UNIVERZA V MARIBORU

FAKULTETA ZA ELEKTROTEHNIKO,

RAČUNALNIŠTVO IN INFORMATIKO

Marko Watzak

PAMETNI POMOČNIK ZA VOŽNJO PO DIRKAŠKI STEZI NA OPERACIJSKEM SISTEMU ANDROID

Diplomsko delo

Maribor, avgust 2021

PAMETNI POMOČNIK ZA VOŽNJO PO DIRKAŠKI STEZI NA OPERACIJSKEM SISTEMU ANDROID

Diplomsko delo

Študent:	Marko Watzak
Študijski program:	Visokošolski strokovni,
	Računalništvo in informacijske tehnologije
Smer:	/
Mentor:	red. prof. dr. Milan Zorman, univ. dipl. inž. rač. in inf.
Lektorica:	Polonca Zlodej, prof. SLJ–PTHV

Zahvala

Zahvaljujem se mentorju prof. dr. Milanu Zormanu za vodstvo in pomoč pri ustvarjanju in pisanju diplomskega dela. Zahvaljujem se tudi svojim staršem, ki so mi omogočili študij in mi stali ob strani.

Pametni pomočnik za vožnjo po dirkaški stezi na operacijskem sistemu Android

Ključne besede: OBD, OBD II, strojno učenje, Java, Android

UDK: 629.052.025.14:004.451.9(043.2)

Povzetek

Že vse odkar se je človek trudil naučiti stroj določenih veščin predvidevanja in lastne inteligence, je tukaj prisotno strojno učenje. Nemalo kdo je že velikokrat poprej prišel na misel, da bi lahko z uporabo strojnega učenja izboljšali različne športe, med drugim tudi avtomobilske. V tem diplomskem delu smo podali namen, kjer lahko skupaj s strojnim učenjem poskusimo izboljšati vožnjo posameznika na dirkaški stezi in to uspešno zabeležiti ter predstaviti na človeku razumljiv način. V našem delu smo opisali načine za reševanje problema, podali razvojna okolja in tehnologijo ter prikazali izsledke. Analizo vožnje je bilo mogoče uspešno izvesti in uspešno interpretirati dobljene rezultate v različnih scenarijih.

Smart race track assistant for Android OS

Keywords: OBD, OBD II, machine learning, Java, Android

UDK: 629.052.025.14:004.451.9(043.2)

Abstract

Ever since people have been trying to teach a machine the skills of prediction and own intelligence, machine learning was present. Countless individuals have thought about using the machine learning for improving different sports, amongst them motorsports. In this diploma we provided a purpose to use machine learning approach to improve an individual's driving on a race track and record the data to be presented in humanly readable form. In our work, we have described various ways of solving the given problem, provided the necessary tools and technology and interpolated the results. Analysis was successfully performed and data interpreted in various scenarios.

SEZNAM KRATIC

API – aplikacijski programski vmesnik (ang. “Application Programming Interface”)

CSV – vrednosti, ločene z vejico (ang. “Comma Separated Values”)

KERS – Sistem za shranjevanje kinetične energije (ang. “Kinetic Energy Recovery System”)

DRS – Sistem za zmanjševanje upora (ang. “Drag reduction system”)

OBD – diagnostika “na krovu” oziroma znotraj vozila (ang. “On Board Diagnostics”)

Weka – Okolje Waikato za analizo znanja (ang. “Waikato Environment for Knowledge Analysis”)

GNU – Splošna javna licenca (ang. “General Public License”)

DTC – diagnostične kode težav (ang. “Diagnostic Trouble Codes”)

GPS – Globalni sistem za pozicijo (ang. “Global Positioning System”).

ARFF – Format za atributne povezave (ang. "Attribute-Relation File Format")

FIFO – Prvi noter, prvi ven (ang. “First In First Out”).

KAZALO

1 UVOD ... 1
2 PREGLED STROJNEGA UČENJA V DIRKALNIH ŠPORTIH ... 3
3 UPORABLJENE METODE IN RAZVOJNO OKOLJE ... 4
3.1 Razvojno okolje Android Studio ... 4
4 NAČRTOVANJE APLIKACIJE ... 5
4.1 Weka ... 6
4.2 FireBase ... 7
5 PROTOKOL OBD ... 8
5.1 Protokol OBD-II ... 8
5.2 Protokol EOBD ... 9
5.3 EOBD kode napak ... 9
6 DELOVANJE APLIKACIJE ... 10
7 IMPLEMENTACIJSKE PODROBNOSTI ... 11
7.1 OBD II Java API ... 11
7.2 Razred OBDCommand (ObdUkaz) ... 13
7.3 OBD servis ... 14
7.4 Pospeškometer ... 22
7.5 Servis za lokacijo ... 29
7.6 Merjenje sektorjev preko GPS lokacije ... 33
7.7 Zapis podatkov v datoteko ... 40
7.8 Strojno učenje in prikaz podatkov ... 41
8 GRAFIČNI VMESNIK APLIKACIJE ... 44
8.1 Glavni zaslon aplikacije ... 44
8.2 Konfiguracija podatkov in povezave ... 47
8.3 Postavitev sektorjev ... 50
9 TESTIRANJE APLIKACIJE ... 51

9.1 Testiranje aplikacije izven avtomobila 51
9.2 Aplikacija Lockito 52
9.3 Predstavitev rezultatov simulacije 53
9.4 Testiranje aplikacije v avtomobilu brez uporabe vmesnika Bluetooth 56
9.5 Testiranje aplikacije z uporabo vmesnika Bluetooth 60
10 ZAKLJUČEK...63
11 VIRI..65

KAZALO SLIK

Slika 3.1.1: Osnovni prikaz strukture delovanja aplikacije....................... 5
Slika 7.1.1: Prikaz povezave z Bluetooth vmesnikom 12
Slika 7.2.1: Razred ObdUkaz 14
Slika 7.3.1: Primer zagonskih ukazov 15
Slika 7.3.2: Konfiguriranje zagonskih ukazov..................................16
Slika 7.3.3: Dodajanje ukaza v vrsto..17
Slika 7.3.4: Prikaz izvedbe vrste ukazov 18
Slika 7.3.5: Prikaz izvajanja ukazov ter prikazovanje na števcih 19
Slika 7.3.6: Zagon branja podatkov v živo 20
Slika 7.3.7: Zagon servisa za zajem podatkov 21
Slika 7.3.8: Ustavitev servisa..22
Slika 7.4.1: Primer uporabe senzorjev..22
Slika 7.4.2: Skica pospeškometra [14] 24
Slika 7.4.3: Pristop filtriranja vrednosti pospeškometra.........................25
Slika 7.4.4: Prebiranje vrednosti iz senzorja 26
Slika 7.4.5: Filtriranje podatkov po mediani...................................27
Slika 7.4.6: Osveževanje pospeškometra na grafičnem vmesniku 28
Slika 7.5.1: Dovoljenja ter vključevanje paketov 29
Slika 7.5.2: Pošiljanje podatkov na servis za lokacijo.............................30
Slika 7.5.3: Komunikacija in implementacija servisa za lokacijo 31
Slika 7.5.4: Zagon in ustavitev servisa v glavni aktivnosti aplikacije 32

Slika 7.5.5: Preverjanje aktivnega servisa za lokacijo 33
Slika 7.6.1: Pokrivanje možnih scenarijev pri vožnji skozi sektorje 34
Slika 7.6.2: Lastnosti posameznega zabeleženega sektorja 36
Slika 7.6.3: Primer preverjanja sektorja za nov krog 36
Slika 7.6.4: Shranjevanje podatkov ob prevoženem sektorju 38
Slika 7.6.5: Preverjanje sektorja, kjer naslednji ni nov krog....................39
Slika 7.7.1: Zapis podatkov v datoteko..40
Slika 7.8.1: Vzorčni primer željenega prikaza 41
Slika 7.8.2: Prikaz grafa vožnje...43
Slika 8.1.1: Osnovni pogled aplikacije (slika levo) in pogled ob polni funkcionalnosti (slika desno) 45
Slika 8.2.1: Konfiguracija aplikacije – nastavitev pnevmatik in pritiska (slika levo) ter Bluetooth in pripadajoče nastavitve (slika desno) 49
Slika 8.3.1: Postavitev sektorjev 50
Slika 9.2.1: Prilagajanje hitrosti in ustavitev simulacije (slika levo) in zagon simulacije z nastavitvami natančnosti ter načina (slika desno).....53
Slika 9.3.1: Rezultati strojnega učenja na grafu (slika levo) in izmerjeni časi (slika desno) 55
Slika 9.4.1: Postavitev sektorjev 57
Slika 9.4.2: Graf vožnje 59
Slika 9.5.1: Postavitev sektorjev za vožnjo 60
Slika 9.5.2: Rezultati testiranja z OBD II vmesnikom 61
Slika 9.5.3: Graf vožnje 62

KAZALO TABEL

Tabela 1: Rezultat simulacije vožnje 54
Tabela 2: Zbrane vrednosti in predikcija 57
Tabela 3: Podatki vožnje...58
Tabela 4: Tabela časov vožnje z OBD II vmesnikom...............................62

1 UVOD

V realnem življenju se v teku zadnjih desetih let področje računalništva seli iz prenosnih in namiznih računalnikov v vozila. Iz vozil, kot kompletno mehanskih naprav, so se počasi razvila vozila, ki premorejo tako rekoč že nekakšno moč razmišljanja in predvidevanja. Ti računalniki so namenjeni različnim stvarem; od merjenja in upravljanja z motorjem ter prikazovanjem dodatnih parametrov vozila. S temi vgrajenimi sistemi v avtomobilih hitreje zaznamo morebitne napake, prav tako pa nam le-ti tudi omogočajo večjo varnost ter lažje upravljanje z vozilom. Nasprotno pa se lahko pridobljena moč računalniškega znanja uporabi tudi v drugih ekstremih, in sicer čedalje večji uporabi v avtomobilskih dirkah. Ogromno sistemov, kateri so domala revolucionizirali avtomoto šport, se je razvilo iz ideje posameznika, kako bi lahko računalniško moč usmeril v čim bolj optimizirano dirkalno vožnjo, ter dvignil meje tega športa na novo raven. Eden izmed teh primerov se lahko dobro vidi v uporabi sistemov KERS in DRS v dirkalnikih formule 1. Sistem KERS (ang. "Kinetic energy recovery system") je bil zasnovan tako, da omogoča zbiranje energije ob zaviranju, katero nato shrani v baterije, nameščene na pogonih, in jo lahko sprosti ob pritisku na gumb. Sistem DRS (ang. "Drag reduction system") omogoča spuščanje in dviganje zadnjega krilca ter s tem maksimizira oz. minimalizira upor ob posredovanju voznika. V primerih rekreativne vožnje navdušencev nad avtomobilskimi športi takšni radikalni posegi v optimalnost vožnje niso potrebni, a vendar vedno obstajajo kakšni mikro popravki, ki omogočajo izboljšanje časa na krog, ohranjanje hitrosti skozi zavoje, optimalna dirkaška linija ipd. Tukaj nastopi izhodišče naše diplomske naloge. Naš namen je razviti aplikacijo, ki bo lahko vozniku na podlagi analize njegove vožnje pomagala izpostaviti napake ter izboljšati njegovo vožnjo. Z merjenjem podatkov, kot so podatki avtomobila (obrati, hitrost, obremenitev …) ter merjenje pospeška v vse smeri bomo pridobili nekakšno predstavo vožnje, katero bomo potem izpostavili strojnem učenju in poskušali predvidevati najboljši možen izid.

V tem diplomskem delu se bomo osredotočili na podatke avtomobila oziroma OBD (ang. "On Board Diagnostic"), ki jih lahko iz avtomobila pridobimo s pomočjo vmesnika Bluetooth ter jih obdelamo preko pristopa strojnega učenja. Za nas so predvsem zanimivi podatki o hitrosti, obratih, obremenitvi motorja in položaju lopute. Za ostale sile, ki vplivajo na avtomobil med vožnjo, bomo implementirali pospeškometer, ki lahko na osnovi senzorjev pametnega telefona določi gibanje po vseh treh oseh (x, y in z). Pridobljeni podatki se bodo v obliki vožnej zapisovali v povezano storitev ponudnika Google, imenovano Firebase, iz katere se bodo nato prenesli in obdelali preko strojnega učenja. Namen te diplomske naloge je množico podatkov obdelati s pomočjo strojnega učenja in jih čim bolj približati optimalni oceni prevoženega kroga.

Struktura tega diplomskega dela je napisana po sledečem vrstnem redu: Najprej pogledamo obstoječe postopke strojnega učenja v dirkalnih športih ter njihove prednosti in slabosti. V naslednjem poglavju spoznamo uporabljene metode in razvojno okolje ter vozilo, na katerem bodo izvedeni testi. V sledečem poglavju se spustimo globlje v samo aplikacijo, kjer se bomo dotaknili in predstavili samo delovanje aplikacije, njene implementacijske podrobnosti in uporabniški vmesnik ter navodila za uporabo. V poglavju 5 bomo spoznali vzpostavljeno eksperimentalno okolje za izvedbo meritev. V poglavju Rezultati bodo predstavljeni rezultati testiranja aplikacije na dirkaški stezi ter uporabljeni pristopi strojnega učenja. Sledita poglavji z naslovom Zaključek, kjer bomo ocenili uspešnost aplikacije in podali možne predloge za izboljšave in Viri, kjer bodo zapisani uporabljeni viri v tem diplomskem delu.

2 PREGLED STROJNEGA UČENJA V DIRKALNIH ŠPORTIH

Strojno učenje je v zadnjih letih pridobilo na popularnosti pri uporabi v dirkalnih športih, saj lahko z uporabo le-tega natančneje opredelimo pridobljene zbrane podatke ter z njim ustvarjamo simulacije, ki pomagajo pri nadaljnjem razvoju komponent dirkalnikov in prog. Uporabo strojnega učenja so namenili tudi predvidevanju vremena na lokacijah dirk, kjer bi s tem lahko pridobili pomembno prednost pred tekmeci, druga uporaba pa je tudi profiliranje voznika med samo dirko, s čimer lahko potem določijo ter predvidijo samo obrabo pnevmatik in drugih materialov. Pred leti je ameriški inštitut MIT razvil programsko opremo za predvidevanje raznih dejavnikov med dirko, vse od samega štarta do ekipe za postanek [18] . Amazon pa je pred časom razvil platformo, ki temelji na oblaku, imenovano AWS DeepRacer. AWS DeepRacer omogoča razvijalcev z različnimi izkušnjami, da lahko preko 3D-simulatorja trenirajo avtomobilček na progi ter z njim tudi tekmujejo na Amazon-ovih tekmovanjih. Amazonov izdelek je plačljiv, vendar ponuja najboljši način interakcije pristopa strojnega učenja različnim razvijalcem, bodisi izkušenim bodisi neizkušenim. Uporaba strojnega učenja se je začela pogosteje pojavljati tudi v Moto GP, saj je prav tako možno zbrati podatke ter na podlagi le-teh napovedati določene dejavnike, ki lahko potem vplivajo na uspešnost posameznega moštva. Ekipa Ducati je ena prvih v tem športu, kjer je strojno učenje postalo sestavni del njihovega razvoja ter testiranja novih tekmovalnih motociklov. Pri zbiranju podatkov jim pomagajo različni senzorji motorja, ki zagotavljajo najrazličnejši nabor podatkov, od temperature do zračnega upora. Ti podatki jim nato pomagajo simulirati različne razmere, kar zmanjša čas ter stroške v primerjavi s klasičnim testiranjem na progi. Zbrani podatki tudi pomagajo pri boljših konfiguracijah ter posledično boljših časih na progi, saj lahko na simulaciji kontrolirano izvajajo tudi najbolj kompleksne teste, kar jih tudi naredi bolj konkurenčne. Kot zanimivost, japonski proizvajalec motorjev in avtomobilov Honda, je pred kratkim predstavil motor, ki se lahko samostojno uravnoveša pri vožnji, pri čemer mu je tako pri razvoju, kot tudi delovanju pomagalo strojno učenje [9].

3 UPORABLJENE METODE IN RAZVOJNO OKOLJE

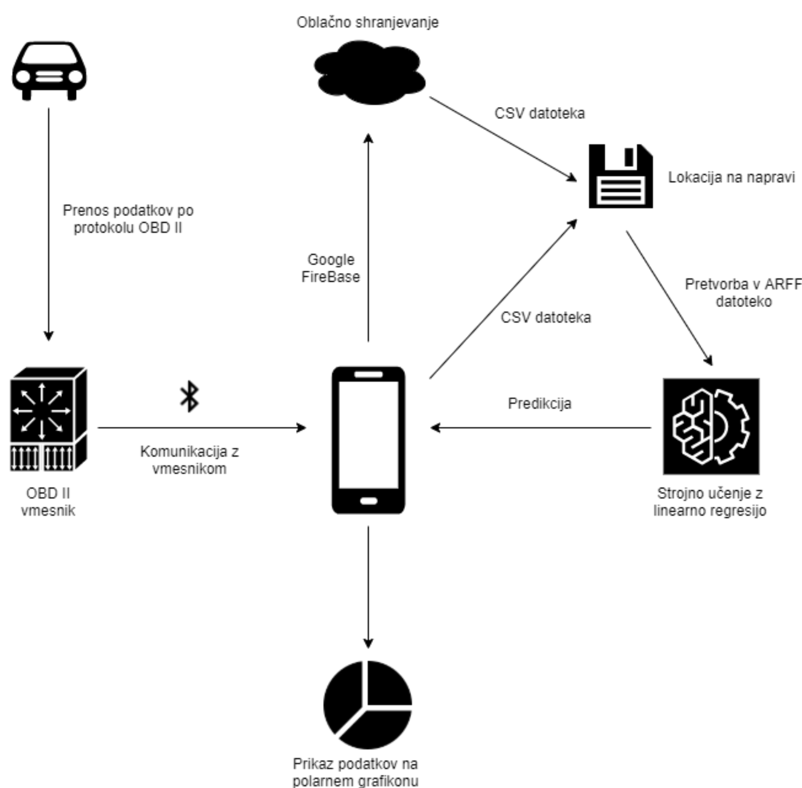
Pri predvidenih metodah smo v diplomskem delu izbrali povezavo z avtomobilskim sistemom OBD in uporabo Android-ovih senzorjev za meritev pospeškov ter najaktualnejši nabor algoritmov za izvajanje strojnega učenja, natančeje regresije. Regresija je v našem primeru uporabljena zato, ker lahko na posamezno instanco predvidi določeno vrednost, torej čas. Večina algoritmov za strojno učenje pa deluje na podlagi uspešno oz. neuspešno klasificiranih instanc, kar pa v našem primeru ni zadovoljivo, saj moramo uporabniku prikazati, kako in če le lahko, lahko izboljša svoj čas vožnje.

