**ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ**

**към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ – СОФИЯ**

**ДИПЛОМНА РАБОТА**

Тема: IoT система за отдалечен достъп, контрол и анализ на устройства

|  |  |
| --- | --- |
| Дипломант: | Научен ръководител: |
| *Огнян Барух* | *Атанас Атанасов* |

СОФИЯ

2 0 2 1

СЪДЪРЖАНИЕ

**УВОД ........................................................................................................................... 5**

**ГЛАВА I. МЕТОДИ И ТЕХНОЛОГИИ НА РЕАЛИЗАЦИЯ. ПРОУЧВАТЕЛНА ЧАСТ.**

1.1 Лицево засичане

1.2 Лицево верифициране

1.3 NumPy

1.4 Виртуални среди

1.5 MQTT

1.6 Дигитални близнаци

1.7 Raspberry Pi

1.8 Bosch IoT Suite

1.9 Bosch IoT Hub

1.10 Bosch IoT Things

1.11 Bosch IoT Edge Services

1.12 Умни устройства

1.13 ONVIF

1.14 Z-Wave

1.15 Flutter

1.16 Firebase

**ГЛАВА II. ИЗИСКВАНИЯ КЪМ ПРИЛОЖЕНИЕТО. СТРУКТУРА НА СИСТЕМАТА ОТ УСТРОЙСТВА.**

2.1 Изисквания към приложението

2.1.1 Изисквания към модела за лицево верифициране

2.1.2 Изисквания към системата от свързани устройства

2.1.3 Изисквания към мобилното приложение

2.2 Съображения за избор на програмни средства

2.2.1 Алгоритми за машинно самообучение

2.2.2 Устройства в системата

2.3 Основни алгоритми за лицево разпознаване

2.3.1 Алгоритъм за лицево засичане

2.3.2 Алгоритъм за лицево верифициране

2.4 Структура на Flutter мобилно приложение

**ГЛАВА III. ПРОГРАМНА РЕАЛИЗАЦИЯ НА СИСТЕМА ОТ СВЪРЗАНИ УСТРОЙСТВА.**

3.1 Файлово и функционално разпределение на приложението

3.2 Файлова структура на управляващия код за системата от свързани устройства

3.3 Програмна реализация на алгоритъма за лицево верифициране

3.3.1 Файлова структура

3.3.2 Създаване на изображенията за трениране

3.3.3 Помощни функции за лицево засичане

3.3.4 Помощни функции за лицево верифициране

3.3.5 Трениране на модела за лицево верифициране

УВОД

През последните години терминът „IoT“ (Internet of Things – Интернет на нещата) придобива огромна популярност, тъй като такива решения улесняват нашето ежедневие. Вече съществуват решения, които ни позволяват по-лесно да контролираме нашите домове, коли, градини и други. Интернет на нещата навлиза и в проекти, свързани със сигурността, измерването и предаването на данни, както и с идентификация на лице, глас и пръстов отпечатък.

Машинното самообучение също става основен фактор в множество софтуерни и хардуерни решения през последните години. Развитието на машините позволява да бъдат тренирани по-сложни алгоритми с цел постигане на по-точни резултати. Машинното самообучение навлиза в света на технологиите все повече и повече, което ни позволява да заменим човешките усилия с работа на машини. То е използвано както за лични проекти, така и за глобални решения с цел подобряване на услуги като градски транспорт, имейл, персонални асистенти, преводи и други.

Целта на настоящата дипломна работа е да бъде изградена система, която комбинира две от най-широко използваните технологии и която създава по-лесен и по-сигурен начин за влизане в нашия дом, както и начин да следим кой е влязъл или се опитва да влезне в нашия дом. Основните компоненти на дипломната работа са камера, която засича обекти, и алгоритъм за машинно самообучение, който проверява дали засечените обекти са хора и дали тези хора са в списъкa с позволени хора, за да отключи вратата и да ги пусне в техния дом.

**ГЛАВА I. МЕТОДИ И ТЕХНОЛОГИИ НА РЕАЛИЗАЦИЯ. ПРОУЧВАТЕЛНА ЧАСТ.**

**1.1 Лицево засичане**

**1.1.1 Определение**

Лицевото засичане е технология, използвана в множество решения за идентифициране на човешки лица в изображения или във видео връзки на живо. То произлиза от засичането на обекти, като в случая търсените обекти са човешки лица. Един алгоритъм за лицево засичане се тренира върху множество изображения на различни човешки лица – различни полове, различни раси, различни черти на лицето. При изпълняване на алгоритъма той обхожда пиксел по пиксел даденото изображение и сравнява пикселите със съществуващите снимки на лица, за да открие приликите между тях. В зависимост от приликите между потенциалното засечено лице и снимките на познатите лица, алгоритъмът „взима решение“ дали даденият обект е лице или не, като резултатът е процентът сигурност, че разглежданият обект е лице. Съществуват алгоритми, които се тренират по време на изпълнение – при засечено лице алгоритъмът го добавя към множеството от познати лица. По този начин всяко следващо разпознато лице довежда до по-точни резултати.

**1.1.2 Приложения**

Лицевото засичане намира множество приложения в различни сфери –системи за лицево разпознаване, автоматичен фокус във фотографията, разпознаване на емоции, както и четене по устни. С развиването на алгоритмите и моделите за лицево засичане, то набира все повече и повече популярност в различни решения.

**1.2 Лицево верифициране**

**1.2.1 Определение**

Лицевото верифициране използва резултатите от лицевото засичане, за да сравни непознато лице с познати такива. За разлика от лицевото разпознаване, което отговаря на въпроса: „Чие е това лице?“, лицевото верифициране отговаря на въпроса: „Това ли е правилното лице?“. Един алгоритъм за лицево верифициране сравнява характерните черти на непознатото и познатите лица – разстоянието между зениците на двете очи, разстоянието между външния и вътрешния ъгъл на всяко око, разстоянието между носа и устата и други. Тази информация се записва като вектор със 128 измерения, а впоследствие се сравняват данните от всички измерения, за да се верифицира непознатото лице.

