Assignment of bachelor's thesis

Title:	Stolen paintings recognition app
Student:	Michal Janeček
Supervisor:	Mgr. Tomáš Karella
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2021/2022

Instructions

The goal of the thesis is to design, implement and test a mobile app for matching user-taken photos with images in the database of stolen paintings. The student will create a test database containing several pictures of paintings from publicly available sources. The included artworks are not required to be stolen due to the possible copyright issues.

Electronically approved by Ing. Karel Klouda, Ph.D. on 22 December 2020 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Stolen paintings recognition app

Michal Janeček

Department of Applied Mathematics

Supervisor: Mgr. Tomáš Karella

May 12, 2021

Acknowledgements

I would like to thank my supervisor, Mgr. Tomáš Karella, for his guidance and for making me work consistently. My thanks also go to Ing. Marek Sušický for his help in communicating with police organizations.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 12, 2021 . . . .

Czech Technical University in Prague

Faculty of Information Technology

© 2021 Michal Janeček. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Janeček, Michal. *Stolen paintings recognition app.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

Tato práce se zabývá vývojem mobilní aplikace k rozpoznávání kradených obrazů. Nejdříve uvede do problému krádeže uměleckých děl a popisuje současné možnosti jejich rozpoznávání. Dále obsahuje návrh aplikace s třívrstvou architekturou a popis technologií, které byly pro její vývoj použity. Pro rozpoznávání obrazů, které se provádí na serveru, používá algoritmus ORB, který je v práci detailně popsán. Testování bylo prováděno na více datasetech s 1800 obrazy a jednom datasetu obsahujícím přes 9000 obrazů. Při testování s lehce augmentovanými obrazy byla dosažena přesnost okolo 90 %, pro více augmentované obrazy zhruba 50 %. Výsledný software zahrnuje mobilního klienta, server a databázi a může být použit pro rozpoznávání kradených obrazů.

Klíčová slova krádeže obrazů, rozpoznávání obrazů, strojové vidění, ORB, mobilní aplikace, docker, třívrstvá architektura

Abstract

This thesis focuses on developing a mobile application that can be used to recognize stolen paintings. First, the problem of stolen art is described, and the current options of recognizing stolen paintings are analyzed. Then an application with three-tier architecture is proposed and the technologies used are described. For the painting recognition which occurs on the server, the ORB algorithm is used, which is then thoroughly described in the thesis. Testing was done using multiple datasets with 1800 paintings and one dataset containing over 9000 paintings. The achieved accuracy is about 90 % when tested with lightly augmented paintings but drops to around 50 % for heavier augmentations. The resulting software includes a mobile client, server, and database, which can be used for stolen painting recognition.

Keywords painting theft, painting recognition, computer vision, ORB, mobile application, docker, three-tier architecture

Contents

Introduction **1**

1 Painting theft **3**
1.1 Brief history . . . 3
1.2 Current situation . . . 3
1.3 Lost paintings recovery . . . 4
1.4 Painting validation . . . 5

2 Solution architecture **7**
2.1 High-level use case description . . . 7
2.2 Requirements . . . 8
2.2.1 Functional requirements . . . 8
2.2.2 Nonfunctional requirements . . . 9
2.3 Architecture . . . 9
2.3.1 Server . . . 9
2.3.2 Client . . . 10
2.4 Docker . . . 10
2.4.1 Docker Compose . . . 11
2.5 Django . . . 11
2.5.1 Django REST framework . . . 11
2.6 PostgreSQL . . . 12
2.7 Android . . . 12
2.7.1 Retrofit . . . 13

3 Painting recognition **15**
3.1 Overview . . . 15
3.2 Feature extraction . . . 16
3.3 FAST . . . 16
3.3.1 Non-max suppression . . . 16
3.4 Brief . . . 17

3.5 ORB . . . 18
3.5.1 Oriented FAST . . . 19
3.5.1.1 Multiscale image pyramid . . . 19
3.5.1.2 Harris corner measure . . . 19
3.5.1.3 Orientation component . . . 19
3.5.2 Adding rotation awareness to BRIEF . . . 21
3.5.2.1 Steered BRIEF . . . 21
3.5.2.2 rBRIEF . . . 22
3.6 Feature matching . . . 22
3.6.1 Brute-Force matcher . . . 22
3.6.2 FLANN-based Matcher . . . 22
3.6.3 Ratio test . . . 22

4 Implementation **25**
4.1 Setting up Docker containers . . . 25
4.1.1 Docker image for Django . . . 25
4.1.2 PostgreSQL container . . . 26
4.1.3 Composing the containers . . . 26
4.2 Django . . . 28
4.2.1 Creating the project . . . 28
4.2.2 Configuring Django to use PostgreSQL . . . 29
4.2.3 Custom admin commands . . . 29
4.2.3.1 wait_for_db command . . . 30
4.2.3.2 load_paintings_from_dir command . . . 30
4.2.3.3 test_accuracy command . . . 30
4.3 Database . . . 30
4.3.1 Models . . . 30
4.3.1.1 Painting model . . . 31
4.3.1.2 PaintingDescriptors model . . . 31
4.4 Feature extraction . . . 32
4.5 Feature matchers . . . 33
4.5.1 FLANN-based matcher . . . 33
4.5.2 Brute-force matcher . . . 34
4.6 API . . . 34
4.6.1 SearchPaintingViewSet . . . 34
4.7 Mobile client . . . 35
4.7.1 Search tab . . . 35
4.7.2 Report stolen tab . . . 36
4.7.3 Models . . . 36
4.7.3.1 FoundPaintingModel . . . 37
4.7.3.2 Painting . . . 38
4.7.4 Retrofit implementation . . . 38

5 Dataset **39**

5.1 Real datasets 39
5.1.1 Interpol 39
5.1.2 PSEUD 39
5.2 Datasets for testing 40
5.2.1 Tate Collection 40
5.2.1.1 Downloading the data 41

6 Testing **43**
6.1 Data Augmentation 43
6.1.1 imgaug library 43
6.1.2 Sequential augmenter 43
6.1.3 Used augmenters 44
6.1.4 Data Augmentation Notebook 44
6.2 Measuring the results 45
6.2.1 Datasets used for evaluation 45
6.2.2 Used augmentation 46
6.2.3 Evaluation metrics 47
6.2.4 Calculating results 47
6.2.5 Results 47

Conclusion **49**
Future work 50

Bibliography **51**

A Acronyms **57**

B Contents of enclosed CD **59**

List of Figures

2.1 The application architecture 10
2.2 Architecture of app using Django REST framework [25]. 12

3.1 The pixels used by the FAST corner detector [39] 17
3.2 Five different approaches to choosing the test locations proposed in [37] 18
3.3 Example flow chart of object recognition software [41] using the ORB algorithm 19
3.4 Illustrative multiscale image pyramid for the Mona Lisa [40] 20

4.1 The two main tabs in the app 36
4.2 Taking a picture and making a search request 37

6.1 Comparison of a random painting before and after augmentation . 46

List of Tables

4.1 Description of ORB parameters [52] and their default values used in the project. 32

5.1 Sample rows from the metadata csv file, important columns. . . . 41

6.1 Description of the augmenters used. 44
6.2 Sub-datasets of the Tate Collection that were used for testing. . . 45
6.3 Complete averaged results (%) for all datasets and both augmentation styles. 48

Introduction

Art theft is as old as art itself, and it is still a very relevant topic these days. This thesis focuses specifically on one type of art - paintings. Many paintings that have been stolen even decades ago have never been found. A simple tool, like a mobile application that can recognize whether a piece of art has been stolen, could improve the situation. Nowadays, when someone comes across a painting that they suspect to be stolen, they have no simple way to determine it.

There are many databases of lost paintings all over the world. The most popular ones are private and law enforcement controlled. Many of them charge for searching, are not accessible to the general public, or lack proper filtering options.

This work focuses on delivering an easy-to-deploy server with a database and a simple, user-friendly mobile application. The server implements a RESTful API that allows clients to upload stolen/lost paintings and search the database by painting to validate if it has been reported stolen. The mobile application is designed to communicate with the server's API. It makes it simple to upload and search paintings in the database by taking pictures using the phone's camera. It also presents the results in a friendly way to the user.

Chapter 1 introduces the reader to the history and problematics of stolen art, then analyzes the current options for painting theft victims and people wanting to validate paintings. Chapter 2 contains high-level analysis of the application and describes the technology used. The related work and techniques used for painting recognition are described in Chapter 3. Implementation of the server and the mobile app is captured in Chapter 4. Lastly, Chapters 5 and 6 describes the process behind obtaining the data, how the data were used for testing, experimenting with different setups, and the results.

Chapter 1

Painting theft

This chapter introduces the reader to the problematics of painting theft. It contains a brief history and significant events of art theft. It also describes the current situation regarding the theft of paintings and discusses painting recovery/validation options.

1.1 Brief history

The courts in ancient Egypt seem to have dealt with tomb robber cases on an almost daily basis. The criminals testified to the theft of various artifacts and other objects. The Mayer Papyri (c. 1108BCE) details the torture of those caught; the punishment was as severe as amputation of the hands. Despite these punishments, several theft cases continued, such as amulets and exotic artifacts mentioned in [1].

There are many well-known cases of painting theft throughout history. Perhaps the most notable art theft is the stealing of the Mona Lisa in 1911. The entire country of France was shocked, and even the country's borders were closed [2].

In 1990, two thieves stole thirteen paintings from the Italianate mansion Isabella Stewart Gardner Museum in Boston. In total, the estimated value of those paintings is $500 million, making it the largest art heist in the history of humankind. To this day, the paintings are still missing. [3]

1.2 Current situation

"*Art theft statistics say that more than 50,000 pieces of artwork are stolen each year around the world and the black market for stolen art is valued at between $6 billion and $8 billion annually.*" [4]

Due to low recovery rates, the number of unsolved painting thefts is rising rapidly. The recovery rate of art registered on the world's largest private art database, the Art Loss Register (ALR), between 2000 and 2009 was approximately 1.9% [5].

Solving art theft is a low-priority issue among most law enforcement agencies worldwide [6]. "*The public, too, prefers that the police focus on 'real' crime rather than on purloined art. Unsolved assaults are scandals; missing paintings are mysteries.*" [7]

One of the most notable recent painting thefts occurred on March 30, 2020. *The Parsonage Garden at Nuenen in Spring*, an early Vincent Van Gogh's painting was stolen from Singer Laren, a Dutch museum that was closed at the time due to the COVID-19 global pandemic [8]. These kinds of thefts may become more common as "*the lack of crowds and security potentially compromised by staffing issues during the virus outbreak may present an invitation to opportunistic thieves.*" [9]

1.3 Lost paintings recovery

There is no single free unified database of lost paintings nor any central authority taking care of art theft [6]. Therefore, the recommended action is to report the theft to local authorities and stolen art databases [10].

There are a few private entities that run databases of stolen art. Unfortunately, they often charge victims a fee for registering their stolen pieces, including the ALR - a private online database with more than 700,000 items in their database [11]. The company has been criticized many times for the period of its operation. Many express concerns about their hardball tactics pushing ethical and sometimes legal boundaries [12]. People continue to use ALR because of the lack of proper alternatives and their extensive database [6]. This is granting them a competitive advantage that could prevent the establishment of new, more open, and transparent databases.

