SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1713

SUSTAV ZA AUTOMATSKO OPTIČKO OČITANJE DIGITALNIH PRIKAZNIKA

Luka Škorić

Zagreb, lipanj 2018.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA DIPLOMSKI RAD PROFILA

Zagreb, 9. ožujka 2018.

DIPLOMSKI ZADATAK br. 1713

Pristupnik: **Luka Škorić (0036473237)**
Studij: Elektrotehnika i informacijska tehnologija
Profil: Elektroničko i računalno inženjerstvo

Zadatak: **Sustav za automatsko optičko očitanje digitalnih prikaznika**

Opis zadatka:

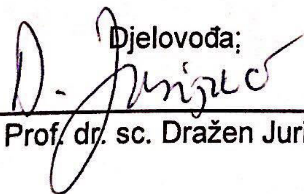
U okviru diplomskog rada potrebno je istražiti raspoložive sustave i algoritme koji se koriste za automatsko prepoznavanje numeričkog očitanja iz fotografija digitalnih prikaznika. Digitalni 7-segmentni prikaznici, najčešće u LCD tehnologiji se koriste kao prikazna jedinica na brojnim mjernim uređajima, pa bi se očitanjem prikazanog višeznamenkastog broja iz fotografije prikaznika moglo ostvariti automatizirano očitanje izmjerenog podatka bez intervencije u mjerni uređaj (npr. automatsko očitanje stanja brojila električne energije). Prema primjerima raspoloživih sustava i temeljem literature, potrebno je razviti vlastito rješenje za ovu zadaću. Predobrada snimljene fotografije mora osigurati uklanjanje uobičajenih problema i ograničenja u sustavu oslikavanja, kao što su neravnomjerno osvjetljenje i kontrast, problem odbljeska okoline od stakla prikaznika, problem geometrijske deformacije zbog perspektive kamere i slično. Algoritam mora identificirani rubove prikaznika u slici, te unutar tih rubova, mora odrediti očekivane pozicije svake pojedine znamenke prikaznika. Segmentirane slike pojedinih znamenki potrebno je zatim automatski prepoznati odgovarajućom metodom: usporedbom s predloškom ili primjenom postupaka umjetne inteligencije i strojnog učenja. Analizirati točnost prepoznavanja numeričkog očitanja u različitim realnim uvjetima degradacije kvalitete slike prikaznika.

Zadatak uručen pristupniku: 16. ožujka 2018.
Rok za predaju rada: 29. lipnja 2018.

Mentor:

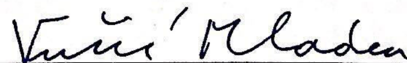
Prof. dr. sc. Davor Petrinović

Djelovođa:



Prof. dr. sc. Dražen Jurišić

Predsjednik odbora za diplomski rad profila:



Prof. dr. sc. Mladen Vučić

Sadržaj

1. Uvod ... 1

2. Prepoznavanje alfanumeričkih znakova ... 3

3. Metodologija programskog rješenja ... 5

4. Biblioteke otvorenog koda ... 10

4.1. OpenCV ... 10

4.2. NumPy ... 11

5. Proces predobrade slike ... 12

5.1. Biblioteka pomoćnih funkcionalnosti ... 40

6. Proces donošenja odluke o sadržaju slike ... 42

6.1. Prepoznavanje standardnim predloškom ... 43

6.2. Prepoznavanje metodama strojnog učenja ... 48

6.2.1. Izrada baze podataka ... 48

6.2.2. Algoritam neuronske mreže ... 51

6.2.3. Algoritam K najbližeg susjeda ... 58

7. Rezultati ... 61

8. Računalna implementacija sustava ... 63

8.1. Utrošak procesorskog vremena ... 66

8.2. Upute za korištenje ... 67

9. Zaključak ... 69

Literatura ... 70

Sažetak ... 71

Abstract ... 72

1. Uvod

Suvremeni računalni sustavi velikih memorijskih kapaciteta i naprednih procesorskih karakteristika, omogućuju realizaciju najzahtjevnijih sustava obrade podataka. Oblici implementacije sustava računalnog vida (eng. *Computer Vision*) jedan su primjer složenijih metoda unosa podataka iz okoline u računalni sustav.

Računalni vid je interdisciplinarno područje računarstva koje se bavi postizanjem visoke razine razumijevanja sadržaja digitalnih slika ili video materijala, od strane računala. Osnovno područje interesa pri bavljenju računalnim vidom obuhvaća procese automatskog prikupljanja, analize i razumijevanja korisne informacije iz pojedinačne slike ili sekvencijalnog slijeda slika. Razumijevanje promatranog sadržaja u kontekstu računalnog vida znači transformaciju informacije sadržane na slici, u podatkovni format prikladan interakciji sa računalom. Izradu takvog programskog sustava obrade podataka, možemo promatrati kao proces reorganizacije podatkovne strukture koja u memoriji računala predstavlja promatranu sliku, sa ciljem izolacije sadržaja koji na slici predstavlja traženu karakteristiku. U tu svrhu koriste se složeni programski modeli temeljeni na pravilima geometrije, fizike, statistike, strojnog učenja i sl.

Organizacija računalnog sustava namjenjenog implementaciji računalnog vida uvelike ovisi o predviđenoj aplikacijskoj funkcionalnosti tog sustava. Sustavi se najčešće realiziraju se kao samostalne (eng. *Stand Alone*) aplikacije koje obavljaju konkretan problem mjeriteljske ili klasifikacijske prirode, ili kao organizirana hijerarhija manjih podsustava čija funkcionalnost obuhvaća širi opseg određene problematike. Većina funkcionalnih karakteristika sustava računalnog vida strogo ovise o potrebama primjene razvijanog sustava. Neke od tipičnih funkcionalnih karakteristika većine sustava računalnog vida redom podrazumjevaju: prihvat slike kao ulazne informacije, predobrada primljenih podataka (slike), izvlačenje značajki, segmentacija određenih regija slike, te donošenje odluke o odabranom sadržaju.

Pri hardverskoj realizaciji sustava računalnog vida naglasak se stavlja na memorijske i procesorske kapacitete korištenog računala, te implementaciju zadovoljavajućih sučelja za unos podataka u sustav i prikaz krajnjih rezultata.

Većina suvremenih stolnih ili prijenosnih računala sa lakoćom izvodi složenije algoritme. U slučaju implementacije algoritama računalnog vida na ugradbena računala, potrebno je prilagoditi cjelokupni sustav i sve njegove karakteristike zahtjevima i složenosti korištenog algoritma. Računalni sustavi realizirani na taj načn ograničeni su na uži opseg prethodno određenih funkcialnosti razvijenog algoritma.

U okviru ovog rada naglasak se stavlja na razvoj programske podrške sustava s karakteristikama računalnog vida čija je zadaća ostvarivanje funkcionalnosti optičkog očitanja digitalnih numeričkih i/ili alfanumeričkih prikaznika. Digitalni 7-segmentni prikaznici, najčešće u LCD tehnologiji, koriste se kao prikazna jedinica na brojnim mjernim uređajima. Očitanjem prikazanog sadržaja iz fotografije takvih prikaznika može se ostvariti automatizirano očitanje prikazanog podatka, bez intervencije u mjerni uređaj. Takav pristup unosu podataka u računalo izuzetno je povoljan u slučajevima gdje pojedini mjerni uređaji ne sadržavaju određene računalne komponente i komunikacijsko sučelje koje bi omogućilo direktno spajanje računala na mjerni uređaj. Razvijeni algoritam pokriva sve korake u procesu obrade ulazne slike, te prosljeđuje obrađene podatke u tri različita algoritma donošenja odluke o sadržaju. Cjelokupnu programsku podršku razvijenu u okviru ovog rada moguće je implementirati na većini suvremenih stolnih i prijenosnih računala. Primjer implementacije razvijenog sustava na komercijalno dostupnom ugradbenom računalu Raspberry Pi 3, naveden je u kasnijim poglavljima ovog rada u demonstrativne svrhe.

U narednim poglavljima detaljno je opisan programski kod iza razvijenog sustava predobrade, obrade i segmentacije slike, te kod triju različitih algoritama donošenja odluke o lokaliziranoj vrijednosti. U kasnijim poglavljima navedene su upute za korištenje razvijenog sustava, te su prokomentirani dobiveni rezultati.

2. Prepoznavanje alfanumeričkih znakova

Prepoznavanje alfanumeričkih znakova (eng. *Optical Character Recognition*) proces je mehaničkog ili električkog prepoznavanja značenja sadržaja prikazanog na nekoj slici, te prilagodbe dobivenih podataka u oblik prihvatljiv komunikaciji sa računalom. Ulazne informacije računalnih sustava koji imaju zadaću prepoznavanja alfanumeričkih znakova najčešće su fotografije promatranih dokumenata, skenovi dokumenata ili ambijentalne fotografije koje u sebi sadrže promatrani znakovni uzorak. Optičko prepoznavanje alfanumeričkih znakova je područje računarstva sadržano u širim disciplinama računalne znanosti poput strojnog učenja (u vidu prepoznavanja uzoraka), umjetne inteligencije, te računalnog vida.

Razvoj najranijih oblika tehnologije prepoznavanja alfanumeričkih znakova započinje u prvim desetljećima 20. stoljeća. Uređaji razvijani u prvoj polovici 20. stoljeća u većini slučajeva radili su na principu pretvorbe očitanih štampanih znakova na priloženom papiru u signale koji su odgovarali pripadajućem simbolu očitanog znaka u Morseovom kodu. Primjer jednog takvog uređaja iz najranijeg razdoblja razvoja optičkih čitača je opthophone, izum irskog fizičara Edmunda Fourniera d'Albea. Napretkom suvremenih računalnih sustava poput pametnih telefona, mogućnosti računalnog prepoznavanja alfanumeričkih znakova iz slike poprimaju široki opseg mogućnosti. Funkcionalnosti optičkog prepoznavanja znakova na suvremenim računalnim sustavima ostvaruju se programskom implementacijom standardiziranih programerskih aplikacijskih sučelja. Podatci korišteni u ovakvim procesima obrade fotografije najčešće su sadržani u slici zabilježenoj kamerom ili drugim optičkim senzorom integriranim u računalni sustav.

Stuktura programske podrške razvijane za potrebe sustava optičkog prepoznavanja alfanumeričkih znakova organizirana je s obzirom na dvije osnovne zadaće koje čine potpuni proces prepoznavanja informacije na slici. To su predobrada primljene slike i konkretno prepoznavanje pojedinog znaka, tj. donošenje odluke o prikazanoj vrijednosti. Algoritmi optičkog prepoznavanja

znakova su u velikoj većini slučajeva offline procesi, koji analiziraju sadržaj statičkog dokumenta.

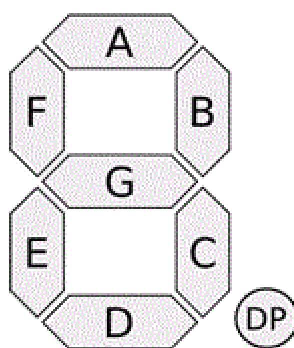
Predobrada slike kao ulaznog seta podataka koje sustav prima, podrazumjeva opširan proces prilagodbe sadržaja te slike kako bi u konačnici, iz pojedinačno izoliranih područja slike bilo moguće donjeti odluku o vrijednosti odabranog sadržaja. Složenost procesa predobrade uvelike ovisi o konačnim aplikacijskim funkcionalnostima koje je planirano ostvariti u gotovom proizvodu. Neke od najčešće primjenjivanih metoda, neophodne i u slučaju razvoja aplikacija minimalnih funkcionalnih sposobnosti su: korekcija dimenzijskog omjera i veličine slike, prilagodba oštrine primjenom raznih filtara, binarizacija boja na slici, te segmentacija područja interesa potrebnih za daljnu obradu.

U procesu prepoznavanja sadržaja prikazane vrijednosti sa slike, prethodno prilagođeno i izolirano područje interesa sa jednim ili više alfanumeričkih znakova se prosliježuje algoritmu čija je zadaća donijeti odluku. Takvi algoritmi se najčešće temelje na karakteristikama metoda prepoznavanja uzoraka, izvlačenja značajki iz sadržaja slike, ili složenijeg pristupa poput neuronskih mreža, tj. strojnog učenja.

Tehnologija optičkog prepoznavanja alfanumeričkih znakova od strane računala primjenu pronalazi u širokom području mjeriteljskih i analitičkih procesa. Ovi oblici računalne funkcionalnosti, konkretnu primjenu pronalaze u zadatcima poput: unosa podataka pri procesima obrade dokumenata, analize sadržaja slike u stvarnom vremenu, digitalizacije tekstualnog sadržaja dokumenta sa ciljem manipulacije tekstom, unos podataka za potrebe analitičkih financijskih sustava, pomoć osobama sa invaliditetom, itd. Algoritmi za prepoznavanje alfanumeričkih znakova, uz sve mogućnosti, modifikacije i funkcionalne opcije, osnovnu primjenu pronalaze u zadatcima unosa sadržaja tekstualnog formata u memoriju računala.

3. Metodologija programskog rješenja

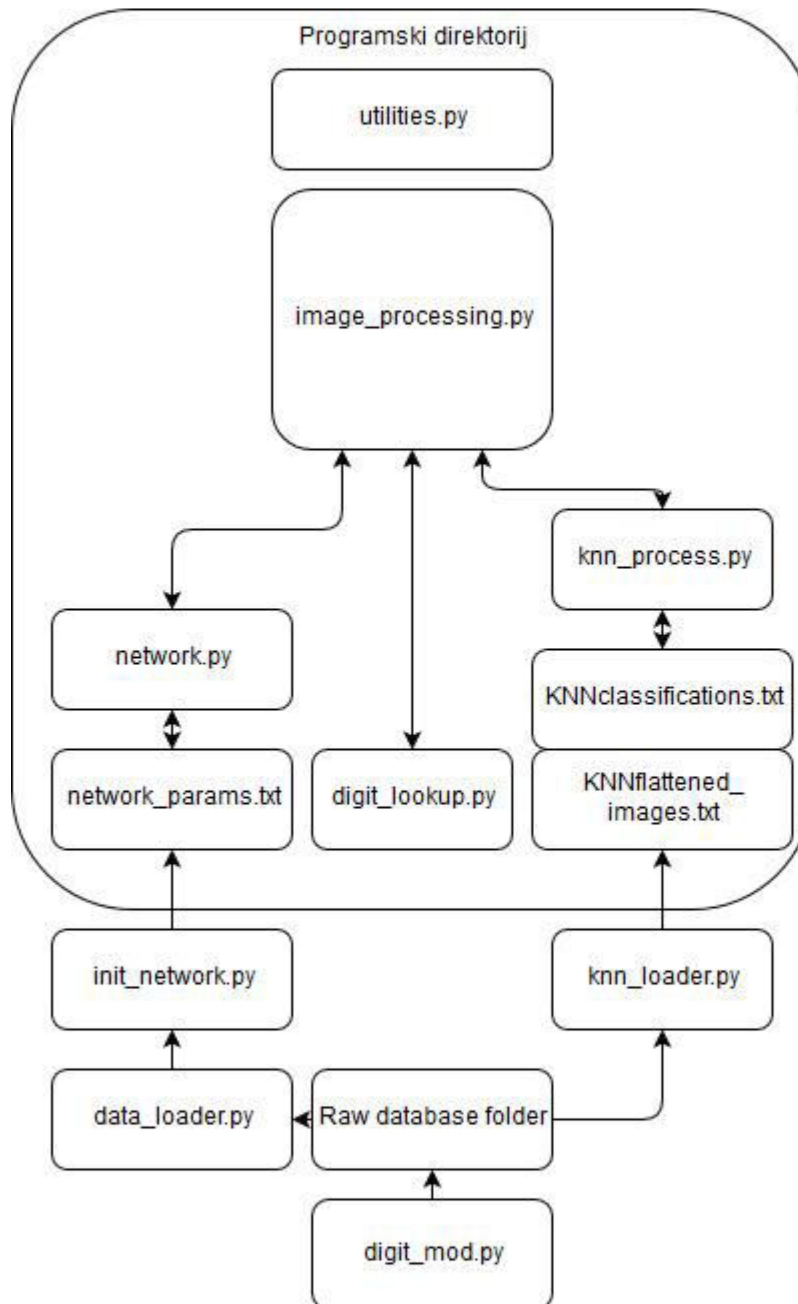
U okviru ovog rada naglasak pri razvoju sustava stavljen je na razvoj programske podrške u obliku algoritma za prepoznavanje vrijednosti prikazanih na digitalnim prikaznicima sadržanima u slici kao ulaznom argumentu sustava. Razvijeni algoritam prilagođen je prepoznavanju vrijednosti izražene 7 – segmentnim alfanumeričkim znakom. 7 – segmentni način prikaza decimalnih znamenki je format zapisa svake od deset znamenki arapskog znakovnog sustava, koji se ostvaruje manipulirajući aktivacijskim stanjima svakoga od 7 raspoloživih segmenata tog formata. Zbog visoko standardiziranog izgleda pojedine znamenke, ali i zbog jednostavnosti implementacije u elektroničko sklopovlje, ovaj format čestu primjenu pronalazi u elektroničkim uređajima najšireg raspona mogućih funkcionalnosti. Primjer strukture jedne znamenke 7 – segmentnog formata prikaza dan je slikom (Slika 3.1).



Slika 3.1 Struktura 7 - segmentnog formata [3]

Programski kod algoritma razvijenog u okviru ovog rada pisan je u programskom jeziku Python, verzije 2.7.12. Cjelokupna aplikacijska programska podrška inicijalno je prilagođena uporabi unutar okruženja operacijskih sustava Unix porodice. Uz manje preinake razvijenu aplikaciju moguće je implementirati i na drugim tipovima operacijskih sustava.

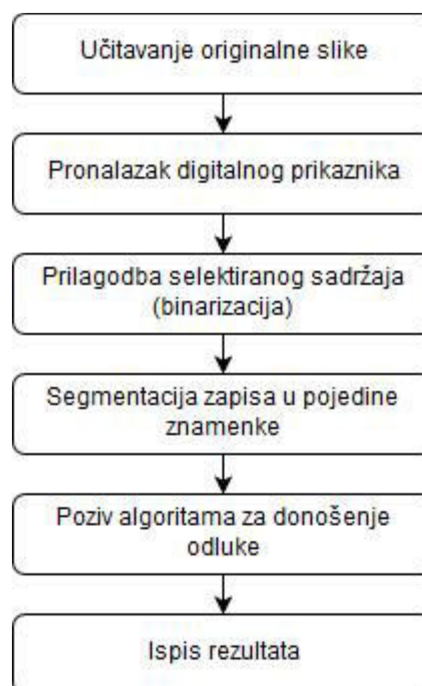
Arhitektura programskog rješenja razvijene aplikacije, organizirana je u većem broju skripti s kodom programskog jezika Python. Ulogu pojedinog odsječka programskog koda sadržanog u pojedinoj skripti, te njihovu poziciju unutar arhitekture određuje funkcionalna zadaća tog segmenta koda. Grafički prikaz arhitekture programske podrške razvijene aplikacije prikazan je slikom (Slika 3.2).



Slika 3.2 Grafički prikaz arhitekture sustava

Glavnina programskog koda vezanog uz proces prilagodbe i predobrade ulazne slike, te uz segmentaciju područja interesa, što su u ovom slučaju 7 – segmentne znamenke, nalazi se u modulu *image_processing.py.* Ovaj modul zauzima svojevrsnu „centralnu ulogu“ unutar organizacijske strukture programskog koda cjelokupnog projekta. Funkcionalnosti sadržane u preostalim skriptama s programskim kodom se u većoj ili manjoj mjeri integriraju u sadržaj koda modula *image_processing.py.* Modul *utilities.py* objedinjuje metode korištene na više mjesta u programskom kodu cjelokupnog projekta, poput rotacije slike, promjene dimenzija slike ili prilagodbe perspektive.

Kao što je ranije navedeno, funkcionalnost sustava optičkog prepoznavanja alfanumeričkih znakova dijeli se na fazu predobrade učitane slike koja završava segmentacijom pojedinog znaka i fazu donošenja odluke o značenju tog znaka. Programski kod skripte *image_processing.py* implementira sve procese prve od dviju navedenih faza. Funkcionalnosti obuhvaćene ovim dijelom programskog koda, organizirane su kako je prikazano slikom (Slika 3.3).



Slika 3.3 Organizacija sadržaja image_processing.py

Program započinje s radom učitavanjem slike na kojoj je prikazan predmet promatranja. Prvi zadatak algoritma koji sada raspolaže sa svim potrebnim podatcima je lokalizacija digitalnog prikaznika sa alfanumeričkim znakovima čija vrijednost se traži. Segment programskog koda korišten u tu svrhu obuhvaća kratak algoritam isticanja svih kontura unutar slike. Podrazumjevajući pravokutan oblik traženog digitalnog prikaznika, eliminacijskim postupkom dolazi se do konture čiji oblik najviše odgovara pretpostavci. Poznavajući koordinate digitalnog prikaznika unutar slike, digitalni prikaznik se u nastavku algoritma, uz prilagodbu perspektive, pohranjuje kao zasebna slika. Sljedeći zadatak algoritma modula *image_processing.py* se odnosi na postupak uklanjanja svih boja na slici, ostavljajući sadržaj prikazan kao potpuno crne ili potpuno bijele konture. Takav postupak se naziva binarizacija sadržaja slike. Ovako prilagođena slika digitalnog prikaznika, prosljeđuje se kroz algoritam segmentacije alfanumeričkog zapisa prikazanog na slici, tj. lokaliziranja pojedinog znaka. Lokalizacija pojedine 7 – segmentne znamenke ostvaruje se nizom eliminacijskih provjera binariziranog sadržaja slike. Završni korak algoritma prepoznavanja sadržaja digitalnog prikaznika predaje listu slika sa individualnim znakovima algoritmu donošenja odluke o vrijednosti pojedinog znaka.