3.1 Razvojno okolje Android Studio

Za uporabo razvojnega okolja, glede na to da izdelujemo aplikacijo za mobilno napravo z operacijskim sistemom Android, pride v poštev Android Studio. Samo razvojno orodje tukaj omogoča vgrajeno podporo za delo z razvojem aplikacije za Android, kjer je potrebno od standardnega programiranja v programskem jeziku Java upoštevati tudi posebnosti. Android Studio je na voljo kot odprtokodno razvojno orodje za prenos na spletu. V primerjavi z ostalimi orodji za delo z Javo (IntelliJ, Eclipse itd.) Android Studio omogoča ustvarjanje manifesta ter datoteke Gradle, ki je unikatna razvoju aplikacij Android, prav tako pa ima vgrajeno tudi podporo za emulator, kar lahko znatno pripomore k razvoju. Za programski jezik smo se odločali med Javo in Kotlin-om, vendar ima, kljub preprostosti slednjega, Java še vedno večjo podporo ter večjo razvojno bazo, saj se je Kotlin kot programski jezik začel uporabljati šele nekaj časa nazaj, medtem ko je Java že dobro uveljavljen programski jezik.

4 NAČRTOVANJE APLIKACIJE

Našo aplikacijo smo zasnovali tako, da le-ta uporabniku omogoča celovit in enostaven pregled parametrov vozila. Glede na to, da OBD protokol, katerega uporabljajo vsa vozila že od nekje sredine 90. let 20. stoletja, omogoča spremljanje vseh parametrov vozila, od temperature hladilne tekočine ter vse do obratov motorja, se nam je zdelo smiselno izpostaviti le najbolj ustrezne. V ta namen smo zato za vožnjo ustrezne podatke izbrali obrate motorja, pozicijo lopute, obremenitev vozila in hitrost. Ostali parametri, kot so recimo temperatura hladilne tekočine in recimo merjenje zmesi goriva in zraka so tudi pomembne pri vožnji, vendar za naš namen obdelave podatkov preko strojnega učenja gledamo le osnovne parametre vozila, s katerimi lahko prikažemo oziroma simuliramo prevoženo pot v uporabne podatke za strojno učenje.



Slika 3.1.1: Osnovni prikaz strukture delovanja aplikacije

Slika 3.1.1 prikazuje zelo grob prikaz delovanja aplikacije. Pri zagonu snemanja vožnje oziroma zbiranja podatkov v živo najprej preberemo podatke iz vozila, ki podpira in uporablja protokol OBD II v samo aplikacijo preko uporabe vmesnika Bluetooth. Po uspešno odvoženih krogih oziroma ko zaključimo snemanje vožnje, lahko podatke shranimo v Google-ov FireBase ali pa neposredno v mapo, ki se nahaja na notranjem pomnilniku telefona ali v spominski kartici. Vsi zbrani podatki bodo zapisani v datoteko tipa CSV (ang. "Comma separated values"). V primeru, da to datoteko prenašamo iz FireBase, pa moramo izvoziti podatke kot datoteko CSV. Nad temi podatki potem izvedemo strojno učenje, kot to prikazuje čisto desna ikona, ter pri tem uporabimo linearno regresijo, same podatke pa nato prikažemo v aplikaciji kot polarni grafikon.

4.1 Weka

Okolje za analizo znanja Weka (ang. "Waikato Environment for Knowledge Analysis") je odprtokodna programska oprema, ki so jo razvili na univerzi v Waikato-u na Novi Zelandiji [14]. Programska oprema deluje pod licenco GNU (ang. "General Public License"). WEKA vsebuje zbirko vizualizacijskih orodij in algoritmov za analizo podatkov ter modeliranje na osnovi predvidevanj oz. predikcij. Primarna verzija programa je bila izdelana kot orodje za analizo podatkov iz agrokulturnih domen, sedaj pa se uporablja v izobraževalne namene in raziskave. Orodje WEKA je napisano primarno v programskem jeziku Java in je na voljo na vseh operacijskih sistemih (Windows, različne distribucije Linux-a, OS X…). Podpora programa omogoča tudi več standardnih postopkov za podatkovno rudarjenje in preprocesiranje podatkov, grozdenje (ang. "clustering"), klasifikacijo, regresijo ter vizualizacijo. Vse tehnike orodja WEKA so uporabljene pod domnevo, da so podatki na voljo v eni datoteki oziroma kot relacije, kjer je vsak podatek opisan z določenim številom atributov, ki pa so primarno numerični oz. nominalni (opisni).

4.2 FireBase

Firebase je platforma, katero je razvil računalniški gigant Google, namen le-te pa je razvoj mobilnih in spletnih aplikacij. Začetek Firebase-a je nastal iz ideje zagona podjetja (ang. "startup") in je bil takrat imenovan Envolve. Njegov glavni namen je bil, da je omogočal razvijalcem možnost uporabe API-ja za integracijo klepetalnika v spletne strani, a so ga razvijalci raje uporabili v namen prenosa podatkov v aplikaciji. Iz te ideje pa je potem nastal nov projekt, imenovan Firebase, ki omogoča realnočasovno bazo (ang. "Real Time Database") čez platforme, kot so iOS, Android in spletne naprave, ter podatke shranjuje v oblaku (ang. "cloud"). Pri razvijalcih se najbolj uporablja pri izdelavi realnočasovnih aplikacij, ki sodelujejo med seboj [20].

5 PROTOKOL OBD

Protokol OBD oz. diagnostika “na krovu” (ang. “On-board diagnostics”) je avtomobilistični izraz, ki se nanaša na samodiagnostično in sporočevalno zmožnost vozila [19]. Ti sistemi omogočajo vozniku oziroma mehaniku možnost dostopa do različnih podsistemov vozila. Začetki vgrajene diagnostike so bili že v 80. letih prejšnjega stoletja, vendar pa so le-te omogočale samo prižig lučke za napako, ne pa tudi podrobnega opisa, ki bi pomagal pri točnem določanju napake. Moderna uporaba OBD omogoča standardizirano digitalno komunikacijo preko vtičnika na avtomobilih za prikaz realnočasovnih podatkov pri dodanih standardiziranih serijah različnih kod, imenovanih kode za diagnostiko oziroma s kratico DTC (ang. “Diagnostic Trouble Codes”), ki omogočajo hitro ugotavljanje napak. Za nas je predvsem zanimiv protokol OBD-II, katerega še danes uporablja velika večina avtomobilov, ki so bili narejeni v časovnem razponu do 20 let nazaj.

5.1 Protokol OBD-II

Protokol OBD-II je prišel v uporabo kot nadgradnja zastarelemu protokolu OBD-I in je bolj izpopolnjen, tako po kapaciteti kot tudi standardizaciji. OBD-II standard specificira tip diagnostičnega priključka in razdelitev pinov, signalizacijo protokolov, ki so na voljo, in format, v katerem se pošiljajo sporočila. Kot največja nadgradnja nad predhodnikom pa je njegov seznam kod za diagnostiko napak. Rezultat te standardizacije je to, da lahko ena sama naprava izvede poizvedbo nad vsemi sistemi in podsistemi vozila. Običajna koda napake na vozilu je štirimestna, pred njo pa je ponavadi črka: P označuje motor in menjalnik, B označuje karoserijo, C označuje šasijo in U označuje omrežje [19].

5.2 Protokol EOBD

Protokol EOBD je bil zaradi predpisov narejen kot evropska verzija opisanega protokola OBD-II in se nanaša na vsa vozila kategorije M1 (torej vsa vozila z manj kot 9 sedeži za potnike in skupna teža, katera je manjša od 2500 kg). V uporabo je za vozila na bencinski pogon prišel s 1. januarjem 2001, tri leta pozneje pa tudi za vozila na dizelski pogon [19].

5.3 EOBD kode napak

Vsaka koda napake ima 5 znakov, začne se s črko, nato pa sledijo štiri številke. Tudi tukaj velja način, opisan zgoraj, kjer različne črke pomenijo različne sklope vozila z zabeleženo napako, medtem ko drugo število, ki je za evropska vozila ponavadi 0, pomeni, da je skladno z EOBD standardom [19]. Klasična oblika kode z napako je tako P0XXX. Naslednje število se nanaša na podsistem, zadnji dve mesti pa pripadata posamezni napaki v vsakem podsistemu. Poznamo različne podsisteme:

- Merjenje goriva in zraka in zunanja emisijska kontrola (ang. "Fuel and Air Metering and Auxillary Emission Controls"), koda P00XX,
- Merjenje goriva in zraka (ang. "Fuel and Air Metering), koda P01XX,
- Merjenje goriva in zraka – elektronika vbrizgalnih šob (ang. "Fuel and Air Metering (Injector Circuit), koda P02XX,
- Sistem uspešnega oziroma neuspešnega vžiga (ang. "Ignition System or Misfire"), koda P03XX,
- Zunanja kontrola emisij (ang. "Auxillary Emissions Controls"), koda P04XX,
- Kontrola hitrosti vozila in prostega teka (ang. "Vehicle Speed Controls and Idle Control System), koda P05XX,
- Izpis vezja računalnika (ang. "Computer Output Circuit"), koda P06XX,
- Menjalniški sklop (ang. "Transmission"), kodi P07XX in P08XX.

6 DELOVANJE APLIKACIJE

Pri opisu delovanja aplikacije se bomo nekoliko globlje posvetili sami zasnovi ter predstavitvi ideje. Aplikacijo smo zasnovali tako, da lahko v realnem času preko OBD protokola iz avtomobila prebira podatke, katere tudi nato prikazuje v uporabniškem vmesniku aplikacije, o katerem bomo govorili nekoliko kasneje. Prikazani podatki, kot so omenjeni zgoraj, so obrati motorja, pozicija lopute za plin, absolutna obremenitev ter hitrost, prikazane pa imamo tudi podatke o linearnem pospešku ter položaju GPS. Samo beleženje podatkov o poziciji uporabnika ni pomembno samo za strojno učenje, ampak pride v poštev pri izrisovanju prevožene poti ter pri postavljanju sektorjev. Ti zbrani podatki se tekom merjenja vožnje zapisujejo na spletno storitev Google-a, imenovano Firebase [20], kjer pa se nato shranjujejo v realnočasovno bazo. V bazo torej zapisujemo zgoraj naštete podatke, poleg tega pa tudi čas vožnje (le-ta se zapisuje v milisekundah), dimenzijo pnevmatik, tip pnevmatik, pritisk v sprednjih ter zadnjih pnevmatikah ter podatke o položaju GPS (ang. "Global Positioning System"). Tako shranjene podatke potem pridobimo iz realnočasovne baze in jih zapišemo kot datoteko tipa CSV (ang. "Comma Separated Values"), kjer so, kot že pove ime samo, podatki ločeni z vejicami. Takšen način zapisa je pomemben za nadaljnjo obdelavo podatkov, saj se potem pri pripravi podatkov za strojno učenje ta format pretvori v ARFF datoteko, katero lahko potem bere ogrodje za strojno učenje, v našem primeru je to Weka. Strojno učenje nato te podatke preprocesira in zgradi model preko uporabljene metode, ki se imenuje regresija. Regresija je način strojnega učenja, ki nadzorovano pomaga pri iskanju korelacij med podatki, torej med njimi išče povezave in nam omogoča predvidevanje izhodne vrednosti glede na spremenljivke, na kateri izvajamo predvidevanje. V našem primeru torej uporabimo metodo regresije na izmerjenem času proge, kjer glede na izmerjen čas preko strojnega učenja predvidevamo boljši čas. Aplikacija nato prikaže primerjavo med starim in novim časom.

7 IMPLEMENTACIJSKE PODROBNOSTI

V tem poglavju se bomo nekoliko podrobneje poglobili v same implementacijske podrobnosti aplikacije. Dotaknili se bomo vseh uporabljenih obstoječih metod in pristopov, ki so že bili razviti v ta namen reševanja problema diagnostike ter merjenja parametrov vozila med vožnjo. Vse metode, katere bomo opisali, temeljijo na OBD II protokolu, ki je bil opisan že v prejšnjem poglavju.

7.1 OBD II Java API

V samem začetku razvoja smo se najprej lotili povezave pametne naprave (mobilnega telefona) s 16-pinskim Bluetooth vmesnikom, kateri nato preko protokola OBD II omogoča komunikacijo ter pridobivanje podatkov iz vozila. V primeru programskega jezika Java in razvojnega okolja Android Studio smo se tega lotili tako, da smo najprej prebrali dokumentacijo o uporabi in komunikaciji z Bluetooth na Android napravah [1][2]. V veliko pomoč nam je bil že spisan API, kateri že skrbi za razčlenjevanje podatkov iz vmesnika Bluetooth ter formatiranje nizov podatkov z ukazi, katere nato razume OBD II protokol. Ta API je, kljub temu da se je razvoj le-tega opustil že kar nekaj let nazaj, še vedno dostopen na spletni strani GitHub [2]. Za ta API smo se odločili, ker je sama implementacija protokola dokaj velik zalogaj, prav tako tudi ni primarni namen tega diplomskega dela. V nadaljevanju bomo pojasnili delovanje nekaj ključnih komponent za branje vhodnih podatkov iz vmesnika Bluetooth ter pretvorbo le-teh v OBD II ukaze ter zapis le-teh, katere vsebuje zgornji API.

```java
public class BluetoothPovezava {
    private static final String TAG = bluetoothPovezava.class.getName();
    private static final UUID MY_UUID = UUID.fromString("00001101-0000-1000-
8000-00805F9B34FB");
    public static BluetoothSocket povezi(BluetoothDevice dev) throws
IOException {
        BluetoothSocket sock = null;
        BluetoothSocket sockFallback = null;
        Log.e(TAG, "Začenjam povezavo z OBDII vmesnikom..");
        try {
            sock = dev.createRfcommSocketToServiceRecord(MY_UUID);
            sock.connect();
        } catch (Exception e1) {
            Log.e(TAG, "Napaka pri vzpostavljanju Bluetooth povezave!", e1);
            Class<?> clazz = sock.getRemoteDevice().getClass();
            Class<?>[] paramTypes = new Class<?>[]{Integer.TYPE};
            try {
                Method m = clazz.getMethod("createRfcommSocket", paramTypes);
                Object[] params = new Object[]{Integer.valueOf(1)};
                sockFallback = (BluetoothSocket)
m.invoke(sock.getRemoteDevice(), params);
                sockFallback.connect();
                sock = sockFallback;
            } catch (Exception e2) {
                Log.e(TAG, "Ne morem nazaj iz vzpostavljanja povezave.", e2);
                throw new IOException(e2.getMessage());
            }
        }
        return sock;
    }
}
```

Slika 7.1.1: Prikaz povezave z Bluetooth vmesnikom

Sprva se najprej definira prazen univerzalno-unikaten identifikator (ang. “Universally unique identifier”), katerega se potrebuje za serijsko povezavo z vhodom vmesnika Bluetooth. V tej obliki mora biti definirati tako, da je na nivoju aplikacije le-ta unikaten. Naprej se definirata spremenljivki tipa *BluetoothSocket*, ki nam služita kot ustvarjanje novega vtičnika, kamor se nato sproži varovana povezava do vmesnika Bluetooth s specificiranim identifikatorjem. S klicem funkcije *connect* se nato na omenjeno napravo lahko poveže. V catch bloku pa v primeru napake pri povezavi ponovno izvede klic. To omogoča lažje povezovanje, saj se velikokrat zgodi, da Bluetooth naprava še ni povsem

postavljena in bi brez tega privedlo do napake pri povezavi, kjer bi nato morali aplikacijo pognati znova. Prikaz same implementacije kode lahko vidimo na Slika 7.1.1.

7.2 Razred OBDCommand (ObdUkaz)

V tem poglavju se bomo spustili v delovanje razreda, ki služi pretvarjanju ukaza v opravilo. Ta razred je specifično namenjen temu, da lahko objekt tipa oz. razreda *ObdCommand*, kateri je del OBD II API-ja pretvorimo v opravilo za pridobitev vrednosti parametrov vozila [2]. Na povezavi je razred opisan bolj podrobno, vendar je za potrebe razumevanja naše implementacije potrebno samo bistvo. Našo implementacijo smo zato povzeli po že obstoječem razredu zgoraj, ampak nekoliko prilagojeno za delovanje naše aplikacije, kot jo predstavlja Slika 7.2.1. Objekt razreda v osnovi vsebuje dva parametra, in sicer ukaz in nabor znakov kot seznam besednih nizov. Funkcije razreda skrbijo za kreiranje novega objekta (konstruktor), filtriranje šumov med podatki (iskanje po vzorcih) in preverjanje napak (kreiranje specifičnih izjem, katere lovijo vhodno-izhodne napake, napake v procesiranju ukazov itd.) ter formatiranje rezultata – le-ta je lahko predstavljen v miljah na uro oz. kilometrih na uro, za odstotkovne ukaze je na voljo drug razred, o katerem pa malo več kasneje.

V naše namene moramo iz tega razreda v naš namen ustvarjanja ukaza kot nekega parametra za procesiranje pretvoriti v opravilo z dodajanjem parametra o stanju ter unikatnem identifikatorju ukaza. Le-ta nam omogoča, da lahko z gotovostjo vemo, v kakšnem stanju je ukaz, ko ga OBD servis skuša prožiti (o tem nekoliko kasneje).

