HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

ins
INSTITUTE FOR
NETWORKED SOLUTIONS

Study on interactive face recognition

Term Project

Department of Computer Science

University of Applied Science Rapperswil

Fall Term 2017

Author(s): Michael Gerber & Victor Ruch

Advisor: Prof. Dr. Luc Bläser

Project Partner: INS Institute for Networked Solutions

1 TABLE OF CONTENTS

Aufgabenstellung Studienarbeit für Michael Gerber und Victor Ruch:

Studie über interaktive Gesichtserkennung

1. Auftraggeber und Betreuer

Diese Studienarbeit findet für das INS Institut für vernetzte Systeme statt.

Betreuer HSR:

- Prof. Dr. Luc Bläser, Institut für vernetzte Systeme, lblaeser@hsr.ch

2. Ausgangslage

In dem letzten Jahrzehnt wurden riesige Fortschritte bei der maschinellen Gesichtserkennung (Face Recognition) erreicht. Es existiert hierfür eine Vielzahl von gut erprobter Algorithmen, wie z.B. mit Deep Learning oder Algorithmen wie Eigenfaces, Fisherfaces und andere. Zudem gibt es etliche Libraries und Tools, die Gesichtserkennung bereits implementieren, wie z.B. doe Cortona Intelligence Suite auf Basis von Microsoft Azure, OpenCV und weitere. Ferner existieren aufbereitete Bildergallerien von Gesichtern, um die Erkennung zu trainieren und evaluieren.

In dieser Studienarbeit soll zuerst ein Spektrum von den relevantesten Algorithmen und Libraries für die Gesichtserkennung analysiert und anhand von Testdaten konkret bezüglich Eignung und Schwierigkeiten evaluiert werden. Schliesslich soll ein geeignetes Framework/Library benutzt werden, um einen Prototyp für die interaktive Erkennung von Gesichtern zu realisieren. Als idealer Show Case wäre es vorstellbar, innerhalb der Microsoft Hololens Personen anhand deren Gesichter wiederzuerkennen. Im Erfolgsfall liesse sich dies als ein Ausstellungsobjekt für die «Informatik zum Anfassen» für das Department Informatik benutzen.

3. Ziele und Aufgabenstellung

Das Ziel dieser Studienarbeit ist es, bestehende Face Recognition Verfahren zu evaluieren und einen Anwendungsprototyp für die Wiedererkennung von Gesichtern zu realisieren.

Dafür können verschiedene geeignete Technologien wie Cortona Intelligence Suite, OpenCV u.a. eingesetzt werden.

Die Arbeit hat folgende spezifische Ziele:

- Recherche des Standes der Technik auf dem Gebiet der maschinellen Face Recognition.

- Konkrete Evaluation der wichtigsten Face Recognition Verfahren anhand von Testdaten.
- Entwurf, Implementation und Testen eines Prototyps zur interaktiven Wiederkennung von Gesichtern basierend auf den analysierten Technologien, allenfalls mit eignen Algorithmen ergänzt. Idealerweise, aber nicht zwingend, als Use Case auf Basis der Microsoft Hololens.
- Experimentelle Evaluation des Prototyps und Fazit der gewonnenen Erkenntnisse.

4. Zur Durchführung

Mit dem HSR-Betreuer finden wöchentliche Besprechungen statt. Zusätzliche Besprechungen sind nach Bedarf durch die Studierenden zu veranlassen. Besprechungen mit dem Auftraggeber werden nach Bedarf durchgeführt.

Alle Besprechungen sind von den Studenten mit einer Traktandenliste vorzubereiten und die Ergebnisse in einem Protokoll zu dokumentieren, das dem Betreuer und dem Auftraggeber per E-Mail zugestellt wird.

Für die Durchführung der Arbeit ist ein Projektplan zu erstellen. Dabei ist auf einen kontinuierlichen und sichtbaren Arbeitsfortschritt zu achten. An Meilensteinen gemäss Projektplan sind einzelne Arbeitsresultate in vorläufigen Versionen abzugeben. Über die abgegebenen Arbeitsresultate erhalten die Studierenden ein vorläufiges Feedback. Eine definitive Beurteilung erfolgt auf Grund der am Abgabetermin abgelieferten Dokumentation.

5. Dokumentation

Über diese Arbeit ist eine Dokumentation gemäss den Richtlinien der Abteilung Informatik zu verfassen. Die zu erstellenden Dokumente sind im Projektplan festzuhalten. Alle Dokumente sind nachzuführen, d.h. sie sollten den Stand der Arbeit bei der Abgabe in konsistenter Form dokumentieren. Die Dokumentation ist vollständig auf CD/DVD/USB in 3 Exemplaren abzugeben. Auf Wunsch ist für den Auftraggeber und den Betreuer eine gedruckte Version zu erstellen.

6. Termine

Siehe auch Terminplan auf dem Skripteserver Informatik -> Fachbereich -> Studienarbeit_Informatik

15.09.17	Beginn der Studienarbeit, Ausgabe der Aufgabenstellung durch die Betreuer.
19.12.17	Die Studierenden geben den Abstract zur Kontrolle an Ihren Betreuer/Examinator frei. Die Studierenden erhalten vorgängig vom Studiengangsekretariat die Aufforderung mit den Zugangsdaten zur Online-Erfassung des Abstracts im DAB-Tool. Die Studierenden senden per Email das A0-Poster zur Prüfung an ihren Examinator/Betreuer. Das Poster muss nicht ausgedruckt werden. Es dient zu Übungszwecken für die Bachelorarbeit. Vorlagen sowie eine ausführliche Anleitung betreffend Dokumentation stehen auf

	dem Skripteserver zur Verfügung.
22.12.17	Der Betreuer/Examinator gibt das Dokument mit dem korrekten und vollständigen Abstract zur Weiterverarbeitung an das Studiengangsekretariat frei.
22.12.17	Abgabe des Berichtes an den Betreuer bis 17.00 Uhr.

7. Beurteilung

Eine erfolgreiche Studienarbeit erhält 8 ECTS-Punkten (1 ECTS Punkt entspricht einer Arbeitsleistung von ca. 25 bis 30 Stunden), siehe auch die Modulbeschreibung der Studienarbeit Informatik der HSR.

Gesichtspunkt	Gewicht
1. Organisation, Durchführung	1/5
2. Berichte (Abstract, Mgmt Summary, techn. u. persönliche Berichte) sowie Gliederung, Darstellung, Sprache der gesamten Dokumentation	1/5
3. Inhalt *)	3/5

*) Die Unterteilung und Gewichtung von 3. Inhalt wird im Laufe dieser Arbeit festgelegt.

Im Übrigen gelten die Bestimmungen der Abt. Informatik zur Durchführung von Studienarbeiten.

Rapperswil, den 18. August 2017

Der verantwortliche Dozent

Prof. Dr. Luc Bläser
Hochschule für Technik Rapperswil

3 ABSTRACT

In recent years, there have been significant advances in the field of face recognition. Many of them are based on deep learning. In our thesis, we investigated the state of the art face recognition technology and its possible applications. The main goal of our work is to build a prototype for the exhibition "Informatik zum Anfassen".

At the beginning, we evaluated different classification technologies (OpenCV, Microsoft Cognitive Services Face API, TensorFlow) and their capabilities. On the results we gathered, we built an early prototype for classification from phones. We changed direction and developed the second prototype that is capable of fully autonomous tracking.

Our prototype demonstrates with a full-stack example how embeddings can be used to train an unsupervised online clustering algorithm to track previously unknown people over different locations. To accomplish this, we developed a client-server architecture comprising multiple services to achieve high performance and scalability.

We are confident that we have achieved the goal of our thesis with our prototype. It shows the capabilities of modern face recognition technology by applying them effectively to a real world setting. In our opinion it would be a worthwhile addition to the exhibition.

4 MANAGEMENT SUMMARY

4.1 INITIAL POSITION

Recognizing faces is a complex problem that humans are able to solve rather well. Since birth the face is a crucial element of social interaction. A study made by the Stanford Vision and NeuroDevelopment Lab[1] suggests human infants are able to process faces earlier than other objects. Describing the human face in form of a computer problem has always been a difficult task. Most traditional algorithms for recognizing faces are limited. For example, face recognition system used for recognition in biometric passports have many constraints in face positioning, expression and lighting conditions.

The Information Age has brought exponential growth in generated data and computing power. This development opens the door for approaches that were not feasible in the past. One area of research that has seen great achievements over the last few years is the field of deep learning, which is inspired by the way neurons work and learn.

Our goal was to explore the capabilities of already existing face recognition services and technologies and document its possibilities and limitations. As a result of these newly gained insights, we developed a prototype as a possible showcase for the exhibition "Informatik zum Anfassen".

