed from the right side of the screen is identified as the correct match with 75% accuracy. Observing at the left-hand side histograms in Figure 6.2, it can be noticed that skewed queries cause the system to produce more potential video matches. Indeed, the first query produces six potential videos matches, which are quite different from the mainly-green *"butterfly"* query, e.g. the main colours in the *"autumn"* and *"dunes"* are not green. This contrasts with the previous non-skewed queries that produce less potential matches. Nevertheless, the probability of the closest match remains greater than the probability of the second closest match despite the skewed query.

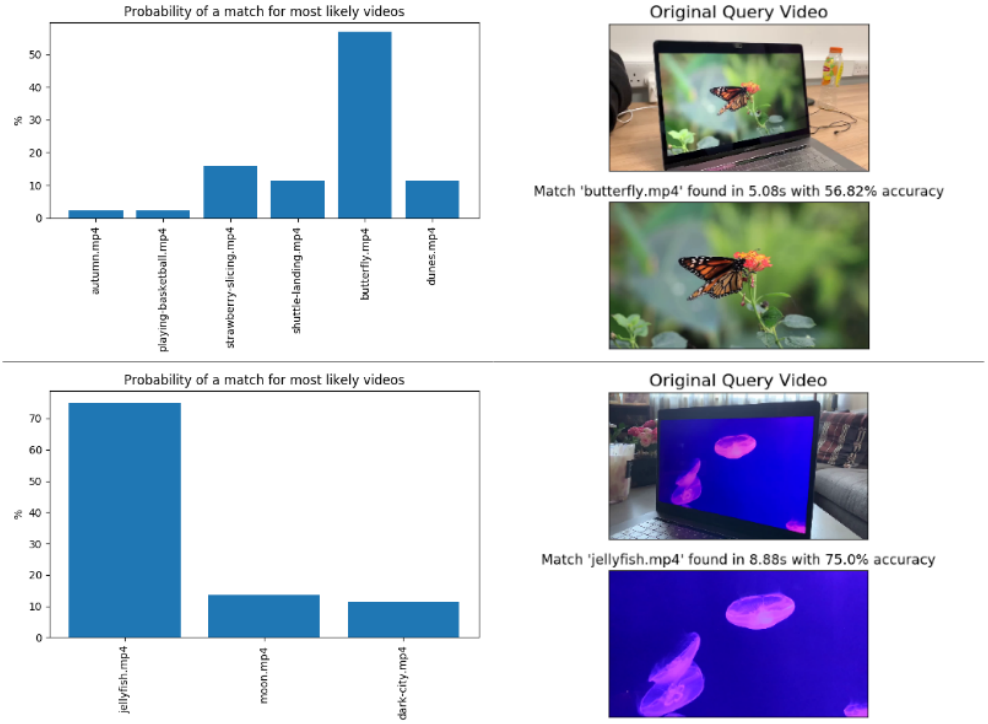

Figure 6.2: Examples of results using skewed queries (recorded at an angle).

Although the results demonstrated in the first two tests using down-scaled and skewed queries portrayed positive results with more than 50% true positives, some queries result in a more mediocre accuracy, reaching a minimum of 45.5% for a correct match, as shown in Figure 6.3. This low accuracy is partly caused by a combination of factors that are further explored in the next paragraph.

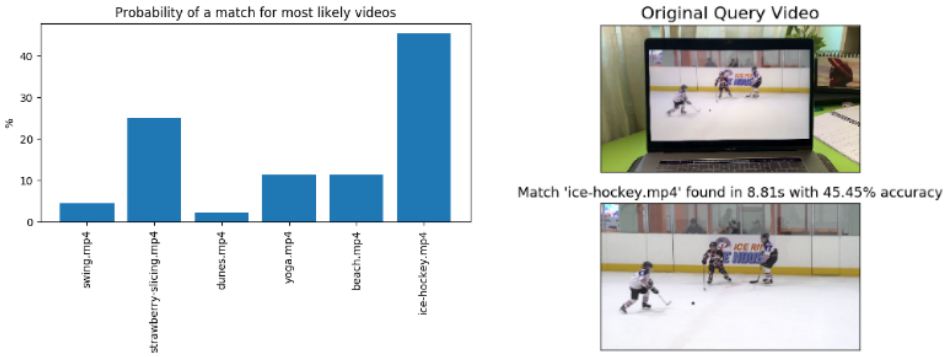

Figure 6.3: Examples of results with poor accuracy.

External Factors Observations

External factors directly affecting the quality of the query can drastically influence the accuracy of the system and even produce wrong matches. These factors are caused by the conditions of the environment rather than by the actual recording's quality.

The first noticeable environmental factor that causes the system to produce low-accuracy results and possibly wrong matches is the lighting in the room. While testing the system with different queries, the system had trouble coping with queries recorded at night-time in low luminosity environments. Indeed, during the filming of these queries, the lamps in the room produced warm light with colour temperatures ranging between 2000K and 3000K[4], which corresponds to orange/yellow light. These caused an alteration of the overall colour of the recorded screen, leading to the histograms to shift towards yellow/orange colours. This shift provoked the query's average histograms to be very disparate from the original database video's histograms, ultimately causing the system to produce the wrong input. Two cases of the consequences of this type of environment are illustrated in Figure 6.4.

The query from the first case (top of Figure 6.4) is matched to the correct database video, but with a very low accuracy of 45.45%. Furthermore, the system found five other potential matches, most with shades of yellow/orange colours (e.g. *"dunes"*, *"bird-walking"* and *"offroad-car"*). The second query used from Figure 6.4 does not even find the correct match, pairing the *"butterfly"* query to the *"bird-walking"* video. Moreover, the system does not list the correct video as a potential match and places the closest and second closest matches within 5% of each other. After watching the second query of the but-

[4]What is colour temperature? `http://www.westinghouselighting.com/color-temperature.aspx`

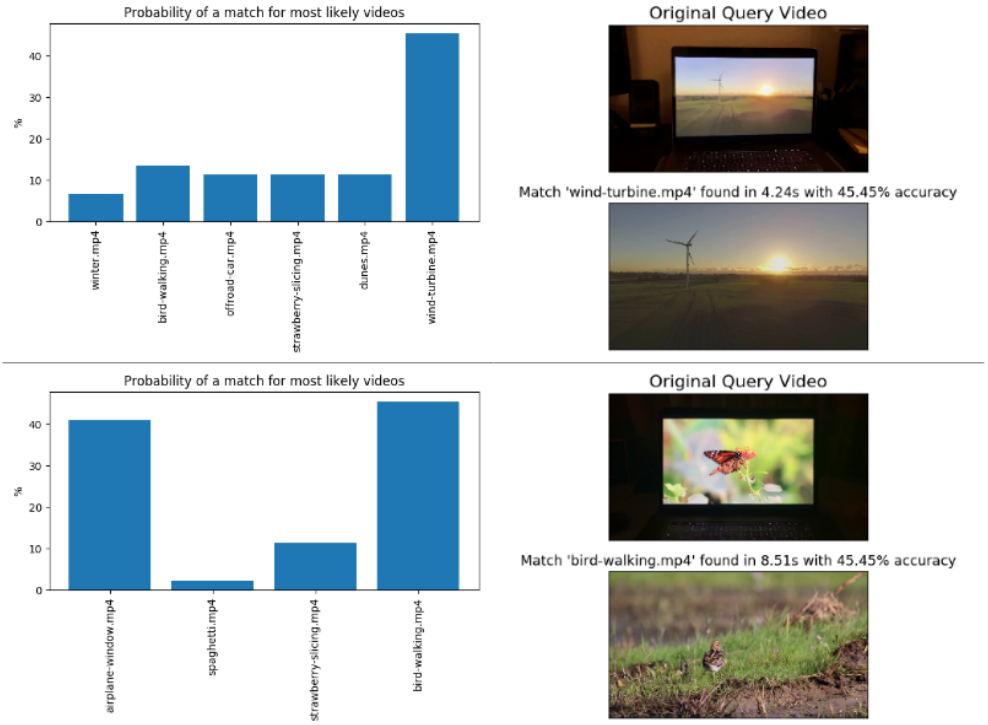

Figure 6.4: Examples of results using queries recorded in a low-light environment.

terfly in more detail, it can be noticed that the dark environment is the source of another negative factor. Due to the poorly lighted room, the luminosity of the screen stands out more, which can, in turn, shift the intensity of the pixels in the histogram towards the right (toward a value of 255). This contributes to the poor accuracy of the system in such conditions.

The second noticeable external factor is the presence of light reflections on the screen displaying the database videos. This factor, which is already probed in Section 4.2.2 Figure 4.2, depends on the positioning of the light source and the camera recording the query. Based on these, light glares might appear on the screen, significantly affecting the accuracy of the system as new colours appear on the screen, causing the histograms' pixel intensities to converge towards that new colour. Two cases reproducing the consequences of the light glares on the screen are represented in Figure 6.5.

The first query of *"people-dancing"* is correctly matched to its database video regardless of the light glare on the right-hand side of the screen (highlighted in red), but again with a low accuracy of 45.45%. The system detects potential matches in videos that are quite distant to the query video, such as

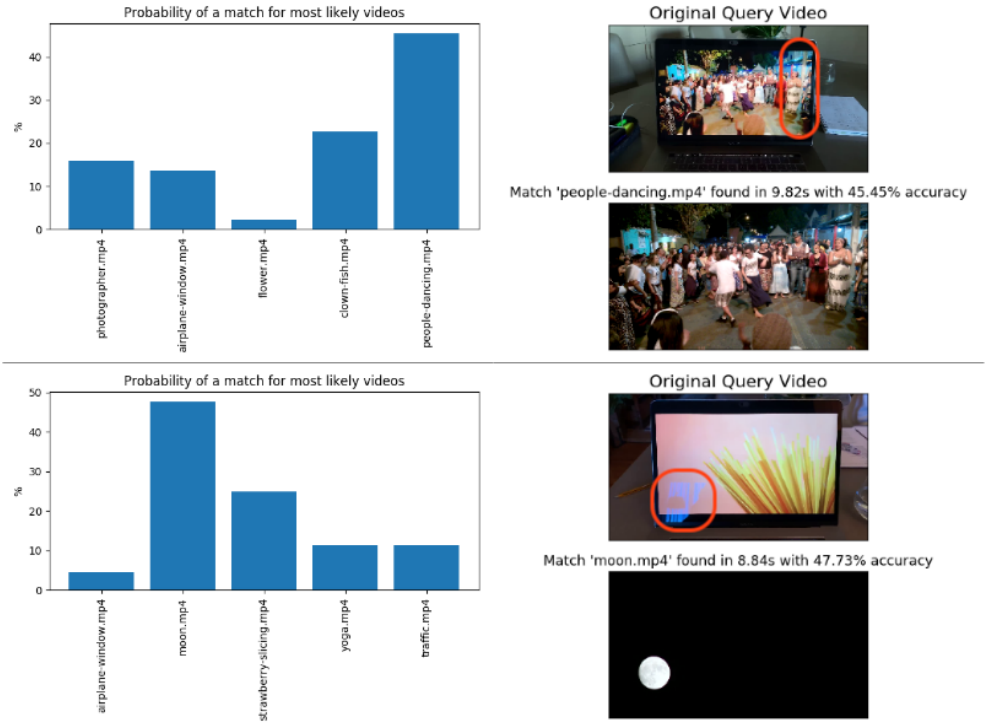

Figure 6.5: Examples of results using queries recorded with light reflections on the screen.

"airplane-window" or *"flower"* which both have shades of blue similar to the glare. The second query of *"spaghetti"* is matched to the wrong video, and is not even listed as a potential match despite its unique combination of yellow and pink colours because of the more substantial light glare in the bottom right of the screen (highlighted in red).

This section depicted the worst case performances of the system under challenging conditions, all caused by different lighting factors such as the colour of the light or the reflection of light on the screen playing the database videos. The next section critiques the combination of histogram models and distance metrics in more detail.

6.2.2 Histogram Models and Distance Metrics Analysis

Reflecting on the results laid out in the previous section, a few analytical words can be shared regarding the combination of the histogram models and the distance metrics used to compute the results.

Figure 6.6: Comparison of the accuracy before and after removing metrics such as the KL Divergence and alternative Chi-Square distance.

First, regarding the histogram models, the HSV histogram model was by far the most performant one, followed by the RGB histogram model in some cases and the least performant greyscale histogram model. For every single correct result depicted in the previous chapter, the HSV histogram models coupled with the Earth Mover's Distance and the Energy Distance always found the correct match[5]. The leading cause of false positives originates from the greyscale and RGB histograms often mismatching the query video. This explains the reason why 45.45% is the minimum accuracy in the results, as the combined weight of the two HSV models is worth 20 according to Table

[5]Excluding the two mismatches caused by yellow light and light reflections on the screen.

5.1. With 44 different weights assigned across the models and distance metrics, $20/44 = 0.4545$, which corresponds to the minimum accuracy produced by the system. If only the HSV were to be used, then results with 45.45% accuracy would reach 100%.