Funkcionalnost donošenja odluke o vrijednosti individualizirane znamenke, u okviru ovog rada je ostvarena trima različitim metodama. Metoda određivanja vrijednosti pomoću predefiniranog predloška realizirana je programskim kodom u skripti *digit_lookup.py.* Funkcionalnost ove metode počiva na pretpostavci visoko standardiziranog formata 7 – segmentnih, dekadskih znamenki. Na slici svake učitane znamenke, algoritam provjerava sadržaj prethodno definiranih područja interesa i ovisno o usklađenosti sa očekivanim vrijednostima prepoznaje rezultat. Prednost ove metode je i mogućnost jednostavne provjere postojanja decimalne točke. Druga metoda donošenja odluke korištena u okviru ovog rada je implementirana algoritmom neuronske mreže. Riječ je o jednostavnoj feedforward neuronskoj mreži stohastičkog gradijentnog spusta, računatog algoritmom povratnog prostiranja (eng. *Backpropagation algorithm*). Programski kod koji ostvaruje neuronsku mrežu obuhvaćen je skriptom *network.py*. Treća metoda donošenja odluke o vrijednosti pojedine znamenke ostvarena je implementacijom funkcionalnosti algoritma K najbližeg susjeda (eng. *K nearest neighbour algorithm,*

skraćeno *Knn*). Aktivacija ovog algoritma realizirana je programskim kodom sadržanim u skripti *knn_process.py*.

Navedeni algoritmi neuronske mreže i K najbližeg susjeda pripadaju metodama strojnog učenja. S obzirom da osnovni koncept svih algoritama strojnog učenja počiva na mogućnosti postupnog poboljšanja performansi algoritma, višestrukim iteriranjem kroz odgovarajuću bazu podataka, u okviru ovog rada kreirana je baza podataka namjenjena učenju navedenih algoritama. Baza podataka se sastoji od 10000 slika 7 – segmentnih znamenki namjenjenih „treningu" i dodatnih 1500 slika namjenjenih provjeri. Sve slike ove baze podataka kreirane su implementacijom jednostavnog programa sadržanog u skripti *digit_mod.py*. Navedeni programski odsječak obuhvaća kratak algoritam namjenjen iterativnom postupku modifikacije učitane slike dekadske znamenke, primjenom raznih morfoloških efekata i rotacije. Prilagodba sadržaja ovakve baze podataka za potrebe neuronske mreže vrši se aktivacijom skripte *data_loader.py*. Postupak inicijalizacije svih parametara, te učenje navedene neuronske mreže ostvaruje se aktivacijom skripte *init_network.py*. Parametri izračunati izvedbom procesa učenja, spremaju se u obliku json stringa u tekstualnu datoteku *network_params.txt*, kako bi navedena mreža postala primjenjiva aktivacijom od strane drugih programa. Prilagodba baze podataka za potrebe Knn algoritma ostvaruje se programskim odsječkom iz skripte *knn_loader.py*. Rezultat izvršavanja tog programskog odsječka su dvije tekstualne datoteke *KNNclassifications.txt* i *KNNflattened_images.txt.* Prva navedena datoteka sadrži zapis dekadskih vrijednosti svake znamenke iz baze podataka, dok druga datoteka sadrži podatkovni zapis svake učitane slike, u formatu prilagođenom potrebama Knn algoritma. Pronalaskom uzorka koji najviše sliči podatkovnom sadržaju predane slike, Knn algoritam donosi odluku o vrijednosti znamenke na toj slici.

Proces izrade programskog sustava opisanog u ovom poglavlju, vođen je pretpostavkom nesavršeno pozicionirane kamere kojom se bilježi ulazni argument sustava. Sa ciljem postizanja što veće robusnosti razvijenog sustava, na mnogim mjestima se pojavljuju naizgled redundantni segmenti programskog koda.

U narednim poglavljima nalazi se detaljan opis programskog koda svih navedenih dijelova sustava.

4. Biblioteke otvorenog koda

Programski sustav razvijen u okviru ovog rada realiziran je programskim jezikom Python 2.7.12. Uz redovne biblioteke integrirane u standardni paket ovog programskog jezika, pri realizaciji programskog rješenja korištene su mnogobrojne funkcionalnosti dviju vanjskih biblioteka: OpenCV i NumPy. To su javno dostupne biblioteke otvorenog koda, a sadrže mnogobrojne metode manipuliranja digitalnim zapisom slike, te skup matematičkih metoda namjenjenih toj svrsi, koje se od standardnog Pythona razlikuju prvenstveno u načinu interpretacije raznih podatkovnih struktura.

4.1. OpenCV

Biblioteka OpenCV (eng. *Open Source Computer Vision Library*) je programska biblioteka otvorenog koda, sadržaja prilagođenog zadatcima programske implementacije sustava računalnog vida i/ili strojnog učenja. Izrada i održavanje ove biblioteke motivirano je idejom stvranja zajedničke objedinjene infrastrukture korištene pri razvoju aplikacija složenih programskih funkcionalnosti, te promoviranja takvih programskih tehnologija u komercijalnoj uporabi. [6] OpenCV biblioteka se sastoji od više od 2500 optimiziranih algoritama koji podrazumjevaju opsežan skup programske podrške jednostavnijih, ali i veoma složenih funkcionalnih karakteristika. Primarna tematika većine sadržanih algoritama odnosi se na razvoj programskih sustava računalnog vida u stvarnom vremenu. Poznatija razvojna okruženja podržana bibliotekom OpenCV su TensorFlow, Torch, Caffe i sl. Programsko sučelje ove biblioteke razvijeno je u okruženju programskog jezika C++, ali sadrži adaptacije koje omogućuju primjenu biblioteke u razvojnim okruženjima programskih jezika Python, Java, te razvojnog paketa MATLAB. Velik broj najzastupljenijih operacijskih sustava podržava implementaciju sadržaja biblioteke OpenCV.

4.2. NumPy

NumPy (eng. *Numeric Python*) je biblioteka namjenjena radu unutar razvojnog okruženja programskog jezika Python. Biblioteka predstavlja skup proširenja razvojnog okruženja ovog programskog jezika koji korisniku omogućuje efikasno manipuliranje velikim setovima objekata organiziranih u matričnom obliku. Takvi setovi objekata nazivaju se redovi (eng. *array*) i mogu imati proizvoljan broj dimenzija. [5] Jednodimenzionalni redovi slični su standardnim Python slijedovima, dvodimenzionalni redovi slični su matricama linearne algebre, itd. Razlika između interpretacije podataka implementacijom NumPy proširenja i standardnog programskog sučelja jezika Python zasniva se na memorijskom zapisu određenog tipa podatka. Osnovna funkcionalnost NumPy biblioteke manifestira se zapisom podataka n – dimenzionalnim redovima nazvanima „ndarray“. Ovi redovi, za razliku od ugrađenih Python podatkovnih struktura za implementaciju nizova, pretpostavljaju točno definiranu veličinu memorijskog buffera svakog pojedinog elementa reda, izraženu u byteovima, te isti tip svakog zapisanog elementa. Takav oblik realizacije podatkovnog zapisa ostvaruje velike vremenske i memorijske uštede pri izvođenju algoritama koji obrađuju velike količine podataka, a funkcionalno se ne razlikuje od standardnog načina Python programskog sučelja.

Primjenom NumPy redova ostvaruje se mogućnost uvida u memorijske buffere alocirane programskom sučelju jezika Python od strane aplikacija izvedenih drugim programskim jezicima, bez potrebe za posebnim kopiranjem memorijskog sadržaja. Navedena karakteristika predstavlja visoku razinu integracije ove biblioteke sa drugim sličnim bibliotekama, ali i sučeljima drugih programskih jezika. Primjena sadržaja NumPy biblioteke u programima ostvarenima programskim jezikom Python rezultira postizanjem funkcionalnosti usporedivih programskom razvojnom okruženju MATLAB.

5. Proces predobrade slike

U okviru ovog rada razvijen je programski sustav namjenjen automatskom prepoznavanju vrijednosti znamenki prikazanih na promatranom digitalnom prikazniku. Proces predobrade i prilagodbe slike kao ulaznog argumenta sustava obuhvaća korake lokalizacije prikaznika, obrade boja u slici, tj. binarizaciju, te segmentaciju zapisa u individualne znamenke. Metodologija ovog dijela programskog sustava prethodno je opisana u poglavlju 3. Programska implementacija procesa realizirana je programskim jezikom Python, te u potpunosti obuhvaćena skriptom *image_processing.py*. U nastavku ovog poglavlja detaljno je opisana programska izvedba cjelokupnog procesa, od inicijalne slike do pohranjivanja izolirane znamenke sa prikaznika. Tijekom opisa programskog koda, u demonstrativne svrhe je korištena slika (Slika 5.1). Zadatak sustava, u ovom slučaju je prepoznavanje prikazane cijene u kunama.

Slika 5.1 Izvorna slika kao ulazni argument sustava

Programski kod sadržan u skripti *image_processing.py* organiziran je linearno. Integracija programskog sadržaja ove skripte u cjelokupan projekt, opisana je u poglavlju 3.

Programski kod ove, ali i svih ostalih skripti sustava započinje inicijalnom naredbom iz linije 1. Ova naredba predstavlja uobičajnu sekvencu za zadavanje interpretera prilagođenog sučelju operacijskog sustava Unix porodice. Implementacijom ove sekvence pojedina skripta se deklarira izvršivom, te se može direktno pozivati, ovisno o potrebama korisnika.

```
1    #!/usr/bin/python
2
3    import sys
4    import json
5    import os
6    import cv2
7    import numpy as np
8
9    import utilities as ut
10   import digit_lookup as dl
```

Linije 3 – 6 sadrže naredbe uključivanja pojedinih biblioteka ugrađenih u programsko sučelje jezika Python, te ranije opisanih biblioteka otvorenog koda. U programsku skriptu učitava se biblioteka *utilities.py,* sa skupom metoda kreiranih za potrebe ovog projekta, te skripta *digit_lookup.py* koja sadrži algoritam donošenja odluke o 7 – segmentnoj znamenci na osnovi predloška. Detaljniji opis programskih funkcionalnosti realiziranih tim skriptama, dan je u narednim poglavljima. U nastavku ovog programskog odsječka inicijaliziraju se parametri potrebni za aktivaciju algoritama neuronske mreže i K najbližeg susjeda. Taj segment koda prikazan je u poglavljima posvećenima navedenim algoritmima.

```
28   # preparing output folder
29   folder = "./debug_znamenke"
30   for file_index in os.listdir(folder):
31
32       item = os.path.join(folder, file_index)
33       if os.path.isfile(item): os.remove(item)
```

U linijama 28 – 33 izvodi se inicijalno pražnjenje direktorija *debug_znamenke* namjenjenog pohranjivanju slika individualnih znamenki segmentiranog zapisa.

Znamenke spremljene na ovaj način koriste se isključivo u svrsi kontrole ispravnog rada programa (eng. *Debugging*), te ne utječu na funkcionalnost nekog od dijelova programskog sustava. Lokacija ovog direktorija postavljena je u glavni programski direktorij.

Do sada opisan programski sadržaj skripte *image_processing.py* odnosio se isključivo na inicjalizaciju svih biblioteka, direktorija i parametara. Proces predobrade i prilagodbe slike započinje učitavanjem promatrane slike (Slika 5.1) kao ulaznog argumenta programskog sustava naredbom `imread` u liniji 112.

```
112  img = cv2.imread(sys.argv[1])
113  resized = ut.resize(img, height = 500)
114  points = find_screen(resized)
```

Funkcija `imread` je dio biblioteke OpenCV i namjenjena je učitavanju slike n piksela visine i m piksela širine u radno okruženje sustava. Slika učitana na ovaj način, u varijabli `img` je pogranjena kao n – člani ndarray red. Svaki od n članova ovog reda predstavlja jedan redak piksela učitane slike, a pohranjen je kao m – člani ndarray. Svaki od m elemenata ovog ndarraya predstavlja pojedini piksel i zapisan je kao 3 – člani ndarray. Piksel se zapisuje na ovaj način jer je učitana slika inicijalno reprezentirana RGB formatom piksela. Svaki piksel takvog formata u memoriji zauzima 24 bita (industrijski standard od 2005. godine [3]) koja se dijele na tri 8 – bitna kanala: kanal crvene boje, kanal zelene boje i kanal plave boje. Intenzitet boje svakog kanala (dubina) predstavlja se dekadskom vrijednošću u rasponu od 0 (najtamniji) i 255 (najsvjetliji). Boja pojedinog piksela dobiva se odgovarajućim omjerom intenziteta svakog od tri kanala (boje).

Učitana slika zapisana u varijabli `img` prosljeđuje se metodi `resize`, iz biblioteke *utilites.py*, uz zadanu dimenziju. Svrha ove promjene dimenzije učitane slike je standardizacija s ciljem povećanja robusnosti algoritma, koji u idućim koracima na jednak način obrađuje svaku sliku, bez obzira na inicijalne dimenzije. Prilagođena, učitana slika je spremna za poziv metode `find_screen` čija je zadaća lociranje digitalnog prikaznika. Metoda `find_screen` nije uključena u biblioteku *utilities.py*, s obzirom da se njezina funkcionalnost koristi na samo

jednom mjestu unutar programske podrške cjelokupnog programskog sustava. Programski kod ove metode prikazan je u nastavku.

```
67   def find_screen(img):
68   
69   filtered = cv2.bilateralFilter(img.copy(), 9, 75, 75)
70   blurred = cv2.GaussianBlur(filtered, (5,5), 0)
71   cv2.imwrite('debug_01.jpg', blurred)
72   
73   edged = auto_canny(blurred)
74   
75   
76   
77   cv2.imwrite('debug_02.jpg', edged)
78   
79   (cnts, _) = cv2.findContours(edged.copy(),
                                   cv2.RETR_EXTERNAL,
                                   cv2.CHAIN_APPROX_SIMPLE)
80   cnts = sorted(cnts, key = cv2.contourArea, reverse = True)[:10]
81   
82   screenCnt = None
83   for i in cnts:
84   
85       peri = cv2.arcLength(i, True)
86       approx = cv2.approxPolyDP(i, 0.02 * peri, True)
87   
88       if len(approx) == 4 and cv2.isContourConvex(approx):
89           screenCnt = approx
90           break
91   
92   return screenCnt
```

Prvi konkretni koraci predobrade sadržaja učitane fotografije ostvaruju se pozivom OpenCV funkcija `bilateralFilter` i `GaussianBlur`, kroz prve dvije linije metode `find_screen`. Funkcija `bilateralFilter` na učitanu fotografiju primjenjuje bilateralni filtar koji mijenja intenzitet boje triju kanala svakog piksela sa prosječnom vrijednošću intenziteta boje susjednih piksela. Kao ulazni parametri zadaju se promjer promatranog područja, te pragovi razmatranog intenziteta i zahvata [6]. Funkcija `GaussianBlur` primjenom gausovog filtra zamućuje sliku i time uklanja sitne detalje, što smanjuje količinu potencijalnih oblika čije konture mogu biti detektirane. Zajedničko svojstvo ovim filtrima je funkcionalnost uklanjanja visokofrekvencijskih šumova uz očuvanje oštrine rubova oblika na slici. Rezultat ovog koraka predobrade učitane fotografije prikazan je slikom (Slika 5.2).

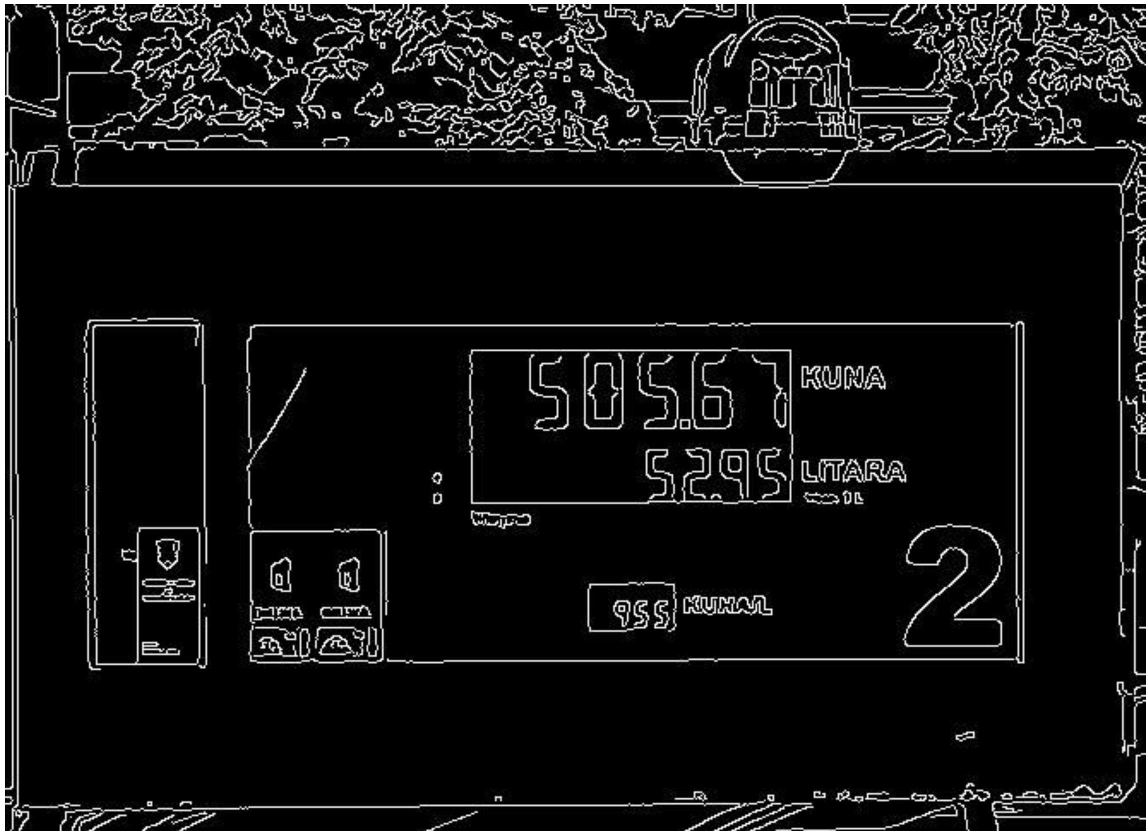
Slika 5.2 Ulazna slika nakon filtriranja

Rezultantna slika prilagođena je procesu obrade u algoritmu naglašavanja rubova svih prisutnih kontura sa slike, implementiranom kroz funkciju `Canny` iz biblioteke OpenCV. Funkcija `Canny` predstavlja implementaciju Canny detektora rubova prilagođenu izravnim korištenjem pri izradi programske podrške na većem broju platformi. Navedeni detektor rubova obuhvaća skup složenih algoritama manipulacije sadržajem slike, sa ciljem ekstrakcije rubova oblika i kontura, što rezultira značajno manjom količinom podataka koje se provjerava. Algoritam Canny detektora sastoji se od 5 osnovnih koraka: filtriranje slike, pronalazak gradijenta intenziteta piksela, otklanjanje netočnih odziva nemaksimalnom supresijom, određivanje pragova doljnje i gornje granice iznosa gradijenta, te primjena histereze pri konačnom razdvajanju „jakih“ i „slabih“ rubova. Sintaksa funkcije `Canny` iz biblioteke OpenCV, uz ulaznu sliku, kao argumente zahtjeva i predaju vrijednosti pragova selekcije iznosa gradijenta. Budući da vrijednosti tih pragova uvelike ovise o sadržaju slike, veoma teško ih je pretpostaviti. Kako bi se otklonila potreba za učestalim intervencijama u programski kod upitne

pouzdanosti, uz pomoć literature [2] izvedena je kratka metoda `auto_canny` namjenjena automatskom izračunu iznosa pragova.

```
54   def auto_canny(img, sigma=0.75):
55
56       # compute the median of the single channel pixel
     intensities
57       v = np.median(img)
58
59       # apply automatic Canny edge detection using the computed
     median
60       lower = int(max(0, (1.0 - sigma) * v))
61       upper = int(min(255, (1.0 + sigma) * v))
62       edged = cv2.Canny(img, lower, upper)
63
64       # return the edged image
65       return edged
```

Metoda `auto_canny`, kako se vidi iz ovog segmenta programskog koda, prima dva ulazna argumenta: `img` i `sigma`. Prvi argument predstavlja sliku u kojoj želimo istaknuti rubove, dok drugi argument predstavlja parametar za modifikaciju postotnih odnosa relacije za dorеđivanje doljnjeg i gornjeg praga, a njegovo zadavanje pri pozivu metode nije obavezno. Programski odsječak ove metode započinje računanjem mediana vrijednosti intenziteta piksela učitane slike. Ovako izračunati iznos mediana se množenjem sa postotnom vrijednošću određenom `sigma` varijablom koristi pri određivanju iznosa nižeg i višeg praga. Dobivene pragove sada se prosljeđuje metodi `Canny`, biblioteke OpenCV. Zadavanje niže vrijednosti `sigma` operatora rezultira užim rasponom traženih pragova. Tako zadani pragovi predani metodi `Canny` rezultiraju jačim prigušenjem kontura objekata na slici, čime se neki rubovi na rezultantnoj slici gube. Suprotno tome, veći iznos `sigma` operatora rezultira širim rasponom pragova, što rezultira detekcijom i onih manje istaknutih kontura, koje često nisu predmet promatranja. Pronalaskom optimalnog iznosa ovog operatora ostvaruje se zadovoljavajuća funkcionalnost metode isticanja kontura objekata na slici `Canny`. U slučaju programskog koda ovog projekta, optimalna vrijednost iznosi 0.75. Rezultat obrade slike (Slika 5.2) metodom `Canny` prikazan je slikom (Slika 5.3).



