Tampereen yliopisto

Jussi Nevanperä

TEKOÄLYN HYÖDYNTÄMINEN OHJELMISTOTESTAUKSESSA

Informaatioteknologian ja viestinnän tiedekunta
Kandidaattitutkielma
Toukokuu 2020

TIIVISTELMÄ

Jussi Nevanperä: Tekoälyn hyödyntäminen ohjelmistotestauksessa
Kandidaattitutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Toukokuu 2020

Tämä tutkielma on kirjallisuuskatsaus, jossa tarkastellaan millaisia vaikutuksia tekoälyn hyödyntämisellä on ohjelmistotestauksessa. Motiivina tutkielman aiheelle toimi ohjelmistotestauksen samanaikainen tärkeys, sekä sen potentiaalinen riski tuhlata kehitystyön tärkeitä resursseja. Tutkielman tarkoituksena on aiempien tutkimuksien ja esimerkkien avulla perustella, miksi tekoälyn hyödyntäminen ohjelmistotestauksessa on tärkeää. Perustelut tekoälyn hyödyntämiselle pitävät sisällään ohjelmistotestauksen työvaiheita, joissa tekoälyä voidaan hyödyntää, erillisen osuuden tekoälyn vahvuuksista yleensä, sekä yksityiskohtaisempia esimerkkejä tekoälyn soveltamistavoista ohjelmistotestauksessa.

Tutkielman tutkimusmateriaalia on kerätty kolmesta eri tietokannasta: IEEE Xploresta, ScienceDirectistä sekä ACM:stä. Aineisto koottiin vuosien 2005 ja 2020 välillä julkaistuista artikkeleista ja konferenssipapereista, jotka käsittelevät sekä ohjelmistotestausta, että tapoja soveltaa tekoälyä sen toteutuksessa. Lopullisia tutkimukseen valittuja lähteitä kertyi 16 kappaletta.

Tutkimuksen tulokset tukevat tekoälyn käytön tärkeyttä ohjelmistotestauksessa. Tekoälyä osataan soveltaa ohjelmistotestauksen jokaisessa vaiheessa. Jo ensimmäisiä testejä varten kyetään tekemään ennustuksia eri osa-alueiden virheherkkyydestä. Näin säästetään testaajien aikaa ja voidaan kohdistaa testaamista oikeisiin alueisiin. Tekoäly kykenee yhdistelemään aiemmista testeistä kerättyä dataa ja tämän pohjalta muokkaamaan tulevia testejä. Myös testien onnistumista voidaan seurata ja tuloksia kategorisoida tekoälyn avulla. Kun testi on ajettu ja ohjelmistoa on sen pohjalta muutettu, tekoäly mahdollistaa nopean tavan löytää kaikki osa-alueet, joihin muutos on vaikuttanut. Näin voidaan valikoida, mitkä testit on syytä ajaa ja mitkä voidaan jättää välistä. Kaikki nämä sovellustavat edesauttavat ohjelmistojen nopeampaa ja halvempaa valmistusta.

Avainsanat: tekoäly, ohjelmistotestaus, laaduntarkkailu, automatisointi, koneoppiminen

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck –ohjelmalla.

Sisällysluettelo

1 Johdanto **1**

2 Tutkimusmenetelmä **2**

3 Tekoäly ohjelmistotestauksessa **3**

3.1 Tekoälyn sovellustavat testauksessa 4

3.2 Tekoälyn asema testauksessa nyt ja tulevaisuudessa 5

3.3 Tekoälyn vahvuudet 5

3.4 Tekoälyn heikkoudet 6

4 Käyttöesimerkkejä **7**

4.1 Sumea logiikka (fuzzy logic) regressiotestauksessa 7

4.2 Testioraakkeli (Test oracle) 8

4.3 Käyttömallien luominen tilastollisella valinnalla 8

4.4 Virtuaaliassistentti 9

5 Pohdintaa **10**

6 Yhteenveto **11**

Lähdeluettelo **12**

1 Johdanto

Ohjelmistotestaus tarkoittaa ohjelmiston testaamista erilaisilla syötteillä. Ensisijaisesti syötteiden tulee heijastaa ohjelmiston oletettua käyttötapaa, mutta perusteellisessa testauksessa on otettava huomioon myös poikkeustapaukset ja ohjelmiston mahdollinen väärinkäyttö. Testien avulla suunnittelijat paikantavat ohjelmiston virheitä, joita korjaamalla he pääsevät lähemmäs ideaalia lopullista tuotetta (Whittaker, 2000).

Niin tärkeä osa kuin ohjelmistotestaus onkin, se on vähiten ymmärretty osa kehitysprosessia. Artikkelissaan Whittaker (2000) tarjoaa esimerkin testaustilanteesta, jossa käyttäjä antaa ohjelmalle syötteenä kellonajan. Syötteen oikeellisuus tarkistetaan for-loopissa ja oikeellisia vaihtoehtoja on 86400 kpl. Tämän lisäksi tietenkin virheellisiä syötteitä on loputtomasti. On helppoa kuvitella, kuinka vaihtoehtojen määrä kasvaa, kun testauksen kohteena on jotain monimuotoisempaa ja vaikeammin rajattavaa kuin kellonaika. Mitä enemmän vaihtoehtoja ohjelmistossa ilmenee, sitä vaikeampi testaajien on löytää pienin mahdollinen määrä testejä, joilla kaikki ajotavat ja syötekentät voidaan tutkia.

Kun ensimmäinen kierros testejä on ajettu, mahdolliset virheet raportoidaan kehittäjille, joiden tehtävänä on korjata ne. Tämän jälkeen on luotava uusia testejä tai vähintään muokattava aiempia skriptejä. Skriptit ovat testaamisessa käytettäviä komentosarjoja, joiden mukaan ohjelmisto yrittää toimia. Mikä tahansa korjaus voi onnistuessaankin kuitenkin rikkoa jonkin muun osan koodista. Koska testaus on hidasta, nämä uudet virheet saattavat jäädä huomaamatta. Testaajat eivät aikataulujen puitteissa aina aja uudelle koodille kaikkia aiempia testejä, vaan vain testin, joka korjattavan virheen on löytänyt (Whittaker, 2000; Briand, 2008).