Vsakemu novemu ukazu ob inicializaciji dodelimo stanje NOV, kar pomeni, da smo ustvarili nov ukaz in ga dodali v vrsto ukazov, katere nato izvaja OBD servis. Pri tem smo ukaz razreda *ObdCommand* uspešno pretvorili v naš razred *ObdUkaz*, ki je nato pripravljen na izvajanje v vrsti ukazov, za katero pa skrbi OBD servis.

```java
package com.example.diplomskaracedroid;

import com.github.pires.obd.commands.ObdCommand;

public class ObdUkaz {
    private long _id;
    private ObdBaseUkaz _ukaz;
    private ObdCommandJobState _stanje;

    public ObdUkaz(ObdBaseUkaz ukaz) {
        _ukaz = ukaz;
        _stanje = ObdCommandJobState.NOV;
    }

    //GET&SET methods

    public long getId() { return _id; }
    public void setId(Long id) { _id = id; }
    public ObdBaseUkaz getKomanda() {return _ukaz; };
    public ObdCommandJobState getStanje() { return _stanje; }
    public void setStanje(ObdCommandJobState stanje) { _stanje = stanje; }


    public enum ObdCommandJobState {
        NOV,
        V_IZVAJANJU,
        ZAGNAN,
        NAPAKA_V_IZVEDBI,
        NAPAKA_V_VRSTI,
        NI_PODPRTO
    }
}
```

Slika 7.2.1: Razred ObdUkaz

7.3 OBD servis

Po uspešnem pretvarjanju OBD ukaza v opravilo se le-ta prenese v OBD servis, ki je implementiran v razredu *ObdServis*. V tem servisu se ustvari povezava z razredom *BluetoothPovezava*, katerega smo omenili že v prejšnjem poglavju, prav tako pa se sproži tudi procesiranje ukazov. Namen tega je, da lahko iz *MainActivity*, ki je glavni razred aplikacije (kliče se ob inicializaciji aplikacije), prožimo ta servis, namesto da podatke obdelujemo direktno tam, kar pomaga pri preglednosti kode. Najprej bomo nekoliko

opisali celoten postopek, nato pa sledi tudi podrobnejša razlaga ob implementaciji. Ob samem začetku delovanja servisa se moramo držati zaporedja nekaterih sistemskih ukazov, ki jih uporablja protokol OBD II za uspešno inicializacijo. Ti ukazi so definirani na način, ki ga prikazuje Slika 7.3.1.

```
        case "AT E0": //echo off
            rezultat = "Echo Off";
            break;
        case "AT L0": //line feed off
            rezultat = "Line Feed-off";
            break;
        case "AT ST 62": //timeout
            rezultat = "Timeout";
            break;
        case "AT SP 0": //protocol
            rezultat = "Select Protocol";
            break;
    }
    return rezultat;
}

public String ukaziKonfiguracija(String kodaUkaza) {
    String rezultat = "";
    switch(kodaUkaza) {
        case "AT Z": //reset obd
        case "AT E0": //echo off
        case "AT L0": //line feed off
        case "AT ST 62": //timeout
        case "AT SP 0": //protocol
            rezultat = _goliPodatki;
            break;
    }
    return rezultat;
}
```

Slika 7.3.1: Primer zagonskih ukazov

Na sliki lahko vidimo primer klicev zagonskih (ang "boot") ukazov. Pri izvajanju le-teh OBD II protokol vrne rezultat kar v obliki niza znakov, zato tukaj ni potrebno posebej razčlenjevati bitne kode. Glavni namen teh razredov je čiščenje morda predpomnjenih podatkov in nastavitev. Na hitro bomo razložili pomen in delovanje posameznega zagonskega ukaza za nekoliko boljše razumevanje namembnosti [7]:

- Reset – ukaz "AT Z" (ukaz poskrbi za ponovno nastavitev naprave in vrne ELM-USB identifikator, na katerega se lahko povežemo),

- Izbira privzetega protokola – ukaz “AT SP 0” (to je pomembno zaradi različnih protokolov, ki jih lahko uporabljajo različni proizvajalci avtomobilov, vendar pri nas ni bil problem s tem, zato smo uporabili privzetega),
- Odstranjevanje izpisov – ukaz “AT E0” (s tem ukazom odstranimo izpis nepotrebnih podatkov iz vmesnika Bluetooth),
- Odstranjevanje nove vrstice – ukaz “AT L0” (s tem ukazom onemogočimo ustvarjanje nove vrstice, saj pri branju podatkov ni tako potrebno, da so zapisani v ločenih vrsticah. V našem primeru je pomembno da imajo čim manj dodatnih znakov, da jih lahko lažje filtriramo),
- Nastavitev poteka časa za povezavo – ukaz “AT ST” ter števila določenih milisekund za potek.

```
protected void konfiguracijaBootUkazov() {
    Log.d(TAG, "Konfiguracija začetne sekvence OBD..");
    dodajJob(new obdUkazJob(new obdBootReset()));
    dodajJob(new obdUkazJob(new obdBootEchoOff()));
    dodajJob(new obdUkazJob(new obdBootLineFeedOff()));
    dodajJob(new obdUkazJob(new obdBootTimeout(62)));

    //dodamo AUTO oz. default OBDII protokol
    dodajJob(new obdUkazJob(new obdBootSelectProtocol()));

    stQue = 0L;
    Log.d(TAG, "Konfiguracija uspešna.");
}
```

Slika 7.3.2: Konfiguriranje zagonskih ukazov

V funkciji *konfiguracijaBootUkazov,* kot jo predstavlja Slika 7.3.2, poskrbimo za nastavitev ob inicializaciji vrste ukazov. Le-to definiramo na začetku OBD servisa, kjer uporabimo Java vgrajeno podatkovno strukturo, imenovano *BlockingQueue*. Ta podatkovna struktura skrbi za to, da lahko vanjo neomejeno pošiljamo ukaze, vrsta pa jih lahko sprejema tako hitro, kot jih lahko dodajamo. V tem primeru ne pride do izjeme *OutOfMemory*, saj je začetna velikost te vrste postavljena na maksimalno vrednost

numerične vrednosti. To vrsto nato pripravimo za prejemanje podatkov s funkcijo *dodajJob*. V to funkcijo prejmemo objekt razreda *ObdUkazJob*, kateremu nato nastavimo unikatni identifikator in ga dodamo v vrsto. V primeru napake ukazu namesto stanja *NOV* dodamo stanje *NAPAKA_V_VRSTI*. Delovanje vrste predstavlja Slika 7.3.3.

```
public void dodajJob(obdUkazJob job) {
    stQue++;
    Log.d(TAG, "Dodajanje joba[" + stQue + "] v vrsto..");

    job.setId(stQue);
    try {
        vrstaJobov.put(job);
        Log.d(TAG, "Job dodan v vrsto.");
    } catch (InterruptedException e) {
        job.setState(obdUkazJob.obdUkazStanje.NAPAKA_V_VRSTI);
        Log.e(TAG, "Ne morem dodati joba v vrsto.");
    }
}
```

Slika 7.3.3: Dodajanje ukaza v vrsto

Na začetku moramo prav tako definirati tudi novo nit, ki bo skrbela za izvajanje vrste ukazov. Takšen pristop veliko pripomore k učinkovitejšemu obdelovanju podatkov, saj nekoliko razbremeni glavno nit. Prikaz uporabe niti v programskem jeziku Java prikazuje Slika 7.3.4 [2].

```
protected BlockingQueue<obdUkazJob> vrstaJobov = new LinkedBlockingQueue<>();
// Run the executeQueue in a different thread to lighten the UI thread
Thread t = new Thread(new Runnable() {
    @Override
    public void run() {
        try {
            izvediVrstoUkazov();
        } catch (IOException | InterruptedException e) {
            t.interrupt();
        }
    }
});
```

Slika 7.3.4: Prikaz izvedbe vrste ukazov

V niti izvajamo spodnjo funkcijo, kot prikazuje Slika 7.3.5, katera pa skrbi za izvajanje ukazov, ki smo jih dodali s klicem funkcije *dodajJob*. Funkcija se izvede nad vrsto objektov razreda obdUkazJob, kjer pobira ukaze po principu FIFO (ang. "First In First Out"). Vsak nov ukaz, pobran iz vrste ukazov, se preveri, če ima ustrezno stanje – v našem primeru mora imeti stanje *NOV*, v nasprotnem primeru se prožijo ustrezne izjeme. Če ukaz ni prazen, pa se lahko simultano posodablja stanje na uporabniškem vmesniku s klicem sistemske funkcije *runOnUIThread* [2].

```java
protected void izvediVrstoUkazov() throws InterruptedException, IOException {
    while(!Thread.currentThread().isInterrupted()) {
        obdUkazJob job = null;
        try {
            job = vrstaJobov.take();
            Log.d(TAG, "Job[" + job.getId() + "] iz vrste..");
            if(job.getState().equals(obdUkazJob.obdUkazStanje.NOV)) {
                job.setState(obdUkazJob.obdUkazStanje.V_IZVAJANJU);
                job.getUkaz().runUkaz(sock.getInputStream(),
sock.getOutputStream());
            }
            else {Log.e(TAG, "Job ni nov, ne bi smel biti v vrsti."); }
        } catch(InterruptedException ex) {
            Thread.currentThread().interrupt();
        } catch (UnsupportedCommandException u) {
            if (job != null) {
                job.setState(obdUkazJob.obdUkazStanje.NI_PODPRTO);
            }
            Log.d(TAG, "Ukaz ni podprt. -> " + u.getMessage());
        } catch (Exception e) {
            if (job != null) {
                job.setState(obdUkazJob.obdUkazStanje.NAPAKA_V_IZVAJANJU);
            }
            Log.e(TAG, "Ne morem zagnati ukaza. -> " + e.getMessage());
        }
        if(job != null) {
            final obdUkazJob job1 = job;
            ((MainActivity) ctx).runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    ((MainActivity) ctx).stateUpdate(job1);
                }
            });
        }
    }
}
```

Slika 7.3.5: Prikaz izvajanja ukazov ter prikazovanje na števcih

Na sliki, katero prikazuje Slika 7.3.6, v razredu *ObdServis* vidimo funkcijo *zacniPovezavoOBD*, ki se nato kliče v funkciji *zazeniServis*, do katere pa pridemo malo kasneje. Ta funkcija pokliče razred, katerega smo opisali prej, in sicer za inicializacijo povezave preko Bluetooh. V primeru, da je povezava uspešna, izvede oz. pokliče funkcijo za inicializacijo zagonskih ukazov [2].

```
public void zacniPovezavoOBD() throws IOException {
    Log.d(TAG, "Starting OBD connection..");
    flag = true;
    try {
        sock = bluetoothPovezava.povezi(dev);

    }catch(Exception ex) {
        Log.e(TAG, "Napaka pri povezovanju z BT. Ustavljam aplikacijo...",
ex);
        ustaviServis();
        throw new IOException();
    }
    konfiguracijaBootUkazov();
}
```

Slika 7.3.6: Zagon branja podatkov v živo

Uporaba glavne funkcije, ki jo prožimo iz glavne aktivnosti aplikacije (ang. “Main activity”) služi kot celosten zagon celega servisa in jo prikazuje Slika 7.3.7. Iz uporabniških nastavitev pridobimo izbrano napravo Bluetooth, na katero se nato skušamo povezati preko javanskih sistemskih funkcij. Če smo v nastavitvah pravilno izbrali napravo, se začne ustvarjati povezava, ki je opisana v funkciji zgoraj [2].

```
public void zazeniServis() throws IOException {
    Log.d(TAG, "Zaganjam servis OBD..");
    final String btNaprava = prefs.getString(Konfiguracija.BT_KLJUC, null);
    Log.d("BT_KLJUC", btNaprava);
    if(btNaprava == null || "".equals(btNaprava)) {
        Toast.makeText(ctx, "Aplikacija deluje v načinu brez Bluetooth
povezave", Toast.LENGTH_LONG).show();
        Log.e(TAG, "Ni izbrane BT naprave.");
        //ustaviServis();
    }else {
        final BluetoothAdapter btAdapter =
BluetoothAdapter.getDefaultAdapter();
        dev = btAdapter.getRemoteDevice(btNaprava);
        Log.d(TAG, "Ustavljam iskanje BT.");
        btAdapter.cancelDiscovery();
        try {
            zacniPovezavoOBD();
        } catch (Exception e) {
            Toast.makeText(getBaseContext(),
                    "Napaka pri vzpostavljanju povezave BT. -> "
                            + e.getMessage(),Toast.LENGTH_LONG);

            // in case of failure, stop this service.
            ustaviServis();
            throw new IOException();
        }
        Toast.makeText(getBaseContext(),"Servis OBD v teku.",
Toast.LENGTH_LONG);
    }
}
```

Slika 7.3.7: Zagon servisa za zajem podatkov

Ko se servis iz nepričakovanih razlogov ustavi, bodisi pa se lahko zaključi preko ukaza iz menija aplikacije, se kasneje pokliče spodnja funkcija, ki izprazni celotno vrsto ukazov in zapre vtičnik za povezavo Bluetooth naprave [2]. Delovanje in upravljanje s samo nepričakovano ustavitvijo servisa prikazuje Slika 7.3.8.

```
public void ustaviServis() {
    Log.d(TAG, "Ustavljam servis OBD");
    vrstaJobov.removeAll(vrstaJobov);
    flag = false;
    if(sock != null) {
        try {
            sock.close();
        }catch(IOException ex) {
            Log.e(TAG, ex.getMessage());
        }
    }
};
```

Slika 7.3.8: Ustavitev servisa

7.4 Pospeškometer

Za potrebe merjenja pospeškov smo se v primeru naše mobilne aplikacije odločili uporabiti vgrajene funkcije OS Android [1]. Večina pametnih telefonov, na katerih teče operacijski sistem Android, že ima vgrajene različne senzorje, le-ti pa omogočajo ogromno dodatnih možnosti hkrati za razvijalce ter tudi uporabnike. Novejši telefoni poleg vgrajenih funkcij, kot so GPS in Bluetooth, omogočajo merjenje pospeškov v različne smeri po vseh treh smereh v prostoru (smer x, y in z), nekateri telefoni pa imajo vgrajen tudi že žiroskop, pri ostalih pa lahko dosežemo podobno delovanje s pravilno izrabo teh senzorjev. V našem primeru smo se lotili tega na način, kot ga prikazuje Slika 7.4.1:

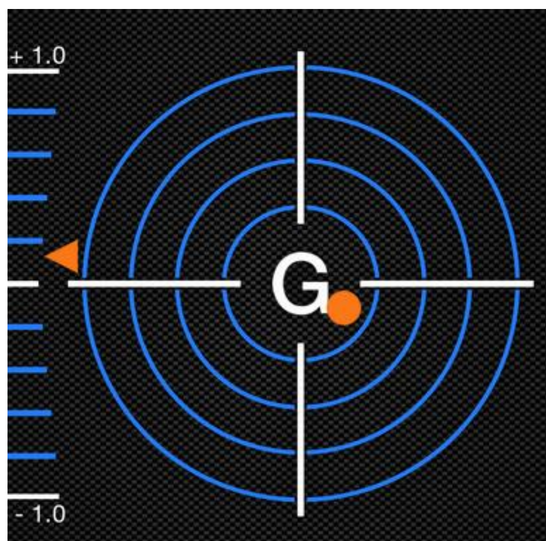
```
sensorManager = (SensorManager)
context.getSystemService(Context.SENSOR_SERVICE);
sensor = sensorManager.getDefaultSensor(Sensor.TYPE_LINEAR_ACCELERATION);
```

Slika 7.4.1: Primer uporabe senzorjev

V programu Android Studio smo preko vgrajene funkcije dostopali do objekta, imenovanega *SensorManager*, ki med tekom delovanja deluje nad senzorji na mobilni napravi. Temu objektu moramo tudi določiti, za katero vrsto senzorja oz. na kakšen način želimo prejemati te podatke, torej v našem primeru gre za linearni pospešek. Seveda to ni edina izbira, operacijski sistem Android omogoča ogromno različnih možnosti teh senzorjev, med katere pa spadajo tudi:

- *TYPE_ACCELEROMETER*: Ta način prav tako meri pospeške, vendar zraven upošteva tudi gravitacijo. V našem primeru to ni najbolj zaželjeno, saj ne vidimo velikih odstopanj pri spremembi pospeška.
- *TYPE_ACCELEROMETER_UNCALIBRATED*: Podobno kot prejšnji način se tukaj merijo pospeški, vendar pa le-ta tudi dopušča možnost za nekaj pristranskosti pri merjenju.
- *TYPE_GRAVITY*: Ta način senzorja meri spremembe v gravitacijski sili v primeru, ko telefon premaknemo v eno od treh smeri v prostoru.
- *TYPE_GYROSCOPE*: Način giroskopa omogoča merjenje rotacijskih pospeškov.
- *TYPE_GYROSCOPE_UNCALIBRATED*: Deluje enako kot prejšnji način, vendar pa dopušča nekoliko pristranskosti pri merjenju.
- *TYPE_LINEAR_ACCELERATION*: Za nas najbolj primeren, saj omogoča merjenje linearnih pospeškov brez gravitacije. To nam omogoča, da lahko na progi zaznamo očitnejše spremembe v pospeških.
- *TYPE_STEP_COUNTER*: Način senzorja, ki omogoča štetje korakov.

Pri implementaciji pospeškometra smo se morali tudi vprašati, kako bi to najbolje prikazali. Prikaz mora biti dovolj jasen, da bo uporabnik aplikacije takoj lahko razbral, kaj mu pospeški sporočajo in se glede na nastalo situacijo tudi lahko prilagodil. V tem primeru smo se obrnili na že obstoječe dirkalne aplikacije, katere imajo implementiran takšen senzor. Večina le-teh uporablja t.i. “merilec g-sile” (ang. “G-force meter”), ki ga prikazuje Slika 7.4.2.



Slika 7.4.2: Skica pospeškometra [15]

V tem primeru vidimo, da gre za klasičen prikaz različnih krogov, ki sporočajo intenziteto pospeška. Sam pospešek prikazuje oranžna pika, ki se prilagaja vrednostim. Če to postavimo v perspektivo, izgleda to v praksi nekako takole:

Pri vožnji s konstantno hitrostjo, kjer ne zavijamo v nobeno smer in ne pospešujemo, bo vsota vseh pospeškov, ki delujejo na vozilo, enaka nič. V tem primeru bi se morala ta oranžna pika zadrževati v sami sredini, vendar pri takšni vožnji zaradi zunanjih dejavnikov (cesta, naklon) lahko nekoliko odstopa. V primeru, da ostro zavijemo v levo smer, se mora zaradi povečane sile, ki deluje na desno stran vozila, ta oranžna pika premakniti v desno smer. Če v tem primeru vmes zaviramo, se bo ta pika premaknila v zgornji desni del grafa, v primeru pospeševanja pa v spodnjo desno četrtino grafa. V primeru zavijanja v drugo smer je situacija enaka z izjemo tega, da se pika premakne na levo stran grafa.

V naši aplikaciji smo se torej tega lotili tako, da smo z uporabo senzorja za linearni pospešek brez upoštevanja gravitacije najprej prebrali podatke iz pametne naprave. V primeru starejših verzij operacijskega sistema, kjer še možnost senzorja za linearni pospešek brez gravitacije ni obstajala, obstajala pa je samo navadna možnost za

pospeškometer, so za meritev realnega pospeška morali uporabiti sledeč filter vrednosti [1].

```
override fun onSensorChanged(event: SensorEvent) {
    // In this example, alpha is calculated as t / (t + dT),
    // where t is the low-pass filter's time-constant and
    // dT is the event delivery rate.

    val alpha: Float = 0.8f

    // Isolate the force of gravity with the low-pass filter.
    gravity[0] = alpha * gravity[0] + (1 - alpha) * event.values[0]
    gravity[1] = alpha * gravity[1] + (1 - alpha) * event.values[1]
    gravity[2] = alpha * gravity[2] + (1 - alpha) * event.values[2]

    // Remove the gravity contribution with the high-pass filter.
    linear_acceleration[0] = event.values[0] - gravity[0]
    linear_acceleration[1] = event.values[1] - gravity[1]
    linear_acceleration[2] = event.values[2] - gravity[2]
}
```

Slika 7.4.3: Pristop filtriranja vrednosti pospeškometra

Slika 7.4.3 prikazuje primer filtra, ki ga imenujejo tudi filter nizkega prehoda (ang. "low pass filter"). Osnovna ideja tega filtra je, da se prednastavi konstanta alfa, katera predstavlja izračun časovne konstante filtra in stopnje osveževanja dogodka za pridobitev vrednosti. V vsaki vrednosti dogodka – vrednosti od 0 do 2 predstavljajo tri smeri v prostoru, torej koordinate x, y in z, v katerih dogodek dostavlja podatke iz pametne naprave, izoliramo vrednost gravitacije, nato pa nad podatki še preko filtra visokega prehoda odstranimo vrednost gravitacije od dejanske zaznane vrednosti. Na ta način torej dobimo čisto vrednost pospeška, kjer zraven ni zabeležena gravitacija.

