Universitatea POLITEHNICA din București

Facultatea de Automatică și Calculatoare,
Departamentul de Calculatoare

LUCRARE DE DIPLOMĂ

Sistem de recunoaștere a siglelor

Conducători Științifici:
Șl.dr.ing. Răzvan Deaconescu
Ing. Tudor Mihordea

Autor:
Flavius–Valentin Anton

București, 2015

University POLITEHNICA of Bucharest

Faculty of Automatic Control and Computers,
Computer Science and Engineering Department

BACHELOR THESIS

Pipelined Logo Recognition System

Scientific Advisers:
Șl.dr.ing. Răzvan Deaconescu
Ing. Tudor Mihordea

Author:
Flavius–Valentin Anton

Bucharest, 2015

I would like to express my very great appreciation to
Tudor Mihordea, who provided me with regular feedback
and valuable suggestions during the implementation stage.
His willingness to give his time so generously has been
very much appreciated.

I would also like to extend my thanks to Răzvan Deaconescu
for his constructive suggestions during the writing of this
thesis and, finally, I would like to thank my parents for
reminding me, with very high frequency, that I have to
write and finish my thesis on time.

Abstract

In this thesis we propose a pipelined, distributed system that allows processing of images originating from online social networks in order to recognize logos and build statistics upon them with the general purpose of enriching an existing social media analytics platform. Therefore, we have built several stand-alone components logically organized as a processing pipeline and communicating with each other using messages. The logo recognition algorithm used in this implementation is based on pattern matching and has been tested on images coming from the Twitter public stream.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

In this chapter we introduce the subject of this thesis, which is the social media analytics, and outline the motivation of this study as well as the aims and objectives, all these being followed by examples of uses cases where a project like the one we have build would be a valuable addition.

1.1 Motivation

Social Media has become an integral part of modern society, thus many activities in the software industry emerge around it. In the last few years, it has been observed that a social network like Facebook or Twitter started to overcome their initial purpose of easing access to information sharing or connecting people in the online environment. One relatively new purpose of social networks is using them as an advertisement platform. This *use case* is not that hard to imagine if we make a reference to the latest digits that social networks have to bear these days.

According to an online statistics portal[7] Twitter has passed 300[2] million monthly active users whereas Facebook is just 50 million users under the 1.5 billion barrier, which is roughly one fifth of the total world population. Needless to say, Instagram, Google+ or even Tumblr are over the 200 million monthly active users barrier. Moreover, there are 350 million new photos uploaded every day on Facebook[3] and 70 million on Instagram[4]. Twitter also faces more than 600 million tweets in a normal day of which approximately 172[5] million contain images.

With these rapidly increasing numbers, naturally comes a new area of development and research called *social media analytics*, which is "the practice of gathering data from blogs and social media websites and analyzing that data to make business decisions. The most common use of social media analytics is to mine customer sentiment in order to support marketing and customer service activities"[6]. Understanding social media analytics gives companies a glimpse about how their products and services are perceived by clients and potential clients. For example, most of the people using a social network share content about their own personal life, their thoughts, images and videos from their everyday life. All this content can be used by companies to analyze how well they are performing and try to improve their services (i.e. offer better advertisements based on that user generated content).

[2] http://www.statista.com/statistics/272014/global-social-networks-ranked-by-number-of-users/
[3] www.businessinsider.com/facebook-350-million-photos-each-day-2013-9
[4] https://instagram.com/press/
[5] Numbers determined experimentally while working on this project
[6] http://searchbusinessanalytics.techtarget.com/definition/social-media-analytics

That being said, in the last few years there was an increasing demand in the industry for tools of analyzing unstructured data found on social networks. These tools are specifically designed and built to consume as much data as they can from the social platforms, process it and offer a variety of representations (such as graphs, charts or tables) with aggregated and meaningful results. In this way, a client company not only can find out how much user reach it has, but also the general feeling about their services. For example, the presence of a company logo in a tweet or Facebook post is a strong indication that the user made a reference to a service or product offered by the respective company, such that an analytics tool that has a NLP (Natural Language Processing) component will feed that component with additional information (i.e. it is very likely that the text is related to our company) so it can be more effective.

1.2 Objectives

In this thesis we propose a system that focuses on logo mentions on social networks streams. Although the tool is designed to work and run independently, its major power would be paired with an existing social media analytics platform, like the one that uberVU via Hootsuite[1] offers. The project was designed and built from scratch in order to enrich their existing tools by adding statistics about logo mentions and was done in collaboration with engineers from Hootsuite.

One of the main objectives is to offer a very scalable system that accepts multiple data sources (social media streams), identifies posts with images and process those images in order to find logos, all in real-time. Before going next, let us be clear with two apparently exhaustive terms used here: *very scalable* and *real-time*. By a very scalable system we understand a software program that, if offered N[2]% more machines (processing nodes), will perform approximately N% faster. Of course, we say *approximately* because we cannot ignore Amdahl's Law[3]. Probably not as intuitive, by *real-time* we understand, in this case, a system that can fetch and process data from a streaming endpoint of a social network without falling behind. To be even more clear, if Twitter offers 1% of total tweets, that gives us approximately 6 million tweets to process in a day. A simple math yields that this system has to process around 420 tweets per minute, or, more interestingly, 30 images per second only from Twitter.

Another objective of this project is to build a modular system with loosely coupled components, so that a new component can be inserted into the pipeline at any future time without too much effort. With this in mind, this system is designed as a chain of smaller self contained systems that can function independently and only require an input stream of data and produce an output stream of data. For example, one component can be the one that fetches data from the social networks and outputs only relevant posts, in our case the ones that contain images. The key objective here is that we can add a new component in-between two existing ones without changing them.

Coupled with the above statements, the system not only has to identify logos in downloaded images, but it can do any type of image processing. Although it may seem so, the primary focus of this project was not building a tool that only recognizes logos in social media streams, but a tool that can do any type of processing with social media streams, providing it has a dedicated component that does the work. Because we obviously needed a component of this type in order to have a fully functional pipeline and because it is indeed useful for social media analytics, a significant part of this thesis concentrates on the logo detection component, but the idea here is that the bigger purpose is the actual pipelined infrastructure.

[1] https://hootsuite.com/products/social-media-analytics/ubervu-via-hootsuite
[2] Where N is an arbitrarily chosen number
[3] http://home.wlu.edu/~whaleyt/classes/parallel/topics/amdahl.html

1.3 Use Cases

From the user perspective, we designed the system far from being verbose, the only visible output that it produces being a real time web dashboard that offers various statistics about detected logos built on top of annotated data stored in a database that gets polled once in a while. Nothing very complicated, all the magic happening all the way from fetching web streams to storing identified data in the database, but it is supposed to be like that, the system aiming to be a smaller part in a large dashboard for a complete analytics tool.

Another use case is, for example, to couple this system (or even add a separate component) with another one that does object or scene recognition, so that the upper layers of analytics have a better insight on the context that the logo was identified. It is clearly an advantage to know that Coca Cola's logo appeared on a 330ml can that takes 50% of the image or on some random guy's T-shirt that points a bended iPhone 6[1] to the camera and looks angry.

Along these lines, we can also imagine a company that recently launched a new product and wants to know how much buzz has created around that product, especially on social networks. It is true that there are many factors that matter here, like user sentiment analysis, for example, but if we only see a recent spike in the number of company logo occurrences on social media streams, it can be an indicator that people are talking about it, that they know about the product. Of course, it is not enough to have a decisive conclusion, but combined with other tools we can definitely have richer results.

From the technical point of view, we designed the system to have multiple, usually replicated, components that run as background jobs and do not do any interaction with a human user. Communication between components is done using JSON objects, such that the components are completely decoupled. This facilitates running the whole system in a cloud environment, like Amazon EC2[2]. So, using the system consists in starting some jobs and using the graphical interface that polls a database that gets updated every time we have some new results (identified logos in our case).

1.4 Thesis Summary

We have seen in the previous three sections what was the motivation of building this project, what objectives we concentrated on and we also saw a couple of situations where it would be useful to have a logo recognition system.

In the next chapter, *Related Work*, we talk about some interesting algorithms and ideas that are currently emerging in the logo recognition world. We explain that logos have very specific features that sometimes may seem useful (i.e. most of them have 2-3 colors and those are usually primary colors), but in fact are rather unhelpful.

In the third chapter we present the detailed implementation of the system, with deep insight in every component as well as some general facts about a messaging system used to pass around information between components. In addition, we talk briefly about Scala, the programming language used in our implementation and we offer a detailed description of the algorithm used for logo recognition.

The fourth chapter focuses on testing and evaluation of current solution. We will crunch some numbers regarding performance, machine costs, scalability or even a precision and recall comparison.

[1] http://goo.gl/2w3Lwz
[2] https://aws.amazon.com/ec2/

The last two chapters present the conclusions and some ideas of future improvements to the present implementation, not only from the algorithmic point of view, but also regarding the underlying infrastructure.

Chapter 2

Existing Work

Like we saw in the previous chapter, social media analytics has become more and more important, therefore the businesses centred around it have gained a lot more attention because they produce a lot more impact. Over the past few years, whole companies like Hootsuite or Sprout Social[2] emerged around the social media ecosystem offering a large variety of tools not only for analytics, but for optimizing social media experience in general. Not unexpectedly, gigants like Google or Facebook have built tools that help their users to better understand how they perform on their very own social network.

However, all these above solutions can be called rudimentary, at most, when it comes to logos. Most of them do not even take logos into account, while others have recently started to do some work in the domain. Of course, there is already some progress done in the industry and it seems quite professional, but its main disadvantage is that it is proprietary technology. The solution that Ditto Labs, Inc[3] is promoting is one of the very few in the industry and seems promising offering visual search in social media streams, but it is not open source so we cannot say much about their approach.

