

UNIVERZITA MATEJA BELA V BANSKEJ BYSTRICI
FAKULTA PRÍRODNÝCH VIED

ANALÝZA A SPRACOVANIE OBRAZU V
MOBILNÝCH APLIKÁCIÁCH TVORENÝCH
POMOCOU WEBOVÝCH TECHNOLOGIÍ

Diplomová práca

39c6251f-a57c-4a1a-8fd0-f17107f6a18f

Študijný program: Aplikovaná informatika

Študijný odbor: Aplikovaná informatika

Katedra: KIN FPV - Katedra informatiky

Vedúci bakalárskej práce: Mgr. Vagač Michal PhD.

Banská Bystrica, 2017

Bc. Tomáš Stankovič

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Tomáš Stankovič
Študijný program: aplikovaná informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: aplikovaná informatika
Typ záverečnej práce: Magisterská záverečná práca
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický


Názov: Analýza a spracovanie obrazu v mobilných aplikáciách tvorených pomocou webových technológií

Anotácia: Analyzujte možnosti tvorby mobilných aplikácií použitím webových technológií. Zamerajte sa na možnosti práce s obrazovými dátami. Vytvorte ukážkovú aplikáciu a overte funkčnosť takéhoto riešenia.

Vedúci: Mgr. Michal Vagač, PhD.
Katedra: KIN FPV - Katedra informatiky
Vedúci katedry: RNDr. Miroslav Melicherčík, PhD.

Dátum zadania: 05.04.2016

Dátum schválenia: 10.04.2017


prof. RNDr. Roman Nedela, DrSc.
garant študijného programu

Čestné prehlásenie

Čestne prehlasujem, že som túto diplomovú prácu vypracoval samostatne s použitím uvedených zdrojov.

V Banskej Bystrici, dňa 20. 4. 2017

.....

Pod'akovanie

Touto cestou chcem poďakovať Mgr. Michalovi Vagačovi PhD., vedúcemu mojej diplomovej práce za pomoc, čas a nespočetné množstvo cenných rád pri písaní a vývoji aplikácie.

Abstrakt

STANKOVIČ, Tomáš: Analýza a spracovanie obrazu v mobilných aplikáciach tvorených pomocou webových technológií

[Magisterská práca] / Bc. Tomáš Stankovič. - Univerzita Mateja Bela v Banskej Bystrici. Fakulta prírodných vied; Katedra informatiky. - Školiteľ: Mgr. Vagač Michal PhD., Banská Bystrica FPV UMB, 2017, 77 s

Práca sa venuje preskúmaniu možnosti spracovania obrazu v mobilných aplikáciách vyvíjaných v jazyku JavaScript. Konkrétne sa zaoberá spojením technológie React Native pre vývoj mobilných aplikácií, Node.js pre spracovávanie dát na serveri, knižnicou OpenCV pre spracovanie obrazu a neurónovou sieťou Tensorflow. Východiskové predpoklady sú overené vytvorením funkčnej mobilnej aplikácie slúžiacej k rozpoznávaniu stromov na základe fotografie ich listov.

Kľúčové slová

JavaScript, React Native, OpenCV, Node.js, Tensorflow

Abstract

STANKOVIČ, Tomáš: Image analysis and processing in mobile applications built using web technologies

[Diploma thesis] / Bc. Tomáš Stankovič. - Matej Bel University Banská Bystrica. Faculty of Natural Sciences; Department of Computer Science. - Thesis supervisor: Mgr. Vagač Michal PhD., Banská Bystrica FPV UMB, 2017, 77 s

This work is concerned with research of image processing options in mobile applications developed in JavaScript. Specifically, it deals with the combination of technology React Native for development of mobile application, Node.js for data processing on the server, OpenCV library for image processing and Tensorflow neural network. Initial assumptions are verified by creating a functional mobile app, that recognizes trees based on pictures of their leaves.

Keywords

JavaScript, React Native, OpenCV, Node.js, Tensorflow

Predhovor

Mobilné aplikácie sa stali bežnou súčasťou životov ľudí a vzniká veľké množstvo rôznych aplikácií. Vďaka veľkým výpočtovým výkonom je možné vytvárať aplikácie, ktoré sú schopné v reálnom čase vykonávať zložité analýzy, čítanie textu či rozpoznávanie objektov. V súčasnosti sa mnoho ľudí orientuje prioritne na technológie, ale zabúda na základné poznávanie prírody, ktoré nám ponúka príroda. V tejto práci máme za cieľ overenie spolupráce medzi vyššie uvedenými technológiami a vytvorenie mobilnej aplikácie, ktorá okrem toho prináša aj návrat k poznaniu prírody. Rozpoznávanie listov pomocou počítačového videnia či neurónových sietí je téma, ktorej sa už pokúšalo venovať viacero svetových univerzít, nevedli však k plnohodnotným aplikáciám použiteľným mimo laboratórneho prostredia.

Obsah

Úvod	13
1 Mobilné aplikácie v JavaScripte	14
1.1 Možnosti tvorby v JavaScripte	14
1.1.1 Dôvody vzniku technológie	14
1.1.2 Charakteristika mobilných JavaScript frameworkov	15
1.2 React Native	16
1.2.1 Komunikácia medzi natívnym kódom a JavaScriptom	17
1.2.2 Spracovanie obrazových dát v JS aplikácii	19
1.2.3 WebView ako simulácia rozšírenej reality	19
1.2.4 Komunikácia klient - server	20
2 Počítačové videnie v Node.js s použitím OpenCV	22
2.1 Reprezentácia, načítanie a uloženie obrázku	23
2.2 Základné metódy úpravy obrazu	24
2.3 Morfologické operácie	28
3 Neurónové siete	30
3.1 Úvod do neurónových sietí	30
3.2 Porovnanie knižníc	31
3.3 Pretrénovanie finálnej vrstvy pre rozpoznávanie objektov na obrázkoch	32

3.4	Zvyšovanie presnosti tréovania siete	35
4	Prípadová štúdia	36
4.1	Architektúra aplikácie	36
4.2	Tvorba mobilnej aplikácie	38
4.2.1	Dizajn mobilnej aplikácie	39
4.2.2	Zachytenie fotografie a jej odoslanie na server	40
4.2.3	Spracovanie dát zo servera	42
4.3	Spracovanie obrazu	44
4.3.1	Vyhľadávanie hrán	44
4.3.2	Aproximácia polygónom	48
4.3.3	Rotácia obrázku bez orezania	50
4.3.4	Porovnanie tvarov pomocou Fourierových deskriptorov	53
4.4	Rozpoznávanie obrazu pomocou neurónovej siete	54
4.4.1	Práca s neurónovou sieťou v jazyku Python	54
4.4.2	Tvorba HTTP rozhrania pre prístup k neurónovej sieti	56
	Záver	58
	Zoznam bibliografických odkazov	59

Zoznam obrázkov

1.1	Vytvorenie súboru do správnej zložky projektu v prostredí Xcode	17
2.1	Načítaný obrázok	24
2.2	Obrázok po použití metódy <code>Matrix.convertGrayscale()</code>	24
2.3	Obrázok po použití metódy <code>Matrix.convertHSVscale()</code>	26
2.4	Obrázok po použití metódy <code>Matrix.crop()</code>	27
2.5	Obrázok pred použitím metód erose a dilate	28
2.6	Obrázok po dilatácii	29
2.7	Obrázok po erózii	29
3.1	Diagram vrstiev neurónovej siete	31
3.2	Výsledok rozpoznávania triedy obrázku v našom modeli	34
4.1	Architektúra aplikácie	37
4.2	Implementačný diagram aplikácie	37
4.3	Ukážka dizajnu mobilnej aplikácie	39
4.4	Obrázok po použití metód <code>Matrix.convertGrayscale()</code> a <code>Matrix.gaussianBlur()</code>	45
4.5	Všetky nájdené hrany na obrázku po Canny Edge analýze	46
4.6	Obrázok s najväčšou nájdenou hranou	47
4.7	Obrázok s vybranými kontúrami	48
4.8	Aproximácia obdĺžnikom	49
4.9	Aproximácia polygónom so špecifikovanou presnosťou	49

4.10 Aproximácia rotovaným polygónom	50
4.11 Obrázok po správnej rotácii	51
4.12 Obrázok po správnej rotácii a Canny Edge vyhľadávani hrán	52

Zoznam symbolov a skratiek

Bazel - Nástroj pre kompilovanie zdrojových súborov

Bottleneck - Výpočet hodnôt pre proces učenia v neurónovej sieti

Build - Automatický proces vytvorenia produkčnej verzie aplikácie

Flask - HTTP framework jazyka Python

Flux - Aplikačná architektúra používaná spolu s knižnicou React.js

HTTP - Hypertextový prenosový protokol

MVC - Aplikačná architektúra: Model - View - Controller

NPM - Balíčkovací systém Node.js

REST API - Serverová architektúra rozhrania navrhnutá pre distribuované prostredia

Úvod

JavaScript vznikol v roku 1995 ako programovací jazyk, slúžiaci k manipulácii a doplneniu webového front-endu. V poslednom desaťročí sa však stal veľmi obľúbeným, najmä vďaka jeho vstupu do sveta webových serverov na základe vzniku projektu Node.js, ktorý je jedným z najobľúbenejších projektov s otvoreným kódom na svete. Jeho preniknutie do mnohých ďalších oblastí tak bolo úplne prirodzené. Mobilné aplikácie sa stali každodennou súčasťou života ľudí a vývojári začali hľadať cestu, ako pomocou tohto jazyka vytvárať takéto aplikácie. Dnes už tak môžeme tvoriť veľmi rýchlo kvalitné, natívne mobilné aplikácie pre najpopulárnejšie platformy - Android a iOS. Mnohé veľké firmy, ako tomu bolo pri prechode z klasického prostredia serverov do platformy Node.js, prechádzajú s vývojom svojich mobilných aplikácií do použitia takýchto technológií, práve vďaka rýchlosti vývoja, univerzálnosti a multiplatformovosti.

Počítače dnes disponujú vďaka rýchlemu vývoju veľkým výpočtovým výkonom a v informatike sa stala bežnou téma počítačového videnia a neurónových sietí. To nám umožňuje spracovávanie, analýzu a predikciu nad rôznymi dátovými množinami. Mobilné telefóny disponujú taktiež pomerne veľkým výpočtovým výkonom a množstvo výpočtov je možné vykonávať priamo v nich.

Cieľom tejto práce je preskúmať možnosti prepojenia všetkých týchto technológií a vytvoriť mobilnú multiplatformovú aplikáciu, slúžiacu na vzdelávanie ľudí v poznaní prírody. Konkrétne nám pôjde o rozpoznávanie stromov z fotografie ich listov.

Kapitola 1

Mobilné aplikácie v JavaScripte

Mobilné aplikácie sa v súčasnosti dostávajú v používaní pred webové aplikácie. Na vrchole popularity je aj jazyk Javascript, ktorý zasahuje takmer do všetkých technických oblastí, mobilné aplikácie nevynímajúc.

1.1 Možnosti tvorby v JavaScripte

Vďaka rastúcej popularite, ale najmä univerzálnosti JavaScriptu sa komunita začala zaujímať o presun webových technológií z prostredia desktopového prehliadača do mobilných zariadení. Po rokoch vývoja sa možnosti tvorby výrazne posunuli a výsledkom je mnoho natívnych aplikácií vytvorených v jazyku, ktorý nevznikol primárne pre tento účel.

1.1.1 Dôvody vzniku technológie

V roku 2009 bol predstavený prvý projekt - Apache Cordova, venujúci sa problematike, na ktorého základoch stavajú aj niektoré zo súčasných frameworkov. Tento projekt bol vydaný s otvoreným kódom a pôvodne podporoval len platformu iOS. Cordova umožnila prístup k niektorým hlavným natívnym funkcionalitám (kamera, kompas, kontakty, atď.) cez Cordova API. Aplikácia samotná bola tvorená HTML, CSS a JavaScript kódom. Aplikácia bežala pomocou engine webového prehliadača skrytého za mobilnú aplikáciu, s odstráneným užívateľským rozhraním samotného prehliadača.

Táto technológia si prešla zreteľným vývojom, počas ktorého pridala podporu až pre 10

mobilných platforiem. V praxi to znamená, že napíšete jednu aplikáciu, ktorú môžete následne vybuildovať na viacero platforiem. Do tvorby mobilných aplikácií pomocou webových technológií to prináša veľkú výhodu, ktorá je hlavným dôvodom aktívneho vývoja až do dnešných dní.

V súčasnosti sa situácia okolo mobilných aplikácia výrazne posunula k lepšiemu. Na webe sa stali bežnou súčasťou Single-page aplikácie, ktoré preberajú množstvo business logiky zo servera na klientský JavaScript kód. Server zohráva úlohu poskytovania REST API pre aplikáciu. Hlavnými frameworkami pre tvorbu webových aplikácií sa stali dva open-source projekty. Angular, ktorý vyšiel v roku 2016 vo svojej druhej verzii, a je vyvíjaný v spoločnosti Google. Druhý populárny framework je React.js. Priniesol veľké zmeny aj v architektúre aplikácií, ktoré sa postupne presunuli z MVC do FLUX architektúry. Tento framework je vyvíjaný spoločnosťou Facebook, ktorý ju aj aktívne používa v produkčných aplikáciách. Na týchto dvoch webových frameworkoch stavajú aj mobilné frameworky, na ktoré sa bližšie pozrieme.

1.1.2 Charakteristika mobilných JavaScript frameworkov

Ionic framework je jeden z prvých frameworkov, ktorý som zvažoval pre implementáciu v tejto magisterskej práci. Pomocou neho som pracoval na dvoch mobilných aplikáciách, ktoré sú v súčasnosti v produkcii. Je založený na mobilnom frameworku Angular 2, a JavaScript sa píše v TypeScripte, ktorý prináša typovosť do tohto jazyka. Framework je postavený na už spomínanej technológii Apache Cordova, a pomocou neho vytvoríme webovú aplikáciu, ktorá sa tvári ako natívna. Získame tak prístup ku všetkým častiam webu, ako ich poznáme. Jeho nevýhodou je zložitá tvorba natívnych modulov a už spomínaný beh v skrytom prehliadači.

React Native bol uverejnený v roku 2015 s podporou iOS, ktorú neskôr doplnil aj o podporu Androidu. Technológia, ako aj ostatné z dielne Facebooku, bola v čase vydania nasadená v produkcii na niekoľkých aplikáciách. Tento stav pretrváva do súčasnosti, technológia sa stále rýchlo vyvíja a je pomerne mladá. React Native má výborne vytvorené nástroje pre vývoj a debugovanie aplikácie, silnú a progresívnu komunitu a výhodu oproti Ionic v priamom vývoji na telefóne či emulátore. Ionic používa pre zobrazovanie

aplikácie pri vývoji samotný webový prehliadač. Veľkou výhodou tejto technológie je fakt, že výsledné používateľské rozhranie aplikácie je kompilované do natívneho kódu platformy a aplikácia tak umožňuje plne natívny pocit z mobilnej aplikácie.

NativeScript je postavený, rovnako ako Ionic na Angular 2 a Typescript technológiách. Bol však inšpirovaný už spomínaným React Native a pracuje veľmi podobne. Rozhranie aplikácie je kompilované do natívneho kódu platformy.

Všetky tri frameworky podporujú platformy Android a iOS.

Predmet práce nie je podrobne oboznamovať čitateľa s týmito frameworkami, ale len s časťami podstatnými pre spracovanie obrazu. Pre implementáciu mobilnej aplikácie som si vybral framework React Native, kvôli lepšiemu výkonu práve z dôvodu kompilovania aplikácie do natívneho kódu a lepších nástrojov pre vývojárov v porovnaní s ostatnými frameworkami.

1.2 React Native

React Native je technológia vydaná ako open source firmou Facebook v marci roku 2015. Vychádza z webovej knižnice React, ktorá slúži ako zobrazovacia vrstva vo webových aplikáciách. React Native, ako už z názvu vyplýva je kombináciou Reactu a natívneho kódu na mobilných zariadeniach. V súčasnosti má podporu operačných systémov iOS a Android.

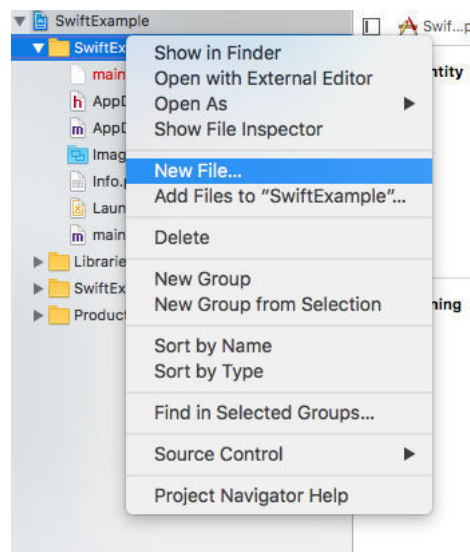
„React Native je prvá JavaScriptová knižnica, ktorá umožňuje kompilovanie aplikácie napísanej v jazyku JavaScript do natívneho kódu - Objective C, Java podľa platformy. Presnejšie celá UI vrstva je kompilovaná do natívneho kódu a vrstva business logiky ostáva v JavaScripte, s tým že máme pochopiteľne prístup k natívnym API (kamera, kontakty, bluetooth, atď.). Taktiež môžeme kritické časti aplikácie napísať v natívnom kóde a komunikovať s nimi pomocou nášho JavaScriptového kódu.” [3]

Nasledujúca podkapitola sa venuje častiam React Native podstatným pre potreby tejto práce, ako je implementácia komunikácie s natívnym kódom, spracovanie obrazových dát a komunikácia klient-server.

