TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Imre Peeter Prikk
155691

HIIRE KASUTUSE MODELLEERIMINE VISUAALSE PROGRAMMEERIMISKEELE ÕPPIMISE JOOKSUL

Bakalaureusetöö

Juhendaja: Sven Nõmm
PhD

Tallinn 2018

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Imre Peeter Prikk

23.05.2018

Annotatsioon

Käesoleva bakalaureusetöö eesmärgiks on uurida inimeste kinemaatiliste parameetrite muutumist visuaalse programmeerimisega tutvumise ja kohanemise vältel. Samuti uuritakse, kas vigade tegemise ja parameetrite vahel on seos. Nende uurimiseks kasutati statistiliste hüpoteeside kontrollimise meetodit ja standartset masinõppe meetodit. Meetodid implementeeriti Java programmiga. Andmed on kogutud programmist *Enrect*, mis on visuaalse programmeerimiskeel. Tulemused saadi kasutades statistiliste hüpoteeside kontrollimise meetodit ja standardset masinõppe meetodit vastavalt. Töö tulemusteks saadi, et blokkide liigutamise kinemaatilisted parameetrid muutusid visuaalse programmeerimisega kohanemise jooksul. Saadi ka, et vigaselt paigutatud blokkide liigutuste ja parameetrite vahel on olemas seos. Õppimissessiooni alguse ja lõpu vahel leitud parameetrite alamhulk, mille väärtused on oluliselt erinevad. Samuti suudeti treenida klassifikaator, mis kinemaatiliste parameetrite alusel suudab tuvastada liigutused, mille tulemusel tekkis viga.

Lõputöö on kirjutatud Eesti keeles ning sisaldab teksti 20 leheküljel, 4 peatükki, 16 joonist, 2 tabelit.

Abstract

Mouse activity modelling during the learning of a visual programming language

The aim of this bachlor's thesis is to investigate the change in the kinematic parameters properties of mouse activity during the learning of a visual programming language and the correlation between mistakes and the kinematic parameters. To analyze the initially set problems a Java program was created to implement the said methots. The data for the analysis has been gathered from a visual programming implementation called *Enrect*. The results were attained by the use of statistical hypothesis testing and supervised machine learning. The results of the study concluded that there was a change in the parameters during the learning of the visual programming language and that there was a strong correlation between mistakes and the parameters of movements. A subset of the parameters was found that had changed between the start and end phase of the learning sessions. A classifier could also be trained which, taking into consideration the kinetic parameters of a movement, could accurately detect a falsely placed block.

The thesis is in Estonian and contains 20 pages of text, 4 chapters,16 figures, 2 tables.

Lühendite ja mõistete sõnastik

Liigutuse massi parameetrid	• Am – *Acceleration mass* – kiirenduse mass. Andmepunktide vaheliste kiirenduste summa. • Vm – *Velocity mass* – kiiruse mass. Andmepunktide vaheliste kiiruste summa. • t – Tegevusele kulunud aeg • l – Trajektoori kogupikkus • e – Liigutuse eukleidiline pikkus ehk teepikkus.
Blokk	Funktsionaalsust omav visuaalse programmeerimiskeele *Enrect* osa, millest koosneb visuaalne programmikood. Tegemist võib olla muutujaga, tingimuslause, arvutamise jms.

Sisukord

1 Sissejuhatus 10
2 Andmetöötlus 12
2.1 Andmete kogumine 14
2.2 Andmete töötlemine 15
3 Metoodika 17
3.1 Statistiliste hüpoteeside kontrollimine 17
3.2 Fischeri väärtus (*score*) 18
3.2.1 Fisheri väärtuse implementeerimine 18
3.3 Masinõpe 18
3.3.1 KNN 19
3.3.2 SVM 19
3.3.3 Logistiline regressioon 19
3.4 Programmikoodi selgitus 19
3.4.1 Andmete töötlemine logifailist 19
3.4.2 Fischeri väärtuse arvutamine 21
3.4.3 Statistiliste hüpoteeside kontrollimine 21
3.4.4 Klassifitseerimine 22
4 Tulemused 24
4.1 Statistiliste hüpoteeside tulemused 24
4.2 Masinõppe tulemused 26
5 Tulemuste tõlgendamine 28
6 Kokkuvõte 29
7 Kasutatud kirjandus 30
Lisa 1 – Fisheri väärtuse arvutamise programm 31
Lisa 2 – Klassifikaatorite ehitamise programm 33
Lisa 3 – Liigutuse klass 36
Lisa 4 – Vektori klass 38
Lisa 5 – Punkti klass 39
Lisa 6 – Statistiliste hüpoteeside arvutamise klass 40

Lisa 7 – Sessiooni klass 43
Lisa 8 – Faasi klass 45

Jooniste loetelu

Joonis 1 Lihtne sekundite loendur *Enrect* programmis 13
Joonis 2 Väärtuste sisestamine blokile programmis *Enrect* 13
Joonis 3 *Enrect* kasutajaliides programmi jooksmise ajal 14
Joonis 4 Bloki lohistamise liigutus 15
Joonis 5 Bloki lohistamise liigutuse vektorid (illustratiivne) 16
Joonis 6 Programmi töö 16
Joonis 7 Näide bloki lohistamise liigutusest logifailis 20
Joonis 8 Statistiliste hüpoteeside kontrollimise peameetod 22
Joonis 9. Fisheri väärtuse arvutamise programm 32
Joonis 10. Klassifikaatorite ehitamise programm 35
Joonis 11. Liigutuse klass 37
Joonis 12. Vektori klass 38
Joonis 13. Punkti klass 39
Joonis 14 Statistiliste hüpoteeside arvutamise klass 42
Joonis 15 Sessiooni klass 44
Joonis 16 Faasi klass 46

Tabelite loetelu

Tabel 1 Statistiliste hüpoteeside tulemused 24
Tabel 2 Masinõppe algoritmide tulemused 27

1 Sissejuhatus

Praegusel ajal näevad graafilised liidesed ja graafilised programmeerimiskeskkonnad palju kasutust. Seoses sellega tekib vajadus uurida, kuidas inimesed kohanevad selliste süsteemidega.

Jaapanis arendatud ja laiemale turule suundunud visuaalse programmeerimiskeele arenduskeskkonnaga, *Enrect*, on arendamisfaasis lisatud võimalus kõiki programmis tehtavaid liigutusi salvestada logifaili. Seda programmi kasutades on kogutud andmed subjektide kohta, ning on aluseks käesolevale tööle.

Käesoleva töö eesmärk on uurida, kuidas on seotud kasutaja poolt tehtavad vead liigutuse parameetritega ning kuidas muutuvad liigutuste kinemaatilised parameetrid visuaalse programmeerimiskeelega kohanemise jooksul. Töö käigus koostati Java programm, mis suudab tõlgendada logifailist andmed programmi poolt kasutatavateks andmeteks. Seejärel peab olema programm suuteline teostama andmetega statistiliste hüpoteeside kontrolli ning klassifitseerimise.

Uuringu eesmärk on välja selgitada, kuidas muutuvad kinemaatilised parameetrid programmeerimiskeelega kohandumise jooksul. Samuti, selgitada välja parameetrite järgi, kas tehtud liigutus on viga või mitte.

Töös käsitletakse kolme põhiprobleemi: parameetrite käitumine visuaalse programmeerimiskeelega kohanemise jooksul, vigade ja parameetrite seoste välja selgitamine, ning analüüsi teostava programmi välja töötamine.

Käesoleva töö uurimishüpoteesideks on:

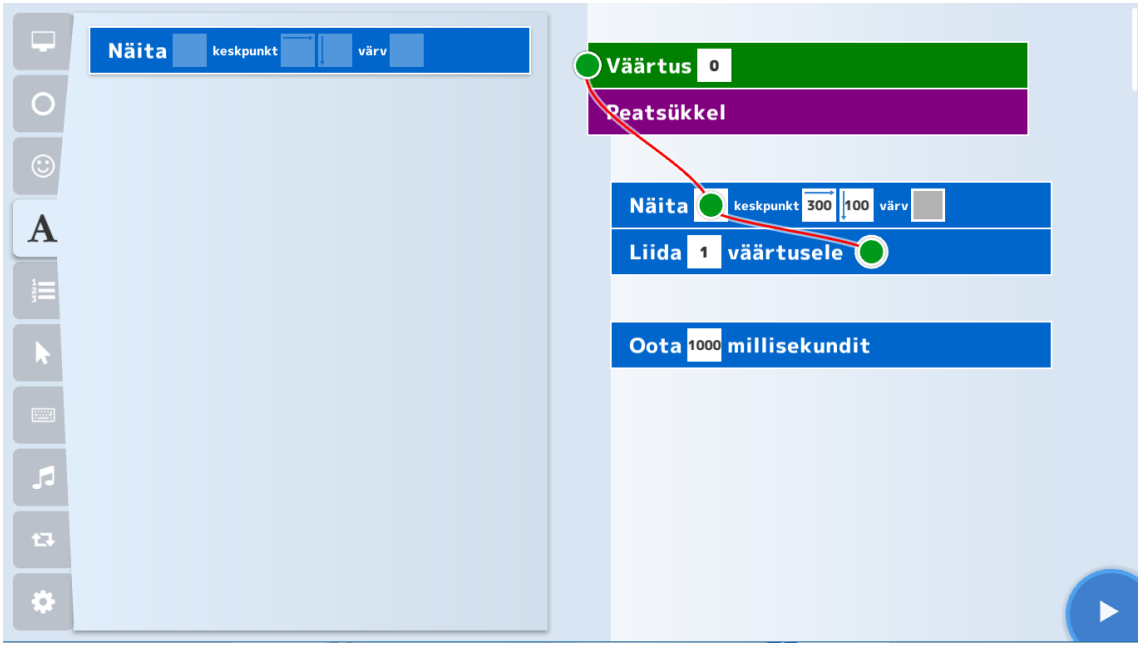
- Liigutuste parameetrid muutuvad õppeprotsessi jooksul.
- Liigutuste parameetrite ja vigade arvu vahel on olemas seos.
- Liigutuse parameetrite järgi on võimalik ennustada, kas asetatud blokk on vigane.

