VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
PROGRAMŲ SISTEMŲ KATEDRA

Neuroninių tinklų panaudojimas testavimo proceso gerinimui

Using neural networks to improve the testing process

Bakalauro darbas

Atliko:	Ričardas Mikelionis	(parašas)
Darbo vadovas:	asist. dr. Vytautas Valaitis	(parašas)
Darbo recenzentas:	partn. prof., dr. Aldas Glemža	(parašas)

Vilnius – 2018

Santrauka

Glaustai aprašomas darbo turinys: pristatoma nagrinėta problema ir padarytos išvados. Santraukos apimtis ne didesnė nei 0,5 puslapio. Santraukų gale nurodomi darbo raktiniai žodžiai.

Raktiniai žodžiai: raktinis žodis 1, raktinis žodis 2, raktinis žodis 3, raktinis žodis 4, raktinis žodis 5

Summary

Santrauka anglų kalba. Santraukos apimtis ne didesnė nei 0,5 puslapio.

Keywords: keyword 1, keyword 2, keyword 3, keyword 4, keyword 5

TURINYS

ĮVADAS 4

1. PROGRAMINIO KODO SUDĖTINGUMO IR DEFEKTYVUMO SĄSAJOS 5
   1.1. McCabe sudėtingumo metrikos 5
   1.2. Halstead sudėtingumo metrikos 5

2. SUDĖTINGUMO METRIKŲ PANAUDOJIMAS APRAŠANT PROGRAMINĖS ĮRANGOS DEFEKTYVUMĄ NUSPĖJANČIĄ BESIMOKANČIĄ MAŠINĄ 6
   2.1. Sprendimų medžio metodu paremta besimokanti mašina 6
   2.2. Giliuoju neuroniniu tinklu paremta besimokanti mašina 6

3. DUOMENŲ ATRINKIMAS 7
   3.1. Rankiniu būdu ieškant atributų kurie yra tiesiogiai proporcingi kitiems atributams 7
   3.2. Duomenų analizė naudojant įrankį Weka 7

4. NEURONINIO TINKLO EFEKTYVUMO GERINIMAS NAUDOJANT ATRINKTUS DUOMENIS 8

REZULTATAI IR IŠVADOS 9

SANTRUMPOS 10

PRIEDAI 10
   1 priedas. Niauroninio tinklo struktūra 11
   2 priedas. Eksperimentinio palyginimo rezultatai 12

Įvadas

Smarkiai augant Continuous Delivery principų populiarumui auga poreikis testuoti daugiau per trumpesnį laiko tarpą. Paprastas sprendimas šiai problemai būtų be abejo nuolat besisukantis automatinių testų paketas, tačiau automatiniais testais padengti programinį kodą 100% praktiškai neįmanoma. 100% padengimas įmanomas tik pagal kokią nors specifinę metriką, o vaikytis tikrojo 100% padengtumo testais pagal kiekvieną metriką ar funkcinę sritį prilygsta šviesos geičio vaikymuisi, kuo arčiau esame tikslo tuo daugiau pastangų ir resursų reikia pasistūmėti į priekį. Todėl, žinoma, vis dar išlieka poreikis rankiniam testavimui. Norint išleisti programinės įrangos versiją kuo dažniau ištestuoti visko kiekvieną kartą neįmanoma. Taip šis problemos sprendimas iškelia dar vieną, mažesnę, problemą: kaip efektyviai pasirinkti testavimo sritis kiekvienai naujai programos laidai (angl. release). Alan M. Davis [**Davis:1995:PSD:203406**] tegia, jog pareto principą galima pritaikyti ir programinės įrangos testavime: čia 80% kode esančių defektų surandami 20% viso kodo.

Tobulame pasaulyje iteracijai pasirinktos testavimo sritys ir apims tuos 20% problematiškojo kodo. Tačiau dažnai net remiantis visa turima istorine informacija testuotojui atsakingam už testavimo sričių parinkimą yra sunku efektyviai atrinkti testavimo sritis, kuriose atliktas darbas turės didžiausią įtaką programinės įrangos kokybei.

