NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

CLÁUDIO NUNO RODRIGUES PEREIRA
Bachelor in Computer Science

DYNAMIC CONTENT-BASED INDEXING IN MOBILE EDGE NETWORKS

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
September, 2021

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

DYNAMIC CONTENT-BASED INDEXING IN MOBILE EDGE NETWORKS

CLÁUDIO NUNO RODRIGUES PEREIRA

Bachelor in Computer Science

Advisers: Hervé Miguel Cordeiro Paulino
Associate Professor, NOVA University Lisbon
Nuno Miguel Cavalheiro Marques
Assistant Professor, NOVA University Lisbon

Examination Committee:

Chair: Carla Maria Gonçalves Ferreira
Associate Professor, NOVA University Lisbon

Rapporteur: Luis Manuel Pereira Sales Cavique Santos
Assistant Professor, Universidade Aberta

Adviser: Hervé Miguel Cordeiro Paulino
Associate Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
September, 2021

Dynamic Content-based Indexing in Mobile edge Networks

To my grandmother, stepfather and my mother.

Acknowledgements

I would like to begin and greatly thank both my advisor and co-advisor, professors Hervé Paulino and Nuno Marques for all the guidance, teachings and all the unconditional and restless support during the elaboration of this dissertation. Due to the global pandemic most lifestyles where altered to a more difficult environment to work on however, they would always find time to help, discuss and guide me through the elaboration of this dissertation. I would also like to thank the NOVA School of Science and Technology and the Informatics Department for being such a great home and providing such memorable events during these last 6 years.

I would like to thank all my family, for their immense support done during all these years, everything I achieved could not be done without them. I would like to thank my mother and grandmother for being always present for both the good and bad times, for being always available for me and for always believing in me. I also want to thank my stepfather Vitor, for all his teachings, patience and for always believing in me.

I would also like to thank my friends, who were so important to me during these years. A thank you from the bottom of the heart to my long lasting friends - Miguel Mira, Maria Inês Santos and Catarina Anastácio - for always being there for me, and for all the spontaneous hangouts during those difficult times, to the friends I had the luck to meet during these years - Bruno Anjos, Bruno Calapez, Eduardo Fernandes, João Mota, João Antão, Lucas Dias e Rafael Conceição - who where always available for more than just work and a personal thanks to Daniel Singh for accompanying me since childhood and always sharing me his wisdom.

This work was partially supported by Fundação para a Ciência e Tecnologia (FCT-MCTES) in the scope of the DeDuCe (PTDC/CCI-COM/32166/2017) research project.

Abstract

Recently, we have seen a huge growth in the usage of mobile devices, and with this growth, the data generated has also increased, being in a huge scale, user generated, e.g, photos, books, texts or messages/e-mails. Usually this data requires a permanent storage and its respective indexing for users to efficiently access it however, due to the unpredictability of this data, a concern regarding its indexing starts to raise as it can be hard to predict labels and indexes capable of representing every possible set of data.

For instance, during a birthday party, users may want to share photos and videos of this event which can be seen as uploading streams of data to a content sharing system. This content stream will most likely have no index, unless it is explicitly generated, making its retrieval difficult. However, when clustering this stream, as data keeps increasing, we might, somewhere in the future, be capable of detecting similarities between each photo (e.g. a guest's face) and might want to index them. Indices can directly impact a system's performance however, there is a drawback from having either too many or too few indices, posing a challenge when it comes to evolving content.

We propose *Chives*, a Content-Based Indexing framework, built on top of a content sharing publish/subscribe system at the edge named *Thyme*, where we evaluate unsupervised learning in data stream techniques to generate indices. It also offers a content-based query to automatically subscribe to indices containing similar content, e.g images.

After evaluating our proposal in a simulated environment, we can see that our framework offers a great abstraction, allowing an easy extension, furthermore our implementation can generate indices from data streams and the indexing follows a clustering criteria, generating the indices as conditions are met. Furthermore, results show that our clustering quality and consequently its generated indices rely strongly on the quality of the image discrimination and its ability to extract features representing its face. In Conclusion, more studies should be done regarding this framework as such, our solution is built in a way where we can exclusively study each component and upgrade it in future work.

Keywords: Content Sharing at the edge, Machine Learning, Unsupervised Learning, Content-Based Indexing, Computer Vision

Resumo

Recentemente, tem-se observado um enorme crescimento na adesão a dispositivos móveis e com este crescimento, tem também aumentado a quantidade de dados partilhados, sendo em grande escala, gerado pelos utilizadores, por exemplo, fotos, livros, textos ou até mensagens/*e-mails*. Normalmente estes dados necessitam de um local de armazenamento permanente e a sua respectiva indexação de modo a poderem ser acedidos de forma eficiente por parte dos utilizadores no entanto, dada a imprevisibilidade destes dados, pode surgir um problema relativamente à indexação dado que poderá ser difícil prever etiquetas e índices capazes de representar qualquer conjunto de dados.

Por exemplo, durante uma festa de anos, utilizadores poderão partilhar fotografias e vídeos deste evento que poderá ser também interpretado como um upload de dados em *stream* para um sistema de partilha de conteúdo. Esta *stream* de dados, muito provavelmente não terá nenhum índice capaz de o descrever, tornando difícil a obtenção deste visto que não existe representação semântica desta. No entanto, ao agrupar esta *stream*, à medida que os dados vão crescendo, poderemos, algures no tempo ser capaz de detectar semelhanças entre cada fotografia (por exemplo. a cara de um convidado) e podemos querer indexar. Índices podem causar um impacto directo sobre o sistema, no entanto o inverso pode acontecer quando existe índices em défice ou em excesso, apresentando um desafio acerca de dados evolutivos.

Nós propomos uma framework de indexação baseada em conteúdo, construído por cima de um sistema de partilha de conteúdo que usa um sistema de *Publish/Subscribe* na *edge* denominado *Thyme*, onde avaliamos técnicas de aprendizagem não supervisionada em data streams para gerar dinamicamente índices.

Depois de avaliar a nossa framework, conseguimos concluir que esta oferece uma boa abstração, facilitando a sua extensão, para além disso a nossa proposta permite gerar índices quando as condições definidas para o clustering são respeitadas. Para além disso, os resultados demonstram que o clustering realizado pelo nosso algoritmo dependem fortemente da qualidade de discriminação de imagens e das características obtidas por este discriminador em relação às faces. Concluindo, mais estudos devem feitos em relação à framework, como tal esta foi construída de modo a permitir uma rápida e fácil extensão

para futuros melhoramentos.

Palavras-chave: Partilha de conteúdo na *edge*, Aprendizagem Automática, Aprendizagem não supervisionada, Indexação baseada em conteúdo, *Android*, *Computer Vision*

Contents

List of Figures

List of Tables

List of Listings

Acronyms

CRDT Conflict-Free Replicated Data Type 25, 26, 45, 48

IoT Internet of Things 1

1 Introduction

1.1 Context & Motivation

Looking at past years, we have seen an incredible technological evolution. Mobiles are now as powerful as computers available some years ago and we reached to a point where computation has become ubiquitous. Following this evolution, users have also become closer and closer to these technologies, up to a point where almost everyone has a mobile, a computer or a wearable.

In 2018, 2.9 Billions of users worldwide were using smartphones and this number was expected to increase [58]. Figure 1.1 shows mobile users worldwide from 2016 to 2021, plus forecasts regarding the following years.

This technological synergy led to the implementation of brand new services and applications with the main objective of trading their services for personal information. The Internet of Things (IoT) has gained popularity in such a way that we can see IoT-related technologies deployed in public places or even in our homes. These systems are usually used for information gathering or conditional computations, e.g. if certain data changes

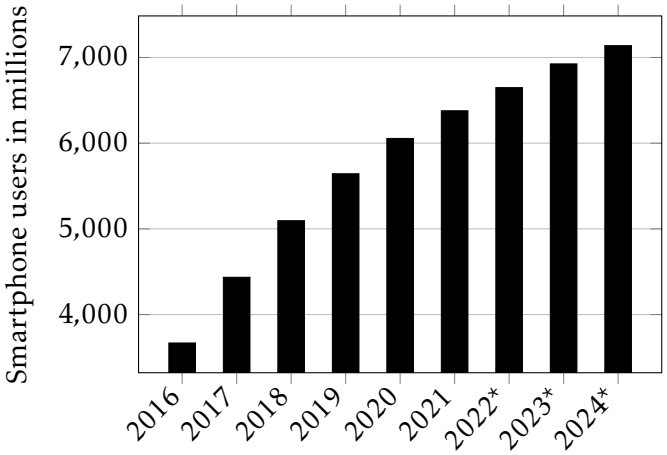

Figure 1.1: Number of smartphone users worldwide from 2016 to 2024(forecast). Adapted from [58].

beyond a defined threshold. Furthermore, Social media are a trending example, where images are constantly being published, generating a huge amount of visual data where users play a huge part in its production.

All these systems are constantly generating data while attending to data retrieval requests for various reasons. This raises a problem for centralized services: as the amount of data increases, it becomes more difficult to find the requested data due to the increasing searching space.

To improve the searching of content, systems use indices. An index is a value used for labeling a certain piece of data, providing a fast and simple method to get to the data itself, as now we only need to know the value used for the index. Indices are used in most systems as they speed up all searches, and decrease computation time, however misusing them may do the exact opposite.

1.2 Edge Computing

Mahadev Satyanarayanan defines *Edge Computing* as "a new paradigm in which substancial computing and storage resources-variously referred to as cloudlets, micro datacenters, or fog nodes-are placed at the Internet's edge in close proximity to mobile devices or sensors." [64]

Edge computing proposes an infrastructure with a higher proximity to the user in mind. This can be achieved by placing intermediate computer resources at the edge of the network [27]. This proximity is of big importance as it can reduce infrastructure's overall latency by two means. If the edge server is able to handle some of the (less powerful) tasks, it can attend to the user's request, hence reducing network hops and reducing user request travel time. If the edge server is not able to handle a more powerful request, it then redirects the request to the main server. This approach allows the edge servers to answer the less powerful requests, decreasing the network congestion directed to the main server, hence reducing latency in resolving these requests [25].

However there are also challenges that must be taken into consideration. These challenges include overall management, security [64], resource limitation and programming abstractions.

Regarding the data and programming abstractions, it comes a challenge in which server should handle a request in order to guarantee as low latency as possible but also the optimal and most accurate response. For instance, to endorse privacy, it may be good idea to process data right at the edge and then send the result to the core server, but if this processing requires heavy computations we need to question whether or not it should be done instead at the core. Another question we can make is whether or not a less expensive computation should be done instead in order to guarantee a fast response even if it causes a less optimal result.

Resource limitation comes from the edge infrastructure and must be carefully endorsed since they are the main distinction between the main server and its edge. This

creates, specifically for mobile edge computing, more aspects to consider whilst planning a system under this paradigm. There is a trade-off between resource saving and infrastructure latency.

1.3 Problem

Our aim with this thesis is to define and implement a new framework for content-based image retrieval capable of generating indexes automatically as needed.

Let us assume a common scenario like a social event, for instance, a birthday party. During the event, users will produce photos and share with other users using an arbitrary content-sharing system. These photos can be seen as a data stream that will be submitted continuously and might represent distinct scenarios (e.g. a birthday party or a concert) and distinct people that may not be known beforehand, making infeasible to the system to index it. With the ongoing of the event, users might want to retrieve content shared from the other users, however they will not be able to efficiently retrieve it as the content is not indexed.

This can be avoided since, as content keeps being published, the size of this data increases and may begin to present some patterns, e.g, several photos containing a new party guest that could arrive. These patterns would allow the content to be grouped, using similarity metrics, increasing its size as more similar content is published.

These groups would then increase into a point where not only it would be recommended to index, but the system could also make assumptions about the label to assign it to, for instance, using the above example, it could assume the new guest's name.

During our research, we found work on dynamic indexing [35, 41] and on content-based indexing applications [47, 74] however, to the best of our knowledge, no system was found that dynamically generates content-based indices with little to no knowledge regarding the data stream.

1.4 Proposed Solution

We propose a Content-Based Indexing framework, built on top of a content sharing publish/subscribe system at the edge named Thyme, which uses the unsupervised learning clustering nature to offer a novel data retrieval framework.

This system will receive data and, using unsupervised learning, group the content based on a similarity metric. For this thesis we will focus on images of faces, thus meaning that each cluster will contain similar faces.

The system will also keep track of each cluster, evaluating if it meets certain conditions and indexing in case it does, allowing a dynamic indexing of the uploaded content.

For the index value the system will evaluate the content inside the cluster and, using a popularity heuristic, generate a label capable of representing the content inside of it.

Furthermore, this system will offer two efficient and distinct content-retrieval methods:

- Semantic Based Retrieval which receives a index and returns the content being represented by that value.
- Content Based Retrieval which receives an image and returns to the user images considered similar.

1.5 Challenges

Regarding our solution there are several challenges that must be faced:

1. How can we efficiently extract facial features?
2. How can we group images based only on their similarity?
3. How do we index the result of the grouping, so a user can retrieve the content, let it be using content-based search or semantic?
4. How can we dynamically index a cluster?

There are also some extra considerations that needs to done regarding some challenges. Regarding the second point, we must also consider two aspects:

- Since there is initially no shared content in the system, we must consider strategies that can evolve from a state where no data was shared yet.
- Due to the features of the system, data will be published continuously, hence strategies considering data streams are important for the efficiency of the implementation.

Finally, being this system implemented over Thyme, another challenge raises as this framework is implemented with the edge computing paradigm in mind, thus providing a non optimal environment for most of the algorithms used for the above mentioned challenges.

1.6 Contributions

With this dissertation, our expected contributions would be as follows:

- Implementation of a content-based indexing system, located at the edge capable of generating indexes for content as needed whilst providing two distinct query methods.
- A comparative study between different feature extractors and clustering algorithms.
- Performance evaluation of our proposal, namely the index generation and similarity between content.

1.7 Document Outline

The remainder of this document is organized as follows. In Chapter 2 we introduce the major areas of interest regarding the context of this thesis and state-of-the-art implementations of technologies regarding those areas In chapter 3 we make a brief description of Thyme [69] and Oregano [63] which are the building blocks for our solution. In chapter 4 we deeply describe our proposal, beginning with an overview of the framework and how to extend it, and the workflows for each agent intervening in the system, followed by a description of the designed architecture to attend this problem and our contributions. In chapter 5 we describe the tests done to validate our solution and conclusions drawn from these validations. We end the document with conclusions drawn from this thesis and an enumeration of future work points in Chapter 6.

2

State-of-the-Art

The purpose of this chapter is to point out relevant concepts, implementations and similar work regarding this thesis. For the proper development of our solution, we will approach each problem with research done regarding that area. We will start by presenting research done regarding Face Detection and Feature Extraction techniques at section 2.1. Next we will describe and summarize our main problem regarding the content-based image retrieval and its indexing in Section 2.2, ending this chapter with a description of related work in Section 2.3 and concluding remarks in Section 2.4.

2.1 Face Detection & Feature Extraction

For this thesis, since the case study will be focused on image retrieval (more specifically images with faces), the extraction should result in facial features. In order to successfully extract these features, it is necessary to first detect the faces. After correct detection of the faces, it is then necessary to extract descriptive, comparative values from these.

Regarding our system, we can either do this task at the edge server or at the mobile devices themselves. This task is required both when the user publishes new content and when it queries based on content. When the user publishes new content, it is expected from the system to store it and share between other users, therefore assigning this task at the mobile level would require each device to upload the content plus its features. This increases network load and may not present any improvement to the overall system at all. Moreover, assigning this task to the mobile devices during queries would bring several improvements to the whole system, including:

Privacy Enforcement: The image used for the query would never be uploaded to the edge node but instead only the resultant features extracted from it. This would ensure that other privacy-sensitive content would not be submitted.

Increase availability: Assigning these tasks to the device would decrease the number of operations done at the edge and the size of the data to upload, thus decreasing the time necessary to attend the request and increasing server availability.

However this would bring a major challenge to our solution, falling inside the resource limitations and management of a mobile device as it forces the algorithm to be as lightweight as possible. Furthermore, assigning this task to the devices may increase operation time due to its limited resources, thus increasing query's overall time. These challenges could be solved when assigning this task to an edge server, but would cause an overall latency increase, since the edge would now need to do a lot more operations to attend a single request.

In this scenario, if the algorithm is too heavy, it could also be interesting to implement an hybrid solution, allowing more powerful devices to do the extraction locally whilst not making mandatory for less powerful cellphones.

Regarding the process, we can divide this task into two main stages: detection and feature extraction. These stages represent two common computer vision problems and their proposals have been increasingly machine learning algorithms.

2.1.1 Machine Learning

In this section we introduce Machine Learning and some important concepts regarding this area as it will be an important branch for our solutions. The following content are mostly adapted from [9, 44, 53, 56].

Machine Learning is a branch of computer science that focuses on building systems or algorithms that can improve with the collection of more data or by past experience. Machine Learning shines by its autonomy. It allows us to solve problems where we do not have, or do not want to make, an algorithm but can compensate with data, both quantitatively and qualitatively.

Looking at our scenario, we struggle to implement an algorithm that is capable of grouping images by their similarity and even if we find a good approximation, we know beforehand that it would be computer expensive. However we know that its core functioning will be to, as data is received, detect relevant features and then search for any group that contains similar features.

However, in order to effectively formulate a machine learning problem, [44] exposes three main issues that must be considered:

- Which task must be performed?
- How should we evaluate the performance of the system, so we can compare each solution?
- What data do we have?

Taking a closer look to our problem, these questions would be answered as follows: The task would be to cluster content by its similarity, thus allowing to return the cluster as a content grouping. Regarding the evaluation of our learner, we will need to use cluster specific metrics, more details in Section 2.2.4. Regarding the existing data, initially

it would be none but, with continuous uploads from the users, this would gradually increase.

As we can see from the questions mentioned earlier, machine learning can have applications for a diversity of problems. Each task, can have different approaches based on the expected result and/or the problem analysis. For instance, in our scenario we could, from the same problem, do other two tasks. We might want to know exactly if an arbitrary image is similar to another given image, thus expecting either a "right" or "wrong". Alternatively we might want to know how similar the images are.

To endorse each problem in Machine Learning, different algorithms and approaches have emerged. We can split these algorithms into two main groups [53], following we list these groups and briefly describe each.

Supervised Learning A data set is provided and their respective classification. The goal of this type of learning is to, whilst training, for each pattern learns how to provide the best classification and then use the classifier to determine the classification of the unknown data.

Unsupervised Learning uses a data set without any classification. The objective of this technique is to find similarities between the given data, resulting in several clusters each with as much similarities as possible. More on clustering in Section 2.2.2. This learning process receives a structure of the data unlabelled, and adjusts a model based only on the data itself. Since labelling is not expected in this learning paradigm, unsupervised learners do not consider them. The results of this model often expose interesting relations between the data, being some capable of human interpretation, thus allowing interesting results. However, as data does not have any label, the traditional evaluation methods can not be used to evaluate the learner's performance and the quality of the model since it is not possible to measure the error with reference to known labels [56]. For our thesis, we will focus solely on this type of learning as it will be the one used in our strategies.

Conclusively, machine learning can be used for a wide variety of problems, however, due to the high amounts of data processing done by machine learning algorithms, two concerns emerge regarding its usage. First we might need an enormous data set in order to guarantee more accurate solutions, requiring information gathering before engaging the problem. For some problems, this requirement can be mandatory. Second, since, as mentioned, these algorithms require high data processing, it might be needed a lot of computer resources. Although some algorithms offer optimizations regarding this issue, they always eventually converge to this need in order to process the data in less time and more efficiently.

However, data is being generated continuously in huge amounts. It is estimated that in 2025, 175 zetabytes of data will the be created worldwide [78] (more data in Figure 2.1).

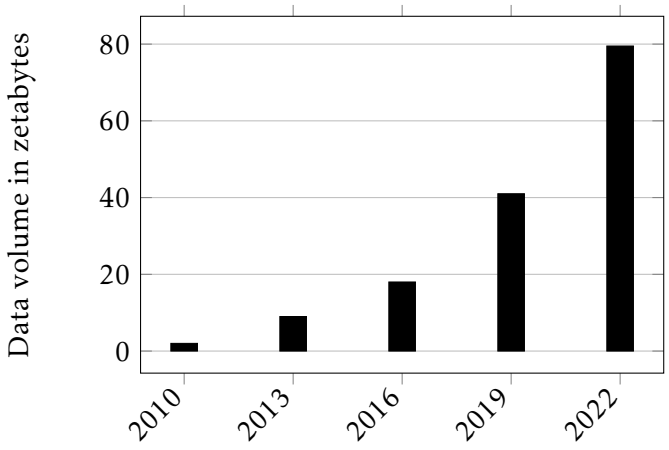

Figure 2.1: Data generated worldwide from 2010 to 2022 (forecast). Adapted from [78]

Furthermore, all systems gradually evolving at a fast pace [29], thus enabling a faster processing of the data. With this increase, not only machine learning is slowly becoming more and more accessible but is also becoming more useful. This led to such a huge increase in its popularity that nowadays its hard to find systems not using this technology to offer a better user experience.