In the next section we will explore one of the most innovative applications of image recognition in social media, while for the rest of this chapter we will concentrate on two techniques of logo recognitions, one of them featuring many written papers in the Computer Vision domain.

2.1 Image Based Analytics in Social Media

One of the new startups in the social media analytics industry is Curalate[4], which, according to their claims, is the world's leading marketing and analytics suite for the visual web. "Their business started around Pinterest and tried to analyze Pins for their potential to expose campaigns and products. As they began to build the business, they realized many people do not use text on Pinterest as they do on other social media sites and therefore needed to figure out a way to specifically analyze images on their own"[5].

Curalate's main idea is that people started to prefer communicating using images, especially on social networks dedicated for this, like Pinterest or Tumblr, and so they try to use object recognition techniques to offer better insight to their clients. Curalate CEO, Apu Gupta, said in an interview: "Before people buy, they save an image of a product and pin it on Pinterest

[2] http://sproutsocial.com
[3] http://ditto.us.com
[4] http://www.curalate.com
[5] http://www.psfk.com/2014/06/curalate-social-image-analysis.html

or post it on Tumblr. After they buy, they might take a selfie with their new sweater and post it on Instagram."

One true thing that stands for all the work done in the domain is that none[1] of the companies involved released an open source platform that allows multiple types of social media image processing, not only logos or objects.

2.2 Object and Logo Recognition

Let us dive now in the realm of object recognition as a stand alone activity, not paired up with social media or marketing purposes. Fortunately, the Computer Vision is a well studied and researched field and there is constant progress in the area. Before proceeding, let's be clear with the term *object recognition* which can be defined as the "process for identifying a specific object in a digital image or video. Object recognition algorithms rely on matching, learning, or pattern recognition algorithms using appearance-based or feature-based techniques. Common techniques include edges, gradients, Histogram of Oriented Gradients (HOG), Haar wavelets, and linear binary patterns."[2]

In the next sections, we will see some of the techniques used for object or logo recognition together with advantages and disadvantages. Before that, let's see what is different in recognizing a logo compared to recognizing an object, because if there were no differences we could simply use any type of object recognition approach and apply it in our case.

The first one that comes to mind, if we look at Figure 2.1, may probably be the *color*. One of the most definitive characteristics of a logo is that its color changes very rarely, or, if it actually changes, we can almost consider it to be another logo. There are brands that change their logo color drastically over time (i.e. Apple), but that is not the common case. Another important fact about the color is that not only there are few or inexistent changes, but there are few colors that appear in the logo itself, to begin with. If we take the Twitter, Dropbox or Honda logos, we can clearly see that they are made of one single color, maybe with slight gradients.

Figure 2.1: Random Logos

[1] To the best of our knowledge
[2] http://www.mathworks.com/discovery/object-recognition.html

Even the Metro Goldwyn Mayer logo, which seems very different compared to the other 7 logos, has one dominant color. Exceptions from this rule make Google and the old Apple logo, but even though they comprise of several colors, these are mostly primary and they do not overlap, so we can see a very abrupt transition from one color to another, one that do not usually occurs in a natural environment.

This observation brings us to the next major difference between a logo and a random object, the *shape*. While an object, despite being the same type of object, can have different shapes and colors (imagine that we are trying to do facial recognition), a brand's logo is always the same. It is one of the key ideas in using a logo that it never has to change in order for people to remember and recognize it easily, so we can also use it to our advantage. Therefore, we can assume that a specific logo, like the Google one, for example, will appear with the same font, the same aspect ratio, the same amount of space between its letters and so on and so forth. In some cases, it is even illegal to change the aspect of a logo, so we can rely on this and use it to our advantage.

That being said, we can also do another important observation regarding logos: they are very rarely, if at all, rotated. There are cases when we have to analyze an image with a Coca Cola can that is flipped on one side so the logo is also rotated 90 degrees, but these are situations that happen in less than 10% of the images used to build this project. With this in mind, we can safely state that considering logo rotation is not the biggest concern regarding the scope of this thesis.

To sum up the above observations, we have seen that three features of a logo usually stay the same across multiple occurrences: *color*, *shape* and *orientation*. Taking this into consideration, when we speak about logo recognition a natural idea should come to mind: *Template Matching*.

2.2.1 Template Matching Approach

This technique is one of the simplest, yet effective ones used in this area of study. In template matching, the query image is matched against an existing template according to a similarity measure, for example the Euclidean distance. "The templates can be represented as intensity or color images, when the appearance of the object has to be considered. Appearance templates are often specific and lack of generalization because the appearance of an object is usually subject to the lighting condition and surface property of the object. Therefore, binary templates representing the contours of the object are often used in object detection since the shape information can be well captured by the templates. Given a set of binary templates representing the object, the task of detecting whether an image contains the object eventually becomes the calculation of the matching score between each template and the image."[10]

One place where we can find a good implementation of the template matching algorithm is the open source computer vision library, OpenCV[11]. We will see later, in Chapter 3, that we are actually using the OpenCV C++ library within the logo detection component. As Figure 2.2 shows, the main idea here is to have a template smaller than the actual query image (in case we find ourselves in the trivial case where the template is bigger than the query image it means there is no possible match) and try and fit the template in every possible position on the original image. If the query image is big enough and the sliding box small compared to the former, it is not that hard to notice an obvious flaw of this approach: it can be very slow in some cases. Fortunately, in real life situations, there is usually no need for matching at high resolutions, so both images can be safely resized to much smaller sizes as long as we do not lose too many details, especially in the template image.

In order for the template matching approach to work, there is a need for a good matching function. In a real implementation, like the one in the OpenCV library, a third image, sometimes called a *result image*, is obtained after the sliding step and, on that image, we look for the *best*

Figure 2.2: Sliding Template[6]

match. When we slide the template on the image, and by sliding we understand moving the box one pixel at a time from left to right and up to down, we compute a value for the same pixel in the result image based on a metric. So, for each pair of pixels (x, y), where we overlapped the template image T over the query image I, we set the corresponding (x, y) pixels in the result image R according to the match metric.

The OpenCV library exposes several metrics to be used together with the `matchTemplate` function of which four of the most interesting ones are:

- `CV_TM_SQDIFF`

$$R(x,y) = \sum_{(x',y')\in T} (T(x',y') - I(x+x',y+y'))^2$$

- `CV_TM_SQDIFF_NORMED`

$$R(x,y) = \frac{\sum_{(x',y')\in T}(T(x',y') - I(x+x',y+y'))^2}{\sqrt{\sum_{(x',y')\in T} T(x',y')^2 \cdot \sum_{(x',y')\in T} I(x+x',y+y')^2}}$$

- `CV_TM_CCORR`

$$R(x,y) = \sum_{(x',y')\in T} (T(x',y') \cdot I(x+x',y+y'))^2$$

- `CV_TM_CCORR_NORMED`

$$R(x,y) = \frac{\sum_{(x',y')\in T}(T(x',y') \cdot I(x+x',y+y'))^2}{\sqrt{\sum_{(x',y')\in T} T(x',y')^2 \cdot \sum_{(x',y')\in T} I(x+x',y+y')^2}}$$

As the four formulas above show, we have, in fact, only two types of metrics, the other two being only the normalized forms of the formers. The first two, `CV_TM_SQDIFF_NORMED` and

CV_TM_SQDIFF rely on the squared difference between the value in the template image and the original image. If we think of what a good match would mean in this case, namely when the two pixels are as close as possible, than it means that the smaller the value is, the better the match we have. So, in the case of the first two methods, all we have to do is look for the smallest value in the result image in order to find the best match.

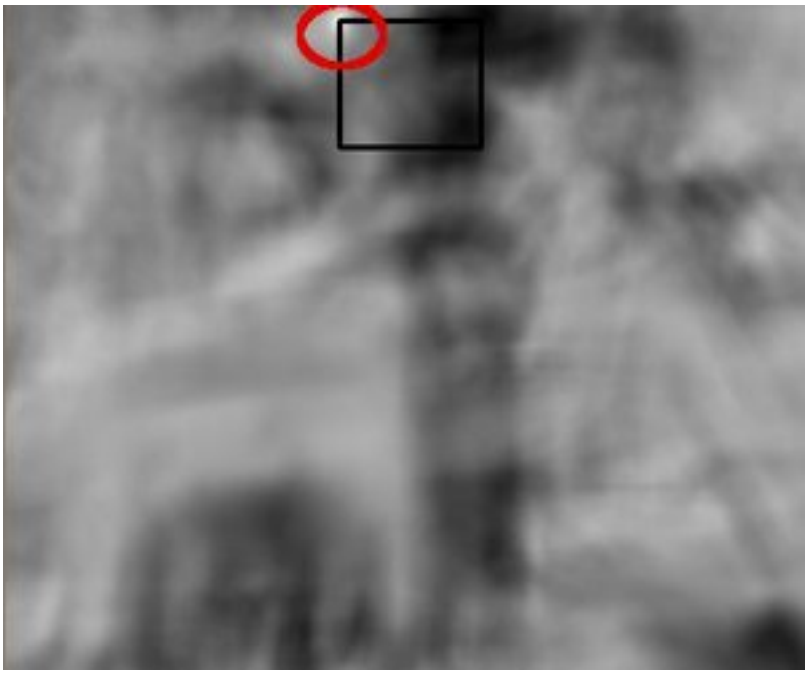

Figure 2.3: Result matrix R obtained with a metric of CV_TM_CCORR_NORMED[6]

In Figure 2.3 we illustrate the result matrix obtained after the sliding procedure shown in Figure 2.2 using the CV_TM_CCORR_NORMED metric. Contrary to the squared difference metrics, the cross correlation methods yield the best match when the result value is at its maximum, in our case the small white spot inside the red circle. Once we have determined the location of the best match, all that remains to be done is to determine the contour of the template in the original image. This can be easily achieved knowing the size of the template and the fact that the white spot represents the upper left corner.