Tekoälyn määritelmään kuuluu kyky jäljitellä ihmisen älykästä toimintaa. Tässä tutkielmassa näistä älykkäistä toiminnoista keskitytään oppimiseen. Tekoälyn kykyä oppia kutsutaan koneoppimiseksi. Koneoppimisen avulla ohjelma oppii muokkaamaan toimintaansa saamansa datan pohjalta (Techopedia, 2020). Tekoäly kykenee löytämään saamastaan datasta yhteyksiä ja toimintamalleja. Mitä enemmän dataa on tarjolla, sitä enemmän ja sitä monipuolisempia yhteyksiä ohjelma kykenee luomaan, ja mitä enemmän yhteyksiä on, sitä luotettavammin ohjelma toimii.

Tekoälyn vaikutus työpaikkoihin on puhuttanut tutkijoita niin kauan kun sen mahdollistama automatisointi on ollut olemassa. Chuin, Manyikan ja Miremadin (2015) tutkimus Four fundamentals of workplace automation on yksi aiheeseen perustuvista tutkimuksista. Tutkimuksen mukaan yleinen virheajatus työpaikkojen automatisoinnissa on työntekijän korvaaminen kokonaan. Tutkimuksen tekoaikana arviolta vain 5 % työpaikoista olisi mahdollista korvata tekoälyllä kokonaan. Sen sijaan on paljon yksittäisiä työtehtäviä, joiden automatisointi on erittäin helppoa. Tutkimuksessa

arvioitiin, että 30 % työtehtävistä 60 % amerikkalaisista yrityksistä voitaisiin jo automatisoida. Monissa näistä tehtävistä, tekoälyllä voidaan jo yltää samoihin tai korkeampiin tuloksiin kuin keskiverron työntekijän toimesta.

Kaksi tärkeintä osa-aluetta, joihin tekoäly voi ohjelmistotestauksessa vaikuttaa ovat aika ja tarkkuus. Manuaalisesti tehtävä testaus on erittäin hidasta, koska ajettavien testien välissä on analysoitava löydetyt virheet ja muokattava seuraavia testejä niiden perusteella. Taloudellisista syistä manuaalisesti tehtävät testaukset pyritään minimoimaan. Tämä tarkoittaa, että kehitetään vain sen verran testejä, jotta jokainen osa-alue saataisiin testattua kerran. Tällaisen rajoitetun testaamisen huono puoli on, että se johtaa herkästi kriittisten virheiden jäämiseen koodiin. Esimerkkinä vuonna 1999 tapahtunut Mars Polar Lander ohjelmistobugi, jonka hinnaksi tuli 165 miljoonaa dollaria (Abraham & Horst, 2004). Tällaisien kriittisten virheiden ehkäisemiseksi tekoäly on erittäin tärkeää. Joskus testauksen tarkkuus on suurempi taloudellinen säästö, kuin automatisoidun testauksen mahdollistama ajansäästö.

Tekoälyä hyödynnetään testauksessa laajasti (Hourani et al., 2019). Sen käyttöä sovelletaan jo ensimmäisistä testikierroksista pitkäjännitteiseen regressiotestaukseen asti (Yadav & Dutta, 2016). Vaikka tekoälyä onkin hyödynnetty testaukseen kuuluvassa virheentunnistuksessa jo pitkään, on niiden algoritmeissa vieläkin parannettavaa (Hall & Bowes, 2012). Lisääntynyt kiinnostus tekoälyn hyödyntämiseen saattaa kuitenkin auttaa karsimaan näitä puutteita lähitulevaisuudessa (King et al., 2019).

Tutkielmani perustuu suurimmalta osin konferenssipapereihin, jotka käsittelevät tekoälyn tämänhetkistä vaikutusta ohjelmistotestaukseen, tekoälyn eri käyttömalleihin ohjelmistotestauksessa ja siihen miltä ohjelmistotestauksen tulevaisuus näyttää tekoälyn suhteen. Esimerkkien ja perustelujen pohjalta kysymys, johon tutkielmani vastaa on ”Miksi tekoälyn hyödyntäminen on tärkeää ohjelmistotestauksessa?”. Tutkielma on rakennettu seuraavasti. Kappaleessa kaksi esittelen tutkimusmenetelmäni. Tämä tarkoittaa käyttämieni hakusanojen, rajausten ja tietokantojen valinnan sekä sopivien tutkimuksien suodatuksen. Kappaleessa kolme esittelen lähteiden pohjalta kerätyt tutkimustulokset tekoälyn hyödyntämisestä. Nämä olen jakanut useampaan alakategoriaan. Kappaleessa neljä esittelen esimerkkitutkimuksia, joista jokainen esittelee perusteellisemmin yhden tavan käyttää tekoälyä ohjelmistotestauksessa. Kappaleessa viisi kerron omia pohdintojani aiheeseen liittyen. Mitä odotin tutkimukselta ja kuinka se vastasi odotuksiani. Viimeisessä kappaleessa on tutkielman yhteenveto.

2 Tutkimusmenetelmä

Tutkimukseni on tyypiltään kirjallisuuskatsaus. Aloitin tutkimusmateriaalin keräämisen Tampereen yliopiston Andor -palvelusta. Sieltä valitsin kolme tietojenkäsittelyalan tietokantaa: IEEE Xplore, ScienceDirect ja acm. Kirjallisuuskatsaukseni hakusanoina

käytin pääasiassa termejä ”artificial intelligence” ja ”software testing”. Lyhensin ”testing” sanan ”test*”:iksi sivustoilla, missä tekstiä täydentävää ominaisuutta oli mahdollista käyttää.

Aloitin kirjallisuuskatsauksen IEEE Xplore tietokannassa. Ensimmäinen hakuni oli muotoa: (("Document Title":"Artificial Intelligence") AND ("Document Title":"Software testing")). Termien haku oli siis rajattu artikkeleiden otsikoihin. Tämän lisäksi rajasin haun vuosille 2005 – 2020. Tämä haku tuotti vain kaksi tulosta. Ensimmäinen liittyi verkkosivujen testaamiseen, minkä päätin rajata tutkimukseni ulkopuolelle. Toinen tuloksista oli Houranin ja muiden (2019) konferenssipaperi ”The Impact of Artificial Intelligence on Software Testing”. Kyseinen paperi sopi sisällöltään tutkimukseeni niin hyvin, että tarkistin sen lähteet löytääkseni lisää tutkimusmateriaalia. 22 lähteestä viisi sopi tutkimustarkoituksiini.