Regarding the distance metrics used to calculate the difference between the query's histograms and the database videos' histograms, some performed better than others. EMD and Energy Distance obviously performed better than the Correlation, Intersection, Chi-Square and Hellinger Distance due to their association with the HSV histogram models. However, all of these performed efficiently to some extent. The interesting results to analyse are the metrics excluded from the system that were used in earlier versions before being removed. Amongst these distances were the Kullback-Leibler Divergence and an alternative to the Chi-Square distance. These metrics never found the correct match, thus lowering the overall accuracy of the system due to the higher number of false positives. Significant improvements were made, with some queries resulting in 21% increases in true positives when excluding these two metrics. An example of the improvement made by removing these metrics can be seen in Figure 6.6, as they were not meant to be used as distance metrics for colour features.

6.2.3 Runtime Measurements

The runtime of each of the ten query videos used to test the system in the previous sections is plotted in a graph which can be found in Figure 6.7.

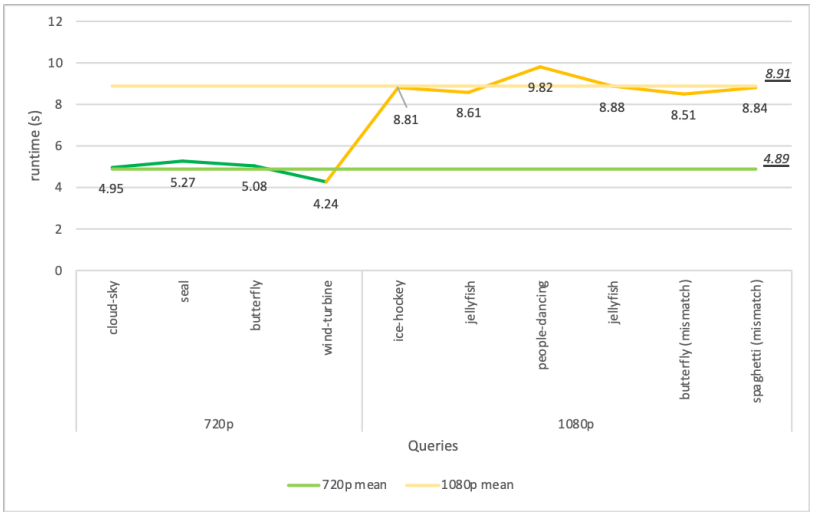

Figure 6.7: Graph depicting the runtime for each query used to test the system, along with the resolution of each query.

A few points can be made based on this graph. First, the runtime never exceeds 10s, thus fulfilling requirement F13. Second, the difference between the 1080p query videos and lower-quality 720p query videos' runtime is almost halved, with the runtime of 720p queries averaging 4.89 seconds and the runtime of 1080p queries averaging 8.91 seconds. However, the accuracy of the system when processing the 720p queries is as high as the accuracy of the 1080p queries. Indeed, the *"cloudy-sky"* query yields an accuracy of 93.18%, while the accuracy of the *"jellyfish"* query yield 75% true positives. This vital measurement reveals that using higher-quality does not necessarily translate to better results, meaning that any decent mobile device camera can be used with the system to generate descriptive histograms.

6.3 Off-line Feature Extraction Phase Scalability

Evaluating the off-line colour-based feature extraction phase is more tricky than evaluating the on-line retrieval phase as the output is always the same between different runs: three averaged histograms for each database video are generated and stored in plain text files. This phase is nevertheless essential as it dictates how large the database of videos can be. Therefore, a quantitative measure can be used to determine the scalability of this phase and how large the database can be.

The measure used to estimate the scalability of the system is the runtime of the phase. The mean runtime for generating the three averaged histograms for each video in a database of 50 videos is 163 seconds. With the way the features are extracted and the feature vectors are generated, the growth of the system is linear as it is equal to $163/50 = 3.26$ seconds per video. Adding a 51st video to the database will increase the runtime by 3.26 seconds. To predict how large the database of videos can be, the runtime in hours is plotted in a graph starting with 50 videos and growing up to one million videos based on the previous linear calculations. The results can be found in Figure 6.8.

These results clearly betray the demands of processing large databases of videos. Up until 1000 videos, the runtime of processing the database would be inferior to one hour. The runtime reaches 9 hours for reaching 10000 videos, which is still acceptable to calculate in a single execution. However, the runtime to process more extensive databases would need to be calculated in days rather than hours when the database reaches one million videos, with runtimes of $905.61/24 = 37.73$ days. This would be feasible with dedicated hardware running this training phase to generate the feature vectors for each database video in multiple batches through long periods of times.

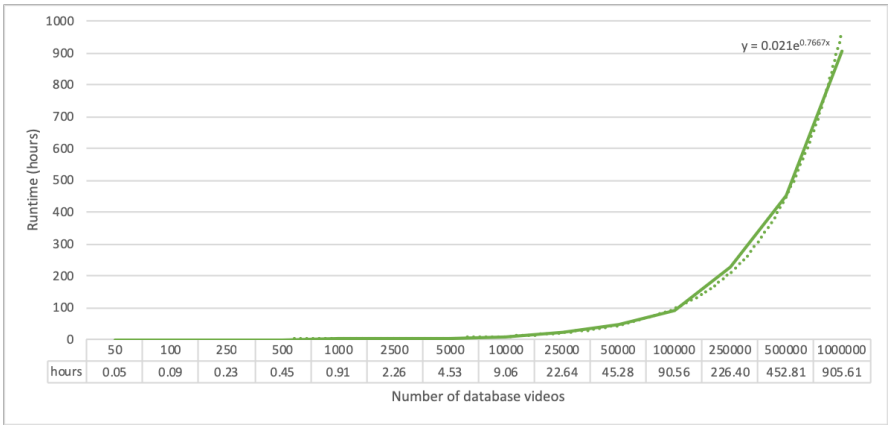

Figure 6.8: Prediction of the off-line colour-based feature extraction phase runtime for different database sizes. A trendline $y = 0.021e^{0.7667x}$ can be fitted to the resulting series to determine the demand

6.4 Database Pre-Processing Test: Movie Segmentation

The initial motivation behind the project being to create a CBVR system for feature-length movies, the phase dedicated to pre-process the database of movies is tested with a feature-length movie to observe how it can be segmented using the implemented shot boundary detection. The movie used for the test is Inception (2010), a 2h28 movie composed of 213098 frames in total. The shot boundary detection algorithm is tested once with the KL Divergence and once with the Intersection metric, as depicted in Figure 6.9.

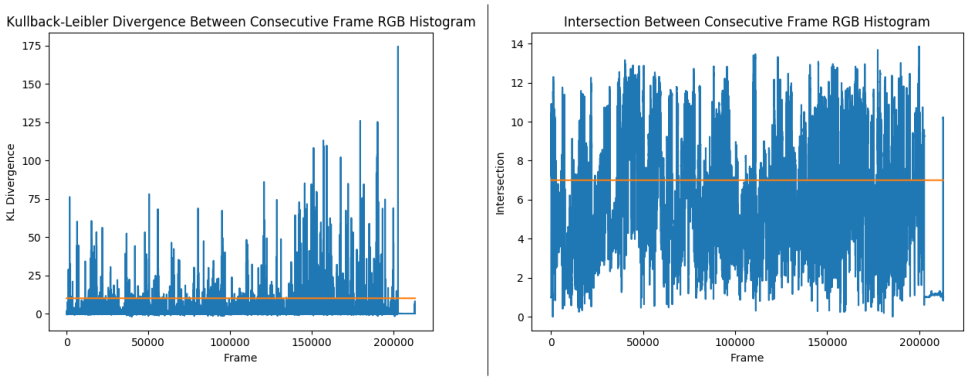

Figure 6.9: Result of the shot boundary detection algorithm on the Inception movie using the KL Divergence (left) and Intersection metric (right) between consecutive frames' RGB histograms.

The first test uses the KL Divergence to detect the difference between consecutive frames while employing a global threshold $t = 10$, which detects a total of 661 different shot boundaries throughout the entire film in 2h29. Alternatively, the second test uses the Intersection metric while employing a global threshold $t = 7$, which detects a total of 1730 shot boundaries 2h41. Although the difference between the two results seems considerable as the Intersection metric detects almost three times more shot boundaries than the KL Divergence, the difference between the two is actually quite small when considering the bigger picture. Indeed, the 661 shot boundaries represent only 0.31% of the movie, while the 1730 shot boundaries represent 0.81% of the movie. When considering that the goal of the database pre-processing phase is to segment the database videos in order to have fewer frames to process in the off-line feature extraction phase, then cutting down the number of frames to process to less than 1% is a satisfactory result that will significantly improve the results when working with long videos.

6.5 Comparison With Ground Truth Experiment

An experiment with 20 participants was conducted to establish ground truth video matching. The experiment was conducted online through the use of Google Forms[6]. During the experiment, the users were first presented six different database videos to simulate the off-line colour-based feature extraction phase. Next, they were shown one query video and asked to rank the six videos they watched from most likely to least likely to match the query video in order to simulate the on-line retrieval phase. The same six database videos and query video were then used with the system to compare with the participants' rankings. To render the matching task compelling, the six videos, which can be found in Figure 6.10, all have different shades of blue.

Figure 6.10: The query video and the six database videos shown to the participants.

[6]Google Forms: `https://www.google.com/forms/about/`

The results of the online experiment along with the results of the system are plotted in Figure 6.11. The most essential aspect between the experiment's results and the systems results is that both ranked the correct video (*"cloudy-sky"*) as the most likely video to match the query with 100% accuracy, meaning that every participant matched the query to the correct database video, and so did the system for each histogram model and distance metric calculation. This confirms with certainty that *"cloudy-sky"* is the correct match to the query. However, while observing the five other matches, it can be seen that they are not ranked in the same order between the ground truth and the algorithm. On the one hand, the results of the ground truth experiment ranks the other videos as 2: *earth* 3: *shuttle*, 4: *winter*, 5: *sunset* and 6: *beach*. On the other hand, the results of the system ranks the other videos as 2: *winter*, 3: *earth*, 4: *shuttle*, 5: *beach*, 6: *dusk*. Interestingly, despite the different rankings between the other videos, a parallel can be drawn between the accuracy for each match. Indeed, the participants and the system ranked the second closest match with 50% accuracy, while they ranked the third closest match with the lowest accuracy (25% for the participants and 22.7% for the system). The common points between the ranking accuracies are through-provoking as they indicate similar behaviour between the participants and the system. As a matter of fact, the participants were asked what visual aspect of the video they considered the most when ranking the videos, and 80% of them stated that they considered the colours of the query video to match it to the six other videos. This explains the similar accuracies between the matches, but the discrepancy in the ranking order could be explained by the fact that 70% of the participants considered visual aspects other than colour when ranking the videos, such as motion, shapes, objects, sound and even semantic meaning.

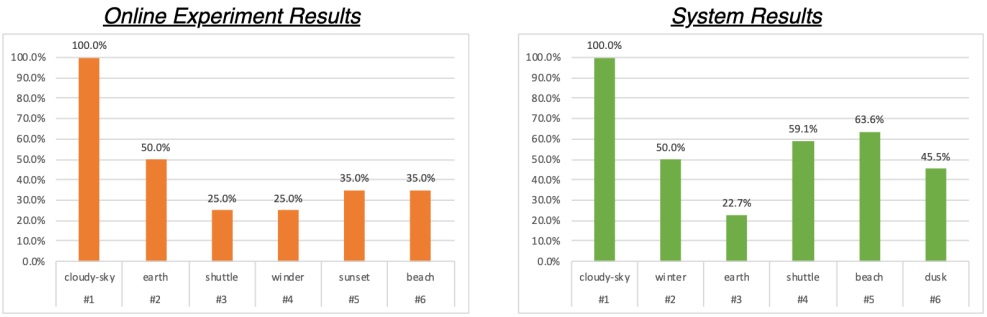

Figure 6.11: Comparison of the online experiment results and the system's results.

Supporting documents covering the online experiment can be found in the appendix, including the Ethics Checklist in Appendix E, the Experiment Survey in Appendix F and the experiments results in Appendix G.

6.6 Summary

This chapter analysed the different results obtained when testing the system. The system was tested with a database of fifty short videos, with the runtime used to measure the scalability of the off-line feature extraction phase. Next, nine different queries were used to test the on-line retrieval phase through different conditions before using a feature-length movie to test the database pre-processing phase. With the system successfully tested and objectively evaluated, the final chapter of this report will look back upon the different stages of the project to discuss what was successful and what could be improved.

Chapter 7

Conclusions

This chapter summarises and reflects upon the project as a whole, discussing the various aspects that went well and those that can be improved. The initial project aims defined in the Introduction in Section 1.4 are first reviewed, followed by the future work for the project. Finally, the chapter concludes with an analysis of the limitations of the current project and a future commercial application before ending with general reflections about what was accomplished and learned through this project.