2.1.2 Features

When tackling a machine Learning problem it is often practical to process the input data and convert them to a new set that might facilitate the original problem [15] This new set of data can also be viewed as a set of features, and should not be confused with its computer vision counterpart as in Machine Learning we describe features as a simple, measurable, data unit, and are usually compiled to be represented as a vector, where for Computer Vision it usually represents a new piece of information regarding the content of an image or part of it, let it be a human-readable value (for example a color) or a numerical, less readable value (e.g an eigenvalue).

Being our thesis focused on images with faces, it is important to implement a processing stage that can rewrite our input to a set of features that can be fed to our machine learning system. For instance, when interpreting an image, we can process it, using a computer vision algorithm, to look for features in the input images and then transform the resulting features into a set of data that can later be used by our machine algorithm. Furthermore, when processing an image of a face other, more specific, features come to mind that can be used as a good descriptor, namely the eyes, mouth, nose. In the following sub-sections, we will describe procedures and adopted techniques in the fields of computer vision to aid our processing stage for extraction of these features, thus allowing the transformation of our images into data that can be analysed by our machine learning system.

2.1.3 Face Detection

When an image is published or used as a query, its content needs to be reduced into identifiable faces, which can be done instantly and almost effortless by a human but the same does not apply to describing an algorithm with the same objective. For that we need to use algorithms capable of detecting any faces existing in an arbitrary image, falling inside a computer vision problem. Since this algorithm is crucial for reduction of the image, we need to use robust algorithm, capable of identifying, in most cases and under any conditions let it be face rotation or distance, the correct faces whilst maintaining a low computational power requirements.

During our research we found several classifiers from whom we will list and describe some relevant discoveries.

For feature based detection methods, we found Haar Classifier and LBP Classifier that are lightweight but present some issues, e.g, they might not detect rotated faces affecting the overall results of the system. However, some proposals were also found, offering an upgrade to these algorithms by making them rotation-invariant[8, 54].

Another popular technique discovered is the Cascade Classifier [76].

The Cascade Classifier [76] is a machine learning approach for face detection, which uses a ensemble method named Adaptive Boost, or AdaBoost [32]. This approach was rapidly improved to a faster and more robust face detection algorithm [77].

During our research, we discovered available state-of-the-art implementations for these three algorithms using the Open Source Computer Vision library (also referred as OpenCV) [1]. OpenCV is an open source, cross-platform computer vision and machine learning library [1] that offers several optimized algorithms for these areas and can help improving the development of our solution with its Java [40] and Android [2] interface.

Alternatively to these algorithms we can make use of Deep learning, which made huge leaps in the computer vision area. Deep learning is a branch of machine learning designated to every model containing layers of nonlinear transformations [45] until the final, output layer. These layers apply consecutively, meaning each nonlinear transformation apply to its previous layer, allowing the discovery of new representation of the data. This makes deep learning a broad concept which can be used for any "traditional" machine learning method, let it be supervised or unsupervised.

However, due to this adaptability, these models can easily result in overfitting, which happens when a model, while learning, tries to adapt too much to data, thus being evaluated in training by a low error value, but as soon as new data appears (in example when testing with new data) it presents poor results due to this over-adaptation. This should be avoided and in these techniques the best way of preventing is by using data that can represent the universe as much as possible, leading to a huge amount of data to be processed which, for more limited systems, might not be feasible to implement.

However, some pre-trained deep learning models are available online which can remove some complexity to our system. During our research we found the following software packages:

- **Deep Learning For Java (DL4J) [3]** is an open source, deep learning library written in Java [40], which allows us to compose deep neural nets from various shallow nets.
- **TensorFlow [4]** is an end-to-end open source platform for machine learning. It offers several state-of-the-art implementations and tools. It also offers a lightweight mobile-driven variant which allows the deployment of pre-trained models.

We also found three proposals: Cascade CNN [48], Multi view Face Detection [31] and R-CNN [20], however none of the software packages appeared to offer any available tools or modules that meet our problem requirements, thus making infeasible the use of deep learning in our solution.

In conclusion, deep learning allowed the creation of several new proposals using Convolutional Neural Networks that outperformed the traditional methods in computer vision problems, however, it is usually resource expensive to train the models, thus making infeasible its use without a state-of-the-art model. This makes the Cascade Classifier our possible choice as the face detection algorithm.

Being, as already mentioned, the face detection at the mobile level a big upgrade to our system, we researched and also discovered a library named Machine Learning Kit [55] developed by Google [5] which offers face detection and other machine learning related operations at the mobile level. However it is important to mention that, despite being possible to do face detection at the mobile level, it is still required to implement one at the edge level since it is still necessary for the edge to do the feature extraction by himself when content is published. This makes the openCV [1] library, a more interesting choice as it may reduce aleatory inconsistencies between both implementations.

2.1.4 Feature Extraction

After the images are reduced, the features can then be extracted using image descriptors. Image descriptors allow the comparison between each image using characteristics such as colors, textures and shapes [24, 72].

After some research, it was found some experimental comparisons [24, 38] between various image descriptors and combination techniques. These comparisons point out some aspects regarding this topic.

Firstly, color histograms tend to present relatively good results by average, furthermore they do not require high computational power in order to effectively perform the algorithm. However, due to its focus on the image colors, this descriptor can perform poorly with gray scale images, plus it can bring false positives and negatives when images present the same color spectrum but different shapes or vice-versa [24]. Furthermore,

color histograms can be, at the cost of higher computational power requirements, be surpassed by the local image descriptors [24]. Some popular local features are SIFT [51], SURF [11] and ORB [61].

Secondly, the accuracy of the results can be improved by combining features extracted from several image descriptions [24, 38].

During our research, we also discovered a texture-based image descriptor named Local Binary Pattern[37, 79]. This algorithm got its attention because it was also mentioned in facial features extraction.

For our thesis, we will focus mostly in comparing algorithms available using the openCV [1] implementations to aid our development. This evaluation should be prior as it is connected to every other part of the clustering processing, meaning that these results may also affect the system overall results.

2.2 Content-Based Data Retrieval & Indexing

Regarding this topic, for this thesis, we want to focus our research mostly in state-of-the-art implementations of algorithms and systems created for both Content-based data retrieval and Indexing. This allows an increase in productivity since it allows us to refrain from implementing all algorithms and systems necessary for the implementation of our solution.

In order to better understand our problem, we will split it into content-based data retrieval and indexing of the content. For the former, we need to implement a system capable of evaluating a specific set of features and group them by their similarity which must be reflected to the user as facial similarities. For the latter, we need to deliver to the user the result of its queries as images. Furthermore, it is important to generate new indexes as a content group starts to increase in size in order to allow a dynamic indexing of content.

2.2.1 Content-Based Retrieval

When published content is unknown and indexes have not been generated beforehand, there should be a way for the system to group this content, so it can later automatically label this grouping and generate the index. Furthermore, these content should still be available for users to request it via content-based searches, i.e., it should be possible for the user to obtain those images as a result of another image he queried.

However, there are some conditions that must be mentioned beforehand. At start, little to no data exists in the system which should not be an issue to the system and in worst case scenario, affect initial responses under the premise of further improvement. Furthermore, we need to ensure that our clustering algorithm is capable of handling a large, continuous sequence of data and still offer improvement in an evolutionary paradigm.

A good approach for this system would be to use the clustering methods provided by unsupervised learning as they naturally offer content grouping by comparison.

2.2.2 Clustering Algorithm

As mentioned in section 2.1.1, clustering algorithms can be classified as unsupervised learners. Clustering acts as a classification over data with the main purpose of agglomerating, thus allowing further learning from the resultant clusters [81]. Its agglomeration then returns clusters with the premise of holding content that must be similar as much as possible [81]. The opposite also applies: content that does not belong in the same cluster must be as different as possible [81].

Clustering requires both an algorithm specification and a similarity function. The similarity function is of big importance, since it is the responsible for the data agglomeration and distinction between each cluster. These functions can be divided by its comparative primitive: distance or similarity.

Regarding the algorithm specification, several have emerged each with its own objectives and properties being the most popular the K-means which belongs to the partitional clustering group. This group has the sole objective of partitioning data into K clusters, which is usually assumed and not predicted. These clusters represent a disjoint set and can be graphically identified as spherical.

K-Means [36]: Its main objective is, given a K value which represents the number of clusters to be created, to generate clusters with the average proximity between each respective point as high as possible. In other words, K points are chosen as the centroids of the clusters, then a non assigned point is assigned to its closer centroid, thus updating it to the mean distance between its points. After this update, this process is repeated until all points are assigned to a cluster.

This algorithm became very popular due to its utility, simplicity and performance. It offers a good overall behaviour and its usually enough to fulfill the desired tasks, however it can be naive, thus bringing several limitations. These limitations include its difficulty in detecting and protecting from noisy data, its dependency in the initial assignment of the centroids [23] and variance in its final result, since this algorithm depends on the initial assignments of the centroids which is randomly defined.

In an attempt to solve some of these limitations, alternatives to the classical k-means have appeared.

A strong alternative to this algorithm is the model-based algorithm SOM. Model-based clustering focuses on selecting a model for each cluster and measure its best fitting, keeping the best one [81].

Self-Organizing Maps(SOM) [42]: Self-Organizing maps are an artificial neural network which uses competitive learning to assign and adjust the weight to its output neurons. The idea behind SOM is to find and build a mapping with reduced dimensions from the input data [81] making them particularly useful in data visualization as they can reduce the dimensions of data while keeping their topological structure, usually generating a uni or bi-dimensional map however, as mentioned, this learning algorithm can also be used as model-based clustering algorithm as this mapping also groups similar data together [23].

Another strong candidate for our problem is the popular Density-based algorithm, DBSCAN.

Density-Based algorithms try to formalize our perception of a cluster and what we trivially identify as noisy data [28], generating clusters by evaluating the connection between several points [33] and density between each.

DBSCAN [28]: This algorithm tries to automatically separate the data by analysing which points are close enough to be considered in the same cluster. This procedure allows the cluster to grow in any direction as long as the density allows to, thus allowing the existence of clusters with arbitrary shape and an innate capability of detecting noisy data[23].

Unlike k-means, this algorithm does not require beforehand the number of clusters we wish to separate the data into. However, this algorithm does require two parameters: Eps and MinPts[28]. To better explain these parameters let us assume two points p and q. Eps denotes the max distance between p and q which we can consider in the same cluster, if two points have a distance equal or lower to this value, we can consider them density-reachable. It is important to mention that even if p and q have a higher distance than eps, they can still be in the same cluster as long as there is a point s which is density-reachable by both. MinPts denotes the minimum points required to be in the Eps radius of point p in order to be considered core point.

Luckily both parameters offer methods of achieving an optimal value beforehand by using trivial operations. For MinPts it is set by default to 4 as for the Eps it is advised to generate a set of values for each point with its distance to the k-th nearest neighbor, being k the value chosen as MinPts, then sort this set in descending order and generate a graph. The optimal result is the first point inside the first "valley" [28].

This algorithm can be a great choice, however, due to its heavy computational requirements, it can be very time consuming for larger data sets [23].

Conclusively, algorithms requiring a pre-defined number of clusters might become troublesome, thus reducing our choices to both SOM and DBSCAN.

However, these algorithms are not tailored for a data stream environment, thus making the application of these algorithms to our problem infeasible. Luckily, some clustering algorithms were built in order to allow clustering in such a restrictive environment.

2.2.3 Clustering in Data Streams

In order to ensure the clustering properties in this environment we need to use algorithms tailored for data-streams.

These algorithms offer new features to the basic clustering, ensuring efficiency and correctness in a model where the input is a continuous flow of data of data[13, 71]. These features include time window, and learning approach[57]. During our research, it seemed these terms did not have an agreement on the names[57, 71], therefore we will refer to them as mentioned earlier. Following we will describe each feature plus the differences between each strategy regarding them.

Regarding the time window, there are four/five different window models. The two most common are Landmark and Sliding-window[57, 71]. The former selects the whole data received giving it as input to the model at the time of training. This data is separated and stored as a landmark A landmark summarizes all data received within a certain period (which can be either time or a specific entry number). Using this approach no data is discarded and all is equally important, making old data no different from recent data. This approach can be naive when applied to a data stream, since data streams can suffer concept drifts. When this is true, some strategies can be applied, e.g, using a small window size to store the landmarks [57, 71].

The latter, when processing data uses a window of size N with a FIFO structure meaning recent data is considered the first element in this window. N represents the most recent data since the last model update, causing data falling out of this window to be considered, outdated and usually discarded. This makes the window size very important. Smaller window sizes requires less computational resources but may not contain enough data or may not contain enough data to properly simulate the universe, leading to over fitted models and returning worst results [57, 71]. Bigger window sizes may require more computational resources but ensure better models. However, if the size is too large there are cases where the accuracy of the model decreases. There are some proposals that offer dynamic size to these windows thus allowing a better adaptation to the system.

Regarding the learning approach, it can be either Incremental or Two-phase [57]. Incremental approaches update every time new data arrives, thus incrementally updating its model. Two-phase approaches separate the learning process into two parts. The first part, receives the data submitted and abstracts it, adding the result into memory. The second is triggered when a user requests the system, and will update the model with the content that was saved in memory [57].

Algorithms with state-of-the-art implementations are listed in Table 2.1. A summary about each, excepting UbiSOM [67] can also be found in [33, 57, 71].

Table 2.1: Algorithms grouped by clustering method adapted from [33, 71, 57].

Algorithm	Method	Clustering Algorithm
[7] Clustream	Hierarchical	K-means
[43] ClusTree	Hierarchical	K-means/DBSCAN
[17] Den-Stream	Density	DBSCAN
[21] D-Stream	Grid	-
[67] UbiSOM	Model	SOM

Looking at our problem, being infeasible to predict the number of clusters existing in our ever increasing data set, algorithms requiring this parameter might present some issues, thus making K-means derivatives a non optimal choice. However several algorithms can still fit our need and cluster shapes are not known beforehand, hence requiring a more experimental comparison between each solution.

After some research, we discovered two libraries offering state-of-the-art implementations of these algorithms and others thus enabling a more easy testing between each algorithm:

- **Massive Online Analysis(MOA) [14]** is an open source framework for data stream mining[18]. It offers a GUI, a library and Maven Dependencies with methods of classification and clustering in data streams. These methods include implementations of the CluStream [7], ClusTree [43], Den-Stream [17] and D-Stream [21] algorithms [14, 18].

- **multiSOM [52]** is a Java implemented interface, which uses the UbiSOM algorithm to model the iris dataset which is interpreted as a data stream. This project allows to change our dataset, thus allowing to visualize and export other models whilst offering the whole implementation of UbiSOM [67].

However, in order to decide the best cluster, it might be interesting to use comparable metrics capable of scoring each cluster without the need of a constant human interaction and analysis.

2.2.4 Cluster Validation Metrics

Since our data does not initially have neither any data set nor user submitted labels, we cannot apply a generic classification evaluation, i.e, we cannot measure the error in the predictions but instead we need to define a metric that allows to define a cluster as "good". This raises a problem regarding the evaluation and comparison of each algorithm. After some research [44] we found two metrics: cluster cohesion which measures similarity of all points inside each cluster and separation which measures the similarity, or difference, between different clusters. It was also studied three metrics that we aim to use in our unsupervised validation: Silhouette [60] score measures consistency inside each

cluster and distance between each, the Davies-Bouldin index [22] measures how well is the cluster by analysing its data and the Calinski-Harabasz index [16] measures the disparity between each cluster. Using UbiSOM [67] we are also capable, with the aid of MultiSOM [52] software, of visualizing this data and its respective clustering, allowing us to also evaluate the impact of the selected features. However, these metrics evaluate the clusters itself, being useful for early detection of data separation and perhaps relevant features however, to better evaluate image similarity these metrics will not be enough requiring other solutions.

2.2.5 Content-Based Indexing

Regarding the indexing of the results, as mentioned earlier, some challenges need to be faced.

Starting by the location of the indexing system, it will be located at the edge, thus providing some interesting properties for our solution:

- **Local Exclusivity:** Providing the index at the edge will naturally generate indexes for the content shared between the users in the same location as the edge, thus providing to them exclusively those same indexes.
- **Reduced Latency:** As this computation is closer to the user, the requests will require less hops reducing their travel time.

However, this also bring some challenges:

- **Limited Resources:** Servers at the edge do no have as much power as cloud servers, which may impact the performance of the algorithms, thus affecting overall performance of the system.
- **Standardization of the Learners:** Having each edge a instance of the learner and due to the local exclusivity property of the edge, each learner will have their own clusters and labels, if the overall system desires to have this values stored in a centralized cloud, it will require to merge all these values in order to do so.

Splitting up the query possibilities, we must consider two types from which a user can make, being them semantic-based and content-based.

Regarding content based indexing we will need to use our unsupervised learner since it is the responsible for clustering similar content, in other words, we need to use the learner to properly predict the cluster belonging to the submitted image and return to the user the data inside it.

Regarding semantic, the user must be capable of requesting content by asking for certain index specific labels. This indexes must be generated dynamically and automatically however, the labels representing the index must be human-readable and properly represent the content inside the respective cluster which may not be a trivial task.

To solve this problem, we can implement in our learner a "analyser" which under some circumstances assigns a label to a cluster automatically.

For that we could implement heuristics from which this "analyser" would be triggered. To simplify our solution we will use a simple heuristic which evaluates whether or not a cluster reaches a certain size.

Once triggered, this "analyser" could apply the automatic tag by simply assigning to the cluster an automatically generated label. However, when assigning the label to the cluster it may be also interesting to assign the same value to each content inside of it. This label could be a unique random value however, it would not be human-readable and wouldn't represent semantically the content. Alternatively, we could analyse the metadata or the content between each image inside the cluster and assign a label depending those values. This method allows the assigning of labels that are common between similar images, thus being a more user-friendly alternative and making it a good and simple methods to solve this problem.

2.3 Machine Learning At The Edge

In this section we overview works that share goals or approaches close to what we pretend to develop in this thesis.

We characterized the studied systems according to the following attributes: application of the system, i.e, the problem it is trying to solve, their edge nodes, architecture, the learner used and how they handle the training of the model.

Following we will discuss each of these attributes. To summarize this section we also built the Table 2.2.

2.3.1 Application

Regarding the applications of the systems, we found the following:

- **Data analysis for anomaly detection:** J. Schneibl et al [65], HiCH [10] for Arrhythmia detection and DeepIns [49] for defect detection in manufactured items.
- **Computation Reuse:** FoggyCache [34].
- **Video analytics:** DeepDecision [59].
- **Image recognition:** Edge Learning [30], DeepCham [46], Cachier [27] and Precog [26].

From the researched systems, the closest to our problem are the ones solving image recognition problems however, these are still not similar to the problem we are aiming to solve.

Regarding Edge Learning [30] implementation, it focused on image analytics, implementing a system for the sole purpose of making the recognition of the face and evaluating its results.

DeepCham [46] focuses on object recognition, trying to identify a generic image.

Cachier [27] is really close to our solution as it uses machine learning for caching, allowing a faster content retrieval. However, it is used for a classification problem, i.e, a correct classification of this algorithm works as a cache hit. This does not meet our requirements as it neither measures content popularity, as it only uses the data to improve its results, nor generate automatically indexes.

Precog [26] works as Cachier [27] and also does not meet our requirements for the same motive, however this system offers an interesting feature as it tries to use a similarity approach for pre-fetching.

In conclusion, it was not discovered any application trying to solve the same problem as us, however, both Precog [27] and Precog [27] offer interesting solutions as they are similar in some steps.

2.3.2 Architecture

Regarding the system architecture, we found several architectures being the mostly used by these systems the edge server-cloud architecture [30, 65, 10, 27, 26, 49] which allowed an efficient distribution between tasks, assigning the most expensive ones to the cloud server. This is an interesting synergy because it works well with machine learning algorithms as, to train models, they require more powerful computers, thus making any training step done at the cloud as no edge server is capable of training a whole model by himself whilst ensuring a low latency to the user. In the machine learning end, this architecture also opens doors to a new powerful deep learning technique named Federated Learning, which allows the training of the model at the device level, allowing a more collaborative model training, G. Zhu et al [83] explains with more detail this learning technique and how it can improve machine learning at the edge.