Za potrebe naše implementacije smo morali torej uporabiti merilec linearnega pospeška brez gravitacije. V primeru strukturiranja tega razreda smo morali implementirati razredne funkcije, katere delujejo posebej s pospeškometrom. Najprej smo morali

definirati poslušalec teh dogodkov, katere zaznava senzor, v njem pa se nato implementirajo določene metode, odvisno od tega, kako želimo podatke manipulirati. V našem primeru je to dogodek, imenovan *OnSensorChanged*, kateri kot vhod funkcije prejme vrednosti senzorja v objektu, le-ta pa vsebuje podatke o vseh treh smereh v prostoru. Primer delovanja ponazarja spodnja Slika 7.4.4.

```
public Pospeskometer(Context context) {
    vrednosti = new ArrayDeque<>();
    sensorManager = (SensorManager)
context.getSystemService(Context.SENSOR_SERVICE);
    sensor = sensorManager.getDefaultSensor(Sensor.TYPE_LINEAR_ACCELERATION);
    sensorEventListener = new SensorEventListener() {
        @Override
        public void onSensorChanged(SensorEvent sensorEvent) {
            if (zacetek == 0) {
                zacetek = System.nanoTime();
            }
            timestamp = System.nanoTime();
            float[] mean = new float[3];
            float hz = (st++ / ((timestamp - zacetek) / 1000000000.0f));
            int filterWindow = (int) Math.ceil(hz * konstanta);
            vrednosti.addLast(Arrays.copyOf(sensorEvent.values,
sensorEvent.values.length));
            while (vrednosti.size() > filterWindow) {
                vrednosti.removeFirst();
            }
            if(!vrednosti.isEmpty()) {
                izpis = mediana(vrednosti);
            } else {
                izpis = new float[sensorEvent.values.length];
                System.arraycopy(sensorEvent.values, 0, izpis, 0,
sensorEvent.values.length);
            }
            if (listener != null) {
                listener.prenosPodatkov(izpis[0], izpis[1], izpis[2]);
            }
        }
    };
}
```

Slika 7.4.4: Prebiranje vrednosti iz senzorja

V tem dogodku nato podatke obdelamo na način, kjer preko uporabe časovnih funkcij najprej izračunamo frekvenco dogodkov, nad katerim potem izračunamo tudi filter. Namen le-tega je prilagajanje vrednosti, s katerim lahko preprečimo počasno delovanje. V primeru, da pride do več vrednosti, kot je frekvenca osveževanja, odvečne stare vrednosti porežemo, v nasprotnem primeru pa jih pošljemo naprej v funkcijo, ki jih nato iz tega razreda prenese naprej v glavno aktivnost aplikacije. V primeru, da vrednosti trenutno niso vnešene, se naredi kopija tabele podatkov, notri pa se napolnijo uteženi podatki. Za računanje vrednosti, katere ne bodo preveč odstopale, imamo narejeno posebno funkcijo, ki izračuna mediano, torej sredinsko vrednost (Slika 7.4.5). To funkcijo smo ustvaril z namenom, da preprečimo prevelika odstopanja med podatki, ki se lahko zgodijo v primeru zunanjih dejavnikov (npr. luknja na cesti).

```
private float[] mediana(ArrayDeque<float[]> data) {
    float[] mean = new float[data.getFirst().length];

    for (float[] axis : data) {
        for (int i = 0; i < axis.length; i++) {
            mean[i] += axis[i];
        }
    }

    for (int i = 0; i < mean.length; i++) {
        mean[i] /= data.size();
    }

    return mean;
}
```

Slika 7.4.5: Filtriranje podatkov po mediani

Za potrebe prenosa podatkov, katerega delovanje ponazarja Slika 7.4.6, smo v ta namen v glavno aktivnost implementirali tudi posebno strukturo, imenovano poslušalec (ang. “listener”), preko katere lahko podatke pošiljamo v glavno aktivnost. Ta struktura služi kot vmesnik, kateri specificira funkcijo, ki se izvaja v ločeni niti, neodvisno od glavne niti.

V našem primeru nam to služi za nemoteno pridobivanje podatkov iz razreda *Pospeskometer* v glavno aktivnost tekom izvajanja.

```
acc = new Pospeskometer( context: this);

registerReceiver(uiUpdated, new IntentFilter( action: "LOCATION_UPDATED"));
handler = new Handler();
runnable = new Runnable() {
    @Override
    public void run() {
        acc.setListener(new Pospeskometer.Listener() {
            @Override
            public void prenosPodatkov(float x, float y, float z) {
                if (x > 3) {
                    napotki.setText("Zavoj v desno.");
                    if (x > 6) {
                        napotki.setText("Možnost zdrsa!");
                    }
                } else if (x < -3) {
                    napotki.setText("Zavoj v levo.");
                    if (x < -6) {
                        napotki.setText("Možnost zdrsa!");
                    }
                } else if (y > 3) {
                    napotki.setText("Pospeševanje.");
                } else if (y < -3) {
                    napotki.setText("Zaviranje.");
                } else {
                    napotki.setText("Mirovanje.");
                }

                x_pos.setText(Float.toString( f: Math.round(x * 100.0f) / 100.0f));
                y_pos.setText(Float.toString( f: Math.round(y * 100.0f) / 100.0f));
                z_pos.setText(Float.toString( f: Math.round(z * 100.0f) / 100.0f));
                pospesek[0] = Math.round(x * 100.0f) / 100.0f;
                pospesek[1] = Math.round(y * 100.0f) / 100.0f;
                pospesek[2] = Math.round(z * 100.0f) / 100.0f;
                gauge.updatePoint( x: Math.round(x * 100.0f) / 100.0f,  y: Math.round(y * 100.0f) / 100.0f);
            }
        });
    }
};
```

Slika 7.4.6: Osveževanje pospeškometra na grafičnem vmesniku

Preko funkcije, imenovane *prenosPodatkov* v razredu *Pospeskometer*, lahko nato v glavni aktivnosti pridobivamo podatke. Za nas je najbolj pomembno to, da lahko podatke prejemamo ter pošiljamo naprej v razred *GaugeAcceleration* [5], ta pa nam nato podatke grafično izriše ter prikaže na merilcu pospeškov. O tem razredu bo več govora v poglavju GRAFIČNI VMESNIK APLIKACIJE vmesnik aplikacije.

7.5 Servis za lokacijo

Za možnost postavljanja sektorjev, tudi merjenje in prikazovanje prevožene poti, smo morali v naši aplikaciji uporabiti tudi lokacijo. Lokacija je na programskem sistemu Android razvijalcem dostopna kot vgrajeni servis, do katerega lahko dostopamo. Pri implementaciji smo si pomagali s prosto dostopnim virom preko spletne strani YouTube [10]. Na začetku implementiranja lokacijskega servisa smo morali v Android Studio-u v *Gradle* datoteko dodati reference za lokacijo (Slika 7.5.1), kateri omogočata dostop do vgrajenih funkcij, potem pa smo v datoteko *Manifest* dodali tudi dovoljenje aplikaciji, da lahko le-ta dostopa do lokacije naprave, ter da lahko izvaja servis za neprekinjen dostop do lokacije.

```
implementation 'com.google.android.gms:play-services-maps:17.0.0'
implementation 'com.google.android.gms:play-services-location:17.0.0'

<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.FOREGROUND_SERVICE" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
```

Slika 7.5.1: Dovoljenja ter vključevanje paketov

Pri implementiranju samega razreda, prikazanega na Slika 7.5.2, za lokacijo pa smo morali najprej razred razširiti z uporabo *Service*. To nam omogoča, da potem ta razred definiramo kot servis in da se bo tudi obnašal kot servis. Na začetku se osredotočimo na objekt tipa *LocationCallback*, ki nam kot struktura omogoča pridobivanje podatkov položaja GPS pametne naprave. Podobno kot pri razredu *Pospeskometer* deluje tudi ta objekt. Vanj se ob implementaciji funkcij tega objekta pridobijo vhodni podatki o lokaciji, ki jih potem shranimo ter dodamo v objekt tipa Intent, kateri nam omogoča, da lahko te podatke pošljemo na neko drugo aktivnost.

```
public class LocationService extends Service {
    public double lat = 0.0, longit = 0.0, alt = 0.0;
    private LocationCallback locationCallback = new LocationCallback(){
        @Override
        public void onLocationResult(LocationResult locationResult) {
            super.onLocationResult(locationResult);
            if(locationResult != null && locationResult.getLocations() !=
null) {
                lat = locationResult.getLastLocation().getLatitude();
                longit = locationResult.getLastLocation().getLongitude();
                alt = locationResult.getLastLocation().getAltitude();

                //Pošljemo podatke nazaj na MainActivity
                Intent i = new Intent("LOCATION_UPDATED");
                i.putExtra("latitude",String.valueOf(lat));
                i.putExtra("longitude",String.valueOf(longit));
                i.putExtra("altitude",String.valueOf(alt));
                sendBroadcast(i);
            }
        }
    };
```

Slika 7.5.2: Pošiljanje podatkov na servis za lokacijo

Ob zagonu našega servisa, kot to predstavlja Slika 7.5.3, za lokacijo moramo podati nek črkovni niz znakov, ki služi za identifikator kanala, po katerem bomo komunicirali, hkrati pa tudi pognati nek obveščevalec, ki nam bo omogočal prikaz stanja samega servisa – torej začetek, izvajanje ter ugašanje [10]. V Android Java programskem jeziku to storimo preko *NotificationManager*-ja, ki koristi vgrajeno obveščevalsko storitev. To nam pride prav pri performančnih zahtevah, saj se zna ogromno poznati razlika, kadar tečejo vsi servisi in kadar ne. V našem primeru koristimo mobilno omrežje za shranjevanje podatkov, Bluetooth za komunikacijo z vmesnikom in pridobivanje lokacije za merjenje sektorjev, kar pa zna vplivati na akumulator telefona. Naprej v implementaciji definiramo tudi nov obveščevalni kanal preko *NotificationChannel* in mu podamo visoko pomembnost. To pomeni, da se bo ob spremembi statusa vedno pojavilo sporočilo na

telefonu. Potrebujemo pa tudi lokacijsko zahtevo, kateri moramo podati interval zajema podatkov, kar storimo preko klica funkcije *setInterval*, podamo pa tudi *setFastestInterval*. Razlika med dvema funkcijama sprva ni očitna, ampak pri prvi podamo nek fiksen interval v milisekundah zajema lokacije, pri drugi pa lahko spremembo lokacije dobimo tudi hitreje, ko in če je seveda ta na voljo. S klicem *getFusedLocationProvider* pa se povežemo na ponudnika storitve in servis poženemo.

```
private void startLocationService() {
    String channelId = "location_notification_channel";
    NotificationManager notificationManager =
(NotificationManager)getSystemService(Context.NOTIFICATION_SERVICE);
    Intent resultIntent = new Intent();
    PendingIntent pendingIntent = PendingIntent.getActivity(
            getApplicationContext(),
            0,
          resultIntent,
            PendingIntent.FLAG_UPDATE_CURRENT
    );
    NotificationCompat.Builder builder = new NotificationCompat.Builder(
            getApplicationContext(),
            channelId
    );
    if(Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        if (notificationManager != null &&
notificationManager.getNotificationChannel(channelId) == null)  {
            NotificationChannel notificationChannel = new
NotificationChannel(
                    channelId,
                    "Service lokacija",
                    NotificationManager.IMPORTANCE_HIGH
            );
            notificationChannel.setDescription(("Ta kanal uporablja service
lokacija"));

notificationManager.createNotificationChannel(notificationChannel);
        }
    }
    LocationRequest locationRequest = new LocationRequest();
    locationRequest.setInterval(0); //4000
    locationRequest.setFastestInterval(0); //2000
    locationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
LocationServices.getFusedLocationProviderClient(this).requestLocationUpdates(
locationRequest,locationCallback, Looper.getMainLooper());
    startForeground(175, builder.build());
}
```

Slika 7.5.3: Komunikacija in implementacija servisa za lokacijo

Seveda moramo servis za lokacijo pravilno poklicati tudi v naši glavni aktivnosti aplikacije, kjer jo moramo pravilno inicializirati in ob koncu uporabe tudi pravilno zapreti. Za to skrbi naslednji del kode, ki je prikazan na Slika 7.5.4.

```
private void zazeniServisZaLokacijo() {
    if(!aliServiceZaLokacijoTece()) {
        Intent intent = new Intent(getApplicationContext(),
LocationService.class);
        intent.setAction("pozeni_servis");
        startService(intent);
        Toast.makeText(this, "Lokacija vklopljena",
Toast.LENGTH_LONG).show();
    }
}

private void ustaviServisZaLokacijo() {
    if(aliServiceZaLokacijoTece()) {
        Intent intent = new Intent(getApplicationContext(),
LocationService.class);
        intent.setAction("ustavi_servis");
        startService(intent);
        Toast.makeText(this, "Lokacija izklopljena",
Toast.LENGTH_LONG).show();

    }
}
```

Slika 7.5.4: Zagon in ustavitev servisa v glavni aktivnosti aplikacije

Ti funkciji se kličeta ob začetku ter koncu snemanja vožnje, delujeta pa glede na preprost parameter, kateremu v računalništvu rečemo zastavica (ang. "flag"). Tekom izvajanja se ob inicializaciji servisa za lokacijo pokliče funkcija *aliServisZaLokacijoTece* in preveri, ali je na nivoju aktivnosti vklopljen servis za lokacijo (Slika 7.5.5). Za to poskrbi *ActivityManager*, ki je namenjen upravljanju z aktivnostjo in tekom izvajanja beleži delovanje – na kratko, sledi izvajanju aplikacje. V tem *ActivityManager*-ju torej preverimo, ali obstaja servis z imenom razreda kot je naš, funkcija pa nato kot rezultat vrne vrednost tipa Boolean, katera lahko vsebuje samo vrednosti resnično (ang. "true") oziroma napačno (ang. "false").

```
private boolean aliServiceZaLokacijoTece() {
    ActivityManager activityManager =
(ActivityManager)getSystemService(Context.ACTIVITY_SERVICE);
    if(activityManager != null) {
        for(ActivityManager.RunningServiceInfo service :
activityManager.getRunningServices(Integer.MAX_VALUE)) {

if(LocationService.class.getName().equals(service.service.getClassName())) {
                if(service.foreground) {
                    return true;
                }
            }
        }
        return false;
    }
    return false;
}
```

Slika 7.5.5: Preverjanje aktivnega servisa za lokacijo

7.6 Merjenje sektorjev preko lokacije GPS

Pri snovanju naše aplikacije se je pojavilo vprašanje, kako bi določene segmente na progi kar najbolje izmerili, torej kako bi lahko z gotovostjo prišli do dejanskih vrednosti, le-te pa bi nam definirale uspešnost vožnje. Navdih za to smo črpali iz različnih disciplin avto-moto športa, kjer uporabljajo meritve sektorjev, ter končno idejo izoblikovali s pomočjo članka, najdenega preko Google Scholar [17]. V primeru formule 1 se pri kvalifikacijah proga razdeli v več sektorjev, na katerih se potem meri čas. Meritve potekajo tako, da se čas za sektor začne šteti pri vstopu v sektor in neha pri izhodu iz njega, po odvoženih krogih posameznika pa se nato sestavijo štartna mesta glede na dosežene čase.

V našem primeru smo se tega lotili tako, da smo v aplikaciji definirali posebno nastavitev, ki nam omogoča nastavitev sektorjev. Stran oz. aktivnost, kot se temu reče v jeziku Java Android, je tipa zemljevid, kar omogoča enostavno postavitev točk za merjenje, le-te pa potem postanejo točke, ki na dirkališču oz. predvideni progi označujejo začetek in konec sektorjev (Slika 7.6.1). Pri merjenju krogov se to prikazuje na sledeč način:

- Pred začetkom vožnje definiramo svoje sektorje na predvideni progi.
- Pri začetku vožnje se čas pri prevozu prvega sektorja začne šteti in se konča pri začetku naslednjega.
- Uporabniški vmesnik beleži vsak uspešno prevožen sektor ter odvožene kroge.

Tukaj sicer velja omeniti, da pri računanju lokacije včasih pride do odstopanj, saj lokacijski modul v mobilnih napravah včasih podleže zunanjim dejavnikom, kot so na primer izolirano sprednje steklo na avtomobilih, kjer je lahko zelo slab signal, geografska pozicija vožnje (izpad signala GPS ter mobilnih podatkov) ter netočna postavitev sektorja.

Implementacijsko smo se tega lotili tako, da smo uporabili funkcijo, katero omogoča programski jezik Java, in sicer imenovano *Thread* oziroma *Runnable*, katerega izvajanje smo nastavili na 100 milisekund za čim bolj točno delovanje in spremljanje lokacije. To predstavlja ločeno nit, ki nato tekom delovanja izvaja oz. ponavlja segment kode na določen čas. Sektorje, katere smo nastavili prej na zemljevidu, smo shranili v tako imenovane preference aplikacije, ki nam omogočajo dostop do le-teh širom aplikacije, nato zvrstimo in dodamo v seznam sektorjev za merjenje.

```
String time;
if(myPrefs.contains("sektorji"))
{
    float[] currentDistance = new float[1];
    LatLng sektor;
        if(stevecSektorjev == sektorjiCopy.size() + 1) {
            stevecSektorjev = 0;
            sektor = sektorjiCopy.get(stevecSektorjev);
            resetKroga = true;
        }
        else if(stevecSektorjev == sektorjiCopy.size()) {
            sektor = sektorjiCopy.get(stevecSektorjev - 1);
            lastSector = true;
        }
        else {
            sektor = sektorjiCopy.get(stevecSektorjev);
        }
```

Slika 7.6.1: Pokrivanje možnih scenarijev pri vožnji skozi sektorje

V funkciji *run*, ki skrbi za pogon niti, nato izvršimo našo logiko. Najprej preverimo, ali v preferencah aplikacije obstajajo določeni sektorji. Potem definiramo spremenljivko

currentDistance, v katero bomo shranjevali trenutno razdaljo naše lokacije do najbližjega sektorja v vsaki iteraciji niti (Slika 7.6.3). Več o tem bo predstavljeno malo kasneje. Preverjati moramo tudi, ali je trenutno sektor kateri od ključnih, zato imamo tukaj zapisano spremljanje treh različnih scenarijev:

- Prvi scenarij preverja, ali je trenutni sektor že začetek novega kroga; v tem primeru ponastavimo števec sektorjev ter postavimo zastavico *resetKroga*, kar upoštevamo v nadaljevanju in poskrbimo za to, da merimo razdaljo do prvega sektorja v krogu.
- Drugi scenarij preverja, če je trenutni sektor zadnji del kroga med zadnjim sektorjem ter začetkom novega kroga. V tem primeru za trenutni sektor vzamemo zadnji sektor v seznamu in poskrbimo za nastavitev nove zastavice *lastSector,* katera nam bo pomagala v nadaljevanju.
- Scenarij v primeru, da trenutni sektor ne ustreza zgornjim kriterijem pa je samo nastavljanje trenutnega sektorja na naslednjega v seznamu.