4.2 METHOD

We decided to compare two different cutting edge deep learning technologies: Face API a closed source cloud service from Microsoft Azure's Cognitive Services and Facenet an open source project based on Google's deep learning Framework TensorFlow. We performed various tests with a large set of publicly available celebrity face photographs and a small set of photographs taken by ourselves.

[1] http://jov.arvojournals.org/article.aspx?articleid=2121335

4.3 RESULTS

The result of this thesis is a working prototype of a distributed system capable of recording and differentiating people in real time without requiring any user interaction. It can support many camera clients at once and comes with an admin web user interface to explore the data.

4.4 OUTLOOK

While this prototype could theoretically be used in real world (e.g. as part of a security system), there are some adjustments that would need to be made. Primarily an authentication and fraud protection layer between client and server is required and scalability concerns are to be analyzed for a large number of clients.

5 INTRODUCTION

The task description of our thesis distinguishes itself by not defining what shape the end result was to take. Instead we were to explore the problem domain of face recognition, examine the capabilities of modern technologies ourselves and ultimately engineer a showcase application that would be representative of our findings. This allowed us a lot of freedom and contributed to our rather diverse portfolio of findings.

We set out to learn about the different problems and solutions in the field of face recognition. It was our goal to be able to compare the multitude of different approaches and subproblems in an informed manner. At the same time, we wanted to rapidly prototype a solution that we could take out into the wild and gather real world experiences with.

The first step was to evaluate data sets that fit the problem domain, so we could test and compare libraries and APIs. After this we narrowed down the candidates for comparison and conducted our tests on each of them. Armed with those results we finished the first prototype which was a native Android-Application connected to a REST-backend capable of classification and supervised learning.

This is where we realized some issues with the classification showcase we were working towards. After some consideration and exploratory design, we pivoted our showcase to an unsupervised solution. For this we built a desktop application that uses the webcam to record faces and send them to a central backend that clusters them to be able to distinguish between people without needing anyone to label them. As the final polish we wrote a Web-UI that would allow an Administrator to explore the collected data and gather insights.

6 METHODS

6.1 DATA SETS

The first thing we realized when we started looking into the topic of face recognition was the need to work on good data sets in order to gather results that would be representative of the real world. Our definition for good was that they needed to be reasonably diverse in people, ages, camera and lighting quality and angles of the pictures taken. In addition, we wanted to be able to recognize the people in the pictures by ourselves in order to reason about mistakes the software made in an informed manner.

There are some well documented public data sets available: The MsCeleb[2] data set seemed to fit our needs since it contained 10 million pictures of celebrities taken from the internet. A sample is shown in Figure 1. At further inspection we had to realize that a lot of the people in it only had one picture in many different variations. Even most of the good image sets for the more famous celebrities contained many duplicates. They occasionally also contained images of masks or even the odd car wheel[3]. After realizing this we decided to curate our own subset of the MsCeleb data set containing 74 celebrities with around 60 images on average. We removed all duplicates, non-human objects and obviously misplaced images from this data set while retaining images of the people with massive age differences, changes in hair color and style, as well as pictures with bad lighting or angles. This set is referred to as MsCeleb-Subset in the rest of this document.

Figure 1: Samples from the MsCeleb data set

A couple of weeks later, we also created a personal data set with 15 HSR students and at least 10 pictures each to verify against a bias of the pre-trained model for the MsCeleb data set. We took the photographs from different angles, with different lighting conditions and facial expression. See Figure 2. For privacy reasons, only images of the authors are included to give an impression of what that data set looked like.

Figure 2: Test samples from our personal data set

6.2 TensorFlow and Facenet

Ultimately, we decided to use the TensorFlow-based Facenet as the foundation for our showcase. This chapter outlines the reasons for this decision and our experiences using this technology.

[2] Microsoft Celebrities data set http://www.msceleb.org

[3] Because the images were automatically collected from the internet there were occasionally non-human objects in the data set

TensorFlow is a deep learning framework created by Google that is based on the idea of constructing an execution graph that can then be run on one or multiple CPUs, GPUs or even computing clusters. This flexibility allowed us to experiment with computationally expensive training tasks on our most powerful machine without having to worry about getting our application to run on other machines or, as we later decided, inside a Docker image.

We quickly realized that creating our own deep learning model and training it to a point where it could compete with the solutions that were already available as open-source was beyond what we could reasonably achieve within the scope of this thesis. Therefore, we elected to use one of the existing models that was licensed as open-source, since it would allow us to make adjustments as needed.

Consequently, Facenet was an obvious choice since it showed great performance and was easy to integrate. It is MIT-licensed software that has the TensorFlow model implementation at its core and also provides some helper scripts to do alignment or training of support vector machine classifiers. At first, we experimented with some of these classification approaches to gain a better understanding of the model's capabilities. Later, we only used this model graph definition and the corresponding pre-trained model within our showcase.

We considered training the model from scratch but had to abandon this idea because we learned it would take between 1000 and 2000 CPU hours to get it to the same point the existing pre-trained version is already at. Since that model was already trained on the same data sets we were considering, it did not seem particularly appealing to invest time into something that would most likely not yield better results.

We gained some interesting insights into the workings of the model though: Most notably, the fact that it was not trained to classify any particular face. Instead its output was a 128-element float vector (an embedding) that was describing the face in the input image. The provided classification example would take these embeddings and use them as the features for a support vector machine that was completely independent of Facenet or TensorFlow. This would later become important when we were looking into alternatives to the supervised classification showcase we initially planned (as described in the section Limitations).

6.3 Comparing Different Technologies

The initial task description suggested comparing OpenCV, Microsoft Cognitive Services Face API and possibly others. It was important to us to include a modern deep learning approach that was not a closed source solution. Thus, Facenet became the third candidate.

Our online research showed that deep learning models have been outperforming the traditional methods implemented in the OpenCV face recognition library. The OpenCV Website stated an accuracy of 0.62 on the LFW (Labeled Faces in the Wild) data set[4], while Facenet states an accuracy of 0.992 on the same data set[5]. Therefore, we decided to focus on comparing Microsoft Cognitive Services Face API with Facenet. While exploring the Facenet classification approach, we observed that we could improve the accuracy by replacing the support vector machine with a stochastic gradient descent model using a logistic regression loss function. This lead to the third set of measurements labeled "SGD (log)" in Figure 3.

[4] OpenCV 3.0.0 Benchmarks on LFW data set: https://docs.opencv.org/3.0-beta/modules/datasets/doc/datasets/fr_lfw.html

[5] Facenet Benchmarks on LFW data set: https://github.com/davidsandberg/facenet/wiki/Validate-on-lfw

There are, of course, different ways to measure the performance of classification algorithms but we decided that accuracy was our main criterion. Since our initial use case had people taking and labeling pictures of themselves, we felt that high accuracy at a low number of samples was important to us.

Because of this, we defined that the experiment would be to select a training subset of one to seven images from each of the 15 people in our personally collected data set and compare the results against three images per person that were not part of the training set.

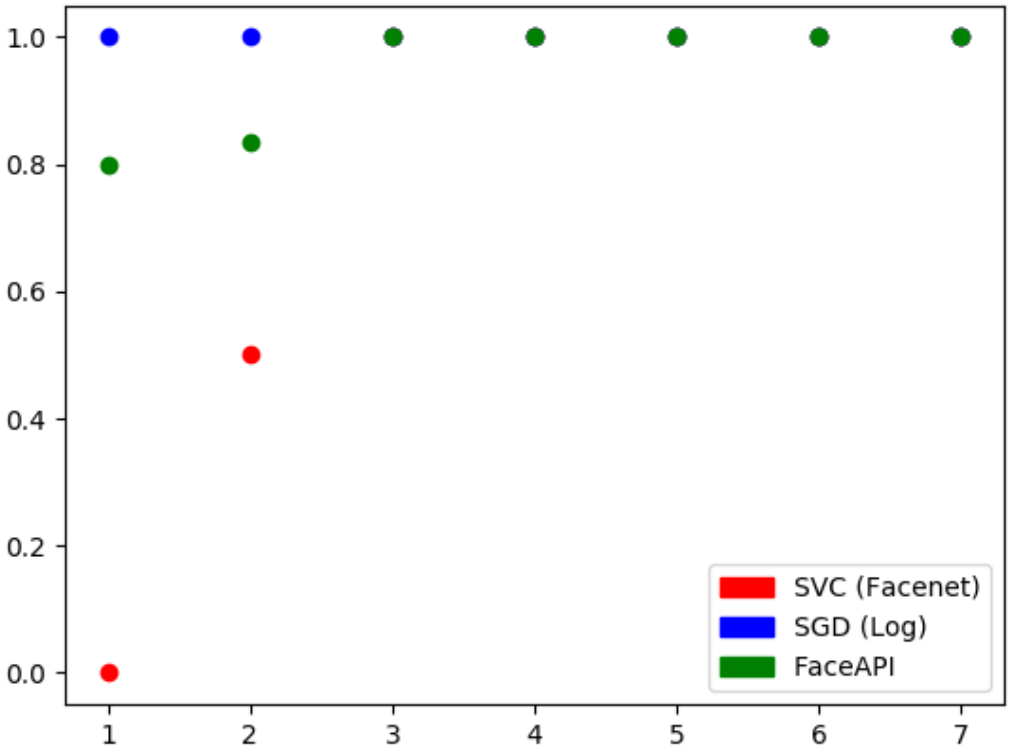

Figure 3: Number of samples per person to accuracy of classification as fraction on the personal data set

While engaging with the Cognitive Services Face API we encountered that the detection step would reject many of our images because it could not be certain enough about the position of the face. Unfortunately, there was no way to override this behavior and we had to exclude many of the images that the Facenet implementation could detect without issues. In other words, it should be kept in mind that this measurement was performed on the 10 easiest images per person.

