MyWatson: A system for interactive access of personal records

Pedro Miguel dos Santos Duarte

Thesis to obtain the Master of Science Degree in
Information Systems and Computer Engineering

Supervisor: Prof. Arlindo Manuel Limede de Oliveira

Examination Committee

Chairperson: Prof. Alberto Manuel Rodrigues da Silva
Supervisor: Prof. Arlindo Manuel Limede de Oliveira
Member of the Committee: Prof. Manuel Fernando Cabido Peres Lopes

November 2018
Acknowledgments

I would like to thank my parents for raising me, always being there for me and encouraging me to follow my goals. They are my friends as well and helped me find my way when I was lost. I would also like to thank my friends: without their continuous support throughout my academic years, I probably would have quit a long time ago. They were the main reason I lasted all these years, and are the main factor in my seemingly successful academic course.

I would also like to acknowledge my dissertation supervisor Prof. Arlindo Oliveira for the opportunity and for his insight, support, and sharing of knowledge that has made this thesis possible.

Finally, to all my colleagues that helped me grow as a person, thank you.
Abstract

With the number of photos people take growing, it's getting increasingly difficult for a common person to manage all the photos in its digital library, and finding a single specific photo in a large gallery is proving to be a challenge. In this thesis, the MyWatson system is proposed, a web application leveraging content-based image retrieval, deep learning, and clustering, with the objective of solving the image retrieval problem, focusing on the user.

MyWatson is developed on top of the Django framework, a powerful high-level Python Web framework that allows for rapid development, and revolves around automatic tag extraction and a friendly user interface that allows the user to browse its picture gallery and search for images via query by keyword. MyWatson's features include the ability to upload and automatically tag multiple photos at once using Google's Cloud Vision API, detect and group faces according to their similarity by utilizing a convolution neural network, built on top of Keras and Tensorflow, as a feature extractor, and a hierarchical clustering algorithm to generate several groups of clusters.

Besides discussing state-of-the-art techniques, presenting the utilized APIs and technologies and explaining the system's architecture with detail, a heuristic evaluation of the interface is corroborated by the results of questionnaires answered by the users. Overall, users manifested interest in the application and the need for features that help them achieve a better management of a large collection of photos.

Keywords

Content-based image retrieval; Deep learning; Clustering; Django; Face detection; Convolutional neural network; Google Cloud Vision; Keras; Tensorflow;
# Contents

1 Introduction .................................................. 1
   1.1 Content-Based Image Retrieval .......................... 3
   1.2 Problem and Objectives ................................ 6
   1.3 Contributions ............................................ 7
   1.4 Outline .................................................... 8

2 Methods and Algorithms ...................................... 9
   2.1 Image tagging ............................................. 10
      2.1.1 Low-level Features ................................. 10
      2.1.2 High-level Features ............................... 15
      2.1.3 In MyWatson ......................................... 21
   2.2 Face Detection and Face Recognition .................... 21
      2.2.1 State of the art in face detection ............... 21
      2.2.2 State of the art in face recognition ............ 25
   2.3 Convolutional Neural Networks .......................... 27
   2.4 Clustering ................................................ 31
      2.4.1 Clustering Algorithms ............................ 32
      2.4.2 Metrics .............................................. 35

3 Technologies .................................................. 39
   3.1 Django .................................................... 40
   3.2 Google Cloud Vision ..................................... 45
   3.3 Tensorflow ................................................. 47
   3.4 Keras ..................................................... 48

4 System Architecture .......................................... 51
   4.1 Overview ................................................ 55
   4.2 Django controller ....................................... 57
   4.3 Core ....................................................... 65
      4.3.1 Automatic Tagger .................................. 65
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3.2 Retrieve and Rank</td>
<td>68</td>
</tr>
<tr>
<td>4.3.3 Learning</td>
<td>69</td>
</tr>
<tr>
<td>4.4 Front-End</td>
<td>73</td>
</tr>
<tr>
<td>5 Evaluation</td>
<td>77</td>
</tr>
<tr>
<td>5.1 Heuristic Analysis</td>
<td>78</td>
</tr>
<tr>
<td>5.2 User Feedback</td>
<td>80</td>
</tr>
<tr>
<td>5.3 System Limitations</td>
<td>82</td>
</tr>
<tr>
<td>6 Conclusions</td>
<td>85</td>
</tr>
<tr>
<td>A MyWatson Application Feedback Form Sample</td>
<td>99</td>
</tr>
<tr>
<td>B MyWatson Application Feedback Results</td>
<td>103</td>
</tr>
<tr>
<td>C Automatic Tagger API Comparison</td>
<td>109</td>
</tr>
</tbody>
</table>
List of Figures

1.1 General CBIR workflow as discussed by Zhou et al. (image taken from [1]) ............... 4

2.1 Keypoint fingerprint (image taken from [2]) ...................................................... 14
2.2 Gradients to descriptor (image taken from [3]) .................................................. 14
2.3 Object-ontology (image taken from [4]) .............................................................. 16
2.4 Multi-layer perceptron (image taken from [5]) ..................................................... 17
2.5 Relevance feedback CBIR system (image taken from [6]) ................................. 18
2.6 Semantic network (image taken from [7]) .......................................................... 19
2.7 Semantic template generation (image taken from [8]) ......................................... 19
2.8 Snakes used in motion tracking (image taken from [9]) ..................................... 23
2.9 Point model of the boundary of a transistor (image taken from [10]) ................... 23
2.10 Sung and Poggio’s system (image taken from [11]) ......................................... 25
2.11 Input image and its Gabor representations (image taken from [12]) ..................... 26
2.12 Single neuron perceptron (image taken from [13]) .......................................... 28
2.13 CNN filtering (image taken from [14]) ............................................................ 29
2.14 CNN fully connected layer (image taken from [14]) ........................................ 30
2.15 Architecture of the VGG network (image taken from [15]) ............................... 30
2.16 A residual building block from ResNet (image taken from [16]) ....................... 31
2.17 K-means clustering (image taken from [17]) ..................................................... 32
2.18 Hierarchical clustering dendogram (image taken from [18]) ............................. 33
2.19 DBSCAN clusters. Notice that the outliers (in gray) are not assigned a cluster (image taken from [19]) ................................................................. 35
2.20 EM clustering on Gaussian-distributed data (image taken from [20]) .......... 36

3.1 Model-View-Controller architectural pattern (image taken from [21]) ............ 41
3.2 A model definition in Django. Note the relations with other models as foreign keys. .... 42
3.3 Template definition in Django. Note that the blocks defined are re-utilized. .......... 43
3.4 A function-based (a) and a class-based view (b). Note that both receive a request and render a specific template, and may also send additional data. .......................... 44
3.5 An example of url-view mapping in URLconf .................................................. 44
3.6 Forms in Django. Note that the view must specify which form is rendered, as it can be seen in figure 3.4(a). ................................................................. 44
3.7 Google CV API examples ................................................................................. 46
3.8 Google CV API facial recognition example ....................................................... 46
3.9 An example of a TensorFlow computation graph (image taken from [22]) .......... 48
3.10 An example showing the basic usage of keras-vggface library to extract features – by excluding the final, fully-connected layer. .................................................. 49
4.1 MyWatson’s upload page. .................................................................................. 53
4.2 Deleting the incorrect tag “sea” and adding the tag “Portugal” ......................... 54
4.3 The return results for the query “beach” .......................................................... 54
4.4 Some clusters aggregated according to face similarities. In edit mode, the cluster can be re-arranged and the names can be changed. .................................................. 55
4.5 An informal diagram picturing the architecture of the MyWatson system. .......... 56
4.6 An ER diagram picturing the data models of the MyWatson system. ................. 57
4.7 Two groups of clusters, with $n = 2$ and $n = 3$. ............................................... 61
4.8 URL dispatcher used in the MyWatson application ........................................... 63
4.9 Redirecting to another view ............................................................................. 63
4.10 Redirecting to another view ............................................................................ 64
4.11 Accessing objects from the view in the template .............................................. 64
4.12 Delegating work to the MyWatson core ......................................................... 64
4.13 Creating a new tag .......................................................................................... 65
4.14 Cluster groups score ...................................................................................... 72
4.15 MyWatson landing page .................................................................................. 74
5.1 Google Cloud Vision API label limitations ...................................................... 82
5.2 Google CV API noise experiment .................................................................... 83

B.1 Bar chart with user’s ages ............................................................................. 104
B.2 Bar chart with how easy users consider MyWatson is to use ......................... 104
B.3 Bar chart with how often users would use MyWatson ..................................... 105
B.4 Pie chart with the most useful feature according to users ............................... 105
B.5 Bar chart with the users’ opinion about the UI .............................................. 106
B.6 Bar chart with how useful the users consider MyWatson ............................... 106
B.7 Pie chart with the likelihood users would recommend MyWatson . . . . . . . . . . . . . . 107

C.1 The images utilized in the comparison . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109

C.2 An informal comparison of different Automatic Tagger APIs . . . . . . . . . . . . . . . . . 110
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Django model fields</td>
<td>58</td>
</tr>
<tr>
<td>5.1</td>
<td>Heuristic evaluation of MyWatson's UI</td>
<td>78</td>
</tr>
</tbody>
</table>
Acronyms

**AJAX**  Asynchronous JavaScript And XML

**API**  Application Programming Interface

**BIRCH**  Balanced Iterative Reducing and Clustering using Hierarchies

**CBIR**  Content-Based Image Retrieval

**CLARA**  Clustering LARge Applications

**CLARANS**  Clustering LARge Applications based on RANdomized Search

**CNS**  Color Naming System

**CTA**  Call to Action

**DBSCAN**  Density-based Spatial Clustering of Applications with Noise

**DoG**  Difference of Gaussians

**DRY**  Don’t Repeat Yourself

**EM**  Expectation-Maximization

**ERD**  Entity Relationship Diagram

**FA**  Factor Analysis

**HTML**  Hyper Text Markup Language

**HMM**  Hidden Markov Model

**ILSVRC**  ImageNet Large Scale Visual Recognition Challenge

**JSON**  JavaScript Object Notation

**LDA**  Linear Discriminant Analysis
LFW  Labeled Faces in the Wild
LoG  Laplacian of Gaussians
MNIST  Modified National Institute of Standards and Technology
MVC  Model-View-Controller
MVT  Model-View-Template
NIST  National Institute of Standards and Technology
OCR  Optic Character Recognition
OOP  Object-Oriented Programming
ORM  Object-Relational-Mapping
PAM  Partitioning Around Medoids
PBKDF2  Password-Based Key Derivation Function 2
PCA  Principal Component Analysis
PDM  Point Distributed Model
REST  REpresentational State Transfer
SIFT  Scale-Invariant Feature Transform
SURF  Speeded Up Robust Features
ST  Semantic Template
STING  STatistical INformation Grid
SVM  Support Vector Machine
SVT  Semantic Visual Template
VGG  Visual Geometry Group
VLAD  Vector of Locally Aggregated Descriptors
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Content-Based Image Retrieval</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Problem and Objectives</td>
<td>6</td>
</tr>
<tr>
<td>1.3 Contributions</td>
<td>7</td>
</tr>
<tr>
<td>1.4 Outline</td>
<td>8</td>
</tr>
</tbody>
</table>
With a third of the world population owning a smartphone in 2017 and with that percentage estimated to rise to 40% until 2021 [23], as well as that percentage being around 65% in both Western Europe and North America (in 2017), it is safe to say that a lot of people have smartphones.

This popularity of smartphones caused a boom in the number of digital pictures in the world [24]. This is because, although the integrated cameras in smartphones are not comparable to DSLR cameras, they have improved a lot, and are a lot more practical to use and carry than the bulky DSLR cameras [25]. This means that a lot more “common” people, not just professional photographers, take pictures: some to post them on social media platforms, others just to keep to themselves. The rise of social media platforms like Facebook and Instagram also seem to have helped the rise of smartphone cameras ( [24]) and therefore the number of pictures a common person takes.

More and more people are driven by the desire of showing the world their lifestyle and sharing their life with the rest of the world through social media and this, of course, also includes photos and other types of media. People actively post what they do, who they are with and where they are in a pursuit of social media popularity, which is translated in the number of followers one has or “likes” a photo gets, for example. To get followers and likes, however, one’s profile needs to be accessible and easily found, i.e. they must appear on the most relevant results of a certain query within the social media platform. The easiest way to do this is by tagging the photos, e.g. hashtags on social networks, so the photos are found when another user searches for that tag. Of course, this means that the greater the number of tags a photo has, the more queries it appears in, and the greater the chance of another user discovering the photo, which may translate in a “like” or an additional follower and, therefore, more popularity. Often, the tags may not even be related to the content of the photo itself, as in this social media setting, quantity often triumphs quality (for the tags).

The assignment of tags is often boring and slow, especially for a large number of tags. The process of manually assigning tags is motivated, in this social media setting, by the pursuit of popularity, as explained by Ames [26]. However, in a more personal environment, e.g. a person that likes to take photographs but does not want to show them to the public, tagging is also important to make the photos searchable and manageable, and this is a crucial necessity once a collection of photos starts getting bigger and bigger. In spite of thinking that tagging is useful and a necessity, people still do not bother with manual tagging (as presented by Ames) because the cons largely outweigh the pros: it is a very boring process, it takes a lot of time and the tagging usually needs to be exhaustive in order to cover all the content in the photo.

Ultimately, what the user wants is to retrieve their photos via query. There are two main techniques for retrieving photos this way: Content-Based Image Retrieval and Text-Based Image Retrieval. Content-Based Image Retrieval approaches focus on “reading” the actual content of the image and extracting information (features) that can later be checked to see if it matches the user’s query. Content-Based
Image Retrieval (CBIR) will be discussed in more detail in the next sub-section.

On the other hand, Text-Based Image Retrieval approaches completely ignores the content of the image itself, and tries to find useful information in the “textual context” of the image, i.e. the meta-data of a image on the web, such as its title, caption or description, or other text present on the same page as the image. This type of approach has many shortcomings, as pointed out by Liu et al. [6], the main ones being the dependency on the text surrounding the image and the need for manual work. Additionally, because the process of describing a picture is done by humans, the descriptions are prone to human subjectivity, i.e. two different people would probably describe it differently. Furthermore, it is very unlikely that the description would be fully extensive and cover every single aspect of the content of the image. Finally, there would be no guarantees that the text around an image would accurately describe it or even be about the image.

Due to the fact that text-based approaches still requires manual work, they will not be further discussed in this document. In the next sub-section, I will further discuss the CBIR approach, its shortcomings and the state of the art.

### 1.1 Content-Based Image Retrieval

Eakins [27] defines CBIR as the “process of retrieving desired images from a large collection on the basis of features (such as colors, texture and shape) that can be automatically extracted from the images themselves”. Note that the word “automatically” plays a huge role in the definition – if the feature extraction is not automatic, then the retrieval is not CBIR, even if the keywords describe the content of the image –, as explained by Eakins.

The general flow of a CBIR process has an online and an offline phase, as presented by Zhou et al. [1]. The offline phase is where the preparation for the user query is made, by keeping a database of images, representing the images, and indexing those representations. When the user wants to find an image, the online phase begins, i.e. when the user formulates his intention and forms the query that will be introduced in the retrieval system. After that, the system translates the user query to a query vector that is of the same type as the features vector of the offline phase, and scores the images in the database using a scoring function – usually, a relevance or similarity function. Finally, the images are retrieved and ranked accordingly, e.g. by relevance to the query.

In this thesis, the image crawling, image database and retrieval browsing are not very important, as they are very linear and simple with almost no challenge. Therefore, the focus will be on the brief discussion of the other steps.

Because the offline stage is a requirement so that the user can begin to utilize a system relying on CBIR, it will be the first stage to be discussed. The most meaningful step in this stage is the image
representation step, which is also present in the online stage. It is assumed that a database with images is already present, e.g. a server has some images on disk.

The goal of the image representation step is to provide a more feasible way to compare two images than with a pixel-by-pixel comparison. This comparison is necessary to check if a certain image in the database is related or similar with another image, being a query by the user (an image query – types of query will be further discussed) or a database image. However, a pixel-by-pixel comparison is not feasible mainly because there are many pixels in an image, and therefore it would take a lot of computational power and time to do a single comparison between two images. There is also the problem that even if the two images being compared had the exact same objects, if they suffered a transformation, i.e. a rotation, a translation or a distortion, they would not be considered similar using this method. The solution encompasses translating the images into smaller representations that can be more easily compared: each image is represented by a vector of features with a fixed size, as discussed by Zhou et al. [1]. With this, the similarity of two images can be easily calculated by a distance function between two vectors, for example.

To build the image representations, the features must first be extracted from the images. But how does a computer looks at an image of an apple and understands and labels it as the object “apple”, if what it sees are just pixels with RGB values? This gap between the RGB values that the computers “sees” and the actual name of the objects is called the semantic gap, i.e. the gap between low-level features – RGB values, but the term also includes colors, shapes and textures and will be used more commonly for the latter group – and high-level features – which are, ultimately, the tags on an image. In short, to output tags based on the content of a given image, computers need to extract low-level features and “understand” them so that they can output a human-readable tag.

Low-level features – which will henceforth be just called features, or descriptors, unless the distinction is not obvious – can be local or global. Local features are mainly based on detecting keypoints and regions of interest of an object in an image. Scale-Invariant Feature Transform (SIFT) by Lowe [3] is an example of such descriptor, being invariant to transformations such as scale and rotation. On the other hand, global features are extracted from the entire image, such as the color histogram or a collection of
shapes or structures. An example of this is the GIST feature [28], in which the main idea is to capture the gist of a scene into a low-dimensional vector.

Features can be categorized into hand-crafted and learning-based features, according to the way they are extracted. Hand-crafted features, such as the SIFT feature by Lowe [3], are extracted by following a specific algorithm to apply “filters” that detects points of interest and regions. Detectors include the Difference of Gaussians (DoG) [3] or Maximally Stable Extremal Regions (MSER) detector [29]. On the other hand, learning-based features are extracted via machine-learning algorithms, i.e. there is no specific algorithm to extract features with filters – the filters are learned and applied automatically. Deep Learning algorithms are the most used, where a perceptron with multiple layers is used to learn the features to be extracted using previously annotated data by trying to predict and learning from the error. Particularly, Convolutional Neural Networks (CNNs) have been proven to be very good at extracting features and classifying images [30], with the advantage that they can be trained to classify a given set of object categories and the features learned can also be used to classify another completely different set of categories with good results. CNNs will be further discussed along this thesis.

After the low-level features are extracted, high-level features (tags) need to be computed. Both vectors of features can be indexed in a database: either a vector full of numbers (representing the low-level features, colors, shapes and such) or a vector of tags (a Bag of Words model, from the standard Text-Based Information Retrieval area). With CNNs, the feature extraction and the tag computation are done in the “same” step. Machine learning methods can also be utilized where hand-crafted features are extracted, as a means of closing the semantic gap.

