Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

FIIT-5212-96898

Róbert Jačko

Využitie strojového učenia pri získavaní znalostí zo softvérových repozitárov

Bakalárska práca

Vedúci práce: Ing. Peter Kapec, PhD.

Máj, 2021

Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

FIIT-5212-96898

Róbert Jačko

Využitie strojového učenia pri získavaní znalostí zo softvérových repozitárov

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	9.2.1 Informatika
Miesto vypracovania:	Ústav počítačového inžinierstva a aplikovanej informatiky
Vedúci práce:	Ing. Peter Kapec, PhD.

Máj, 2021

Slovenská technická univerzita v Bratislave
Ústav počítačového inžinierstva a aplikovanej informatiky

Fakulta informatiky a informačných technológií
Akademický rok: 2020/2021
Evidenčné číslo: FIIT-5212-96898

ZADANIE BAKALÁRSKEJ PRÁCE

Študent: **Róbert Jačko**
ID študenta: 96898
Študijný program: informatika
Študijný odbor: informatika
Vedúci práce: Ing. Peter Kapec, PhD.

Názov práce: **Využitie strojového učenia pri získavaní znalostí zo softvérových repozitárov**

Jazyk, v ktorom sa práca vypracuje: slovenský jazyk

Špecifikácia zadania:

V súčasnosti zaznamenávame značný záujem o metódy strojového učenia. Veľmi populárne sú metódy hlbokého učenia, nakoľko umožňujú riešiť komplexné problémy často s lepšími výsledkami ako bežné prístupy. Preskúmajte možnosti ako využiť grafové štruktúry na reprezentáciu informácií a znalostí o softvérových artefaktoch a ako pomocou metód strojového učenia získavať z týchto reprezentácií užitočné znalosti. V rámci práce sa zamerajte na využitie možností znalostných grafov, relačného strojového učenia a grafových neurónových sietí (Graph Neural Network). Navrhnite a realizujte vlastné riešenie pre konkrétne softvérové repozitáre, z ktorých sa pomocou strojového učenia získajú „generické“ znalosti s cieľom využiť ich na vizualizáciu podobných resp. výrazne odlišných čŕt alebo na zvýraznenie zaujímavých resp. dôležitých softvérových artefaktov v repozitári. Navrhnuté riešenie overte na sade existujúcich a testovacích repozitárov.

Rozsah práce: 40

Riešenie zadania práce od: 17. 02. 2020
Dátum odovzdania práce: 17. 05. 2021

Róbert Jačko
študent

Ing. Katarína Jelemenská, PhD.
vedúci pracoviska

doc. Ing. Valentino Vranić, PhD.
garant študijného programu

Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

ZADANIE BAKALÁRSKEHO PROJEKTU

Meno študenta:	**Jačko Róbert**
Študijný odbor:	Informatika
Študijný program:	Informatika
Názov projektu:	**Využitie strojového učenia pri získavaní znalostí zo softvérových repozitárov**

Zadanie:
V súčasnosti zaznamenávame značný záujem o metódy strojového učenia. Veľmi populárne sú metódy hlbokého učenia, nakoľko umožňujú riešiť komplexné problémy často s lepšími výsledkami ako bežné prístupy. Preskúmajte možnosti ako využiť grafové štruktúry na reprezentáciu informácií a znalostí o softvérových artefaktoch a ako pomocou metód strojového učenia získavať z týchto reprezentácií užitočné znalosti. V rámci práce sa zamerajte na využitie možností znalostných grafov, relačného strojového učenia a grafových neurónových sietí (Graph Neural Network). Navrhnite a realizujte vlastné riešenie pre konkrétne softvérové repozitáre, z ktorých sa pomocou strojového učenia získajú „generické" znalosti s cieľom využiť ich na vizualizáciu podobných resp. výrazne odlišných čŕt alebo na zvýraznenie zaujímavých resp. dôležitých softvérových artefaktov v repozitári. Navrhnuté riešenie overte na sade existujúcich a testovacích repozitárov.

Práca musí obsahovať:

Anotáciu v slovenskom a anglickom jazyku
Analýzu problému
Opis riešenia
Zhodnotenie
Technickú dokumentáciu
Zoznam použitej literatúry
Elektronické médium obsahujúce vytvorený produkt spolu s dokumentáciou

Miesto vypracovania: Ústav počítačového inžinierstva a aplikovanej informatiky, FIIT STU, Bratislava
Vedúci projektu: Ing. Peter Kapec, PhD.

Termín odovzdania práce v zimnom semestri : 7.12.2020
Termín odovzdania práce v letnom semestri : 11.5.2021

Bratislava 21.9.2020

doc. Ing. Valentino Vranić, PhD.
poverený riadením ÚISI

Anotácia

Slovenská technická univerzita v Bratislave
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Študijný program:	Informatika
Bakalárska práca:	Využitie strojového učenia pri získavaní znalostí zo softvérových repozitárov
Autor:	Róbert Jačko
Vedúci práce:	Ing. Peter Kapec, PhD.

Máj, 2021

V súčasnosti zaznamenávame značný záujem o metódy strojového učenia. Veľmi populárne sú metódy hlbokého učenia, nakoľko umožňujú riešiť komplexné problémy často s lepšími výsledkami ako bežné prístupy. V tejto práci je opísaný princíp, na ktorom fungujú neurónové siete a podrobnejšie sa venujeme grafovým neurónovým sieťam a technikám slúžiacim na spracovanie zdrojových kódov. Počas práce sme zo zdrojových kódov extrahovali ich reprezentácie vo forme grafov a následne vytvorili model neurónovej siete, ktorou sme tieto grafy klasifikovali do viacerých tried.

Annotation

Slovak University of Technology Bratislava
FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES

Study program:	Informatika
Bachelor thesis:	Use of machine learning in obtaining knowledge from software repositories
Author:	Róbert Jačko
Supervisor:	Ing. Peter Kapec, PhD.
2021, May	

Nowdays we see considerable interest in machine learning methods. Deep learning methods are very popular, as they make it possible to solve complex problems, often with better results than conventional approaches. This work describes the principles on which neural networks work and we focus in more detail on graph neural networks and techniques used to process source codes. During the work, we extracted representations from the source codes in the form of graphs and then created a neural network model, which we used to classify these graphs into several classes.

POĎAKOVANIE

Ďakujem môjmu školiteľovi Ing. Petrovi Kapcovi, PhD. za pravidelné konzultácie, užitočné rady a čas, ktorý mi poskytol pri tvorbe tejto bakalárskej práce.

ČESTNÉ PREHLÁSENIE

Čestne prehlasujem, že som túto prácu vypracoval samostatne s použitím uvedenej literatúry a na základe konzultácií a svojich vedomostí.

.....

Róbert Jačko

Obsah

Zoznam obrázkov

1 Úvod

Umelá inteligencia a strojové učenie vo všeobecnosti v poslednom čase preukazujú pozoruhodné výsledky v mnohých úlohách, od spracovania obrazu po spracovanie prirodzeného jazyka, najmä s nástupom hlbokého učenia [35].

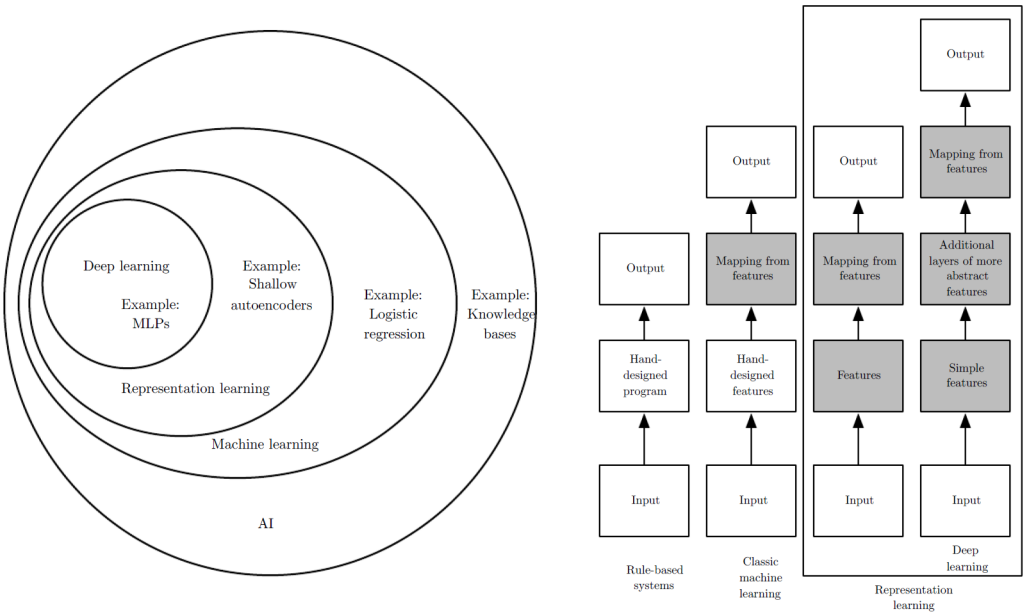
Umelá inteligencia je vlastne snaha umožniť počítačom učiť sa zo skúseností a porozumieť svetu v zmysle hierarchie pojmov, pričom každý pojem je definovaný z hľadiska jeho vzťahu k jednoduchším pojmom. Zhromažďovaním poznatkov zo skúseností tak tento prístup nepotrebuje ľudí aby formálne špecifikovali všetky vedomosti, ktoré počítač potrebuje [4].

Skutočnou výzvou pre umelú inteligenciu je však riešenie úloh, ktoré sú pre človeka ľahko vykonateľné, ale ťažko ich opísať. Sú to problémy, ktoré riešime intuitívne a ktoré sú pre nás automatické, ako napríklad rozpoznávanie hovorených slov alebo tvárí v obrazoch.

V tejto bakalárskej práci je najprv opísaná jedna z oblastí *Umelej inteligencie* a to *Učenie sa reprezentácií*. Spolu s *Učením sa reprezentácií* je zhrnutá aj jej podtrieda *Hlboké učenie*, jeho základné modely a využitie. V ďalšej sekcii sa práca venuje grafovým neurónovým sieťam, kde je takisto opísaný základný model, jeho limity a ich celkové využitie. Neskôr je už práca zameraná konkrétne na spracúvanie zdrojových kódov a ich reprezentáciu. V tejto sekcii sú aj opísané niektoré existujúce riešenia v oblasti klasifikácie, predikcie a hľadaní zraniteľností v zdrojových kódoch.