This experiment shows that the approach using Facenet embeddings to fit the stochastic gradient descent model (labeled SGD(Log) in Figure 4) outperformed the others at a level that is hard to quantify since it could perfectly classify the 45 training samples even with just one sample per class.

It should also be considered that the differences show in test runs with only one or two samples per class which means there were only 15 (and 30) samples used for training and three times that amount for verification. With a sample set this small there is a considerable margin for error. We would like to note that our decision against Face API was mainly due to its inability to accept the harder samples and its worse performance in this test was only a minor factor.

Since the sample size was restricted by the limitations of Cognitive Services Face API we decided to run a similar test again without Face API on the full data set. We also included another model using a modified Huber loss function for gradient descent[6].

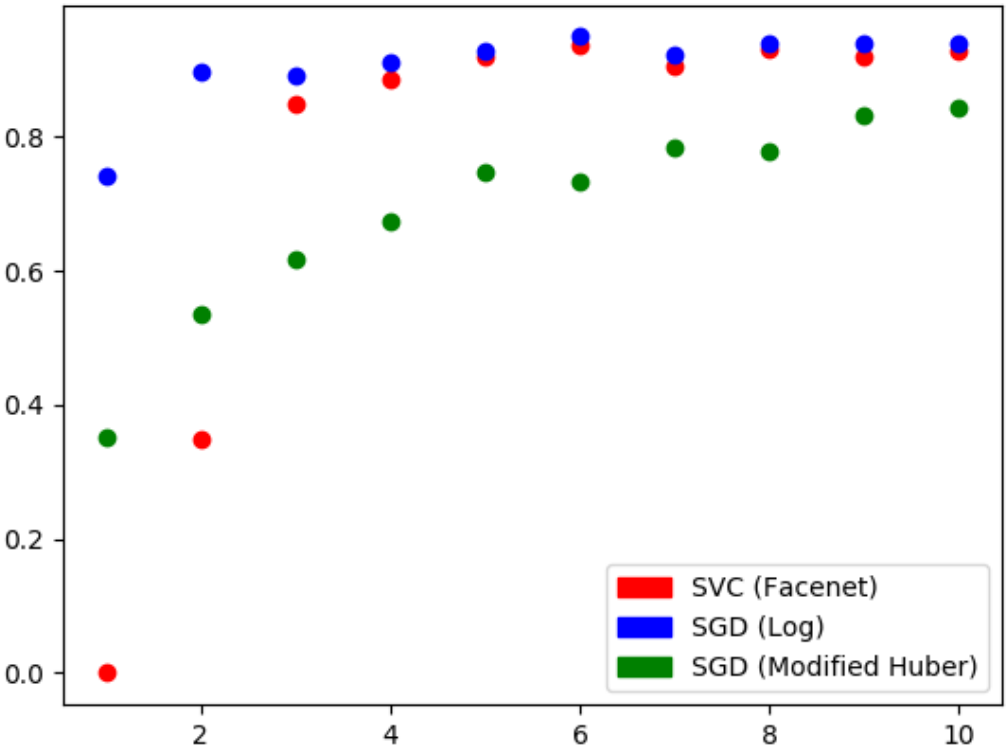

Figure 4: Number of samples per person to accuracy of classification as fraction run on the personal data set.

This showed us two things. Firstly, the SGD(Log) model was a lot better suited for smaller sample sizes. Secondly, even at larger sample sizes the SGD(Log) model consistently outperformed the support vector machine Model included in the Facenet examples.

This looked very promising but there was a different metric by which to measure the models that could be equally as decisive for our showcase: The model's behavior with an increasing number of classes.

[6] While the scikit-learn library supports many different loss functions, the only ones that support probabilistic predictions (meaning we receive some measure of how certain the model is about the result) are Logistic Regression and Modified Huber. We included it to see if changing the loss function would yield different results.

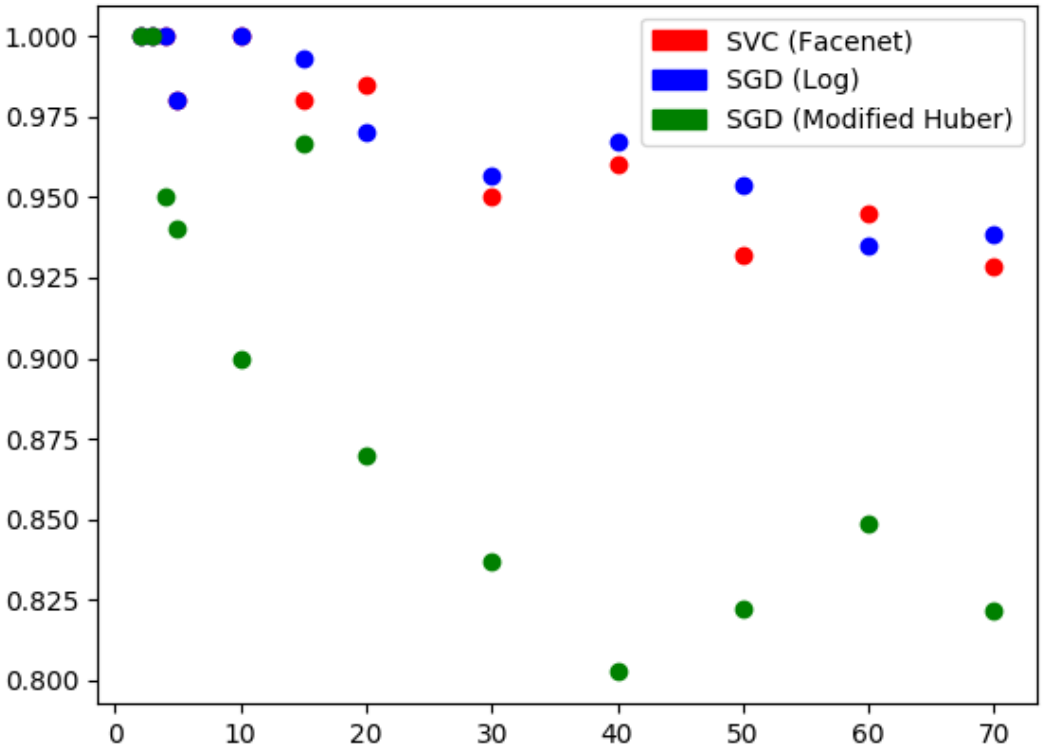

Figure 5: Number of classes to accuracy of classification as fraction on the MsCeleb data set.

We ran a similar test in Figure 5 to the previous one in Figure 4 but instead of increasing the number of samples we set a fixed number of 10 samples per class and increased the number of classes. For this we had to use the MsCeleb-Subset data set because the personal data set we collected simply did not have enough classes to yield significant results.

The test in Figure 5 shows us that the SGD(Log) and SVC models behave identical up to about 15 samples. After that they diverge on some samples but stay more or less on par with each other. Again, the SGD(Modified Huber) model performs a lot worse. Since the Huber loss function is known for being more robust against outliers we assume its worse performance stems from its inability to adapt when only few samples per class are provided.

6.4 The First Showcase

After we had gathered some experiences and data by training models on static data sets, we started working on a showcase. We wanted to highlight the impressive accuracy achievable with face classification. To this end we came up with a use case that supported classification.

In our initial showcase visitors entered a room with two stations called A and B. In station A a visitor could take a picture of himself and enter a name. As soon as he got to station B a camera would recognize his face and show the name he entered on a large display. This showcase requires station A and station B to communicate either directly or through a server. Both stations would need a display and a camera. Station A would require a keyboard, so the user could enter their name.