Usually, indexing the features in the database is solved by borrowing another technique from Text-Based Image Retrieval: the inverted-index, where a feature (low-level or high-level) is mapped to the image that contains that feature.

With the database complete, the user can now query the system and retrieve photos. The queries that the user can do depend on what the system allows, and the queries are split into five different categories according to Zhou et al. [1]:

- **Query by keyword**: In a query by keyword, the user introduces one or more keywords that describe the content of the images he wants to retrieve. Eakins also sub-categorizes this type of query by levels [27], according to its semantic level, from level 1 to level 3. This is the most common query, and it is the most simple one for a user to execute. However, sometimes it is hard for a user to formulate his intention just by keywords.

  A level 1 query comprises primitive elements of an image, e.g. “find pictures with a large uniform blue region on top”. However, this type of query level can also be found in the next categories, i.e. by asking the system to “find pictures like this”, introducing an example picture or of a layout – except if the system is required to have knowledge of the names of real life entities.
A level 2 query jumps the semantic gap, because here the system already needs to have at least a minimal understanding of objects and their names. This query includes types of objects (“find pictures of a person”) or even individual objects or entities (“find pictures of Obama”).

Finally, a level 3 query is comprised of abstract concepts, such as activities (e.g. “people dancing”) or emotional or religious significance (e.g. “happy people”). The last two levels are classified as semantic image retrieval.

- **Query by example:** In a query by example, the user introduces one or multiple example images that he wants to find pictures similar to. The system then tries to find images that are closely related or similar in terms of color, shapes, textures or layout. This query type – and the following that are similar – has the inconvenience of requiring the user to have an example image at hand to introduce.

- **Query by sketch:** A query by sketch is very similar to the previous one, but instead the user submits a drawn sketch of its intention.

- **Query by color layout:** In a query by color layout, the user introduces a layout of the colors and their locations of the intended images. The layout is usually a simple scheme of names of colors separated by lines to indicate zones of colors. For example, a scheme of “blue” on top and “green” on the bottom separated by a line would indicate that the user wants to retrieve images of green fields with a clear sky.

- **Query by concept layout:** Very similar to the color layout query, but instead of colors the user introduces names of objects or entities.

After the user translates his intention into a query of one of the discussed categories, the query must be transformed into the same type as the feature vectors in the database, i.e. features must also be extracted from the query. If the query is an example image, the features are extracted in the same way as the images in the database. On the other hand, if the query is a set of keywords, the query vector is already specified, minus a few details that will be further discussed, such as weights and expansion of the terms in the query.

Finally, the relevance score is computed for each feature vector in the database, ordered according to their relevance to the query and the top relevant ones are returned to the user.

### 1.2 Problem and Objectives

The large influx of photos previously discussed raises the problem of how to adequately manage large collections of personal photos. An adequate solution to this problem would need to focus on the man-
agement of a user’s gallery and should have the following objectives and requirements:

- **User non-involvement**: the user should not need to be involved in the automatic tagging process – the system must produce automatic tags for all photos without needing to bother the user.

- **User controllable**: in spite of the previous point, the system must allow the user to add or remove tags that he thinks do not adequately describe the content of an image.

- **Storage and accessibility**: the system should allow the user to store and access photos anytime and anywhere and, of course, the ability to query the system.

- **Gallery management**: the system must offer additional functionality to facilitate the management of the user’s photo gallery, apart from the ability to query the system.

- **Easy and simple**: the user interface should be simple and easy to use, as this complementary system has the goal of saving the time of not having to tag and search for images manually, and not to waste the user’s time.

Therefore, there are some challenges to overcome. The first challenge is to solve the automatic tagging of the photos. In parallel with the CBIR framework previously addressed, the most challenging part here is how to extract the features of an image or query.

The second challenge corresponds to making a user-friendly and accessible interface in order to facilitate the life of the user.

Finally, the last challenge is what can be done to further enhance the user experience when managing photos, i.e. finding a way to help the user find the information he needs faster. This thesis proposes utilizing facial detection for face clustering, by classifying face images in a CNN pre-trained on a large face dataset and then removing the last classification layer in order to extract facial features.

### 1.3 Contributions

The main contribution of this thesis falls on the general area of CBIR. By focusing on the automatic tagging of photos, I will be testing the usefulness of a system that eases the process of managing a photo gallery without having to waste a potential user’s time and effort. Furthermore, machine learning also plays a very important role in this goal: by utilizing a CNN which was pre-trained to classify faces and cutting off the final classification layer, I will be able to recognize faces in photos and cluster them – by utilizing the output from the feature extraction layer as features for the clustering –, increasing the user’s available information, which further enhances the ability to manage a photo gallery.
Implementing a web interface is also a contribution in itself, as it should be interesting to see the usefulness that an easily accessible system may have on a user’s habit, i.e. how often would a user use the system to manage or look through his photo gallery.

The entirety of the code implementing the MyWatson system can be found at the following GitHub repository: https://github.com/psnint/MyWatson.

1.4 Outline

After this introductory chapter briefly explaining some CBIR general terms and techniques, challenges, objectives and contributions, the thesis will cover the algorithms and technologies used, as well as the system architecture and evaluation, in the following chapters:

**Chapter 2** The methods and algorithms used in the MyWatson system will be discussed. There is a section for each important module of the solution, namely the automatic tagging, the facial recognition, convolutional neural networks and clustering. The objective is to provide a general understanding of the techniques used.

**Chapter 3** Discusses the technologies used in the development of the MyWatson system. Focuses on the frameworks and not the algorithms themselves, but also on the Application Programming Interface (API) utilized.

**Chapter 4** Provides an overview of the system architecture, i.e. how its modules are connected and how common use cases are processed, such as the automatic tagging of photos or the clustering by faces.

**Chapter 5** This chapter provides a heuristic evaluation of the user interface and discusses the results of a system evaluation done by real users, as well as possible reasons for those results and a comparison between the heuristic evaluation and these results.

**Chapter 6** In the final chapter, conclusions will be drawn from the current status and evaluation of MyWatson, as well as future improvements and work that can be done.
## 2 Methods and Algorithms

### Contents

2.1 Image tagging .................................................. 10
   2.1.1 Low-level Features ........................................ 10
   2.1.2 High-level Features ....................................... 15
   2.1.3 In MyWatson ............................................. 21

2.2 Face Detection and Face Recognition .......................... 21
   2.2.1 State of the art in face detection ....................... 21
   2.2.2 State of the art in face recognition .................... 25

2.3 Convolutional Neural Networks ................................. 27

2.4 Clustering ..................................................... 31
   2.4.1 Clustering Algorithms ..................................... 32
   2.4.2 Metrics .................................................. 35
This chapter will discuss the algorithms used to solve some of the CBIR problems, as well as those used in the MyWatson system in general. This includes the algorithms and methods used on the image tagging and face detection, as well as in the clustering of faces, which first requires the facial features extraction through a CNN. The discussion will include a general idea of how they work and what they do, as well as some of their uses and, finally, how they were implemented in the MyWatson system.

2.1 Image tagging

The most important aspect of the MyWatson system is the automatic image tagging itself. It is the module that does most of the work and so it is fitting that it gets to be the first one to be discussed. Although a brief explanation was given on CBIR in the introductory chapter 1, it is such a complex field that further detail is needed, at least on the most critical parts: low-level feature extraction and high-level feature extraction.

2.1.1 Low-level Features

The low-level feature extraction is the core of a CBIR system: it's how a computer sees an image and extracts any kind of low-level information from it, which allows for further refinement into high-level semantic objects and concepts. Currently, low-level features can be extracted in a manual or an automated manner, and manually extracted features can be local or global features, depending on the focus of the algorithm.

Manual or hand-crafted features are not extracted manually by a human, despite its name – instead, they are derived via an algorithm which was specially engineered (by humans) to extract those specific features. This is the difference versus automatically extracted features: in the latter, it is not known what type of features are extracted, because they are automatically learned. Of this type, the most common techniques nowadays include the usage of neural networks with deep architectures to learn the extraction process with high-level abstractions close to those used by humans, which will be further discussed in section 2.3. Therefore, the remainder of this section will be focused on discussing manually engineered processes.

The main difference between global and local features is that the resulting output of extracting global features is a vector that describes the entire image, i.e. it's a point in an high-dimensional space and it is very easy to compare with other images. On the other hand, the output of extracting local features is a vector of vectors of regions, and some sort of aggregation is needed to compact those vectors into a single, fixed size vector. In short: global features describe the whole image in terms of colors, shapes and textures, and the local features describe multiple regions that belong to an image.
Global features are attractive, because they are very compact in terms of representation, and as such any standard classifier can be utilized, as discussed by Lisin et al. [31], for example to produce high-level features. However, they are often not directly related to any high-level semantics [32]. Global features are also very sensitive to background clutter, so they should mainly be utilized in images that only have one object in a front-plane, or with a good and clear segmentation [31]. Examples of global feature extraction techniques include computing a color histogram — which, in short, tells the percentage of each color in an image —, such as the bag-of-colors of Wengert et al. [33], or GIST [28] which represents a scene with a *Spatial Envelope* — i.e. the shape of the space that is present in a certain image — using perceptual dimensions such as the naturalness, openness, roughness, expansion and ruggedness of an image.

On the other hand, local feature focus on keypoints, the salient image patches that contain the rich local information of an image [32]. Generally, there are two steps in local feature extraction: the keypoint detection and the region description. A feature is comprised of a keypoint and a descriptor: the former is the location of a certain region or interest point in an image, and the latter is some set of values that describe the image patch around the keypoint. Descriptors should be independent from the keypoint position, and should also be independent from scale and transformations. Keypoints are detected such as that even with some kind of change — a scaling or rotation, for example — they can still be identified. Usually, some kind of detector is used, and one such example is the Difference of Gaussians (DoG) [3] detector. The DoG is used for edge detection by subtracting two Gaussians with different standard deviations and detecting where the intensity of the image changes abruptly (hence detecting contours or edges). The very basic idea is to blur the same image twice, with different blur intensities, and capture edges that are simplified by one and preserved by the other by subtracting them. One other such detector is the MSER (Maximally Stable Extremal Regions) detector by Matas et al. [29]. The general idea of this detector is to try multiple intensity thresholds in an image, and select the regions that maintain unchanged shapes over a large set of thresholds (on several gray-scale levels, from completely white to completely black, and vice-versa) — usually connected areas with almost uniform intensity, surrounded by contrasting background. The FAST detector [34] by Rosten et al. is also an example of a local keypoint detector — in this case, it is a corner detector. For each pixel, it tries to see if it is a corner or not by checking within a circle of 16 pixels if there are at least N contiguous pixels (usually N = 12) which are all brighter or darker (within a given threshold). This is a high-speed test, as only some pixels are checked in the negative case.

With the interest points detected, descriptors of the area around those points should also be extracted. They should be invariant to transformations, whether positional or illumination, for example. They also should be distinctive enough so as to match a single feature with high probability against a lot of features [1]. The most popular choice of descriptor is the SIFT, as in 2004 Lowe [3] solved the
problem of extracting keypoints and descriptors independent from scale. SIFT is a patented algorithm which extracts both keypoints and descriptors. The general idea of the algorithm is as follows [2]:

1. **Create scale space:** take the original image and generate progressively blurred images. Then, resize the original image to half the size, and generate blurred out images again. Images of the same size form an octave, i.e. images are progressively blurred within an octave:

   \[ L(x, y, \sigma) = G(x, y, \sigma) \ast I(x, y) \]

   The blurred image \( L \) with a scale factor \( \sigma \) is equal to the original image \( I \) with a Gaussian blur with scale factor \( \sigma \) applied to it. The actual Gaussian blur operator is

   \[ G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2} \]

   Usually, a constant \( k \) is chosen and multiplied to \( \sigma \) for each octave. The set of octaves forms the scale space of the image.

2. **Difference of Gaussians:** then, the DoG is used as a scale-invariant alternative to the Laplacian of Gaussians (LoG), which is more computationally expensive and is not invariant to scale. This means that the second order derivative – which is the Laplacian – are dependant on the scale (\( \sigma \)), whereas the Difference is not, because we get rid of the factor \( \sigma^2 \) and then multiply with it again. For each octaves, and within each octaves, the “adjacent” gaussians from the previous step are substracted and form a DoG.

3. **Finding keypoints:** maxima and minima are located in the DoG images by, for each DoG image – except the first and last ones – within an octave, looking at each pixel and, for that image and its adjacent DoG images, checking its neighbours. This means that, in the worst case, 26 pixels are checked (9 x 3 - 1), but usually only a few checks are made before discarding the pixel as a maximum or minimum. Then, if it is a maximum or minimum, it is marked as a keypoint. However, these are just approximate, as usually the maxima and minima are on subpixels. These subpixels are found using the Taylor expansion of the image around the keypoint by differentiating it and equating it to zero, i.e. finding its extremes:

   \[ D(x) = D + \frac{\partial D^T}{\partial x} x + \frac{1}{2} x^T \frac{\partial^2 D}{\partial x^2} x \]

   This improves the stability of the algorithm and matching.

4. **Reducing keypoints:** because the previous step generates a lot of keypoints, and some of them are not useful as features – either they do not have enough contrast, or lie along an edge –, in this
step, low contrast keypoints and edges are removed. To remove low contrast points, for each pixel in the DoG image that is being checked for maxima/minima, if the magnitude of the intensity is less than a threshold, it gets rejected. Here, the intensity of the subpixels is also checked, again using the Taylor expansion. To remove the edges and keep just the corners, SIFT borrows the method from the Harris Corner Detector [35], using an Hessian matrix. This matrix is used to calculate two gradients perpendicular to each other around a keypoint, and find out if both the gradients are big enough. If they are, they are considered a corner and are accepted as a keypoint. Otherwise, they are either a flat area or an edge, and are rejected.

5. **Orientation:** from the last step, we now have scale invariant keypoints. To have rotation invariance, we now have to assign orientation to each keypoint. The general idea is to collect gradient directions and magnitudes around each keypoint. Then, the orientation with most magnitude is assigned to that point. For each pixel of an image $L(x, y)$ with a given scale, the magnitude of the gradient $m(x, y)$ and orientation $\theta(x, y)$ are calculated in the following way, respectively:

$$
m(x, y) = \sqrt{(L(x + 1, y) - L(x - 1, y))^2 + (L(x, y + 1) - L(x, y - 1))^2}
$$

$$
\theta(x, y) = \tan^{-1}\left(\frac{(L(x, y + 1) - L(x, y - 1))}{(L(x + 1, y) - L(x - 1, y))}\right)
$$

After the magnitude and orientation are calculated for all pixels around the keypoint, an histogram is created. This orientation histogram has 36 bins, covering the 360 degree range of orientations, and each sample added is proportional by its gradient magnitude. Then, the keypoint is assigned the orientation from the bin of the peak of the histogram. Also, orientations with 80% of the peak of the histogram are also converted into a keypoint with that orientation and the same scale and position as the original one – i.e. keypoints with the same position and scale can be split according to their orientation.

6. **The descriptor:** the idea of this step is to create a fingerprint to distinguish keypoints from each other by making a $16 \times 16$ with the appropriate scale window around the keypoint. Then, this window is split into $16 \times 4 \times 4$ windows, as shown in figure 2.1.

Within each $4 \times 4$ window, the gradient magnitudes and orientations are calculated and put into an histogram with 8 bins. The values that are added to the bin, however, are now weighted according to the distance between the gradient and the keypoint, using a Gaussian circle – which can be seen in blue in figure 2.2. After doing this to all $16 \times 4 \times 4$ windows, the feature vector is formed by the 128 normalized numbers from the $4 \times 4 \times 8$ values. Then, to achieve rotation independence, the keypoint’s rotation is subtracted from each of the descriptors’ orientation, so that each gradient orientation is relative to the keypoint’s orientation. Finally, to achieve illumination independence,
any number in the feature vector that is greater than 0.2 is changed to 0.2, and the vector is normalized again.

The Speeded Up Robust Features (SURF), presented by Bay et al. [36], is a more robust and faster alternative to SIFT. Instead of using the DoG to approximate the LoG, SURF utilizes the box filter, the advantages of this being that the convolution box for the box filter can be easily calculated with the help of integral images, and it can be done in parallel for different scales. Other improvements include the feature matching and orientation assignment.
To further maximize the performance of the feature matching, similar features can also be clustered together. As a Bag-of-(Visual)-Words, a codebook (analogous to a text dictionary) can be generated using a clustering algorithm over the vectors: the number of codewords will then be the same as the number of clusters.

Other techniques include adding more information to the codebook, called Spatial Context Embedding [37]. Examples of this technique include adding visual phrases (analogous to text phrases) as presented by Zhang et al. [38].

When a representation is achieved by a set of local features instead of a single vector, some kind of aggregation is needed for a more convenient similarity checking. One such technique is the Bag-of-Visual-Words representation, which assigns each local feature to the closest visual word of a codebook [1], which results in a vector with the same size as the visual codebook. Other such approach is the Vector of Locally Aggregated Descriptors (VLAD) as presented by Jégou et al. [39], which aggregates the differences between the vectors and the visual word into a single vector, characterizing the distribution of vectors with respect to the center of the cluster.

2.1.2 High-level Features

With the finality to compare two images, i.e. when the user introduces an example image as a query, the feature vectors are compared using the Euclidean distance measure. However, when the user inputs a query using high-level keywords, how to relate those words with a vector of numbers? The previously discussed semantic gap requires an higher level alternative representation for the indexed images.

Liu et al. [6] discusses five state-of-the-art techniques to bridge the semantic gap and implement high-level semantic image retrieval. These techniques can also be combined, complementing each other.

- **Object-ontology** The basic idea of an object ontology system is to map low-level features to high-level keywords based on our knowledge. For example, we know that an “uniform, light blue region on the upper part of an image” is (likely) the “sky”. These are intermediate-level descriptors and are called the object-ontology which define high-level semantic concepts. The system proposed by Mezaris et al. [4] is an example of such technique. High-level keywords are defined in function of these intermediate-level descriptors, which are themselves defined by an ontology.

An ontology includes definitions, representations, categories, properties and relations of concepts and entities in one or more domains. An example of an ontology is shown in figure 2.3, which is the object-ontology used in the system by Mezaris. The keywords are first translated into intermediate-level descriptors, and are either already defined in the ontology or must be manually translated. The low-level descriptors are automatically mapped to the intermediate-level ones by quantizing
the color, texture, position and size of a certain region.

Quantizing is how millions of colors that can be defined in a computer are also able to be named by users, and is also used to describe textures. The Color Naming System (CNS) proposed by Berk et al. [40] quantizes the HSL (Hue, Saturation and Lightness) space into 627 distinct colors. On the other hand, texture naming is a lot more difficult to standardize. Rao and Lohset identified the three most significant dimensions of textures [41], namely the repetitiveness, directionality and complexity. Chiu et al. [42] quantize Tamura features – coarseness, contrast, directionality, line-likeness, regularity and roughness – into different levels, such as very coarse, high contrast, very fine, etc., using logical operators such as “and” or “or”.