**1.2.2 Приложения**

Лицевото верифициране намира широко приложение в системи, свързани със сигурността, като това могат да бъдат както лични системи, така и публични такива. Лицевото верифициране се използва за контрол на достъпа до различни помещения, както и за отключване на нашите смартфони.

**1.3 NumPy**

NumPy е най-използваната библиотека за работа с многоизмерни масиви, написана на Python. Библиотеката предлага множество функционалности, свързани с създаване и обработване на масиви с много измерения. Освен това NumPy има огромна документация, която много улеснява използването на функционалностите, които предлага. Библиотеката позволява запазването на такива масиви в специални файлове с разширение .npz, което представлява компресиран многоизмерен масив. Друга широко използвана функционалност е че масивът, използван в NumPy, е обект, който съдържа и друга лесно достъпна информация като брой на измеренията, големина на всяко измерение и други.

**1.4 Виртуални среди**

В много случаи при разработка на приложение на програмния език Python се изисква инсталиране на външни библиотеки. Различните приложения обаче може да се нуждаят от различни версии на някоя библиотека, което означава, че при някоя изтеглена версия някои приложения няма да функционират правилно. Решението на този проблем е да се създаде т.нар. виртуална среда, която съдържа в себе си необходимите библиотеки за съответното приложение. Най-често всяко приложение има отделна виртуална среда, за да се предотврати конфликт на версии. Към всяка виртуална среда върви автоматично генериран текстов документ, в който са описани имената и версиите на всички външни библиотеки. Това позволява лесното пренасяне на всяко приложение през публично хранилище – всеки, който иска да стартира такова приложение, трябва да създаде виртуална среда на машината си и да изтегли всички библиотеки, описани в текстовия документ, като за тази цел за изпълняват две команди в конзолата и целият процес отнема до 1 минута.

**1.5 MQTT**

**1.5.1 Определение**

MQTT представлява стандартен протокол за комуникация, който намира широко приложение в проектите, свързани с „Интернет на нещата“.

**1.5.2 Принцип на работа**

MQTT работи на принципа на публикуване и абониране за съобщения, което позволява едно съобщение да достига до множество крайни устройства. Всяко устройство може да се абонира за даден канал и да получава всички съобщения в този канал. Протоколът дефинира два типа устройства – един брокер и много клиенти. Брокерът е центърът на комуникация, като той отговаря за получаването на всяко съобщение и изпращането му до всеки абонат. Той контролира кои клиенти са свързани и кои клиенти към кои канали са се абонирали. Клиентите могат да бъдат много и те се свързват към даден брокер по IP. Те могат да се абонират към отделни канали и да получават съобщения, изпратени от други клиенти. В зависимост от приложението съобщенията могат да бъдат криптирани.

**1.5.3 Защо MQTT е толкова разпространен?**

Най-силните страни на MQTT, е че има много висока скорост и много ниска консумация. Една MQTT заявка се изпълнява над 10 пъти по-бързо от една HTTP заявка. Освен това този протокол е с малко, но все пак достатъчни възможности, за да се постигнат необходимите функционалности за работа с устройства. MQTT заявките са с много малък размер, което позволява на устройства с ограничени възможности да се възползват от услугите на този протокол. MQTT позволява едно съобщение да бъде получено от безброй много абонати, докато при HTTP ще трябва да се изпращат отделни съобщения до всеки получател. Има много примери за използването на този протокол, дори извън Интернет на нещата. Messenger на Facebook използва този протокол, което ни позволява бързо изпращане и получаване на съобщения, без да се използват много ресурси.

**1.6 Дигитални близнаци**

Дигитален близнак на едно устройство представлява съвкупността от информация, описваща съществуващо устройство. Терминът добива популярност в началото на XIX век с навлизането в ерата на „Интернет на нещата“. Дигиталният близнак може да се разглежда като база от данни за едно устройства, съдържаща само текущата информация. Ако устройството има връзка със своя дигитален близнак, то тогава то може да сигнализира за промяна в неговата информация, която да се отбележи в дигиталния близнак. Съществуването на дигитални репрезентации позволява на хората да следят от разстояние информацията и работата на всяко устройство.

**1.7 Raspberry Pi**

Raspberry Pi е един от най-популярните микрокомпютри на пазара. То представлява по-немощен, но и много по-малък и по-евтин компютър. Най-често тези устройства се използват в решения, свързани с Интернет на нещата, тъй като повечето имат вграден Wi-Fi модул и Bluetooth модул, което им позволява да обменят информация с други устройства и да се създаде система от свързани устройства на сравнително ниска цена. Тези микрокомпютри нямат много памет, нито много добър процесор, но са способни да извършват предостатъчно операции, свързани с комуникация и обработка на данни от устройства. Някои от по-новите модели (Raspberry Pi 3 и Raspberry Pi 4) имат възможност да изпълнят и някои сравнително леки модели за машинно самообучение. Всяко Raspberry Pi използва Linux базирана операционна система, предоставяща на потребителите конзола, чрез която да управляват своя компютър. Свързването на Raspberry Pi към компютър или лаптоп отнема много малко време и програмирането върху него е сравнително лесно, което го прави много приятно за по-начинаещи, а освен това има множество форуми и видео уроци, свързани с използването на този микрокомпютър.

**1.8 Bosch IoT Suite**

Bosch IoT Suite представлява платформа за свързване на устройства, тяхното контролиране, както и обработването на данните, които изкарва всяко устройство. Услугите в Bosch IoT Suite са разделени спрямо функционалността си – свързване на устройства, дигитална репрезентация на устройства, управление на устройства и подновяване на софтуера върху устройства. Към днешна дата над 15 милиона устройства са свързани към Bosch IoT Suite. Също така над 250 международни проекта се възползват от услугите, предлагани от тази платформа.