The effectiveness of police organizations varies from country to country. Over the years, concerns have been expressed about the degree of co-operation between them. The International Criminal Police Organization (INTERPOL), the world's largest police organization, operates a "Stolen Works of Art Database" (SWOA), which accepts entries from law enforcement agencies worldwide [13]. Because they do not take victim's entries directly, they have to rely on national police forces for information. Allegedly, many of them send the data only occasionally, or not at all [6]. Given all this and the fact that the primary purpose of the database is to disseminate information and not actively search for the missing art, the chance of recovery depends heavily on

the abilities of local institutions.

1.4 Painting validation

In this context, painting validation is a process of checking whether a particular painting has been reported stolen/lost.

The lack of central authority as described in section 1.3 means that there is no option to validate a painting against various databases quickly. The best way to validate a painting is to search the databases one by one, which may be complicated and lengthy and in some cases useless. There could be a situation such as suspicion of illicit trafficking where the painting has to be validated quickly.

Another problem is that many of the databases lack the feature of search by image. For example, in PSEUD [14] it is only possible to search the database by a name and other attributes, but this is often the type of information we do not have when validating a painting.

On the other hand, INTERPOL has the feature to search by image, but when I started writing this work, October 2020, it was mandatory to fill an application to access the database. There was not an option to quickly validate a painting like via using a mobile application. The only available mobile application for stolen art identification was the *iTPC Carabinieri* [15] managed by the Italian police agency. However, it is only available in the Italian language and only validates the painting against one database. We reached out to the INTERPOL and the Police of the Czech Republic to obtain the data about the lost paintings, to use them in this thesis and make it simple for anyone to quickly validate an artwork using a mobile client.

The institutions were proven to be unwilling to provide the data. About four months later, INTERPOL started to promote an app called ID-Art on their website. The app was released at the start of 2021, by chance, a few months after our email communication [16].

Chapter 2

Solution architecture

This chapter describes the project use cases, requirements, and structure. It contains the general outline of how the application's architecture looks like and explains the core technologies used for the implementation. It does not focus on the painting recognition algorithms described in Chapter 3.

2.1 High-level use case description

The main goal of the implementation part of this thesis is to deliver a mobile application that allows reporting stolen/lost paintings to a shared database and validating paintings against that database. Three types of personas will use the software:

1. **Admin**
   - Manages the server and the database.
   - Uploads large batches of paintings to the database.
2. **Victim of painting theft**
   - Reports stolen/lost paintings, thus uploading them to the database used by painting validators.
3. **Painting validator**
   - Uses a mobile application to validate paintings quickly.

2.2 Requirements

This section describes the requirements for the software as a whole.

2.2.1 Functional requirements

1. **Taking a picture**
    - It must be possible to take a picture of a painting using the mobile app. This feature will be either implemented directly or delegated to a camera application and get the resulting image.
2. **Picking a picture from gallery**
    - The user must be able to pick an existing image from his phone and use it within the app.
3. **Uploading a batch of paintings**
    - The admin must be able to upload a folder with several paintings at once to the shared database using a simple command.
4. **Painting recognition**
    - Algorithms for determining whether a certain painting is in the database must be implemented.
5. **Uploading a stolen/lost painting**
    - The mobile app has to have the functionality of uploading a new painting to the database and any information about the painting that the database accepts and the user wishes to provide.
6. **Painting validation**
    - The mobile client must have a function to quickly validate a painting against the database to check if it has been reported lost/stolen. After the validation, the app will either inform the user that the painting is not known, or if found, display all known information about the painting.

2.2.2 Nonfunctional requirements

1. **Fast processing of new painting**
   - When a new painting is uploaded to the database, it must be processed immediately, i.e., it will be available for validation in a several minutes at most.
2. **Lightweight application**
   - The mobile application size has to be smaller than 20 MB, so it can be downloaded fast enough even with a very slow internet connection.
3. **Simple multi-platform deployment**
   - The backend has to be simple to deploy on a new platform. It should be able to run on most hosts without significant configuration changes.

2.3 Architecture

To ensure that the resulting application will be responsive and lightweight on all devices, I decided to create and implement the three-tier architecture. All data will be stored in a centralized database, and the complex calculations and painting recognition will be carried out on a server. The mobile application will serve the purpose of the presentation layer and delegate the work to the server.

2.3.1 Server

To encapsulate everything the server needs to run, I decided to use Docker (see 2.4) containers. This way, any host with Docker installed can run the server. Containers are generally more portable than using the alternative approach of virtual machines, while also easier to maintain [17].

The server side was written in Python, one of the most popular programming languages for computer vision tasks. The most popular Python frameworks are Django and Flask [18], both having large communities, which was a large decision factor, as I wanted to use a thoroughly tested and maintained server technology. I chose to use Django (see 2.5) as the backbone as it provides more features than Flask.

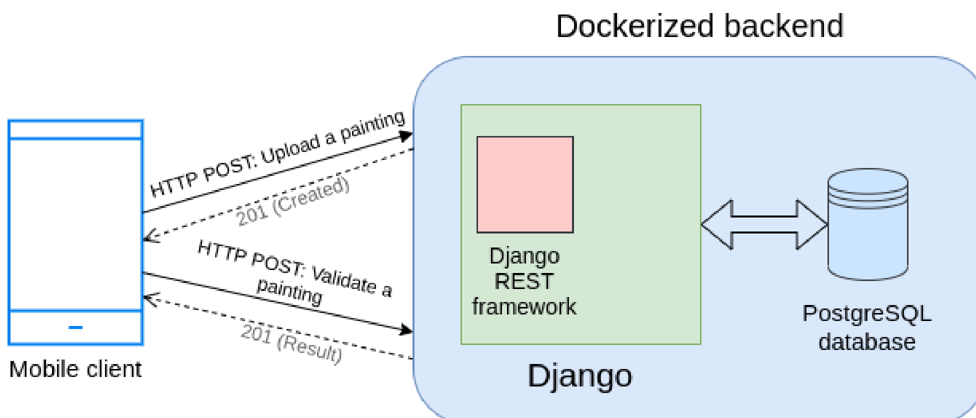


Figure 2.1: The application architecture

The server uses the Django REST framework (detailed in section 2.5.1) to deliver a robust RESTful API.

2.3.2 Client

Ultimately, the mobile application should be available for both iOS and Android, which together take almost 100 % of the global mobile operating system market share [19]. However, for the purpose of this thesis, the mobile application was developed only for Android (see 2.7). It can be used as a proof of concept when communicating with institutions and demonstrating the product in other ways.

As the mobile application serves mainly to communicate with the server and present the results, anyone can fairly easily implement their mobile client for iOS or even Android if the client's design delivered in this thesis does not suit the needs of anyone who wishes to use it.

For implementing the REST client in the app, I used Retrofit (see 2.7.1), which is a helper library widely used in Android and Java that simplifies creating HTTP requests and processing responses.

2.4 Docker

To make the deployment simple, I decided to use Docker.

Docker describes itself as "*an open platform for developing, shipping, and running applications.*" [20] It makes it very easy to package programs. Docker

runs applications in isolated containers that are generally more lightweight than virtual machines. Containers contain everything needed to run the application without relying on what is currently installed on the host. [20]

2.4.1 Docker Compose

To separate the database and the server in different containers for security reasons and good practice while also keeping it simple, I use Docker Compose.

"*Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.*" [21]

2.5 Django

Django is a powerful server-side web framework that enables the development of complex solutions. It provides choices for almost any functionality and follows the "Batteries included" philosophy [22]. I chose to use Django to ensure future scalability, security, and rapid development [23]. In addition, Django is written in Python, which fits my needs as it is one of the most robust and machine learning friendly programming languages.

2.5.1 Django REST framework

Django REST Framework is a library that builds on top of standard Django models to create a flexible and powerful API.

It is composed of three layers [24]:

1. **Serializer**
    - Converts complex data structures into a format that is easily transmitted via an API.
    - Transforms information submitted by a user to data (Models).
2. **ViewSet**
    - Defines available operations for a given set of data.
    - *create*, *list*, *retrieve*, *update*, *destroy* ...
3. **Router**
    - The surface layer of the API.

- Defines the URLs that provide access to the ViewSets.

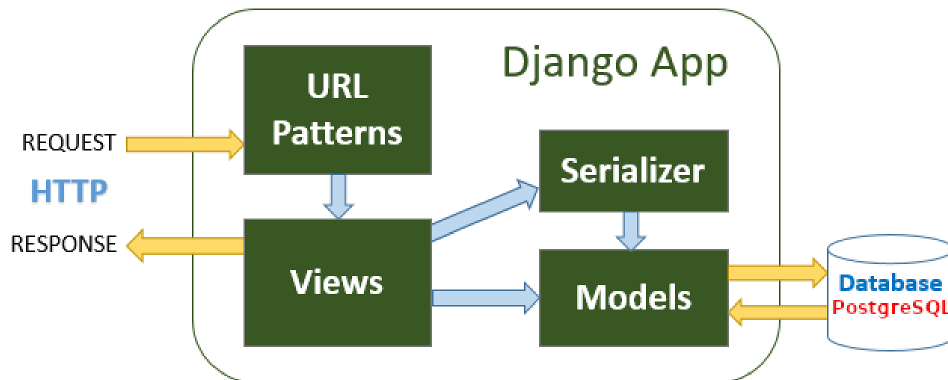


Figure 2.2: Architecture of app using Django REST framework [25].

2.6 PostgreSQL

"*PostgreSQL is a powerful, open-source object-relational database system that uses and extends the SQL language combined with many features that safely store and scale the most complicated data workloads.*" [26]

I chose to use PostgreSQL for its proven architecture, reliability, and security. Moreover, the community behind the software is very active and consistently works on delivering new solutions [26]. These factors make it a good choice for a project like this, where the amount of data is expected to grow over time, and security and active maintenance are essential.

In this project, I use dockerized PostgreSQL. I have a separate docker image for it, specifically the `postgres:10-alpine` downloaded from Docker Hub [27].

2.7 Android

Android is a mobile operating system developed by Google based on the Linux kernel [28]. As of March 2021, it holds over 71% of the mobile operating system market share worldwide according to [19].

I decided to develop the first version of the mobile client just for Android, mainly for its reach. Other reasons that make **prototype development** for Android, in my opinion, superior to its biggest competitor, iOS, are:

- Developing for Android, unlike for iOS, does not require specific hardware, which makes it **more accessible** for independent developers.
- **Cheaper** deployment and maintenance fees.
- Google Play has **less restrictive app review guidelines** than Apple, thus allowing more freedom in application design and making it easier to publish prototypes.

2.7.1 Retrofit

Retrofit is a networking library used in Android development. It is used to create type-safe REST Clients. It consists of three main parts [29]:

1. **Model classes** which are used as a JSON models.
2. **Interfaces** defining the available operations.
3. **Retrofit.Builder class** where the base URL is specified.

I chose to use Retrofit for its performance [30], convenience in **uploading image files**, and extensibility [31].

Chapter 3

Painting recognition

This chapter describes painting recognition, compares different approaches to this task, and introduces the reader to feature extraction. Then it details the algorithms used for feature detection/description and feature matching.

3.1 Overview

Painting recognition is a two-dimensional object recognition computer vision task. Each painting is a specific object which a two-dimensional model can easily represent. That makes the task easier compared to three-dimensional objects because there is no need to recognize the object from different sides. Because of this property, it is possible to extract a consistent set of features that can then be used to represent the said painting.