6.5 The First Prototype (Android)

We felt it was important to quickly build a prototype, shown in Figure 6, that would allow us to interactively take pictures in different lighting and from different angles and see how the models would handle them. We came up with the idea of writing an Android App that could communicate with a

RESTful JSON-API that would calculate the embeddings, try to classify the image and be able to retrain the classifier if the user entered a label for the image. This was very important in helping us understand what types of images it performed well on and which ones it could not handle.

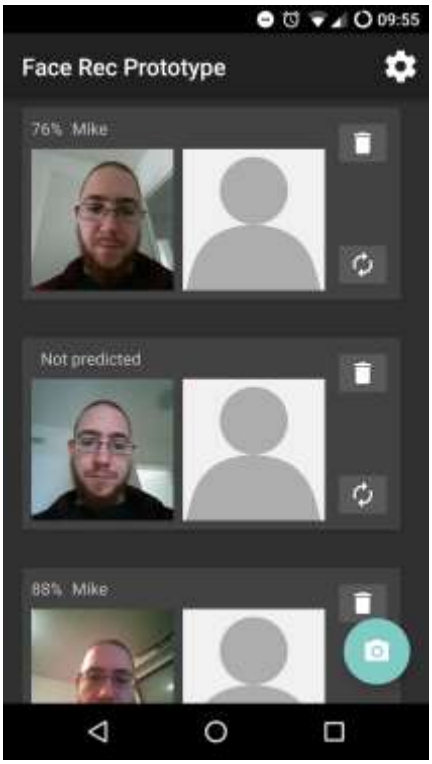

Figure 6: Screenshot from the Android prototype

This would also help us with the showcase we had planned, since the requirements on the server side would be the same and first experience in OpenCV face detection could be gathered.

In general, we were pleased with the accuracy of the classifier. A new person needed to only enter one or two sample images to get recognized with a very high probability. However, for this use case to be viable we had to make sure the time to retraining the classifier would not become a problem once a lot of labeled samples had been collected. To measure this, we trained the classifier with differently sized sets of random samples from the MsCeleb-Subset data set. The test results are shown in Figure 7.

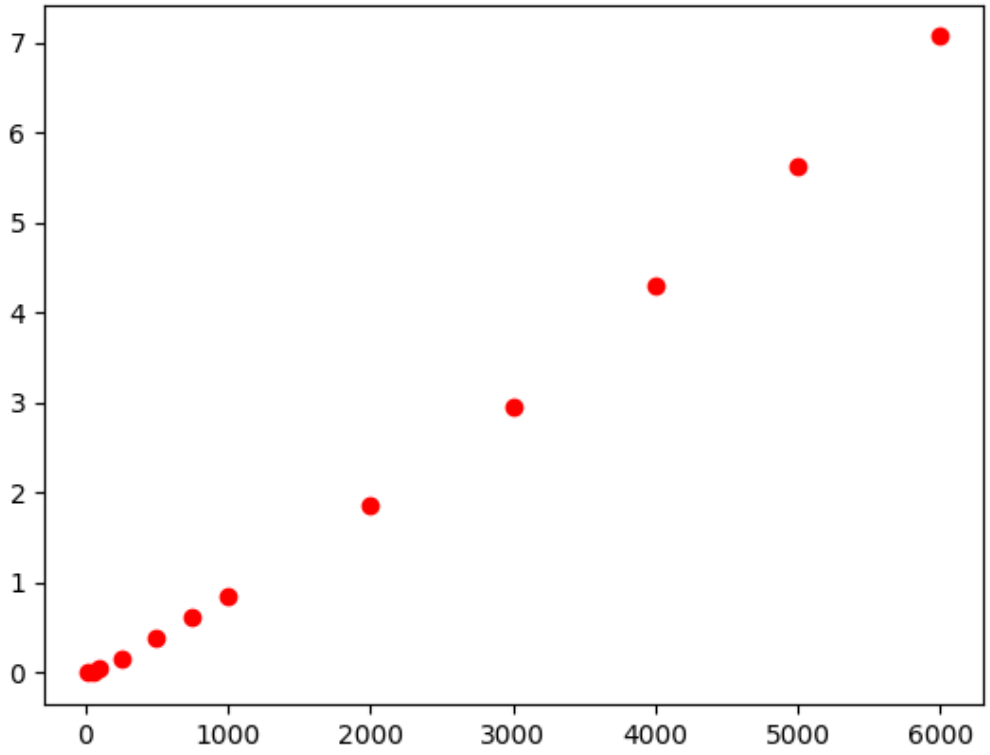

Figure 7: Number of unique samples to time consumption in seconds

6.5.1 The API-Server

The first proof of concept implementation of the server component was an API-only Rails Application. It would take the image, store it on the disk and call a python script to do classification or retraining of the model. While this full-stack architecture did in fact serve as a proof of concept we quickly realized it would not be viable for the finished product. The main issue was that the loading time of the TensorFlow model took almost a minute even on our strongest machine. By invoking it through a script there was no way for us to reduce the loading time significantly. The best we could do was storing it on a ram-disk which only marginally improved the performance. We had to somehow keep the model loaded between requests.

Our first approach was to use TensorFlow Serving[7] which is a way to expose TensorFlow models as gRPC[8] services. It seemed like an elegant solution to our problem but proofed to be tedious to set up. Basically, it would require rewriting some core parts of the model and there was very little documentation on how to do it with image inputs. While trying to convert the model, we encountered many obstacles and ultimately decided to look for alternative solutions.

Since our only requirement was that we would be able to keep the model in memory and grant access to it through a web protocol we elected to give Flask[9] a try. Flask was able to turn any python script into a web service with just a few lines of python code. This meant that we could simply load the model at startup and then continue to serve the model via JSON interfaces. Note that at this point, there was only one Flask service that contained both the embedding model and the classification model. We would later extract them into separate services which would ultimately allow us to perform embedding on the client.

---

[7] TensorFlow Serving: https://www.tensorflow.org/serving/
[8] gRPC: https://grpc.io/about/
[9] Lightweight Python Web Framework http://flask.pocoo.org/

6.6 LIMITATIONS

From the start, we always had some kind of showcase in mind that would demonstrate to people what was possible with current technology. Our prototype was built around the idea of a classification use case. This requires a human to enter appropriately labeled images such that the system can learn the identity of a person. We explored the idea of having people - at a convention or something similar - come up to a PC with a webcam, take their photo and enter their name (e.g. “Mark”). Then they could walk around on the show floor and there would be another computer with a camera that analyzed people walking by in real time and if it recognized them it would show their picture and say something like “Hello Mark!”.

Unfortunately, there were two big issues with this showcase. For one thing, it was very clunky since it required active user participation at the designated PC. The larger issue were privacy and legal concerns. The main problem here was that people would enter identifying labels, most likely their names. In order for us to be allowed to store that information we would have to get a written statement of consent from people. This would require us to have a printer on the premises, have someone to collect the signed statement of consent and would most likely guarantee that nobody would be interacting with the showcase.

6.7 THE SOLUTION (CLUSTERING)

At that point we decided to deviate from the original idea to one that would not require any active participation. What if we could recognize people without labeling them and just show them the previously taken images? We could set up any number of computers with webcams that would recognize people walking by and show them images taken by all the other cameras that recognized them earlier.

For this to work, we would have to switch from a classification model to a clustering one. And since it would have to be constantly updated, it would have to be capable of online training. This was not something we had much experience with but knowing the nature of the underlying embeddings we were confident that an existing solution could be adapted for what we were trying to do.

Before we looked at available clustering models we decided that we wanted the model to have two important properties. Firstly, it needed to be trainable online since it should be able to accept new samples quickly (cameras would send multiple images per second). Secondly, we wanted it to support a threshold parameter for “sample to cluster similarity” instead of simply a parameter to control the number of clusters since that would change as time and number of recorded people increases.

The python library scikit learn[10] offers a collection of different clustering algorithms but only two of them are capable of online training or partial updates as it is called in their documentation. The first one is a K-Means variant while the other is a tree-based model called Birch. Of the two, only Birch supported a threshold parameter referring to the cluster radius. Therefore, we decided to build a prototype with Birch and only start looking at alternatives if we were unsatisfied with its performance.

The next step was to come up with a measurement for the performance of the model that would not solely rely on empirical findings on the working system but could be expressed as comparable numerical values. This was not as simple as with classification models where one can just calculate the ratio of correctly classified samples.

[10] Scikit Learn, Machine Learning in Python: http://scikit-learn.org/

We decided on two numbers, that would sufficiently express the accuracy of the model for our needs. The first one, which is depicted as red dots in Figure 8, is the silhouette-coefficient[11] which essentially expresses how well defined the clusters of a model are. It ranges from -1.0 for the worst values to 1.0 for the best. The second one, which is depicted as blue dots in Figure 8, is a measurement of how close the number of different classes in the training data reflects the number of classes/clusters in the trained model. It ranges from 0.0 for the worst values to 1.0 for the best. We calculated these measurements with 500 randomized samples from the MsCeleb-Subset for different thresholds of the Birch model.