Töö on jagatud kolmeks peatükiks: andmetöötlus, metoodika ja tulemused. Andmetöötluses selgitatakse kust saadakse andmed ja mis nendega tehakse. Metoodikas tutvustatakse kasutatavaid meetodeid nii üldisemalt kui ka lähemalt. Tulemustes tuuakse välja metoodikas tutvustatud meetodite tulemused.

Olulisemateks infoallikateks on teemaga seotud raamatud, ning mõningad internetiallikad.

2 Andmetöötlus

Andmeid koguti Ryo Suzuki poolt loodud visuaalse programmeerimisega nimega *Enrect* [1]. Antud programm on väga sarnane ühe teise, enam levinud visuaalse programmeerimiskeelega *Scratch* [2]. Programmi koostamine on mõlemas programmis sarnane. Esmalt valitakse liidese vasakust äärest alammenüü, mille tulemusel esitatakse selles alammenüüs olevad blokid. Blokki saab asetada ajajoonele lohistades seda üle keskjoone ja paigutades selle soovitud kohale. *Enrect* võimaldab kasutajatel kasutada tsükleid, tingimuslauseid, muutujaid, teksti kuvamist, häälitsuste mängimist, tausta muutmist, lihtsaid arvutsi ja muud sarnast. Enamustel blokkidel on võimalus sisestada mingi väärtus. Väärtust saab sisestada vajutades muudetava väärtuse peale, mille tegevus on välja toodud Joonisel 2. Mitme bloki vahel saab luua seoseid, nagu toodud õhukeste punaste joontega ajajoone blokkide vahel.



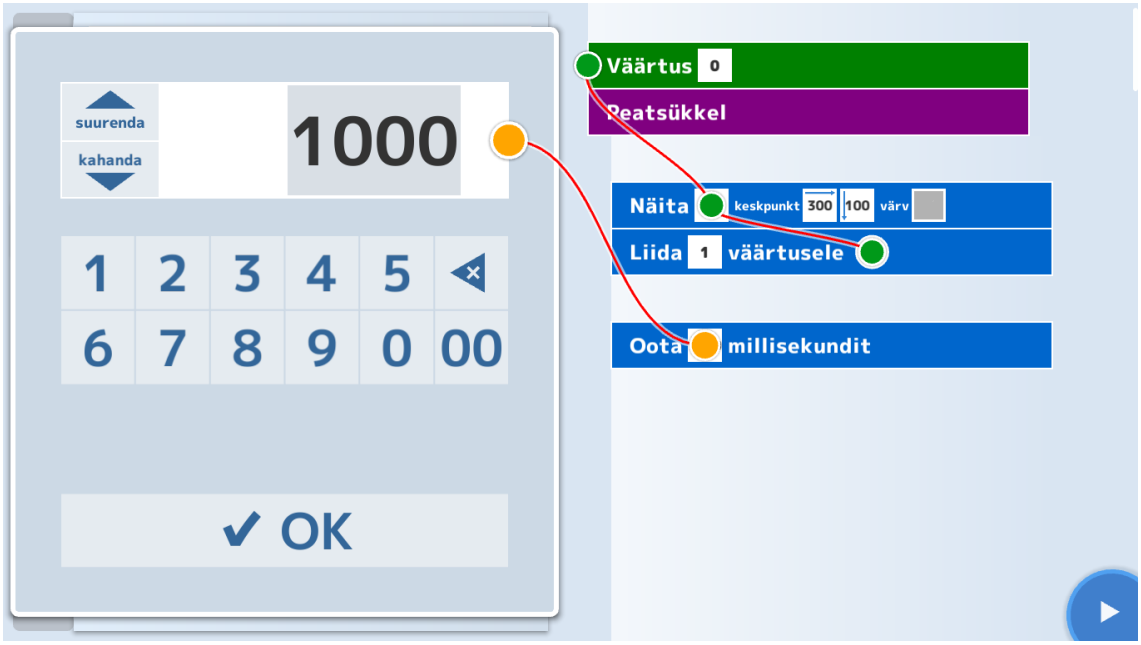
Joonis 1 Lihtne sekundite loendur *Enrect* programmis

Seosed on programmi tööks vajalikud, kuna nende abil saab viidata teistele blokkidele.

Joonisel 1 on näidatud kaks loodud seost: üks, kus väärtust näidatakse ekraanil, ning teine, kus väärtust vähendatakse ühe võrra. Antud programmis on näitamine ja lahutamine peatsükklis, mis tähendab, et neid kahte tegevust korratakse lõpmatuseni. Ajajoont saab käivitada all paremal nurgas olevale kolmnurgale vajutades. Ajajoone käivitamisel kompileeritakse masinkood ning käivitatakse see.

toodud programmi käivitamisel asendub menüü vaatega, kus on näidatud viidatud väärtus. Programmi jooksev olek on toodud Joonisel 3, kus on loendur käinud 2 sekundit. Programi saab peatada kasutades all paremal nurgas asuvat ruutu. Erinevus *Enrecti* ja Scratchi vahel on, et *Enrectile* on lisatud võimalus jäädvustada liigutuste parameetreid, millest lähemalt räägitakse peatükkides 2.2 ja 3.4.1.

Kuigi programm on suunatud eelkõige kolmanda klassi õpilastele, on see sobilik kõikidele teistele inimestele, kes soovivad programmeerimist õppida olenemata emakeelest. Eelis visuaalsel programmeerimisel on, et selle meetodiga ei kirjutata reaalset koodi, seega ei pea meelde jätma süntaksit ega funktsioonide nimesid. Õpitakse mõtlema nagu programmeerija, mille alusel saab kergema vaevaga edasi õppida mõnda keerulisemat programmeerimiskeelt nagu Java, Python, C# või muid kõrgemaid programmeerimiskeeli. Programm oli algselt vaid Jaapani keeles, kuid turgu laiendades on programm tõlgitud ka eesti ja inglise keelde. (E.k. tõlkinud Jaagup Irve) [1].