In [32], Yiyu Hong and Jongweon Kim trained a Convolutional Neural Network (CNN) for Art Painting Identification and compared it to state-of-the-art image descriptor SIFT which it outperformed by 13.6%. However, some of the images in the test set were heavily augmented, which could skew the results slightly in a way that does not reflect real situations. The CNN was trained on 30,000 randomly distorted versions of 100 images. The downside of this approach is that adding/removing new paintings takes relatively a lot of computational power.

Unlike in the case with CNN's, the feature extraction approach only looks at one image at a time. Adding a new known painting is as simple as extracting its features and inserting it into the "known" list, allowing near real-time manipulation with it.

3.2 Feature extraction

Feature extraction is a process of finding and describing the features of an image. Features, typically interesting points like corners, are pieces of information about the image's content. The result is a set of descriptors, usually vectors, that can be compared and matched.

Some of the most well-known algorithms detecting and describing local features are SIFT [33] and SURF (inspired by SIFT, improving speed) [34]. To address the computational complexity and the performance speed of these algorithms, Ethan Rublee with his team in 2011 came up with an alternative called ORB [35], a feature extraction algorithm based on state-of-the-art feature detector FAST [36] and feature descriptor BRIEF [37]. It has a similar matching performance to SIFT while being almost two orders of magnitude faster. Another advantage is that it is not patented and free from licensing restrictions, unlike the alternatives.

3.3 FAST

Features from accelerated segment test (FAST) is a corner detection algorithm, first proposed by Rosten and Drummond in [36]. It decides whether a certain pixel p is a corner by analyzing a circle of sixteen pixels around it, as shown in figure 3.1. Pixel p is classified as a corner if there are at least n consecutive pixels on the circle that are brighter than $I_p + t$ or darker than $I_p - t$, where I_p is the intensity of p and t is a brightness threshold. Equation 3.1 shows a formula for assigning group $S_{p\to x}$ to a pixel x with intensity $I_{p\to x}$ [38].

$$S_{p\to x} = \begin{cases} d, I_{p\to x} \leq I_p - t & \text{(darker)} \\ s, I_{p\to x} - t < I_{p\to x} < I_p + t & \text{(similar)} \\ b, I_p + t \leq I_{p\to x} & \text{(brighter)} \end{cases} \tag{3.1}$$

To quickly find and exclude a vast number of non-corners, FAST first inspects pixels $1, 5, 9, 13$ (the four compass directions). At least three of these pixels have to be brighter/darker than p. Otherwise, there can not exist a contiguous set of more than 12 pixels, which is the original value of n [36].

3.3.1 Non-max suppression

Marking multiple pixels in adjacent locations as corners is an unwanted behavior because they generally represent the same feature in the image. This is solved by non-maximum suppression - only taking the pixel with the highest

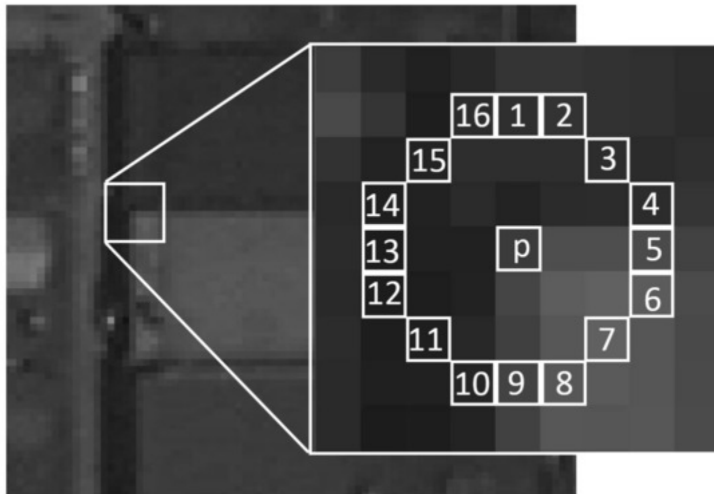


Figure 3.1: The pixels used by the FAST corner detector [39]

value and discarding others. To determine the point value, a score function is used.

$$V = \max\left(\sum_{x \in S_{bright}} |I_{p\to x} - I_p| - t, \sum_{x \in S_{dark}} |I_p - I_{p\to x}| - t\right) \quad (3.2)$$

It takes the sum of the absolute differences between the intensities of the pixels in the contiguous arc and the center pixel. We then compare the values of all adjacent keypoints and discard all except the one with the highest score [36].

3.4 Brief

First proposed in [37], Binary Robust Independent Elementary Features (BRIEF) is an efficient feature point descriptor as a binary string. It can be used to describe the corner detected by FAST or any other feature detection algorithm. To describe a feature point p, a patch $\mathbf{s}$ of size $S \times S$ pixels with p in the center is considered. To obtain the elements of the binary descriptor, a required number of pairwise intensity comparisons are performed on the *smoothed* patch using a test τ defined as:

$$\tau(\mathbf{s}; \mathbf{x}, \mathbf{y}) = \begin{cases} 1 & \text{if } \mathbf{s}(\mathbf{x}) < \mathbf{s}(\mathbf{y}) \\ 0 & \text{otherwise} \end{cases} \quad (3.3)$$

where $\mathbf{x}$, $\mathbf{y}$ are the two pixels being compared and $\mathbf{s}(\mathbf{x})$, $\mathbf{s}(\mathbf{y})$ are the pixel intensities in a smoothed version of $\mathbf{s}$. There are several different approaches in which we may select the pairs of pixels $\mathbf{x}, \mathbf{y}$, as shown in figure 3.2.

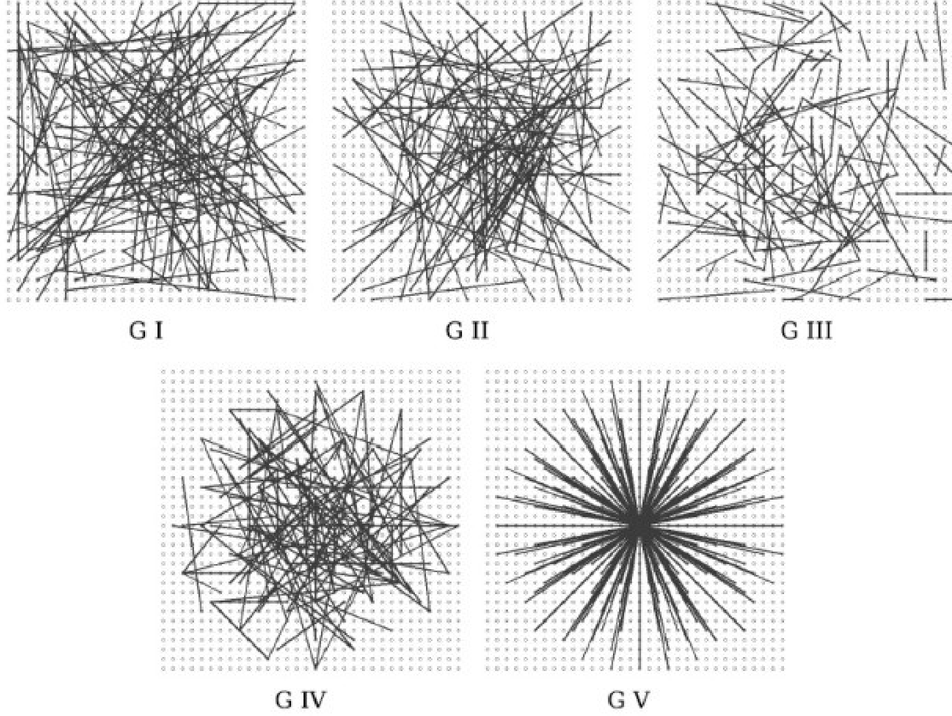


Figure 3.2: Five different approaches to choosing the test locations proposed in [37]

BRIEF descriptor of patch $\mathbf{s}$ is then a n_d-dimensional vector of bits computed as follows:

$$f_{n_d}(\mathbf{s}) = \sum_{1 \leq i \leq n_d} 2^{i-1} \tau(\mathbf{s}; \mathbf{x}_i, \mathbf{y}_i), \tag{3.4}$$

where n_d is the number of performed tests.

3.5 ORB

Oriented FAST and rotated BRIEF (ORB) is a local feature detector and descriptor, first presented by Ethan Rublee in 2011 [35], based on FAST and BRIEF. It addresses the core problems of these algorithms without losing out on the speed aspects. It makes the keypoint detection more scale-invariant, employs a method for ordering the keypoints, and introduces an orientation

component. Because the matching performance of BRIEF falls off sharply for in-plane rotation of more than a few degrees [40], ORB adds the functionality of rotation awareness and makes it rotationally invariant with outstanding results.

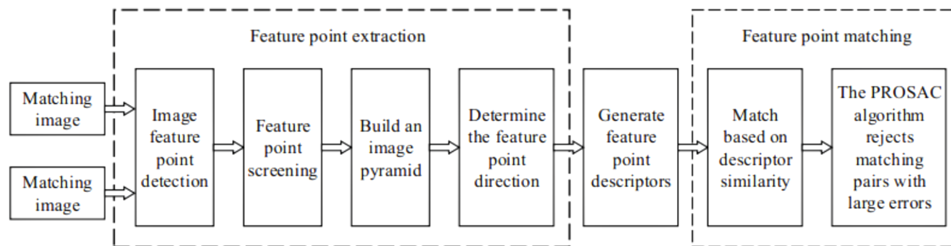


Figure 3.3: Example flow chart of object recognition software [41] using the ORB algorithm

3.5.1 Oriented FAST

This section goes through the keypoint detection process in ORB. The orientation component is discussed in the last part.

3.5.1.1 Multiscale image pyramid

For reliable detection of keypoints on one image with different sizes, ORB algorithm uses a multiscale image pyramid. The pyramid contains several levels where each level is represented by a downsampled version of the image in the previous level, the first level being the original image, as shown in figure 3.4. Then, the keypoint detection algorithm is used to detect keypoints at each level, effectively locating keypoints at a different scale. [40]

3.5.1.2 Harris corner measure

FAST has large responses along edges, which are considered worse features than corners, because of their lesser distinctive properties – there are many similar patches along the edge [42]. Because FAST alone does not produce any value representing the "quality" of the keypoints, ORB adopts Harris corner measure to assign each keypoint a value of "cornerness". The value computed using Harris corner measure can then be used to order the keypoints, which is very useful when we only want a maximum of N keypoints.