Jatkoin hakua kohdistamalla hakusanani ensin artikkelin avainsanoihin ja tämän jälkeen koko metatekstiin. Pidin haussani aina saman vuosirajauksen 2005 – 2020. Nämä haut tuottivat runsaammin tuloksia, mutta tiivistelmien pohjalta jouduin rajaamaan suurimman osan niistä tutkimukseni ulkopuolelle. Yleisimmät syyt artikkelien poisrajaamiselle olivat tekoälyn liian pieni rooli ja ohjelmistotestauksen kohdistuminen tekoälyohjelmaan sen sijaan, että tekoälyä hyödynnettäisiin itse testauksessa.

Kun olin tyytyväinen IEEE Xploresta saamiini tuloksiin, jatkoin kirjallisuuskatsaustani ScienceDirect tietokannassa. Kyseinen kanta ei tuottanut yhtä paljon sopivaa materiaalia tutkimukselleni. Hakujen jälkeen sain kuitenkin suodatettua 3 kelvollista lähdettä. Viimeiseksi tein hakuja ACM-tietokannassa. ACM osoittautui heikoimmaksi lähteiden lähteeksi tutkimuksiani varten. Useimmat tutkimukset jäivät aiheeni ulkopuolelle tutkimalla joko verkkosivuihin tai koodikieliin liittyvää testaamista. Marijanin (2016) artikkeli testien muokkaamisesta tilastollisen testivalinnan avulla oli kuitenkin oleellinen.

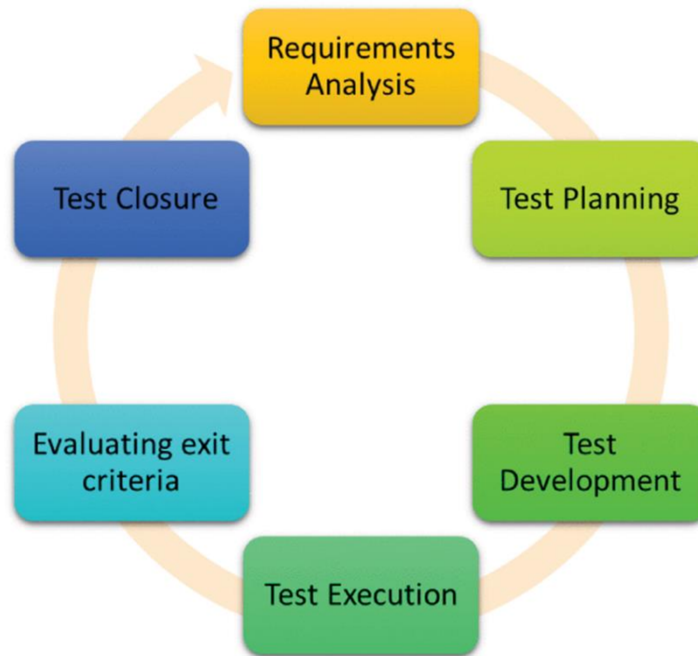
Lopuksi hyödynsin Google Scholaria ja hain tutkimukseni tueksi lisämateriaalia työpaikkojen automatisoinnista. Tämän jälkeen minulla oli 20 lähdettä. Tässä oli liikaa lähteitä tutkimukseni tarpeisiin, joten karsin 7 lähdettä syvällisemmän lähdeanalysoinnin jälkeen. Syinä karsimisille olivat lähes identtisten lähteiden olemassa olo ja käyttöesimerkit jotka päätin jättää tutkimuksen ulkopuolelle.

3 Tekoäly ohjelmistotestauksessa

Tämä kappale esittelee kirjallisuudesta johdettuja perusteita tekoälyn hyödyllisyydelle ohjelmistotestauksessa. Näihin perusteisiin kuuluvat tekoälyn laajat käyttömahdollisuudet, sen asema testauksessa tällä hetkellä suunnittelijoiden näkökulmasta, sekä yleisesti tekoälyn vahvuuksista puhumista. Otan kuitenkin myös kantaa tekoälyn heikkouksiin ohjelmistotestauksessa.

3.1 Tekoälyn sovellustavat testauksessa

Ohjelmistotestausta ei ikinä tehdä yhdellä optimoidulla testillä. Testien on kehityttävä sen mukaan, miten ohjelmisto muuttuu ja mitä aiemmilla testeillä on saatu selville. Ohjelmistotestauksen kiertokulku tarveanalyysistä testin lopetukseen ja takaisin analyysiin on esitetty kuvassa 1 (Hourani et al., 2019). Tekoälyä on mahdollista käyttää kaikissa näissä testauksen vaiheissa: Tarveanalyysi, testin suunnittelu, testin kehitys, testin suoritus, testin onnistumisen arviointi ja testin yhteenveto.



Kuva 1 Testauksen kiertokulku (Hourani et al., 2019)

Ohjelmistoa rakentaessa ja viimeistään sen ensimmäisen testattavan version valmistuessa, voidaan tekoälyä hyödyntämällä paikantaa alueita jotka tulevat olemaan virhealttiita (Hourani et al., 2019; Yadav & Dutta, 2016). Kun virhealttiit alueet ovat tiedossa, testaajat voivat panostaa enemmän niiden tutkimiseen. Kun ohjelmisto on valmis, eli se on läpäissyt testit niin hyvin, että se voidaan ottaa käyttöön, sen osien muuttumisherkkyyttä (Chandra et al., 2016) on mahdollista tutkia erilaisilla koneoppimisen menetelmillä. Näin voidaan ennustaa, mitkä osa-alueet todennäköisimmin muuttuvat ja näin ollen ovat alttiita muutoksista johtuville virheille.

Aina kun valmiissa ohjelmistossa tapahtuu muutoksia, testaajat joutuvat käymään kaikki osa-alueet läpi uusien virheiden varalta. Mutta koko ohjelmiston kattavan testin ajaminen jokaisen muutoksen kohdalla olisi erittäin kallista ja aikaa vievää. Tekoälyä hyödyntämällä voidaan nopeasti tarkistaa kaikki alueet joihin kyseessä oleva muutos vaikuttaa. Tämä nopeuttaa optimaalisten testien valitsemista ja säästää aikaa (Yadav & Dutta, 2016; Abraham & Horst, 2004; Rauf & Alanazi, 2014).

Sen lisäksi, että tekoäly osaa rajata testattavat alueet, muokata testejä aiempien testien mukaan ja ajaa testejä itsenäisesti, se voi myös arvioida testin onnistumista testioraakkelin muodossa. Testioraakkelin luominen vie paljon aikaa, koska testaajien on perusteellisesti opetettava sitä ennalta kerätyn datan avulla, kunnes sen tekemät päätelmät vastaavat ihmistestaajan päätelmiä (Briand, 2008).