DeepDecision [59] used edge devices with cloud architecture, thus allowing a faster response and task simplification by using the end device itself to do several simpler tasks. Although at first glance this architecture looks similar to the cloud computing paradigm, it is important to highlight that in this architecture, the end user is also part of the architecture as the mobile device used by him is also responsible for some tasks.

FoggyCache [34] and DeepCham [46] used edge nodes with mobile devices architecture which allowed a more local evolution of the system that might never affect another edge as it can represent local exclusive properties.

2.3.3 Feature Extraction

Since the feature extraction technique relies on the data used as input, we will focus this analysis on systems also extracting image features, namely [26, 27, 30, 46].

From these systems, Edge Learning [30] and DeepCham [46] used pre-trained layers for feature extraction, thus removing them from our comparison as these extractors would required pre-trained-models. Alternatively, Cachier [27] and Precog [26] used the ORB for feature extraction, which is an available image descriptor for our solution. However it is important to note that these systems are tailored for image recognition, and not specifically faces, raising the question to whether or not it may be the most efficient descriptor for faces exclusively.

2.3.4 Learner

We noticed a trend in the usage of the learner however, this can be biased as most problems requirements of these systems was to classify content. Even so, the trend shown that most of the learners were using deep learning [30, 46, 49, 59, 65], more specifically Convolutional Neural Networks, which reflects the popularity growth of this technique.

However HiCH [10] used Support Vector Machines instead and Precog [26] used Local-Sensitive Hashing Classifiers and both presented overall good results in their solutions.

Overall, this research has shown an absence in unsupervised clustering techniques, thus reducing our awareness to possible challenges this learner may raise during the implementation.

2.3.5 Model Training

Another interesting aspect to research is how they addressed the model training task as it represents one of the most difficult challenges to address when combining Machine Learning with the Edge Computing paradigm.

The systems presented several proposals and as such we will list each excluding beforehand systems using solely pre-trained models, namelly FoggyCache [34], as it removes the desired complexity.

Training at the cloud

During our research, we found three systems training the model at the cloud, namely, HiCH [10], Edge Learning [30] and Precog [26]. This solution can be simple to implement however to attend to operations using the models trained, the request must be sent to the cloud unless the cloud transfer its models to the edge.

An example of this model transfer is Precog [26] as it initially trains the model at the cloud but also implements, using the LSH Classifier properties, a method to transfer this training between the edge server and mobile device and another between the edge server and cloud. This allows to do the recognition at the mobile as Precog prefetches the training components based on recent requests from this device and sends it back.

Although Edge Learning [30] isolates the model Training at the cloud, it also uses machine learning at the edge for lightweight operations, more specifically for the extraction of the features. This architecture as mentioned at Section 2.1 has several upsides.

Training at the Edge

During our research, for systems including learning at the edge, we found [65] and DeepCham [46].

J. Schneibl [65] implemented the above mentioned (Subsection 2.3.2) federated learning technique, allowing a collaborative learning at the edge with agglomerate learning at the cloud. As soon as the agglomerate learning task finishes, the edge server models are updated with the result of this task, thus allowing the edge learner to update as well. However this may remove any local exclusive feature as, during merge, each node model is assigned with a weight, which can be different between each or a simple average.

Alternatively, DeepCham [46] uses both a pre-trained deep learning model and a shallow learning model to, using a method called Late-Fusion, generate the recognition result. To create the shallow model, the system has a training task done at the edge named Adaptation training, consisting in, after initiation from a device, assigning a node as master and other devices as workers. Then using the workers, it generates training instances that are used to create this model.

Both DeepIns [49] and DeepDecision [59] use Convolutional Neural Network Strategies to split the learning task through the system, therefore we won't mention in detail their strategies however, a short description of each is mentioned in Table 2.2.

2.4 Summary

This chapter has presented a survey related to state-of-the-art implementations and systems to our solution as they may aid the implementation of our solution. In order to better understand each problem and solutions we introduced concepts such as, Machine Learning and some respective areas and Clustering.

We ended this chapter by researching systems implementing the edge paradigm whilst using machine learning and concluded that, for feature extraction, using local descriptors like ORB [61] might present interesting results. Regarding the learner, we didn't find any system using similar approaches, thus making this area unexplored.

Before presenting our solution, in the next chapter we will present two systems that will serve as a base for the implementation of our solution, *Thyme* [69] and *Oregano* [63].

Table 2.2: Systems with machine learning at the edge. Adapted from [50, 82]

System	Application	Edge Nodes	Architecture	Learner	Model Training
Edge Learning [30]	Face Recognition	Devices & Edge Learning Server	Devices, Edge Servers & Cloud	Autoencoder & Deep learning	At cloud with Feature extraction at edge
Technica* [65]	Anomaly Detection	Edge devices	Distributed set of nodes & Cloud	Autoencoders	At edge & Aggregated at cloud
HiCH [10]	Medical data analysis for arrhythmia detection	Layered Fog Network	Centralized cloud & distributed fog nodes	Support Vector Machine	At cloud
FoggyCache [34]	Cross-device approximate computation reuse system	Edge Server	Devices & Edge Servers	Adaptive Locality Sensitive Hashing Classifier	Pre-Trained Models
DeepCham [46]	Object Recognition	Edge Server & Mobile Devices	Edge Server & Mobile Devices	Convolutional Neural Networks	Adaptation Training at edge & Pre-Trained Deep Model
Cachier [27]	Image Recognition	Edge Server	Edge Server & Cloud	Not Mentioned	Not Mentioned
Precog [26]	Image Recognition & Prefetching	Edge Server	Device & Edge Server & Cloud	LSH Classifier	Initially trained at cloud & sent, using the edge, to device as prefetch response
DeepIns [49]	Manufacture Inspection	Sensors & Fog Nodes	Sensors, Fog Nodes & Central Server	Convolutional Neural Networks	Lower-Level CNN Layers at Edge & Higher-Level CNN Layers Servers
DeepDecision [59]	Video Analytics	Mobile Devices	Mobile Devices & Cloud	Convolutional Neural Networks	Small CNN at edge & Bigger CNN at back-end
Our proposal	Group and Index user-submitted content by similarity	Edge Server & Mobile Devices	Mobile Devices & Edge Servers	Unsupervised Stream Clustering (4)	Principal Component Analysis (4)

* although it is not referred directly, the paper uses this name as a reference

3

Thyme & Oregano

This chapter focuses on the building blocks of our project, Thyme, Oregano and GardenBed. These systems belong to the EdgeGarden ecosystem built under projects "Hyrax: Crowd-Sourcing mobile devices to develop edge cloud" [1] and "DeDuCe: Distributed Data-Centric Concurrency Control" [2], and are now used as building blocks for many projects and thesis. We will begin by giving an overview on the Thyme system in Section 3.1 and then, describe the system we will work on, GardenBed, in Section 3.2. Finally we will end this chapter with a description of the Oregano Framework in Section 3.3.

3.1 Thyme

Thyme [19, 70, 69] is a topic-based time aware publish/subscribe system built for mobile edge networks allowing a persistent collaborative data storage using as backbone for communication the peer-to-peer architecture. The mobile devices are clustered along regions, scattering and accessing data contained within. Every device's functionality is symmetrical, meaning that no master is needed to maintain the system and that each device can be both a publisher and a subscriber. Thyme can be divided into three layers: connectivity, network and services.

Missing bridge here

Connectivity Layer are responsible for the communication between each device and their respective edge server node by several communication protocols.

Network Layer is responsible for handling communication between the peers and maintaining the Publish/Subscribe system connected. This layer is also responsible for discovering each peer through broadcasting.

Services Layer contains three main modules: Time-Aware Publish/Subscribe, Storage, Grid Manager, whose description follows.

[1] https://hyrax.dcc.fc.up.pt
[2] https://sites.google.com/fct.unl.pt/deduce

Time-Aware Publish/Subscribe offers the Time-Aware publish/Subscribe interface to the application, allowing the specification of a time-interval for both the subscriptions and publishes, regardless its time. This layer manages both publish and subscriptions from the users plus all the notifications involving them.

Grid Manager defines and manages a geographical space named *cells*. These *cells* point to a group of physical devices (i.e. mobile devices) that are located in the same space.

Storage contains both the Publish/Subscribe infrastructure and the actual storage system. This service handles the replication within the nodes, allowing a more reliable storage, preventing data loss and reducing the load at the original publisher as data is also located at other nodes. Thyme offers two types of replication mechanisms:

Active Replication uses the virtual nodes considered in the same grid and, after a publish, somewhere in the future, all nodes will download the published data.

Passive Replication leverages on the nodes that already contain the data to provide more replicas within the network.

3.1.1 Thyme's API

Thyme offers several operations. These include the regular publish/subscribe operations: *Publish*, *Subscribe* and a their reversed operations, *Unpublish* and *Unsubscribe*.

Publish indexes the metadata associated to the object by its respective tags, hashing each and sending and sending it for storage in the resultant cell. The cells are then responsible for managing the object's metadata and evaluate if subscriptions match the publication. The content however is only stored in the publisher node.

Subscribe allows the user to subscribe to a set of tags, and define a time interval from which he wishes to keep the subscription. After subscribing, if any content matches both parameters, the user is then notified and can then download the content. The user is not limited in the interval provided, allowing him to subscribe to both past and future content.

Unsubscribe removes the subscription from the user to the provided tags.

Unpublish is an operation consisting in a complete erase of the data from the system. Since *Thyme* considers time as a first order dimension, this operation is possible as by preserving data in a Publish/Subscribe system with awareness to past content, it allows the deletion of content that is no longer shared and preventing the retrieval of content that users did not want to share anymore.

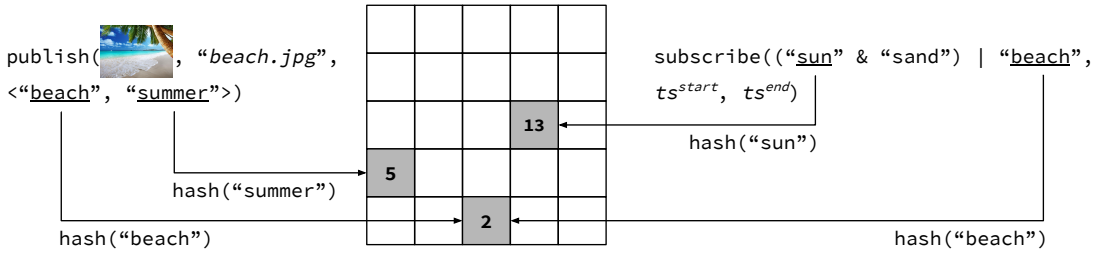

Figure 3.1: Publish and subscribe operations. The tags' hashing determines the cells responsible for managing the object metadata (cells 2 and 5) and the subscription (cells 2 and 13). If a subscription has overlapping tags with a publication (and vice versa) it will also have overlapping (responsible) cells, guaranteeing the matching and sending of notifications to the subscriber. Taken from [70, 75]

Figure 3.1 gives an overall insight of the publish and subscribe operations.

The work done by Thyme is around metadata objects under the format $\langle id_{\text{obj}}, T, s, ts^{\text{pub}}, id_{\text{owner}}, L_{\text{rep}} \rangle$. This metadata represents respectively the object id, its associated tags, summary of the object, e.g, a thumbnail of an image, publication timestamp, the owner id and a list of replicas under the format $\langle id_{\text{node}}, cell_{\text{node}} \rangle$ and are crucial to fully take advantage of the replication methods.

3.1.2 Mutable Data

Thyme supports mutable datasets for data sharing. This is done through the implementation of Conflict-Free Replicated Data Type (CRDT) [66]. CRDTs are a family of replicated data structures designed for highly available systems. CRDTs allows for its objects and replicas to be modified without the need for an expensive synchronization, while following a strong eventual consistency model, guaranteeing from its replicas the convergence of the data in an uncoordinated and failure-safe way. These assurances are done by respecting the order of execution of all operations and respecting all dependencies.

CRDTs can synchronize by two distinct methods: State or Operations. CRDTs who synchronize using state, do so by periodically sending their local values. When another state is received it is merged using a defined merging function. Although simpler to design and implement, this type of synchronization usually requires a huge ammount of requests for the propagation of the entire state. On the other hand, CRDT who use Operation-based synchronization do so by propagating the update operations. This is done using a *prepare* function responsible for generating the operation to be replicated and an *effect*, executed at the destination and responsible for aplying the update operations. Using operation-based synchronization can be more communication efficient, however it also makes stronger assumptions regarding the environment.

On Thyme, the available CRDTs offer a set of conflict-free operations with causal consistency and ensure both operation order and dependencies are respected. This offers

an easy and accessible interface for using them, requiring only to publish the changes done to the dataset and then subscribe for its updates. As updates are published by other devices, they can be downloaded and be used to update locally.

3.1.3 Sharing the Index on Thyme

Due to the unpredictability of data, and the possible alteration of its concepts in a continuous nature, our indices need to be dynamic and allow us to both add and remove them with relative easy and without affecting the quality of the solution. However, these must also be as descriptive and consistent as possible, since our queries rely strongly on them. Indices must also be accessible to everyone, let it be a mobile device being used by a user, or an edge server who needs to evaluate if an index is adequate for an arbitrary content. Furthermore, these indices must be mutable, and modified by at least one end whilst being consistent for its readers. This requirement rapidly extends to various ends as we tailor our solution for *GardenBed* [75], more details about this system will be provided in the next section.

Given the aforementioned requirements, our solution must strongly depend on *Thyme* interface for the CRDT, and generate a where we will store our indices. These indices must then be subscribed by each mobile device so they can stay updated.

3.2 GardenBed

GardenBed [68] is a distributed system composed by a set of stationary nodes located at the edge and a set of devices, capable of storing and sharing content between themselves using device-to-device communications. These sets are named regions and are unique, meaning that they do not contain neither a repeating station node nor a repeating mobile device from other regions.

The edge servers form a distributed system located at the edge and allow a cross-region topic-based publish/subscribe abstraction. In turn, the mobile devices run a system that allows for content sharing and storage between the devices contained in a region. In our setting, mobile devices run both Thyme (Section 3.1) and Oregano (Section 3.3). Overall, the system offers a cohesive and consistent storage network between multiple regions under a Publish/Subscribe paradigm, allowing users to publish content and provide a set of tags and a description of the content, and subscribe to its interests based on the same properties.

Like *Thyme*, GardenBed offers a similar interface to a regular publish/subscribe system to interact with the system, namely a *Publish*, *Subscribe*, *Unsubscribe*, *Unpublish* and *Download* operations.

Mobile devices can also communicate with outer-region edge servers however, they must do it so indirectly, communicating first with the local edge server. This happens because only the edge servers are linked to each other (more details in Section 3.2.1).

Initially, the data published by a mobile device remains on the device itself, being only available to the other devices in the region however, as data popularity increases, the region's edge server begins to collect and cache the published content so it can be accessed globally. This brings two upsides:

1. Since, as mentioned earlier, the edge servers are linked to each other, it allows the sharing of the cached content between other regions.

2. Reduces loading between mobile devices as edge server can now also attend to subscription requests.

3.2.1 Edge Servers

The deployment of an edge-server requires the implementation of several modules. The *Cell-Head Lookup* algorithm determines which region should be contacted when operating over a given data item. The *Matching Logic algorithm* which matches subscriptions with its respective publishes. The Ranking Algorithm which determines the data items that must be uploaded to the edge. The Notification Priority Policy algorithm decides who must send a notification, being either the server, the mobile nodes or both (in case the data is cached on the edge).

Communication between the server and its mobile devices is bidirectional. If the communication is from the client to the server, then the later is informed of the operations performed within the region, and can enable, if requested the download of data only available in other regions. The local information is sent by the cluster node at cluster level and contains the subscriptions of the cluster nodes, the number of "unpublish" operations and statistical data regarding the number of downloads done for each data item, allowing the ranking of the items based on their popularity.

If the communication is from the server to the client, then it can be because of four different reasons:

- notify the clients about new data published in other regions.

- notify the clients of changes in the server so they may update the metadata of a data item so it complies with those changes.

- download a data item from other region and cache it or, in the opposite, attend a download operation from a client from other region

- to "unpublish" a data item, triggering its removal from the system

Being the edge nodes served mainly as a cache, it is important to upload only content that are considered popular, for that an asynchronous process, using the Ranking algorithm gathers the most popular items and caches it in the Local Popularity Cache.

Our system could be implemented on top of this infrastructure, beginning its process as soon as this task retrieves the data.

3.2.2 Mobile Devices

As mentioned, mobile devices are clustered by regions, operating as a distributed, region-specific storage. Using this architect tailored for *Thyme* and its intrinsic operations we can implement a photo sharing application. When an image is published it can be done using the publish method. Following the image, we can also set descriptive tags allowing other users to fetch it via subscription. As mentioned earlier, despite a regular Publish/Subscribe nature, *Thyme* allows content to be erased, allowing us to delete images as well.

Furthermore, we can use *GardenBed* to upgrade this storage into a multi-region storage with content indexing, allowing us to extend the indexing process using Machine-Learning's clustering. This can be done by processing the images downloaded (using the aforementioned algorithm done by *GardenBed*) and grouping the content based on its similarity using feature extraction techniques. Afterwards, as a group starts to become relevant we index it so it can be accessed. However, this proposal is not strong enough as it does not offer a content-based image retrieval for the devices. To solve this issue, we use *Oregano*, which can be run as well in mobile devices and allows the execution of computations over the data in a distributed manner.

3.3 Oregano

Oregano [62, 63] is a mobile executed framework for distributed computation, capable of processing data without the need for Internet services and has the main goal of transferring computation to where the data is located. Oregano uses groups data stored in Thyme and does the necessary computations over the published data on the network. Oregano provides two main functionalities, stored data computation and real-time data streams computation.

Figure 3.2 illustrates the system architecture. Network and Link layers handle the communication between the devices. Persistent Publish/Subscribe refers to Thyme which was already mentioned earlier.

3.3.1 Computation

This layer represents the Oregano framework and is responsible for the execution of distributed data computation. As mentioned earlier, it focuses on moving the computation to where the data is located. To do that, it uses the network layer to send and receive system specific messages. It interacts with Thyme, making use of its properties, namely replication and churn, to handle and optimize user operations allowing, for instance, balancing of the computation between the devices, avoiding overloading. For that, Oregano requires the metadata stored in Thyme as it is the main data used for the whole process.

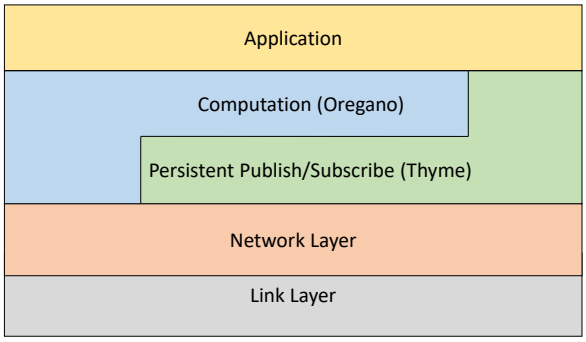

Figure 3.2: Oregano architecture. Taken from [62]

3.3.2 Application

This layer represents the applications using the Oregano framework. Oregano offers an API, allowing the interaction and use of its services and the ones provided by Thyme by any application.

A device using an application using the Oregano framework can play one or more of the following roles: client, scheduler or computing. These roles can be interpreted as three important components of the computation layer, being crucial for the proper termination of a computation request. This means that assigning more than one role to a mobile device, will assign to it multiple operations which can triggered simultaneously.

Following we will briefly describe each role.

Client is responsible for the management of the subscriptions with computations including messages associated with each subscription, e.g, failures. Since this operation adds a new computing step after the subscription, in order to store both data, the client does two subscriptions: one with the tag specified, pointing to the desired content and another with the resulting tag, pointing to the content after the computation. This role is also responsible for controlling the processing of the results, allowing users to stop or continue a computation request. While a user can stop the process at any time, the same does not apply for the continue operation and instead it is only available under some conditions dependent on the amount of data processed and the amount left including new publishes. Finally, the user is also responsible for any publish with pre-processing.

Scheduler is responsible for the management of all subscriptions with computation it receives, scheduling tasks with the devices owning the desired data. This role is more selective, assigning only to devices positioned in the cell from where the requested tag maps to, thus having the complete set of metadata necessary. Then, for every matching publish, it will distribute the computation between the available computation devices and, using a mechanism of heartbeat, keep track of the status of the computation. If for

```
public class Service1 extends Service<Input, Output, ServiceArgs> {
    protected Class<Input> getInputClass()
    protected Class<ServiceArgs> getArgsClass()
    public IMDD<Output> process(IMDDStream<Input> inputStream,
                                List<ServiceArgs> args)
    protected DataItem outputToDescription(Output output)
}
```

Listing 1: Class extending Service and required implementations

some reason a computation node fails to compute the data, the scheduler is responsible for rescheduling the data that could not be computed.