- **Machine learning** Learning is also a common technique that is used to either predict semantic concepts based on a training set of previously annotated images, in the case of supervised learning, or to understand how data is clustered, in the case of unsupervised learning.

Examples of supervised learning include utilizing an Support Vector Machine (SVM) that learns labels for regions by utilizing labeled image regions as the training input, as seen [43] by Feng and Chua. The general idea of an SVM is to find an optimal separating plane (OSP) that divides a set of data points into two classes so that the margin between the nearest data point of each class is maximum. Because an SVM is a binary classifier, one SVM must be trained for each high-level concept, i.e. for each concept, the outputs of the classifier will be \( y \in \{-1, 1\} \), where 1 belongs to that concept class and -1 does not. Other techniques include the use of neural networks (Town and Sinclair [44]) to automatically establish the link between low-level features and a set of categories. Multi-layer perceptrons (MLP) are a common neural network class with at least one hidden layer which uses backpropagation for training and can distinguish data which is not linearly separable, i.e. it uses a non-linear activation function.
The main downside of using supervised learning is the required amount of data to train a classifier. On the other hand, unsupervised learning does not need any previously annotated data, and has the objective of grouping similar data points together – for example, image regions or features – so that it maximizes the similarity between points within a cluster and minimizes the similarity between points in different clusters. One of the most common techniques for clustering is the $k$-means clustering. In this algorithm, $k$ centroids – which define a cluster – are randomly initialized in the data space, and each data point is assigned to the nearest centroid, based on the squared Euclidean distance measure. Then, the centroids’ position are recalculated by taking the mean of all data points assigned to the centroid. The algorithm repeats these two steps (minus the initialization) until it converges, i.e. no data points change cluster. The system proposed by Stan [45] utilizes $k$-means clustering as a way to find patterns in the low-level feature space – namely, colors – of the image, using a non-Euclidean distance function instead. Then, a mapping between clusters and keywords is derived using statistics measuring the variation within each cluster as IF-THEN rules. Another example is the system proposed by Jin et al. [46] which employs a semi-naive Bayes model to compute the posterior probability of concepts given region clusters, after these clusters have been determined using PCK-means (pair-wise constraints $k$-means). As clustering plays a huge role in the MyWatson application, section 2.4 will further discuss this topic.

- **Relevance feedback** These techniques differs from the previously discussed ones since it tries to learn the user’s intentions by asking him for feedback directly – it is, therefore, an on-line processing algorithm, as opposed to the other ones which are off-line. The general idea of a Relevance Feedback (RF) algorithm, as discussed by Zhou and Huang [47], is to provide some initial query results, then wait for the user feedback on whether those results are relevant to the query or not.
– usually, the user marks them relevant/irrelevant – and finally apply a machine learning algorithm to learn the user’s feedback. These steps are repeated until the user is satisfied with the results, as pictured in figure 2.5.

Figure 2.5: Relevance feedback CBIR system (image taken from [6])

Rui et al. [48] proposed a RF-based interactive approach and experimentally showed that the proposed approach greatly reduces the user’s efforts of composing a query and capture the user’s need more precisely, as opposed to systems that are computer-centric and require the user to describe high-level concepts using weights and low-level features. Re-weighting the low-level features from both the query and the images is a very typical approach to the last step of the RF loop, which removes the burden of assigning weights from the user. Other techniques include the query-point movement (QPM) [49] and [50], which improves the estimation of the query point by moving it towards positive examples and away from negative examples, or using machine learning methods such as the SVM which corresponding effectiveness is demonstrated by Jing et al. [50] or decision-tree algorithms such as the system proposed by MacArthur et al. [51].

RF techniques work well when low-level feature vectors describe the query well, as these techniques do not directly address the semantic content of images but rather their low-level features. However, some concepts cannot be sufficiently described by such low-level features, even with a lot of feedback loops. To address this problem, the framework proposed by Lu et al. [7] builds a semantic network using RF and a simple voting scheme to update the links in the network, on top of using RF in low-level features.
Overall, the main disadvantage of this technique is the requirement of user interaction.

- **Semantic template** A semantic template is a mapping between high-level concepts and low-level features, and is usually defined as the "representative" feature of a concept calculated from a collection of sample images.

Chang et al. introduced the Semantic Visual Template (SVT) [52] for bridging the semantic gap in video queries. The SVT is defined as a set of icons or example scenes/objects – from which feature vectors are extracted in for query process – that represents the semantic that the template is associated with. The users need to define a certain time and space in the video that contains a certain object, as well as the features weights, which is treated as an initial query scenario. Through interacting with the users, the system converges to a small set of example queries that best match – i.e. maximizes the recall – the user's intention.

However, common users would have a hard time defining these templates, as it requires an in-depth understanding of image features. An automatic alternative to this system was introduced by Zhuang et al. [8]. In this system, the generation of the Semantic Template (ST) is integrated with the RF loop, as pictured in figure 2.7 and its generation is imperceptible to the user, which means anyone can use it.
When an user submits an annotated image as a query example, he also submits a concept represented by the image. The system then returns an initial list of images based on the initial weights and visual feature vector. The user inputs the relevance of each of the retrieval image. The weights are updated based on the relevance feedback, and after some iterations, the most relevant images are returned to the user. Finally, the system calculates a centroid of the most relevant and treats it as the representation of that semantic concept. The semantic template is thus created, defined as $ST = \{C, F, W\}$, where $C$ is the initial query concept given by the user, $F$ is the feature vector corresponding to $C$ and $W$ is the weight vector of $F$.

In this system, WordNet [53] is used to construct semantic associations. WordNet has concepts, synsets – synonym sets, i.e. each word has different meanings – and relations, such as IS-A, PART-OF, etc. WordNet also provides associations between concepts. For example, the words "man" and "gentleman" are linked by an IS-A relation, "kill" and "die" are linked by a causal relation, and "beautiful" and "pretty", "attractive", "charming", etc. are "linked" by a synonym relation.

When a user introduces a keyword as a query, the system finds a list of all the semantic associations in WordNet that have distance smaller than a threshold $h$. Thus, a list of ordered terms is obtained according to the distance measure. Finally, for every term in the list, the system finds a corresponding ST and uses the corresponding feature vector $F$ and weights $W$ to find similar images.

- **Web image retrieval** Liu et al. [6] classify web image retrieval as one of the state-of-the-art techniques in semantic image retrieval rather than a specific application domain, as it has some technical differences from image retrieval in other applications.

Usually, only textual clues about the image are used, for example, title information, text surrounding the picture, Hyper Text Markup Language (HTML) tags, hierarchical structure of links, etc. This information, however, is usually not good enough. To solve this, visual evidences are put together with textual evidences to improve this type of retrieval. An example of such technique is the system proposed by Feng et al. [54], which employs machine learning approaches to automatically annotate web images based on a predefined list of concepts with evidences from associated HTML elements and visual contents. Co-training is utilized to mitigate the common supervised learning problem of needing a large set of training samples. Two orthogonal classifiers – one for textual evidence, other for visual evidence – are used to successively annotate a large set of unlabeled images starting from a small set of labeled training images.


2.1.3 In MyWatson

Because MyWatson utilizes the Google Cloud Vision API [55], which is a closed source API, no further discussion on the actual algorithm that is used to automatically tag the images can be made. However, the previous discussion on this topic should be enough to get an understanding on the current state-of-the-art techniques that are used to perform automatic tagging, with the exception of the CNNs which will be further discussed in section 2.3. More details on the Google Cloud Vision API will also be discussed in section 3.2.

2.2 Face Detection and Face Recognition

Facial recognition plays a major role in the MyWatson system, as it is a powerful tool to help users manage their photography collection, mainly when other people appear. It could be used to find pictures of friends or family faster, or simply to differentiate portrait pictures from landscapes, for example.

The difference between face detection and face recognition is that in the first, the faces and their landmarks (nose, eyes, mouth, etc) are only detected, and in the latter the faces from an individual are matched or verified against a database. Therefore, face detection is a necessary first step for face recognition. In MyWatson, both techniques are used: the faces are first detected and cropped, and then are clustered together, which acts as facial recognition.

In the following subsection, some state-of-the-art algorithms for facial detection and recognition will be briefly discussed.

2.2.1 State of the art in face detection

Hjelmas and Low [56] define the problem of face detection as “given a still or video image, detect and localize an unknown number (if any) of faces”, and defend that a face detection system should be able to achieve the task regardless of illumination, orientation and camera distance. They categorize face detection techniques based on how to utilize a priori face knowledge: feature-based approaches and image-based approaches. The first type of approach focuses on deriving low-level features, much like the approaches discussed in section 2.1, and uses properties such as the skin color or face geometry. The second type of approach incorporates face knowledge implicitly through mapping and training schemes, and uses training algorithms without feature derivation or analysis.

Most of the techniques that fall into the first type are ideally similar to those already discussed, and they include the analysis of edges, grey-levels, color or motion in the video case. For example, in edge-detection-based approaches, the most primitive of features, edges are labeled and matched against a face model in order verify correct detections: in an example by Govindaraju [57], this is accomplished by
labeling edges as the left side, hairline or right side of a front view face, and then by matching the edges against a face model by utilizing the golden ratio for an ideal face. Gray information within a face can also be used as features. This is because facial features like eyebrows, pupils and lips usually appear darker than their surroundings.

Feature searching also falls on this type of approach, and its general idea is that the confidence of a facial feature’s existence is increased by the detection of nearby features. It starts by finding a prominent reference facial feature, usually the eyes or even the body, i.e. directly below the head. Algorithms using this approach then detect other features based on the distance between them and some reference ratio. Another closely related technique is the constellation analysis, which general idea consists in grouping facial features in face-like constellations using, however, more robust modeling methods such as statistical analysis.

These approaches fall into a further refined classification by Hjelmas and Low [56]: edge and gray-scale in low-level analysis and feature searching in feature analysis. The first deals with segmentation of visual features using pixel properties, such as grey-scale and color, which results in ambiguous features. In feature analysis, visual features are grouped into a more global concept of facial features using face geometry, which helps to locate the facial features.

The last type of feature-based approach involves the use of active shape models, with the purpose of extracting complex and non-rigid facial features such as the eye pupil and lips. The basic idea is to gradually deform an active shape model close to a facial feature so it interacts with low-level features – e.g. edges, brightness - and takes the shape of the whole facial feature. This way, higher-level features are modeled, close to the actual physical appearance. The main examples of usage of active shape models include the use of generic active contour called snakes by Kass et al. [9], deformable templates by Yuille et al. [58] and Point Distributed Model (PDM) by Cootes et al. [10, 59].

Snakes are active contour models, i.e. they lock onto nearby edges and follow them, influenced by external constraint forces and internal images forces that pull them towards features, such as lines and edges. More formally, the evolution of a snake is achieved by minimizing the energy function:

\[ E_{\text{snake}} = E_{\text{internal}} + E_{\text{external}}. \]

While the internal energy depends on the intrinsic properties of the snake and defines its natural evolution, the external energy is what enables the contour to deviate from the feature and eventually take the shape of nearby features – usually, the head boundary, which is the state of equilibrium. Thus, the main considerations when implementing snakes should be made on the choice of both energy functions. However, snakes are not very efficient when extracting non-convex features due to their tendency to attain minimum curvature. Furthermore, part of the contour often becomes trapped on false image features.
Deformable templates takes the concept of snakes a step further by including global information of
the eye to improve the reliability. The general idea is to initialize a deformed eye template near an eye
feature and it will deform itself towards its optimal feature boundaries. A deformable template will try to
minimize the following energy function:

\[ E = E_v + E_e + E_p + E_t + E_{\text{internal}} \]

where the external energy takes into account the energy of the valley, the edge, the peak and the
image brightness, \( E_v \), \( E_e \), \( E_p \) and \( E_t \), respectively. Besides the eye, mouths are also detected using
similar templates. This type of templates, however, is very sensible to its initial position. Yuille et al.
demonstrated that if the template’s initial position is put above the eye, it will deform itself into the eyebrow
instead. It also has a very high processing time, which can be reduced using additional information or
using simpler versions of the template.

Finally, Cootes and Taylor [10] describe PDM as a way of representing a class of shapes using a
flexible model of the position of labelled points placed on examples of the class. The model consists
of mean points and nodes of variation that describe how the points tend to move from the mean, i.e. a
point on the PDM is defined as

\[ x = \bar{x} + P v \]

where \( \bar{x} \) is the mean feature in the training set for that point, and \( P = [p_1 p_2 ... p_t] \) is the matrix of the most
significant variation vectors of the covariance of deviations, and \( v \) is the weight vector for each node.
PDM was first used to detect faces by Lanitis et al. [59] as a flexible model, which depicts the global appearance of a face, including all facial features such as the eyes, nose, mouth and eyebrows. In order to fit a PDM in a face, the mean shape model (the set of $\bar{x}$) is first placed near the face. The face model is then deformed with the use of a local grey-scale search strategy, but constrained to be consistent with the information modeled from the training set. PDM was shown by Lanitis that it could approximate up to 95% of the face shapes in the training set. The advantages of this model include providing a compact parametrized description of a face, and further improvements by Lanities et al. [60] which include the use of genetic algorithms as a global search procedure allows all the features to be located simultaneously. Furthermore, it is shown that the occlusion of some features is not a severe problem, since other features in the model can still contribute to a global optimal solution.

The problem with feature-based type of approach is that they are very sensible to background clutter, and the need to perform face detection in more hostile scenarios motivated the research in the area to shift the treatment of the problem of face detection to that of pattern recognition. Thus, with image-based approaches, the need for specific face knowledge is avoided, and the potential modeling error due to incomplete or inaccurate knowledge is eliminated, as now the approaches include learning face patterns from examples, assigning positive or negative classes – face or non-face – to images. Generally, image-based approaches apply a window scanning technique to detect faces at multiple scales, and different implementations vary the size of the scanning window, the sub-sampling rate, the step size or the number of iterations. Hjelmas and Low roughly divide these approaches into linear subspace methods, neural networks – which will be discussed in section 2.3 and statistical approaches.

Linear subspace methods include the usage of techniques such as the Principal Component Analysis (PCA), Linear Discriminant Analysis (LDA) and Factor Analysis (FA) to represent the image subspace in which the face lies. Sirovich and Kirby developed a technique that utilizes PCA [61] to efficiently represent human faces. Eigenfaces are derived, which are individual faces in a face set approximated by linear combinations of eigenvectors. Turk and Pentland [62] later used this technique for facial recognition, as will be further discussed in the next subsection.

Another technique, introduced by Sung and Poggio [11] leverages a distribution-based face model, with the idea that because a face is usually a structured objects with the same key features geometrically arranged in the same way, a “canonical face” structure can be derived to perform pattern matching. The system exhaustively scans the image at all possible scales searching for those patterns by, within a given scale – and a window size –, dividing the image into multiple and possible overlapping sub-images with a given window size. At each window, the system attempts to classify that sub-image as a face or not a face.

More specifically, the algorithm can be divided into four steps:

1. The input sub-image is re-scaled into 19x19 pixels and is applied a mask.
2. A distribution-based model of canonical face patterns in the masked 19\times19 dimensional image window vector space is built. The model is comprised of 12 clusters, with an half representing “face” and the other representing “non-face” pattern prototypes.

3. For each new window pattern, a set of image measurements is computed, as an input to the “face” vs “non-face” classifier. The measurements are a vector of distances from the new window pattern to the window pattern prototypes in the image window vector space.

4. A multi-layered perceptron (MLP) is trained in order to classify the input vector from the last step into “face” or “non-face”.

Steps 1 and 3 are applied whenever a new image is to be classified, and the MLP provides the classification.

As a statistical approach example, Schneiderman and Kanade [63, 64] present two face detectors based on the Bayesian decision rule:

\[
\frac{P(\text{image}|\text{object})}{P(\text{image}|\text{non-object})} > \frac{P(\text{non-object})}{P(\text{object})} .
\]

If the left side – the likelihood ratio – is greater than the right side, then it is assumed that a face is present at the current location. In the systems by Schneiderman and Kanade, the visual attributes of the image are either represented by local eigenvectors coefficients or by a locally sampled wavelet transform, which can capture information such as the space, frequency and orientation.

### 2.2.2 State of the art in face recognition

Sharif et al. [65] discuss the current challenges of face recognition and divide the factors that affect it into two categories: intrinsic and extrinsic factors. Intrinsic factors are related to the physical face of humans, like aging or expression, whereas the latter are related to external conditions such as the illumination
condition or pose variation. Aging causes significant alterations in facial appearance. Some systems, for example biometric systems, should take this intrinsic factor into account, as with the passage of time faces should still be recognized. On the other hand, the facial expression of a person may alter the geometry of a face significantly, and such variations in an image may hamper the recognition. Similarly, factors that alter the face in any way, whether it be with a change in pose, an object occluding the face or even changes in illumination are also proven to be a challenge to solving the problem of face recognition. Techniques used to recognize faces include the Eigenface, the Gabor Wavelet, Neural Networks and Hidden Markov Model (HMM).

The Eigenface technique was introduced by Sirovich and Kirby [61]. The basic idea of this technique is to distinguish multiple faces using eigenvectors, computed from a covariance matrix. Eigenfaces are extracted by measuring the distance between key facial features, such as the nose tip, mouth and eye corners and chin edges. Eigenfaces were successfully used to perform face recognition by Turk and Pentland [62]. The idea is that a human face can be considered as a general, standardized face plus some variations of some eigenfaces. A set of eigenfaces is generated by using Principal Component Analysis (PCA) on a large set of different human faces. This training set of faces should be pictures taken with the same lightning conditions, and their eyes and mouths should be aligned across all images.

The Gabor Wavelet, also known as Gabor filters, were introduced as a tool for signal processing in noise by Dennis Gabor, and has the property of minimizing the product of its standard deviations in the time and frequency domains. In 2010, Barbu [12] utilized Gabor filters for facial feature extraction, or Gabor features. For each facial image, a 3D face vector is obtained, containing a robust content descriptor of the input face, or Gabor representations, as shown in figure 2.11. A supervised classifier is then used to perform feature vector classification.

![Figure 2.11: Input image and its Gabor representations (image taken from [12])](image)

Deep Neural Networks are also a common technique for face recognition and have been successfully utilized to solve such problem, with the only problem being the need for a large amount of training data.
Hidden Markov Models (HMM) have also been successfully used in face recognition, such as the one used in 2007 by Salah et al. [66], which swaps the usual raster scan that analyses local features in a sequential manner with an HMM based on the idea that the distribution of discriminative information is not uniform over the face, e.g. the eyes and the mouth are more discriminative than the cheeks. It simulates a person's selective attention mechanism and focuses more resources on the more informative parts, and it is shown in the paper that high accuracy is reached with much less feature sequences.