3.2 Tekoälyn asema testauksessa nyt ja tulevaisuudessa

Tutkimuksien mukaan tekoälyn hyödyntäminen useimmissa testauksen vaiheissa on mahdollista ja kannattavaa. Zhu (2018) tutki tekoälyn hyödyntämisen mahdollisuutta luomalla matemaattisia lausekkeita, jotka toimivien testien pohjalta kävisivät toteen. Itkin ja muut (2019) visoivat virtuaaliassistenttia, koska tekoälyn hyödyntäminen tuntuu väistämättömältä nykyaikaisten ja yhä kehittyvien monimutkaisten ohjelmistojen laadun ylläpitämiseksi. Roper (2019), Yadav & Dutta (2016), Agarwal ja muut (2012), Hall & Bowes (2012) sekä Chandra ja muut (2016) ovat testanneet olemassa olevien testiohjelmistojen toimivuutta ja luoneet testausta varten uusia.

Tekoälyn soveltamisessa on kuitenkin vielä puutteita (Ishani et al., 2015; Hall & Bowes, 2012). Vaikka tekoälyä sovelletaankin jo testauksen jokaisessa vaiheessa, optimaalisiin tuloksiin ei vielä ylletä jokaisella osa-alueella. On vielä epäselvää johtuvatko puutteet siitä, että optimaalinen ohjelmoiminen veisi niin paljon aikaa, ettei todellista ajan säästöä enään saavutettaisi, vai siitä ettei heikkoa tekoälyä yksinkertaisesti voi ohjelmoida huomioimaan kaikkia poikkeustapauksia.

Kingin ja muiden (2019) konferenssissa USA:ssa analysoitiin tutkijoiden ajatuksia tekoälyyn liittyen. Analysoinnin pohjana oli kysely, joka oli tehty 328 IT-alan työntekijälle, joista kaikilla on jotain tietämystä tekoälyyn liittyen. Suurin osa oli sitä mieltä, että tekoälyllä tulee olemaan suuri vaikutus niin manuaaliseen testaukseen, kuin automatisoituunkin. Tarkemmin sanottuna tekoälyn oletetaan korvaavan automatisoidun testaamisen tyystin, mutta suurin osa ei usko sen täysin korvaavan manuaalista testaamista koskaan.

3.3 Tekoälyn vahvuudet

Automatisoinnilla ja tekoälyn hyödyntämisellä tähdätään lähtökohtaisesti työntekijöiden ajansäästöön ja tuotteiden nopeampaan valmistukseen. Läpikäymieni tutkimuksien perusteella tekoälyn avulla kyetään so saavuttamaan molemmat tavoitteet. Tämä saavutettu ajansäästö ei kuitenkaan näytä vaikuttavan tuotteen laatuun negatiivisesti. Tekoäly kykenee siis löytämään suurimman osan, ellei kaikkia ihmistestaajankin löytämistä virheistä. Ajansäästöä lisää muun muassa tekoälyn mahdollistamasta virheherkkyyksien esilaskeminen, testien optimoiminen sekä kyky ajaa ja analysoida

enemmän testejä annetussa ajassa kuin ihmistestaaja (Chandra et al., 2016; Itkin et al., 2019; Hourani et al., 2019; Yadav & Dutta, 2016; Briand, 2008; Marijan, 2016).

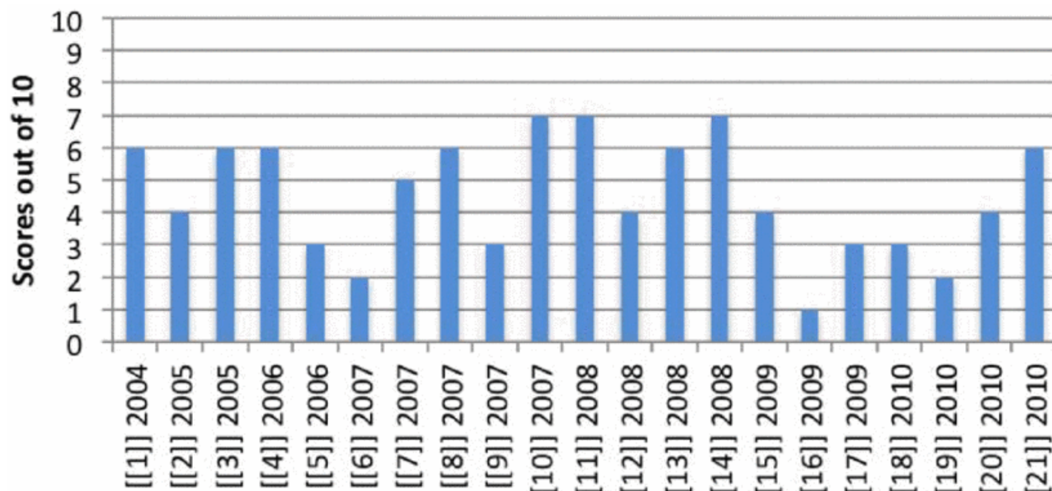
Tekoälyn avulla on myös mahdollista käyttää laitteiston tehoja optimaalisesti (Rauf & Alanazi, 2014). Siinä missä ihmistestaajan kyky käyttää laitteistoa rajoittuu hänen fyysisiin ja kognitiivisiin kykyihinsä, tekoälyn käyttämät resurssit voidaan ennalta päättää ja delegoida laitteiston tehojen mukaisesti. Tällä tavoin voidaan säästää energiaa ja välttää laitteiston turhaa päivittämistä.

Laitteiston tehojen lisäksi tekoäly mahdollistaa myös testaajien optimaalisen hyödyntämisen. Testaaja, joka hyödyntää tekoälyä virheherkkien alueiden paikantamisessa, testien valinnassa, sekä testien varsinaisessa ajamisessa, kykenee tehokkaammin hyödyntämään työhön käyttämänsä ajan. Tekoäly vapauttaa aikaa varsinaiselle luovalle työlle.