7.1 Achievements

The main goal of this project was to design and implement a prototype system to tackle the problem of exponential unstructured data growth in the form of a content-based video retrieval system oriented towards matching mobile-recorded video queries to a movie. After investigating a wide array of potential feature extraction and comparison methods to combine in order to come up with a unique solution, a functional system divided into three phases was created. The first phase processed the videos in the databases to generate feature vectors by extracting colour features, generating averaged histograms and storing them in plain text files. The second phase repeated the process with an input query video that is matched to one of the database videos by calculating the distances between the videos' feature vectors. Finally, the third phase was dedicated to making the system suitable for movies by segmenting the videos in the database through a shot boundary detection algorithm.

This pipeline of three phases that constituted the system produced positive results with a custom database of 50 videos ranging from 7 to 14 seconds. Indeed, it accurately found the correct matches to queries recorded with a mobile device that included undesired camera movement and poor framing (recorded at a distance and an angle from the screen) with accuracies reaching a maximum of 93.18%. However, the system fell short when dealing with

unfavourable environmental conditions such as the non-white light sources in dark environments and light reflections glaring on the screen, with a minimum of 45.45% accuracy and in some extreme cases, mismatches.

Unfortunately, the system could not be tested with a database of movies due to availability and legal copyright constraints, but a single movie was used to test the database pre-processing phase of the system. The movie was reduced to less than 1% of its original size thanks to the shot boundary detection algorithm, which could have been used as one of the database videos in the two other phases, thus showing the links between the pipeline phases. Overall, almost every aim initially set in Section 1.4 was achieved, except for testing the system with a database of movies and creating an actual link between the database pre-processing and the off-line feature extraction phases which was only conceptualised.

7.2 Future Work

A lot of different aspects that were skipped due to time constraints can be added to the system in its current state to improve the runtime and accuracy in order to ultimately use more extensive databases and longer videos.

7.2.1 Current Pipeline Improvements

Many improvements could be accomplished while keeping the original system's design concepts laid out in Chapter 4, including the colour-based features and distance metric calculations. The first change involves the histograms. Rather than generating averaged global histograms for each shot, region-based histograms should be generated instead. For example, the frames used to generate the histograms can be divided into several regions, and an average histogram could be generated for each region. This would ensure a much higher level of accuracy when matching the query video to a database video as the histograms no longer describe the colour distribution throughout the entire image but for a smaller localised region. For example, if the two frames contain both 40% of blue, where one corresponds to the sky and one to the sea, then a global histogram would not be able to differentiate the two frames easily, whereas a region-based histogram would as the blues would not be in the same region. An illustration of a frame segmented into five distinct regions can be found in Figure 7.1.

Additionally, histogram models that did not produce accurate results, such as the greyscale and RGB histograms, could be excluded from future versions. Instead, only HSV histograms and new histograms with similar levels of com-

Figure 7.1: Example of a frame segmented into 5 regions.

plexity such as HSL[1] histograms could be used to maintain high accuracy. As a matter of fact, considering the queries used in Chapter 6, if only the HSV histograms with the EMD and energy distance metrics were used, 100% would be achieved for every single query. One of the observations from Section 6.2.2 is that the minimum accuracy of 45.45% achieved is due to the HSV histograms outweighing the greyscale and RGB histograms thanks to the pre-assigned weights from Table 5.1. Therefore, using only similar histogram models and distance metrics would increase the performance of the system both in terms of accuracy and runtime since fewer histograms would need to be generated and fewer distances to be calculated.

7.2.2 Interface-Related Improvements

In terms of interface, simple changes can be carried out to make the system more usable, without the need to create a commercial mobile application yet. The first change is the interface itself, replacing the CLI with a GUI. A GUI would allow the results to be displayed more easily on the screen along with more straightforward interactions through button presses rather than console argument parsing. The generated data, such as the matched video, the histograms and the results tables would be more easily accessible and readable. Improving the system to a GUI could be done using either Tkinter, the native GUI library for Python if it remains the programming language used, or by rendering a clean HTML webpage.

Regarding the manual ROI selection, two improvements could be made. The first is a simple enhancement to allow four distinct points to be chosen to select the region of interest. Currently, allowing only two points limits the ROI to be a rectangle which cannot be rotated. Supporting four points would allow quadrilateral shapes other than rectangles to be drawn and rotated. This

[1] Hue Saturation Lightness

would significantly boost the performance of the on-line retrieval phase since skewed queries could be cropped without losing any information and queries with undesired artefacts such as light reflections on the edges of the screen could be easily pruned.

An improvement that was not implemented as it was outside the project scope is automatic ROI selection. Eventually, in a commercial application, edges could be detected using horizontal and vertical Sobel operators to outline the contour of the screen, and once the edges are determined pixels outside the area could be excluded. This could not be implemented due to the sheer complexity of integrating such a feature, which could be a dissertation project on its own.

7.2.3 Fundamental System Redesign for Large-Scale Databases

Moving away from the current system pipeline developed as part of this project, fundamental changes need to be applied. The simple choice of features and model used to build the system functions well with small databases of short videos, but in order to build a system that could work with larger databases and movies, significant modifications to these need to be made. More advanced features such as shape-based, motion and object features need to be extracted rather than extracting only colours. However, the main change required to build a commercial application that would in theory work with vast databases of movies is a deep learning system, such as a neural network, which functions well with large amounts of data to process.

7.3 Limitations

7.3.1 Project Limitation

The main limitation of this dissertation project is the time constraint. Due to the short amount of time available to design and implement a functional system, many compromises had to be made to develop a prototype that could be tested. For this reason, plenty of various aspects had to be either be simplified or skipped. The first main compromise came within the features to extract from the video. As mentioned in the literature review, a lot of potential features could be extracted to represent a video, ranging from static to dynamic features. Colour-based features were chosen, but complementing the system with dynamic features as well would have significantly increased its accuracy. However, these being much more complex to extract and compare efficiently, thus could not be included in the code. A second compromise made was regarding the learning model. The simplest option available, corresponding to storing the extracted features into histograms which were then compared using distance metrics to find the closest one to the query, was

elected to accommodate a small database. More sophisticated models such as deep learning models (e.g. neural networks, convolutional neural networks) could not be considered due to their complexity and the time required to implement them. The focus was to create a functional system by developing the different phases of a pipeline without necessarily coding all the intermediate steps. For instance, third-party libraries were used for the essential computer vision functions and the video stabilisation, while other aspects were simplified such as an intuitive interface replaced by a simple command line interface or automatic ROI selection replaced by manual cropping.

7.3.2 Commercial Application Limitations

The leading limitation of a future marketable application to match movies lies within the nature of the data itself. Indeed, videos, and in particular movies, are substantial objects. No matter the efficiency of the algorithm, processing them requires a lot of time and computing power, which is not always available. This fact is verified with the evaluation test carried out in Section 6.4, where pre-processing a single feature-length movie took an average of 2 hours and 35 minutes. This step includes the actual visual content extraction to represent a movie with features. Given a database of at least 100,000 movies, which would be a realistic number to start a commercial movie matching application, the amount of time to pre-process and later train the model would be colossal and unrealistic. Additionally, publicly available databases of movies do not exist due to copyright and legal issues, as most of them are owned by studios that only sell viewing rights to streaming services. Therefore, coming up with a database of movies with enough relevant titles to make the application interesting would be a challenge on its own.

7.4 Project Summary & Reflections

This project focused on multiple interests of mine by combining movies with programming. The main inspiration for this project originated from the goal to re-create the music matching application *"Shazam"* for movies. With the main limitations mentioned in Section 7.3 already known before starting the project, it was an exciting challenge to explore how the system could be designed and to create a functional prototype that could be tested and showcased to teachers, friends and family.

The code developed for this dissertation can be found online at the following URL: `https://github.com/Adamouization/Content-Based-Video-Retrieval-Code`.

Bibliography

Amir, A., Berg, M., Chang, S.-F., Hsu, W., Iyengar, G., Lin, C.-Y., Naphade, M., Natsev, A., Neti, C., Nock, H. et al. (2003), 'Ibm research trecvid-2003 video retrieval system', *NIST TRECVID-2003* **7**(8), 36.

Araujo, A. and Girod, B. (2018), 'Large-scale video retrieval using image queries', *IEEE Transactions on Circuits and Systems for Video Technology* **28**(6), 1406–1420.

Awad, G., Butt, A., Curtis, K., Lee, Y., Fiscus, J., Godil, A., Joy, D., Delgado, A., Smeaton, A. F., Graham, Y., Kraaij, W., Quénot, G., Magalhaes, J., Semedo, D. and Blasi, S. (2018), Trecvid 2018: Benchmarking video activity detection, video captioning and matching, video storytelling linking and video search, *in* 'Proceedings of TRECVID 2018', NIST, USA.

Blumberg, R. and Atre, S. (2003), 'The problem with unstructured data', *Dm Review* **13**(42-49), 62.

Bradski, G. and Kaehler, A. (2008), *Learning OpenCV: Computer vision with the OpenCV library*, " O'Reilly Media, Inc.", pp. 493–636.
URL: *http://shop.oreilly.com/product/0636920044765.do*

Brownlow, K. (1980), 'Silent films: What was the right speed?', *Sight and Sound* **Summer**, 164–167.
URL: *https://web.archive.org/web/20110708155615/http://www.cinemaweb.com/silentfilm/bookshelf/18_kb_2.htm*

Camara-Chavez, G., Precioso, F., Cord, M., Phillip-Foliguet, S. and Araujo, A. d. A. (2007), Shot boundary detection by a hierarchical supervised approach, *in* '2007 14th International Workshop on Systems, Signals and Image Processing and 6th EURASIP Conference focused on Speech and Image Processing, Multimedia Communications and Services', IEEE, pp. 197–200.

Cernekova, Z., Pitas, I. and Nikou, C. (2006), 'Information theory-based shot cut/fade detection and video summarization', *IEEE Transactions on circuits and systems for video technology* **16**(1), 82–91.

Chang, Y., Lee, D.-J., Hong, Y. and Archibald, J. (2007), 'Unsupervised video shot detection using clustering ensemble with a color global scale-invariant feature transform descriptor', *EURASIP Journal on Image and Video Processing* **2008**(1), 860743.

Deutscher, M. (2012), 'Big data is the new natural resource: New ibm tool powers decision making', `https://siliconangle.com/2012/04/03/big-data-is-the-new-natural-resource-new-ibm-tool-powers-decision-making/`. [Online] Accessed: 2018-10-28.

Fablet, R., Bouthemy, P. and Perez, P. (2002), 'Nonparametric motion characterization using causal probabilistic models for video indexing and retrieval', *IEEE Transactions on Image Processing* **11**(4), 393–407.

Fanning, J., Mullen, S. P. and McAuley, E. (2012), 'Increasing physical activity with mobile devices: a meta-analysis', *Journal of medical Internet research* **14**(6).

Feng, B., Cao, J., Bao, X., Bao, L., Zhang, Y., Lin, S. and Yun, X. (2011), 'Graph-based multi-space semantic correlation propagation for video retrieval', *The Visual Computer* **27**(1), 21–34.
URL: *https://doi.org/10.1007/s00371-010-0510-6*

Hanjalic, A., Lagendijk, R. L. and Biemond, J. (1999), 'Automated high-level movie segmentation for advanced video-retrieval systems', *IEEE Transactions on Circuits and Systems for Video Technology* **9**(4), 580–588.

Hauptmann, A., Baron, R. V., Chen, M.-y., Christel, M., Duygulu, P., Huang, C., Jin, R., Lin, W.-H., Ng, T. and Moraveji, N. (2004), Informedia at trecvid 2003: Analyzing and searching broadcast news video, Technical report, NIST.
URL: *https://www-nlpir.nist.gov/projects/tvpubs/tvpapers03/cmu.final.paper.pdf*

Heo, Y. S., Lee, S. and Jung, H. Y. (2016), 'Consistent color and detail transfer from multiple source images for video and images', *The Visual Computer* **32**(10), 1273–1289.
URL: *https://doi.org/10.1007/s00371-015-1162-3*

Hoi, S. C., Wong, L. L. and Lyu, A. (2006), Chinese university of hongkong at trecvid 2006: Shot boundary detection and video search, *in* 'TRECVid 2006 Workshop', Gaithersburg, Maryland, NIST, pp. 76–86.

Hu, W., Xie, N., Li, L., Zeng, X. and Maybank, S. (2011), 'A survey on visual content-based video indexing and retrieval', *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* **41**(6), 797–819.

Huang, J., Kumar, S. R., Mitra, M., Zhu, W.-J. and Zabih, R. (1997), 'Image indexing using color correlograms', *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition* **1**, 762.

IMDb (2018), 'Imdb statistics', `https://www.imdb.com/pressroom/stats/`.

Janwe, N. J. and Bhoyar, K. K. (2013), Video shot boundary detection based on jnd color histogram, *in* '2013 IEEE Second International Conference on Image Information Processing (ICIIP-2013)', IEEE, pp. 476–480.