1.2.1 Komunikácia medzi natívnym kódom a JavaScriptom

React Native obsahuje vo svojom jadre množstvo natívneho kódu, ktorý je sprístupnený vývojárovi cez takzvaný most. Ide zväčša o situácie, kedy sa potrebujeme odvolávať priamo na natívnu funkcionálnosť, ako napríklad kamera a podobne. Vývojár má tiež možnosť písať vlastnú natívnu funkcionálnosť. To môže byť vhodné pre výpočtovo náročnejšie časti aplikácie, alebo tiež pre sprístupnenie natívnych modulov či knižníc, ktoré ešte svoje prepojenie s React Native nemajú naprogramované.

Najskôr je potrebné vytvoriť si súbor s logikou, ktorú chceme spúšťať v natívnom prostredí. V prípade iOS môžeme vytvoriť túto časť v programovacích jazykoch Objective-C alebo Swift. Pri platforme Android je to kód jazyka Java. My si ukážeme, ako vytvoriť Swift kód pre platformu iOS, tak aby komunikoval s našou React Native aplikáciou. Všetky natívne súbory, ktoré budeme v tejto kapitole vytvárať, je potrebné uložiť do /ios zložky v našom projekte (Obr. 1.1).



Obr. 1.1: Vytvorenie súboru do správnej zložky projektu v prostredí Xcode (Zdroj: autor)

Nasleduje ukážka natívneho kódu našej aplikácie v programovacom jazyku Swift: `MyModule.swift`.

```
1 import Foundation
2
3 @objc(MyModule)
```

```

4 class MyModule: NSObject {
5     @objc func callbackMethod(_ callback: RCTResponseSenderBlock) {
6         let resultsDict: Any = [
7             "pole" : [1,2,3],
8             "pole2": [4,5,6],
9             "retazec": "ahoj"
10        ];
11        callback([NSNumber() ,resultsDict])
12    }
13
14    @objc func simpleMethod(_ a: String, b: String, c: Int) {
15        NSLog("SWIFT \(a) \(b) x \(c)");
16    }
17 }

```

V predchádzajúcom kóde môžeme vidieť definíciu našej triedy MyModule, ktorá obsahuje dve metódy: `callbackMethod`, ktorá nám vracia pole s dátami, a metóda `simpleMethod`, ktorá prijíma 3 hodnoty a vypisuje ich do konzoly. Na sprístupnenie týchto metód potrebujeme súbor v jazyku Objective-C, ktorý bude takzvaným prepojujúcim mostom: `MyModule.m`

```

1 //React Native version < 0.40.0
2 #import "RCTBridgeModule.h"
3 //React Native version >= 0.40.0
4 #import <React/RCTBridgeModule.h>
5
6 @interface RCT_EXTERN_MODULE(MyModule, NSObject)
7 RCT_EXTERN_METHOD(callbackMethod:(RCTResponseSenderBlock)callback)
8 RCT_EXTERN_METHOD(simpleMethod:(NSString *)a b:(NSString *)b c:(
9     NSInteger *)c)
9 @end

```

V predchádzajúcom kóde môžeme vidieť jednoduchosť, s akou sa dá akýkoľvek natívny kód prepojiť s JavaScriptovou aplikáciou v React Native.

Následne je nutné vytvoriť súbor s názvom: `MojProjekt-Bridging-Header.h` s nasledujúcim obsahom. Xcode editor kódu nám pri vytváraní `.m` súboru ponúka automatické vytvorenie práve tohto `Bridging-Header` súboru. V prípade že sme potvrdili jeho vytvorenie, zmeníme len obsah súboru.

```

1 #import <React/RCTBridgeModule.h>

```

V aplikácii nám tak zostáva už len import našej funkcionality a jej používanie.

```
1 let MyModule = NativeModules.MyModule;
2 export default class MyApp extends Component {
3   constructor() {
4     MyModule.simpleMethod2('i', 'love code', 3);
5     MyModule.callbackMethod((err, data) => {
6       console.log(data);
7     });
8   }
9 }
```

1.2.2 Spracovanie obrazových dát v JS aplikácii

Pre zobrazovanie obrazových dát v React Native máme k dispozícii komponent `Image`. Ten umožňuje zobrazovanie statických súborov, lokálnych súborov v disku zariadenia a súborov dostupných pomocou HTTP. Použitie komponentu je jednoduché a rýchle.

Pokiaľ ide o získavanie obrazových dát, tie môžeme získavať z kamery zariadenia pomocou npm modulu `react-native-camera` vyvinutého komunitou. Po inštalácii modulu ho môžeme používať ako klasický React komponent, ktorý konfigurujeme pomocou vlastností, do neho posielaných.. Tie umožňujú nastavenie kvality, umiestnenie fotografie, orientáciu a ďalšie. Metóda `capture` tejto knižnice vyvolá vytvorenie fotografie v zariadení, čo má za následok vrátenie *Promise* s cestou k zachytenej fotografii.

S touto fotografiou môžeme následne pracovať buď pomocou niekoľkých dostupných npm modulov, pomocou vlastného natívneho kódu pre zložitejšie operácie, alebo odslaním a následným spracovaním na serveri.

1.2.3 WebView ako simulácia rozšírenej reality

Rozšírená realita je špecifickým problémom počítačového videnia, v ktorej je potrebné upravovať a dopĺňať o informácie obraz z kamery v reálnom čase. Jednou z možností, ako ju simulovať pomocou React Native je zobrazovanie živého náhľadu z kamery, nad ktorým je umiestnená priehľadná vrstva `WebView`, poskytujúca potrebné informácie na rozšírenie našej reality. „`WebView` dokáže vykresliť klasický HTML, CSS a JavaScript

kód v používateľskom rozhraní aplikácie a môže tiež v základe fungovať na zobrazenie celých webových stránok, teda ako náš vlastný webový prehliadač vo vnútri aplikácie. Ak potrebujeme pracovať s klasickým webovým kódom v používateľskom rozhraní, WebView je naša jediná možnosť keďže ostatné komponenty sú kompilované do natívneho rozhrania.” [10]

Pre predstavu môže vyzeráť naša render metóda s použitím WebView vrstvy s vlastným JavaScript a HTML kódom nasledovne:

```
1  render() {
2      let html = `
3          <div id='helloElement">
4              Hello
5          </div>
6      `;
7      let jsCode = `
8          document.querySelector('#helloElement').style.
9              backgroundColor = 'yellow';
10     `;
11     return (
12         <View>
13             <WebView
14                 ref="myWebView"
15                 html={html}
16                 injectedJavaScript={jsCode}
17                 javaScriptEnabledAndroid={true}
18             >
19         </WebView>
20     </View>
21 );
22 }
```

1.2.4 Komunikácia klient - server

React Native komunikuje so serverom cez HTTP API pomocou JavaScript Fetch API. Oslovovanie servera s GET požiadavkou môže vyzeráť nasledovne:

```
1  fetch('https://mywebsite.com/mydata.json')
```

Požiadavka POST, v ktorej posielame serveru aj nejaké dáta môže pomocou Fetch API vyzeráť takto:

```
1 fetch('https://mywebsite.com/endpoint/', {
2   method: 'POST',
3   headers: {
4     'Accept': 'application/json',
5     'Content-Type': 'application/json',
6   },
7   body: JSON.stringify({
8     firstParam: 'yourValue',
9     secondParam: 'yourOtherValue',
10  })
11 })
```

Fetch API vytvára asynchrónnu operáciu na server, čo znamená, že tok našej aplikácie sa nezablokuje a môžeme s ňou pracovať aj pri dlhších časoch spracovania na serveri. Odpoveď servera dostávame zachytávaním jej odpovede na volanej fetch metóde, pomocou takzvaného Promise API dostupného v JavaScripte, ktoré sa stalo v posledných rokoch štandardom pre spracovanie asynchrónnych operácií v tomto jazyku.

```
1 function getData() {
2   return fetch('https://mywebsite.com/endpoint/')
3     .then((response) => response.json())
4     .then((responseJson) => {
5       return responseJson.movies;
6     })
7     .catch((error) => {
8       console.error(error);
9     });
10 }
```

Kapitola 2

Počítačové videnie v Node.js s použitím OpenCV

Počítačové videnie je časť informatiky, ktorá získava údaje zo zachyteného obrazu či videa pomocou rôznych algoritmov.

„Vizuálna detekcia a rozpoznávanie objektov patrí v poslednom desaťročí medzi najväčšie výzvy počítačového videnia. Potencionálne uplatnenie systémov detekcie a rozpoznávania objektov je veľmi široké, od bezpečnostných systémov (napr. identifikácia oprávnených osôb) cez medicínske techniky (napr. detekcia tumorov), priemyselné aplikácie (napr. vizuálna inšpekcia výrobkov), robotiku (napr. navigácia robota v nedostupnom teréne) až po rozšírenú realitu a mnohé ďalšie.” [17]

Implementovanie algoritmov počítačového videnia je netriviálna úloha. Preto vznikla v roku 1999 Open Source knižnica OpenCV, ktorá sa vyvíja aktívne až dodnes. Oficiálne podporuje jazyky C, C++, Python a Java. Avšak JavaScriptová komunita na čele s programátorom Petrom Bradenom začala pracovať na interface medzi týmto jazykom a knižnicou pod názvom node-opencv. Vďaka nej môžeme budovať aplikácie postavené na Node.js s možnosťou analýzy obrazu pomocou počítačového videnia. Nie je možné s ňou pracovať na strane klientskeho JavaScriptového kódu, preto je nutné presunúť logiku spracovania obrazu na Node.js server. Táto knižnica momentálne nemá implementované úplne všetky časti OpenCV, a najmä tie z tretej verzie OpenCV, avšak používať ju je možné už dnes. Knižnica má v súčasnosti žiaľ slabú dokumentáciu a vývojár sa nevyhne študovaniu C++ implementácie, aj pri banálnych metódach a ich parametroch.

To je dôvod vzniku tejto kapitoly a obsahu kapitoly o Prípadovej štúdii vyvíjanej aplikácie a ukážky niekoľkých základných operácií, ktoré sa však nenachádzajú v oficiálnej

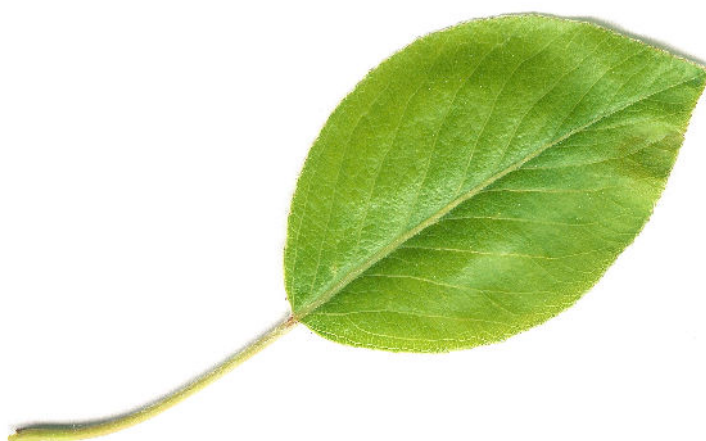
dokumentácii node-opencv knižnice.

2.1 Reprezentácia, načítanie a uloženie obrázku

Základom práce s OpenCV je možnosť načítania a uloženia obrázku. To môžeme uskutočniť pomocou nasledujúceho kódu:

```
1  const cv = require('opencv');
2
3  cv.imread('./img/myImage.jpg', function (err, img) {
4    if (err) {
5      throw err;
6    }
7
8    const width = img.width();
9    const height = img.height();
10
11   if (width < 1 || height < 1) {
12     throw new Error('Image has no size');
13   }
14
15   img.save('./img/myNewImage.jpg');
16 });
```

Načítaný obrázok je objekt, ktorý reprezentuje základnú dátovú štruktúru pre prácu s obrazom v OpenCV - Matrix. Každý načítaný obrázok je reprezentovaný takouto maticou, s ktorou môžeme následne pracovať, kde jedna hodnota poľa je jeden pixel obrázku, a tak veľkosť Matrix matice je definovaná veľkosťou načítaného obrázku (Obr. 2.1).

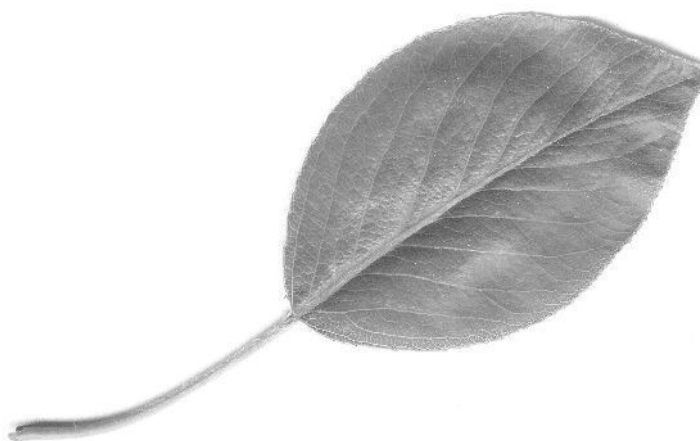


Obr. 2.1: Načítaný obrázok (Zdroj: autor)

2.2 Základné metódy úpravy obrazu

Príkladom zo základných funkcií, ktoré môžeme volať na načítanom obrázku (a často ich aj používame pri analýze obrazu) sú funkcie na konvertovanie farebného priestoru. Jednoducho týmto spôsobom môžeme získať obrázok v stupňoch šedej pomocou `Matrix.convertGrayscale()` metódy (Obr. 2.2).

```
1 img.convertGrayscale();  
2 img.save('./img/myGrayscaleImg.jpg');
```



Obr. 2.2: Obrázok po použití metódy `Matrix.convertGrayscale()` (Zdroj: autor)

Túto funkciu používame pomerne často, napríklad pred používaním detekcie hrán pomocou Canny Edge metódy.

Konvertovať na HSV cylindrický farebný profil môžeme pomocou funkcie *Matrix.convertHSVscale()* (Obr. 2.3).

```
1 img.convertHSVscale();
2 img.save('./img/myGrayscaleImg.jpg');
```

Zavolaním metódy `convertHSVscale` v našom JavaScriptovom kóde sa zavolá nasledujúci kód jazyka C++. Matica `self` je vytvorená z obrázku, nad ktorým je naša metóda v JavaScriptovom kóde volaná, následne je vytvorená nová matica `hsv`, ktorá slúži na uloženie modifikovaného obrázku. Na záver je späť skopírovaný obsah matice `hsv` do nášho objektu v JavaScripte.

```
1 NAN_METHOD(Matrix::ConvertHSVscale) {
2   Nan::HandleScope scope;
3
4   Matrix *self = Nan::ObjectWrap::Unwrap<Matrix>(info.This());
5   if (self->mat.channels() != 3) {
6     Nan::ThrowError("Image is no 3-channel");
7   }
8
9   cv::Mat hsv;
10
11  cv::cvtColor(self->mat, hsv, CV_BGR2HSV);
12  hsv.copyTo(self->mat);
13
14  info.GetReturnValue().Set(Nan::Null());
15 }
```



Obr. 2.3: Obrázok po použití metódy `Matrix.convertHSVscale()` (Zdroj: autor)

Jednou z ďalších základných operácií spracovania obrazu je orezanie obrázku. Pri používaní predchádzajúcich metód sa menila priamo matica `Matrix`, pri volaní `Crop` funkcie však dostávame novú maticu v návratovej hodnote, ktorá je v skutočnosti podobrázkom ukazujúcim na obrázok, nad ktorým bola metóda zavolaná. Takáto implementácia prispieva k rýchlosti spracovania a šetreniu pamäte. Na orezávanie obrázku používame nasledujúcu metódu `Matrix.crop(x, y, width, height)` (Obr. 2.4).

```
1 let croppedImg = img.crop(1000, 1000, 1000, 1000);
2 croppedImg.save('./img/croppedImg');
```

Po zavolaní `crop` metódy nad našim cieľovým obrázkom je volaný nasledujúci blok C++ kódu, kde prebieha kontrola správnosti parametrov, z ktorých sa vytvárajú nové premenné s pozíciou a veľkosťou nového obrázka. Následne sa vytvára nový orezaný obrázok, ktorý je vrátený ako nová premenná.

```
1 NAN_METHOD(Matrix::Crop) {
2     SETUP_FUNCTION(Matrix)
3
4     if ((info.Length() == 4) && (info[0]->IsNumber()) && (info[1]->
5         IsNumber())
6         && (info[2]->IsNumber()) && (info[3]->IsNumber())) {
7         int x = info[0]->IntegerValue();
```

```

8     int y = info[1]->IntegerValue();
9     int width = info[2]->IntegerValue();
10    int height = info[3]->IntegerValue();
11
12    cv::Rect roi(x, y, width, height);
13
14    Local < Object > im_h =
15        Nan::New(Matrix::constructor)->GetFunction()->NewInstance()
16        ;
17    Matrix *m = Nan::ObjectWrap::Unwrap<Matrix>(im_h);
18    m->mat = self->mat(roi);
19
20    info.GetReturnValue().Set(im_h);
21 } else {
22     info.GetReturnValue().Set(Nan::New("Insufficient or wrong
23         arguments").ToLocalChecked());
24 }

```



Obr. 2.4: Obrázok po použití metódy `Matrix.crop()` (zdroj: autor)

Takýmto spôsobom môžeme pracovať so základnými funkciami, ktoré prebiehajú nad Maticou `Matrix`. Nájde tu rôzne rozostrovacie filtre, funkcie pre kreslenie a ďalšie úpravy obrázku.

Pokiaľ chceme skopírovať obrázok z jednej premennej do druhej, používame na to metódu `Matrix.copy()`, s ktorou potom pracujeme ako so samostatným obrázkom.

```

1 let newImg = img.copy();

```

2.3 Morfologické operácie

Erózia a dilatácia sú základnými metódami matematickej morfológie. Ich fungovanie nám objasní úprava nasledujúceho obrázku pomocou týchto algoritmov (Obr. 2.5).