3.5.1.3 Orientation component

As a corner orientation measure, the intensity centroid method introduced by Paul L. Rosin in [43] is used. The orientation is assigned depending on how the levels of intensity change around the keypoint (the center of the patch). "*The*

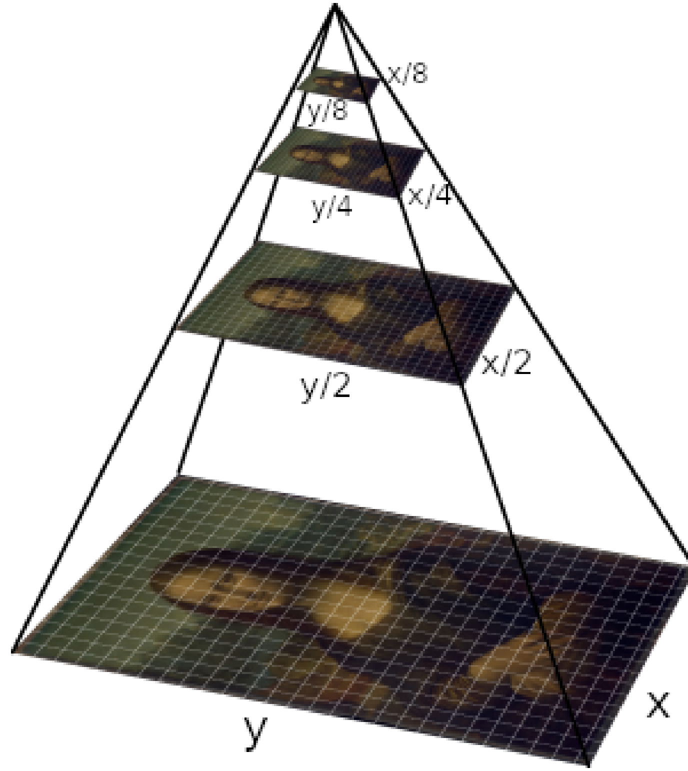


Figure 3.4: Illustrative multiscale image pyramid for the Mona Lisa [40]

intensity centroid assumes that a corner's intensity is offset from its center, and this vector may be used to impute an orientation."[35] The centroid is determined using the moments of a patch **s** defined as:

$$m_{pq} = \sum_{x,y \in \mathbf{s}} x^p y^q I(x, y) \tag{3.5}$$

Where x, y are pixel coordinates and $I(x, y)$ is the intensity of the corresponding pixel. Then the intensity centroid is:

$$C = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}} \right) \tag{3.6}$$

Finally, the geometric center O (the keypoint coordinate) and the centroid C of the patch **s** are used to obtain a vector $\overrightarrow{OC}$. The angle of this vector is used as the orientation of the keypoint, calculated as follows:

$$\theta = \arctan \left(\frac{m_{01}}{m_{00}} / \frac{m_{10}}{m_{00}} \right) = \arctan \left(\frac{m_{01}}{m_{10}} \right) \tag{3.7}$$

In the original paper, Rosin corrects the angle by 180 degrees if the corner color is darker than its background, but this is not used in ORB as the angle measures are consistent regardless of the corner color. After calculating the orientation, we can rotate the patch and then compute the descriptor, thus obtaining some rotation invariance [40].

3.5.2 Adding rotation awareness to BRIEF

Rotation-Aware BRIEF (rBRIEF) is based on the BRIEF descriptor, detailed in section 3.4. BRIEF needs the described patch to be smoothed to reduce the sensitivity to noise, which in the case of ORB is achieved using an integral image (summed-area table), where each test point is represented by a 5×5 sub-window around it. For the feature vector length $n = 256$ is used.

The original BRIEF descriptor proposed in [37] is very sensitive to in-plane rotation – when the rotation is greater than 30 degrees, the matching ratio is close to zero.

3.5.2.1 Steered BRIEF

Steered BRIEF respects the obtained orientation θ using Oriented FAST (described in 3.5.1) of a keypoint. It does so by modifying the test locations for each keypoint according to its orientation so that the descriptor has direction information. We create a $2 \times n$ matrix S, where n is the number of binary tests, as follows:

$$S = \begin{pmatrix} x_1, & \dots, & x_n \\ y_1, & \dots, & y_n \end{pmatrix} \tag{3.8}$$

This matrix represents n keypoint with coordinates $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$. Then we use a rotation matrix R_θ to create a steered version of S. The matrix R_θ is obtained using the patch orientation θ:

$$R_\theta = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \tag{3.9}$$

Then steered matrix S_θ is simply $S_\theta = R_\theta S$.

Finally, we define the directional descriptor. The method of comparing two points does not change, so we can use the function f_{n_d} from the original BRIEF

(see equation 3.4). The modified descriptor can then be represented as:

$$g_{n_d}(p,\theta) = f_{n_d}(p)|(x_i, y_i) \in S_\theta \quad (3.10)$$

3.5.2.2 rBRIEF

When performing experiments with steered BRIEF, the authors found that its descriptors have lesser variance than the original BRIEF, which means that the features are less discriminative. Another thing they noticed is that the binary tests (see equation 3.3) become more correlated. To combat this problem, they implemented a greedy algorithm for choosing a good subset of binary tests described in detail in [35].

3.6 Feature matching

Feature matching is a process of matching features in one image with features in another image. The distance of the two features represents their similarity. For features represented by binary descriptors, like in the case with ORB, Hamming norm is often the best option, whereas for SIFT or SURF, it is better to use L^2 norm [44].

3.6.1 Brute-Force matcher

Brute-Force matcher works on a simple basis – it compares a feature f_{1_i} from one image to every feature from the other image and returns the closest one f_{2_j}. Optionally, it can check whether the closest match for the returned feature f_{2_j} in the first image is f_{1_i}, which means it only considers two features as a match if they match each other.

3.6.2 FLANN-based Matcher

Fast Library for Approximate Nearest Neighbours (FLANN) [45][46] contains a "*collection of algorithms optimized for FAST nearest neighbour search in large datasets and for high dimensional features*" [44]. As a tradeoff for its much higher speed than Brute-force matcher, it does not guarantee to find the best match, only an approximation. In the case of matching ORB descriptors, the most notable algorithms it uses are the randomized k-d tree algorithm, and LSH described in [47].

3.6.3 Ratio test

The ratio test is a method for deciding whether a match is good, which is useful for filtering out points that do not have any valid match, like random noise. David Lowe introduced it in [48]. It looks at the two closest matches of a

point, and if they are not different enough, it discards them. More specifically, for two matches m_1, m_2 let d_1, d_2 be their distances. Then if $d_1 < d_2 * \alpha$, then d_1 is considered a good match, otherwise it is discarded. The ratio α is a configurable parameter; in the original paper, 0.8 is used.

In his implementation, Lowe had great success with this method – it discarded 90 % of the bad matches while only discarding 5 % of the good ones.

Chapter 4

Implementation

This chapter describes how the core parts of the project were implemented.

4.1 Setting up Docker containers

This section describes the Docker setup used in the project.

4.1.1 Docker image for Django

My main selection criteria for the image were:

- Reliability, security, size and the amount of testing it has undergone.
- Provides all the tools to install Django and related components.
- Allows for simple installation of `OpenCV` `opencv-python` library.

After experimenting with the most popular lightweight images used for Django deployment like `python:3.6-alpine`, I found that the amount of additional packages needed in order to install `opencv-python` greatly mitigates the pros of using a lightweight image. Additionally, installing a massive number of packages, each in specific versions, raises the probability of dependency errors and reduces the system's flexibility for future updates.

For these reasons, I decided to use the latest version of the Debian image as a container for the Django server. Debian is a stable and secure Linux based operating system with a large community. Over the years, it has been thoroughly tested and is viewed as a reliable choice. [49]

Most of the requirements are given in the `requirements.txt` file, where the name of the package and version range is specified, e.g., `opencv-python>=4,<5` requires version at least 4.0 but less than 5.0. This approach allows minor up-

dates and security patches but reduces the probability of potential errors from a major update.

```
# latest Debian version available
FROM debian:latest

# copy the requirements file from the local build context
# into the container
COPY ./requirements.txt /requirements.txt

# installing python3 + pip
RUN apt-get install -y python3
RUN apt-get install -y python3-dev
RUN apt-get install -y python3-pip
RUN pip3 install scikit-build
RUN pip3 install requests
RUN pip3 install --upgrade setuptools pip

# installing requirements
RUN pip3 install -r /requirements.txt

# creating the server directory and copying it from the
# local build context
RUN mkdir /app
WORKDIR /app
COPY ./app /app
```

Source Code 4.1: The core part of the `Dockerfile`

4.1.2 PostgreSQL container

When choosing the container for PostgreSQL there are generally two options – the `postgres-alpine` based on the alpine Linux or just `postgres` based on Debian. In this case, I decided to go with the `postgres-alpine` as it takes up almost two times less space and is sufficient for cases when it is not needed to install any additional packages.

The image uses environment variables to set the superuser password, name of the database, initial arguments send to `postgres initdb` [27], etc. These variables can be set in using Docker.

4.1.3 Composing the containers

To combine the containers described in the above sections into one multi-container Docker application, a tool Docker Compose (see 2.4.1) is used. It

needs an `docker-compose.yml` file with instructions and where the services used in the app are defined.

My application is composed of two services, configured as follows:

1. **Server service** (*app*)
    - Is build from the local `Dockerfile`.
    - Has port 8000 mapped to a port 8000 on the Docker host.
    - Uses local `app` folder as a persistent data storage.
    - Runs Django commands in following order:
        a) Waits for database container to start.
        b) Propagates changes to models if there are any.
        c) Launches the server.
    - Sets the environment variables that are later used for accessing the database.
2. **Database service** (*db*)
    - Specifies the name of the image on Docker Hub.
    - Sets the environment variables.

```
version: "3"

services:
  app:
    build:
      context: .
    ports:
      - "8000:8000"
    volumes:
      - ./app:/app
    command: >
      sh -c "python3 manage.py wait_for_db &&
             python3 manage.py migrate &&
             python3 manage.py runserver 0.0.0.0:8000"
    environment:
      - DB_HOST=db
      - DB_NAME=app
      - DB_USER=postgres
      - DB_PASS=supersecretpassword
    depends_on:
      - db

  db:
```

```
image: postgres:10-alpine
environment:
  - POSTGRES_DB=app
  - POSTGRES_USER=postgres
  - POSTGRES_PASSWORD=supersecretpassword
```

Source Code 4.2: The full implementation of the `docker-compose.yml` file

4.2 Django

This section contains the process of Django setup, configuration, and the implementation of custom admin commands.

4.2.1 Creating the project

The first step was to create a new Django project using the `startproject` command from `django-admin.py`. The command needs to be executed on the server's image described in section 4.1.1 like this:

```
admin@admin:~\$ docker-compose run app sh -c "django-admin.py
↪  startproject app ."
```

In Django, the project consists of components called applications. These applications provide logical separation of reusable components. For this project, I created two applications:

1. **Core** app
   - Holds the central code shared across the other applications.
   - Defines the database models.
   - Contains all the migrations (changes to the database schema over time).
   - Contains custom Django admin commands.
2. **Painting** app
   - Contains all painting recognition algorithm and tools.
   - Implements REST API.

```
admin@admin:~\$ docker-compose run app sh -c "python manage.py
↪  startapp core"
admin@admin:~\$ docker-compose run --rm app sh -c "python
↪  manage.py startapp painting"
```

Source Code 4.3: Creating the applications

After creating the applications, I included them in the `INSTALLED_APPS` dictionary in the project's `setting.py` file, where all used applications must be listed.