It is true that we presented here only the happy case when the template size matches exactly the size of the object looked for in the query image, which is not very often true in a real environment where we know the size of the template, but we know nothing about the bigger image. However, there is some work done in this domain [5] [12] such that the template matching approach can be used in real applications, like the one presented in this thesis: recognizing logos.

Given its nature, the template matching technique is mainly used in situations where the object characteristics are rather constant or where speed is more important than very high accuracy. If we remember the numbers stated in Section 1.1 from Chapter 1, it is not a surprise that this type of algorithm was the main choice in the implementation.

2.2.2 Learning Approach

This approach is totally different from the Template Matching one. If the main idea for the latter is to have a stateless algorithm that takes a template and a query image and tries to produce a binary response based on whether it finds the template or not, the Learning Approach

requires a set of training templates upon it builds a model and constantly upgrades that model. Obviously, it is called the Learning Approach, because it mainly relies on Machine Learning techniques.

Because the main goal of this thesis was not building the strongest logo recognition algorithm, but rather a reliable infrastructure that can support any type of image processing algorithm, and, therefore, it exceeds the scope of this project, we would not give a detailed description of this approach. However, we would like to mention that there is currently a lot of research in this domain and the *state-of-the-art* is continuously pushed forward. Among the most important recently published, written papers in this field of study, we can count: Bundle Min-Hashing for Logo Recognition[8], Scalable Triangulation-based Logo Recognition[14], Realtime Storefront Logo Recognition[2] or Scalable Logo Recognition in Real-World Images[13].

Chapter 3

Proposed Solution

In this chapter we present a detailed description of the system we have built in the scope of this thesis. In Section 3.1 we present what were the requirements and constraints of this project together with what technologies we used and why we chose them. In the next part of this chapter, Section 3.2, we take each component in the pipeline and present how it works. Finally, the third part, Section 3.3, consists in the detailed description of the template matching algorithm variation that we used for the logo recognition component.

We see in the next section that one of the biggest requirements (and achievements for that matter) of this thesis was to build a system that is algorithm agnostic and can do any type of computation on images fetched from social media streams. In order for this to happen, it was necessary to divide the work into multiple separate components each one being responsible for a very specific task, i.e. fetching images, storing images, recognizing logos etc. But before diving into each component, let us see what were the requirements and technologies used in building this system.

3.1 Requirements and Technologies Used

Even though the project is open source and we had almost full freedom in designing the system and choosing the technologies, we still had to conform with some requirements came from the proposing company, Hootsuite. One of the rigid requirements was that the project has to be written in Scala, not necessarily because they considered Scala is the best choice here, but because most, if not all, of their platform is built upon Scala. More details about Scala, what are its advantages, disadvantages or why it is fun to use, can be found in Section 3.1.2.

Another requirement that did not come from Hootsuite, but rather from the common sense is that the system has to scale very easily. Like we saw in Section 1.1, the number of images that this system has to face is quite big, even with the computation power we have available nowadays. The most intuitive way to imagine a scalable system is if we add an arbitrary number of machines, let's say M_{new}, than the task has to be completed $1 + \frac{M_{new}}{M_{old}}$ times faster, where M_{old} is the old number of machines available. In practice, mostly due to Amdahl's Law[2], but also because of other factors, like network latency, hardware or software fails etc, the system will never reach that idealized speedup, but if the speed gain is linear with $1 + \frac{M_{new}}{M_{old}}$, we can declare that the system is scalable. Fortunately for us, the task we have to perform is very easy to distribute across multiple machines. Suppose that we have M machines and each of them is running an instance of the logo recognition algorithm, then, if we have $2 \times M$ images to

[2] http://home.wlu.edu/~whaleyt/classes/parallel/topics/amdahl.html

process, we can easily distribute 2 images per machine and then collect the results in the end. Because images that come from the social media streaming endpoints are rather independent and processing them represents perfectly independent tasks, we do not even need to collect the results, but this is a discussion we will have in Section 3.2.

One of the main characteristics of the system is that it has to allow introducing new *known* logos or removing old ones in an easy manner. It is allowed to have a small delay of a few minutes before the system can recognize the new logo when we want to insert it in the *known logos database*, but not hours or, even worse, days. By easy we mean here fast and reliable rather than easy from the end user point of view. The actual process is fine to involve digging into the filesystem, updating some directory, restarting some components, but this process has to be seamless from the whole system perspective, we cannot afford shutting everything down and restarting just because we want to insert a new logo because the service has to run continuously.

The process of moving an image through the pipeline from one end to the other should finish either with a positive result (i.e. we found this list of logos in the image) or with a negative one. In the former case, one of the components has to store the image (or the image link) in a database together with some data to describe the result, for example the logo list. This type of metadata is often called *annotated data* along this thesis and we may also refer to the pair $(image, logolist)$ as the *annotated image*.

Despite being designed as mostly a background job that sometimes stores results in a database, our system have to have at least a minimum interaction with a human and we decided that interaction to be in the form of a minimal web interface where one can basically see the pairs $(image, logolist)$ from the database, but in a nicer form. Also, the dashboard has to offer at least a table or chart with statistics regarding recognized logos. To increase the usefulness of the dashboard it should be real time and compute the statistics as long as new logos are discovered. It is fine to have a slight delay in updating the dashboard, but the web application should poll the database at least once a minute.

Because our focus is rather on the infrastructure and reliability rather than the algorithm approach itself, which can be changed at any future time, but also because we cannot miss important data and have false statistics, the system is not allowed to lose messages. In the scope of this thesis, by *message* we understand the output of one component and the input of the next component in the pipeline. Of course, the first component, Fetch Module (see Section 3.2.1) and the last component, Annotated Data Storage Module (see Section 3.2.4), make an exception from this rule as they only output messages, in the case of the former, or consume messages, if we speak about the latter. That does not mean that the first component has no input, it means that its input comes directly from the outside world and represent unstructured data, i.e. tweets or posts, streamed by the Social Networks. Because the volume of steamed data can become very large, we have to make sure that all this data gets processed on time by our system, so in order for this to happen, the first component has to quickly get rid of the posts and pass the work to the next layers.

When it comes to passing messages around, we needed a good infrastructure that supports not necessarily large amounts of data, but a tool that allows multiple producers, multiple consumers and does not offer the same message twice for processing, because we do not want the same image to be analyzed multiple times. We say that we do not need large amounts of data, because we do not have to send entire images between components. We describe in Section 3.2.2 the way we store images so that each component can fetch them directly from the centralized storage, thus avoiding passing them around each time we send a message. Therefore, our messaging system has to be reliable, seamlessly scalable and have a fast way of partitioning the data channels, because we do not want to mix messages around, we only need one channel between two types of components without allowing other processes to have access to that data. One of the mature, distributed open source, messaging systems available is Apache Kafka[1].

[1] http://kafka.apache.org

3.1.1 Kafka – A Messaging System

"Apache Kafka is a distributed, partitioned, replicated commit log service. It provides the functionality of a messaging system, but with a unique design."[4]

First, let us be clear with their terminology for some of the bits that are of direct interest to us. Kafka maintains feeds of messages in categorized queues called *topics*. These topics are, at the low level view, nothing more than large buffer caches where some processes write data (*producers*) and other processes read data (*consumers*). Kafka is designed to run over a cluster of servers, each one of those being called a *broker*, but for the scope of this thesis it is not necessary to know what a broker is.

The most important abstraction that Kafka provides is the topic because it is practically the way we can sort and categorize the large amount of data that we are dealing with. The Kafka cluster maintains, for each topic, a commit log where data is appended and cached for a configurable amount of time. For example, if the log retention is set up at one day, the messages received into one Kafka topic are kept for 24 hours, after which they will be deleted in order to free disk space.

The two main types of active entities in a Kafka environment are the consumers and the producers. A producer is responsible for connecting to one Kafka broker and sending the message to the correct topic, but also for choosing the right *partition* to which it wants to send the message.

The consumers are a little bit more special. Kafka introduces another abstraction here, namely the *consumer group* which is nothing more than several consumers grouped by a common identifier. What is actually interesting here is that, by this approach, one can choose from two models of usage: message queue and publish-subscribe model. What Kafka does is that all the messages from one partition go to all consumer groups (publish-subscribe model), but only one consumer from a consumer group can read a given message (queueing model). This proved, over time, to be extremely useful and it is, indeed, in our case also. Figure 3.1 offers a visual interpretation of the two models that Kafka provides. It is a two server Kafka cluster hosting 4 partitions with 2 consumer groups. Consumer group A has 2 consumers, whereas consumer group B has 4 consumers. The interesting part is that each partition is consumed by both consumer groups (load balancing), but only one consumer from each consumer group reads the message (uniqueness). The good part here is that Kafka is able to provide both load balancing and ordering in the scope of the same partition.

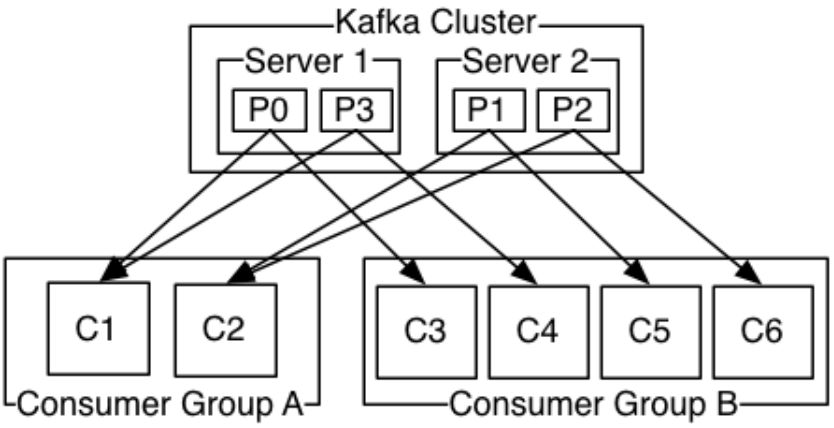

Figure 3.1: Kafka Cluster[3]

3.1.2 Scala

Scala is a programming language created by a team conducted by prof. Martin Odersky at the École Polytechnique Fédérale de Lausanne (EPFL), Switzerland. "Scala fuses object-oriented and functional programming in a statically typed programming language. It is aimed at the construction of components and component systems."[9]

As prof. Martin Odersky et al state in their paper[9], Scala is a language designed for building components. And "true component systems have been an elusive goal of the software industry. Ideally, software should be assembled from libraries of pre-written components, just as hardware is assembled from pre-fabricated chips. In reality, large parts of software applications are written from scratch, so that software production is still more a craft than an industry."