Obr. 2.5: Obrázok pred použitím metód erózie a dilatacie (Zdroj: [11])

Dilatácia A podľa B , kde A_b je prekladanie A podľa b je

$$A \oplus B = \bigcup_{b \in B} A_b$$

OpenCV nám ponúka možnosť volať funkciu `Matrix.dilate(iterations, structEl)`, kde `iterations` znamená, koľkokrát sa dilatácia vykoná a `structEl` je štruktúrálny element používaný pri dilatácii (prednastavené 3x3).

Ak zavoláme funkciu `dilate` nad obrázkom s nasledujúcimi parametrami

```
1 img.dilate(3);
```

volá sa OpenCV metóda s parametrami:

```
1 cv::dilate(self->mat, self->mat, structEl, cv::Point(-1, -1), 3);
```

Po zavolaní tejto funkcie sa na daný obrázok aplikuje dilatácia s jadrom o veľkosti 3. Výsledkom tohto volania je obrázok (2.6).



Obr. 2.6: Obrázok po dilatácii (zdroj: autor)

Erózia A podľa B, kde E je integer grid:

$$A \ominus B = \{z \in E \mid B_z \subseteq A\}$$

V OpenCV môžeme túto funkciu volať podobne ako funkciu `dilate`, `Matrix.erode(iterations, structE1)`, kde `iterations` znamená, koľkokrát sa erózia vykoná a `structE1` je štrukturovaný element používaný pri erózii (prednastavené 3x3).

Keď spustíme metódu `erode` nasledovne

```
1 img.erode(3);
```

JavaScriptové API volá nasledujúci C kód na našom obrázku podobne, ako pri metóde `dilate` (Obr. 2.7).

```
1 cv::erode(self->mat, self->mat, structE1, cv::Point(-1, -1), 3);
```



Obr. 2.7: Obrázok po erózii (zdroj: autor)

Kapitola 3

Neurónové siete

Neurónová sieť je jeden z výpočtových modelov umelej inteligencie. Umelá neurónová sieť je určená pre distribuované paralelné spracovávanie dát. Sieť sa skladá z umelých neurónov, pre vytvorenie ktorých bol vzorom skutočný biologický neurón. Neuróny sú navzájom pospájané a signály, ktoré si medzi sebou posielajú transformujú pomocou prenosových funkcií. Neurón môže mať ľubovoľný počet vstupov, ale vždy len jeden výstup.

Neurónové siete majú široké využitie od predpovedania vývoja burzových akcií, rozpoznávanie písma, cez predpovedanie správania používateľov až po úpravu, tvorbu či rozpoznávanie zvukových alebo obrazových dát. Práve tomu sa budeme venovať v tejto práci.

3.1 Úvod do neurónových sietí

Jeden z najpoužívanejších modelov umelého neurónu je McCullocheov a Pittsov model, ktorý môžeme definovať nasledovne:

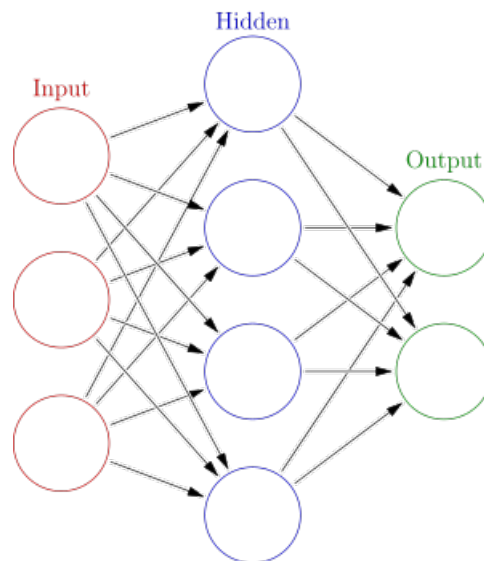
$$Y = S\left(\sum_{i=1}^N (w_i x_i) + \Theta\right)$$

kde:

- x_i sú vstupy neurónu
- w_i sú synaptické váhy

- \emptyset je prah
- $S(x)$ je prenosová funkcia neurónu
- Y je výstup neurónu

Neurónovú sieť môžeme vďaka učeniu nastaviť tak, aby nám poskytovala požadované výsledky. Počas učenia sieť upravuje váhy svojich neurónov. Učenie neurónovej siete môžeme rozdeliť do dvoch kategórií: Učenie s učiteľom a Učenie bez učiteľa. Prvá spomínaná možnosť pracuje s predloženým vzorom na jeho analýzu, ktorého výslednú hodnotu porovnáme s požadovaným výsledkom a určíme chybu. Následne spravíme korekcie a upravíme hodnoty váh, aby sme znížili chybu. Tento postup opakujeme až kým sa nepriblížime k požadovanej presnosti, respektíve veľkosti chyby. Učenie bez učiteľa nevyhodnocuje výstup, pretože ho nepoznáme. Sieť si na základe vzoriek sama prispôsobuje topológiu podľa vlastností vstupu (Obr. 3.1). [6]



Obr. 3.1: Diagram vrstiev neurónovej siete (zdroj: [6])

3.2 Porovnanie knižníc

Pre strojové učenie, a teda aj budovanie neurónových sietí existuje viacero knižníc s otvoreným zdrojovým kódom pre rôzne programovacie jazyky. Jednou z pomerne starších, ale veľmi obľúbených knižníc je Torch pre programovací jazyk Lua v ktorom je

aj naprogramovaná, s výnimkou jadra naprogramovanom v jazyku C. Knižnica vznikla v roku 2002 a stále je udržiavaná, čo dokazuje aj vydanie jej siedmej verzie vo februári roku 2017. Svojím obsahom je veľmi pokročilá a poskytuje širokú škálu algoritmov pre strojové učenie.

Tensorflow je takisto knižnica s otvoreným zdrojovým kódom, ktorá je v porovnaní s knižnicou Torch značne mladšia. Počiatočné vydanie verzie 0.1 sa uskutočnilo v novembri roku 2015 a vo februári 2017 sa knižnica dočkala vydania svojej prvej oficiálnej verzie, pod vedením spoločnosti Google. Projekt historicky vychádza zo staršej knižnice DistBelief z roku 2011, ktorá bola vyvíjaná v rámci projektu Google Brain. Po jej masívnom refaktorovaní a priblížení funkcionality bližšie k bežnejším vývojárom bola vydaná práve pod súčasným názvom Tensorflow.

Tensorflow je takisto veľmi pokročilá knižnica s obrovským množstvom algoritmov pre strojové učenie a budovanie neurónových sietí. Prichádza tiež s výbornou podporou rozpoznávania obrazu, kde pre mnoho prípadov stačí pretrénovať len poslednú vrstvu neurónovej siete, ako si ukážeme v nasledujúcej podkapitole. Spoločnosť Google ponúka tiež dobrú podporu pre prácu s Tensorflow v rámci ich vlastného cloudového riešenia Google Cloud Platform, kde je predpripravená architektúra pre tréovanie a testovanie siete. Obe spomínané knižnice poskytujú podporu výpočtov ako na CPU, tak aj na GPU procesoroch.

3.3 Pretrénovanie finálnej vrstvy pre rozpoznávanie objektov na obrázkoch

Moderné rozpoznávanie objektov je nesmierne náročná úloha a zaberie mnoho času konfiguráciou, a ešte viac tréovaním siete. Vývojári knižnice Tensorflow mysleli na túto situáciu, ktorú môžu používatelia riešiť pomerne často. Práve tento prístup dokazuje snahu Googlu priblížiť sa viac do produkčného prostredia aplikácií, ktorú sme spomínali v predchádzajúcej kapitole. Tensorflow pre toto použitie umožňuje použiť plne natrénovaný model a pretrénovať len finálnu vrstvu siete, pre rozpoznávanie našej dátovej množiny. Čas tréovania sa rapídne skraca a je možné bez väčších problé-

mov natrénovať sieť aj na osobnom počítači. Samozrejme to závisí od veľkosti dátovej množiny a prostredia, pre ktoré sieť trénujeme. V tejto podkapitole si ukážeme, ako pretrénovať finálnu vrstvu siete, pre rozpoznávanie našich obrázkov pomocou spúšťania príkazov z terminálu.

V prvom kroku potrebujeme dátový set obrázkov, rozdelený do zložiek podľa typu obrázkov (tzn. auto, motorka, bicykel). Následne je potrebné spustiť build pretrénovacieho skriptu pomocou nasledujúceho *bazel* príkazu.

```
$ bazel build tensorflow/examples/image_retraining:retrain
```

Po jeho dokončení, čo môže trvať približne polhodinu môžeme spustiť skript preučania nad našou konkrétnou dátovou množinou. Tento skript načíta predtrénovaný model, odstráni jeho finálnu vrstvu a trénuje novú nad našimi dátami. Dĺžka učenia závisí od množstva obrázkov a tiež parametrov, s akými preučací skript spúšťame.

```
$ bazel-bin/tensorflow/examples/image_retraining/retrain --  
  image_dir ~/leaf_dataset
```

Prvou fázou skriptu je takzvaná bottleneck analýza každého z obrázkov, čo znamená výpočet jeho hodnôt pre klasifikáciu používanú v predposlednej vrstve siete. Tieto bottleneck dáta sú uložené v textovom súbore v adresári */tmp/bottleneck*. Nie je nutné túto analýzu vykonávať zakaždým, čo značne zrýchľuje prípadné ďalšie pretrénovanie našej siete. Pokiaľ pridáme nové obrázky do našej dátovej množiny, prebehne bottleneck analýza len nad týmito obrázkami. Výsledky analýzy skript doplní do našej */tmp/bottleneck* zložky na disku.

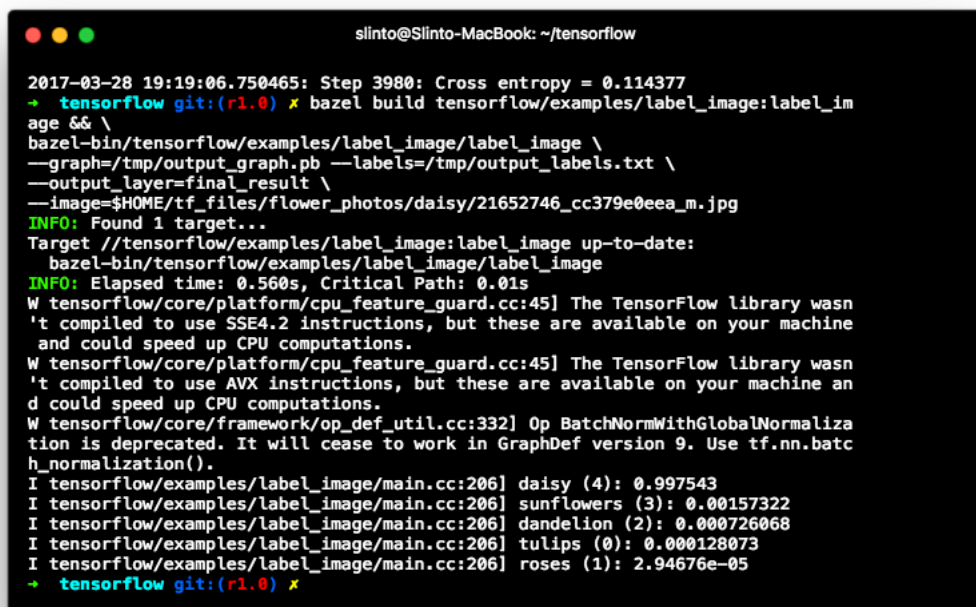
Po skončení bottleneck analýzy je sieť pripravená pre trénovanie. Na obrazovke sa bude postupne vypisovať séria hodnôt ako presnosť trénovania, validácie a cross entropia. Presnosť trénovania ukazuje, koľko percent z obrázkov používaných v použitej tréningovej množine bolo označené správnou triedou. Presnosť validácie je presnosť na náhodne vybranej skupine obrázkov. Dôležitý je pomer týchto dvoch validácií. Ak je hodnota trénovania príliš vysoká a hodnota validácie je naopak nízka, naša sieť je pretrénovaná a pamätá si zbytočné detaily niektorých obrázkov, ktoré nie sú užitočné

vo všeobecnosti pre rozhodovanie siete. Cross entropia je stratová funkcia, ktorá nám dáva informáciu o tom, ako dobre proces učenia pokračuje.

Po skončení tréningu je sieť pripravená pre používanie. Tensorflow nám dáva možnosť oslovovať sieť s našou požiadavkou pomocou C++ a Python API, takisto ako aj pomocou príkazov volaných z terminálu so správnymi parametrami, čo si ukážeme na nasledujúcej ukážke.

```
$ bazel build tensorflow/examples/label_image:label_image && \
bazel-bin/tensorflow/examples/label_image/label_image \
--graph=/tmp/output_graph.pb --labels=/tmp/output_labels.txt \
--output_layer=final_result \
--image=$HOME/leaf_dataset/malus-florentina/2746.jpg
```

Parametrami skriptu sú cesty k natrénovaným dátam a tiež cesta k obrázku, ktorý chceme v našej sieti priradiť do jednej z našich tried. Príklad výsledku skriptu môžeme vidieť na obrázku 3.2.



```
slinto@Slinto-MacBook: ~/tensorflow
2017-03-28 19:19:06.750465: Step 3980: Cross entropy = 0.114377
+ tensorflow git:(r1.0) x bazel build tensorflow/examples/label_image:label_image && \
bazel-bin/tensorflow/examples/label_image/label_image \
--graph=/tmp/output_graph.pb --labels=/tmp/output_labels.txt \
--output_layer=final_result \
--image=$HOME/tf_files/flower_photos/daisy/21652746_cc379e0eea_m.jpg
INFO: Found 1 target...
Target //tensorflow/examples/label_image:label_image up-to-date:
  bazel-bin/tensorflow/examples/label_image/label_image
INFO: Elapsed time: 0.560s, Critical Path: 0.01s
W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to use SSE4.2 instructions, but these are available on your machine and could speed up CPU computations.
W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to use AVX instructions, but these are available on your machine and could speed up CPU computations.
W tensorflow/core/framework/op_def_util.cc:332] Op BatchNormWithGlobalNormalization is deprecated. It will cease to work in GraphDef version 9. Use tf.nn.batch_normalization().
I tensorflow/examples/label_image/main.cc:206] daisy (4): 0.997543
I tensorflow/examples/label_image/main.cc:206] sunflowers (3): 0.00157322
I tensorflow/examples/label_image/main.cc:206] dandelion (2): 0.000726068
I tensorflow/examples/label_image/main.cc:206] tulips (0): 0.000128073
I tensorflow/examples/label_image/main.cc:206] roses (1): 2.94676e-05
+ tensorflow git:(r1.0) x
```

Obr. 3.2: Výsledok rozpoznávania triedy obrázku v našom modeli (zdroj: autor)

3.4 Zvyšovanie presnosti tréovania siete

Náš spúšťaný `retrain` skript má prednastavený počet učiacich krokov na hodnotu 4000, ktorú však môžeme meniť pomocou parametra `--how_many_training_steps` pri spúšťaní skriptu. Každý krok vyberie 10 náhodných obrázkov z tréningovej množiny, načíta ich `bottlenecks` hodnoty z `tmp/` zložky a vkladá ich do finálnej vrstvy siete pre získanie predikcie. Tieto hodnoty sú následne porovnané s kategóriami obrázkov a upraví váhy vo finálnej vrstve prostredníctvom procesu spätného šírenia. So zvyšovaním čísla kroku by sa mala zvyšovať aj presnosť siete. Zvyšovaním počtu krokov môžeme zvyšovať presnosť siete, až kým nenarazíme na limit modelu.

Ďalšou z bežných metód pre zvyšovanie presnosti je náhodná deformácia, orezávanie či úprava jasnosti nad tréňovanými obrázkami. Týmto zvyšujeme efektivitu siete, pretože môžeme znovu použiť rôzne variácie jedného obrázku z databázy, a tak predpokladať napríklad rôzne svetelné podmienky našich obrázkov. Nevýhodou tohto postupu je, že `bottleneck` dáta sa každým spustením skriptu menia. Je nutné ich stále regenerovať čo zaberie značný čas pri opätovnom tréňovaní navyše. Tieto zmeny nad dátami môžeme aktivovať spúšťaním retrainovacieho skriptu s nasledujúcimi parametrami `--random_crop`, `--random_scale` a `--random_brightness`. Každý z parametrov prijíma hodnotu v percentách, čo je hodnota úpravy každého z obrázkov.

Ďalšou z dôležitých nastavení pretrainovacieho skriptu, ktorú nerobí automaticky je rozdelenie množiny našich dát na tri rôzne podmnožiny. Tréningová množina je obvykle najväčšia s bežným odporúčaním 80% z celkového množstva všetkých dát. Obrázky tejto množiny sú spracovávané počas tréňovania a slúžia pre aktualizáciu váh modelu. 10% všetkých dát potom ostáva na množinu validačnú, ktorá porovnáva výsledky už počas tréningu, a zvyšných 10% slúži k reálnemu testovaniu na už pretrainovanej sieti. Tieto hodnoty môžeme upravovať znovu pomocou nasledujúcich parametrov pri spúšťaní retrainovacieho skriptu: `--testing_percentage`, `--validation_percentage`.