Karr, D. (2012), 'Big data brings marketing big numbers', `https://martech.zone/ibm-big-data-marketing/`. [Online] Accessed: 2018-10-28.

Kurt, W. (2017), 'Kullback-leibler divergence explained', `https://www.countbayesie.com/blog/2017/5/9/kullback-leibler-divergence-explained`. [Online] Accessed: 2019-04-25.

Lai, Y. and Yang, C. (2015), 'Video object retrieval by trajectory and appearance', *IEEE Transactions on Circuits and Systems for Video Technology* **25**(6), 1026–1037.

Lars, K. (2012), 'Improving rights management at youtube', `https://youtube-creators.googleblog.com/2012/04/improving-rights-management-at-youtube.html`.

LeCun, Y., Bengio, Y. and Hinton, G. (2015), 'Deep learning', *Nature* **521**, 436–444.
URL: *https://doi.org/10.1038/nature14539*

Li, H. and Doermann, D. (2002), Video indexing and retrieval based on recognized text, *in* 'IEEE Workshop on Multimedia Signal Processing.', pp. 245–248.

Liu, W., Mei, T. and Zhang, Y. (2014), 'Instant mobile video search with layered audio-video indexing and progressive transmission', *IEEE Transactions on Multimedia* **16**(8), 2242–2255.

Lo, C.-C. and Wang, S.-J. (2001), 'Video segmentation using a histogram-based fuzzy c-means clustering algorithm', *Computer Standards & Interfaces* **23**(5), 429–438.

Manjunath, B. S. and Ma, W.-Y. (1996), 'Texture features for browsing and retrieval of image data', *IEEE Transactions on pattern analysis and machine intelligence* **18**(8), 837–842.

Okabe, M., Dobashi, Y. and Anjyo, K. (2018), 'Animating pictures of water scenes using video retrieval', *The Visual Computer* **34**(3), 347–358.
URL: *https://doi.org/10.1007/s00371-016-1337-6*

OpenCV (2015), 'Opencv: 3.0.0-rc1, open source computer vision, image processing, histograms', https://docs.opencv.org/3.0-rc1/d6/dc7/group__imgproc__hist.html. [Online] Accessed: 2019-04-25.

Park, F. (2011), 'Shape descriptor/feature extraction', https://www.math.uci.edu/icamp/summer/research_11/park/shape_descriptors_survey.pdf. [Online] Accessed: 2019-03-18.

Patel, B. V. and Meshram, B. B. (2012), 'Content based video retrieval', *The International Journal of Multimedia & Its Applications* **4**(5), 77–98.
URL: *http://arxiv.org/abs/1211.4683*

Petković, M. (2000), Content-based video retrieval, *in* 'Extending DataBase Technology PhD Workshop'.
URL: *http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.33.7317*

Puzicha, J., Hofmann, T. and Buhmann, J. M. (1997), Non-parametric similarity measures for unsupervised texture segmentation and image retrieval, *in* 'Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition', IEEE, pp. 267–272.

Sivic, J., Everingham, M. and Zisserman, A. (2005), Person spotting: video shot retrieval for face sets, *in* 'International conference on image and video retrieval', Springer, pp. 226–236.

Smeaton, A. F., Over, P. and Kraaij, W. (2006), Evaluation campaigns and trecvid, *in* 'MIR '06: Proceedings of the 8th ACM International Workshop on Multimedia Information Retrieval', ACM Press, New York, NY, USA, pp. 321–330.

TIOBE (2019), 'Tiobe programming community index definition', https://www.tiobe.com/tiobe-index/programming-languages-definition/. [Online] Accessed: 2019-04-11.

Villas-Boas, A. (2017), 'Google's new lens product uses your phone's camera to do clever tricks, like connecting your phone to a wifi network', http://uk.businessinsider.com/googles-lens-feature-can-connect-your-phone-to-wifi-using-your-camera-2017-5?op=1&r=US&IR=T.

Wang, H., Oneata, D., Verbeek, J. and Schmid, C. (2016), 'A robust and efficient video representation for action recognition', *International Journal of Computer Vision* **119**(3), 219–238.
URL: *https://doi.org/10.1007/s11263-015-0846-5*

Yan, R. and Hauptmann, A. G. (2007), 'A review of text and image retrieval approaches for broadcast news video', *Information Retrieval* **10**(4-5), 445–484.

Yang, J., Jiang, Y.-G., Hauptmann, A. G. and Ngo, C.-W. (2007), Evaluating bag-of-visual-words representations in scene classification, *in* 'Proceedings of the international workshop on Workshop on multimedia information retrieval', ACM, pp. 197–206.

Yuan, J., Wang, H., Xiao, L., Zheng, W., Li, J., Lin, F. and Zhang, B. (2007), 'A formal study of shot boundary detection', *IEEE transactions on circuits and systems for video technology* **17**(2), 168–186.

Zhao, Z.-C. and Cai, A.-N. (2006), Shot boundary detection algorithm in compressed domain based on adaboost and fuzzy theory, *in* 'International Conference on Natural Computation', Springer, pp. 617–626.

Appendix A

Potential Programming Languages Comparison

	Python	**C++**	**Java**	**MATLAB**
Familiarity	Very familiar: used in industry and personal projects	Unfamiliar	Familiar in the past: used in industry and coursework	Somewhat familiar
Support for video manipulation and computer vision functions	OpenCV library support	OpenCV library support	OpenCV library support	Native video manipulation and computer vision functions
Speed*	Slow** (interpreted)	Very fast (compiled)	Fast (compiled)	Slow (interpreted)
Code Readability	Simple syntax to follow, easy to read***	Complex and wordy syntax	Complex and wordy syntax	Complex syntax

Table A.1: Table comparing the main pros and cons for using different programming languages to build the system.

* Based on *Benchmarks Game*[1] and *"A Comparison of Programming Languages", J. Kinlay*[2].

[1] `https://benchmarksgame-team.pages.debian.net/benchmarksgame/`
[2] `http://jonathankinlay.com/2018/10/comparison-programming-languages/`

** Python code using OpenCV functions executes as fast as the original OpenCV library C++ code since it is the actual OpenCV C++ code that is running in the background, thus maintaining high-speed despite using Python[3].

*** Following the PEP8[4] coding style guide will ensure that code readable by professionals will be written, meeting the goal set by requirement F21 in Chapter 3.

[3]OpenCV Python Tutorials, `https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_setup/py_intro/py_intro.html#intro`

[4]PEP8 – Style Guide for Python Code: `https://www.python.org/dev/peps/pep-0008/`

Appendix B

Text and Binary Files Comparison

Text files	Binary files
Are human readable: text files are easier to debug since people can see what was written to the file.	Are not human readable: binary files are a lot more difficult to debug because people cannot easily see what was written to the file.
When dealing with large data, compression may be a factor. For example, a 10-digit number will take at least 10 bytes as text.	Will store the same data in less space. For example: a 10-digit number will take as little as four or two as binary.
More restrictive than binary files since they can only contain textual data.	Binary file formats are less restrictive as they may include multiple types of data in the same file such as image, video, and audio data.
Less likely to become corrupted. A small error in a text file may simply show up once the file has been opened.	More likely to become corrupted. A small error in a binary file may make it unreadable.
Many programs are capable of reading and editing text files.	Less programs are capable of reading and editing text files.

Table B.1: Table comparing the pros and cons for storing data in text files and binary files.

Appendix C

Raw Console Output Example

An example of the console output produced, depicting the first option to stabilise the query video, followed by a request to manually select the ROI, followed by a printing of the results. The results are printed in the form of tables for each histogram model and each distance metric used, with the match found located under the table. Global results are printed at the end of the program, in parallel to the histogram and figure displayed with Matplotlib for a visual representation of the results.

This example console output uses the skewed butterfly query and six database videos in order to fit within a small minimum amount of pages.

File - ALL Test

```
 1 /Users/ajaamour/Environments/Content-Based-Video-Retrieval-
   Code/bin/python3 /Users/ajaamour/Projects/Content-Based-
   Video-Retrieval-Code/app/main.py --model all --mode test --
   showhists
 2 Do you wish to stabilise the recorded query video? [y/N] y
 3
 4 Stabilised version of query already found: 'stable-
   recording3.avi'
 5
 6 Using query: 'stable-recording3.avi'
 7
 8 Please crop the recorded query video for the signature to
   be generated.
 9
10 Greyscale Histogram Comparison Results:
11
12 ╔CORRELATION═══════╦═══════════╗
13 ║ sushi.mp4        ║ -0.04906  ║
14 ╠══════════════════╬═══════════╣
15 ║ disco-ball.mp4   ║ -0.24845  ║
16 ╠══════════════════╬═══════════╣
17 ║ ice-hockey.mp4   ║ -0.25634  ║
18 ╠══════════════════╬═══════════╣
19 ║ beach.mp4        ║ 0.10313   ║
20 ╠══════════════════╬═══════════╣
21 ║ butterfly.mp4    ║ 0.33377   ║
22 ╠══════════════════╬═══════════╣
23 ║ spaghetti.mp4    ║ 0.64516   ║
24 ╚══════════════════╩═══════════╝
25 Greyscale CORRELATION match found: spaghetti.mp4
26
27
28 ╔CHI-SQUARE════════╦═══════════╗
29 ║ sushi.mp4        ║ 3410.27354 ║
30 ╠══════════════════╬═══════════╣
31 ║ disco-ball.mp4   ║ 86621.92583 ║
32 ╠══════════════════╬═══════════╣
33 ║ ice-hockey.mp4   ║ 4358.13383 ║
34 ╠══════════════════╬═══════════╣
35 ║ beach.mp4        ║ 55.08272  ║
36 ╠══════════════════╬═══════════╣
37 ║ butterfly.mp4    ║ 9.34003   ║
38 ╠══════════════════╬═══════════╣
39 ║ spaghetti.mp4    ║ 7.59116   ║
40 ╚══════════════════╩═══════════╝
41 Greyscale CHI-SQUARE match found: spaghetti.mp4
42
43
44 ╔INTERSECTION══════╦═══════════╗
45 ║ sushi.mp4        ║ 7.25432   ║
```

disco-ball.mp4	0.8675
ice-hockey.mp4	1.06405
beach.mp4	6.10959
butterfly.mp4	6.51279
spaghetti.mp4	5.6198

Greyscale INTERSECTION match found: sushi.mp4

HELLINGER

sushi.mp4	0.39752
disco-ball.mp4	0.77503
ice-hockey.mp4	0.73212
beach.mp4	0.41016
butterfly.mp4	0.4124
spaghetti.mp4	0.36488

Greyscale HELLINGER match found: spaghetti.mp4

RGB Histogram Comparison Results:

CORRELATION

sushi.mp4	-0.19274
disco-ball.mp4	-0.24466
ice-hockey.mp4	-0.24504
beach.mp4	-0.08755
butterfly.mp4	0.45455
spaghetti.mp4	0.14436

RGB CORRELATION match found: butterfly.mp4

CHI-SQUARE

File - ALL Test

```
 96 ╔═══════════════════╦═══════════════════╗
 97 ╠═══════════════════╬═══════════════════╣
 98 ╠═══════════════════╬═══════════════════╣
 99 ╠═══════════════════╬═══════════════════╣
100 ╠═══════════════════╬═══════════════════╣
101 ╠═══════════════════╬═══════════════════╣
102 ╠═══════════════════╬═══════════════════╣
103 ╠═══════════════════╬═══════════════════╣
104 ╠═══════════════════╬═══════════════════╣
105 ╠═══════════════════╬═══════════════════╣
106 ╠═══════════════════╬═══════════════════╣
107 ╚═══════════════════╩═══════════════════╝
108 RGB CHI-SQUARE match found: butterfly.mp4
109
110
111 ╔INTERSECTION═══════╦═══════════════════╗
112 ╠═══════════════════╬═══════════════════╣
113 ╠═══════════════════╬═══════════════════╣
114 ╠═══════════════════╬═══════════════════╣
115 ╠═══════════════════╬═══════════════════╣
116 ╠═══════════════════╬═══════════════════╣
117 ╠═══════════════════╬═══════════════════╣
118 ╠═══════════════════╬═══════════════════╣
119 ╠═══════════════════╬═══════════════════╣
120 ╠═══════════════════╬═══════════════════╣
121 ╠═══════════════════╬═══════════════════╣
122 ╚═══════════════════╩═══════════════════╝
123
124 RGB INTERSECTION match found: butterfly.mp4
125
126
127 ╔HELLINGER══════════╦═══════════════════╗
128 ╠═══════════════════╬═══════════════════╣
129 ╠═══════════════════╬═══════════════════╣
130 ╠═══════════════════╬═══════════════════╣
131 ╠═══════════════════╬═══════════════════╣
132 ╠═══════════════════╬═══════════════════╣
133 ╠═══════════════════╬═══════════════════╣
134 ╠═══════════════════╬═══════════════════╣
135 ╠═══════════════════╬═══════════════════╣
136 ╠═══════════════════╬═══════════════════╣
137 ╠═══════════════════╬═══════════════════╣
138 ╚═══════════════════╩═══════════════════╝
139
140 RGB HELLINGER match found: butterfly.mp4
141
142
143
144 HSV Histogram Comparison Results:
145
```

