VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS KATEDRA

Kodo probleminių vietų aptikimas naudojant automatinį kodo savybių mokymąsi

Code Smells Detection Using Automatic Code Features Learning

Magistro baigiamasis darbas

Atliko:	Dariuš Butkevičius	(parašas)
Darbo vadovas:	Dr. Rimantas Kybartas	(parašas)
Darbo recenzentas:	Dr. Valdas Dičiūnas	(parašas)

Vilnius – 2019

Santrauka

Egzistuojantys sprendimai kodo probleminių vietų aptikimui reikalauja rankinio išeities kodo metrikų parinkimo kiekvienai naujai programų sistemai, tai apsunkina to paties sprendimo taikymą skirtingoms programų sistemoms. Šio darbo tikslas yra pasiūlyti automatinio mokymosi (angl. *machine learning*) modelį, kuris be rankinio metrikų parinkimo, gebėtų aptiki kodo problemines vietas pagal automatiškai ištrauktas išeities kodo sintaksines ir semantines savybes. Išeities kodo savybių išgavimui darbe apmokytas ilgalaikės trumpalaikės atminties dirbtinis neuroninis tinklas (ITA DNT, angl. *long short-term artificial neural network*). Taip pat išeities kodo savybių ištraukimui bandytas iš anksto apmokytas dėmesio dirbtinis neuroninis tinklas *Code2Vec*. Pasitelkiant išmoktų žinių perdavimo techniką (angl. *transfer learning*) buvo sudarytas modelis iš ITA DNT apmokyto išeities kodo savybių ištraukimui ir tiesioginio sklidimo dirbtinio neuroninio tinklo kodo probleminių vietų klasifikavimui. Darbe pasiūlyto modelio, su automatiniu savybių išmokymu, didelės klasės (angl. *large class*), duomenų klasės (angl. *data class*) ir ilgo metodo (angl. *long method*) kodo probleminių vietų aptikimo rezultatai yra pakankamai artimi egzistuojančio modelio su rankiniu kodo savybių (metrikų) parinkinėjimu rezultatams, todėl šio darbo modelis be papildomų pakeitimų gali efektyviai aptikti kodo problemines vietas naujuose projektuose.

Raktiniai žodžiai: kodo probleminė vieta, didelė klasė, duomenų klasė, ilgas metodas, automatinis mokymasis, išmoktų žinių perdavimo technika, ilgalaikės trumpalaikės atminties dirbtinis neuroninis tinklas, dėmesio dirbtinis neuroninis tinklas.

Summary

Existing code smells detection solutions rely on software metrics handcrafting for every system, therefore, it is not possible to easily apply the same solution for different software systems. The main objective of this work is to propose a machine-learning model that is able to detect code smells using automatically detected source code syntactic and semantic features. External pre-trained attention neural network *Code2Vec* and long short-term memory recurrent neural network (LSTM) were tried for source code features extraction. In this work, a transfer learning based model was proposed which uses LSTM for source code features extraction and feed forward neural network for code smells classification. The proposed model with automatic features learning showed only slightly worse results in classification of large class, data class and long method smells than a model with handcrafted features. Thus, the proposed model can be applied for different software systems without additional changes.

Keywords: code smell, large class, data class, long method, machine learning, transfer learning, long short term memory recurrent neural network, attention neural network.

Turinys

ĮVADAS 5

1. KODO PROBLEMINĖS VIETOS 8

1.1. Kodo probleminių vietų poaibio ir jų apibrėžimų parinkimas 8

1.2. Metrikos aptiktų kodo probleminių vietų matavimui 11

2. DIRBTINIAI NEURONINIAI TINKLAI 13

2.1. Dirbtiniai rekurentiniai neuroniniai tinklai 14

2.1.1. Ilgalaikės trumpalaikės atminties dirbtinis neuroninis tinklas 15

2.2. Dėmesio mechanizmas skirtas DNT 17

2.3. Išmoktų žinių perdavimo technika 18

3. KODO PROBLEMINIŲ VIETŲ APTIKIMO STRATEGIJOS 20

3.1. Aptikimas naudojantis taisyklėmis 20

3.2. Aptikimas euristinių paieškos algoritmų pagalba 22

3.3. Aptikimas automatinio mokymosi pagalba 24

3.4. Aptikimas pagal kodo struktūrą 26

3.4.1. Išeities kodo reprezentavimas abstrakčios sintaksės medžio keliais 27

4. KODO PROBLEMINIŲ VIETŲ APTIKIMO MODELIS 29

4.1. Duomenų aibės paruošimas 29

4.2. Kodo savybių ištraukimas 31

4.3. Kodo probleminių vietų klasifikavimas 35

4.4. Modelių mokymas 36

4.5. Įgyvendinimo detalės 39

5. KODO PROBLEMINIŲ VIETŲ APTIKIMO VERTINIMAS 41

REZULTATAI IR IŠVADOS 46

ŠALTINIAI 47

Įvadas

Programinės įrangos palaikymo darbai yra svarbūs tiek kūrimo procese, tiek priežiūros procese. Robert L. Glass darbe „Frequently Forgotten Fundamental Facts about Software Engineering“ tvirtina, kad dažniausiai palaikymo darbai išnaudoja nuo 40% iki 80% procentų programinės įrangos vystymui suplanuoto biudžeto [Gla01]. Taip pat Troy Pearse darbe „Maintainability Measurements on Industrial Source Code Maintenance Activities“ pateikia, kad HP įmonė išleidžia nuo 40% iki 60% nuo vystymui skiriamų lėšų su programinės įrangos palaikymu susijusioms veikloms [PO95]. Dėl programinės įrangos palaikymo svarbos ilgalaikiuose ir dideliuose programinės įrangos vystymo projektuose paprastai skiriamas dėmesys kodo palaikomumui (angl. *maintainability*) ir skaitomumui (angl. *readability*). Tai yra sistemos kūrimo fazėje skiriamas papildomas laikas vidinėms kodo peržiūroms (angl. *internal code reviews*), naudojami statinės kodo analizės įrankiai (angl. *static code analysis tools*), galimai užsakomos išorinės sistemos kodo peržiūros (angl. *external code reviews*). Visų išvardintų priemonių tikslas yra sumažinti programinės įrangos palaikymo kaštus.

Kodo peržiūros metu kitas programuotojas nei pakeitimo autorius patikrina pakeitimo atitikimą komandos viduje sutartiems kodo standartams, bendrai architektūrinei sistemos vizijai, komandos pasirinktoms gerosioms praktikoms. Tai leidžia aptikti dalį problemų ankstyvoje kūrimo fazėje, dėl to išvengti dalies pakeitimų ir defektų vėlesnėje kūrimo arba priežiūros fazėse [ABL89, AFE84]. Mažiau pakeitimų ir defektų vėlesnėse programinės įrangos gyvavimo ciklo fazėse sąlygoja mažesnes išlaidas programinės įrangos palaikymui. Pagrindinis kodo peržiūrų trūkumas yra jų kaina, nes peržiūros dažniausiai reikalauja labiau patyrusių, brangiai apmokamų specialistų darbo laiko. Darant kodo peržiūras didelėms sistemos dalims, kodo peržiūros ir su ja susijusios diskusijos gali užtrukti ženklų laiko tarpą.

Dėl kodo peržiūrų riboto pritaikomumo dideliuose programų sistemų kūrimo projektuose kartu su kodo peržiūromis taikomi ir statinės kodo analizės įrankiai (SKAĮ, angl. *static code analysis tools*). Statinė kodo analizė – tai programos analizės procesas jos nepaleidžiant. SKAĮ palengvina kodo analizę programuotojui, pateikdama taisytinų vietų sąrašą. Labai didelio masto projektuose SKAĮ gali būti vienintelė įmanoma kodo peržiūros dalinė alternatyva. Pavyzdžiui, didelio atviro kodo (angl. *open source*) projekto atveju SKAĮ galėtų atfiltruoti netvarkingus, netinkamus siūlomus išeities kodo pakeitimus (angl. *pull requests*). Tačiau SKAĮ dažnai taiko aproksimacijos schemas, nes pilna paieška nėra įmanoma dideliose sistemose [KE03]. Pagal atliktus empirinius tyrimus nuo 30% iki 100% aptiktų taisytinų vietų yra klaidingai teigiami (angl. *false positive*) įspėjimai. Pagal

kitą empirinį tyrimą SKAĮ gali aptikti apie 40 taisytinų vietų per 1 tūkstantį eilučių [KE03]. Vidutiniško dydžio projekte (50 tūkstančių eilučių) tai būtų apie 2000 taisytinų vietų. Remiantis dviejų empirinių tyrimų rezultatais mažiausiai 600 įspėjimų būtų klaidingi. Programuotojas susidūręs su keliolika klaidingų įspėjimų gali prarasti pasitikėjimą SKAĮ rezultatais ir nekreipti dėmesio net į teisingus įspėjimus. Turint tūkstančius taisytinų vietų gali nepakakti ir laiko visų įspėjimų peržiūrėjimui. Todėl SKAĮ rezultatų gerinimas yra aktuali problema analizuojant didelių sistemų išeities kodus.

SKAĮ įspėjimus skirsto į problemines kodo vietas (angl. *code smell*), projektavimo problemines vietas (angl. *design smell*), galimas saugumo spragas kode (angl. *vulnerability*). Šiame darbe bus fokusuojamasi į probleminių kodo vietų aptikimą. Pavyzdžiui, tokie SKAĮ kaip IPlasma [MMM+05], Fluid Tool [Non12] geba aptikti tam tikras kodo, projektavimos problemines vietas. 1998 metais Brown knygoje „Anti patterns: Refactoring Software, Architectures, and Projects in Crisis" pristatė 40 neigiamų projektavimo ir kūrimo šablonų (angl. *anti-pattern*) [BMM+98]. Neigiamas projektavimo šablonas – tai neoptimalus, blogas sprendimas pasikartojančiai projektavimo problemai [BMM+98]. Pavyzdžiui, viską apimanti klasė (angl. *God class* arba *Blob*) yra neigiamas sprendimas palaikymo prasme bandant įgyvendinti buhalterinės sistemos apskaitos verslo logiką. Viską apimanti klasė – tai didelė klasė su daug metodų mažo susietumo, kuri valdo daug duomenų iš kitų klasių. Coplien neigiamą šabloną dar apibrėžė kaip patrauklų sprendimą iš pirmo žvilgsnio, tačiau turintį daug neigiamų pasekmių vėlesniame programų sistemos kūrimo etape [Cop04]. Todėl svarbu yra kuo anksčiau aptikti neigiamą šabloną. 1999 metais Martin Fowler kartu su Kent Beck knygoje „Refactoring: Improving the design of existing code" pristatė 22 kodo problemines vietas [FBB+99]. Kodo probleminė vieta – tai indikatorius apie galimą gilesnę problemą kode, kuri gali vesti prie defektų ar sudėtingesnio sistemos plėtimo ateityje [FBB+99]. Viena iš gilesnių problemų kode yra pritaikytas neigiamas projektavimo šablonas arba dar kitaip projektavimo probleminė vieta. Tad neigiamą šabloną (projektavimo probleminę vietą) galima aptikti per kodo problemines vietas. Pavyzdžiui, vienas iš viską apimančios klasės simptomų yra Fowler apibrėžta kodo probleminė vieta didelė klasė (angl. *large class*). Ankstyvas kodo probleminių vietų aptikimas turi teigiamą įtaką programų sistemos vystymo ir palaikymo išlaidų mažinimui [Pre01].

Šiame darbe bus analizuojama ilgalaikės trumpalaikės atminties dirbtinio neuroninio tinklo (ITA DNT, angl. *long short-term artificial neural network*) [HS97] ir dėmesio DNT *Code2Vec* [AZL+18a] (angl. *attention artificial neural network*) taikymo galimybė kodo probleminių vietų

aptikimui. ITA DNT paremti modeliai geba automatiškai išmokti kodo savybes [DTP+17, KSF+17, WTV+16]. *Code2Vec* buvo sukurtas išeities kodo savybių ištraukimui [AZL+18a]. Tokiu būdu siekiama sukurti modelį gebantį aptikti kodo problemines vietas be išankstinio, rankinio programų sistemų metrikų, kurios atspindėtų kodo savybes, parinkinėjimo. Taip pat tikimasi, kad detalios žinios apie programos struktūrą gali padėti sumažinti klaidingai teigiamų SKAĮ įspėjimų skaičių [KSF+17].

Egzistuojančiuose darbuose galima išskirti tris pagrindines kryptis kodo ir projektavimo probleminių vietų aptikimui: pagal aptikimo taisykles [Mar04, MGD+09], naudojant euristines paieškos strategijas [KKS+11, MKM+17], naudojant automatinį mokymąsi (angl. *machine learning*) iš kodą atitinkančių metrikų [MAB+12a, MAB+12b, FZM+13, FZ+17]. Darbai iš visų trijų krypčių palieka programuotojui parinkti svarbių metrikų sąrašą kiekvienam iš projektų atskirai. Tai įveda papildomo rankinio darbo ir eksperimentavimo prieš gebant aptikti kodo problemines vietas. Parinktų metrikų aibė vienai projektų aibei nėra universaliai pritaikoma kitiems projektams ar kitoms kodo ir projektavimo probleminėms vietoms [NPT+18]. Bendrai yra galimas atvejis, kai metrikos yra identiškos kodo vietai turinčiai ir neturinčiai probleminės vietos [DTP+17]. Taip pat pasiekiamas ribotas tikslumas (angl. *precision*) ir atpažintų objektų dalies įvertis (angl. *recall*), kai norima ne tik aptikti kodo problemines vietas, bet, pavyzdžiui, aptiktas kodo problemines vietas suskirstyti dar į 3 grupes pagal prioritetą [FZ+17].

Šio darbo tikslas yra pasiūlyti automatinio mokymosi modelį, kuris be rankinio metrikų parinkimo gebėtų aptikti kodo problemines vietas pagal automatiškai ištrauktas išeities kodo sintaksines ir semantines savybes. Tikslui pasiekti reikia įgyvendinti sekančius uždavinius:

1. Pasirinkti probleminių kodo vietų poaibį ir pasirinkti jų apibrėžimus.
2. Sudaryti arba adaptuoti probleminių kodo vietų duomenų aibę.
3. Paruošti modelį aptinkantį pasirinktas kodo problemines vietas.
4. Palyginti egzistuojančio įrankio, gebančio aptikti tas pačias kodo problemines vietas, su sukurto modelio rezultatais.

Toliau pateikiami laukiami rezultatai įgyvendinus uždavinius:

1. Probleminių kodo vietų duomenų aibė.
2. Modelis kodo probleminių vietų aptikimui pagal kodo struktūrą.

1. Kodo probleminės vietos

Šiame skyriuje pristatomi kodo probleminių vietų apibrėžimai ir parenkama probleminių vietų aibę analizei šiame darbe. Taip pat yra aprašytos kodo probleminių vietų aptikimo sėkmingumą matuojančios metrikos.

1.1. Kodo probleminių vietų poaibio ir jų apibrėžimų parinkimas

Po „Refactoring: Improving the design of existing code“ knygos paskelbimo kodo probleminės vietos buvo nagrinėjamos straipsniuose siekiant pagerinti atskirų kodo probleminių vietų pagal Fowler supratimą ir sukurti įrankius jų aptikimui [ZHB11]. Taip pat buvo nagrinėjamas kodo probleminių vietų neigiamas poveikis sistemos plečiamumui ir poveikis didėjančiai sistemos defekto atsiradimo tikimybei [ZHB11]. Pagal sisteminę 319 straipsnių apie kodo problemines vietas analizę nuo 2000 iki 2009 metų buvo atrinktos daugiausiai dėmesio susilaukusios kodo problemines vietos ir iš jų atrinktos kodo probleminės vietos turinčios empirinių įrodymų apie neigiamą poveikį sistemos defekto atsiradimo tikimybei [ZHB11, LS07]. Tokiu būdu šiame darbe pasirinkta analizuoti didelės klasės, ilgo metodo (angl. *long method*) problemines vietas. Pagal atliktą egzistuojančių kodo probleminių vietų duomenų aibių analizę vienintelė viešai prieinama kodo probleminių vietų duomenų aibė, kurioje yra didelės klasės ir ilgo metodo kodo probleminės vietos buvo Fontana ir jos darbo grupės paruošta kodo probleminių vietų duomenų aibė [FMZ16]. Tai buvo naudinga siekiant palyginti naujai kuriama kodo probleminių vietų aptikimo modelį su egzistuojančiu įrankiu. Fontana ir jos darbo grupės kodo probleminių vietų duomenų aibėje dar buvo duomenų klasės (angl. *data class*) ir pavydaus metodo (angl. *feature envy*) kodo probleminės vietos. Todėl šiame darbe nuspręsta pasirinkti didelės klasės, ilgo metodo, duomenų klasės ir pavydžios funkcijos kodo problemines vietas. Toliau 1 lentelėje pateikiami pasirinktų kodo probleminių vietų apibrėžimai pagal Fowler ir grupės pagal Mäntylä [ML06].