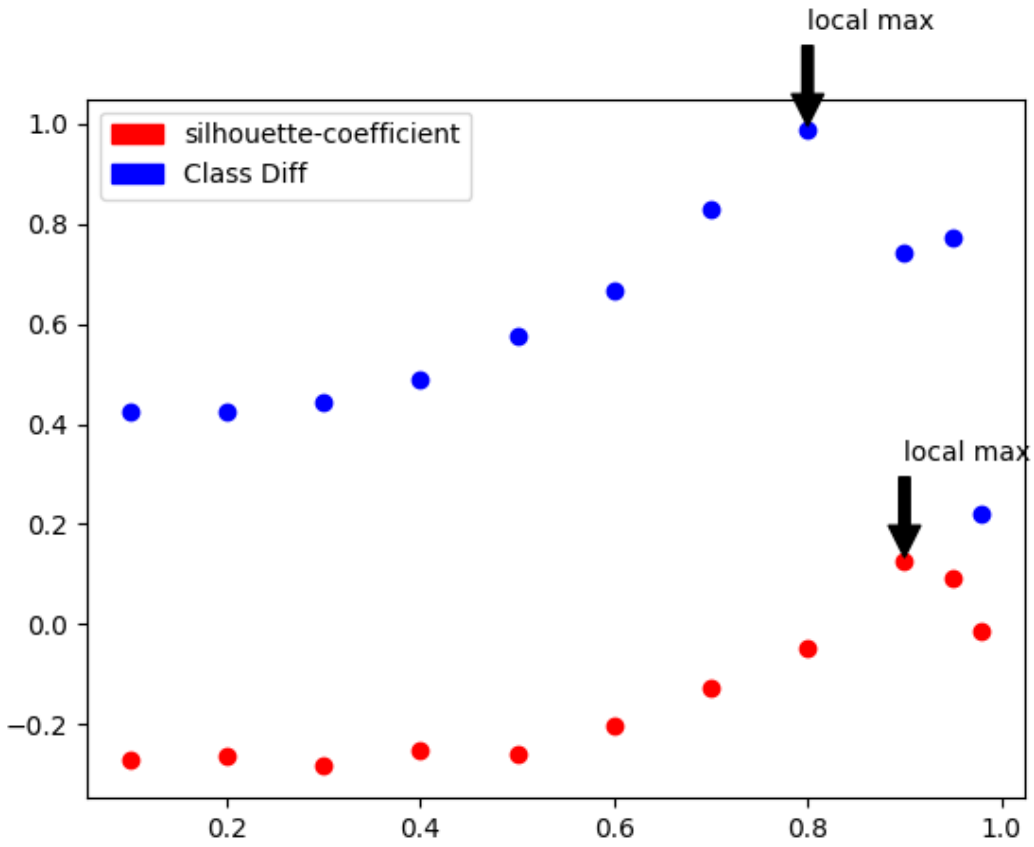

Figure 8: Silhouette-coefficient and class difference for Birch threshold from 0.0 to 1.0 in 0.2 increments

Figure 8 showed us that the optimum would be located somewhere between a threshold of 0.8 and 1.0 so we ran the same test again for smaller increments in that range.

[11] Silhouette Coefficient : http://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score.html#sklearn.metrics.silhouette_score

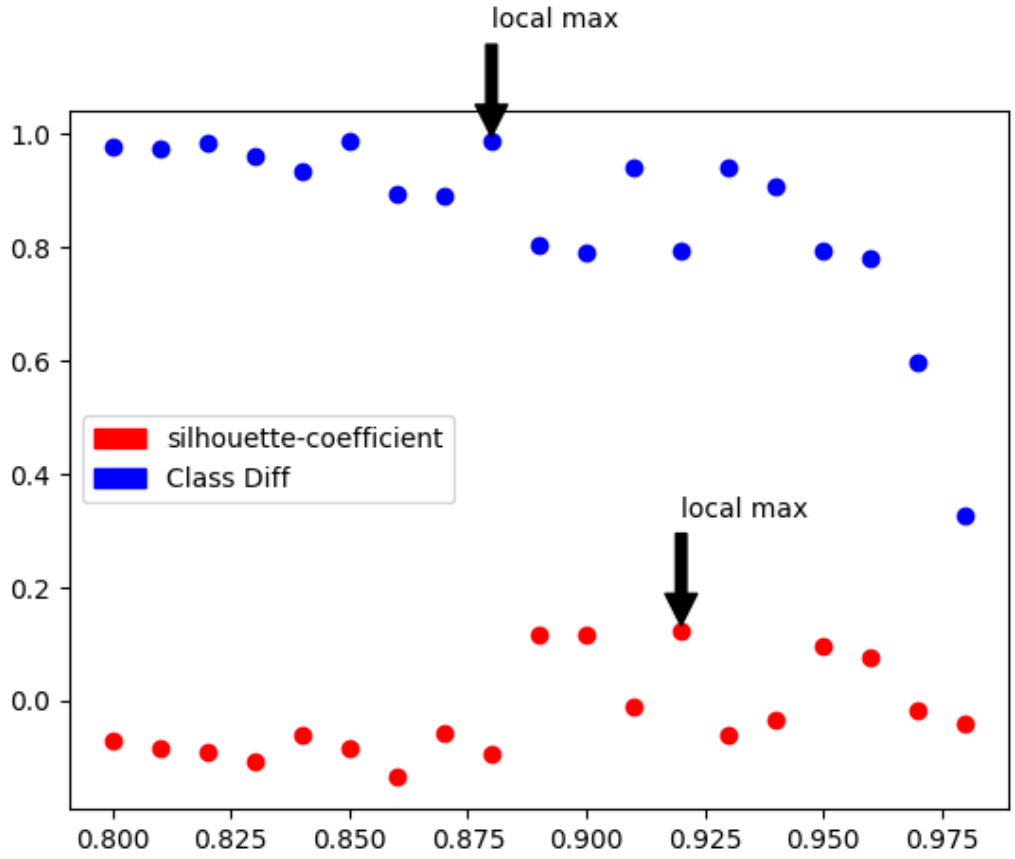

Figure 9: Silhouette-coefficient and class difference for Birch threshold from 0.8 to 1.0 in 0.025 increments

From Figure 9 we learned that, while there is no perfect number to satisfy both requirements simultaneously, a threshold of 0.9 should perform reasonably well. So, we decided to collect the same measurement with a fixed threshold of 0.9 for different number of samples to see if accuracy would decrease with a high number of processed images.

The answer was no, as the performance was relatively stable between 1000 or 6000 samples as we see in Figure 10.

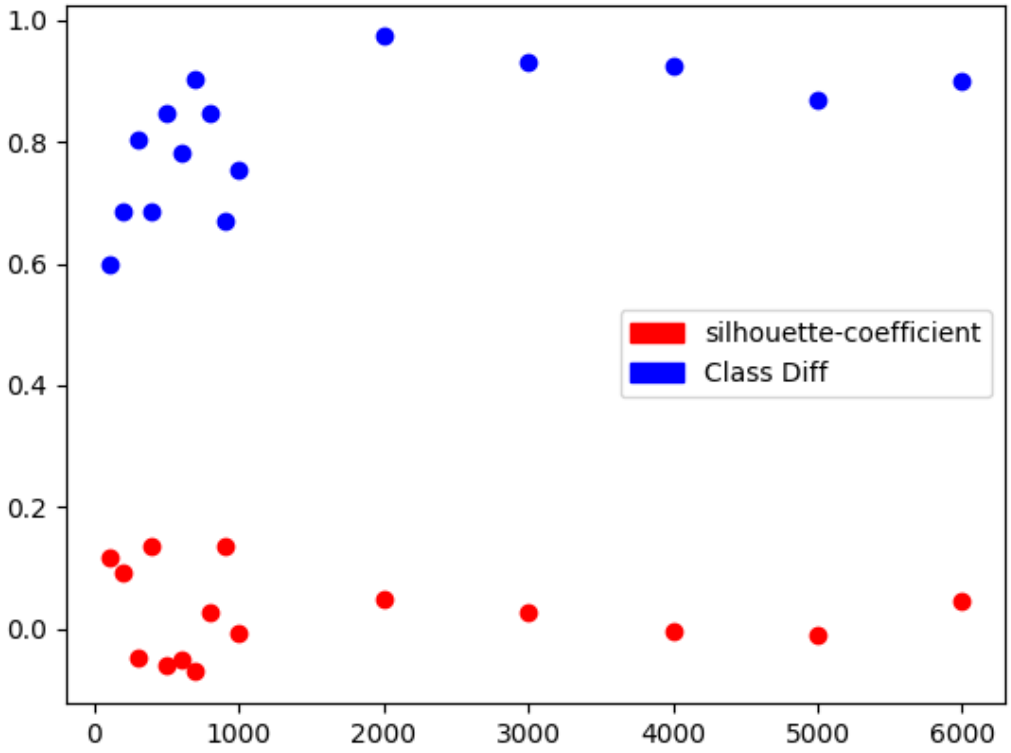

Figure 10:With fixed threshold of 0.9 measuring sample size to accuracy.