2 Učenie sa reprezentácií

Výkon a presnosť metód strojového učenia výrazne závisia od voľby reprezentácie (vlastností) údajov, na ktorých sa tieto metódy aplikujú. Dobrá reprezentácia údajov značne uľahčuje získavanie užitočných informácií pri vytváraní klasifikátorov alebo iných prediktorov. Z tohto dôvodu veľká časť úsilia pri nasadzovaní algoritmov strojového učenia smeruje práve do predspracovania a transformácií údajov s cieľom vytvorenia ich reprezentácií [3]. Manuálne navrhovanie reprezentácií pre nejakú zložitú úlohu si však vyžaduje veľmi veľa ľudského času a úsilia. Preto sa zaviedli techniky, ktoré umožňujú strojovému učeniu naučiť sa reprezentáciu samotných dát. Naučené reprezentácie dokonca často vedú k oveľa lepšiemu výkonu, ako je možné dosiahnuť pomocou ručne navrhnutých reprezentácií. Pekný príklad na učenie sa reprezentácií je opísaný v knihe *Deep learning* [4]: *"Vieme, že autá majú kolesá, takže by sme možno chceli prítomnosť kolesa využiť ako nejakú významnú vlastnosť. Bohužiaľ je ťažké presne opísať v hodnotách pixelov ako vyzerá koleso. Koleso má jednoduchý geometrický tvar, ale jeho obraz môžu komplikovať tiene, ktoré naň dopadajú, slnko odrážajúce jeho kovové časti, blatník automobilu alebo predmet v popredí zakrývajúci jeho časť... Jedným z riešení tohto problému je použitie strojového učenia nielen na objavenie mapovania z reprezentácie na výstup, ale aj objavenie samotnej reprezentácie. Tento prístup sa nazýva učenie sa reprezentácie."* Základným modelom učenia sa reprezentácií je autoenkóder, ktorý je ďalej opísaný v sekcii 2.3.4 [4]. Na obrázku 2.1 vľavo je zobrazený Vennov diagram, ktorý zachytáva vzťah medzi hlbokým učením, učením reprezentácií, strojovým učením a umelou inteligenciou.

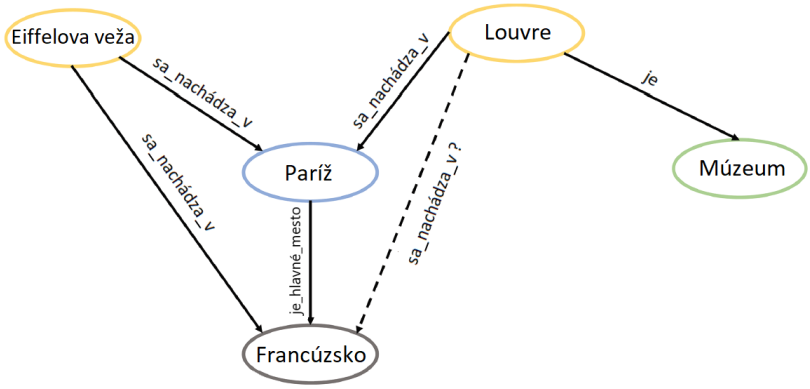


Obr. 2.1: *Diagram naľavo zobrazuje hierarchiu jednotlivých pojmov v umelej inteligencii. Diagram napravo zobrazuje, ako spolu súvisia rôzne časti umelej inteligencie. Šedé políčka označujú komponenty, ktoré sú schopné učiť sa z údajov [4].*

2.1 Znalostné grafy

Znalostné grafy slúžia na reprezentáciu znalostných báz prostredníctvom často orientovaného grafu, ktorého uzly predstavujú entity a jeho hrany predstavujú vzťahy medzi týmito entitami [22]. Entity môžu byť objektmi z reálneho sveta alebo abstraktnými pojmami. Znalostné grafy možno považovať ako vedomostnú bázu pre interpretáciu a odvodzovanie faktov, inými slovami sú to vlastne štruktúrované znázornenia nejakých znalostí. Znalosti môžu byť vyjadrené trojicou `(subjekt, predikát, objekt)` napr. `(Eiffelova veža, sa_nachádza_v, Paríž)`. Často sa používajú grafy vlastností alebo atribútové grafy, v ktorých majú uzly a hrany svoje vlastnosti alebo atribúty. Znalostné grafy sa používajú napr. v oblasti odporúčaní, vyhľadávania na webe a odpovedí na otázky [16] [22]. Na obrázku 2.2 je znázornený príklad znalostného grafu, v ktorom sú uzly (entity) navzájom prepojené orientovanými hranami (vzťahy) a každá z hrán má svoje označenie, ktoré opisuje typ vzťahu

medzi uzlami. Smer orientácie hrán zobrazuje úlohu uzlov. Prerušovaná čiara predstavuje chýbajúcu informáciu (vzťah), ktorú je možné prostredníctvom algoritmov strojového učenia odvodiť.



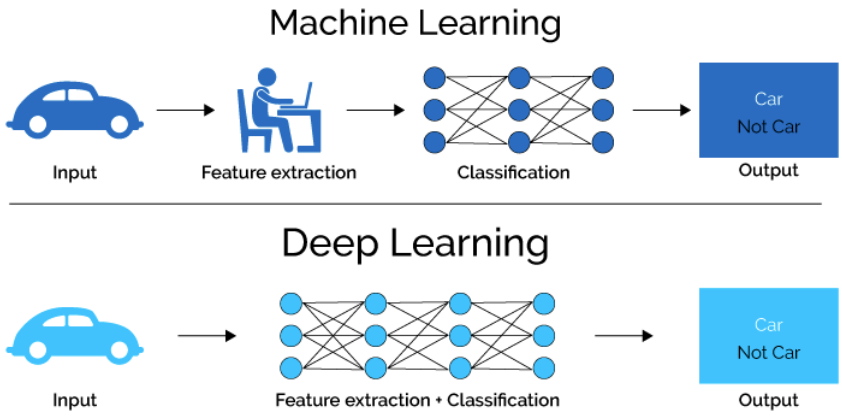
Obr. 2.2: *Príklad znalostného grafu [22].*

2.2 Hlboké učenie

Hlboké učenie je pomerne nová oblasť výskumu Strojového učenia, ktorý bol zavedený s cieľom posunutia strojového učenia bližšie k jednému zo svojich pôvodných cieľov, ktorým je Umelá inteligencia. Prinieslo prielomové výsledky v mnohých výskumných oblastiach. Schopnosť hlbokej neurónovej siete extrahovať jedinečné vlastnosti pre klasifikáciu jej poskytla náskok pred ostatnými technikami strojového učenia [10]. Hlboké učenie je založené na umelých neurónových sieťach, ktoré sú inšpirované štruktúrou a funkciou ľudského mozgu. Zakladá sa na učení viacerých úrovní reprezentácií a abstrakcií dát, ktoré pomáhajú porozumieť údajom ako sú napríklad obrázky, zvuky a texty za účelom modelovania ich zložitejších vzťahov, ktoré sú pre človeka často skryté [9].

Hlboké učenie sa osvedčilo vo viacerých oblastiach a aktívne sa využíva napríkad v medicíne, chémii, robotike, automobilovom priemysle, bioinformatike, pri spracovaní reči a zvukov, klasifikácii dát, rozpoznávaní objektov a v mnoho ďalších odvetviach [4].

Na obrázku 2.3 je zobrazený rozdiel medzi strojovým učením a hlbokým učením na základe extrakcie reprezentácie dát. V strojovom učení musí človek najprv označiť napríklad na obrázkoch základné reprezentačné znaky auta, napríklad kolesá, čím poskytne ďalšie znaky, s ktorými môže systém pracovať. Narozdiel od toho sa v hlbokom učení systém sám pokúša určiť, ktoré časti obrázku tvoria auto, napríklad kolesá, čelné sklo, svetlá atď. Hlboké učenie tak aj môže znížiť množstvo programovania (tzv. hardkódovania), ktoré musia ľudia vykonať na definovanie reprezentácií v datasetoch. Rozdiel je teda ten, že v strojovom učení je nutné manuálne vytvoriť reprezentáciu auta, zatiaľ čo v hlbokom učení stačí označiť, že na obrázku sa nachádza auto a systém sa tak reprezentáciu auta naučí sám.



Obr. 2.3: *Diagram zobrazujúci základný rozdiel medzi strojovým učením a hlbokým učením [19].*

2.3 Umelé neurónové siete

Umelé neurónové siete sú výpočtové systémy, ktoré sú výrazne inšpirované fungovaním biologických nervových systémov (napríklad ľudského mozgu). Skladajú sa z vysokého počtu vzájomne prepojených výpočtových uzlov (neurónov), ktoré pracujú rozmiestnené po celom systéme a spoločne sa učia zo vstupu s cieľom optimalizovať konečný výstup [27].

Neurón je základná jednotka neurónovej siete, ktorá produkuje postupnosť aktivácií. Vstupné neuróny sú aktivované prostredníctvom senzorov

vnímajúcich prostredie a ostatné neuróny sú aktivované prostredníctvom ich vážených vstupov, ktoré vlastne predstavujú výstupy z predošlých aktivovaných neurónov. Výstup neurónu je definovaný jeho aktivačnou funkciou. Klasická štruktúra neurónu je zobrazená na obrázku 2.5.

Aktivačná funkcia sa používa na zavedenie nelinearity do modelovacích schopností siete. Je to funkcia, ktorá mapuje reálne číslo do určitého intervalu (napr. do intervalu (0, 1), kde 0 označuje deaktiváciu a 1 zase plnú aktiváciu neurónu) [22] [29]. Medzi základné aktivačné funkcie patria napríklad:

- **Sigmoid** - normalizuje vstup do intervalu $(0, 1)$.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

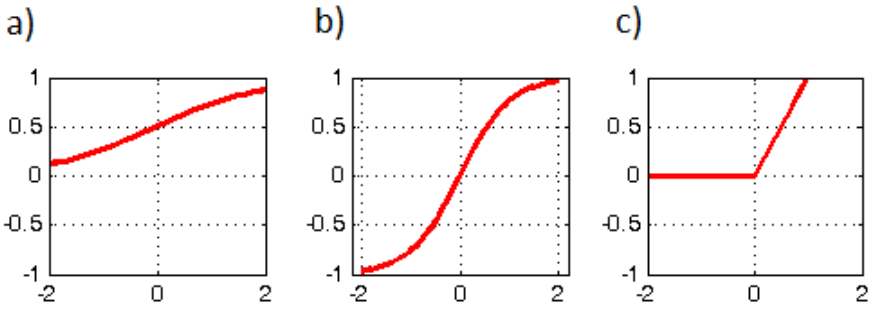
- **Hyperbolický tangens** - normalizuje vstup do intervalu $(-1, 1)$.