Computing is responsible for doing the computation over the objects locally stored in Thyme. For that, it provides the data to be processed plus the service arguments provided by the scheduler to the Service deciding, after completing the computation, to publish the results, assigning a result tag or notify the requesting Client device directly, informing the location of the data so the user can download it.

3.3.3 Computation Services

As mentioned in the system overview, we can use *Oregano* to handle several operations regarding our solution. These can be done by establishing computation servers, and the service it offers for computation. To establish a computation Service we can extend one of the two Service classes offered by *Oregano*: **Service** and **ServiceWithPP**.

As the name suggests, the former is used for services that convert input data directly into an output. Its extension requires the definition of three generic data types, namely its input, its output and the arguments used by the service. It also requires the implementation of four methods, one to retrieve the Class of the input object, one for the arguments of the service, one to describe the output and the other for the computation to be done. The latter implementation receives both the arguments of the service and the input data as a Mobile Dynamic Data Set and must result in a similar dataset containing the output Class (an example of its signatures can be viewed at 1).

The latter is used for a service that does require the processing of the input before executing the implemented computation. To extend this class, it is required to follow the suggestions mentioned earlier with only some additions and alterations. It is also required to define the Class of the input after being processed. The method referring to the input class now refers to the input after being processed while it exists a new method to get the original data type. We also need to implement the computation to do before the main computation can be done, it receives the original input and the arguments of the service and must return an object of the type declared as the input after being processed (an example of the signature can be viewed at 2).

```
public class Service2 extends ServiceWithPP<Input, ProcessedInput,
                                            Output, ServiceArgs> {
    // ... Similar methods as the Service1 ...
    protected Class<ProcessedInput> getInputClass() // Instead of Class<Input>
    protected Class<Input> getOriginalInputClass()
    public ProcessedInput preProcess(Input input, List<ServiceArgs> args)
}
```

Listing 2: Class extending a Service with pre-processing and required implementations

After defined the computation services, it is still required to define the computation Servers.

3.3.4 Computation Servers

To attend requests with computation using *Oregano* it is required to establish Computation Servers. This is done by extending the **ComputationInfrastructureServer** and assigning the service to run. After these implementations, it is only required to request for these services by doing a publish (or subscribe) with computation operation, and *Oregano* handles the rest. In this thesis we propose two computation server and processes: One server for the upload of a photo and another for a content-based image retrieval.

3.3.5 Photo Sharing

Photo sharing applications for mobile are a pratical solution for image visualization, sharing and processing. Furthermore, mobiles are evolving rapidly both in hardware and software, allowing a pratical use of several features, namelly image processing and adding computation possibilities to the region nodes. Local computation reduces the workload for the stationary server, thus increasing its availability for the core operations.

Oregano allows us to make use of theese available features in our solution using the aforementioned pratices. We can do so, by establishing computation servers that process the image and publish its result so it can be accessed later on. This collaborative behaviour also provides access to computations that older devices would have more dificulty in running due to its limitations. Establishing this servers would then allow each device to process the image (let it be locally or using *Oregano*) and search the index using its result. More details in the next Chapter.

4

Chives

In this chapter we apply every knowledge obtained in the earliest chapters to deeply design and implement our system proposal. We begin by presenting an overview of the architecture design for our solution in Section 4.1, describing the proposed architecture and behaviour for each main role, including its contributions and interactions with the system. Furthermore, we present our Framework *Chives* in section 4.2, followed by a more detailed description of its requirements and design. We begin by describing the image processing in section 4.2.1, followed by our proposed implementation in section 4.2.2. Thereupon, we describe the clustering algorithm interface and its tasks in section 4.2.3, followed by our implementation in section 4.2.4 and our integration with framework specific properties (4.2.5). After describing how to extend our framework, we portray its design and how it manages its indices in section 4.2.6, followed by a description of our solution for the dynamic generation of indices in section 4.2.6.1 and a description of a component aiding this process, the overseer (section 4.2.6.2). Lastly we describe our designed workflows in section 4.3, followed by the designed integration between *Chives* and *Oregano* regarding the aforementioned workflows in 4.4 and end this chapter with some final remarks in section 4.5.

4.1 Architecture

Chives is a framework designed to generate content-based indices dynamically and automatically. It provides the means to: a) build an index of any type of data and b) query these indices by providing an exemplar set of the data, returning content considered as similar. As mentioned in Chapter 1, in this thesis we focus on the indexing of photographs, based on the faces they contain. Accordingly, although Chives was built with a level of abstraction capable of handling arbitrary content (more details in Section 4.2.6), we will center its description on the photo indexing scenario.

Chives comprises two main components (an overview is depicted in Figure 4.1):

1. a logical region, containing a set of devices (potentially mobile) connected to each other through an access point, and

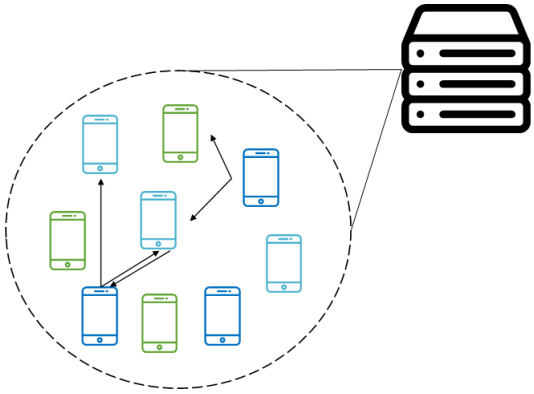

Figure 4.1: System overview from a single mobile cluster.[1]

2. a stationary server located at the access point or in close proximity to it.

These components will have *Chives* built inside *Oregano*, who is built on top of *GardenBed*. These components are tailored for the mobile edge, bringing the data storage and processing closer to the devices. This reduces the navigation of both requests and data during its operations, which reduces latency and operation time.

4.1.1 Mobile Devices

The mobile devices are the main contributors for the system's, as they are the actors publishing the content. As mentioned in sub-section 3.2.2, each device is connected to both the other devices and to the *GardenBed* server via an access point. The user interacts with the device using a mobile application that makes use of *Thyme* and *Oregano* (Software architecture can be seen at figure 4.2). The use of *Thyme* allows a Publish/Subscribe abstraction and a data storage to publish and subscribe to data indexed by the tags. *Oregano* extends this concept by adding allowing queries (on the data) that may require computation, such as content-based search. The position of Chives in this software stack is to be an integrated component that can be used by *Oregano*, allowing the latter to access the indices generated by the former and verify if the content being used for searching is indexed.

As a result, this system will offer to the device both a publish and a subscribe method, and its respective computations. *Oregano* can use *Chives* to add the extraction of its features as a pre-processing computation. When subscribing to content, we add a query allowing to do a content-based search of the content. To do so, the user can provide an image which will be used in the query to retrieve the tags of similar content. This will be processed by *Chives*, who will check for similarities between the image and its stored samples, returning its deduction.

[1] This and Future Images may contain icons made by prettycons and Freepik, available at flaticon.

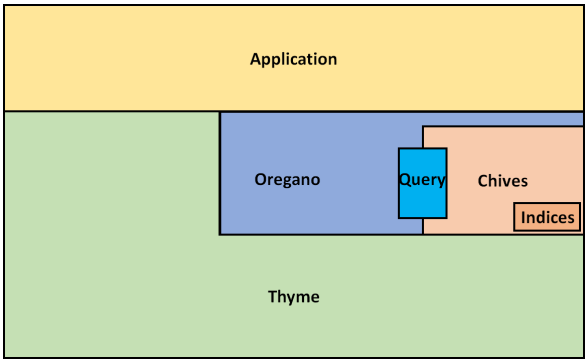

Figure 4.2: Overview of the software used by the mobile user.

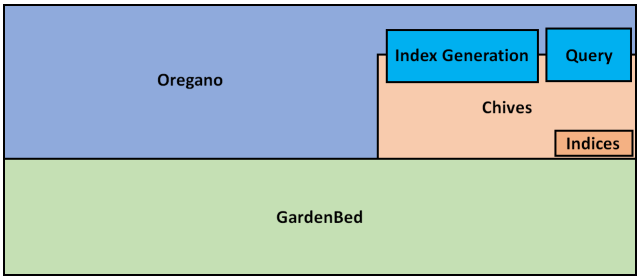

Figure 4.3: Overview of the software used by the stationary server.

4.1.2 Stationary Server

The stationary server is responsible for downloading the content shared between the users, allowing him to store, cluster and index the content. It is built over *GardenBed* as presented in Section 3.2. *GardenBed* periodically inspects the region it manages to download the most popular items. In order to provide for content-based queries on the region, *Chives* will complement the functionalities of *GardenBed* with the ability of processing all incoming data to build a region-wide index. This is done using *Oregano* who will allow the processing of the data through the use of its services (*Oregano* services are described in section 3.3, design of the software architecture can be seen in figure 4.3).

Being the server our centralized infrastructure, it will also have available other essential services, i.e. the index query. This is done to ensure that it exists at least one computation server attending these services. These servers are crucial for the system, as they allow mobile devices with less resources to still operate in the system. During the elaboration of this thesis, we focused on two main services, the index generation (described in section 4.4.1) and the index query (described in section 4.4.2).

As mentioned earlier, this is done by using *Chives* inside the *Oregano* services, allowing them to process the images, cluster its features and get the generated indices. For the server, *Chives* also includes an indexer module, responsible for the generation of indices so it can disseminate them to the devices.

This module is composed by three main components: the clustering algorithm, the

index generator and the overseer. The clustering algorithm will save and cluster each projection, grouping projections regarding to their proximity. The index generator will receive clusters who meet the indexing conditions and generate an index. It is also responsible for updating them. The overseer will try to update the indexer and look for clusters who can be indexed and provide them to the index generator. More details regarding each component will be mentioned in the next sections, alongside the design of the architecture 4.4. In conclusion, both the server and the mobile devices use *Chives*, however only the server requires the indexer.

4.2 Chives API

Chives was built with abstraction and extension in mind, enabling further studies over the already implemented process and even the addition of new ones. Our Framework allows to store, extract and cluster an image, by implementing specific interfaces aimed to follow the aforementioned workflow. By implementing the interfaces designed, *Chives* is then responsible for clustering the images, evaluate its clusters, index the clusters meeting certain conditions and propagating any changes to each mobile device.

Chives is also aimed to be part of Oregano's services, however, to best assert over the quality of our solution without any specific integration, it was also built under a "testing" environment.

For both cases, Chives can be instantiated and provided with the following parameters:

An indexing condition: This allows us to evaluate a cluster's projections and index it if such condition is true (Default: Cluster has more than 10 projections).

Image Batch Size: Number of images inserted until the clustering algorithm re-evaluates the mapped clusters (Default: 100 insertions).

Feature Extractor: A feature extractor interface for Chives to operate over.

Cluster DataType: A comparablle and descriptive cluster datatype.

Clustering Algorithm: The algorithm performing the clustering.

In the following sections we will better describe these requirements and their underlying processes, followed by some implementations.

4.2.1 Feature Extractor

The *Feature Extractor* is an abstraction of the task responsible for processing the images being inserted. Being our thesis focused on facial features, the extractor extends the process by running first a face detection technique, followed by a feature extraction of each detected face. This allows the extractor to focus solely on the facial features of an

```
public abstract class FeatureExtractor {
    // Face detection Algorithm
    public abstract List<Image> detectFaces(Image img);
    // Feature Extraction Algorithm
    public abstract double[] extractFeatures(Image img);

    ...Pre-Defined Feature Extraction methods
}
```

Listing 3: Feature Extractor Interface

```
public interface IProjection {
  // Returns a projection features into a set of coordinates
  double[] asDoubleArray()
  // Gets a specific feature giving the dimension value
  double get(int index)
  // Obtain the number of dimensions of this projection
  int getDimensionSize()
}
```

Listing 4: Projection Interface

image, and generate more focused features, however this may also add a dependency layer to the final result.

Chives offers a huge abstraction from this process, requiring from the user to implement two main operations during the implementation of an algorithm: a face detection algorithm and a feature extraction algorithm (each method signature can be seen in Listing 3). The face detection algorithm should focus on detecting faces inside the image and extract each detected face into a new, reduced, image. Since many faces can be detected, the result is expected to be a new set of images with reduced noise for the extractor. The feature extraction algorithm should receive an image and transform into a set of descriptive coordinates, this set of coordinates are crucial for the clustering, as they will evaluate the similarity between each face.

This methods are used by a pre-defined method in the same class that will process the image and create an image projection. Projections are the basis of our clustering and indexing and represent a position in a N-dimension subspace, containing methods to access its coordinates and dimension size (this interface can be seen in Listing 4).

Despite the aforementioned abstraction, due to the scope of our thesis, the *Feature Extractor* generates an implementation of the aforementioned projection that also contains a reference for its original image. This is important so each face can be referenced back to its original image.

More details regarding implementation of the feature extractor, will be described in the following section.

4.2.2 OpenCV-Based Feature Extractor

Each image can be of different kinds and contain different, possible multiple, descriptive features in it, making the extraction of its features not an easy task. Being the focus during this thesis, to generate indices according to faces contained in the image, some studies were done regarding face detection, image feature extraction algorithms and, giving the complexity of this problem, it was decided to add a step before the extraction of the facial features, regarding the detection of the face. This can cause unpredictably errors and inaccuracies when being processed by a feature extractor which may, later on, be reflected in our indexer.

Regarding feature extraction, being our main focus to provide a solution portable to both *GardenBed* and *Android* systems, we used out of the box implementations offered by *OpenCV* [1], namely pre-trained face detection models. Being *OpenCV* built to be cross platform, it offers integration with several commercial systems, namely Android, allowing us to access the same implementations and models from two distinct systems [1]. *OpenCV* provides several face detection models, as such we asserted over the best model (details can be seen in Section 5.4), choosing the one providing less noise and with higher accuracy. This is important as we prefer to ignore content than adding noise to our algorithm as this could lead to noise being clustered and generate inaccurate indices from the indexer.

An important factor to consider as well is the size of the face extracted, as each has no guarantees regarding its size and most algorithms and strategies may require them to be of equal size. As such, using *OpenCV* [1], we resize each face extracted into a pre-defined size, thus ensuring this may no longer be an issue, however resizing an image may lead to inaccuracies.

After detecting and isolating the face from a (possibly more generic) image, we can then proceed to the feature extraction algorithm. The algorithms used are implemented in the *OpenCV* [1] library. During the elaboration of this thesis we used *JavaCPP Presets* [6], to access the libraries provided by the library for Java. As mentioned in sub-section 2.1.2, to ensure the processed data can be clustered, it is required to convert the facial features detected by our computer vision algorithm into a set of features that can be provided into the clustering algorithm.

During our research, we discovered a face recognition algorithm that offered a similar approach to a regular clustering algorithm named *EigenFaces* [73]. *EigenFaces* was built to attain only the most relevant features of a face image and encode it in a way that it can, later on, be used for comparison with other, similarly encoded images. This algorithm uses *Principal Component Analysis* [80] in its process and consists on finding the *Eigen-Vectors* of the provided images. *EigenVectors* represent a specific type of vector whose changes are reflected by a scalar (the eigenvalue) when a linear transformation is applied. These vectors represent the most characteristic features of the images provided, being each face described by its respective weights.

As requisite from a classification problem, *EigenFaces* requires an initialization phase that consists in training the algorithm with an initial set of images. This initialization calculates the corresponding vectors and define an adaptable "face space"containing the images with the highest resulting eigenvalues [73]. The known images are then projected inside this space and will serve to determine future, inserted images. After the initialization is complete, this algorithm performs its recognition by calculating the set of weights for the input and projecting it into the "face space", returning either the label of the known image or a default value representing "unknown".

Our solution replicates this procedure but with a clustering system. Three models were generated to create this projections: a *Principal Component Analysis* [80] model and two *Linear Discriminant analysis* models receiving both eigenvalues and eigenvectors from two trained face recognition algorithms: *EigenFaces* [73] and *FisherFaces* [12]. These models can project the faces into a subspace, generating a set of coordinates, which allows an euclidean evaluation and comparison of the data. However, these models require training, raising an issue regarding its performance at low amounts of data as it requires a dataset to improve its results. Furthermore, this model is susceptible to bias and inaccurate projections.

Since the quality of the clusters returned by our indexer will depend highly in the projections returned by this model, it will be required an evaluation phase regarding the model, thus adding an evaluation layer. More information regarding this evaluation in Section 5.4.

4.2.3 Clustering Algorithm

Conceptually, an index should represent a group of similar images, containing little to no differences between them. Theoretically, considering a good feature extraction as a good detector of the facial features, we can consider a photo similar if its features are similar as well (or relatively close). Consequently, we can measure the difference of a face by measuring the difference of its features and group photos with a lower difference between them. If we consider each face as a point in a subspace, having each of its features projected as a coordinate, then we can measure the difference between each face by measuring the euclidean distance of its point. Furthermore, we can group similar photos by grouping their points, using clusters as a reference of a group.

The *Clustering Algorithm* is the component responsible for creating, maintaining and updating the clusters inside the indexer. It contains both a clustering algorithm and mapping between the data and the clusters. To detach the logic behind each clustering algorithm and the infrastructure operations, we propose a solution that receives a clustering algorithm implementing a specific interface and uses it for its operations.

This interface (Listing 5) contains a training, an estimate and a clustering operation. The `train` method must insert the projection inside the cluster algorithm, updating its internal clusters. These projections do not need to be saved inside the algorithm as

```
public interface ClusteringAlgorithmInterface<PROJ extends IProjection> {
    // receives a projection and inserts inside the algorithm
    void train(PROJ point)
    // receives a list of projections and inserts them inside the algorithm
    void train(List<PROJ> points)
    // Returns the cluster containing the projection
    ICluster estimate(PROJ projection)
    // Returns all clusters
    List<ICluster> getClustering()
}
```

Listing 5: Interface for the clustering algorithms

the indexer already does that. Instead the training method must focus on updating its clusters, allowing the *Indexer* to query (using the `estimate` operation) the cluster whom the projection was inserted into. The `estimate` method should query the algorithm for a representative, it receives the projection to query and returns the cluster object considered adequate. The `getClustering` method should return all clusters in the algorithm.

The `ClusteringAlgorithm` interface also requires the return of clusters implementing the `ICluster` interface (Listing 6). We require this interface so the indexer can operate over these clusters and send them to the users for more accessibility over the data. The `ICluster` interface contains methods `getId` and `setId` to, respectively, get and set an Id, `getInclusionProbability` to measure the inclusion probability of a projection, i.e. the probability of a projection being included in the cluster and `getCenterProjection()` to get a projection of the cluster's center. Since the id of a cluster is its means of identification and being difficult to distinguish clusters without this value, the indexer carries the id generated by the algorithm itself, otherwise it would be unfeasible to keep track of each cluster and its transformations.

Every insertion of the projection is reflected on the clustering algorithm, however to keep track of any changes that happened inside of it, the *Indexer* queries, in the background and periodically, for the clusters. This query allows the *Indexer* to also update its internal mapping between each projection and cluster. This internal mapping is the data being reflected to the user, meaning that data may be outdated before this update operation is done. After the update is done, the clusters that fulfill the indexing condition are marked for indexing, adding it to the indexer or updating it. Furthermore, this component also stores the points and their representative, allowing the user to retrieve cluster specific projections.

In the following section, we describe the process of extending our clustering algorithm using the *Massive Online Analysis* [14] library.

```
public interface ICluster extends Comparable<ClusterId> {
    // Gets the id of the cluster
    double getId()
    // Measures the inclusion probability of this cluster
    double getInclusionProbability(IProjection projection)
    // Gets the center of this cluster as a projection
    IProjection getCenterProjection()
}
```

Listing 6: Interface for the cluster

```
public abstract class MOAClusterer<PROJ extends IProjection>
            implements ClusteringAlgorithmInterface<PROJ> {
    // Receives a MOA specific object as the clusterer.
    // This enables re-usability for each of its algorithms
    public MOAClusterer(AbstractClusterer clusterer) { ... }
    // Interface Methods
    public void train(PROJ point) { ... }
    public void train(List<PROJ> points) { ... }
    public ICluster estimate(IProjection projection) { ... }
    public List<ICluster> getClustering() { ... }
}
```

Listing 7: MOA Clustering Algorithm

4.2.4 A MOA-Based Clustering Algorithm

During this thesis, we implemented *Chives* Cluster Algorithms for four algorithms from the *Massive Online Analysis* library: *Clustream* [7], *Clustree* [43], *Denstream* [17] and *Dstream* [21]. This was done by adding classes who convert the MOA data types to implement the required interfaces. For the algorithms, we propose a class who wraps MOA clustering algorithms and implements our `ClusteringAlgorithmInterface` (for more details regarding the class signature and some methods can presented in Listing 7). This class contains a training method consisting in the operations required for insertion of a new instance into the clustering algorithm, an estimation method which returns the cluster whom the projection should belong to (this operation does not insert the queried projection) and a method to return the existing clusters at that moment. This return is simplified into a list of clusters, allowing us to make our own estimations using their inherent `getInclusionProbability(...)` method.