2.3 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are a class of Artificial Neural Networks (ANN) with an architecture specifically made to analyze visual data. This means that any kind of data that can be arranged to look like an image can successfully be learned with a CNN. Because CNNs leverage the same principles behind other ANNs, only the former will be discussed, as well as some of the most common architectures, as it is the one utilized as the backbone of facial feature extraction in the MyWatson system. In other words, it is being used as a black-box that, given a face image, outputs a vector of features.

Artificial Neural Networks are systems that are inspired in actual biological neurons in animal brains: computational units (neurons) have scalar inputs and outputs, with each input having an associated weight. Neural networks are trained to predict the class of an input by repeatedly adjusting the weights associated with each input according to the difference between their prediction and the actual, correct class, called the error. For example, in the most simple case of Neural Networks, with a single neuron, all the input values are multiplied by their respective weights and added, resulting in the output value, as shown in figure 2.12. The difference between the output and the correct value, i.e. the error, will then serve as the basis to make adjustments on the weights.

This method of weight adjusting is called backpropagation, and is a common method when training deep neural networks, i.e. neural networks with more than one hidden layer. It is very commonly used when performing gradient descent, by adjusting the weights for a small amount and watching how the error changes. The training is done iteratively by adjusting the weights until they stop changing or, more realistically, until a certain number of iterations is reached.

In order to better understand how CNNs work, the Modified National Institute of Standards and Technology (MNIST) handwritten digits dataset [67] will be used as an example. One can view a CNN as a detective trying to understand the content of a very large image using only a very small field of view. An image big enough cannot be seen all at once, so the detective must gather evidence from what he can see in order to reconstruct similar, already known patterns. The detective was trained to identify numbers by understanding that they can be deconstructed into smaller patterns, that can be utilized by different numbers: for example, the number “eight” is comprised of two “loops” on top of each other, and
the number “nine” also has a loop on top, but has a straight line below it. Thus, different classes may have patterns in common. Likewise, “loops” and “lines” can also be deconstructed into smaller patterns: a loop is composed of several “edges” and “corners” in a circle fashion. The job of the detective is to search in the entire image for these known, low-level patterns, and form more high-level patterns in order to gather enough evidence that allows him to distinguish between the classes.

In this – very basic – example, to recognize both levels of patterns, the CNN would need two hidden layers: one to recognize low-level patterns, such as edges and corners, and another one to, given the previous patterns, capture more composed shapes, such as loops and straight lines. However, how does the CNN looks at a set of pixels and understands that a pattern is present or not?

Each main building block of the CNN, i.e. a layer, has a singular purpose, and belongs to one of the three following types: a convolution layer, a pooling layer, or a fully connected layer. Different CNN architectures arrange these layers in an sequential fashion in different manners, but the idea is generally the same:

- The input layer transforms an image into a 3D vector: one dimension for the width, other for the height, and other for the color channels. For example, an RGB image with 224x224 pixels will be transformed into a (224, 224, 3) vector.

- Filters are what allow the CNN to recognize patterns, and this is done in the convolution layer: each filter that corresponds to a pattern is convoluted around the entire image and the scalar product between the image pixels and the filter is computed, as seen in figure 2.13. The picture displays a 3x3 filter that detects, in an image containing an “X”, diagonals from the upper-left to the bottom-right. The product is then stored in the center of the 3x3 area’s corresponding position in a matrix with the same size as the image. Because multiple filters are convoluted, one images becomes a stack of images, i.e. the matrices containing the scalar product.

- In order to shrink the image stack of the previous layer, a technique known as pooling is used. Pooling consists of walking across the image with a window of a given size and with a specific stride – commonly 2 for both – and saving a combination of the values from the image within the
window in a new, smaller matrix. Common types of pooling include the **max pooling** and **average pooling**, where the result is the maximum value of the window and the average of the values in the window, respectively. However, pooling is optional and is not uncommon to not see it being used in a given architecture.

- Another important layer in CNNs is the normalization layer, which prevents negative values. This is commonly done by a **ReLU layer**, which uses a Rectified Linear Unit function that simply swaps all negative values to 0, keeping the other values the same.

- Finally, a **fully connected layer** flattens the input image and treats it as a vector with one dimension. Each element of the vector is assigned a weight as a product of the training and, during the classification, each element essentially casts a weighted vote on a given class, as pictured in figure 2.14. Thus, the highest voted class is assigned to the input image.

If this is the last layer, it is usually where the most important features are located before the classification. When this layer is removed, a vector of features is the result, as it is done in MyWatson for the purposes of clustering similar faces. Intermediate features are also detected as a result of an hidden fully connected layer, as in the discussed example of recognizing MNIST handwritten digits.

The weights of the votes in the fully connected layers and the filters are a result of the training phase, found by utilizing the backpropagation technique. All of the discussed layers can be stacked, and different architectures are originated by doing so in any given order. The most relevant CNN architectures for the MyWatson application are the VGG16 and the ResNet50.

The **Visual Geometry Group (VGG)** CNN architecture was introduced by Simonyan and Zisserman [68] in 2014 is characterized by its simplicity. Its VGG16 variation includes 13 convolution layers and
3 fully connected layers, as opposed to its VGG19 counterpart which has 16 convolution layers and 3 fully connected layers. The convolution layers are $3 \times 3$ in size and are stacked on top of each other with an increased depth, and the volume reduction is handled by the pooling layers which implement max pooling. Right before the end are three fully connected layers, the first two with 4096 nodes and the last one with 1000 and, finally, a softmax classifier, which outputs probabilities for each class. In 2014, this architecture was considered a very deep CNN. The major drawbacks of the VGG network include the slowness to train and the weights themselves being quite large. To train the VGG with a large number of layers, Simonyan and Zisserman first trained smaller versions of the architecture with less layers, in a process called pre-training. Only after the smaller versions were trained the larger ones could be initialized with the previous.

On the other hand, the deep residual network know as ResNet, first introduced by He et al. [16], aims to solve the problem of degradation found in deeper architectures: as the depth of an architecture
increases, the accuracy gets saturated and degrades rapidly. The solution is based on the usage of residual building blocks, shown in figure 2.16, which essentially introduce identity shortcut connections between the convolution layers. The basic intuition is that it is easier to optimize the residual mapping function than the original mapping, i.e. the focus changes from learning functions to learning the residual functions with reference to the layer inputs.

![Figure 2.16: A residual building block from ResNet (image taken from [16])](image)

This particular type of architecture demonstrates that extremely deep networks can still be trained using the standard stochastic gradient descent commonly used, through the use of residual modules.

Another useful technique in the field of deep learning is transfer learning, defined as using knowledge of one domain in another domain. More specifically, by training a neural network in one dataset corresponding to a given domain and then use it another dataset by fine-tuning the same network. In practice, this technique is used to speed up the training process, by obtaining a pre-trained model on a very large dataset and then use it as initialization or as a feature extractor for the task of interest. It is common to tweak the weights by continuing the backpropagation process on the new dataset. In My-Watson, this technique is leveraged as a fixed feature extractor: a CNN pre-trained on 2.6 million images of 2.6K different people, as described in the “Deep Face Recognition” paper by Parkhi et al. [69], is used to extract the features from images containing faces, which are then utilized for clustering purposes.

### 2.4 Clustering

Face clustering is a good alternative for facial recognition, specially in the case where actual identities – i.e. names – are not needed and only some kind of distinction needs to be done between faces. Otto et al. [70] utilizes a rank-order clustering algorithm to achieve a 0.87 F-measure score (harmonic mean of precision and recall) in the Labeled Faces in the Wild (LFW) dataset [71], which has a better cluster accuracy than other know clustering methods. In this section, some known clustering algorithms will be discussed.

The main purpose of clustering or cluster analysis is to transform a large set of raw, unstructured data into groups of semi-structured data with related properties or features – i.e. data within a cluster is more
semantically close to each other than with data outside of a cluster. Clustering falls into the unsupervised learning category since it doesn’t need previously labeled data. While supervised learning methods have the goal of determining the class to which a given data point belongs to, clustering methods aim to find if there are relations between the data points by looking for some hidden pattern in the data.

### 2.4.1 Clustering Algorithms

Some authors, such as Mann and Kaur [72] and Fahad et al. [73], divide clustering algorithms into five general categories: hierarchical, partitioning, density-based, grid-based and model-based.

- **Partitioning-based:** In partitioning-based clustering algorithms, the number of clusters must be specified *a priori*, and data is divided according to these partitions, with the requirements that each cluster must have at least one data point, and a data point must only belong to one cluster. The most well-known example of this category is the K-means algorithm, which has been previously discussed. The basic idea is to define $k$ initial centers (means) and then, in each iteration, associate each data point to its nearest mean – the one with the least squared distance, forming a cluster – and compute new centroids based on the points in each cluster, until no more changes are observed. Other algorithms include the Clustering LARge Applications (CLARA) and Clustering LARge Applications based on RANdomized Search (CLARANS) algorithms [74] or the Partitioning Around Medoids (PAM).

  ![K-means Clustering](image)

- **Hierarchical-based:** This type of cluster analysis seeks to build hierarchies of clustering and may do it in two different ways: agglomerative and divisive. In agglomerative approaches, also called "bottom-up", each observation starts with its own cluster, and pairs are merged as the
hierarchy moves up. Divisive or "top-down" approaches begin with a cluster containing all the data points, and as the hierarchy moves down the clusters split in pairs. Usually, the number of clusters \( k \) is a requirement and a stopping condition for the algorithm. In both cases, to find which clusters are aggregated or how to split one cluster into two, a dissimilarity measure between sets of observations is required, generally using a distance metric (such as the Euclidean or Manhattan distance) and a linkage criterion which specifies the dissimilarity measure as a function of the distances between pairs of observations. Common linkage criteria include:

- **Maximum or complete linkage**: Takes the maximum pairwise distance of the two sets of data points: 
  \[
  \max \{d(a, b) : a \in A, b \in B\},
  \]
  where \( A \) and \( B \) are two sets of observations. This linkage generally produces elongated clusters.

- **Minimum or single linkage**: Takes the minimum pairwise distance of the two sets of data points: 
  \[
  \min \{d(a, b) : a \in A, b \in B\},
  \]
  where \( A \) and \( B \) are two sets of observations. This type of linkages generally produces compact clusters. Both the minimum linkage and the maximum linkage are sensitive to outliers.

- **Average linkage**: Takes the mean of all of the pairwise distances between all of the data points in both sets of observations:
  \[
  \frac{1}{|A||B|} \sum_{a \in A} \sum_{b \in B} d(a, b)
  \]

- **Centroid linkage**: Takes the distance between the centroids of each set of observations.

- **Ward linkage**: The distance between two clusters \( A \) and \( B \) is how much the variance will increase if they are merged, and thus needs to utilize the euclidean distance. This distance can be seen as a cost to merge two sets of observations, and can be a good clue to when to stop joining clusters: if the cost took a "big" step, it is probably indicative that the \( k \) before the jump is the correct number of clusters. Of course, the threshold of this jump is dependant on the application of the algorithm.

**Figure 2.18**: Hierarchical clustering dendrogram (image taken from [18])

An advantage of hierarchical-based approaches is their flexibility, as the level of granularity of
clusters is embedded in the approach itself. On the other hand, most algorithms of this type do not look back to a cluster for improvement purposes once it has been built, and termination criteria can sometimes be vague.

An example of hierarchical-based clustering is the Balanced Iterative Reducing and Clustering using Hierarchies (BIRCH) algorithm by Zhang et al. [75]. The main advantage of BIRCH is its capacity to handle noise – data points that do not fit an underlying pattern - and its effectiveness on very large databases, by exploiting the fact that not every data point is equally important and that the data space is not uniformly occupied.

- **Density-based:** In density-based clustering, data points are separated according to regions of density, connectivity and boundaries. Sparse areas are required, with the purpose of separating clustering, and data points in those sparse areas are usually seen as noise or border points and thus are not assigned a cluster.

A prime example of this type of clustering algorithm is the Density-based Spatial Clustering of Applications with Noise (DBSCAN) [76] algorithm by Ester et al. The DBSCAN algorithm considers a dense region any region with at least $\text{minPts}$ points, provided that the distance between any pair of those points is not larger than $\epsilon$: these are the two parameters of this algorithm. Each point is classified as follows:

- **Core point:** has at least $\text{minPts}$ points as neighbours within an $\epsilon$ radius.
- **Border point:** is within an $\epsilon$ radius of a core point, but does not have at least $\text{minPts}$ neighbours within its own $\epsilon$ radius.
- **Noise point:** it is neither a core point nor a border point.

A disadvantage of DBSCAN is that it does not behave well when faced with clusters that have different densities, and may have problems separating nearby clusters.

- **Grid-based:** Grid-based algorithms divide the data space into grids, and have a fast processing time, as the dataset is visited once to build the statistical values for the grids. The idea is to find clusters from the cells in the grid structure. STatistical INformation Grid (STING) [77] by Wang et al. is a paramount example of this type of algorithm, where the grid structure has different levels corresponding to different levels of resolutions, and layers have different number of grids.

- **Model-based:** Model-based methods optimize the fit between the given data and some predefined mathematical model, based on the assumption that data is generated by a mixture of underlying probabilistic distributions. Clusters are then identified by seeing which objects are more likely to belong to the same distribution. The Expectation-Maximization (EM) algorithm [78] by Dempster...
et al. is an example of model-based clustering that, although commonly used as a clustering algorithm, is a general methodology for finding distributions parameters. The idea behind the EM is to find parameters that best fit the data given. This algorithm follows two steps iteratively: the expectation step (E) and the maximization step (M). Parameters are first initialized as an initial estimate. Then, the likelihood that each parameters computes a given data point is calculated. In the expectation step, weights are calculated for each data point, based on the likelihood of it being produced by a parameter. Then, in the maximization step, a better estimate is computed using the weights and data. The algorithms stops when convergence is achieved, i.e. the estimates stop being different. The problem with this type of approach is that real world data is not always described as some mathematical probabilistic distribution and thus an underlying distribution cannot be found.

2.4.2 Metrics

When finding out in how many $k$ clusters should the data be split in, some kind of metric is needed to evaluate the quality of the clusters, much like in supervised learning to find out the precision or recall of a certain algorithm. However, this is not so trivial when utilizing clustering algorithms to do so. Considerations about this are done in the Scikit-learn [79] website, particularly that the separation of data should be compared to some ground-truth set of classes rather than cluster labels. Another alternative is to take into consideration if the data points are more similar to each other within a cluster than with other points outside of it.

The following metrics fall in the first class of metrics, where a set of ground-truth labels is needed for evaluation.
• **Adjusted Rand Index:** given a set of ground-truth labels and a set of the assigned clusters, this metric measures the similarity between the two, ignoring permutations and with chance normalization. This means that even if wrong “labels” are assigned to the points in a cluster, but the data is clustered in the same way, the score will be maximum. The score is a value between $-1$ and $1$, where $-1$ is a bad clustering and $1$ is a perfect clustering.

• **Mutual information based scores:** very similar to the previous metric, the MI (Mutual Information) score measures the level of agreement that the ground-truth set and the predicted clusters have, ignoring permutations. The possible score values are bounded by the range $[0, 1]$. However, random partitionings have an expected AMI score of around $0$ and thus may return negative values.

• **Homogeneity, completeness and V-measure scores:** homogeneity and completeness are intuitive measures based on the ground-truth classes in a given cluster. The homogeneity score is based on the assumption that each cluster should only contain members of a single class, and the completeness score leverages the idea that all members of a single class should be assigned to the same cluster. Both scores are bounded by the interval $[0, 1]$. Analogously to the precision, recall and F-measure in classification, homogeneity and completeness also have an harmonic mean score, the V-measure.

• **Fowlkes-Mallows index:** the FMI is defined as the geometric mean of precision and recall. Given the number of true positives $TP$ (number of pairs of data points that belong to both the ground-truth labels and the assigned predicted clusters), the number of false positive $FP$ (pairs that belong to
the same clusters in the ground-truth but not in the assigned clusters) and the number of false negatives $FN$ (belong to the same assigned clusters but not in the ground-truth set), the FMI score is given by

$$\text{FMI} = \frac{TP}{\sqrt{(TP + FP)(TP + FN)}}$$

The possible values of the score is also bounded by $[0, 1]$, and random labeling has a score near 0.

The common disadvantage of the previous metrics is the requirement for ground-truth labels. If they are not known, an evaluation should be done using the model itself. The following metrics output an higher value the more well separated and dense the clusters are, as is the definition of a cluster, and do not require knowledge about the ground-truth classification.

- **Silhouette coefficient**: evaluates the level of definition of a cluster, and for a single data point, the silhouette score $s$ is defined as

$$s = \frac{b - a}{\max(a, b)}$$

where $a$ is the mean distance between the sample and all the other points in the same class, and $b$ is the mean distance between the sample and all the other points in the next nearest cluster. The score of the whole cluster is the mean of all the scores for each sample, and is bounded between $-1$ and $1$, indicating incorrect clustering and dense clustering, respectively. Furthermore, scores near 0 indicate overlapping clusters.

- **Calinski-Harabaz index**: the Calinski-Harabaz score $s$ for $k$ clusters is given as the ratio of the dispersion mean between clusters and dispersion within clusters.
3 Technologies

Contents

3.1 Django ......................................................... 40
3.2 Google Cloud Vision ........................................... 45
3.3 Tensorflow ..................................................... 47
3.4 Keras ........................................................... 48
WITH the objective of developing a system that provides a worry-free management of a personal collection of photos and yet still allows the user to fully control it, and because the point of this thesis is to provide a working example of such a system, the focus was mainly on building a web application that implements it. The decision of developing a web application supports the idea that the user should have as little trouble as possible when using the system, and having to install a program in his personal computer to do so may not be the best approach. Furthermore, it can be accessed anywhere as long as the user has internet connection, and by any device such as computers, cellphones and tablets. Also, future updates that improve the application or fix issues do not need to be downloaded by users as patches, as these changes are made on the server side. Finally, because multiple frameworks and API already exist to speed up the development of such web applications, building MyWatson as a service on the web seemed the right decision. The usage of existing frameworks implies several advantages, such as:

- **Efficiency**: less time writing repetitive code and implementing already existent code, and more focusing on developing the logic of the actual application.
- **Less bugs**: most frameworks are open-source and are therefore tested by an active community.
- **Integration**: frameworks provide ease of connection between different technologies, such as database engines and web servers.

In this chapter, the technologies that were used on the development of the MyWatson application are discussed, such frameworks and APIs, including their features and advantages, as well as some reasoning behind the choices.