4.2.2 Configuring Django to use PostgreSQL

To configure the server to use PostgreSQL database running in container *db* described in section 4.1.2, I modified the `DATABASES` dictionary in the `settings.py` file, which contains the settings for all databases used in the project. I set the default database to PostgreSQL by modifying the following fields [50]:

ENGINE the database backend to use, in this case PostgreSQL, which is built-in in Django
HOST the host of the database
NAME the name of the database
USER username used when connecting to the database
PASSWORD password for the user

The fields are set as environment variables in the `docker-compose.yml` file when specifying the services in the application as a whole. Then, in `setting.py` when setting the `DATABASES` dictionary, they are retrieved using `os.environ.get` function which takes the variable's name as a parameter and returns its value. This approach is used to avoid hardcoded values and makes the system more resistant to changes.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'HOST': os.environ.get('DB_HOST'),
        'NAME': os.environ.get('DB_NAME'),
        'USER': os.environ.get('DB_USER'),
        'PASSWORD': os.environ.get('DB_PASS'),
    }
}
```

Source Code 4.4: My Django project's database configuration

4.2.3 Custom admin commands

Writing custom Django commands can be done by creating Python modules in application's `management/commands` directory. Django will then register a

manage.py command for each module. The modules are required to define a class Command that extends BaseCommand or one of its subclasses. [50]

4.2.3.1 wait_for_db command

To prevent database errors that can occur when the Django server tries to access the database (running in the *db* image) before it is ready, I adopted wait_for_db command. This command was implemented by Mark Winterbottom in his project [51] that is publicly available on GitHub. It ensures that the database is up and running before accessing it. The command is used in the servers Dockerfile as shown in source code 4.2.

4.2.3.2 load_paintings_from_dir command

In load_paintings_from_dir.py I implemented a command for uploading a large batch of paintings from a directory on a local disk to the database. By implementing add_arguments method in the Command I added a positional string parameter directory_name which is the name of the directory containing the data. The directory **must** be located in the project's datasets directory.

```
admin@admin:~\$ docker-compose run --rm app sh -c "python3
↪  manage.py load\_paintings\_from\_dir Tate-500-paintings"
```

Source Code 4.5: An example usage of the command. Uploading all paintings in the Tate-500-paintings directory to the database.

4.2.3.3 test_accuracy command

This command is only for testing purposes and metrics evaluation. It works similarly to the load_paintings_from_dir command described in the section above, but instead of uploading the contents of the directory into the database, it performs a search query for each painting in the directory provided in the parameter. It tracks the statistics, and after all the paintings were searched, it prints the results in the console.

4.3 Database

This section describes the database models.

4.3.1 Models

Django model is a feature included in Django which is used to create database tables, defining their fields and various constraints [50]. All models used in this project are located in the core app of the project, in the models.py file.

4.3.1.1 Painting model

The `Painting` model represents a stolen/lost painting. A new instance is created when a victim uploads a new painting or when the server's administrator calls the command for mass upload, described in section 4.2.3. It has two fields:

- **name**
  - The name of the painting.
- **image**
  - A picture of the painting.
  - Implemented by an `ImageField` which in parameter accepts a function that generates the destination's filename. A filename provided by the user cannot be used for storing the image on the system due to the risk of clashing filenames (see code 4.6).

```
def painting_image_file_path(instance, filename):
    """Generate a file path for new painting image"""
    ext = filename.split('.')[-1]
    filename = f'{uuid.uuid4()}.{ext}'
    return os.path.join('uploads/painting/', filename)
```

Source Code 4.6: The function generating a universal unique filename using `uuid` that will be used for storing the image on the system.

It is worth mentioning that the model also has an **id** field, which is added automatically by Django.

4.3.1.2 PaintingDescriptors model

This model represents the binary descriptors acquired by the ORB algorithm. Each painting is represented by a maximum of 500 features, where each feature is an array of 256 bits; the whole process is explained in section 3.5.2. The model has two fields:

- **painting**
  - An **id** of the painting descriptors are tied to.
- **descriptors**
  - A 2-D integer array of size $n \times 32$, where n is the number of features.

– A row is composed of 32 numbers from 0-255, i.e., 32 bytes that together represent the 256 bit feature descriptor.

4.4 Feature extraction

Feature extraction is done using the ORB algorithm, described in section 3.5. The algorithm has been implemented in the `OpenCV` library. In this project, I use that implementation.

File `orb.py` contains my implementation of a class that wraps the ORB object. The class has a method that accepts a list of filenames and returns a list with a descriptor for each of them. The `__init__` method (constructor), specifies the default ORB parameters used in the project, described in table 4.1.

Parameter	**Value**	**Description**
`nfeatures`	500	The maximum number of features. If more features are found, only top $nfeatures$ are selected.
`scaleFactor`	1.2	Scale factor of the images in the multi-scale image pyramid (see 3.5.1).
`nlevels`	8	The size of the pyramid (number of levels).
`edgeThreshold`	31	Size of the border where the features are not being detected.
`firstLevel`	0	The level of the pyramid where the original image should be put. Images on below levels are upscaled.
`WTA_k`	2	Number of points used to compute one element of the descriptor. Here I use only 2, as described in the ORB [35] paper.
`scoreType`	HARRIS	Score used to order features.
`patchSize`	31	Size of the patch where the descriptor will be computed.
`fastThreshold`	21	The FAST threshold described in section 3.3.

Table 4.1: Description of ORB parameters [52] and their default values used in the project.

4.5 Feature matchers

For matching, I use FLANN-based matcher by default, but the project also contains a brute-force matcher, and it is possible to switch between the two (both described in section 3.6). As a basis, I use already implemented versions from the `OpenCV2` library. I created wrapper classes for the matchers in `matchers.py` file, which contains all matching-related functionality.

Then I implemented `get_best_matches` function that accepts the searched painting's descriptors in a parameter and returns $n_{nearest}$ matches, where $n_{nearest}$ is a number specified when calling the function – by default set to one, as now I only want one closest match. Using the specified matcher, a number of good matches between the searched painting and every painting in the database is obtained. Then, the paintings are ordered descendingly by the number of matches, and the **id**s of $n_{nearest}$ top matches are returned. If there are no paintings with at least a threshold number of matches or if not enough source features were provided (the searched descriptor is too small), none are returned – painting was not found in the database.

For both matchers, I implemented a `get_good_matches_number` method, which takes two descriptors as arguments and returns the number of good matches.

4.5.1 FLANN-based matcher

As recommended in [44], I configured the matcher to use the LSH algorithm for identifying the nearest neighbors. This is done by defining the `index_params` dictionary, which specifies the algorithm parameters and passing it as an argument when creating the matcher. The parameters are shown in code 4.7.

```
FLANN_INDEX_LSH = 6
index_params = dict(algorithm = FLANN_INDEX_LSH,
                    table_number = 6, # the number of hash
                    ↪  tables to use
                    key_size = 12, # length of key in hash
                    ↪  tables
                    multi_probe_level = 1 # number of levels to
                    ↪  use in multiprobe
                    )
```

Source Code 4.7: The index parameters

To determine the number of good matches between two descriptors, a k-nearest neighbor search (by default $k = 2$) is performed using the matcher. Then, the matches are compared using the ratio test to filter out ambiguous matches (see section 3.6.3).

4.5.2 Brute-force matcher

Because I use ORB with `WTA_k` set to 2, which means that the descriptor is a binary vector, I set the matcher to use Hamming distance (`cv2.NORM_HAMMING`) to compute the difference between two descriptors. A modified version of Hamming distance (`cv2.NORM_HAMMING2`) must be used if `WTA_k` is set to 3 or 4 [44].

By default, I use cross-check (described in section 3.6.1) to filter out bad matches. If `crossCheck` is set to `False`, the ratio test is performed instead.

4.6 API

Two endpoints were implemented:

1. **paintings**
   - Lists all paintings in the database.
   - Allows uploading a new painting.
2. **search**
   - Searches the database for a given painting and return k most similar paintings in the response.

In Django, defining operations for a given set of data is done using `ViewSets`. Then, using `routers` provided by Django REST framework (section 2.5.1), I define the URLs and specify the `ViewSets` that they provide access to, see code 4.8.

```
router = DefaultRouter()
router.register('paintings', views.PaintingViewSet, 'paintings')
router.register('search', views.SearchPaintingViewSet, 'search')
```

Source Code 4.8: Defining the URLs

4.6.1 SearchPaintingViewSet

This `ViewSet` makes the search operation available. The search parameters are an image (lost/stolen painting) and a value of k, signifying how many closest matches should be returned.

The `ViewSet` uses `ImageSearchParamsSerializer`, which is a serializer that defines the search parameters (see code 4.9).