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

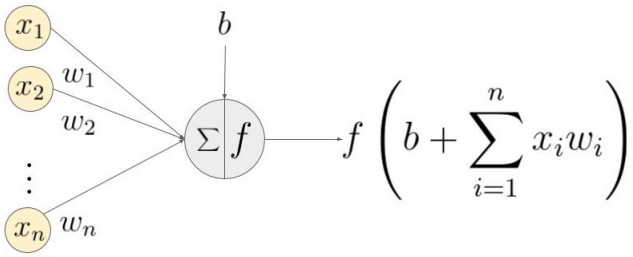
- **ReLU** (Rectified Linear Unit) - vstupné hodnoty, ktoré sú menšie ako nula, nahradí nulou

$$ReLU(x) = \begin{cases} 0, & x \leq 0 \\ x, & x > 0 \end{cases}$$

Na obrázku 2.4 sú zachytené grafické zobrazenia týchto troch funkcií.



Obr. 2.4: *Grafické zobrazenie aktivačných funkcií: a) sigmoid, b) hyperbolický tangens, c) ReLU.*



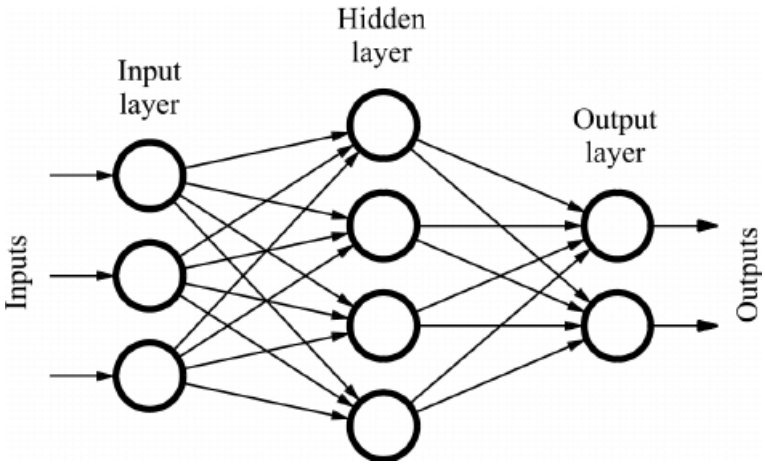
Obr. 2.5: *Zobrazenie klasickej štruktúry neurónu. Vstup neurónu tvoria hodnoty* $\{x_1, x_2, \ldots, x_n\}$*, ktoré sa prenásobia ich príslušnými váhami* $\{w_1, w_2, \ldots, w_n\}$*, následne sa vypočíta suma týchto súčinov a potom sa k nej pripočíta bias* b*. Nakoniec je na tento súčet aplikovaná aktivačná funkcia* f*. Táto konečná hodnota potom tvorí výstup neurónu.*

Vstup neurónových sietí je zvyčajne vo forme viacrozmerného vektora, ktorý je potom rozdelený vstupnou vrstvou do skrytých vrstiev. V dnešnej dobe existuje mnoho typov neurónových sietí, ktoré sa používajú na rôzne účely.

2.3.1 Dopredné neurónové siete

Dopredná neurónová sieť je základný a najjednoduchší typ neurónovej siete. Informácie sa v nej šíria iba jedným smerom (dopredu) zo vstupných uzlov, cez skryté uzly (ak existujú) až do výstupných uzlov. Spojenia medzi jej uzlami

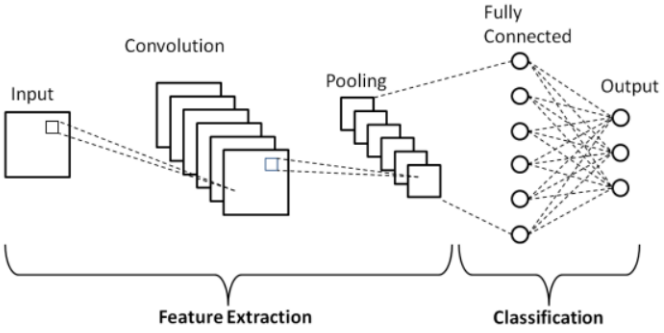
netvoria cyklus: nie sú v nej žiadne spätnoväzbové spojenia, v ktorých by sa výstup z modelu privádzal naspäť do toho istého modelu. Cieľom doprednej siete je aproximovať niektoré funkcie. Dopredné neurónové siete tvoria základ mnohých dôležitých komerčných aplikácií [4].



Obr. 2.6: *Architektúra doprednej neurónovej siete*

2.3.2 Konvolučné neurónové siete

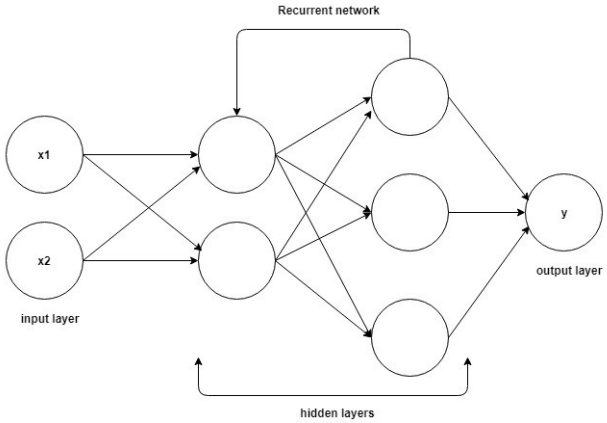
Konvolučná neurónová sieť je špecializovaný druh neurónovej siete na spracovanie dát s mriežkovou topológiou. Typická konvolučná sieť je okrem vstupnej a výstupnej vrstvy zložená z konvolučných vrstiev, pooling vrstiev a plne prepojených vrstiev. Účelom prvej konvolučnej vrstvy je extrakcia bežných vzorov nachádzajúcich sa v lokálnych regiónoch vstupných dát. Pooling vrstvy zase slúžia na redukciu priestorovej dimenzionality, aby sa znížilo množstvo parametrov a výpočtov v sieti [26]. Konvolučné neurónové siete sú schopné extrahovať a skladať viacrozmerne lokalizované priestorové prvky ako prvky s vysokou reprezentačnou silou, následkom čoho nastáva prielom takmer vo všetkých oblastiach strojového učenia [30]. Využívajú sa napríklad pri spracovaní obrázkov, reči, textu alebo aj pri objavovaní nových liekov [26].



Obr. 2.7: *Architektúra konvolučnej neurónovej siete*

2.3.3 Rekurentné neurónové siete

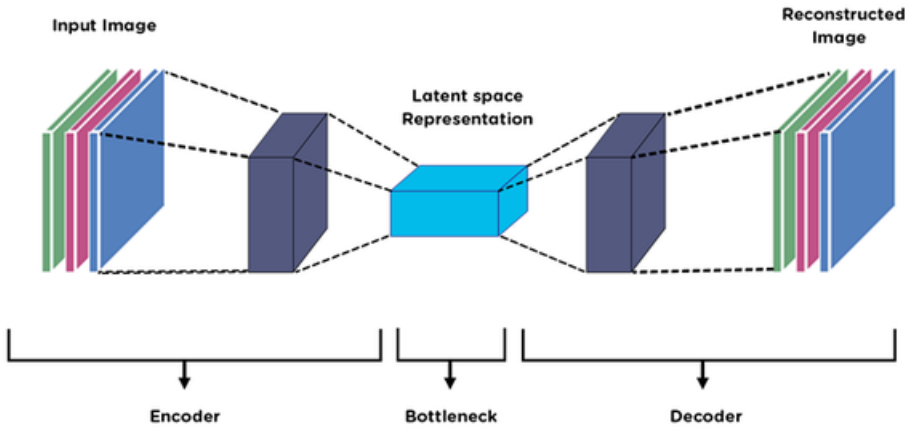
Rekurentné neurónové siete sú určené na spracovanie sekvenčných dát. Sú navrhnuté tak, aby sa naučili postupné alebo časovo odlišné vzory. Rekurentná sieť je neurónová sieť so spätnou väzbou (uzavretá slučka). Výstup neurónu môže byť vstupom nejakého neurónu nachádzajúceho sa v rovnakej vrstve alebo dokonca v predchádzajúcich vrstvách. Využívajú sa pri problémoch s učením postupností ako sú napríklad spracovanie jazyka, kompozícia hudby alebo predpovedanie predajov [4] [20].



Obr. 2.8: *Architektúra rekurentnej neurónovej siete*

2.3.4 Autoenkódery

Autoenkóder je v podstate neurónová sieť, ktorá je trénovaná tak, aby sa pokúsila skopírovať svoj vstup na svoj výstup. Obsahuje skrytú vrstvu, ktorá opisuje kód použitý na reprezentáciu vstupu. Pozostáva z dvoch častí: enkóder (funkcia $h = f(x)$) a dekóder, ktorý produkuje rekonštrukciu ($r = g(h)$). Zvyčajne sa enkódery využívali na redukciu dimenzionality alebo učenie funkcií. O enkóderoch sa dá uvažovať ako o zvláštnom prípade dopredných sietí a môžu sa trénovať rovnakými technikami. Autoenkódery je možné trénovať aj pomocou recirkulácie, trénovacieho algoritmu založeného na porovnaní aktivácií siete na pôvodnom vstupe a aktivácií na rekonštruovanom vstupe [4]



Obr. 2.9: *Architektúra autoenkódera*

2.4 Trénovanie neurónovej siete

Algoritmus strojového učenia je algoritmus, ktorý je schopný učiť sa z údajov [4]. *"O počítačovom programe môžeme povedať, že sa učí zo skúsenosti E v súvislosti s niektorými triedami úloh T a výkonom P, ak sa jeho výkon P v úlohách T so skúsenosťami E zvyšuje."*(Mitchell, 1997, [23])

Pri učení v neurónových sieťach nastávajú dva zásadné problémy, a to tré-

novanie architektúry siete a trénovanie parametrov siete. Zatiaľ čo trénovanie architektúry siete zostáva stále otvorenou otázkou, pre trénovanie parametrov siete existuje efektívny algoritmus [31].