**1.9 Bosch IoT Hub**

Bosch IoT Hub представлява облачна услуга, която свързва устройства с приложения, които ги управляват. Чрез тази услуга разработчиците на софтуер получават бърз и лесен начин да свържат най-различни устройства към своите приложения. Тя поддържа няколко протокола за комуникация, най-известните от които са HTTP, MQTT и AMQP. Свързването на устройства чрез Bosch IoT Hub е лесно и най-вече сигурно, тъй като се разчита на информация, достъпна само за притежателя на устройствата.

**1.10 Bosch IoT Things**

Bosch IoT Things представлява облачна услуга, която позволява на всеки потребител да съхранява и обработва дигителните близнаци на устройствата си. Това може да се случва през сайта на Bosch IoT Things, през HTTP API или посредством код с HTTP заявки и MQTT съобщения. Функционалностите на услугата са регистриране и обработване на дигитални близнаци, контролиране на достъпа до тях, както и търсене на дигитални близнаци по динамични стойности. Bosch IoT Things е част от Bosch IoT Hub, което позволява на всички услуги в рамките на Bosch IoT Suite да обменят информация с цел подобряване на качеството на услугата и улесняването на процеса на контрол на дигиталните близнаци за потребителите.

**1.11 Bosch IoT Edge Services**

Bosch IoT Edge Services представлява връзката между устройствата и тяхната дигитална репрезентация в Bosch IoT Suite. Освен връзката, Bosch IoT Edge Services осъществява и комуникацията между едно устройство и Bosch IoT Suite посредством протокола MQTT. Те позволяват на устройства, работещи върху всякакви протоколи, да се свържат със своя дигитален близнак. По този начин всяко устройство, което работи с Bluetooth, Z-Wave, ONVIF, ZigBee и други може по много бърз и лесен начин да се свърже и управлява чрез Bosch IoT Suite.

**1.12 Умни устройства**

През последните години всички устройства, използвани в ежедневието, биват заменяни от умни устройства. Умни устройства са устройства или сензори, които са свързани към други устройства или към други машини и които могат да се използват интерактивно. Съществуват множество протоколи за комуникация между отделни устройства, като най-известните са Bluetooth, Wi-Fi, Z-Wave, ZigBee, ONVIF и други. Такива устройства биват смартфони, умни автомобили, умни часовници, умни камера, умни хладилници, умни ключалки и други. Появата на умните устройства улеснява много човешкото ежедневие и тяхната употреба става необходима във все повече индустрии.

**1.13 Z-Wave**

Z-Wave представлява протокол за безжична комуникация, като се използва предимно в решения, свързани с „Интернет на нещата“. Най-често протоколът се използва, за да се осъществи комуникацията с устройства в дома като автоматични ключалки, автоматични прозорци, автоматични врати за гаражи, термостати и други. Вълните достигат до 100 метра, което позволява пълен контрол в дома и градината. Всяко устройство, поддържащо Z-Wave, може да бъде контролирано през Wi-Fi, което улеснява работата с тези устройства.

**1.14 ONVIF**

ONVIF е глобална организация, която цели да стандартизира работата с базираните на IP устройства, свързани със сигурността. Създаденият стандарт описва връзката и комуникацията между различни устройства за видеонаблюдение и централни машини, които следят тяхната работа и потока, който записват. Друга основна цел на организацията е да се осигури съвместимостта между продукти на различни компании, за да се постигне по-сигурна и стандартна система.

**1.15 Flutter**

Flutter е пакет за разработка на софтуер, създаден от Google. Първата версия излиза 2017 г. и Flutter за кратко време става една от най-използваните платформи за създаване на мобилни приложения. Във Flutter се използва обектно-ориентираният езикът за програмиране Dart. Основната единица в едно Flutter приложение е т.нар „widget“, което представлява клас за даден компонент в приложението (бутон, параграф, изображение), който има отделни параметри и функционалности. Едно от най-големите предимства на Flutter е че позволява компилиране за iOS и Android, което означава, че може да се създаде едно приложение на един език за двете операционни системи вместо две отделни приложения на два отделни езика за програмиране. Освен това съществуват множество курсове и форуми за Flutter, което помага на по-начинаещи да направят своите първи стъпки в разработката на мобилни приложения

**1.16 Firebase**

Firebase е платформа, разработена от Google за създаване на мобилни и уеб приложения. Платформата съдържа 18 продукта, които улесняват разработката на приложения. Всеки продукт има отделна документация, в която е описан процесът на инсталиране и функционалностите, които предлага. Най-често процесът на инсталиране изисква няколко реда код, което помага на разработчиците да се фокусират върху функционалностите на приложението, което създават.

**1.16.1 Firebase Authentication**

В почти всички приложения се изисква имплементирането на някакъв вид автентикация. Firebase Authentication позволява да се вградят над 10 вида автентикация в приложението. Firebase позволява изграждането на база от данни, където се пази потребителските имена и пароли за използване от Firebase Authentication. Този продукт поддържа сесии, което позволява по всяко време да се знае дали някой потребител е влезнал в своя профил.

**ГЛАВА II. ИЗИСКВАНИЯ КЪМ ПРИЛОЖЕНИЕТО. СТРУКТУРА НА СИСТЕМАТА ОТ УСТРОЙСТВА.**

**2.1 Изисквания към приложението**

**2.1.1 Изисквания към модела за лицево верифициране**

• Моделът за лицево верифициране да включва програма, която да позволи на потребителя да отвори камерата на машината си и да снима изображения на потребителя, с които да се тренира алгоритъма.

• Моделът за лицево верифициране да включва модел за лицево засичане, който да засича всички лица на хора в едно изображение и да ги запазва в масив.

• Процесът за лицево верифициране да прекъсва изпълнението си, ако няма засечени лица в изображението.