3.4 Tekoälyn heikkoudet

Tekoälyn soveltamisessa on vielä puutteita. Tutkimuksien mukaan koneoppimiseen pohjautuvien virheentunnistusohjelmien laatu ei ole kehittynyt tasaisesti, eivätkä ne edelleenkään täytä kaikkia tärkeitä kriteerejä (Hall & Bowes, 2012; Arora et al., 2015). Hallin ja Bowesin tutkimuksessa perehdyttiin 21 eri virheentunnistusta suorittavaan tekoälyalgoritmiin aikavälillä 2004-2010. Tutkimuksessa määriteltiin 10 tärkeää ominaisuutta, jotka algoritmin kuuluisi omata. Näiden ominaisuuksien olemassaoloa testattiin ja jokaisesta ominaisuudesta metodi sai yhden pisteen. Testin tulokset on esitelty kuvassa 2. X akselille on merkitty metodin numero, eli ohjelmistot 1-21, sekä vuosi jona kyseinen ohjelmisto on luotu, 2004-2010. Y akseli osoittaa, kuinka monta tutkimuksessa määriteltyä ominaisuutta metodi sisältää.

11 algoritmia, yli puolet tutkimuksen kohteista, sai alle viisi pistettä. Parhaiten pärjänneetkään algoritmit eivät saaneet enempää kuin seitsemän pistettä. Tämä johti noin neljän ja puolen pisteen keskiarvoon 21 algoritmin välillä.



Kuva 2 Testausohjelmistojen arvosanat (Hall & Bowes, 2012)

Poikkeuksien ohjelmoiminen on vaikeaa, siksi ihmistestaajat kykenevät edelleen löytämään enemmän virheitä, kuin tekoäly. Siinäkin tapauksessa, että tekoäly löytää yhtä paljon tai enemmän virheitä kuin ihminen, sen toiminta on sitä rakentaneen testaajan huolellisuuden varassa. Tämä ongelma ei tietenkään ole rajattu vain testaukseen, vaan on läsnä aina tekoälyä ohjelmoitaessa.

Tekoäly on edelleen vain työkalu. Täydellinen automatisointi ei siis tule Heikon tekoälyn puitteissa olemaan mahdollista. Heikolla tekoälyllä viitataan tekoälyyn, joka ei kykene itsenäiseen ajatteluun, vaan pystyy suoriutumaan vain tehtävistä, joiden tekemiseen se on varta vasten ohjelmoitu. Koska testauksessa ilmenee säännöllisesti poikkeustapauksia, Heikko tekoäly tarvitsee aina ihmisen tuloksien oikeellisuuden tarkistamiseen.

4 Käyttöesimerkkejä

Tässä kappaleessa esittelen, sekä tutkittuja käyttötapoja, että tutkijoiden visioimia toistaiseksi toteuttamattomia keinoja hyödyntää tekoälyä ohjelmistotestauksessa. Esimerkit eivät sisällä kaikkia tutkimusmateriaalissa esiintyneitä tekoälyn sovelluskeinoja, vaan tutkijan mielestä oleellisimmat.

4.1 Sumea logiikka (fuzzy logic) regressiotestauksessa

Sumea logiikka on arviointitekniikka, missä tulosta ei arvioida binäärisesti, vaan ääripäiden välissä olevat arvot lasketaan ja huomioidaan myös (Techopedia, 2020). Tämä tarkoittaa, että ohjelmiston osille luodaan joukko sääntöjä. Sääntöjen toteutuvuus eri osa-alueiden yhteistyössä määrittää sen saaman pisteellisen arvon (Abraham & Horst, 2004).

Regressiotestaus on testausta mitä ohjelmistolle tehdään päivityksien jälkeen. Sen tarkoituksena on löytää päivityksen mahdollisesti aiheuttamia virheitä ohjelmistosta. Sumeaa logiikkaa on mahdollista hyödyntää regressiotestauksessa eri osa-alueiden virheherkkyyden määrittelemisessä, jotta löydetään optimaalinen tärkeysjärjestys testien ajamiselle (Yadav & Dutta, 2016).

APFD (Average Percentage of Fault Detected) on arvo, jonka mukaan ohjelmiston osa-alueiden virheherkkyys ilmaistaan. Kuvassa 3 on esitetty APFD:n laskukaava niin, että ”T” tarkoittaa testiä, ”F” testien löytämiä virheitä, ”n” testien lukumäärää ja ”m” virheiden lukumäärää.

$$\mathrm{APFD} = 1 - \frac{\mathrm{TF_1} + \mathrm{TF_2} + \mathrm{TF_3} + \ldots.. + \mathrm{TF_m}}{\mathrm{n} * \mathrm{m}} + \frac{1}{2\mathrm{n}}$$

Kuva 3 APFD -kaava (Yadav & Dutta, 2016).

Esimerkkitutkimuksessa määriteltiin kolme arvioitavaa piirrettä, joiden mukaan testien tärkeys laskettiin. Nämä olivat: ”Löydettyjen virheiden lukumäärä”, ”Suoritusaika” ja

”Testin vaikutusalue”. Vaikutusalue tarkoittaa, kuinka suuren osan ohjelmistosta testin on käytävä läpi toimiakseen. Näille piirteille laskettiin sumeat arvot tarkempien pisteellisten tulosten pohjalta, jotka pystyttiin jakamaan kolmeen arvoluokkaan: ”matala”, ”keskiverto” ja ”korkea”. Sumean logiikan käyttöä varten määriteltiin sääntö jokaista arvoyhdistelmää kohden. Sääntöjä tuli siis 27kpl (3^3). Näiden sääntöjen pohjalta määriteltiin lopulta ”Testin sopivuus”. Esimerkkinä sääntö 1: ”Jos löydetään *vähän* virheitä, suoritusaika on *matala* ja vaikutusalue on *pieni* päätetään, että *testi ei ole sopiva*.” (Yadav & Dutta, 2016).

Tekniikkaa sovellettiin 6 eri ohjelmaan. Sumealla logiikalla laskettu APFD kattoi huonoimmillaan 91,7 % ja parhaimmillaan 96,2 % optimaalisesta tuloksesta. Optimaalinen tulos on tässä tapauksessa testi, joka löytää kaikki mahdolliset virheet. Tämän perusteella tekniikkaa voidaan pitää kelvollisena tapana priorisoida testausalueita.