Trénovanie neurónovej siete môžeme teda definovať ako učenie sa hodnôt parametrov (váhy a bias) a tento proces môžeme brať ako iteratívny proces dopredného a spätného šírenia informácií cez vrstvy neurónov. Počas tréningu neurónovej siete sa najčastejšie používa algoritmus spätného šírenia (backpropagation). Je to algoritmus založený na optimalizácií parametrov na základe klesajúceho gradientu. Funkcia, ktorú takto chceme minimalizovať sa volá **stratová (chybová) funkcia**.

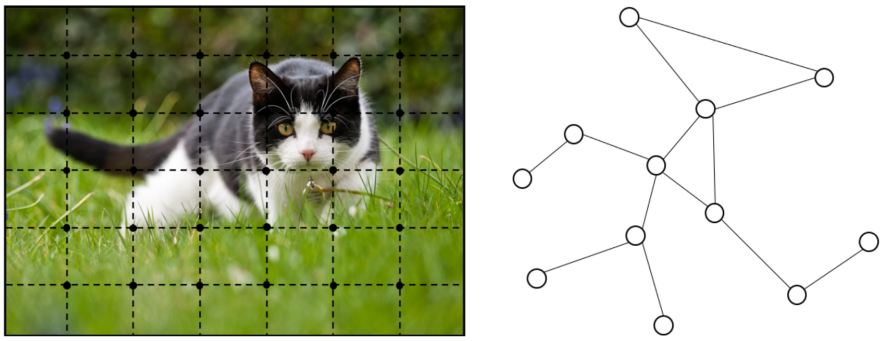
Stratová funkcia sa používa na odhad chyby a na porovnanie a meranie toho, aký dobrý, resp. zlý, bol náš výsledok predikcie vo vzťahu k správnemu výsledku. Na základe tejto funkcie sa počas tréningu modelu postupne upravujú váhy vzájomných prepojení neurónov, až kým sa nedosahujú uspokojivé predikcie [29] [31].

Existujú dve základné metódy trénovania neurónových sietí: učenie s učiteľom (supervised learning) a učenie bez učiteľa (unsupervised learning). Pri učení s učiteľom sa v datasete nachádzajú údaje, ktoré sú označené a model sa trénuje porovnávaním jeho predikovaného výstupu s týmto označením. Pri učení bez učiteľa dáta nie sú označené a model sa snaží zachytiť ich štruktúru. Výsledkom učenia bez učiteľa je teda model, ktorý obsahuje reprezentáciu trénovacích dát. Okrem týchto dvoch základných metód trénovania neurónových sietí existujú aj iné metódy ako napríklad čiastočné učenie s učiteľom (semi-supervised learning) [42] a učenie s posilňovaním (reinforcement learning) [34].

3 Grafové neurónové siete

Grafmi môžeme reprezentovať veľké množstvo systémov z rôznych oblastí. Sú to užitočné dátové štruktúry, ktoré sa vyskytujú v zložitých reálnych aplikáciách, ako sú napríklad modelovanie fyzikálnych systémov, učenie sa molekulárnych odtlačkov prstov, ovládanie dopravných sietí a odporúčanie priateľov v sociálnych sieťach. Tieto úlohy si však vyžadujú prácu s neeuklidovskými grafmi, ktoré obsahujú rozsiahle informácie o vzťahoch medzi prvkami a ktoré sa nedajú dobre spracovať tradičnými modelmi hlbokého učenia [22]. Rozdiel medzi euklidovským a neeuklidovským priestorom je zobrazený na obrázku 3.1.

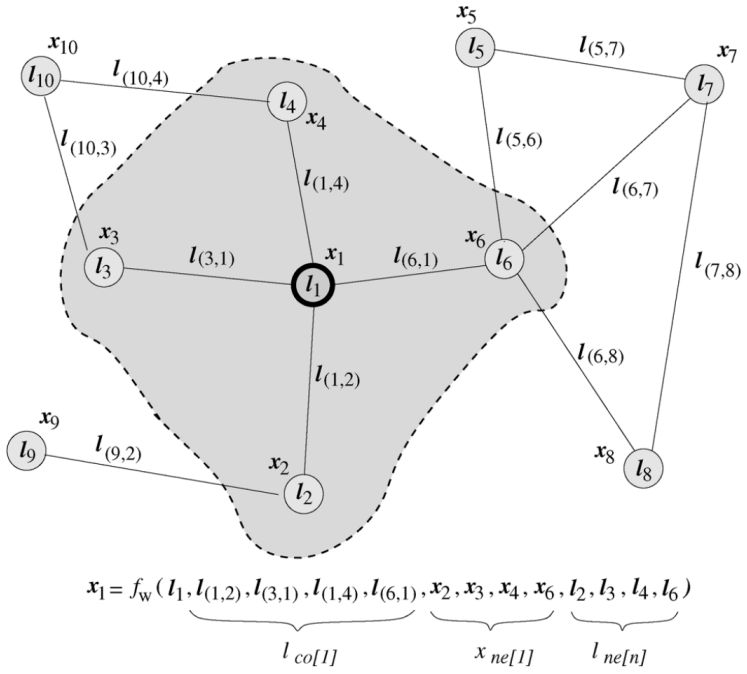
Graf je usporiadaná dvojica, $G = (V, E)$, kde $V = \{1, ..., |V|\}$ je množina uzlov a $E \subseteq \{V \times V\}$ je množina hrán (vzťahov). Hrany grafu môžu byť orientované alebo neorientované. Neorientované hrany sú definovane neusporiadanou dvojicou vrcholov $\{u, v\}$, orientované hrany sú definované usporiadanou dvojicou vrcholov u, v, kde u je začiatočný a v koncový vrchol. **Stupeň vrcholu** v, je počet hrán spojených s v [30].



Obr. 3.1: *Zobrazenie euklidovského priestoru (obrázok vľavo), kde sú pixely rozmiestnené v dvojrozmernej mriežke (ako tretí rozmer môžeme považovať farbu daného pixelu) a grafu v neeuklidovskom pristore (obrázok vpravo), kde graf nevieme zobraziť v n-rozmernom priestore bez toho, aby sme prišli o nejaké podkladové informácie [5] [40].*

Grafové neurónové siete sú metódy založené na hlbokom učení, ktoré sa využívajú v grafovej doméne. Vďaka svojmu výkonu a vysokej interpretovateľnosti sú v poslednej dobe široko používanou metódou analýzy grafov [22]. Zachytávajú závislosti grafov prostredníctvom výmeny správ medzi ich uzlami. Na rozdiel od štandardných neurónových sietí si grafové neurónové siete zachovávajú stav, ktorý môže reprezentovať informácie z jeho okolia s ľubovoľnou hĺbkou. Základný model grafu, ktorý sa využíva pri grafových neurónových sieťach je zobrazený na obrázku 3.2 [40].

Uzol grafu je prirodzene definovaný svojimi vlastnosťami a súvisiacimi uzlami v grafe. Cieľom grafovej neurónovej siete je naučiť sa pre každý uzol zapúzdrenie jeho stavu (angl. state embedding) $h_v \in \mathbb{R}^s$, ktorý kóduje informácie z okolia daného uzla. Zapúzdrenie stavu h_v je s-dimenzionálny vektor uzla v a používa sa na vytvorenie výstupu o_v, ako je napríklad distribúcia predikovaného označenia uzla [28] [40].



Obr. 3.2: *Zobrazenie základného modelu grafu a susedov uzla l_1: uzly sú označené ako l_n a hrany ako $l_{(n_1,n_2)}$, $ne[l]$ prestavuje susedov uzla l, $co[l]$ označuje množinu hrán, ktoré majú l ako vrchol. Vlastnosti x_1 uzla l_1 závisia od informácií obsiahnutých v jeho susedných uzloch [28].*

3.1 Základný model grafovej neurónovej siete

Základný model grafovej neurónovej siete bol definovaný v článku [28] a zhrnutý v článkoch [22] a [40] a celá táto sekcia sa venuje jeho opisu.

Na aktualizáciu stavu uzla na základe jeho vstupu, ktorý predstavujú jeho susedné uzly, je medzi všetkými uzlami zdieľaná parametrická funkcia f, ktorá sa nazýva *funkcia lokálneho prechodu*. Zapúzdrenie stavu h_v je potom definované ako:

$$h_v = f(x_v, x_{co[v]}, h_{ne[v]}, x_{ne[v]}),$$

kde x_v, $x_{co[v]}$, $h_{ne[v]}$, $x_{ne[v]}$ sú vlastnosti uzla v, vlastnosti jeho hrán, stavy jeho susedných uzlov a vlastnosti jeho susedných uzlov v uvedenom poradí. Na vytvorenie výstupu o_v z daného uzla existuje parametrická funkcia g, ktorá sa nazýva *funkcia lokálneho výstupu*:

$$o_v = g(h_v, x_v).$$

Výpočty funkcie lokálneho prechodu f a funkcie lokálneho výstupu g možno interpretovať ako dopredné neurónové siete. Naskladaním všetkých stavov, výstupov, vlastností a vlastností uzlov sa vytvoria vektory H, O, X a X_N v uvedenom poradí. Získa sa tak *funkcia globálneho prechodu* F a *funkcia globálneho výstupu* G. F a G sú naskladané verzie f a g pre všetky uzly v grafe.

$$H = F(H, X)$$

$$O = G(H, X_N)$$

Grafová neurónová sieť používa na výpočet stavu túto klasickú iteračnú schému:

$$H^{t+1} = F(H^t, X),$$

kde H^t označuje t-tú iteráciu H a X označuje všetky naskladané vlastnosti. Učenie parametrov pre funkcie lokálneho prechodu f a funkcie lokálneho výstupu g s označením t_v pre každý vrchol (učenie s učiteľom) je vykonávané

pomocou stratovej funkcie:

$$loss = \sum_{n=1}^{p} (t_i - o_i),$$

kde p je počet uzlov, učených s učiteľom, t_i je cieľové označenie uzla i a o_i je výstup uzla i. Algoritmus učenia je založený na stratégii klesania gradientu a skladá sa z nasledujúcich krokov:

- Stavy h_v^t sú iteratívne aktualizované funkciou lokálneho prechodu až do časového kroku T. Potom je získané približné riešenie $H(T) \approx H$
- Zo stratovej funkcie sa vypočíta gradient váh W
- Váhy W sa aktualizujú podľa gradientu vypočítaného v predchádzajúcom kroku

Aj keď sa grafová neurónová sieť preukázala ako výkonná architektúra na modelovanie štrukturálnych údajov, základná grafová neurónová sieť má stále svoje obmedzenia. Je výpočtovo neefektívne iteratívne aktualizovať skryté stavy uzlov, aby sa získal ich pevný bod H. V iterácii sa používajú rovnaké parametre, zatiaľ čo populárne neurónové siete používajú rôzne parametre v rôznych vrstvách, čo slúži na extrakciu hierarchických prvkov. Hrany tiež obsahujú niektoré informatívne prvky, ktoré sa v základnej grafovej neurónovej sieti nedajú efektívne modelovať. Napríklad hrany v znalostnom grafe majú typ vzťahov a šírenie správy cez rôzne hrany by sa malo líšiť podľa ich typov.