• Моделът за лицево верифициране да включва програма, която да тренира алгоритъма, използвайки заснетите снимки за познати лица.

• Моделът за лицево верифициране да включва програма, която да приема изображение чрез линк или чрез файл с възможни разширения *.jpg*, *.png* и *.jpeg*. Програмата трябва да засече всички лица в изображението и да провери кои лица са на познати хора. Алгоритъмът трябва да връща масив с познатите засечени лица или „Unknown“, ако сред засечените лица няма нито едно познато.

**2.1.2 Изисквания към системата от свързани устройства**

• Системата да включва Raspberry Pi, което да има инсталирани Bosch IoT Edge Services, за да може да се свържат устройствата в система към това Raspberry Pi.

• Системата да включва камера, която да е свързана към Raspberry Pi посредством Bosch IoT Edge Services. Тази камера трябва да засича движение, да прави снимка и да изпраща линк на снимката към Raspberry Pi, за да се верифицират човешките лица, ако има такива.

• Системата да включва ключалка, която да е свързана към Raspberry Pi посредством Bosch IoT Edge Services. Ключалката да се отключва за 5 секунди, ако моделът за лицево верифициране разпознае лице. Съобщенията към ключалката да се изпращат и получават посредством Z-Wave контролер, свързан към Raspberry Pi.

**2.1.3. Изисквания към мобилното приложение**

• Мобилното приложение да може да изпраща HTTP заявки към Bosch IoT Suite с цел достъп, четене и редактиране на дигиталните близнаци на устройствата.

**2.2 Съображения за избор на програмни средства**

**2.2.1 Алгоритми за машинно самообучение**

За реализация на алгоритмите за лицево засичане и лицево верифициране се използва скриптовият език Python. Има много причини, защо Python е най-използваният език за реализиране на алгоритми за машинно самообучение. Този език е сред първите 10 на най-използваните в програмирането. Освен това Python е скриптов език и за изпълнението на програма, написана на Python, не се изискват допълнителни процеси като компилация, както при компилируеми езици като C и Java. Друго предимство на Python е че може да се поддържа от всякакви операционни системи, дори и на микроконтролери. Освен това съществуват множество библиотеки за машинно самообучение, написани на Python, което позволява на разработчиците на код да създават всякакви приложение, използвайки Python и съществуващите библиотеки.

**2.2.2 Устройства в системата**

Системата от свързани устройства включва камера и ключалка, свързани към Raspberry Pi посредством Bosch IoT Edge Services. Използва се този микроконтролер, тъй като той разполага с достатъчно добър процесор, за да поддържа връзките към устройствата, връзката към облачната услуга Bosch IoT Hub и да изпълнява тренирания алгоритъм за лицево верифициране.

Камерата, използвана за реализацията на дипломната работа, е Bosch FLEXIDOME IP 4000i. Камерата използва ONVIF за комуникация, което позволява лесното й свързване и контролиране от друга машина. Тя съдържа сензор за движение и сигнализира, когато има засечен обект от сензора. Камерата може да изпраща линк към изображение и линк към видео поток, които могат да се достъпят с автентикация с потребителско име и парола.

Ключалката, използвана за реализацията на дипломната работа, е Danalock V3 Smartlock. Тя може да бъде свързана към смартфон или към смарт часовник и да се контролира от съответното устройство. Ключалката използва протокола Z-Wave за комуникация. Този протокол позволява лесното ѝ свързване към Bosch IoT Suite, както и лесното задаване на команди към нея. Освен това тя може да бъде контролирана през Wi-Fi, а създателите на тази ключалка са разработили и мобилно приложение, което може да се свърже с нея и да я контролира.

**2.3 Основни алгоритми за лицево засичане и лицево верифициране**

**2.3.1 Алгоритми за лицево засичане**

Съществуват много алгоритми, създадени и оптимизирани специално за целта да засичат лица в снимки. Според много източници MTCNN е един от най-добрите модели откъм ефективност и постигнати резултати. Този модел се състои от 3 конволюционни невронни мрежи, като освен лица в снимка, те могат и да открият координатите на характеристични точки на лице на човек – двете очи, носа и двата края на устата. В контекста на това приложение няма нужда от координатите на характеристични точки на лицето, затова най-подходящият е Cascade Classifier, който е изграден по-подобен начин, но се изпълнява с по-голямо бързодействие, тъй като засича само лице в изображение, а не и характеристични точки в лицето. За всяко лице моделът връща като резултат масив с 4 стойности – първите две са координатите на горния ляв ъгъл на лицето, а последните 2 са дължината и широчината на лицето, като по този начин се обрисува правоъгълник около лицето, който може се изрязва от оригиналната снимка, за да се използва по-лесно и ефективно от модела за лицево верифициране

**2.3.2 Алгоритми за лицево верифициране**

Както при лицевото засичане, така и при лицевото верифициране невронните мрежи водят до най-точни резултати. Един от най-известните модели за лицево верифициране е VGG-Face, разработен от специалисти от Оксфордски университет. Алгоритъмът за лицево верифициране приема изображение, съдържащо лице и създава т.нар „*face embedding*“, което представлява вектор в 128 измерения, като всяко измерения представлява различна част от този *face embedding*. Този вектор пази информация за много разстояния между характеристичните части на човешкото лице – разстоянието между зениците на двете очи, разстоянието между вътрешния и външния ъгъл на всяко око и други.