Kapitola 4

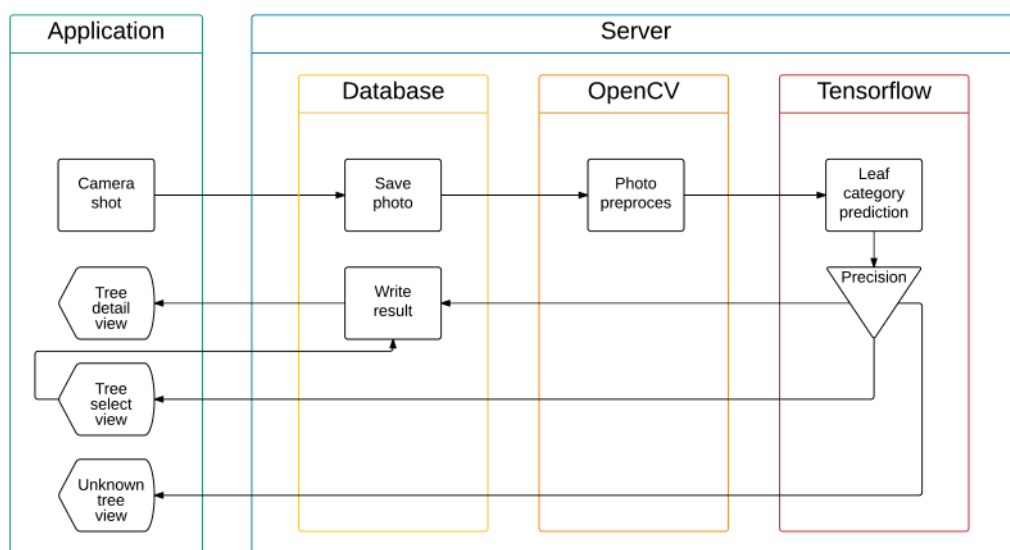
Prípadová štúdia

V tejto kapitole sa budeme venovať ukážke viacerých použitých techník na reálnom príklade vyvíjanej aplikácie. Rozhodli sme sa pre vytvorenie mobilnej aplikácie pre platformy iOS a Android, ktorá bude slúžiť na rozpoznávanie názvov stromov z fotografie ich samostatného listu. Motiváciou pre vývoj tejto myšlienky bolo umožniť ľuďom a deťom spoznávať stromy v svojom okolí hravou formou a priblížiť im tak práve prostredníctvom technológie prírodu, ktorá sa im práve vďaka nej častokrát vzdáľuje.

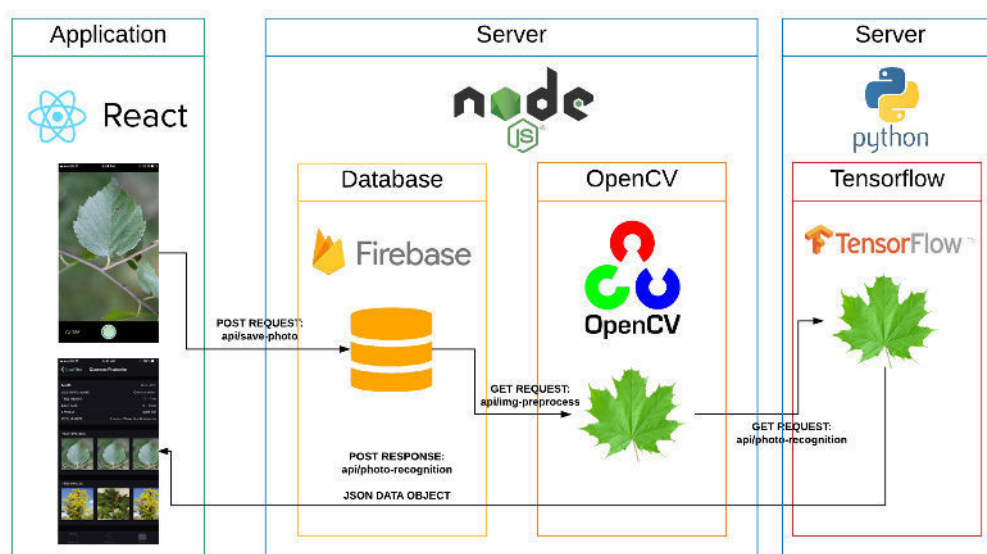
Postupne si ukážeme návrh architektúry aplikácie, niektoré techniky analýzy a spracovania obrazu pomocou knižnice OpenCV, dizajn aplikácie a tiež vybrané techniky z tvorby natívnych mobilných aplikácií v jazyku JavaScript.

4.1 Architektúra aplikácie

Celá aplikácia je rozdelená na 2 hlavné časti: Mobilná aplikácia a webový server, ktorý obsahuje časti na obsluhu databázy, prácu s OpenCV a neurónovú sieť Tensorflow. Obsluha neurónovej siete sa nachádza na druhom serveri, ktorý komunikuje s neurónovou sieťou pomocou Tensorflow API v programovacom jazyku Python, z dôvodu chýbajúceho API pre jazyk JavaScript. Práve tento kód je jediným napísaným kódom v jazyku inom ako je JavaScript.



Obr. 4.1: Architektúra aplikácie (zdroj: autor)



Obr. 4.2: Implementačný diagram aplikácie (zdroj: autor)

Mobilná aplikácia po jej spustení bude vyžadovať registráciu, resp. prihlásenie používateľa. Po vykonaní týchto krokov sa používateľ môže dostať k jej hlavnej časti fotoaparátu, kde môže zachytiť fotografiu listu, skontrolovať jej kvalitu a následne ju odoslať na spracovanie serveru. Po vrátení odpovede zo servera dostáva používateľ príslušnú obrazovku s výsledkom analýzy, kde môžu nastať až tri scenáre:

- list bol úspešne analyzovaný a používateľ sa dostáva na detail konkrétneho stromu s informáciami o ňom,
- neurónová sieť sa nevie rozhodnúť a je nutné používateľom vybrať správny strom z ponúknutých, medzi ktorými sa sieť nevie rozhodnúť na základe fotografií listov,
- list nie je v databáze a teda výsledok rozpoznávania v neurónovej sieti je nejasný.

Ďalšou časťou aplikácie je zoznam už rozpoznaných stromov z používateľových fotografií, ktorý si tak vytvára svoj osobný mobilný herbár. Používateľ môže ísť na detail každého rozpoznaneho stromu, dozvedieť sa o ňom základné informácie či si pozrieť jeho fotografie.

Tretia z hlavných častí aplikácie, dostupných po prihlásení je používateľský profil, kde má používateľ možnosť úpravy informácií, odoslania spätnej väzby či nahlásenia chyby v aplikácii.

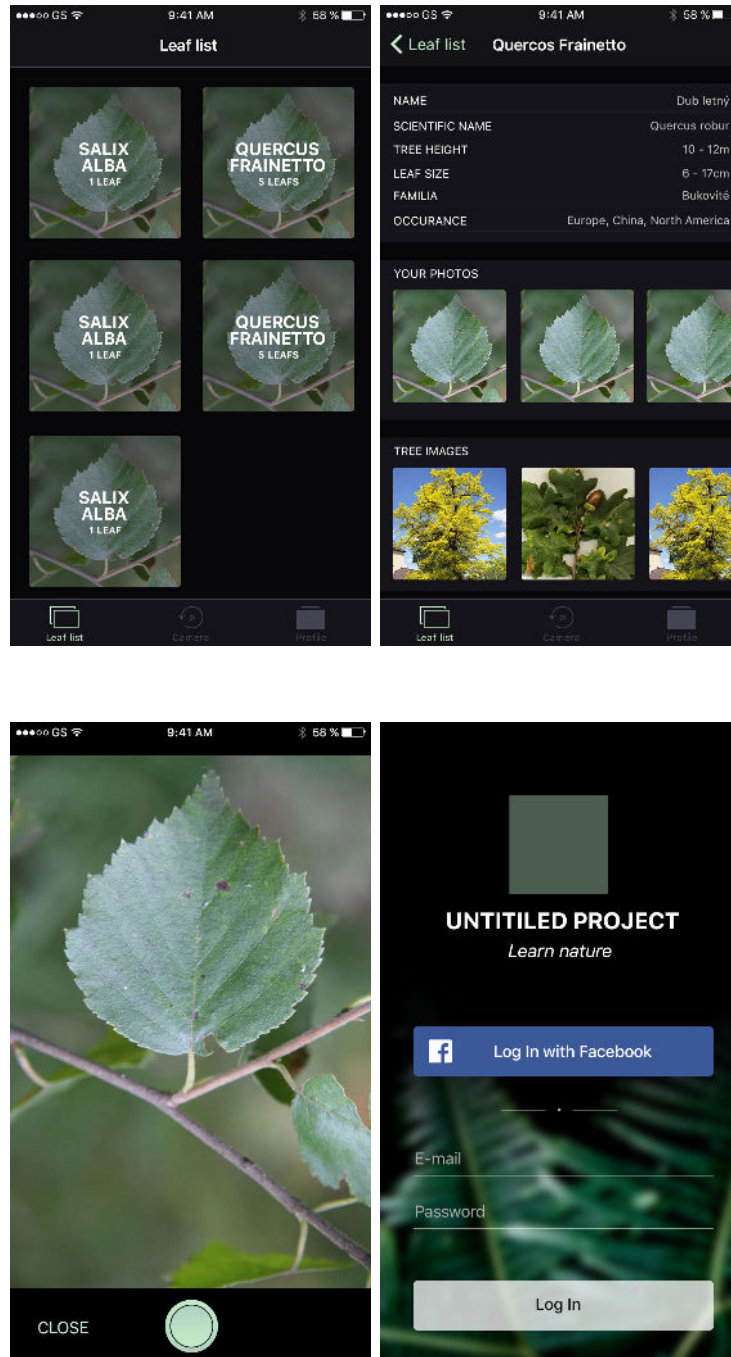
Aplikácia však ponúka do budúca širokú škálu aktualizácií a doplnkov. Napríklad štatistiky a gamifikácia aplikácie, ktorá by zahŕňala rebríčky používateľov s najväčším počtom rozpoznaných rastlín či interaktívne odznaky za plnenie rôznych cieľov: 100 úspešne analyzovaných listov, 10 analyzovaných listov z jedného stromu a podobne. Práve takéto odmeny používateľa podnecujú k častejšiemu používaniu aplikácie, vďaka čomu sa môže naučiť lepšie spoznávať stromy. Aplikácia by sa tiež mohla rozširovať o väčšiu databázu stromov a neskôr aj napríklad kvetov rastlín, čo by len zväčšilo jej možnosť využívania.

4.2 Tvorba mobilnej aplikácie

Kód našej mobilnej aplikácie je pomerne komplexný a aplikácia obsahuje mnoho rôznych obrazoviek a scenárov použitia. Celá aplikácia je implementovaná pomocou JavaScriptového frameworku React Native. Na nasledujúcich stranách tak vyberáme niekoľko jej zaujímavých častí s ukážkami kódu.

4.2.1 Dizajn mobilnej aplikácie

Pre lepšiu predstavu o používaní a dizajne aplikácie tu uvádzam štyri obrazovky z našej aplikácie - zoznam analyzovaných stromov, detail stromu, obrazovku kamery a obrazovku prihlásenia (Obr. 4.3).



Obr. 4.3: Zoznam analyzovaných stromov (vľavo hore), detail stromu (vpravo hore), obrazovka kamery (vľavo dole), prihlásenie používateľa (vpravo dole). (zdroj: autor)

4.2.2 Zachytenie fotografie a jej odoslanie na server

Naša aplikácia komunikuje so serverom cez HTTP API, ktorý zabezpečuje všetko od prihlásenia a registrácie až po našu najdôležitejšiu časť - analýzu fotografie. Server je napísaný taktiež v jazyku JavaScript, pomocou serverového rozhrania Node.js.

Ako prvé je nutné importovať si potrebné knižnice do nášho súboru *camera-snap.js*.

```
1 import React, { Component } from 'react';
2 import {
3   TouchableHighlight,
4   View
5 } from 'react-native';
6 import Camera from 'react-native-camera';
```

Následne si môžeme zdefinovať novú triedu `CameraSnap`, ktorá bude slúžiť ako náhľad a obsluha kamery v zariadení, na ktorom bude aplikácia bežať. V nasledujúcom kóde môžeme vidieť metódu `render`, ktorá má za úlohu vykreslenie používateľského rozhrania aplikácie.

```
1 export default class CameraSnap extends Component {
2   constructor(props) {
3     super(props);
4   }
5
6   render() {
7     return (
8       <View style={styles.container}>
9         <Camera
10           ref={(cam) => {
11             this.camera = cam;
12           }}
13           captureTarget={Camera.constants.CaptureTarget.temp}>
14           <View>
15             <TouchableHighlight onPress={this.takePicture} />
16           </View>
17         </Camera>
18       </View>
19     );
20   }
21 }
```


Na elemente `TouchableHighlight` môžeme vidieť poslucháč `onPress`, ktorý vždy po jeho stlačení zavolá metódu `takePicture`. Tá oslovuje priamo hardvérovú kameru zariadenia cez prepojenie napísané v jazyku C. Následne čakáme odpoveď cez Promise API JavaScriptu, ktoré sa stalo v jazyku v posledných rokoch štandardom pre spracovanie asynchrónnych operácií v posledných rokoch. Ako odpoveď z kamery získavame adresu fotografie v dočasnej pamäti zariadenia, ktorú následne opäť cez asynchrónnu operáciu `uploadImage` posielame na náš server. Po obdržaní odpovede zo servera voláme metódu `getPrediction()` pre ďalšiu prácu s dátami.

```
1  takePicture() {
2    this.camera.capture()
3      .then(data => {
4        FirebaseStorage.uploadImage(data)
5          .then(url => {
6            this.getPrediction(url);
7          })
8        .catch(error => {
9          console.log(error);
10       });
11     });
12   .catch(error => {
13     console.log(error);
14   });
15 }
```

Samotné nahranie fotografie do nášho dátového úložiska prebieha opäť ako väčšina operácií asynchrónne. Popisovaný fragment kódu patrí metóde `uploadImage`, vracajúcej novú Promise, ktorej výsledok sme zachytávali v predchádzajúcej ukážke kódu.

```
1  return new Promise((resolve, reject) => {
2    fs.readFile(uploadUri, 'base64')
3      .then((data) => Blob.build(data, { type: `${mime};BASE64` }))
4      .then((blob) => {
5        uploadBlob = blob;
6        return imageRef.put(blob, { contentType: mime });
7      })
8      .then(() => {
9        uploadBlob.close();
10       resolve(imageRef.getDownloadURL());
11     });
12 });
```

Po spracovaní metódy `uploadImage` je volaná opäť ďalšia metóda - `getPrediction()`. Pomocou nej oslovujeme ďalšou asynchrónnou operáciou server s neurónovou sieťou. V HTTP požiadavke typu POST posielame URL adresu k fotografii, ktorú máme záujem analyzovať. Po obdržaní výsledku siete voláme v aplikácii metódu `analyzePredictionData` pre spracovanie dát z neurónovej siete na úrovni aplikácie.

```
1 RNFetchBlob.fetch('POST', `${Api.tensorflow.test}/photo-prediction-  
  mock`, {  
2   'Content-Type': 'multipart/form-data'  
3 }, [{ name: 'image_data', data: url }])  
4   .then((res) => res.json())  
5   .then((res) => {  
6     this.setState({ processing: false });  
7     this.analyzePredictionData(res);  
8   })
```

4.2.3 Spracovanie dát zo servera

Metóda `analyzePredictionData` volaná po prijatí dát z predchádzajúcej kapitoly prechádza všetky výsledky a vyberá len tie kategórie, ktoré majú hodnotu predikcie väčšiu ako 80 percent.

```
1   const validResults = [];  
2   data.results.forEach(item => {  
3     if (item.value >= 0.80) {  
4       validResults.push(item);  
5     }  
6   });
```

Následne podľa počtu vybraných položiek v poli `validResults` vykonávame správnu akciu. Pokiaľ aplikácia zanalyzovala našu fotografiu ideálne a vo výsledkoch máme len jednu položku, vytvoríme nový objekt s detailami o fotografii ako dátum, identifikátor, url adresa a GPS súradnice. Následne objekt zapíšeme do databázy používateľa, k správnej kategórii. Po úspešnom zápise voláme funkciu `Database.getTreeDetail()` pre zobrazenie detailu stromu, ktorého list sme zachytili na fotografii.

```

1 let userLeafRef = 'user/${this.state.user.uid}/trees/${validResults
    [0].id}';
2 firebase.database().ref(userLeafRef).once('value').then((snapshot)
    => {
3   let leafData = snapshot.val();
4   let date = new Date();
5   let newPhoto = {
6     id: date.valueOf(),
7     url: this.state.uploadedURL,
8     date: date.toString(),
9     gps: false
10  }
11  leafData.photos.unshift(newPhoto);
12
13  firebase.database().ref(userLeafRef).set(leafData).then(() => {
14    Database.getTreeDetail(validResults[0].id, (tree) => {
15      Actions.detail({ tree: tree, title: tree.name });
16    });
17  });
18 });

```

Ak máme v poli `validResults` viac ako jednu položku, používateľovi zobrazíme možnosť ručného priradenia jeho zachytenej fotografie k vzorkám listov stromov, ktoré neurónová sieť vyhodnotila ako podobné, respektíve ich priradila ku kategóriám s hodnotou väčšou ako 80 percent. Zápis do databázy v tejto chvíli neprebíha. Ten sa vykoná až na základe používateľovej akcie - správneho priradenia fotografie ku kategórii.

```

1 Actions.detail({ trees: validResults });

```

V prípade, že neurónová sieť nedokázala priradiť fotografiu ku niektorej z kategórii s hodnotou nad 80 percent, používateľovi posielame obrazovku so správou o neúspešnej analýze.

```

1 Actions.treeAnalyzeUnsuccessful({ img_url: url });

```

4.3 Spracovanie obrazu

Jednou zo základných stavebných častí nášho systému je analýza obrazu pomocou knižnice OpenCV, ktorá má za úlohu predspracovanie zachytenej fotografie a základné rozpoznanie typu a kategórie listu.