One of the main goals of Scala is to provide a *scalable* language in the sense that the same concepts can describe small as well as large parts. Therefore, their main focus is on building upon composition and decomposition and concentrate on abstraction, rather than adding a large set of primitives that might be useful in the scope of one component at the bare code level, but not at other levels.

Apart from these thoretical considerations, Scala has several characteristics that make it not only suitable for industry applications, but also fun to use. Among them, one can count:

- seamless Java interoperability: Scala runs on the JVM, so Java and Scala stacks can be freely mixed for totally seamless integration.
- type inference: Scala has static types, but one does not have to specify types for every variable they use, the Scala type system being clever enough to deduct almost all of them.
- concurrency and distribution: Scala uses data-parallel operations on collections, actors for concurrency and distribution, or futures for asynchronous programming.
- traits: combines the flexibility of Java-style interfaces with the power of classes. Think at principled multiple-inheritance.
- pattern matching: match against class hierarchies, sequences and more. Basically, one can use any type of variable in a `switch`-like manner.
- higher-order functions: functions are first-class objects in Scala. You can compose them with guaranteed type safety and use them anywhere or pass them to anything.

To better understand the constrast between Scala and Java (we are directly comparing these two languages because they are very related as they both run over the JVM), let's look at Listing 3.1, which is a Java method that returns a copy of the list of products in an object, and compare it with Listing 3.2 which is the equivalent Scala code.

```
1 public List<Product> getProducts() {
2   List<Product> products = new ArrayList<Product>();
3   for (Order order : orders) {
4     products.addAll(order.getProducts());
5   }
6   return products;
```

Listing 3.1: Code written in Java

```
1 def products = orders.flatMap(o => o.products)
```

Listing 3.2: Code written in Scala

3.2 System Architecture

In this section we explore the system architecture as a whole and then we go one level up and explore each separate component. Figure 3.2 shows a general overview of the whole pipeline. As shown, data comes from the social networks and enters the pipeline through the Fetcher Module which filters every post or tweet and writes the output to a Kafka topic (see Section 3.1.1) from which several processes read and download the images. The downloaded data then gets sent by the Retrievers to the centralized Image Store from which the Detectors pull images for processing purposes. The Detectors are also connected with the Retrievers by a Kafka topic. After the detection process, the life of an image can stop, in case nothing was detected, or can continue to the Storage Module which talks to a MongoDB[1] database. The database gets polled periodically by a web application which can be accessed directly by a human user.

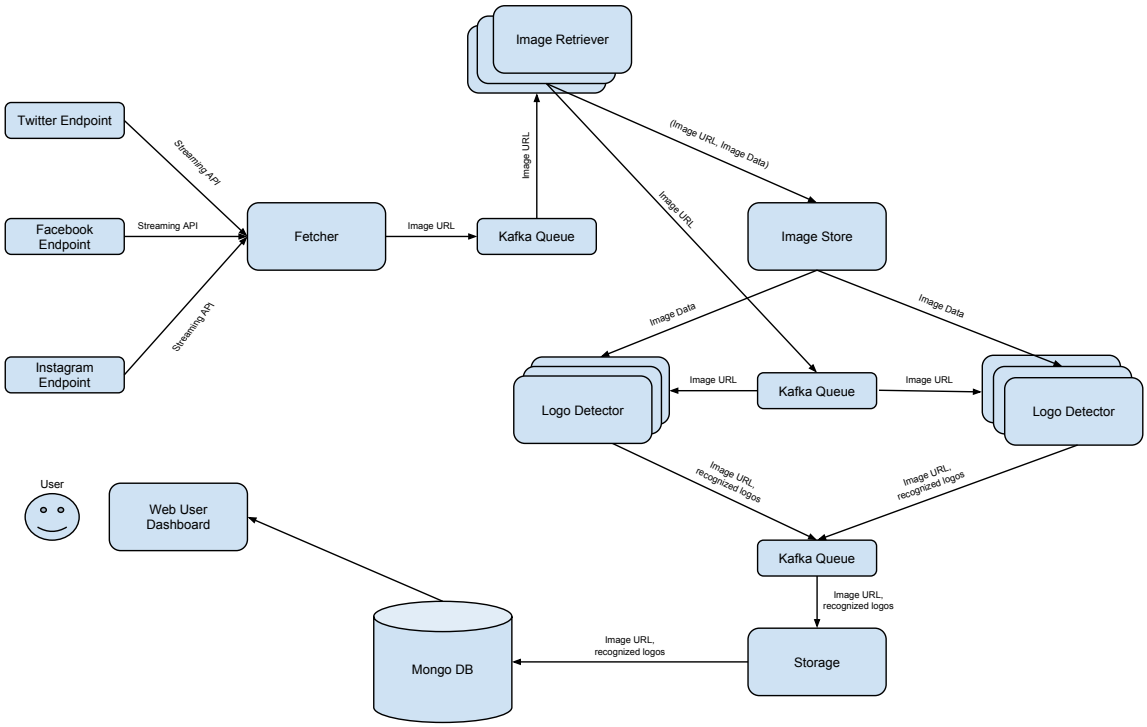

Figure 3.2: High Level System Architecture

Even though some of the components may seem that could have been very well integrated into other ones (for example Image Retriever into Fetcher), we will shortly see that if we do some computations there is impossible to use only one machine for some of them, in our case for the Image Retriever component. Therefore, the reason why some of the modules are drawn as duplicated rectangles is to emphasize that they are replicated processes, so not necessarily need to be run on different machines, but there is a need for more instances of them. The replication is not needed, in this case, for reliability and fault tolerance purposes, but in order to increase the power of consuming more incoming data.

Apart from the replicated components, Figure 3.2 shows another strange characteristic: there are two piles of Logo Detectors. By this, we only try to show that some detectors are specialized in recognizing some logos, while other pile of detectors are specialized in another set of logos. We took this decision because the process of testing an image against each and every possible

[1] https://www.mongodb.org

logo in the database would take too long on a single detector, so it is better to have an even work distribution. Moreover, the most important reason we chose to do this is that, if we want to add a new logo in the equation, we only have to restart one pile of detectors, which may be considerably faster than restarting all detectors.

As we have seen in Section 3.1.1, the messaging system we chose for our implementation can keep messages for as long as they are needed and producers or consumers can come and go at any given time without impacting the stored data. To use this to our advantage, each component basically runs an infinite loop where it waits for input, do some computation and writes to the corresponding queue towards the next component in the chain, as the code in Listing 3.3 shows.

```
1 initializeComponent();
2 consumer = new KafkaConsumer(INPUT_TOPIC);
3 producer = new KafkaProducer(OUTPUT_TOPIC);
4
5 while (true)
6   message = consumer.consumeOne();
7   result = process(message);
8   producer.produce(result);
```

Listing 3.3: General Component Workflow

Having this type of workflow for each component is also very convenient from the update or upgrade perspective. Basically, each component is a stateless process that takes some input, applies a function that does not need any state and then returns an output. The only entity responsible for keeping the state is Kafka, our distributed messaging system, so fortunately for us, we can turn down and up any component at any given time. In a hypothetic situation where all Detectors have failed and went down, the system is well able to recover if the Detectors restart, providing there is enough disk space to cache all those incoming messages and images. However, disk have become so cheap nowadays that we can afford having a lot of it, such that the probability of complete system fail (data loss) is very low.

As far as component restarts concern, they are also needed in case of a new logo being added to the known logo database or in case of a code base update. It is true that in the latter situation, we also have to make sure the changes are compatible, but the biggest concern should be how do we keep the whole system running, in spite of updating some of the components. This problem is also automatically solved by the fact that our clever messaging system is caching for as long as we need it to. If a consumer simply goes down for whatever reason, Kafka will notice that, but there is no action required as the remaining consumers will simply take that workload. If there is no consumer left or too few of them, Kafka will cache the messages until there is again a sufficient number of consumers to continue working.

That being said, in the following sections, we present each separate component that takes part into the processing pipeline.

3.2.1 Fetch Module

The Fetch Module is the frontend of our system as it is the only component that connects to the outside world. It is responsible for fetching streams of unstructured data, i.e. posts, tweets etc, filtering this data in order to keep only what is of interest to us, namely images and downloading these images. After that, the downloaded image gets stored by the Image Store module and the Detectors are notified by the incoming message in the corresponding Kafka queue. This is what happens in a few words, but let us explore some of the subtleties of this module.

First of all, there is no need to have multiple machines, nor processes that are responsible for fetching posts or tweets from the streaming endpoints. To back this up, we make a reference to Section 1.2 and the calculations we did there. If we sum all the incoming tweets and posts we barely reach around 250 messages per second. Given the fact that images are not included as data inside the tweets, but as links and we know that a tweet has at most 140 characters (let us assume that all the data comes from Twitter and a link has at most 100 bytes for simplicity) simple math yields that this component has to face 60KB per second ($(140B + 100B) \times 250 = 60KB$), a value that was modest even 10 years ago.