При трениране на алгоритъма за лицево верифициране се използва метода „*triplet loss*“. За неговата реализация се използват три изображения на човешки лица – базова снимка, вярна снимка и грешна снимка. Базовата снимка е снимката, върху която се тренира, вярната снимка е на същото лице, а грешната снимка е на друго лице. По този начин алгоритъмът се „учи“ как да разграничава отделните лица по стойностите от различните измерения на вектора *face embedding*. Функцията на *triplet loss* изглежда така:

*Фиг. 2.1: Формула за функцията на triplet loss*

В тази формула е базовата снимка, е вярната снимка, а е грешната. Функцията представлява функцията за извличане на *face embedding* от изображение на лице, a е допустимата граница между вярната и грешната снимка. Тази граница се въвежда, за да има по-голямо разграничение между отделни лица и да може да се разграничават хора, които си приличат в лицето. За трениране на алгоритъма за лицево верифициране се използва следната формула:

*Фиг 2.2: Функция за трениране на модела за лицево верифициране*

Резултатът от тази формула е сумата от резултатите на функцията с всички снимки от колекцията от снимки за трениране. Този резултат се запазва във файл с разширениe *.npz* и впоследствие се използва при изпълнението на алгоритъма върху тестови лица.

**2.4 Структура на Flutter мобилно приложение**

Във *Фиг. 2.3* е показана примерна файлова структура на едно Flutter приложение, което е разположено в папката „flutter-application“.

flutter-application /

├── android/

│ ├── app/

│ ├── build.gradle

│ └── gradle/

├── build/

├── fonts/

├── ios/

│ ├── Flutter/

│ ├── Podfile

│ ├── Pods/

│ └── Runner/

├── lib/

│ ├── main.dart

│ ├── models/

│ ├── screens/

│ └── services/

├── pubspec.yaml

└── README.md

*Фиг. 2.3: Примерна файлова структура на Flutter приложение*

В папката „android” е поместено приложението за устройства с Android операционна система. В подпапката „app“ се намира функционалния код на мобилното приложение, а в подпапката „gradle“ се намира кода за сглобяване на приложението. Файлът „build.gradle“ съдържа в себе си константи, които се използват при сглобяването на приложението.

В папката „build“ са поместени сглобените приложения за Android OS и iOS, както и сорс код на изтеглените външни библиотеки.

В папката „fonts“ се поставят семействата от шрифтове, използвани в приложението.

В папката „ios“ се намира приложението за устройства с операционната система iOS. В подпапката „Flutter“ е поместена логиката за сглобяване на мобилното приложение. В подпапката „Pods“ се намират необходимите външни библиотеки за сглобяване на приложението, а в подпапката „Runner” се пазят конфигурационни файлове за стартирането на приложението върху реално устройство. Във файлът „Podfile“ се пазят версиите и имената на необходимите външни библиотеки, които при сглобяване се изтеглят и се поставят в папката „Pods”.

В папката „lib“ е поместен основния функционален код на приложението, написан на програмния език Dart. Там се намира кодът, написан от разработчикът на едно мобилно приложение. При сглобяване на приложението кодът, написан на Dart, се „превежда“ на Swift (програмният език за разработване на приложения за iOS) и Kotlin (програмният език за разработване на приложения за Android OS), за да могат да се подготвят за стартиране приложенията за двете операционни системи. Подпапката „models“ съдържа различни помощни класове, които се използват в приложението. В подпапката „screens“ са поместени класовете за различните екрани в мобилното приложение. В подпапката „services“ се намират различни помощни класове, свързани с функционалностите на приложението. Файлът „main.dart“ е файлът, който съдържа началната страница на приложението и който се изпълнява при неговото стартиране.

**ГЛАВА III. ПРОГРАМНА РЕАЛИЗАЦИЯ НА СИСТЕМА ОТ СВЪРЗАНИ УСТРОЙСТВА.**

**3.1 Файлово и функционално разпределение на приложението**

Приложението е съставено от няколко отделни модула, както и от няколко конфигурационни файла, които се изпълняват при стартиране на програмата. Основната файлова структура на сорс кода е изобразена във *Фиг. 3.1*.

src/

├── clean.sh

├── install.sh

├── iot\_system/

├── mobile/

├── pc.txt

├── requirements.txt

*Фиг 3.1: Файлова структура на кода на проекта*

Директориите „iot\_system“ и „mobile“ са двата модула на проекта – кода, управляващ схемата от свързани устройства и кода за мобилното приложение. Файлът „requirements.txt“ съдържа необходимите външни библиотеки за изпълняване на приложението. В него на отделен ред са дефинирани имената на всички библиотеки, както и техните версии, които да се изтеглят.

За реализацията на дипломната работа е използван микроконтролерът Raspberry Pi, като върху него се изпълнява алгоритъмът за лицево верифициране, то е свързано към устройствата и към облачните услуги на Bosch. Тъй като Tensorflow версия 2 не е излязла официално за Raspbian OS, процесът за инсталиране на Tensorflow за Raspberry Pi е по-различен от този за Windows или Linux операционни системи. За инсталиране на необходимите външни библиотеки се изпълнява скрипта „install.sh“, изобразен във *Фиг. 3.2*, който инсталира библиотеките в зависимост от операционната система, върху която работи потребителя.

#!/bin/bash  
if [[ $OSTYPE == "linux-gnueabihf" ]]  
then  
 python3 -m pip install --upgrade pip  
 python3 -m pip install virtualenv  
 python3 -m virtualenv env  
 python3 -m pip install -r requirements.txt  
 python3 -m pip install https://github.com/bitsy-ai/tensorflow-arm-bin/releases/download/v2.4.0-rc2/tensorflow-2.4.0rc2-cp37-none-linux\_armv7l.whl  
else  
 python3 -m pip install --upgrade pip  
 python3 -m pip install pipenv  
 pipenv install -r pc.txt  
fi

*Фиг 3.2: Скриптът install.sh*

За намиране на операционната система на машината се използва предефинирана константа „OSTYPE“. Независимо от операционната система се инсталира последната версия на PIP (Package Installer for Python), който е най-широко разпространеният инсталатор на пакети за Python. Tensorflow 2 се нуждае от минимална версия 20.0 на PIP, затова се инсталира последната версия, която към момента на разработване на тази дипломна работа е 21.0.1. Скриптът инсталира и виртуална среда, в която ще инсталира всички библиотеки, необходими за изпълнението на приложението. При Raspbian OS библиотеките се инсталират от „requirements.txt“, а за всички останали операционни системи се инсталират от „pc.txt“, който инсталира всичко от „requirements.txt“ плюс Tensorflow 2 за Windows или Linux OS. В „install.sh“ се инсталира отделно Tensorflow 2 за Raspbian OS.