V nadaljevanju bomo za potrebo shranjevanja podatkov pri meritvah sektorja implementirali tudi razred *MachineLearningData*, katerega predstavlja Slika 7.6.2. Ta preprost razred bo vseboval vse podatke, ki jih lahko zberemo pri vožnji.

```java
import com.google.android.gms.maps.model.

public class MachineLearningData {
    public String StKroga;
    public String StSektorja;
    public String Cas;
    public long Cas_pospesevanja;
    public long Cas_zaviranja;
    public double TPS_zac;
    public double GPS_lat_zac;
    public double GPS_long_zac;
    public int obrati_zac;
    public int obrati_kon;
    public double obremenitev_zac;
    public double obremenitev_kon;
    public String Spd_zac;
    public String Spd_kon;
    public double TPS_kon;
    public double GPS_lat_kon;
    public double GPS_long_kon;
    public double PospesekX;
    public double PospesekY;
    public double PospesekZ;
    public String Dimenzija;
    public String TipGume;
    public double PritiskSpredaj;
    public double PritiskZadaj;
}
```

Slika 7.6.2: Lastnosti posameznega zabeleženega sektorja

```java
MachineLearningData entry = null;
if(entry == null) {
    entry = new MachineLearningData();

    String tps = (String) tps_counter.getText();
    String load = (String) load_counter.getText();
    entry.TPS_zac = Double.parseDouble(tps.substring(0, tps.length()-1));
    entry.GPS_lat_zac = sektor.latitude;
    entry.GPS_long_zac = sektor.longitude;
    entry.Spd_zac = String.valueOf(value_speed);
    entry.obrati_zac = Integer.parseInt((String) counter_ui.getText());
    entry.obremenitev_zac = Double.parseDouble(load.substring(0, load.length()-1));
    entry.TipGume = tip_gume;
    entry.Dimenzija = dim_gum;
    entry.PritiskSpredaj = pritisk_spredaj;
    entry.PritiskZadaj = pritisk_zadaj;
}

if(lastSector) {
    Location.distanceBetween(sirina, dolzina, sektorjiCopy.get(0).latitude, sektorjiCopy.get(0).longitude, currentDistance);
    lastSector = false;
}
else {
    Location.distanceBetween(sirina, dolzina, sektor.latitude, sektor.longitude, currentDistance);
}
```

Slika 7.6.3: Primer preverjanja sektorja za nov krog

Ko uspešno določimo pravilen scenarij meritve, usmerimo našo pozornost k shranjevanju podatkov pri merjenju sektorja. V izseku kode zgoraj najprej preverimo, ali že obstaja objekt našega razreda, v nasprotnem primeru vanj shranimo trenutne podatke o vrednosti pozicije plina, položaja GPS ter trenutni hitrosti vozila. Delovanje predstavlja Slika 7.6.4.

Za potrebe merjenja naše razdalje do sektorja smo uporabili vgrajeno funkcijo ponudnika Google, in sicer *distanceBetween*. Ta funkcija nam omogoča, da lahko vnesemo koordinate dveh točk, katere nato shranimo v že prej omenjeno spremenljivko *currentDistance.* Le-ta nam v posamezni iteraciji potem pove, koliko smo oddaljeni do določenega sektorja. Ponavadi računamo razdaljo med našo trenutno pozicijo ter trenutnim sektorjem, za potrebe meritve zadnjega sektorja pa razdaljo namesto do trenutnega sektorja merimo do prvega sektorja, kjer pa nam prav pride zastavica *lastSector.*

```java
if (currentDistance[0] < 10)
{
    if(isChronometerRunning) {
        time = (String) chronometer.getText();
        chronometer.stop();
        chronometer.setBase(SystemClock.elapsedRealtime());
        chronometer.start();

        if(resetKroga == true) { //
            if(stevecSektorjev != 0) {
                String tps = (String) tps_counter.getText();
                String load = (String) load_counter.getText();
                entry.StKroga = String.valueOf(lap_count);
                entry.StSektorja = String.valueOf(stevecSektorjev);
                entry.TPS_kon = Double.parseDouble(tps.substring(0, tps.length()-1));
                entry.GPS_lat_kon = sektor.latitude;
                entry.GPS_long_kon = sektor.longitude;
                entry.Spd_kon = String.valueOf(value_speed);
                entry.Cas = time;
                entry.PospesekX = Double.parseDouble((String) x_pos.getText());
                entry.PospesekY = Double.parseDouble((String) y_pos.getText());
                entry.PospesekZ = Double.parseDouble((String) z_pos.getText());
                entry.obrati_kon = Integer.parseInt((String) counter_ui.getText());
                entry.obremenitev_kon = Double.parseDouble(load.substring(0, load.length()-1));
                entry.TipGume = tip_gume;
                entry.Dimenzija = dim_gum;
                entry.PritiskSpredaj = pritisk_spredaj;
                entry.PritiskZadaj = pritisk_zadaj;
                data.add(entry);
                lap.setText("Krog: " + lap_count);
                ListElementsArrayList.add("Krog: " + lap_count + "| Sektor " + stevecSektorjev + ": " + time); //
                resetKroga = false;
                stevecSektorjev++;
            }
            else {
                Toast.makeText(context,  text: "Začetek merjenja sektorja...", Toast.LENGTH_LONG).show();
                stevecSektorjev++;
                lap_count += 1;
                lap.setText("Krog: " + lap_count);
                st_sektorja.setText("Sektor: " + stevecSektorjev);
            }
        }
```

Slika 7.6.4: Shranjevanje podatkov ob prevoženem sektorju

V zadnjem delu naše niti sledi preverjanje, kdaj lahko registriramo naš sektor kot prevožen oz. dosežen, kot to prikazuje implementacija na Slika 7.6.5. Ponavadi bi to pomenilo, da bi sektor bil prevožen takrat, ko je spremenljivka *currentDistance* enaka 0, saj to pomeni, da se koordinate sektorja ter naše trenutne pozicije povsem ujemajo. Zaradi v določenih primerih nenatančne lokacije moramo pri preverjanju razdalje pustiti neko toleranco, saj se v nasprotnem primeru zgodi, da sektor zgrešimo. Moduli GPS v današnjih pametnih telefonih so sicer dovolj natančni, ampak zaradi samega delovanja aplikacije ter upoštevanja dejavnikov delovanja, kot so opisani zgoraj, raje pustimo neko toleranco. Zdelo se nam je, da je 5 metrov dovolj dobra toleranca, saj bi lahko tako še vedno pravilno predvideli prevožen sektor in ne preveč okrnili delovanja.

V nadaljevanju programske kode pri izmerjeni razdalji, ki je manjša od tolerančne, pridemo v segment, kjer nato preverjamo, če trenutno že teče štoparica (Slika 7.6.4). V primeru da ni, vemo, da gre za vožnjo do prvega sektorja in pri dosegu le-tega to zaženemo. Ta korak je nujno potreben, saj ne želimo, da bi nam tik ob zagonu aplikacije začelo šteti čas, saj v večini primerov voznik takrat še ni pripravljen in tudi test ne bi bil izveden učinkovito. V nasprotnem primeru pa potem preverimo, ali je postavljena zastavica *resetKroga* in potem ustrezno zabeležimo podatke o sektorju na grafični vmesnik in v podatke, katere bomo kasneje obdelali preko strojnega učenja. Nit *runnableSektorji* se neha izvajati, ko prekinemo prenos podatkov v živo iz menija v aplikaciji.

```
else {
    if(stevecSektorjev != 0) {
        String tps = (String) tps_counter.getText();
        String load = (String) load_counter.getText();
        entry.StKroga = String.valueOf(lap_count);
        entry.StSektorja = String.valueOf(stevecSektorjev);
        entry.TPS_kon = Double.parseDouble(tps.substring(0, tps.length()-1));
        entry.GPS_lat_kon = sektor.latitude;
        entry.GPS_long_kon = sektor.longitude;
        entry.Spd_kon = String.valueOf(value_speed);
        entry.Cas = time;
        entry.PospesekX = Double.parseDouble((String) x_pos.getText());
        entry.PospesekY = Double.parseDouble((String) y_pos.getText());
        entry.PospesekZ = Double.parseDouble((String) z_pos.getText());
        entry.obrati_kon = Integer.parseInt((String) counter_ui.getText());
        entry.obremenitev_kon = Double.parseDouble(load.substring(0, load.length()-1));
        entry.TipGume = tip_gume;
        entry.Dimenzija = dim_gum;
        entry.PritiskSpredaj = pritisk_spredaj;
        entry.PritiskZadaj = pritisk_zadaj;

        data.add(entry);

        Toast.makeText(context, text: "Prišli ste skozi sektor!", Toast.LENGTH_LONG).show();
        st_sektorja.setText("Sektor: " + stevecSektorjev);
        ListElementsArrayList.add("Krog: " + lap_count + "| Sektor " + stevecSektorjev + ": " + time); //
        stevecSektorjev++;
    }
    else if(stevecSektorjev == sektorjiCopy.size()) {

    }
    else {
        Toast.makeText(context, text: "Začetek merjenja sektorja...", Toast.LENGTH_LONG).show();
        stevecSektorjev++;

        st_sektorja.setText("Sektor: " + stevecSektorjev);
    }
```

Slika 7.6.5: Preverjanje sektorja, kjer naslednji ni nov krog

7.7 Zapis podatkov v datoteko

Po odvoženih krogih je potrebno podatke zapisati v datoteko, da jih bomo lahko kasneje strojno obdelali. Pri zapisu smo upoštevali sledeče: Za podatke strojnega učenja uporabimo datoteko CSV (ang. “Comma delimited values”) oziroma vrednosti, ločene z vejico, saj lahko te podatke nato brez večjih težav obdelamo preko programa za strojno učenje Weka (Slika 7.7.1). V objekt *StringBuilder* programskega jezika najprej zapišemo glavo CSV dokumenta, ki bo kasneje služila za prepoznavo naših atributov pri strojnem učenju, potem pa z iteracijo skozi zbrane podatke le-te zapišemo ločene s podpičjem. Spodnji izsek kode sicer prikazuje vrednosti v primeru priključenega OBD II vmesnika Bluetooth, v primeru vožnje brez pa podatki o loputi, hitrosti, obratih ter hitrosti ne obstajajo in jih zato izvzamemo iz zapisa, saj predstavljajo slabšo kvaliteto strojnega učenja z ničnimi vrednostmi.

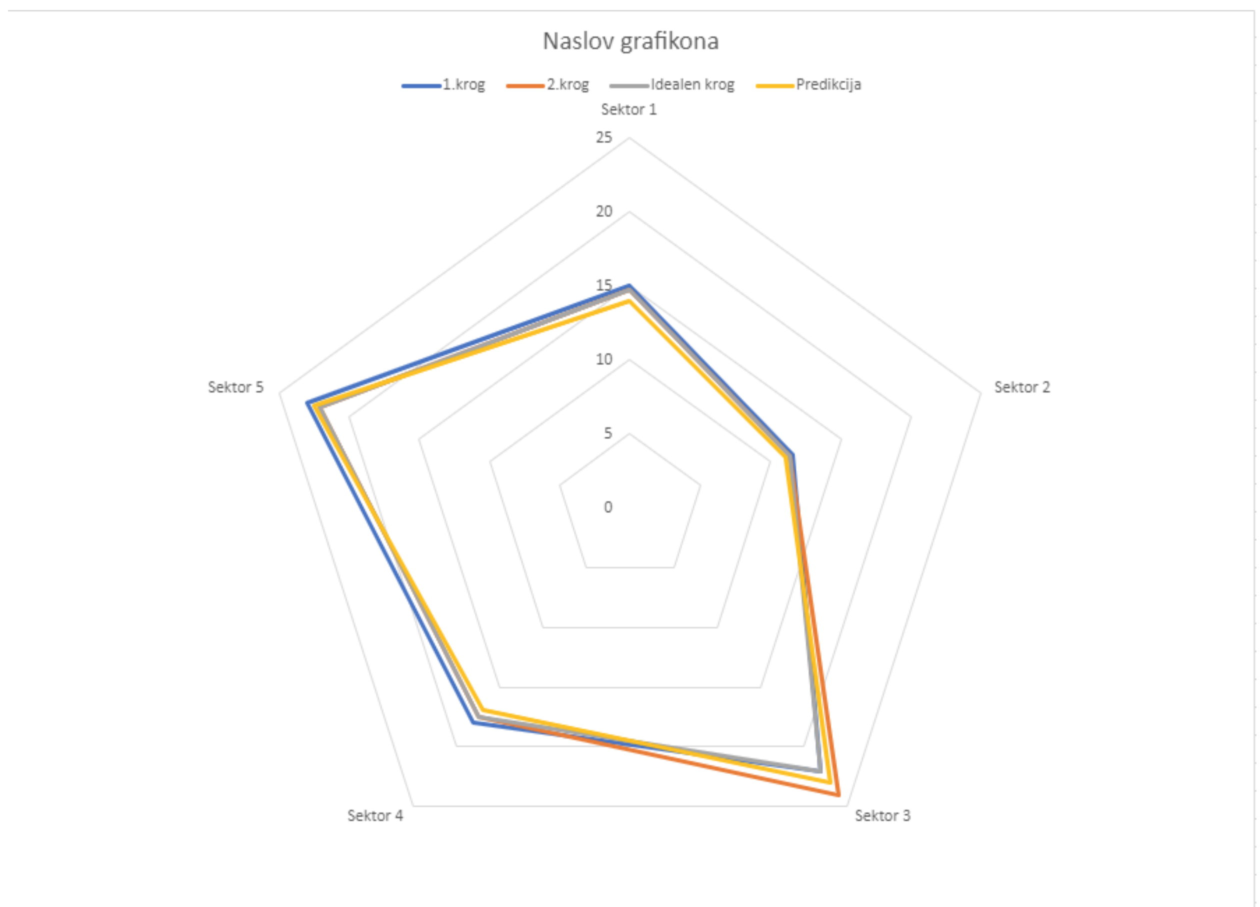
```
public void zapisiPodatkeVDatoteko() throws IOException {
    StringBuilder sb = new StringBuilder();
    if(pozeniOffline == false) {
        sb.append("ST_KROGA;ST_SEKTORJA;CAS;TPS_ZAC;TPS_KON;OBRATI_ZAC;OBRATI
        for(MachineLearningData line : data) {
            String[] t = line.Cas.split( regex: ":"); //will break the string u
            int minutes = Integer.parseInt(t[0]); //first element
            int seconds = Integer.parseInt(t[1]);
            int duration = 60 * minutes + seconds;

            sb.append("\n"+String.valueOf(line.StKroga) + ";"
                    + String.valueOf(line.StSektorja) + ";"
                    + String.valueOf(duration) + ";"
                    +  String.valueOf(line.TPS_zac) +";"
                    + String.valueOf(line.TPS_kon) +";"
                    + String.valueOf(line.obrati_zac) +";"
                    + String.valueOf(line.obrati_kon) +";"
                    + String.valueOf(line.obremenitev_zac) +";"
                    + String.valueOf(line.obremenitev_kon) +";"
                    + String.valueOf(line.GPS_lat_zac) +";"
                    + String.valueOf(line.GPS_long_zac) +";"
                    + String.valueOf(line.GPS_lat_kon) + ";"
                    + String.valueOf(line.GPS_long_kon) + ";"
                    + String.valueOf(line.Spd_zac) + ";"
                    + String.valueOf(line.Spd_kon) + ";"
                    + String.valueOf(line.PospesekX)+ ";"
                    + String.valueOf(line.PospesekY)+ ";"
                    + String.valueOf(line.PospesekZ)+ ";"
                    + String.valueOf(line.TipGume)+ ";"
                    + String.valueOf(line.Dimenzija)+ ";"
                    + String.valueOf(line.PritiskSpredaj)+ ";"
                    + String.valueOf(line.PritiskZadaj)
            );
        }
    }
```

Slika 7.7.1: Zapis podatkov v datoteko

7.8 Strojno učenje in prikaz podatkov

Po uspešno izvedenem shranjevanju podatkov se je porajalo vprašanje, kako bi lahko te podatke najlažje prikazali ter dokazali uspešnost strojnega učenja. V našem primeru bi to najlažje storili tako, da bi lahko videli zbrane čase različnih krogov glede na posamezen sektor, hkrati pa tudi celotne kroge, kot tudi razlike med njimi. Polarni grafikon (Slika 7.8.1) nam omogoča spremljanje več različnih spreminjajočih se vrednosti hkrati in tudi daje celosten pregled nad dobljenimi rezultati.



Slika 7.8.1: Vzorčni primer željenega prikaza

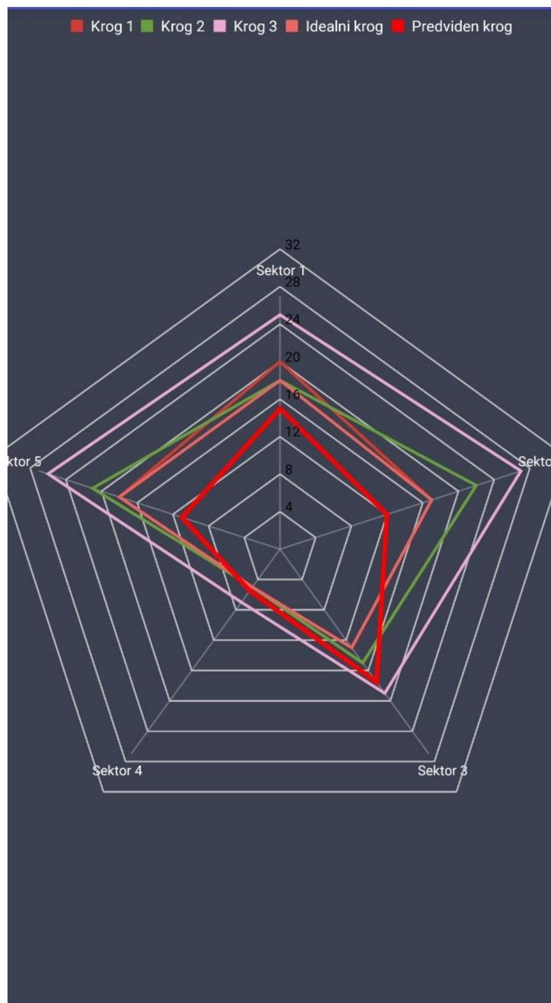
V primeru na sliki zgoraj opazimo, da grafikon predstavlja podatke za štiri kroge; torej dva odvožena kroga, idealen krog – poberemo najboljše čase vsakega sektorja ter predviden krog preko strojnega učenja. Sam graf je postavljen v obliki petkotnika, kar ponazarja v tem primeru pet postavljenih sektorjev, znotraj le-tega pa prikazujemo

merilno lestvico s sekundami za posamezni sektor. Vsak lom posamezne črte glede na lestvico prikazuje dosežen čas na posameznem sektorju; s tem lahko spremljamo uspešnost vožnje posameznega kroga, kot tudi uspešnost predikcije glede na podane podatke vožnje. V primeru naše implementacije enakega grafikona v aplikaciji smo se tega lotili na identičen način: Iz posamezne datoteke je bilo potrebno izluščiti sektorje in pripadajoče podatke, nad njimi izvesti strojno učenje in potem prikazati podatke čim bolj podobno grafu zgoraj.