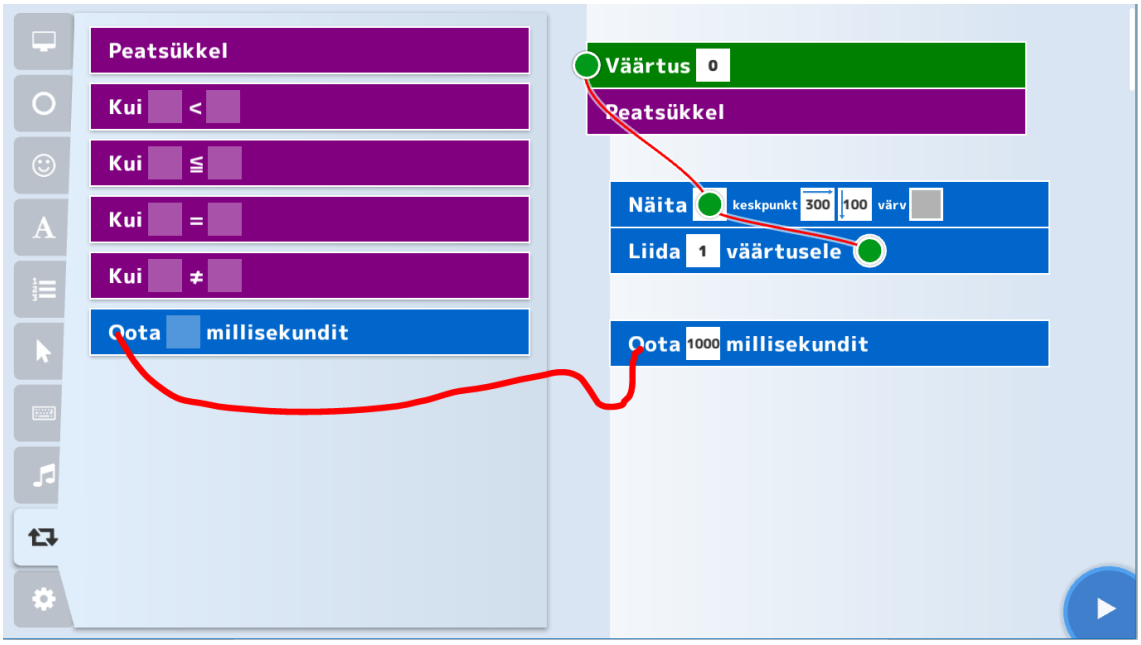


Joonis 2 Väärtuste sisestamine blokile programmis *Enrect*

Joonis 3 *Enrect* kasutajaliides programmi jooksmise ajal

2.1 Andmete kogumine

Arendades programmi *Enrect* lisas Ryo Suzuki programmile võimaluse jäädvustada kasutaja tehtud liigutusi. Bloki lohistamise vältel kogutud hiire liigutamise andmepunktide kogumit nimetatakse liigutuseks. Liigutuse näide on toodud Joonisel 4. Liigutuste andmed salvestatakse tekstikujul logifaili. Logifail sisaldab andmeid terve programmi oleku kohta. Lisaks liigutustele sisaldab logifail andmeid programmi käivitamise, bloki kustutamise ja paigutamise kohta. Liigutuse korral on salvestatud liigutuse ajal toimunud hiire liikumise x-y koordinaadid ja vastav ajatempel.

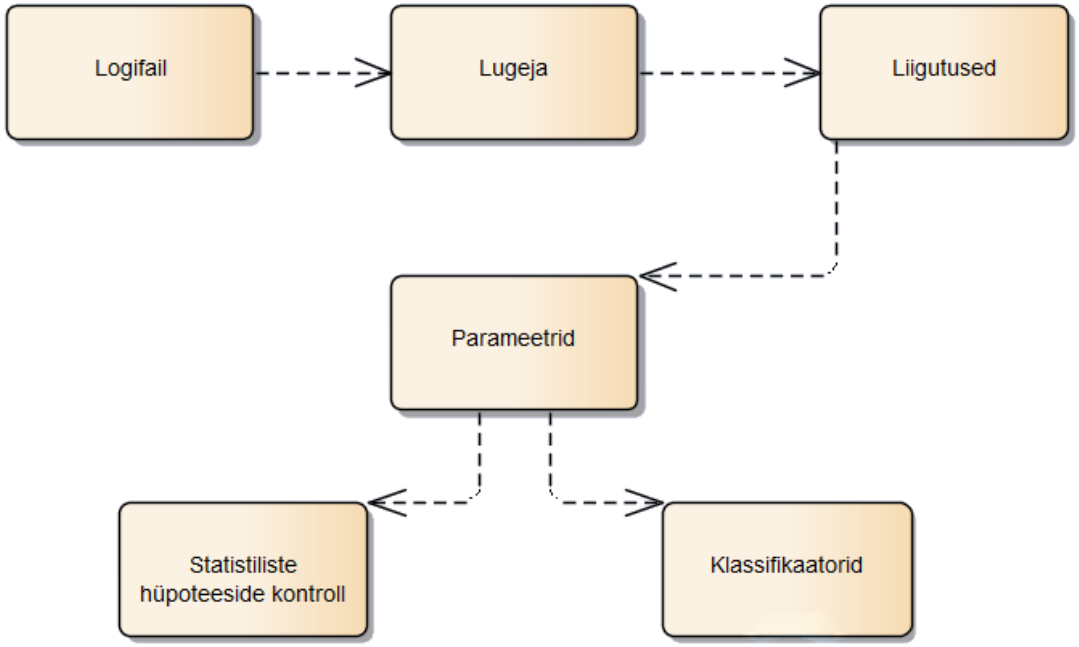


Joonis 4 Bloki lohistamise liigutus

2.2 Andmete töötlemine

Andmed olid tekstifaili formaadis ning koosnesid ajatemplist ja tegevusest, näide logifailist on toodud lisas 6. Tegevuste alla loetakse menüüvahetusi, bloki lohistamist, bloki asetust ajajoonele, bloki kustutamist ja ajajoone käivitamist. Antud töö raames vaadatakse vaid bloki kustutamist ning bloki asetust ajajoonele. Bloki lohistamisel salvestatakse selle liigutuse kestvusel hiire koordinaadid koos ajatempliga. Koordinaatide ja ajatempliga saab paika panna punkti teatud ajahetkel. Koostatud punktide kogumiga saab välja arvutada liigutuse kestvuse ning vektorid. Vektorite abil on võimalik arvutada liigutuse trajektoori ja eukleidilise teepikkuse, nagu on näidatud Joonisel 5, ning kiiruse ja kiirenduse massid [3]. See info koguti igal sessioonil iga tehtud liigutuse kohta. Kokku saadi 1546 kasutatavat liigutust 12 sessioonist. Vastavate objektide klassid on kirjeldatud lisades 3-5. Liigutuse valmimisel arvutatakse kõik parameetrid ning seda saab kasutada edasises analüüsis, mille voog on kujutatud Joonisel 6. Punktis 3.4.1 räägitakse lähemalt andmete töötlusest.

Joonis 5 Bloki lohistamise liigutuse vektorid (illustratiivne)



Joonis 6 Programmi töö

3 Metoodika

3.1 Statistiliste hüpoteeside kontrollimine

Statistilist hüpoteeside kontrollimist kasutatakse erinevate probleemide analüüsimiseks. Selle meetodi abil vastatakse küsimusele, kas erinevus on põhjustatud mõne teguri poolt või on seletatav juhusliku varieerumisega. Üldiselt esitatakse kontrollimiseks hüpoteeside paar, mis koosneb nullhüpoteesist (H_0) ja alternatiivsest (sisukast) (H_1) hüpoteesist. Nullhüpotees kehtib vaikimisi ning vastab mingile konkreetsele väärtusele. Sisukaks hüpoteesiks valitakse üldiselt suurus, mida soovitakse uurida. Mõlemad hüpoteesid püstitatakse sama valimi jaoks, mille uurimisel võib juhtuda, et nullhüpotees lükatakse ümber ja võetakse vastu alternatiivne hüpotees. Nullhüpotees lükatakse tagasi kui erinevus nullhüpoteesis püstitatud väärtusest on oluline. Alternatiivne hüpotees võetakse vastu kui erinevus püstitatud väärtusest on ebaoluline. [3, p. 294]

Kontrolli teostamiseks püstitatakse hüpoteesipaar: $H_0: \mu = \mu_0$ ja $H_1: \mu \neq \mu_0$. Seejärel arvutatakse teststatistik(z) vastavalt valemiga:

$$z = \frac{(\bar{x} - \mu_0)\sqrt{n}}{s}$$

kus $\bar{x}$ on valimi keskmine, s on standardviga, μ_0 on parameetri keskmine ja n on valimi maht. Mida rohkem erineb valimi keskmine nullhüpoteesis püstitatud väärtusest, seda kergem on tagasi lükata alternatiivset hüpoteesi. Otsus, milline hüpoteesidest võtta vastu toimub järgnevate tingimuste alusel: H_0, kui $|z| \leq z_{\alpha/2}$ ja H_1, kui $|z| > z_{\alpha/2}$. Väärtus $z_{\alpha/2}$ on normaaljaotuse täiskvantiil, ning arvutatakse olulisuse nivoo α alusel. Tegemist on suurusega, millest suurem erinevus on oluline erinevus. [3, p. 301]

Antud töös püstitati iga vaadeldava parameetri jaoks kolm hüpoteeside paari. Esimene alguse ja keskkoha kohta, teine keskkoha ja lõpu kohta ning kolmas alguse ja lõpu kohta. Andmed jagati sessioonideks ning igast sessioonist võetud keskmised alg-, kesk- ja lõppfaasist. Saadud keskmised lahutati omavahel vastavalt hüpoteesidele ning leiti faaside keskmised erinevused. Faaside keskmistest erinevustest arvutati standardhälve ja nendest omakorda keskmine standardhälve, et saada keskmine sessioonide erinevus. Standardhälbe alusel saab arvutada standardvea, mille abil saab leida z väärtuse.