Slika 5.3 Istaknuti rubovi kontura na slici

Važno je napomenuti da metoda `Canny`, biblioteke OpenCV svaku rezultantnu sliku izlaznom argumentu predaje u potpuno binariziranom formatu. Svaki piksel ovako pohranjene slike prikazan je jednim 8 – bitnim kanalom intenziteta sive boje, što se naziva grayscale format. Binarizirani format slike je, u suštini, varijanta grayscale formata ograničena na dvije krajnosti: piksele vrijednosti 0 (potpuno crna boja) i piksele vrijednosti 255 (potpuno bijela boja). Pojam kontura u terminologiji računalnog vida predstavlja neprakinutu krivulju piksela iste boje ili intenziteta, te je stoga ovakav format slike prikladan za daljnju obradu.

Izlazni parametar metode `Canny` pohranjuje se u varijablu `edged`, iz koje se u nastavku programskog odsječka metode `find_screen` detektiraju tražene konture rubova. Navedena funkcionalnost detekcije kontura svih područja potpuno bijele boje ostvaruje se pozivom metode `findContours` biblioteke OpenCV. Ulazni argumenti ove metode su kopija binarizirane slike (algoritam verzije 2.4.9 OpenCV mijenja sadržaj ulaznog parametra), te dvije zastavice, u ovom slučaju

RETR_EXTERNAL i CHAIN_APPROX_SIMPLE. Prva navedena zastavica postavlja vrstu odnosa prema hijerarhiji promatranih kontura, tako da algoritam metode limitira na obradu isključivo vanjskih kontura promatranih objekata. Druga zastavica modificira oblik izlaznog argumenta metode, što je u ovom slučaju vektor točaka koje predstavljaju koordinate segmenata pojedine konture unutar slike [6]. Vektori točaka svih detektiranih kontura spremaju se u „običnu“ Python listu nazvanu cnts. Vektor točaka je ndarray red čiji su elementi koordinate svakog piksela pojedine konture. Metoda findContours vraća još jedan izlazni parametar koji sadrži matrični zapis hijerarhijskog položaja svake pronađene konture, ali je on potisnut s obzirom da se u ovom algoritmu taj podatak ne koristi.

Lista vektora svih detektiranih kontura se u idućem koraku sortira silazno u ovisnosti o površini koju kontura na izvornoj slici zauzima. Površinu svake konture računa metoda contourArea biblioteke OpenCV. Sadržaj liste cnts sada se prepisuje novosortiranom listom zadržavajući samo 10 kontura najveće površine. Ovo ograničenje proizlazi iz pretpostavke da izvorna slika sadrži veliki broj malih, neželjenih objekata, te da traženi digitalni prikaznik na slici zauzima znatno veću površinu od tih objekata.

Poslijednji korak algoritma metode find_screen odnosi se na selekciju konture koja najvjerojatnije predstavlja rubove traženog digitalnog prikaznika. Provjera se izvodi iterirajući kroz sadržaj liste vektora 10 najvećih kontura. Dvije metode biblioteke OpenCV svaki pojedini element liste cnts, tj. vektor segmenata svake odabrane konture aproksimiraju u pravokutni oblik. To su metode arcLength i approxPolyDP. Funkcionalnost aproksimacije kontura iz vektora njihovih segmenata, u navedenim metodama je ostvarena primjenom Douglas–Peucker algoritma [6]. Metoda arcLength računa parametar peri koji određuje maksimalno dozvoljeno odstupanje od originalnog oblika konture izraženo u postotnoj vrijednošću. To je najvažniji ulazni argument druge navedene metode koja iz vektora segmenata konture vraća vektor koordinata pravokutne aproksimacije promatrane konture, u varijabli approx. Pretpostavljajući pravokutni oblik traženog prikaznika na izvornoj slici, ali i činjenicu da će Douglas–Peucker algoritam sve veće i složenije oblike na izvornoj slici (uz isti parametar peri) vjerojatno aproksimirati kroz više od 4 točke, postavlja se konačno uvjetovanje.

Uvijet ispituje broj točaka kroz koje je algoritam aproksimirao pojedinu konturu, te prilikom pronalaska prve aproksimacije izvedene kroz samo 4 točke, prekida petlju i izlazi iz programskog odsječka funkcije `find_screen`.

Rezultat izvođenja te funkcije iz skripte *image_processing.py* je lista koordinata točaka uglova konture traženog digitalnog prikaznika. Lista je spremljena u varijablu nazvanu `points`.

```
116  # error check
117  if points is not None:
118      example = resized.copy()
119      cv2.drawContours(example, [points], 0, (255, 0, 0), 3)
120      cv2.imwrite('SCREEN1.jpg', example)
121  else:
122      points = np.array([(0, 0), (resized.shape[1] - 1, 0),
123                         (resized.shape[1] - 1, resized.shape[0] - 1),
124                         (0, resized.shape[0] - 1)],
125                          dtype = "float32")
```

Prije nastavka obrade sadržaja prikaznika, implementirana je jedna jednostavna sigurnosna provjera sa svrhom omogućavanja nastavka izvršavanja algoritma skripte *image_processing.py* i u slučaju neuspješnog pronalaska konture prikaznika u prethodno opisanoj funkciji. Uvijet pretpostavlja da ako kontura prikaznika nije uspiješno detektirana, povratna vrijednost funkcije `find_screen` ostaje `None`. U tom slučaju kao sadržaj varijable `points` se postavljaju krajnji kutovi originalne slike učitane u programski sustav. Koordinate uglova izvorne slike u ostatku algoritma mogu biti interpretirane kao uglovi prikaznika u specifičnim slučajevima kada prikaznik zauzima većinu kadra kamere sustava koja snima ulaznu sliku ili kada je riječ o prikaznicima bez naglašenih rubova što je čest slučaj sa LED prikaznicima. U slučaju ispravno detektirane konture digitalnog prikaznika, algoritam se nastavlja, a konture se u demonstrativne svrhe iscrtavaju na izvornoj slici, što je prikazano slikom (Slika 5.4).

```
127  scrImg = ut.transform_perspective(resized, points.reshape(4, 2))
128  scrImg = cv2.cvtColor(scrImg, cv2.COLOR_BGR2GRAY)
129  scrImg = ut.resize(scrImg, width = 300)

130  cv2.imwrite('SCREEN2.jpg', scrImg)
```

Koordinate uglova prikaznika sadržane u varijabli points prosljeđuju se funkciji `transform_perspective` biblioteke *utilities.py* čija je zadaća ekstrakcija digitalnog prikaznika iz izvorne slike, te prilagodba perspektive kako bi „izrezana“ regija bila prikazana potpuno vertikalno. Programski kod te funkcije detaljno je opisan u nastavku poglavja. Dobivena slika sadržaja digitalnog prikaznika sprema se u varijablu `scrImg`. U idućoj liniji programskog koda, uz pomoć funkcije `cvtColor`, format slike prikaznika se pretvara u Grayscale, zadržavajući samo jedan 8 – bitni kanal boje za svaki piksel koji sada ima ranije opisana svojstva. Ovako obrađena slika izoliranog digitalnog prikaznika, dana je slikom (Slika 5.5). Dimenzije ove slike se postavljaju s ciljem standardizacije rada ostatka algoritma.

Slika 5.4 Istaknuta kontura detektiranog prikaznika

Slika 5.5 Vertikalni prikaz sadržaja prikaznika, grayscale format

Slika izoliranog, dimenzijski i formatno prilagođenog sadržaja promatranog digitalnog prikaznika obuhvaća sav predmet interesa preostalog dijela algoritma, te se izvorna slika više ne koristi.

Slijedeći korak algoritma skripte *image_processing.py* se odnosi na proces potpune binarizacije slike (Slika 5.5), zapisane u varijabli `scrImg`. Cilj ovog procesa je ostvarivanje prikaza 7 – segmentnih znamenki, na slici prikaznika, kao piksele potpuno bijele boje (vrijednost 255), a svega ostaloga kao piksele potpuno crne boje (vrijednost 0). Tražena funkcionalnost, programski se ostvaruje postupkom određivanja pragova intenziteta pojedinih piksela (eng. *Thresholding*), te primjenom određenih morfoloških metoda biblioteke OpenCV.

```
135  # thresholding
136  average_val = int(np.mean(scrImg))
137  print ("Debug parameter_1: {}".format(average_val))
138
139  if average_val <= 50:
140      scrImg = cv2.inRange(scrImg, 50, 255)
141
142  else:
143      scrImg = cv2.equalizeHist(scrImg)
144
145      alpha = float(2.5)
146      scrImg = cv2.multiply(scrImg, np.array([alpha]))
147
148      v = np.median(scrImg)
149      sigma = 0.5
150      lower = int(max(0, (1.0 - sigma) * v))
151      upper = int(min(255, (1.0 + sigma) * v))
152      scrImg = cv2.inRange(scrImg, lower, upper)
153      scrImg = cv2.bitwise_not(scrImg)
154
155  cv2.imwrite('debug_03.jpg', scrImg)
```

Slika pohranjena u varijabli `scrImg` je ndarray red koji se sastoji od n ndarray redova od kojih svaki ima 300 cijelobrojnih elemenata i predstavlja jedan redak piksela u slici. Svaki piksel ove slike izražen je vrijednošću raspona od 0 do 255. Proces thresholdinga započinje računanjem srednje vrijednosti svih piksela u slici. Taj parametar se pohranjuje u varijablu `average_val`, a zadaća mu je realizacija uvjetovanja kojim se donosi odluka o vrsti snimljenog digitalnog prikaznika i osvjetljenju znamenki unutar istog. Digitalni prikaznici 7 – segmentnih zapisa najčešće se realiziraju u LCD ili LED tehnologiji. Primjer LCD prikaznika je i prikaznik sa slike (Slika 5.5). Za ovu vrstu prikaznika karakteristično je jače

pozadinsko osvjetljenje, što rezultira pozadinom svjetlijom od znamenaka. U slučaju digitalnih prikaznika realiziranih LED tehnologijom same znamenke su najčešće svjetleći segmenti većeg broja LED dioda, što znači da je pozadina zapisa u pravilu tamnija od znakova. U slučaju kada je srednja vrijednost iznosa intenziteta grayscale kanala slike manja od ~50, možemo sa sigurnošću pretpostaviti da je na slici puno više tamnih piksela nego svjetlih piksela, što upućuje na slučaj prikaznika realiziranog LED tehnologijom. Algoritam, u tom slučaju, zapisu slike sadržanom u `scrImg` varijabli primjenjuje unaprijed određene pragove 50 i 255 metodom `inRange` biblioteke OpenCV. Navedena metoda kao ulazni argument prima ndarray slike i zadane pragove, te svakom pikselu čija je vrijednost broj unutar opsega tih pragova, vrijednost postavlja u 255 (potpuno bijelo).

Ako je iznos srednje vrijednosti intenziteta piksela veći od uvjetovanih 50, pretpostavlja se da je na promatranoj slici sadržaj LCD digitalnog prikaznika. Postupak određivanja pragova binarizacjie, u ovom slučaju je složeniji, te podrazumjeva dvije dodatne prilagodbe promatranog sadržaja varijable `scrImg`. Pozadina ovako realiziranog zaslona (Slika 5.5), u grayscale formatu, u većini slučajeva je samo nekoliko nijanski sive boje svjetlija od znamenki. Primjenom funkcija `equalizeHist` i `multiply`, nad sadržajem promatrane slike normalizira se bistrina i povećava kontrast prikazanih oblika. Željene vrijednosti pragova binarizacije piksela pronalaze se postupkom izračuna mediana, korištenim i pri pronalasku pragova za potrebe funkcije Canny. Opis navedenog postupka nalazi se na stranici 17 ovog dokumenta. Metodologija takvog izračuna je u potpunosti primjenjiva i u ovom slučaju jer je potrebno odrediti pragove kojima se najviše naglašava kontrast između najistaknutijih objekata, tj. znamenki, i okoline. Aktivacijom funkcije `inRange` i primjenom izračunatih pragova, najtamniji pikseli postavljeni su u vrijednost 0, a svi ostali u vrijednost 255. Kako bi rezultantna slika bila sukladna slici iz drugog slučaja ovog uvjetovanja, potrebno je još izvršiti svojevrstan *bitwise not* postupak nad svim pikselima, efektivno mijenjajući bijele piksele crnima i obrnuto. Rezultat potpunog procesa binarizacije prikazan je slikom (Slika 5.6).

Slika 5.6 Binarizirani sadržaj slike prikaznika

Poslijednji korak predobrade i prilagodbe inicijalne slike prije početka postupka segmentacije znakovnog zapisa, podrazumjeva primjenu nekih morfoloških metoda dostupnih kroz biblioteku OpenCV. Zadaća takvih metoda je priprema binariziranog sadržaja promatrane slike u vidu uklanjanja neželjenih kontura i smetnji, te povezivanja svake znamenke u konzistentnu cjelinu bijelih piksela.

```
157  # cleaning up, dilating digits
158  kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (1, 6))
159  scrImg = cv2.morphologyEx(scrImg, cv2.MORPH_OPEN, kernel)
160
161  scrImg = erodeImg(scrImg, 3, 2, 1)
162  cv2.imwrite('debug_04.jpg', scrImg)
```

Željena funkcionalnost ostvaruje se morfološkim metodama biblioteke poput `morphologyEx`, te pozivom funkcije `erodeImg`, definirane u skripti *image_processing.py*. Morfološke metode ove biblioteke implementiraju se zadavanjem parametara poput zastavice načina rada, te zrna (eng. *Kernel*) proizvoljno zadanog oblika i dimenzija. Funkcionalnosti ovakvih morfoloških metoda utječu na područja slike visokog kontrasta većeg broja susjednih piksela (rub objekta), a odnose se na procese erozije ili proširivanja rubova promatranih objekata.

```
45   def erodeImg(img, dim1=2, dim2=2, iterations=1):
46
47   img2 = cv2.bitwise_not(img)
48   kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (dim1, dim2))
49   eroded = cv2.erode(img2, kernel, iterations)
50   eroded = cv2.bitwise_not(eroded)
51
52   return eroded
```

Programski sadržaj funkcije `erodeImg` predstavlja jednu dodatnu iteraciju primjene morfološkog efekta na promatranoj binariziranoj slici, prethodno opisane funkcionalnosti. Razlog odvojenosti ovog odsječka koda u zasebu funkciju je postizanje mogućnosti korištenja takve metode i u drugim dijelovima programskog sustava, ako je to potrebno. Postavljanje inicijalizirajućih parametara opisanih morfoloških metoda postupak je koji uvelike ovisi sadržaju aktivno promatrane slike, te je podložan mnogobrojnim korekcijama od strane korisnika.

Morfološkom korekcijom sadržaja binarizirane slike pohranjene u varijabli `scrImg`, promatrana slika digitalnog zaslona, zajedno sa traženim objektima, u potpunosti je prilagođena procesu lokalizacije znamenki. Rezultantna slika, nakon morfoloških korekcija prikazana je slikom (Slika 5.7).

Slika 5.7 Digitalni zaslon nakon morfoloških korekcija

Završni korak algoritma realiziranog programskim kodom skripte *image_processing.py* odnosi se na proces segmentacije prikazane vrijednosti. Navedeni proces obrade slike (Slika 5.7) pohranjene u varijabli `scrImg` podrazumjeva lokalizaciju svake znamenke promatranog zapisa, te pohranu tako izdvojene znamenke u elemente novokreirane liste, koji se na kraju posljeđuju algorotmima donošenja odluke.

```
166  imgCrop = scrImg.copy()
167
168  # detection of all white areas of input image as an individual
     contour
169  (cnts, _) = cv2.findContours(scrImg.copy(), cv2.RETR_EXTERNAL,
     cv2.CHAIN_APPROX_SIMPLE)
170
171  if len(cnts) > 0:
172      contours, _ = contour_sort(cnts, option = 0)
173
174  # for print purposes
175  scrImg = cv2.cvtColor(scrImg, cv2.COLOR_GRAY2BGR)
176  debugWriteImg = scrImg.copy()
```

Proces segmentacije započinje linijom 166 programskog koda skripte *image_processing.py*. Stvara se kopija slike prikazane slikom (Slika 5.7) koja se pohranjuje u varijablu `imgCrop`, a biti će korištena pri kraju algoritma za potrebe „izrezivanja“ lokaliziranog sadržaja. Sva cjelovita područja bijelih piksela na promatranoj slici, potom se spremaju u obliku vektora segmenata konture tog područja, primjenom algoritma funkcije `findContours`. Metodologija algoritma ove funkcije prethodne je opisana na stranici 19 ovog dokumenta. Sadržaj liste `cnts`, u idućem koraku sortira pozivom funkcije `contour_sort`. Navedena funkcija je također definirana unutar skripte *image_processing.py*, a opisuje ju priloženi segment programskog koda.

```
35   def contour_sort(cnts, option):
36
37   if option == 0:
38       boundingBoxes = [cv2.boundingRect(i) for i in cnts]
39       (cnts, boundingBoxes) = zip(*sorted(zip(cnts, boundingBoxes),
                                  key=lambda b: b[1][0],
                                  reverse = False))
40       return cnts, boundingBoxes
41   elif option == 1:
42       cnts = sorted(cnts, key = lambda x: x[0], reverse = False)
43       return cnts
```

Osnovna funkcionalnost ovog segmenta programskog koda je postupak sortiranja elemenata liste `cnts`, u ovisnosti o položaju konture koju pojedini element predstavlja unutar slike, s lijeva na desno. Kako bi se tražena funkcionalnost uspješno realizirala potrebno je poznavati koordinate kontura unutar liste `cnts`, što se ostvaruje primjenom funkcije `boundingRect`, biblioteke OpenCV. Navedena funkcija kao ulazni argument prima element liste `cnts`, tj. vektor koordinata svih pikslela konture, te za razmatranu konturu „pronalazi“ pravokutnik koji ju opisuje (eng. *Bounding Rectangle*). Opisni pravokutnik se zapisuje kao Python lista sa 4 elementa: x i y koordinate gornjeg lijevog ugla tog pravokutnika, te njegova širina i visina. Lista opisnih pravokutnika svih kontura liste `cnts`, zapisuje se u listu `boundingBoxes`. Iteracijom kroz tu listu, uz postavljanje odgovarajućeg uvjeta, sve konture se sortiraju od lijeve strane prikaznika prema desno. Zbog promjene zapisa kontura u nastavku programskog koda algoritma segmentacije, funkcija `contour_sort` omogućuje dva načina rada, potpuno jednake funkcionalnosti.

Nakon procesa sortiranja elemenata liste `cnts`, u linijama 175 i 176 provode se dvije manje prilagodbe razmatrane slike prikaznika (Slika 5.7), potrebne pri kasnijem ispisu za potrebe debugginga.

Algoritam segmentacije slike provodi se kroz nekoliko faza, tj. iteracija kontura bojelih piksela detektiranih na razmatranoj slici. Svrha tih provjera je pronalazak potencijalnih kandidata za znamenke u skupu kontura kojima raspolažemo, te odbacivanje neželjenih kontura za koje se ustanovi da sigurno nisu traženi znakovi. Željeni rezultati se postižu implementacijom više serija uvjetovanja koja se prvenstveno odnose na položaj i dimenzije razmatranih kontura.

Serija eliminacijskih uvjetovanja implementirana u algoritmu ovog projekta započinje provjerom potencijalnih bijelih područja u krajnjim uglovima razmatrane binarizirane slike digitalnog prikaznika. Pojava neželjenih bijelih područja u uglovima veoma je česta u slučajevima snimanja digitalnog prikaznika iz kojega se želi očitati vrijednost. Razlozi nastanka ovakvih smetnji najčešće su uvjeti osvjetljenja u kojima je ulazna fotografija snimljena ili refleksija u staklu digitalnog prikaznika, što je česta pojava kod LCD prikaznika. Zbog tako prouzrokovane visoke razlike u osvjetjenju zaslona i visokog kontrasta nijansi pozadinskih boja

prikaznika, algoritam binarizacije ne izbacijue takva područja, a zbog veće površine to se ne postigne ni morfološkim metodama. Traženi alfanumerički zapis u velikoj većini slučajeva nije pozicioniran u ekstremnoj blizini uglova prikaznika. Priloženi odsječak programskog koda provjerava krajnje uglove razmatrane slike, koristeći koordinate uglova opisnog pravokutnika svake konture, dobivenih metodom `boundingRect`. Sve konture izvorne liste koje zadovolje ovo uvjetovanje spremaju se u novoinicijaliziranu listu `contours2`.

```
178  height, width = scrImg.shape[:2]
179  contours2 = []
180
181  # removing white areas in corners
182  for i in contours:
183
184      [x, y, w, h] = cv2.boundingRect(i)
185      if ((0 <= x <= 2 and 0 <= y <= 2) or
186          (width-2 <= (x+w) <= width and 0 <= y <= 2) or
187          (width-2 <= (x+w) <= width and height-2 <= (y+h) <= height) or
188          (0 <= x <= 2 and height-2 <= (y+h) <= height)):
189           continue
190  else:
191      contours2.append(i)
192      cv2.rectangle(debugWriteImg, (x,y), (x+w,y+h), (255, 0, 0), 2)
```

Sadržaj novokreirane liste `contours2` nakon opisanog postupka eliminacijskog uvjetovanja za slučaj prikaznika sa slike (Slika 5.7), grafički je ilustriran slikom (Slika 5.8).