4.2 Testioraakkeli (Test oracle)

Testioraakkeli on ohjelma, jonka tehtävänä on arvioida läpäisikö ohjelmisto sille suoritetun testin hyväksytysti (Computer Notes, 2020). Automatisoidun oraakkelin luominen on eniten pohjatyötä vaativa tapa soveltaa tekoälyä, mitä tutkimusmateriaalini piti sisällään. Automatisoidun testioraakkelin perusteellinen opetus vaatii paljon ammattilaistestaajan aikaa. Automatisoiduilla testioraakkeleita on hyödynnetty esimerkiksi lääketieteessä röntgenkuvien analysointiin (Briand, 2008). Omassa testissään Briand (2008) tiiminsä kanssa onnistui luomaan yksinkertaisen testioraakkelin päätöspuun avulla. Oraakkelin arvioitiin tuottavan luotettavia tuloksia yli 80% käyttökerroista.

Testioraakkelin opetus tapahtuu ohjatun oppimisen avulla. Ohjatussa oppimisessa ohjelmalle annetaan dataa, jonka pohjalta sen on tarkoitus oppia kaavoja, joilla datan itsenäinen tulkinta tehdään ohjelmalle mahdolliseksi. Kaavojen oppimiseksi ohjelmoijan on tarkasti analysoitava ohjelman tekemiä tulkintoja ja korjata niitä tarvittaessa. Oppimiseen käytettävää dataa muokataan ja monimutkaistetaan sitä myöten kun ohjelma oppii enemmän kaavoja ja yhteyksiä. Jos kyseessä on toimenpide missä virhemarginaali on erittäin pieni, testioraakkelin koulutus saatetaan joutua aloittamaan alusta useita kertoja. Riittävän opetuksen lopuksi testioraakkeli pystyy itsenäisesti arvioimaan testien tuloksia.

4.3 Käyttömallien luominen tilastollisella valinnalla

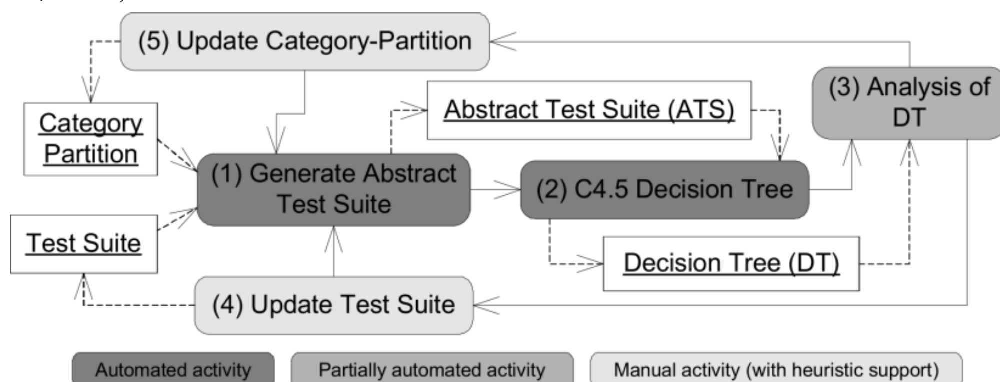
Jotkin ohjelmistot tarjoavat käyttäjilleen mahdollisuuden muokata ohjelmistoa omien tarpeidensa mukaan. Mitä enemmän ohjelmaa on mahdollista muokata, sitä enemmän siinä on testattavaa. Tällaisten ohjelmistojen kohdalla optimaalisten testien valitsemista voidaan helpottaa käyttömallien avulla. Käyttömallit ovat esimerkkejä, miten ohjelmistoa voidaan käyttää ja todennäköisesti käytetään.

Käyttömallien luomiseen tarvitaan kuitenkin dataa. Datan kerääminen edellyttää, että ohjelmisto on joko ollut vapaassa käytössä tai sille on tehty kattava betatestaus. Datan pohjalta päästään rakentamaan käyttömalleja tilastollisen valikoinnin avulla. Tämän valikoinnin periaatteena käytetään Markovin ketjua, joka laskee seuraavan askeleen todennäköisyyden aiemman askeleen pohjalta (Techopedia, 2020). Tilastollisessa valikoinnissa lasketaan toimintojen käyttötodennäköisyyksiä ja niiden välisiä siirtymätodennäköisyyksiä, eli mitä toimintoja eniten käytetään ja missä järjestyksessä. Käyttömalli pitää sisällään tiedon ohjelmiston kaikkien ominaisuuksien suhteesta toisiinsa. Tällä tavoin voidaan luoda ennustuksia eri osa-alueiden käyttötiheydestä ja näin ollen valikoida testejä, jotka keskittyvät eniten käytettyihin alueisiin ja mukailevat toimintojen käyttöjärjestystä (Marijan, 2016). Testien perusteella tilastollinen valinta voi tehdä testaamisesta viisi tai jopa kahdeksan kertaa nopeampaa.

4.4 Koneoppimiseen perustuva Black box -testauksen parantaminen (MELBA)

MELBA (Briand, 2008) hyödyntää Category-Partition -metodia (CP). Se on ohjelmistotestaukseen suunniteltu metodi, jossa tutkittava ohjelma jaetaan ensin pienimpiin mahdollisiin tutkittaviin osiin. Nämä osat kattavat kaikki ohjelmiston sisällä tapahtuvat tiedonsiirrot käyttäjän syötteestä aina viimeiseen ohjelmiston ulosantiin asti. Analysoidessaan ohjelmiston osia, CP ottaa huomioon myös kaikki ohjelmiston tilat ja erilaiset syötteet, jotka kyseessä olevaan osaan vaikuttavat. Tämän lisäksi kaikista toisiinsa vaikuttavista osista rakennetaan toimintapareja. CP:n antaman tiedon pohjalta voidaan testin vaikutusalue rajata niin, että se kattaa ohjelmiston kaikki kriittisimmät funktiot (Ostrand & Balcer, 1988).

MELBAn toimintaprosessi on kuvattu kuvassa 4. CP:n luoman rajauksen pohjalta luodaan abstrakti testisarja kohdassa 1. Seuraavaksi testisarjasta etsitään sääntöjä, joiden mukaan CP:n toimintaparit vaikuttavat toisiinsa. Näiden sääntöjen pohjalta rakennetaan päätöspuu. Tämä vaihe tapahtuu C4.5 algoritmin avulla kohdassa 2. Tätä päätöspuuta analysoidaan kohdassa 3. Analyysin pohjalta CP:tä ja testisarjaa päivitetään kohdissa 4 ja 5. Prosessia iteroidaan niin kauan kuin CP:ssä ja testisarjassa löydetään parannettavaa (Briand, 2008).