```
class ImageSearchParamsSerializer(serializers.Serializer):
    image = serializers.ImageField()
    k = serializers.IntegerField()
```

Source Code 4.9: `ImageSearchParamsSerializer` serializer

After a user makes a search request, the serializer is used to validate the data. Then, a descriptor is computed for the validated image. The `get_best_matches` described in section 4.5 is then used to obtain the **ids** of k nearest matches. Finally, the paintings with corresponding **ids** are returned in the response.

4.7 Mobile client

The mobile client is made of two main tabs (see figure 4.1). Both are implemented as `Fragments`, managed by applications `MainActivity`.

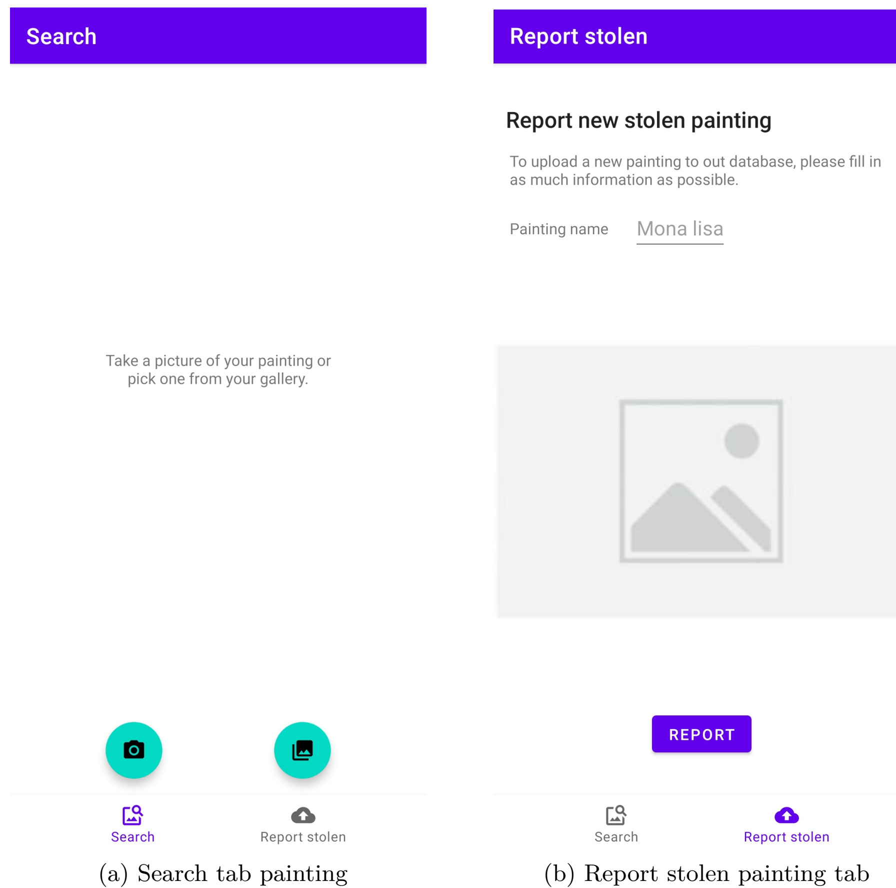
4.7.1 Search tab

Using this tab, a user (painting validator) can make search requests to the server. The user either takes a picture of the painting or selects it from the gallery. This is done using `Intents`, which are used to delegate work to other applications. In this case, I use two types of `Intent`:

1. `Intent.ACTION_PICK`
    - Allows the user to pick an image from any app registered for this action. Typically, the user will choose an image using the default gallery app.
2. `Intent.ACTION_IMAGE_CAPTURE`
    - Allows the user to take a picture using any camera app. The resulting picture is saved in a directory specified in the parameter.

After selecting a picture, the user can make the search request by pressing a button. Then, a request is made using the Retrofit interface described in section 4.7.4.

If the search fails for any reason (e.g., the painting was not found on the server or the connection failer), the app informs the user using a `Toast` popup. If a match is found, a new fragment is brought to the front (`FoundPaintingDetailFragment`), containing the information about the found painting.



(a) Search tab painting

(b) Report stolen painting tab

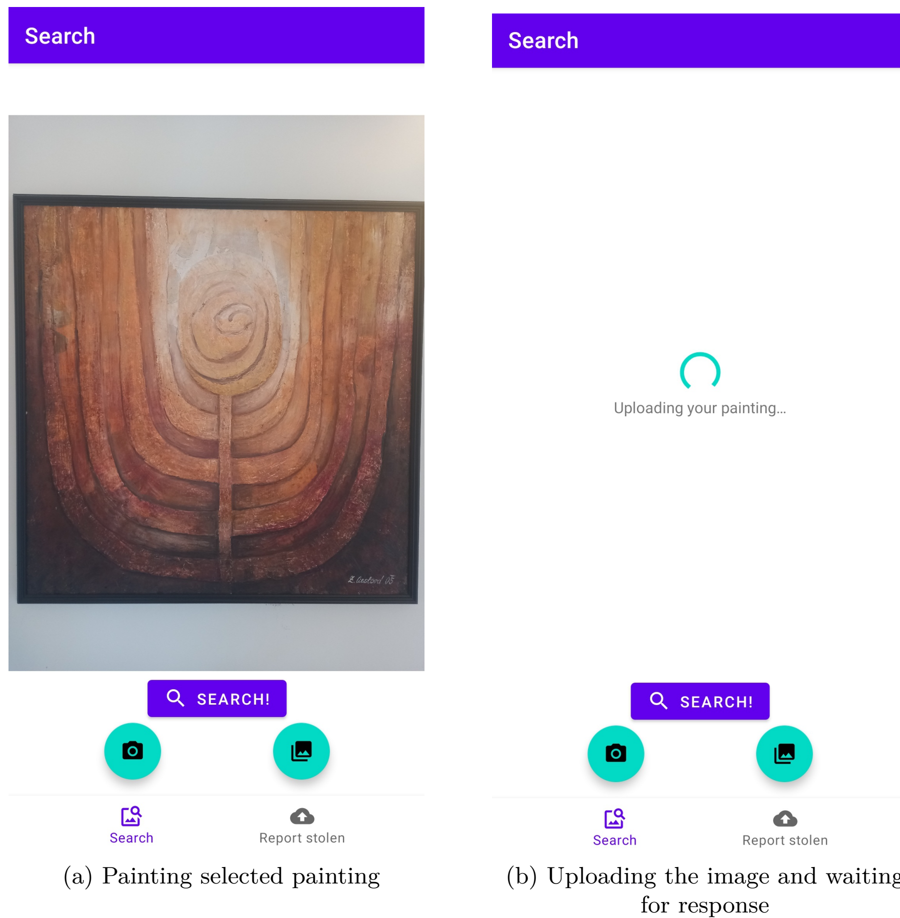
Figure 4.1: The two main tabs in the app

4.7.2 Report stolen tab

Using this tab, a user can upload a new painting to the server. Before making the request, a check is performed on whether all required fields are filled. The user is informed if the operation was successful based on the response. The implementation is similar to the section 4.7.1 above. The tab is visualized in figure 4.1.

4.7.3 Models

Two different painting models are defined in the `models` package, each serving a different function.



(a) Painting selected painting

(b) Uploading the image and waiting for response

Figure 4.2: Taking a picture and making a search request

4.7.3.1 FoundPaintingModel

This model has the same fields as the painting model on the server described in section 4.3.1. It is used for parsing the server responses into Java objects. This model does not store the image of a painting but only the URL of it (like the model on the server). When making a new call, it is passed as a type parameter, which signifies the successful response body type. An example of usage can be seen in code 4.10

```
SearchDBService service =
↪   APIClient.getRetrofitInstance().create(SearchDBService.class);

Call<List<FoundPaintingModel>> call =
↪   service.searchByPainting(body, k);
```

Source Code 4.10: SearchDBService interface.

4.7.3.2 Painting

The Painting model is similar to FoundPaintingModel, but in addition to the painting attributes, it also contains a bitmap of the painting, which was downloaded using the FoundPaintingModel URL attribute.

Because this model is mainly used for transferring painting data between fragments, it implements the Parcelable Java interface. This allows the Painting instance to be put into Bundle in source fragment using putParcelable method. In the destination fragment, it then can be simply retrieved by calling the getParcelable method on the Bundle.

4.7.4 Retrofit implementation

Networking and Retrofit related implementation is in the networking package.

The Retrofit instance is created in APIClient, a class with one static method getRetrofitInstance, which returns the instance or creates one if it has not been created yet. Note that before deploying the application, a correct BASE_URL **must be specified**. That is the address of the server where the requests are sent.

For making requests to the server, two interfaces were implemented:

1. SearchDBService
    - Used for searching the database by image, i.e., painting validation.
    - Expects list of k nearest matches (FoundPaintingModels). In current implementation, value for k is always 1.
    - See code 4.11.
2. ReportStolenService
    - Used for reporting new stolen painting.

```
public interface SearchDBService {
    @Multipart
    @POST("api/painting/search/")
    Call<List<FoundPaintingModel>> searchByPainting(
                                        @Part MultipartBody.Part image,
                                        @Part("k") RequestBody k);
}
```

Source Code 4.11: SearchDBService interface.

Chapter 5

Dataset

This chapter talks about the difficulty of obtaining real data of stolen paintings and communicating with institutions. Then it describes the dataset used for testing and evaluating the results.

5.1 Real datasets

As described in sections 1.3 and 1.4, there is not any central authority tracking all art thefts, which means that the data are scattered among institutions and private organizations.

The intention was to get data from police institutions that, unlike private organizations, have no commercial interest in not providing us with the data.

5.1.1 Interpol

As mentioned in section 1.3, INTERPOL operates a "Stolen Works of Art Database", a database of stolen works of art containing more than 50,000 items [13]. It is not completely public, but anyone can apply for authorization, which I did, and I was granted access.

Because it is not allowed to scrape the data from the website, I contacted INTERPOL and thoroughly explained my intention to create a noncommercial mobile application as a part of this thesis and asked them if there was a way to provide me with the data in their database. Unfortunately, I was not given the data nor received permission to download them myself.

5.1.2 PSEUD

The Police of the Czech Republic operates a database of stolen art called PSEUD. The database contains around 18,000 artworks. It has poor filter-

ing options, but from my analysis, it contains around 6,000 paintings. The database provides public unauthenticated access, unlike INTERPOL's SWOA. A lot of additional information about the registered items is visible on the website, like the size/weight of the art or material.

However, a major downside is that the maximum resolution in which the photographs of the paintings are provided is 200×200 pixels, even though the police often has the images in much better quality. This is clear because some artworks in PSEUD can be found in INTERPOL's SWOA, but in much higher quality, and as explained in section 1.3, it is the Police of the Czech Republic which is responsible for providing the data to INTERPOL.

I contacted the Police of the Czech Republic by e-mail in November 2020 with a request for higher quality images. Unfortunately, the request was declined without further explanation. After that, Ing. Marek Sušický send an official request based on Czech law "Zákon č. 106/1999 Sb." (Act on Free Access to Information). This request was declined as well, the reason being the victims' rights. As of April 2021, the next step is under negotiation.

5.2 Datasets for testing

For demonstration and testing purposes, I decided to search for nonstolen painting datasets. Obtaining a real dataset of stolen artworks proved very difficult and is not needed for testing and metric evaluation.

5.2.1 Tate Collection

Tate [53] is an institution operating four art galleries in Europe. It houses more than 75,000 artworks, including the United Kingdom's national collection of British art.

The Tate Collection is a GitHub repository containing metadata for around 3,500 artists and a large metadata file *artwork_data.csv* containing information about approximately 70,000 artworks, most of them being paintings [54]. The repository is no longer actively maintained, but it is being kept available for research purposes.

Using the URL columns, it is possible to access the image of almost every artwork in the metadata file.
The URL always starts with *http://www.tate.org.uk/art/images/work/* and the full resolution image is stored on the Tate website.

id	...	Title	...	medium	...	thumbnailUrl
1053	...	Job's Despair	...	Line engraving on paper	...	/A/A00/A00019_8.jpg
10542	...	Grotto in the Snow	...	Oil paint on canvas	...	/N/N05/N05254_8.jpg
9407	...	Entering the Fold	...	Oil paint on canvas	...	/N/N05/N05570_8.jpg
15003	...	Rievaulx Abbey	...	Watercolour on paper	...	/N/N05/N05615_8.jpg
4166	...	The Lute Player	...	Oil paint on mahogany	...	/N/N00/N00359_8.jpg

Table 5.1: Sample rows from the metadata csv file, important columns.

5.2.1.1 Downloading the data

This thesis includes a jupyter notebook that can be used for downloading images from the Tate website. It uses the `pandas` library to load the *Tate_artwork_data.csv* file (originally named *artwork_data.csv* in the GitHub repository) as a pandas dataframe. Then it filters out unwanted rows (each row represents one artwork) and, using the *thumbnailUrl* column of the remaining rows, downloads n images, where n is a variable specified in the notebook. At the moment of release of this thesis, the notebook supports filtering only by the medium. "*Medium can refer to both to the type of art (e.g. painting, sculpture, printmaking), as well as the materials an artwork is made from*" [55] is how Tate defines the term.

```
wanted_mediums = ['Oil paint on canvas', 'Oil paint on
↪   mahogany']
artworks_filtered =
↪   artworks_csv.loc[artworks_csv.medium.isin(wanted_mediums)]

# specify name of the destination folder, where the paintings
↪   will be stored
download_to = "Tate-oil-paintings-1000-samples"

download_images(artworks_filtered, download_to, 1000,
↪   generic_name = True)
```

Source Code 5.1: Downloading 1000 oil paintings.

In source code 5.1 we can see a short code that downloads 1000 randomly chosen paintings from the Tate Collection that satisfies the given constraints

– the medium is either 'Oil paint on canvas' or 'Oil paint on mahogany'. The files are saved in the folder specified by a parameter; if the folder does not exist, a new one with the same name is created.

Chapter 6

Testing

This chapter describes how the software was tested using augmented data and how to use the data augmentation notebook to generate custom data for testing. Then it discusses how well does the algorithm perform on various datasets with different augmentations.

6.1 Data Augmentation

As explained in section 5.2, datasets of nonstolen paintings were used for testing. Because it would not make sense to test the product with the same pictures of the paintings in the database, as that would not simulate the real conditions, data augmentation was used to obtain more pictures of one painting. The augmented images were then used for testing, regarded as stolen/lost paintings.

6.1.1 imgaug library

The official *imgaug* documentation [56] states: "*imgaug* is a library for image augmentation in machine learning experiments. It supports a wide range of augmentation techniques, allows to combine these easily and execute them in random order."

6.1.2 Sequential augmenter

The crucial part of my augmentation process is a sequential augmenter. `Imgaug.augmenters.Sequential` is an augmenter that contains a list of other augmenters and applies them in a sequence or in random order.

6.1.3 Used augmenters

This section contains an overview of all augmenters used in the project.

Augmenter	Parameters	Description
`GaussianBlur`	sigma range	Blurs the image with a gaussian kernel with sigma from the specified range.
`JpegCompression`	compression	Simulates the loss of quality caused by (repeated) JPEG compression.
`Affine`	rotate range, translate percent	Rotates the image by angle from the specified range. Translates it by x percent on the specified axis.
`Crop`	x pixels	Removes columns/rows of pixels at the side of the image.
`LinearContrast`	alpha interval	Modifies the contrast of the image according to $127 + alpha * (pixel_value - 127)$. Alpha is in the specified range.
`Cutout`	size, fill mode	Cuts out a random rectangular area of the specified size and fills it according to the specified fill mode.

Table 6.1: Description of the augmenters used.

6.1.4 Data Augmentation Notebook

The augmentation algorithm is implemented in `data_augmentation.ipynb` jupyter notebook. It augments images from a given directory with augmenters specified in the `augmenters_list` variable. A new folder is created, and the augmented images are saved here with their original name and `aug_` prefix. The notebook also contains a function to visualize the changes. An example of how to use the notebook is shown in source code 6.1.