One may simply ask what happens with this component in case of hardware or software failures and it is a valid question. Well, the answer is so simple that it may be shocking at the first glance: nothing happens immediately. By this, we do not mean that the system stops working entirely, but for a period of time, the fetch component will simply be down and there will be no new tweets fetched. We chose to do this mainly for two reasons. First, because you cannot have multiple connections to the same social network endpoint. The Twitter APIs, for example, limit the number of connections at 1 per application, so there is no point in having replicas for this component. Second, because even if we would have additional replicas being ready to jump in when the main replica fails, there is still a number of streamed tweets that will be lost forever. With this in mind and the fact that Twitter sends us only 1% of their data, there is nothing to worry about if we lose a number of streamed tweets, because we do not have control of which tweets they stream. Those N tweets that we presumably cannot afford to lose could have been in the other 99% from the beginning. The most important thing about our system is to keep the Detectors busy all the time, because they are the real bottleneck of the whole pipeline, even if the Detectors have the biggest number of machines assigned. Due to cost limitations, (see Section 4.2) we periodically need to shut down the Fetch Module, to allow the Detectors to keep up with all the incoming data.

In the future, if the hardware power is increased considerably, there is also the possibility of adding dedicated Fetcher replicas that are ready to jump in the game in case of a failure, but for now this is not a big concern. Moreover, if we add more social networks to the system such that one single Fetcher has to download from too many endpoints, we might also change the design and split the work between multiple Fetchers.

As we have stated before, the Fetcher is only responsible for getting the stream of data and filter the images and by filtering means discarding the whole message content but the image URL. The image URL is the only thing that gets past the Fetcher and gets written in a corresponding Kafka queue on which more Image Retriever processes await.

It should be clear to the reader, by now, that the Fetcher and Image Retriever (see Figure 3.2) are two separate processes, but they do not need to run on different machines. Because the processing power needed here is very low, but also because these two components are very bound to each other in terms of logic, we are considering these two types of processes as being one single component. Even though there is a Kafka queue between them, this is used more as a caching level rather than as a logical boundary.

The Image Retriever processes are, therefore, replicated mostly regarding failure resistance, but also because of bandwidth. If we know that an image is approximately 50–200KB and we have about 30 images per second to process that gives us $30 \times 200KB = 6MB$ of data to download per second in the worst case. If we also take peaks due to busy hours into consideration, the amount of data to download in one second can exceed 10MB so we do not want all this incoming traffic on one single machine. This is the main reason we should have at least 2 or 3 Image Retrievers if our data connection is good.

After the download, an Image Retriever is responsible for two more tasks. First, it has to call the Image Store component (Section 3.2.2) and send the image to it. Secondly, it has to produce a message containing the image ID (image URL) and write it to the Kafka queue on which the

Detectors await.

3.2.2 Image Store

The Image Store component is basically an interface that is an abstraction of the storage engine used underneath. It represents a one common entity that knows where images are stored and can do two types of operations: `putImage(imageID, imageBytes)` and `getImage(imageID)`.

As Figure 3.3 shows, in the current version of the implementation, the Image Store component uses the disk, because most of the images we deal with need to be written and read only once in 99% of the cases. In the future, if we decide to move the whole system in the Amazon Cloud, the underneath storage can be Amazon S3[1]. For now, it is sufficient to use the disk in order to store images and more important to have the storage hidden behind a good abstraction layer such that other components do not know who they are talking to, besides the Image Store component.

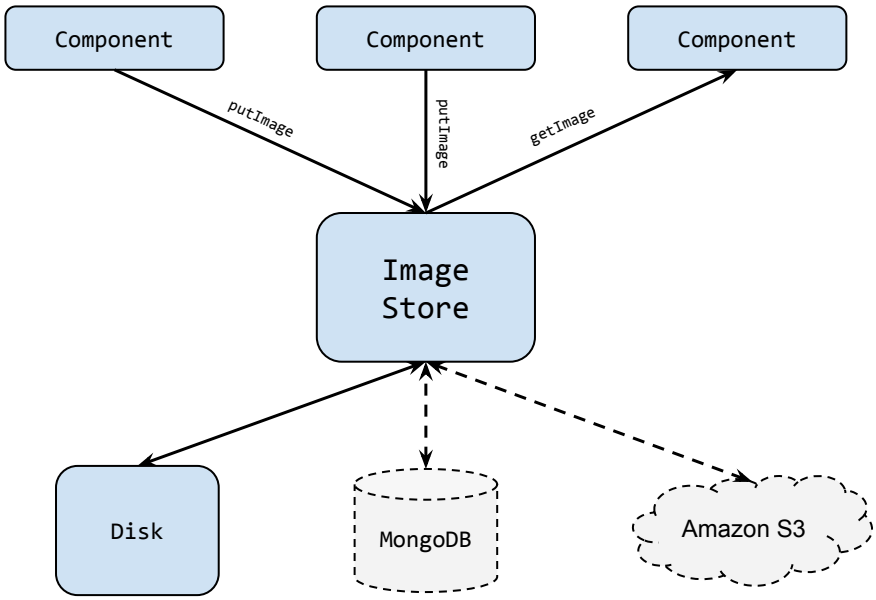

Figure 3.3: Image Store Component

3.2.3 Detector Module

This module is the most computationally intensive one and requires more than 90% of the whole processing power available. In our logo recognition situation, these processes are the bottleneck for the entire system, so if we were to look for performance optimizations this should have been the first place to check. However, we designed the system such that it is entirely algorithmic agnostic, so in a situation where a different, less intensive, algorithm is run inside those Detectors, the situation may change, at least with regard to the bottlenecking problem.

[1] https://aws.amazon.com/s3/

Truth be told, this would be very unlikely, because image processing tasks are computationally intensive by their nature.

A Detector process pulls its data from the Kafka queue where Image Retrievers write imageIDs, in our case image URLs. Once a Detector has the image URL, it requests the actual image from the Image Store (Section 3.2.2) and starts processing it. The processing consists in calling an abstract interface that represents an algorithm. One such interface takes the image as parameter and outputs a list of found features of the image, in our case logos. This list of found features is then compacted in a message and quickly wrote to the next Kafka queue on which the Annotated Data Storage Module (Section 3.2.4) awaits. This workflow is better described in Figure 3.4 below. As the figure shows, the detectors do not rely on a specific algorithm and can run any type of processing, providing they are fed up with the implementation. We tried to illustrate this idea in the two lightly colored boxes.

In our type of implementation, with the Detectors being actually logo Detectors, there is also a need to store somewhere the known logos. Our logo detection algorithm (see Section 3.3) does not recognize logos out of nowhere, it rather compares them to a list of known logos. The design decision we took in this case, at least for now, was to bundle the list of images with the logo detection algorithm such that the process will have them in the `resources/` folder. The downside of this approach is, of course, the fact that every time a logo needs to be updated we have to rebuild the Detector and re-run it. The good part is that this process is very easy to do, we just have to add the new set of images to a folder and recompile the program. Moreover, we only need to do this operations only on one subset of detectors, namely 3 of them. We chose 3 as the magic number here in order to have triple replication. For more details about the queueing system and how the Detectors are organized, please follow Section 3.2.5.

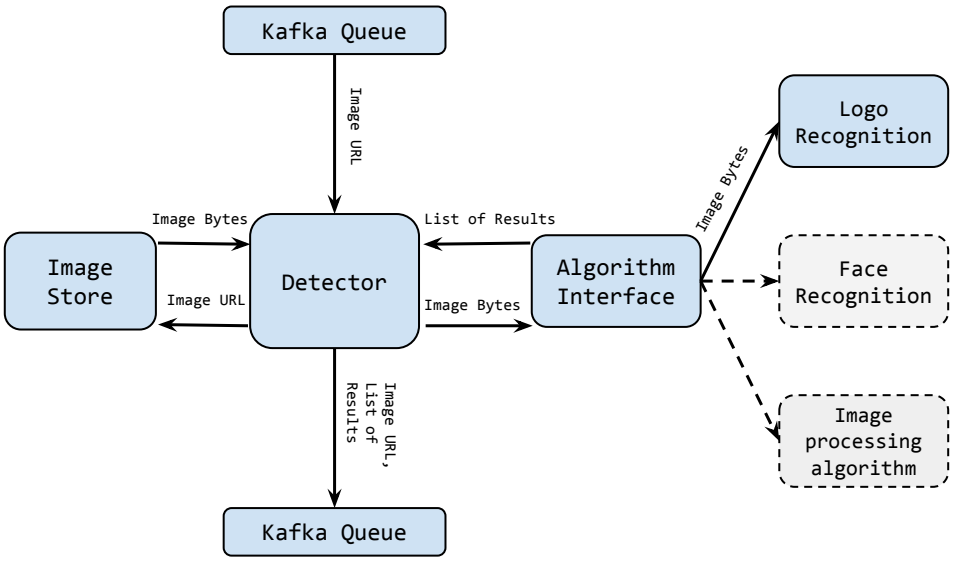

Figure 3.4: Detector Workflow

3.2.4 Annotated Data Storage Module

At this moment we find ourselves beyond the Detector wall and the volume of incoming traffic has diminished considerably, because at this stage we only have *interesting* images from the logical, human point of view. These images are about 1% of the total images analyzed, so, if we assume that the Detector cohorts can keep up with the 30 images pushed into the pipeline by the Fetcher, the storage module has to face 1 image every 3–4 seconds on average. This calculation proves that we only need one process of this kind. In case of an unlikely event of a hardware of software failure, we only have to restart the component. The good part is that we are again backed up by the messaging system that caches the messages until the process goes back up, so we do not lose precious analyzed and found positive data.