Za potrebe prikaza samega grafa smo uporabili že obstoječo knjižnico z grafikoni, imenovano Charting [12], saj bi bila implementacija samega grafa še dodatno delo, kar pa ni primarna tema tega diplomskega dela. Knjižnica Charting že vsebuje polarni grafikon s konfigurabilnimi parametri, kar nam je v veliko pomoč, saj ne moremo tega postaviti fiksno zaradi spreminjanja podatkov. Uporabili smo tudi preprosto implementacijo izbiranja datotek [13], saj programski jezik Java v okolju Android ne vsebuje vgrajene funkcionalnosti za izbiro datotek.

Pri implemetaciji grafikona smo se oprli na primer implementacije polarnega grafikona, ki se nahaja na zgornjem viru (Slika 7.8.2). V našem primeru bomo morali to obstoječo funkcionalnost nekoliko prilagoditi, da bo bolj fleksibilna glede na vhodne podatke, saj so ti lahko zbrani preko OBD II vmesnika ali brez, lahko imajo različno število sektorjev in tudi različne razpone podatkov. V ta namen smo si pripravili nekaj funkcij, katere nam bodo pomagale ugotoviti najdaljši čas sektorja, idealni krog ter maksimalno število sektorjev in krogov. Zaradi trivialnosti funkcij bomo omenili samo, da gre v tem primeru za razčlenjevanje vhodnih podatkov, nad katerimi potem iščemo vrednosti in jih v skladu s funkcijo uporabimo pri mapiranju grafa.

V nadaljevanju implementacije najprej naložimo podatke iz datoteke preko izbire datotek. Tukaj lahko odpiramo samo CSV datoteke, katere vsebujejo podatke, v nasprotnem primeru se bo izpisalo samo opozorilo, da graf nima podatkov. Pri uspešni izbiri se nam nato generira graf, ki vsebuje podatke o vožnji in je identičen zgornjemu.



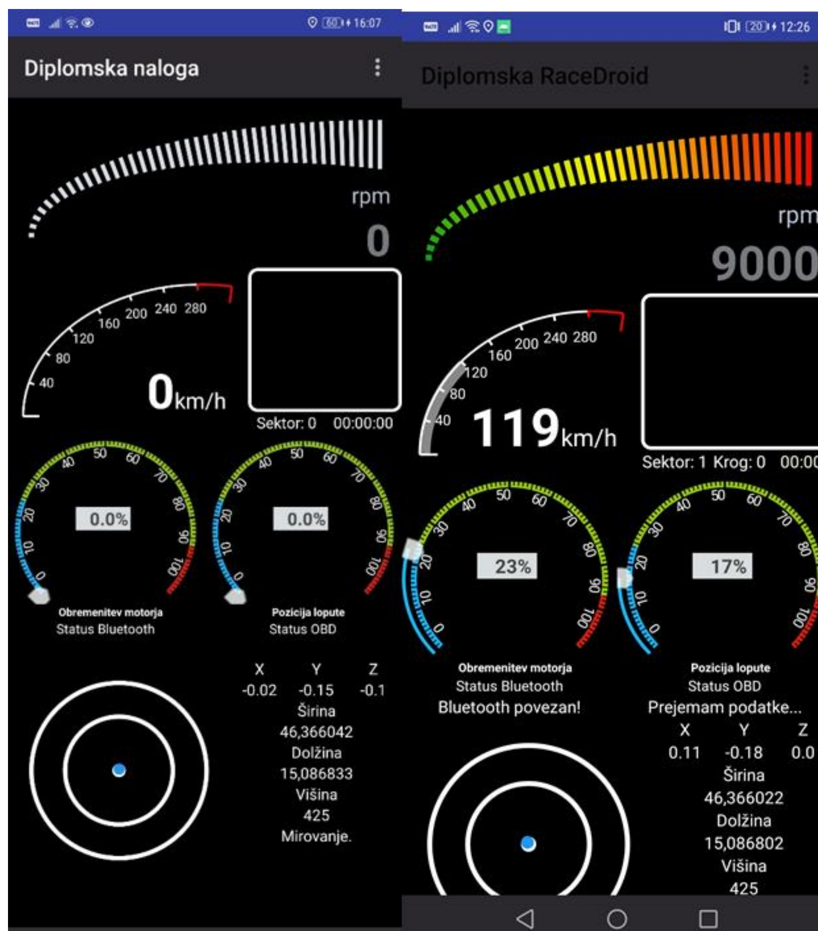
Slika 7.8.2: Prikaz grafa vožnje

8 GRAFIČNI VMESNIK APLIKACIJE

V tem poglavju bomo malo podrobneje pogledali grafični vmesnik ter funkcionalnosti, ki jih ponuja aplikacija. Ogledali si bomo možne nastavitve aplikacije, pojasnili glavni zaslon aplikacije z vsemi števci ter si pogledali nastavitev sektorjev. Konfiguracija teh elementov je nujna za optimalno delovanje aplikacije, saj je uspeh predikcije odvisen od pravilnega vnosa podatkov vozila [5] [6] [12].

8.1 Glavni zaslon aplikacije

Ob zagonu aplikacije se nam najprej pokaže glavni zaslon aplikacije (Slika 8.1.1), ki vsebuje vse potrebne števce za prikaz podatkov vozila. Leva slika prikazuje aplikacijo ob zagonu, kjer še process branja podatkov ni zagnan, na desni sliki pa lahko vidimo polno delovanje aplikacije. Na vrhu aplikacije imamo števec obratov, kjer različne črtice prikazujejo približen odstotek obratov motorja, spodaj pa se prikazuje dejanska zajeta številčna vrednost v danem trenutku. Pod števcem za obrate se nahaja števec za hitrost, kjer smo poskušali intuitivno prikazati hitrost vozila z analognim in digitalnim prikazom.



Slika 8.1.1: Osnovni pogled aplikacije (slika levo) in pogled ob polni funkcionalnosti (slika desno)

Verjetno najbolj pomemben pa je spremljevalnik odvoženih krogov, kateri pa se nahaja zraven števca za hitrost. Notri lahko beležimo odvožene kroge ter spremljamo izmerjene čase krogov. Pod njim se nahaja tudi števec krogov in trenutni sektor, zato da vozniku ni potrebno vedeti, v katerem sektorju oz. krogu se trenutno nahaja. Merjenje časa se vsakič znova ponastavi ob prehodu skozi sektor, pridobljena vrednost pred ponastavitvijo pa se zapiše na spremljevalnik krogov zraven pripadajočega kroga in sektorja.

Naslednja števca, ki ju bomo opisali, pa sta števca za beleženje absolutne obremenitve vozila ter pozicijo lopute za plin. Oba vsebujeta majhni analogni prikazovalnik vrednosti, kot tudi odstotke, ki prikazujejo točnejšo obremenitev, razpon vrednosti tukaj pa je med

0 in 100 odstotki. Morda se tukaj pojavi vprašanje, zakaj spremljamo te vrednosti? V motošportih je velikokrat to zelo pomemben podatek, saj potencialnega voznika vedno zanima, s kolikšno zmožnostjo deluje njegovo vozilo, ter kje se nahaja meja vozila. Seveda velik del te meje predstavljajo že samo obrati motorja na minuto, vendar šele pri prikazu absolutne vrednosti vidimo, kolikšen je seštevek vseh obremenitev, ki vplivajo na vozilo v danem trenutku. Pri merjenju odstotka pozicije lopute za plin pa je situacija zelo podobna; večji odstotek pomeni, da je loputa bolj odprta. Če nekoliko pojasnimo, celotni cikel izgorevanja motorja je pogojen s pravilno mešanico zraka in goriva, katera mora biti ustrezna, da lahko iz vozila iztisnemo kar največ. Ravno zaradi tega je dobro vedeti, kako se le-ta vrednost spreminja pod obremenitvijo in je lahko izkušenemu vozniku zelo uporaben podatek.

Na spodnjem delu zaslona lahko vidimo števec obremenitev oziroma sil, ki delujejo na vozilo med pospeški, zato takšen merilec imenujemo tudi pospeškometer. Delovanje pospeškometra lahko razložimo na naslednji način: Ob čistem mirovanju bo vsota vseh sil, ki delujejo na neko telo, enaka nič, saj telo ne pospešuje oz. zavira, enako pa bo tudi, če se telo premika s konstantno hitrostjo, brez pospeševanja oziroma zaviranja. Ob zavijanju začne na telo vplivati lateralni pospešek in kazalnik se bo premaknil v nasprotno smer zavoja. V primeru nenadnega pospeška se bo kazalnik premaknil v spodnji del kroga, v primeru nenadnega zaviranja pa na sprednji del kroga. Ko to uporabimo v praksi pri vožnji in pridemo npr. do situacije, kjer je potrebno za vstop v zavoj hkrati zavirati in zavijati, pa v tem primeru pride ko kombinacije vektorjev pospeška. Vrednost pospeškometra ponavadi ob dirkalni vožnji ne preseže več kot 1G. Zraven pospeškometra pa prikazujemo tudi vrednosti koordinat ter trenutni položaj GPS in nadmorsko višino.

Na spodnjih slikah je prikazana razlika, kjer imamo na desni sliki povezavo preko vmesnika Bluetooth, na levi sliki pa le-te ni. Aplikacija sicer lahko deluje v obeh načinih, vendar pri uporabi aplikacije brez vmesnika Bluetooth števci ne delujejo. Vožnjo lahko sicer še vedno izmerimo z nekaj razlikami oziroma kljub manjkajočim podatkom, kar pa je opisano v poglavju Strojno učenje in prikaz podatkov.

8.2 Konfiguracija podatkov in povezave

Za uspešno uporabo aplikacije je potrebno nastaviti pravilne parametre vozila, saj nam le-ti omogočajo kar najbolj točno zajetje vseh podatkov ob vožnji. V ta namen smo naredili posebno aktivnost, ki se odpre ob kliku na tri pikice v imenski vrstici aplikacije in izbere s klikom na Nastavitve, in je prikazana na Slika 8.2.1.

Pri aktivnosti Konfiguracija smo poskušali čim bolj zajeti vse nastavitve, katere bi lahko poenotili na eno mesto v aplikaciji. Kot najpomembnejša nastavitev aplikacije se čisto na vrhu nahaja preklopni gumb za vklop oziroma izklop vmesnika Bluetooth, ki nam omogoča povezavo z OBD II vmesnikom. Pri vklopu vmesnika se nam najprej pojavi potrditveno okno, kjer moramo najprej aplikaciji dati pravice za upravljanje z Bluetooth na pametnem telefonu. Po odobrenih pravicah se na napravi prižge Bluetooth in potem lahko izberemo, iz katere naprave želimo brati podatke. Pod možnostjo Naprave BT se nam prikaže seznam seznanjenih naprav, s katerimi je pametni telefon seznanjen. Tam lahko preprosto samo izberemo napravo iz seznama in ob zagonu branja podatkov v živo aplikacija poskuša iz naprave prebrati podatke.

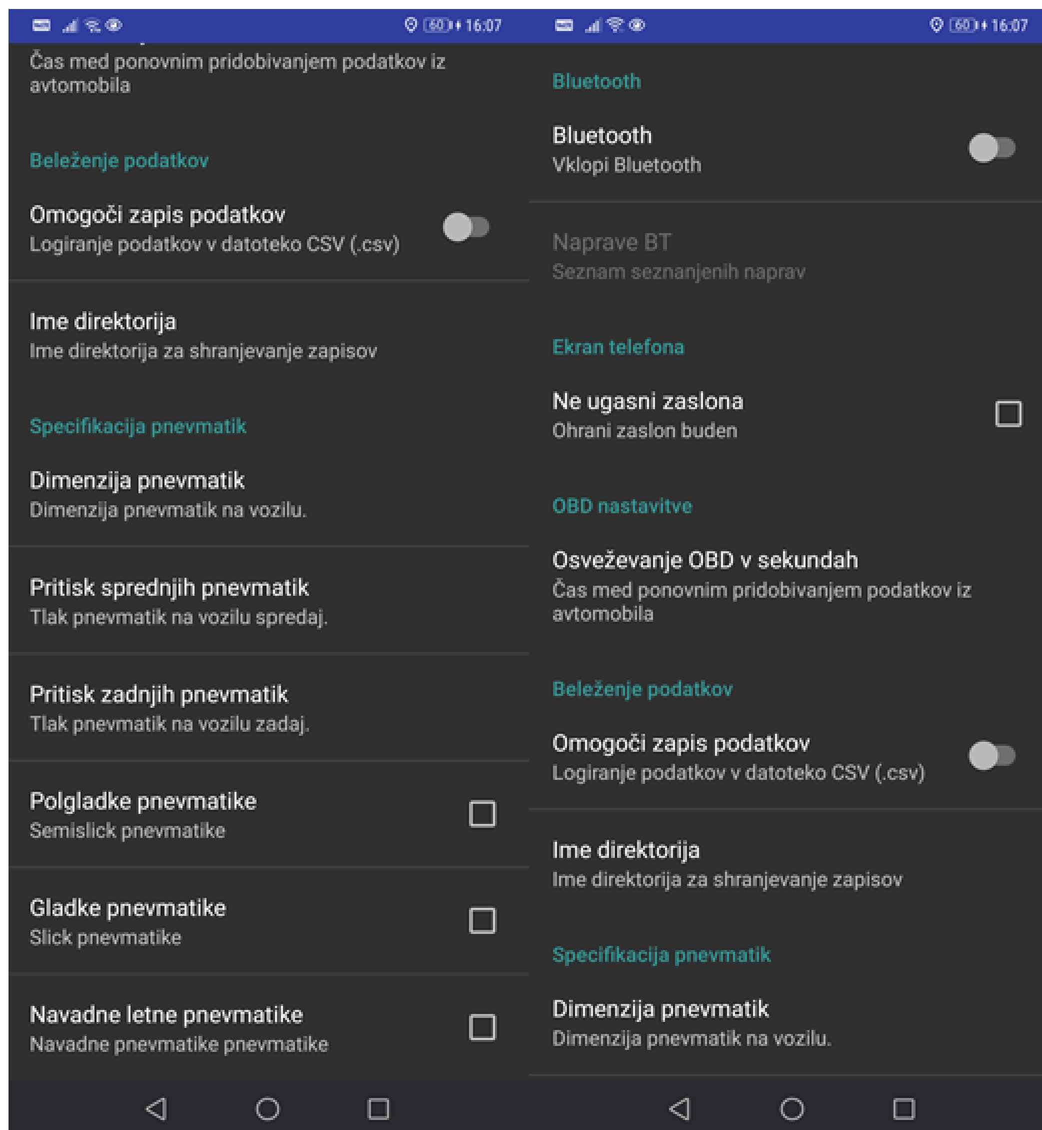
Konfiguracija vsebuje naprej tudi še druge dodatne nastavitve, med njimi tudi budnost zaslona, ki pri vklopu poskrbi, da se zaslon ob neuporabi ne zatemni samodejno, kar bi povzročilo nevšečnosti med vožnjo. Ta nastavitev je seveda izbirna, saj bi vedno prižgan zaslon preveč porabljal baterijo.

V poglavju OBD nastavitve lahko nastavimo čas med zajemom podatkov iz avtomobila. Seveda želimo, da je ta čas čim manjši, saj lahko le tako dobimo kar najbolj točne in realne podatke iz samega vozila. V primeru OBD II vmesnika je lahko to pri določenih modelih problem, saj nekateri slabši ne uspejo brati in pošiljati teh podatkov dovolj hitro ter učinkovito.

Poglavje Beleženje podatkov je namenjeno konfiguraciji zapisa v napravo, ker vsi pametni telefoni žal še vedno nimajo dovolj notranjega pomnilnika ter morajo podatke shranjevati na spominsko kartico. V našem primeru to lahko privede do napake pri privzetem direktoriju aplikacije, zato smo odločitev, kam se bodo podatki zapisovali,

prepustili vsakemu posameznemu uporabniku. Pod temi nastavitvami je moč nastaviti pravice za zapis CSV datoteke v telefon ter v kateri imenik.

Zadnje poglavje aktivnosti Konfiguracija pa je namenjeno podatkom vozila. Tukaj se predvsem usmerimo na uporabo pnevmatik. Zelo dolgo že velja, da je od zmožnosti vozila na progi ogromno odvisno od pnevmatik ter pritiska v njih. Ker poznamo več vrst pnevmatik, smo v aplikaciji naredili možnost izbire le-teh. Izbiramo lahko med gladkimi (ang. "slick"), polgladkimi (ang. "semi-slick") in navadnimi letnimi pnevmatikami. Seveda tukaj obstaja še veliko drugih vrst, vendar se nam je za potrebe testiranja in uporabo aplikacije zdelo dovolj, da izpostavimo samo te, saj se najpogosteje uporabljajo. Za izbrane pnevmatike tudi za čim bolj točne podatke podamo dimenzije ter pritisk v sprednjem in zadnjem paru pnevmatik vozila. Sam pritisk pnevmatik prav tako igra veliko vlogo, saj lahko s tem na nek način reguliramo trdoto pnevmatik [2].



Slika 8.2.1: Konfiguracija aplikacije – nastavitev pnevmatik in pritiska (slika levo) ter Bluetooth in pripadajoče nastavitve (slika desno)

8.3 Postavitev sektorjev

Pri izbiri možnosti Postavitev sektorjev iz glavnega menija aplikacije se nam prikaže zemljevid s trenutno lokacijo in je prikazan na Slika 8.3.1. Če smo prej že odvozili kakšen krog, lahko izberemo možnost Izriši pot, ki nam bo na mapi prikazala prevoženo pot, drugače pa lahko s pritiskom na ekran na željeno mesto postavimo sektor. Pri izhodu iz te aktivnosti nas aplikacija vpraša, ali želimo shraniti te sektorje. S pritiskom na potrditev se bodo ti sektorji upoštevali ob vsakem merjenju vožnje, katera gre čez te geografske koordinate, katere vsebuje sektor.