1 lentelė. Pasirinktų kodo probleminių vietų apibrėžimai ir grupės

Kodo probleminė vieta	**Apibrėžimas**	**Grupė**
Duomenų klasė	Klasė turinti tik duomenis (be jokių operacijų su jais).	Perteklinis kodas (angl. *dispensables*)
Pavydus metodas	Metodas pasiima duomenis dažniau iš kitos klasės nei iš klasės, kurioje randasi.	Nereikalinga susietumą didinanti grupė (angl. *couplers*)

Didelė klasė	Klasė prisiima per daug atsakomybių, turi per daug klasės kintamųjų. Tai didina kodo kartojimo, betvarkės ir sudėtingo palaikymo tikimybes.	Supratimą ir palaikymą sunkinanti grupė (angl. *bloaters*)
Ilgas metodas	Ilgi, sudėtingi, su daug atsakomybių metodai, kuriuos sudėtinga suprasti vien perskaičius pavadinimą.	Supratimą ir palaikymą sunkinanti grupė

Duomenų klasė verslo logikos dalyje dažnai reiškia, jog yra kitos klasės manipuliuojančios jos duomenimis [FBB+99]. Tokiu būdu pažeidžiamas klasės logikos susiejimo su duomenimis principas (angl. *tell-don't-ask*), kuris teigia jog operacijos susijusios su duomenimis turi rastis toje pačioje klasėje [Fow13]. Taip pat duomenų klasė gali privesti prie vieno iš objektinio programavimo (angl. *object-oriented programming*) principų – tai inkapsuliacijos (angl. *encapsulation*) pažeidimo. Laikantis inkapsuliacijos, vidinė klasės būsena turi būti paslėpta nuo išorės ir klasės duomenys gali būti pasiekiami, modifikuojami tik per tos pačios klasės pateiktus metodus [PKA08]. 1 paveiksle yra pateiktas potencialios duomenų klasės Java kalba pavyzdys. *Phone* klasė turi savyje tik duomenis be susijusios logikos. Tokiu būdu sudaromas pagrindas klasės logikos susiejimo su duomenimis principo pažeidimo pagrindas.

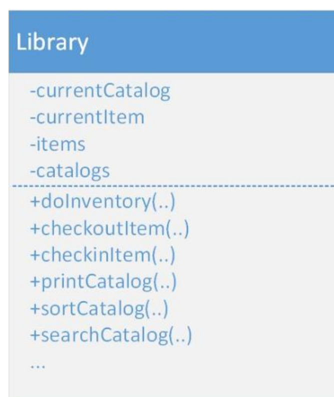
```
public class Phone {
   private String unformattedNumber;

   public Phone(String unformattedNumber) {
      this.unformattedNumber = unformattedNumber;
   }
   public String getAreaCode() {
      return unformattedNumber.substring(0,3);
   }
   public String getPrefix() {
      return unformattedNumber.substring(3,6);
   }
   public String getNumber() {
      return unformattedNumber.substring(6,10);
   }
}
public class Customer {
   private Phone mobilePhone;

   public String getMobilePhoneNumber() {
      return "(" +
         mobilePhone.getAreaCode() + ") " +
         mobilePhone.getPrefix() + "-" +
         mobilePhone.getNumber();
   }
}
```

1 paveikslas. Adaptuoti duomenų klasės ir pavydaus metodo pavyzdžiai [MR15]

Pavydus metodas pagrinde atlieka veiksmus su kitos klasės duomenimis nenaudodamas savo klasės duomenų [FBB+99]. Tokiu būdu įvedamas nereikalingas susietumas tarp klasių [FBB+99].

Pavydaus metodo kodo probleminė vieta yra dažnai aptinkama kartu su duomenų klasės kodo problemine vieta. Metodas, kuris randasi netinkamoje klasėje, atlieka veiksmus pagrinde su duomenų klasės laukais. Tai yra duomenų klasei turėjusi priklausyti logika yra aprašoma išorinėse klasėse. 1 paveiksle *Customer* klasės *getMobilePhoneNumber* metodas yra pavydaus metodo pavyzdys. Metodas *getMobilePhoneNumber* grąžina specialaus formato telefono numerį ir operuoja tik su *Phone* klasės laukais, todėl randantis *Customer* klasėje, jis nereikalingai didina *Customer* susietumą su *Phone* vidinėmis įgyvendinimo detalėmis.

Pagal dalį didelės klasės kodo probleminės vietos apibrėžimo šios probleminės vietos atsiradimui įtakos turi daug klasės laukų, metodų, metodų sudėtingumas [FBB+99]. Tačiau negalima vienareikšmiškai pasirinkti kiek klasės laukų, metodų yra priimtina, nes svarbu taip pat išsiaiškinti kiek atsakomybių prisiima analizuojama klasė. Pavyzdžiui, jei klasė turi tik vieną atsakomybę, tai yra priimtina turėti daug metodų. Pagal Robert Martin vienos atsakomybės principą (angl. *single responsibility principle*) klasė turėtų turėti tik vieną priežastį keistis [Mar05]. 2 paveiksle pateikta supaprastinta *Library* klasės diagrama. *Library* yra didelės klasės kodo probleminės vietos pavyzdys, nes turi daugiau nei vieną atsakomybę, tai yra daugiau nei vieną priežastį keistis. Metodai *checkinItem*, *checkoutItem* ir metodai *printCatalog*, *sortCatalog*, *searchCatalog* turi atskiras priežastis keistis. Pirma grupė metodų yra atsakinga už bibliotekos knygos/žurnalo logiką, o antra metodų grupė yra atsakinga už bibliotekos katalogo logiką.



2 paveikslas. Adaptuotas didelės klasės pavyzdys [Sou]

Ilgo metodo probleminė vieta turi panašius atsiradimo faktorius kaip didelė klasė. Paprastai metodas prisiima per daug atsakomybių ir dėl to yra sunkiai suprantamas. 3 paveiksle yra pavaizduotas ilgo metodo pavyzdys. Metodas prisiima 4 skirtingas atsakomybes, kurios yra sugrupuotos komentarais. Ilgas metodas *doSetup* yra atsakingas už palaikomų įrenginių atfiltravimą, rašymo į žurnalą konfigūravimą, nustatymų ištraukimą, numatytos spalvos nustatymą.

```java
public void doSetup {

    // (1) make sure the code only runs on mac os x
    boolean mrjVersionExists = System.getProperty("mrj.version") != null;
    boolean osNameExists = System.getProperty("os.name").startsWith("Mac OS");

    if (!mrjVersionExists || !osNameExists) {
      System.err.println("Not running on a Mac OS X system.");
      System.exit(1);
    }

    // (2) do all the logfile setup stuff
    int currentLoggingLevel = DEFAULT_LOG_LEVEL;

    File errorFile = new File(ERROR_LOG_FILENAME);
    File warningFile = new File(WARNING_LOG_FILENAME);
    File debugFile = new File(DEBUG_LOG_FILENAME);

    if (errorFile.exists()) {
      currentLoggingLevel = DDLoggerInterface.LOG_ERROR;
    }
    if (warningFile.exists()) {
      currentLoggingLevel = DDLoggerInterface.LOG_WARNING;
    }
    if (debugFile.exists()) {
      currentLoggingLevel = DDLoggerInterface.LOG_DEBUG;
    }

    logger = new DDSimpleLogger(CANON_DEBUG_FILENAME, currentLoggingLevel, true, true);

    // (3, 4) do all the preferences stuff, and get the default color
    preferences = Preferences.userNodeForPackage(this.getClass());
    int r = preferences.getInt(CURTAIN_R, 0);
    int g = preferences.getInt(CURTAIN_G, 0);
    int b = preferences.getInt(CURTAIN_B, 0);
    int a = preferences.getInt(CURTAIN_A, 255);
    currentColor = new Color(r,g,b,a);
}
```

3 paveikslas. Adaptuotas ilgo metodo pavyzdys [Ale18]

1.2. Metrikos aptiktų kodo probleminių vietų matavimui

Susijusiuose darbuose kodo probleminių vietų aptikimo sėkmingumo matavimui naudotos tikslumo, atpažintų objektų dalies, bendro tikslumo (angl. *accuracy*), F įvertis (angl. *F-measure*) metrikos. 2 lentelėje pateikiamos šių metrikų formulės kodo probleminių vietų aptikimo kontekste pagal Maiga [MGD+09].

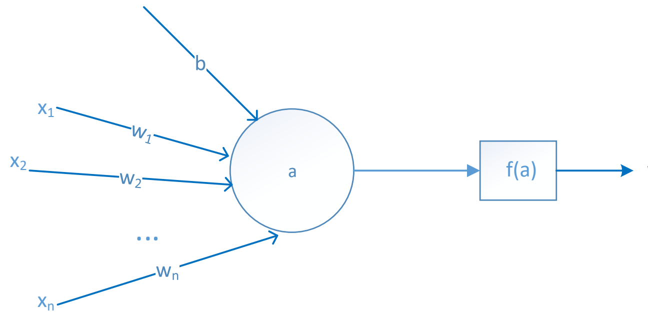
2 lentelė. Metrikos kodo probleminių vietų aptikimo metodo gerumui matuoti

Metrika	**Formulė**
Tikslumas	$\frac{\lvert\{\text{egzistuojančios probleminės vietos}\} \cap \{\text{aptiktos probleminės vietos}\}\rvert}{\lvert\{\text{aptiktos probleminės vietos}\}\rvert}$
Atpažintų objektų dalis	$\frac{\lvert\{\text{egzistuojančios probleminės vietos}\} \cap \{\text{aptiktos probleminės vietos}\}\rvert}{\lvert\{\text{egzistuojančios probleminės vietos}\}\rvert}$
Bendras tikslumas	$\frac{\lvert\{\text{teisingai klasifikuoti kodo elementai}\}\rvert}{\lvert\{\text{nagrinėti kodo elementai (metodai, klasės)}\}\rvert}$

F įvertis	$2 * \frac{\text{tikslumas} * \text{atpažintų objektų dalis}}{\text{tikslumas} + \text{atpažintų objektų dalis}}$

2. Dirbtiniai neuroniniai tinklai

Dirbtinis neuroninis tinklas (DNT, angl. *neural network*) yra automatinio mokymosi modelis, įkvėptas supaprastinto modelio žmogaus smegenų funkcionavimo. DNT susideda iš dirbtinių neuronų, kurio modelis pavaizduotas 4 paveiksle. Dirbtinio neurono išeitis skaičiuojama pagal 1 formulę, kur x_i yra įeitys, w_i išmokstami svoriai, o b išmokstamas slenkstis. Dirbtinio neurono rezultatui pritaikoma aktyvacijos funkcija.



4 paveikslas. Dirbtinio neurono modelis

$$a = \sum_{k=1}^{n} w_k x_k + b \quad (1)$$

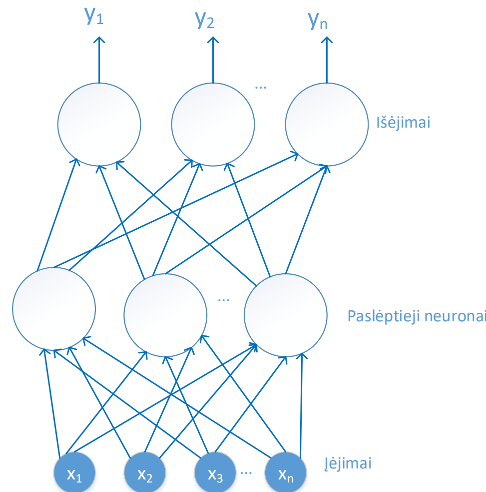
2-4 formulėse pateiktos dažniausios aktyvacijos funkcijos, tai yra sigmoidinė, hiperbolinio tangento, pataisyta tiesinė aktyvacijos funkcija.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \quad (2)$$

$$tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \quad (3)$$

$$ReLU(z) = \max(0, z) \quad (4)$$

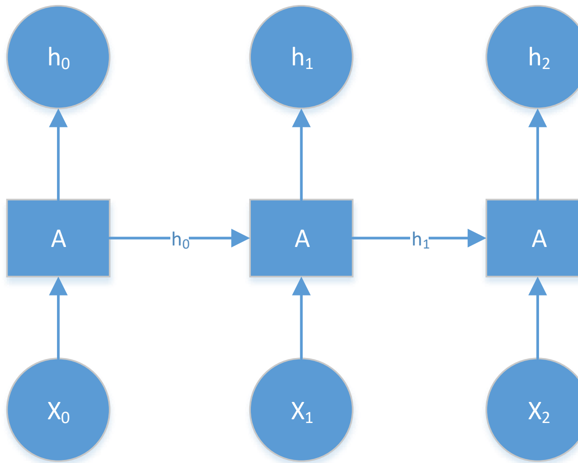
Vienas iš paprasčiausių DNT yra tiesioginio sklidimo DNT (angl. *feed forward neural network*), kuriame galimi dirbtinių neuronų ryšiai tik į priekį, tai yra iš įėjimų į išėjimus. Tiesioginio sklidimo DNT gali būti vienas ar daugiau paslėptų sluoksnių ir visi praeito sluoksnio dirbtiniai neuronai yra sujungti su visais sekančio sluoksnio dirbtiniais neuronais (žiūrėti 5 paveikslą).



5 paveikslas. Tiesioginio sklidimo DNT modelis

2.1. Dirbtiniai rekurentiniai neuroniniai tinklai

Dėl nuoseklaus kodo pobūdžio (angl. *sequential data*) rekurentiniai dirbtiniai neuroniniai tinklai (RDNT, angl. *reccurent artificial neural network*) yra tinkamas variantas kodo analizei, siekiant aptikti kodo ir projektavimo problemines vietas. RDNT – tai vieno sluoksnio DNT pakartotas daug kartų [HBF+01]. Jei tiesioginio sklidimo DNT priima vektorių ir grąžina vektorių, tai RDNT priima vektorių seką ir grąžina vektorių seką. 6 paveiksle parodyta išvynioto RDNT struktūra. RDNT parodytas pirmuose 3 laiko vienetuose. Galima išvynioti priklausomai nuo įeities vektorių skaičiaus. X_i – tai įeities vektorius i-me laiko vienete. Analizuojant kodą X_i reprezentuoja kodo elementą, pavyzdžiui, žodį *catch*. 5 formulė parodo kaip t-jame A bloke apskaičiuojama išeitis, kur b – tai slenksčių vektorius, W_h – tai praeitos išeities svoris (neaktualu pirmame laiko vienete), W_x – tai dabartinio laiko momento įeities svoris. σ – tai netiesinė aktyvacijos funkcija, kuri pritaikoma kiekvienai vektoriaus komponentei. Mokant RDNT nustatinėjami svorių vektoriai W_x ir W_h, kurie yra vienodi kiekviename laiko momente ir apsprendžia kiek yra svarbi praeitis, ir kiek svarbus dabartinis įeities vektorius.

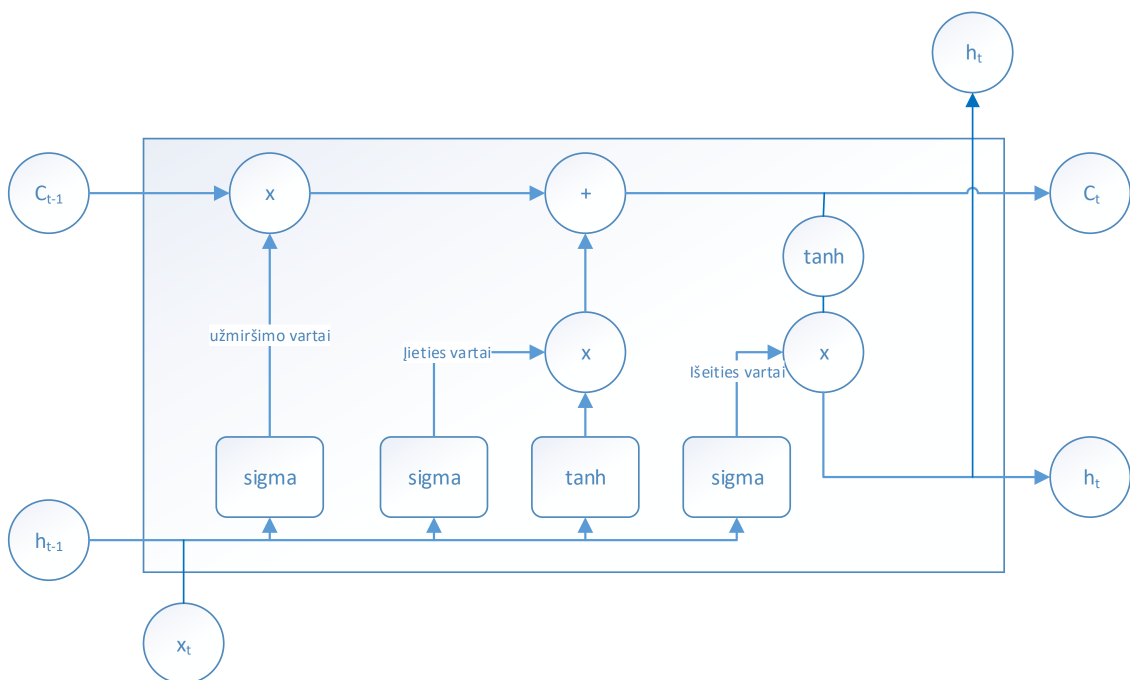


6 paveikslas. Išvynioto laike RDNT struktūra

$$h_t = \sigma(b + W_h h_{t-1} + W_x X_t) \quad (5)$$

2.1.1. Ilgalaikės trumpalaikės atminties dirbtinis neuroninis tinklas

Vienas iš sėkmingiausių RDNT atstovų yra ITA DNT, kuris pirmą kartą pristatytas Sepp Hochreiter ir Jurgen Schmidhuber 1997 metais [HS97]. ITA DNT turi sėkmingus taikymus natūralios kalbos supratime, vertime, laiko eilučių (angl. *time series*) prognozavime [WSC+16, Vog16]. Analizuojant kodą labai svarbu suprasti kodo elementų tarpusavio priklausomybes [DTP+17], kurias ITA DNT geba įžvelgti. 7 paveiksle parodyta ITA DNT vieneto struktūra. Bendroje RDNT struktūroje, kuri pavaizduota 6 paveiksle, vienas ar keletas ITA vienetų formuotų A bloką.