Mostly because our messages are basically JSON messages, but also because of reliability, we chose MongoDB[1] as the database for our system. The storage module is also agnostic about the type of computation we did in the previous layers, so the data that gets stored is nothing more than the image URL and the list of features found in the image. We call this features *annotations*, as the Listing 3.4 shows.

```
1  {
2     "imageURL":    "http://example.com",
3     "annotations": [ "Honda",
4                      "Ford",
5                      "Mazda",
6                    ]
7  }
```

Listing 3.4: MongoDB Document

The database is also polled once every 10 seconds by the simple web dashboard in order to check for new logo occurrences and compute new statistics. Because of the rare frequency of the event when we identify a known logo, there is very little load on the database, so we only need one small server to keep it running. In terms of replicas, we rely on the internal MongoDB replications and failure recovery mechanisms.

3.2.5 Messaging and Queing System

As stated before, we rely on the Apache Kafka when it comes to passing messages around and exchange data between different components. Each component is connected to the next one by a Kafka channel (topic) through which only the same type of messages should be sent. Kafka does not have a mechanism of data introspection, such that we can discard potentially malformed messages came from defective processes, thus we are obliged to do sanity checks every time we read a message.

The logo Detectors in our implementation are very slow, mainly because they have to check an image against a list of logos, not just one single known logo. So, if we were to keep adding logos to the known database, we would have ended up in the situation where a single logo Detector needs minutes or, even worse, hours, to check only one image. Imagine that a check against a single logo needs about 1-2 seconds. If we have 1000 known logos, the Detector needs 1000-2000 seconds, which roughly means half an hour. By any definitions of benchmarking, evaluations or even the common sense, that is a lot of time.

Apart from the main abstraction, the topic, Kafka provides another abstraction, the *partition* (see Section 3.1.1). Because Kafka provides these two powerful abstractions, we make use of

[1] https://www.mongodb.org

them especially in the case of Detectors. To avoid the above described behaviour, we split all available Detectors in detector groups, like Figure 3.5 shows. This idea allows two important things, the first one being this problem with computation taking too much for one photo. It is true that by this method we do not diminish the overall needed computational time, i.e. in our case we still have 9 logos to test against, but we diminish the time needed per process, i.e. one Detector has to test against 3 instead of 9 logos. Besides the advantage of having a fair work load distribution, there is also another perk in having this Detector organization, namely when adding a new known logo: we only have to do it for one group of Detectors (see Section 3.2.3).

One problem that is avoided by Kafka's design is the race between Detectors in the same Detector group. If Kafka did not offer a guarantee that messages wrote in the same partition are ordered and read by exactly one consumer, we would have needed to solve the race condition by manually implementing a locking mechanism between Detectors from the same group which would have complicated the whole system. So, once again, Kafka proved itself a suitable choice in this situation.

Messages that we send through Kafka topics from one component to another are serialized as JSON documents. Until now, mainly because of the incipient stage of the project, the messages are very small and have 2-3 keys, so we did not feel the need of using a compression library, like zLib[1], therefore we send messages in clear text. In terms of numbers, we are currently using 3 Kafka topics and 3 types of messages. The 3 topics are: `toDownload`, `toDetect` and `toStoreDetected` and the 3 types of messages are: `DownloadMessage`, `DetectMessage` and `StoreDetectedMessage`. The first two types of messages consist only in the imageID (image URL in our case), whereas the last one has an additional list of annotated data (found logos) which is, in most cases, empty.

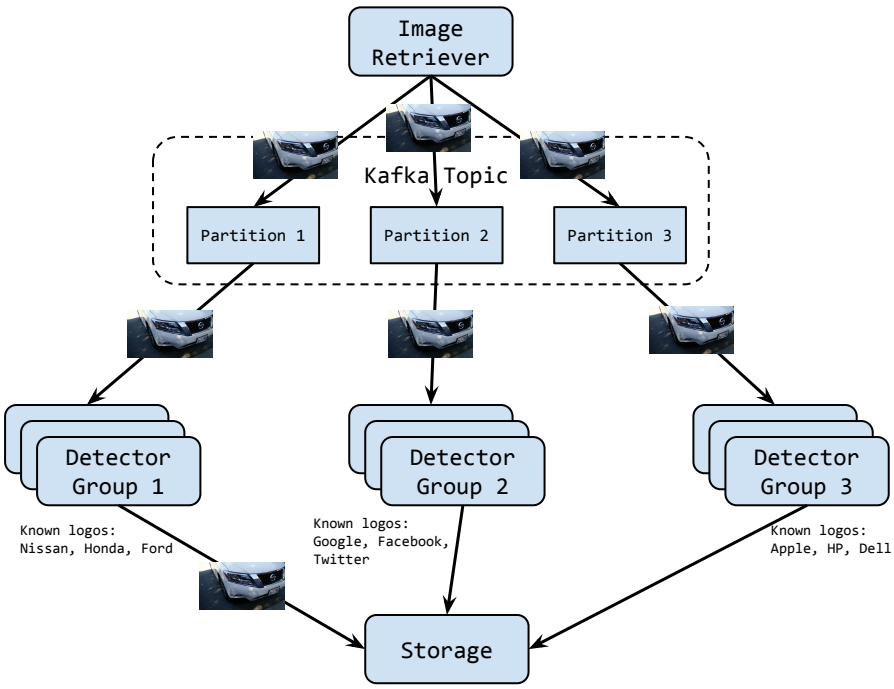

Figure 3.5: Detector Groups

[1] http://www.zlib.net

3.3 Logo Recognition Algorithm

As stated before, the general idea of this project was to create a processing pipeline that allows any type of image computations done on images fetched from social networks streaming endpoints. In order to have a fully functional complete implementation, we also had to choose one algorithm and apply it on the input data. Therefore, we chose a Logo Recognition algorithm that attempts to find logos in a random image based on a database of known logos. Because the main requirement was not the algorithm, but because we needed a fairly fast and simple one, we chose the Template Matching Approach (see Section 2.2.1).

The main idea of this approach is to obtain a template from the known logos and try to match it in every possible position of the query image and it works very well for images that have multiple colors and rich textures or details. Unfortunately, logos are most of the time monocolor and, to worsen things up, without any distinguishable details, not even gradients, i.e. the Twitter logo (see Figure 2.1). The Twitter logo is basically a normal, weird shaped light blue spot. With this in mind, if we try to match it against a completely blue image (same blue color) it would perfectly match in every location. A naïve algorithm would even find thousands of matches in the given image. To solve this problem we had to come up with an original approach and this is: split the image into 4 equally sized squares, match each one of the squares independently and, finally, compare the *best matched* positions and see if they are in the same order as they were in the template. Figure 3.6 attempts to explain it better. We have the Twitter logo template, one random picture containing the logo, the 4-way split template and the matched template over the query image.

As the figure shows, the 4 matched squares are approximately in the same relative position as in the original template, thus, according to this algorithm standards, is a strong belief that the logo have been identified in that position. One thing to notice, though, is that the matches are not in exactly the same relative position, and this is mainly caused by the matching function used