3.2 Využitie grafových neurónových sietí

Existuje mnoho aplikácií v reálnom svete, ktoré zahŕňajú analýzu grafovo štruktúrovaných údajov ako sú sociálne siete, molekulárne štruktúry, znalostné grafy atď. Grafová neurónová sieť je navrhnutá špeciálne na spracovanie takýchto štruktúrovaných údajov [22]. Grafové neurónové siete majú potenciál v mnohých odvetviach a aktívne sa využívajú v chémii pri tvorbe nových mole-

kúl, vo fyzikálnych systémoch na predikciu trajektórie objektov, pri klasifikácii obrázkov alebo pri kombinatorickej optimalizácii.

Napríklad pri objavovaní liekov je molekula reprezentovaná súborom atómov (uzlov) a hraníc (hrán) medzi atómami.

Dajú sa využiť aj pri grafoch scény [38]. Graf scény je všeobecná dátová štruktúra, v ktorej uzly grafu predstavujú objekty a hrany predstavujú vzťahy medzi nimi. Je to veľmi užitočná reprezentácia pri mnohých úlohách, ako sú napríklad generovanie titulkov k obrázkom alebo odpovedanie na vizuálne otázky [17].

Pri spracovaní prirodzeného jazyka [7] predstavuje znalostný graf súbor vzájomne prepojených popisov entít, ktoré sú užitočné pri úlohách, ako je napríklad odpovedanie na otázky v textovej forme.

V internetových obchodoch môže model grafickej neurónovej siete vytvárať presné odporúčania učením sa vzťahov medzi zákazníkmi a produktmi [41].

V sociálnej sieti môžu byť jednotlivci, organizácie a ich sociálne interakcie taktiež efektívne znázornené grafom [11].

4 Spracovanie zdrojových kódov

S nedávnym úspechom v oblasti hlbokého učenia sa stáva čoraz populárnejším aplikovať ho na obrázky, prirodzený jazyk, audio a podobne. Jedným z kľúčových aspektov tohto výskumu je výpočet numerického vektora predstavujúceho predmet záujmu. Tento vektor sa nazýva zapúzdrenie (angl. embedding). Pokiaľ ide o slová v prirodzenom jazyku, vedci prišli na súbor techník nazývaných zapúzdrovanie slov, ktoré mapujú slová alebo frázy na vektor reálnych čísel [6].

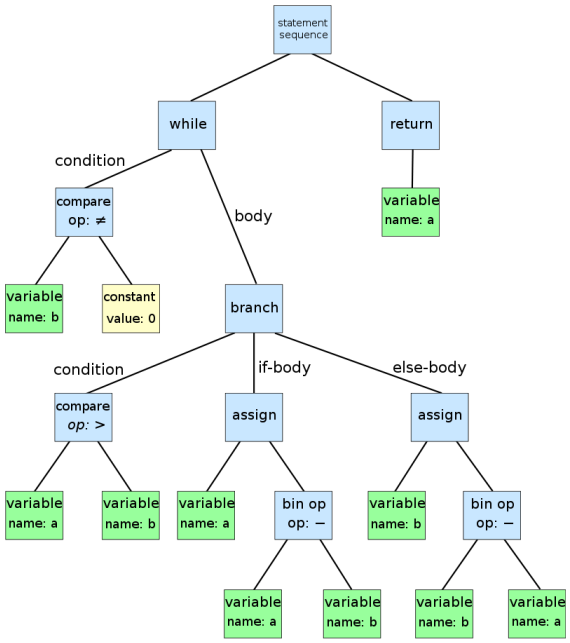
Metódy hlbokého učenia sa začali používať aj na spracúvanie zdrojových kódov (t.j. formálnych jazykov) [1]. Na zlepšenie vývoja a údržby softvéru bolo navrhnutých veľa metód softvérového inžinierstva, ako je klasifikácia zdrojového kódu [25], detekcia klonov kódu [37], predikcia chýb [36], sumarizácia kódu [15], generovanie komentárov [13]... Jednou z hlavných výziev, ktorá je spoločná pre všetky tieto metódy, je spôsob, ako reprezentovať zdrojový kód, aby bolo možné efektívne zachytiť syntaktické a sémantické informácie, ktoré zdrojový kód obsahuje [39].

4.1 Reprezentácia zdrojového kódu

Spočiatku sa väčšina prác pokúšala pristupovať k zdrojovému kódu pomocou metód prirodzeného jazyka [32] a nevyťažili tak z jedinečných možností, ktoré ponúka známa sémantika zdrojového kódu. Napríklad neuvažovali so závislosťami na veľké vzdialenosti spôsobenými použitím rovnakej premennej alebo funkcie vo vzdialených lokalitách v zdrojovom kóde [1]. Pri spracúvaní zdrojových kódov ako textov v prirodzenom jazyku chýbajú dôležité sémantické informácie zdrojového kódu a mnoho jeho aspektov, ako sú napríklad názvy, formátovanie alebo lexikálne poradie metód nemá žiadny vplyv na sémantiku programu. V poslednej dobe však najnovšie štúdie ukazujú, že modely založené na abstraktných syntaktických stromoch (AST) môžu oveľa lepšie reprezentovať zdrojový kód [39]. Na obrázku 4.1 je zobrazený AST vytvorený

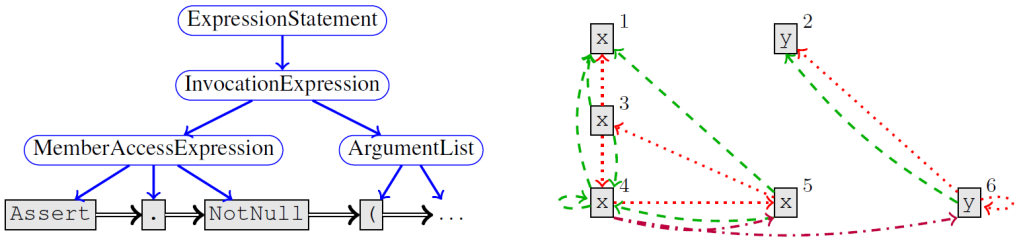
z príslušného úseku kódu.

```
while b ≠ 0
  if a > b
    a := a − b
  else
    b := b − a
return a
```



Obr. 4.1: *Zobrazenie úseku kódu (vľavo) a abstraktného syntaktického stromu (vpravo), ktorý bol vytvorený na základe tohoto kódu.*

Veľkosti AST sú však zvyčajne veľmi veľké a základné modely sú náchylné na problém vzdialených závislostí v kóde [39]. V základnom AST nie sú dobre zachytené volania rovnakých funkcií a používanie rovnakých premenných v rôznych miestach kódu. Preto sa na reprezentáciu zdrojových kódov začali používať grafy, ktoré tieto vlastnosti kódu zachytiť dokážu [1]. Príklad takejto reprezentácie je zobrazený na obrázku 4.2.



Obr. 4.2: *Zobrazenie grafovej reprezentácie kódu [1]. V syntaktickom grafe (vľavo) sú syntaktické uzly ohraničené modrým oválom a syntaktické tokeny čiernym obdĺžnikom. Modré hrany sú prepojenia smerom na potomkov a čierne dvojité hrany prepájajú tokeny. Na obrázku vpravo sú zobrazené hrany toku údajov pre úsek kódu* $(x^1, y^2) = Foo();\ while(x^3 > 0)\ x^4 = x^5 + y^6;$ *(indexy pri premenných boli pridané kvôli prehľadnosti). Červené bodkované hrany zobrazujú posledné použitie, zelené prerušované hrany zobrazujú posledný zápis a prerušované fialové hrany zobrazujú odkiaľ boli hodnoty vypočítané*

4.2 Existujúce riešenia

4.2.1 Identifikácia programovacieho jazyka

V práci [12] ukázali, že identifikáciu programovacieho jazyka je možné vykonať automaticky využitím umelej neurónovej siete založenej na učení s učiteľom a inteligentnej extrakcii reprezentácií zo zdrojových kódov. Použili pri tom viacvrstvovú neurónovú sieť, ktorá obsahovala vrstvy na zapúzdrovanie slov a konvolučnú neurónovú sieť. Na trénovanie modelu použili dataset vytvorený z tisícov súborov zdrojových kódov z úložísk na stránke GitHub a na označenie každej tejto vzorky jej programovacím jazykom boli použité prípony názvov týchto súborov (t.j. subor.py bol označený ako zdrojový kód v jazyku Python). Celkovo použili 1 183 242 súborov zdrojových kódov a pre každý jazyk použili maximálne 20 000 príkladov, aby tak zabránili nevyváženému trénovaniu. V súboroch, ktoré obsahovali aj iné jazyky ako ich primárny jazyk získaný z ich prípon (napr. HTML, CSS, JavaScript...) tieto časti odstránili na základe známych výrazov a vyhradených slov pre každý jazyk tak, aby sa v každom dokumente nachádzal práve jeden programovací jazyk. Nakoniec nahradili všetky texty medzi úvodzovkami reťazcom *"strv"* a vymazali všetky znaky

nových riadkov a tabulátorov.