7 paveikslas. Standartinė ITA vieneto struktūra laiko momentu t. Paveikslas adaptuotas iš Cristopher Olah pateiktos versijos [Ola15]

ITA DNT vieneto pagrindinis skirtumas nuo bazinio RDNT vieneto yra ilgalaikės atminties kintamasis C_t, kuris saugo sukauptą kontekstinę informaciją. ITA vieneto rezultatas laiko momentu t apskaičiuojamas atsižvelgiant ne tik į praeito vieneto rezultatą h_{t-1}, kas yra laikoma trumpalaike atmintimi, ir dabartinę įeitį X_t, bet ir atsižvelgiant į kontekstą (ilgalaikę atmintį) C_t. C_t būsena kontroliuojama užmiršimo ir įeities vartais (žiūrėti 7 paveikslą). 6 formulėje pateikta užmiršimo vartų išraiška, kur trumpalaikės atminties h_{t-1} ir X_t vektoriai yra sujungiami (užrašant vieną po kito), W_f yra svorių vektorius, o b_f slenksčių vektorius. Formulėse 6, 7 ir 10 σ žymi sigmoidinę aktyvacijos funkciją, kuri atskirai pritaikoma kiekvienai vektoriaus komponentei. f_t gaunamas vektorius su reikšmėmis nuo 0 iki 1, kur 0 reiškia reikia pilnai pamiršti dalį ilgalaikės atminties, o 1 – tai pilnas išsaugojimas dalies ilgalaikės atminties.

$$f_t = \sigma\left(W_f[h_{t-1}, X_t] + b_f\right) \tag{6}$$

i_t įeities vartai yra apskaičiuojami tuo pačiu principu kaip užmiršimo vartai tik su savais išmokstamais svoriais W_i ir slenksčiais b_i (žiūrėti 7 formulę).

$$i_t = \sigma(W_i[h_{t-1}, X_t] + b_i) \tag{7}$$

ITA vieneto įeitis CU_t apskaičiuojama atsižvelgiant į trumpalaikę atmintį ir dabartinę įeitį pagal išmokstamus svorius W_c ir slenksčius b_c (žiūrėti 8 formulę). Kiekvienai vektoriaus komponentei taikoma hiperbolinio tangento aktyvacijos funkcija sutalpina rezultatus intervale tarp -1 ir 1. Tada

atnaujinama ilgalaikė atmintis C_t užmirštant dalį senos informacijos pagal užmiršimo vartus ir pridedant dalį naujos informacijos CU_t pagal įeities vartus (7 paveikslas, 9 formulė). 9 formulėje +, * reiškia atitinkamų pozicijų vektorių komponenčių sudėjimą ir daugybą (angl. *element-wise*). Ilgalaikė atmintis perduodama sekančiam ITA vienetui.

$$CU_t = tanh(W_c[h_{t-1}, X_t] + b_c) \tag{8}$$

$$C_t = f_t * C_{t-1} + i_t * CU_t \tag{9}$$

Pabaigoje apskaičiuojami išeities vartai panašiai kaip įeities ir užmiršimo su išmokstamais svoriais W_o ir slenksčiais b_o (žiūrėti 10 formulę). Pagal ilgalaikę atmintį, tanh sutalpinta į intervalą nuo -1 iki 1, pritaikius išeities vartus gaunama ITA vieneto išeitis (žiūrėti 11 formulę).

$$o_t = \sigma(W_o[h_{t-1}, X_t] + b_o) \tag{10}$$

$$h_t = o_t * \tanh(C_t) \tag{11}$$

2.2. Dėmesio mechanizmas skirtas DNT

Dėmesio technikų (angl. *attention*) taikymas DNT buvo pristatytas teksto apdorojimo, vertimo problemose [LPM15]. Abstrakti idėja yra išmokti koncentruotis ties maža dalimi įeities duomenų, kuri yra svarbiausia efektyviam uždavinio sprendimui. Toliau yra pristatoma supaprastinta dėmesio technika [RE15] su tiesioginio sklidimo DNT, kuri buvo naudojama šiame darbe (žiūrėti 8 paveikslą). Įeityje yra n m-mačių (1 x m) vektorių H_1, ..., H_n už kurių apjungimą yra atsakingas dėmesio mechanizmas. 8 paveiksle dėmesio mechanizmas pavaizduotas kaip tiesioginio sklidimo DNT, tačiau tai gali būti ir kitas modelis. Toliau darysime prielaidą, kad yra vienas dirbtinis neuronas be aktyvacijos funkcijos su m-mačiu svorių vektoriumi a (1 x m). Tada i-oji išeitis skaičiuojama pagal 12 formulę, rezultate gaunamas q_i yra skaliaras.

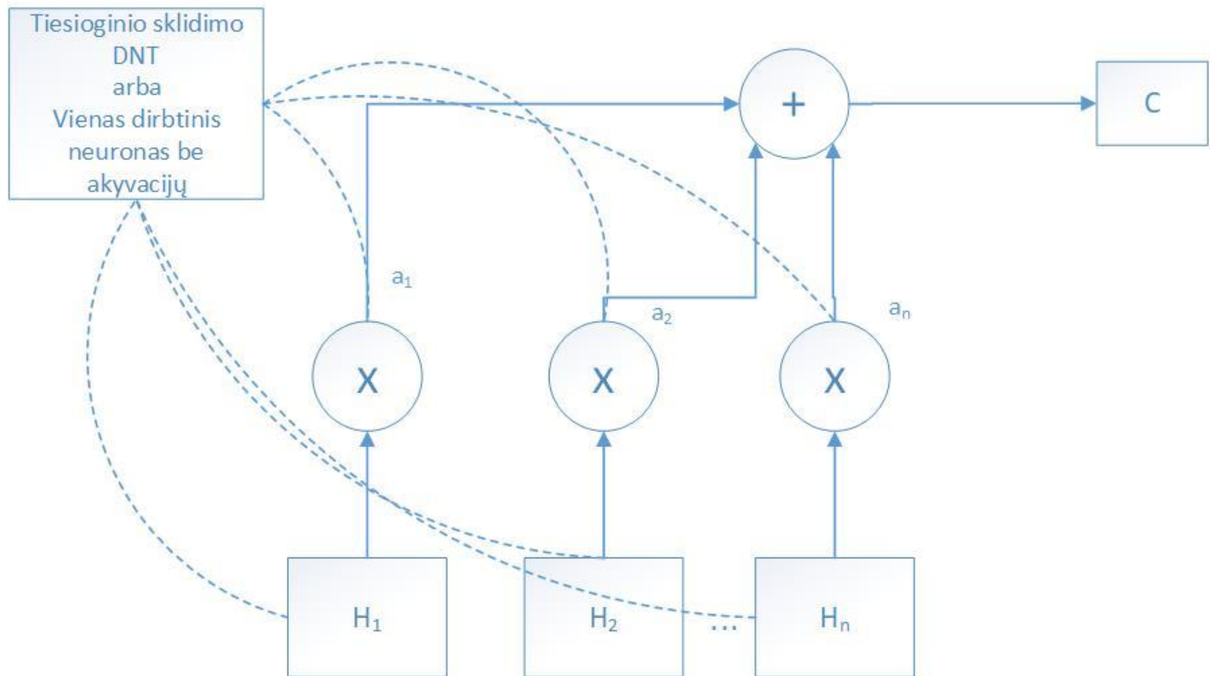
$$q_i = aH_i^t \tag{12}$$

Kiekvienai iš H_i įeičių dėmesio svoris yra apskaičiuojamas naudojant lankstaus maksimumo funkcija (angl. *softmax*). 13 formulėje yra parodytas a_i skaičiavimas. Dėl lankstaus maksimumo taikymo visų svorių suma yra lygi 1.

$$a_i = \frac{e^{q_i}}{\sum_{k=1}^{n} e^{q_k}} \tag{13}$$

Įeities vektoriai yra apjungiami į vieną vektorių, atsižvelgiant į išmoktus svorius a_1, ..., a_n (14 formulė), todėl fokusuojamasi ties maža įeities duomenų dalimi, kuri yra svarbiausia sprendžiamam uždaviniui.

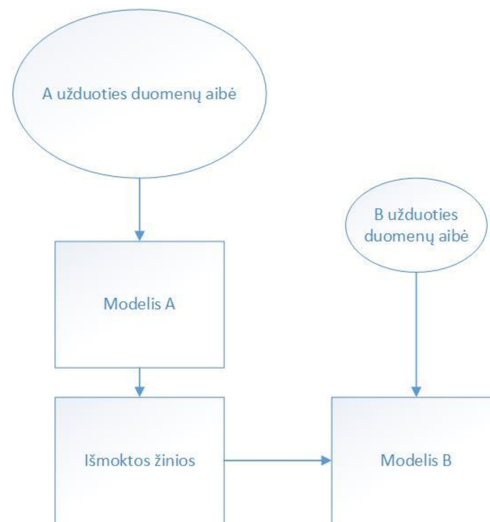
$$C = \sum_{k=1}^{n} a_k H_k \tag{14}$$



8 paveikslas. Dėmesio technika naudojant tiesioginio sklidimo DNT [RE15]

2.3. Išmoktų žinių perdavimo technika

Pirmos išmoktų žinių perdavimo technikos (angl. *transfer learning*) užuominos buvo dar 1995 metais, kai buvo nagrinėjama išmoktų žinių perdavimo technikos motyvacija automatinio mokymosi srityje [Lea95]. Išmoktų žinių perdavimo technika – tai sugebėjimas pritaikyti šaltinio užduotyse išmoktas žinias, sprendžiant esamą užduotį [PY10]. Paprastai sėkmingam išmoktų žinių perdavimo technikos taikymui šaltinio užduotys yra panašios su esama užduotimi ir šaltinio užduočių duomenų aibės yra kur kas didesnės. 9 paveiksle parodyta išmoktų žinių perdavimo technika. Šaltinio užduotis yra žymima A, o esama užduotis yra žymima B. Galima matyti, kad A turi kur kas didesnę duomenų aibę nei B. B modelio apmokymui naudojamos jau išmoktos žinios išmokius A modelį ir B duomenų aibė.



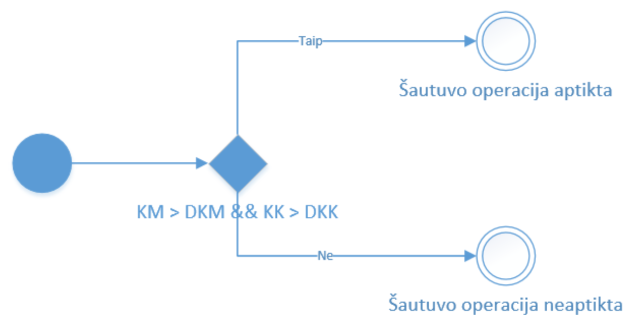
9 paveikslas. Išmoktų žinių perdavimo technikos schema, kuri adaptuota iš [PY10]

3. Kodo probleminių vietų aptikimo strategijos

Didžioji dalis mokslininkų, nagrinėjusių kodo ar projektavimo problemines vietas, siekė aprašyti konkrečių probleminių vietų aptikimo metodus ar sukurti įrankius [ZHB11], nes rankinė paieška užima daug laiko ir turi didelę klaidos tikimybę. Šiame skyriuje apžvelgtos 3 pagrindinės probleminių vietų aptikimo metodų vystymosi kryptys. Tai yra probleminių vietų aptikimas naudojantis iš anksto apibrėžtomis taisyklėmis, euristiniais paieškos algoritmais, automatiniu taisyklių išmokymu. Taip pat yra pateiktas kodo probleminių vietų aptikimo pagal kodo struktūrą kontekstas.

3.1. Aptikimas naudojantis taisyklėmis

Kodo ir projektavimo probleminių vietų aptikimo taisyklės susideda iš požymių ir metrikų apibūdinančių konkrečias kodo vietas. Požymiams parenkamos norimos reikšmės, o metrikoms parenkami slenksčiai. Pavyzdžiui, jei ieškoma klasės lygmens projektavimo probleminės vietos šautuvo operacija (angl. *shotgun surgery*), tai galima taikyti taisyklę, pavaizduota 10 paveiksle, kiekvienam iš klasės metodų. Šautuvo operacija apibūdina klasės lygmens projektavimo probleminę vietą, kai darant pakeitimą probleminėje vietoje, reikia keisti daug kitų klasių [BMM+98]. Klasėje būtų aptikta šautuvo operacijos probleminė vieta, jei bent vienam metodui iš klasės taisyklė būtų teisinga. Taisyklei suformuoti parinktos metrikos KM ir KK, kur KM – tai konkretų metodą kviečiančių metodų skaičius, o KK – tai konkretų metodą kviečiančių metodų nesikartojančių klasių skaičius. DKM ir DKK – tai natūralus skaičiai, slenksčiai parinktoms metrikoms. Požymio pavyzdys būtų ar klasės pavadinimas lygus „Tvarkytojas“. Toks požymis galėtų būti naudingas, bandant surasti viską apimančios klasės projektavimo probleminę vietą.



10 paveikslas. Pavyzdinis šautuvo operacijos probleminės vietos aptikimas

Radu Marinescu aprašė kodo ir projektavimo probleminių vietų aptikimo strategijų formavimo būdą [Mar04]. Pateikiama metodologija kaip iš kodo ir projektavimo probleminių vietų aprašymų nustatyti svarbias metrikas ir parinkti joms slenksčius. Iš nustatytų metrikų ir jų slenksčių suformuojamos aptikimo taisyklės SOD interpretuojamos kalbos pavidalu, kurios taikomos naudojant PRODEOOS įrankį, objektinės sistemos sugeneruotam meta modeliui. Meta modelio generavimas, Marinescu pristatyto PRODEOOS įrankio pagalba, užtikrina galimybę dirbti su skirtingomis objektiškai orientuotomis kalbomis parašytomis sistemomis. Rezultate gaunamos klasės ir metodai, kurie įtariami turintys probleminių vietų. Pristatytas būdas išbandytas su 9 skirtingom probleminėmis vietomis su dviem sistemomis (700 tūkstančių ir 2 milijonų eilučių). Pasiekti bendri tikslumai nuo 50% iki 81%. Pavyzdžiui, šautuvo operacijos ir viską apimančios klasės problemines vietos buvo aptinkamos su 60% tikslumu. Vėliau Marinescu pristatytos kodo ir projektavimo probleminių vietų aptikimo strategijos buvo įgyvendintos IPlasma įrankyje [MMM+05].

Naouel Moha pristatė metodą DECOR kodo ir projektavimo probleminių vietų specifikavimui ir aptikimui [MGD+09]. DECOR metodas susideda iš 5 žingsnių. Iš pradžių programuotojas analizuoja probleminių vietų aprašymus ir sudaro žodyną esminių žodžių apibūdinančių problemines vietas. Tada, naudojantis dalykinei sričiai pritaikyta kalba (angl. *domain specific language, DSL*), programuotojas parengia formalias taisyklių aibes kiekvienai iš probleminių vietų. Iš taisyklių aibių automatiškai generuojami probleminių vietų aptikimo algoritmai. Aptikimo algoritmai yra taikomi sistemos modeliui, kuris automatiškai parengiamas PADL (angl. *Pattern and Abstract-level Description Language*) meta modelio pagalba. PADL – tai nepriklausomas nuo programavimo kalbos meta modelis objektiškai orientuotoms sistemoms reprezentuoti [GA08]. Rezultate gražinamos klasės, kuriose potencialiai yra kodo ar projektavimo probleminė vieta. Paskutiniame žingsnyje grąžintos klasės tikrinamos programuotojo ar iš tikro jose yra probleminių vietų. Metodo privalumas yra galimybė palaipsniui tobulinti probleminių vietų aptikimo taisykles dalykinei sričiai pritaikyta kalba nesigilinant į paieškos logikos detales. Straipsnyje taip pat pateikiama konkreti DECOR metodo realizacija DETEX. DETEX paruoštos aptikimo taisyklės 4 projektavimo probleminėms vietoms: viską apimanti klasė, funkcinė dekompozicija (angl. *functional decomposition*), painus kodas (angl. *spagheti code*), šveicariškas peilis (angl. *swiss army knife*). Klasė su funkcine dekompozicija nenaudoja pakankamai objektinio programavimo privalumų paveldėjimo, polimorfizmo [BMM+98]. Dažniausiai tai yra klasė įgyvendinanti logiką procedūriniu stiliumi [BMM+98]. Klasė su painiu kodu neturi laukų ir turi daug ilgų metodų be parametrų

[BMM+98]. Klasė su šveicariško peilio problemine vieta bando atitikti visus įmanomus poreikius ir dėl to įgyvendina daug sąsajų [BMM+98]. Pateiktoje DECOR metodo realizacijoje dar įgyvendintas probleminių vietų aptikimo algoritmų generavimas ir objektinės sistemos PADL modelio generavimas. DETEX vertinimo eksperimentas atliktas su Xerces versijos 2.7.0 [Xer] atviro kodo sistema. Atpažintų objektų dalis sudarė 100% viską apimančios klasės, funkcinės dekompozicijos, painaus kodo, šveicariško peilio probleminių vietoms. Tikslumas buvo lygus 88.6%, 51.7%, 60.5%, 41.1% atitinkamai viską apimančios klasės, funkcinės dekompozicijos, painaus kodo, šveicariško peilio probleminių vietų atvejais. Tai parodo, jog nors ir visos probleminės vietos Xerces atviro kodo sistemoje buvo aptiktos, dar daug rezultatų rankinio tikrinimo reikėjo atlikti programuotojui tam, kad klaidingai teigiamos problemines vietos būtų surastos.