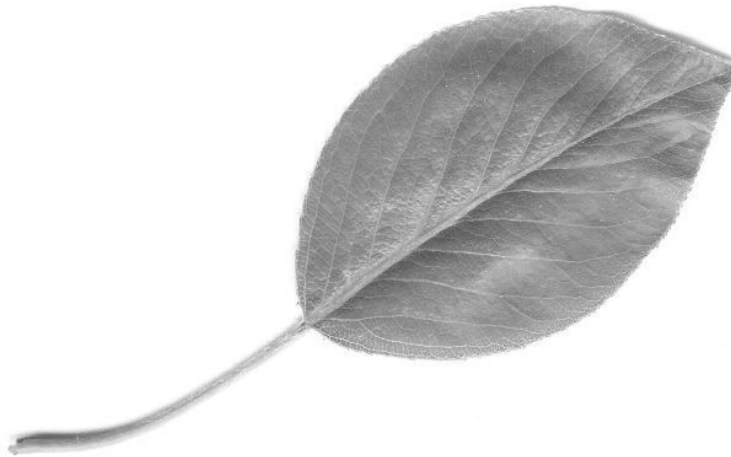
4.3.1 Vyhľadávanie hrán

Na vyhľadávanie hrán môžeme použiť Canny Edge Detector algoritmus, ktorý bol vyvinutý v roku 1986 a stal sa veľmi obľúbeným. Často sa nazýva aj optimálnym detektorom. Tento algoritmus spĺňa nasledujúce 3 kritéria, ktoré sú dôležité pri hľadaní hrán:

1. Detekcia hrany s nízkou mierou chýb,
2. Správna lokalizácia hrany - hrana by mala byť lokalizovaná presne v jej strede,
3. Zobrazenie každej hrany len raz.

Obrázok môžeme skonvertovať do farebného formátu odtieňov šedej, čo môže niekedy prinášať lepšie výsledky. Následne pre zjemnenie obrázku a odstránenie zbytočného šumu použijeme *Gaussian Blur filter*, ktorý prijíma ako argument pole *Gaussian kernel size*. Použitie týchto dvoch metód nám môže priniesť lepšie a presnejšie výsledky (Obr. 4.4).

```
1 im.convertGrayscale();  
2 im.gaussianBlur([3, 3]);
```



Obr. 4.4: Obrázok po použití metód `Matrix.convertGrayscale()` a `Matrix.gaussianBlur()` (zdroj: autor)

Obrázok je teraz pripravený na vyhľadávanie hrán pomocou Canny Edge algoritmu. Tento algoritmus prijíma ako vstupné hodnoty `lowThreshold` a `highThreshold` parametre. Dva prahy umožňujú rozdeliť pixely hrán do troch skupín. Ak je hodnota gradientu pixelu vyššia ako hodnota vysokého prahu, pixely sú označené ako silne okrajové pixely. Ak je hodnota gradientu medzi vysokým a nízkym prahom, pixely označíme ako slabé okrajové pixely. A ak je hodnota nižšia ako hodnota nízkeho prahu, tieto pixely úplne potlačíme. Neexistuje nič ako globálne nastavenie prahov pre každý obrázok. Je potrebné ich správne nastaviť pre konkrétny obrázok zvlášť.

Po volaní Canny Edge algoritmu nad obrázkom zavoláme tiež metódu dilatácie, ktorej fungovanie sme si už vysvetlili.

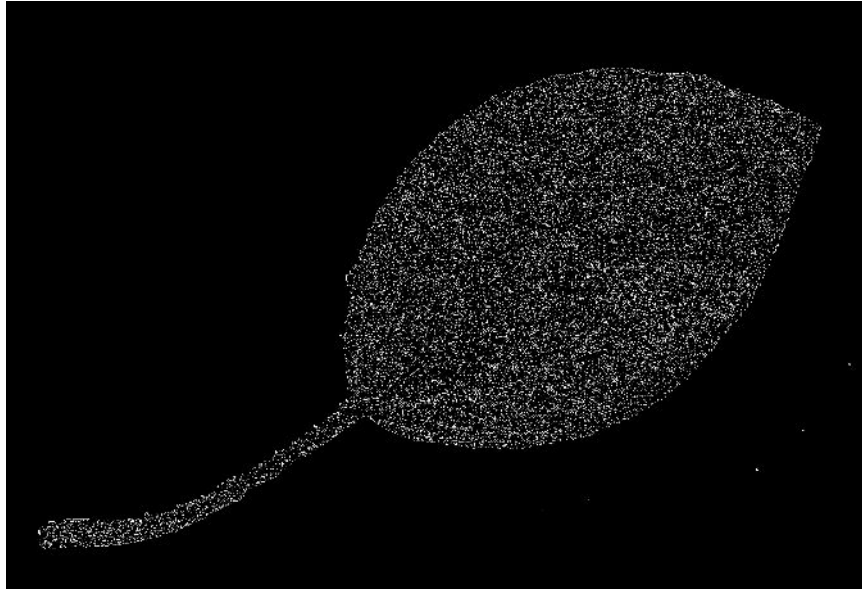
```
1 const lowThresh = 0;
2 const highThresh = 150;
3 const iterations = 2;
4
5 img.canny(lowThresh, highThresh);
6 img.dilate(iterations);
```

Po týchto krokoch máme obrázok, ktorý prešiel Canny Edge analýzou. Z tohto obrázku si teraz môžeme vybrať všetky kontúry volaním metódy `Matrix.findContours()` a následne ich vykresliť do nového obrázku (Obr. 4.5).

```

1 const WHITE = [255, 255, 255];
2 let contours = img.findContours();
3 let allContoursImg = img.drawAllContours(contours, WHITE);
4 allContoursImg.save('./img/allContoursImg.jpg');

```



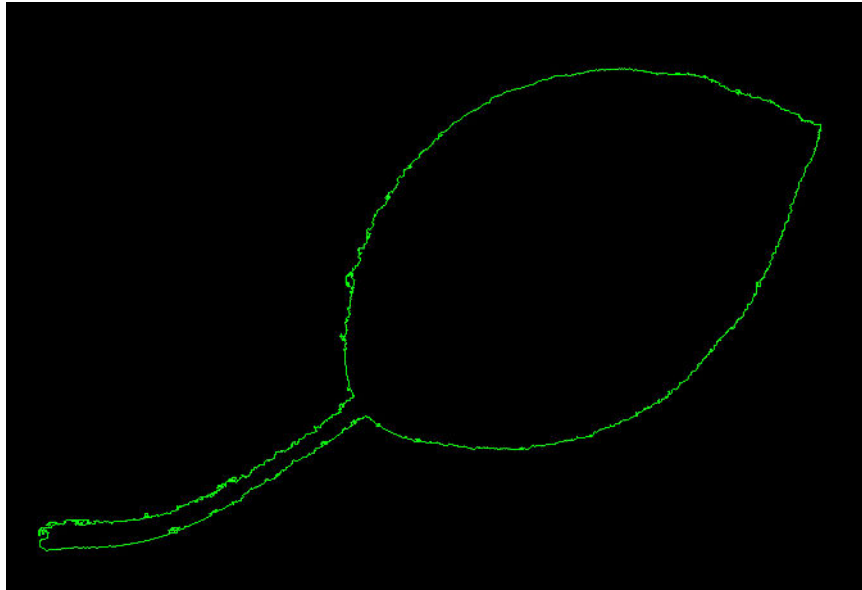
Obr. 4.5: Všetky nájdené hrany na obrázku po Canny Edge analýze (zdroj: autor)

Na tomto obrázku vidíme všetky kontúry, ktoré Canny Edge detector našiel. My však chceme len jednu, a to tú najväčšiu z nich. To docielime nasledujúcim kódom, v ktorom prejdeme každú z kontúr a uložíme si tú najväčšiu. Následne ju vykreslíme do novej premennej pomocou metódy `Matrix.drawContour()`, ktorá prijíma niekoľko parametrov a obrázok uložíme (Obr. 4.6).

```

1 const WHITE = [255, 255, 255];
2 let contours = img.contours();
3 let largestContourImg;
4 let largestArea = 0;
5 let largestAreaIndex;
6
7 for (let i = 0; i < contours.size(); i++) {
8   if (contours.area(i) > largestArea) {
9     largestArea = contours.area(i);
10    largestAreaIndex = i;
11  }
12 }
13
14 largestContourImg.drawContour(contours, largestAreaIndex, WHITE,
    thickness, lineType);

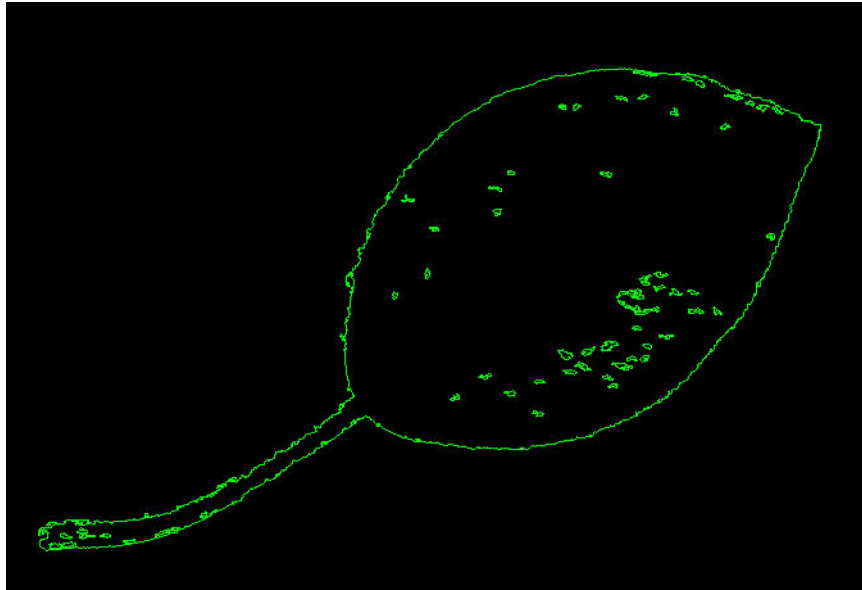
```



Obr. 4.6: Obrázok s najväčšou nájdenou hranou (zdroj: autor)

Pokiaľ by sme chceli vykresliť viac kontúr, ktoré sú napríklad väčšie ako nejaká hodnota $\langle \text{Integer} \rangle$, metódu `Matrix.drawContour()` len presunieme do for cyklu a upravíme podmienku. Výsledok tejto operácie môžeme vidieť na obrázku 4.7.

```
1  const WHITE = [255, 255, 255];
2  let contours = img.contours();
3  let largestContourImg;
4  let largestArea = 500;
5  let largestAreaIndex;
6
7  for (let i = 0; i < contours.size(); i++) {
8    if (contours.area(i) > largestArea) {
9      largestContourImg.drawContour(contours, i, WHITE, thickness,
10                                   lineHeight);
11    }
12  }
13  largestContourImg.drawContour(contours, largestAreaIndex, WHITE,
14                                thickness, lineHeight);
```

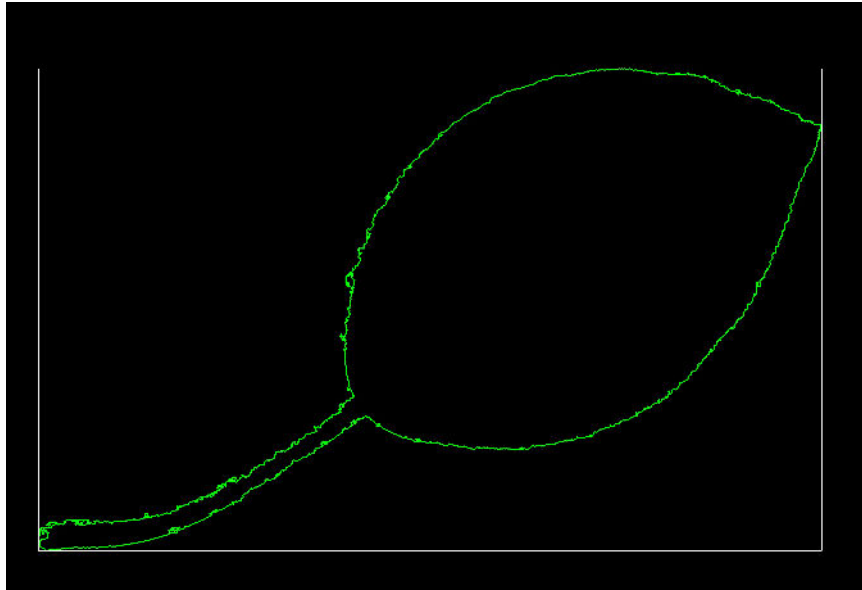


Obr. 4.7: Obrázok s vybranými kontúrami (zdroj: autor)

4.3.2 Aproximácia polygónom

Aproximáciu polygónom môžeme využiť na viacero vecí, tou najtriviálnejšou je opísanie objektu ohraničujúcim obdĺžnikom pomocou metódy `Contours.boundingRect(index)` (Obr. 4.8). Túto metódu voláme na objekte `Contours`, ktorý sme získali zavolaním `Matrix.findContours()` metódy na obrázku, po Canny Edge detekcii v predchádzajúcom príklade.

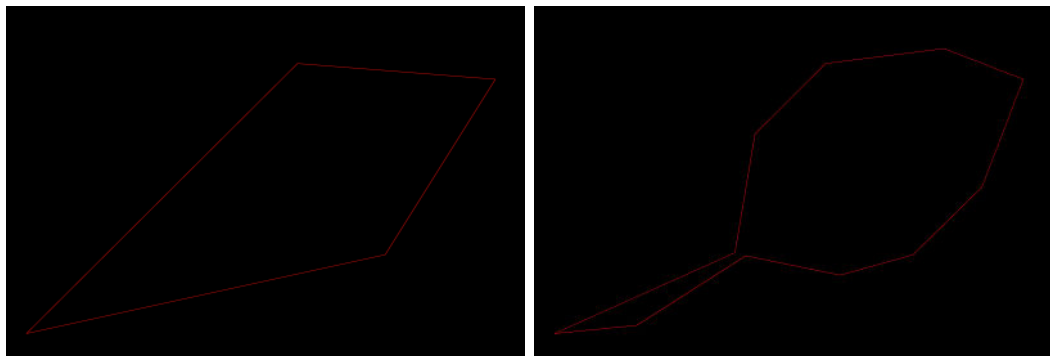
```
1 let bound = contours.boundingRect(largestAreaIndex);  
2 largestContourImg.rectangle([bound.x, bound.y], [bound.width, bound  
   .height], WHITE, 2);
```

Obr. 4.8: Aproximácia obdĺžnikom (zdroj: autor)

Druhou možnosťou je aproximácia polygónom so špecifikovanou presnosťou. Na dvojici výsledných obrázkov 4.9 môžeme vidieť aproximácie s dvomi úrovňami presnosti. Pomocou `Contours.cornerCount(index)` zavolanou nad modifikovanými kontúrami dostávame počet uhlov polygónu, s čím môžeme neskôr pracovať.

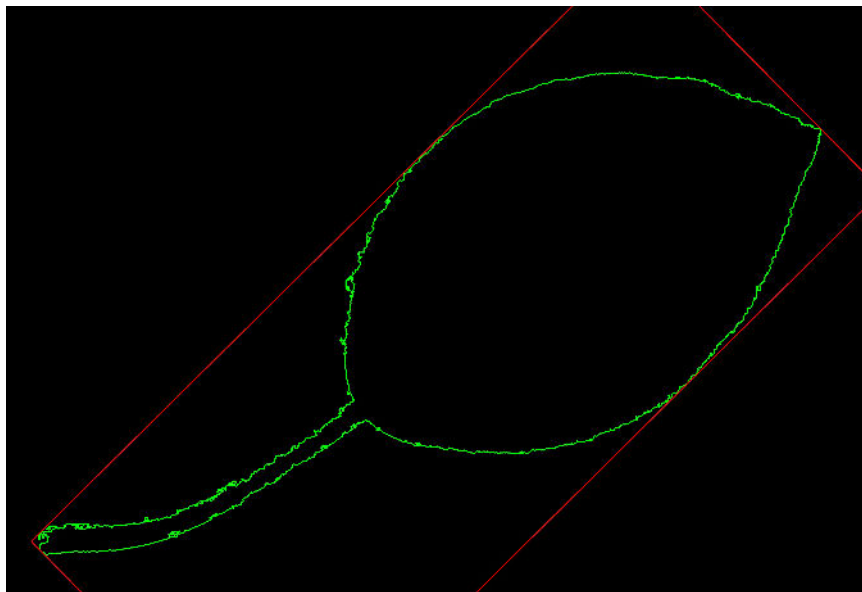
```
1 let poly;  
2 let RED = [0, 0, 255];  
3 let arcLength = contours.arcLength(largestAreaIndex, true);  
4 contours.approxPolyDP(largestAreaIndex, arcLength * 0.05, true);  
5 poly.drawContour(contours, largestAreaIndex, RED);  
6  
7 // number of corners  
8 console.log(contours.cornerCount(largestAreaIndex));
```



Obr. 4.9: Aproximácia polygónom so špecifikovanou presnosťou (zdroj: autor)

Zaujímavé je tiež použitie aproximácie rotovaním polygónom pomocou metódy `Contours.minAreaRect()` (Obr. 4.10). Túto metódu používame v našom projekte na zistenie uhlu konkrétneho objektu a jeho správnu rotáciu do vodorovnej polohy, keďže nám okrem bodov vracia aj uhol tohto polygónu. V nasledujúcom príklade si do obrázku v premennej `largestContourImg` pridáme rotovaný polygón a vypíšeme uhol tohto polygónu.

```
1 let rect = contours.minAreaRect(largestAreaIndex);
2 for (let i = 0; i < 4; i++) {
3     largestContourImg.line([rect.points[i].x, rect.points[i].y], [
4         rect.points[(i+1)%4].x, rect.points[(i+1)%4].y], RED, 3);
5 }
6 // uhol polygonu
7 console.log(rect.angle);
```



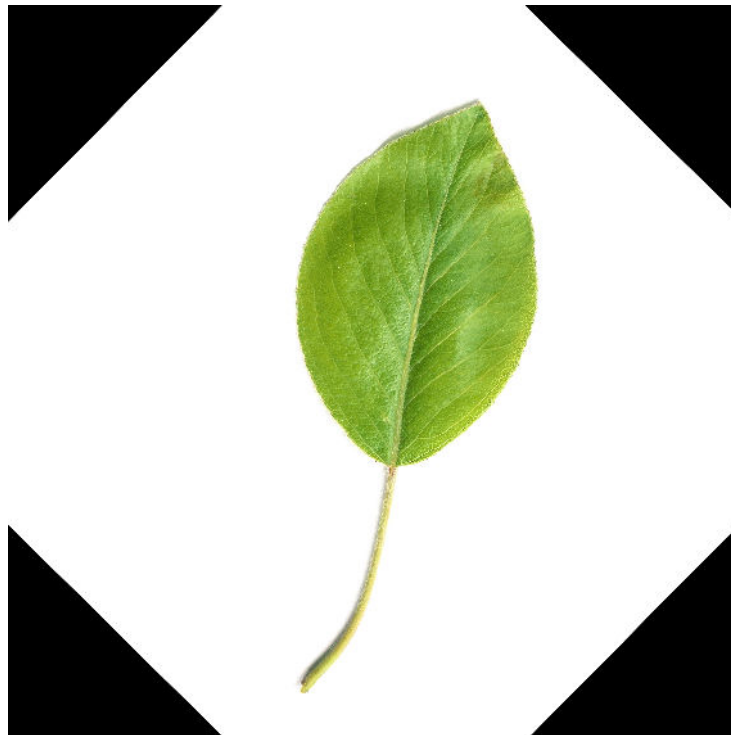
Obr. 4.10: Aproximácia rotovaným polygónom (zdroj: autor)

4.3.3 Rotácia obrázku bez orezania

Jedna z vecí, ktorú sme potrebovali vyriešiť a OpenCV ju v sebe nemá je rotácia obrázku bez jeho orezania. Pred samotnou rotáciou si vytvoríme novú štvorcovú 8-bit 3 kanálovú maticu `Matrix` s názvom `bgImg` vo veľkosti diagonály obrázku, ktorý chceme rotovať. Následne vypočítame pozíciu pre obrázok, ktorý budeme do našej novej matice vkladať. Nad celým `bgImg` zavoláme požadovanú rotáciu a obrázok môžeme uložiť (Obr.