3.2 Fischeri väärtus (*score*)

Fischeri väärtus aitab mõista kui tugevalt on mingi parameeter seotud sõltuva muutujaga. Fisheri väärtustamist kasutatakse klassifitseerimisülesannete lahendamiseks. See on loodud keskmise klasside vahelise eristuse ja keskmise klasside sisemise eristuse suhte mõõtmiseks. Mida suurem on parameetri Fischeri väärtus, seda olulisem on see sõltuva muutuja arvamises. Fischeri väärtuse järgi saab vaadata, milliseid parameetreid ennustuse otsustamisse valida ehk millised parameetrid sobivad enim kasutamiseks klassifikatsiooni algoritmis. Fisheri väärtuse arvutamise valem on:

$$F = \frac{\sum_{j=1}^{k} p_j (\mu_j - \mu)^2}{\sum_{j=1}^{k} p_j \sigma_j^2}$$

kus p_j on klassi j kuuluvate andmepunktide arvu ja kõikide andmepunktide arvu suhe, μ on parameetri keskmine väärtus, μ_j on klassi j kuuluvate parameetri vaartuste keskmine ning σ_j on klassi j kuuluvate parameetri väärtuste standardhälve. [4, p. 290]

3.2.1 Fisheri väärtuse implementeerimine

Väärtuste arvutamiseks loodi Java klass, mis võttis arvesse ühe parameetri kogumi ja uuritava muutuja kogumi. Väärtustamise oodatud tulemus üle kõigi parameetrite oli, et mõnel parameetril on märkimisväärselt suurem väärtus kui teistel. Paraku ei andnud väärtustamine soovitud tulemust ning kasutusele tuli võtta muu meetod. Kirjutatud programm testiti andmetega ning töötas korrektselt. Koodi on kirjeldatud Lisas 1.

3.3 Masinõpe

Masinõppe kasutamise eesmärk käesolevas töös on uurida, kas töödeldud andmete põhjal on võimalik konstrueerida klassifikaator, mis suudab eristada kahte klassi. Esimene klass kujutab ette andmepunkte, kus eemaldamist ei toimunud ning teine, kus toimus. Töös kasutati kolme populaarsemat juhendamisega õppe algoritmi: logistilist regressiooni, k-lähimat naabrit (*k-Nearest-Neighbors*) ja tugivektormasinat (*Support-Vector-Machine*). Meetodite enda sisu antud töös lähemalt uurima ei hakatud, vaid pigem kasutati neid idee kontrollimiseks. Juhendatud masinõppe puhul antakse algoritmile sisendväärtuseks hulk andmeid, mis koosnevad parameetritest ja nendele vastavatest väljundväärtustest.

Algoritmi eesmärk on koostada funktsioon, mille alusel oleks programm suuteline nii samade kui ka uute andmetega arvama väljundi väärtuse. [5]

Kui leidub vähemalt üks algoritm, mis suudab andmed klassifitseerida korrektselt, tähendab see, et vigade tegemine ja parameetrid on omavahel seotud. Töös kasutatud klassifikaatorite ehitamise kood on toodud Lisas 2.

3.3.1 KNN

KNN (*k-Nearest-Neighbors*) ehk k-lähimat naabrit. Lähimate naabrite otsimine ja nendest kauguse leidmine on arvutusrohke. KNN arvutab esmalt testandmete kauguse kõigist teistest treeningandmetest, sorteerib kõik treeningandmed kasvavas järjekorras listi. Seejärel võetakse listist k esimest instantsi ning tehakse otsus nende põhjal. See tähendab, et KNN on instantsipõhine algoritm. [6]

3.3.2 SVM

SVM (*Support-Vector-Machine*) ehk tugivektormasin on klassifikaator, mis kasutab lineaarset klassifitseerimist. Tugivektormasin suurendab andmete dimensioone nii, et uues ruumis on andmed lineaarselt eristatavad. [7]

3.3.3 Logistiline regressioon

Tegemist on tavalise regressiooniga, kus normaaljaotus asendatakse binaarjaotusega ehk sõltuv muutuja on 0 või 1. Konstrueeritakse lineaarne otsustuspiir ning mudel antakse võrrandi vormis. [8]

3.4 Programmikoodi selgitus

3.4.1 Andmete töötlemine logifailist

Esmalt määratletakse, milliseid faile vaadeldakse. Seejärel vaadatakse need ükshaaval läbi, mille jaoks on koostatud *Parser* klass. Klass võtab sisse faili nime ning väljastab kõik liigutused, mis klass failist leidis. Faili töötlus käib rea haaval. Iga rea kohta vaadatakse, kas selles reas on sees sõne “Dragging”. Vastava rea leidmisel luuakse uus *Item* instants ja alustatakse uut tsüklit, mis vaatab iga rida, kus on seesama sõne sees. Iga lisanduva rea korral lisatakse *Item* instantsile x-y koordinaadid ja ajatempel. Tsükkel jätkub kuni jätkub sõne sisaldavaid ridu. Näide logifailist on Joonisel 7, kus on näha

liigutuse logimise kõik osad. Logifaili uurides leiti, et tõmbamine lõpeb alati mingi tegevusega. Programmis toodi välja kaks tähtsamat tegevust: bloki eemaldamine ja bloki paigutamine. Liigutuse lõppedes antakse *Item* instantsile teada, et liigutus on lõppenud, mille järel arvutatakse liigutuse kinemaatilised parameetrid.

```
389771 BlockGrabbedFromMenu      7      0      (105,30,530,56)     (347,82)
389773 Dragging      (349,81)
389782 Dragging      (352,81)
389799 Dragging      (362,79)
...
390532 Dragging      (883,156)
390548 Dragging      (883,156)
390565 BlockPlaced   2      (704,126,530,56)    (883,156)
```

Joonis 7 Näide bloki lohistamise liigutusest logifailis

Parameetrid mida arvutatakse *Item* klassis on:

- Am – kiirenduse mass $v = \frac{\sqrt{x^2+y^2}}{t}$ $Am = \sum_{i=1}^{n} \frac{v_{i-1}-v_i}{t_{i-1}-t_i}$
- Vm – kiiruse mass $Vm = \sum_{i=0}^{n} \frac{\sqrt{x_i^2+y_i^2}}{t_i}$
- t – liigutusele kulunud aeg $t = t_n - t_0$
- l – liigutuse trajektoori kogupikkus $l = \sum_{i=0}^{n} \sqrt{x_i^2 + y_i^2}$
- e – liigutuse eukleidiline pikkus. $e = \sqrt{(x_n - x_0)^2 + (y_n - y_0)^2}$

Lisaks arvutati parameetrite suhted:

- $Am \,/\, e$
- $Vm \,/\, e$
- $l \,/\, e$
- $Am \,/\, t$
- $Vm \,/\, t$
- $l \,/\, t$

Kiiruse mass saadakse vektorite koostamise abiga. Nimelt on iga salvestatud punkti vahele tõmmatud vektor. Teades vektori pikkust ja kahe punkti vahelist aega, saab arvutada kiiruse sellel lõigul. Kiiruse mass saadakse kõiki kiiruseid kokku liites.

Kiirenduse mass leitakse, kasutades eelneva parameetri arvutamise jaoks loodud vektoreid. Nimelt kahe järjestikuse vektori kiiruseid teades on võimalik arvutada kiirendus. Liites iga kahe järjestikulise vektori kiirendused kokku arvutatakse kiirenduse mass.

Liigutuse trajektoori pikkuse saamiseks tuleb kokku liita iga vektori pikkus. Liigutuse eukleidilise teepikkuse leidmiseks arvutatakse vahemaa liigutuse esimese ja viimase punkti vahel. Sarnasel viisil arvutatakse ka liigutusele kulunud aeg. *Item*, *Vector* ja *Point* klassid on esitatud Lisades 3-5.

Liigutusele antakse ka kaasa, kas tegemist on bloki eemaldamise või paigutusega. Seda infot on vaja talletada, kuna eemaldamise liigutusi töös kasutada vaja pole. Tegemist on lihtsalt bloki tõmbamisega prügikasti, mis pole töös oluline. Käies läbi kõik liigutused, mis failist saadi üksteise järgi, saab ära määrata bloki kustutamine ja see vastavale blokile kinnitada. Seejärel eemaldatakse liigutused, mis lõppesid kustutamisega.

Igast liigutusest toodi välja iga parameeter ning pandi järjekorras vastavasse listi. Kokku saadi 12 parameetrit.

3.4.2 Fischeri väärtuse arvutamine

Fisheri väärtus arvutati parameetri kogumi ja uuritava muutuja kogumi vahel. Klass on implementeeritud valemi järgi.

Esmalt leiti mõlema koguse keskmised, loeti kokku mõlema klassi esindajad, arvutati klasside osakaalu suhted, arvutati standardviga ning lõpuks pandi valem kokku. Valem ja muu meetodiga seonduv on välja toodud punktis 3.2 Fischeri väärtus (*score*).

3.4.3 Statistiliste hüpoteeside kontrollimine

Statistiliste hüpoteeside kontrollimiseks tehti esmalt seda käsitsi paberil, et aru saada, kuidas see käib ning seejärel programmeeriti see valmis.