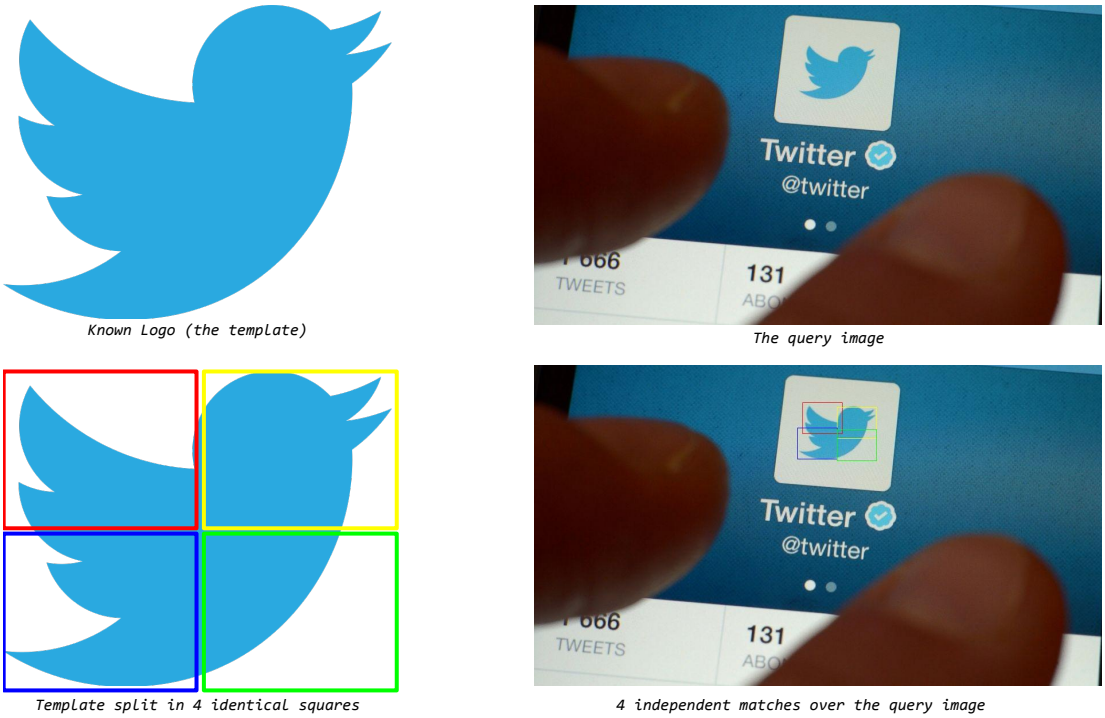

Figure 3.6: 4-way matching algorithm

(see Section 2.2.1) which does an approximation. Throughout our implementation we chose the CV_TM_CCORR metric, because it proved itself to be the most accurate in the experiments we have run. Another important thing, that we have also determined by running multiple experiments and doing the average, was setting the maximum threshold distance allowed between two squares, both horizontally and vertically. To be more precise, there is a 20% (of the square size) allowance between the upper row squares and lower row squares and 31% between right and left. Any match that *looks like* the template, but has larger distances between its squares, is considered a false match. However, we have also determined experimentally that, in some cases, 3 of the squares perfectly match themselves onto the query image (less than 5% gap between them), but one of them which matches in a totally different place. In such cases, we consider the situation a bad luck event and still return a correct match with the condition that the 3 squares that match in the same place have to comply with the 20% and 31% boundaries. Listing 3.5 is the pseudocode of the 4-way split algorithm that runs inside a Detector process.

```
1  def match(template_image, query_image):
2    squares = ['red', 'yellow', 'blue', 'green']
3    for square in squares:
4      template_pos[square] = 4way_split(square, template_image)
5
6    for square in squares:
7      best_match[square] = find_best_match(template_pos[square],
           template, query_image)
8
9    sq_size_x = template_image.x / 2
10   sq_size_y = template_image.y / 2
11   matches_count = 0
12
13   if x_difference(best_match['red'], best_match['yellow']) > 0.31 *
       sq_size_x:
14     matches_count += 1
15
16   if x_difference(best_match['blue'], best_match['green']) > 0.31 *
       sq_size_x:
17     matches_count += 1
18
19   if y_difference(best_match['red'], best_match['blue']) > 0.20 *
       sq_size_y:
20     matches_count += 1
21
22   if y_difference(best_match['yellow'], best_match['green']) > 0.20
       * sq_size_y:
23     matches_count += 1
24
25   if matches_count >= 3:
26     return true
27   return false
```

Listing 3.5: 4 way splitting template matching algorithm

Listing 3.5 presents only the general idea of the algorithm. In reality, we have to do all sorts of other checks, like, for example, check if the squares respect the distance threshold with their neighbours lest they are actually mirrored. For example, the checks in Listing 3.5 do pass in the event when the red and yellow squares are on the lower row instead of the upper.

A problem that is not very hard to see with this situation is what happens if the template has

a difference size compared to the logo in the query image, say 10 times bigger. In this case, it is obvious that the result will be most likely false because even one square would still be 5 times larger that the whole logo. To solve this problem we do a multiple scale scan, with the template size decreased in exponential steps until we find the correct match or until the template becomes to small. To be more precise, we start with the template size approximately equal with the query image and do a full round trip (split the template, match each square separately etc). After that, we decrease the size of the template by 1.15 (determined experimentally) and redo the process. In practice, it turned out that we need between 12–17 round trips to do a complete check. This process is represented in Listing 3.6.

```
1  def match_one(template, query):
2    if template.x + template.y < 50px:
3      return false
4
5    if template.x > query.x or template.y > query.y:
6      return false
7
8    if match(template, query):
9      return true
10   else:
11     return match_one(shrink(template, 15%), query)
```

Listing 3.6: Multiple scale template matching

One last shortcoming of this implementation is that it is very likely to produce wrong results in the event when the same logo can be found more than once inside the query image. The reasons why this is happening are pretty obvious: two squares can match on one logo and the other two squares on another one, the final result being false. So, not only we miss one occurrence of the logo, but we miss all of them. Fortunately for us, the experiments proved that this situation happens so rarely in practice that is far from being a concern right now.

Chapter 4

Testing and Evaluation

In this chapter we talk about the performance of the logo recognition algorithm as well as the supporting pipelined infrastructure. As well as with other computer software programs, testing is important to ensure that the system does not produce wrong results and evaluation is a way of quantifying the performance. In our case, testing is a little bit more trickier, especially for the end result and that is because if we had a program that would tell us if a logo is or is not present in an image, we could have used that program in the first place. This correctness problem is very common in Computer Vision and we also have to face it here.

To ensure that we produce results as sane as possible, we turned to the human eye for verification. Thus, once in a while, we manually checked random images where the system supposedly found a logo to see if the logo is actually present. Unfortunately, this type of verification only gives us an approximate percentage of the false positives that the system produces. In order to have a more accurate understanding over the accuracy of our program, we ended up using the precision and recall measurements which are a common way of quantifying relevancy, especially in pattern recognition. We present more details about precision and recall in Section 4.3.

Although, do not forget the main purpose of this thesis: building a pipelined infrastructure that allows any type of image computation done on images fetched from social networks. In order to have a general idea about how well the system performs, we also did some cost and resource consumption analysis of the current system. We present a more detailed view over this facts in Section 4.2, but first let us see some of the representative numbers of the current algorithm.

4.1 Algorithm Performance

The current algorithm that runs inside a Detector is the logo recognition one, presented in Section 3.3. Although it may seem that the 4 way split combined with the multiple scale scan we are doing, considerably slows down the whole process, in practice this proved to be not bad at all. The first argument that comes in the defense of this practice is that it is, indeed, producing better results. Either if we drop the 4 way split or the multiple scale scan, the results we get are far worse. It is true that in a normal situation you would prefer an algorithm that runs 4 times faster, but has 10 to 20% worse results, but in our case, if we drop one of the steps, the recall we get goes as down as 25% of the total input images, which is a decrease in performance of over 100% (we currently have a recall of 55–75%). The second argument in the favour of our algorithm is that both the 4 way split and the multiple scale scan are heavily and easily parallelizable, which is a very good property for an algorithm, especially nowadays when mainstream processors come with 4 or even 8 cores.

That being said, the total time needed to test a template of 680×480 pixels against an image of the same size, dropping the time needed to read and load images into memory is roughly 0.075 seconds or 75 milliseconds, which seems blazing fast. If we take into consideration that we have to do 4 separate matches for one single template, the time goes up to $4 \times 0.075 = 0.25s$ and starts to get bigger. Adding the average of 15 round trips per query, the time seems to become $15 \times 0.25 = 3.75s$ which looks like a long running time for an algorithm of this type (template matching). In practice, using a processor with 4 cores and parallelizing each one of the steps the time needed for the whole process goes down about 3.5 times to an approximate value of 1.25 seconds needed per query. It is still a large value, but, at least, starts to look like something we can use in order to achieve a real time system.

The 1.25 seconds per query we have seems very decent at a first glance, but let's explore what this actually means in terms of cost and resource consumption, because a small *0.1s* improvement at the algorithm level, means a $3s$ gain for the whole system at the speed of 30 images per second, so this actually means that we save up computation time as for **90 images**, if we assume that the system keeps up the pace.

4.2 Cost and Resource Consumption

To better understand what processing power we have, what performance the algorithm really has and what are the costs of running the system at full capacity we do a cost evaluation in this section.

First, let us assume that we want to have enough power to process 30 images per second. In the event of busy hours and peaks of more than 30 images, we can rely on the queueing system that powers the whole infrastructure, Kafka, that will cache data until we can process it. We choose 30 images here because it is the actual average number of images streamed by Twitter's Streaming API endpoints and because it is a fairly round number. If testing a query image against a template logo takes about 1.25s (see Section 4.1), this means that we need 5s to test it against 4 templates with variations of the same logo. To worsen things up, do not forget that we have to recognize multiple logos and suppose that we want our system to *know* 100 different logos. Doing the math results in $5 \times 100 = 500s$ of total processing time needed for one single round of 30 images that come in a second.

The 500s of total processing time need to be absorbed in just one second, if we want to keep up the pace with the Fetcher and process 30 images per second, therefore we need a total of 500 processing nodes or processes. If we suppose that we can run 4 different processes on a single physical machine, that gives us a total number of $500/4 = 125$ machines.

Using the above computation, but approaching it from the cost point of view results in a possibly unexpected result. A standard machine with 4-8 processing cores and 8GB of RAM costs around \$0.25 per hour[1] so, counting the total of 125 machines that we need only for the Detectors results in $\$0.25 \times 125 = \$31.25$ per hour. With this in mind, let's do the math for an entire month: $\$31.25 \times 24h \times 30days = \$22.500$ per month.

Neglecting the additional 2 or 3 machines to run the other *light* components, \$22.500 is the cost that need to be split between the 100 companies that supposedly have to pay in order to find their logo among the ones that we are searching for. $\$22.500/100logos = \$225$ per brand. This means that each brand has to pay us at least \$225 a month and this is only the raw cost of keeping the pipeline up. Add other costs like engineer salaries or VAT that needs to be paid and the price can easily come over \$300 per month per brand.

Although \$300 seems like a significant amount to pay each month, it may worth it when we talk about companies that have millions of clients and billions of dollars in revenue. For the

[1] https://aws.amazon.com/ec2/pricing/

moment, the budget of this project does not allow spinning up so many machines at once, therefore the temporary solution for not cluttering the Kafka logs was to shut down the Fetcher from time to time (see Section 3.2.1) in order to allow the 4-5 Detectors that we had to recover from behind.

4.3 Precision and Recall

4.3.1 Precision and Recall in Pattern Recognition

In pattern recognition and information retrieval with binary classification, precision (also called positive predictive value) is the fraction of retrieved instances that are relevant, while recall (also known as sensitivity) is the fraction of relevant instances that are retrieved. Therefore, they are a measure for relevance.

For example, suppose that we have a computer program that has to recognize apples in an image and we provide it with a picture of 10 apples and 10 oranges. If the program says that it recognized 9 apples, but only 7 out of those 9 were really apples, then we say that the program had a precision of $7/9 = 77\%$ and a recall of $7/10 = 70\%$. Precision can be seen as a measure of exactness or quality, whereas recall is a measure of completeness or quantity.