Slika 5.8 Konture pohranjene u listi contours2

Drugi korak serije procesa uvjetovanja algoritma segmentacije digitalnog zapisa i lokalizacije znamenaka odnosi se na postupak pronalaska kontura koje s najvećom vjerojatnošću predstavljaju oblik traženog zapisa, te eliminacije svih ostalih kontura iz razmatranja.

```
194  digitBox = []
195  decimalBox = []
196
197  digit_aspect = 0.6
198  digit_one_aspect = 0.2
199  aspect_buffer = 0.15
200
201  max_height = 0
202  max_width = 35
203  max_width_one = 10
204  y_min = 1000
205  for i in contours2:
206
207      [x, y, w, h] = cv2.boundingRect(i)
208      aspect = float(w) / h
209      area = w * h
```

Za potrebe cijelog niza uvjetovanja implementiranih u ovom koraku potrebno je inicijalizirati vrijednosti odrednica na osnovu kojih će uvjeti biti postavljeni. Inicijaliziraju se dvije prazne Python liste `digitBox`, u koju će se spremati konture koje zadovolje postavljene uvjete i `decimalBox` namjenjenu konturama koje zadovolje posebno uvjetovanje prilagođeno decimalnoj točki. Nadalje, postavljaju se očekivane vrijednosti omjera dimenzija (širine i visine) potencijalnih znamenki i potencijalnih jedinica, varijablama `digit_aspect` i `digit_one_aspect`. Znamenka broja 1 u ovome je procesu uvjetovanja tretirana kao poseban slučaj s obzirom da je, u standardnom 7 – segmentnom prikazu, širina znamenke znatno manja od širine ostalih znamenaka. Varijablom `aspect_buffer` određen je pojas tolerancije odstupanja dimenzijskog omjera potencijalnih znamenki. Varijable `max_height`, `max_width`, `max_width_one`, te `y_min` postavljene su u inicijalne vrijednosti dimenzija i položaja kontura koje budu prepoznate kao potencijalne znamenke. Vrijednosti tih varijabli biti će ažurirane pri pronalasku zadovoljavajućih kontura, a primjenu pronalaze u nastavku algoritma. U liniji 178 bilježe se i iznosi broja piksela širine i visine cijele razmatrane slike (Slika 5.7), također potrebni u nastavku algoritma. Parametre opisane u ovom segmentu programskog koda moguće je inicijalizirati i u zasebnoj skripti, te po potrebi

učitavati u ostale skripte cjelokupnog sustava. U ovom primjeru inicijalizacija je zadržana unutar programskog koda algoritma zbog bolje preglednosti nad obrađivanim podatcima.

Proces uvjetovanja raspoloživih kontura realiziran je `for` petljom koja iterira sadržajem liste `contours2` primjenjujući seriju uvjeta na svaki element te liste, tj. svaku konturu. Početak svake iteracije odnosi se na izračun vrijednosti koje opisuju razmatranu konturu, čiji iznos se provjerava u narednim uvjetima. Ti iznosi podrazumjevaju koordinade gornjeg lijevog ugla i dimenzije pravokutnika opisanog konturi, dimenzijski omjer tog pravokutnika, te njegovu površinu.

```
211  # possible decimal points
212  if area > 5 and area < 250
                 and aspect >= 1 - aspect_buffer
                 and aspect <= 1 + aspect_buffer:
213      decimalBox.append([x, y, w, h])
214      cv2.rectangle(scrImg, (x,y), (x+w,y+h), (0, 255, 0), 2)
```

Prvi postavljeni uvjet odnosi se na provjeru potencijalnih decimalnih točka među konturama liste `contours2`. Svaka kontura čiji opisani pravokutnik ima relativno malu površinu, te teži kvadratnom obliku (uz dozvoljeno odstupanje `aspect_buffer`) može se smatrati potencijalnom decimalnom točkom promatranog znakovnog zapisa. Važno je primjetiti kako se u novodeklariranu listu `decimalBox` u ovom slučaju sprema Python lista s podatcima o koordinatama gornjeg lijevog ugla, te dimenzijama opisanog pravokutnika, dobivena pozivom funkcije `boundingRect`. Ovaj format podataka o opisanom pravokutniku, zauzima mnogo manje memorije, te je jednostavniji od formata zapisa funkcije `findContours`, stoga će se koristiti u ostatku algoritma. Opisani pravokutnik koji zadovoljava zadani uvjet, iscrtava se na razmatranu sliku, za potrebe debugginga.

```
216  # eliminating white areas that cannot be digits
217  if area < int((height * 0.5)*5) and
        (aspect < 1 - aspect_buffer or aspect > 1 + aspect_buffer):
218      continue
```

Sljedeći u nizu uvjeta odnosi se na selekciju kontura liste `contours2` koje nikako ne mogu biti znamenka ili decimalna točka. Navedenom uvjetu odgovaraju konture opisnog pravokutnika izrazito male površine i dimenzija koje od kvadratnog oblika odstupaju više od dozvoljenog `aspect_buffer` pojasa. Takve konture se izbacuju iz razmatranja.

```
220  # possible digits
221  if (area > int((height * 0.5)*25) and
          aspect >= digit_aspect - aspect_buffer and
          aspect <= digit_aspect + aspect_buffer):
222          if h > max_height: max_height = h
223          if w > max_width: max_width = w
224          if y < y_min: y_min = y
225          digitBox.append([x, y, w, h])
226          #cv2.rectangle(debugWriteImg, (x,y), (x+w,y+h), (255,
     0, 0), 2)
```

Treća u nizu postavljenih provjera, svake iteracije navedene `for` petlje, definira uvjete površine i dimenzija opisnog pravokutnika koji odgovaraju očekivanoj znamenci. Iznos očekivane površine je relativan u odnosu na visinu cijelog digitalnog prikaznika. U ovom konkretnom primjeru ta vrijednost je polovica visine prikaznika. Očekivani dimenzijski omjer zadovoljavajuće znamenke uvjetovan je varijablom `digit_aspect` i dozvoljenim pojasom odstupanja `aspect_buffer`. Za potrebe ovog projekta, zadana vrijednost dimenzijskog omjera iznosi 0.6 (linija 197). Kada pojedina kontura zadovolji zadani uvjet, ažuriraju se vrijednosti varijabli `max_height`, `max_width` i `y_min`. Podatci o opisanom pravokutniku zadovoljavajuće konture spramaju se u listu `digitBox`, a pravokutnik se iscrtava u izvornoj slici, zbog potreba debugginga.

Posljednja provjera implementirana unutar navedene `for` petlje odnosi se na detekciju kontura koje potencijalno predstavljaju 7 – segmentnu znamenku broja jedan. Razlog potrebe za implementacijom posebnog uvjetovanja koje se odnosi samo na znamenku broja jedan je značajna razlika u promatranim parametrima, u odnosu na ostalih devet znamenaka. 7 – segmentna znamenka broja jedan jedina je znamenka tog standarda koja se ne sastoji niti od jednog horizontalnog segmenta. To znači da je takva znamenka u pravilu je mnogo uža od ostalih, što rezultira i manjom površinom. Iz tog razloga znamenke broja jedan ne zadovoljavaju ranije opisani uvjet koji se odnosi na preostale znamenke.

```
228 # possible digit one
229 if (area > int((height * 0.5)*5) and
         aspect >= digit_one_aspect - aspect_buffer and
         aspect <= digit_one_aspect + aspect_buffer):
230         if h > max_height: max_height = h
231         if w > max_width_one: max_width_one = w
232         if y < y_min: y_min = y
233         digitBox.append([x, y, w, h])
234         #cv2.rectangle(debugWriteImg, (x,y), (x+w,y+h), (255,
    0, 0), 2)
```

Metodologija zadanog uvjetovanja identična je slučaju prethodno opisanog segmenta programskog koda namjenjenog preostalim znamenkama, uz napomenu da se u slučaju znamenke broja jedan iznos širine zadovoljavajuće konture pohranjuje u zasebnoj varijabli `max_width_one`. Opisni pravokutnik se dodaje listi `digitBox` i iscrtava za potrebe debugginga.

Prethodno opisanim postupkom analize sadržaja liste `contours2` odbacuju se smetnje i ostvaruje selekcija svih poželjnih kontura promatrane slike, čiji su opisani pravokutnici po završetku izvođenja prethodne `for` petlje pohranjeni u listama `digitBox` i `decimalBox`. Prethodno opisan niz uvjetovanja može se smatrati „centralnim" dijelom algoritma segmentacije zapisa i lokalizacije individualnih znamenaka. Ostatak programskog koda skripte *image_processing.py* odnosi se na nekoliko iteracija dodatnih provjera detektiranih kontura i konačno pozivanje algoritama donošenja odluke o vrijednosti prepoznatih kontura.

Osnovni nedostatak koji se kroz sljedećih nekoliko odsječaka programskog koda uklanja iz algoritma je greška do koje dolazi kada se u procesu binarizacije i morfološke modifikacije sadržaja promatranog prikaznika ne ostvari potpuna međusobna povezanost svih segmenata, svake znamenke. Naime, s obzirom da osnovna funkcija detekcije kontura (`findContours)` potpuno bijelih područja na crnoj pozadini, u rezultantnu listu zasebno pohranjuje svaki neprekinuti oblik bijele boje, cilj je u zapisu slike (Slika 5.7) postići potpunu povezanost svih segmenata pojedine znamenke. Iz priloženog primjera vidiljivo je kako su u ovom slučaju morfološke funkcije uspješno modificirale rubove bijelih segmenata i znamenke su zbog toga cjeloviti objekti. Ovakav rezultat nije uvijek slučaj. Implementirane morfološke funkcije rade sukladno zadanim parametrima, ranije opisanom metodologijom (str. 24.). Prilikom postavljanja vrijednosti tih parametara, potrebno

je prvenstveno voditi računa o svim neželjenim pojavama i smetnjama na slici, koje pogrešno postavljene morfološke funkcije mogu spojiti sa korisnim sadržajem i tako onemogućiti daljnju obradu. Slučajeve razdvojenih segmenata stoga korigira programski kod opisan u nastavku.

```
238  # detection of digits with unconnected segments
239  for i in contours2:
240
241      [x, y, w, h] = cv2.boundingRect(i)
242      flag = False
243      for j in digitBox:
244          if [x, y, w, h] == j: flag = True
245
246      if flag is True: continue
247
248      if (y >= y_min - 5 and y <= y_min + 5):
249          if (w >= h):
250              w = max_width
251              h = max_height
252          if (w < h):
253              if w <= max_width_one: w = max_width_one
254              elif (max_width_one < w <= max_width_one + 5): w =
     max_width_one
255              else: w = max_width
256              h = max_height
257          if (x + max_width >= width or x + max_width_one >= width):
258              w = width - x - 3
259          digitBox.append([x, y, w, h])
```

Proces povezivanja razdvojenih segmenata ostvaren je programskim kodom kroz tri `for` petlje. Prva u nizu petlja provjerava postojanje manjih kontura unutar pojasa vrijednosti na veritkalnoj osi slike, određenih vrijednostima `y_min` i `y_min + max_height`. Ranije u ovom tekstu je objašnjen zapis učitane slike u odgovarajući ndarray red. Važno je napomenuti da je ishodište zamišljenog koordinatnog sustava slika interpretiranih na ovaj način, piksel gornjeg lijevog ugla slike. Horizontalna os je pozitivna desno od ishodišta i predstavlja širinu slike, dok je vertikalna os pozitivna u smjeru prema dolje, te predstavlja visinu slike. Vrijednost varijable `y_min` postavljena u prethodnom segmentu koda, predstavlja najnižu vrijednost vertikalne osi (najvišu razinu na slici) na kojoj je detektirana potencijalna znamenka. Podrazumjeva se da je svaki traženi zapis više 7 – segmentnih znamenki u istoj ravnini, stoga se očekuje da se sve znamenke nalaze u pojasu vertikalne osi između `y_min` i `y_min + max_height`. Priložena `for` petlja iterira sadržajem liste `contours2` i uspoređuje svaki element sa ranije

detektiranim segmentima koji predstavljaju potencijalne znamenke. U slučaju da je element iteracije jedan od ranije prepoznatih elemenata, korak se preskače i petlja prelazi na sljedeći element. Svaki od elemenata liste `contours2` koji ranije nije zadovoljio uvjete, prolazi kroz novi niz uvjetovanja čija je zadaća utvrditi je li razmatrani element odvojeni segment tražene znamenke. Primjer rezultata izvođenja korekcijskog algoritma implementiranog ovom petljom, prikazan je slikom (Slika 5.9). Iz slike i priloženog koda je vidljivo da su vrijednosti dimenzija opisnog pravokutnika odvojenog segmenta, zamjenjene najvećom vrijednošću visine i najvećom vrijednošću širine potencijalne znamenke i dodane listi `digitBox`. Redundantni elementi liste `digitBox`, nastali ovako realiziranom korekcijom biti će uklonjeni u nastavku algoritma.

Slika 5.9 Povezivanje razdvojenih segmenata

Druga u nizu korekcijska `for` petlja ostvaruje sličnu funkcionalnost povezivanja segmenata, ali se odnosi isljučivo na slučaj kada je tražena znamenka razdvojena horizontalno, po sredini. Takva komplikacija se pokazala vrlo učestalom pojavom tijekom razvoja ovog programskog sustava, te je za potrebe korekcije razvijen slijedeći programski odsječak.

```
261  for i, value in enumerate(digitBox):
262      [x, y, w, h] = value
263
264      for k in contours2:
265          [xk, yk, wk, hk] = cv2.boundingRect(k)
266          if (y + h <= yk <= y + h + 3):
267              h = h * 2
268              digitBox[i] = [x, y, w, h]
```

Za svaki element liste `digitBox`, koji potencijalno predstavlja polovicu tražene znamenke, algoritam ispituje položaje svih kontura sadržanih u listi `contours2`. U slučaju pronalaska konture pozicionirane točno ispod razmatrane konture polovičnog obuhvata, visina razmatrane konture se uduplava, efektivno obuhvaćajući i dio znamenke koji je nedostajao. Rezultat izvođenja ovog programskog odsječka na primjeru jedne znamenke, prikazan je slikom (Slika 5.10).

Slika 5.10 Spajanje polovično obuhvaćene znamenke

Posljednji segment programskog koda sa korekcijskom ulogom do sada obrađenih kontura obavlja zadatak uklanjanja nepotrebnih kontura iz liste `digitBox`, koje su izvršavanjem prethodno opisanih dviju korekcijskih petlji postale redundantne. Algoritam je prikazan sljedećim programskim odsječkom:

```
270 # removing excess contours
271 digitBox_final = []
272 for i in digitBox:
273     [x, y, w, h] = i
274 
275     flag = False
276     for j in digitBox:
277 
278         if j == i: continue
279         [xj, yj, wj, hj] = j
280 
281         if ((xj <= x <= xj + wj and yj <= y <= yj + hj) or
282             (xj <= x + w <= xj + wj and yj <= y <= yj + hj) or
283             (xj <= x + w <= xj + wj and yj <= y + h <= yj + hj) or
284             (xj <= x <= xj + wj and yj <= y + h <= yj + hj) and
285              w * h < wj * hj):
286             flag = True
287 
288     if flag is False:
289         digitBox_final.append(i)
```

Prethodno navedeni redundantni elementi liste `digitBox` predstavljaju pojedinačne, tj. nepovezane segmente tražene znamenke. Problematična znamenka je, nakon izvršavanja korekcijskih postupaka, u listi `digitBox` pohranjena kao novi element, opisani pravokutnik, koji sada obuhvaća cijelu znamenku. Priloženi programski odsječak svaki element liste `digitBox` položajno uspoređuje sa svim elementima te iste liste. Nizom postavljenih uvjeta utvrđuje se pripada li promatrani element, tj. opisni pravokutnik konture koju on predstavlja, površini obuhvaćenoj nekim drugim elementom. Svi elementi za koje se ovom provjerom ustanovi da nisu obuhvaćeni elementom većih dimenzija, dodaju se novoinicijaliziranoj listi `digitBox_final`.

Provedbom svih prethodno opisanih lokalizacijskih i korekcijskih postupaka, algoritam segmentacije rezultira konturama pohranjenima u listi `digitBox_final`. Grafički prikaz sadržaja te liste (plavo) i sadržaja liste `decimalBox` (zeleno), na primjeru slike mjerača goriva, dan je slikom (Slika 5.11). Programski kod razvijen za potrebe ovog demonstrativnog primjera, podešen je tako da bilježi dimenzijski najveće znamenke (u odnosu na prikaznik), s obzirom da velik broj prikaznika mjernih uređaja ostvaruje ispis u samo jednom redu. Uz istu metodologiju ,ali manje promjene regulatornih parametara, moguće je ostvariti prepoznavanje znamenaka drugačijih dimenzija, ovisno o potrebama mjerenja.

Slika 5.11 Rezultat algoritma segmentacije

Algoritam obuhvaćen skriptom *image_processing.py* u ovom stadiju raspolaže sa potpuno lokaliziranim konturama razmatranog zapisa. Preostaje integrirati funkcionalnost algoritama donošenja odluke o vrijednosti pojedine znamenke, te ispisati konačne rezultate na korisničko sučelje.

```
295 digitBox_final = contour_sort(digitBox_final, option = 1)
296 output = []
297 outputNet = []
298 outputKnn = []
299 for i, value in enumerate(digitBox_final):
300
301     [x, y, w, h] = value
302
303     singleDigit = imgCrop[y:y + h, x:x + w]
304     # decision by pattern
305     output += dl.lookup(singleDigit, max_width_one)
306     # decision by network
307     outputNet += run_network(singleDigit)
308     # decision by knn algorithm
309     outputKnn += run_knn(singleDigit)
310
311     # first decimal point check
312     output += dl.decimal_lookup(singleDigit)
313     outputNet += dl.decimal_lookup(singleDigit)
314     outputKnn += dl.decimal_lookup(singleDigit)
315     # additional decimal point check
316     for j in decimalBox:
317
318         [xj, yj, wj, hj] = j
319         if (y_min + max_height - 5 <= yj + hj <= y_min + max_height
                 + 5):
320             if (value == digitBox_final[-1] and x < xj):
321                 if (output[-1] != "."): output.append(".")
322                 if (outputNet[-1] != "."): outputNet.append(".")
323                 if (outputKnn[-1] != "."): outputKnn.append(".")
324             elif (value != digitBox_final[-1] and x < xj <
                       digitBox_final[i+1][0]):
325                 if (output[-1] != "."): output.append(".")
326                 if (outputNet[-1] != "."): outputNet.append(".")
327                 if (outputKnn[-1] != "."): outputKnn.append(".")
328
329     singleDigit = cv2.bitwise_not(singleDigit)
330     singleDigit = ut.resize(singleDigit, height = 125)
331     cv2.imwrite(folder + '/digit_' + str(i) + '.jpg', singleDigit)
332
333     cv2.rectangle(scrImg, (x,y), (x+w,y+h), (255, 0, 0), 2)
```

Proces integracije metoda donošenja odluke o vrijednosti individualnih znamenkaka ostvaruje se jednom iteracijom svih vrijednosti sadržanih u listi `digitBox_final`. Ta lista se prethodno ovom procesu, još jednom sortira, u ovisnosti o kordinatama opisnih pravokutnika, sa lijeva na desno. Sortiranjem se potvrđuje sigurnost ispravnog poretka pojedinačnih kontura znamenaka promatranog zapisa. S obzirom da su u okviru ovog projekta implementirana tri različita algoritma donošenja odluke o vrijednosti znamenke prikazane slikom, prethodno procesu inicijalizacije tih motoda potrebo je definirati i tri prazne liste. Te liste će sadržavati znakovne elemente tipa string, koji će pretstavljati prepoznatu

vrijednost pojedine znamenke, te predstavljaju konačni, rezultantni skup podataka cjelokupnog razvijenog programskog sustava.

Proces obrade svakog elementa liste digitBox_final započinje „izrezivanjem“ područja određenog koordinatama opisnog pravokutnika odgovarajuće znamenke, iz inicijalne, binarizirane slike prikaznika, prikazane slikom (Slika 5.7). Svako područje izdvojeneo na ovaj način, predstavlja novu sliku čiji se ndarray zapis pohranjuje u varijablu `singleDigit`. Individualizirana znamenka se kroz sljedeće tri linje programskog koda predaje metodama `lookup`, `run_network` i `run_knn` kao ulazni parametar. Navedene metode predstavljaju algoritme donošenja odluke o vrijednsoti znamenke, čija je metodologija opisana u narednim poglavljima. Izlazne vrijednosti tih metoda su podatci tipa string, koji se dodaju prethodno inicjaliziranim rezultantnim listama.

Sljedeći korak procesa obrade lokaliziranih znamenaka odnosi se na provjeru postojanja decimalne točke unutar promatranog zapisa. Prilikom ispitivanja postojanja decimalne točke implementirana su dva „sloja“ provjere. Prva provjera je ostvarena metodom programskog koda skripte *digit_lookup.py*, čija metodoloija je opisana u narednim poglavljima. Druga provjera se odnosi prvenstveno na sadržaj liste `decimalBox`, te ostvaruje korekcijsku ulogu neuspjelih pronalazaka prvog pokušaja. Ukoliko se unutar navedene liste nalazi nekoliko kontura potencijalnih decimalnih točaka, algoritam provjerava položaj detektiranih kontura u odnosu na položaj trenutno razmatrane znamenke. U slučaju da se razmatrana kontura iz liste `decimalBox` nalazi na očekivanom položaju, a prethodni „sloj“ detekcije nije zabilježio decimalnu točku na tom mjestu, u rezultantnu listu dodaje se decimalna točka.

Posljednjih nekoliko linija prikazanog programskog odsječka, odnosi se na pohranu slika lokaliziranih znamenki u direktorij *debug_znamenke*, u svrhsi debugginga.

Programska funkcionalnost implementirana u skripti opianoj u okviru ovog poglavlja, *image_processing.py* završava implementacijom ispisa konačnih rezultata programskog sustava pohranjenih u listama `output`, `outputNet`, te `outputKnn`, na korisničko sučelje računalnog sustava.

```
335  # printing the results
336  output = ''.join(output)
337  print ("\nValue detected by pattern: {}".format(output))
338  outputNet = ''.join(outputNet)
339  print ("\nValue detected by network algorithm: {}".format(outputNet))
340  outputKnn = ''.join(outputKnn)
341  print ("\nValue detected by knn algorithm: {}".format(outputKnn))
```

Programska skripta *image_processing.py* središnji je element računalnog sustava razvijenog u sklopu ovog projekta. Obuhvaćajući sve funkcionalnosti implementirane programskim odsječcima pohranjenim u ostalim skriptama sustava (Slika 3.2), te ostvarivanjem serije složenih procesa obrade ulazne fotografije, opisani programski sadržaj predstavlja jezgru sustava i podlogu za sve moguće modifikacije i proširenja u budućnosti.