Once we applied the 0.9 threshold to our prototype we observed some rather sobering results. While the clustering sometimes would work almost flawlessly we also observed instances where most samples would be thrown into one cluster/class even if they contained obviously different faces. There would also be a multitude of miniscule clusters with people who were already in larger clusters.

The problem here was not that our measurements were inaccurate but rather that we omitted two crucial properties of the real-world application. The first one was that perceived accuracy of the system would be impaired a lot more by a false-positive match to a cluster than occasional pictures being wrongly assigned to a new cluster. The larger issue was that our tests were not only based on randomly picked samples but also randomly ordered samples. In the real world many pictures of the same person would be fed to the model consecutively such that the algorithm would be subject to something similar to overfitting. Instead of building the tree in a way to distinguish different people it would focus on the wrong features, the ones it could use to distinguish between the many pictures of the same person (the one first observed by the system).

The observable effect was similar to a black hole where one cluster would grow so large that most new faces would be swallowed by it. This meant we had to conduct tests to create clusters that were similar whether they were created by randomized data or consecutive data. By empirical testing we concluded that a threshold of 0.63 would yield much more desirable results on data captured by the finished prototype. Unfortunately, we did not have the resources to manually label the over 4400 images we used for this finding which is why there is no proof in numbers for it.

Figure 11 shows a snapshot of what our top 4 clusters looked like after the adjustment. The number on the left is the cluster ID and next to it in parentheses is the number of samples in that cluster. People other than the authors have been blurred to protect their privacy.

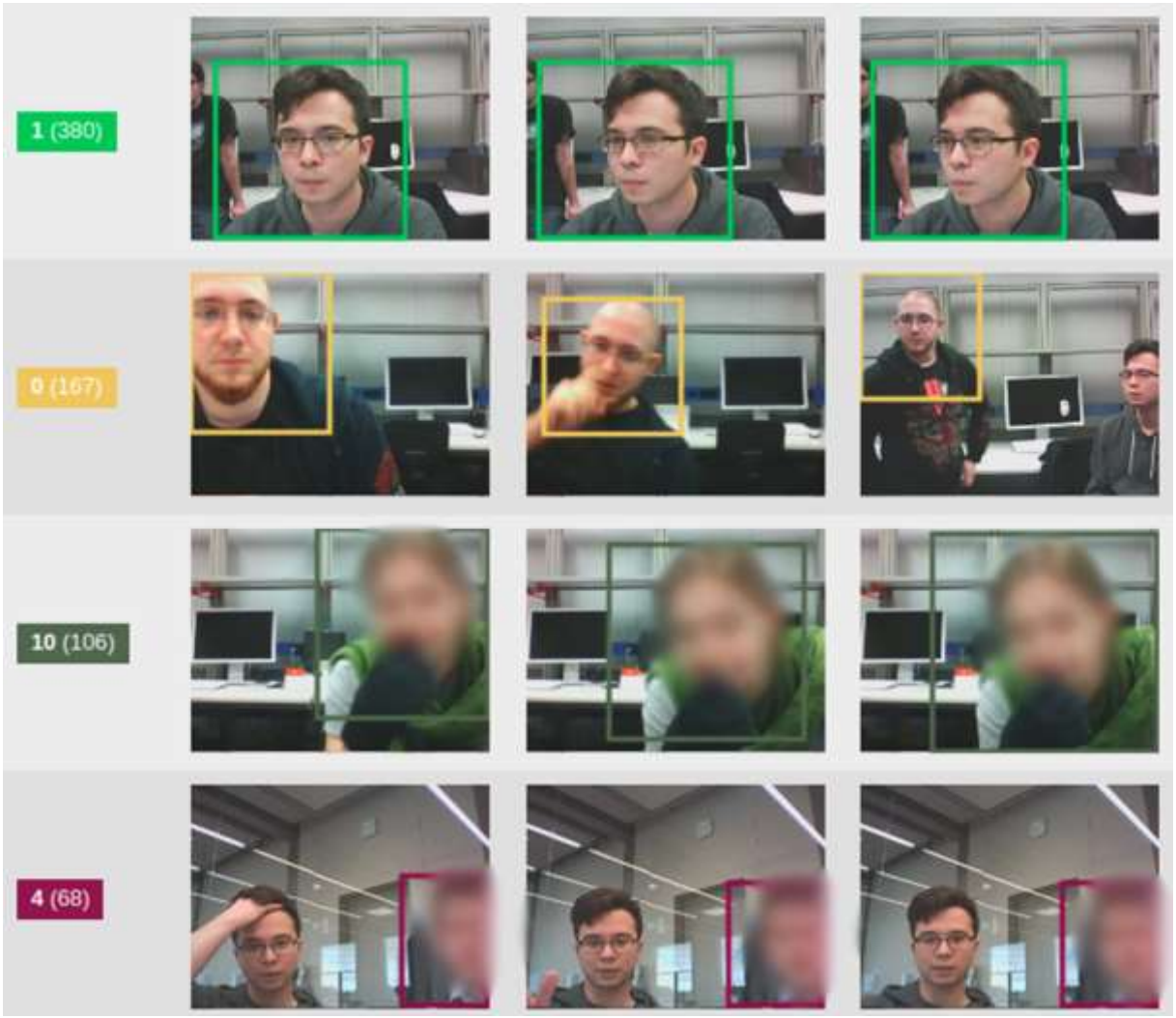

Figure 11: Snapshot of top 4 clusters in our application

7 IMPLEMENTATION

7.1 OVERVIEW

The finished product is a client-server application supporting many simultaneously running clients. It consists of a desktop client capable of interfacing with the webcam, locally embedding images, sending images to the server and displaying the HTML-view containing images for the person last recognized at its location. The server provides the JSON-API for the client to interface with and some HTML-views to browse the relationally stored data.

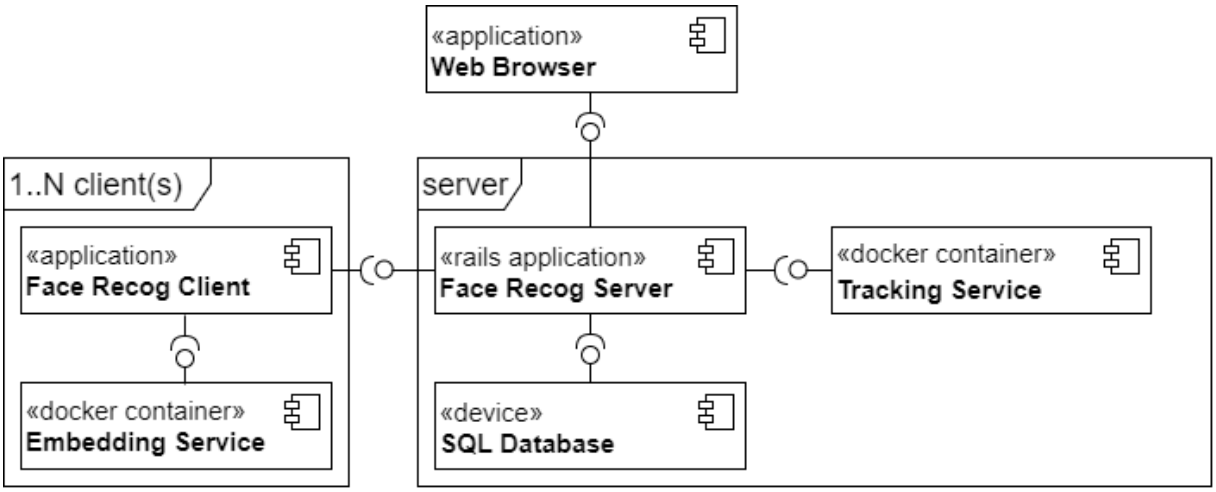

Figure 12: Components and its dependencies

7.2 DOMAIN-MODEL

The data stored on the server side requires only three entities shown in Figure 13. At the top of the hierarchy is the **Location**. It stores a unique name that identifies a client. For it many **TrackedImages** can be can be stored. Each **TrackedImage** can contain many **Trackings** which each refer to a different person in the captured image. The **Tracking** ultimately stores positional information about where in the image the face is located (height, width, left, top), the assigned label and the calculated embedding.