3.2. Aptikimas euristinių paieškos algoritmų pagalba

Programuotojui pačiam sudarinėti probleminių vietų aptikimo taisykles yra daug laiko užimantis darbas. Skirtingos metrikos ir kodo struktūriniai ar leksiniai požymiai gali būti svarbūs, bandant aptikti problemines vietas skirtingose sistemose. Taip pat kiekvienai sistemai iš naujo gali tekti kalibruoti metrikų slenksčius. Tai lemia daug rankinio darbo ir eksperimentavimo poreikį, norint aptikti kodo ir projektavimo problemines vietas. Todėl dalis mokslininkų nagrinėja euristinių paieškos algoritmų taikymo galimybes tam, kad aptiktų kodo ir projektavimo problemines vietas be išankstinio aptikimo taisyklių paruošimo.

Vienas iš euristinių paieškos algoritmų atstovų yra genetinės paieškos algoritmas [Gol89]. Genetinė paieška yra paremta gamtos evoliucijos principais. Pradžioje yra formuojama pradinių sprendinių aibė – tai vadinama populiacija. Sprendiniai yra dažniausiai užkoduoti binariniu pavidalu ir yra vadinami chromosomais, o atskiri elementai (1 ar 0) genais. Tada kiekvienos iteracijos metu kai kurie sprendiniai yra sujungiami (angl. *crossover*) ir gaunami vaikai (angl. *offspring*), kurie atsitiktinai mutuojami, keičiant atskirus genus. Gauti vaikai pridedami į populiaciją. Sekančiai iteracijai atrenkama tiek chromosomų kiek buvo pradinėje populiacijoje. Atranka vyksta pagal tinkamumo (angl. *fitness*) funkciją. Iteracijas vykdoma kol nepasiekiama fiksuoto iteracijų skaičiaus arba populiacija nustoja gerėti. Vienas iš genetinės paieškos praplėtimų yra genetinis programavimas [Koz92]. Pagrindinis skirtumas, jog sprendiniams užkoduoti yra naudojama medžio duomenų struktūra (angl. *tree*) ir jie gali būti skirtingo dydžio.

Marouane Kessentini pristatė genetiniu programavimu paremtą metodą kodo ir projektavimo probleminėms vietoms aptikti ir pataisyti [KKS+11]. Probleminių vietų aptikimo metodas kaip įeitį

gauna galimų kodo metrikų sąrašą ir istoriją iki šiol aptiktų probleminių vietų. Iš gautų metrikų suformuojama pradinė taisyklių aibė. Kiekviena iš taisyklių reprezentuojama kaip medis, kur viršūnėse konjunkcijos arba disjunkcijos operacijos, o lapuose metrikos su konkrečiais slenksčiais. Genetinio programavimo iteracijose probleminių vietų aptikimo taisyklės keičiamos, tai yra atliekamos sujungimo, mutacijos, išrinkimo operacijos, taip, kad aptiktų problemines vietas maksimaliai panašiai į istorinius duomenis. Rezultate gaunama taisyklių aibė probleminėms kodo ir projektavimo vietoms aptikti. Tada genetinės paieškos pagalba ieškoma kodo pertvarkymo galimybių (angl. *refactoring*). Genetinei paieškai kaip įeitis paduodamos probleminių vietų aptikimo taisykles gautos genetinio programavimo pagalba ir galimų kodo pertvarkymų sąrašas. Pavyzdžiui, iškelti metodą x iš klasės y į z yra galima pertvarkymo operacija. Suformuojama pradinių sprendinių aibė, kur vienas sprendinys yra vektorius, kurio dimensija yra lygi paduoto galimų pertvarkymų sąrašo ilgiui, o i-ji vektoriaus komponentė žymi ar reikia atlikti i-jį pertvarkymą (1 arba 0). Genetinės paieškos iteracijų metu siekiama sumažinti skaičių aptiktų probleminių vietų su paduotomis taisyklėmis pritaikius kodo pertvarkymo taisykles iš populiacijos. Probleminių vietų aptikimo ir taisymo metodo vertinimas atliktas su atviro kodo sistemomis GanttProject [Gan], Xerces, ArgoUML [Arg], QuickUML [Qui] viską apimančios klasės, painaus kodo ir funkcinės dekompozicijos projektavimo probleminėms vietoms. Atpažintų objektų dalis svyravo nuo 86% iki 100%, o tikslumas nuo 81% iki 100%. Bendrai vidutinis tikslumas pranoko DETEX parodytą tikslumą, tačiau atpažintų objektų dalies metriką buvo blogesnė nei DETEX atveju. Apie 78-85% probleminių vietų taisymo pasiūlymų buvo patvirtinti po rankinės peržiūros.

Usman Mansoor pristatė daugiakriteriniu genetiniu programavimu paremtą metodą kodo ir projektavimo probleminių vietų aptikimui [MKM+17]. Kaip ir Kessentini metodo atveju probleminių vietų aptikimo taisyklės koduojamos medžio pavidalu, o metodo įeitis yra kodo ir projektavimo probleminių vietų pavyzdžiai ir metrikų sąrašas, išeitis yra probleminių vietų aptikimo taisyklių aibė. Tačiau papildomai genetinio programavimo optimizacijos metodui dar paduodami geri kodo pavyzdžiai. Straipsnyje teigiama, kad dalies probleminių vietų neišeina aptikti dėl nepakankamo istorinių probleminių vietų pavyzdžių kiekio. Dėl to genetinio programavimo metu papildomai siekiama minimizuoti gerų kodo pavyzdžių aptikimą, o ne tik maksimizuoti aptiktų istorinių probleminių vietų kiekį su populiacija formuojančiomis probleminių vietų aptikimo taisyklėmis. Taip pat tvirtinama, jog papildomas optimizacijos kriterijus mažina klaidingai teigiamų probleminių vietų aptikimą. Aprašyto metodo vertinimas atliktas su 7 atviro kodo sistemomis ArgoUML versija 0.26, ArgoUML versija 0.3, Xerces versija 2.7, Ant-Apache versija 1.5 [Ant],

Ant-Apache versija 1.7, GanttProject, Azureus versija 2.3.0.6 [Azu]. Buvo ieškoma viską apimančios klasės, painaus kodo, funkcinės dekompozicijos, duomenų klasės, pavydaus metodo. Atpažintų objektų dalies metrika gauta nuo 89% iki 94%, o tikslumas nuo 79% iki 95% skirtingoms sistemoms.

3.3. Aptikimas automatinio mokymosi pagalba

Kitoje grupėje darbų, nagrinėjusių kodo ir projektavimo probleminių vietų aptikimą, yra analizuojamos automatinio mokymosi pritaikymo galimybės. Kaip ir euristiniais paieškos algoritmais grįstuose metoduose siekiama išvengti kodo ir projektavimo probleminių vietų aptikimo taisyklių sudarymo rankomis.

Abdou Maiga pristatė įrankį SVMDetect kodo ir projektavimo probleminėms vietoms aptikti, kuris remiasi atraminių vektorių klasifikatoriumi (AVK, angl. *Support Vector Machines classifier*) [MAB+12a]. Statistinio mokymosi pagalba AVK siekia surasti hiperplokštumą, kuri kuo geriau atskirtų skirtingų klasių duomenis [Vap79]. Hiperplokštuma – tai geometrinis objektas viena dimensija mažesnėje erdvėje nei kontekstinė erdvė. Pavyzdžiui, jei duomenims apibūdinti užtenka dviejų atributų, tai juos galima pavaizduoti plokštumoje, o skirtingų klasių duomenis galėtų atskirti tiesė. AVK mokomas su mokytoju, tai yra mokymosi duomenys paduodami su žinomomis klasėmis. Kiekvienai iš kodo ir projektavimo probleminių vietų SVMDetect mokomas iš naujo. Pavyzdžiui, jei ieškoma viską apimančios klasės projektavimo probleminės vietos, tai mokymosi aibę sudaro programų sistemos klases reprezentuojantys (p+1) dimensijos vektoriai, kur p yra skaičiuojamų metrikų programų sistemų klasei skaičius. Kiekvienoje iš p pirmųjų vektoriaus komponenčių yra konkreti metrikos reikšmė, o p+1 įgyja reikšmes iš aibės {-1, 1} priklausomai nuo to viską apimanti klasė egzistuoja vektoriaus reprezentuojamoje programų sistemos klasėje. Kitoms probleminės vietoms mokymosi aibės sudaromos lygiai taip pat tik, kad p+1 komponentė reprezentuoja kitos probleminės vietos būvimą ar nebūvimą. SVMDetect mokymosi metu ieškoma hiperplokštumos atskiriančios duomenis pagal p+1 komponentę, tai yra probleminės vietos būvimą ar nebūvimą. SVMDetect buvo įgyvendintas naudojant Weka įrankį [FHW16]. SVMDetect vertinimui parinktos viską apimančios klasės, funkcinės dekompozicijos, painaus kodo, šveicariško peilio projektavimo probleminės vietos ir atviro kodo sistemos ArgoUML versija 0.9.18, Azureus versija 2.3.0.6, Xerces versija 2.7.0. SVMDetect tikslumo metrika buvo geresnė nei DETEX. Pavyzdžiui, Xerces sistemoje tikslumas 95.51%, 66.93%, 86%, 80.76% atitinkamai viską apimančios klasės, funkcinės dekompozicijos, painaus kodo, šveicariško peilio probleminių vietų atvejais. Tačiau atpažintų

objektų dalies metrika svyravo nuo 70% iki 95.29% skirtingoms probleminės vietoms Xerces sistemoje, kai DETEX turėjo po 100% kiekvienai iš probleminių vietų. Tai reiškia, kad SVMDetect atveju nepakanka tik rankinių būdu atfiltruoti klaidingai teigiamas problemines vietas, nes sugrąžinamos ne visos probleminių vietų paliestos programų sistemų klasės. Dar vienas iš ribojimų – tai, kad SVMDetect nedetalizuoja kokios metrikos ir kaip buvo parinktos įrankio mokymosi aibių formavimams.

Maiga dar pratęsė darbą ties SVMDetect ir pristatė iteracinį SMURF įrankį probleminėms vietoms aptikti [MAB+12b]. SMURF taip pat paremtas SVM, tačiau grąžinus rezultatus programuotojas turi galimybę pažymėti klaidingai teigiamas problemines vietas, tai yra pažymėti programų sistemų klases, kuriose iš tikro nėra probleminių vietų. Tada suformuojama nauja mokymosi aibė, SMURF apmokomas iš naujo ir grąžinami pataisyti rezultatai. SMURF mokymą galima kartoti kiek norima kartų. Iteracinis SMURF mokymosi būdas leidžia išvengti dalies klaidingai teigiamų kodo ir projektavimo probleminių vietų atvejų.

Taip pat Francesca Fontana nagrinėjo automatinio mokymosi pritaikymo galimybes kodo ir projektavimo probleminių vietų aptikimui. Darbe „Code Smell Detection: Towards a Machine Learning-based Approach“ [FZM+13] išbandyti šeši automatinio mokymosi klasifikatoriai iš Weka paketo su numatytaisiais parametrais (angl. *default parameters*), tai yra sprendimų medis (angl. *decision tree*) J48, atsitiktinis miškas (angl. *random forest*), naivaus Bajeso klasifikavimo algoritmas (angl. *naive Bayes*), klasifikavimo taisyklių metodas JRip, AVK SMO versija (angl. *sequential minimal optimization support vector machine, SMO SVM*) [Pla98], LibSVM AVK SMO versija [CL11]. Kiekvienos iš probleminių vietų mokymosi aibei formuoti naudotos 76 Java atviro kodo sistemos iš Qualitas Corpus saugyklos [TAD+10]. Kiekvienai iš sistemų praleistas kodo ir projektavimo problemines vietas aptinkantis įrankis IPlasma, kuris grąžino klases ir metodus (priklausomai nuo kodo ar projektavimo probleminės vietos lygmens) turinčius problemines vietas. Tada atsitiktinai traukiama po vieną atvejį ir 3 magistro studentai nepriklausomai vienas nuo kito, besiremiant IPlasma grąžintais rezultatais, priskiria binarinę klasę yra ar nėra probleminės vietos. Taip suformuojama mokymosi aibė kiekvienai iš kodo probleminių vietų iš 420 įrašų, kur 140 įrašų su probleminė vieta, o 280 įrašų be probleminės vietos. Kaip ir SVMDetect atveju vienas įrašas, reprezentuojantis klasę ar metodą, turi fiksuotą skaičių metrikų. Darbe „Code Smell Detection: Towards a Machine Learning-based Approach“ Fontana bandė aptikti kodo problemines vietas: ilgą metodą, pavydų metodą, duomenų klasę, didelę klasę. J48, atsitiktinis miškas, naivus Bajesas, JRip, AVK SMO kiekvienai iš kodo probleminių vietų parodė bendrą tikslumą kiek didesnį nei 90%.

Darbe „Code smell severity classification using machine learning techniques“ Fontana pratęsė automatinio mokymosi taikymo kodo probleminių vietų aptikimui eksperimentavimą [FZ17]. Pagrindinis šio straipsnio praplėtimas yra kodo probleminių vietų klasifikavimas į keletą klasių:

1. Ne probleminė vieta
2. Nesvarbi probleminė vieta
3. Probleminė vieta
4. Svarbi probleminė vieta

Rastų kodo probleminių vietų rikiavimas pagal svarbą leidžia programuotojui prioritezuoti darbus peržiūrint ir taisant problemines vietas, kas yra labai svarbu apriboto laiko programų sistemų kūrimo projektuose. Didelei klasei, duomenų klasei ir ilgam metodui geriausias bendras tikslumas pasiektas su atsitiktiniu mišku, o pavydžiam metodui su sprendimų medžiu J48. Didelei klasei, duomenų klasei, ilgam metodui, pavydžiam metodui pasiektas atitinkamai 74%, 77%, 92%, 93% bendras tikslumas.

3.4. Aptikimas pagal kodo struktūrą

Naudojant išeities kodą ar saugumo spragų, ar klaidų, ar kodo probleminių vietų aptikimui, ji galima traktuoti kaip žodžių seką ir taikyti automatinio mokymosi modelius iš teksto apdorojimo naudojant automatinį mokymąsi srities. Toks būdas yra naudojamas sekančiuose dviejuose susijusiuose darbuose. Taip pat šiame skyriuje apžvelgiama kaip kodą galima vaizduoti abstrakčios sintaksės medžiu (angl. *abstract syntax tree*), kurio pagalba daugiau sintaksinės kodo informacijos iš karto paduodama į automatinio mokymosi modelį.

Ugur Koc nagrinėjo galimybę taikyti ITA DNT klaidingai teigiamų saugumo spragų, rastų statinės kodo analizės, kode atfiltravimui [KSF+17]. Mokymosi aibė buvo sudaryta pagal viešą Owasp duomenų aibę [Owa17], kuri skirta statinės kodo analizės įrankių testavimui. Iš 2371 įrašų 1193 sudarė klaidingai teigiami įspėjimai apie saugumo spragas. ITA DNT, kartu su binarine logistine regresija (angl. *logistic regression*) ant viršaus, modelis priėmė vektorių sekas. Kiekvienas vektorius atitinka programų sistemų metodą, o kiekviena iš vektoriaus komponenčių kodo elementą. Rezultate gaunama viena iš dviejų klasių teisingas įspėjimas apie saugumo spragą ar ne. Aprašytas modelis pasiekė tikslumą 97.3%, atpažintų objektų dalį 81.3% ir bendrą tikslumą 89.6%. Iš atpažintų objektų dalies metrikos galima matyti, jog pasiūlytas modelis neranda 18.7% klaidingai teigiamų probleminių vietų su saugumo spragomis, tačiau labai tiksliai klasifikuoja atpažintus objektus.