Wrapping the MOA abstract cluster algorithm brings modularity to this implementation, allowing the usage of several algorithms from this library just by extending this class and provide it to the clustering algorithm component whom will operate over it. We also implemented Cluster and Projection wrappers with a similar objective as the clustering algorithm.

```
##### Indexer Generic Options #####
insertionInterval = 100
algorithm = clustream

##### Clustream Options #####
clustream.timeWindow = 1000
clustream.maxNumKernels = 100
clustream.kernelRadiFactor = 2
```

Listing 8: Properties File

4.2.5 Customizing Conditions

Since cluster algorithms have their own parameters and these usually need to be considered for a better clustering quality, we implemented a file parser who reads its properties and assigns indexer and cluster specific parameters. As a proof of concept, we use this parser to choose the clustering algorithm to be used by *Chives* as well as define its values. Although each algorithm has its own parameters, thus requiring an implementation beforehand, one parameter can be defined beforehand, the number of insertions to be done before the indexer should update its clusters. We also had a parameter for the algorithm name so we can then search the name inserted in the implemented clustering algorithms and call its dedicated parser. This parser will then parse the file for each required parameter following the syntax: <*algorithm*>.<*parameter*> (an example file of the configurations file can be seen at 8). If there is no value associated or the key is not found, a null value will be returned which, in our use case is reflected by maintaining the default value of the parameter.

After this parse is finished, two properties object are generated, the module and the algorithm. The former contains parameters for *Chives* and the later contains parameters for the clustering algorithm. These are then used to generate both the clustering algorithm and the clustering update condition and are given to the module.

4.2.6 Indexing

As mentioned in Section 4.2, *Chives* is responsible for creating, maintaining and propagating indices between all the devices, this is done periodically and under custom circumstances. The indexer is the most important component in the dynamic generation of indices. It is responsible for storing, clustering and updating the mapping between the projections. It is also responsible for updating, creating and removing indices if certain conditions are met in the cluster.

As mentioned, the indexer can only operate using projections, this done so it allows any type of projection as long as it follows the interface mentioned in listing 4.

As we can see in figure 4.4, the indexer is composed of three main components: the

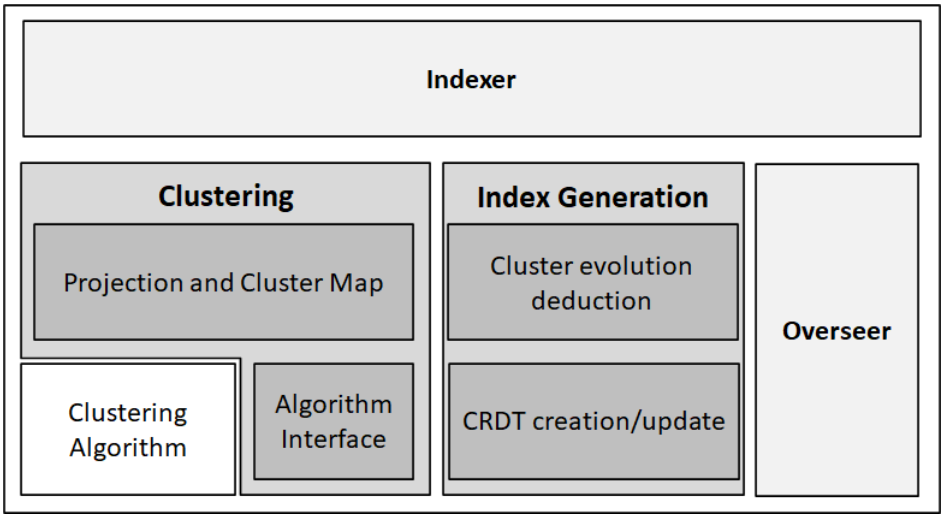

Figure 4.4: Overview of the indexer.

Clustering algorithm, the *Index Generation* and the *Overseer*. In section 4.2.3 we described the first component, in the following sections we will describe the remaining components.

4.2.6.1 Index Generation

The indexer is responsible for keeping track of the clusters to be indexed and removing the ones who should not be. The Index Generation component is triggered by the *Overseer* who gives the clusters to be indexed and then tries to either update the cluster's indices or, if the cluster does not exist in the map, generate one.

An index contains a collection of tagged clusters. These tagged clusters contain the cluster as provided by the clustering algorithm and a collection of comparable strings to be provided and used by the users. Having access to the cluster interface, allows to autonomously ascertain whether a projection belongs or not inside a cluster, this will be useful for both the index update and the queries done by the users (more details will be provided in section 4.4.2). The tags are String representing the cluster content. These tags can then be interpreted by the subscribe methods allowing subscription to content.

If an index is created, the indexer creates and assigns a unique tag to it. The generated tag is unique because each cluster should have a unique tag, as this allows to access the content of the cluster exclusively, this avoids possible wrong mappings caused by a bad tag generation.

Since clusters are constantly evolving, and being these indices of most importance in order to retrieve data, it is mandatory at this level to evaluate the evolution of these clusters in order to best migrate the indices. To do so, this component keeps track of indices being updated and at each stage, it evaluates the state of old indices. In the context of our thesis, we focused on two main states being the merged and diverged state.

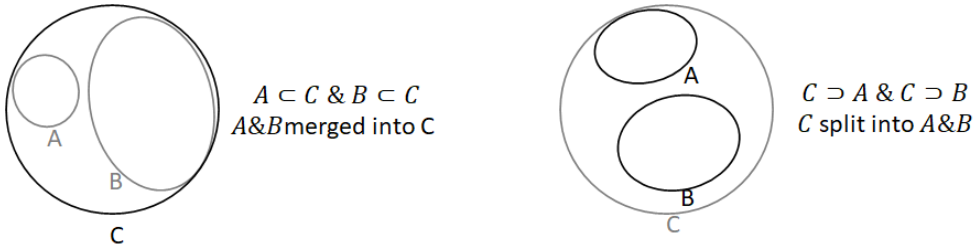

Figure 4.5: States of the cluster and validation.

These states try to evaluate if the old clusters have collapsed into a bigger one, or have split into smaller clusters because a new one was created from them and if so, it assigns its indices to the ones representing them as well. This allows us to delete unused clusters whilst maintaining the tags contents.

Merged State: This state represents a cluster who has stopped updating its state because it was absorbed by another one. If such behaviour happens then it indicates that content was being unnecessarily being specific, thus meaning that this new cluster should represent a more sparse definition of the previously published content. This being said, the indexer simply assigns the indices inside each cluster who was absorbed, into the new cluster. It is important to mention that this procedure removes precision from the index as it merges two distinct indices into one.

Diverged State: This state represents a cluster who has stopped updating its state because it has been split into two distinct clusters. If such behaviour happens then it most probably indicates that content was being too broad, thus meaning that the resulting clusters should represent a more specific definition of the old content. This being said, the indexer simply assigns the indices inside the old cluster to each new cluster who might have been split.

To make these evaluations, we simplified our problem by making some naive assumptions. One of the assumptions consists in the containment of a cluster: Let us assume two clusters, A and B. A is inside B if its center is inside B. The second assumption is that each index may evolve from either a merge or a diverge, otherwise we do not take any action nor conclusion.

Now that we mentioned the simplifications done regarding the evaluation of the clusters, we can now describe the two conditions:

1. Being A an updated cluster, it is a result of a merge if it exists more than one old cluster inside of him (fig. 4.5, left).

2. Being A an old cluster, it has diverged into new clusters if it exists more than one new cluster inside of him (fig. 4.5, right).

As these indices are being created and updated, these changes are being disseminated to the mobile devices so they can update their indices as well. This is possible due to the replication offered by a conflict-free replicated data types (CRDTs) supported by Thyme. To implement this data type, we defined a state for the cluster and a conflict-free replicated data type object who updates and merges this state.

Being the state defined by the index itself, it is only required to wrap this state in order to implement the required methods. Although this optimization is important to ensure both performance but also evolution of the indices, further study may be required to fully understand if such optimization is recommended as it also raises several issues. For instance, when removing two old clusters (due to a merge) we also lose the distinction between these two clusters, which may prove useful for evolution but also delete information.

4.2.6.2 Overseer

The *overseer* is a component designed to run independently from the main module and is responsible for triggering the update from the indexer and updating indices.

This component runs periodically evaluating, when awakened, all projections and the resulting map between these projection and the clusters inside the clustering algorithm. This update consists in re-evaluating all insertions, using the clustering algorithm to re-estimate them and re-assign them to their new representatives. This update is important since data can, as clusters evolve, be reassigned to a new cluster, requiring to re-evaluate its indices as well.

After the update is done, the overseer then evaluates each cluster. If this cluster meets the condition defined then it is called to generate or update its indices. This condition is arbitrary, however its scope is defined to the list of projections inside the cluster, allowing predicates regarding both the size of the cluster and validation of its content.

When both updates are done, the overseer logs statistics done regarding each part of the process, e.g. images inserted, clusters detected and indices registered. This analysis exposes information regarding the indexer, i.e the number of times indices were merged (or diverged), indices created and time of last update.

4.3 Workflow

In this section, we describe and illustrate the overall workflow of our systems three main operations. More details regarding each operation will be provided in the following sections.

The first workflow allows a user to upload photographs into the system (a sequence diagram of the operation is presented in figure 4.6). These photos are stored in *Thyme* for future access (through its subscribe operation) and also contributes to the evolution of *Chives* indices. This is done by the users mobile device, via *Oregano*s publish with

computation, providing both the photo, user defined tags and the computation to add a new projection from *Chives*. As images are added, the stationary server will (using *GardenBed*s ranking algorithm) begin to download popular photographs shared between its users, applying for each the mentioned add projection computation. This computation consists in a pipeline of operations starting at a scan for faces inside the photograph. If the algorithm detects faces, they are then extracted and, for each detected face, it is extracted comparable features from them. Then, these features are mapped to the original image and are clustered.

The second main operation consists in the update of its indices (fig. 4.7) and it does not require an interaction from the user, however it does require some conditions to be met. This task is asynchronous and called after a certain interval, evaluating parameters at the clustering level and only updating if conditions are met, i.e. if n points were added since last update. If these condition are met then *Chives* gets the clustering and for each mapped projection, estimates its new representatives. After this estimation is done, it is then evaluated for each cluster if indexing conditions are met, i.e. if a cluster size is bigger than x, and if positive, the cluster is saved. The saved clusters are then used to generate new indices or, if they already exist, are updated instead. As some clusters may disappear or even merge into other clusters, after the creation/updating phase is finished, indices who were not updated are also evaluated for both cleanup and to ensure the cluster evolution is tracked by its indices, more details will be provided in sub-section 4.2.6.1. Finally these updates are replicated to the mobile devices due to the guarantees offered by the CRDT being used.

The third main operation consists in a query for similar images 4.8). This content-based image retrieval operation, consists in returning to the user images contained in the same group as the photo submitted. *Chives* offers this computation, requiring for the user to run its operation over the image, returning the estimated indices. This is done by scanning the image for faces and extracting for each its features. Then for each projection generated, measure the cluster with more chance to be representative and return it to the user. If no index is considered representative or its chance falls under a confidence threshold then the user does not receive any index. Finally, for each index received, it is only required to subscribe to the tags contained inside the cluster.

Although *Chives* operation can be run locally (due to its indices being propagated to the mobile devices), due to mobile devices limited resources, it is not expected for this operation to run completely local on every device. Furthermore, if resources are met, then running this operation locally would fasten the operation time considerably. As such we offer both alternatives, allowing devices with lower resources to run *Oregano*s subscribe with computation, providing this query operation (an illustration can be seen in figure 4.10).

More details regarding the evaluation of the old indices and cleanup will be mentioned in subsection 4.2.6.1

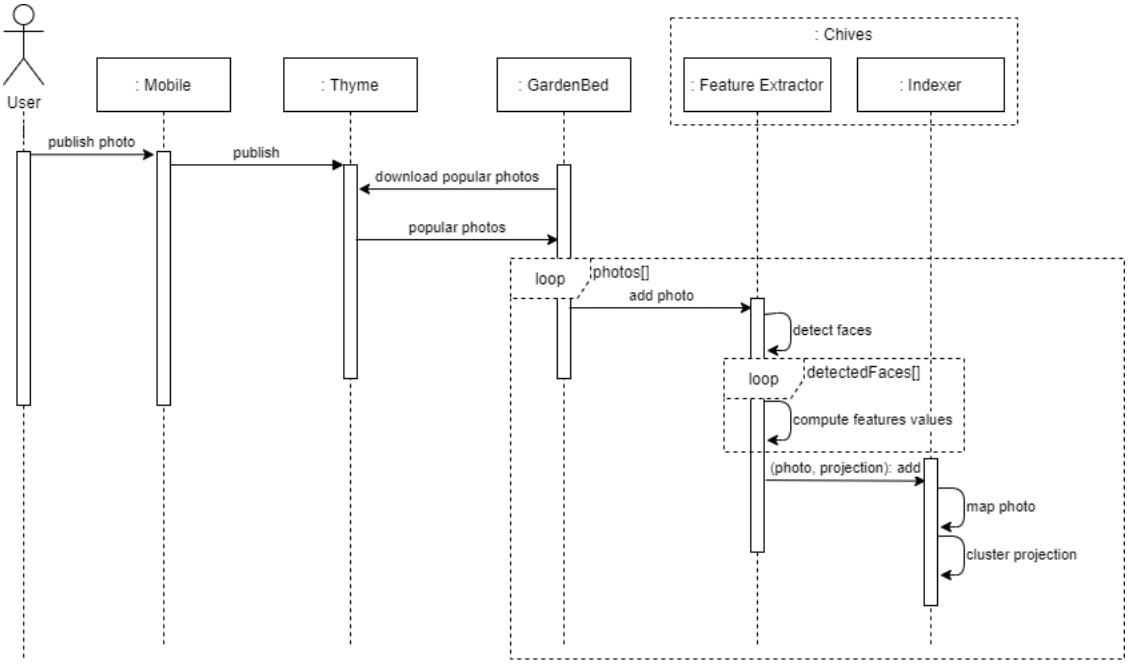

Figure 4.6: Sequence diagram illustrating the upload of a photo.

4.4 Chives & Oregano

As mentioned in the beginning of this chapter, *Chives* is aimed to be part of an *Oregano* service. This is done so we can allow distributed computations in our solution since due to the computational weight of processing an image, not every device should be expected to run all the required processes. Integrating *Chives* into *Oregano* can be done by implementing services who use *Chives* operations and access its data. Furthermore, it is required to implement a computation server running the implemented services, assigning them to the peers responsible for handling the request.

In the following sections we will describe two case-studies: A service that adds an image to *Chives* and a service using *Chives* to retrieve the index containing similar faces.

4.4.1 Index Generation

This service is the one responsible for processing the downloaded images and generate indices. As mentioned in section 4.2.3, clustering will allow us to create groups (or clusters) of points regarding their euclidean distances. As the size of a cluster begins to increase, i.e. the number of points inside it begins to increase, future updates may start to indicate as worthy to generate indices according to its content. An overview of the generation of indexes can be viewed in figure 4.9.

The ranking algorithm provided by *GardenBed*, described in section 3.2, is important for this operation as it is responsible for deciding which content to download, using popularity metrics. After this decision is made, *GardenBed* downloads and stores the

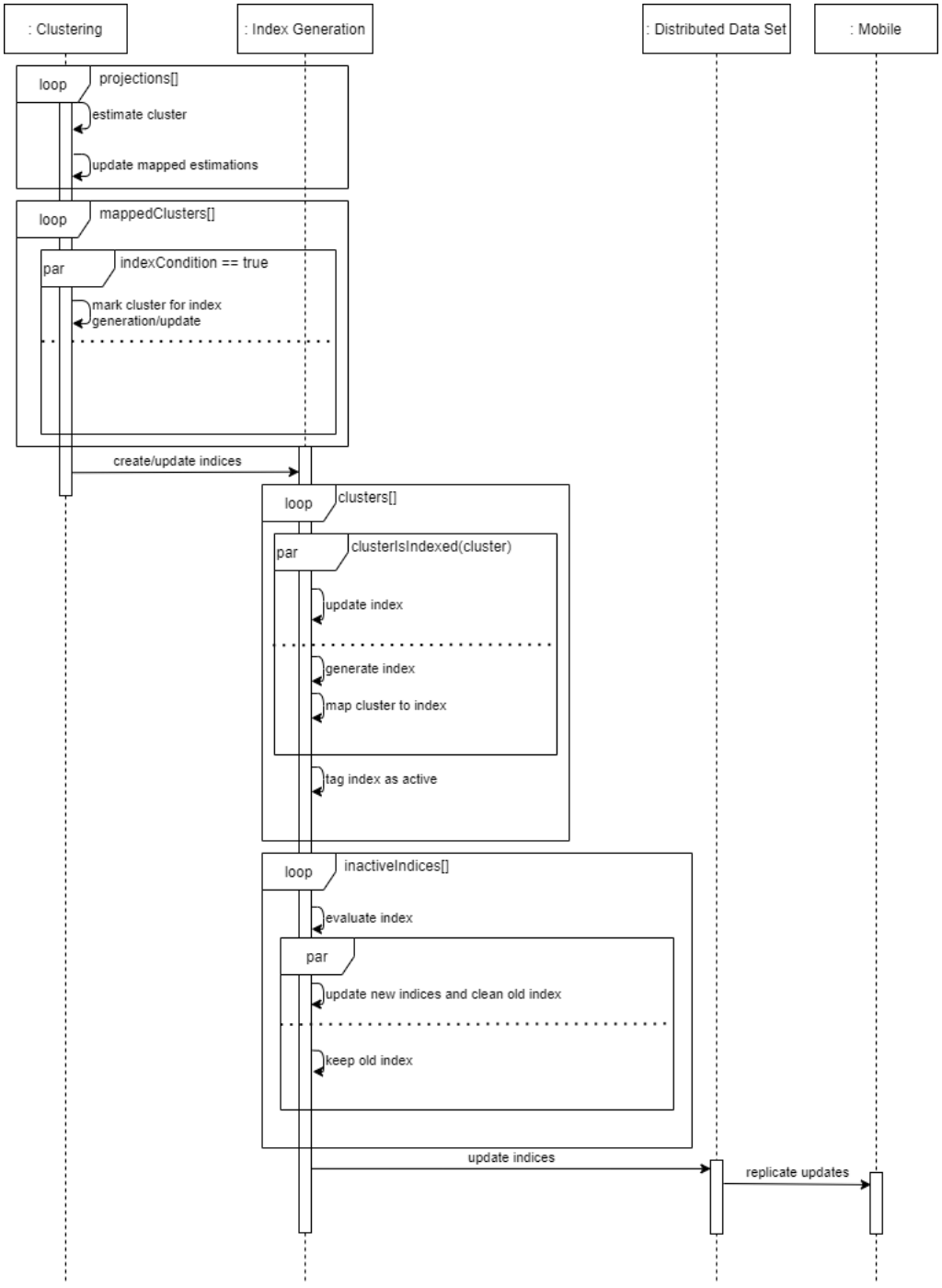

Figure 4.7: Sequence diagram illustrating the creation, update and propagation of the indices.

decided content, as required by our system (mentioned in 4.1.2). After the content is downloaded, *Oregano* will then process all downloaded content using *Chives Indexer* to generate the respective indices.