Sieniant sėkmingai įgyvendinti darbo tikslą siekiama įgyvendinti šiuos uždavinius:

1. Išrinkti metrikas darančias įtaką programinės įrangos sudėtingumui
2. Palyginti efektyvumą tarp sprendimų medžio besimokančios mašinos ir neuroninio tinklo naudojant surinktus duomenis
3. Atrinkti duomenis turinčius mažiausią įtaką galutiniam spėjimui
4. Naudojantis mažesniu atributų kiekiu iš naujo apmokyti neuroninį tinklą ir palyginti rezultatus su pilnų duomenų rinkinių apmokyto neuroninio tinklo

1. Programinio kodo sudėtingumo ir defektyvumo sąsajos

Tiek Halstead tiek McCabe aprašė kodo sudėtingumo metrikas. Ryšys tarp kodo sudėtingumo bei defektų buvimo jame, o gal būt derėtų sakyti defektų buvimo galimybės kode, atrodytų, gana logiškas: kuo sudėtingesnis kodas, tuo sunkiau jį skaityti, kuo sunkiau kodas skaitomas tuo sunkiau jį plėsti, todėl kyla rizika defektų atsiradimui. Tokia prielaida tiriama nuo pat sudėtingumo metrikų aprašymo. Žinoma, dėl kodo sudėtingumo tiksliai defektų kiekui daromo efekto kyla nestuarimų, vieni teigia, jog Halstead metrikos neturi jokio įrodomo ryšio su defektų buvimu, tačiau yra ir tyrimų pagrindžiančių kodo sudėtingumo bei defektų buvimo jame koreliaciją [**Schroeder1999APG**]. Dažniausiai tyrėjai lieka šio argumento viduryje nei visiškai neigdami, nei patvirtindami šią koreliaciją. Gana dažnas sprendimas yra rinktis ir patikrinti metrikas priklausomai nuo situacijos, taip užtikrinant, kad pasirinktosios metrikos tikrai korelioja su tyrimo metu ieškomu fenomenu, šiuo atveju kodo defektyvumu [**Metrics in Evaluating Software Defects:2013**].

Šio tyrimo kontekste pradėsime nuo visų metrikų naudojimo, toliau siekiant efektyvesnio neuroninio tinklo apmokymo rezultatų bus atrenkami duomenys pašalinantys mažiausią įtaką ieškomam rezultatui daranyts.

1.1. McCabe sudėtingumo metrikos

1.2. Halstead sudėtingumo metrikos

2. Sudėtingumo metrikų panaudojimas aprašant programinės įrangos defektyvumą nuspėjančią besimokančią mašiną

2.1. Sprendimų medžio metodu paremta besimokanti mašina

2.2. Giliuoju neuroniniu tinklu paremta besimokanti mašina

3. Duomenų atrinkimas

3.1. Rankiniu būdu ieškant atributų kurie yra tiesiogiai proporcingi kitiems atributams

3.2. Duomenų analizė naudojant įrankį Weka

4. Neuroninio tinklo efektyvumo gerinimas naudojant atrinktus duomenis

Rezultatai ir išvados

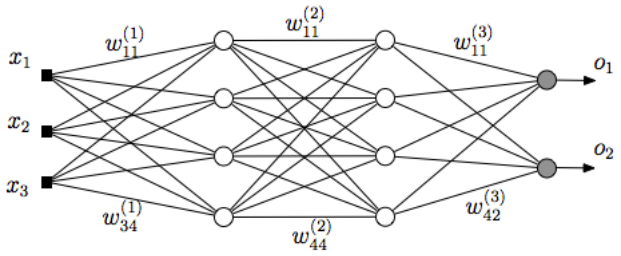
Rezultatų ir išvadų dalyje išdėstomi pagrindiniai darbo rezultatai (kažkas išanalizuota, kažkas sukurta, kažkas įdiegta), toliau pateikiamos išvados (daromi nagrinėtų problemų sprendimo metodų palyginimai, siūlomos rekomendacijos, akcentuojamos naujovės). Rezultatai ir išvados pateikiami sunumeruotų (gali būti hierarchiniai) sąrašų pavidalu. Darbo rezultatai turi atitikti darbo tikslą.

Santrumpos

Sąvokų apibrėžimai ir santrumpų sąrašas sudaromas tada, kai darbo tekste vartojami specialūs paaiškinimo reikalaujantys terminai ir rečiau sutinkamos santrumpos.

Priedas nr. 1

Niauroninio tinklo struktūra



1 pav. Paveikslėlio pavyzdys

Priedas nr. 2

Eksperimentinio palyginimo rezultatai

1 lentelė. Lentelės pavyzdys

Algoritmas	$\bar{x}$	σ^2
Algoritmas A	1.6335	0.5584
Algoritmas B	1.7395	0.5647