За удобство на потребителя е създаден скриптът „clean.sh“, изобразен във *Фиг. 3.3*, който премахва виртуалната среда, създадена за проекта и изтрива от машината на потребителя всички инсталирани библиотеки за стартирането на приложението. В този скрипт отново се проверява вида на операционната система, тъй като процесът по деинсталиране на библиотеките се различава.

#!/bin/bash  
if [[ $OSTYPE == "linux-gnueabihf" ]]  
then  
 rm -r env  
else  
 pipenv --rm  
fi

*Фиг. 3.3 Скриптът clean.sh*

Тъй като всички библиотеки, от които приложението зависи, са инсталирани във виртуална среда, то останалите приложения на потребителя няма да бъдат засегнати по никакъв начин.

**3.2 Файлова структура на управляващия код за системата от свързани устройства**

Основната директория, в която се съхранява функционалния код, управляващ системата от свързани устройства, е с наименование „iot\_system“ (*Фиг. 3.4*). В приложението отделните компоненти са разделени в отделни директории, като има една за алгоритъма за лицево верифициране („face\_verification“), една за управляващия код за камерата („camera\_detection“) и една за управляващия код за ключалката („door\_lock\_controller“).

iot\_system/

├── camera\_detection/

├── door\_lock\_controller/

├── face\_verification/

├── models/

├── \_\_init\_\_.py

├── main.py

*Фиг. 3.4: Файлова структура на кода за системата от устройства*

Подпапката „models“ съдържа тренираните модели за лицево засичане и лицево верифициране. Файлът „\_\_init\_\_.py“ превръща директорията „iot\_system“ в Python пакет, което позволява на всеки скрипт да използва функциите, дефинирани в останалите Python програми. Скриптът „main.py“ е основният скрипт, който се изпълнява при стартиране на системата от свързани устройства. Този файл е нужен, за да може пакетът „iot\_system“ да бъде най-високото ниво при изпълнението на приложението, което позволява достъп до всички Python програми, разположени в директорията „iot\_system“ и във всяка нейна поддиректория.

**3.3 Програмна реализация на алгоритъма за лицево верифициране**

**3.3.1 Файлова структура**

В директорията „face\_verification“ Python кодът е разпределен в отделни файлове според функционалностите, които изпълнява. Файловата структура на директорията е изобразена във *Фиг. 3.5*:

face\_verification/

├── candidate\_face\_check.py

├── dataset/

├── dataset\_loader.py

├── face\_embedding\_utils.py

├── face\_extract\_utils.py

├── recognize.py

├── train.py

*Фиг. 3.5: Файлова структура на кода на алгоритъма за лицево верифициране*

**3.3.2 Създаване на изображенията за трениране**

Първата стъпка за създаване на модел за лицево верифициране е да се създаде набор от снимки, върху които да се тренира модела. Моделът трябва да бъде добре запознат с лицата на хората, на които ще отключва вратата, затова се нуждае от немалък брой изображения. В контекста на тази дипломна работа познатите лица са не повече от 5 и моделът се тренира върху 30 снимки за всяко лице. Всички изображения се съхраняват в директорията „dataset“, като тя е разделена на подпапки, които са имената на хората. При качване на кода в хранилище, се качва празна папката „dataset“, така че всеки, който иска да използва приложението да може да създаде колекция от свои снимки, без да се занимава да трие чужди такива. Python програмата, отговорна за създаване на колекцията от снимки е с името „dataset\_loader.py“. При изпълнение на програмата се подават двете имена на човека, който ще си прави снимки, така че неговите снимки да бъдат запазени в папка с неговото име, като има 2 начина, по които могат да се подадат аргументите, които са показани във *Фиг. 3.6*.

$ python3 dataset\_loader.py -f Ognian -l Baruh  
$ python3 dataset\_loader.py --fname Ognian --lname Baruh

*Фиг. 3.6: Примерни комадни за стартиране на*

*програмата за създаване на изображения*

По този начин ако моделът за лицево верифициране разпознае този човек, той знае неговото име и информира притежателят на системата кой влиза в къщата.

При изпълнение на програмата с грешни аргументи или без такива се изкарва грешка, която показва как е правилно да се стартира. Тази грешка е изобразена във *Фиг. 3.7*.

$ python3 dataset\_loader.py

Wrong usage!

Correct usage:

$ python3 dataset\_loader.py -f <FirstName> -l <LastName>

or

$ python3 dataset\_loader.py --fname <FirstName> --lname <LastName>

*Фиг. 3.7: Съобщение за грешка при изпълнение на*

*програмата за създаване на изображения*

При изпълнение на програмата с „-h“ или „--help“ се изкарва наръчник за стартиране на програмата, който е изобразен във *Фиг. 3.8*.

$ py dataset\_loader.py -h

-h, --help -> shows this screen

-f, --fname -> first name of the person

-l, --lname -> last name of the person

Note: There can't be two people with the same full name!

*Фиг. 3.8: Наръчник за изпълнение на програмата за създаване на изображения*

При нормално изпълнение на програмата процесът за създаване на изображение за трениране е показана на *Фиг. 3.9*.