See osa programmist koosneb kolmest klassist: faas, sessioon ja põhiline klass. Statistiliste hüpoteeside kontrollimiseks on esmalt vaja klassile sessioonide kaupa infot lisada. Sessiooni lisamisel jagab programm sessiooni kolmeks faasiks. Sessioonide lisamise lõpus saab kutsuda välja meetodi *calculate* (Joonis 8), mis käivitab kõik vajaliku õiges järjekorras, et saaks teostada statistiliste hüpoteeside kontrolli.

```
public void calculate() {
    initLists();
    fillList();
    calculateDifferences();
    calculateAverages();
    calculateS();
    calculateT();
    compareAndPrintResults();
}
```

Joonis 8 Statistiliste hüpoteeside kontrollimise peameetod

Esmalt ehitatakse valmis kõik vajaminevad listid täpselt nii suureks kui vaja. Seejärel täidetakse põhiline list andmetega, mis on saadud eelnevast sessioonide sisestamisest. Seejärel arvutatakse faaside vahelised erinevused vastavalt hüpoteesides kirjeldatule ning lisatakse uude listi vastavale kohale. Sellest listist saadakse andmed, et arvutada nende erinevuste keskmine väärtus ja standardhälve. Nende kahe listi abil saab arvutada testistatistiku, mille valem on toodud punktis 3.1. Viimase sammuna prinditakse kõik tulemused konsooli. Kogu protsess toimub üle kõikide sessioonide, sellesse kuuluvate faaside ja nendesse kuuluvate parameetrite. Terve programmikood on olemas Lisades 6-8. Töö alguses teostati statistiliste hüpoteeside kontroll Exceli tabeli abil, kuid töö käigus otsustati see funktsionaalsus lisada programmi.

Antud koodi loomise raskeim osa oli kõikide listide loomine, haldamine, täitmine ja küsimine, kuna tegemist oli mitmedimendsiooniliste listidega.

3.4.4 Klassifitseerimine

Klassifitseerimiseks loodi *Classifiers* klass, mis võtab sisse parameetrite listid ning prindib välja kasutavate meetodite tulemused. Antud klass teostab logistilise regressiooni, *KNN* ja *SVM* algoritmide käivitust ja kontrolli.

Klassifikaatorid saadi *Java ML* teegist [9]. Selles paketis olevad meetodid vajavad töötamiseks *Dataset* objekti. Tehes *Dataset* objekti, saab sellese lisada instantse, mis

koosnevad *double* väärtuste listist. Pärast kõikide parameetrite andmete lisamist saab käivitada klassifikaatorite algoritmid. Pärast iga algoritmi käivitamist saab kontrollida iga algoritmi klassieraldust ja täpsust. Klasside vahelise eristuse väärtus tagastatakse algoritmi enda poolt. Täpsuse saab teada, küsides iga *Dataseti* sees oleva instantsi kohta klassifikaatori liigitust, mida võrreldakse päris väärtusega.

Logistiline regressioon on koostatud Java *smile* teegiga [10]. Antud teegi kasutamiseks tuli andmed konverteerida teegile sobivaks. Klassifikaatori täpsust kontrolliti sama meetodiga nagu teisi klassifikaatoreid. Töö alguses teostati regressioon Excel tabeli abil, kuid töö käigus otsustati see funktsionaalsus programmi kirjutada.

4 Tulemused

4.1 Statistiliste hüpoteeside tulemused

Statistiliste hüpoteeside kontroll teostati töödeldud andmetega ja tulemused on kajastatud tabelis 1.

Tabel 1 Statistiliste hüpoteeside tulemused

		Oluline viga	2,306004	
Parameeter	Nullhüpotees	Alternatiivne hüpotees	Testistatistik	Otsus
Trajektoori pikkus	$H_0: \mu_{algus} = \mu_{kesk}$	$H_1: \mu_{algus} \neq \mu_{kesk}$	2,038964	H_0
	$H_0: \mu_{kesk} = \mu_{lõpp}$	$H_1: \mu_{kesk} \neq \mu_{lõpp}$	2,998296	H_1
	$H_0: \mu_{algus} = \mu_{lõpp}$	$H_1: \mu_{algus} \neq \mu_{lõpp}$	3,688513	H_1
Eukleidiline teepikkus	$H_0: \mu_{algus} = \mu_{kesk}$	$H_1: \mu_{algus} \neq \mu_{kesk}$	5,675821	H_1
	$H_0: \mu_{kesk} = \mu_{lõpp}$	$H_1: \mu_{kesk} \neq \mu_{lõpp}$	0,438415	H_0
	$H_0: \mu_{algus} = \mu_{lõpp}$	$H_1: \mu_{algus} \neq \mu_{lõpp}$	2,816486	H_1
Kiirenduse mass	$H_0: \mu_{algus} = \mu_{kesk}$	$H_1: \mu_{algus} \neq \mu_{kesk}$	2.201361	H_0
	$H_0: \mu_{kesk} = \mu_{lõpp}$	$H_1: \mu_{kesk} \neq \mu_{lõpp}$	3.161206	H_1
	$H_0: \mu_{algus} = \mu_{lõpp}$	$H_1: \mu_{algus} \neq \mu_{lõpp}$	2.149862	H_0
Kiiruse mass	$H_0: \mu_{algus} = \mu_{kesk}$	$H_1: \mu_{algus} \neq \mu_{kesk}$	2.042009	H_0
	$H_0: \mu_{kesk} = \mu_{lõpp}$	$H_1: \mu_{kesk} \neq \mu_{lõpp}$	2.997563	H_1
	$H_0: \mu_{algus} = \mu_{lõpp}$	$H_1: \mu_{algus} \neq \mu_{lõpp}$	3.670459	H_1
	$H_0: \mu_{algus} = \mu_{kesk}$	$H_1: \mu_{algus} \neq \mu_{kesk}$	1.064239	H_0

Kiirenduse massi ja Eukleidilise teepikkuse suhe	$H_0: \mu_{kesk} = \mu_{lõpp}$	$H_1: \mu_{kesk} \neq \mu_{lõpp}$	2.929949	H_1
	$H_0: \mu_{algus} = \mu_{lõpp}$	$H_1: \mu_{algus} \neq \mu_{lõpp}$	1.817896	H_0
Kiiruse massi ja Eukleidilise teepikkuse suhe	$H_0: \mu_{algus} = \mu_{kesk}$	$H_1: \mu_{algus} \neq \mu_{kesk}$	2.695231	H_0
	$H_0: \mu_{kesk} = \mu_{lõpp}$	$H_1: \mu_{kesk} \neq \mu_{lõpp}$	2.148066	H_1
	$H_0: \mu_{algus} = \mu_{lõpp}$	$H_1: \mu_{algus} \neq \mu_{lõpp}$	3.839531	H_0
Trajektoori pikkuse ja Eukleidilise teepikkuse suhe	$H_0: \mu_{algus} = \mu_{kesk}$	$H_1: \mu_{algus} \neq \mu_{kesk}$	1.013065	H_0
	$H_0: \mu_{kesk} = \mu_{lõpp}$	$H_1: \mu_{kesk} \neq \mu_{lõpp}$	3.061956	H_1
	$H_0: \mu_{algus} = \mu_{lõpp}$	$H_1: \mu_{algus} \neq \mu_{lõpp}$	2.145696	H_0
Kulunud aeg	$H_0: \mu_{algus} = \mu_{kesk}$	$H_1: \mu_{algus} \neq \mu_{kesk}$	3.717113	H_1
	$H_0: \mu_{kesk} = \mu_{lõpp}$	$H_1: \mu_{kesk} \neq \mu_{lõpp}$	2.54047	H_1
	$H_0: \mu_{algus} = \mu_{lõpp}$	$H_1: \mu_{algus} \neq \mu_{lõpp}$	1.663384	H_0
Kiirenduse massi ja Kulunud aja suhe	$H_0: \mu_{algus} = \mu_{kesk}$	$H_1: \mu_{algus} \neq \mu_{kesk}$	0.256562	H_0
	$H_0: \mu_{kesk} = \mu_{lõpp}$	$H_1: \mu_{kesk} \neq \mu_{lõpp}$	2.542449	H_1
	$H_0: \mu_{algus} = \mu_{lõpp}$	$H_1: \mu_{algus} \neq \mu_{lõpp}$	2.879638	H_1
Kiiruse massi ja Kulunud aja suhe	$H_0: \mu_{algus} = \mu_{kesk}$	$H_1: \mu_{algus} \neq \mu_{kesk}$	1.028776	H_0
	$H_0: \mu_{kesk} = \mu_{lõpp}$	$H_1: \mu_{kesk} \neq \mu_{lõpp}$	2.505829	H_1
	$H_0: \mu_{algus} = \mu_{lõpp}$	$H_1: \mu_{algus} \neq \mu_{lõpp}$	2.59554	H_1
	$H_0: \mu_{algus} = \mu_{kesk}$	$H_1: \mu_{algus} \neq \mu_{kesk}$	1.027693	H_0

Trajektoori pikkuse ja Kulunud aja suhe	$H_0: \mu_{kesk} = \mu_{lõpp}$	$H_1: \mu_{kesk} \neq \mu_{lõpp}$	2.502007	H_1
	$H_0: \mu_{algus} = \mu_{lõpp}$	$H_1: \mu_{algus} \neq \mu_{lõpp}$	2.595429	H_1
Vigade protsent	$H_0: \mu_{algus} = \mu_{kesk}$	$H_1: \mu_{algus} \neq \mu_{kesk}$	2.331278	H_1
	$H_0: \mu_{kesk} = \mu_{lõpp}$	$H_1: \mu_{kesk} \neq \mu_{lõpp}$	0.323861	H_0
	$H_0: \mu_{algus} = \mu_{lõpp}$	$H_1: \mu_{algus} \neq \mu_{lõpp}$	2.641084	H_1

Statistiliste hüpoteeside kontrolli teostamise alguses püstitati iga parameetri kohta kolm hüpoteeside paari. Tulemuseks saadi, et esimese hüpoteesipaari alternatiivne hüpotees võeti vastu kolme parameetri puhul: eukleidiline teepikkus, aeg ja vigade tegemise sagedus. Teise hüpoteesipaari alternatiivne hüpotees võeti vastu kõikide parameetrite, välja arvatud ühe, eukleidilise teepikkuse, puhul. Kolmanda, kõige tähtsama hüpoteeside paari, puhul võeti alternatiivne hüpotees vastu seitsme parameetri korral: trajektoori pikkus, kiiruse mass, eukleidiline teepikkus, kiirenduse massi ja aja suhe, kiiruse massi ja aja suhe, trajektoori pikkuse ja aja suhe ning vigade tegemise sagedus.