Hoa Khanh Dam pristatė ITA DNT paremtą modelį, kuris apmokomas kodo pavyzdžiais iš skirtingų projektų [DTP+17]. Pristatytas modelis geba įsisavinti tiek semantines, tiek sintaksines kodo savybes tam, kad aptiktų kodo saugumo spragas. Pagal atliktus vertinimus su 18 Android programų Hoa Khanh Dam pasiūlytas modelis aptinka nuo 3% iki 58% tiksliau skirtingas kodo saugumo spragas nei metodai paremti metrikomis.

3.4.1. Išeities kodo reprezentavimas abstrakčios sintaksės medžio keliais

Abstrakčios sintaksės medis – tai medžio pavidalo išeities kodo sintaksės reprezentacija [AZL+18b]. Formaliai abstrakčios sintaksės medį kodo fragmentui apibrėžia rinkinys <N, T, X, s, δ, φ> [AZL+18b], kur:

- N yra aibė viršūnių, kurios nėra lapais
- T yra aibė lapų
- X yra aibė reikšmių naudotų išeities kodo fragmente
- s yra medžio šaknis
- δ yra funkcija, kuri kiekvienam viršūnei, kuri nėra lapas, priskiria vaikų sąrašą (15 formulė)

$$\delta\colon \mathrm{N} \rightarrow (\mathrm{N} \cup \mathrm{T})^{*} \tag{15}$$

- φ yra funkcija, kuri priskiria lapams reikšmes (16 formulė)

$$\varphi\colon \mathrm{T} \rightarrow \mathrm{X} \tag{16}$$

Abstrakčios sintaksės medžio kelias, kurio ilgis k yra sekančios formos seka: $n_1 d_1 \ldots n_k d_k n_{k+1}$ [AZL+18b], kur:

- $n_1, n_{k+1} \in T$ (kelio pradžia ir pabaiga yra lapas)
- $\forall i \in [2..k]$: $n_i \in N$ (tarpinės kelio viršūnės nėra lapai)
- $\forall i \in [1..k]$: $d_i \in \{\wedge, v\}$ (d_i žymi judėjimo kryptis)

Abstrakčios sintaksės medžio kelio p pradžia dar žymima pradžia(p), o pabaiga pabaiga(p) [AZL+18b]. Abstrakčios sintaksės medžio kelio p kontekstas – tai rinkinys <x_s, p, x_t>, kur x_s yra reikšmė priskirta kelio pradžiai (17 formulė), o x_t reikšmė priskirta kelio pabaigai (18 formulė) [AZL+18b]. Pavyzdžiui, abstrakčios sintaksės medžio kelio kontekstas, kuri atspindi x = 7 programavimo sakinį, atrodytų <x, (vardo išraiška, ^, priskyrimo išraiška, v, skaitinė konstanta), 7>.

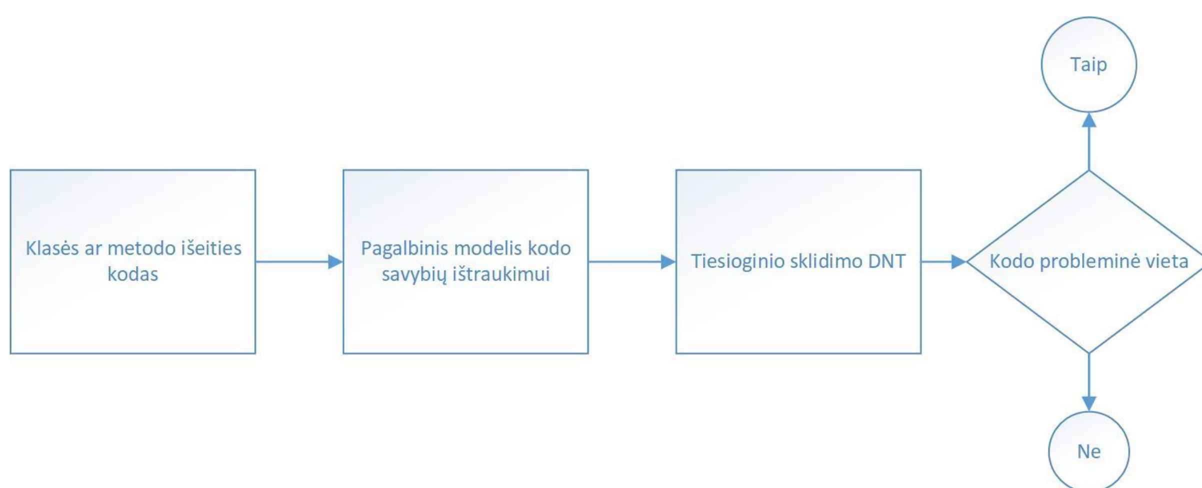
$$x_s = \varphi(\text{pradžia}(\text{p})) \tag{17}$$

$$x_t = \varphi(\text{pabaiga}(\text{p})) \tag{18}$$

Su išeities kodu dirbantis modelis galėtų priimti abstrakčios sintaksės medžio kelių kontekstus, kurie atitinka išeities kodą ir perteikia jo sintaksinę struktūrą.

4. Kodo probleminių vietų aptikimo modelis

Šiame skyriuje yra pristatytas modelis, gebantis aptikti kodo problemines vietas pagal kodo struktūrą. Modelio bendra schema yra pateikta 11 paveiksle. Modelis geba aptikti išeities kodo klasės ir metodo lygmens kodo problemines vietas. Kaip įeities tikimasi visos klasės išeities kodo arba metodo išeities kodo. Tada iš anksto apmokytas pagalbinis modelis ištraukia paduoto kodo savybes. Kodo savybių ištraukimui buvo bandyti dėmesio DNT ir ITA DNT. Tada tiesioginio sklidimo DNT yra atsakingas už klasės ar metodo klasifikavimą jau pagal ištrauktas kodo savybes. Rezultate gaunama klasė, kuri parodo kodo probleminės vietos buvimą ar nebuvimą. Toliau šiame skyriuje pateikti detalūs šio modelio dalių aprašymai.

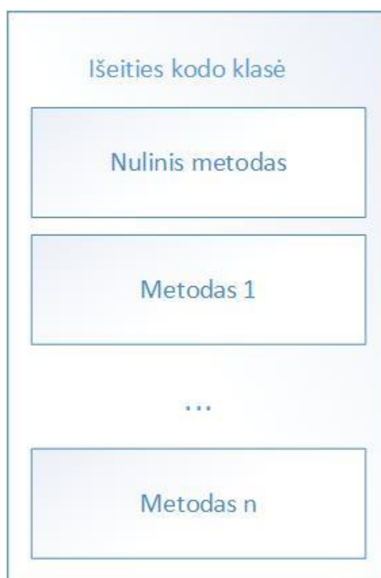


11 paveikslas. Kodo probleminių vietų aptikimo modelio schema

4.1. Duomenų aibės paruošimas

Pagal atliktą egzistuojančių kodo probleminių vietų duomenų aibių analizę nėra vienos lyginamosios duomenų aibės (angl. *benchmark dataset*), kuri būtų naudojama daugelyje mokslinių darbų susijusių su kodo probleminėmis vietomis. Nors 2015 metais buvo išleistas „Landfill" darbas, kurio tikslas buvo sudaryti įrankius visuotinei kodo probleminių vietų aibės rinkimui [PNT+15]. „Landfill" darbo grupė pateikė API sąsaja (angl. *application programming interface*, *API*) kodo probleminių vietų pridėjimui ir ištraukimui, tačiau API netapo plačiai naudojamas. Šiame darbe dėl apriboto laiko pasirinkta adaptuoti, pagal atliktą analizę, vienintelę prieinamą kodo probleminių vietų duomenų aibę, kuri buvo paruošta Fontana ir jos darbo grupės [FMZ16]. Duomenų aibę sudaro po 420 įrašų duomenų klasės, didelės klasės, pavydaus metodo ir ilgo metodo kodo probleminėms vietoms. Kiekvienai iš kodo probleminių vietų yra 280 neigiamų ir 140 teigiamų įrašų, kur teigiamas įrašas turi kodo probleminę vietą, o neigiamas neturi. Duomenų aibės įrašas reprezentuoja išeities

kodo klasę ar metodą iš vienos iš 76 Java atviro kodo sistemų Qualitas Corpus rinkinyje [TAD+10]. Šiame darbe pasitelkta idėja reprezentuoti išeities kodo klasę metodų sąrašu [DTP+17], tai padeda vienodu pavidalu turėti tiek išeities kodo klasių, tiek metodų lygmens problemines vietas (žiūrėti 12 paveikslą). Išeities kodo klasei yra išvedamas dirbtinis nulinis metodas, kurio kūne sudedami visi išeities kodo klasės laukai, o dirbtinio metodo parašas – *void header()*.



12 paveikslas. Pasirinktas išeities kodo klasės reprezentavimo būdas

Toliau pateikiami duomenų aibės adaptavimo žingsniai išeities kodo klasės lygmens probleminėms vietoms:

1. Išeities kodo klasei išvedamas nulinis metodas.
2. Įrašui saugomi išeities kodo klasės metodų ir nulinio metodo išeities kodai atskiruose failuose.

Metodo lygmens kodo probleminėms vietoms taip pat buvo pridėta apgaubiančios išeities kodo klasės kontekstinė informacija nulinio metodo pavidalu. Toliau pateikiami konkretūs duomenų aibės adaptavimo žingsniai metodo lygmens kodo probleminėms vietoms:

1. Metodui surandama apgaubianti išeities kodo klasė.
2. Apgaubiančiai išeities kodo klasei išvedamas nulinis metodas.
3. Įrašui saugomi adaptuojamo ir nulinio metodo išeities kodai atskiruose failuose.

Po adaptacijos vienas duomenų aibės įrašas turi unikalų identifikatorių, kodo probleminės vietos pavadinimą, klasę (kodo probleminė vieta ar ne) ir sąrašą failų su metodų išeities kodais.

Metodų išeities kodai buvo apdoroti pagal standartines praktikas [WVL+15]. Komentarai ir tuščios eilutės buvo ignoruotos. Metodų išeities kodai susidaro iš kodo žymių (angl. *code token*). Kodo žymė – tai eilutė su priskirta prasme nagrinėjamoje programavimo kalboje (šio darbo atveju

Java programavimo kalba) [ALS+06]. Kodo žyme gali būt separatorius, programavimo kalbos raktažodis, kintamasis ar, pavyzdžiui, konstanta. Toliau išvardintos išeities kodo transformacijos:

1. Visi išeities kode sutinkami sveiki, realieji ir kiti skaičiai buvo pakeisti <*num*> žymėmis.
2. Visos išeities kode sutinkamos eilutės konstantos (angl. *constant string*) buvo pakeistos <*str*> žymėmis.
3. Kodo žymės nepakliūnančios į 1000 populiariausių kodo žymių buvo pakeistos <*unk*> žymėmis. Žodyno dydis |Z| = 1000 parinktas pagal susijusį ITA DNT taikymą išeities kodo supratimui, kur Z yra žymimas žodynas [DTP16].

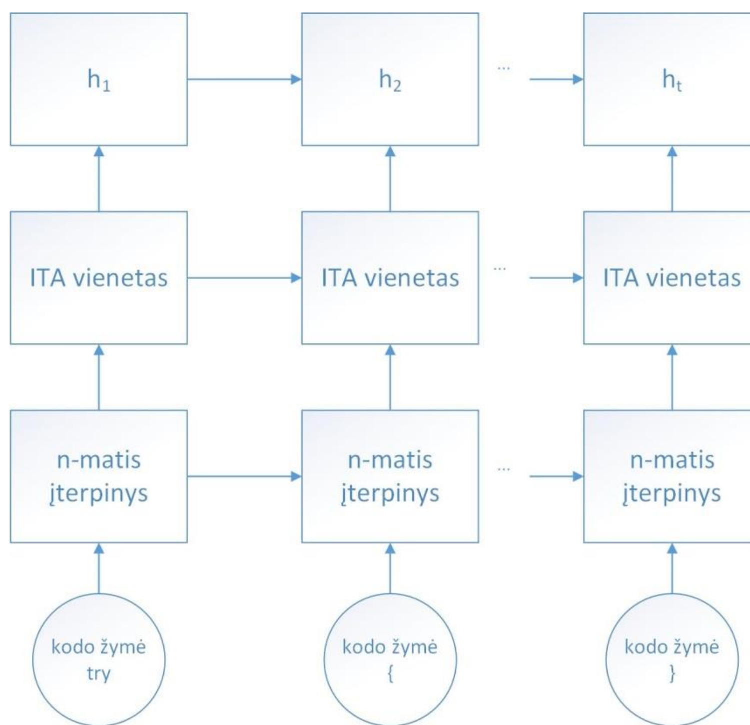
Rezultate transformuotos duomenų aibės metodų išeities kodus sudaro kodo žymės iš fiksuoto dydžio žodyno. Kodo probleminių vietų duomenų aibę, kuriai nepritaikytos išeities kodo transformacijos taisyklės vadinsime netransformuota, o kuriai pritaikytos vadinsime transformuota.

4.2. Kodo savybių ištraukimas

Dėl didelės kodo probleminių vietų duomenų aibės trūkumo nėra įmanoma išmokti ištraukti kodo savybes iš išeities kodo naudojantis probleminių vietų duomenų aibe. Tačiau iki šiol jau yra surinkta daug atviro kodo sistemų išeities kodo, pavyzdžiui Github Java Corpus turi virš 10000 populiariausių Java atviro kodo projektų išeities kodus [AMS13]. Taip pat yra sukurta automatinio mokymosi modelių gebančių išmokti kodo savybes tam, kad sugeneruotų metodo dokumentaciją [IKC+16], pasiūlytų išeities kodo galimą pratęsimo variantą (angl. *autocomplete*) [LWK+17] ar, pavyzdžiui, pasiūlytų metodo pavadinimą [AZL+18a]. Dėl to šiame darbe nuspręsta naudoti pagalbinį modelį kodo savybių ištraukimui, kuris buvo apmokytas spręsti kitą problemą, tai yra išmoktų žinių perdavimo technikos taikymas. Išmoktų žinių perdavimas buvo sėkmingai taikomas teksto klasifikavimo problemoms [DN05], vaizdų klasifikavimo problemoms [DSF16]. Pavyzdžiui, vaizdų klasifikavimo atveju sudėtingi DNT apmokyti su ImageNet [DDS+09] duomenų aibe yra naudojami savybių ištraukimui, turint tik mažą specifinių vaizdų mokymosi aibę. Tokiu būdu didelėje mokymosi aibėje išmokti bendri gebėjimai, pavyzdžiui, atpažinti linijas ar figūras yra pernaudojami mažesniam uždaviniui spręsti. Panašiai kodo probleminių vietų aptikimui reikia bazinių gebėjimų suprasti sintaksines, semantines išeities kodo savybes, kuriuos galima pernaudoti iš kitų modelių. Savybių ištraukimui iš išeities kodo šiame darbe buvo bandyti dėmesio DNT ir ITA DNT modeliai [DTP16], nes ITA DNT yra plačiai naudojamas susijusiuose darbuose [KSF+17, WTV+16]. Tuo tarpu dėmesio DNT metodo pavadinimo siūlymui pritaikiusi darbo grupė parodė,

jog dėmesio DNT suranda išeities kodo savybes, kurios priveda prie geresnio klasifikavimo rezultato nei ITA DNT, todėl nuspręsta išbandyti ir dėmesio DNT.

Kodo sintaksinių ir semantinių savybių išmokymui bandytas paprastas ITA DNT, kuris apmokomas nuspėti sekančią kodo žymę [DTP16]. Susijusiame darbe tokia architektūra buvo sėkmingai naudota išeities kodo savybių aptikimui, kurios buvo panaudotos kodo saugumo spragų klasifikavimui [DTP+17]. 13 paveiksle parodyta naudota ITA DNT architektūra. Metodo išeities kodas yra traktuojamas kaip kodo žymių seka. 13 paveiksle galima matyti, kad kiekvienu laiko momentu yra paduodama viena kodo žymė, kuri paverčiama į n komponenčių vektorių. Viena iš modelio užduočių yra išmokti kodo žymių transformacija į n-matę dimensiją išlaikant panašumus tarp pradinių kodo žymių ir n dimensijos vektorių. Pagal susijusį darbą įterpinių dimensiją parinkta lygi n = 50 [DTP16]. Skaitiniai vektoriai išlaikantys tekstinių eilučių panašumus dar vadinami įterpiniais (angl. *embedding*) konkrečioje dimensijoje. Tada konkrečiu laiko momentu ITA vienetas priima kodo žymės įterpinį ir prieš tai matytų įterpinių kontekstą ir gražina išeitį h_t, kuri gali būti panaudota sekančios kodo žymės atspėjimui. Šiuo atveju pagal susijusį darbą nuspręsta, jog išeities h_t dimensija bus lygi m = 512 [DTP16].