*Фиг. 3.9: Блок схема на програмата за създаване на изображения*

Първата функция, която се изпълнява при стартиране на програмата е функцията „get\_name“, показана във *Фиг. 3.10*, която има за цел да обработи подадените аргументи от потребителя и да генерира пълното име, събирайки първото и последното име на потребителя. Обработката на аргументите се осъществява чрез библиотеката „getopt“, която позволява лесно използване на подадените аргументи.

def get\_name(argv):  
 fname = ''  
 lname = ''  
  
 try:  
 # Взимане на аргументите с фунцкията „getopt“  
 except GetoptError:  
 # Принтиране на съобщение за грешка

# Обходжане на всички аргументи  
 for opt, arg in opts:  
 if opt in ('-f', '--fname'):  
 fname = arg  
 elif opt in ('-l', '--lname'):  
 lname = arg  
 elif opt in ('-h', '--help'):  
 # Приниране на наръчника  
   
 if not fname and not lname:  
 # Приниране на съобщение за грешка

# Връщане на резултат  
 name = fname + ' ' + lname  
 return name

*Фиг. 3.10: Скелет на функцията „get\_name“*

Втората функция има за цел да създаде директория, в която ще се пазят снимките на потребителя, като името на тази директория е името на потребителя, което той е подал при стартиране на програмата. Създаването на директорията става посредством вградената библиотека „os“. Наименованието на тази функция е „create\_dir“ и тя е изобразена във *Фиг. 3.11*.

def create\_dir(name):  
 full\_path = PATH + name  
 mkdir(full\_path)  
 return full\_path

*Фиг. 3.11: Функцията „create\_dir“*

Последната функция в програмата е с най-голяма функционалност, като тя е отговорна за снимането на потребителя. За тази цел се използва библиотеката OpenCV, която позволява на да се отвори отделен прозорец, в който да се покаже камерата на устройството на потребителя (ако потребителят използва настолен компютър, то ще трябва да свърже камера или да превключи на лаптоп). Снимки се правят с натискане на бутона „SPACE“, а изпълнението на програмата се прекратява с натискането на бутона „ESCAPE“. При приключване на програмата се затваря прозореца с камерата и всички направени снимки се запазват. Скелетът на тази функция е изобразен във *Фиг. 3.12*.

def take\_screenshots(full\_path):  
 # Инициализиране на прозореца за камерата  
  
 while True:

# Показване на прозореца за камерата  
  
 # Изчакване за натискане на SPACE или ESCAPE  
  
 # SPACE  
 if key % 256 == 32:  
 # Запазване на снимка

# ESCAPE  
 elif key % 256 == 27:  
 # Прекратяване

*Фиг. 3.12: Скелет на функцията „take\_screenshots“*

**3.3.3 Помощни функции за лицево засичане**

При изпълнението на алгоритъм за лицево верифициране или лицево разпознаване първата стъпка е да се отделят всички лица от оригиналната снимка. Този процес включва изваждане на лицето от снимката и изправянето му, така че лицето да не е под определен ъгъл. Резултатът от алгоритъм за лицево засичане са координатите на правоъгълник, в който се намира лицето. Посредством тези координати, правоъгълникът може да бъде отрязан от оригиналната снимка и да бъде запазен в нов файл или да бъде използван по време на изпълнението на програмата. За реализацията на проекта е създаден отделен файл, съдържащ всички функционалности, свързани с алгоритъма за лицево засичане, чието име е „face\_extract\_utils.py“. В жизнения цикъл на приложението този скрипт не се изпълнява директно, а функции му се използват в други програми като помощни функции. „face\_extract\_utils.py“ съдържа 5 помощни функции, като те се използват както в тренирането на алгоритъма за лицево верифициране, така и в неговото изпълнение.

Първата помощна функция във файла е „get\_pixels“, като тя служи за отваряне на изображението и запазването на неговите пиксели в триизмерен масив. Тя приема като аргумент името на файла и връща като резултат пикселите на съответното изображение. Въпреки че всички изображения, които камерата заснема, са цветни, функцията конвертира всяка черно-бяла снимка в цветна. Това е необходимо, тъй като всеки модел за машинно самообучение има предварително дефинирани размери на входните данни. Моделът за лицево верифициране очаква изображения с дължина 160 пиксела, широчина 160 пиксела и с цветови модел RGB (Red – червено, Green – зелено, Blue – синьо). В повечето случаи черно-белите снимки имат само една стойност за цвят от 0 до 255, тъй като в цветовия модел RGB сивият цвят се получава при еднакво количество от трите цвята. Конвертирайки едно изображение от черно-бяло в цветно, функцията заменя единичната стойност със списък от 3 такива – по една за червения, зеления и синия цвят. За тази цел се използва обектът „Image“ от библиотеката „PIL“. Тази библиотека е най-използваната за работа с изображение в Python. Скелетът на функцията е изобразен във *Фиг. 3.13*.

def get\_pixels(filename):  
 # Отваряне на снимката

# Конвертиране към цветовия модел RGB

# Конвертиране от обект от тип снимка към масив от пиксели

# Връщане на резултатния масив

*Фиг. 3.13: Скелет на функцията „get\_pixels”*

Втората функция във файла се използва само в тренирането на модела за лицево верифициране и нейното име е „extract\_single\_face“. Тази функция засича само едно лице в изображението и се използва при трениране с цел оптимизиране на бързодействието на процеса на трениране, тъй като обработката на десетки снимки отнема немалко време. Тя приема като аргументи името на файла, където е изображението, и модела за лицево засичане, а връща масив съдържащ пикселите на засеченото лице. Стойността на подразбиране на параметъра „required\_size“ е константа със стойност „(160,160)“, като тя може да бъде заменена с друга при извикването на функцията. Функцията „extract\_single\_face“ използва „get\_pixels“, за да получи пикселите на изображението. Впоследствие се изпълнява модела за лицево засичане, който връща като резултат координатите на правоъгълника, в който се намира лицето. Този правоъгълник се отделя от оригиналната снимка и функцията връща нова снимка с размер 160х160, съдържаща само лицето на потребителя. Скелетът на функцията е изобразен във *Фиг. 3.14*.

def extract\_single\_face(filename, classifier, required\_size=REQUIRED\_SIZE):  
 # Отваряне на снимката посредством „get\_pixels“  
  