Slika 8.3.1: Postavitev sektorjev

9 TESTIRANJE APLIKACIJE

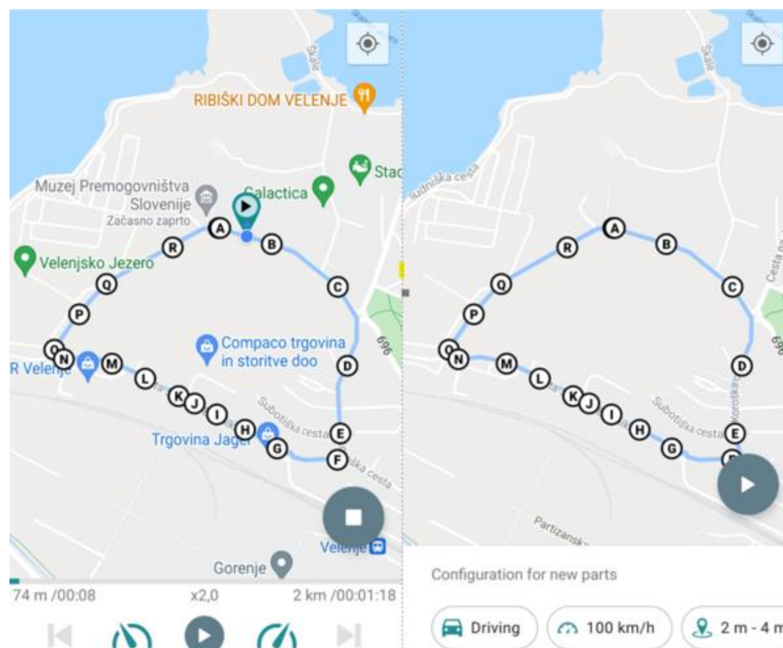
V poglavju Testiranje aplikacije bomo podrobneje predstavili postavljena testiranja ter tudi prikazali rezultate. Naš glavni namen je dodobra in celovito preučiti delovanje aplikacije in preverjanje dobljenih rezultatov. V ta namen bomo za vsak način testiranja podrobneje predstavili pristop in razložili dobljene rezultate.

9.1 Testiranje aplikacije izven avtomobila

Zaradi globalne pandemije, katera je zajela svet v začetku lanskega leta ter tudi vplivala na delovanje sveta še v tekočem letu, se je bilo v vmesnem času potrebno za testiranje nekoliko prilagoditi. Dirkališča, kot tudi mnogo drugih stvari so bila zaprta, zato testiranje na takšen način žal ni bilo mogoče. Seveda bi lahko aplikacijo preizkusili tudi na cesti, vendar zaradi spoštovanja cestno-prometnih predpisov pri tem nismo pretiravali, največji problem pa je bil najti dovolj veliko ter varno mesto, kjer bi to lahko preizkusili. Zaradi naštetih stvari smo zato poskusili nekaj novega. Na spletu smo poiskali možne načine, s katerimi bi bilo mogoče pametni telefon preslepiti ter na nek način simulirati vožnje, tudi ko nismo fizično v avtomobilu. V ta namen smo naleteli na Android aplikacijo, imenovano Lockito, katera bo opisana v naslednjem razdelku. Spodaj bodo opisane še tudi druge metode testiranja, kot so obisk dirkališča ter testiranje drugih voznikov.

9.2 Aplikacija Lockito

Kot že omenjeno zgoraj, smo se pri testiranju poslužili uporabe aplikacije Lockito [21], katera nam omogoča, da nad pametnim telefonom v načinu za razvijalce omogočimo tako imenovano "prevaro" (ang. "spoofing"), ter le-tega prepričamo v poljubno manipulacijo lokacije, kar lahko vidimo na sliki Slika 9.2.1. Na levi sliki lahko vidimo zagnano simulacijo, kjer lahko prilagajamo hitrost in po potrebi tudi ustavimo aplikacijo, na desni sliki pa lahko vidimo aplikacijo pred zagonom. V aplikaciji je mogoče nastavit povsem poljubno progo, katero postavimo z označbami na dotik. V našem primeru smo si izbrali dovolj obsežen del sklenjene ceste, katerega bi lahko smatrali kot približek dirkališča. Nanj smo nato čim bolj točno postavili označbe in si oblikovali progo, nad katero bomo potem lahko izvajali našo simulacijo. Pri tem smo pazili, da se za čim boljšo natančnost in nemoteno delovanje začetna in končna točka čim manj razlikujeta. Ko smo progo uspešno postavili, moramo v aplikaciji še nastaviti sektorje po načinu, opisanem zgoraj in potem smo pripravljeni na zagon zajema podatkov v živo. To v aplikaciji storimo s pritiskom na gumb Začni prenos v živo, potem pa hitro preklopimo na aplikacijo Lockito, ter pritisnemo velik gumb za predvajanje. Tekom delovanja aplikacije je mogoče tudi prilagajati simulacijo hitrosti ter prevrteti naprej na hitrejše izvajanje animacije. Velja tudi omeniti, da aplikacija podpira povsem uporabniško konfigurabilno natančnost zajema lokacije, kjer smo ga v našem primeru nastavili na kar najbolj točno, da bi lahko dobili čim boljše rezultate. Po uspešno izvedenih krogih lahko simulacijo vožnje enostavno ustavimo s pritiskom na gumb ustavi. Pri tem ustavimo tudi zajem podatkov v živo na naši aplikaciji ter preverimo rezultate na izrisanem grafu.



Slika 9.2.1: Prilagajanje hitrosti in ustavitev simulacije (slika levo) in zagon simulacije z nastavitvami natančnosti ter načina (slika desno)

9.3 Predstavitev rezultatov simulacije

Po uspešno izvedenem testiranju s simulacijo lokacije bomo sedaj predstavili dobljene rezultate ter pojasnili graf in ostale podatke. Velja omeniti, da ta simulacija žal deluje samo v načinu brez povezave z Bluetooth vmesnikom OBD II, zato so podatki nekoliko okrnjeni, ampak je nad njimi še vedno mogoče izvesti strojno učenje ter analizirati vožnjo. V spodnji tabeli (Tabela 1) lahko vidimo zbrane podatke, ki jih je generirala aplikacija. Imena stolpcev bomo zaradi lažje preglednosti označili s številkami od ena do štirinajst. Stolpci si sledijo po naslednjem zaporedju:

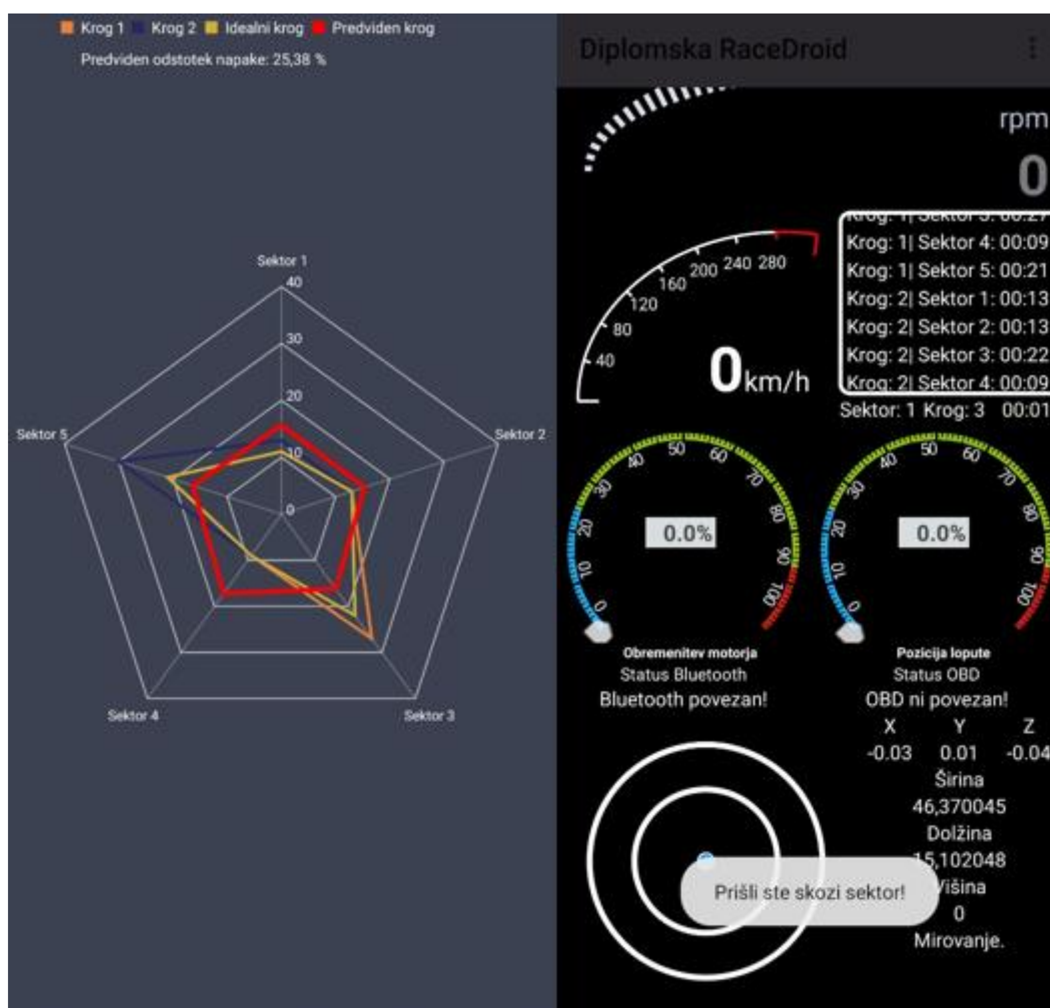
1. Število trenutnega kroga
2. Število trenutnega sektorja
3. Čas sektorja, od začetka do konca
4. GPS zemljepisna dolžina ob vstopu v sektor
5. GPS zemljepisna širina ob vstopu v sektor
6. GPS zemljepisna dolžina ob izstopu iz sektorja

7. GPS zemljepisna širina ob izstopu iz sektorja
8. Pospešek v X osi
9. Pospešek v Y osi
10. Pospešek v Z osi
11. Trenutni tip gume na vozilu (možnost navadne, polgladke ali gladke)
12. Dimenzija gume na vozilu
13. Pritisk gum spredaj
14. Pritisk gum zadaj

Tabela 1: Rezultat simulacije vožnje

1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	1	11	46.36733175225204	15.10277833789587	46.36733175225204	15.10277833789587	0.0	0.0	-0.08	Standard	185/55 R15	2.2	2.2
1	2	13	46.36563777127325	15.100877992808817	46.36563777127325	15.100877992808817	0.01	0.0	-0.09	Standard	185/55 R15	2.2	2.2
1	3	27	46.36786709669322	15.09294468909502	46.36786709669322	15.09294468909502	0.0	0.01	-0.09	Standard	185/55 R15	2.2	2.2
1	4	9	46.369545045944	15.094364918768406	46.369545045944	15.094364918768406	0.0	0.0	-0.07	Standard	185/55 R15	2.2	2.2
1	5	21	46.369545045944	15.094364918768406	46.369545045944	15.094364918768406	0.0	0.0	-0.06	Standard	185/55 R15	2.2	2.2
2	1	13	46.36733175225204	15.10277833789587	46.36733175225204	15.10277833789587	0.0	0.0	-0.09	Standard	185/55 R15	2.2	2.2
2	2	13	46.36563777127325	15.100877992808817	46.36563777127325	15.100877992808817	0.0	-0.01	-0.09	Standard	185/55 R15	2.2	2.2
2	3	22	46.36786709669322	15.09294468909502	46.36786709669322	15.09294468909502	0.02	0.02	-0.13	Standard	185/55 R15	2.2	2.2
2	4	9	46.369545045944	15.094364918768406	46.369545045944	15.094364918768406	0.0	0.0	-0.08	Standard	185/55 R15	2.2	2.2

Podatki v tabeli so enaki zapisu, ki ga ustvari snemanje vožnje brez vklopljenega vmesnika Bluetooth, zato tukaj ni dejanskih podatkov iz vozila. Točnejše delovanje aplikacije nam prikazuje Slika 9.3.1, kjer lahko na sliki levo opazujemo generiran graf, na sliki desno pa delovanje aplikacije v načinu snemanja vožnje. Še vedno pa smo lahko uspešno zajeli podatke o krogih, sektorjih, lokaciji ter pospeških, na koncu pa se nahajajo še podatki o konfiguraciji pnevmatik na vozilu, katere smo nastavili pred zajemom podatkov. Sami goli podatki nam žal ne povedo veliko, ampak po prikazu na grafu, generiranem po prevoženih krogih, nam bo to vizualno bolje predstavljivo.



Slika 9.3.1: Rezultati strojnega učenja na grafu (slika levo) in izmerjeni časi (slika desno)

Na desni sliki lahko vidimo, kako izgleda aplikacija tekom delovanja ter tudi, kako se beležijo sektorji. Števci, kot je že bilo omenjeno, žal ne delujejo, saj smo snemali brez vmesnika Bluetooth. Leva slika pa predstavlja podatke, vizualizirane na grafu. Oranžna ter temno modra barva prikazujeta odvožena kroga, medtem ko rumena barva predstavlja optimalno pot, torej najboljše čase posameznih sektorjev. Za nas je najbolj pomembna rdeča črta, saj ta predstavlja predvideno pot, ki smo jo pridobili iz strojnega učenja nad zbranimi podatki. Kot lahko vidimo, se je predikcija osredotočila na najkrajše čase ter na podlagi zbranih podatkov in klasifikacije po doseženem času podala oceno, da je mogoče našo vožnjo še precej izboljšati. Tukaj sicer velja omeniti, da je zaradi manjkajočih parametrov iz vmesnika Bluetooth predikcija nekoliko nerealna, vendar je glede na podane parametre, ne glede na vse uspešno predvidela možno izboljšanje. Pri meritvi uspešnosti predikcije smo se interpretacije napake pri strojnem učenju lotili z računanjem evklidske razdalje, kjer je rezultat uspešnosti pogojen s seštevkom kvadratov razlik med predvidenimi podatki ter zbranimi podatki.

9.4 Testiranje aplikacije v avtomobilu brez uporabe vmesnika Bluetooth

Pri izdelavi naše aplikacije, predstavljene v tem diplomskem delu, ne moremo povzeti, brez da je ne bi uspeli preizkusiti v vozilu in med vožnjo, kar je tudi prvoten namen aplikacije, zato smo to naredili na varen način ter z upoštevanjem cestno-prometnih predpisov. Izbrali smo si dovolj veliko mesto (v tem primeru parkirišče pred podjetjem Gorenje v Velenju) in tam postavili testno progo, kot to prikazuje Slika 9.4.1. Naša proga v tem primeru sestoji iz štirih sektorjev, ki so postavljeni v obliki nekakšnega kroga. Naš namen je s tem simulirati krog, kot bi ga lahko odvozili na stezi, ter zagotoviti, da lahko posnamemo čim bolj točne podatke vožnje. Proga je bila zamišljena na način, da našo vožnjo začnemo pri spodnji desni označbi, torej takoj pri vhodu na parkirišče, in potem pot nadaljujemo v krogu v obratni smeri urinega kazalca. Omenjena točka je potem naš mejnik za začetek novega kroga.

Slika 9.4.1: Postavitev sektorjev

Po uspešno odvoženih krogih smo zbrali podatke, katere smo povzeli v spodnji tabeli za lažje razumevanje (Tabela 2). Odvozili smo štiri kroge in pri vsakem zabeležili podatke, torej imamo zbranih podatkov za osem prevoženih sektorjev, ker smo v tem primeru prevozili dva kroga. Dodali smo tudi zapis idealnega časa, kateri predstavlja najhitreje prevožen sektor ter služi kot idealna predstava vožnje, pod njim pa se nahaja še rezultat strojnega učenja, ki je nastal po zbranih podatkih v tabeli.

Tabela 2: Zbrane vrednosti in predikcija

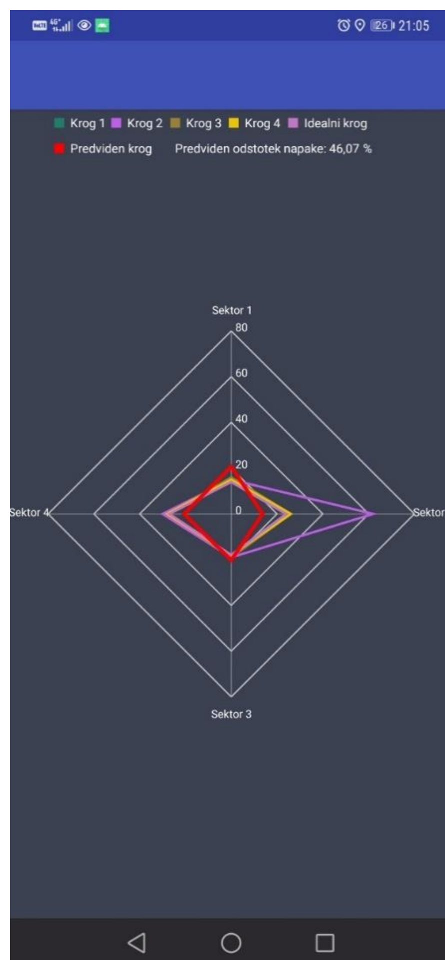
	Sektor 1 (s)	Sektor 2 (s)	Sektor 3 (s)	Sektor 4 (s)
1. krog	14	24	18	28
2. krog	15	62	19	30
3. krog	16	26	19	27
4. krog	15	26	18	28
Idealen krog	14	24	18	27
Predikcija	20.702	13.712	20.218	20.636

Tabela 3: Podatki vožnje

ST_KROGA	ST_SEKTORJA	CAS	GPS_LAT_ZAC	GPS_LONG_ZAC	GPS_LAT_KON	GPS_LONG_KON	POSPESEK_X	POSPESEK_Y	POSPESEK_Z	TIP_GUME	DIMENZIJA_GUME	PR_GUM_SPREDAJ	PR_GUM_ZADAJ
1	1	14	46.3631253910949	15.09354315698147	46.3631253910949	15.09354315698147	0.22	-0.38	-0.09	Standard	185/55 R15	2.2	2.2
1	2	24	46.36361589155923	15.091931149363518	46.36361589155923	15.091931149363518	-1.3	-0.68	-0.18	Standard	185/55 R15	2.2	2.2
1	3	18	46.36290258211814	15.092742182314396	46.36290258211814	15.092742182314396	0.04	-0.74	0.03	Standard	185/55 R15	2.2	2.2
1	4	28	46.36290258211814	15.092742182314396	46.36290258211814	15.092742182314396	0.21	-0.38	-0.31	Standard	185/55 R15	2.2	2.2
2	1	15	46.3631253910949	15.09354315698147	46.3631253910949	15.09354315698147	0.04	-0.36	-0.28	Standard	185/55 R15	2.2	2.2
2	2	62	46.36361589155923	15.091931149363518	46.36361589155923	15.091931149363518	-1.79	-0.6	-0.09	Standard	185/55 R15	2.2	2.2
2	3	19	46.36290258211814	15.092742182314396	46.36290258211814	15.092742182314396	-0.13	-0.41	-0.08	Standard	185/55 R15	2.2	2.2
2	4	30	46.36290258211814	15.092742182314396	46.36290258211814	15.092742182314396	0.24	-0.61	-0.05	Standard	185/55 R15	2.2	2.2
3	1	16	46.3631253910949	15.09354315698147	46.3631253910949	15.09354315698147	-0.03	-0.27	-0.48	Standard	185/55 R15	2.2	2.2
3	2	26	46.36361589155923	15.091931149363518	46.36361589155923	15.091931149363518	-1.44	-0.56	-0.14	Standard	185/55 R15	2.2	2.2
3	3	19	46.36290258211814	15.092742182314396	46.36290258211814	15.092742182314396	0.0	-0.53	0.08	Standard	185/55 R15	2.2	2.2
3	4	27	46.36290258211814	15.092742182314396	46.36290258211814	15.092742182314396	0.18	-0.58	0.02	Standard	185/55 R15	2.2	2.2
4	1	15	46.3631253910949	15.09354315698147	46.3631253910949	15.09354315698147	0.09	-0.33	-0.22	Standard	185/55 R15	2.2	2.2
4	2	26	46.36361589155923	15.091931149363518	46.36361589155923	15.091931149363518	-1.49	-0.59	-0.07	Standard	185/55 R15	2.2	2.2
4	3	18	46.36290258211814	15.092742182314396	46.36290258211814	15.092742182314396	-0.1	-0.42	-0.07	Standard	185/55 R15	2.2	2.2
4	4	28	46.36290258211814	15.092742182314396	46.36290258211814	15.092742182314396	-0.14	-0.75	-0.28	Standard	185/55 R15	2.2	2.2