Beyond this relationally stored data in the database there is some additional persistent data. Firstly, we have the actual images which are stored in the file system adhering to a path schema that can be inferred from the **TrackedImage** record in the database. The other semi-persistent part of the system is the clustering model. Its state is never written to disk, but the data is persistent until the service is restarted. It can be recreated by reprocessing the stored **Trackings** in the same order they were originally received.

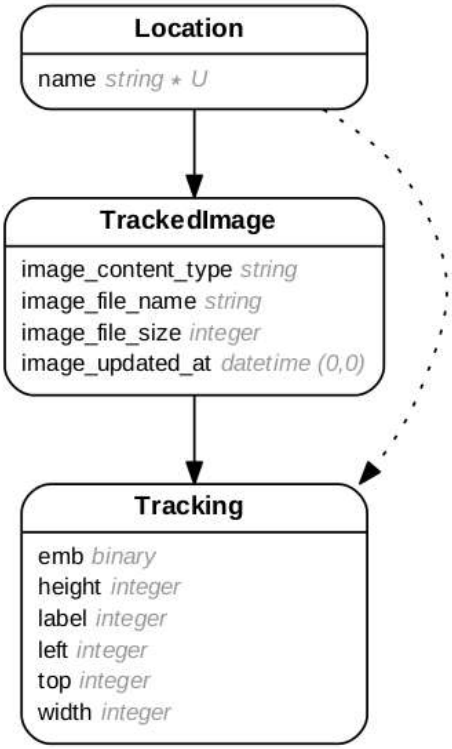

Figure 13: FaceRecogServer Domain Model

7.3 SERVER ARCHITECTURE

The server-side architecture consists of stateful and stateless components. This is to ensure a certain degree of scalability when more camera clients are used.

The application can be deployed in various ways depending on the scalability needs. We outline some of them in the section 8.1 Deployment. This is an in-depth description of the production level setup we used to deploy our application for multi-client testing.

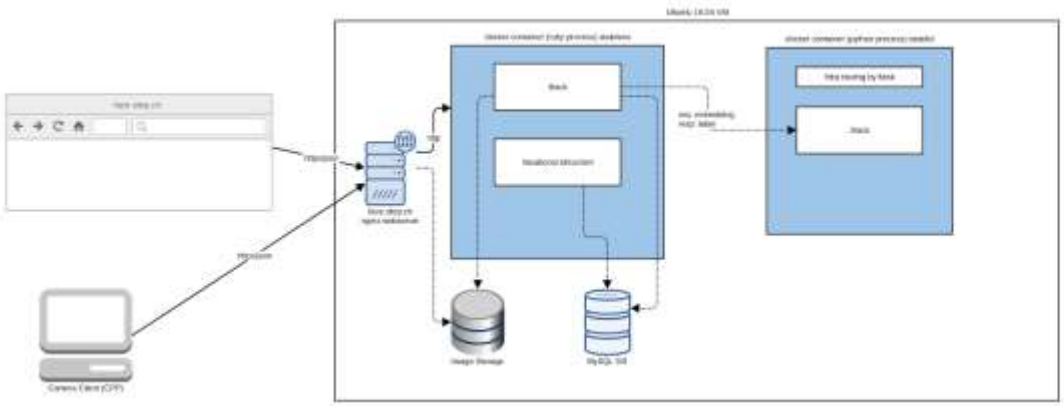

Figure 14: Deployment of the FaceRecogServer as we used during the project

All http requests are first handled by an nginx web server that terminates the SSL connection and handles compression and caching headers. It also serves the image files directly from the filesystem without going through a docker container or ruby process.

It is configured to load balance requests between all application instances each running in their own docker container. It is noteworthy that this can be omitted when testing locally with a single Rails instance.

The Ruby on Rails Deployment is handled by Dokku[12], which uses Heroku[13] images and buildpacks to build the application on each push to this git remote. The Rails Application handles both API requests from the camera clients and plain HTML Rendering for the Web GUI. It is completely stateless and thus can be freely scaled horizontally.

Images are stored to the Docker host file system by mounting it to each container. This allows the Rails application to store the images in a way that lets the load-balancer serve them directly.

All persistent data (locations, trackings, embeddings) is stored in a central MySQL database shared by all Rails instances.

The python tracking service is a separate single-instance docker container that holds the learned clustering in a Birch model[14] and allows access to it through a JSON-API via Flask.

7.4 Client Architecture

The client is written in C++. It uses OpenCV for accessing the video stream from the webcam, converting the raw image into JPG and scaling the image. The client uses Dlib[15] for face detection. The user interface and api client are written with Qt[16].

The client uses Dlib for face detection. The face detection from OpenCV is faster, but also delivers more false positives. In our use case we want as few false positives as possible. Therefore, we decided to use Dlib. Dlib uses the FHOG feature extraction algorithm (P. Felzenszwalb, 2010). Measurements showed us that the face detection needs the most computing time. After scaling the image to 75% of its size, we improved to 60ms from 75ms for each frame.

To achieve the best performance, we used OpenGL 3 to draw the video feed to screen. It gave us a lot flexibility, but maintaining and writing the code became complicated. We decided to use the high-level Qt 5 2D drawing library, this required to transform every frame from BGR to RGBA a format Qt 5 uses for its images.

Once detected each face is cropped and transformed to a 32px x 32px image with 80px padding. This 160px x 160px image is then compressed using JPEG and sent to the embedding service. The Embedding Service takes a pre-aligned image of the face and returns a 128-element float vector (an embedding). This embedding is then used by the tracking service to classify the faces.

The client sends the whole image JPEG compressed, the face positions and the embeddings to the Tracking API.

[12] Doku a Plattform as a Service based on Docker: http://dokku.viewdocs.io/dokku/
[13] Heroku a cloud plattform: https://www.heroku.com/
[14] Birch model: http://scikit-learn.org/stable/modules/generated/sklearn.cluster.Birch.html
[15] Dlib a machine learning C++ library: http://dlib.net/
[16] Qt a cross-platform application framework: https://www1.qt.io/company/

7.4.1 The main loop

Figure 15 shows a high level overview of the Face Recog Client's main loop and its services. Every two seconds the embedded web browser refreshes its view. The view contains the last recognized person from the webcam footage.

The actual main loop needs around 60ms for a loop. First a frame is fetched from the webcam and passed to the face detector. The face detector scans for faces and draw its findings on the webcam footage to the display. The API Client makes two asynchronous HTTP REST calls. The first one goes to the embedding service and the second goes to the tracking service. The API Client ignores all images and detected faces those two requests are fully processed. We decided on this behavior to avoid excessive network load.

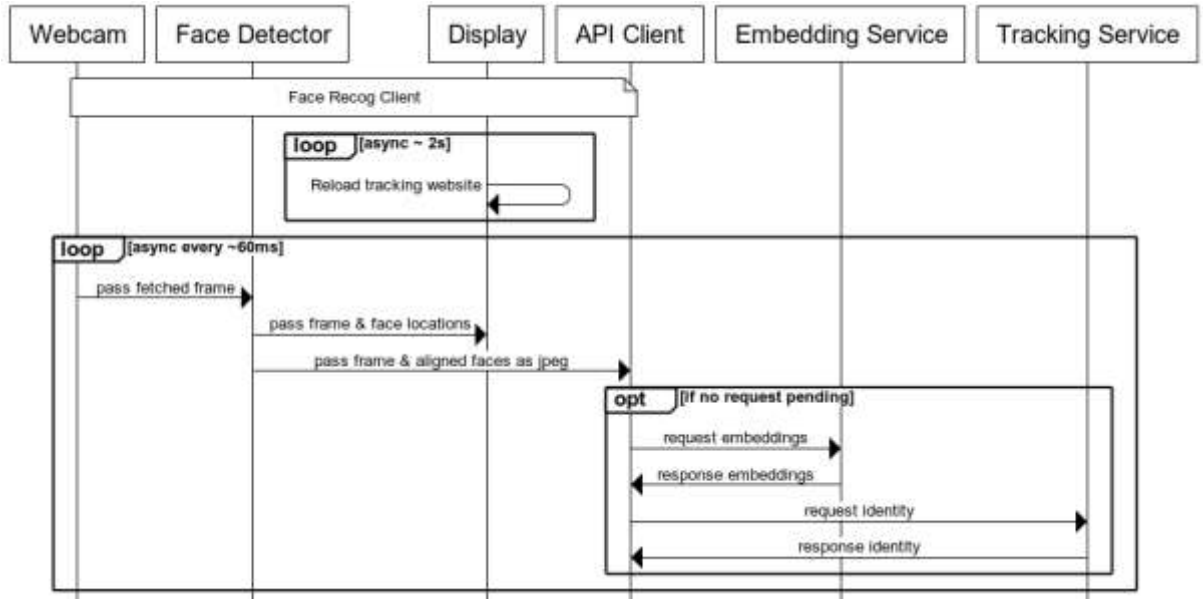

Figure 15: FaceRecogClient conceptual main loop sequence

8 RESULTS

Our prototype detects multiple faces from a webcam and shows earlier footage of the same person across all locations that are connected to the server. After the application starts no user interaction is needed. Although the Face Recog Client was programmed without any platform specific API, it was only tested on an Ubuntu 17.04 and 16.10.