### 3.1 Django

Developing a fully operational web application that can be utilized by users as an actual working system is not a trivial task, and usually takes a lot of time and effort to do so, as it requires a good user interface, a working database as a way to store user data, a server-side module that implements the logic of the application, a client-side scripting that also implements front-end functionality and a way of deploying the application.

Django [80] is a Python web framework that simplifies most aspects of web development. Django rules itself by the Don’t Repeat Yourself (DRY) principle, which aims at reducing repetition of software patterns, leveraging abstractions and using data normalization. When the DRY principle is applied successfully, some modification of an element does not require changes in other elements that are logically unrelated.
The Django framework is based on the Model-View-Controller (MVC) architectural pattern, although Django’s architectural pattern is called the Model-View-Template (MVT) [81] since the controller is handled by the framework itself. A standard MVC architectural pattern has the following components, as pictured in figure 3.1:

- **Model (M)**: a representation of the data, i.e. not the actual data but an interface to it. Usually provides an abstraction layer so that data can be pulled from the database without dealing with the actual database itself, and the same model can be used with different databases.

- **View (V)**: it is the presentation layer of the model, and what the user sees, e.g. the user interface for a web application. It is also a way to collect the user input.

- **Controller (C)**: the controller manages the flow of information between the model and the view by capturing the user input through the view, controls the business logic of the application and decides which data is pulled from the database and how the view changes.

![Figure 3.1: Model-View-Controller architectural pattern (image taken from [21])](image)

On the other hand, in Django’s MTV has a different interpretation of the original MVC architecture, and because the controller part is done by the framework itself, a developer using Django has to deal with the following architectural pattern:

- **Model (M)**: the same as the original MVC architecture. In this layer, Django’s Object-Relational-Mapping (ORM) provides an interface to the database.

- **Template (T)**: contrarily to the standard MVC, the template is the presentation layer, which means that it controls how a given data is displayed and in what form.

- **View (V)**: this layer, while similar to its homonym in the MVC, has more characteristics of a controller, because while in the MVC the view controlled how the data was seen, here the view controls
which data is seen. In other words, the view fetches the content and the template presents that content.

Django’s ORM provides a way of fetching and updating data without executing query commands in whatever database engine is being used, and data is defined as a collection of variables with a given type. Thus, instead of creating a table – in most database types – for each type of object used in the application, the model – akin to an Object-Oriented Programming (OOP) class – is declared once and the framework creates the table without the developer having to actually write complex MySQL code, as seen in figure 3.2. Note that in spite of MySQL being the example given, Django supports four different types of database, which include PostgreSQL, SQLite, Oracle and, of course, MySQL. Each have different advantages and disadvantages, and the fact that Django supports four different databases is in itself an advantage that the developer can leverage, as he can then chose the one he is most comfortable with. However, despite having an additional abstraction layer which swaps common queries – such as those of the type “fetch objects of this type” or “with this ID” – with easy Python functions, which perform additional operations such as “joins” whenever needed, behind the scenes, the database can also be directly called by writing raw queries if a more complex one is required.

```python
class Face(models.Model):
    user = models.ForeignKey(User, null=True, on_delete=models.CASCADE)
    face = models.ImageField()
    photo = models.ForeignKey(Photo, null=True, on_delete=models.CASCADE)
    tag = models.ForeignKey(Tag, null=True, on_delete=models.CASCADE)
```

Figure 3.2: A model definition in Django. Note the relations with other models as foreign keys.

Data from the model is shown to the user through a template, which is defined in a normal HTML file, with some special characteristics. First, logic is separated from design by the usage of tags, as seen in figure 3.3(a). The variable `user` is passed by the view to the template, as the template renders whatever value the variable has when requested. Second, repetition is discouraged, according to the DRY principle, by using template inheritance. Akin to inheritance in OOP, by defining a “super-template” its characteristics are then passed down to its children without additional redefinition. The OOP approach is also useful in the web developing context, as web pages tend to have a layout that is constant throughout the entire website, and generally only the content of the page changes. The usefulness of this property can be seen in the example in figures 3.3(b) and 3.3(c). Furthermore, some security vulnerabilities are also mitigated by prohibiting code execution inside the template, as variables can’t be assigned new values and code cannot be executed, and strings are automatically escaped – unless explicitly said so.

Lastly, views retrieve data from the database and deliver it to the template. Each view is either a Python function or a class that performs a specific function, and each view has a single template
associated, as it can be seen in figures 3.4(a) and 3.4(b), respectively. Generally, a view receives a request from the template, for example when a user clicks a link, containing some information e.g. the user that sent the request. Then, some business logic is done by the view and the requested page (template) is returned and rendered, with the requested information. For example, as shown in figure 3.4(b), when a user executes a query over his gallery, the results of that query are returned to the user, serialized to JavaScript Object Notation (JSON) data in order to be sent to the browser. In the beginning of Django, only function-based views existed, and class-based views were introduced later as a way of writing common views more easily. For example, requesting lists of objects (e.g. a photo gallery) or a single object (e.g. a specific photo) were common, and class-based views – in this case, the ListView and the DetailView, respectively – simplify the process of writing such views, without explicitly telling the template to render and having to format data accordingly. Another aspect of views is that the POST and GET requests are separated, either by conditional branching in function-based views or by different methods in class-based views, which allows for better code organization and separation.

Another functionality that is worth mentioning is Django’s URL dispatcher or configurator, URLconf for short. This module is pure Python file mapping between URL path expressions and Python functions (the views), and can also reference other mappings, providing additional abstraction. An example of a mapping can be seen in image 3.5. Django runs through the urls patterns list and stops at the first match, and calls the corresponding view, passing a HttpRequest object. Additional arguments can also be passed through the URL i.e. for GET requests, and must match against regular expressions in the mapping.

Building authentication procedures and forms is also very simplified in Django. In normal HTML,
authentication and the like require a form tag that includes fields that allow user input, e.g. the username and password fields, and then sends that information back to the server. Although some forms can be fairly simple, handling them can be quite complex: they require validation, cleaning up, sending them via POST request and processing. Building a form in Django usually requires two steps: defining them in a pure Python file and then utilizing them in a template, as pictured in figures 3.6(a) and 3.6(b), respectively.

![An example of a definition of a form based on a specific model.](image)
![Using a pre-defined form in a template.](image)

**Figure 3.6:** Forms in Django. Note that the view must specify which form is rendered, as it can be seen in figure 3.4(a).

User authentication and authorization are entirely managed by the Django framework itself, which verifies if a user is who he claims to be and what that user is allowed to do, respectively. This module...
automatically hashes the password and saves it in a secure manner and maintains a session for the user until he logs out of his account.

Although Django has a lot more excellent features worth discussing, these are the main and most relevant ones, which were here briefly presented.

### 3.2 Google Cloud Vision

If Django is the scaffolding that supports the whole MyWatson system, the Google Cloud Vision API [55] is its heart, as it provides fully automatic tags given a photo, which is the most important aspect of the solution. By using the Google Cloud Vision API, the focus can be shifted from solving the automatic tagging problem, and rather build a working application that fully utilizes the advantages of an automatic tagging system.

In practice, any API that can analyze the content of an image and output high-level tags could be used, and many of them exist. To find out which to use, an informal comparison was performed between the following APIs: Google Cloud Vision, Watson Visual Recognition, Microsoft Computer Vision, Clarif.ai and Cloudsight. A set of 11 photos were annotated with each different API, and the top 5 tags were taken and assessed. A table containing each respective assessment can be seen in appendix C.

The Google Cloud Vision API was the one with the best results in general, with a good accuracy on the present objects as well as returning meaningful tags.

The job of the Google Cloud Vision API is simple: given an image, return information about the contents of the image, which include:

- **Objects**: Objects and concepts belonging to thousands of categories are detected, outputting high-level tags corresponding to those entities, as well as a value that describes the degree of confidence in a specific tag, as seen in picture 3.7(a).

- **Properties**: Several image properties are also extracted, such as dominant colors or crop hints.

- **Text**: Google CV employs Optic Character Recognition (OCR) which detects and extracts text from images, while also supporting and automatically detecting a large set of languages.

- **Landmarks and logos**: Popular logos can be detected using the Google CV API, as well as famous landmarks, be it natural or man-made, which are accompanied by relative latitude and longitude coordinates.

- **Faces**: Face detection is also a very important feature of the Google CV API as it also solves the problem discussed in section 2.2.1 in a single API. Given a picture containing one or more person, the API detects and outputs the coordinates of the detected faces, each one also having a degree
of confidence as well as face landmarks, such as the position of the eyes, eyebrows, mouth, etc. An example of the output of face detection is given in figure 3.8.

- **Web search:** Google CV also searches the web for related images and extracts similar terms called “web entities” which are analogous to high-level tags but not directly extracted based on the content of the image itself. This type of search can sometimes output relevant labels that are not returned by the content of the image itself, such as celebrity names. It is very similar to a reverse image search.

![Concepts and objects are detected from the image accompanied with values describing the degree of confidence](image1)

![Famous landmarks are detected and also have a degree of confidence and are marked with the coordinates](image2)

**Figure 3.7:** Google CV API examples

![A picture containing a person](image3)

![Extraction of facial features and a bounding box for the face, as well as additional information about the person’s emotion](image4)

**Figure 3.8:** Google CV API facial recognition example

Another advantage of Google Cloud Vision is that it gets better annotation results with time as new
concepts are introduced. Furthermore, because Google is known for designing advanced and sophisticated systems, Google Cloud Vision is likely very scalable and should improve in the future. Finally, because of its integration with the REpresentational State Transfer (REST) API, it can be used with different languages and operating systems, allowing for multiple requests at once and with different types of annotations. REST allows for generic HTTP requests (i.e. POST and GET requests) on API endpoints, including arguments embedded in the URI, and returns the result formatted as JSON.

Although Google Cloud Vision is being used as a black-box, the general algorithms/processes for automatic tagging and face detection were previously discussed in chapter 2 and therefore provide a basic understanding of what is happening when a request is made to the API.

3.3 Tensorflow

TensorFlow [22] is an open-source software library created by the Google Brain team for internal Google use, but was made public on November, 2015 [82]. The underlying software is build in high-performance C++, but several API front-ends are provided for convenient use, such as Python and JavaScript. It is cross-platform, and can run on multiple CPUs or GPUs, embedded platforms and mobile.

At its core, TensorFlow is a framework used to build Deep Learning models easily as data flow directed graphs. These graphs are networks of nodes, each one representing an operation that can range from a simple addition to a more complex equation, and represent a data flow computation, allowing some nodes to maintain and update state. An example of a TensorFlow graph can be seen in figure 3.9. Each node has zero or more inputs and outputs, and the values that flow through the nodes are called tensors, which are arrays with arbitrary dimension that treat all types of input uniformly as n-dimensional matrices and whose type is specified or inferred at graph-construction time. There are also some special edges called control dependencies that may also be present in a graph. They do not allow data to flow along them but are used to specify dependency relations between nodes, i.e. the source node must finish executing before the destination node can start its execution, and is a way to enforce order. Operations have names and represent abstract computations, and may have attributes which should be specified or inferred at graph-construction time in order to instantiate a node. Operations are categorized into multiple types such as array operations, mathematical element-wise operations, or Neural-net building blocks, to which belongs the Sigmoid, ReLU, MaxPool, etc.

Another important notion in TensorFlow is the session, which encapsulates the environment in which the graph is built and run, i.e. operations and tensors are executed and evaluated. A default, empty graph is generated when a session is created, and it can be extended with nodes and edges. Additionally, the session can be ran, taking as arguments the set of outputs to be computed and a set of tensors to be fed as input to the graph. Because nodes may have an execution order, the transitive closure of all nodes
to be executed for a specific output is calculated, in order to figure out an ordering that respects the dependencies of the appropriate nodes. Sessions also manage variables. A graph is usually executed more than once, and most tensors do not survive past one execution. However, variables are persistent across multiple executions of the graph. In machine learning applications, model parameters are usually stored in tensors held in variables and are updated as part of the training graph run.

3.4 Keras

In the previous section, the basic building blocks that TensorFlow provides were discussed. However, building fully functional neural networks from the ground up is not a trivial task even with tensors and nodes from TensorFlow. Because the goal is to focus on building a working prototype of a real-life application that users can fully experience, Keras [83] is also used.

Keras is a high-level neural network API written in Python, with François Chollet, a Google engineer, as its primary author, and is capable of running on several back-ends such as Microsoft’s Cognitive ToolKit (CNTK) [84], Theano [85] and, most importantly, TensorFlow. The focus of Keras is to allow for fast experimentation, as the developers themselves say: “Being able to go from idea to result with the least possible delay is key to doing good research.” This policy is supported by the user-friendliness of the API, as well as its extensibility and modularity. Keras can be seen as an interface to TensorFlow, offering more intuitive abstractions of higher level, making it an excellent API not only for developers that are not used to building deep neural networks but also for developers who want to quickly integrate deep learning architectures in their work.

Another advantage of Keras is that it already implements several known CNN architectures developed for general image classification, such as the VGG16 – the 16-layer model used by the VGG
team in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2014 competition [86] – and ResNet50, which were previously discussed in section 2.3, as well as the Xception and InceptionV3. All of these models already have weights pre-trained on ImageNet, an image database organized according to the WordNet hierarchy [87]. Furthermore, Malli [88] implemented on top of Keras the Oxford's VGGFace CNN model, which leverages VGG Face descriptors using the transfer learning technique previously discussed, originally implemented using the Caffe framework [89] – another commonly used deep learning framework developed by Berkeley AI Research. This library, keras-vggface, very similarly to Keras, provides implementations of the VGG16, ResNet50 and Senet50 models and, also like Keras, comes with weights pre-trained on the dataset by Parkhi et al. [69] with over 2.6M images and 2.6K different people.

The most prominent feature of keras-vggface and Keras itself is the ability to import and download models and pre-trained weights, providing the ability to utilize a functional and trained CNN to classify images without the need to train it, fully bypassing the most prominent disadvantage of using neural networks. Another advantage is the ability to customize or tweak the model: parameters and activation functions can be changed and layers can be added or removed, which is an important feature that allows the removal of the fully-connected classification layer – the last layer. This way, by leveraging CNN features “off-the-shelf” as discussed in the paper from Razavian [30], features are the output of face “classification”, turning a CNN into a feature extractor for face images, as shown in figure 3.10. Features can also be extracted from an arbitrary layer, not just the last. Finally, despite the models being already trained, one can train them with one’s own data if one chooses to do so, for example to classify images into categories other than those trained.

```python
model = VGGFace(model='vgg16', input_shape=(224, 224, 3), include_top=False)
img = image.load_img(image_path, target_size=(224, 224))
img_data = image.img_to_array(img)
img_data = np.expand_dims(img_data, axis=0)
img_data = utils.preprocess_input(img_data)
features = model.predict(img_data)
```

**Figure 3.10:** An example showing the basic usage of keras-vggface library to extract features – by excluding the final, fully-connected layer.

With the ending of this chapter discussing the technologies and APIs utilized in the development of MyWatson, and with background in image tagging, face detection and recognition, as well as deep learning and clustering from chapter 2, it is now possible to present, in the next chapter, an in-depth view of the MyWatson system, including a general overview of its modular architecture, and a more elaborate description of how each individual module works, what it does, and how they all work together to produce automatic tags, provide the execution of queries by keyword and agglomerate similar faces.
together, resulting in a system that provides the full experience of a personal assistant for a user’s private photo gallery.
## 4 System Architecture

### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Overview</td>
<td>55</td>
</tr>
<tr>
<td>4.2 Django controller</td>
<td>57</td>
</tr>
<tr>
<td>4.3 Core</td>
<td>65</td>
</tr>
<tr>
<td>4.3.1 Automatic Tagger</td>
<td>65</td>
</tr>
<tr>
<td>4.3.2 Retrieve and Rank</td>
<td>68</td>
</tr>
<tr>
<td>4.3.3 Learning</td>
<td>69</td>
</tr>
<tr>
<td>4.4 Front-End</td>
<td>73</td>
</tr>
</tbody>
</table>
Based on the previously presented chapters discussing the problems and techniques of content-based image retrieval, clustering, face detection and recognition, as well as deep learning as a powerful tool to solve some of the previous problems, the details of how MyWatson leverages the presented techniques in order to build an environment that the user can fully utilize to upload, manage and automatically tag their photos will be discussed.

Because the discussion will be focused around certain uses cases, it is a good idea to first define what the user can and cannot do when utilizing the system. The user can:

- **Create an account and log-in:** in fact, this step is a requirement to utilize the system, due to the uploaded photos being associated with an account, as well as the tags and faces extracted along the way. It is an easy process that, for simplicity's sake at this stage, does not require confirmation by email.

- **Upload photos:** the user can upload one or more photos at once, either by clicking the button that will open the user's operating system explorer and allow him to choose the photo(s), or by dragging and dropping the photos or folders containing photos to the upload area. Selected folders can be nested, i.e. if a folder to be uploaded has sub-folders, the contents of that sub-folder will also be uploaded, and so on. Note that only photos are uploaded, even if other files are selected and, as demonstrated in figure 4.1, images must not be over 10 MB in size due to Google’s Cloud Vision own technical limitation, but it would be a good idea to limit the size even if that would not be the case.

- **Browse the gallery:** the first page of MyWatson is the gallery which is, of course, empty until the user uploads at least one photo. Then, all of the user's uploaded images are displayed here, in the form of medium-sized thumbnails, arranged in a grid-like pattern. Clicking on a photo will take the user to another page, where the photo can be seen in larger size and the assigned tags can be seen. The user can also delete photos.

- **Edit or add tags:** high-level tags are automatically assigned to a photo when it is uploaded. In spite of this, full control is still given to the user, given the fact that he can still add his own personal tags or even remove existent ones. An example of this interaction with the system can be seen in figure 4.2, depicting the photo details page. Furthermore, the user can also re-tag the photo, applying the automatic tag process again and eliminating any custom tag that he eventually added.

- **Perform queries:** there is a search field in the navigation bar, which is always present whichever page is being displayed, thus allowing the user to execute queries by keywords in any page. After performing the query, the user is redirected to a page containing the results retrieved by the MyWatson system, as illustrated in figure 4.3. The resulting images are considered relevant if, putting
it simply, they have at least a tag in common with the introduced query. Further details will be provided on this matter along this chapter.

- **View aggregated similar faces**: the user can also view a page where all the faces are displayed clustered together, according to facial similarities. Clicking on a face will redirect the user to the photo from where the face was cropped. Because the system computes several sets of clusters by varying the number $k$ of clusters in a strategic order – which will be further discussed –, the user may also change $k$ in the slider to view different sets of clusters. Each group of clusters has a score, corresponding to the silhouette score for that cluster group, and, by default, the group with the highest score is displayed. From left to right, the slider shows the group with highest to lowest score, respectively. Furthermore, clusters can also be re-arranged and changed its name when in edit mode, as pictured in figure 4.4. After saving the changes, cluster groups that have been changed by the user will always have the maximum score of 1, unless they need to be recomputed.