```
# the directory with paintings to augment
to_augment_dir = "../datasets/Tate-1000-oil-canvas/"
# load the paintings, including their filenames (new filename is
↪  prefix+old filename)
paintings_filenames, paintings = load_paintings(to_augment_dir)
# specify the augmentors to be used
augmenters_list = [ iaa.JpegCompression(compression=(0, 30)) ]
# augment the images
augmented = get_augmented(paintings, augmenters_list)
# make a new folder in this directory with "original_name"+
↪  "_aug" postfix
```

```
aug_dir = to_augment_dir.split('/')[-2] + "_aug/"
# save the augmented paintings in the folder
save_images(aug_dir, paintings_filenames, augmented)
```

Source Code 6.1: Example of usage; applying a JPEG compression with 0-30 % quality loss on 1000 oil paintings from the Tate's Collection.

6.2 Measuring the results

This section explains which evaluation metrics were measured and how it was done.

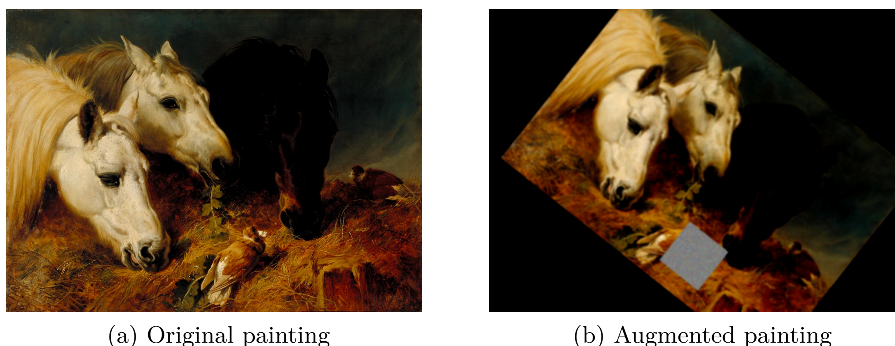
6.2.1 Datasets used for evaluation

Five different sub-datasets of The Tate Collection were used for testing the accuracy. Four datasets represent a different group of paintings based on which medium they were painted on, each containing exactly 1800 paintings chosen randomly from the collection (see table 6.2).

The final dataset includes all paintings that could be downloaded from Tate and falls into one of the four categories from the previous datasets.

#	Medium	Number of paintings	Notes
1	Oil paint on canvas	1800	Mainly "stereotypical" paintings.
2	Screenprint on paper	1800	Contains screenprinted photos/paintings, **modern art**, and images with text. A lot of similar, low-resolution images.
3	Lithograph on paper	1800	High occurrence of grayscale images.
4	Watercolour on paper	1800	Contains a very large number of paintings that look like a blank page of paper, some of them hardly distinguishable on first sight. Contains significantly more abstract paintings than the other datasets.
5	All above	9320	

Table 6.2: Sub-datasets of the Tate Collection that were used for testing.



(a) Original painting (b) Augmented painting

Figure 6.1: Comparison of a random painting before and after augmentation

6.2.2 Used augmentation

As explained in section 6.1, augmented paintings were regarded as stolen paintings and used for testing. Two different augmentation pipelines were tried, i.e., two different lists of augmenters.

The first augmentation pipeline tries to simulate fairly good conditions when the person making the search query has a good camera and has the time to take a good photo of the painting. The second pipeline simulates bad conditions where the person taking the picture has a low-quality camera or perhaps is taking the picture in a hurry. The paintings are more blurred, zoomed out, compressed, and more.

Both `augmenters_lists` variables, which precisely define the augmentation, are in the attachment of this work in separate files `augmenters_list-1.py` and `augmenters_list-2.py`.

6.2.3 Evaluation metrics

Three metrics were used to evaluate the software:

1. **Correctly found paintings**
    - Paintings that were matched correctly.
2. **Incorrectly found paintings**
    - Paintings that were matches, but with another painting.
3. **Not found paintings**
    - Paintings that were not found, i.e., did not match with anything.

6.2.4 Calculating results

The results for all datasets were obtained using the following steps:

1. **Flush the database** using Django built-in command `flush`.

    ```
    admin@admin:~\$ docker-compose run --rm app sh -c "python3
    ↪  manage.py flush"
    ```

2. **Upload the dataset** into the database using the custom command described in section 4.2.3.
3. **Take a random sample** of 250 paintings from the dataset.
4. **Use the** `data_augmentation.ipynb` jupyter notebook on the sample to obtain augmented data.
5. **Call the** `test_accuracy` command for search automation (see section 4.2.3).
6. **Repeat steps 3. - 5.** five times and take the average as the final result.
7. **Repeat** with the next dataset.

6.2.5 Results

The following table shows the achieved results. **Augmentation 1** is lighter than **Augmentation 2**, that simulates bad conditions, as explained in section 6.2.2. The datasets are numbered according to table 6.2.

Dataset	Augmentation					
	1			**2**		
	correct	incorrect	not found	correct	incorrect	not found
1	98.56	0.96	0.48	80.0	14.96	5.04
2	80.64	15.12	4.24	45.68	47.2	7.12
3	88.8	9.6	1.6	53.36	43.52	3.12
4	97.36	1.44	1.2	64.24	9.36	26.4
5	84.88	14.16	0.96	56.88	29.44	13.68

Table 6.3: Complete averaged results (%) for all datasets and both augmentation styles.

When simulating good conditions by mild augmentation, the results show near-perfect accuracy for oil paintings on canvas and watercolour paintings on paper. It achieves good results in the 80-90 % range for the other datasets, including **(5)** that contains over 9000 paintings.

The accuracy drops almost by half for some datasets when tested with the second augmentation. Paintings augmented this way have much lower quality and are rotated/zoomed more aggressively, see section 6.2.2.

Conclusion

The main goal of this thesis was to deliver a simple mobile application for recognizing stolen paintings. After analyzing the possible solutions, I developed software with three-tier architecture. The resulting mobile application is lightweight, simple to use and delegates the work to the server. The server implements RESTful API and is not tied to the mobile client, which opens the possibility of creating other mobile/web clients. All data is stored in a centralized database, which means near real-time data synchronization between the clients.

While working on this thesis, my supervisors and I tried to obtain stolen paintings from police institutions so that the resulting software could be immediately used in real situations. There is a lack of options to validate paintings, and when I started working on this thesis, even INTERPOL had no mobile application for that use, despite operating a database with tens of thousands of stolen works of art. When we communicated with INTERPOL via e-mail and explained our intention to create a mobile application for painting recognition and asked them for data, they did not mention any plans to create such app themselves. Our idea may have inspired INTERPOL, because a few months after our conversation they published their own app for validating paintings. As of now, April 2021, they still have not provided us with the data, and we are considering further steps.

The painting recognition algorithm is based on the concept of feature extraction, detailed in Chapter 3. The application was tested using datasets of nonstolen paintings and achieved fairly high accuracy and thus seems suitable for most real-world use cases. Processing a new painting uploaded to the database is done almost instantly, and deleting paintings from the database is done by simply removing the entries, without the need to retrain any models.

The whole project is publicly available on GitHub [1]. It is open-source and can be bettered by other developers or perhaps adopted by institutions. There are still many official institutions that operate databases of stolen art, which have no mobile application for uploading, and more importantly, validating paintings.

Future work

The goal to deliver a proof of concept to facilitate the discussion with institutions about data acquisition was fulfilled. The application works with good results and could be deployed as is. However, there are still several categories of possible improvements.

To increase the speed of painting search in the database and to sustain it even when there are tens of thousands of paintings, a metric access method for fast similarity search in the database should be implemented. Any metric index would decrease the number of needed comparisons between the paintings by orders of magnitude. Another speed (and perhaps even accuracy) improvement could be training a deep neural network (DNN) to categorize all paintings in the database by style. Any searched painting would then be first categorized by the DNN and then compared using feature matching only with the paintings in the same category.

For the server side to be entirely usable in production, an extension of the database model must be made. As for now, the `Painting` model does not store the information about how/where it was lost nor other known information about the painting. Adding new fields according to the needs should be simple to implement, both on the server and in the mobile application.

Ultimately, the goal would be to gather data from various sources to establish a free and accessible database and a mobile client allowing fast and accurate painting validation. Because this may not be possible due to the unwillingness of official institutions and commercial interests of private organizations to provide data, we hope that some institutions will be encouraged to make use of their data by adopting our application. I also believe that more institutions will be inspired and or even pressured to make their data public after this application gains on popularity.

[1] https://github.com/opendatalabcz/stolen-art

Bibliography

[1] Mark, J. J. Tomb Robbing in Ancient Egypt. [online], 2017, [cit. 2021-04-19]. Available from: https://www.ancient.eu/article/1095/tomb-robbing-in-ancient-egypt/

[2] Chua-Eoan, H. Stealing the Mona Lisa. [online], 2007, [cit. 2021-04-19]. Available from: https://web.archive.org/web/20070303065050/http://www.time.com/time/2007/crimes/2.html

[3] Janson, J. Vermeer Thefts: 1990 - The Concert. [online], 2016, [cit. 2021-04-19]. Available from: http://www.essentialvermeer.com/fakes_thefts_school_of_delft_lost_sp/vermeer_theft_05.html

[4] Artwork Archive. Victim of Art Theft? Do These 5 Things. [online], 2020, [cit. 2021-04-19]. Available from: https://www.artworkarchive.com/blog/victim-of-art-theft-do-these-5-things

[5] Durney, M. How an Art Theft's Publicity and Documentation Can Impact the Stolen Object's Recovery Rate. *Journal of Contemporary Criminal Justice*, volume 27, no. 4, 2011: pp. 438–448, doi:10.1177/1043986211418886, https://doi.org/10.1177/1043986211418886. Available from: https://doi.org/10.1177/1043986211418886

[6] Foley, T. *Art Loss and Databases: The Quest for a Free Single Unified System*. Dissertation thesis, Sotheby's Institute of Art, New York, 2014.

[7] Dolnick, E. Stealing Beauty. [online], 2004, [cit. 2021-04-19]. Available from: https://www.nytimes.com/2004/08/24/opinion/stealing-beauty.html

[8] Krever, M.; Woodyatt, A. Van Gogh painting stolen from museum shuttered by Covid-19 pandemic. [online], 2020, [cit. 2021-04-19].