4.11).

```
1 let rect = contours.minAreaRect(largestAreaIndex);
2 let diagonal = Math.round(Math.sqrt(Math.pow(im.size()[1], 2) +
    Math.pow(im.size()[0], 2)));
3 let bgImg = new cv.Matrix(diagonal, diagonal, cv.Constants.CV_8UC3,
    [255, 255, 255]);
4 let offsetX = (diagonal - im.size()[1]) / 2;
5 let offsetY = (diagonal - im.size()[0]) / 2;
6
7 IMG_ORIGINAL.copyTo(bgImg, offsetX, offsetY);
8 bgImg.rotate(rect.angle + 90);
9
10 bgImg.save('./img/rotatedImg.jpg');
```



Obr. 4.11: Obrázok po správnej rotácii (zdroj: autor)

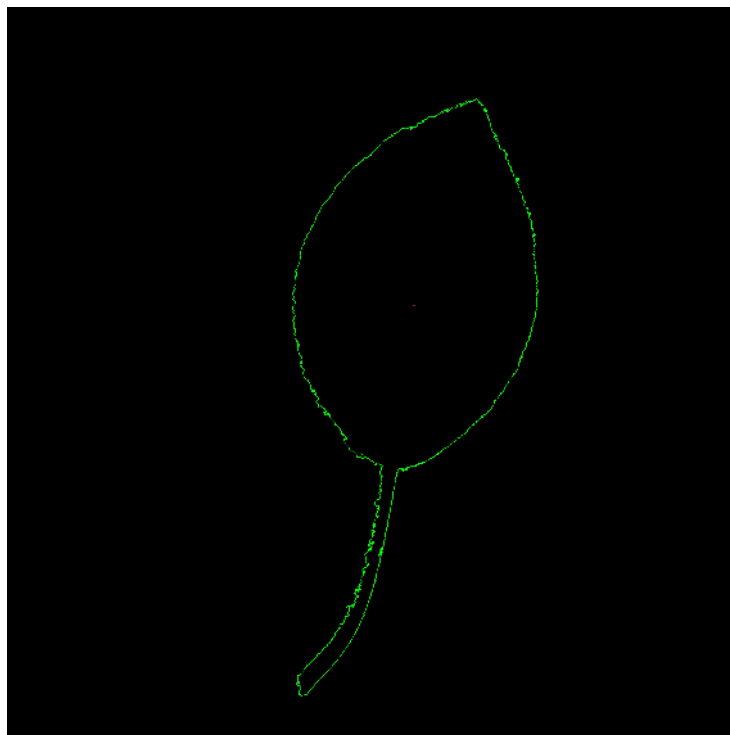
Následne môžeme nad rotovaným obrázkom spustiť Canny Edge detektor pre získanie kontúr obrázku tak, ako sme si to už ukázali.

```
1 const GREEN = [0, 255, 0];;
2 let rotatedContour = new cv.Matrix(diagonal, diagonal);
3 bgImg.canny(lowThresh, highThresh);
4 bgImg.dilate(nIters);
5 let contours = bgImg.findContours();
```

```

6
7 for (let i = 0; i < contours.size(); i++) {
8   if (contours.area(i) > largestArea) {
9     largestArea = contours.area(i);
10    largestAreaIndex = i;
11  }
12 }
13
14 rotatedContour.drawContour(contours, largestAreaIndex, GREEN,
15   thickness, lineType);
15 rotatedContour.save('./img/rotatedImgContour.jpg');

```



Obr. 4.12: Obrázok po správnej rotácii a Canny Edge vyhľadávani hrán (zdroj: autor)

Po spustení Canny Edge detektora dostávame požadovaný výsledok (Obr. 4.12). Je veľmi veľa ďalších metód, ktoré môžeme nad obrázkom spúšťať, ako odstránenie pozadia a podobné úpravy, ktoré sa nám tiež môžu veľmi hodiť. Nie je však možné písať o všetkých v rozsahu tejto práce.

4.3.4 Porovnanie tvarov pomocou Fourierových deskriptorov

Po už opísaných úpravách obrázku v predchádzajúcich kapitolách máme k dispozícii nájdenú hranu okolo celého listu. Na základe týchto dát sme chceli priradiť konkrétne listy ku ich správnym kategóriám pomocou Fourierových deskriptorov. „Tie sú vyjadrením časovo závislých signálov pomocou harmonických signálov, teda matematických funkcií *sin* a *cos*. Fourierova transformácia je definovaná integrálnym vzťahom:” [4]

$$S(\omega) = \int_{-\infty}^{\infty} s(t)e^{-i\omega t} dt$$

Kde funkciu $s(t)$ vypočítame pomocou inverznej Fourierovej transformácie nasledujúcim spôsobom:

$$s(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} S(\omega)e^{i\omega t} d\omega$$

Náš zámer rozpoznávania kategórie listu touto metódou sme testovali pomocou knižnice OpenCV. Kvôli chýbajúcej implementácii Fourierovej transformácie v JavaScriptovej knižnici a lepšej práci s maticami sme boli nútení použiť jazyk Python. Po vykonaní Fourierovej transformácie pomocou príkazu `np.fft.fft(ks)` je potrebné vykonať normalizáciu nad týmito dátami. Konkrétne nastavenie DC komponenty deskriptora na hodnotu 0, vydelenie všetkých deskriptorov magnitúdou druhého deskriptora a následná normalizácia štartovacieho bodu - odčítanie fázy druhého deskriptora od fázy každého deskriptora a nastavenie ich váhy.

```
1 Fu2[0] = complex(0, 0)
2 Fur = []
3 for k in range(0, len(Fu2)):
4     if k == 0:
5         Fur.append(Fu2[k])
6     else:
7         Fur.append(Fu2[k] / abs(Fu2[1]))
8 f1 = math.atan(Fur[1].imag / Fur[1].real)
9 for k in range(0, len(Fur)):
```

```
10 Fur[k] = Fur[k]*cmath.exp(-j * f1 * k)
```

Po následnom vykreslení a porovnaní dát z diametrálne odlišných listov sme dospeli k záveru, že pomocou tejto metódy nie je možné rozpoznávať kategóriu listu ani s približnou presnosťou. Na základe týchto zistení sme sa rozhodli pre použitie neurónovej siete na určenie presného typu analyzovaného listu z fotografie.

4.4 Rozpoznávanie obrazu pomocou neurónovej siete

Po tom, ako máme sieť natrénovanú z našich dát s ňou potrebujeme komunikovať cez webový protokol HTTP. Aby to bolo možné, potrebujeme naprogramovať kód pre klasifikáciu obrázkov pomocou dostupného oficiálneho Tensorflow API, ktoré je dostupné pre jazyky Python a C++. V našom príklade sme zvolili jazyk Python. Následne naprogramujeme HTTP serverové API na komunikáciu medzi aplikáciou a neurónovou sieťou.

4.4.1 Práca s neurónovou sieťou v jazyku Python

Ako prvý potrebujeme spraviť import potrebných knižníc, konkrétne Tensorflow na prácu s neurónovou sieťou, Numpy na pokročilú prácu s matematikou, Flask ako HTTP microframework a systémové knižnice OS a URLLIB.

```
1 import os
2 import numpy as np
3 from six.moves import urllib
4 import tensorflow as tf
5 from flask import Flask, jsonify, render_template, request
```

Následne si zdefinujeme konštanty s cestou k našim natrénovaným dátam: modelovému grafu a popisom so správnou absolútnou cestou na serveri vďaka *os.getcwd()* metóde.

```
1 pwd = os.getcwd()
2 MODEL_PATH = pwd + '/data/output_graph.pb'
```

```
3 LABELS_PATH = pwd + '/data/output_labels.txt'
```

Súbor s grafovými dátami `output_graph.pb` je ich textovou reprezentáciou a pre prácu s nimi je potrebné ich previesť do formátu, ktorému rozumie Tensorflow. To urobíme pomocou nasledujúcej funkcie, v ktorej si načítame náš súbor, získame dáta z textového retazca a importujeme ich do našej siete.

```
1 def create_graph():
2     with tf.gfile.FastGFile(modelFullPath, 'rb') as f:
3         graph_def = tf.GraphDef()
4         graph_def.ParseFromString(f.read())
5         _ = tf.import_graph_def(graph_def, name='')
```

Hlavnou funkciou, ktorú neskôr budeme volať priamo po HTTP POST požiadavke je `run_inference_on_image`, ktorá prijíma ako jediný parameter `image_url`, čo je url adresa k fotografii, ktorú chceme analyzovať. Hneď na jej začiatku tento obrázok načítame do pamäte pre neskoršiu prácu s ním, a voláme už predstavenú funkciu `create_graph()`.

```
1 req = urllib.request.Request(image_url)
2 response = urllib.request.urlopen(req)
3 image_data = response.read()
4 create_graph()
```

V tejto chvíli môžeme prejsť k spusteniu siete s naším načítaným grafom. Z neho získame pomocou funkcie `get_tensor_by_name()` konkrétne dáta, s ktorými vie sieť pracovať. Tieto dáta sa nazývajú tensor a môžeme si ich predstaviť ako n-dimenzionálny zoznam, respektíve pole. Nasleduje získanie predikcií volaním funkcie `sess.run()` s grafovými dátami a načítaným obrázkom.

```
1 softmax_tensor = sess.graph.get_tensor_by_name('final_result:0')
2 predictions = sess.run(softmax_tensor, {'DecodeJpeg/contents:0':
3     image_data})
3 predictions = np.squeeze(predictions)
```

Keď už máme k dispozícii výsledky s našimi predikciami, vykonáme niekoľko operácií - zoradenie výsledkov, priradenie ich k ich typom zo súboru `output_labels.txt` podľa

identifikátora, vybrané piatich kategórií s najväčšou percentuálnou hodnotou predikcie a vrátenie z funkcie vo forme viacrozmerného poľa. V tomto poli obsahuje každá položka záznam s názvom kategórie a jej percentuálnu hodnotu predikcie.

```
1 top_k = predictions.argsort()[-5:][::-1]
2 f = open(LABELS_PATH, 'rb')
3 lines = f.readlines()
4 labels = [str(w).replace("\n", "") for w in lines]
5 results = []
6
7 for node_id in top_k:
8     human_string = labels[node_id]
9     score = predictions[node_id]
10    results.append([human_string, score])
11
12 return results
```

4.4.2 Tvorba HTTP rozhrania pre prístup k neurónovej sieti

Na to, aby sme mohli z našej mobilnej aplikácie osloviť neurónovú sieť a získať výsledky jej analýzy, potrebujeme pripraviť HTTP REST API v rovnakom jazyku ako je obsluha siete, teda v programovacom jazyku Python a webovom frameworku Flask, ktorý je populárny vo svete kvôli svojej minimalistickosti. Na začiatku si vytvoríme novú inštanciu aplikácie pomocou funkcie `Flask()` a následne ju spustíme ako `app.run()`.

```
1 app = Flask(__name__)
2 if __name__ == '__main__':
3     app.run()
```

Po tomto kroku sa už môžeme pustiť do vytvorenia HTTP cesty, ktorú budeme volať v našej aplikácii. Zdefinujeme si tak novú url adresu `/api/photo-prediction`, ktorá bude poslúchať na HTTP POST požiadavku. Po jej zachytení si z požiadavky načítame URL adresu obrázku, ktorú používateľ odoslal k analýze a volaním už popísanej funkcie `run_inference_on_image()` získame jej výsledok. Ten následne pretransformujeme do formátu JSON, ktorý bude prijímať naša aplikácia a pracovať s ním ďalej. Vyberieme tak 5 najpopulárnejších výsledkov, ktoré nám vrátila neurónová sieť a získavame ich

názov a hodnotu predikcie. Tie posielame v odpovedi našej aplikácie ako pole objektov.

```
1 @app.route('/api/photo-prediction', methods=['POST'])
2 def photoPrediction():
3     print(request.form['image_data'])
4     answer = run_inference_on_image(request.form['image_data'])
5
6     list = [
7         {'id': getNormalizedString(answer[0][0]), 'value':
8           getNormalizedNumber(answer[0][1])},
9         {'id': getNormalizedString(answer[1][0]), 'value':
10          getNormalizedNumber(answer[1][1])},
11        {'id': getNormalizedString(answer[2][0]), 'value':
12          getNormalizedNumber(answer[2][1])},
13        {'id': getNormalizedString(answer[3][0]), 'value':
14          getNormalizedNumber(answer[3][1])},
15        {'id': getNormalizedString(answer[4][0]), 'value':
16          getNormalizedNumber(answer[4][1])},
17    ]
18    return jsonify(status='OK', results=list)
```

Odpoveď nášho servera s JSON dátami môže vyzeráť nasledovne.

```
1 {
2   "results": [
3     { "id": "prunus-serrulata", "value": 0.99 },
4     { "id": "salix-urbalix", "value": 0.21 },
5     { "id": "acer-campreste", "value": 0.08 },
6     { "id": "larix-decidua", "value": 0.03 },
7     { "id": "populus-balsamifera", "value": 0.01 }
8   ],
9   "status": "OK"
10 }
```

Pre zrýchlenie vývoja aplikácie a odľahčenie servera od neustálej analýzy dát pri vývoji sme si tiež vytvorili podobnú HTTP URL adresu `/api/photo-prediction-mock`, ktorá vyzerá rovnako ako predchádzajúca s absenciou volania `run_inference_on_image` metódy a nahradenie čítania výsledkov s našimi ručne vygenerovanými vzorovými dátami.

Záver

Táto práca predstavila možnosti tvorby natívnych mobilných aplikácií v jazyku JavaScript pomocou knižnice React Native, ako aj techniky spracovania obrazu v tomto jazyku pomocou knižnice OpenCV. Jej značná časť sa tiež venuje technikám rozpoznávania obrazu pomocou neurónovej siete Tensorflow. Vytvorili sme funkčnú mobilnú aplikáciu slúžiacu na rozpoznávanie stromov z fotografie ich listu.

Spracovanie obrazu nebolo možné vykonávať priamo na mobilnom zariadení kvôli chýbajúcej implementácii knižnice OpenCV pre framework React Native. Výpočtovú logiku sme tak presunuli na webový server, napísaný taktiež v jazyku JavaScript. Pri analýze obrazu sme však narazili na limity pri používaní metód Fourierových deskriptorov, ktoré nám neumožňovali kategorizáciu listov s dostatočnou presnosťou. Na základe týchto zistení sme sa rozhodli pre použitie neurónovej siete Tensorflow, ktorá nám po natrénovaní umožňuje správnu kategorizáciu listov stromov. Obsluha neurónovej siete, ako aj jej webový server je naprogramovaný v jazyku Python, kvôli chýbajúcej implementácii v jazyku JavaScript.

Ďalším cieľom práce je ďalej získavať čo najväčšiu databázu listov z rôznych stromov a rozširovať tak možnosti jej poznania. Po dostatočnom rozšírení databázy z oblasti stromov je možné aplikáciu ponúknuť používateľom prostredníctvom obchodov Apple AppStore a Google Play. Ďalej je v aplikácii veľký priestor pre pridávanie novej funkcionality, ako napríklad zapojenie gamifikácie a motivácie používateľov pre používanie aplikácie, ktorá ich tak vzdeláva v ich poznaní jednej časti prírody. Aplikácia sa tiež neskôr môže rozvíjať do ďalších oblastí, ako je rozpoznávanie kvetov či bylín.