![MyWatson's upload page](image)

**Figure 4.1**: MyWatson’s upload page.

The user cannot, however:

- **Specify a number of clusters**: the set $k$’s to compute is obtained on the fly according to a specific strategy that tends to minimize the number of computations it needs to do. At this point, the user cannot ask MyWatson to compute the cluster group of a given $k$, which would be useful in the case that he knew the exact number of people in all of the photos present in the gallery.

- **Add meta-data**: titles, captions or other types of meta-data cannot be added. All the textual information belonging to a photo that the user wants to add can only be assigned by adding new tags.
• **Delete photos en masse**: although a lot of photos can be uploaded at once, the deletion of photos should be done one at a time, as there is currently no option to do otherwise – admittedly, this would be a useful feature but it is, however, a minor detail and not a very relevant feature to the objective of this thesis.

• **Edit photos**: MyWatson is not a photo-editing program and this feature is not related with the thesis’ objectives.

![Figure 4.2: Deleting the incorrect tag “sea” and adding the tag “Portugal”](image)

![Figure 4.3: The return results for the query “beach”](image)

In the remainder of this chapter, this discussion will be further elaborated. First, an overview of MyWatson architecture will be presented, providing a general insight of the function of each component as common operations are performed, for example, when a user uploads a set of photos or executes a query by keyword. Then, the implementations details and choices of each individual module that composes MyWatson will be further discussed, providing a more in-depth clarification of its inner workings and each of the modules’ duty.
4.1 Overview

MyWatson’s architecture is graphically described in the diagram shown in picture 4.5, and is composed of three different main components:

- **The front-end**: this is what the user sees. It is essentially comprised of the MyWatson website, which hosts the application. The most relevant detail of this component is the website itself, i.e. the user interface. Despite not being the most important element of the system, it is still a very important one: a good user interface facilitates the life of the user, avoids wasting his time by not hiding crucial elements and information and is accessible and simple to use.

- **The Django controller**: the Django framework is what receives requests and returns responses from and to the front-end, respectively. It is also what implements the whole server-side business logic, including the management of the database – which, despite making the database also an element in this component, could be done without Django – and the pre-processing of some data to send to the main MyWatson application, the core. Because of this, it is seen as the mediator between the front-end and the modules that perform the tagging, retrieving and learning operations.

- **The core**: divided into four modules – the main application, the retrieve & rank, the automatic tagger and the learning modules –, the core is where the previously discussed CBIR and learning techniques are implemented. The main module, however, is the intermediary between the three other modules and the rest of the application and therefore, in the future, will not be referenced as a “module”, unlike the other three. This approach – which roughly follows a facade software-design pattern – makes adding new functionality very easy, avoids a lot of complexity and provides
a singular, simplified interface that allows communication with the rest of the application.

To help draw a parallel between the system’s architecture and the CBIR workflow presented by Zhou et al. [1] discussed in the introductory chapter, the architecture can also be divided into the online and offline phases. In other words, the principal module that deals with the offline phase is the automatic tagger module. Technically, offline phase work just has to be done before the user inputs a query, thus categorizing the learning module also as part of the offline phase, further supported by that fact that the learning module can also assign tags to photos. On the other hand, the retrieve and rank module carries out the work in the online phase, in which the user performs a query and the relevant results are retrieved.

The most relevant implementation is done in the core modules, where the main logic of the application is, i.e. the code that is directly related to all the previously discussed problems and techniques is presented in the retrieve and rank, automatic tagger and learning modules. The former two implement techniques that try to solve the CBIR problem by leveraging the Google Cloud Vision API and a text-based information retrieval technique called TF-IDF, while the last module implements and utilizes a convolutional neural network to extract features of face images.

In short, each core module has a distinct role when responding to the common use cases discussed. The retrieve and rank module is responsible for providing the images whose content is relevant to a specific set of keywords. Because this is done following a text-based approach, it means that each image is treated as a document containing words, i.e. following a Bag-of-Words model. The automatic tagger “transforms” each image in a text document, analogously to the BoW model but, putting it simply, it classifies the image into several categories using Google’s Cloud Vision API, outputting tags that describe the content of the image, such as objects, colors or people. Finally, the learning module computes the features of face images, providing groups of people which are easier to visualize and find.
specially if the user has a lot of photos in his gallery. Changing the name of the clusters also assigns a new tag to all the photos that have faces in that cluster, further enhancing the retrieval process.

In the following sections, each of the main components will be further discussed, including their functionality, role, and implementation details. Furthermore, common messages and interactions between the smaller components will also be explored in detail, in addition to a careful look on what happens behind the curtains when the previously presented common use cases are performed.

### 4.2 Django controller

It makes sense to discuss the mechanics of the Django controller first, not only because the framework is the scaffolding of the whole MyWatson system, but also because it contains the models, i.e. direct communication channels to the database containing the data model. In order to understand what MyWatson does, a discussion about what kind of data is managed by the database is in order.

The models can be simply described as MySQL database tables wrote in Python so, to simplify the process of discussing them, the models will be seen as object classes containing information about that specific entity. An Entity Relationship Diagram (ERD) of the models, generated by the MySQL Workbench software, is pictured in figure 4.6, presenting the classes as well as the relationships between them.

---

Figure 4.6: An ER diagram picturing the data models of the MyWatson system.
Django generates other tables for the framework itself to utilize, such as the `django_session` table which stores the users sessions, or the `auth_permission` that stores the potential permissions that can be granted to a user, but these are not relevant for the context of the application and thus will not be presented nor discussed. The exception is the `user` table which, despite being generated by the Django framework, is relevant to the application. For simplicity's sake, some fields in the `auth_user` table presented in the ERD were omitted, such as the first and last names, as their relevance is also minimal.

An example of the creation of a model in Django can be seen in the previously presented picture 3.2: a normal Python class is created within the file `models.py` – which contains the definition of all the models for a given application – extending the class `django.db.models.Model`, the base Model class in Django. Table names are assigned by taking the lowercase name of the class (e.g. "Face"), the lowercase name of the Django application (i.e. "MyWatson") and concatenating the former with the latter with an underscore in-between (e.g. resulting in the table "mywatson_face"). Inside the class, the fields – columns, in a table – are defined through a normal variable assignment, where the left side is the name of attribute, and the right side is the type of the attribute or a reference as a foreign key to another table. Note that the model’s identifier or primary key is not defined by the developer: instead, it is created automatically by Django, and also automatically incremented whenever a new row is added to the table. Field types are also defined by Django in the class `models`, containing the expected types `CharField` for string or `IntegerField` for integers, but also fields such as the `FileField`, which provides an abstraction for the storage of uploaded files, or the `ImageField`, which is just a `FileField` with an additional validation to check if the uploaded file is an image. Generally, fields validate themselves, i.e. if a `FloatField` takes a string, it will return an error. Some other field types are presented in the table 4.1. Foreign keys, on the other hand, are present as the `models.ForeignKey` method, which takes as argument the target table, a flag that designates if the foreign key can be null or not, and also an argument to specify what happens when the target foreign key is deleted.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinaryField</td>
<td>Stores raw binary data</td>
</tr>
<tr>
<td>BooleanField</td>
<td>A true/false field</td>
</tr>
<tr>
<td>DateTimeField</td>
<td>Date and time represented in Python by a <code>datetime.datetime</code> instance</td>
</tr>
<tr>
<td>EmailField</td>
<td>A CharField that checks if the value is a valid email address</td>
</tr>
<tr>
<td>GenericIPAddressField</td>
<td>An IPv4 or IPv6 address represented in string format</td>
</tr>
</tbody>
</table>

Table 4.1: Django model fields

Given the above, the relevant models that describe the types of data necessary for the MyWatson application itself, with simplified names, are as follows:

- **User**: This model defines the user abstraction, representing real people that utilize the MyWatson system. It stores the relevant information about them, i.e. the minimal information that is required
to utilize the system and to manage the users. The most important information, required for the
sign-up and log-in, are the username, email and password fields. The former two must be unique,
as it serves as a way of identifying the users. Generally, it is a good idea to require an email
confirmation as a way to reduce the number of fake accounts, but because the application is live
as a proof of concept, not requiring it speeds up the process of registration and log-in, and allows
users to jump right into experimenting with MyWatson.

Passwords, on the other hand, are not stored in plain text, and instead adopt the format, de-
defined by Django, `<algorithm>$<iterations>$<salt>$<hash>`. By default, the algorithm used is
the Password-Based Key Derivation Function 2 (PBKDF2) with 10000 iterations over the SHA256
hash, working as a one-way function, with the salt as a random seed, which complies with the
recommendations of the National Institute of Standards and Technology (NIST) [90].

- **Photo**: The photo model represents the real world photography, but instead of having the content
directly represented, it keeps the record of where that image is stored in the disk, i.e. its path.
This is expressed by the ImageField attribute `image`, which receives a string argument `upload_to`,
pointing to a given directory to store the image in.

For example, when a given user `UserFoo` uploads a photo, it will be downloaded by the server and
saved in the location `<MEDIA_BASE_DIR>/photos/UserFoo/`. The photo also has an attribute that
points to its owner, i.e. a foreign key to the user table. This is very important, as the only photos
that a given user should see are his own and not anyone else's.

- **Tag**: Expresses a high-level keyword that describes the content of the photo it is associated to,
which directly translates to having a foreign key pointing to the photo model. It also has a tag
field which is a string containing the actual keyword, a score field that represents the degree of
confidence in the tag, which ranges from 0 to 1, and four fields called `startX`, `startY`, `endX` and
`endY` representing, pairwise, the coordinates of the top-left and bottom-right corners, respectively,
of a potential bounding box, when applicable. Because, at this point, Google's Cloud Vision API
does not output the bounding box for a given tag, the coordinate fields are only non-empty when
a tag directly expresses a face. The tag "person" is a special tag that is added by an auxiliary
process: in the tagging process of a photo, if GCV detects a face a bounding box is outputted in
this case and so the special tag is added by the MyWatson system to that particular photo.

Finally, the tag model has a category string attribute: because GCV can extract different types
of high-level tags, these are distinguished and displayed as such to the user, as an additional
cosmetic information. The possible values of categories are: label for normal tags, web_entity
for labels extracted from the reverse image search, color for tags describing the color content
of the image, face for the special face tags, and finally logo, landmark and text which are self
explanatory. Although the distinction is purely cosmetic, clicking on a face tag when viewing a picture in detail will draw a box around the corresponding detected face. Furthermore, the user might want to deal with tags differently, depending on the category. As an additional note, tags added by the user are always categorized as label and have a score of 1.

- **Face:** When a face is detected and extracted from an image, it is also cropped – using the coordinates given by the GCV – and saved on a table in the database, corresponding to the face model. The model's definition can be seen in image 3.2, and consists of a set of foreign keys that point to the user, the photo and the tag it is associated with, and its main attribute is the location of the cropped face as an ImageField, which are utilized to cluster the faces. A record is added to this table whenever a face is detected by GCV during the tag extraction of an image, which will result in an additional, special tag containing the keyword "person" and the coordinates of the face in the image.

- **Face Cluster:** Each instance of the face cluster model essentially corresponds to an assignment of a face to a cluster within a group of clusters, which is defined by the triple (n_clusters, cluster_id, face). To clarify, a graphical example of two groups of clusters can be seen in picture 4.7. From the top-down: a group of clusters, henceforth called cluster group, has a given n and a silhouette score. Within a cluster group, the data points (the faces) are separated into n clusters, according to a certain clustering algorithm. Given the above, we can see that each cluster group operates on the exact same data points, but the same face can belong to two different instances, each belonging to its own cluster group, as it can be seen in the figure: despite the yellow and red points corresponding to the same face, they are two different instances in the model. Therefore, each instance is defined by the n_cluster which is the cluster group identifier, cluster_id which is the identifier of a cluster, and the face, which identifies the face using a foreign key, and has the additional attributes silhouette_score – which is the same across all instances within the cluster group – and name – describing the name of a given cluster and is the same across all instances within the cluster. The default name is designated by "Cluster <cluster_id>", until the user changes the name of the cluster. Additionally, the user is also an attribute of a face cluster instance.

- **User Preferences:** In the website, when the user first opens the page containing the face clusters, the default cluster group that shows is the one with the highest silhouette score. However, there might be some cases when the cluster group with the best score is wrong. In this instance, the user might want to change the number of clusters to display, i.e. the cluster group. After doing so and saving the changes, the cluster group number is saved in a table, corresponding to the model UserPreferences. In summary, at this point, an instance of this model has the attribute user as
a foreign key and the attribute n_clusters, corresponding to the chosen cluster group. Also note that each instance is unique, i.e. when the user changes the cluster group to be displayed, the new preference will overwrite the old one.

• **Features:** Instead of having to extract features from face images every time the clusters need to be recomputed – which, between adding new faces or removing photos, is very common –, the features are computed once and then stored in the database as a JSON string, corresponding to a list of floating point values. The learning module will then check if the features for a certain face were already extracted before doing it himself if they were not. Accordingly, the model has two fields: one points to a face object, and the other is a string containing the representation of the vector of features of the respective face image.

Before discussing more complex Django elements, it is helpful to consider the structure of a Django project. A Django project can be seen as a site, and the site can have many apps. The site has its own global configuration files and apps have their own folders inside the project root. Inside each app folder are the views, models, templates, and other configuration files or folders that are application specific. Despite a folder separation, each app can use files from the other apps. The following is a basic view of the most important folders in the MyWatson Django project, illustrating the file tree described:

```
mywatson_site/
├── manage.py
```

```
mywatson_site/  
├── __init__.py
├── settings.py
└── urls.py
```
The core application folder contains files for the website itself, not the MyWatson core, such as the landing page, the sign-up and the log-out logic, including forms and views utilized. The settings.py file contains pure Python code where Django variables are defined, such as the installed apps – which are the core, mywatson and sorl.thumbnail for enhanced thumbnails in templates –, base folders for some important files – such as the templates and media folder –, the database engine and credentials and used middleware – applications for things like security and sessions.

It is also relevant to have an in-depth discussion about Django views, as they are the interface between the models and the template, as well as the MyWatson core. Whenever a user enters a specific URL, a view is requested with the objective of preparing the data that the page at the given URL needs to display. Thus, a mapping of URLs to views is required as a way of knowing which view handles which data. This is done by the URL dispatcher (also informally called URLConf), which is a module consisting of pure Python code, present in the file urls.py. The URL dispatcher used in the MyWatson application can be seen in figure 4.8. As an illustrative example: when the user clicks on a photo in the gallery, the
browser will be redirected to a link with the form http://mywatson.com/mywatson/<int:pk>/, containing the photo identifier as an integer. To clarify: the view `DetailView` will handle the request, receiving the integer argument `pk` whenever the requested URL matches the specified pattern, i.e., if a user clicked on photo 86, the link http://mywatson.com/mywatson/86/ would match the view `DetailView` and pass the argument 86 (with the name “pk”) to the function in the python file `views.py`.

Another way to redirect is by using the `name` argument as a way of matching. The usefulness of this is clear when, for example, a view wants to redirect the user to another page after some processing. Instead of utilizing URLs, a better abstraction is to find the reverse of the mapping of the view that matches that URL. In the example pictured in figure 4.9, the page is redirected to the view that has the name `photo` in the `mywatson` application namespace, i.e, the `DetailView`, passing the argument as well.

In their essence, views’ primary role is to fetch data from the database – by using the model abstraction – and pass it to the template in order to display it to the user. The most basic view present in MyWatson can be seen in picture 4.10. When the user requests the page http://mywatson.com/mywatson, which corresponds to the first URL pattern in figure 4.8, the `GalleryView` will handle the request. Because the view is a generic `ListView`, this specific class-based view should always return a list of objects. Furthermore, to pass it to the template “mywatson/index.html”, which is specified as an attribute in the view as a string corresponding to the path of the template to display, the function `get_queryset` simply needs to be overridden.

In this case (the view of the photo gallery), all the photos of a specific user are fetched from the database using a Django query, which is an abstraction of normal database queries. Then, in the template, the list of objects – which is received as JSON data – can be accessed using the context object name defined in the view, “photos”, as pictured in the figure 4.11. Once an object is grabbed from the list of objects, its model attributes can be accessed analogously to methods in OOP. Note that, in the beginning, the URL http://mywatson.com/mywatson was matched to the first, empty, pattern because

---

**Figure 4.8:** URL dispatcher used in the MyWatson application

**Figure 4.9:** Redirecting to another view
all patterns in the URL dispatcher in figure 4.8 have mywatson at the beginning of the rest of the URL, excluding the domain name mywatson.com. The reason for this is the “rule” that includes the mywatson app URLconf in the main URLconfig, as it can be seen in figure 3.5. In other words all URLs that begin with http://mywatson.com/mywatson/ are “redirected” to the mywatson URLconf.

When the required data cannot simply be fetched from the database, however, as is the case with face clusters, the view delegates the processing to the MyWatson core, as seen in picture 4.12. First, the view fetches all the face objects in photos that contain people in it. This is done by first executing a query with the keyword “person” and then obtaining the related faces. Then, the view passes the request along to the MyWatson core, coupled with the faces to be clustered, as well as a minimum and maximum number of clusters that the core should try to compute. Finally, some more work is done in order to transform the groups into JSON data so that the template can display the groups accordingly.

 Besides fetching data, views can also update or add new data to the database. An example of this
process is shown in picture 4.13. By using the method `<Object>.objects.create(...)`, the framework executes an SQL INSERT query over the database, receiving as arguments the respective field values.

```
new_tag = Tag.objects.create(tag=tag, score=score, photo=photo, 
  startX=x1, startY=y1,
  endX=x2, endY=y2)
```

Figure 4.13: Creating a new tag

### 4.3 Core

In the core application, each of the four main modules has a specific role. While the main application module works just as a facade, the other modules aim to solve specific CBIR and learning problems that are requested by the user whenever he, for example, uploads a photo, or executes a query.

The main module is a file called `MyWatsonApp.py` (MWA), and essentially has functions that operate like a direct channel for each of the other three modules, i.e. if a view requests the MWA to tag a batch of images, the MWA will simply ask the automatic tagger module to tag the batch of images. Then, the automatic tagger module will return the tags, and the MWA will pass the result along to the view that requested it. The job of MWA is simply to be a facade to the rest of the core in order to provide an abstraction for the Django controller and make it easier to decouple. The real interesting implementation details are present on the other modules, which will be discussed along the rest of this section. In the following discussion, whenever it is mentioned that a view requests a certain module to do something, keep in mind that the request always passes through the MWA.