 # Изпълняване на модела за лицево засичане  
  
 # Намиране на горния ляв и долния десен ъгъл на правоъгълника, в който се намира засеченото лице  
  
 # Създаване на нова снимка, съдържаща само лицето

# Връщане на новата снимка като резултат

*Фиг. 3.14: Скелет на функцията „extract\_single\_face“*

Третата функция, с наименование „extract\_multiple\_faces“, се използва само при изпълнението на алгоритъма за лицево верифициране, като тя засича множество лица в изображение, а не само едно. Това забавя леко изпълнението на програмата, но създава по-голям комфорт за потребителя. Нейното изпълнение наподобява изпълнението на „extract\_single\_face“, но при засичането на лица резултат е масив от координатите на правоъгълниците, в които се намират отделните лица в изображението. Функцията приема името на файла и модела за лицево засичане и връща масив от нови снимки на отделните лица в оригиналното изображение. Скелетът на функцията е показан във *Фиг. 3.15*.

def extract\_multiple\_faces(filename, classifier, required\_size=REQUIRED\_SIZE):  
 # Отваряне на снимката посредством „get\_pixels“  
  
 # Изпълняване на модела за лицево засичане  
  
 for face\_box in faces\_boxes:  
 # Намиране на горния ляв и долния десен ъгъл на правоъгълника, в който се намира засеченото лице  
  
 # Създаване на нова снимка, съдържаща само лицето   
  
 # Добавяне на снимката в резултатен масив   
  
 # Връщане на резултатния масив

*Фиг. 3.15: Скелет на функцията „extract\_multiple\_faces“*

Последните две помощни функции в този файл се използват само при тренирането и те са отговорни за зареждането и отварянето на всички изображения от колекцията от снимки за трениране на модела за лицево верифициране. Функцията „load\_faces“ извиква „extract\_single\_face“ за всеки изображение в подадена директория, като всички тези снимки за на един човек, като неговото име е името на директорията. Тя приема като аргумент името на директорията и модела за лицево засичане и връща като резултат масив с лицата от всички изображения в директорията. Скелетът на функцията е изобразен във *Фиг. 3.16*.

def load\_faces(directory, classifier):  
 # Обхождане на всички файлове в директорията

for filename in listdir(directory):  
 # Засичане на лице в съответното изображение  
  
 # Добавяне на засеченото лице в резултатен масив  
  
 # Връщане на резултатния масив като резултат

*Фиг. 3.16: Скелет на функцията „load\_faces“*

Функцията „load\_dataset“ извиква „load\_faces“ за всяка поддиректория в папката „dataset“, където се намират всички тренировъчни изображения. Всяка поддиректория представлява изображенията на отделен човек. Аргументите на функцията са името на директорията, в която са запазени всички изображения за трениране, която в контекста на дипломната работа е „“, и моделът за лицево засичане, който се инициализира при стартиране на програмата за трениране и се подава към помощните функции. Функцията връща два едноизмерни масива. В първия се пазят всички лица от колекцията за трениране, а във втория се пазят всички имена на хора. Скелетът на функцията е изобразен във *Фиг. 3.17*.

def load\_dataset(directory, classifier):  
 # Обхождане на всички поддиректории, като всяка една от тях съдържа изображения само на един човек  
 for subdir in listdir(directory):  
 # Взима лицата от всички изображения в поддиректорията посредством функцията „load\_faces“

# Добавяне на лицата в първия резултатен масив

# Добавяне на имената, съответстващи на хората от изображенията във втория резултатен масив

# Връщане на двата резултатни масива

*Фиг. 3.17: Скелет на функцията „load\_dataset”*

**3.3.4 Помощни функции за лицево верифициране**

При алгоритмите за лицево разпознаване моделът трябва да постави едно лице в едно 128-измерно пространство и да се намери до кое е най-близко, докато при алгоритмите за лицево верифициране моделът трябва да пресметне дали дадено лице е достатъчно близко до някое друго, така че да бъде познато. За да се постави едно лице в 128-измерно пространство се създава т.нар. *face embedding* („отливка“ на лицето). Тази лицева „отливка“ представлява 128 стойности, които описват лицето на един човек. Това е начинът, по който машината се „учи“ да разпознава или да верифицира лица. След извличане на лице от изображение, това лице се подава на модел за лицево верифициране, който създава тази „отливка“ на лицето. Функцията, който създава „отливката“ е в отделен файл с име „get\_face\_embedding“. Функцията приема отрязаната снимка на лице, както и модел за лицево верифициране и връща като резултат „отливката“ на лицето. Нейният скелет изглежда по следния начин:

def get\_face\_embedding(face\_pixels, model):  
 # Нормализиране на входните данни за модела  
  
 # Изпълняване на модела

# Връщане на лицевата „отливка“

*Фиг. 3.18: Скелет на функцията „get\_face\_embedding”*

**3.3.5 Трениране на модела за лицево верифициране**

Тренирането на модела за лицево верифициране е най-фундаменталната стъпка за постигането на точни резултати при изпълнение. Важно е да бъдат подадени достатъчно тренировъчни материали, за да се постигнат перцизност при изпълнението. Алгоритъмът за лицево верифициране се тества с по 30 изображения на човек, което му предразполага за много голяма прецизност при изпълнението в реална обстановка. Той се „тренира“, като извлича лицевите „отливки“ от всяко изображение. Така моделът знае къде се намира всяко лице в пространството от 128 измерения и при изпълнение може да се верифицира дали тестваното лице е познато или не. Скриптът за трениране на модела се намира във файла „“, като той съдържа само една функция, отговорна за събирането на всички „отливки“ на лицето от колекцията от изображения за трениране. Процесът на трениране на модела за лицево верифициране изглежда по следния начин:



*Фиг. 3.19: Блок схема на процеса на трениране*

*на модела за лицево верифициране*

**3.3.6 Изпълнение на алгоритъма за лицево верифициране**