Tulemuse alusel võib väita, et tõepoolest leidub parameetreid, mis erinevad vastavalt sessiooni faasile. Iga faasi vahel on erinevus vähemalt kolme või enama parameetriga.

4.2 Masinõppe tulemused

Töös kasutavate andmete põhjal oli võimalik konstrueerida kaks klassifikaatorit, mille edukus oli üle 70 protsendi. Nendeks klassifikaatoriteks on logistiline regressioon ja *KNN*. See tulemus tähendab, et valesti paigutatud blokkide liigutuste ja korrektselt paigutatud blokkide liigutuste vastavad parameetrid erinevad sellisel määral, et neid saab eristada ning kasutada ennustamisel. Kõikide kasutatud algoritmide tulemused on välja toodud tabelis 2.

Tabel 2 Masinõppe algoritmide tulemused

Algoritm	Kokku andmeid	Õigeid arvamisi	Valesid arvamisi	Protsent
KNN	1546	1117	428	72,3%
SVM	1546	810	763	52,4%
logistiline regressioon	1546	1224	322	79%

Kuna kahel algoritmil õnnestus erinevate liigutuste klasside parameetrites erinevusi, peavad parameetrid olema seotud vigade tegemisega. Seega tulemuste põhjal saab kinnitada käesoleva töö teise hüpoteesi: liigutuste parameetrite ja vigade arvu vahel on olemas seos.

5 Tulemuste tõlgendamine

Töö käigus selgus, et seitsme parameetri väärtustes toimusid muutused erinevate sessioonide faasides. Parameetrite muutumine sessiooni alguse ja lõpu vahel tähendab, et subjektide liigutused muutusid programmiga kohanemise jooksul. Vaadates andmeid saab tuua välja, et liigutused muutusid aeglasemaks, trajektoori pikkused kahanesid, ning vigaseid blokke sisestati vähem. See võib olla tulenenud sellest, et subjektid kohanesid programmeerimiskeelega ja hakkasid bloki paigutamisel rohkem mõtlema ning leidsid otsemaid teid bloki paigutamisel. Töö käigus selgus ka, et vigaselt paigutatud blokke lohistati teistmoodi. Antud liigutuste parameetrite muutuste uurimine on keeruline ning ei ole käesoleva bakalaureusetöö käsitlusala ja seega ei uuritud neid lähemalt.

Antud probleemi edasisel uurimisel saab koostada metodoloogia, mille alusel saab hinnata, millises õppestaadiumis mingi kasutajaliidese õppija parasjagu on. Samuti saab selle meetodi edasise uurimise abil koostada kergemini õpitavaid kasutajaliideseid.

6 Kokkuvõte

Bakalaureusetöö eesmärgiks oli välja selgitada, kas liigutuste parameetrid muutuvad programmiga kohanemise jooksul ning kas vigade arvu ja parameetrite vahel on seos. Selle uurimiseks koostati Java programm, mis võttis Scratchi sarnasest programmist tulevast logifailist andmed, töötles need ja teostas metoodika. Meetoditeks olid statistiliste hüpoteeside kontroll, Fisheri väärtuse arvutamine ja klassifitseerimine erinevate algoritmidega.

Statistiliste hüpoteeside kontrollimise tulemusel selgus, et programmiga kohanedes subjekti liigutuste parameetrid muutusid.

Samuti leiti, et vigade ja liigutuste parameetrite vahel on tugev seos, mida uuriti standardse masinõppe strateegiaga. Selle uurimiseks kasutati kolme juhendatud masinõppe algoritmi: lineaarsete regressiooni, *KNN* ja *SVM*. Andmete põhjal oli võimalik konstrueerida klassifikaator, mis tähendab, et viga tegevate liigutuste ja korrektsete liigutuste parameetrid erinesid piisaval määral. Klassifikaatori abil sai ka vaadata uute sisendandmete klassikuuluvust, mis tähendab, et selle abil saab teha ennustusi. Selle põhjal saab väita, et vigade tegemise ja liigutuse parameetrite vahel tugev korrelatsioon.

Kõik töös püstitatud uurimishüpoteesid leidsid kinnitust.

Töö käigus omandas autor uusi programmeerimisoskuseid ja uuendas ka ununenud teadmisi. Samuti sai autor töö jooksul sai esmase reaalse kokkupuute klassifitseerimise ja andmetöötlusega ning andmete analüüsiga.

Tulemusena tehti programm, mis suudab teostada andmete põhjal statistiliste hüpoteeside kontrolli ja koostada klassifikaatorid.

Sarnasel põhimõttel töötavat koodi oleks võimalik integreerida visuaalse programmeerimiskeele enda tarkvarasse, eesmärgiga saada jooksvalt infot tehtavate vigade kohta.

7 Kasutatud kirjandus

[1] R. Suzuki, „Enrect,“ 25 märts 2018. [Võrgumaterjal]. Available: https://enrect.org/. [Kasutatud 10 mai 2018].

[2] MIT, „Scratch,“ 2018. [Võrgumaterjal]. Available: https://scratch.mit.edu/. [Kasutatud 10 mai 2018].

[3] S. Nõmm ja A. Toomela, „An alternative approach to measure quantity and smoothness of the human limb motions,“ *Estonian Journal of Engineering*, kd. 19, nr 4, p. 298–308, 2013.

[4] A. Sauga, Statistika, Tallinn: TTÜ Kirjastus, 2017.

[5] C. C. Aggarwal, Data Mining, London: Springer, 2015.

[6] J. Brownlee, „Machine Learning Mastery,“ 16 märts 2016. [Võrgumaterjal]. Available: https://machinelearningmastery.com/supervised-and-unsupervised-machine-learning-algorithms/. [Kasutatud 10 mai 2018].

[7] T. Srivastava, „Analytics Vidhya,“ 26 märts 2018. [Võrgumaterjal]. Available: https://www.analyticsvidhya.com/blog/2018/03/introduction-k-neighbours-algorithm-clustering/. [Kasutatud 10 mai 2018].

[8] S. Ray, „Analytics Vidhya,“ 13 september 2017. [Võrgumaterjal]. Available: https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/. [Kasutatud 10 mai 2018].

[9] Statistics Solutions, „What is logistic Regression,“ 2018. [Võrgumaterjal]. Available: https://www.statisticssolutions.com/what-is-logistic-regression/. [Kasutatud 10 mai 2018].

[10] T. Abeel, „Java Machine Learning,“ 2012. [Võrgumaterjal]. Available: http://java-ml.sourceforge.net/. [Kasutatud 21 march 2018].

[11] H. Li, „Smile,“ 2018. [Võrgumaterjal]. Available: https://haifengl.github.io/smile/. [Kasutatud 2018].