Aby stroj porozumel jazyku, musí najskôr vytvoriť myšlienkovú mapu pojmov, ich definícií a asociácií s inými výrazmi. To si vyžaduje vytvorenie slovníka takýchto výrazov a pochopenie ich vzájomných vzťahov, logicky aj sémanticky. Za týmto účelom sa generujú zapúzdrenia slov. Jedná sa o mapovanie každého výrazu v slovníku na usporiadanie čísel vo viacrozmernom priestore. Podobné pojmy sú si v tomto priestore navzájom bližšie, zatiaľ čo rozdielne pojmy sú si vzdialené [12]. Na tokenizáciu použili regex:

[\w']+|[""!"#$%&'()*+,-./:;<=>?@[\]^_‘{|}~""\\]

Takto vygenerované tokeny fungujú ako vektory reprezentácií používané na klasifikáciu. Na obrázku 4.3 je zobrazené predspracovanie a tokenizácia, ktorou prejdú všetky zdrojové kódy v jednolivých súboroch.

Zdrojový kód v jazyku Python

```
print "This is an example program."
for dir in os.getcwd():
   print dir
```

Predspracovanie

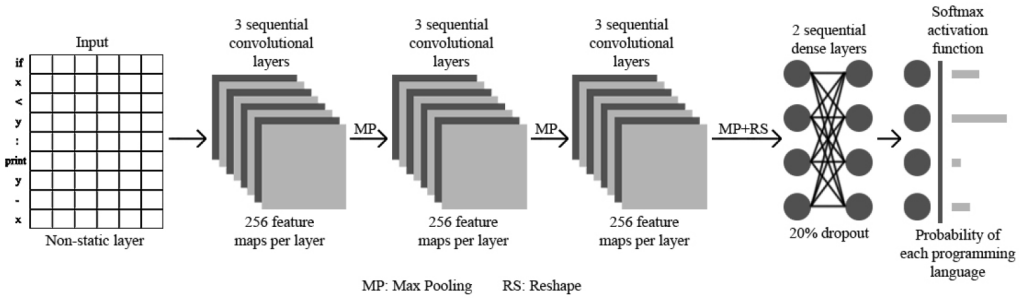
```
print strv for dir in os.getcwd():print dir
```

Tokenizácia

```
['print' , 'strv' , 'for' , 'dir' , 'in' , 'os
    ' , '.' , 'getcwd' , '(' , ')' , ':' , '
    print' , 'dir']
```

Obr. 4.3: *Príklad predspracovania a tokenizácie zdrojového kódu [12].*

Na obrázku 4.4 je zobrazený model klasifikátora programovacieho jazyka, ktorý použili pri riešení tohoto problému. Model obsahuje vrstvu na vkladanie slov nasledovanú viacvrstvovou konvolučnou sieťou s viacerými filtrami, vrstvy ReLU, vrstvy max-poolingu, plne prepojené vrstvy a nakoniec vrstvu softmax.



Obr. 4.4: *Model neurónovej siete, ktorý bol použitý na klasifikáciu [12].*

Sieť trénovali dopredným šírením tréningových údajov a následným spätným šírením chýb. Použili 20% riedenie, aby tak zabránili pretrénovaniu.

Na trénovanie použili 1 064 918 a na validáciu 118 324 zdrojových kódov. Celkové výsledky pre presnosť, precíznosť, návratnosť a f1-skóre sú pri klasifikácii 60 jazykov 97%, 96%, 96% a 96% v uvedenom poradí a čas vykonania tejto identifikácie programovacieho jazyka je v priemere približne 0,1 sekundy na jeden súbor so zdrojovým kódom.

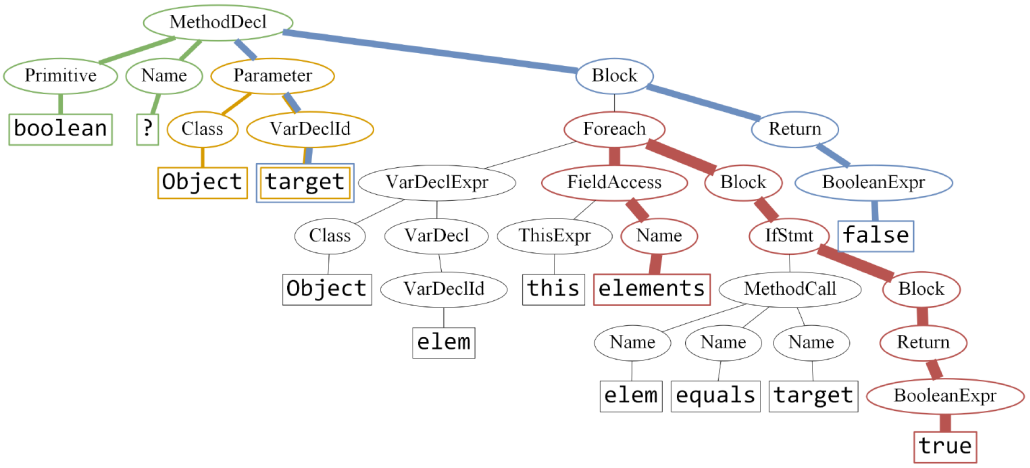
4.2.2 Predikcia názvov metód

V práci [2] s názvom *code2vec* sa rozhodli zdrojový kód reprezentovať ako spojitý distribuovaný vektor. Hlavnou myšlienkou tejto práce bolo reprezentovať útržky kódov ako kolekciu ciest v AST a rozumne a škálovateľne tieto cesty agregovať do jedného vektora kódu s fixnou dĺžkou. Tieto vektory potom použili na predikciu sémantických vlastností týchto častí kódov. Ukázali, že ich riešenie dokázalo predikovať názvy metód zo súborov, ktoré vôbec neboli použité počas tréningu. Na obrázku 4.5 sú zobrazené tri metódy napísane v jazyku Java. Tieto metódy majú veľmi podobnú syntaktickú štruktúru: všetky majú jeden parameter s názvom *target* iterujú cez pole s názvom *elements* a majú vo vnútri cyklu podmienku *if*. Hlavný rozdiel spočíva v tom, že metóda (a) vracia hodnotu *true*, keď prvky obsahujú hľadaný objekt inak false, metóda (b) vracia objekt, ktorého hash kód sa rovná hľadanému hash kódu a metóda (c) vracia index hľadaného objektu. Napriek podobnej syntaktickej štruktúre model dokáže predikovať názvy metód: *contains*, *get* a *indexOf*

v uvedenom poradí.

Obr. 4.5: *Príklad predikcie názvov troch metód, ktoré majú veľmi podobnú syntaktickú štruktúru, ale model aj tak úspešne zachytáva ich sémantické rozdiely a dokáže predikovať zmysluplné názvy. Šírky farebných čiar sú úmerné pozornosti, ktorá bola každej ceste venovaná. [2].*

Obrázok 4.6 zobrazuje kontexty ciest v AST vytvoreného z metódy z obrázku 4.5(a). Hrúbka farebných čiar je úmerná pozornosti, ktorú model venoval každému kontextu cesty.



Obr. 4.6: *[2].*

Pri **extrakcii cesty** sa najskôr každá metóda v tréningovej sade parsuje tak, aby sa vytvorili AST. Potom sa AST prehľadajú a extrahujú sa syntaktické cesty medzi ich listami. Každá cesta je reprezentovaná ako postupnosť uzlov

AST prepojených šípkami nahor a nadol, ktoré symbolizujú prepojenie nahor k rodičom alebo nadol k potomkom medzi susednými uzlami v strome. Cesta je reprezentovaná ako trojica (x_s, p, x_f), kde x_s je list AST, v ktorom, cesta začína, p je cesta, ktorá sa musí prejsť aby sme sa dostali ku koncovému listu a x_f je koncový list AST. Táto cestu sa označuje ako *kontext cesty*. Príklad takéhoto kontextu cesty z obrázka 4.6 je:

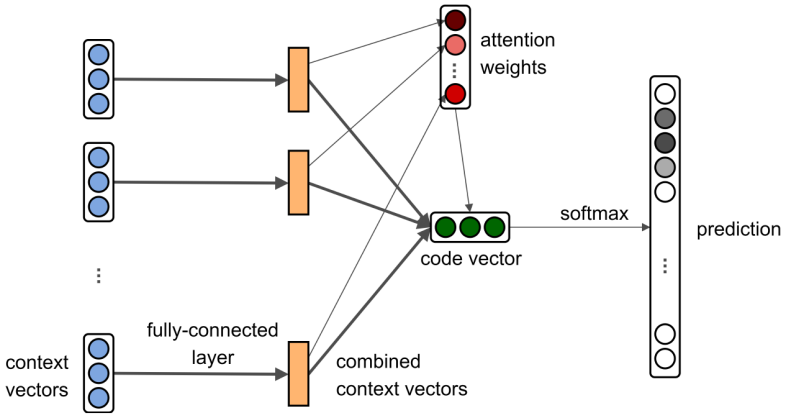
$$(elements, \text{Name}\uparrow\text{FieldAccess}\uparrow\text{Foreach}\downarrow\text{Block}\downarrow\text{IfStmt}\downarrow\text{Block}\downarrow\text{Return}\downarrow\text{BooleanExpr}, true),$$

kde *elements* a *true* sú listy stromu a výraz medzi nimi je cesta, ktorú treba prejsť aby sme sa dostali z jedného listu do druhého.

Distribuovaná reprezentácia kontextov sa získa tak, že sa každá hodnota z trojice, ktorou je reprezentovaná cesta (oba listy a cesta medzi nimi) namapuje na jej zodpovedajúcu vektorovú reprezentáciu. Následne sa tieto tri vektory každej cesty zreťazia do jedného vektora, ktorý reprezentuje kontext cesty.

Sieť zameraná na cestu agreguje viacero zapúzdrení kontextov ciest do jedného vektora, ktorý predstavuje celé telo metódy. Zameranie je mechanizmus, ktorý sa učí ohodnotiť každý kontext cesty tak, že sa vyššia pozornosť odrazí na vyššom skóre. Tieto viacnásobné zapúzdrenia sa agregujú pomocou skóre zamerania do jedného vektora kódu. Sieť potom predpovedá pravdepodobnosť pre každý názov cieľovej metódy daný vektorom kódu.