As these indices are being generated and/or updated, they are being replicated as well to every device who has subscribed to its updates. This is made possible due to *Thyme*

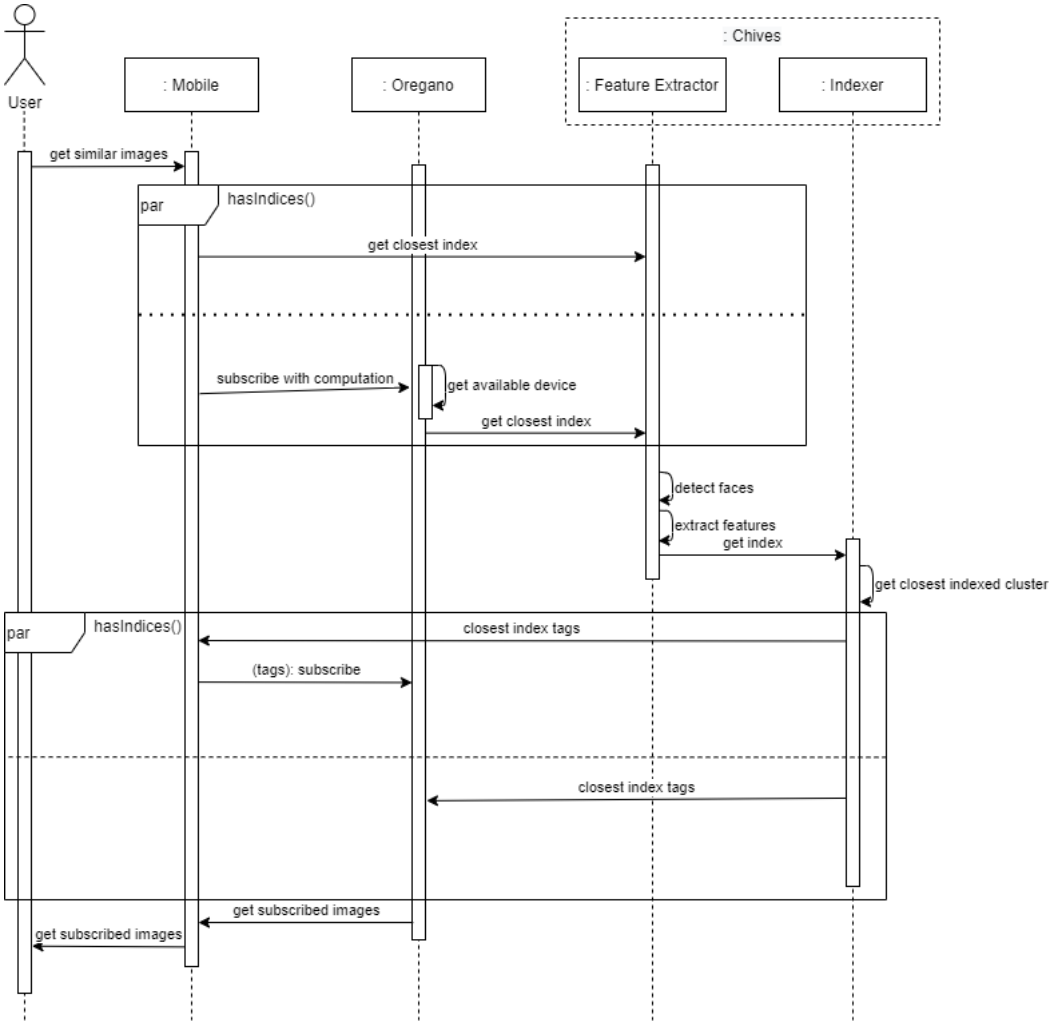

Figure 4.8: Sequence diagram illustrating the query for similar images.

support of CRDT since, as mentioned in 3.1.2, they propagate their state to every replica. This allows for the users to, later on, operate over those indices for numerous operations.

It is important to mention that although the insertion into the system is done immediately, the indexes may not reflect it as fast. This may happen due to two main factors, the process aforementioned works asynchronously under an update criteria and the consistency offered by the CRDTs which is eventual.

In the following sub-sections we deeply describe *Chives* contribution to the system. We begin by describing the module responsible for extracting the features of the faces inside the images, followed by the indexer We end this section with a more detailed description of the workflow behind this service.

4.4.2 Index Query

Users may not wish to use subscribe to content by providing keywords. This can happen due to many reasons and as such, we propose a content-based query, where the user can

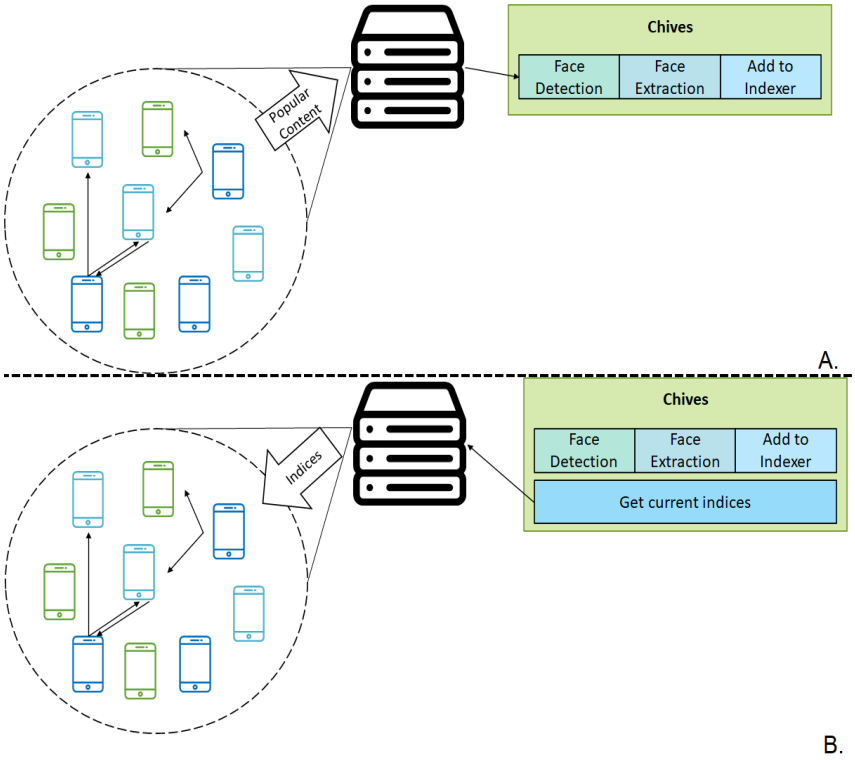

Figure 4.9: Overview of the generation of Indices operation: A. The images are downloaded and inserted into the indexer. B. The Indices are disseminated to the users.

subscribe to content just by providing an image as reference. This type of query has become very popular as it can be a very efficient method to search for similar content without any analysis nor interpretation beforehand from the user.

Our proposal consists of another computational pipeline with four stages:

Stage 1 - Face Detector: detects all faces found in a given photo.

Stage 2 - Feature Extractor: extracts the features of all face images received.

Stage 3 - Infer Index: we measure, for each projection, the inclusion probability of each index, returning the most probable. In case of a tie, one of the indices is returned. If there is not an index with a high enough probability, it is provided feedback and returned the closest index. The cluster returned in this case is the most probable one or, as a last resort, a Knn from the projection to each cluster center.

Stage 4 - Retrieve Tags: extracts the tags inside the inferred index and returns them.

If the device has both the indices and enough resources to run the pipeline, then it can determine the tags by himself, running the pipeline locally. Doing this not only fastens the operation, but also reduces overall requests to the server, increasing its availability. Using *Oregano*, we can increase this availability further by allowing mobile devices capable of querying the index locally, to act as a server for the mobile devices who can not. Then,

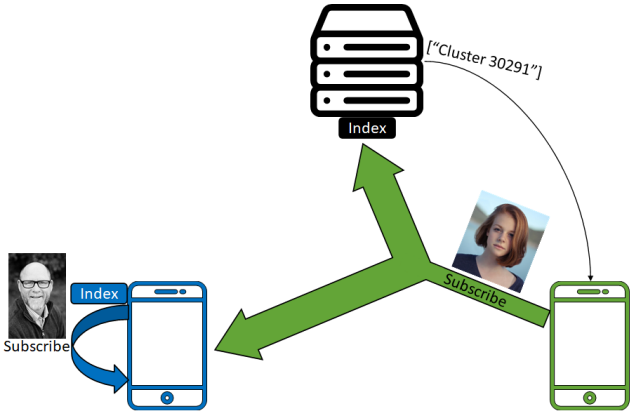

Figure 4.10: Index Query: The user can behave according to one of the possibilities: Left - Do the operation by himself. Right - Request assistance.[2]

if a device cannot do this operation, it only needs to ask for a server who can realize the computation for him. This can be done by using *Oregano* subscribe with computation method and providing an image (an illustration of this process can be viewed at figure 4.10). The server attending to the request will then send to the device requesting the tags extracted from its inference, allowing him to subscribe to the content contained inside the tag.

However, it is important to mention that requests attend by the mobiles, may produce different results from the server. This happens due to the nature of the conflict-free replicated data type. Since the data is centralized in the server and being it the responsible for generating and updating tags, it will be the first to provide the updated results to the user while the devices will receive, eventually, the same updates. Until then, even though it is available to attend requests, these results will be delivered regarding the older version.

4.5 Final Remarks

In this chapter we introduced and explained the design of the system plus its integration with the foundations described in chapter 3. We described the contributions from each device and the server in the system, how it should work and behave during several operations and under to some conditions. We also described our proposal, *Chives* and how it contributes to the design, describing its two main modules, the *feature extractor* and the *indexer*. We also proposed two important services for the development of our proposal, the index generator and the index query, describing their workflow as well. Being our indexer designed with a certain level of abstraction, to allow its use for any content rather than just images and even any other clustering algorithm implementation,

[2]This Image contains photos made by Christopher Campbell and Robert Godwin, available at Unsplash.

we also described some implementation details regarding it in order to aid in any extension or reuse of this module, starting by describing the projection interface at 4.2.1 and the process behind the addition of a clustering algorithm, including our implementation for the clustering algorithms provided by *MOA* [14], followed by the implementation of algorithm specific properties (allowing to set parameters).

In the following sections we will describe how we validate our proposal mentioning our three stages of evaluation and its results, followed by a brief conclusion of the thesis and enumerate future work.

5

Evaluation

In this chapter we will present our experimental evaluations and how they validate our solution. We begin by establishing the goals of our evaluations in Section 5.1, splitting into two main points: the framework and our implementations. After establishing the goals, we begin by evaluating the former in Section 5.2, followed by the latter in the next sections. We begin with a clarification of our case study in Section 5.3, followed by our evaluation of the implemented facial extractor in Section 5.4 and Indexer in Section 5.5. We end this chapter in Section 5.6 with a summary of the attained results and the conclusions drawn from the aforementioned evaluations.

5.1 Goal

The main goal of this thesis was to introduce a framework for dynamic index generation of images, allowing us to bring machine learning's clustering to the edge and evaluate its impact. As such, to validate our studies we will validate *Chives* in both its states: as a framework and using our own implemented processes. Regarding the former, it is required to assert over the quality of the code and how can it be further improved for future related research, to do so we will evaluate three aspects:

Expressivity: How adapted is our framework for distinct scenarios? How much can we customize from our framework?

Extension Effort: What is the effort to extend our framework?

Abstraction Level: How much does our framework abstract from its underlying logic and utilities?

Following the evaluation of our framework, it is also required to validate our proposed processes, namely the quality of the indices generated. In a first stage, during the implementation of our solution, we tested each component to assert over its quality. This

evaluation was of most importance as each further implemented component had a dependency from the former, meaning that a bad validation could compromise the quality of future components, thus affecting the following validations. Hereupon, we divided the evaluation into two main phases, with the following goals:

***Facial Extractor*:** Does it detect and extract faces from a photo and convert the extracted faces into comparable features?

***Clustering Algorithm*:** Cluster images by their similarity. Build representative clusters (for example a cluster could represent a person). Allow evolution, permitting adaptation to new content.

In the following sections we will begin by validating our infrastructure, asserting over the aforementioned metrics, followed by a description over the environment implemented to isolate and test *Chives*, followed by a deep description of the tests realized.

5.2 Framework

A framework is built to aid, simplify and abstract the user from possible complex problems. Furthermore, frameworks aim to reduce possible code repeatibility by providing modular solutions to the problem. Having said that, to evaluate *Chives* quality we looked into aspects that could reflect this ease in extending and usage.

5.2.1 Expressivity

Looking at *Chives* expressivity, although limited by our computational pipeline, the user has total control over the algorithms being used in the infrastructure, being *Chives* only responsible for storing, processing and generate data running the provided implementation. This allows a bigger personalization of the process, allowing new and more distinct techniques to be implemented, however it may also require more implementation due to the lack of details (e.g. Clustering customization for a clustering algorithm).

5.2.2 Extension effort

Regarding implementation, extending *Chives* can be done with relative ease, requiring implementation of the processes mentioned in section 4.2. Looking at our proposal we implemented three datatypes, a projection representing a MOA's Data Point (containing currently 82 code lines), an Image extension offering OpenCV specific operations (91 lines of code) and a representation of the Cluster representing a generic MOA Cluster (42 lines of code) and containing a customized ID (41 lines). This datatypes are solution specific, so they meet the requirements in our implementation, however *Chives* is already expecting and thus requiring in its interface operations such as projection estimation.

For the clustering algorithm, as we proposed more than a cluster algorithm, we implemented an abstract class wrapping a specifically created algorithm and implementing the expected interface by translating each operation (75 lines). Furthermore, for each specific algorithm implemented, it was then required to setup the algorithm (using customized parameters) and expose its clustering (from 54 to 90 lines). Lastly, for the Feature extractor, we integrated OpenCV algorithms into both our Face Detector and Feature Extraction. The Face Detection uses a pre-trained model, requiring mostly to integrate with our requirements, implement an operation for face detection (77 lines). This algorithm is then coupled with the feature extraction into an implementation of the required interface. To use the three mentioned algorithms (Section 4.2.4), we implemented two levels of abstraction scope specific and its setup, after setup is done this algorithm is connected with *Chives* framework by implementing the feature extraction process (more than 503 lines for the 3 algorithms implementations).

5.2.3 Abstraction Level

Regarding abstraction level, *Chives* focuses on offering a relation between computer vision, machine learning and edge computing whilst abstracting the edge computing process. This allows to focus our implementation in the clustering process whilst abstracting the user from the underlying logic behind Index formation, update and sharing.

In the following sections we will evaluate our solution using machine learning techniques and metrics in order to evaluate the quality of our solution.

5.3 Case Study: Face Indexing

5.3.1 Setup

As already mentioned, *Chives* is built to integrate with *Oregano* (Section 4.2), however to reduce complexity during our evaluation it is first required to isolate all processes and evaluate the quality for each stage. Furthermore, being our environment focused on the process quality, several assumptions were done to test our solution. In this section we aim to describe our testing environment and the assumptions made to ease the evaluation process.

Regarding the Extraction of the Features, it was required to define a model capable of processing each image, this is done so we can effectively extract the required features and reduce noise in the clustering as much as possible. Our proposal falls under this requirement of providing face recognition and a face clustering. So, in order to improve the feature extractor result, we should train a model (or use a pre-trained model instead), before trying to project the "real"images. To train our model we customized several datasets that would provide images for both the training and the "real"process.

5.3.2 Datasets

During the elaboration of this thesis, to represent images uploaded to the system, we have used a dataset of facial images: CASIA-WebFace. This dataset contains a total of 494414 images from several celebrities faces grouped by each celebrity, having a total of 10575 folders however, due to its enormous size, smaller subsets were derived from this original dataset. As our solutions makes use of a trainable model for the extraction of features and being its quality important for our solution's quality, for each subset generated, we built both a Training and a Prediction Set, being the former used for the training of the feature extractor and the latter for the usual tests and represent the user submitted photographs.

One of the built datasets was named as "PerfectWorld"containing 9 distinct persons and a total of 119 images for the training set and 155 for the prediction. This dataset was purposely small and its main objective was to pick images who were visually easier to distinguish, providing an "easier"dataset for our solution. However, this dataset would neither represent the real world, nor contain enough data to effectivelly evaluate our cluster, thus being unreliable for evaluation of our proposal.

For a more reliable evaluation we created a new dataset, named as "SmallerOurWorld"that tries to provide more data for the training and evaluation process whilst representing a more distinct set of people. This dataset aims to train our Feature Extractor with distinct people, providing 179 distinct sets of photos (totalling around 3400 images for training) however, to prevent possible bias, the training dataset is also aimed to contain distinct faces from its evaluation counterpart. Due to limitations regarding the hardware using OpenCV during the training of these models, the number of images being processed needed to be reduced to 1600, thus reducing the quality of our model. For the evaluation process, the generated dataset contains 11 sets of photos (totalling around 4300 images). During model training, due to the nature of the implemented processes, models were also trained with distinct dimensionalities sizes, namelly five, ten and twenty dimensions. These dimensions vary acording to the number of components established in the algorithm and this variation may also allow to best define the optimal value for our solution.

5.4 Facial Extractor

5.4.1 Face Detection

The facial extractor is not connected to the indexer, however it is a crucial component of our system as most of the data processed and indices generated strongly depend on it. That being said, in order to guarantee an easier and more accurate clustering, this extractor should provide accurate and descriptive features, so the *Clustering Algorithm* can evaluate and operate over it with more ease and accuracy as well without compromising the clustering algorithm. This brought a need to test this extractor early as it would allow us to preemptively assert its quality.

Being the thesis, focused on facial features, we tested the face detection models for face detection, provided by *OpenCV*.

We tested several models, evaluating the number of images correctly detected and wrongly. When an image is detected wrongly, we distinguish between whether it was not detected or the classifier detected more than one. This is done because it is known beforehand that the dataset contains one face to be detected and it is mostly probable that an image lacking a face was also detected (these results can be seen in the table 5.1).

Table 5.1: Face Detection Results.

Model	Correct Faces	No Faces	Multiple Faces	Accuracy
HaarCascade FrontalFace Alternative	3357	916	51	77.6
HaarCascade FrontalFace Alternative 2	3415	807	102	79
HaarCascade FrontalFace Alternative 2 CUDA	3505	714	105	81.1
HaarCascade FrontalFace Alternative Tree CUDA	2787	1523	14	64.5
HaarCascade FrontalFace Alternative CUDA	3453	808	63	79.9
HaarCascade FrontalFace Default	3452	561	311	79.8
HaarCascade FrotalFace Default CUDA	3444	541	339	79.6
HaarCascade ProfileFace	1492	2803	29	34.5
HaarCascade ProfileFace CUDA	1778	2504	42	41.1
LBPCascade FrontalFace	3163	1054	107	73.1
LBPCascade FrontalFace Improved	2959	1347	18	68.8

Looking at the results, *HaarCascade FrontalFace Alternative 2 CUDA*[1] offers highest accuracy whilst reducing a great amount of images with zero or many faces, however due to practical motives we have chosen an algorithm that could reduce this number the most: *HaarCascade FrontalFace Alternative Tree CUDA*.

5.4.2 Feature extraction data analysis

After deciding the model for face detection, there were three algorithms used for extracting features from the face image for our use case: *PCA*, *EigenFaces*, *FisherFaces* (section 4.2.2). To assert over the most adequate algorithm, we did a comparative study using each of these algorithms. Using the aforementioned (subsection 5.3.2) dataset "DistinctPeople"we trained several models, alternating between the algorithm and the dimension size, then using the prediction set, we tested each generated model and analysed its resulting points. To analyse the generated results, the projections were stored in a file compatible with the software *MulltiSOM* [52] (section 2.2.3), allowing the visualization of the data. *MulltiSOM* [52] offers a great graphical interface, which allows us to reduce the dimensions of the data and visualize possible clusters.