13 paveikslas. Kodo savybes išgaunančio ITA DNT architektūra [DTP16]

Sekančios kodo žymės tikimybės vertinimui buvo įvesti m-mačiai išmokstami parametrai U_k kiekvienai kodo žymei k iš žodyno Z. 19 formulėje pateikiamas sekančios kodo žymės ω tikimybės vertinimas pagal ITA vieneto išeitį laiko momentu t [DTP16].

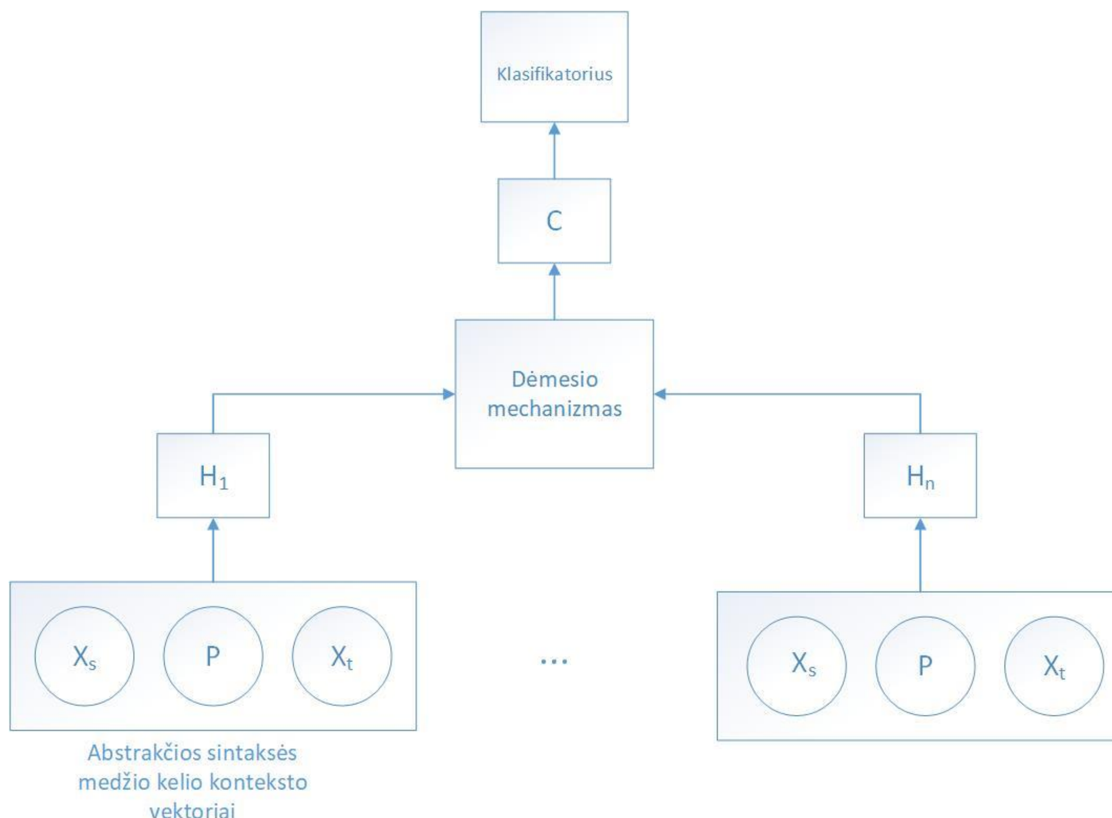
$$P(\omega) = \frac{e^{U_{\omega}^{T} h_t}}{\sum k \,\epsilon Z \, e^{U_k^T h_t}} \tag{19}$$

Mokant ITA DNT nuspėti sekančią kodo žymę, ITA DNT išmoksta išeities kodo programavimo kalbos gramatiką, o ITA DNT išeitys perteikia kodo žymės prasmę kartu su panaudojimo kontekstu [GLT01]. Dėl to sprendžiant kodo probleminių vietų aptikimo problemą, su išeities kodu apmokytą ITA DNT galima panaudoti išeities kodo savybių ištraukimui. Praleidus išeities kodą pro ITA DNT, gautas išeitis h_1, .., h_t galima traktuoti kaip kodo savybes. Toliau pateikiami detalūs išeities kodo savybių ištraukimo žingsniai:

1. Kiekvienam transformuotos kodo probleminių vietų duomenų aibės įrašui kiekvieno metodo kodo žymių seka praleidžiama pro iš anksto apmokytą ITA DNT.
2. Metodo gautoms išeitims daromas sutelkimas, tai yra suskaičiuojamas įterpinių vidurkis. Tokiu būdu gaunamas vienas savybių vektorius išeities kodo metodui.
3. Kiekvienas transformuotos kodo probleminių vietų duomenų aibės įrašas turi sąrašą metodų, todėl skaičiuojamas metodų savybių vektorių vidurkis (daromas dar vienas sutelkimas).

Taip pat šiame darbe bandytas dėmesio DNT *Code2Vec* yra sukurtas tam, kad gebėtų ištraukti išeities kodo semantines ir sintaksines savybes [AZL+18a]. 14 paveiksle yra pateikta *Code2Vec* architektūra. Nagrinėjamas dėmesio DNT priima kodo fragmento abstrakčios sintaksės medžio kelių kontekstus (žiūrėti 3.4.1 skyrių dėl notacijos ir paaiškinimo). Kelio pradžios lapas X_s, kelias P, ir kelio pabaigos lapas X_t paduodami 128-mačių vektorių įterpinių pavidalu, todėl visą kelio kontekstą $<X_s, P, X_t>$ sudaro 384 komponenčių vektorius. Prieš *Code2Vec* mokymą įterpiniai yra inicializuojami atsitiktinai ir *Code2Vec* viena iš užduočių išmokti semantinę prasmę išlaikančius įterpinius. H_1, ..., H_n yra 384-mačiai vektoriai. Tarp atitinkamai kiekvienos poros H_i ir i-ojo abstraktaus sintaksės medžio kelio konteksto sujungtų įterpinių yra pilnai sujungtas DNT sluoksnis, kurio užduotis yra išmokti labiausiai tinkamą X_s, P ir X_t sujungimą. Toliau išeities kodo fragmento kontekstų kontekstinę informacija koduojantis H_1, ..., H_n paduodami į dėmesio mechanizmą, kuris yra aprašytas 2.2 skyriuje. Išeityje gaunamas 384 komponenčių paduoto išeities kodo sintaksinių ir semantinių savybių vektorius C. Dėl mažos aibės turimų kodo probleminių vietų buvo paimtas jau apmokytas su virš 14 milijonų metodų *Code2Vec.* Aprašinėjamas dėmesio DNT buvo mokytas atspėti metodo pavadinimą, 14 paveiksle pavaizduoto klasifikatoriaus užduotis buvo priskirti labiausiai tinkamą metodo pavadinimą pagal paduoto išeities kodo ištrauktas savybes C. Šiame darbe daroma prielaida, kad dėmesio DNT *Code2Vec* modelio apmokyti parametrai išskyrus klasifikatoriaus yra naudingi bandant surasti semantines ir sintaksines kodo savybes, kurios padėtų

atskirti kodo probleminę vietą nuo neprobleminės. Toliau pateikiami detalūs žingsniai kaip buvo panaudotas dėmesio DNT *Code2Vec* savybių ištraukimui:



14 paveikslas. Dėmesio DNT *Code2Vec* architektūra. Schema yra adaptuota iš [AZL+18a]

1. Kiekvienam metodui iš netransformuotos kodo probleminių vietų duomenų aibės buvo sukurtas abstrakčios sintaksės medis ir gauti visi abstrakčios sintaksės medžio keliai. Kelio pradžia (lapas medyje), kelias ir kelio pabaiga (lapas medyje) reprezentuojami 128-mačiais vektoriais. Reprezentuoti bet kokią kelio pradžią, kelią ir kelio pabaigą 128-matėje erdvėje dėmesio DNT *Code2Vec* buvo išmokytas naudojantis metodo pavadinimų duomenų aibe.
2. Vieno metodo abstrakčios sintaksės medžio keliai paduodami į apmokytą dėmesio DNT *Code2Vec*. Priešpaskutinio sluoksnio reikšmės sudaro 384 savybės, kurios buvo gautos iš metodo išeities kodo.
3. Kadangi kodo probleminių vietų duomenų aibės įrašai turi po 2 ar daugiau metodų, tai visiems gautiems savybių 384-mačiams vektoriams yra daromas sutelkimas, tai yra paliekamas tik visų vektorių vidurkis, kuris ir reprezentuoja duomenų aibės įrašo savybes.

4.3. Kodo probleminių vietų klasifikavimas

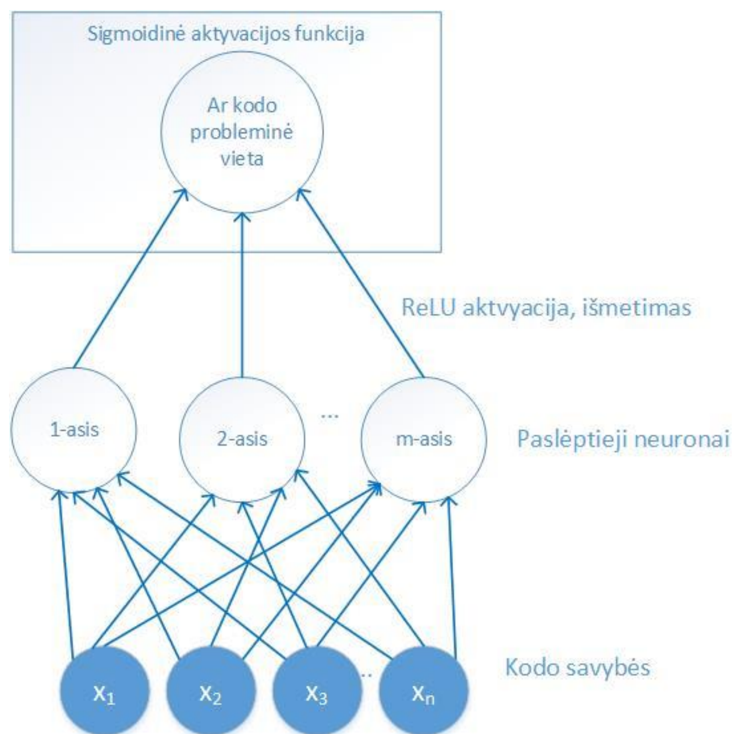
Kodo probleminių vietų klasifikavimui buvo naudotas tiesioginio sklidimo DNT. Turint išeities kodą atitinkančias savybes galima bandyti skirtingus automatino mokymosi modelius, tačiau pasiremta sėkmingais išmoktų žinių perdavimo technikos taikymo pavyzdžiais vaizdų klasifikavimo srityje, kai ištraukus savybes naudojamas tiesioginio sklidimo DNT klasifikavimui [AYX+08]. Kodo probleminių vietų klasifikavimo modelio architektūra pavaizduota 15 paveiksle. Tiesioginio sklidimo DNT turi įeities sluoksnį, kuriame priklausomai nuo išeities kodo savybių ištraukimo metodo:

- 384 įeitys, jei savybių ištraukimui naudotas *Code2Vec*
- 512 įeičių, jei savybių ištraukimui naudotas ITA DNT

Pagal paslėptų neuronų skaičiaus pasirinkimo rekomendacijas paslėptame sluoksnyje siūloma bandyti 2/3 bendro neuronų skaičiaus įeityje ir išeityje [PP14]. Taip pat paslėptame tiesioginio sklidimo DNT sluoksnyje neturėtų būti daugiau nei dvigubai neuronų nei įeities sluoksnyje [PP14]. Todėl vieninteliame tiesioginio sklidimo DNT paslėptame sluoksnyje buvo bandyti skirtingi paslėptų neuronų skaičiai:

- 256 arba 384 paslėpti neuronai, jei savybių ištraukimui naudotas *Code2Vec*
- 341 arba 512 paslėptų neuronų, jei savybių ištraukimui naudotas ITA DNT

Paslėpto sluoksnio neuronuose pritaikoma pataisyta tiesinė aktyvacijos funkcija. Po paslėpto sluoksnio buvo pridėtas išmetimo sluoksnis (angl. *dropout layer*) tam, kad modelis neprisitaikytų per daug prie mokymosi duomenų ir geriau veiktų su nematytais duomenimis. Kiekvieno paslėpto sluoksnio neurono išeitis buvo ignoruojama su tikimybe x, kur x yra išankstinis modelio parametras (angl. *hyperparameter*), kurio geriausia reikšmė buvo nustatoma empiriniu būdu. Išeities sluoksnyje yra 1 neuronas, kuriam pritaikyta sigmoidinė aktyvacijos funkcija tam, kad būtų gautas kodo probleminės vietos buvimo tikimybės įvertis.



15 paveikslas. Kodo probleminių vietų klasifikavimo modelio architektūra

4.4. Modelių mokymas

Pagalbinis modelis *Code2Vec* savybių ištraukimui iš išeities kodo buvo paimtas jau apmokytas su 14 milijonų metodų iš skirtingų Java atviro kodo projektų, kurie patalpinti Github [TAD+10].

Pagalbinis ITA DNT modelis buvo mokytas atskirai šiame darbe, nes nebuvo viešai prieinamo apmokyto ITA DNT Java išeities kodo supratimui. Mokymui buvo sudaryta 26021 metodų, kurie buvo atsitiktinai parinkti iš Github Java Corpus virš 10000 Java atviro kodo projektų [AMS13], duomenų aibe. Suformuotoje duomenų aibėje yra 21090674 kodo žymių, iš kurių 60936 unikalios. Susijusiame darbe [DTP16], kuriuo remtasi konstruojant ITA DNT, buvo naudota duomenų aibė iš 6103191 kodo žymių, todėl padaryta prielaida, jog suformuota duomenų aibė yra pakankamo dydžio. Metodų išeities kodams buvo pritaikytos transformavimo taisyklės aprašytos 4.1 skyriuje. Po transformacijos metodo išeities kodą sudaro kodo žymės iš fiksuoto dydžio žodyno. Šiame darbe parinktas žodyno dydis yra N = 1000 pagal susijusį darbą [DTP16]. Duomenų aibė buvo padalinta į mokymosi (90%) ir testavimo (10%) aibes. Nebuvo sudaroma atskira patikros duomenų aibė, nes išankstiniai modelio parametrai buvo parinkti pagal susijusius darbus [DTP16, DTP+17]. Kodo žymės patekusios į testavimo aibę, tačiau neegzistuojančios mokymosi aibėje, buvo pakeistos *<unk>* žymėmis.

ITA DNT mokymui naudota praplėsta versija gradientinio nusileidimo optimizavimo algoritmo Adam. Buvo naudota stochastinė gradientinio nusileidimo atmaina, kuriame įrašai paduodami mažomis grupėmis (angl. *mini-batch gradient descent*), tam, kad būtų sumažinta tikimybė pastrigti lokaliame minimume [GHJ+15]. Pagal susijusį darbą modelis buvo mokytas naudojant grupes po 50 sekų iš T = 100 kodo žymių [DTP+17]. Fiksuotas sekos ilgis išsprendžia nevienodo ilgio metodų problemą. Toliau aprašomas procesas kaip metodo išeities kodas buvo pakeičiamas į sekas po T kodo žymių.

1. Iš metodo išeities kodo ištraukiama X kodo žymių seka.
2. Jei |X| < T, tai prie X pridedama T – X <*unk*> kodo žymių.
3. Iš X šalinama |X| mod T paskutinių kodo žymių.

20 formulėje parodyta nuostolių funkcija naudota ITA DNT mokymui. L priima T ilgio, išlanksto žinomą kodo žymių seką s = <ω_1, ω_2, ..., ω_T>. Apmokyto ITA DNT išeitys gali būti panaudotos 19 formulės pagalba įvertinti kiekvienos iš kodo žymių tikimybes P(ω_k), kur k nuo 1 iki T. $-\log P\ (\omega_k)$ išraiška sparčiai didėja kuo mažesnė yra įvertinta tikros kodo žymės tikimybė ir lygi 0, kai įvertinta kodo žymės tikimybė yra lygi 1.

$$L(s) = -\log P\ (\omega_1) - \sum_{k=2}^{T} \log P(\omega_k | \omega_{1:k-1}) \tag{20}$$

Kartu su nuostolio funkcija buvo skaičiuojamas ir sumišimo įvertis (angl. *perplexity*) tam, kad galima būtų įvertinti ar šiame darbe apmokytas ITA DNT modelis pasiekė panašų lygį kaip susijusio darbo ITA DNT [DTP16]. 21 formulėje yra parodytas sumišimo skaičiavimo būdas, kur #kodo žymių žymi bendrą skaičių kodo žymių per visas kodo žymių sekas s, o $-\log P(s) = L(s)$. Kuo mažesnis sumišimas tuo modelis pasiekė geresnį rezultatą. Intuityviai sumišimas parodo iš kiek kodo žymių renkamasi, bandant nuspėti sekančią kodo žymę. Šiame darbe apmokyto ITA DNT pasiektas kodo sumišimas yra 4.9, kuris yra panašus į geriausią pasiektą sumišimą susijusiame darbe 4.72 [DTP16], todėl galima daryti prielaidą, jog apmokytas pakankamai geras ITA DNT savybių ištraukimui kodo probleminių vietų klasifikavimui.

$$e^{-\frac{\sum_s \log P(s)}{\#kodo\ žymių}} \tag{21}$$

Mokant ITA DNT pagal susijusį darbą buvo parinkta išmetimo tikimybė lygi 0.5 tam, kad modelis per daug neprisitaikytų prie mokymosi duomenų [DTP16]. Mokymosi greitis buvo paliktas numatytasis Adam optimizavimo algoritmui (0.001). ITA DNT buvo mokomas 3 epochas ir buvo

išsaugotas modelio variantas pasiekęs mažiausią nuostolių funkcijos rezultatą tam, kad vėliau galima būtų ji naudoti savybių ištraukimui mokant kodo probleminių vietų klasifikatorius.