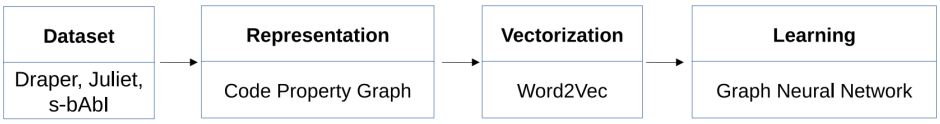
Model siete zameranej na cestu je zobrazený na obrázku 4.7. Plne prepojená vrstva sa učí kombinovať vloženie každého kontextu cesty. Váhy pozornosti sa učia pomocou kombinovaných kontextových vektorov a používajú sa na výpočet vektora kódu. Vektor kódu sa používa na predikciu označenia.



Obr. 4.7: *Architektúra siete zameranej na cestu [2].*

4.2.3 Detekcia zraniteľností v kóde

V práci [33] skúmali použiteľnosť grafových neurónových sietí na zisťovanie zraniteľností zdrojových kódov. Zamerali sa na zdrojové kódy napísané v programovacom jazyku C. V tejto práci neriešili lokalizáciu zraniteľností, iba sa pokúšali zistiť, či je kód zraniteľný alebo nie. Na obrázku 4.8 je zobrazené zreťazené spracovanie (angl. pipeline) s názvom *AI4VA*, ktoré použili na zdrojové kódy. Na vstupe berie tieto surové zdrojové kódy a na výstupe ich klasifikuje ako zraniteľné alebo nie.



Obr. 4.8: *Zobrazenie AI4VA pipeline. Draper, Juliet a s-bAbI sú názvy označených datasetov, ktoré použili pri trénovaní a validácii neurónovej siete [33].*

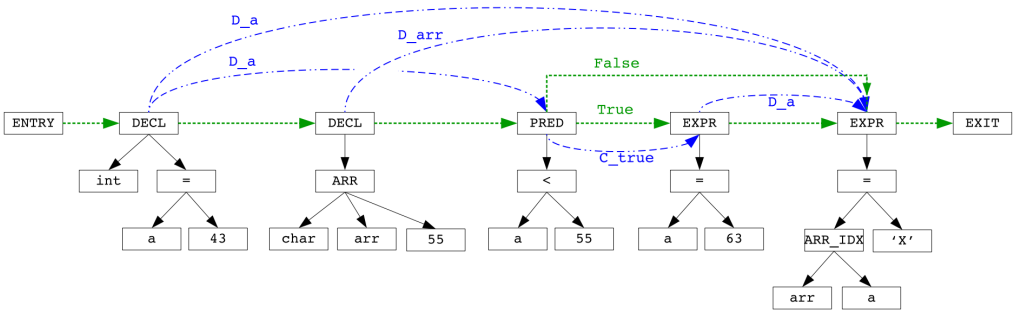
Extrakciu reprezentácií zdrojových kódov nerobili manuálne ale použili automatickú extrakciu reprezentácií, ktorú umožňujú neurónové siete. Zamerali sa na konverziu zdrojového kódu na vhodnú reprezentáciu, v ktorej sa zachovajú sémantické informácie a ktorú môže neurónová sieť vhodne spracovať. Použili grafovú reprezentáciu zdrojového kódu, ktorá je pre zdrojový kód

prirodzená a dokáže zachytiť jeho sémantické informácie. Konkrétne použili grafy vlastností kódov (angl. code property graphs). Graf vlastností kódu je v podstate kombináciou AST, grafu toku (angl. control flow graph) a grafu závislosti programu (angl. program dependency graph) extrahovaných zo zdrojového kódu. Na túto konverziu každého zdrojového kódu do zodpovedajúceho grafu vlastností kódu použili open-source nástroj s názvom *Joern*.

Na obrázku 4.9 je zobrazený príklad zdrojového kódu, v ktorom sa v poslednom riadku nachádza chyba spôsobená pretečením zásobníka. Na obrázku 4.10 je zobrazený graf vlastností kódu, ktorý bol vytvorený na základe tohoto zdrojového kódu.

```
void foo()
{
   int a = 43
   char arr[55];
   if (a < 55)
   {
     a = 63;
   }
   arr[a] = 'X';
}
```

Obr. 4.9: *Príklad zdrojového kódu s chybou [33].*



Obr. 4.10: *Graf vlastností kódu. Čierne plné hrany zobrazujú AST, zelené prerušované hrany zobrazujú graf toku a modré prerušované hrany zobrazujú graf vlastností kódu. Hrana s označením D_a zobrazuje závislosť údajov od podstromu, ktorý definuje premennú a, k pod stromu, ktorý používa túto definovanú hodnotu. Zelené hrany zachytávajú poradie vykonávania programu [33].*

Následne konvertovali tieto reprezentácie zdrojových kódov do vekto-

rových reprezentácií potrebných pre neurónovú sieť. Hrany konvertovali na trojicu `{id_zdrojového_uzla, typ_hrany, id_cieľového_uzla}`. Trojice všetkých hrán sú potom spracované neurónovou sieťou, ktorá z nich vytvorí matice susednosti na základe typov hrán. Všetky uzly, ktoré môžu obsahovať rôzne informácie v závislosti od ich zdrojov (AST, graf toku, graf závislosti programu) sa prevedú na n-dimenzionálne vektory. Na tento prevod použili program *Word2vec* kvôli zachovaniu sémantických atribútov kódu a jazyka.

Fázu učenia založili na hypotéze, že ak pre kód existuje prirodzená štruktúra, malo by byť možné ju využiť na automatické zistenie, či určitá časť štruktúry kódu súvisí s konkrétnymi typmi zraniteľnosti. Použili pri tom najmodernejšiu grafovú neurónovú sieť [21]. Vektorová reprezentácia každého uzla funguje ako počiatočná informácia, ktorú nesie. Váženým súčtom výstupov od bezprostredných susedov sú vytvorené nové vstupy pre každý uzol. Počas procesu tréningu sa model učí hodnoty váh, ktoré sa priradia rôznym uzlom a hranám.

Takto natrénovaný model otestovali na súboroch z troch rôznych datasetov a porovnali s ostatnými existujúcimi prístupmi a v dvoch z týchto troch datasetov nad nimi dominoval.

5 Opis riešenia

Cieľom tejto bakalárskej práce bolo extrahovať vhodnú grafovú reprezentáciu zo zdrojových kódov napísaných v jazyku *Lua*, tieto grafy následne prostredníctvom grafových neurónových sietí klasifikovať do viacerých tried a vhodne vizualizovať úspešnosť modelu. Hlavným cieľom bolo klasifikovať zdrojový kód do 4 tried:

- shebang - spustiteľný zdrojový kód (typicky nejaký program)
- spec - zdrojový kód obsahujúci unit testy využívajúce *Busted* framework
- test - zdrojový kód obsahujúci nejaký test
- obyčajný zdrojový kód - modul, ktorý sa dá načítať v inom module prostredníctvom funkcie *require*

Počas práce sme ale zistili, že štruktúry grafov pre obyčajné zdrojové kódy a test sú veľmi podobné a model si ich mýlil. Rozhodli sme sa teda vyskúšať klasifikáciu aj len pre 3 triedy, ktorej výsledky boli oveľa uspokojivejšie. Posledným cieľom bolo vytvoriť vhodné rozhranie, ktoré využíva už natrénovaný model a jednoducho klasifikuje graf na vstupe.

5.1 Extrakcia grafov

V tomto kroku sme nadviazali na predošlú prácu [14] a použili sme modul *Module Extractor*. Tento modul berie ako argument cestu k repozitáru a následne vytvorí veľký graf nad celým týmto repozitárom. Keďže repozitáre obsahujú viacero zdrojových kódov a iných súborov, graf sme museli ďalej spracúvať.

V prvom kroku sme odfiltrovali všetky (pre túto prácu) nepotrebné uzly, ako napríklad komentáre a metriky. Po tomto kroku nám teda ostal graf nad celým priečinkom, ale len s dôležitými uzlami a ich hranami.

Ďalej sme v grafe hľadali uzly reprezentujúce (.lua) súbory so zdrojovými kódmi a všetky ostatné uzly, ktoré sa týkajú týchto súborov. Týmto krokom sme teda vlastne rozdelili veľký graf na menšie grafy, reprezentujúce konkrétne súbory so zdrojovými kódmi.

Extrahovaný graf však nezachytával vnorenia funkcií, používanie anonymných funkcií ako argumenty a používanie rovnakých volaní, preto sme museli ešte dorobiť pár takýchto hrán na základe textového porovnávania tiel funkcií.

Následne sme každému grafu pridali atribúty ako názov súboru, z ktorého je graf vytvorený, cestu k súboru a označenie či je to shebang, spec alebo test súbor. Na označenie, či je súbor spec alebo test sme sa spoliehali na konvenciu, že dané súbory sa nachádzali v "/spec" resp. "/test" adresároch. Grafy teda dostali toto označenie ak sa v ceste k týmto súborom nachádzal podreťazec "/spec" resp. "/test". Označenie shebang dostali tie súbory, v ktorých sa po otvorení nachádzal na začiatku reťazec "#!".

Grafy teda mohli byť označené viacerými označeniami. Ak sa napríklad súbor nachádzal v "/spec" adresári a na začiatku obsahoval reťazec "#!", bol označený ako spec a aj ako shebang.

Pozn.: Pri vytváraní datasetu pre model klasikátora sme však nechali vždy iba jedno označenie a to s nasledovnou prioritou: shebang, spec, test.