Kuva 4 MELBA prosessi kuvattuna

4.5 Virtuaaliassistentti

Virtuaaliassistentti (VA) on tekoälyllä varustettu ohjelma, joka tyypillisesti vastaa kysymyksiin ja tarjoaa tietoa oman toiminta-alueensa sisältä. Tällaisia assistentteja hyödynnetään jo monissa tehtävissä. Esimerkkeinä opetusmaailmassa käytettävät assistentit ja vanhustenhoitoon käytetyt seurarobotit (Itkin et al., 2019).

Ohjelmistotestaus on alue missä VA:ta ei ole sovellettu, mutta visioita aiheeseen liittyen on jo olemassa. Testien suunnitteleminen, suorittaminen ja niiden datan kerääminen on jo mahdollista suorittaa tekoälyn keinoin. AV tarjoaisi apukeinoja testaajan ja automatisoidun testiohjelman saumattomampaan yhteistyöhön tarjoamalla käyttöliittymän tiedon tutkimiseen. Suuresta datamäärästä olisi entistä helpompi suodattaa oleellisinta tietoa. Jos käyttöliittymään sovelletaan avoimen keskustelun työkaluja, saadaan testauksen kehittämisestä vielä nopeampaa. Parhaassa tapauksessa VA kykenisi perustelemaan testeihin tekemänsä muutokset keräämänsä datan pohjalta. Käyttöliittymä tarjoaisi molemminsuuntaista tiedonvälitystä, jonka avulla VA voisi kerätä ja yhdistää testauksesta muodostettua dataa testaajien syöttämän datan kanssa ja tehdä alustavia laskuja kuten virheherkkyyden laskemista.

5 Pohdintaa

Tutkielman päämääränä oli tutkia tapoja hyödyntää tekoälyä ohjelmistotestauksessa ja kerätyn tiedon pohjalta perustella tekoälyn hyödyntämisen tärkeyttä. Tutkimusaiheeni valintaan vaikutti oma alustava mielenkiintoni tekoälyä kohtaan, sekä vahva uskoni sen kykyyn nopeuttaa mitä tahansa, mihin sitä on mahdollista soveltaa. Lähtökohtainen hypoteesini olikin, että tekoälyä käyttämällä voidaan lyhentää huomattavasti ohjelmistotestaukseen käytettävää aikaa.

Houranin ja muiden (2019) konferenssipaperi oli rakennettu hyödyntämällä suosituimpia aiheeseen liittyviä tutkimuksia. Näistä tutkimuksista nostettiin 22 esimerkkiä erilaisista tavoista käyttää tekoälyalgoritmeja tai -tekniikoita ohjelmistotestauksessa. Esimerkit osoittivat tekoälyn monipuolisia käyttömahdollisuuksia ohjelmistotestauksen eri vaiheissa aina virheherkkyyden tunnistamisesta regressiotestien valintaan asti. Tutkijana olin positiivisesti yllättynyt, koska käyttömahdollisuuksien määrä ylitti odotukseni, joiden oletin jo ennestään olevan korkeat.

Tutkimusmateriaalia valitessani en eettisten velvoitteiden takia tietenkään valinnut artikkeleita sen mukaan oliko niiden tulos tekoälyn käyttöön liittyen positiivinen vai negatiivinen. Tästä syystä kirjallisuuskatsaukseni sisälsi myös kritiikkiä tekoälyn sovelluksia kohtaan. Näistä, tuloksiltaan negatiivisista tutkimuksista, tärkein oli mielestäni Hallin ja Bowesin (2012) tutkimus virheentunnistusalgoritmien laadusta. Tutkimustaan varten Hall ja Bowes olivat määritelleet kymmenen toimenpidettä, jotka

virheentunnistusalgoritmin pitäisi suorittaa luotettavien tuloksien saamiseksi. Neljän ja puolen pisteen keskiarvo tutkimuksen 21 algoritmille, testissä jonka maksimi pistemäärä on kymmenen, ei herätä luottamusta tämänhetkisiä algoritmien suunnittelijoita kohtaan.

Kingin ja muiden (2019) tekemän kyselyn perusteella tekoäly ja sen mahdollistamat edut testaukseen liittyen edelleen kiinnostavat IT-alan osaajia. Suurin osa kyselyyn osallistuneista ihmisistä ei ollut juurikaan perehtynyt tekoälyyn. Tästä huolimatta yleinen mielipide oli, että tekoäly tulee vaikuttamaan vahvasti niin manuaaliseen, kuin automatisoituun testaamiseen vuonna 2020 ellei aiemmin. Uskon, että tällaiset mielikuvat ja kiinnostus tekoälyä kohtaan lisäävät osaltaan myös halua kehittää tekoälyalgoritmeja eteenpäin. Siksi tämän tutkielman aiheen uudelleenkäsitteleminen lähitulevaisuudessa olisi tarpeellista.

Vaikka tekoälyn soveltamisen vahvuuksia on esitelty tutkielmassa laajasti, lopullinen kirjallisuuskatsaus on ollut puutteellinen. Käyttämällä mielikuvituksellisempia hakusanoja ja tutkimalla useampia tietokantoja, olisi tutkielmaan voitu löytää monipuolisempia lähteitä. On hyvin mahdollista, että on olemassa tutkimuksia liittyen tekoälyyn liittyvään koneoppimiseen ja sen hyödyntämiseen ohjelmistotestauksessa, joiden tiivistelmässä ei korosteta ”tekoälyn” tai ”ohjelmiston” osuutta. Lisäksi tekoälyn käyttöä olisi voitu tutkia monipuolisemmin myös ohjelmistotestauksen ulkopuolella. Viimeiseksi on hyvä huomioida, että tietokantoja käytiin läpi tietyssä järjestyksessä. Tietokantoja kannattaisi tutkia samanaikaisesti, jottei olisi riskiä, että jokin niistä korostuu tutkimuksessa enemmän kuin muut. IEEE -tietokannan osuus saattoi jäädä tutkielmassa liian suureksi juuri siksi, että sitä tutkittiin ensimmäisenä.