Po savybių ištraukimo iš išeities kodo pritaikymo buvo gauta duomenų aibė, kur įrašas turi 384 komponenčių (ITA DNT atveju 512 komponenčių) vektorių, atspindinti išeities kodo (klasės ar metodo) sintaksines ir semantines savybes, kodo probleminės vietos tipą ir klasę. 4 atskiri klasifikavimo modeliai buvo mokomi 4 tipams kodo probleminių vietų, todėl gauta savybių duomenų aibė buvo padalinta į 4 dalis pagal kodo probleminės vietos tipą (po 420 įrašų). Toliau skirtingoms kodo probleminėms vietoms modelio mokymo procesas buvo identiškas. Kodo savybių duomenų aibė buvo padalinta į mokymosi (60%), patikros (20%) ir testavimo (20%) aibes. Padalijimo metu buvo išlaikytas įrašų turinčių ir neturinčių kodo probleminę vietą santykis.

Tiesioginio sklidimo DNT mokymui naudota praplėsta versija gradientinio nusileidimo optimizavimo algoritmo Adam. Buvo naudota stochastinė gradientinio nusileidimo atmaina, kuriame įrašai paduodami mažomis grupėmis. Pagal susijusių darbų rekomendacijas modelis buvo mokytas naudojant grupes po 32 įrašus [ML18]. Kaip nuostolių funkciją naudota binarinę kryžminės entropijos nuostolių funkciją. 22 formulėje yra parašyta nuostolio funkcija vienam įrašui, kur y žymi tikrąją įrašo klasę (1 – kodo probleminė vieta, 0 – nėra kodo probleminės vietos), o x žymi tiesioginio DNT grąžintą tikimybės įvertį, kad įrašas turi kodo probleminę vietą. Galima matyti, kad turint įrašą su kodo problemine (y = 1) vieta, tik pirmoji nuostolių funkcijos dalis –y*log(x) yra aktuali. Ji sparčiai didėja esant mažam tikimybės įverčiui x, tuo tarpu lygi 0 esant idealiam tikimybės įverčiui (x = 1). Atvirkščiai turint įrašą be kodo probleminės vietos (y = 0), tik antroji nuostolių funkcijos dalis –(1-y)*log(1-x) yra aktuali. Ji sparčiai didėja dideliam tikimybės įverčiui x, tuo tarpu lygi 0 esant idealiam tikimybės įverčiui (x = 0).

$$l(x, y) = -y * \log(x) - (1 - y) * \log(1 - x) \tag{22}$$

Mokant tiesioginio sklidimo DNT buvo naudota ankstyvo sustojimo technika (angl. *early stopping*) tam, kad būtų išvengta tiek pernelyg didelio prisitaikymo prie mokymosi duomenų, tiek perteklinių mokymosi epochų. Ankstyvas sustojimas taikytas pagal Prechelt rekomendacijas [Pre96]. Kas antrą mokymosi epochą buvo tikrinama vidutinė nuostolių funkcijos reikšmė patikros duomenų aibėje, jei nuostolių funkcijos reikšmė patikros duomenų aibėje paaugo, tai reiškia modelis pradeda pernelyg prisitaikyti prie mokymosi aibės ir metas sustabdyti mokymąsi. Todėl kaip galutinis modelis yra parenkamas ne modelis po paskutinios epochos, o modelis parodęs mažiausia nuostolio funkcijos reikšmę po bet kurios iš mokymosi epochų.

Tiesioginio sklidimo DNT paslėpto sluoksnio neuronų skaičius buvo parenkamas iš aibės {256, 384}, kai savybės ištrauktos *Code2Vec* modelio arba iš aibės {341, 512}, kai savybės ištrauktos ITA DNT modelio. Išmetimo sluoksnio tikimybė, jog neuronas bus ignoruojamas buvo parenkama iš aibės {0.1, 0.2, 0.3, 0.4, 0.5}. Optimalių išankstinių modelio parametrų parinkimas vyko, mokant modelį su visomis įmanomomis išankstinių parametrų poromis ir parenkant modelį su didžiausiu F įverčių patikros duomenų aibėje. F įvertis naudotas, nes kodo probleminės vietos buvimas yra žymiai retesnė klasė nei kodo probleminės vietos nebuvimas. Mokymosi greitis buvo parinktas iš karto pakankamai mažas (0.0005), kad būtų išvengta minimumo peršokimo efekto (angl. *overshooting*). Mažas mokymosi greitis nesukėlė skaičiavimo laiko problemos, nes mokymosi aibė buvo nedidelė.

4.5. Įgyvendinimo detalės

Fontana ir jos darbo grupės paruoštos kodo probleminių vietų duomenų aibė turėjo metodų arba klasių pavadinimus ir metrikas. Šiam darbui reikėjo išeities kodo failų, kurie turėjo Fontana duomenų aibės išeities kodo klases ir metodus. Reikalingų išeities kodo failų paieškai Qualitas Corpus rinkinyje buvo paruošta .NET programa *DataPreparation*. Surastų Java išeities kodo failų apdorojimui buvo sukurta Java programa *CodeAnalysis* [But19a]. JavaParser biblioteka buvo naudojama klasių, laukų, metodų išeities kodo ištraukimui. Java programos rezultate gauta netransformuota kodo probleminių vietų duomenų aibė, iš kurios savybes geba ištraukti *Code2Vec*. Vienas duomenų aibės įrašas atitinka direktoriją, kurios pavadinimo šablonas yra sekantis {kodo probleminės vietos tipas}_{klasė}_{unikalus identifikatorius}. Kiekvienos direktorijos viduje yra nulinio metodo ir kitų metodų išeities kodų failai. Transformuotos kodo probleminių vietų duomenų aibės paruošimui buvo sukurta Java programa *CodeModellingPreparation* [But19b]. Ta pati programa buvo atsakinga už Java išeities kodo duomenų aibės paruošimą iš Github Java Corpus. Rezultate gautas žodynas *dictionary_large.csv* ir *code_corpus_large.csv*, kurie buvo naudoti ITA DNT mokymui [But19b].

Iš anksto apmokytas dėmesio DNT *Code2Vec* modelis ir metodo išeities kodą apdorojantis Python kodas, kad ji galima būtų paduoti į *Code2Vec* modelį, buvo paimti iš oficialios *Code2Vec* autorių GitHub saugyklos [AZL+18c]. Tada buvo paruošta Python programa, kuri perbėga per netransformuota kodo probleminių vietų duomenų aibę, paleidžia dėmesio DNT *Code2Vec* modelį kiekvienam išeities kodo metodui, padaro rezultatų sutelkimą ir suformuoja išeities kodo savybių duomenų aibę.

ITA DNT buvo aprašomas ir mokomas interaktyvioje Gooble Colab aplinkoje, kuri suteikė galimybę naudoti Tesla K80 GPU. Pytorch bibliotekos pagalba įgyvendinto ITA DNT kodas yra pasiekimas Github saugykloje [But19c]. Tiesioginio sklidimo DNT buvo taip pat aprašomas ir mokomas interaktyvioje Google Colab aplinkoje, naudojant Pytorch biblioteką [But19a].

5. Kodo probleminių vietų aptikimo vertinimas

Išankstiniai klasifikavimo modelio (tiesioginio sklidimo DNT) parametrai buvo parinkti naudojantis patikros kodo savybių duomenų aibe. Klasifikavimo modelis buvo mokomas po 5 kartus su visom išankstinių parametrų (neuronų skaičius paslėptame sluoksnyje, išmetimo tikimybė išmetimo sluoksnyje) porų reikšmėmis. Kiekvienai išankstinių parametrų porai buvo parenkamas geriausias F įvertis patikros duomenų aibėje. Tada buvo parenkama išankstinių modelio parametrų pora pagal geriausią F įvertį parametrų porai patikros duomenų aibėje. Tokiu būdu gauti klasifikavimo modeliai kiekvienai iš kodo probleminių vietų. 3 lentelėje pateikti parinkti išankstiniai tiesioginio sklidimo DNT parametrai ir klasifikavimo rezultatą testavimo duomenų aibėje apibūdinančios metrikos, kai savybių ištraukimui buvo naudotas *Code2Vec* modelis.

3 lentelė. Klasifikavimo modelių rezultatai testavimo aibėje ir parinkti išankstiniai parametrai, panaudojus *Code2Vec* kodo savybių ištraukimui

	Didelė klasė	**Duomenų klasė**	**Ilgas metodas**	**Pavydus metodas**
Parinktas neuronų skaičius paslėptame sluoksnyje	256	256	256	256
Išmetimo tikimybė	0.2	0.5	0.2	0.1
Atpažintų objektų dalis	0.67	0.75	0.80	0.70
Tikslumas	0.89	0.82	0.89	0.46
Bendras tikslumas	0.86	0.86	0.90	0.69
F įvertis	0.76	0.78	0.84	0.56

Iš rezultatų 3 lentelėje galima matyti, kad tiesioginio sklidimo DNT architektūra su 256 neuronais paslėptame sluoksnyje buvo pranašesnė, kai yra 384 įeičių. Didžiausios paslėpto sluoksnio neurono rezultato išmetimo tikimybės prireikę duomenų klasės probleminės vietos klasifikatoriui. Kiti klasifikavimo modeliai (kitoms kodo probleminės vietoms) veikė geriausiai su maža (0.1, 0.2) išmetimo tikimybe. Pagal pasiektus rezultatus atskirai reikėtų vertinti didelės klasės,

duomenų klasės ir ilgo metodo klasifikatorius. Duomenų klasės, didelės klasės klasifikatoriai pasiekė bendrą tikslumą virš 0.85, o ilgo metodo klasifikatorius netgi 0.9 bendrą tikslumą. Šių trijų kodo probleminių vietų klasifikatoriai turėjo virš 0.75 F įverti, tai netgi dar svarbiau nei bendras tikslumas dėl nesubalansuotų klasių. Analizuojami trys klasifikatoriai turėjo pakankamai aukštą tikslumą (virš 0.82 visi), tai reiškia padorų skaičių klaidingai teigiamų įspėjimų realiame taikyme. Atpažintų objektų dalis buvo kiek žemesnė didelės klasės klasifikatoriui (0.67), tai galima interpretuoti, kad realiame taikyme būtų praleista 33% iš tikro didelės klasės kodo probleminių vietų. Blogesnius rezultatus nei duomenų klasės, didelės klasės ir ilgo metodo klasifikatoriai parodė pavydaus metodo klasifikatorius. Visų pirma tai atsispindi tikslumo metrikoje, kas reiškia didelį skaičių potencialiai klaidingai teigiamų rezultatų, jei klasifikatorius būtų naudojamas. Galima būtų numanyti, kad geresniems pavydaus metodo klasifikatoriaus rezultatams reikia ne tik programavimo klasės išeities kodo, kurioje randasi kodo probleminė vieta, tačiau ir kitų klasių išeities kodų, kurios susietos per kompoziciją.

4 lentelėje pateikti kodo probleminių vietų klasifikavimo rezultatai, kai išeities kodo savybių ištraukimui naudotas iš anksto apmokytas ITA DNT. Galima matyti, kad didelės klasės ir ilgo metodo klasifikavimui geriau tiko tiesioginio sklidimo architektūros su mažiau neuronų paslėptame sluoksnyje (341). Metodų lygmens kodo probleminių vietų klasifikatoriai veikė geriau su nežymiai didesne išmetimo tikimybe (0.4) nei išeities kodo klasės lygmens kodo probleminių vietų klasifikatoriai (0.3). Didelės klasės, duomenų klasės, ilgo metodo kodo probleminių vietų klasifikatoriai pasiekė gerą F įvertį, nes mažiausias iš jų 0.87. Pavydaus metodo F įvertis buvo pastebimai žemesnis nei kitų kodo probleminių vietų, tačiau visų kodo probleminių vietų klasifikatoriai pasiekė geresnį F įvertį, kai kaip įeitis naudojamos savybės ištrauktos ITA DNT vietoj savybių ištrauktų *Code2Vec*.

4 lentelė. Klasifikavimo modelių rezultatai testavimo aibėje ir parinkti išankstiniai parametrai, panaudojus ITA DNT kodo savybių ištraukimui

	Didelė klasė	**Duomenų klasė**	**Ilgas metodas**	**Pavydus metodas**
Parinktas neuronų skaičius paslėptame sluoksnyje	341	512	341	512

Išmetimo tikimybė	0.3	0.3	0.4	0.4
Atpažintų objektų dalis	0.96	0.89	0.93	0.71
Tikslumas	0.85	0.84	0.93	0.67
Bendras tikslumas	0.87	0.83	0.96	0.84
F įvertis	0.9	0.87	0.93	0.69

5 lentelėje pateiktos vidutinės klasifikavimo modelių metrikų reikšmės išeities klasės lygmens kodo probleminėms vietoms, metodo lygmens kodo probleminėms vietoms ir bendras klasifikavimo modelių vidurkis. Vidurkių skaičiavimui buvo paimti klasifikatorių rezultatai, kurie buvo apmokyti ištraukus savybes su ITA DNT, nes jie parodė geresnius rezultatus nei klasifikatoriai su *Code2Vec* ištrauktomis savybėmis. Galima pastebėti, kad klasės lygmens kodo probleminių vietų klasifikatoriai parodė geresnius rezultatus pagal tikslumą, atpažintų objektų dalį ir F įvertį, tik bendras tikslumas yra nežymiai didesnis metodo lygmens probleminių vietų klasifikatorių. Pagrinde taip yra dėl prastesnių pavydaus metodo teigiamų atvejų klasifikavimo rezultatų.

5 lentelė. Klasifikavimo modelių rezultatų vidurkiai išeities kodo klasės, metodo lygmens kodo probleminėms vietoms ir bendras vidurkis

	Atpažintų objektų dalis	**Tikslumas**	**Bendras tikslumas**	**F įvertis**
Išeities kodo klasės lygmens probleminės vietos	0.93	0.85	0.85	0.89
Metodo lygmens kodo probleminės vietos	0.82	0.80	0.90	0.81
Bendras	0.88	0.83	0.88	0.85

Kodo probleminių vietų klasifikavimo pagal išeities kodą rezultatai buvo palyginti su Fontana ir jos darbo grupės pasiektais rezultatais [FZM+13]. Klasifikavimui pagal išeities kodą panaudotas

ITA DNT išeities kodo savybių ištraukimui. Fontana ir jos darbo grupė rankomis parinkinėjo programų sistemų metrikas ir mokė automatinio mokymosi modelį kodo probleminių vietų aptikimui. Palyginimo rezultatai pateikti 6 lentelėje. Galima matyti, kad didelės klasės ir ilgo metodo bendri tikslumai ir F įverčiai yra labai artimi klasifikavimo rezultatams su rankomis parinktomis metrikomis. Duomenų klasės klasifikavimo bendras tikslumas šiek tiek žemesnis, kai tuo tarpu pavydaus metodo F įvertis yra pastebimai žemesnis. Visgi susijusiame darbe parodyta, jog Fontana ir jos darbo grupė labai gerai rankomis parinko metrikas būtent savo sukurtai kodo probleminių vietų duomenų aibei ir labai aukšti klasifikavimo rezultatai neparodo kodo probleminių vietų aptikimo technikos idealumo [NPT+18]. Atlikus nedidelius kodo probleminių vietų duomenų aibės pertvarkymus, Fontana ir jos darbo grupės aprašyta aptikimo technika nesugebėjo prisitaikyti prie pakeitimų ir rezultatai ženkliai sumažėjo [NPT+18]. Šiame darbe pasiūlytas kodo probleminių vietų aptikimo modelis neįtraukia rankinio metrikų ar kodo savybių parinkinėjimo (jos yra nustatomos automatiškai iš išeities kodo), dėl to pasiūlytas modelis turi didesnius šansus išlaikyti gerus rezultatus su nematytais duomenimis iš naujų projektų nei automatinio mokymosi modelis su rankomis parinktomis metrikomis. Todėl, jei kodo problemines vietas aptinkantis modelis rodo rezultatus artėjančius prie Fontana ir jos darbo grupės rezultatų Fontana ir jos darbo grupės paruoštoje duomenų aibėje, tai yra pakankamai perspektyvus modelis.