Zoznam bibliografických odkazov

- [1] CANTELON, M.; HARTER, M.; HOLOWAYCHUCK, T. J.: *Node.js in Action*. Manning Publications Co., 2014, ISBN 978-1-617-29057-2, 690 s.
- [2] CROCKFORD, D.: *JavaScript: The Good Parts*. O'Reilly Media, 2008, ISBN 978-0-596-51774-8, 311 s.
- [3] EISENMAN, B.: *Learning React Native*. O'Reilly Media, 2015, ISBN 978-1-4919-2900-1, 272 s.
- [4] GONZALEZ, R. C.; WOODS, R. E.: *Digital Image Processing*. Pearson Education, 2008, ISBN 0-13-168728-2, 943 s.
- [5] GOOGLE: *Tensorflow - oficiálna dokumentácia*. [online]. 2017, [cit. 2017.04.20]. Dostupné na internete:, <https://www.tensorflow.org/api_docs/>.
- [6] GURNEY, K.: *An Introduction to Neural Networks*. CRC Press, 1996, ISBN 1857285034, 234 s.
- [7] LINDLEY, C.: *JavaScript Enlightenment*. O'Reilly Media, 2013, ISBN 978-1-449-34288-3, 166 s.
- [8] LUTZ, M.: *Learning Python*. O'Reilly Media, 2013, ISBN 978-1-4493-5573-9, 1648 s.
- [9] MARTIN, R. C.: *Čistý kód*. Computer press, a.s., 2009, ISBN 978-80-251-2285-3, 423 s.
- [10] NAHUM, D.: *Programming React Native*. O'Reilly Media, 2016, ISBN 978-1-4919-3200-2, 172 s.
- [11] Node.js: *oficiálna dokumentácia*. [online]. 2017, [cit. 2017.04.20]. Dostupné na internete:, <<https://nodejs.org/en/docs/>>.

- [12] OpenCV: *oficiálna dokumentácia*. [online]. 2017, [cit. 2017.04.20]. Dostupné na internete:, <<http://docs.opencv.org/2.4/>>.
- [13] OSMANI, A.: *Learning JavaScript Design Patterns*. O'Reilly Media, 2012, ISBN 978-1-449-33181-8, 254 s.
- [14] POLÁK, J.; MERUNKA, V.; CARDA, A.: *Umění systémového návrhu*. Grada Praga, 2003, ISBN 80-247-0424-02, 196 s.
- [15] ZAKAS, N. C.: *JavaScript pro webové vývojáře*. Computer press, a.s., 2009, ISBN 978-80-251-2509-0, 832 s.
- [16] ZAKAS, N. C.: *Maintainable JavaScript*. O'Reilly Media, 2012, ISBN 978-1-449-32768-2, 242 s.
- [17] ŠIKUDO VÁ, E.; ČERNEKOVÁ, Z.; BENEŠOVÁ, W.; aj.: *Počítačové videnie*. Wikina, 2011, ISBN 978-80-87925-06-5, 397 s.

Zoznam príloh

Príloha A: Systémová dokumentácia

Príloha B: Používateľská príručka

Príloha C: Sprievodné CD so zdrojovým kódom

Príloha A: Systémová dokumentácia

**UNIVERZITA MATEJA BELA V BANSKEJ BYSTRICI
FAKULTA PRÍRODNÝCH VIED**

**ANALÝZA A SPRACOVANIE OBRAZU V
MOBILNÝCH APLIKÁCIÁCH TVORENÝCH
POMOCOU WEBOVÝCH TECHNOLOGIÍ**

Systémová dokumentácia

Mobilná aplikácia

Systémové požiadavky, inštalácia a vývoj

Pre vývoj aplikácie je potrebné mať na počítači nainštalovaný Node.js verzie 4 a vyšší. Takisto pre potreby publikovania, resp. testovania na zariadeniach je potrebné mať nainštalované vývojárske programy Android Studio a Apple Xcode, ktoré nie sú potrebné pre vývoj, ale pre podpisovanie certifikátov. Nainštalovanie všetkých potrebných závislostí môžeme na operačnom systéme macOS vykonať spustením nasledujúcich príkazov v termináli.

```
brew install node
brew install watchman
npm install -g react-native-cli
npm install
```

Pre spustenie emulátora a začiatok vývoja aplikácie môžeme použiť jeden z nasledujúcich príkazov v závislosti od platformy, na ktorej chceme aplikáciu spúšťať.

```
react-native run-ios
react-native run-android
```

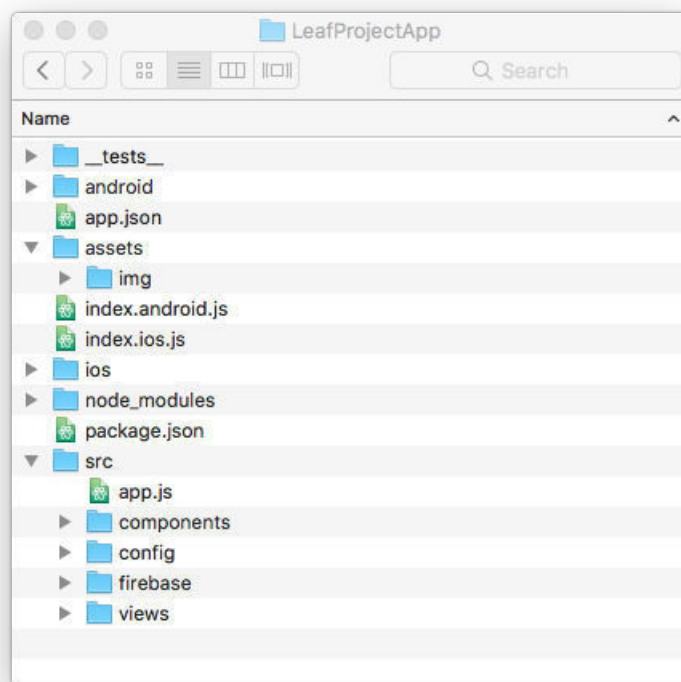
Pre produkčný build aplikácie je nutné zmeniť schému aplikácie na produkčnú a následne aplikáciu vybuildovať v prostredí Android Studio, respektíve Apple Xcode.

```
react-native run-ios --configuration Release
react-native run-android --variant=release
```

Adresárová štruktúra

Štruktúru aplikácie môžeme vidieť na obrázku nižšie. Aplikácia obsahuje zložku `__tests__`, ktorá obsahuje integračné testy. Zložky `android` a `ios` obsahujú natívne projekty pre tieto platformy, ktoré boli automaticky vytvorené pri spúšťaní príkazov *React Native*

v termináli. Zložka *assets* obsahuje statické súbory používané v aplikácii, ako napríklad *obrázky*. Súbory *index.android.js* a *index.ios.js* sú základnými súbormi, ktoré sú potrebné pre *build* aplikácie pre konkrétnu platformu. Zložka *node_modules* obsahuje stiahnuté balíčky potrebné pre vývoj a spustenie aplikácie a súbor *package.json* obsahuje zas definíciu verzií týchto balíčkov. Najzaujímavejšou je zložka *src*, ktorá v sebe obsahuje základ aplikácie v podobe súboru *app.js*. Tento súbor obsahuje najmä obsluhu zobrazovania konkrétnych obrazoviek a ich hierarchiu. Zložka *components* obsahuje malé komponenty používané vo vnútri celých obrazoviek, ktoré sú definované vo vnútri zložky *views*. Konfiguračné súbory sa nachádzajú v zložke *config* a súbory potrebné pre prácu s databázou, ako napríklad prihlásenie, registrácia či uloženie novej fotografie sa nachádzajú vo vnútri zložky *firebase*, čo je názov databázového servera, ktorý v aplikácii využívame.



Štruktúra zdrojových súborov mobilnej aplikácie

Webový server - aplikácia

Systémové požiadavky, inštalácia a vývoj

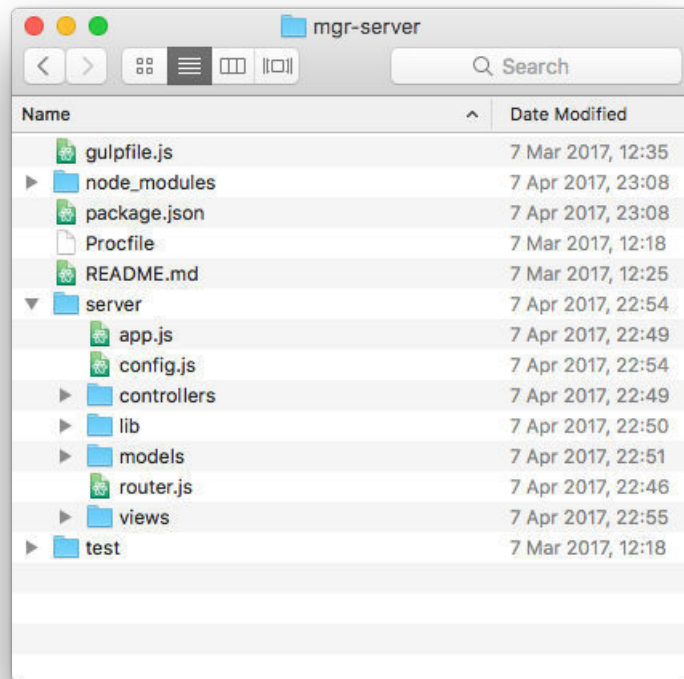
Nutnosťou pre lokálny vývoj aplikácie je nainštalovaný Node.js v minimálnej verzii 4. Taktiež server, na ktorom bude aplikácia bežať potrebuje rovnakú verziu Node.js, čo je aj jedinou požiadavkou na tento server. Pre lokálnu inštaláciu aplikácie je potrebné zavolať v termináli nasledujúce príkazy.

```
brew install node
npm install -g gulp-cli
npm install
```

Následný vývoj aplikácie môžeme začať príkazom `gulp server`, ktorý spustí lokálny server dostupný na adrese `localhost:8080`. Server sleduje zmeny súborov a automaticky sa reštartuje pri ich zaznamenaní.

Adresárová štruktúra

Štruktúra Node.js serverovej aplikácie obsahuje opäť zložku `node_modules` a súbor `packages.json` pre definíciu a uloženie používaných balíčkov. Súbor `gulpfile.js` pre spúšťanie vývojových skriptov. V zložke `server` sa nachádza hlavný súbor aplikácie `app.js`, konfiguračný súbor `config.js` a súbor s definíciou `HTTP` ciest `router.js`. V zložke `lib` sa nachádzajú pomocné funkcie volané v aplikácii, zložka `controllers` obsahuje súbory s aplikačnou logikou, databázové modely sa nachádzajú v zložke `models` a HTML šablóny v zložke `views`.



Štruktúra zdrojových súborov Node.js servera

Webový server - neurónová sieť

Systemové požiadavky, inštalácia a vývoj

Neurónovú sieť *Tensorflow* spúšťame na virtuálnom serveri od spoločnosti *Amazon*. Jej inštalácia spolu s inštaláciou python webového serveru *Flask* zahŕňa spustenie niekoľkých príkazov v prostredí terminálu.

```
// instalacia apache & flask
sudo apt-get update
sudo apt-get install apache2
sudo apt-get install libapache2-mod-wsgi
sudo apt-get install python-pip
sudo pip install flask

// vytvorenie adresara pre flask aplikáciu
mkdir ~/flaskapp
cd ~/flaskapp
```

```
// stiahnutie nasej serverovej aplikacie
git clone <repo_url> .
```

Pre správne namapovanie *flask* aplikácie je potrebné pridať do konfiguračného súboru */etc/apache2/sites-enabled/000-default.conf* nasledujúce riadky.

```
WSGIDaemonProcess flaskapp threads=5
WSGIScriptAlias / /var/www/html/flaskapp/flaskapp.wsgi

<Directory flaskapp>
    WSGIProcessGroup flaskapp
    WSGIApplicationGroup %{GLOBAL}
    Order deny,allow
    Allow from all
</Directory>
```

Teraz je všetko pripravené pre reštartovanie servera a následné načítanie našej aplikácie. Tiež môžeme nainštalovať neurónovú sieť Tensorflow a jej potrebné závislosti.

```
// restartovanie servera
sudo apachectl restart

// instalacia Tensorflow
sudo apt-get install python-dev
pip install tensorflow
```

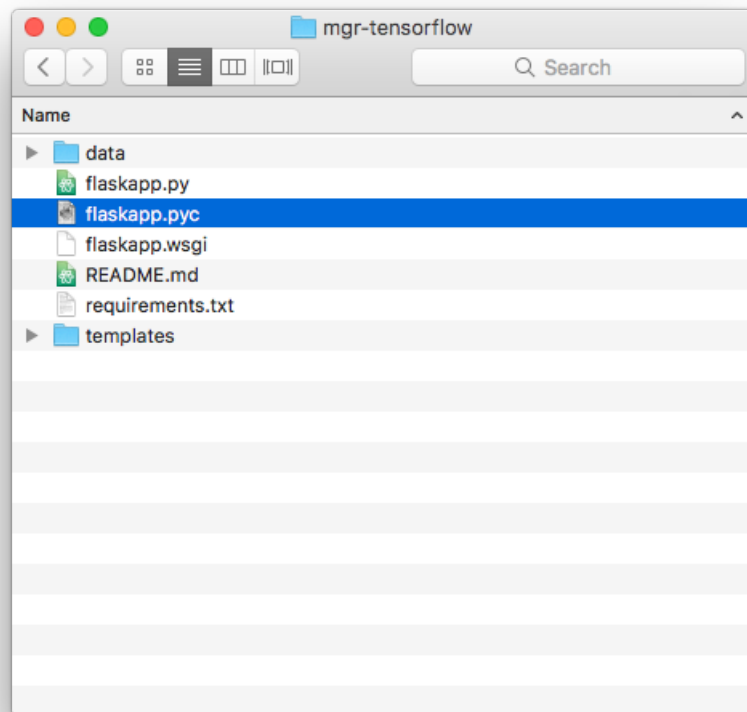
Po úspešnej inštalácii všetkých častí je aplikácia pripravená na prácu. Logy aplikácie môžeme sledovať zavolaním nasledujúcich príkazov z prostredia terminálu nášho servera.

```
// error logy
tail -f /var/log/apache2/error.log

// http a aplikacne logy
tail -f /var/log/apache2/access.log
```

Adresárová štruktúra

Štruktúra Python aplikácie na obsluhu neurónovej siete je najjednoduchšou z našich aplikácií. V *data* zložke sa nachádzajú sieťou vygenerované grafové a popisné dáta pre neskoršie používanie sieťou. Súbor *flaskapp.py* obsahuje celú logiku od obsluhovania *Tensorflow API* až po *HTTP* server vo frameworku *Flask*. Súbor *flaskapp.wsgi* slúži na automatické spúšťanie našej aplikácie serverom. Zložka *templates* obsahuje *HTML* šablóny pre zobrazovanie obsahu na webe. Súbor *requirements.txt* obsahuje závislosti potrebné pre inštaláciu cez *pip* inštalátor jazyku *Python*.



Štruktúra zdrojových súborov Tensorflow servera

Príloha B: Používateľská príručka

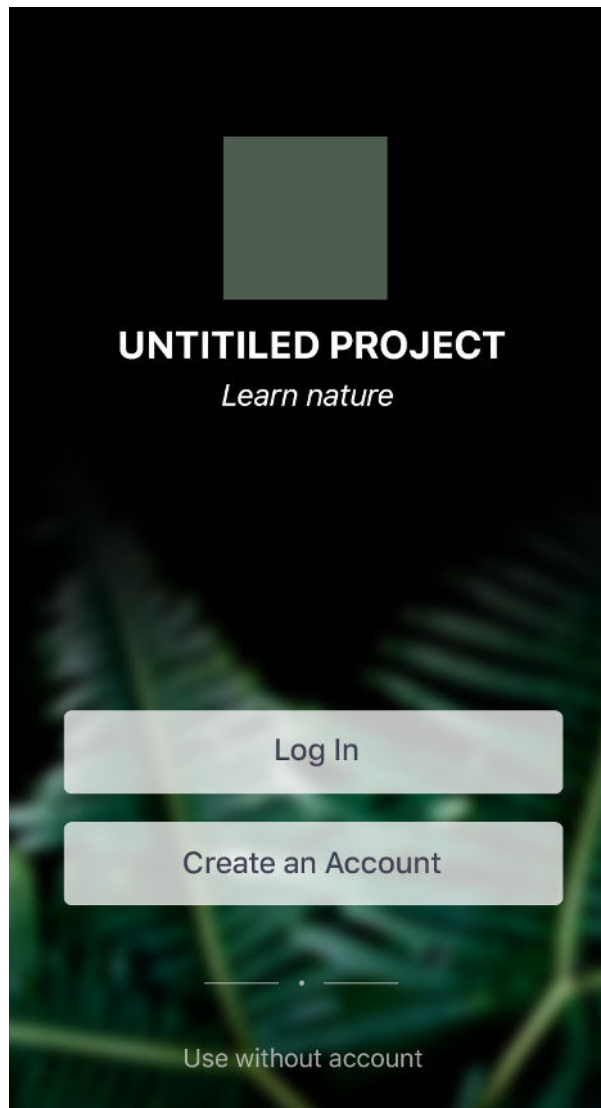
**UNIVERZITA MATEJA BELA V BANSKEJ BYSTRICI
FAKULTA PRÍRODNÝCH VIED**

**ANALÝZA A SPRACOVANIE OBRAZU V
MOBILNÝCH APLIKÁCIÁCH TVORENÝCH
POMOCOU WEBOVÝCH TECHNOLOGIÍ**

Používateľská príručka

Úvodná obrazovka

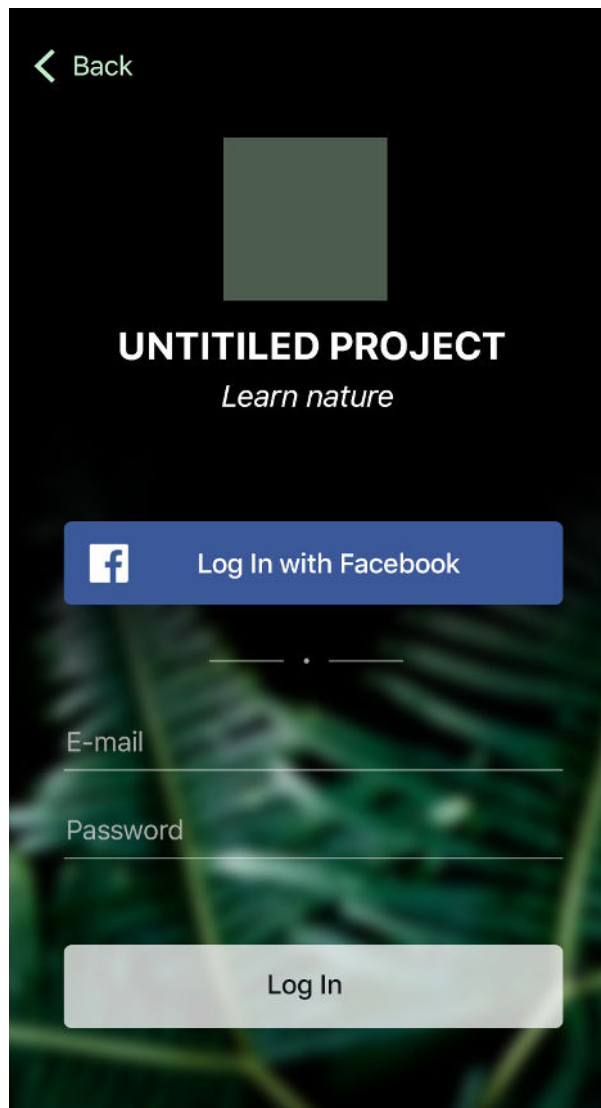
V prípade, že používateľ otvára aplikáciu prvýkrát, respektíve otvára ju ako neprihlásený dostáva sa na úvodnú obrazovku, kde si môže vybrať jednu z možností: prihlásenie, registrácia.