5.1. Biblioteka pomoćnih funkcionalnosti

Programska podrška sustava razvijenog u okviru ovog projekta na nekoliko lokacija unutar arhitekture programskog sustava (Slika 3.2) primjenjuje seriju istovjetnih funkcionalnosti. Implementacija tih konkretnih funkcionalnosti ostvarena je programskim kodom skripte *utilities.py*.

Unutar ovog potpoglavlja opisana je programska implementacija funkcija objedinjenih navedenom skriptom. Prilikom razvoja svih opisanih funkcionalnosti, korišteni su programski alati biblioteka OpenCV i NumPy. Potpuni programski kod opisanih funkcija nalazi se u privitku ovog dokumenta.

Funkcija rotacije slike

Funkcija rotacije slike predstavlja programsku implementaciju funkcionalnosti rotiranja slike, koja je memorijski zapisana u obliku prethodno opisanog dvodimenzionalnog NumPy ndarray reda. Navedena funkcija je unutar biblioteke *utilities.py* definirana imenom `rotate_crop`. Funkcija kao ulazne parametre prima ndarray zadane slike i iznos željenog kuta rotacije (u stupnjevima). Za zadanu sliku se, na osnovu dimenzija, izračunavaju koordinate centralnog piksela. Taj podatak se prosljeđuje metodi `getRotationMatrix2D` biblioteke OpenCV. Navedena metoda izvršava preslagivanje svih elemenata ndarray reda u skladu sa željenim kutem rotacije (suprotno smjeru kazaljke na satu), te rezultira odgovarajućom transformacijskom matricom zadane slike. Sadržaj matrice se na kraju „prepisuje“ preko sadržaja ulaznog reda pozivom funkcije `warpAffine` biblioteke OpenCV, efektivno rotirajući ulaznu sliku, uz zadržavanje početnih dimenzija.

Funkcija promjene dimenzija slike

Ova funkcija biblioteke *utilities.py* predstavlja programsku implementaciju kratkog algoritma namjenjenog promjeni dimenzija (visine i širine) zadane slike, uz očuvanje izvornog omjera tih dimenzija. Unutar biblioteke *utilities.py*, navedena funkcija je definirana imenom `resize`. Funkcija kao ulazne parametre prima dvodimenzionalni ndarray red zadane slike, proizvoljan iznos željenih dimenzija, tj. visine ili širine. Serija uvjetovanja provjerava koji od dva moguća dimenzijska

parametra je zadan te na osnovu tog podatka računa iznos preostalog parametra proporcionalan izvornoj dimenziji. Proširenje ulaznog ndarray reda se ostvaruje pozivom funkcije `resize` biblioteke OpenCV, kojoj se kao ulazni parametri predaju ndarray izvorne slike i izračunate vrijednosti novih dimenzija.

Funkcija prilagodbe perspektive

Navedena funkcija predstavlja programsku implementaciju postupka izdvajanja zadanog područja iz izvorne slike i vertikalizacije perspektive pogleda na tako izdvojeno područje (pogled „ptičje perspektive“). Unutar biblioteke *utilities.py*, ova funkcija je definirana imenom `transform_perspective`. Funkcija kao ulazne parametre prima ndarray izvorne slike i listu koordinata četiriju točaka, koje definiraju uglove odabranog područja interesa. Pretpostavlja se da je područje interesa pravokutnog oblika. Lista koordinata se u prvoj liniji programskog koda funkcije `transform_perspective` obrađuje pozivom lokalne funkcije `order_points`, također realizirane unutar skripte *utilities.py*. Zadaća ove funkcije je osigurati konzistentnost redosljeda zadanih točaka ulazne liste. Jednostavni programski odsječak funkcije `order_points` preslaguje ulazu listu zadanih točaka primjenom metode zbrajanja i oduzimanja x i y koordinate. Zadani redosljed točaka je: gore – lijevo, gore – desno, dolje – desno, dolje – lijevo. To znači da prva toča ima najmanju sumu x i y koordinata, a treća točka najveću. Jednako tako, druga točka ima najmanju razliku koordinata x i y, a četvrta točka najveću. Pravilno posložene točke uglova područja interesa vraćaju se algoritmu funkcije `transform_perspective`. Algoritam računa najveću apsolutnu udaljenost y koordinata zadanih ulaznih točaka, što postaje željena dimenzija visine, te najveću apsolutnu udaljenost x koordinata ulaznih točaka, što postaje željena dimenzija širine. Poznavajući dimenzije slike koja će nastati ekstrakcijom područja interesa, definiraju se koordinate odredišta, tj. uglovi te nove slike (konzistentnim redosljedom!). Algoritam završava pozivom funkcije `getPerspectiveTransform` biblioteke OpenCV, koja iz posloženih zadanih točaka, i novodefiniranih uglova rezultantne slike, računa transformacijsku matricu. Pozivom funkcije `warpPerspective`, izvorna slika se modificira na osnovu matrice, što rezultira slikom područja interesa, prilagođene perspektive.

6. Proces donošenja odluke o sadržaju slike

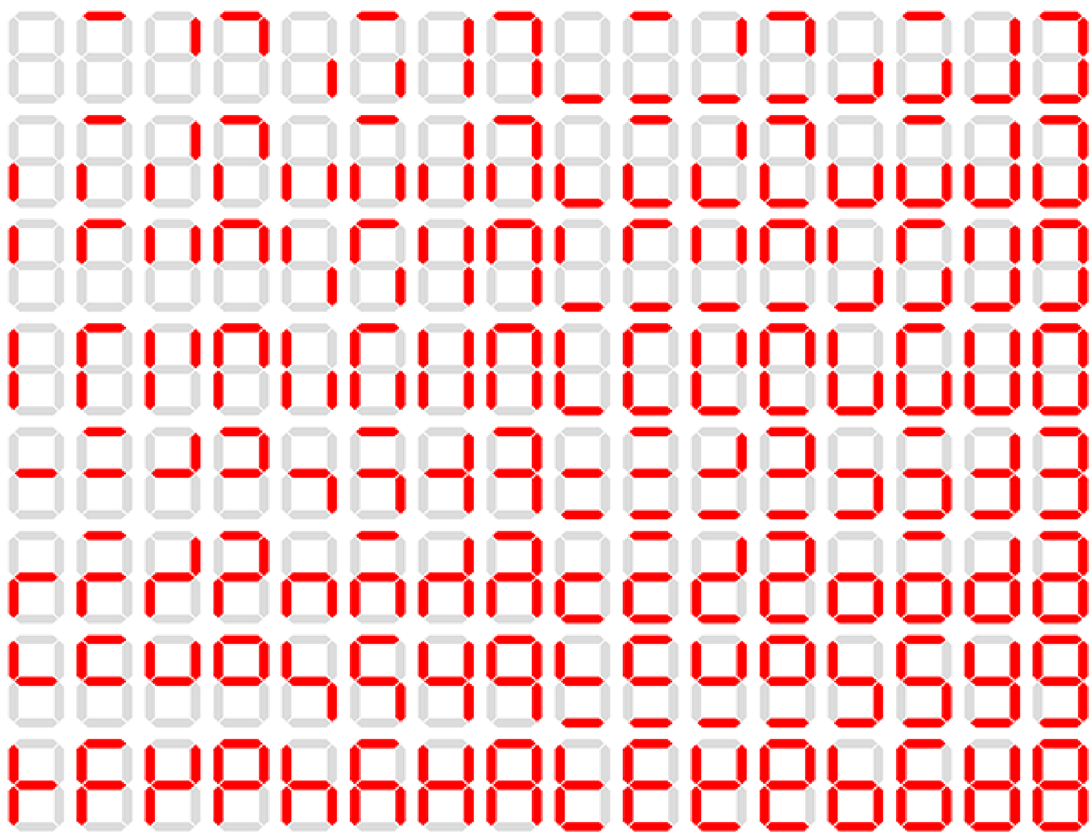
Programska podrška računalnih sustava namjenjenih prepoznavanju vrijednosti alfanumeričkih zapisa unutar slike, kako je prethodno opisano, podrazumjeva dva osnovna procesa analize ulaznih podataka. Izvršavanjem algoritma programske skripte *image_processing.py*, opisane u prethodnom poglavlju, u potpunosti je završen proces predobrade učitane fotografije i segmentacije zapisa. Računalni sustav tada raspolaže sa uspješno lokaliziranim, te individualno pohranjenim slikama pojedinačnih znamenki promatranog numeričkog zapisa. Primjer rezultantnog sadržaja programske skripte *image_processing.py*, ostvarenog obradom sadržaja slike (Slika 5.1), prikazan je slikom (Slika 6.1).

Slika 6.1 Sadržaj direktorija debug_znamenke

Potpuna funkcionalnost svakog računalnog sustava ovog tipa, ostvaruje se implementacijom odgovarajućeg algoritma prepoznavanja vrijednosti sadržaja slike lokalizirane znamenke. U okviru ovog projekta implementirana su tri različita algoritma koja ostvaruju ovu zadaću. U ostatku teksta ovog poglavlja, opisane su funkcionalnosti korištenih algoritama, te način njihove integracije u programski sustav.

6.1. Prepoznavanje standardnim predloškom

Prepoznavanje vrijednosti sadržaja slike primjenom prethodno definiranog predloška jedna je od najjednostavnijih metoda očitavanja vrijednosti 7 – segmentnih alfanumeričkih znakova. S obzirom na visoko standardizirani izgled znamenaka 7 – segmentnog formata (Slika 3.1), izradom jednostavnog predloška moguće je ispitivati razine stanja aktivnosti pojedinog segmenta. Standardni format 7 – segmentnog prikaza zauzima ukupno 128 različitih stanja, prikazanih slikom (Slika 6.1.1).



Slika 6.2 128 stanja 7 - segmentnog prikaza [3]

Funkcionalnost prepoznavanja vrijednosti prikazane pojedinom rezultantnom slikom (Slika 6.1) prethodno opisanog algoritma, skripte *image_processing.py*, programski je implementirana kratkim algoritmom programske skripte imena *digit_lookup.py*. Razvijeni algoritam izvršava zadatak prepoznavanja vrijednosti

znamenke na učitanoj slici i provjere postojanja decimalne točke, neposredno uz tu znamenku. Funkcionalnost algoritma ograničena je na ispitivanje 10 različitih stanja sa slike (Slika 6.1.1), koja odgovaraju dekadskim znamenkama 0 – 9.

```
1     #!/usr/bin/python
2
3     import sys
4     import cv2
5     import numpy as np
6
7     LOOKUP_TABLE = {
8         (1, 1, 1, 0, 1, 1, 1): 0,
9         (1, 1, 1, 1, 1, 1, 1): 1,
10        (1, 0, 1, 1, 1, 1, 0): 2,
11        (1, 0, 1, 1, 0, 1, 1): 3,
12        (0, 1, 1, 1, 0, 1, 0): 4,
13        (1, 1, 0, 1, 0, 1, 1): 5,
14        (1, 1, 0, 1, 1, 1, 1): 6,
15        (1, 0, 1, 0, 0, 1, 0): 7,
16        (1, 1, 1, 1, 1, 1, 1): 8,
17        (1, 1, 1, 1, 0, 1, 1): 9
18    }
```

Programski sadržaj algoritma skripte *digit_lookup.py* započinje prethodnim odsječkom. Nakon uobičajne inicijalizacije potrebnih alata definiran je Python riječnik `LOOKUP_TABLE` koji čini referentni predložak korišten u ostatku algoritma. Ključevi ovog riječnika su n – torke od 7 elemenata vrijednosti 0 ili 1, koji predstavljaju aktivnost pojedinog segmenta, pri čemu 1 znači da je segment upaljen. Redosljed elemenata svake n – torke odgovara segmentima sa slike (Slika 3.1): A, F, B, G, E, C, D. Vrijednosti pridjeljene pojedinom ključu predstavljaju stvarno značenje znamenke prikazane takvom kombinacijom segmenata.

Nastavak programskog koda skripte *digit_lookup.py* organiziran je u dvije funkcije kojima se ostvaruje donošenje odluke o vrijednosti znaka i provjera decimalne točke.

Funkcija `lookup` prikazana slijedećim programskim odsječkom implementira zadaću prepoznavanja vrijednsoti znaka.

```
20   def lookup(digitImg, digit_one_w = 10):
25
26       digH, digW = digitImg.shape[:2]
27       (dW, dH) = (int(digW * 0.25), int(digH * 0.15))
28
29       if (digit_one_w - 5 <= digW <= digit_one_w + 5):
30           return str(1)
31       else:
32           # initialising regions of interest
33           ROI = [
     # top
34   (((digW // 2) - (dW // 2), 0), ((digW // 2) + (dW // 2), dH)),
     # top-left
35   ((0, (digH // 4) - (dH // 2)), (dW, (digH // 4) + (dH // 2))),
     # top-right
36   ((digW - dW, (digH // 4) - (dH // 2)), (digW, (digH // 4) + (dH
     // 2))),
     # center
37   (((digW // 2) - (dW // 2), (digH // 2) - (dH // 2)) , ((digW //
     2) + (dW // 2), (digH // 2) + (dH // 2))),
     # bottom-left
38   ((0, digH - (digH // 4) - (dH // 2)), (dW, digH - (digH // 4) +
     (dH // 2))),
     # bottom - right
39   ((digW - dW, digH - (digH // 4) - (dH // 2)), (digW, digH - (digH
     // 4) + (dH // 2))),
     # bottom
40   (((digW // 2) - (dW // 2), digH - dH), ((digW // 2) + (dW // 2),
     digH))
41   ]
```

Ulazni argumenti prikazane funkcije su ndarray slike lokalizirane znamenke promatranog zapisa i podatak o najvećoj širini potencijalne znamenke broja 1, detektiran algoritmom skripte *image_processing.py*. Taj podatak je u ovom slučaju koristan, jer se usporedbom širine ulazne slike i registrirane širine detektiranih jedinica (ako ih je bilo), može zaključiti da je učitana slika znamenka broja 1, te na taj način zaobići izvođenje ostatka algoritma. U slučaju obrade svih ostalih znamenaka, slijedi korak izračuna koordinata područja interesa unutar kojih će se provjeravati sadržaj piksela učitane slike. Svakom od 7 tipičnih segmenata 7 – segmentnog formata, pridjeljuje se jedno područje interesa. Parovi koordinata gornjeg lijevog i doljnjeg desnog ugla kvadratića područja interesa izračunavaju se relativno dimenzijama učitane slike i pohranjuju u listu `ROI`, redoljedom koji odgovara redosljedu elemenata prethodno inicijaliziranih n –torki. Grafički prikaz promatranih područja interesa dan je slikom (Slika 6.1.2).

Slika 6.3 Definirana područja interesa (inverzne boje znamenke)

U sljedećem koraku, inicijalizira se lista `setup` koja predstavlja „masku“ za upis detektiranog stanja pojedinog segmenta. Lista sa područjima interesa se, potom, iterira, provjeravajući količinu bijelih piksela unutar promatranog područja interesa. Za uspješnije razumjevanje ovog koraka, potrebno je znati da je svaka slika predana kao ulazni argument ovoj funkciji, unutar algoritma skripte *image_processing.py* „izrezana“ iz binarizirane slike digitalnog prikaznika (Slika 5.7). To znači da je promatrana znamenka na slici ispisana pikselima bijele boje (255), a okolina pikselima crne boje (0). Količina bijelih piksela unutar promatranog područja interesa se računa funkcijom `countNonZero` biblioteke OpenCV. Ukoliko je barem 15% površine nekog područja interesa iz liste `ROI` prekriveno bijelim pikselima, vrijednost elementa sa istim indeksom iz liste `setup` se postavlja u 1, te tako popunjava maska.

```
42        setup = [0] * 7
43
44        # iterating through image of single digit
45        for i, ((Xt, Yt), (Xb, Yb)) in enumerate(ROI):
46
47            seg = digitImg[Yt:Yb, Xt:Xb]
48            whites = cv2.countNonZero(seg)
49            segArea = (Xb - Xt) * (Yb - Yt)
50
51            if whites / float(segArea) > 0.15:
52                setup[i] = 1
53
54            cv2.rectangle(imgDbg, (Xt,Yt), (Xb,Yb), (255, 255, 0), -1)
55
56        setup = tuple(setup)
57        digit = "X"
58        for key in LOOKUP_TABLE:
59
60            if setup == key: digit = str(LOOKUP_TABLE[key])
61
62        return digit
```

Nakon provjere boje svih 7 područja interesa, lista `setup` je popunjena, te se prevodi u n –torku. Jednostavnom iteracijom svim elementima referentnog dnevnika `LOOKUP_TABE` i usporedbom ključa svakog elementa sa sadržajem n –torke `setup`, donosi se odluka o vrijednosti znamenke prikazane ulaznom slikom.

Programska skripta *digit_lookup.py* sadrži i programski odsječak koji ostvaruje funkcionalnost detekcije decimalne točke. Navedeni programski odsječak realiziran je funkcijom `decimal_lookup`.

```
64   def decimal_lookup(digitImg):
65
66       digH, digW = digitImg.shape[:2]
67
68       seg = digitImg[digH - 2:digH, digW - 2:digW]
69       whites = cv2.countNonZero(seg)
70       if whites / float(4) > 0.1:
71           return "."
72       else:
73           return ""
```

Ulazni parametar ove funkcije je slika lokalizirane znamenke, korištena i pri prethodno opisanom procesu prepoznavanja vrijednosti. U slučaju kada je decimalna točka izvornog promatranog zapisa dovoljno blizu pojedinoj znamenki, opisni pravokutnik konture znamenke iz algoritma skripte *image_processing.py*, obuhvatiti će i dio te decimalne točke. Ovaj programski odsječak provjerava boju piksela doljnjeg desnog kuta učitane binarizirane slike pojedine znamenke, te u slučaju detekcije zadovoljavajuće količine piksela vrijednosti 255, pozivnom programu vraća detektiranu decimalnu točku.

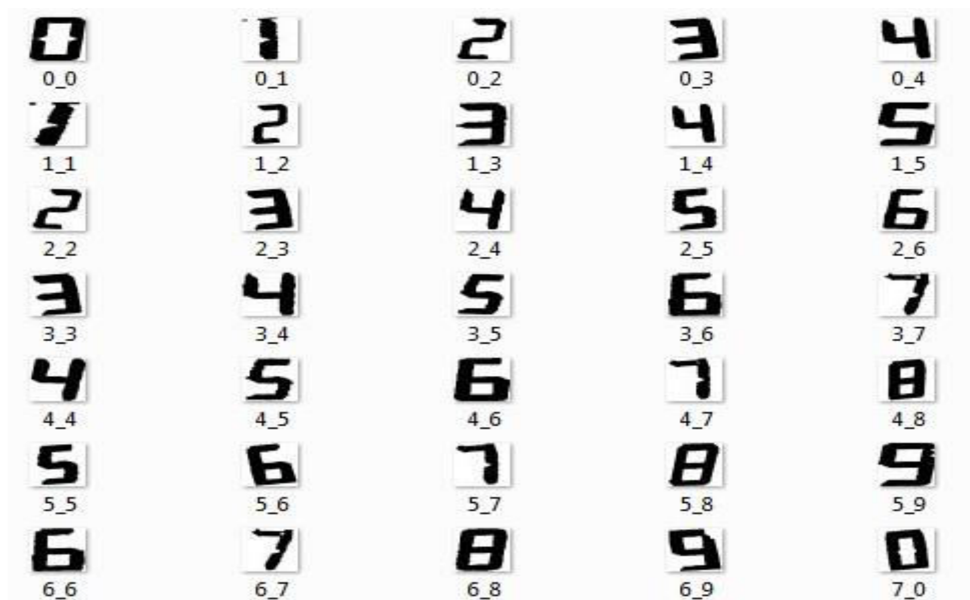
6.2. Prepoznavanje metodama strojnog učenja

Odluku o značenju alfanumeričkog znaka prikazanog zasebnom slikom moguće je ostvariti raznim metodama strojnog učenja. Uvjet uspješne implementacije nekog od algoritama tog tipa u razvijani programski sustav su uspješno proveden postupak segmentacije promatranog zapisa, te prilagodba boja na slici individualne znamenke u odgovarajući binarizirani format. U okviru ovog projekta u programski sustav implementirane su funkcionalnosti algoritama jednostavne neuronske mreže i algoritma K najbližeg susjeda.

6.2.1. Izrada baze podataka

Strojno učenje je područje računalne znanosti koje se bavi razvojem algoritama koji primjenom statističkih metoda nad velikim bazama podataka, ostvaruju prividnu mogućnost učenja algoritma. Algoritmi implementirani metodama strojnog učenja povećavaju performanse obavljanja očekivane fukcionalnosti višestrukim obradama velikih baza podataka, bez da se takve funkcionalnosti direktno programski „ugrade“ u sam algoritam.

Očekivana funkcionalnost navedenih algoritama strojnog učenja implementiranih u programski sustav ovog projekta je prepoznavanje značenja individualne dekadske znamenke sadržane na slici kao ulaznom parametru algoritma. U polju strojnog učenja, prepoznavanje značenja znaka na slici je problem klasifikacijske prirode. Kako bi se navedena funkcionalnost ostvarila, u okviru ovog projekta kreirana je baza podataka koja se sastoji od 11500 slika dekadskih znamenki (0 – 9), 7 – segmentnog formata. Primjer sadržaja kreirane baze podataka dan je uzorkom prikazanim slikom (Slika 6.2.1.1).