6 Yhteenveto

Tekoälyä käyttömahdollisuudet levittäytyvät ohjelmistotestauksen koko elinkaarelle. Jo ennen ensimmäisiä testejä, tekoäly voidaan valjastaa etsimään ohjelmistosta virheherkkiä alueita, jotta testien pääpaino osataan sijoittaa niiden mukaisesti. Kun testit on valittu ja niiden suorittamiseen tarkoitetut skriptit on kirjoitettu, tekoälyn avulla voidaan itse testien ajamistakin automatisoida. Ajetuista testeistä kerätyn datan pohjalta tekoälyä käyttämällä voidaan automaattisesti rakentaa tai muokata tulevia testejä. Myös testien onnistumista voidaan seurata ja tuloksia kategorisoida tekoälyn avulla. Kun testi on ajettu ja ohjelmistoa on sen pohjalta muutettu, tekoäly mahdollistaa nopean tavan löytää kaikki osa-alueet joihin muutos on vaikuttanut. Näin voidaan valikoida, mitkä testit on syytä ajaa ja mitkä voidaan jättää välistä.

Tekoäly vähentää siis huomattavasti testaukseen käytettävää aikaa testauksen monessa eri vaiheessa. Oikein opetettuna ja kalibroituna se voi yltää yhtä tarkkoihin tuloksiin kuin ihmistestaaja. Tekoälyn varassa toimivien virheentunnistusohjelmien tarkkuus ei ole noussut tasaisesti, mutta aiheeseen liittyvä kiinnostus IT-alalla saattaa

muuttaa asian lähitulevaisuudessa. Vaikka tekoälyn sovelluksissa on puutteita, sen käytön lopettaminen ohjelmistotestauksessa johtaisi testaukseen käytettävien budjettien kasvamiseen, ohjelmistojen valmistumisajan venymiseen ja yleisesti kehityksen hidastumiseen. Tekoäly on korvaamaton työkalu rutiininomaisten tehtävien automatisoimiseen, niiden suorittamisen nopeuttamiseen sekä valtavien datamassojen analysoimiseen.

Lähdeluettelo

Abraham, K., & Horst, B. (2004). Artificial intelligence methods in software testing (Vol. 56). World Scientific.

Agarwal, D., Tamir, D. E., Last, M., & Kandel, A. (2012). A comparative study of artificial neural networks and info-fuzzy networks as automated oracles in software testing. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans, 42(5)*, 1183-1193. https://www.doi.org/10.1109/TSMCA.2012.2183590

Arora, I., Tetarwal, V., & Saha, A. (2015). Open issues in software defect prediction. *Procedia Computer Science*, *46*, (pp. 906-912). https://doi.org/10.1016/j.procs.2015.02.161

Briand, L. C. (2008). Novel applications of machine learning in software testing. *2008 The Eighth International Conference on Quality Software* (pp. 3-10). IEEE. https://doi.org/10.1109/QSIC.2008.29

Chandra, K., Kapoor, G., Kohli, R., & Gupta, A. (2016). Improving software quality using machine learning. *2016 International Conference on Innovation and Challenges in Cyber Security (ICICCS-INBUSH)* (pp. 115-118). IEEE. https://www.doi.org/10.1109/ICICCS.2016.7542340

Chui, M., Manyika, J., & Miremadi, M. (2015). Four fundamentals of workplace automation. *McKinsey Quarterly*, *29*(3), 1-9. https://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/four-fundamentals-of-workplace-automation

Computer Notes (2020). *Software Engineering*. Tietosivusto joka tarjoaa vastauksia teknologiaan liittyviin kysymyksiin. http://ecomputernotes.com/software-engineering/what-are-test-oracles
(haettu 19.03.2020)

Hall, T., & Bowes, D. (2012). The state of machine learning methodology in software fault prediction. *2012 11th International Conference on Machine Learning and*

Applications (Vol. 2, pp. 308-313). IEEE. https://www.doi.org/10.1109/ICMLA.2012.226

Hourani, H., Hammad, A., & Lafi, M. (2019). The Impact of Artificial Intelligence on Software Testing. *2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)* (pp. 565-570). IEEE. https://www.doi.org/10.1109/JEEIT.2019.8717439

Itkin, I., Novikov, A., & Yavorskiy, R. (2019). Development of Intelligent Virtual Assistant for Software Testing Team. *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)* (s. 126-129). IEEE. https://www.doi.org/10.1109/QRS-C.2019.00036

King, T. M., Arbon, J., Santiago, D., Adamo, D., Chin, W., & Shanmugam, R. (2019). AI for testing today and tomorrow: industry perspectives. *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)* (pp. 81-88). IEEE. https://www.doi.org/10.1109/AITest.2019.000-3

Marijan, D. (2016). Improving configurable software testing with statistical test selection. *Proceedings of the International Workshop on Formal Methods for Analysis of Business Systems* (pp. 5-8). https://www.doi.org/10.1145/2975941.2975942

Ostrand, T. J. & Balcer, M. J. (1988). The category-partition method for specifying and generating fuctional tests. *Communications of the ACM*, *31*(6), 676-686. *https://doi.org/10.1145/62959.62964*

Rauf, A. & Alanazi, M. N. (2014). Using artificial intelligence to automatically test GUI. *2014 9th International Conference on Computer Science & Education* (pp. 3-5). https://www.doi.org/10.1109/ICCSE.2014.6926420

Roper, M. (2019). Using machine learning to classify test outcomes. *IEEE International Conference On Artificial Intelligence Testing (AITest)* (pp. 99-100). IEEE. https://www.doi.org/10.1109/AITest.2019.00009

Techopedia (2020a). *Sanakirja.* Selitys termille ”fuzzy logic”. https://www.techopedia.com/definition/1809/fuzzy-logic (haettu 19.03.2020)

Techopedia (2020b). *Sanakirja.* Selitys termille ”machine learning”. https://www.techopedia.com/definition/8181/machine-learning (haettu 10.04.2020)

Techopedia (2020c). *Sanakirja.* Selitys termille ”markov chain”. https://www.techopedia.com/definition/8249/markov-chain

(haettu 13.04.2020)

Whittaker, J. A. (2000). What is software testing? And why is it so hard?. *IEEE software*, *17*(1), (pp.70-79). https://doi-org.libproxy.tuni.fi/10.1109/52.819971.

Yadav, D. K., & Dutta, S. (2016). Test case prioritization technique based on early fault detection using fuzzy logic. *2016 3rd international conference on computing for sustainable global development (INDIACom)* (pp. 1033-1036). https://ieeexplore-ieee-org.libproxy.tuni.fi/document/7724418

Zhu, H. (2018). Software testing as a problem of machine learning: towards a foundation on computational learning theory. *Proceedings of the 13th International Workshop on Automation of Software Test* (pp. 1-1). https://ieeexplore-ieee-org.libproxy.tuni.fi/document/8536340