In addition, we run our clustering algorithm over the generated projections as well and evaluate the resulting clusters. Since our focus was not yet to evaluate our clustering algorithm, we used this method whilst varying all its relevant parameters. To visualise the resulting cluster we saved the projections and used pyhton's matplotlib [39] to generate graphical spaces. Although limited, the two-dimensional generated graphs, allowed

[1] https://github.com/opencv/opencv/tree/master/data/haarcascades_cuda

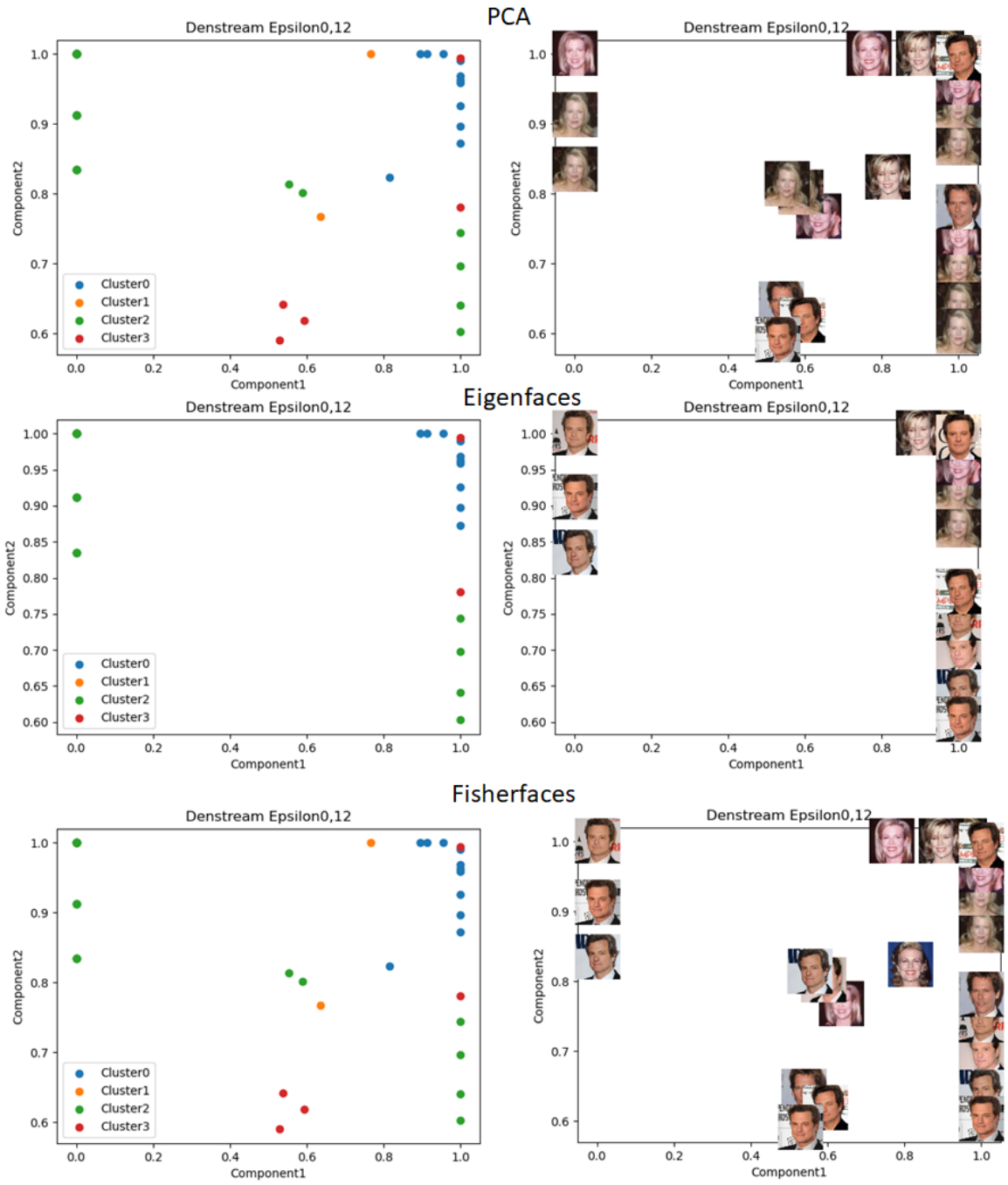

Figure 5.1: Feature Extractor Algorithm Comparison.

to comparatively evaluate each algorithm, figure 5.1 demonstrates the resulting coordinates for the first two principal-component analysis using the *DenStream* algorithm with epsilon value of 0.12.

After analysing these results we can see that neither feature extractor algorithm could

effectively extract the facial features as their results shown distinct faces containing similar coordinates and the exact opposite. Furthermore, we decided to progress our studies using *EigenFaces* as it proved to be a reasonable half-term.

5.5 Clustering Projections

Although the results provided in subsection 5.4.2 allowed a preemptive assertion over the data, those results required both human intervention and analysis, resulting in a qualitative and less reliable evaluation. As such, in this section we will evaluate the quality of our clustering algorithm and its respective clusters using both unsupervised and supervised metrics.

For our extractor, in consideration with the aforementioned evaluation (section 5.4), we used a feature extractor with the HaarCascade FrontalFace Alternative Tree CUDA model as our face detection algorithm and EigeinFaces with dimensionality 10 as our feature extraction algorithm. For the clustering algorithm, although implemented an integration with four *MOA* algorithms, due to technical issues regarding integration of some of the algorithms that rendered their results incoherent and uninterpretable, our evaluation focused solely on the density-based clustering algorithm *DenStream*. These tests consist in varying the algorithm's parameters whilst analysing its clustering. After evaluating the disparity between results when altering each parameters of the algorithm, we decided to focus these tests on the epsilon parameter.

5.5.1 Unsupervised metrics

More than evaluating the labels of each image inside the cluster, it is important to understand if the data can be discriminated and clustered. In this subsection we aim to use unsupervised metrics to best assert over the generated clusters and determine the more adequate parameters for the established algorithm.

To evaluate our clusters, we created several instances of the clustering algorithm alternating the epsilon parameter and clustered the projections generated by our feature extractor. After the insertions were done, we registered each projection and its representative cluster and measured the silhouette, Calinski-Harabasz and Davies-Bouldin score (section 2.2.4) for the registered clustering (table 5.2 shows the unsupervised score for the first six variations of epsilon).

Comparing these results we can conclude several aspects. Analysing the first coefficient, the silhouette score, we notice that for epsilon values of 0.1 this score is higher, meaning that clusters are better defined when this parameter is used. However this result can also happen due to the smaller clusters, as decreasing the epsilon value reduces the cluster radius threshold. When evaluating the second score, the Calinski-Harabasz index, we can notice that scores are effectively higher for higher epsilon values, however this increase is not proportional as for epsilon value of 0.16, the result was higher than for

Table 5.2: Unsupervised scores for a range of epsilon values in the Denstream algorithm

Epsilon	Silhouette	Calinski-Harabasz	Davies-Bouldin
0.10	0.4792	28.7847	0.6190
0.12	0.3148	34.9826	0.8375
0.14	0.3167	72.3828	0.8747
0.16	0.2898	106.6503	0.9610
0.18	0.1162	98.3177	0.9990
0.20	0.2409	122.1313	1.0272

0.18. Joining with Davies-Bouldin index, we notice that score is lower for lower epsilons as well.

Considering these results we can detect that lowering our epsilon may provide a better clustering and, accordingly, a better separation between each cluster. However, a lower epsilon may also decrease the number of images clustered (and be classified as noise). Furthermore, regarding their absolute values, it is also important to note that for convex clusters, the resulting scores from these metrics tend to increase, meaning that density based clusters may have inherently increased scores. As such, we focus our conclusions in a comparative manner between the same algorithm.

Since lowering our parameter increases our score but may contain less images clustering, we also evaluated each clustering content. In this evaluation we analysed each cluster images based on its original label and introduced a purity metric following this formula:

$$Popularity_{cluster} = \frac{n_x}{\sum_{i=0}^{n_{labels \setminus x}}}$$

Where **x** represents the most occurring label and **labels** represents all distinct labels.

Tables 5.3 and 5.4 present the resulting analysis for both the clustering using an epsilon parameter value of 0.1 and 0.14.

Table 5.3: Cluster purity regarding faces clustered for Epsilon 0.1.

Cluster	P1	P2	P6	P7	P8	P10	Purity
0	0	0	1	0	0	1	0.5
1	1	0	0	0	1	0	0.5
2	0	1	1	0	0	0	0.5
3	1	0	0	0	1	0	0.5
4	1	0	1	0	0	0	0.5
5	0	0	0	2	0	0	1.0

Analysing the tables we confirm the lack of content when providing the lower value of epsilon. The presented result for epsilon value of 0.1 also indicated a required trade-off between the scores presented in the beginning of this section and the content inside each cluster. Furthermore, we can conclude that for epsilon values of 0.14, no cluster is pure, meaning that there is not a cluster containing images of a specific person but instead

Table 5.4: Cluster purity regarding faces clustered for Epsilon 0.14.

Cluster	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	Purity
0	1	0	0	1	0	0	1	0	0	0	0	0.33
1	0	0	0	0	0	0	5	0	0	0	1	0.83
3	0	0	1	0	2	0	0	0	0	0	0	0.67
4	1	1	1	1	0	0	2	0	0	0	0	0.33
5	1	0	0	0	0	1	0	0	2	0	0	0.5
7	0	1	0	0	0	0	2	0	0	0	0	0.67
8	0	2	1	1	0	0	0	0	2	0	0	0.33
9	0	1	1	1	0	0	1	0	0	1	0	0.2
11	4	6	0	0	0	1	3	2	3	0	2	0.29
12	2	3	0	0	0	0	1	0	2	0	2	0.3
16	3	3	1	1	1	2	10	0	2	0	1	0.42
17	11	0	4	0	0	0	7	0	1	0	0	0.48
18	3	1	1	0	0	0	7	1	0	0	0	0.54
19	0	4	0	2	0	0	1	1	2	0	0	0.4

contains several images of distinct people. For epsilon values of 0.14 we can also notice distinct clusters representing the same face label (e.g. Cluster 7 and 16), thus scattering content that should be grouped. Both cluster's purity and label ambiguity affect the descriptiveness of an index, reducing index labels accuracy as well.

5.5.2 Supervised metrics

In this section we evaluate the quality of the cluster considered the closest to the queried projection. We started, by projecting ten images for each face group being represented in the prediction set (totalling 110 images) and added a ground truth label to all our images (based on their face group). After adding the labels, we compared each projected image with the resulting cluster most popular label, being the popularity measured by evaluating the ground truth labels of each image inside the cluster and assigning the most occurring to the cluster. If the returned cluster shared a similar label as the image being queried, we assumed a correct query. We then measured the precision, recall, f1-score and support of the resulting labels.

When evaluating the scores for the multiple epsilon values, we can conclude that the measured scores tend to be low, meaning that the clusters could not represent all faces inside the set(table 5.5 shows the scores for an epsilon value of 0.14). Furthermore several face groups (e.g. two and three) have mostly scored zero, this can happen when either no image could be exclusively identified, i.e. the images could not be projected into a cluster containing similar labels or if the face is not clustered, for instance, it could be scattered through other clusters.

Table 5.5: Supervised scores for epsilon values of 0,14

Face	Precision	Recall	F1-Score	Support
0	0.3077	0.4	0.3478	10.0
1	0.1667	0.625	0.2632	8.0
2	0.0	0.0	0.0	9.0
3	0.0	0.0	0.0	9.0
4	0.25	0.6	0.3529	5.0
5	0.0	0.0	0.0	7.0
6	0.2273	0.5556	0.3226	9.0
7	0.0	0.0	0.0	6.0
8	0.0	0.0	0.0	7.0
9	0.0	0.0	0.0	5.0
10	0.0	0.0	0.0	5.0
accuracy	0.2125	0.2125	0.2125	0.2125
macro average	0.0865	0.1982	0.1170	80.0
weighted average	0.0963	0.2125	0.128	80.0

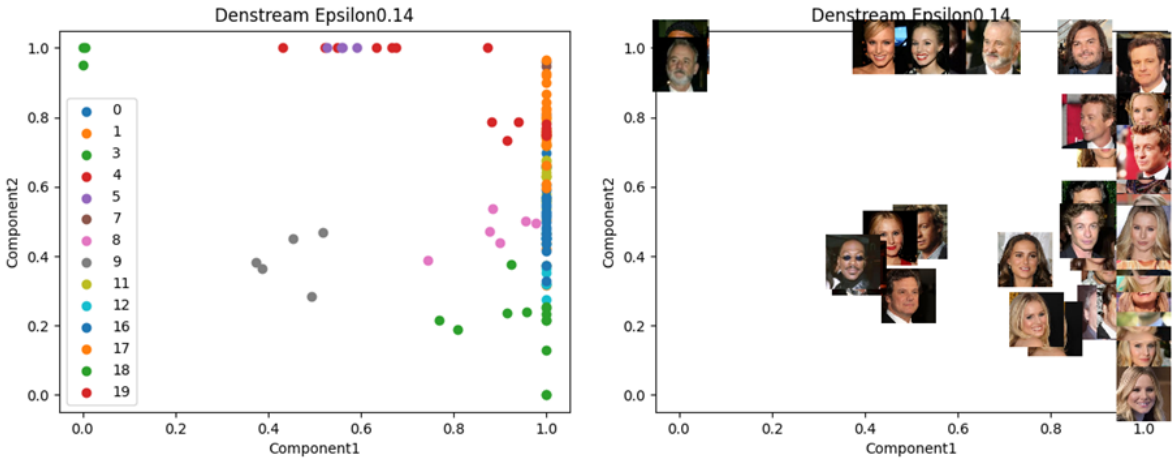

Figure 5.2: Clustering Results for epsilon values of 0.14.

5.5.3 Image Clustering and Indexing

After evaluating the clustering algorithm, in this section we visualise the resulting clustering and discuss over the indexing of our system. For data visualization we used an epsilon value of 0.14 as it presented to be a reasonable trade-off between the obtained scores and the number of images clustered. Figure 5.2 shows both the cluster labels and its original images projected in a two dimensional space representing its first two coordinates.

As noticed in the earlier evaluations, only a small set of the projected data is being clustered, this can happen due to the noise detection of *DenStream*. Density-based algorithms can easily be affected by noise, meaning that their algorithms are usually built with this factor in mind. However, since our solution is not expected to contain any noise, this behaviour reinforces the lack of discrimination of the data projected.

When evaluating the mapping of the indices we can also validate their creation, existing only indices for cluster who do meet the requirements established, however further studies are required in order to validate the implemented operations and adaptation to a constantly changing cluster. These validations are important to also detect and discuss the drawbacks caused by the assumptions made in sub-section 4.2.6.1.

5.6 Summary

After analysing the results, we can see that, providing our face detection, for 81%, we can detect and reduce the original images to its faces. Providing us with 3500 images for clustering. However, When processing these images with the feature extractor, the features provided do not effectively represent each face, meaning that distinct people can often end in close proximity. Furthermore, similar faces can also end in distinct positions. Since our clustering algorithm clusters based on each points euclidean distances, this downside generated a difficult subspace to cluster on, thus diminishing the results provided by our clustering algorithm. However, despite its results we could detect, although in smaller dimensions, clustering. This could happen due to *DenStream*'s noise detection, imbued in density-based clustering algorithms. Furthermore, as cluster increased their size, clusters containing a total size bigger than the one defined in the predicate, were saved in the index maps, however further studies should be done in order to best assert and validate over its process.

In conclusion, further studies and improvements should be done in regards to the final solution, this could be done by introducing new algorithms to our processing pipeline or improving the already proposed ones.

6

Conclusions

This chapter will contain our final remarks regarding this thesis. We will begin by enumerating our conclusions regarding our solution as described in chapter 4 and its respective results, shown in chapter 5. Following our conclusions, in section 6.2 we will enumerate aspects to be improved in this thesis, these aspects range from infrastructure improvement to work planned to extend this thesis.

6.1 Conclusions

In this thesis, we present *Chives*, a solution for dynamic generation of indices for devices running in the edge. *Chives* is built for use by *Oregano*, to allow both automatic generation of indices by the centralized stationary server and content-based query for indices of content.

It is comprised of two main, detached, components: the feature extractor and the indexer. The feature extractor implements algorithms to project images regarding the focus of our thesis: extracting faces and projecting each to a subspace understandable by the indexer. This extractor is also built with the clustering algorithm used in mind. The indexer is purposely built with a level of abstraction, allowing the clustering and indexing of any content and also allows the usage of a specific clustering algorithm (detaching the clustering algorithm from the indexer).

This implementation is not yet finished, therefore we proposed, in chapter 4, the architecture designed during the elaboration of this thesis. This design uses the foundations mentioned in 3, to provide a system running at the edge, using both mobile devices and a stationary server located at the access point or close to it.

We can also conclude that, by using clustering, we can cluster content by its similarity and even use the generated clusters to reduce the complexity of a content-based query to both the devices and the server. This enables a new method of querying content for a system running with limited resources on the edge. However this solution also revealed aspects that must be considered as it can easily affect these results and compromise its

accuracy. The feature extractor can hide or wrongly detect faces in an image, which can increase the number of outliers being inserted in the algorithm and lead to more "noise"inside of it. To prevent these outliers, we can use a density-based clustering algorithm which can filter this noise, however this can also cause the algorithm to falsely accuse a projection as noise, which will reduce the number of clusters, hence reducing the number of indices as well as their sizes. Our results shown these aspects, providing the means of clustering content but also ignoring other great clusters.

This study, serves as a proof of concept of our solution and presents results who can motivate to further study this solution.

6.2 Future Work

Chives is a framework built with our problem in mind, however as we can see in chapter 5 there is still a lot of room for improvement regarding our proposal. In this section we present several considerations that could be done so our solution could be improved and worked on to get closer to the designed architecture. These considerations will be listed according to our evaluation method, followed by architectural proposals.

6.2.1 Feature Extractor

As already mentioned, the *Feature Extractor*, plays a very important role in the system, meaning that improving the extractor would mean a direct improvement in the indexer as well (as seen in the 5.4 and 5.5). However, extracting features is not an easy task moreover considering the limited resources from the mobile devices. Following, we propose two possible improvement points to extend our work.

Accuracy: As seen in the section 5.4, although this extractor provides means to cluster, it still has trouble distinguishing some faces and not every face is detected for clustering. This is caused by both the model for face detection and *PCA*. A good aproach could be to improve this processing pipeline, either by upgrading both the face detection algorithm and feature extractor or by providing better models. The former could consist in researching better techniques for feature extraction, improving our processing pipeline or the already defined stages by introducing better algorithms, e.g. deep learning algorithms. Alternatively, as later described, we could try to improve the already proposed algorithm by deriving better models and evaluate if it does provide an interesting result that may suffice in comparison to other, more computationally expensive, solutions.

Image Range: For now, this extractor was only tested using photos containing only a single face. This has eased our research however, this is only true under a simulated environment (as image-sharing software can receive any content, who may or may not have faces). To enhance our solution and extend it to more than just a proof of concept, it could

be also interesting to extend the researching environment by removing this assumption and evaluate our framework adaption to this solution.

6.2.2 Clustering Algorithm

Although limited by the subspace generated, the clustering algorithm allowed to visualise potential clusters, however it also provided several false positives in regards to noise, clustering only a small set of the provided images. That being said, it could be also interesting to invest in improving the clustering algorithm as well by providing new algorithms or test the already implemented exhaustively. However, it is important to consider that before upgrading the clustering algorithm, the feature extractor should already provide a reasonable projection set for clustering as most of the presented results for the clustering algorithm being tested may have been hindered by this factor.

6.2.3 Indexer

As mentioned in chapter 5, the *Indexer* could not be effectively validated, as such several validations should be done to *Chives* as well to better understand and detect possible drawbacks. Future work in this topic could consist in studying and improving the dynamic evolution of these indices as for now we only know they exist.

6.2.4 System

After improving our solution, there are still points to work upon in order to integrate our solution with *Oregano*.

Services and Servers: Our proposed services and computation servers should be evaluated and improved in order to best fulfill the defined workflows and integrate with the remaining architecture.

Mobile devices: A mobile application can be implemented running the software stack mentioned in figure 4.2 and offer a photo sharing service inside our environment that meets the defined workflows.

Access Points: Access points can be extended to add processing during the download of popular content, thus allowing the usage of *Chives* for image insertion and clustering whilst the content is being downloaded.

6.2.5 Data streaming environment

More than upgrading our solution, evaluation must also be improved to represent a more realistic environment. Since our focus is on edge computing, future upgrades in the evaluation process could be done by simulating a data stream dataset and evaluate the

adaptability of the solution. Furthermore, this validation should also be improved by integrating with other components being implemented as suggested in the earlier sections up to a scenario where it is possible to evaluate and simulate the designed architecture.

Bibliography

[1] OpenCV. URL: https://opencv.org (cit. on pp. 10–12, 37).

[2] Android. URL: https://www.android.com (cit. on p. 10).

[3] Deep Learning For Java(DL4J). URL: https://deeplearning4j.org/ (cit. on p. 11).

[4] TensorFlow. URL: https://www.tensorflow.org/lite (cit. on p. 11).

[5] Google. URL: https://www.google.com (cit. on p. 11).

[6] JavaCPP Presets. URL: https://github.com/bytedeco/javacpp-presets (cit. on p. 37).

[7] C. C. Aggarwal et al. "- A Framework for Clustering Evolving Data Streams". In: *Proceedings 2003 VLDB Conference*. Ed. by J.-C. Freytag et al. San Francisco: Morgan Kaufmann, 2003, pp. 81–92. ISBN: 978-0-12-722442-8. DOI: https://doi.org/10.1016/B978-012722442-8/50016-1. URL: http://www.sciencedirect.com/science/article/pii/B9780127224428500161 (cit. on pp. 16, 40).

[8] T. Ahonen et al. "Rotation Invariant Image Description with Local Binary Pattern Histogram Fourier Features". In: *Image Analysis*. Ed. by A.-B. Salberg, J. Y. Hardeberg, and R. Jenssen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 61–70. ISBN: 978-3-642-02230-2 (cit. on p. 10).

[9] E. Alpaydin. *Introduction to Machine Learning*. 2nd. The MIT Press, 2010. ISBN: 026201243X, 9780262012430 (cit. on p. 7).

[10] I. Azimi et al. "HiCH: Hierarchical Fog-Assisted Computing Architecture for Healthcare IoT". In: *ACM Trans. Embed. Comput. Syst.* 16.5s (Sept. 2017). ISSN: 1539-9087. DOI: 10.1145/3126501. URL: https://doi.org/10.1145/3126501 (cit. on pp. 18–20, 22).