Slika 6.4 Uzorak baze podataka

Sadržaj baze podataka organiziran je u dvije podskupine: 10000 slika namjenjenih učenju algoritma, u direktoriju *znamenke_train* i 1500 slika namjenjenih provjeri točnosti algoritma, u direktoriju *znamenke_test*. Slike sadržane u tim direktorijima razvrstane su u 10 mapa, sukladno znamenci koja se na njima nalazi. Svaka od tako organiziranih mapa unutar navedenih direktorija imenovana je znamenkom prikazanom na svim slikama koje se unutar nje nalaze. Organizacija jednog od direktorija baze podataka prikazana je slikom (Slika 6.2.1.2).

Name	Size	Type	Modified
0	1000 items	Folder	Svi 28
1	1000 items	Folder	Svi 28
2	1000 items	Folder	Svi 28
3	1000 items	Folder	Svi 28
4	1000 items	Folder	Svi 28
5	1000 items	Folder	Svi 28
6	1000 items	Folder	Svi 28
7	1000 items	Folder	Svi 28
8	1000 items	Folder	Svi 29
9	1000 items	Folder	Svi 29

Slika 6.5 Organizacija direktorija znamenke_train

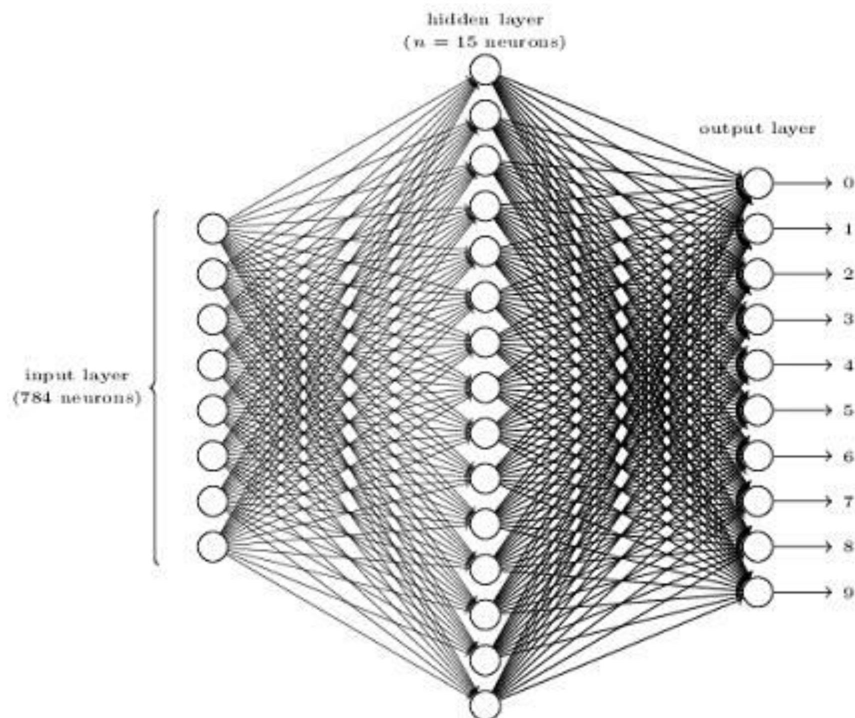
Opisana baza podataka kreirana je implementacijom jednostavnog algoritma programske skripte pod imenom *digit_mod.py*. Zadaća tog algoritma je modificirati sadržaj slike pojedinačne znamenke kao ulaznog parametra, mjenjajući položaj znamenke i njezine morfološke karakteristike. Programski kod unutar navedene skripte je organiziran kroz dvije funkcije `erodeImg` i `dilateImg` čija je zadaća u velikom broju iteracija modificirati ulazni sadržaj i rezultat svake pojedinačne modifikacije spremiti u zadani direktorij (*znamenke_train* ili *znamenke_test*). Navedene modifikacije realizirane su primjenom funkcija biblioteke OpenCV poput ranije opisanih: `morphologyEx`, `erode`, `dilate`, ali i funkcije biblioteke *utilities.py* `rotate_crop`. Funkcije `erodeImg` i `dilateImg` su strukturalno potpuno identične, a razlikuju se jedino u vrsti morfološke funkcije koju pozivaju. Nizom nasumično postavljenih programskih petlji, mijenaju se parametri navedenih morfoloških funkcija što rezultira velikim brojem rezultantnih slika za svaku iteraciju petlje. Cjeloviti programski kod opisane skripte nalazi se u privitku ovog dokumenta.

Programska skripta *digit_mod.py* nije dio programskog paketa gotovog sustava namjenjenog korisničkoj upotrebi. Jedina zadaća ove skripte je stvaranje baze podataka čiji sadržaj je potreban za inicijalizaciju parametara algoritama strojnog učenja. Prije pokretanja algoritma skripte *digit_mod.py* nužno je pozicionirati se u datoteku unutar koje se želi stvoriti sadržaj. Svaka od mapa datoteke *znamenke_train* prikazana na slici (Slika 6.2.1.2) inicijalno je sadržavala samo jednu sliku znamenke odgovarajuće vrijednosti. Algoritam opisane skripte, primivši lokaciju slike, stvara proizvoljan broj varijacija te slike u odabranom direktoriju.

6.2.2. Algoritam neuronske mreže

Programski sustavi zasnovani na algoritmima neuronskih mreža pronalaze široku primjenu u sustavima koji ostvaruju funkcionalnosti računalnog vida. Programski sustav razvijen u okviru ovog projekta integrira takav algoritam kao jednu od opcija pri realizaciji procesa donošenja odluke o vrijednosti pojedinačne znamenke sadržane unutar zasebne slike.

U okviru ovog programskog sustava implementirana je jednostavna, troslojna feedforward neuronska mreža, stohastičkog gradijentnog spusta, računatog algoritmom povratnog prostiranja pogreške. Na slici (Slika 6.2.2.1) grafički je prikazana shematska struktura takve neuronske mreže.



Slika 6.6 Shematski prikaz implementirane mreže [1]

Ulazni sloj mreže sadrži neurone koji enkodiraju vrijednosti piksela slike koja se predaje izravno mreži. Određivanjem konačne vrijednosti dimenzija ulazne slike, postiže se određeni stupanj standardizacije pri obradi podatka. Iz tog razloga ulazni sloj se postavnja na 784 neurona. Očekivani format piksela ulazne slike je

„greyscale“ float tipa, sa vrijednostima 0.0 koja predstavlja potpuno bijelu boju i 1.0 koja predstavlja potpuno crnu boju.

Srednji sloj, po terminologiji deep learninga i neuronskih mreža, nazvan i hidden layer se sastoji od ne specificiranog broja neurona, ovom prilikom označenih slovom n. Tu vrijednost moguće je proizvoljno mijenjati, što utječe na performanse mreže poput preciznosti ili njezine sposobnosti učenja.

Izlazni sloj mreže sastoji se od 10 neurona numeriranih indeksima od 0 do 9. Ovisno o aktivacijskoj vrijednosti pojedinog neurona, mreža „donosi odluku“ o znamenci koja se nalazi na slici, tj. ulaznom argumentu algoritma mreže.

Neuroni implementirani u sklopu ove neuronske mreže su tipični sigmoidalni neuroni, čija aktivacijska funkcija je dana izrazom (1).

$$\frac{1}{1 + exp\left(-\sum_j w_j x_j - b\right)} \tag{1}$$

U prethodnom izrazu parametri w i b predstavljaju parametre neurona težinu i pomak (eng. *Bias*), dok x predstavlja listu ulaznih vrijednosti za pojedini neuron.

Osnovno svojstvo implementirane neuronske mreže je algoritam gradijentnog spusta. Taj algoritam omogućuje mreži da učenejm konstantno ažurira vrijednosti svih težina i biasa na svakom neuronu, što predstavlja i osnovni smisao algoritama neuronskih mreža. Osnovna matematička funkcija na kojemu počiva algoritam gradijentnog spusta, nazvana funkcija troška, dana je izrazom (2).

$$C(w,b) \equiv \frac{1}{2n} \sum_x \left\| y(x) - a \right\|^2 \tag{2}$$

Metodologija algoritma zasniva se na velikom broju iteracija potrage za vrijednostima parametara w i b za koje će vrijednost vektora a (izlaz iz mreže kada je x ulaz), biti što bliži vrijednosti vektora y (idealni izlaz, tj. vektor koji predstavlja

pojedinu znamenku). To znači da se ponavljajućim postupkom traži minimum funkcije (2). Pronalaženje tog minimuma je znak mreži da su vrijednosti parametara svih neurona najpoželjnije kalibrirane za prepoznavanje vrijednosti znamenke na ulaznoj slici. Kako se pri svakoj iteraciji ne bi računao gradijent navedene funkcije za sve ulazne parametre, nasumično se odabire manji broj training ulaza (eng. *Mini Batch*) za koje je lako izračunati pojedinačni gradijent. Računanjem prosjeka svih pojedinačnih gradijenata lako se aproksimira vrijednost stvarnog gradijenta. Ovakav algoritam gradijentnog spusta uvelike ubrzava proces treniranja mreže, a zove se stohastički gradijentni spust.

Opisana neuronska mreža programski je izvedena programskim kodom obuhvaćenim skriptom *network.py*. Programski kod opisane neuronske mreže, uz određene modifikacije, u razvijeni sustav je implementiran vodeći se primjerom neuronske mreže iz prvog poglavlja knjige *Neural Networks and Deep Learning* [1], autora Michaela Nielsena.

Središnji objekt programskog koda skripte *network.py* je razred `Network`.

```
7     class Network(object):
8
9         def __init__(self, sizes):
10
11            self.num_layers = len(sizes)
12            self.sizes = sizes
13            self.biases = [np.random.randn(y, 1) for y
                           in sizes[1:]]
14            self.weights = [np.random.randn(y, x) for x, y
                            in zip(sizes[:-1], sizes[1:])]
```

Lista `sizes` sadrži tri elementa od kojih svaki predstavlja broj neurona u odgovarajućem sloju mreže. Inicijalizacijska metoda objekta `Network` postavlja vrijednosti biasa i težina 2. i 3. sloja (prvi je ulazni) dodjeljujući im nasumične vrijednosti funkcijom `random.randn` biblioteke NumPy. Tako inicijalizirane vrijednosti predstavljaju početno stanje izvođenja algoritma stohastičkog gradijentnog spusta.

Sigmoidalna aktivacijska funkcija (1) ostvarena je public funkcijom `sigmoid`, programskim odsječkom unutar skripte *network.py*, prikazanim u nastavku.

```
101 def sigmoid(z):
102     return 1.0/(1.0+np.exp(-z))
```

Varijabla z u priloženom odsječku predstavlja ndarray red nastao kao skalarni produkt ndarray redova težina i ulaznih vriednosti kojemu se pribraja vektor odgovarajući vektor biasa.

Navedena sigmoidalna funkcija se u objektu mreže implementira kroz metodu razreda nazvanu feedforward. Ova metoda kao ulazni argument prima ndarray red od n elemenata posloženih u jednu dimenziju, a kao izlaz daje rezultat mreže proračunat aktualnom verzijom parametara težina i biasa.

```
16  def feedforward(self, a):
17
18      for b, w in zip(self.biases, self.weights):
19          a = sigmoid(np.dot(w, a)+b)
20      return a
```

Funkcionalnost učenja algoritam opisane mreže ostvaruje nizom metoda razreda Network od kojih centralnu ulogu ima metoda SGD. Ova metoda implementira algoritam stohastičkog gradijentnog spusta. Ulazni parametri su prethodno incijalizirani ndarray baze podataka za učenje nazvan training_data, proizvoljan broj iteracija treninga zadan varijablom epochs, veličina mini batcha zadana varijablom mini_batch_size, stopa učenja eta i opcionalno zadani ndarray red baze podataka za provjeru preciznosti, test_data. Unutar svake iteracije učenja, algoritam nasumično, iz ukupne training baze podataka, odabire sadržaj mini bacha. Potom nad svakim mini batchom primjenjuje jedan korak gradijentnog spusta pozivom metode razreda update_mini_batch.

Navedena metoda ažurira parametre težine i biasa sukladno jednom izvršavanju algoritma gradijentnog spusta. Najbrži način izračuna gradijenta funkcije troška (2), nad sadržajem mini batcha implementira se pozivom metode klase backprop koja implementira algoritam povratnog prostiranja pogreške.

Potpuni programski kod skripte *network.py* nalazi se u privitku ovog dokumenta.

Prethodno inicijalizaciji opisane neuronske mreže i pokretanja procesa učenja, potrebo je prilagoditi bazu podataka, opisanu u prethodnom potpoglavlju. Naime, kako je u ovom poglavlju navedeno, opisana neuronska mreža u ulaznom sloju ima ukupno 784 neurona. Svaki od tih neurona predstavlja jedan piksel, grayscale formata, vrijednosti skalirane u rasponu od 0.0 do 1.0. To znači da pojedina slika koja se dovodi na ulaz algoritma neuronske mreže mora imati 784 piksela posloženih u jednodimenzionalni ndarray od 784 elementa. Baza podataka opisana u prethodnom potpoglavlju sastoji se od slika neodređenih dimenzija. Algoritmom programske skripte *data_loader.py* ostvaruje se prilagodba sukladna potrebama algoritma mreže.

Programski sadržaj skripte *data_loader.py* organiziran je u dvije funkije *main* i *load_data*.

Algoritam funkcije *main* pokreće se direktnim pokretanjem skripte iz korisničkog sučelja. Jedina zadaća ovog algoritma je učitavanje sadržaja direktorija *znamenke_train* i *znamenke_test*, te prilagodba svake slike traženom formatu i dimenzijama. Obrađene slike se pohranjuju u dva nova direktorija *prepared_train* i *prepared_test*. Važno je napomenuti da, za razliku od organizacije prikazane slikom (Slika 6.2.1.2), prilagođene slike unutar novokreiranih direktorija nisu razvrstane po mapama. Klasifikacijska obilježja u ovom slučaju su upisana u ime svake pojedine slike.

Uz ispravno pripremljenu bazu podataka moguće je pokrenuti proces inicijalizacije i treniranja neuronske mreže. To se ostvaruje pokretanjem kratke skripte *init_network.py* iz korisničkog sučelja. Potpuni programski sadržaj navedene skripte prikazan je u nastavku:

```
1    #!/usr/bin/python
2
3    import data_loader
4    training_data, test_data = data_loader.load_data()
5
6    import network
7    net = network.Network([784, 40, 10])
8    net.SGD(training_data, 150, 15, 0.0125, test_data=test_data)
9    net.save('network_params.txt')
```

Prikazana skripta poziva ranije spomenutu funkciju `load_data` implementiranu u skripti *data_loader.py*. Zadaća funkcije `load_data` je učitavanje baze podataka raspoređene u direktorije *prepared_train* i *prepared_test* u format prikladan daljnoj obradi metodom mreže `SGD`. Kako je ranije navedeno, metoda `SGD` kao ulazne parametre prima i dvije liste: `training_data` i `test_data`. Lista `trainig_data` sastoji se od 10000 2 – torki `(training_inputs, training_results)` koje predstavljaju pojedinu sliku i vrijednost broja koji je njome prikazan. Red `taining_inputs` jednodimenzionalni ndarray od 784 elementa koji predstavljaju pojedini piksel, a `training_results` jednodimenzionalni ndarray od 10 elemenata. Svaki `training_results` ndarray red u ovom zapisu predstavlja vektoriziranu dekadsku vrijednost klasifikatora znamenke na slici. Logiku zapisa reda `training_results` najbolje ilustrira programski odsječak iz skripte *data_loader.py* korišten pri njegovom oblikovanju:

```
48   def vectorized_result(j):
49
50       e = np.zeros((10, 1))
51       e[j] = 1.0
52       return e
```

Lista `test_data` se sastoji od 1500 2 – torki sličnog formata, pri čemu je jedina razlika u tome što drugi član svake 2 – torke, koji predstavlja klasifikator slike, ovdje nije vektoriziran, već je zapisan kao integer vrijednost od 0 do 9.

Potpuni programski kod skripte *data_loader.py* nalazi se u privitku ovog dokumenta.

Nakon ispravnog učitavanja baze podataka training i test slika, skripta *init_network.py* inicijalizira neuronsku mrežu razreda `Network` predajući mu listu `sizes` sa veličinama slojeva mreže. Veličina srednjeg sloja (eng. *Hidden layer*) je proizvoljna a odabire se ovisno o performansama mreže. U narednim linijama inicijaliziraju se parametri metode `SGD` i poziva metoda `save` čija je zadaća pohraniti json string sa krajnjim parametrima svih neurona nakon završetka sesije učenja. Rezultantni parametri se pohranjuju u tekstualnu datoteku *network_params.txt*.

Provedbom prethodno opisanih procesa inicijalizacije i treniranja neuronske mreže, programski sustav raspolaže s optimalnim parametrima kojima se u svakom trenu može lako pristupiti. Učitavanjem datoteke *network_params.txt* i „raspakiravanjem“ njezinog sadržaja u objekte razreda `Network`, ostvaruje se mogućnost implementacije algoritma donošenja odluke o vrijednosti znamenke na slici u svakoj programskoj skripti sustava.

Programski kod koji obuhvaća proces postavki parametara neuronske mreže unutar skripte *image_processing.py* prikazan je sljedećim odsječkom:

```
16   # initializing network parameters
17   import network
18   np.seterr(over='ignore')
19
20   f = open('network_params.txt', "r")
21   data = json.load(f)
22   f.close()
23
24   net = network.Network(data["sizes"])
25   net.weights = [np.array(w) for w in data["weights"]]
26   net.biases = [np.array(b) for b in data["biases"]]
```

Funkcionalnost prepoznavanja znamenke neuronskom mrežom unutar programskog sadržaja skripte *image_processing.py*, ostvarena je funkcijom `run_network`, prikazanom sljedećim odsječkom:

```
94   def run_network(singleDigit):
95
96       singleDigit = cv2.bitwise_not(singleDigit)
97       img_input = cv2.resize(singleDigit, (28, 28), interpolation =
                                                        cv2.INTER_AREA)
98       img_input = np.reshape(img_input, (784, 1))
99       result = np.argmax(net.feedforward(img_input))
100
101      return str(result)
```

Ulazni parametar ove funkcije je lokalizirano područje znamenke sa slike (Slika 5.7). Zbog činjenice da se baza podataka sastoji od slika crnih znamenaka na bijeloj podlozi, ulaznoj slici potrebno je invertirati boje. Sadržaj ulaznog parametra se potom, prevodi u odgovarajući format od 784 piksela i prosljeđuje metodi `feedforward`. Rezultat prikazane funkcije je vrijednost znamenke na slici.

6.2.3. Algoritam K najbližeg susjeda

Posljednja od tri metode donošenja odluke o vrijednosti lokalizirane znamenke prikazane slikom implementirana u okviru ovog programskog sustava je algoritam K najbližeg susjeda (skraćeno Knn).

Knn algoritam je jedna od metoda strojnog učenja koja pripada procesima prepoznavanja uzoraka. Knn algoritam pripada nadgledanoj (eng. *Supervised*) porodici algoritama strojnog učenja. To znači da za skup podataka x označen klasama y želimo pronaći odnos x prema y. Formalno, knn algoritam pronalazi funkciju $h: X \rightarrow Y$, tako da za primjer prethodno neviđenog ulaza x, $h(x)$ uspješno pronalazi klasifikator y [4]. Knn algoritam je neparametarska, „lijena“ metoda učenja s obzirom da ne pravi eksplicitne pretpostavke forme funkcije h i odgađa izgradnju generalne ciljne funkcije za klasificiranje sve do trenutka početka procesa klasifikacije novog zadanog primjera [4].

Knn algoritmi mogu se koristiti u funkciji klasificiranja ili regresije. Algoritam implementiran programskom sustavu razvijenom u okviru ovog projekta koristi se za klasifikaciju slike koja prikazuje znamenku. Izlaz algoritma je klasna pripadnost zadanog objekta (slika). Uz dostupnu bazu podataka klasificiranih objekata, algoritam se svodi na izračun ekulidske (može i ostalih) udaljenosti između zadanog objekta i svih objekata sadržanih u bazi podataka. Zadani objekt se klasificira „većinom glasova“ njegovih susjeda, tako da se objekt dodjeljuje klasi koja je najzastupljenija među k njegovih najbližih susjeda u funkciji prostora. Kako bi se izbjegle situacije „izjednačenog glasovanja“ za vrijednost k se uzima neki neparan broj.

Važno je primjetiti kako minimalna faza treniranja Knn algoritma istovremeno predstavlja opterećenje memorije, s obzirom da se pohranjuje potencijalno velika baza podataka, kao i opterećenje procesorskih kapaciteta u fazi klasificiranja, s obzirom da određivanje vrijednosti zadanog objekta zahtjeva prolazak algoritma kroz cijelu bazu podataka.

Programska implementacija algoritma K najbližeg susjeda podržana je bibliotekom OpenCV. U okviru razvijenog programskog sustava funkcionalnost prepoznavanja vrijednosti znamenke na slici implementirana je razredom `Knn_processing`, definiranim unutar programske skripte *knn_process.py*. Navedeni razred sastoji se od dvije osnovne metode `train_knn` i `predict_digit`.

Referentna baza podataka korištena pri inicijalizaciji Knn algoritma u okviru ovog projekta je skup 1500 slika direktorija *znamenke_test*. Ova baza podataka se potrebama Knn algoritma prilagođava kratkim algoritmom realiziranim unutar skripte *knn_loader.py*. Kako je ranije opisano (Slika 6.2.1.2), sve slike navedene baze podataka razvrstane su u zasebne mape imenovane njihovim klasifikatiorima. Algoritam skripte *knn_loader.py* prolazi kroz tako organiziran direktorij *znamenke_test* stvarajući dvije ndarray matrice `npa_flattened_images` i `npaClassifications`. Prva matrica se sastoji od 1500 elemenata od kojih je svaki ndarray vektor 784 elementa, floating – point jednostruke preciznosti. Svaki element svakog vektora ndarray matrice `npa_flattened_images` predstavlja pojedini piksel slike iz navedene baze podataka, u grayscale formatu. Matrica `npaClassifications` je ndarray vektor 1500 elemenata, gdje je svaki element klasifikator odgovarajuće slike unutar druge matrice. Sadržaji opisanih matrica kreiranih skriptom *knn_loader.py* pohranjuju se u tekstualne datoteke *KNNflattened_images.txt* i *KNNclassifications.txt*.