6 lentelė. Kodo probleminių vietų aptikimo pagal išeities kodą ir rankomis parinktas metrikas palyginimas

	Bendras tikslumas	**F įvertis**
Didelė klasė (pagal išeities kodą)	0.87	0.90
Didelė klasė (pagal rankomis parinktas metrikas)	0.96	0.96
Duomenų klasė (pagal išeities kodą)	0.83	0.87
Duomenų klasė (pagal rankomis parinktas metrikas)	0.98	0.98
Pavydus metodas (pagal išeities kodą)	0.84	0.69
Pavydus metodas (pagal rankomis parinktas metrikas)	0.94	0.94
Ilgas metodas (pagal išeities	0.96	0.93

kodą)		
Ilgas metodas (pagal rankomis parinktas metrikas)	0.98	0.98

Įsitikinimui, jog šiame darbe apmokyti kodo probleminių vietų modeliai geba aptikti prasmingus pavyzdžius buvo atliktas pirminis testas (angl. *sanity check*) su didelės klasės, duomenų klasės, pavydaus metodo, ilgo metodo pavydžiais iš 1.1 skyriaus. Šių pavyzdžių pilni išeities kodais yra prieinami Github [But19a]. Apmokyti modeliai sugebėjo aptikti kodo probleminių vietų pavyzdžius.

Šiame darbe nebandyta apjungti *Code2Vec* ir ITA DNT ištrauktas savybes kiekvienam iš kodo probleminių vietų klasifikavimo uždavinių, nes turimos kodo probleminių vietų aibės yra per mažos klasifikatorių mokymui su daug savybių. Nepadidinus kodo probleminių vietų duomenų aibės per didelis automatinių savybių skaičius sukeltų tik didesnį klasifikatoriaus prisitaikymą prie mokymosi duomenų.

Rezultatai ir išvados

Šio darbo rezultatai yra:

1. Kodo probleminių vietų duomenų aibė su išeities kodais.
2. ITA DNT modelis gebantis nuspėti sekančias išeities kodo žymes.
3. Modelis, gebantis aptikti didelės klasės, duomenų klasės, ilgo metodo, pavydaus metodo kodo problemines vietas pagal išeities kodą.

Tiek sudarinėjant kodo probleminių vietų aptikimo taisykles rankomis, tiek taikant kodo probleminių vietų aptikimo įrankius, kurie remiasi euristiniais paieškos algoritmais ar automatinio mokymosi modeliais nagrinėtais susijusiuose darbuose, reikia kiekvienam projektui parinkinėti programų sistemų metrikas. Šiame darbe pateiktas kodo probleminių vietų aptikimo modelis pats aptinka efektyviam kodo probleminių vietų klasifikavimui reikalingas išeities kodo savybes. Darbe pasiūlyto modelio didelės klasės, duomenų klasės ir ilgo metodo kodo probleminių vietų aptikimo rezultatai yra pakankamai artimi Fontana ir jos darbo grupės rezultatams be rankinio kodo savybių (metrikų) parinkinėjimo, todėl šio darbo modelis be papildomų pakeitimų gali efektyviai aptikti kodo problemines vietas naujuose projektuose.

Kituose darbuose galima būtų plėsti kodo probleminių vietų duomenų aibę, pridedant naujus kodo probleminių vietų tipus, kurie dar labiau priklausytų nuo išeities kodo struktūros. Padidinus kodo probleminių vietų duomenų aibę galima būtų atsisakyti išmoktų žinių perdavimo technikos taikymo ir tikėtina pagerinti modelio efektyvumą, mokant jį nuo pradžios. Ateityje dideliu kodo probleminių vietų aptikimo modelio pagerinimu būtų prioritetų priskyrimas aptiktiems rezultatams. Taip pat įdomu būtu paeksperimentuoti modelio efektyvumą bandant suformuoti kodo savybių vektorių, kuris atitiktų ne tik išeities kodo klasės kontekstą, bet ir priklausomybėmis susietas išeities kodo klases. Kitas spektras ateities darbų būtų naujų modelių bandymas išeities kodo savybių ištraukimui, pavyzdžiui, ITA DNT su dėmesio mechanizmu.

Šaltiniai

[ABL89] A. Ackerman, L. Buchwald, and F. Lewski. Software inspections: An effective verification process. Software IEEE, 1989, pp. 31–36.

[AFE84] A. Ackerman, P. Fowler, and R. Ebenau. Software inspections and the industrial production of software. Proceedings of a symposium on Software validation: inspection-testing-verification, 1984, pp. 13-40.

[Ale18] A. Alexander. A Java "extract method" refactoring example. [Žiūrėta 2019-05-07]. Prieiga per internetą: https://alvinalexander.com/java/refactoring-extract-method-java-example.

[ALS+06] A. Aho, M. Lam, R. Sethi, J. Ullman. Compilers: Principles, Techniques, and Tools (2nd edition). Addison Wesley, 2006.

[AMS13] Allamanis, Miltiadis, C. Sutton. Mining Source Code Repositories at Massive Scale using Language Modelling. Proceedings of the 10th Working Conference on Mining Software Repositories, 2013, pp. 207-216.

[Ant] Ant-Apache. [Žiūrėta 2018-06-13]. Prieiga per internetą: https://ant.apache.org/srcdownload.cgi.

[Arg] ArgoUML. [Žiūrėta 2018-06-13]. Prieiga per internetą: http://argouml.tigris.org/source/browse/argouml/trunk/src.

[AYX+08] A. Ahmed, K. Yu, W. Xu, Y. Gong, E. Xing. Training Hierarchical Feed-Forward Visual Recognition Models Using Transfer Learning from Pseudo-Tasks. Proceedings of the 10th European Conference on Computer Vision, 2008, pp. 69-82.

[AZL+18a] U. Alon, M. Zilberstein, O. Levy, E. Yahav. Code2Vec: Learning Distributed Representations of Code. arXiv preprint arXiv:1803.09473, 2018.

[AZL+18b] U. Alon, M. Zilberstein, O. Levy, E. Yahav. A General Path-based Representation for Predicting Program Properties. In Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2018, pp. 404-419.

[AZL+18c] U. Alon, M. Zilberstein, O. Levy, E. Yahav. Code2Vec, 2018. [Žiūrėta 2019-05-07]. Prieiga per internetą: https://github.com/tech-srl/code2vec.

[Azu] Azureus. [Žiūrėta 2018-06-13]. Prieiga per internetą: https://www.sourceforge.net/projects/azureus.

[But19a] Dariuš Butkevičius. Code Smells Detection, 2019. [Žiūrėta 2019-05-07]. Prieiga per internetą: https://github.com/D-a-r-e-k/Code-Smells-Detection.

[But19b] Dariuš Butkevičius. Source Code Modelling Preparation, 2019. [Žiūrėta 2019-05-07]. Prieiga per internetą: https://github.com/D-a-r-e-k/Source-Code-Modelling-Preparation.

[But19c] Dariuš Butkevičius. Source Code Modelling, 2019. [Žiūrėta 2019-05-07]. Prieiga per internetą: https://github.com/D-a-r-e-k/Source-Code-Modelling.

[BMM+98] W. H. Brown, R. C. Malveau, H. W. McCormick, T. J. Mowbray. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley & Sons, 1998.

[Cop04] J. O. Coplien. Organizational Patterns of Agile Software Development. Prentice Hall, 2004.

[CL11] C. C. Chang, C. J. Lin. LIBSVM: A library for support vector machines. Journal ACM Transactions on Intelligent Systems and Technology, vol. 2, no. 27, 2011.

[DDS+09] J. Deng, W. Dong, R. Socher, L. Li, K. Li, L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. CVPR09, 2009.

[DN05] C. B. Do, A. Ng. Transfer Learning for Text Classification. Proceedings of the 18th International Conference on Neural Information Processing Systems, 2005, pp. 299-306.

[DSF16] Z. Ding, M. Shao, Y. Fu. Transfer Learning for Image Classification with Incomplete Multiple Sources. International Joint Conference on Neural Networks, 2016.

[DTP16] H. K. Dam, T. Tran, T. Pham. A deep language model for software code. arXivpreprint arXiv:1608.02715, 2016.

[DTP+17] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, A. Ghose. Automatic feature learning for vulnerability prediction. arXiv preprint arXiv:1708.02368, 2017.

[FBB+99] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts. Refactoring: Improving the design of existing code. Addison Wesley Professional, 1999.

[FHW16] E. Frank, M. A. Hall, I. H. Witten. Online Appendix for Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann, Fourth Edition, 2016.

[FMZ16] F. A. Fontana, M. V. Mäntylä, M. Zanoni. Comparing and Experimenting Machine Learning Techniques for Code Smell Detection. Empirical Software Engineering, vol. 21, no. 3, 2016, pp. 1143-1191.

[Fow13] M. Fowler. TellDontAsk. [Žiūrėta 2019-05-07]. Prieiga per internetą: https://martinfowler.com/bliki/TellDontAsk.html.

[FZ17] F. A. Fontana, M. Zanoni. Code smell severity classification using machine learning techniques. Knowledge-Based Systems, vol. 128, 2017, pp. 43-58.

[FZM+13] F. A. Fontana, M. Zanoni, A. Marino, M. V. Mäntylä. Code smell detection: towards a machine learning-based approach. Proceedings of the 29th IEEE International Conference on Software Maintenance, 2013, pp. 396-399.

[GA08] Y. G. Gueheneuc, G. Antoniol. DeMIMA: A multi-layered framework for design pattern identification. Transactions on Software Engineering, vol. 34, no. 5, 2008, pp. 667-684.

[Gan] GanttProject. [Žiūrėta 2018-06-13]. Prieiga per internetą: https://www.ganttproject.biz.

[GHJ+15] R. Ge, F. Huang, C. Jin, Y. Yuan. Escaping From Saddle Points-Online Stochastic Gradient for Tensor Decomposition. COLT, 2015, pp. 797-842.

[Gla01] Robert L. Glass. Frequently Forgotten Fundamental Facts about Software Engineering. Software IEEE, 2001, pp. 111-112.

[GLT01] C. L. Giles, S. Lawrence, A. C. Tsoi. Noisy time series prediction using recurrent neural networks and grammatical inference. Machine learning, vol. 44, no. 1, 2001, pp. 161–183.

[Gol89] D. E. Goldberg. Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing, 1989.

[HBF+01] S. Hochreiter, Y. Bengio, P. Frasconi, J. Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[HS97] S. Hochreiter, J. Schmidhuber. Long short-term memory. Neural Computations, 1997.

[IKC+16] S. Iyer, I. Konstas, A. Cheung, L. Zettlemoyer. Summarizing Source Code using a Neural Attention Model. ACL, 2016.

[KE03] T. Kremenek, D. Engler. Z-ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. Proceedings of the 10th International Conferrence on Statis Analysis, 2003, pp. 295-315.

[KKS+11] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, A. Ouni. Design defects detection and correction by example. 19th IEEE international conference on program comprehension, 2011, pp. 81-90.

[Koz92] J. R. Koza. Genetic Programming: On the Programming of Computer by Means of Natural Selection. MIT Press, 1992.

[KSF+17] U. Koc, P. Saadatpanah, J. S. Foster, A. A. Porter. Learning a classifier for false positive error reports emitted by static code analysis tools. Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, 2017, pp. 35-42.

[Lea95] Learning to Learn: Knowledge Consolidation and Transfer in Inductive Systems. NIPS*95 Post-Conference Workshop, 1995.

[LPM15] M. Luong, H. Pham, C. Manning. Effective Approaches to Attention-based Neural Machine Translation. arXiv preprint arXiv:1508.04025, 2015.

[LS07] W. Li, R. Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. Journal of Systems and Software, 2007, pp. 1120-1128.

[LWK+17] J. Li, Y. Wang, I. King, M. R. Lyu. Code Completion with Neural Attention and Pointer Networks. arXiv preprint arXiv:1711.09573, 2017.

[MAB+12a] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y. Gueheneuc, G. Antoniol, E. Aimeur. Support Vector Machine for Anti-Pattern Detection. Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, 2012.

[MAB+12b] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y. Gueheneuc, E. Aimeur. Smurf: svm-based incremental anti-pattern detection. 19th Working Conference on Reverse Engineering, 2012, pp. 466-475.

[Mar04] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. Proceedings of the 20th International Conference on Software Maintenance, 2004, pp. 350-359.

[Mar05] R. C. Martin. SRP: The Single Responsibility Principle. ObjectMentor, 2005.

[MGD+09] N. Moha, Y. Gueheneuc, L. Duchien, A. Le Meur. DECOR: A Method for the Specification and Detection of Code and Design Smells. IEEE Transactions on Software Engineering, 2009, pp. 20-36.

[MKM+17] Usman Mansoor, Marouane Kessentini, Bruce R. Maxim, Kalyanmoy Deb. Multi-objective code-smells detection using good and bad design examples. Software Quality Journal 25, no. 2, 2017, pp. 529-552.

[ML06] M. V. Mäntylä, C. Lassenius. Subjective Evaluation of Software Evolvability Using Code Smells: An Empirical Study. Journal of Empirical Software Engineering, vol. 11, no. 3, 2006, pp. 395-431.

[ML18] D. Masters, C. Luschi. Revisiting Small Batch Training for Deep Neural Networks. arXiv:1804.07612v1, 2018.

[MMM+05] C. Marinescu, R. Marinescu, P. Mihancea, D. Ratiu, R. Wettel. iPlasma: An Integrated Platform for Quality Assesment of Object-Oriented Design. Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005, pp. 77-80.

[MR15] N. Mathur, Y. R. Reddy. Correctness of Semantic Code Smell Detection Tools. Proceedings of APSEC 2015 Workshops: QuASoQ, WAWSE & CMCE, 2015.

[Non12] K. Nongpong. Integrating code smells detection with refactoring tool support. Ph.D. dissertation. University of Wisconsin Milwaukee, 2012.

[NPT+18] D. Nucci, F. Palomba, D. Tamburri, A. Serebrenik, A. Lucia. Detecting code smells using machine learning techniques: Are we yet there?. 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering, 2018, pp. 612-621.

[Ola15] C. Olah. Understanding LSTM Networks. 2015. [Žiūrėta 2018-06-13]. Prieiga per internetą: http://colah.github.io/posts/2015-08-Understanding-LSTMs.

[Owa17] F. Owasp. Open web application security Project. 2017. [Žiūrėta 2018-06-13]. Prieiga per internetą: https://owasp.org/index.php/Benchmark.

[PKA08] D. Poo, D. Kiong, S. Ashok. Object-Oriented Programming and Java. Springer, 2008.

[Pla98] J. Platt. Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines. Microsoft Research, 1998.

[PNT+15] F. Palomba, D. Nucci, M. Tufano, G. Bavota, R. Oliveto, D. Poshyvanyk, A. Lucia. Landfill: An Open Dataset of Code Smells with Public Evaluation. IEEE/ACM 12th Working Conference on Mining Software Repositories, 2015.

[PO95] T. Pearse, P. Oman. Maintainability measurements on industrial source code maintenance activities. Proceedings of International Conference on Software Maintenance, 1995.

[PP14] F. S. Panchal, M. Panchal. Review on Methods of Selecting Number of Hidden Nodes in Artificial Neural Network. IJCSMC, vol. 3, no. 11, 2014, pp. 455-464.

[Pre01] R. S. Pressman, Software Engineering – A Practitioner's Approach, 5 th edition. McGraw-Hill Higher Education, 2001.

[Pre96] L. Prechelt. Early Stopping – but when? NIPS workshop, 1996, pp. 55-69.

[PY10] S. J. Pan, Q. Yang. A Survery on Transfer Learning. IEEE Transactions on Knowledge and Data Engineering, vol. 22, no. 10, 2010, pp. 1345-1359.

[Sou] Source Making. The Blob. [Žiūrėta 2019-05-07]. Prieiga per internetą: https://sourcemaking.com/antipatterns/the-blob.

[Qui] QuickUML. [Žiūrėta 2018-06-13]. Prieiga per internetą: https://sourceforge.net/projects/quj.

[RE15] C. Raffel, D. P. W. Ellis. Feed-Forward Networks with Attention Can Solve Some Long-Term Memory Problems. arXiv preprint arXiv:1512.08756, 2015.

[TAD+10] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, J. Noble. The qualitas corpus: A curated collection of java code for empirical studines. Proceedings of the 17th Asia Pacific Software Engineering Conference, 2010, pp. 336-345.

[Vap79] V. Vapnik. Estimation of Dependences Based on Empirical Data. Nauka, 1979.

[Vog16] W. Vogels. Bringing the Magic of Amazon AI and Alexa to Apps on AWS. All Things Distributed, 2016.

[WSC+16] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. arXiv:1609.08144, 2016.

[WTV+16] M. White, M. Tufano, C. Vendome, D. Poshyvanyk. Deep learning code fragments for code clone detection. Proccedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016, pp. 87-98.

[WVL+15] M. White, C. Vendome, M. Linares-Vasquez, D. Poshyvanyk. Toward deep learning software repositories. Proceedings of the 12th Working Conference on Mining Software Repositories, 2015, pp. 334–345.

[Xer] Xerces. [Žiūrėta 2018-06-13]. Prieiga per internetą: https://xerces.apache.org.

[ZHB11] M. Zhang, T. Hall, N. Baddoo. Code bad smells: a review of current knowledge. Journal of Software Maintenance and Evolution: Research and Practice, vol. 23, no. 3, 2011, pp. 179-202.