Lisa 1 – Fisheri väärtuse arvutamise programm

```
public class Fisher {
    /**
     * Method to calculate the fischer score between list in and list
removed.
     * @param in  the parameter of which the score is desired.
     * @param removed  the dependant variable.
     * @return fischer's score for the parameter.
     */
    public double score(List<Double> in, List<Integer> removed) {

    //Calculating the means
        //mean of total data
        double mean = 0;
        for (int i = 0; i < in. size(); i++) {
            mean += in.get(i);
        }
        mean /= in.size();

        //mean of data of classes 1 and 0
        double mean0 = 0;
        double mean1 = 0;

        int tot0 = 0;
        int tot1 = 0;
        for (int i = 0; i < in.size(); i++) {
            if (removed.get(i) == 1) {
                mean1 += in.get(i);
                tot1++;
            } else {
                mean0 += in.get(i);
                tot0++;
            }
        }
        mean0 /= tot0;
        mean1 /= tot1;

        //total count of class 0 divided by the total count.
        double pj0 = tot0 / in.size();
        //total count of class 0 divided by the total count.
        double pj1 = tot1 / in.size();

    //Calculating the standard deviation
        //standard deviation of data of classes 1 and 0
```

```
        double s0 = 0;
        double s1 = 0;

        double sSum0 = 0;
        double sSum1 = 0;
        for (int i = 0; i < in.size(); i++) {
            if (removed.get(i) == 0) {
                sSum0 += Math.pow(in.get(i) - mean0, 2);
            } else {
                sSum1 += Math.pow(in.get(i) - mean1, 2);
            }
        }

        s0 = Math.sqrt(sSum0 / (tot0-1));
        s1 = Math.sqrt(sSum1 / (tot1-1));

    //Put the formula together and return it
        return (pj0 * Math.pow(mean0 - mean, 2) +
             pj1 * Math.pow(mean1 - mean, 2)) /
            (pj0 * Math.pow(s0, 2) + pj1 * Math.pow(s1,2));
    }
}
```

Joonis 9. Fisheri väärtuse arvutamise programm

Lisa 2 – Klassifikaatorite ehitamise programm

```
public class Classifiers {
    List<Double> Am = new ArrayList<>();
    List<Double> Vm = new ArrayList<>();
    List<Double> L = new ArrayList<>();
    List<Integer> removed = new ArrayList<>();
    List<Double> E = new ArrayList<>();
    List<Integer> T = new ArrayList<>();
    List<Double> AmE = new ArrayList<>();
    List<Double> VmE = new ArrayList<>();
    List<Double> LE = new ArrayList<>();
    List<Double> AmT = new ArrayList<>();
    List<Double> VmT = new ArrayList<>();
    List<Double> LT = new ArrayList<>();

    Dataset data = new DefaultDataset();
    /**
     * Constructor with lists.
     */
    public Classifiers(List<Double> am,
                       List<Double> vm,
                       List<Double> l,
                       List<Integer> removed,
                       List<Double> e,
                       List<Integer> t,
                       List<Double> amE,
                       List<Double> vmE,
                       List<Double> LE,
                       List<Double> amT,
                       List<Double> vmT,
                       List<Double> LT) {
        Am = am;
        Vm = vm;
        L = l;
        this.removed = removed;
        E = e;
        T = t;
        AmE = amE;
        VmE = vmE;
        this.LE = LE;
        AmT = amT;
        VmT = vmT;
        this.LT = LT;

        makeDataSet();
    }

    /**
```

```
 * Constructor with data set.
 */
public Classifiers(Dataset data) {
    this.data = data;
}

/**
 * Builds the dataset from the lists.
 */
private void makeDataSet() {
    for (int i = 0; i < removed.size(); i++) {
        Instance instance = new DenseInstance(new double[]{
                Am.get(i),
                Vm.get(i),
                L.get(i),
                E.get(i),
                T.get(i),
                AmE.get(i),
                VmE.get(i),
                LE.get(i),
                AmT.get(i),
                VmT.get(i),
                LT.get(i)
        });
        instance.setClassValue(removed.get(i));
        data.add(instance);
    }
}

/**
 * Prints out the results of the K-Nearest-Neighbor algorithm.
 */
public void KNN() {

    Classifier knn = new KNearestNeighbors(15);
    knn.buildClassifier(data);

    int correct = 0, wrong = 0;
    for (Instance inst : data) {
        Object predictedClassValue = knn.classify(inst);
        Object realClassValue = inst.classValue();
        if (predictedClassValue.equals(realClassValue))
            correct++;
        else
            wrong++;
    }

    System.out.println("Classifier performance");
    System.out.println(
         "\tCorrect: " + correct + "\n\t  Wrong: " + wrong);
```

```
        System.out.println("KNN Classifier evaluation");
        Map<Object, PerformanceMeasure> knnPm =
                                EvaluateDataset.testDataset(knn, data);
        for (Object o : knnPm.keySet())
            System.out.println("\t" + o + ": " + knnPm.get(o).getAccuracy());
    }

    /**
     * Prints out the results of the Support-Vector-Machine algorithm.
     */
    public void SVM() {

        Classifier svm = new LibSVM();
        svm.buildClassifier(data);

        int correct = 0, wrong = 0;
        for (Instance inst : data) {
            Object predictedClassValue = svm.classify(inst);
            Object realClassValue = inst.classValue();
            if (predictedClassValue.equals(realClassValue))
                correct++;
            else
                wrong++;
        }

        System.out.println("Classifier performance");
        System.out.println(
             "\tCorrect: " + correct + "\n\t  Wrong: " + wrong);

        System.out.println("SVM Classifier evaluation");
        Map<Object, PerformanceMeasure> svmPm =
                                EvaluateDataset.testDataset(svm, data);
        for(Object o:svmPm.keySet())
            System.out.println("\t" + o+": "+svmPm.get(o).getAccuracy());
    }

}
```

Joonis 10. Klassifikaatorite ehitamise programm

Lisa 3 – Liigutuse klass

```
public class Item {
    double Am;
    double Vm;
    private int timeStart;
    private int timeEnd;
    private boolean removed = false;
    private boolean fromMenu = false;
    private boolean run = false;
    private int startX;
    private int startY;
    private int endX;
    private int endY;
    private List<Point> points = new ArrayList<>();
    private List<Vector> vectors = new ArrayList<>();
    private List<Double> acceleration = new ArrayList<>();
    private List<Integer> velocity = new ArrayList<>();
    private int timeTotal;
    private double euclidDistance;
    private double actualDistance;
    private double speed;

    public Item() {
    }

    public Item(String special) {
        if (special.equals("Removed")) removed = true;
    }

    /**
     * a method that finalizes the Item aka sets and calculates all the
     * parameters
     */
    public void endItem() {

        //Set/calculate times
        timeStart = points.get(0).time;
        timeEnd = points.get(points.size() - 1).time;
        timeTotal = timeEnd - timeStart;

        //Calculate distance
        euclidDistance = Math.sqrt(
           Math.pow(
             (points.get(points.size() - 1).x - points.get(0).x), 2) +
             Math.pow(
              (points.get(points.size() - 1).y - points.get(0).y), 2));
        speed = euclidDistance / timeTotal * 10;

        //Set starting and ending values
```

```
        startX = points.get(0).x;
        startY = points.get(0).y;
        endX = points.get(points.size() - 1).x;
        endY = points.get(points.size() - 1).y;

        if (!removed && !run) doVectors();

        //Calculate Am and Vm
        for (Vector vector : vectors) {
            actualDistance += vector.length;
            Vm += Math.abs(vector.velocity);

        }
        for (Double accel : acceleration) {
            Am += Math.abs(accel);
        }
    }
```

Joonis 11. Liigutuse klass

Lisa 4 – Vektori klass

```
public class Vector {

    double length;
    double velocity;
    private Point a;
    private Point b;
    private double x;
    private double y;


    public Vector(Point a, Point b) {

        this.a = a;
        this.b = b;
        x = b.x - a.x;
        y = b.y - a.y;

        /**
         length = sqrt(X^2 + Y^2)
         X = b.x - a.x
         unit is px
         */
        length = Math.sqrt(
             Math.pow((b.x - a.x), 2) + Math.pow((b.y - a.y), 2));

        /**
         V = distance / time
         unit is px/ms
         */
        velocity = length / (a.time - b.time);

    }

    public Point getA() {
        return a;
    }

    public Point getB() {
        return b;
    }
}
```

Joonis 12. Vektori klass

Lisa 5 – Punkti klass

```
public class Point {
    int x;
    int y;
    int time;

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public Point(int x, int y, int time) {
        this.x = x;
        this.y = y;
        this.time = time;

    }
}
```