V zgornji tabeli (Tabela 3) lahko vidimo, da se pri vožnji skozi sektor pozna naklon vrednosti na pospeškometru, na ravninskih sektorjih pa je ta vrednost enaka običajnim odstopanjem med samo vožnjo. Na sektorju številka tri je ta odklon še bolj viden, saj gre za zelo oster zavoj, kjer je pričakovano delovanje gravitacijske sile na premikajoče se

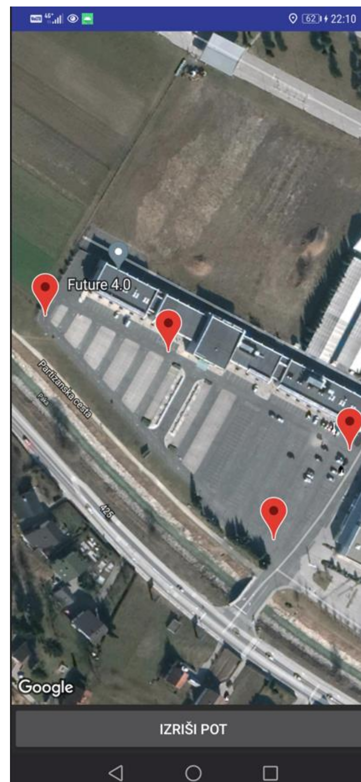
vozilo večje. Na sliki spodaj (Slika 9.4.2) lahko vidimo graf, ki ga je preračunala aplikacija. V drugem krogu lahko zelo dobro vidimo izstopajočo vrednost, kjer je vožnja skozi izmerjen sektor dva trajala toliko dlje od ostalih. Predviden krog, katerega po legendi simbolizira rdeča barva, pa lahko vidimo, da je predvidel boljši čas pri sektorju dva ter sektorju številka štiri, pri sektorju številka tri pa se vrednost precej ujema z dobljeno. Pri sektorju številka ena pa je predikcija predvidela daljše trajanje sektorja v primerjavi z izmerjenimi vrednosti pri vožnji.



Slika 9.4.2: Graf vožnje

9.5 Testiranje aplikacije z uporabo vmesnika Bluetooth

Pri izvajanju testiranja naše aplikacije moramo za celostno testiranje izvesti tudi še zadnji scenarij, in sicer uporabo vmesnika Bluetooth pri branju podatkov vozila. Na podlagi preteklih testiranj se tukaj zanašamo na to, da bo lahko aplikacija predvidela boljše rezultate, saj iz vozila prejemamo bolj točne podatke o vožnji. Pri tem smo upoštevali, da za čim bolj podobne rezultate uporabimo enako vozilo ter enako progo, saj bomo le tako lahko dobili kar najbolj natančne rezultate. Tako smo pri tem testiranje ponovno izvedli na parkirnem prostoru pred proizvodnjo podjetja Gorenje v Velenju, kar je razvidno iz Slika 9.5.1. Pri tem testiranju morda velja omeniti tudi, da smo progo nekoliko prilagodili oziroma le-ta ni popolnoma enaka prejšnji, vendar pa smo jo poskušali čim bolj poustvariti. Za večji učinek gravitacijske sile, in koliko bi le-ta mogoče znala vplivati na predvidene podatke, smo sektor številka 4 premaknili nekoliko dlje navzdol po progi, kjer je zadnji levi ovinek nekoliko ostrejši.



Slika 9.5.1: Postavitev sektorjev za vožnjo

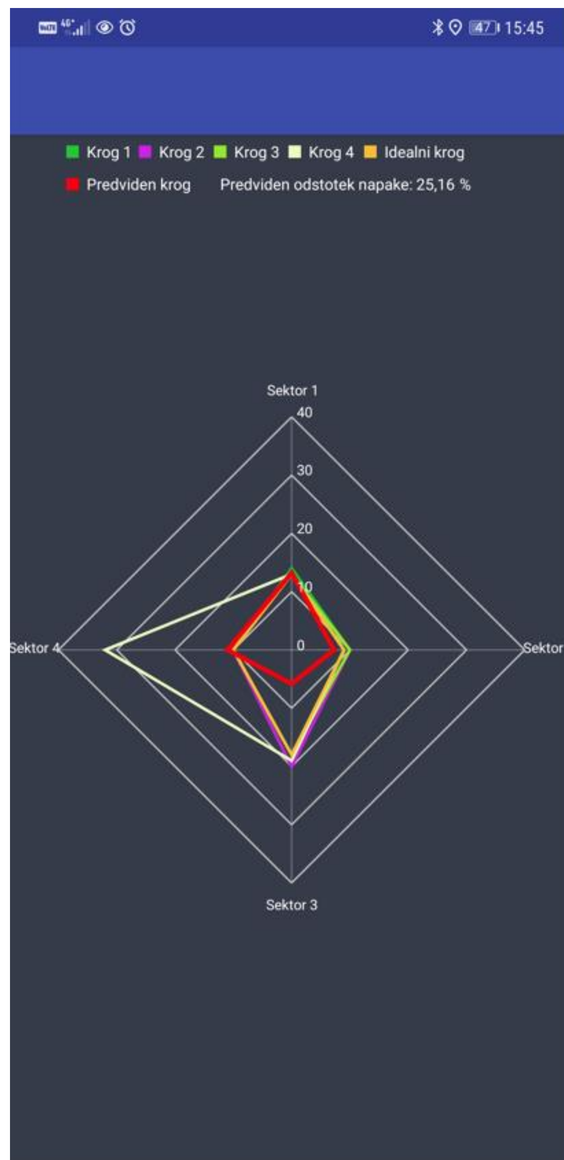
ST_KROGA	ST_SEKTORJA	CAS	TPS_ZAC	TPS_KON	OBRATI_ZAC	OBRATI_KON	OBREMENITEV_ZAC	OBREMENITEV_KON	GPS_LAT_ZAC	GPS_LONG_ZAC	GPS_LAT_KON	GPS_LONG_KON	SPD_ZAC	SPD_KON	POSPESEK_X	POSPESEK_Y	POSPESEK_Z
1	1	14	44.2	94.1	1802	2446	44.2	94.1	46.363.37...	15.092.856...	46.363.379....	15.092.856...	37	46	0.14	-0.54	0.06
1	2	10	27.8	32.1	843	1012	27.8	32.1	46.363.57...	15.091.908...	46.363.571....	15.091.908...	22	53	-3.26	-1.22	0.0
1	3	20	28.6	66.7	1039	778	28.6	66.7	4.636.238...	15.093.666...	4.636.238.2...	15.093.666...	32	26	-0.75	-1.46	0.06
1	4	10	52.2	44.4	930	2482	52.2	44.4	4.636.238...	15.093.666...	4.636.238.2...	15.093.666...	25	18	-2.05	-1.95	0.08
2	1	13	56.8	98.0	1932	1475	56.8	98.0	46.363.37...	15.092.856...	46.363.379....	15.092.856...	39	21	0.53	-0.78	-0.01
2	2	10	16.7	11.0	1503	1503	16.7	11.0	46.363.57...	15.091.908...	46.363.571....	15.091.908...	21	34	-3.17	-1.0	0.04
2	3	20	25.0	66.1	955	1717	25.0	66.1	4.636.238...	15.093.666...	4.636.238.2...	15.093.666...	32	59	-0.62	-1.18	-0.05
2	4	10	25.8	47.4	923	2729	25.8	47.4	4.636.238...	15.093.666...	4.636.238.2...	15.093.666...	31	23	-2.08	-2.34	0.19
3	1	13	40.7	73.2	1937	1937	40.7	73.2	46.363.37...	15.092.856...	46.363.379....	15.092.856...	39	32	0.33	-0.89	-0.12
3	2	10	51.9	19.2	805	1845	51.9	19.2	46.363.57...	15.091.908...	46.363.571....	15.091.908...	21	53	-3.31	-0.87	-0.1
3	3	18	55.2	94.1	946	1551	55.2	94.1	4.636.238...	15.093.666...	4.636.238.2...	15.093.666...	36	25	-0.32	-1.65	-0.4
3	4	10	3.13	26.6	907	1102	3.13	26.6	4.636.238...	15.093.666...	4.636.238.2...	15.093.666...	26	42	-2.74	-2.04	0.1
4	1	13	85.8	28.6	1905	988	85.8	28.6	46.363.37...	15.092.856...	46.363.379....	15.092.856...	39	51	0.24	-0.6	-0.01
4	2	9	11.2	30.9	876	1532	11.2	30.9	46.363.57...	15.091.908...	46.363.571....	15.091.908...	21	29	-3.22	-1.39	0.15
4	3	19	61.4	32.1	1097	1764	61.4	32.1	4.636.238...	15.093.666...	4.636.238.2...	15.093.666...	37	22	-0.36	-1.88	-0.46
4	4	32	82.0	94.9	1315	2719	82.0	94.9	4.636.238...	15.093.666...	4.636.238.2...	15.093.666...	19	24	-1.81	-1.03	0.74

Slika 9.5.2: Rezultati testiranja z OBD II vmesnikom

Zaradi obširnejše količine podatkov smo morali namesto tabele tokrat uporabiti sliko, ki zajame najvažnejše zbrane podatke, saj je drugače enostavno preveč nepregledna datoteka, da bi jo lahko prikazali kot tabelo. Na zgornji Slika 9.5.2 lahko vidimo, da se pri pospeševanju in zaviranju skozi zavoje skladoma s tem spreminja pozicija lopute. Primer delovanja lahko lepo vidimo pri sektorju številka tri, kjer se ob vhodu v sektor pri močnejšem zaviranju zabeleži manjša vrednost pozicije lopute, pri izstopu pa je le-ta nato večja, ko lahko začnemo pospeševati. Podobno lahko vidimo pri obratih, ki se pri vstopu v zavoj zmanjšujejo, pri izstopu iz njega pa narastejo. Spodaj smo zbrali rezultate, kot jih je predvidela aplikacija, katere pa predstavlja Tabela 4. Seveda ti podatki niso čisto točni, saj prihaja pri branju OBD podatkov včasih do zamikov, prejmemo pa lahko tudi slabo poslan signal, vendar so v veliki večini zbrani podatki precej točni. Pri generiranju grafa lahko točneje opazimo odstopanja ter predviden čas glede na podane zbrane podatke (Slika 9.5.3). Opazimo lahko, da je bilo predvidevanje v kar treh izmed štirih sektorjev precej natančno, saj je glavna črta predikcije precej enaka krogom, odstopa le v sektorju številka tri, kjer pa je predvidena vrednost precej manjša od dejanske. Do te napake klasifikatorja lahko pride tudi zaradi neravnin na cesti, kjer lahko merilec pospeškov zmoti kakšna luknja oziroma nepravilnost na vozišču. V bolj kontroliranem okolju, kot je na primer dirkališče, lahko v veliki večini te dejavnike izključimo.

Tabela 4: Tabela časov vožnje z OBD II vmesnikom

	Sektor 1 (s)	Sektor 2 (s)	Sektor 3 (s)	Sektor 4 (s)
1. krog	14	10	20	10
2. krog	13	10	20	10
3. krog	16	26	19	27
4. krog	13	10	18	10
Idealen krog	13	9	19	32
Predikcija	13.219	7.217	5.813	11.004



Slika 9.5.3: Graf vožnje

10 ZAKLJUČEK

Strojno učenje kot način zbiranja in obdelovanja podatkov je takorekoč nepogrešljiv pristop v današnjem času. Z našim delom lahko skoraj z gotovostjo zatrdimo, da je takšen pristop uporabe strojnega učenja v avtomobilskih oziroma moto športih lahko zelo uporaben, saj nam ponuja vpogled v nove načine izboljšav športa. S porastom uporabe mobilnih aplikacij pa je to postala tudi dostopnejša možnost, saj razen pametnega telefona ne potrebujemo skoraj ničesar drugega.

V našem zaključnem delu smo prikazali celoten postopek, ki zajema vse od prebiranja avtomobilskih podatkov po protokolu OBD II do obdelave podatkov s strojnim učenjem in na koncu tudi vizualizacijo rezultatov uporabniku. Prikazali smo uporabo najbolj razširjenih orodij za razvoj mobilnih aplikacij Android, strojno učenje in shranjevanje podatkov. Izpostavili smo tudi morebitne težave, ki se lahko pojavijo tekom prenosa podatkov iz avtomobila, kot tudi možnost napačnih meritev položaja GPS oziroma pospeškov. Težave nam je tukaj povzročalo tudi malenkost neravno cestišče, kar je morda tudi nekoliko botrovalo napaki klasifikatorja.

Pri našem delu nam je uspelo dokazati, da je mogoče razviti aplikacijo za mobilni telefon z operacijskim sistemom Android, katera se lahko poveže s komunikacijskim kanalom vozila OBD II ter v njej uspešno prebrati in prenesti podatke, ki jih s strojnim učenjem preko knjižnice Weka obdelamo in prikažemo na človeku razumljiv način. Velik izziv je predstavljala komunikacija ter zanesljivo prebiranje podatkov, prav tako tudi določitev sistema merjenja sektorjev in krogov, kateri bo lahko uspešno določil zadano progo.

Velja pa tudi omeniti, da je ta aplikacija samo približek nekega digitalnega osebnega inštruktorja. V ekstremnih pogojih, kot jih predstavljajo vsa večja dirkališča po svetu, bi za natančno merjenje ter točne podatke potrebovali še veliko več ostalih senzorjev in drugih naprav, ki bi lahko vozilo spremljale s kirurško natančnostjo. Če bi se tega lotili na takšen način, bi potem seveda potrebovali tudi večji vložek ter bolj nadzorovano okolje. Za prenos podatkov in obdelavo bi potrebovali predvsem več shranjevalnega prostora

ter kar precejšnjo zmogljivost računalnikov, kjer bi potem lahko rekli, da imamo podatke, ki nam lahko v določenih avtomobilističnih športih dajejo prednost pred konkurenco.

11 VIRI

[1] Senzorji gibanja ter delo z njimi v programskem jeziku Java Android. Dostopno na: https://developer.android.com/guide/topics/sensors/sensors_motion [26. 6. 2020].

[2] OBD II programski vmesnik v jeziku Java. Dostopno na: https://github.com/pires/obd-java-api [18. 6. 2020].

[3] Strojno učenje, Wikipedia. Dostopno na: https://en.wikipedia.org/wiki/Machine_learning [18. 6. 2020].

[4] Merjenje pospeškov v programskem jeziku Java Android. Dostopno na: https://github.com/KalebKE/FSensor [27. 7. 2020].

[5] Grafični prikaz pospeškov s ponazoritvijo pospeškometra. Dostopno na: https://github.com/KalebKE/AccelerationExplorer [27. 7. 2020].

[6] Knjižnica ScArcGauge kazalnikov vrednosti v programskem jeziku Android Studio. Dostopno na: sc-gauges/ScArcGauge.java at master · Paroca72/sc-gauges · GitHub [7. 7. 2020]

[7] Seznam ukazov protokola OBD II. Dostopno na: https://www.sparkfun.com/datasheets/Widgets/ELM327_AT_Commands.pdf [30. 6. 2020].

[8] Choo, Christopher Ledesma Weisen. *Real-time decision making in motorsports: analytics for improving professional car race strategy*. Massachusetts: Inštitut za tehnologijo v Massachusettsu, divizija za načrtovanje sistemov, program za načrtovanje in upravljanje sistemov, 2015. Dostopno na: https://dspace.mit.edu/handle/1721.1/100310 [22. 6. 2020]

[9] Moštvo Ducati o uporabi strojnega učenja pri testiranju novih motociklov. Dostopno na: https://analyticsindiamag.com/ducati-goes-big-data-machine-learning-improve-motogp-bikes-testing-process/ [22. 6. 2020]

[10] Primer implementacije servisa za lokacijo v programskem jeziku Android. Dostopno na: https://www.youtube.com/watch?v=4_RK_5bCoOY [27. 6. 2020]

[11] Pristop za risanje poti med dvema točkama na zemljevidu aktivnosti Maps v okolju Android Studio. Dostopno na: https://www.specbee.com/blogs/android-tutorials-google-map-drawing-routes-between-two-points [27. 6. 2020]

[12] Knjižnica za risanje grafov v programskem jeziku Java Android. Dostopno na: https://github.com/PhilJay/MPAndroidChart/tree/master/MPChartLib/src/main/java/com/github/mikephil/charting [15. 7. 2020]

[13] Preprost izbirnik datotek v programskem jeziku Java Android. Dostopno na: https://github.com/criss721/AndroidFileSelector/blob/master/FileSelector.java [20. 7. 2020]

[14] Zbirka funkcij in metod knjižnice Weka v programskem jeziku Java Android. Dostopno na: https://www.cs.waikato.ac.nz/ml/weka/ [7. 8. 2020]

[15] Primer skice pospeškometra. Dostopno na: https://www.iphonecake.com/app_623440332_.html [15. 8. 2021]

[16] Uporaba zemljevida v Android Studio aplikaciji. Dostopno na: https://developers.google.com/maps/documentation/android-sdk/map-with-marker

[17] Tokarz, K., Czekalski, P., & Raszka, R. (2016). *Raspberry Pi based lap counter for amateur car racing*. *Studia Informatica*, *37*(4A), 7–13. Dostopno na: https://scholar.archive.org/work/tifuzvkna5ez3p72lq2gxwhntm/access/wayback/http://studiainformatica.polsl.pl:80/index.php/SI/article/download/783/753 [27. 7. 2020]

[18] Platforma Amazon AWS DeepRacer za uporabo strojnega učenja v pri načrtovanju dirkalne vožnje. Dostopno na: https://aws.amazon.com/deepracer/

[19] Članek o protokolu OBD na Wikipedii. Dostopno na: https://en.wikipedia.org/wiki/On-board_diagnostics [20. 7. 2020]

[20] Članek o platformi FireBase na Wikipedii. Dostopno na: https://en.wikipedia.org/wiki/Firebase [18. 7. 2020]

[21] Aplikacija Lockito na Google-ovi trgovini Play. Dostopno na:

https://play.google.com/store/apps/details?id=fr.dvilleneuve.lockito&hl=en&gl=US [5. 5. 2021]