Available from: https://edition.cnn.com/style/article/van-gogh-stolen-covid-19-intl-scli/index.html

[9] Hickman, M. Thieves snatch Van Gogh painting from Dutch museum shuttered during COVID-19 outbreak. [online], 2020, [cit. 2021-04-19]. Available from: https://www.archpaper.com/2020/03/thieves-snatch-van-gogh-painting-from-singer-laren/

[10] Amineddoleh, L. What It Takes to Recover a Stolen Work of Art. [online], 2016, [cit. 2021-04-19]. Available from: https://www.artsy.net/article/artsy-editorial-what-it-takes-to-recover-a-stolen-work-of-art

[11] The International Art and Antique Loss Register Limited. Art Loss Register. [online], 2021, [cit. 2021-04-19]. Available from: https://www.artloss.com/register/

[12] Taylor, K.; Manly, L. Tracking Stolen Art, for Profit, and Blurring a Few Lines. [online], 2013, [cit. 2021-04-19]. Available from: https://www.nytimes.com/2013/09/21/arts/design/tracking-stolen-art-for-profit-and-blurring-a-few-lines.html

[13] INTERPOL. Stolen Works of Art Database. [online], 2021, [cit. 2021-04-19]. Available from: https://www.interpol.int/Crimes/Cultural-heritage-crime/Stolen-Works-of-Art-Database

[14] Policie České Republiky. Portál systému evidence uměleckých děl. [online], 2021, [cit. 2021-04-19]. Available from: https://pseud.policie.cz/PSEUD_Internet/Vyhledani.aspx

[15] Arma del Carabinieri. iTPC Carabinieri. [Mobile application software], 2014, [cit. 2021-04-19]. Available from: https://apps.apple.com/cz/app/itpc-carabinieri/id858588594

[16] Internet Archive. The Wayback Machine: Site history of SWOA. [online], 2021, [cit. 2021-04-19]. Available from: https://web.archive.org/web/20210117222034/https://www.interpol.int/Crimes/Cultural-heritage-crime/Stolen-Works-of-Art-Database

[17] Engine Yard. Containers vs Virtual Machines Differences, Pros, & Cons. [online], 2020, [cit. 2021-04-19]. Available from: https://blog.engineyard.com/containers-vs-virtual-machines-differences-pros-cons

[18] Petlovana, Y. Top 13 Python Web Frameworks to Learn in 2020. [online], 2020, [cit. 2021-04-19]. Available from: https://steelkiwi.com/blog/top-10-python-web-frameworks-to-learn/

[19] Statcounter. Mobile Operating System Market Share Worldwide. [online], 2021, [cit. 2021-04-19]. Available from: https://gs.statcounter.com/os-market-share/mobile/worldwide

[20] Docker Inc. Docker overview. [online], 2021, [cit. 2021-04-19]. Available from: https://docs.docker.com/get-started/overview/

[21] Docker Inc. Overview of Docker Compose. [online], 2021, [cit. 2021-04-19]. Available from: https://docs.docker.com/compose/

[22] Mozilla and individual contributors. Django introduction. [online], 2021, [cit. 2021-04-19]. Available from: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Introduction

[23] DataFlair. Django Advantages and Disadvantages – Why You Should Choose Django? [online], 2021, [cit. 2021-04-19]. Available from: https://data-flair.training/blogs/django-advantages-and-disadvantages/

[24] Mays, C. Basics of Django Rest Framework. [online], 2018, [cit. 2021-04-19]. Available from: https://www.caktusgroup.com/blog/2018/02/26/basics-django-rest-framework/

[25] Bezkoder. Django: POST, PUT, GET, DELETE requests example. [online].

[26] The PostgreSQL Global Development Group. Official PostgreSQL Documentation. [online], 2021, [cit. 2021-04-19]. Available from: https://www.postgresql.org/about/

[27] The PostgreSQL Docker Community. Docker Hub postgres image. [online], 2021, [cit. 2021-04-19]. Available from: https://hub.docker.com/_/postgres

[28] Cervantes, E. What is Android? Here's everything you need to know. [online], 2021, [cit. 2021-04-19]. Available from: https://www.androidauthority.com/what-is-android-328076/

[29] Vogel, L.; Scholz, S.; et al. Using Retrofit 2.x as REST client. [online], 2020, [cit. 2021-04-19]. Available from: https://www.vogella.com/tutorials/Retrofit/article.html

[30] AbhiAndroid. Performance benchmarks for Android AsyncTask, Volley and Retrofit. [online], 2018, [cit. 2021-04-19]. Available from: https://abhiandroid.com/programming/retrofit#Need_of_Retrofit_In_Android

[31] Tamada, R. Android Working with Retrofit HTTP Library. [online], 2017, [cit. 2021-04-19]. Available from: https://www.androidhive.info/2016/05/android-working-with-retrofit-http-library/

[32] Hong, Y.; Kim, J. Art painting identification using convolutional neural network. *International Journal of Applied Engineering Research*, volume 12, no. 4, 2017: pp. 532–539.

[33] Lowe, D. G. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, volume 60, no. 2, 2004: pp. 91–110.

[34] Bay, H.; Tuytelaars, T.; et al. SURF: Speeded Up Robust Features. In *Computer Vision – ECCV 2006*, edited by A. Leonardis; H. Bischof; A. Pinz, Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, ISBN 978-3-540-33833-8, pp. 404–417.

[35] Rublee, E.; Rabaud, V.; et al. ORB: An efficient alternative to SIFT or SURF. In *2011 International Conference on Computer Vision*, 2011, pp. 2564–2571, doi:10.1109/ICCV.2011.6126544.

[36] Rosten, E.; Drummond, T. Machine learning for high-speed corner detection. In *European conference on computer vision*, Springer, 2006, pp. 430–443.

[37] Calonder, M.; Lepetit, V.; et al. Brief: Binary robust independent elementary features. In *European conference on computer vision*, Springer, 2010, pp. 778–792.

[38] Huang, J.; Zhou, G.; et al. A New FPGA Architecture of FAST and BRIEF Algorithm for On-Board Corner Detection and Matching. *Sensors*, volume 18, no. 4, 2018, ISSN 1424-8220, doi:10.3390/s18041014. Available from: https://www.mdpi.com/1424-8220/18/4/1014

[39] Huang, J.; Zhou, G.; et al. A New FPGA Architecture of FAST and BRIEF Algorithm for On-Board Corner Detection and Matching. *Sensors*, volume 18, no. 4, 2018, ISSN 1424-8220, doi:10.3390/s18041014. Available from: https://www.mdpi.com/1424-8220/18/4/1014

[40] Tyagi, D. Introduction to ORB (Oriented FAST and Rotated BRIEF). [online], 2019, [cit. 2021-04-19]. Available from: https://medium.com/data-breach/introduction-to-orb-oriented-fast-and-rotated-brief-4220e8ec40cf

[41] Luo, C.; Yang, W.; et al. Overview of Image Matching Based on ORB Algorithm. *Journal of Physics: Conference Series*, volume 1237, 06 2019: p. 032020, doi:10.1088/1742-6596/1237/3/032020.

[42] Tyagi, D. Introduction to Harris Corner Detector. [online], 2019, [cit. 2021-04-19]. Available from: https://medium.com/data-breach/introduction-to-harris-corner-detector-32a88850b3f6

[43] Rosin, P. Measuring Corner Properties. *Computer Vision and Image Understanding*, volume 73, 02 1999: pp. 291–307, doi:10.1006/cviu.1998.0719.

[44] Mordvintsev, A. OpenCV Docs: Feature Matching. [online], 2013, [cit. 2021-04-19]. Available from: https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_matcher/py_matcher.html

[45] FLANN contributors. FLANN - Fast Library for Approximate Nearest Neighbors C++ implementation. [online], 2021, [cit. 2021-04-19]. Available from: https://github.com/flann-lib/flann

[46] Muja, M.; Lowe, D. G. Fast approximate nearest neighbors with automatic algorithm configuration. *VISAPP (1)*, volume 2, no. 331-340, 2009: p. 2.

[47] Muja, M.; Lowe, D. G. Scalable Nearest Neighbor Algorithms for High Dimensional Data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 36, no. 11, 2014: pp. 2227–2240, doi:10.1109/TPAMI.2014.2321376.

[48] Lowe, D. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, volume 60, 11 2004: pp. 91–, doi:10.1023/B:VISI.0000029664.99615.94.

[49] The Debian Project. Reasons to Choose Debian. [online], 2021, [cit. 2021-04-19]. Available from: https://www.debian.org/intro/why_debian

[50] Django Software Foundation. Django documentation. [online], 2021, [cit. 2021-04-19]. Available from: https://docs.djangoproject.com/en/3.2/

[51] Winterbottom, M. REST API Course. [online] commit: 2626047ed2293b66efdfaccfec6c8954d7a6a9e4, 2018, [cit. 2021-04-19]. Available from: https://github.com/LondonAppDev/course-rest-api

[52] OpenCV developers. cv::ORB Class Reference. [online], 2021, [cit. 2021-04-19]. Available from: https://docs.opencv.org/3.4/db/d95/classcv_1_1ORB.html

[53] The Board of Trustees of the Tate Gallery. The Collection. [online], 2021, [cit. 2021-04-19]. Available from: https://www.tate.org.uk/about-us/collection

[54] Barrett-Small, R. The Tate Collection. [online], 2020, [cit. 2021-04-19] commit: a51d8afc988ed083557e2950f4d0b644e7719f4a. Available from: https://github.com/tategallery/collection

[55] The Board of Trustees of the Tate Gallery. Medium - Art Term. [online], 2021, [cit. 2021-04-19]. Available from: https://www.tate.org.uk/art/art-terms/m/medium

[56] Jung, A. imgaug 0.4.0 documentation. [online] Revision: 7443efbf, 2020, [cit. 2021-04-19]. Available from: https://imgaug.readthedocs.io/en/latest/

Appendix A

Acronyms

ALR Art Loss Register
API Application Programming Interface

BRIEF Binary Robust Independent Elementary Features

CNN Convolutional Neural Network
COVID-19 Coronavirus disease of 2019

DNN Deep Neural Network

FAST Features from Accelerated Segment Test
FLANN Fast Library for Approximate Nearest Neigbours

HTTP Hypertext Transfer Protocol

INTERPOL The International Criminal Police Organization
iOS iPhone Operating System

JSON JavaScript Object Notation

LSH Locality-Sensitive Hashing

ORB Oriented FAST and Rotated BRIEF

PSEUD Portál Systému Evidence Uměleckých Děl

REST Representational State Transfer

SIFT Scale-Invariant Feature Transform
SQL Structured Query Language
SURF Speeded-Up Robust Features
SWOA Stolen Works of Art Database

URL Uniform Resource Locator

YAML YAML Ain't Markup Language

Appendix **B**

Contents of enclosed CD

```
├── jupyter_notebooks ....the directory containing all the jupyter notebooks
│   ├── augmentation................the data augmentation implementation
│   ├── datasets .............. the implementation of downloading paintings
│   └── orb ........................the demonstration of the ORB algorithm
├── client ..........................the directory containing the client files
│   └── Stolen-Art-Finder..........the full source code of the mobile client
├── server.........................the directory containing the server-side
│   ├── app.............................the Django server implementation
│   ├── docker-compose.yml...................the docker-compose.yml file
│   ├── Dockerfile ..................... the Dockerfile for the server image
│   └── requirements.txt .................. the server requirements.txt file
└── results .................................... the directory with results
    └── full_results.csv ............................... the results in csv
```