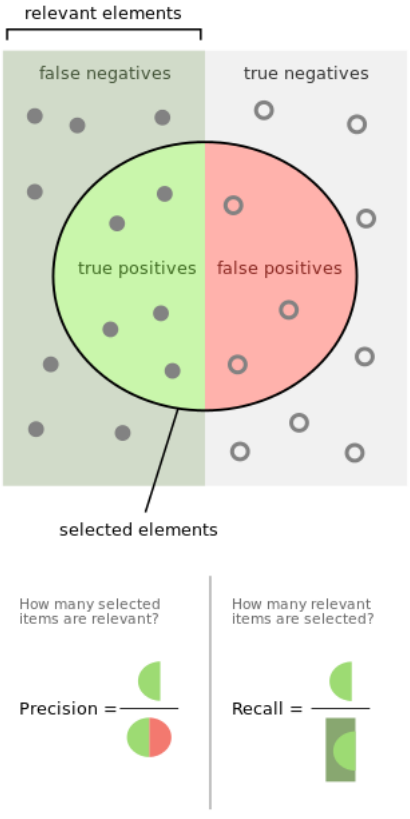

Figure 4.1: Precision and recall[1]

Precision and recall can be defined as:

$$Precision = \frac{tp}{tp + fp}$$

$$Recall = \frac{tp}{tp + fn}$$

The abbreviations `tp`, `tf` and `fn` mean, respectively, true positives, false positives and false negatives. A better illustration of the definitions above can be found in Figure 4.1.

4.3.2 Precision and Recall for our Algorithm

In the case of our logo recognition algorithm precision and recall are very easy to compute. In order to evaluate these two parameters in an automatic fashion, from time to time we had to make the input constant over time. For increased relevance of this type of evaluation, we did not change the images nor their numbers. For that, we took 80 images, 10 with the Twitter logo, 10 containing the Honda logo and 60 random images that did not contain the Twitter nor the Honda logo, but there were 10 of them that contained other logos. We ran the algorithm on all those 80 images and the results are presented in Table 4.1.

Table 4.1: Precision and recall for Twitter and Honda logos

Logo	Img with logo	Img w/o logo	fp	fn	tp	Precision	Recall
Honda	10	70	0	3	7	100%	70%
Twitter	10	70	2	4	6	75%	60%

4.4 Scalability and Portability

Scalability is a very important factor in a distributed environment and it should not be treated as a second tier requirement or concern, although, the first instinct is to focus on raw performance, big O complexities and running times for algorithms. A scalable system proves itself to be extremely useful, once the data grows a lot in volume and, after some point, the only way to adapt it. Right now, if we had an algorithm that needs linear time and runs 100 times faster than the current one, we would have required only 5 processing nodes (the amount we need right now divided by 100). But imagine that some day, the incoming volume of data grows 100 times as well, so, at that moment we will be back right where we find ourselves now: make hundreds of machines to work together.

Fortunately for our pipelined infrastructure, it was designed with these problems in mind, so we can scale it easily. Say we have 20% more machines available, we can accommodate approximately 20% more data, which we will need for sure with the rapid development of social networks.

Another thing worth mentioning is the reliability of our system. One of the requirements (see Section 3.1) was to build a reliable system that does not stop when we want to do a code update or when we want to add a new logo to the known logo database. To obtain this, we have to be careful about how we restart some of the Detectors and, to be more precise, how many of them we update at the same time. In Section 3.2.5 we talked about the fact that we chose to split Detectors into several Detector groups, each group containing 3 Detectors. We chose the number 3 in order to have 2 additional replicas in the same groups so we can swallow 2 failures per group while the system can still recognize all types of known logos, even if it does that slower. But the main idea behind this distribution is when we do an update, hence we have to restart some Detectors.

Suppose that we have to add a new logo to the known logo database. The first step in the updating process is to add the 4 templates associated with that logo into the `resources/` folder of the Detector component. The second step is to build the new Detector component and the third step is to choose a detector group that has the smallest number of known logos. The last step consists in taking down, one by one, the Detectors in that group and introducing the new Detector in their place, but the obvious problem that comes here is what happens with the partition assigned to these two Detectors while the old one is taken down and the new one has not started yet. Fortunately, Kafka caches data long enough until the new Detector comes up again. But let's assume that we have recently got some new machines that run faster than the old ones we have, so we can afford to run only two Detectors in some group. In this case, Kafka proves again to be extremely handy in the way that it can reassign the abandoned partition to another Detector in the group. This can be done after a configurable amount of time in which nobody has read from that partition.

Last, but not least, a useful feature of this system is its portability, because it can basically run on any operating system that has a Java Virtual Machine[1] with a version bigger that 1.7, because Scala compiles to Java bytecode. Although, we do not take too much credit for this achievement, as it is a generally true fact for any Scala or Java built project.

[1] https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf

Chapter 5

Further Work

This chapter provides an overview of the current lacks of the system and possible future enhancements. Given the nature of this project, we have to take into consideration both the logo recognition algorithm, as a separate entity that can be developed completely separately in an isolated environment, and the underlying infrastructure that supports everything.

5.1 System Architecture Enhancements

The first improvement that can be made would be to replicate the Fetcher Module presented in Section 3.2.1. This would not introduce a great benefit in the present situation, but in the future could prevent losing data in the event of a hardware or software failure that would affect the Fetcher. It is important to understand that currently the Fetcher is a single point of failure for this system and, if it fails, all other components behind it will starve after a period of time. The obvious solution to this problem is to have at least another replica that can come in and take over the job of the failed Fetcher.

The second *todo* on the list of improvements is, of course, supporting more social networks, because, right now, Twitter is the only source of data. Adding Facebook or Instagram would definitely increase the relevance of the statistics we produce, but would also require a considerable amount of additional processing nodes.

In terms of storage, a *nice-to-have* feature is to move everything we store on disk right now, especially downloaded images, on a cloud service that is reliable and integrated in the cloud infrastructure that we are using. One example of this technology is Amazon S3[2].

Last, but not least, the Annotated Data Storage Module (see Section 3.2.4) is also a single point of failure in the system that needs to be replicated in the future. Fortunately, there is very little traffic that reaches this point, so we can safely rely on the Kafka log caching for now.

5.2 Better Logo Recognition Algorithm

A large amount of the time available was spent on building the pipelined system that is basically the supporting infrastructure for image processing algorithms that run inside the Detectors. Therefore, the accent has not been stressed on building the best possible logo recognition algorithm, but a *good enough* one that can prove the functionality of the underlying system.

[2] https://aws.amazon.com/s3/

With this in mind, a decent improvement for the present implementation would be a faster algorithm. It is hard to imagine a ground breaking improvement while still using the template matching approach, but others could make a difference. At the amount of machines that we currently need, even a 10% faster algorithm will result in about $2000 saved per month, if we are running at full Fetcher capacity.

Chapter 6

Conclusions

In this thesis we have presented a pipelined, distributed, social media image processing infrastructure with exemplification on logo recognition. The biggest objective we had and fulfilled was to build the system in an algorithm agnostic manner, such that it supports any type of image processing providing it has the implementation of an algorithm.

In order to have a fully functional, complete system, we have also implemented a logo recognition algorithm that is based on pattern matching and better described in Section 3.3. We chose this approach for two main reasons. First, because it provides good results without using too much computing power and second, because the amount of time allocated for developing the algorithm was only a fraction of the total time needed to build the project and we needed an algorithm that can be well understood and implemented relatively fast.

Even with these several constraints, mainly presented in Section 3.1, the results we have obtained (see Section 4.2 and Section 4.3) are more than satisfactory and we have a strong belief that the system can be well integrated in a real social media analytics platform, like the one that Hootsuite provides.

Bibliography

[1] Precision and recall. https://en.wikipedia.org/wiki/Precision_and_recall.

[2] Frank Liu. Yanlin Chen. Realtime storefront logo recognition. http://cvgl.stanford.edu/teaching/cs231a_winter1415/prev/projects/report-2.pdf, 2014.

[3] Apache Foundation. Kafka 0.8.2 documentation. http://kafka.apache.org/documentation.html#introduction.

[4] Apache Foundation. Kafka, a high-throughput distributed messaging system. http://kafka.apache.org/documentation.html.

[5] Kai Briechle. Uwe D. Hanebeck. Template matching using fast normalized cross correlation. http://i81pc23.itec.uni-karlsruhe.de/Publikationen/SPIE01_BriechleHanebeck_CrossCorr.pdf, 2001.

[6] Ana Huaman. Template matching. http://docs.opencv.org/doc/tutorials/imgproc/histograms/template_matching/template_matching.html.

[7] Statista Inc. The statistics portal. http://www.statista.com, 2015.

[8] Stefan Romberg. Rainer Lienhart. Bundle min-hashing for logo recognition. http://www.multimedia-computing.de/mediawiki/images/d/da/Bundle_Min-Hashing_-for_Logo_Recognition_-_ICMR2013.pdf, 2013.

[9] Martin Odersky. An overview of the scala programming language. http://www.scala-lang.org/docu/files/ScalaOverview.pdf, January 2006.

[10] Duc Thanh Nguyen. Wanqiq Li. Philip Ogunbona. An improved template matching method for object detection. http://link.springer.com/chapter/10.1007%2F978-3-642-12297-2_19, 2010.

[11] opencv dev team. Opencv documentation. http://docs.opencv.org, 2011-2015.

[12] Kavita Ahuja. Preeti Tuli. Object recognition by template matching using correlations and phase angle method. http://www.ijarcce.com/upload/2013/march/11-kavita%20ahuja%20-%20object%20recognition-c.pdf, March 2013.

[13] Stefan Romberg. Lluis Garcia Pueyo. Rainer Lienhart. Roelof van Zwol. Scalable logo recognition in real-world images. http://www.multimedia-computing.de/mediawiki/images/3/34/ICMR2011_Scalable_Logo_Recognition_in_-Real-World_Images.pdf, 2011.

[14] Yannis Kalantidis. Lluis Garcia Pueyo. Michele Trevisiol. Roelof van Zwol. Yannis Avrithis. Scalable triangulation-based logo recognition. http://image.ntua.gr/iva/files/logo_triangulation.pdf.