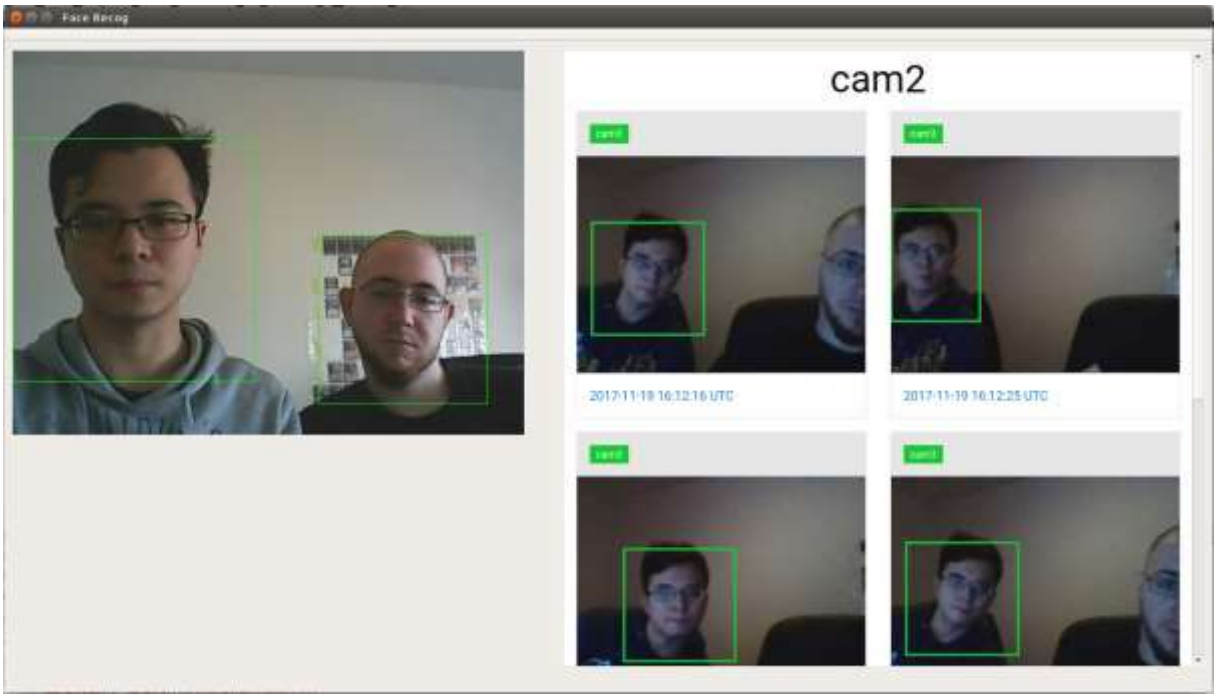

Figure 16: Snapshot from the FaceRecogClient

8.1 Deployment

Our application can be deployed in various ways. We suggest two particular setups. Each of our components are documented alongside the source code.

8.1.1 Development & Test Setup

All services can be deployed on one machine. This setup is recommended for development or for trying out.

- 1 Ubuntu Machine:
  - FaceRecogClient
  - Embedding service
  - Tracking service
  - Rails frontend

8.1.2 Recommended Setup

Our use case was designed with this setup in mind.

- 1 to N Ubuntu Client:
  - FaceRecogClient
  - Embedding service
- 1 Linux Server:
  - Tracking service
  - Rails frontend

8.2 Scaling Possibilities

Each component could be deployed on a separate machine. We aimed to design our showcase in a way that would be easy to scale.

9 CONCLUSION

The goal of this thesis was to show what modern day face recognition is capable of. We wanted to prove with our prototype that it is possible to build an impressive application only using publicly available resources.

We are confident that we have achieved this goal. Our prototype is entirely built on open-source software. It is sufficiently accurate and can be scaled to many different cameras. It consists of services connected through APIs that can be replaced or improved individually. The implementation of these services is based on the results of our testing and reflects the optimal performance we were able to achieve.

There is, as usual, room for improvement or extensions. If we were to continue working on it there are some areas that could be improved. For the application to be deployable in a real-world setting there needs to be an authentication scheme in place. Not only should the Web-GUI require some sort of authentication, there should also be an authentication token passed from the camera clients to the API. The clustering service could potentially be rewritten to be less vulnerable to bad embeddings and to produce better results when the majority of samples belongs to a single class. The client could be extended to remember face location and send only requests, when new faces are detected by the camera.

10 BIBLIOGRAPHY

Farzin, F., Hou, C., & Norcia, A. M. (n.d.). *Piecing it together: Infant's neural responses to face and object structure.* Retrieved from Investigative Ophthalmology & Visual Science: http://jov.arvojournals.org/article.aspx?articleid=2121335

Jack, R. E., Garrod, O. G.B., & Schyns, P. G. (n.d.). *Dynamic facial expressions of emotion transmit an evolving hierarchy of signals over time.* Retrieved from University of Glasgow: http://eprints.gla.ac.uk/88928/

Russell, S., & Norvig, P. (2016). Artificial Intelligence: A Modern Approach.

Schroff, F., Kalenichenko, D., & Philbi, J. (n.d.). *FaceNet: A Unified Embedding for Face Recognition and Clustering.* Retrieved from Cornell University Library: http://arxiv.org/abs/1503.03832

Zhang, K., Zhang, Z., Li, Z., & Qiao, Y. (n.d.). *Joint Face Detection and Alignment using Multi-task Cascaded Convolutional Networks.* Retrieved from Cornell University Library: https://arxiv.org/abs/1604.02878

11 APPENDIX

11.1 REPORT: MICHAEL GERBER

Working on this project was a surprisingly positive experience for me. I feel lucky that I got to work in the field of artificial intelligence which has been one of my main interests for a couple of years now. Although working without a clear goal in the beginning was quite challenging, it has proven to be quite the boon by giving us a strong incentive to explore a wide array of different solutions and use-cases. I

like to learn by exploring and tinkering with technology and working on this thesis has given me ample opportunity to do so.

As a team Victor and I worked very well together. We have very similar expectations for good software design and are not shy in calling out flaws in the other's work. Our background is similar enough to communicate efficiently and work out architecture and API decisions quickly. Yet it is different enough for each of us to bring our own strengths to the table. An example of this is would be Victor's in-depth understanding of C++ or my previous experience with deep learning models.

11.2 Report: Victor Ruch

The field of deep learning and face recognition were both new for me. I was excited to work on a term project where we could research in these fields. The term project was very interesting and challenging. I am satisfied with the results. The current state of face recognition technologies has surprised me.

Michael and I have similar backgrounds and know each other for many years. We have done various projects together, which contributed to an efficient communication. I am glad Michael had previous experience in the field of deep learning. He took the time to explain many concepts to me and pointed out useful resources.

11.3 Appendix: API Specifications

The finished product features three notable APIs that will be outlined here.

Rails Server API

This is the API the camera client is sending its images to for permanent storage. It has a single endpoint to receive images on.

POST /track

Request

image	JPG image of one frame captured by the client
data	JSON Encoded Meta information for the image
data['positions']	Array of position objects
data['positions'][x]	Positional information about a face in the image. Contains **width** and **height** of the face and starting positions **top** and **left** relative to the sent image
data['embeddings']	Array containing embeddings for each face in the image. Order needs to match positions.
data['embeddings'][x]	The embedding calculated on the client for the face at position data['positions'][x]

data['location']	The location the image was taken at. Has to be a string configured to be unique across all clients.

Embedding Service API

This is the API the camera client is sending cropped images of faces to to receive embeddings. It has a single endpoint and is stateless.

POST /embed

Request

images[]	JPG images of faces that are exactly 160x160 pixels and have a margin of 32 pixels around the face.

Response

embedding[]	Array of embeddings for the sent images. Same order as images.

Tracking Service API

Service that holds the clustering model. Used by the Rails server to assign labels to newly received images.

POST /track

Request

embeddings[]	Array of embedding vectors to be added to model.
ids[]	Array of IDs corresponding to embeddings.

Response

labels[]	Array of labels to which the embeddings were assigned to.