Prilikom inicijalizacije razreda `Knn_processing`, poziva se metoda `train_knn`. Navedena metoda učitava prethodno opisane matrice iz pripadajućih tekstualnih datoteka, inicijalizira Knn funkciju prostora i poziva metodu `train` biblioteke OpenCV.

```
k_nearest = cv2.KNearest()
k_nearest.train(npa_flattened_images, npa_classifications)
```

Treniranje baze podataka se u kontekstu metode `train` odnosi na povezivanje pojedine slike zapisane unutar matrice `npa_flattened_images` i njoj odgovarajuće klase iz matrice `npa_classifications`.

Metoda predict_digit, razreda Knn_processing implementira funkcionalnost prepoznavanja znamenke unutar slike. Ulazni argument te metode je uobičajni ndarray zapis promatrane slike. Sadržaj ulaznog argumenta se prilagođava traženom zapisu od 784 elementa u jednom redu, floating – point aritmetike, jednostruke preciznosti. Takav zapis ndarray matrice promatrane slike prosljeđuje se metodi find_nearest biblioteke OpenCV. Navedena metoda nad zapisom promatrane slike provodi ranije opisan Knn algoritam, te kao rezultat vraća vektor rezultata, vektor k najbližih susjeda i vektor euklidskih udaljenosti ulaznog uzorka od k najbližih susjeda.

Cjelokupan programski kod skripti *knn_loader.py* i *knn_process.py* nalazi se u privitku ovog dokumenta.

Programski kod koji obuhvaća postavke parametara Knn algoritma unutar skripte *image_processing.py* prikazan je sljedećim odsječkom:

```
12   # initializing knn parameters
13   import knn_process
14   knn = knn_process.Knn_processing("_1_0")
```

Funkcionalnost prepoznavanja znamenke pomoću Knn algoritma, unutar skripte *image_processing.py* ostvarena je funkcijom:

```
103  def run_knn(singleDigit):
104
105      singleDigit = cv2.bitwise_not(singleDigit)
106      result = knn.predict_digit(singleDigit)
107
108      return str(result)
```

Logika inverzije boje primljene slike jednaka je prethodno opisanoj na primjeru funkcije run_network pri pozivu neuronske mreže.

7. Rezultati

Programski sustav optičkog očitanja 7 – segmentnog sadržaja čija je struktura opisana u prethodnim poglaviljima realizira funkcionalnost unosa promatranih podataka u vidu niza numeričkih znakova, u memorijski prostor računala jednostavnim snimanjem fotografije ciljanog sadržaja.

Uspješnost interpretacije numeričkog zapisa promatrane slike prvenstveno ovisi o uspješnoj provedbi procesa nekoliko „kritičnih“ stadija cjelokupnog algoritma. Uspješna detekcija rubova prikaznika i ekstrakcija sadržaja zaslona iz ostatka slike preduvjet je za uspješan proces binarizacije prikaza zaslona. Bez kvalitetno provedenog postupa binarizacije, nije moguće segmentirati promatrani zapis i izdvojiti pojedine znakove.

Obradom sadržaja slika lokaliziranih znakova nekom od tri opisane metode donošenja odluke, ostvaruje se završni, korisniku vidljiv korak prikaza promatrne vrijednosti. Funkcionalnost cjelokupnog sustava uvelike ovisi i o preciznosti algoritama donošenja odluke.

Algoritam prepoznavanja vrijednosti znamenke pomoću predloška, jedini od tri navedena ne pripada metodama strojnog učenja, već generičkim predloškom ispituje područja interesa. Najveći utjecaj na preciznost ovog algoritma ima proces binarizacije zapisa, te obrada morfološkim metodama. Netočna očitanja događaju se u situacijama kada dođe do neželjenih pojava bijelih piksela unutar ispitivanih područja interesa, najčešće kao razlog pogrešno kalibriranih morfoloških metoda. Ovo je generalno najprecizniji algoritam prepoznavanja znamenke, u slučaju uspješne izvedbe svih ključnih dijelova cjelokupnog procesa.

Preciznost algoritma neuronske mreže ovisi prvenstveno o sadržaju i opsegu prethodno opisane korištene baze podataka, ali i hiperparametrima mreže odabranima pri njegovoj inicijalizaciji. Pri učenju algoritma u sklopu ovog projekta optimalni hiperparametri iznosili su: 150 iteracija učenja (epoha), 15 slika po mini batchu i stopa učenja u iznosu 0.0125. Uz ovako postavljene parametre i kreiranu bazu podataka, postignuta je preciznost od ~69% (72% u vrhuncu). Poslijednjih 25 epoha učenja prikazano je slikom (Slika 7.1).

```
Epoch 125: 1080 / 1500
Epoch 126: 988 / 1500
Epoch 127: 1018 / 1500
Epoch 128: 1059 / 1500
Epoch 129: 1063 / 1500
Epoch 130: 1042 / 1500
Epoch 131: 1037 / 1500
Epoch 132: 1026 / 1500
Epoch 133: 991 / 1500
Epoch 134: 1018 / 1500
Epoch 135: 1066 / 1500
Epoch 136: 1013 / 1500
Epoch 137: 1015 / 1500
Epoch 138: 1012 / 1500
Epoch 139: 1008 / 1500
Epoch 140: 978 / 1500
Epoch 141: 996 / 1500
Epoch 142: 1000 / 1500
Epoch 143: 1022 / 1500
Epoch 144: 1044 / 1500
Epoch 145: 1044 / 1500
Epoch 146: 1020 / 1500
Epoch 147: 1033 / 1500
Epoch 148: 1036 / 1500
Epoch 149: 1048 / 1500
```

Slika 7.1 Primjer učenja kreiranom bazom podataka

Algoritam K najbližeg susjeda implementiran u razvijeni programski sustav, treniran je skupom od 1500 slika iz direktorija *znamenke_train* i testiran skupom jednakog broja slika iz direktorija *znamenke_test*. Izračunata preciznost opisanog Knn algoritma iznosi 83.93%.

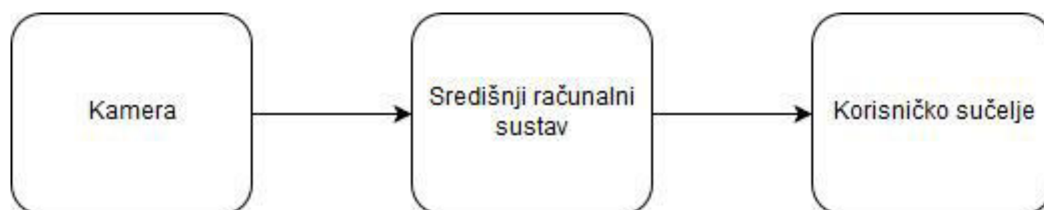
Razvijeni programski sustav testiran je na većem broju slika digitalnih prikaznika u različitim uvjetima i okolini. Rezultati nekoliko testiranih primjera prikazani su slikama u privitku ovog dokumenta.

Funkcionalnost razvijenog programskog sustava uspoređena je sa sličnim online dostupnim proizvodima. Uz najčešću razliku u načinu registriranja lokacije digitalnog prikaznika unutar šire slike, algoritam ostvaruje zadovoljavajuće rezultate u području binarizacije zaslona i lokalizacije znakova. Česti pogreške nastale uslijed neravnomjernog osvjetljenja ili odbljeska okoline moguće je otkloniti složenijim algoritmima binarizacije. Problem lokalizacije rubova prikaznika moguće je pojednostaviti zadavanjem koordinata očekivane lokacije prikaznika jednostavnom intervencijom u programski kod algoritma.

8. Računalna implementacija sustava

Razvijeni programski sustav optičkog očitanja zapisa digitalnih prikaznika namjenjen je implementaciji unutar računalnih sustava podržanih operacijskim sustavom Unix porodice. Uz manje preinake razvijenu aplikaciju moguće je implementirati i na drugim tipovima operacijskih sustava.

Računalni sustavi sa odgovarajućom programskom podrškom i pratećim perifernim jedinicama, pronalaze primjenu u širokom rasponu zadataka vezanim za područje mjeriteljstva i/ili računalnog vida. Vrsta računalnog sustava na koji je ovu aplikaciju moguće implementirati nije specifično određena sve dok računalo raspolaže zadovoljavajućim memorijskim i procesorskim kapacitetima, te potrebnim perifernim jedinicama. Shematski prikaz minimalnog sadržaja cjelovitog sustava na kojemu je moguće koristiti razvijenu aplikaciju prikazan je slikom (Slika 8.1).



Slika 8.1 Blok dijagram cjelovitog sustava

U slučaju kada se promatrana slika sa sadržajem kojemu je potrebno očitati vrijednost, ne nalazi inicijalno u memoriji računala, unos u računalni sustav se ostvaruje perifernom jedinicom. Vrijednost dobivenu obradom podataka razvijenom aplikacijom potrebno je prikazati korisniku putem neke vrste korisničkog sučelja i podatke pohraniti u bazu podataka. Korisničko sučelje može se realizirati kao monitor, web aplikacija, mobilna aplikacija, baza podataka, itd. Korisničko sučelje realizira se ovisno o potrebi krajnjeg korisnika.

Praktičnu primjenu razvijena aplikacija ostvaruje u slučaju implementacije ugradbenim računalnim sustavom. S obzirom na ograničene memorijske kapacitete i procesorsku moć većine takvih računala, za razvoj funkcionalnog sustava sa složenim zadaćama, potrebno je voditi računa o efikasnosti programske podrške implementirane na takav sustav. Implementacija složenijih algoritamskih rješenja na računala koja upravljaju nekim ugradbenim sustavaom, doprinosi pouzdanosti i povećanju robusnosti samog sustava, što je izrazito važno zbog otežanog pristupa većini ugradbenih računala.

Primjer ugradbenog računala karakteristika odgovarajućih potrebama razvijene aplikacije je sustav Raspberry Pi 3. Raspberry Pi 3 je mikrokontroler implementiran na jednoj tiskanoj pločici, system on chip arhitekture, sa integriranom ARM kompatibilnom procesorskom jedinicom (CPU), on – chip grafičkom procesorskom jedinicom (GPU) i pratećim upravljačkim i perifernim sklopovljem. Programska memorija, na kojoj je pohranjen i operacijski sustav, izvedena je microSD karticom. Složeno periferno sklopovlje, preko priključaka mikrokontrolera omogućuje uključivanje ovog ugradbenog računala u složenije računalne sustave i njegovu komunikaciju sa drugim elektroničkim uređajima. U okviru ovog projekta, veza između korištene web kamere i mikrokontrolera Raspberry Pi je ostvarena USB standardom.

Značajniji operacijski sustavi koje ovo ugradbeno računalo podržava su Ubuntu MATE, Kali Linux, Windows 10, RISC OS, Debian FreeBSD, NetBSD. Raspbian je najčešće korišten operacijski sustav kada je riječ o ugradbenim računalima branda Raspberry Pi. To je Debian bazirana, Linux distribucija prilagođena ugradbenim računalima.

Za potrebe pristupa razvijenom programskom sustavu od strane krajnjeg korisnika, implementirana je jednostavna skripta naredbi Bash ljuske operacijskog sustava nazvana *run.sh*.

```
#!/bin/bash

rm *.jpg
name="input_image.jpg"

while true; do

    fswebcam -d v4l2:/dev/video1 -S 15 --no-banner $name
    python image_processing.py $name
    echo -e "\n"
    sleep 5
    #read -p "Press enter to continue"
done
```

Navedena skripta objedinjuje sve komponente programske strukture razvijenog sustava i predstavlja centralni izvršivi modul kojemu krajnji korisnik direktno pristupa. U prikazanom primjeru algoritam ove skripte je podešen slučaju kada se promatrana fotografija unosi kamerom kao perifernom jedinicom. Skripta je jednako primjenjiva i kada se predmet interesa učitava iz memorije računala. Algoritam prikazane skripte omogućuje korištenje razvijenog sustava u mjeriteljskim ili nadzornim zadatcima periodičkim praćenjem ulaznog sadržaja.

Kao ogledni primjer tijekom implementacije i razvoja opisanog programskog sustava korišteno je nekoliko javno dostupnih aplikacija, slične funkcionalnosti. Primjer takve aplikacije je i jednostavni čitač jednolinijskog zapisa 7 – segmentnih znamenki, primjenjiv na većem broju platformi [7]. Većina javno dostupnih sustava automatskog prepoznavanja vrijednosti digitalnog zapisa varira ovisno o njima pretpostavljenoj primjeni. Navedena aplikacija [7] ostvaruje funkcionalnost prepoznavanja jednog reda znamenki iz prethodno odabranog područja ulazne slike. Aplikacija ovog tipa moguću primjenu pronalazi u zadaćama praćenja vrijednosti statičkih prikaznika. U usporedbi sa navedenim primjerom [7], aplikacijski sustav razvijen u okviru ovog projekta obuhvaća šire područje primjene zahvaljujući mogućnosti detekcije samog prikaznika unutar slike.

8.1. Utrošak procesorskog vremena

Opisani sustav automatskog prepoznavanja sadržaja digitalnih prikaznika, kako je prethodno navedeno, implementiran je na osobnom računalu s procesorom Intel Core i7 – 2670QM i 8 GB RAM memorije, te na komercijalnom ugradbenom računalu Raspberry Pi 3 s četverojezgrenim ARM Cortex - A53 procesorom i 1GB RAM memorije. Sa ciljem procjene efikasnosti i primjenjivosti ovakvog sustava u realnim uvjetima, provedena je analiza dijelova sustava s obzirom na vremenski utrošak procesorskog vremena. Prosječno vrijeme izvršavanja pojedinih procesa, izračunato na temelju nekoliko slijednih očitanja, prikazano je tablicom (Tabela 1).

Tabela 1 Vremenski utrošak procesora

Operacija	Intel Core i7 – 2670QM	ARM Cortex – A53
Lokalizacija prikaznika	246.946 ms	1539.99 ms
Binarizacija slike	1.519 ms	18.579 ms
Segmentacija zapisa	1.160 ms	16.338 ms
Prepoznavanje predloškom	0.068 ms	0.679 ms
Prepoznavanje mrežom	0.141 ms	1.457 ms
Prepoznavanje Knn	1.208 ms	10.774 ms
Ukupno	251.042 ms	1587.817 ms

Iz priloženog se uočava kako je operacija lokalizacije prikaznika procesorski najzahtjevnija. Razlog tome je i dio algoritma koji se odnosi na inicijalni unos ulaznog parametra obuhvaćen ovim procesom, ali i primjena nekoliko ranije opisanih, složenijih funkcija biblioteke OpenCV. Trajanje izvođenja istog programskog sustava implementiranog računalnim sustavom Raspberry Pi 3 osjetno je duže, a razlog tome su ograničeni kapaciteti takvog sustava. Uz pravilno razmotreno područje primjene razvijenog sustava, implementacija na ugradbenom računalu, u konačnici ne predstavlja potpuno ograničenje pri upotrebi razvijenih funkcionalnosti.

8.2. Upute za korištenje

Na korištenom računalu potrebno je koristiti operacijski sustav neke od Linux distribucija.

Instalacija programske podrške podrazumjeva kopiranje svih skripti sa programskim kodom u jedan direktorij, sukladno rasporedu (Slika 3.2). Lokacija direktorija u sustavu nije specificirana, jedino je bitno pobrinuti se da je pisanje u taj direktorij dozvoljeno. Potrebno je pozicionirati se u kreirani direktorij.

Radnje kreiranja i učitavanja baze podataka, te inicijalizaciju algoritama strojnog učenja krajnji korisnik ne poduzima.

Na korištenom operacijskom sustavu potreban je interpreter Python programskog jezika. Taj interpreter inicijalno dolazi u paketu sa većinom operacijskih sustava UNIX tipa. Ukoliko je instalacija ipak potrebna to je moguće ostvariti unošenjem slijedećih neredbi u terminal:

```
wget https://www.python.org/ftp/python/2.7.12/Python2.7.12.tgz
tar xzf Python-2.7.12.tgz
cd Python-2.7.12
sudo ./configure
sudo make altinstall
```

Provjeru uspješnosti instalacije moguće je provesti unosom slijedeće naredbe:

```
python --version
```

Potrebno je instalirati biblioteku OpenCV. To se ostvaruje unosom slijedeće naredbe:

```
sudo apt-get install python-opencv
```

Potrebno je instalirati numpy biblioteku sa matematičkim funkcijama i standardima za prilagodbu podataka algoritmima mreže. To se ostvaruje unosom slijedeće naredbe:

```
sudo apt-get install python-pip
```

U slučaju potrebe za web kamerom, web kamera se aktivira direktno iz terminala, bash naredbama u programskom kodu skripte *run.sh*. Potrebno je instalirati odgovarajuću funkcionalnost pozivom slijedeće funkcije:

sudo apt-get install fswebcam

Uz sve datoteke i pomoćne module ispravno učitane, krajnji korisnik aplikaciju pokreće pokretanjem skripte *run.sh*.

9. Zaključak

Programski sustav namjenjen optičkom očitanju sadržaja digitalnih numeričkih prikaznika, razvijen u okviru ovog projekta, objedinjuje sve funkcionalnosti konvencionalnih aplikacija ovog tipa. Složeni algoritam nad ulaznom slikom provodi procese lokalizacije digitalnog prikaznika, predobrade i binarizacije sadržaja prikazanog tim prikaznikom, lokalizaciju pojedinih znakova, te na poslijetku donosi odluku o značenju svakog pojedinog znaka. Funkcionalnost donošenja odluke o vrijednosti pojedinog znaka implementirana je trima različitim algoritmima, s ciljem eksperimentalnog utvrđivanja optimalnog rješenja primjenjivog u realnom okruženju. Mogućnost uspješne interpretacije zapisa učitane slike prvenstveno ovisi o provedbi prvih koraka algoritma koji podrazumjevaju procese predobrade slike i lokalizaciju znakova. Unutar programske strukture opisanog algoritma, implementirano je nekoliko metoda sa isključivom svrhom pružanja dodatne razine otpornosti sustava na učestale i očekivane komplikacije proizašle iz negativnog utjecaja okoline.

Cjelokupnu programsku podršku razvijenu u okviru ovog rada moguće je implementirati na većini suvremenih stolnih i prijenosnih računala. Praktičnu primjenu razvijena aplikacija ostvaruje u slučaju implementacije ugradbenim računalnim sustavom. Specijalizirani sustavi namjenjeni optičkom očitanju vrijednosti digitalnih prikaznika implementirani su ugradbenim računalom sa pratećim perifernim sučeljima za unos, te pohranu ili prikaz podataka i pogonjeni opisanim aplikacijskim sadržajem. Takvi sustavi ostvaruju široku primjenu u područjima mjeriteljstva, nadzora i automatizacije upravljanja složenijim uređajima, bez potrebe za direktnom intervencijom u sklopovlje promatranog uređaja.

Programski sustavi namjenjeni automatskom očitavanju sadržaja digitalnih prikaznika predstavljaju jedno od osnovnih područja istraživanja na temelju kojih je razvijana znanstvena disciplina računalnog vida.

Literatura

[1] Michael Nielsen, „*Neural Networks and Deep Learning*“, prosinac 2017.
http://neuralnetworksanddeeplearning.com/ [p. 20. prosinca 2017.]

[2] Adrian Rosebrock, „*Pyimagesearch Blog*“, travanj 2015.
https://www.pyimagesearch.com/ [p. 22. travnja 2018.]

[3] Wikipedia, „*Seven-segment display*“
https://en.wikipedia.org/wiki/Seven-segment_display [p. 10. lipnja 2018]

[4] Kevin Zakka, „*Kevin Zakkas Blog*“, srpanj 2016.
https://kevinzakka.github.io/2016/07/13/k-nearest-neighbor/
[p. 12. svibnja 2018]

[5] David Ascher, Paul F. Dubois, Konrad Hinsen, Jim Hugunin, Travis Oliphant, „*Numerical Python*“, 1999.
https://www.cs.mcgill.ca/~hv/articles/Numerical/numpy.pdf [p. 22. travnja 2018]

[6] https://opencv.org/
https://docs.opencv.org/2.4/ [p. 22. travnja 2018]

[7] Erik Auerswald
https://www.unix-ag.uni-kl.de/%7Eauerswal/ssocr/ [p. 4. lipnja 2018]

[8] Slika za potrebe testiranja
http://www.hannainstruments.co.uk/ [p. 10. lipnja 2018]

[9] Slika za potrebe testiranja
https://www.circuitspecialists.com/digital-panel-meters [p. 10. lipnja 2018]

[10] Slika za potrebe testiranja
https://es.hudsonreed.com/info/blog/ [p. 10. lipnja 2018]

[11] Slika za potrebe testiranja
http://www.directindustry.com/ [p. 10. lipnja 2018]

[12] Slika za potrebe testiranja
https://www.amazon.com/HiLetgo-MAX7219-8-Digital-Segment-Digital/
[p. 10. lipnja 2018]

Sažetak

Sustav za automatsko optičko očitanje digitalnih prikaznika

U okviru ovog rada razvijen je programski sustav namjenjen automatskom optičkom očitavanju sadržaja digitalnih prikaznika. Razvijeni sustav objedinjuje algoritam lokalizacije prikaznika unutar učitane slike, predobradu sadržaja prikaznika i segmentaciju promatranog zapisa lokalizacijom znakova. Značenje pojedinog lokaliziranog znaka prepoznaje se obradom od strane tri, u tu svrhu implementirana algoritma. Algoritmom usporedbe slike sa zadanim predloškom, direktno se iz prikaza lokaliziranog znaka saznaje tražena informacija, dok algoritmi neuronske mreže i K najbližeg susjeda to ostvaruju primjenom metoda strojnog učenja. Za potrebe učenja i testiranja navedenih algoritama strojnog učenja kreirana je baza podataka sa slikama 7 – segmentnog formata. Razvijeni programski sustav potpunu funkcionalnost ostvaruje implementacijom na ugradbenom računalu s odgovarajućim ulazno - izlaznim periferijama. Takvi sustavi primjenu pronalaze u polju mjeriteljstva, nadzora ili automatiziranog upravljanja.