Úvodná obrazovka

Prihlásenie a registrácia

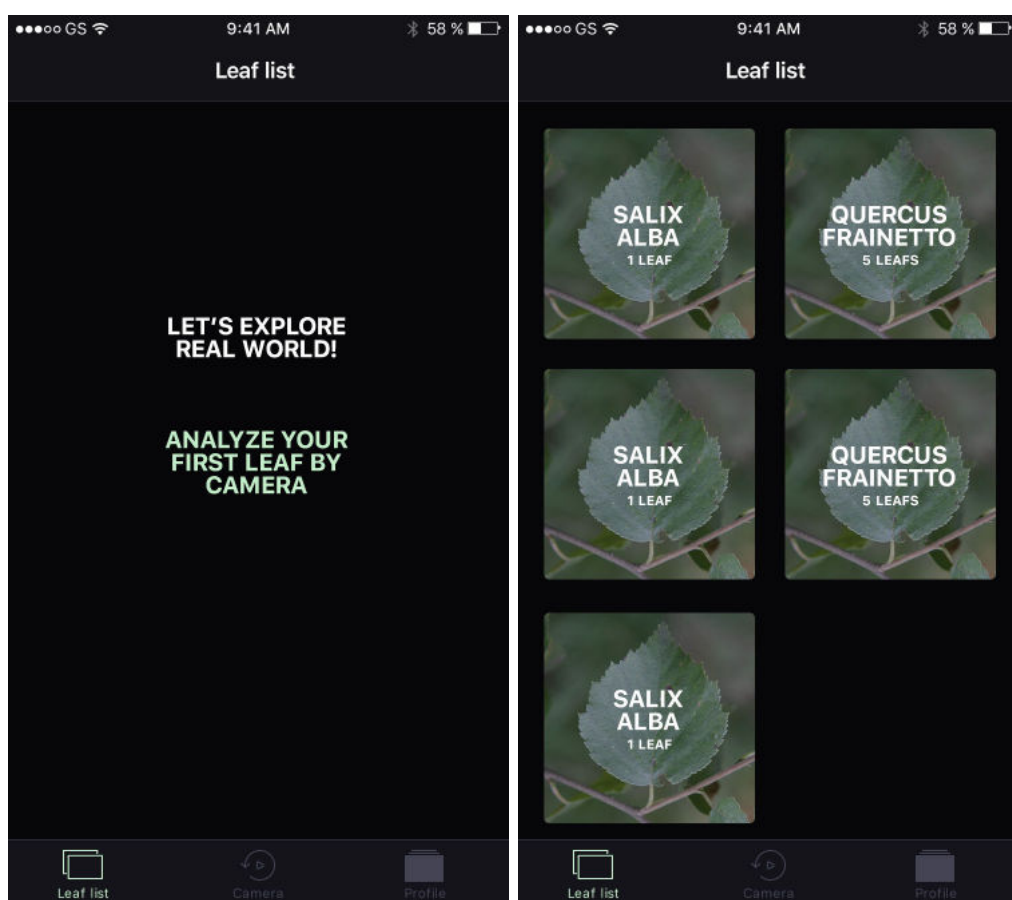
Používateľ sa po výbere jednej z volieb dostáva na obrazovku, kde má možnosť prihlásenia, resp. registrácie cez sociálnu sieť Facebook alebo klasickým spôsobom pomocou emailu a hesla.



Obrazovka prihlásenia

Môj herbár

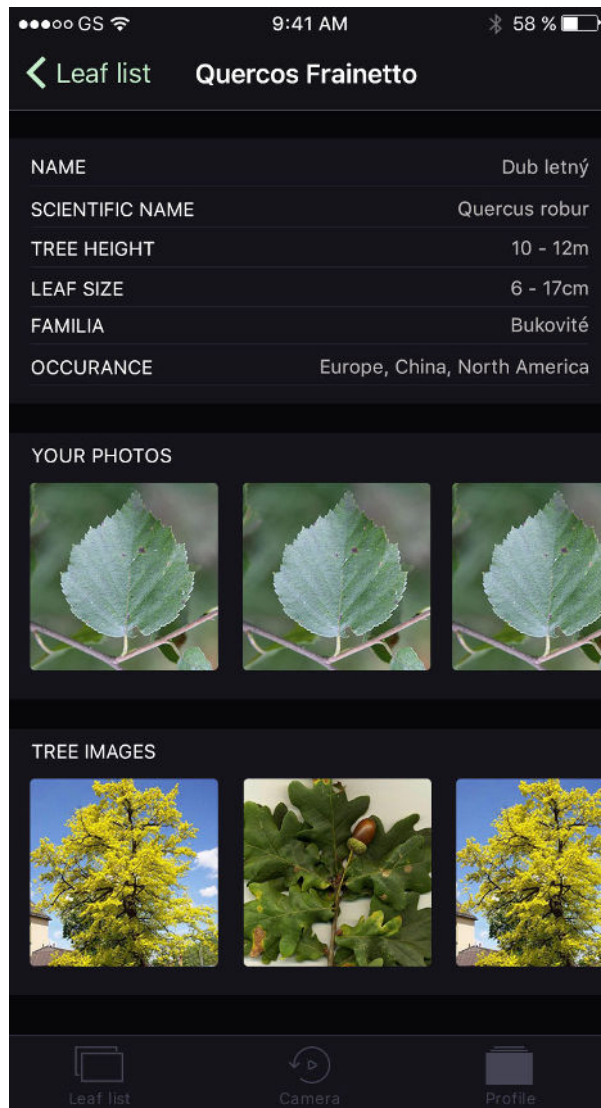
Po prihlásení, resp. po spustení aplikácie ako prihlásený používateľ sa otvára obrazovka s osobným herbárom, kde sa nachádzajú všetky stromy, ktorých listy sa používateľovi podarilo úspešne odfotografovať. Na obrázku vľavo vidíme situáciu, keď používateľ ešte nezachytil žiadny z listov. Na druhej strane vidíme položky s názvami stromu, počtom používateľových fotografií listov prislúchajúcich ku konkrétnemu stromu a pozadím položky, ktoré je tvorené vždy poslednou používateľovou fotografiou ku konkrétnemu stromu.



Obrazovka so zoznamom zachytených stromov

Detail stromu

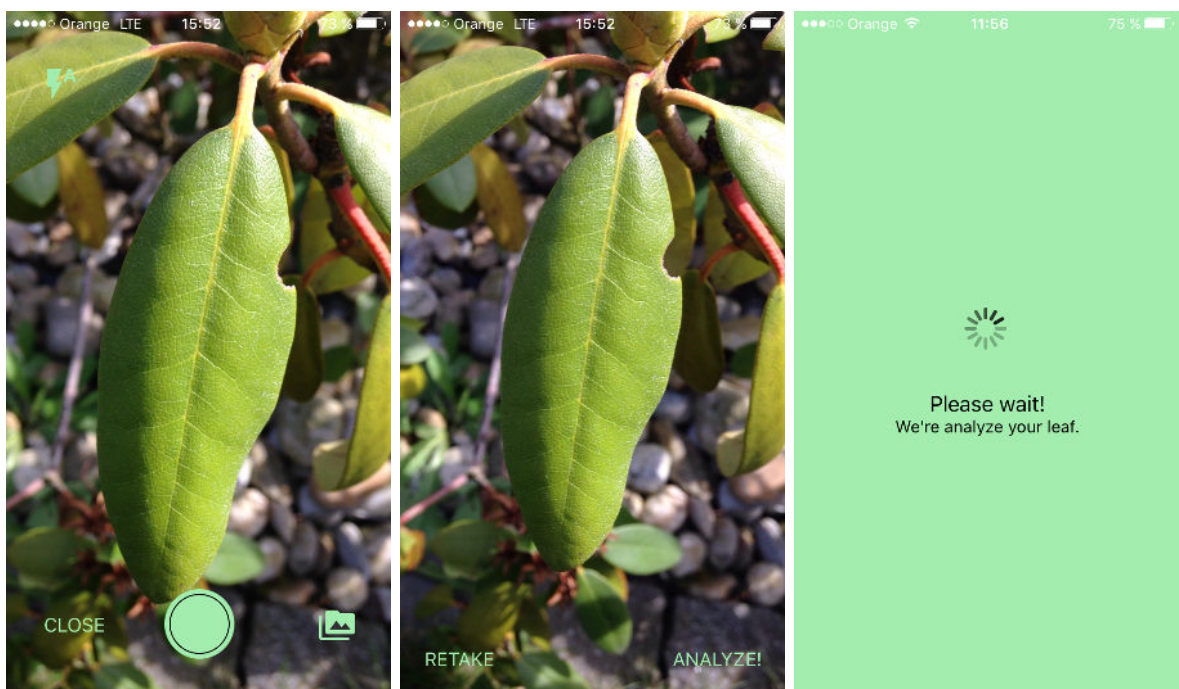
Po kliknutí na jednu z položiek v predchádzajúcej obrazovke so zoznamom stromov sa používateľ dostáva na obrazovku s detailom konkrétneho stromu. Tá obsahuje popisné detaily o konkrétnom strome, používateľove fotografie listov a tiež ukážkové fotografie celého stromu z databázy aplikácie.



Obrazovka s detailom konkrétneho stromu

Zachytenie fotografie a analýza

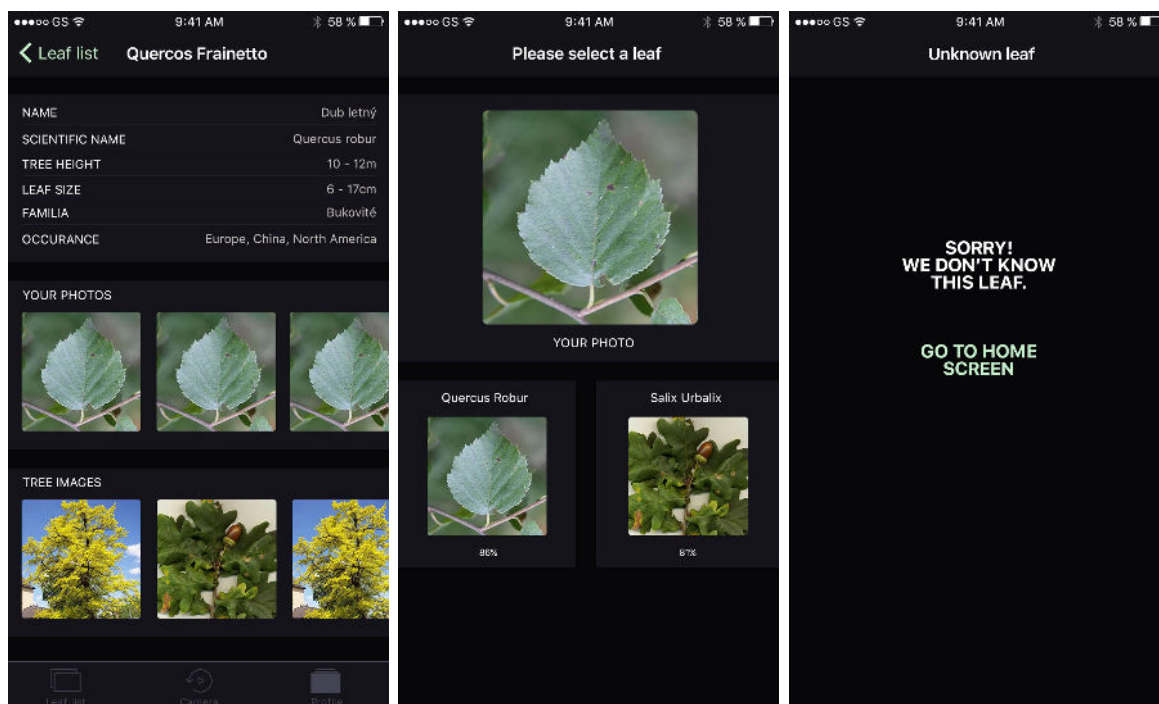
Po zvolení možnosti *Camera* z hlavného menu sa používateľ dostáva na obrazovku s kamerou, kde môže priamo zachytiť list. Tiež môže otvoriť svoj zoznam fotografií a vybrať jednu zo svojich starších fotografií. Po odfotografovaní, resp. vybratí fotografie z galérie sa používateľ dostáva na obrazovku, kde si môže svoju fotografiu ešte raz prezrieť a rozhodnúť sa či ju chce zahodiť, alebo poslať na analýzu. V prípade výberu analýzy sa dostáva na obrazovku s čakaním na odpoveď servera a dokončenie analýzy neurónovou sieťou.



Proces zachytávania a analýzy fotografie

Výsledok analýzy

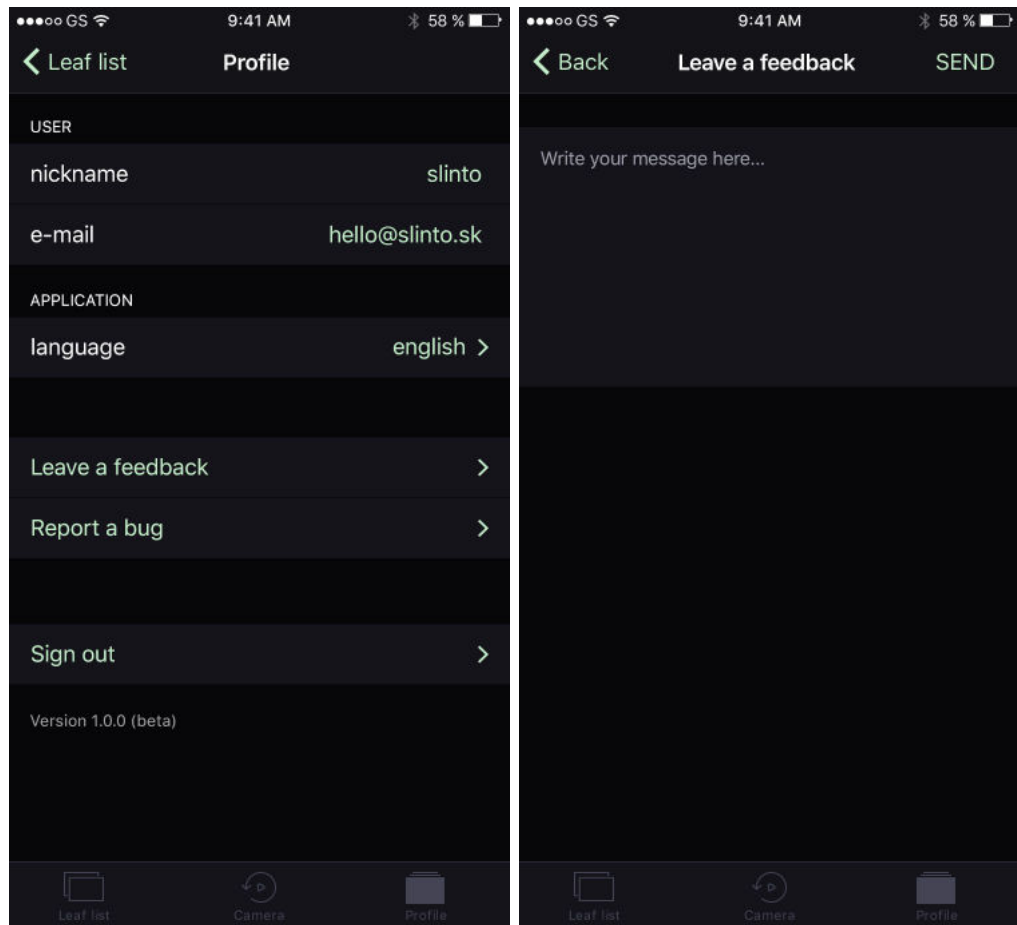
Po skončení analýzy, dostáva aplikácia odpoveď a môžu nastať tri varianty výsledku.. Prvý znamená, že list bol správne analyzovaný a používateľ sa dostáva na detail konkrétneho stromu, ku ktorému patrí analyzovaný list. Druhý variant nastáva v prípade, že sieť nejednoznačne predpokladá klasifikáciu listu a je na používateľovi, aby priradil svoju fotografiu do kategórie, medzi ktorými sa sieť rozhoduje. Tretí variant znamená, že sieť nenašla podobnosť so svojou databázou.



Obrazovky s možnými výsledkami analýzy fotografie

Profil používateľa

Ak používateľ zvolí v hlavnom menu položku *Profile*, dostáva sa na svoj osobný profil, kde má možnosti ako zmenu mena, hesla či jazyka. Tiež je možné zaslať spätnú väzbu resp. nahlásiť chybu v aplikácii pomocou vyplnenia a odoslania formulára. Takisto je tu možnosť odhlásiť sa a veľký priestor pre budúce štatistiky a hodnotenia.



Profil používateľa a príslušné obrazovky

Príloha C: Sprievodné CD so zdrojovým kódom