#### 4.3.1 Automatic Tagger

The automatic tagger module is probably the most important module of the whole MyWatson core. It is what powers the image retrieval by keywords and implements the face detection so it is, consequently, a hard prerequisite for the other two modules. The role of the automatic tagger is to attach textual information that describes the content of the image or is in any way relevant. As previously discussed, the information can come from generic entities present in the image, but also from known logos or landmarks, colors, text or faces.

The problem that the automatic tagger tries to solve can then be described, consequently, as: given a set of paths corresponding to image files, extract high-level keywords that correspond to the content of the image. The underlying steps to solving this problem include obtaining the respective set of image paths, actually tagging the set of photos in an efficient manner, and finally returning the organized results.
There are two events that can trigger the tagging process: the user either uploads a set of photos in the upload page, or requests a re-tagging of a photo. When the former happens, the images are first uploaded and saved on disk, under the user’s respective photos folder. Then, an Asynchronous JavaScript And XML (AJAX) POST request is sent from the browser client to the web server at the URL “mywatson/upload”, signaling that the upload is complete, as a way of indicating that the tagging process can begin. This signal is captured by the view that handles the upload page, the upload_photo view. As opposed to the previously seen views, this one is not a class-based view. Instead, it is a function that receives a request and returns any kind of HTTP response or JSON response. To distinguish between POST and GET request, the view has a conditional branch that asserts the type of method. However, because the upload page also has a form – which also sends the upload data using the POST method –, further checking is required. In this case, another conditional branch is used to check if the POST request contains the signal for the completed upload by accessing the variable upload_done in the POST request, which was previously sent by the JavaScript script in the client’s web browser. As photos are uploaded, photo objects are created in the database and added to a list of images to be tagged, which is then sent to the function that handles the tagging. After the work is done, the signal tagging_complete is returned to the client’s browser as JSON data, indicating that the tagging process has been complete.

On the other hand, if the user requires for a photo to be re-tagged, which is a better way than deleting a photo and uploading it again, the tagging process is also triggered. First, the view responsible for the photo details page, the DetailView, receives a POST request containing a signal retag, sent by the client as the user presses the button to reload the tags. The signal contains the identifier of the photo to be re-tagged in its value. The corresponding photo is fetched, and all the tags that belong to the photo are then deleted from the database. Finally, a list containing solely the target photo is sent to the function that handles the tagging process.

Before asking the core and, consequently, the automatic tagger module, to tag the photos, some pre-processing must be done. The list of photo objects is transformed into a list of paths, and the file’s size is checked again, discarding photos that are too big to be tagged. There are two last steps remaining until the process of tagging is considered completely done: computing a list of tags, and creating the tag objects, which includes assigning them to the photos.

To further improve the efficiency of the whole process, the extraction of tags is also done in batches. In the automatic tagger module, the process of obtaining a list of tags itself is divided into three steps: building the requests for the Google Cloud Vision API, grouping the requests in batches and, finally, requesting the API client to annotate the batches, one by one.

A Google Cloud Vision API client must be requested before any tags can be extracted. This client implements the Python methods that allow MyWatson to utilize the API. The process of building the list of REST requests then begins, one for each photo. The REST request that GCV expects is in JSON
form, so a dictionary is built, containing the image data content – obtained by utilizing an API function for the purpose over the image path – and a list of features to extract. These features correspond to the categories previously discussed in section 3.2. Each element of the feature list is a dictionary and must, at least, contain the field “type” with the value from the list ["FACE_DETECTION", "LANDMARK_DETECTION", "LOGO_DETECTION", "LABEL_DETECTION", "TEXT_DETECTION", "IMAGE_PROPERTIES", "WEB_DETECTION"]. Another useful field to specify is the maxResults, which defines the maximum number of high-level tags of a given category that can be extracted.

After obtaining a list of requests, these are then grouped with batches of 15 requests, as this seems to be the maximum number of requests in a batch that Google’s Cloud Vision API accepts. Additionally, the batch size must not exceed 10485760 bytes, which is the exact number specified in the returned errors whenever a batch is too large.

Finally, the GCV client is asked to annotate each batch of requests, and the responses are aggregated in a list. Then, the responses are organized into a dictionary: each photo, represented by an index which is in conformity with the photo list in the view – i.e. a index with the same number corresponds to the same photo –, is mapped to a list, called info. Each response has high-level information from each of the categories requested, and the info list consists of dictionaries with the fields “description”, “score” and “category”. However, some categories have additional fields, such as the “fraction” field for colors, specifying the fraction of a certain color in the image and “box” for detected faces, indicating the coordinates of the detected face’s bounding box. To summarize: a mapping photo-info is the product of the automatic tagging process, and info is a list of information objects that describe the content of the photo.

Completed the task of the automatic tagger module, given the list of dictionaries containing a mapping photo-info, the tags are then created in bulk, in the view, using Django’s built-in function bulk_create(), which is a lot more efficient when creating multiple objects than the normal create() function. First, the field description is truncated to the tag’s max size, which is, for readability purposes, 100. Then, the creation of repeated tags is prevented, excepted in the case where the tag has the description “person” and has a bounding box. Tags that passed the clean-up process are then created in bulk. Tags that are associated with faces in the image, i.e. tags that have the “person” description and have bounding boxes, are passed to another function, along with the path of the photo, for the creation of the face object and cropping. This is done by utilizing the crop function from the Pillow library – which is a fork of PIL (Python Imaging Library) – and then saving the cropped photo under the user’s face folder, in "/public/media/faces/<username>". Finally, the face object is created and recorded in the database, containing the path of the cropped face, the id of the photo and the associated tag object.
4.3.2 Retrieve and Rank

The work of the retrieve and rank module begins when the user executes a query in the search box, located in the navigation bar. Submitting the query redirects the browser to a "mywatson/results" URL, which is handled by the view QueryResultsView, whose implementation can be seen in the previously presented figure 4.3. First, the view obtains the string of keywords which were sent through a GET request as the value of the argument "q", present in the URL. In other words, when searching for "blue sky", the URL would have the form "mywatson/results?q=blue+sky". After obtaining the string containing the query terms, the view asks the MyWatson core to get the relevant results to that query.

When receiving this request, the MyWatsonApp first checks if the query string is empty and returns an empty list to the view if this is the case, without ever delegating the task to the retrieve and rank module, as no results would be retrieved. Otherwise, the query string is pre-processed, by changing all characters to lowercase and splitting the terms, essentially transforming the query string to a query set. The job of retrieving the results is then delegated to the module.

The objective of the module is to return a list of Django photo objects that are relevant to the given query set. To this end, the TF-IDF model from text-based information retrieval is leveraged, as the photos can be approximated as documents containing words (tags). The zero-th step is to add the stems and lemmas of all the query words to the query set, i.e. if the original query set contains the words “dogs” and “walking”, it will contain “dogs”, “walking”, “dog” and “walk” after the stemming and lemmatization. In other words, some expansion of the query terms is done in order to improve retrieval results at a very basic level, with the idea of including word roots and singular forms.

The retrieval process begins after, by transforming the query terms into a query vector. For each query term, a weight is derived, describing the importance of a term, ranging from 0 to 1, respectively indicating low or high relevance. As a very basic weighting strategy, the original query terms are assigned the weight 1, and the stemmed and lemmatized terms have a 0.7 weight. The idea is that terms directly input by the user should have higher importance than their expanded forms. Analogously, a bag of words model is computed as a mapping between photo objects (documents) and term weights, by first transforming the photo objects into vectors of separated tags – i.e. a tag “labrador retriever” will return the terms “labrador” and “retriever” – and then, for each term, computing its IDF score according to the formula

$$IDF(term) = \log \left( 1 + \frac{N}{N_{term}} \right)$$

where $N$ is the number of documents (photos from the user) and $N_{term}$ is the number of documents in which the term appears. Note that the TF weight is not utilized because it would always be 1, as no term is repeated in a document, and smoothing is utilized in order to prevent an IDF value of 0 in the case
where a term appears in all documents.

The BoW vectors of the collection are then extracted from the previously computed weights: a mapping photo-vector, where the vector contains the IDF weights for each of the terms present in the query vector, or 0 if a query term is not present in the document. The result is a document vector with the same size as the query vector, where elements of the same index in both vectors correspond to the respective weights of the exact same terms.

Then, similarities are calculated, utilizing the cosine similarity function

\[ \text{similarity}(x, y) = \cos(x, y) = \frac{x \cdot y}{||x|| \cdot ||y||} = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2} \sqrt{\sum_{i=1}^{n} y_i^2}} \]

defining the similarity between two term vectors \(x\) and \(y\) is given by the cosine of the angle between the vectors, i.e. the query vector and a document vector. Finally, the results are sorted according to their similarity values in order to display the most relevant photos first, and all the photos that contain at least one of the query terms are returned.

### 4.3.3 Learning

The learning module is the last and most complex module of the MyWatson core. Its job is to, given a list of face objects, compute a configurable number of sets of clusters, each with a different number of clusters, by grouping faces according to their similarity. Analogously to other modules, it is requested by the browser whenever the user is redirected to the URL “mywatson/people”, triggered by clicking the link “People” in the navigation bar. The view responsible for handling this URL, the ClusterView, begins the process by obtaining all the face objects from the respective user and, if no faces were found, returns an empty list, skipping the learning module entirely. The user’s browser will then display a message with this information, much like when a query does not return any results. The range of number of clusters that the learning module will try to compute, \(\text{min\_clusters}\) and \(\text{max\_clusters}\), is usually between 1 and the number of faces, corresponding, respectively, to the cases where a user only has photos of one person and all the photos have a different person in it, i.e. there is not a single repeated face in all the user’s photos. Finally, the view requests the learning module to compute the cluster groups, passing along the set of face objects and the minimum and maximum number of clusters.

The request is received by the learning module and triggers the start of the algorithm that will return a list of clusters in the end. The general idea of the process is to dynamically change the list of values of \(n\_clusters\), the number of clusters to compute. This way, even if the user has, as an example, 10000 photos, instead of trying to compute 10000 clusters, guaranteeing this way that the best cluster will be present — i.e. it is guaranteed that there is a cluster group where the number of clusters corresponds the number of different faces in the user’s photos (optimally, because this also depends on the face
features and clustering algorithm)—, the module will apply a variation of a local search algorithm in order to optimally find the best cluster groups with a high probability. The algorithm acts in accordance with the following steps:

1. The order of the number of clusters in which the module will compute the groups is initialized as a list, called `cluster_try_order`, of three numbers, corresponding to the first, second and third quartiles. The second quartile corresponds to the median of the range of number of clusters to compute, and first and third quartiles correspond to the median of each halves resultant by the division according to the second quartile. These values are calculated according to equations 4.2, 4.3 and 4.4.

   \[
   \text{offset} = \left\lfloor \frac{\# \text{faces}}{4} \right\rfloor \tag{4.1}
   \]

   \[
   \text{middle}_n_{\text{cluster}} = \left\lfloor \frac{\# \text{faces}}{2} \right\rfloor \tag{4.2}
   \]

   \[
   \text{min}_{n_{\text{cluster}}} = \text{middle}_n_{\text{cluster}} - \text{offset} \tag{4.3}
   \]

   \[
   \text{max}_{n_{\text{cluster}}} = \text{middle}_n_{\text{cluster}} + \text{offset} \tag{4.4}
   \]

Then, a balance threshold is also initialized, corresponding to the number of times the score must drop more than it rises for the algorithm to stop computing cluster groups. It is a polynomial function with a degree of one, i.e. a linear function, with a negative slope, and its definition can be seen in equation 4.5. The \( \text{ratio} \) is a negative real value that defines the rate at which the necessary number of clusters decreases with the number of faces, and the \( \text{minimum}_{\text{clusters}} \) is an integer value, describing the minimum number of clusters that should always be computed. The ratio must be a value between 0 and 1, and should not be too high, because if the amount of face images is too large a lot of clusters will have to be computed. For the same reason, the minimum number of clusters should not be too high either. After some informal, empirical analysis, the parameters \( \text{ratio} = -0.25 \) and \( \text{minimum}_{\text{clusters}} = 25 \) were deemed as acceptable, as they allow the computation of all cluster groups in cases where there are not a lot of face images, and still compute a good number of clusters to have a good chance of finding a good maxima when the number of face images is high. For example, if there were 25 face images, all of the cluster groups, with \( n \in [1, 25] \), would be computed, but if there were 1000, only \( 275 + 2 \) (first and last) would be computed, in the best case. Some other variables are also initialized to 0 or \text{None}, such as
the balance and reference_cluster_group which, respectively, keeps track of the ratio between rising and dropping scores and saves the current cluster group that is being used as a reference.

\[
\text{balance\_threshold} = \lfloor \text{ratio} \times \#\text{faces} - \text{minimum\_clusters} \rfloor
\] (4.5)

2. The three initial cluster groups, corresponding to the groups with \(n_1\) clusters within the initial vector, i.e. \(n_1 \in \text{cluster\_try\_order}_0\), are computed, along with their respective silhouette score. From these three, the cluster with the highest silhouette score is chosen as the reference cluster group, i.e. because it has the highest score, it is deemed likely that a decent local maxima is closer to it that to the other groups.

The clustering algorithm is as follows:

(a) Given a set of faces and an integer \(n\) corresponding to how many clusters will the faces be divided into, the algorithm starts by obtaining the features from the face images, which can be retrieved from the face objects. In the case that the features of a certain face object have already been computed, they are simply retrieved from the database. Otherwise, the CNN model, which uses the ResNet50 architecture with no pooling, is used to extract the features. This is done by pre-processing an image in order to make sure that the input image has the format expected by the CNN, e.g. three sets of lists with a size of \((224, 224)\) pixels each, one set for each of the RGB channels. Furthermore, Keras pre-processing functions are utilized, resulting in the image data that is fed to the actual CNN model. As a result of this step, a vector containing all of the face image features is obtained.

(b) An instance of an algorithm in the sklearn.cluster library, the AgglomerativeCluster, which performs hierarchical clustering with ward linkage is then used to predict the clusters of the data points corresponding to the faces, agglomerating them via a bottom-up approach and stopping when the number of clusters reaches the requested value. Furthermore, the average of this group is also calculated using the function silhouette_score from sklearn.metrics, and is returned along with the labels of each data point.

3. After the reference cluster group is chosen, its adjacent neighbours are added to the cluster_try_order list, and their clusters and silhouette scores are also calculated. The neighbour with highest score defines the direction in which the search will be conducted. As an illustrative example, consider that, from neighbors of the reference cluster group \(n = 5\), namely \(n = 4\) and \(n = 6\), the one with the highest score is the group with \(n = 4\) clusters. In this case, the search would be conducted “downwards”, to lower clusters. Finally, all of the \(n\) values corresponding to that direction will be added to the cluster_try_order until their lower and upper limits. In continuation with the previous
example, this would mean that the values 3, 2 and 1 would be added.

4. In the main loop, the added groups are computed. Each time a cluster group is computed — including in the previous steps —, the highest score is updated if that cluster group has higher score. If the score is updated, i.e. if a certain cluster group has higher score than the current highest score, the balance variable is incremented by one. On the other hand, if it is not updated, meaning that the score keeps dropping, the balance is decremented by one. If the balance reaches the threshold, it means that enough evidence has been gathered to indicate that it is not probable that a better cluster group can be found, and the algorithm stops. Otherwise, the algorithm runs until all of the groups in the cluster_try_order have been computed.

5. Finally, in the case that the main loop broke due to the balance reaching the threshold, the limit cluster groups are calculated, i.e. the groups with 1 and \#faces clusters. The list of clusters is then returned to the view and the FaceCluster objects are created and added to the database.

Note that cluster groups are only computed if they have not been computed yet, i.e. if a FaceCluster object identified by a triple (n_clusters, cluster_id, face) is not present in the database. Furthermore, because a user’s cluster groups are deleted whenever he removes a face or a photo, or adds a photo that has a face in it, this scenario will also be triggered.

On another note, the Keras CNN model used by the learning module to extract features from face images is only instantiated once, whenever the Django server starts. The instantiation implies downloading the model and pre-trained weights once, which will then be saved in a JSON and H5 file, respectively. For further instantiations, the model and weights will simply be loaded from their respective files.

![Cluster groups score](image)

**Figure 4.14:** Cluster groups score

In summary, the idea of this strategy is to find the spot were clusters groups most likely have the
highest score, i.e. better model the face images, and then to find the direction in which the score rises, gathering further evidence of the presence of a better cluster group along the way by keeping score of the balance. If no better group has been found for a certain amount of rounds, defined by the value of balance_threshold, the algorithm gives up, preventing the waste of computation time. The initial cluster references are evenly spaced in order to reduce the number of cluster computations needed to gather enough evidence, i.e. so that the initial references have more chance of falling in the best zone of scores. Figure 4.14 depicts the evolution of the silhouette score as the number of clusters in a group increase, during a run of the algorithm in a generic set of 97 face images. Note that the groups with 1 and 97 clusters were omitted, as their score was 0. During the run, the group with the maximum score, with 43 clusters, is found.

The picture further supports the idea that the zones near the quartiles usually do not differ too much from the quartile itself, and the direction from its neighbors is generally monotonic. In other words, in a given direction, even if the score drops a bit, the score is higher overall, as it can be seen by the trendline.

4.4 Front-End

The MyWatson website follows a very simple design, where a fixed navigation bar is always present on the top of the page and the rest of the page displays the respective content. The HTML files are defined in the Django templates, and are divided into two apps: the "core", which does not have anything to do with MyWatson Core but, instead, contains the necessary logic for the registration and log-in, and the "mywatson", according to the file tree previously presented.

The web files present in the core are related to the landing page, where the sign-up and log-in forms are also located. The landing page is the first page that a person sees, and aims at transforming potential users into actual users by focusing on the objective known as a Call to Action (CTA). The main HTML file and some CSS and JavaScript files were downloaded as a template from Colorlib [91], and were then customized and adapted to include Django template language. The landing page can be divided into six different, ordered areas:

1. **Home**: displays the banner with a slogan and a logo. If the user is already logged in, a welcome back message and a button to directly enter the MyWatson web application are also present.

2. **Features**: briefly introduces some of the features of MyWatson, including automatic tagging and face detection.

3. **How to**: explains how to start using MyWatson by showing a series of easy to follow steps that start with the account registration.
4. **Inside**: explains in a bit more depth how MyWatson works from the inside, as well as some techniques utilized in layman’s terms.

5. **Register / Log-in**: this is the Call-to-Action area, where a small registration form is shown, requiring a username, a password, and an e-mail. In the case that the user already has an account, the log-in form can also be toggled.

6. **About**: displays the motivation for the MyWatson system, as well as a small introduction about it’s author. A footer with copyright information about the website template and the logo is present.

![Figure 4.15: MyWatson landing page](image)

After registering an account and logging in, the user is redirected to the MyWatson web application, an independent Django app. The web application currently has four web pages, of which some images were already presented, the first three being accessible through the navigation bar:

1. **Gallery**: displays all of the photos that the user uploaded in a grid-like pattern. Clicking on a photo thumbnail will redirect the user to that photo’s details.