File - ALL Test

```
╒EARTH'S MOVER DISTANCE═══╕
│ sushi.mp4        │ 3.07203 │
├──────────────────┼─────────┤
│ disco-ball.mp4   │ 2.43814 │
├──────────────────┼─────────┤
│ ice-hockey.mp4   │ 2.14705 │
├──────────────────┼─────────┤
│ beach.mp4        │ 2.37272 │
├──────────────────┼─────────┤
│ butterfly.mp4    │ 0.99484 │
├──────────────────┼─────────┤
│ spaghetti.mp4    │ 2.25072 │
╘══════════════════╧═════════╛
HSV EARTH'S MOVER DISTANCE Match found: butterfly.mp4


╒ENERGY DISTANCE══════════╕
│ sushi.mp4        │ 12.32629 │
├──────────────────┼──────────┤
│ disco-ball.mp4   │ 10.07701 │
├──────────────────┼──────────┤
│ ice-hockey.mp4   │ 8.6981   │
├──────────────────┼──────────┤
│ beach.mp4        │ 9.68431  │
├──────────────────┼──────────┤
│ butterfly.mp4    │ 5.1075   │
├──────────────────┼──────────┤
│ spaghetti.mp4    │ 9.17143  │
╘══════════════════╧══════════╛
HSV ENERGY DISTANCE Match found: butterfly.mp4


Matches made: Counter({'butterfly.mp4': 40, 'spaghetti.mp4
    ': 3, 'sushi.mp4': 1})
% of matches made: {'spaghetti.mp4': 6.82, 'sushi.mp4': 2.
    27, 'butterfly.mp4': 90.91}


Generated all histograms for all videos


MATCH FOUND: butterfly.mp4

--- Runtime: 4.17 seconds ---
--- Accuracy: 90.91 % ---

Process finished with exit code 0

```

Appendix D

Stored Feature Files

Examples of the different compact features stored in plain text files using the butterfly video as an example. These correspond to averaged histograms in greyscale, RGB and HSV colour models.

D.1 Greyscale Histogram Features

The averaged greyscale histogram only has one channel, represented by 255 values for the 255 bins. Only bins 1-5, 124-128 and 250-255 are represented below.

File: *butterfly.mp4/hist-gray*

```
# Greyscale Histogram (255 bins) [normalised]
0.000677
0.000453
0.000588
0.000803
0.001338
...
0.147575
0.149898
0.141548
0.023301
0.108333
...
0.000152
0.000164
0.000087
0.000080
0.000128
```

D.2 RGB Histogram Features (Red channel only)

The averaged RGB histogram has three channels, each represented by 255 values for the 255 bins. Only the red channel is represented in the example below. The same file format is used to represent the green and blue channels. Only bins 1-5, 124-128 and 250-255 are represented below.

File: *butterfly.mp4/hist-r*

```
# 'R' channel of RGB histogram (255 bins) [normalised]
0.001610
0.000436
0.000395
0.000468
0.000833
...
0.150552
0.112711
0.133993
0.113074
0.103523
...
0.002662
0.002867
0.003141
0.003351
0.003299
```

D.3 HSV Histogram Features

The averaged HSV histogram has 8 total slices for the 8 different hue bins. Each hue bin has 12 saturation bins, each with 3 value bins. Only the 1st and 8th hue bins are represented below.

File: *butterfly.mp4/hist-hsv*

```
# HSV Histogram shape: (8, 12, 3) [normalised]
# New slice
1.690998567606915226e-03 9.581707232758741389e-04
    ↪ 1.511106754399158718e-03
2.772998953746123787e-03 4.584945844147692712e-03
    ↪ 3.237609486942264088e-03
4.736755551262335738e-03 6.970707496458833710e-03
    ↪ 4.525759765370326003e-03
5.926002092151479682e-03 7.984556638720359881e-03
    ↪ 9.167456248013133510e-03
5.528808910061012299e-03 1.160728825594891170e-02
    ↪ 8.409987489523535267e-03
8.208146847953850730e-03 1.193360518664121628e-02
    ↪ 6.526554860597984752e-03
```

```
7.324515270407904594e-03 9.202591588043353144e-03
    ↪ 1.104022278873757930e-02
7.212228395722128892e-03 7.106588645414872209e-03
    ↪ 1.543889956718141367e-02
6.635527232323180230e-03 7.380225738002495726e-03
    ↪ 1.792533712630922302e-02
5.477001382545990983e-03 5.517539758743209878e-03
    ↪ 1.022866905920884889e-02
3.716428414918482304e-03 1.813905805730345495e-03
    ↪ 6.794666538057340423e-04
8.167625489560041349e-03 1.257265304130586632e-03
    ↪ 1.215253951985770014e-04
...
# New slice
8.166000294626097689e-04 1.444111869204789400e-03
    ↪ 2.556757863864979764e-03
2.281072126193480019e-03 1.405416374010118492e-03
    ↪ 6.197975771184163719e-04
3.227299312129616737e-03 4.016239108750596642e-04
    ↪ 5.364667876247867172e-05
4.634697921574115753e-03 1.376390735624061679e-04
    ↪ 2.150440096474168000e-04
4.121060758321123084e-03 1.419378115149976904e-04
    ↪ 1.943936631505494006e-04
3.336825863119553554e-03 2.044695773458277588e-04
    ↪ 1.442692149050179185e-04
3.074369096959178940e-03 3.068208203661594103e-04
    ↪ 5.189062966647642284e-04
2.311520511284470558e-03 2.902706883932378913e-04
    ↪ 2.334064346293664594e-03
1.362986608662388561e-03 1.100270561768080737e-04
    ↪ 2.477648943154649262e-03
7.060457216787406187e-04 7.275994308160575492e-05
    ↪ 1.679020986722951645e-03
2.139106257924471488e-04 7.810963734300457872e-05
    ↪ 1.976605589416894021e-04
2.936312653632326707e-04 3.317900460718771412e-05
    ↪ 1.067189321640233325e-05
```

Appendix E

13 Point Ethics Check List

This document describes the 13 issues that need to be considered carefully before students or staff involve other people ("participants") for the collection of information as part of their project or research.

1. **Have you prepared a briefing script for volunteers**?
   All participants will be introduced with a briefing script explaining what the project consists of, what their role in the experiment is, and how their data will be used.

2. **Will the participants be using any non-standard hardware?**
   No. Participants can conduct the experiment from their personal computers or mobile devices.

3. **Is there any intentional deception of the participants?**
   No deception will occur during this experiment; all the details of the experiment will be provided in the debriefing script.

4. **How will participants voluntarily give consent?**
   The participants will give consent when they submit their Google Form survey, which is specified in the briefing script.

5. **Will the participants be exposed to any risks greater than those encountered in their normal work life?**
   Participants will take part in the experiment from their environment of choice as they will receive a link to the online survey in their inbox, which they can use to access the survey online from their personal computers and mobile devices.

6. **Are you offering any incentive to the participants?**
   No incentive will be offered to participants.

7. **Are any of your participants under the age of 16?**
   All participants are above the age of 18.

8. **Do any of your participants have an impairment that will limit their understanding or communication?**
   None of the participants have any known impairment which may affect the experiment's results.

9. **Are you in a position of authority or influence over any of your participants?**
   The participants will complete the survey in their own free time using their hardware with no presence of authority present during the experiment.

10. **Will the participants be informed that they could withdraw at any time?**
    All the participants will be informed that they can withdraw from the experiment at any time in the briefing script.

11. **Will the participants be informed of your contact details?** My contacts details and my supervisor's details will be shared with all participants.

12. **Will participants be de-briefed?**
    All participants will be debriefed through the introductory briefing script.

13. **Will the data collected from the participants be stored in an anonymous form?**
    The data collected from the online survey will be downloaded and stored locally in a CSV file anonymously and will be used anonymously in the final report, as informed in the briefing script.

Appendix F

Experiment Survey

This chapter presents the survey used for the online experiment. The database videos and the query video are included in the survey using YouTube. The online survey is powered by Google Forms, and is available online: `https://forms.gle/HyQAGyks2Tnj7wgc9`

F.1 Title

Content-Based Video Retrieval Experiment

F.2 What is the project about?

This dissertation project aims to create a system similar to Shazam for movies.

For background information, Shazam is a mobile application that allows users to match a recording to a piece of music.

The goal of this project is to explore how such a system could be developed to work with movies, where a user would record a screen with his phone, which will pattern match the movie to a database of movies and accurately return the movie title.

F.3 Your role in this experiment

In this experiment, you play the role of the algorithm that matches a user recording to one of the videos in the database.

You will be shown 6 videos from the database, followed by 1 user-recorded video. Your goal is to mentally compare the recorded video to each video in the database to be able to rank each database video from most likely to match

the recorded video to least likely.

When making your decision, take into account aspects of the video such as colour distribution, luminosity, textures, shapes, objects and motion, as a video matching algorithm would in real life.

Note: The data retrieved from this experiment will be compared to the results the of the project's matching algorithm. It will remain anonymous in locally-stored hard copies and in the final report. By submitting this form, you give consent for your data to be evaluated and used in the report. You may withdraw from the experiment at any time.

F.4 Watch

Database Videos Watch the 6 database videos that will train your "knowledge".

Figure F.1: The YouTube video showing the 6 database videos used for the experiment, available online: `https://www.youtube.com/watch?v=BvukbK-sX9A`.

Query Video Now watch the recorded query video. Your goal is now to compare this video to the 6 previous videos you watched and to rank them based on their visual similarities.

Figure F.2: The YouTube video showing the query video used for the experiment, available online: `https://www.youtube.com/watch?v=4JPo0-aSzNE`.

F.5 Rank

Which database video does the recorded query match the most?

	Video A (snowy winter)	Video B (Earth from space)	Video C (shuttle landing)	Video D (cloudy sky)	Video E (sunset)	Video F (beach)
Most likely to match (#1)	☐	☐	☐	☐	☐	☐
2nd most likely to match (#2)	☐	☐	☐	☐	☐	☐
3nd most likely to match (#1)	☐	☐	☐	☐	☐	☐
4th most likely to match (#4)	☐	☐	☐	☐	☐	☐
5th most likely to match (#5)	☐	☐	☐	☐	☐	☐
Least likely to match (#6)	☐	☐	☐	☐	☐	☐

Figure F.3: Screenshot of the checkbox grid used to rank the database videos from most likely to match the query video to least likely.

Which video aspects did you consider the most when ranking them?

Placeholder for long answer from participant.

What do you think is the most important aspect of a video that a matching algorithm should analyse when pattern matching videos?

Placeholder for short answer from participant.

F.6 Confirmation Message

(The message that is displayed to users once they have finished the experiment and submitted the Google Form).

Your response has been recorded.

Thank you taking part in this experiment!

Here are my algorithm's results for comparison:

1. cloudy-sky (video D)
2. winter (video A)
3. earth (video B)
4. shuttle-landing (video C)
5. beach (video F)
6. sunset (video E)

Contact details:

- Adam Jaamour: *aj645@bath.ac.uk*
- Dr. Yong-Liang Yang (project supervisor): *Y.Yang2@bath.ac.uk*

Appendix G

Raw Experiment Results

G.1 Question 1

Which database video does the recorded query match the most?

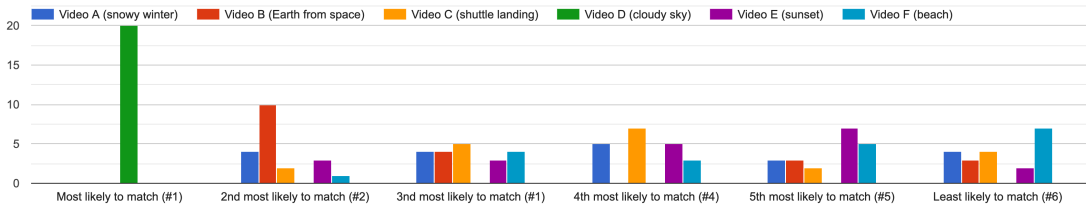

Figure G.1: Survey results of the "Which database video does the recorded query match the most?" ranking question.

G.2 Question 2

Which video aspects did you consider the most when ranking them?