[11] H. Bay, T. Tuytelaars, and L. Van Gool. "SURF: Speeded Up Robust Features". In: *Computer Vision – ECCV 2006*. Ed. by A. Leonardis, H. Bischof, and A. Pinz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 404–417. ISBN: 978-3-540-33833-8 (cit. on p. 12).

[12] P. Belhumeur, J. Hespanha, and D. Kriegman. "Eigenfaces vs. Fisherfaces: recognition using class specific linear projection". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 19.7 (1997), pp. 711–720. DOI: 10.1109/34.598228 (cit. on p. 38).

[13] J. Beringer and E. Hüllermeier. "Online clustering of parallel data streams". In: *Data & Knowledge Engineering* 58.2 (2006), pp. 180–204. ISSN: 0169-023X. DOI: https://doi.org/10.1016/j.datak.2005.05.009. URL: http://www.sciencedirect.com/science/article/pii/S0169023X05000819 (cit. on p. 15).

[14] A. Bifet et al. "MOA: Massive Online Analysis". In: *J. Mach. Learn. Res.* 11 (2010), pp. 1601–1604. URL: http://portal.acm.org/citation.cfm?id=1859903 (cit. on pp. 16, 39, 51).

[15] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387310738 (cit. on p. 9).

[16] T. Caliński and H. JA. "A Dendrite Method for Cluster Analysis". In: *Communications in Statistics - Theory and Methods* 3 (Jan. 1974), pp. 1–27. DOI: 10.1080/03610927408827101 (cit. on p. 17).

[17] F. Cao et al. "Density-Based Clustering over an Evolving Data Stream with Noise". In: *Proceedings of the 2006 SIAM International Conference on Data Mining*, pp. 328–339. DOI: 10.1137/1.9781611972764.29. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611972764.29. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611972764.29 (cit. on pp. 16, 40).

[18] M. Carnein and H. Trautmann. "Optimizing Data Stream Representation: An Extensive Survey on Stream Clustering Algorithms". In: *Business & Information Systems Engineering (BISE)* 61 (2019), pp. 277–297. DOI: 10.1007/s12599-019-00576-5 (cit. on p. 16).

[19] F. Cerqueira et al. "Towards a Persistent Publish/Subscribe System for Networks of Mobile Devices". In: *Proceedings of the 2nd Workshop on Middleware for Edge Clouds & Cloudlets*. MECC '17. Las Vegas, Nevada: Association for Computing Machinery, 2017. ISBN: 9781450351713. DOI: 10.1145/3152360.3152362. URL: https://doi.org/10.1145/3152360.3152362 (cit. on p. 23).

[20] J. Chen et al. "A Real-Time Multi-Task Single Shot Face Detector". In: *2018 25th IEEE International Conference on Image Processing (ICIP)*. Oct. 2018, pp. 176–180. DOI: 10.1109/ICIP.2018.8451649 (cit. on p. 11).

[21] Y. Chen and L. Tu. "Density-Based Clustering for Real-Time Stream Data". In: *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '07. San Jose, California, USA: Association for Computing Machinery, 2007, pp. 133–142. ISBN: 9781595936097. DOI: 10.1145/1281192.1281210. URL: https://doi.org/10.1145/1281192.1281210 (cit. on pp. 16, 40).

[22] D. L. Davies and D. W. Bouldin. "A Cluster Separation Measure". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-1.2 (1979), pp. 224–227. DOI: 10.1109/TPAMI.1979.4766909 (cit. on p. 17).

[23] S. Dehuri et al. "Comparative study of clustering algorithms". In: (2006) (cit. on pp. 13, 14).

[24] T. Deselaers, D. Keysers, and H. Ney. "Features for image retrieval: an experimental comparison". In: *Information Retrieval* 11.2 (Apr. 2008), pp. 77–107. ISSN: 1573-7659. DOI: 10.1007/s10791-007-9039-3. URL: https://doi.org/10.1007/s10791-007-9039-3 (cit. on pp. 11, 12).

[25] K. Dolui and S. K. Datta. "Comparison of edge computing implementations: Fog computing, cloudlet and mobile edge computing". In: *Global Internet of Things Summit, GIoTS 2017, Geneva, Switzerland, June 6-9, 2017*. IEEE, 2017, pp. 1–6. ISBN: 978-1-5090-5873-0. DOI: 10.1109/GIOTS.2017.8016213. URL: https://doi.org/10.1109/GIOTS.2017.8016213 (cit. on p. 2).

[26] U. Drolia, K. Guo, and P. Narasimhan. "Precog: prefetching for Image recognition Applications at the Edge". In: *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. SEC '17. San Jose, California: Association for Computing Machinery, 2017. ISBN: 9781450350877. DOI: 10.1145/3132211.3134456. URL: https://doi.org/10.1145/3132211.3134456 (cit. on pp. 18–20, 22).

[27] U. Drolia et al. "Cachier: Edge-Caching for Recognition Applications". In: *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*. Ed. by K. Lee and L. Liu. IEEE Computer Society, 2017, pp. 276–286. ISBN: 978-1-5386-1792-2. DOI: 10.1109/ICDCS.2017.94. URL: https://doi.org/10.1109/ICDCS.2017.94 (cit. on pp. 2, 18–20, 22).

[28] M. Ester et al. "A density-based algorithm for discovering clusters in large spatial databases with noise". In: AAAI Press, 1996, pp. 226–231 (cit. on p. 14).

[29] *Exponential progress - Moore's Law*. Our World In Data. URL: https://ourworldindata.org/technological-progress (cit. on p. 9).

[30] X. Fan et al. "Exploiting the edge power: an edge deep learning framework". In: *CCF Transactions on Networking* 2.1 (June 2019), pp. 4–11. ISSN: 2520-8470. DOI: 10.1007/s42045-018-0003-0. URL: https://doi.org/10.1007/s42045-018-0003-0 (cit. on pp. 18–22).

[31] S. S. Farfade, M. J. Saberian, and L.-J. Li. "Multi-View Face Detection Using Deep Convolutional Neural Networks". In: *Proceedings of the 5th ACM on International Conference on Multimedia Retrieval*. ICMR '15. Shanghai, China: Association for Computing Machinery, 2015, pp. 643–650. ISBN: 9781450332743. DOI: 10.1145/2671188.2749408. URL: https://doi.org/10.1145/2671188.2749408 (cit. on p. 11).

[32] Y. Freund and R. E. Schapire. "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting". In: *Journal of Computer and System Sciences* 55.1 (1997), pp. 119–139. ISSN: 0022-0000. DOI: https://doi.org/10.1006/jcss.1997.1504. URL: http://www.sciencedirect.com/science/article/pii/S002200009791504X (cit. on p. 10).

[33] M. Ghesmoune, M. Lebbah, and H. Azzag. "State-of-the-art on clustering data streams". In: *Big Data Analytics* 1.1 (2016), p. 13 (cit. on pp. 14–16).

[34] P. Guo et al. "FoggyCache: Cross-Device Approximate Computation Reuse". In: *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*. MobiCom '18. New Delhi, India: Association for Computing Machinery, 2018, pp. 19–34. ISBN: 9781450359030. DOI: 10.1145/3241539.3241557. URL: https://doi.org/10.1145/3241539.3241557 (cit. on pp. 18–20, 22).

[35] A. Guttman. "R-Trees: A Dynamic Index Structure for Spatial Searching". In: *SIGMOD Rec.* 14.2 (June 1984), pp. 47–57. ISSN: 0163-5808. DOI: 10.1145/971697.602266. URL: https://doi.org/10.1145/971697.602266 (cit. on p. 3).

[36] J. A. Hartigan. *Clustering Algorithms*. 99th. USA: John Wiley & Sons, Inc., 1975. ISBN: 047135645X (cit. on p. 13).

[37] D.-C. He and L. Wang. "Texture unit, texture spectrum, and texture analysis". In: *IEEE transactions on Geoscience and Remote Sensing* 28.4 (1990), pp. 509–512 (cit. on p. 12).

[38] D. Heesch and S. Rüger. "Combining Features for Content-Based Sketch Retrieval — A Comparative Evaluation of Retrieval Performance". In: *Advances in Information Retrieval*. Ed. by F. Crestani, M. Girolami, and C. J. van Rijsbergen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 41–52. ISBN: 978-3-540-45886-9 (cit. on pp. 11, 12).

[39] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55 (cit. on p. 56).

[40] *Java*. Oracle. URL: https://www.java.com (cit. on pp. 10, 11).

[41] J. Kinnunen et al. "Dynamic indexing and clustering of government strategies to mitigate Covid-19". In: *Entrepreneurial Business and Economics Review* 9.2 (June 2021), pp. 7–20. DOI: 10.15678/EBER.2021.090201. URL: https://eber.uek.krakow.pl/index.php/eber/article/view/1284 (cit. on p. 3).

[42] T. Kohonen. "The self-organizing map". In: *Proceedings of the IEEE* 78.9 (Sept. 1990), pp. 1464–1480. ISSN: 1558-2256. DOI: 10.1109/5.58325 (cit. on p. 14).

[43] P. Kranen et al. "The ClusTree: indexing micro-clusters for anytime stream mining". In: *Knowledge and Information Systems* 29.2 (Nov. 2011), pp. 249–272. ISSN: 0219-3116. DOI: 10.1007/s10115-010-0342-8. URL: https://doi.org/10.1007/s10115-010-0342-8 (cit. on pp. 16, 40).

[44] L. Krippahl. "Lecture Notes". In: *Aprendizagem Automática(Machine Learning)*. 2018. URL: http://aa.ssdi.di.fct.unl.pt/files/LectureNotes.pdf (cit. on pp. 7, 16).

[45] L. Krippahl. "Lecture Notes". In: *Aprendizagem com Dados Não Estruturados*. 2019. URL: http://adne.ssdi.di.fct.unl.pt/files/ADNE_notes.pdf (cit. on p. 10).

[46] D. Li et al. "DeepCham: Collaborative Edge-Mediated Adaptive Deep Learning for Mobile Object Recognition". In: *2016 IEEE/ACM Symposium on Edge Computing (SEC)*. Oct. 2016, pp. 64–76. DOI: 10.1109/SEC.2016.38 (cit. on pp. 18–22).

[47] G. Li and A. Khokhar. "Content-based indexing and retrieval of audio data using wavelets". In: *2000 IEEE International Conference on Multimedia and Expo. ICME2000. Proceedings. Latest Advances in the Fast Changing World of Multimedia (Cat. No.00TH8532)*. Vol. 2. 2000, 885–888 vol.2. DOI: 10.1109/ICME.2000.871501 (cit. on p. 3).

[48] H. Li et al. "A Convolutional Neural Network Cascade for Face Detection". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2015 (cit. on p. 11).

[49] L. Li, K. Ota, and M. Dong. "Deep Learning for Smart Industry: Efficient Manufacture Inspection System With Fog Computing". In: *IEEE Transactions on Industrial Informatics* 14.10 (Oct. 2018), pp. 4665–4673. ISSN: 1941-0050. DOI: 10.1109/TII.2018.2842821 (cit. on pp. 18–22).

[50] F. Liu et al. "A Survey on Edge Computing Systems and Tools". In: *Proceedings of the IEEE* 107.8 (Aug. 2019), pp. 1537–1562. ISSN: 1558-2256. DOI: 10.1109/JPROC.2019.2920341 (cit. on p. 22).

[51] D. G. Lowe. "Object recognition from local scale-invariant features". In: *Proceedings of the Seventh IEEE International Conference on Computer Vision*. Vol. 2. Sept. 1999, 1150–1157 vol.2. DOI: 10.1109/ICCV.1999.790410 (cit. on p. 12).

[52] N. C. Marques, B. Silva, and H. Santos. "An Interactive Interface for Multi-dimensional Data Stream Analysis". In: *2016 20th International Conference Information Visualisation (IV)*. July 2016, pp. 223–229. DOI: 10.1109/IV.2016.72 (cit. on pp. 16, 17, 56).

[53] S. Marsland. *Machine Learning - An Algorithmic Perspective*. Chapman and Hall / CRC machine learning and pattern recognition series. CRC Press, 2009. ISBN: 978-1-4200-6718-7. URL: http://www.crcpress.com/product/isbn/9781420067187 (cit. on pp. 7, 8).

[54] C. Messom and A. Barczak. "Fast and efficient rotated haar-like features using rotated integral images". In: *Australian Conference on Robotics and Automation*. 2006, pp. 1–6 (cit. on p. 10).

[55] *ML Kit*. Google. URL: https://developers.google.com/ml-kit (cit. on p. 11).

[56] M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of Machine Learning*. Adaptive computation and machine learning. MIT Press, 2012. ISBN: 978-0-262-01825-8. URL: http://mitpress.mit.edu/books/foundations-machine-learning-0 (cit. on pp. 7, 8).

[57] H.-L. Nguyen, Y.-K. Woon, and W.-K. Ng. "A survey on data stream clustering and classification". In: *Knowledge and Information Systems* 45.3 (Dec. 2015), pp. 535–569. ISSN: 0219-3116. DOI: 10.1007/s10115-014-0808-1. URL: https://doi.org/10.1007/s10115-014-0808-1 (cit. on pp. 15, 16).

[58] *Number of smartphone users worldwide from 2016 to 2021*. Statista. URL: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/ (cit. on p. 1).

[59] X. Ran et al. "DeepDecision: A Mobile Deep Learning Framework for Edge Video Analytics". In: *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*. Apr. 2018, pp. 1421–1429. DOI: 10.1109/INFOCOM.2018.8485905 (cit. on pp. 18–22).

[60] P. J. Rousseeuw. "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis". In: *Journal of Computational and Applied Mathematics* 20 (1987), pp. 53–65. ISSN: 0377-0427. DOI: https://doi.org/10.1016/0377-0427(87)90125-7. URL: https://www.sciencedirect.com/science/article/pii/0377042787901257 (cit. on p. 16).

[61] E. Rublee et al. "ORB: An efficient alternative to SIFT or SURF". In: *2011 International Conference on Computer Vision*. Nov. 2011, pp. 2564–2571. DOI: 10.1109/ICCV.2011.6126544 (cit. on pp. 12, 21).

[62] P. Sanches et al. "Computação Distribuída em Redes Formadas por Dispositivos Móveis". Portuguese. In: *INForum 2017 - Atas do Nono Simpósio de Informática*. Oct. 2017 (cit. on pp. 28, 29).

[63] P. Sanches et al. "Data-Centric Distributed Computing on Networks of Mobile Devices". In: *Euro-Par 2020: Parallel Processing - 26th International Conference on Parallel and Distributed Computing, Warsaw, Poland, August 24-28, 2020, Proceedings*. Ed. by M. Malawski and K. Rzadca. Vol. 12247. Lecture Notes in Computer Science. Springer, 2020, pp. 296–311. DOI: 10.1007/978-3-030-57675-2_19. URL: https://doi.org/10.1007/978-3-030-57675-2%5C_19 (cit. on pp. 5, 21, 28).

[64] M. Satyanarayanan. "The Emergence of Edge Computing". In: *IEEE Computer* 50.1 (2017), pp. 30–39. DOI: 10.1109/MC.2017.9. URL: https://doi.org/10.1109/MC.2017.9 (cit. on p. 2).

[65] J. Schneible and A. Lu. "Anomaly detection on the edge". In: *MILCOM 2017 - 2017 IEEE Military Communications Conference (MILCOM)*. Oct. 2017, pp. 678–682. DOI: 10.1109/MILCOM.2017.8170817 (cit. on pp. 18–22).

[66] M. Shapiro et al. "Conflict-Free Replicated Data Types". In: *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*. SSS'11. Grenoble, France: Springer-Verlag, 2011, pp. 386–400. ISBN: 9783642245497 (cit. on p. 25).

[67] B. Silva and N. C. Marques. "The ubiquitous self-organizing map for non-stationary data streams". In: *J. Big Data* 2 (2015), p. 27. DOI: 10.1186/s40537-015-0033-0. URL: https://doi.org/10.1186/s40537-015-0033-0 (cit. on pp. 15–17).

[68] J. A. Silva, P. Vieira, and H. Paulino. "Data Storage and Sharing for Mobile Devices in Multi-region Edge Networks". In: *21st IEEE International Symposium on "A World of Wireless, Mobile and Multimedia Networks", WoWMoM 2020, Cork, Ireland, August 31 - September 3, 2020*. IEEE, 2020, pp. 40–49. DOI: 10.1109/WoWMoM49955.2020.00021. URL: https://doi.org/10.1109/WoWMoM49955.2020.00021 (cit. on p. 26).

[69] J. A. Silva et al. "It's about Thyme: On the design and implementation of a time-aware reactive storage system for pervasive edge computing environments". In: *Future Gener. Comput. Syst.* 118 (2021), pp. 14–36. DOI: 10.1016/j.future.2020.12.008. URL: https://doi.org/10.1016/j.future.2020.12.008 (cit. on pp. 5, 21, 23).

[70] J. A. Silva et al. "Time-aware reactive storage in wireless edge environments". In: *MobiQuitous 2019, Proceedings of the 16th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services, Houston, Texas, USA, November 12-14, 2019*. 2019, pp. 238–247. DOI: 10.1145/3360774.3360828 (cit. on pp. 23, 25).

[71] J. A. Silva et al. "Data Stream Clustering: A Survey". In: *ACM Comput. Surv.* 46.1 (July 2013). ISSN: 0360-0300. DOI: 10.1145/2522968.2522981. URL: https://doi.org/10.1145/2522968.2522981 (cit. on pp. 15, 16).

[72] N. Singhai and S. K. Shandilya. "A survey on: content based image retrieval systems". In: *International Journal of Computer Applications* 4.2 (2010), pp. 22–26 (cit. on p. 11).

[73] M. A. Turk and A. P. Pentland. "Face recognition using eigenfaces". In: *Proceedings. 1991 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. 1991, pp. 586–591. DOI: 10.1109/CVPR.1991.139758 (cit. on pp. 37, 38).

[74] A. Vailaya et al. "Image classification for content-based indexing". In: *IEEE Transactions on Image Processing* 10.1 (2001), pp. 117–130. DOI: 10.1109/83.892448 (cit. on p. 3).

[75] P. Vieira. "A Persistent Publish/Subscribe System for Mobile Edge Computing". MA thesis. FCT NOVA, Nov. 2018. URL: http://hdl.handle.net/10362/71124 (cit. on pp. 25, 26).

[76] P. Viola and M. Jones. "Rapid object detection using a boosted cascade of simple features". In: *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*. Vol. 1. Dec. 2001, pp. I–I. DOI: 10.1109/CVPR.2001.990517 (cit. on p. 10).

[77] P. Viola and M. J. Jones. "Robust Real-Time Face Detection". In: *International Journal of Computer Vision* 57.2 (May 2004), pp. 137–154. ISSN: 1573-1405. DOI: 10.1023/B:VISI.0000013087.49260.fb. URL: https://doi.org/10.1023/B:VISI.0000013087.49260.fb (cit. on p. 10).

[78] *Volume of data/information created worldwide from 2010 to 2025*. Statista. URL: https://www.statista.com/statistics/871513/worldwide-data-created/ (cit. on pp. 8, 9).

[79] L. Wang and D.-C. He. "Texture classification using texture spectrum". In: *Pattern Recognition* 23.8 (1990), pp. 905–910. ISSN: 0031-3203. DOI: https://doi.org/10.1016/0031-3203(90)90135-8. URL: http://www.sciencedirect.com/science/article/pii/0031320390901358 (cit. on p. 12).

[80] S. Wold, K. Esbensen, and P. Geladi. "Principal component analysis". In: *Chemometrics and intelligent laboratory systems* 2.1-3 (1987), pp. 37–52 (cit. on pp. 37, 38).

[81] D. Xu and Y. Tian. "A Comprehensive Survey of Clustering Algorithms". In: *Annals of Data Science* 2.2 (June 2015), pp. 165–193. ISSN: 2198-5812. DOI: 10.1007/s40745-015-0040-1. URL: https://doi.org/10.1007/s40745-015-0040-1 (cit. on pp. 13, 14).

[82] Z. Zhou et al. "Edge Intelligence: Paving the Last Mile of Artificial Intelligence With Edge Computing". In: *Proceedings of the IEEE* 107.8 (Aug. 2019), pp. 1738–1762. ISSN: 1558-2256. DOI: 10.1109/JPROC.2019.2918951 (cit. on p. 22).

[83] G. Zhu et al. "Toward an Intelligent Edge: Wireless Communication Meets Machine Learning". In: *IEEE Communications Magazine* 58.1 (Jan. 2020), pp. 19–25. ISSN: 1558-1896. DOI: 10.1109/MCOM.001.1900103 (cit. on p. 19).