2. **Upload**: a very simple page containing an area to drop or choose images in order to upload them.

3. **People**: contains all of the faces in the user’s photo gallery, divided into groups of variable length and arranged in a grid-like pattern. By default, the faces are divided into the $n_{best}$ clusters, corresponding to the number of clusters that grants the highest silhouette score to the division or the one that the user previously selected by utilizing the save button. On the top of the page is a small control panel, with a slider that controls in how many groups will the faces be divided, the silhouette score of the respective division, and a toggleable button and a save button, both for the edit mode, which allows users to rename and re-arrange the clusters.
4. **Photo details**: clicking on a thumbnail of an image in the gallery or of a face in the clusters will redirect the user to the photo’s details, where the image is displayed in a larger scale – while still fitting the window. Below the image are the tags for that image, either returned by the automatic tagger or added manually by the user. On the left of the image is an “hamburguer menu” which, when toggled, displays options to delete or re-tag the photo, or to add a new tag.

In addition, all pages contain a “mini MyWatson” in the bottom right corner which, when clicked, displays helpful information about the current page, in order to offer some assistance to users that may have trouble understanding some aspects of the interface.

Finally, another relevant aspect of the frond-end logic is the JavaScript functions that regulate some functionality, the most complex one being the script that controls how the clusters are displayed. Because the data is received in a JSON format and is encoded, a JavaScript function is needed to decode the JSON string and transform it into a dictionary of cluster groups as a mapping from $n_{clusters}$ to another mapping, of $cluster_{id}$ to lists of photos, as well as sorting the slider values by descending silhouette score.
# Evaluation

## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Heuristic Analysis</td>
<td>78</td>
</tr>
<tr>
<td>5.2 User Feedback</td>
<td>80</td>
</tr>
<tr>
<td>5.3 System Limitations</td>
<td>82</td>
</tr>
</tbody>
</table>
Table 5.1: Heuristic evaluation of MyWatson’s UI

<table>
<thead>
<tr>
<th>Heuristic</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Visibility of system status</td>
<td>5</td>
</tr>
<tr>
<td>2. Match between system and the real world</td>
<td>4</td>
</tr>
<tr>
<td>3. User control and freedom</td>
<td>3</td>
</tr>
<tr>
<td>4. Consistency and standards</td>
<td>5</td>
</tr>
<tr>
<td>5. Error prevention</td>
<td>5</td>
</tr>
<tr>
<td>6. Recognition rather than recall</td>
<td>5</td>
</tr>
<tr>
<td>7. Flexibility and efficiency of use</td>
<td>3</td>
</tr>
<tr>
<td>8. Aesthetic and minimalist design</td>
<td>5</td>
</tr>
<tr>
<td>9. Help users recognize, diagnose, and recover from errors</td>
<td>2</td>
</tr>
<tr>
<td>10. Help and documentation</td>
<td>5</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>4.2</strong></td>
</tr>
</tbody>
</table>

The evaluation methodology for the MyWatson web application is essentially comprised of user feedback, collected after the development of MyWatson was complete. Additionally, an heuristic inspection is performed as a way of evaluating the usability of the user interface, based on the well-known heuristics by Jakob Nielsen [92]. The latter analysis will be discussed first, and then the user feedback will be presented and compared against the usability evaluation.

5.1 Heuristic Analysis

Heuristic analysis’ objective is to identify usability problems along the user interface of a given application. The final ten Nielsen’s heuristics that were solidified in 1994 are still utilized as a reference to this day, and will be the ones used on this thesis as well, in order to give a formal analysis of MyWatson’s user interface. For each of the ten heuristics, a rating between 0 and 5 is given in table 5.1, corresponding to “does not comply” and “fully complies”, respectively.

The reasoning behind the presented rating for each of the given heuristics is as follows:

1. **Visibility of system status**: the user is always informed with detail, via non-technical language, of what is going on behind the scenes, by real-time messages that display while a given task is being executed. This is seen in the upload page, while the photos are being uploaded and tagged, and also when both tasks are finished. This is also seen in the clusters’ page, while the cluster groups are being obtained, whether by retrieving existent ones or by computing them.

2. **Match between system and the real world**: the system almost always uses non-technical words, either in message prompts, buttons or other types of information. The possible exception to this is in the clusters’ page, where the word “cluster” is used instead of the word “group”. However, I do not consider this a relevant issue, and the word “cluster” is utilized because I want the association with unsupervised learning techniques to be clear. Additionally, another small issue is the fact
that the cluster numbering starts at 0 rather than 1, and this is uncommon outside the scope of computer science.

3. **User control and freedom**: although there is not a lot of room for mistakes, as error prevention measures are taken in a lot of scenarios, eventual ones cannot be undone, at this point. For example, the deletion of a photo or a tag cannot be undone. Similarly, any changes to a cluster group cannot be undone once saved.

4. **Consistency and standards**: consistency is present in almost all of the information displayed at any page. For example, in both confirmation prompts when deleting photos and tags, the buttons are in the same order: “Yes” on the left, “No” on the right. This also true for the task of re-tagging a photo. This could be extended, however, to the words “label” and “tag” which, although meaning the same thing in this context, may leave the user wondering if this is the case. On the other hand, common platform standards are also followed, for example, in the image gallery which displays the thumbnails in a grid-like fashion, similarly to other websites such as Facebook. The “hamburguer menu” in the photo details page is also a common sight in other popular websites.

5. **Error prevention**: common mistakes such as the user misclicking the button to delete a photo or a tag are prevented by displaying a confirmation message, asking the user to confirm the action that he is about to request. This is done in the most crucial cases where the loss of data is imminent which, although can also happen when the user makes changes to the cluster groups, is not so relevant then.

6. **Recognition rather than recall**: rather than requiring the user to remember how actions should be performed, information is repeated when necessary, preventing the user from having to remember it. Easily recognizable buttons further help the user understanding what to click in order to perform a specific action, as is the case with the “trash can” or “plus” buttons in the photo details, which respectively delete the photo and add a new tag. Furthermore, hovering such icons displays a tooltip containing additional textual information.

7. **Flexibility and efficiency of use**: although tasks were developed with the objective to be as efficiently as possible in mind, there are currently no shortcuts for advanced users to perform them even more efficiently, such as keyboard shortcuts – hotkeys – or steps that can be skipped. In reality, this is not very relevant at this point, as currently existent actions are already simple to perform and accelerators are not considered a requirement.

8. **Aesthetic and minimalist design**: during the development of the website, I tried to achieve an aesthetically pleasing user interface. However, this is not easy to objectively evaluate by myself and, consequently, will be done by the users themselves. On the other hand, I believe that a
minimalist design was achieved, where only the relevant information is displayed, such as simplistic icons or messages.

9. **Help users recognize, diagnose, and recover from errors:** errors such as the lack of photos or people in the photos are well diagnosed and presented to the user, respectively in the gallery and in the cluster group page. However, other rare internal errors, such as the Google Cloud Vision API client failing, are not displayed to the user and instead an error with code 500 is presented. Not much time was spent in displaying and treating errors, because the focus was on developing the application in order to work as intended.

10. **Help and documentation:** generally, the user interface was developed so that it can be fully navigated and the system utilized just with the information displayed. However, in case a user is lost or does not understand the purpose of a specific page or action, additional information can be displayed by clicking the small MyWatson logo on the bottom right of most pages. The information shown includes the purpose of the page, as well as the functionality and possible actions that can be taken.

This evaluation was done as critically and objectively as possible, and the results reported in the same manner. The results of the user feedback should provide even more objective criticism, as well as additional suggestions for future improvements.

### 5.2 User Feedback

The user feedback was collected via online forms from Google Forms. The users were asked to experiment with the MyWatson app for a while and perform some tasks, such upload photos, search for photos or analyze the people cluster groups. As such, users were required to have an account to do so. Although the process of registration is very simple and quick, potential users might have been reluctant in creating an account just for trying a website and experimenting for 10 minutes. In order to mitigate this issue, a testing account was previously created, so that users that found registering for an account too troublesome could skip that step and test MyWatson. Some generic photos were uploaded to the testing account beforehand, such as pictures of beaches, cats, dogs, streets and celebrities, to name a few. Additionally, users who used the testing account were encouraged to add more photos and asked not to remove any. This was largely agreed to, as no photos were removed during the testing phase and a lot of additional photos were uploaded.

A sample of the MyWatson application feedback form is present in appendix A. It is comprised of 9 questions, 2 of which ask for the user’s to write their opinion and additional observations and thus are not required to be answered. In summary, users are asked their opinion about the user interface, the
usefulness of MyWatson and their potential use and recommendation. Furthermore, users could also
share improvements that they consider would better meet their needs. The charts corresponding to the
results of the surveys is present in appendix B.

There were 27 replies to the questionnaires, from which the great majority’s age is comprised be-
tween 22 and 24 years, inclusive. Respectively, 30%, 37% and 15% from age 22, 23 and 24, totalling
82%. The maximum age of the questionees is 36, while the modal age is 23, and the mean age is 24.

Users were asked to essentially quantify the usability MyWatson application from 1 to 5, respectively
the worst and best score. This can be specifically observed in two questions, where the users were
asked, respectively, to rate how easy the application is to use, and to give an overall rating to the user
interface. Overall, most users agree that the interface is good and simple to use, as seen in charts B.5
and B.2, respectively, where the overall ratings 4 and 5 were given to the UI by respectively 48% and
44% of the inquired, and 67% gave the maximum rating to the ease of use. These results corroborate
the heuristic evaluation of the previous section, where the user interface was also considered to be
adequate in terms of usability.

The reports also indicate that users consider the automatic labelling of photos the most useful fea-
ture, with 48% votes, while the second most useful feature being the grouping of people by faces, with
37%, as seen in the chart B.4. The search feature is considered the most useful only by 15% of the
inquired. These results are consistent with the fact that the face clustering, while not being considered
the most useful, is still probably interesting to use from a user’s point of view. Furthermore, the automatic
labelling of photos was expected to be considered the most useful, as it is the cornerstone of the entire
system.

On the other hand, the results indicate that users would not utilize the application frequently, as seen
in chart B.3, where a third of the inquired replied with a 3 on a scale of 1 to 5 to the question “How
often would you use MyWatson?”, with a close 30% giving a rating of 4, and only 11% gave a rating
of 5. Despite this, it is still a positive majority, compared to the 7% and 11% that answered 1 and 2,
respectively. These ratings might be evidence of the fact that the application is not meant to be used
very frequently, as the objective is to upload photos and let MyWatson manage them automatically,
without having to worry about them too much. The charts B.6 and B.7, respectively showing the rating
given by the users about MyWatson usefulness and how likely they would recommend it, might be further
evidence that corroborates the fact that users are overall pleased with the system, with 37%, 41% and
19% of the inquired giving ratings 3, 4 and 5, respectively, to the former question, and with 74% of the
users replying affirmatively to the latter.

Finally, inquired were also asked “How can MyWatson be improved to better meet your needs?” and
to input additional observations. Suggestions include the development of a smartphone application or a
computer program, the introduction of the most used tags, and the possibility of adding the geographic
location to a photo and creating albums, to name a few. One user notes that the black outline of the thumbnails in the photo gallery might feel a little heavy on the eyes, and another inquiree mentions that he did not consider the user interface in the people’s page very intuitive, as he did not understand at first that the cluster groups were ordered by score and also expected dragging a face outside of a cluster would create a new cluster. These observations lead me to believe that the possibility of customization of the user interface would be an interesting feature to add, along with the development of a mobile application.

5.3 System Limitations

Up to this point, the known limitations of the MyWatson system include the lack of customization of the user interface and the lack of possibility of creating albums or new clusters, as pointed out by the users. Another limitation is the number of photos that can be uploaded at once: while the face clustering algorithm is scalable and the automatic tagging is done in the most efficient manner using batches, the process of uploading and tagging the photos when a lot of files – informal experiments revealed that the number is at least 200 – are introduced at once is very slow and might ungracefully terminate the system.

Additionally, these APIs, despite speeding up the process of development and leveraging their advantages, also include their limitations. Namely, Google Cloud Vision API still has some limitations, such as the fact that the extracted tags correctly label dominant objects in the image, but do not label every object of interest in a photo, as also noted in an article by Chad Johnson [93]. Despite the article being from 2016, the results can be reproduced with the current version of the API: the image should, objectively, contain the tags “apple” and “bat”, but GCV does not output them, as pictured in figure 5.1.

![Figure 5.1: Google Cloud Vision API label limitations](image)

However, despite its limitations, its clear that Google Cloud Vision is evolving. In 2017, Hosseini et al. [94] reported that the API was not robust to noise, i.e. it could not extract the correct labels from pictures with a small degree of noise introduced. Experiments included inputting two pictures: one normal picture and other being the same picture but with noise. The same picture of a house would
output the labels “Property” and “Ecosystem”, respectively the normal and the noised picture. However, the experiment was reproduced with the same exact same images and the results were correct this time around, despite a small change in the extracted tags, as seen in figure 5.2. The same experiment was repeated with the same photos utilized in the experiment by Hosseini et al., and the same results of noise robustness were achieved.

(a) A picture containing a house                      (b) Same picture with 15% impulse noise

Figure 5.2: Google CV API noise experiment
Conclusions
This thesis aimed to present a practical solution to the problem of Content-Based Image Retrieval. To this end, the development of a modular application that provides an easy and simple way of managing photos was discussed. The MyWatson system leverages the Google Cloud Vision API for automatic tag extraction, including objects, landmarks, colors and others, as well as faces and their coordinates that are present in the content of an image. Detected faces then have their features extracted by a convolution neural network model, pre-trained on a large set of faces and built on top of the Keras API using the Tensorflow backend. Face features are then clustered together according to their similarity using a hierarchical clustering algorithm, generating many cluster groups with different numbers of clusters, corresponding to the hypothetical number of faces in a group. Cluster groups with higher silhouette score correspond to stronger and more probable hypotheses, which are then shown to the user as a way of further enhancing the management of the user's photo library.

Despite utilizing Google Cloud Vision API and the Keras CNN as a black-box, the base ideas and techniques for CBIR, face detection and convolution neural networks were discussed in chapter 2, providing a minimal understanding of the general processes when leveraging such high-level APIs. Additionally, several options for clustering were presented, including metrics to evaluate the quality of the cluster groups in order to approximate the number of clusters to the real number of faces that are presented in the whole photo gallery of a given user. The utilized APIs were also further discussed in chapter 3, including their general motivation, architecture and features, where a small discussion on the Django framework was presented, as it is a very important piece of the presented solution, including the models and views that are contained in the project.

MyWatson's architecture was also explained in-depth, starting by an overview of the system as a whole. Then, each module was analyzed in detail for common operations that the user can perform, including uploading photos and clustering the faces.

Finally, an heuristic analysis was performed over the MyWatson's user interface, according to the important and well-known Nielsen's heuristics for usability. This evaluation was corroborated by the user's opinion on the user interface, extracted from the answers to the questionnaires that 27 users filled in after experimenting the web application hosted publicly. The inquirers also expressed the usefulness of MyWatson, as well as further improvements to the application and user interface. Overall, the objectives discussed in the introductory chapter have been achieved.

Looking at the user's opinions for improvement and the limitations of the system, also discussed in chapter 5, future work should include the inclusion of meta-data to the photos, such as titles, location or the date, which might further help the searching feature. In the scope of further enhancing the image retrieval, which is limited by the tags extracted by the Google Cloud Vision API, query and tag expansion techniques should be a priority, as the low number of tags and information can generate additional relevant information when adding similar or related keywords to both the tags and the query.
For example, when searching for “sun”, related concepts such as “sunny”, “day” or “star” should also be taken into account, with their respective weights adjusted. Developing a better interface with enhanced control is also an important direction to take, as the creation of albums and the possibility of adding titles or caption to the photos should help the management aspect of a photo gallery. The possibility of specifying in how many clusters should the faces be divided into, the creation of new clusters, or a more fine-grained control of the face cluster groups should also further empower the user and further enhance the management element. Finally, the development of a smartphone application should also provide interesting results for the users, as photos could be directly uploaded using the phone.
Bibliography


[84] F. Seide and A. Agarwal, “Cntk: Microsoft’s open-source deep-learning toolkit,” in Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data


MyWatson Application Feedback
Form Sample
MyWatson Application Feedback

MyWatson is a web-based application that provides easy management of a photo collection in the form of search, automatic labeling of photos and face clustering. Think of it as your personal assistant for photos!

Please try it at:

http://192.92.147.76:8000/

and then answer this survey, which takes less than 5 minutes to answer.

NOTE: The website was developed for Google Chrome, so it will probably not work that well on another browser.

NOTE: You need to register to access MyWatson, and upload photos to experiment with it, but if you prefer you can use the following test account which already has some photos:

Username: TestAcc
Password: testing123

You can also upload more photos to this account if you want. Please do not delete any photo!

Thank you for your time!

* Required

1. How old are you? *

2. How easy is it to use? *

   Mark only one oval.

   1 2 3 4 5
   Complex   Easy

3. How would you rate the User Interface design? *

   I.e., is the website pleasing to look at, does it feel clunky or messy, etc.
   Mark only one oval.

   1 2 3 4 5
   Terrible Excelent
4. How would you rate the usefulness of MyWatson? *
   *Mark only one oval.
   
   1  2  3  4  5
   
   Useless  ○  ○  ○  ○  ○  Essential

5. Which MyWatson feature is the most useful for you? *
   *Mark only one oval.
   
   ○ The automatic labeling of photos
   ○ The grouping of people by faces
   ○ The search
   ○ Other: ____________________________

6. How often would you use MyWatson? *
   *Mark only one oval.
   
   1  2  3  4  5
   Never  ○  ○  ○  ○  ○  Every day

7. Would you recommend MyWatson to your colleagues and friends? *
   *Mark only one oval.
   
   ○ Yes
   ○ No
   ○ Maybe

8. How can MyWatson be improved to better meet your needs?
   I.e. by adding new features, different platform, etc.
   
   ____________________________________________
   ____________________________________________
   ____________________________________________
   ____________________________________________

9. Additional observations:
   
   ____________________________________________
   ____________________________________________
   ____________________________________________
   ____________________________________________
MyWatson Application Feedback

Results
Figure B.1: Bar chart with user's ages

Figure B.2: Bar chart with how easy users consider MyWatson is to use
Figure B.3: Bar chart with how often users would use MyWatson

Figure B.4: Pie chart with the most useful feature according to users
Figure B.5: Bar chart with the users’ opinion about the UI

Figure B.6: Bar chart with how useful the users consider MyWatson
Figure B.7: Pie chart with the likelihood users would recommend MyWatson
Automatic Tagger API Comparison

Figure C.1: The images utilized in the comparison
**Figure C.2:** An informal comparison of different Automatic Tagger APIs