- Participant 1: Blue colour
- Participant 2: the sky is cold
- Participant 3: The colours and movements
- Participant 4: The colour of the sky and the movements
- Participant 5: Colours and luminosity
- Participant 6: The actual scenario i.e. the fact that a plane was landing or the fact that someone was walking a dog
- Participant 7: Light, sky and clouds

- Participant 8: clouds
- Participant 9: colour, landscape, sounds
- Participant 10: movement, patterns, image, colour
- Participant 11: luminosity of colours and similarity of object represented
- Participant 12: Images, absence of sound, speed of movement
- Participant 13: Colours
- Participant 14: Colour, movement, size of objects, shape of objects
- Participant 15: Image, colour, light
- Participant 16: The colours (including luminosity) then the movements
- Participant 17: colour blue with blurry whites. sky clearly visible in daytime
- Participant 18: The movement and colour of the image
- Participant 19: Colours
- Participant 20: Colours of the video

G.3 Question 3

What do you think is the most important aspect of a video that a matching algorithm should analyse when pattern matching videos?

- Participant 1: Similar colours, shades of colours
- Participant 2: colours (warm vs cold)
- Participant 3: Colours
- Participant 4: Colours and the pace of movement and change
- Participant 5: Colours
- Participant 6: colour
- Participant 7: The environment surrounding the action
- Participant 8: overall colour and location of major colour blobs

- Participant 9: filter / colours, sounds (as many movies can relate similar situation e.g. Second World War, they can be differentiated by determining how old is the movie and where it takes place). Sounds are very different from one movie to another, even in the same topic, so I believe that it is another core aspect to be analysed.
- Participant 10: movement
- Participant 11: shapes and luminosity
- Participant 12: Image + speed of movement
- Participant 13: Colours / Shapes / moving objects
- Participant 14: Colour scheme
- Participant 15: Sound, quality of image
- Participant 16: The motion and the colours
- Participant 17: objects appearing in video
- Participant 18: The succession of images
- Participant 19: movements
- Participant 20: Shape and colour of objects within the video

Appendix H

Code Listings

This appendix includes the main code sections of the system in a lightweight form, with omitted comments and condensed formats. The code is organised by module.

H.1 Main Module

The *main.py* module contains the program entry point that parses the command line argument in order to decide which phase to execute. The module the three functions to start running each phase, include the off-line colour-based feature extraction, the on-line retrieval and database pre-processing phases.

H.1.1 Argument Parsing

The program entry point. Parses command line input to decide which phase of the system to run.

```
import argparse
import app.config as config

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("-m", "--model", help="The histogram
    ↪ model to use. Choose from the following options: 'rgb',
    ↪ 'hsv' or 'gray'. Leave empty to train using all 3 histogram
    ↪ models.")
    parser.add_argument("--mode", required=True, help="The mode
    ↪ to run the code in. Choose from the following options:
    ↪ 'train', 'test' or 'segment'.")
    parser.add_argument("--showhists", action="store_true",
    ↪ help="Specify whether you want to display each generated
    ↪ histogram.")
    parser.add_argument("-d", "--debug", action="store_true",
    ↪ help="Specify whether you want to print additional logs for
    ↪ debugging purposes.")
```

```
10      args = parser.parse_args()
11      config.debug = args.debug
12      config.mode = args.mode
13      config.show_histograms = args.showhists
14      config.model = args.model
15      if config.mode == "train":
16          off_line_colour_based_feature_extraction_phase()
17      elif config.mode == "test":
18          on_line_retrieval_phase()
19      elif config.mode == "segment":
20          segment_video()
21      else:
22          print("Wrong mode chosen. Choose from the following
    ↪ options: 'train', 'test' or 'segment'.")
23          exit(0)
```

H.1.2 Initial Off-line Feature Extraction Phase Function

Generates and stores averaged greyscale, RGB and HSV histograms for all the videos in the directory-based database.

```
1  from pyspin.spin import make_spin, Spin2
2  from app.histogram import HistogramGenerator
3
4  @make_spin(Spin2, "Generating histograms for database
       ↪ videos...".format(config.model))
5  def off_line_colour_based_feature_extraction_phase():
6      files = get_video_filenames("../footage/")
7      start_time = time.time() # start measuring runtime
8      for file in files:
9          histogram_generator_gray = HistogramGenerator(directory,
       ↪ file)
10
       ↪ histogram_generator_gray.generate_video_greyscale_histogram()
11         histogram_generator_rgb = HistogramGenerator(directory,
       ↪ file)
12         histogram_generator_rgb.generate_video_rgb_histogram()
13         histogram_generator_hsv = HistogramGenerator(directory,
       ↪ file)
14         histogram_generator_hsv.generate_video_hsv_histogram()
15     runtime = round(time.time() - start_time, 2)
16     print_finished_training_message(config.model, directory,
       ↪ runtime)
```

H.1.3 Initial On-line Retrieval Phase Function

Prompts the user to stabilise and crop the query video before generating the same averaged greyscale, RGB and HSV histograms to compare with the database videos' previously stored histograms using distance metrics.

```
def on_line_retrieval_phase():
    directory = "../recordings/"
    file = "butterfly_recording.mp4"

    # ask user to stabilise the input query video or not (see
    ↪ code in section below)
    print("\nUsing query: '{}'".format(file))
    print("\nPlease crop the recorded query video for the
    ↪ signature to be generated.")

    start_time = time.time()
    histogram_generator_gray = HistogramGenerator(directory, file)
    histogram_generator_gray.generate_video_greyscale_histogram(
    ↪ is_query=True)
    cur_reference_points =
    ↪ histogram_generator_gray.get_current_reference_points()
    histogram_generator_rgb = HistogramGenerator(directory, file)
    histogram_generator_rgb.generate_video_rgb_histogram(
    ↪ is_query=True, cur_ref_points=cur_reference_points)
    histogram_generator_hsv = HistogramGenerator(directory, file)
    histogram_generator_hsv.generate_video_hsv_histogram(
    ↪ is_query=True, cur_ref_points=cur_reference_points)

    histogram_generator_gray.match_histograms(
    ↪ cur_all_model='gray')
    histogram_generator_rgb.match_histograms(cur_all_model='rgb')
    histogram_generator_hsv.match_histograms(cur_all_model='hsv')

    # Combine matches from all 3 histogram models
    all_results = histogram_generator_hsv.get_results_array()
    results_count = Counter(all_results)  # count matches
    # convert from count to %
    results_percentage = dict()
    for match in results_count:
        percentage = round((results_count[match] /
    ↪ len(all_results)) * 100.0, 2)
        results_percentage[match] = percentage
    display_results_histogram(results_percentage)
    print("Matches made: {}".format(results_count))
    print("% of matches made: {}".format(results_percentage))

    # find best result
    final_result_name = ""
    final_result_count = 0
    for i, r in enumerate(results_count):
        if i == 0:
            final_result_name = r
```

```
            final_result_count = results_count[r]
        else:
            if results_count[r] > final_result_count:
                final_result_name = r
                final_result_count = results_count[r]

    # print results
    runtime = round(time.time() - start_time, 2)
    accuracy = final_result_count / len(all_results)
    get_video_first_frame(directory + file, "../results",
    ↪ is_query=True)

    ↪ get_video_first_frame("../footage/{}".format(final_result_name),
    ↪ "../results", is_result=True)
    show_final_match(final_result_name, "../results/query.png",
    ↪ "../results/result.png", runtime, accuracy)
    print_finished_training_message(final_result_name,
    ↪ config.model, runtime, accuracy)
```

H.1.4 Initial Database Pre-Processing Phase

Applies a shot boundary detection algorithm to a video for segmentation.

```
def database_preprocessing_phase():
    shot_boundary_detector = HistogramGenerator("../recordings/",
    ↪ "scene-segmentation.mp4")
    video_capture = shot_boundary_detector.get_video_capture()
    frame_count = get_number_of_frames(vc=video_capture)
    fps = get_video_fps(vc=video_capture)
    print("Total Frames: {}".format(frame_count))
    print("FPS: {}\n".format(fps))

    # start processing video for shout boundary detection
    print("Starting to process video for shot boundary
    ↪ detection...")
    start_time = time.time()
    shot_boundary_detector.rgb_histogram_shot_boundary_detection(
    ↪ threshold=7)

    # print final results
    runtime = round(time.time() - start_time, 2)
    print("--- Number of frames in video: {}
    ↪ ---".format(frame_count))
    print("--- Runtime: {} seconds ---".format(runtime))
```

H.2 Histograms Module

The *histogram.py* module contains the HistogramGenerator class with functions to generate, average, store and compare greyscale, RGB and HSV histograms.

H.2.1 High Level Module Overview

A high-level overview of the module's classes and functions.

```
class HistogramGenerator:
    colours = ('b', 'g', 'r')  # RGB channels
    bins = (8, 12, 3)  # 8 Hue, 12 Saturation, 3 Value
    histcmp_methods = [cv2.HISTCMP_CORREL, cv2.HISTCMP_CHISQR,
    ↪ cv2.HISTCMP_INTERSECT, cv2.HISTCMP_HELLINGER]
    histcmp_3d_methods = ["earths_mover_distance",
    ↪ "energy_distance"]
    histogram_comparison_weigths = {
        'gray': 1,
        'rgb': 5,
        'hsv': 10
    }
    results_array = list()

    def __init__(self, directory, file_name):
    def generate_video_rgb_histogram(self, is_query=False,
    ↪ cur_ref_points=None):
    def generate_video_greyscale_histogram(self, is_query=False):
    def generate_video_hsv_histogram(self, is_query=False,
    ↪ cur_ref_points=None):
    def generate_and_store_average_rgb_histogram(self):
    def generate_and_store_average_greyscale_histogram(self):
    def generate_and_store_average_hsv_histogram(self):
    def match_histograms(self, cur_all_model="all"):
    def rgb_histogram_shot_boundary_detection(self, threshold):
    def check_video_capture(self):
    def destroy_video_capture(self):
    def get_video_capture(self):
    def get_current_reference_points(self):
    def get_results_array(self):

def _normalise_histogram(hist):
def _get_frames_to_process(vc):
def _get_chosen_model_string(model):
```

H.2.2 Single Greyscale Histogram Generation

Generates multiple greyscale histograms (one every second) for a video.

```
def generate_video_greyscale_histogram(self, is_query=False):
    frame_counter = 0  # cur frame ID
    while self.video_capture.isOpened():
```

```
        ret, frame = self.video_capture.read()
        if ret:
            if is_query and frame_counter == 0:
                cad = ClickAndDrop(frame)
                self.reference_points = cad.get_reference_points()
            frame_counter += 1
            if frame_counter in
    ↪ _get_frames_to_process(self.video_capture):
                if is_query and len(self.reference_points) == 2:
                    roi =
    ↪ frame[self.reference_points[0][1]:self.reference_points[1][1],
    ↪ self.reference_points[0][0]:self.reference_points[1][0]]
                    roi_grey = cv2.cvtColor(roi,
    ↪ cv2.COLOR_BGR2GRAY)
                    histogram = cv2.calcHist([roi_grey], [0],
    ↪ None, [256], [0, 256])
                else:
                    grey_frame = cv2.cvtColor(frame,
    ↪ cv2.COLOR_BGR2GRAY)
                    histogram = cv2.calcHist([grey_frame], [0],
    ↪ None, [256], [0, 256])
                self.histograms_grey_dict.append(histogram)
        else:
            break
    self.generate_and_store_average_greyscale_histogram()
    self.destroy_video_capture()
```

H.2.3 Single RGB Histogram Generation

Generates multiple RGB histograms (one every second) for a video.

```
def generate_video_rgb_histogram(self, is_query=False,
    ↪ cur_ref_points=None):
    frame_counter = 0  # cur frame ID
    while self.video_capture.isOpened():
        ret, frame = self.video_capture.read()
        if ret:
            if is_query and frame_counter == 0:
                if cur_ref_points is None:
                    cad = ClickAndDrop(frame)
                    self.reference_points =
    ↪ cad.get_reference_points()
                else:
                    self.reference_points = cur_ref_points
            frame_counter += 1
            if frame_counter in
    ↪ _get_frames_to_process(self.video_capture):
                for i, col in enumerate(self.colours):
                    if is_query and len(self.reference_points) ==
    ↪ 2:
                        roi =
    ↪ frame[self.reference_points[0][1]:self.reference_points[1][1],
```

```
17
    ↪ self.reference_points[0][0]:self.reference_points[1][0]]
18                          histogram = cv2.calcHist([roi], [i],
    ↪ None, [256], [0, 256])
19                      else:
20                          histogram = cv2.calcHist([frame], [i],
    ↪ None, [256], [0, 256])
21
    ↪ self.histograms_rgb_dict[col].append(histogram)
22          else:
23              break
24      self.generate_and_store_average_rgb_histogram()
25      self.destroy_video_capture()
```