Ključne riječi: računalni vid, 7 – segmentni format, OCR, binarizacija slike, prepoznavanje predloškom, neuronska mreža, Knn algoritam, Unix, ugradbeni sustavi

Abstract

Automatized System for Optical Readout of Digital Displays

This thesis describes the course of development of an automatized system for optical readout of the content from digital displays. The developed system implements various algorithms with tasks of localizing a display within the input picture, pre - processing of the localized content and segmentation of the observed sequence by character localization methods. Value of the observed localized character is recognized by three different algorithms implemented for this purpose. Template comparison algorithm has the capability of direct extraction of sought after information form the content of the input image. Neural network and K nearest neighbour algorithms accomplish that task by applying the methods of machine learning. Special database of 7 – segment character images has been created for the purpose of training and testing earlier mentioned algorithms. The developed application accomplishes its full functionality when implemented within the embedded computer with appropriate input - output peripherals. Such systems can be used for purposes of metrology, surveillance and automated control.

Key words: computer vision, 7 – segment format, OCR, image thresholding, template recognition, neural network, Knn algorithm, Unix, embedded systems

Privitak

utilities.py

```
1     #!/usr/bin/python
2
3     import sys
4     import cv2
5     import numpy as np
6
7     def rotate_crop(image, angle):
8
9         h, w = image.shape[:2]
10        center = (w // 2, h // 2)
11
12        M = cv2.getRotationMatrix2D(center, angle, 1)
13        rotated = cv2.warpAffine(image, M, (w, h))
14
15        return rotated
16
17    def resize(img, width=None, height=None):
18
19        dim = None
20        h, w = img.shape[:2]
21
22        if width is None and height is None:
23            return img
24        if width is None:
25            r = height / float(h)
26            dim = (int(w * r), height)
27        else:
28            r = width / float(w)
29            dim = (width, int(h * r))
30
31        resized = cv2.resize(img, dim, interpolation = cv2.INTER_AREA)
32        return resized
33
34    def order_points(pts):
35
36        # top left, top right, bottom right, bottom left
37        rect = np.zeros((4, 2), dtype = "float32")
38
39        s = pts.sum(axis = 1)
40        rect[0] = pts[np.argmin(s)]
41        rect[2] = pts[np.argmax(s)]
42
43        diff = np.diff(pts, axis = 1)
44        rect[1] = pts[np.argmin(diff)]
45        rect[3] = pts[np.argmax(diff)]
46
47        return rect
48
49    def transform_perspective(image, pts):
50
51        rect = order_points(pts)
52        (tl, tr, br, bl) = rect
53
54        # width of new image
```

```
55        widthA = np.sqrt(((br[0] - bl[0]) ** 2) + ((br[1] - bl[1]) ** 2))
56        widthB = np.sqrt(((tr[0] - tl[0]) ** 2) + ((tr[1] - tl[1]) ** 2))
57        maxWidth = max(int(widthA), int(widthB))
58
59        # height of new image
60        heightA = np.sqrt(((tr[0] - br[0]) ** 2) + ((tr[1] - br[1]) ** 2))
61        heightB = np.sqrt(((tl[0] - bl[0]) ** 2) + ((tl[1] - bl[1]) ** 2))
62        maxHeight = max(int(heightA), int(heightB))
63
64        # initialising starting points - same order
65        dst = np.array([
66            (0, 0),
67            (maxWidth - 1, 0),
68            (maxWidth - 1, maxHeight - 1),
69            (0, maxHeight - 1)], dtype = "float32")
70
71        # calculation of transformational matrix
72        M = cv2.getPerspectiveTransform(rect, dst)
73        warped = cv2.warpPerspective(image, M, (maxWidth, maxHeight))
74
75        return warped
```

digit_lookup.py

```
1     #!/usr/bin/python
2
3     import sys
4     import cv2
5     import numpy as np
6
7     LOOKUP_TABLE = {
8         (1, 1, 1, 0, 1, 1, 1): 0,
9         (1, 1, 1, 1, 1, 1, 1): 1,
10        (1, 0, 1, 1, 1, 1, 0): 2,
11        (1, 0, 1, 1, 0, 1, 1): 3,
12        (0, 1, 1, 1, 0, 1, 0): 4,
13        (1, 1, 0, 1, 0, 1, 1): 5,
14        (1, 1, 0, 1, 1, 1, 1): 6,
15        (1, 0, 1, 0, 0, 1, 0): 7,
16        (1, 1, 1, 1, 1, 1, 1): 8,
17        (1, 1, 1, 1, 0, 1, 1): 9
18    }
19
20    def lookup(digitImg, digit_one_w = 10):
21
22        global imgDbg
23        imgDbg = cv2.bitwise_not(digitImg)
24        imgDbg = cv2.cvtColor(imgDbg, cv2.COLOR_GRAY2BGR)
25
26        digH, digW = digitImg.shape[:2]
27        (dW, dH) = (int(digW * 0.25), int(digH * 0.15))
28
29        if (digit_one_w - 5 <= digW <= digit_one_w + 5):
30            return str(1)
31        else:
32            # initialising regions of interest
33            ROI = [
34                (((digW // 2) - (dW // 2), 0), ((digW // 2) + (dW // 2), dH)),
35                ((0, (digH // 4) - (dH // 2)), (dW, (digH // 4) + (dH // 2))),
36                ((digW - dW, (digH // 4) - (dH // 2)), (digW, (digH // 4) + (dH
      // 2))),
```

```
37                (((digW // 2) - (dW // 2), (digH // 2) - (dH // 2)) , ((digW //
    2) + (dW // 2), (digH // 2) + (dH // 2))),
38                ((0, digH - (digH // 4) - (dH // 2)), (dW, digH - (digH // 4) +
    (dH // 2))),
39                ((digW - dW, digH - (digH // 4) - (dH // 2)), (digW, digH -
    (digH // 4) + (dH // 2))),
40                (((digW // 2) - (dW // 2), digH - dH), ((digW // 2) + (dW //
    2), digH))
41                ]
42            setup = [0] * 7
43
44            # iterating through image of single digit
45            for i, ((Xt, Yt), (Xb, Yb)) in enumerate(ROI):
46
47                seg = digitImg[Yt:Yb, Xt:Xb]
48                whites = cv2.countNonZero(seg)
49                segArea = (Xb - Xt) * (Yb - Yt)
50
51                if whites / float(segArea) > 0.15:
52                    setup[i] = 1
53
54                cv2.rectangle(imgDbg, (Xt,Yt), (Xb,Yb), (255, 255, 0), -1)
55
56            setup = tuple(setup)
57            digit = "X"
58            for key in LOOKUP_TABLE:
59
60                if setup == key: digit = str(LOOKUP_TABLE[key])
61
62            return digit
63
64    def decimal_lookup(digitImg):
65
66        digH, digW = digitImg.shape[:2]
67
68        seg = digitImg[digH - 2:digH, digW - 2:digW]
69        whites = cv2.countNonZero(seg)
70        if whites / float(4) > 0.1:
71            return "."
72        else:
73            return ""
74
75    def main():
76
77        # shows the regions of interest
78        # debugging purposes
79        img3 = cv2.imread(sys.argv[1])
80        img3 = cv2.cvtColor(img3, cv2.COLOR_BGR2GRAY)
81        img3 = cv2.inRange(img3, 50, 255)
82        img3 = cv2.bitwise_not(img3)
83
84        _ = lookup(img3)
85        cv2.imwrite('debug_znamenke/debug_digit.jpg', imgDbg)
86
87    if __name__ == "__main__":
88        main()
```

digit_mod.py

```
1     #!/usr/bin/python
2
3     import os
4     import sys
5     import cv2
6     import numpy as np
7     import utilities as ut
8
9     def modify_image(path, image):
10
11        folder, file_name_jpg = os.path.split(path)
12        file_name = file_name_jpg.split('.')[0]
13
14        # optional
15        kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (1, 6))
16        img2 = cv2.morphologyEx(image, cv2.MORPH_OPEN, kernel)
17
18        erodeImg(folder, file_name, img2)
19        dilateImg(folder, file_name, img2)
20
21    def erodeImg(folder, file_name, img):
22
23        for iterations in range(1, 4):
24            for dim in range(1, 4):
25
26                kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (dim + 1,
      dim))
27                if dim % 2 == 0:
28                    img = cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel)
29
30                eroded = cv2.erode(img, kernel, iterations + 2)
31                for angle in range(-4, 4):
32
33                    eroded = ut.rotate_crop(eroded, angle)
34
35                    title = "eroded-r-" + str(angle) + "-dim-" + str(dim) + "-
      i-" + str(iterations)
36                    cv2.imwrite(folder + '/' + file_name + title + '.jpg',
      eroded)
37
38    def dilateImg(folder, file_name, img):
39
40        for iterations in range(1, 5):
41            for dim in range(1, 4):
42
43                kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (dim + 1,
      dim))
44                if dim % 2 == 0:
45                    img = cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel)
46
47                dilated = cv2.dilate(img, kernel, iterations + 2)
48                for angle in range(-2, 4):
49
50                    dilated = ut.rotate_crop(dilated, angle)
51
52                    title = "dilated-r-" + str(angle) + "-dim-" + str(dim) + "-
      i-" + str(iterations)
53                    cv2.imwrite(folder + '/' + file_name + title + '.jpg',
      dilated)
```

```
54
55    def main():
56
57        path = sys.argv[1]
58
59        img = cv2.imread(path, 0)
60        img = cv2.inRange(img, 50, 255)
61
62        modify_image(path, img)
63
64
65    if __name__ == "__main__":
66        main()
```

network.py

```
1     #!/usr/bin/python
2
3     import random
4     import sys
5     import json
6     import numpy as np
7     np.seterr(over='ignore')
8
9     class Network(object):
10
11        def __init__(self, sizes):
12
13            self.num_layers = len(sizes)
14            self.sizes = sizes
15            self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
16            self.weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1],
      sizes[1:])]
17
18        def feedforward(self, a):
19
20            for b, w in zip(self.biases, self.weights):
21                a = sigmoid(np.dot(w, a)+b)
22            return a
23
24        def SGD(self, training_data, epochs, mini_batch_size, eta,
      test_data=None):
25
26            if test_data: n_test = len(test_data)
27            n = len(training_data)
28            for j in xrange(epochs):
29                random.shuffle(training_data)
30                mini_batches = [
31                    training_data[k:k+mini_batch_size]
32                    for k in xrange(0, n, mini_batch_size)]
33                for mini_batch in mini_batches:
34                    self.update_mini_batch(mini_batch, eta)
35                if test_data:
36                    print "Epoch {0}: {1} / {2}".format(j,
      self.evaluate(test_data), n_test)
37                else:
38                    print "Epoch {0} complete".format(j)
39
40        def update_mini_batch(self, mini_batch, eta):
41
42            nabla_b = [np.zeros(b.shape) for b in self.biases]
```

```
43          nabla_w = [np.zeros(w.shape) for w in self.weights]
44          for x, y in mini_batch:
45              delta_nabla_b, delta_nabla_w = self.backprop(x, y)
46              nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
47              nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
48          self.weights = [w-(eta/len(mini_batch))*nw
49                          for w, nw in zip(self.weights, nabla_w)]
50          self.biases = [b-(eta/len(mini_batch))*nb
51                         for b, nb in zip(self.biases, nabla_b)]
52
53      def backprop(self, x, y):
54
55          nabla_b = [np.zeros(b.shape) for b in self.biases]
56          nabla_w = [np.zeros(w.shape) for w in self.weights]
57          # feedforward
58          activation = x
59          activations = [x] # list to store all the activations, layer by
    layer
60          zs = [] # list to store all the z vectors, layer by layer
61          for b, w in zip(self.biases, self.weights):
62              z = np.dot(w, activation)+b
63              zs.append(z)
64              activation = sigmoid(z)
65              activations.append(activation)
66          # backward pass
67          delta = self.cost_derivative(activations[-1], y) *
    sigmoid_prime(zs[-1])
68          nabla_b[-1] = delta
69          nabla_w[-1] = np.dot(delta, activations[-2].transpose())
70
71          for l in xrange(2, self.num_layers):
72              z = zs[-l]
73              sp = sigmoid_prime(z)
74              delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
75              nabla_b[-l] = delta
76              nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
77          return (nabla_b, nabla_w)
78
79      def evaluate(self, test_data):
80
81          test_results = [(np.argmax(self.feedforward(x)), y)
82                          for (x, y) in test_data]
83          return sum(int(x == y) for (x, y) in test_results)
84
85      def cost_derivative(self, output_activations, y):
86
87          return (output_activations-y)
88
89      def save(self, filename):
90
91          data = {"sizes": self.sizes,
92                  "weights": [w.tolist() for w in self.weights],
93                  "biases": [b.tolist() for b in self.biases]}
94          f = open(filename, "w")
95          json.dump(data, f)
96          f.close()
97
98
99  def sigmoid(z):
100     return 1.0/(1.0+np.exp(-z))
101
```

```
102  def sigmoid_prime(z):
103      return sigmoid(z)*(1-sigmoid(z))
```

data_loader.py

```
1    #!/usr/bin/python
2
3    import os
4    import sys
5    import cv2
6    import numpy as np
7
8    def load_data():
9
10       path = ["/home/luka/Desktop/prepared_train",
11               "/home/luka/Desktop/prepared_test"]
12
13       for p in path:
14
15           folder = os.path.split(p)[1]
16           if folder.split('_')[1] == "train": train = True
17           else: train = False
18
19           images = os.listdir(p)
20
21           image_values = []#np.empty((0, len(images)))
22           image_labels = []
23           for image in images:
24               #print(image)
25               img = cv2.imread(p + '/' + image, 0)
26               img = cv2.inRange(img, 50, 255)
27
28               image_values.append(img)
29               tmp = image.split("_")
30               tmp = tmp[1].split(".")
31               label = tmp[0]
32               #print (label)
33               image_labels.append(int(label))
34
35           image_values = np.array(image_values)
36           image_labels = np.array(image_labels)
37
38           if train:
39               training_inputs = [np.reshape(x, (784, 1)) for x in
     image_values]
40               training_results = [vectorized_result(y) for y in image_labels]
41               training_data = zip(training_inputs, training_results)
42           else:
43               test_inputs = [np.reshape(x, (784, 1)) for x in image_values]
44               test_data = zip(test_inputs, image_labels)
45
46       return (training_data, test_data)
47
48   def vectorized_result(j):
49
50       e = np.zeros((10, 1))
51       e[j] = 1.0
52       return e
53
54   def file_setup(image_path, digit, index, train):
55
```

```
56          img = cv2.imread(image_path, 0)
57
58          img2 = cv2.inRange(img, 50, 255)
59          img2 = cv2.resize(img2, (28, 28), interpolation = cv2.INTER_AREA)
60          if train:
61              cv2.imwrite('/home/luka/Desktop/prepared_train/' + str(index) + '_'
    + digit + '.jpg', img2)
62          else:
63              cv2.imwrite('/home/luka/Desktop/prepared_test/' + str(index) + '_'
    + digit + '.jpg', img2)
64
65      def main():
66
67          # folder that contains training and testing material
68          path = "/home/luka/Desktop"
69
70          for name in os.listdir(path):
71              if name.split('_')[0] == "znamenke":
72                  if name.split('_')[1] == "train": train = True
73                  else: train = False
74
75                  path2 = os.path.join(path, name)
76                  print path2
77                  for digit in os.listdir(path2):
78                      path3 = os.path.join(path2, digit)
79                      if os.path.isdir(path3):
80                          print("Setting " + digit)
81                          images = os.listdir(path3)
82                          i = 0
83                          for image in images:
84                              #print (image)
85                              if not image.startswith("."):
86                                  file_setup(path3 + '/' + image, digit, i,
    train=train)
87                                  i += 1
88
89
90      if __name__ == "__main__":
91          main()
```

knn_loader.py

```
1       #!/usr/bin/python
2
3       import sys
4       import os
5       import cv2
6       import numpy as np
7
8       version = "_1_0"
9
10      RESIZED_IMAGE_WIDTH = 28
11      RESIZED_IMAGE_HEIGHT = 28
12
13      int_classifications = []
14      npa_flattened_images = np.empty((0, RESIZED_IMAGE_WIDTH *
        RESIZED_IMAGE_HEIGHT))
15      npa_classifications = []
16
17      trained_folder = "."
18
```

```
19  def setup_values(file_path, char):
20
21      global npa_flattened_images, int_classifications
22
23      if char == "dot":
24          char = "A"
25
26      img = cv2.imread(file_path)
27      imgGray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
28      imgThreshCopy = imgGray.copy()
29      imgROIResized = cv2.resize(imgThreshCopy, (RESIZED_IMAGE_WIDTH,
    RESIZED_IMAGE_HEIGHT))
30
31      int_classifications.append(ord(char))
32      npaFlattenedImage = imgROIResized.reshape((1, RESIZED_IMAGE_WIDTH *
    RESIZED_IMAGE_HEIGHT))
33      npa_flattened_images = np.append(npa_flattened_images,
    npaFlattenedImage, 0)
34
35  def main():
36
37      training_dir = "/home/luka/Desktop/znamenke_test"
38
39      for fname in os.listdir(training_dir):
40
41          path = os.path.join(training_dir, fname)
42          if os.path.isdir(path):
43              print("Training " + fname)
44              tfiles = os.listdir(path)
45              for tfile in tfiles:
46
47                  if not tfile.startswith("."):
48                      setup_values(path + "/" + tfile, fname)
49
50      # Save the classifications for use in Python
51      fltClassifications = np.array(int_classifications, np.float32)
52      npaClassifications =
    fltClassifications.reshape((fltClassifications.size, 1))
53      np.savetxt(trained_folder + "/KNNclassifications" + version + ".txt",
    npaClassifications)
54      np.savetxt(trained_folder + "/KNNflattened_images" + version + ".txt",
    npa_flattened_images)
55
56  if __name__ == "__main__":
57      main()
```

knn_process.py

```
1   #!/usr/bin/python
2
3   import sys
4   import os
5   import cv2
6   import numpy as np
7
8   RESIZED_IMAGE_WIDTH = 28
9   RESIZED_IMAGE_HEIGHT = 28
10
11  class Knn_processing:
12
13      def __init__(self, version, debug=False):
```

```
14
15          self.folder = "."
16          self.debug = debug
17          self.version = version
18
19          self.knn = self.train_knn(self.version)
20
21      def train_knn(self, version):
22
23          # read in training classifications
24          npa_classifications = np.loadtxt(self.folder +
    "/KNNclassifications" + version + ".txt", np.float32)
25          # read in training images
26          npa_flattened_images = np.loadtxt(self.folder +
    "/KNNflattened_images" + version + ".txt", np.float32)
27
28          npa_classifications =
    npa_classifications.reshape((npa_classifications.size, 1))
29          k_nearest = cv2.KNearest()
30          k_nearest.train(npa_flattened_images, npa_classifications)
31
32          return k_nearest
33
34      def predict_digit(self, digit_mat):
35
36          # Resize the image
37          imgROIResized = cv2.resize(digit_mat, (RESIZED_IMAGE_WIDTH,
    RESIZED_IMAGE_HEIGHT))
38          # Reshape the image
39          npaROIResized = imgROIResized.reshape((1, RESIZED_IMAGE_WIDTH *
    RESIZED_IMAGE_HEIGHT))
40          # Convert it to floats
41          npaROIResized = np.float32(npaROIResized)
42          _, results, neigh_resp, dists =
    self.knn.find_nearest(npaROIResized, k=1)
43          predicted_digit = str(chr(int(results[0][0])))
44
45          if predicted_digit == 'A':
46              predicted_digit = '.'
47
48          return predicted_digit
```

Primjeri testiranja

```
Debug parameter_1: 54
Debug parameter_2: 10
Debug parameter_3: 5

Value detected by pattern: 505.67

Value detected by network algorithm: 404.47

Value detected by knn algorithm: 505.67
```

```
Debug parameter_1: 148
Debug parameter_2: 10
Debug parameter_3: 3

Value detected by pattern: 79.8

Value detected by network algorithm: 73.4

Value detected by knn algorithm: 79.8
```

```
Debug parameter_1: 111
Debug parameter_2: 15
Debug parameter_3: 3

Value detected by pattern: 037

Value detected by network algorithm: 037

Value detected by knn algorithm: 037
```

```
Debug parameter_1: 66
Debug parameter_2: 10
Debug parameter_3: 3

Value detected by pattern: 34.5

Value detected by network algorithm: 34.5

Value detected by knn algorithm: 34.5
```

```
Debug parameter_1: 134
Debug parameter_2: 10
Debug parameter_3: 2

Value detected by pattern: 0.0

Value detected by network algorithm: 0.0

Value detected by knn algorithm: 0.0
```

```
Debug parameter_1: 18
Debug parameter_2: 13
Debug parameter_3: 5

Value detected by pattern: 41.368

Value detected by network algorithm: 41.344

Value detected by knn algorithm: 41.368
```

```
Debug parameter_1: 39
Debug parameter_2: 10
Debug parameter_3: 4

Value detected by pattern: 8.8.8.8.

Value detected by network algorithm: 4.4.4.4.

Value detected by knn algorithm: 8.8.8.8.
```

```
Debug parameter_1: 163
Debug parameter_2: 10
Debug parameter_3: 5

Value detected by pattern: 10.000

Value detected by network algorithm: 14.445

Value detected by knn algorithm: 10.007
```