Joonis 13. Punkti klass

Lisa 6 – Statistiliste hüpoteeside arvutamise klass

```
public class Hypothesis {

    private List<HypSession> sessions = new ArrayList<>();

    public static final double SIGNIFICANT = 2.306004;

    public void addSession(HypSession session) {
        sessions.add(session);
    }
    List<List<List<Double>>> data = new ArrayList<>();
    List<List<List<Double>>> diff = new ArrayList<>();
    List<List<Double>> avg = new ArrayList<>();
    List<List<Double>> s = new ArrayList<>();
    List<List<Double>> z = new ArrayList<>();

    private void initLists() {
        for (int f = 0; f <= 2; f++) {
            data.add(new ArrayList<>());
            diff.add(new ArrayList<>());
            for (int p = 0; p <= 11; p++) {
                data.get(f).add(new ArrayList<>());
                diff.get(f).add(new ArrayList<>());
            }
        }
        for (int p = 0; p <= 11; p++) {
            avg.add(new ArrayList<>());
            s.add(new ArrayList<>());
            z.add(new ArrayList<>());
        }
    }

    private void fillList() {
        for (int s = 0; s < sessions.size(); s++) {
            for (int p = 0; p <= 11; p++) {
              data.get(0).get(p).add(sessions.get(s).start.getData().get(p));
                data.get(1).get(p).add(sessions.get(s).mid.getData().get(p));
                data.get(2).get(p).add(sessions.get(s).end.getData().get(p));
            }
        }
    }

    private void calculateDifferences() {
        for (int p = 0; p <= 11; p++) {
            for (int s = 0; s < sessions.size(); s++) {
                diff.get(0).get(p).add(data.get(0).get(p).get(s) -
data.get(1).get(p).get(s));
                diff.get(1).get(p).add(data.get(1).get(p).get(s) -
data.get(2).get(p).get(s));
```

```
                diff.get(2).get(p).add(data.get(0).get(p).get(s) -
data.get(2).get(p).get(s));
            }
        }
    }

    private void calculateAverages() {
        for (int p = 0; p <= 11; p++) {
            for (int f = 0; f <= 2; f++) {
                avg.get(p).add(calculateMean(diff.get(f).get(p)));
            }
        }
    }

    private void calculateS() {
        for (int p = 0; p <= 11; p++) {
            for (int f = 0; f <= 2; f++) {
                s.get(p).add(calculateStandardDeviation(diff.get(f).get(p)));
                System.out.println(diff.get(f).get(p));
            }
        }
    }

    private void calculateT() {
        for (int p = 0; p <= 11; p++) {
            for (int f = 0; f <= 2; f++) {
                try {
                    z.get(p).add(Math.abs(avg.get(p).get(f) /
s.get(p).get(f)) * sessions.size() - 1);
                } catch (IndexOutOfBoundsException e) {
                    z.get(p).add(-1.0);
                }
            }
        }
    }

    private void compareAndPrintResults() {
        for (int p = 0; p <= 11; p++) {
            for (int f = 0; f <= 2; f++) {
                System.out.println("Parameter " + (p + 1) + " results: ");
                System.out.println("\tH" + (f + 1) + ": T=" +
z.get(p).get(f));
            }
        }
    }
    private double calculateMean(List<Double> in) {
        double out = 0;
        for (Double i : in) {
            out += i;
        }
        return out / in.size();
```

```
    }

    private double calculateStandardDeviation(List<Double> in) {
        double out = 0;
        double mean = calculateMean(in);
        for (Double i : in) {
            out += Math.pow(i - mean, 2);
        }
        return Math.sqrt(out / in.size() - 1);
    }
    public void calculate() {
        initLists();
        fillList();
        calculateDifferences();
        calculateAverages();
        calculateS();
        calculateT();
        compareAndPrintResults();
    }


}
```

Joonis 14 Statistiliste hüpoteeside arvutamise klass

Lisa 7 – Sessiooni klass

```
public class HypSession {
    List<Double> Am = new ArrayList<>();
    List<Double> Vm = new ArrayList<>();
    List<Double> L = new ArrayList<>();
    List<Integer> removed = new ArrayList<>();
    List<Double> E = new ArrayList<>();
    List<Integer> T = new ArrayList<>();
    List<Double> AmE = new ArrayList<>();
    List<Double> VmE = new ArrayList<>();
    List<Double> LE = new ArrayList<>();
    List<Double> AmT = new ArrayList<>();
    List<Double> VmT = new ArrayList<>();
    List<Double> LT = new ArrayList<>();

    List<Double> startAvg = new ArrayList<>();
    List<Double> midAvg = new ArrayList<>();
    List<Double> endAvg = new ArrayList<>();

    HypPhase start;
    HypPhase mid;
    HypPhase end;

    public HypSession(List<Double> am, List<Double> vm, List<Double> l,
List<Integer> removed, List<Double> e, List<Integer> t, List<Double> amE,
List<Double> vmE, List<Double> LE, List<Double> amT, List<Double> vmT,
List<Double> LT) {
        Am = am;
        Vm = vm;
        L = l;
        this.removed = removed;
        E = e;
        T = t;
        AmE = amE;
        VmE = vmE;
        this.LE = LE;
        AmT = amT;
        VmT = vmT;
        this.LT = LT;
        makePhases();
        calculateSessionAverages();
    }

    private void makePhases() {
        int sizeOfPhase = Am.size() / 3;
        start = new HypPhase(
                this.Am.subList(0, sizeOfPhase),
                this.Vm.subList(0, sizeOfPhase),
                this.L.subList(0, sizeOfPhase),
```

```
                this.removed.subList(0, sizeOfPhase),
                this.E.subList(0, sizeOfPhase),
                this.T.subList(0, sizeOfPhase),
                this.AmE.subList(0, sizeOfPhase),
                this.VmE.subList(0, sizeOfPhase),
                this.LE.subList(0, sizeOfPhase),
                this.AmT.subList(0, sizeOfPhase),
                this.VmT.subList(0, sizeOfPhase),
                this.LT.subList(0, sizeOfPhase)
        );
        mid = new HypPhase(
                this.Am.subList(sizeOfPhase, sizeOfPhase * 2),
                this.Vm.subList(sizeOfPhase, sizeOfPhase * 2),
                this.L.subList(sizeOfPhase, sizeOfPhase * 2),
                this.removed.subList(sizeOfPhase, sizeOfPhase * 2),
                this.E.subList(sizeOfPhase, sizeOfPhase * 2),
                this.T.subList(sizeOfPhase, sizeOfPhase * 2),
                this.AmE.subList(sizeOfPhase, sizeOfPhase * 2),
                this.VmE.subList(sizeOfPhase, sizeOfPhase * 2),
                this.LE.subList(sizeOfPhase, sizeOfPhase * 2),
                this.AmT.subList(sizeOfPhase, sizeOfPhase * 2),
                this.VmT.subList(sizeOfPhase, sizeOfPhase * 2),
                this.LT.subList(sizeOfPhase, sizeOfPhase * 2)
        );
        end = new HypPhase(
                this.Am.subList(sizeOfPhase * 2, sizeOfPhase * 3),
                this.Vm.subList(sizeOfPhase * 2, sizeOfPhase * 3),
                this.L.subList(sizeOfPhase * 2, sizeOfPhase * 3),
                this.removed.subList(sizeOfPhase * 2, sizeOfPhase * 3),
                this.E.subList(sizeOfPhase * 2, sizeOfPhase * 3),
                this.T.subList(sizeOfPhase * 2, sizeOfPhase * 3),
                this.AmE.subList(sizeOfPhase * 2, sizeOfPhase * 3),
                this.VmE.subList(sizeOfPhase * 2, sizeOfPhase * 3),
                this.LE.subList(sizeOfPhase * 2, sizeOfPhase * 3),
                this.AmT.subList(sizeOfPhase * 2, sizeOfPhase * 3),
                this.VmT.subList(sizeOfPhase * 2, sizeOfPhase * 3),
                this.LT.subList(sizeOfPhase * 2, sizeOfPhase * 3)
        );
    }

    private void calculateSessionAverages() {
        for (int p = 0; p < 11; p++) {
            startAvg.add(start.getData().get(p));
            midAvg.add(mid.getData().get(p));
            endAvg.add(end.getData().get(p));
        }
    }
}
```

Joonis 15 Sessiooni klass

Lisa 8 – Faasi klass

```
public class HypPhase {
    List<Double> Am = new ArrayList<>();
    List<Double> Vm = new ArrayList<>();
    List<Double> L = new ArrayList<>();
    List<Integer> removed = new ArrayList<>();
    List<Double> E = new ArrayList<>();
    List<Integer> T = new ArrayList<>();
    List<Double> AmE = new ArrayList<>();
    List<Double> VmE = new ArrayList<>();
    List<Double> LE = new ArrayList<>();
    List<Double> AmT = new ArrayList<>();
    List<Double> VmT = new ArrayList<>();
    List<Double> LT = new ArrayList<>();


    public HypPhase(List<Double> am, List<Double> vm, List<Double> l,
List<Integer> removed, List<Double> e, List<Integer> t, List<Double> amE,
List<Double> vmE, List<Double> LE, List<Double> amT, List<Double> vmT,
List<Double> LT) {
        Am = am;
        Vm = vm;
        L = l;
        this.removed = removed;
        E = e;
        T = t;
        AmE = amE;
        VmE = vmE;
        this.LE = LE;
        AmT = amT;
        VmT = vmT;
        this.LT = LT;
    }




    private double calculateMeanD(List<Double> in) {
        double out = 0;
        for (Double i : in) {
            out += i;
        }
        return out / in.size();
    }
    private double calculateMeanI(List<Integer> in) {
        double out = 0;
        for (Integer i : in) {
            out += i;
        }
```

```
        return out / in.size();
    }

    public List<Double> getData() {
        List<Double> out = new ArrayList<>();
        out.add(calculateMeanD(Am));
        out.add(calculateMeanD(Vm));
        out.add(calculateMeanD(L));
        out.add(calculateMeanI(removed));
        out.add(calculateMeanD(E));
        out.add(calculateMeanI(T));
        out.add(calculateMeanD(AmE));
        out.add(calculateMeanD(VmE));
        out.add(calculateMeanD(LE));
        out.add(calculateMeanD(AmT));
        out.add(calculateMeanD(VmT));
        out.add(calculateMeanD(LT));
        return out;
    }

}
```

Joonis 16 Faasi klass