H.2.4 Single HSV Histogram Generation

Generates multiple HSV histograms (one every second) for a video.

```
1  def generate_video_hsv_histogram(self, is_query=False,
       ↪ cur_ref_points=None):
2      frame_counter = 0  # cur frame ID
3      while self.video_capture.isOpened():
4          ret, frame = self.video_capture.read()
5          if ret:
6              if is_query and frame_counter == 0:
7                  if cur_ref_points is None:
8                      cad = ClickAndDrop(frame)
9                      self.reference_points =
       ↪ cad.get_reference_points()
10                 else:
11                     self.reference_points = cur_ref_points
12             frame_counter += 1
13             if frame_counter in
       ↪ _get_frames_to_process(self.video_capture):
14                 if is_query and len(self.reference_points) == 2:
15                     roi =
       ↪ frame[self.reference_points[0][1]:self.reference_points[1][1],
       ↪ self.reference_points[0][0]:self.reference_points[1][0]]
16                     roi_hsv = cv2.cvtColor(roi, cv2.COLOR_BGR2HSV)
17                     histogram = cv2.calcHist([roi_hsv], [0, 1,
       ↪ 2], None, self.bins, [0, 180, 0, 256, 0, 256])
18                 else:
19                     hsv_frame = cv2.cvtColor(frame,
       ↪ cv2.COLOR_BGR2HSV)
20                     histogram = cv2.calcHist([hsv_frame], [0, 1,
       ↪ 2], None, self.bins, [0, 180, 0, 256, 0, 256])
21                 self.histograms_hsv_dict.append(histogram)
22         else:
23             break
24     self.generate_and_store_average_hsv_histogram()
25     self.destroy_video_capture()
```

H.2.5 Average & Store Greyscale Histogram

Generates a single greyscale histogram by averaging all histograms of a video before normalising it and saving the results to a plain text file.

```
def generate_and_store_average_greyscale_histogram(self):
    avg_histogram = np.zeros(shape=(255, 1))
    hist = self.histograms_grey_dict
    for i in range(0, 255):  # loop through all bins
        bin_sum = 0
        # get value for each colour histogram in bin i
        for arr_index in range(0, len(hist)):
            bin_value = hist[arr_index].item(i)
            bin_sum += bin_value
        # average all bins values to store in new histogram
        new_bin_value = bin_sum / len(hist)
        avg_histogram[i] = new_bin_value
    # normalise
    avg_histogram = _normalise_histogram(avg_histogram)
    # write to file
    if not
    ↪ os.path.exists("../histogram_data/{}/".format(self.file_name)):

    ↪ os.makedirs("../histogram_data/{}/".format(self.file_name))
    with
    ↪ open("../histogram_data/{}/hist-{}.txt".format(self.file_name,
    ↪ "gray"), 'w') as file:
        file.write("# Greyscale Histogram ({} bins)
    ↪ [normalised]\n".format(avg_histogram.shape[0]))
        np.savetxt(file, avg_histogram, fmt='%f')
```

H.2.6 Average & Store RGB Histogram

Generates a single RGB histogram by averaging all histograms of a video before normalising it and saving the results to a plain text file.

```
def generate_and_store_average_rgb_histogram(self):
    avg_histogram = np.zeros(shape=(255, 1))
    for col, hists in self.histograms_rgb_dict.items():
        for i in range(0, 255):  # loop through all bins
            bin_sum = 0
            # get value for each colour histogram in bin i
            for arr_index in range(0, len(hists)):
                bin_value = hists[arr_index].item(i)
                bin_sum += bin_value
            # average all bins values to store in new histogram
            new_bin_value = bin_sum / len(hists)
            avg_histogram[i] = new_bin_value
        # normalise averaged histogram
        avg_histogram = _normalise_histogram(avg_histogram)
        # write to file
        if not
    ↪ os.path.exists("../histogram_data/{}/".format(self.file_name)):
```

```
17
    ↪ os.makedirs("../histogram_data/{}/".format(self.file_name))
18      with
    ↪ open("../histogram_data/{}/hist-{}.txt".format(self.file_name,
    ↪ col), 'w') as file:
19          file.write("# '{}' channel of RGB histogram ({} bins)
    ↪ [normalised]\n".format(
20              col.upper(),
21              avg_histogram.shape[0]
22          ))
23          np.savetxt(file, avg_histogram, fmt='%f')
```

H.2.7 Average & Store HSV Histogram

Generates a single HSV histogram by averaging all histograms of a video before normalising it and saving the results to a plain text file.

```
1  def generate_and_store_average_hsv_histogram(self):
2      avg_histogram = np.zeros(shape=(8, 12, 3))
3      hist = self.histograms_hsv_dict
4      for h in range(0, self.bins[0]):  # loop through hue bins
5          for s in range(0, self.bins[1]):  # loop through
       ↪ saturation bins
6              for v in range(0, self.bins[2]):  # loop through
       ↪ value bins
7                  bin_sum = 0
8                  # get value for each colour histogram in bin
       ↪ [h][s][v]
9                  for arr_index in range(0, len(hist)):
10                     bin_value = hist[arr_index][h][s][v]
11                     bin_sum += bin_value
12                 # average all bins values to store in new
       ↪ histogram
13                 new_bin_value = bin_sum / len(hist)
14                 avg_histogram[h][s][v] = new_bin_value
15     # normalise averaged histogram
16     avg_histogram = _normalise_histogram(avg_histogram)
17     # write to file
18     if not
       ↪ os.path.exists("../histogram_data/{}/".format(self.file_name)):
19
       ↪ os.makedirs("../histogram_data/{}/".format(self.file_name))
20     with
       ↪ open("../histogram_data/{}/hist-{}.txt".format(self.file_name,
       ↪ "hsv"), 'w') as file:
21         file.write("# HSV Histogram shape: {0}
       ↪ [normalised]\n".format(avg_histogram.shape))
22         for arr_2d in avg_histogram:
23             file.write("# New slice\n")
24             np.savetxt(file, arr_2d)
```

H.2.8 Histogram Matching

Compares the greyscale, RGB and HSV histograms of the query video with each of the saved average histograms using different distance metrics such as the Correlation, Intersection, Chi-Square Distance, Hellinger Distance, Earth's Mover Distance and Energy Distance metrics. Finally, prints the results for each histogram model and metric in a console table and writes the data to a CSV file.

2D Histogram matching

Compares 2D histograms, concerning the greyscale and RGB histograms only using *OpenCV's* compareHist function.

```
def match_histograms(self, cur_all_model="all"):
    # variables used for finding the match to the recorded video
    video_match = ""
    video_match_value = 0
    # get histogram for the recorded video to match
    if config.model == "gray":
        query_histogram = {
            'gray': np.loadtxt(
↪ "../histogram_data/{}/hist-gray.txt".format(self.file_name),
↪ dtype=np.float32, unpack=False)
        }
    elif config.model == "rgb":
        query_histogram = {
            'b':
↪ np.loadtxt("../histogram_data/{}/hist-b.txt".format(
↪ self.file_name), dtype=np.float32, unpack=False),
            'g':
↪ np.loadtxt("../histogram_data/{}/hist-g.txt".format(
↪ self.file_name), dtype=np.float32, unpack=False),
            'r':
↪ np.loadtxt("../histogram_data/{}/hist-r.txt".format(
↪ self.file_name), dtype=np.float32, unpack=False)
        }

    print("\n{} Histogram Comparison
↪ Results:\n".format(_get_chosen_model_string(cur_all_model)))
    method = ""
    csv_field_names = ["video", "score"]
    if config.model == "rgb" or config.model == "gray":
        for m in self.histcmp_methods:
            if m == 0:
                method = "CORRELATION"
            elif m == 1:
                method = "CHI-SQUARE"
            elif m == 2:
                method = "INTERSECTION"
            elif m == 3:
```

```
30                  method = "HELLINGER"
31              # CSV file to write data to for each method
32              csv_file =
    ↪ open("../results/csv/{}-{}-{}.csv".format(config.model,
    ↪ cur_all_model, method), 'w')
33              with csv_file:
34                  writer = csv.DictWriter(csv_file,
    ↪ fieldnames=csv_field_names)
35                  writer.writeheader()
36                  table_data = list()
37                  for i, file in
    ↪ enumerate(get_video_filenames("../footage/")):
38                      comparison = 0
39                      if config.model == "gray":
40                          dbvideo_greyscale_histogram =
    ↪ np.loadtxt("../histogram_data/{}/hist-gray.txt".format(file),
    ↪ dtype=np.float32, unpack=False)
41                          comparison =
    ↪ cv2.compareHist(query_histogram['gray'],
    ↪ dbvideo_greyscale_histogram, m)
42                      elif config.model == "rgb":
43                          dbvideo_b_histogram =
    ↪ np.loadtxt("../histogram_data/{}/hist-b.txt".format(file),
    ↪ dtype=np.float32, unpack=False)
44                          dbvideo_g_histogram =
    ↪ np.loadtxt("../histogram_data/{}/hist-g.txt".format(file),
    ↪ dtype=np.float32, unpack=False)
45                          dbvideo_r_histogram =
    ↪ np.loadtxt("../histogram_data/{}/hist-r.txt".format(file),
    ↪ dtype=np.float32, unpack=False)
46                          comparison_b =
    ↪ cv2.compareHist(query_histogram['b'], dbvideo_b_histogram,
    ↪ m)
47                          comparison_g =
    ↪ cv2.compareHist(query_histogram['g'], dbvideo_g_histogram,
    ↪ m)
48                          comparison_r =
    ↪ cv2.compareHist(query_histogram['r'], dbvideo_r_histogram,
    ↪ m)
49                          comparison = (comparison_b + comparison_g
    ↪ + comparison_r) / 3
50
51                      # append data to table
52                      table_data.append([file, round(comparison,
    ↪ 5)])
53                      # write data to CSV file
54                      writer.writerow({"video": file, "score":
    ↪ round(comparison, 5)})
55
56                      if i == 0:
57                          video_match = file
58                          video_match_value = comparison
59                      else:
```

```
                        # Higher score = better match (Correlation and Intersection)
                        if m in [0, 2] and comparison > video_match_value:
                            video_match = file
                            video_match_value = comparison
                        # Lower score = better match (Chi-square, Alternative chi-square, Hellinger and KL Divergence)
                        elif m in [1, 3, 4, 5] and comparison < video_match_value:
                            video_match = file
                            video_match_value = comparison

                # append video match found to results list (using weights)
                if cur_all_model == "gray":
                    for _ in range(0, self.histogram_comparison_weigths['gray'], 1):
                        self.results_array.append(video_match)
                elif cur_all_model == "rgb":
                    for _ in range(0, self.histogram_comparison_weigths['rgb'], 1):
                        self.results_array.append(video_match)

                print_terminal_table(table_data, method)
                print("{} {} match found: ".format(_get_chosen_model_string(cur_all_model), method) + "\x1b[1;31m" + video_match + "\x1b[0m" + "\n\n")
```

Matching 3D Histograms

Compares 3D histograms, concerning the HSV histograms only using *SciPy's* statistical distances functions.

```
def match_histograms(self, cur_all_model="all"):
    # variables used for finding the match to the recorded video
    video_match = ""
    video_match_value = 0
    # get histogram for the recorded video to match
    hsv_data = np.loadtxt("../histogram_data/{}/hist-hsv.txt".format(self.file_name))
    query_histogram = {
        'hsv': hsv_data.reshape((8, 12, 3))
    }

    print("\n{} Histogram Comparison Results:\n".format(_get_chosen_model_string(cur_all_model)))
    method = ""
    csv_field_names = ["video", "score"]
    if config.model == "hsv":
```

```
15          for m in self.histcmp_3d_methods:
16              if m == "earths_mover_distance":
17                  method = "EARTH'S MOVER DISTANCE"
18              elif m == "energy_distance":
19                  method = "ENERGY DISTANCE"
20              # CSV file to write data to for each method
21              csv_file =
    ↪ open("../results/csv/{}-{}-{}.csv".format(config.model,
    ↪ cur_all_model, method), 'w')
22              with csv_file:
23                  writer = csv.DictWriter(csv_file,
    ↪ fieldnames=csv_field_names)
24                  writer.writeheader()
25                  table_data = list()
26                  for i, file in
    ↪ enumerate(get_video_filenames("../footage/")):
27                      dbvideo_hsv_histogram_data =
    ↪ np.loadtxt("../histogram_data/{}/hist-hsv.txt".format(file))
28                      dbvideo_hsv_histogram =
    ↪ dbvideo_hsv_histogram_data.reshape((8, 12, 3))
29                      comparison = 0
30                      for h in range(0, self.bins[0]):  # loop
    ↪ through hue bins
31                          for s in range(0, self.bins[1]):  # loop
    ↪ through saturation bins
32                              query_histogram_slice =
    ↪ query_histogram['hsv'][h][s]
33                              dbvideo_histogram_slice =
    ↪ dbvideo_hsv_histogram[h][s]
34                              if method == "EARTH'S MOVER DISTANCE":
35                                  comparison +=
    ↪ wasserstein_distance(query_histogram_slice,
    ↪ dbvideo_histogram_slice)
36                              elif method == "ENERGY DISTANCE":
37                                  comparison +=
    ↪ energy_distance(query_histogram_slice,
    ↪ dbvideo_histogram_slice)
38
39                      # append data to table
40                      table_data.append([file, round(comparison,
    ↪ 5)])
41                      # write data to CSV file
42                      writer.writerow({"video": file, "score":
    ↪ round(comparison, 5)})
43
44                      if i == 0:
45                          video_match = file
46                          video_match_value = comparison
47                      else:
48                          if comparison < video_match_value:
49                              video_match = file
50                              video_match_value = comparison
51
```

```
52              # append video match found to results list (using
    ↪ weights)
53              for _ in range(0,
    ↪ self.histogram_comparison_weigths['hsv']):
54                  self.results_array.append(video_match)
55
56              print_terminal_table(table_data, method)
57              print("{} {} Match found:
    ↪ ".format(_get_chosen_model_string(cur_all_model), method) +
    ↪ "\x1b[1;31m" + video_match + "\x1b[0m" + "\n\n")
```

H.2.9 Shot Boundary Detection Algorithm

Compares consecutive frames' RGB histograms using the Intersection metric with a global threshold approach. If the metric is smaller than the specified threshold, then a shot boundary has been detected.

```
1  def rgb_histogram_shot_boundary_detection(self, threshold):
2      x_axis = list()
3      y_axis = list()
4      is_under_threshold = True
5      ret, frame = self.video_capture.read()  # get initial frame
6      frame_counter = 0  # cur frame ID
7      shot_changes_detected = 0  # No. shot boundaries detected
8      while self.video_capture.isOpened():
9          prev_frame = frame[:]  # previous frame
10         ret, frame = self.video_capture.read()  # read video
11         if ret:
12             frame_counter += 1
13             cur_rgb_hist = {
14                 'b': list(),
15                 'g': list(),
16                 'r': list()
17             }
18             prev_rgb_hist = {
19                 'b': list(),
20                 'g': list(),
21                 'r': list()
22             }
23             for i, col in enumerate(self.colours):
24                 # calculate RGB histograms
25                 cur_frame_hist = cv2.calcHist([frame], [i], None,
    ↪ [256], [0, 256])
26                 prev_frame_hist = cv2.calcHist([prev_frame], [i],
    ↪ None, [256], [0, 256])
27                 # normalise histograms
28                 cur_frame_hist =
    ↪ _normalise_histogram(cur_frame_hist)
29                 prev_frame_hist =
    ↪ _normalise_histogram(prev_frame_hist)
30                 # save histograms in dict
31                 cur_rgb_hist[col].append(cur_frame_hist)
```

```
32                  prev_rgb_hist[col].append(prev_frame_hist)
33
34              # calculate Intersection between consecutive frames
35              comparison_r = cv2.compareHist(prev_rgb_hist['r'][0],
    ↪ cur_rgb_hist['r'][0], cv2.HISTCMP_INTERSECT)
36              comparison_g = cv2.compareHist(prev_rgb_hist['g'][0],
    ↪ cur_rgb_hist['g'][0], cv2.HISTCMP_INTERSECT)
37              comparison_b = cv2.compareHist(prev_rgb_hist['b'][0],
    ↪ cur_rgb_hist['b'][0], cv2.HISTCMP_INTERSECT)
38              comparison = (comparison_b + comparison_g +
    ↪ comparison_r) / 3
39              # For KL Divergence, use cv2.HISTCMP_KL_DIV
40
41              # append data to lists for plot
42              x_axis.append(frame_counter)
43              y_axis.append(comparison)
44
45              # check if difference big enough for shot boundary
46              if comparison < threshold and is_under_threshold:
47                  shot_changes_detected += 1
48                  is_under_threshold = False
49                  print("Scene Change detected at Frame
    ↪ {}".format(frame_counter))
50              elif comparison > threshold:
51                  is_under_threshold = True
52          else:
53              break
54
55      # Plot results
56      plt.plot(x_axis, y_axis)
57      plt.plot(x_axis, np.full(frame_counter, threshold))
58      plt.title("Intersection Between Consecutive Frame RGB
    ↪ Histogram")
59      plt.xlabel("Frame")
60      plt.ylabel("Intersection")
61      plt.show()
62      print("\n—— Number of shot changes detected: {}
    ↪ ——".format(shot_changes_detected))
```

H.2.10 Frames Pre-Selection Function

Returns the IDs of the frames to calculate a histogram for (1 frame per second).

```
1 def _get_frames_to_process(vc):
2     frame_ids = list()
3     total_frames = vc.get(cv2.CAP_PROP_FRAME_COUNT)
4     fps = vc.get(cv2.CAP_PROP_FPS)
5     for i in range(1, int(total_frames) + 1, math.ceil(fps)):
6         frame_ids.append(i)
7     return frame_ids
```

H.3 Video Operations Module

The *video_operations.py* module contains the code used to perform operations on the query video. It takes care of the pre-processing steps, including video stabilisation and manual ROI selection.

H.3.1 VideoStabiliser Class

Class used to stabilise the recorded video for more optimal matching.

```
from pyspin.spin import make_spin, Box1
from vidstab import VidStab

class VideoStabiliser:
    def __init__(self, directory, file_name):
        """Initialise variables and start stabilising"""
        self.directory = directory
        self.file = file_name
        self.new_file = file_name[:-4]
        self.stabiliser = VidStab()
        self.stabilise_video()

    @make_spin(Box1, "Stabilising video...")
    def stabilise_video(self):
        self.stabiliser.stabilize(input_path="{}{}".format(
    ↪ self.directory, self.file),
    ↪ output_path="{}/stable-{}.avi".format(self.directory,
    ↪ self.new_file), border_type="reflect")
        print("\nVideo stabilised!")
```

H.3.2 ClickAndDrop Class for Manual ROI Selection

Class for selecting a region of interest on a frame by click and dropping the mouse over the desired area, and cropping that frame to include the pixels inside the ROI only.

```
import cv2

class ClickAndDrop:
    window_name = "Crop the recording (top-left click ->
    ↪ bottom-right drop) - 'C' to crop - 'R' to restart"

    def __init__(self, thumbnail):
        """Loads image to crop + controls the callback loop."""
        self.thumbnail = thumbnail
        self.reference_points = list()
        self.cropping = False

        # load image, clone it, setup the mouse callback
        self.image = cv2.resize(self.thumbnail, (1280, 720),
    ↪ interpolation=cv2.INTER_AREA)
```

```
14          clone = self.image.copy()
15          cv2.namedWindow(self.window_name)
16          cv2.setMouseCallback(self.window_name,
    ↪ self.click_and_crop)
17
18          # keep looping until the 'q' key is pressed
19          while True:
20              # display the image and wait for a keypress
21              cv2.imshow(self.window_name, self.image)
22              key = cv2.waitKey(1) & 0xFF
23              if key == ord("r"):  # reset the cropping region
24                  self.image = clone.copy()
25              elif key == ord("c"):  # break from the loop
26                  break
27
28          # crop the ROI from the image
29          if len(self.reference_points) == 2:
30              self.roi =
    ↪ clone[self.reference_points[0][1]:self.reference_points[1][1],
31
    ↪ self.reference_points[0][0]:self.reference_points[1][0]]
32          cv2.destroyAllWindows()
33
34      def click_and_crop(self, event, x, y, flags, param):
35          """
36          Callback function for user to manually crop image.
37          NOTE: must crop from top-left corner to bottom-right
    ↪ corner
38          """
39          # if left mouse button clicked, record the starting (x,
    ↪ y) coordinates and indicate that cropping is being performed
40          if event == cv2.EVENT_LBUTTONDOWN:
41              self.reference_points = [(x, y)]
42              self.cropping = True
43
44          # check if left mouse button was released
45          elif event == cv2.EVENT_LBUTTONUP:
46              # record ending (x, y) coordinates and indicate that
    ↪ the cropping operation is finished
47              self.reference_points.append((x, y))
48              self.cropping = False
49
50              # draw a rectangle around the region of interest
51              cv2.rectangle(self.image, self.reference_points[0],
    ↪ self.reference_points[1], (0, 255, 0), 2)
52              cv2.imshow(self.window_name, self.image)
53
54      def get_roi(self):
55          return self.roi
56
57      def get_reference_points(self):
58          return self.reference_points  # ROI reference points
```

H.4 Helpers Module

This module contains multiple general-use functions with various uses. For example, functions for retrieving file names, removing file extensions from filenames, displaying the final results results in the form of plots or console outputs, parsing command line input and getting video information can be found in this module.

```
def get_video_filenames(directory):
    """Returns a list containing all the mp4 files in a
    ↪ directory"""
    list_of_videos = list()
    for filename in os.listdir(directory):
        if filename == ".DS_Store":
            pass # ignoring .DS_Store file (for macOS)
        elif filename.endswith(".mp4"):
            list_of_videos.append(filename)
        else:
            print("no mp4 files found in directory
    ↪ '{}'".format(directory))
    return list_of_videos

def print_terminal_table(table_data, method_used):
    """Prints a table with the results in the terminal."""
    table = DoubleTable(table_data)
    table.title = method_used
    table.inner_heading_row_border = False
    table.inner_row_border = True
    print(table.table)

def print_finished_training_message(answer, model, runtime,
    ↪ accuracy=None):
    """Prints a message at the end of the training function."""
    print("\n\nGenerated " + "\x1b[1;31m" + "{}".format(model) +
    ↪ "\x1b[0m" + " histograms for all videos")
    if accuracy is not None:
        print("\n\n" + "\x1b[1;31m" + "MATCH FOUND:
    ↪ {}".format(answer) + "\x1b[0m")
    print("\n--- Runtime: {} seconds ---".format(runtime))
    if accuracy is not None:
        print("--- Accuracy: {} % ---".format(round(accuracy *
    ↪ 100, 2)))

def get_video_first_frame(video, path_output_dir, is_query=False,
    ↪ is_result=False):
    """Retrieves the first frame from a video and saves it as a
    ↪ PNG."""
    vc = cv2.VideoCapture(video)
    frame_counter = 0
    while vc.isOpened():
        ret, image = vc.read()
        if ret and frame_counter == 0:
            if is_query:
```

```
38                  cv2.imwrite(os.path.join(path_output_dir,
    ↪ "query.png"), image)
39              elif is_result:
40                  cv2.imwrite(os.path.join(path_output_dir,
    ↪ "result.png"), image)
41              frame_counter += 1
42          else:
43              break
44      cv2.destroyAllWindows()
45      vc.release()
46
47  def show_final_match(result_name, query_frame, result_frame,
    ↪ runtime, accuracy):
48      """Plots the query image and the matched video."""
49      query_img = mpimg.imread(query_frame)
50      result_img = mpimg.imread(result_frame)
51      plt.subplot(2, 1, 1)
52      plt.imshow(query_img)
53      plt.title("Original Query Video", fontSize=16),
    ↪ plt.xticks([]), plt.yticks([])
54      plt.subplot(2, 1, 2)
55      plt.imshow(result_img)
56      plt.title(
57          "Match '{}' found in {}s with {}%
    ↪ accuracy".format(result_name, runtime, round(accuracy *
    ↪ 100, 2)),
58          fontSize=13)
59      plt.xticks([])
60      plt.yticks([])
61      plt.show()
62
63  def display_results_histogram(results_dict):
64      """Displays the results in the form of a histogram."""
65      fig = plt.figure()
66      ax = fig.add_subplot(111)
67      ax.bar(list(results_dict.keys()), results_dict.values())
68      plt.title("Probability of a match for most likely videos")
69      plt.ylabel("%")
70      plt.tight_layout()
71      plt.setp(ax.get_xticklabels(), fontsize=10,
    ↪ rotation='vertical')
72      plt.show()
73
74  def get_number_of_frames(vc):
75      """Retrieves the total number of frames in a video using
    ↪ OpenCV's VideoCapture object cv2.CAP_PROP_FRAME_COUNT
    ↪ attribute."""
76      return int(vc.get(cv2.CAP_PROP_FRAME_COUNT))
77
78  def get_video_fps(vc):
79      """Retrieves the frame rate (Frames Per Second) of a video
    ↪ using OpenCV's VideoCapture object cv2.CAP_PROP_FPS"""
80      return round(vc.get(cv2.CAP_PROP_FPS), 2)
```

```
81
82 def terminal_yes_no_question(question, default="no"):
83     """Ask a yes/no question via input() and return the answer as
   ↪ a boolean."""
84     valid = {"yes": True, "y": True, "no": False, "n": False}
85     if default is None:
86         prompt = " [y/n] "
87     elif default == "yes":
88         prompt = " [Y/n] "
89     elif default == "no":
90         prompt = " [y/N] "
91     else:
92         raise ValueError("invalid default answer: '%s'" % default)
93     while True:
94         sys.stdout.write(question + prompt)
95         choice = input().lower()
96         if default is not None and choice == '':
97             return valid[default]
98         elif choice in valid:
99             return valid[choice]
100        else:
101            sys.stdout.write("Please respond with 'yes' or 'no'
   ↪ (or 'y' or 'n').\n")
102
103 def video_file_already_stabilised(filepath):
104     """Checks if the path to a stable version of the video
   ↪ already exists."""
105     if os.path.isfile(filepath):
106         return True
107     return False
```