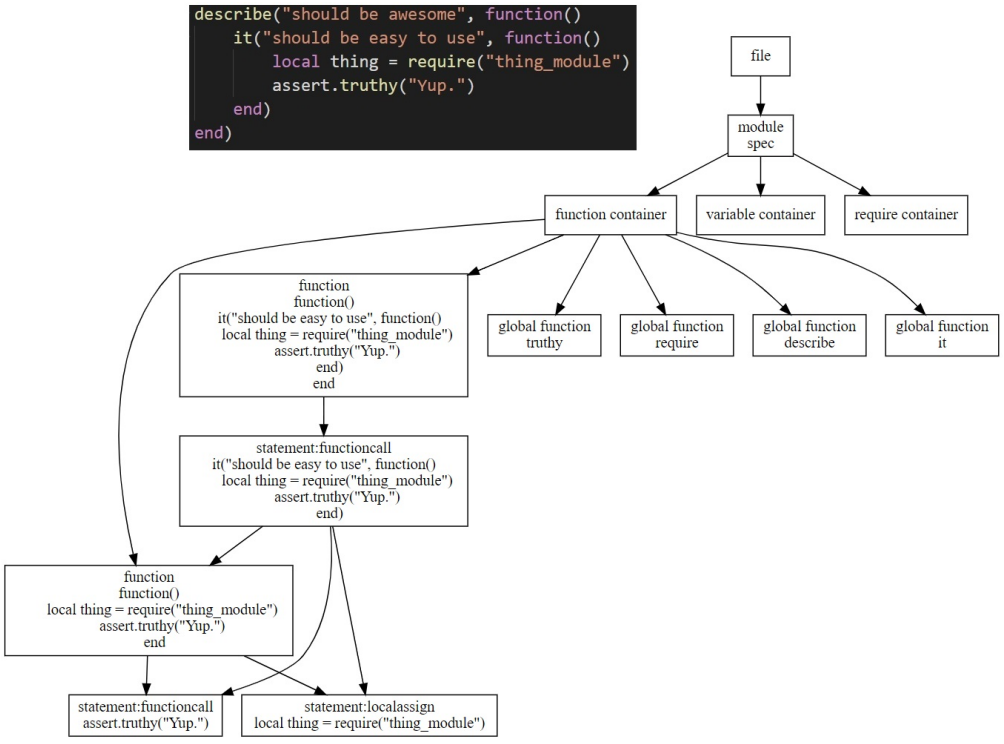
Nakoniec sme graf s uzlami, hranami a ostatnými atribútmi uložili do súboru vo formáte JSON. Tento súbor tak obsahoval:

- _filename [string] - názov súboru, z ktorého je graf vytvorený
- _path [string] - cesta k súboru, z ktorého je graf vytvorený
- _isTest [boolean] - označenie súboru pre test
- _isSpec [boolean] - označenie súboru pre spec
- _isShebang [boolean] - označenie súboru pre shebang
- nodes [array] - uzly grafu
- edges [array] - hrany grafu

Uzol grafu obsahoval atribúty: id daného uzla, jeho typ a textovú informáciu (napr. telo funkcie, ak bol daný uzol funkcia). Textové informácie uzlov však boli použité iba pri vizualizácii grafu.

Hrana grafu obsahovala atribúty: id zdrojového uzla, id cieľového uzla a označenie hrany. Označenia hrán však v modeli neurónovej siete neboli použité, pretože už samotné spojenie dvoch uzlov s ich konkrétnymi typmi nahrádzalo toto označenie.

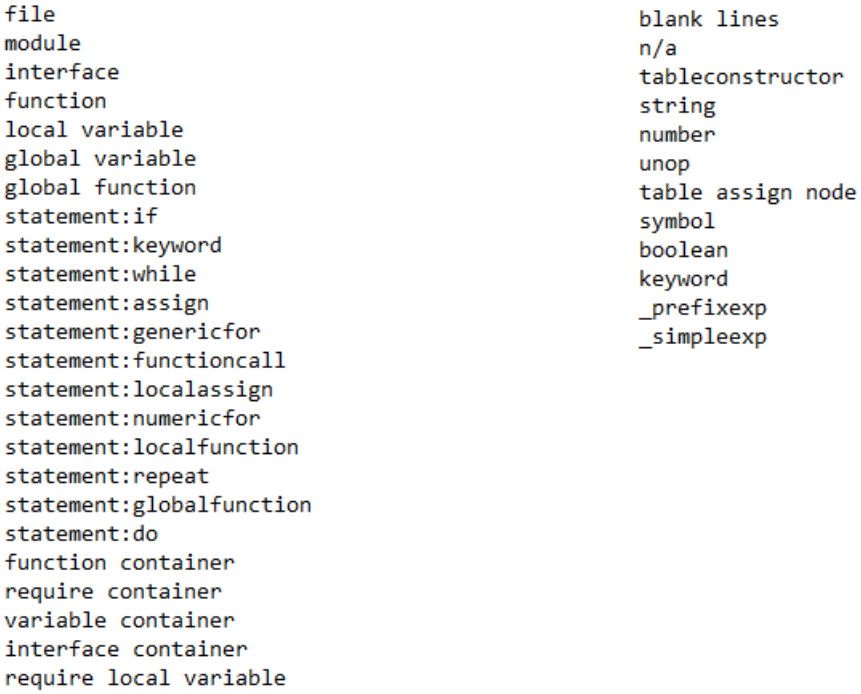
Na obrázku 5.1 je zobrazený graf vytvorený z daného zdrojového súboru. Graf bohužiaľ nezachytáva všetky potrebné vlastnosti daného kódu, ako napríklad použite anonymnej funkcie obsahujúcej volanie funkcie *it* vo funkcii *describe*.



Obr. 5.1: *Reprezentácie zdrojového kódu grafom.*

5.2 Spracovanie datasetu

Vytvorené grafy uložené v JSON súboroch sme načítali v prostredí *Jupyter Notebook*. Keďže extraktor, ktorý sme použili na extrakciu grafov zo zdrojových súborov ma zopár "bugov", v typoch uzlov sa nachádzajú aj také uzly, ktoré by tam byť nemali. Typy uzlov sú zobrazené na obrázku 5.2.



Obr. 5.2: *Zobrazenie typov uzlov. Napravo na obrázku sú uzly, ktoré patria do AST a extraktor ich chybne zaradil do grafu.*

Všetky validné typy uzlov sme zakódovali pomocou *OneHotEncoding*. Ostatné uzly (uzly z pravej strany na obrázku 5.2) sa ignorovali: boli zakódované samými nulami.

Následne sme z grafov vytvorili pole inštancií triedy *StellarGraph*. Pri vytváraní týchto inštancií sme v konštruktore ako reprezentáciu uzlov použili ich (OneHot) kódovanie.

Pre označenie grafov sme vytvorili pole rovnakej dĺžky (označenie na indexe i patrí grafu na rovnakom indexe) a ak existoval graf, ktorý je označený

viacerými označeniami, zvolili sme mu iba jedno podľa priority: shebang, spec, test. Ak graf neobsahoval ani jedno z týchto označení, pokladal sa za obyčajný zdrojový kód. Grafy boli označené nasledovne:

- 0 - obyčajný zdrojový kód
- 1 - shebang zdrojový kód
- 2 - spec zdrojový kód
- 3 - test zdrojový kód

Pri klasifikácii do týchto 4 tried sme však zistili, že grafy označené ako *test* majú veľmi podobnú štruktúru ako grafy označené ako obyčajný zdrojový kód a model si ich veľmi mýlil, čo je ďalej vidno na matici zámen na obrázku 5.8. Grafy označené ako test kvôli nejednoznačnému označeniu datasetu mohli byť v skutočnosti kľudne aj busted testy alebo obyčajné zdrojové kódy. Ďalej tak teda opisujeme klasifikáciu pre 3 triedy, pretože tieto výsledky boli presnejšie a uspokojivejšie.

Z datasetu sme odstránili grafy s odľahlými počtami uzlov, teda tie, ktoré mali príliš odlišný počet uzlov od ostatných grafov. Odstránili sme konkrétne horný a dolný 5% kvantil.

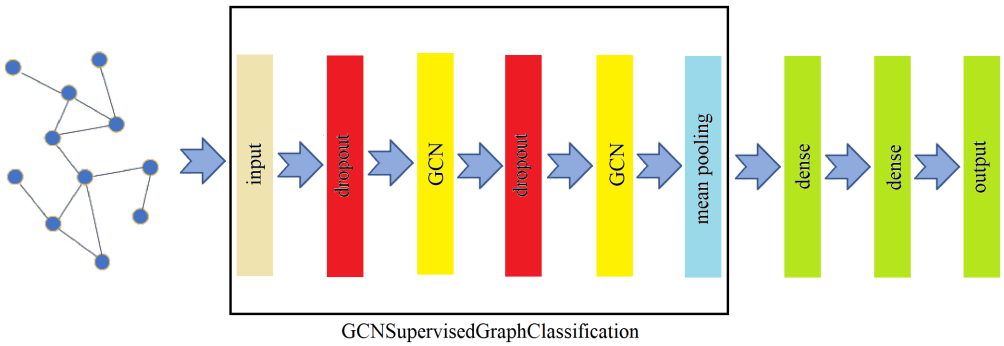
V datasete sa po tomto nachádzalo 2026 obyčajných, 122 shebang a 91 spec zdrojových kódov, dataset bol teda veľmi nevyvážený. Dataset sme trochu vyvážili použitím metódy *Random Undersampling*, kde sme z obyčajných zdrojových kódov vybrali len 500 grafov a počty shebang a spec grafov sme zachovali. Dataset sme rozdelili na trénovací, validačný a testovací v pomere 70:15:15. Z trénovacieho datasetu sme ešte vypočítali váhy pre jednotlivé triedy, ktoré sa neskôr použili v tréningu, pretože bol dataset stále nevyvážený. Tieto váhy slúžili na zvýšenie citlivosti modelu na minoritné triedy pri jeho tréningu.

Finálny dataset teda obsahoval 500 obyčajných, 122 shebang a 91 spec zdrojových kódov rovnomerne rozdelených do trénovacieho, validačného a testovacieho datasetu v pomere 70:15:15.

5.3 Model klasifikátora

Keďže sme dataset označili, zvolili sme model založený na učení sa s učiteľom. Architektúra tohoto modelu je založená na architektúre navrhnutej v článku [24] a využíva grafové konvolučné vrstvy z článku [18].

Vstupom do siete je graf reprezentovaný jeho maticou susednosti a zakódovanými typmi uzlov prostredníctvom metódy *OneHotEncoding*. Keďže sme grafy reprezentovali iba typom a rozložením ich uzlov, museli sme z nich teda najprv extrahovať ich vlastnosti. Táto extrakcia prebieha v grafových konvolučných vrstvách, ktoré produkujú užitočné reprezentácie týchto vlastností prostredníctvom konvolučných filtrov. Obe tieto vrstvy obsahujú 64 neurónov. Vrstvy *dropout* sa využívajú len pri trénovaní modelu a pomáhajú predchádzať jeho pretrénovaniu. Po konvolučných grafových vrstvách nasleduje vrstva *mean pooling*, ktorá slúži na zníženie dimenzionality dát. Nasledujú 2 plne prepojené vrstvy, ktoré vykonávajú samotný proces klasifikácie. Prvá plne prepojená vrstva obsahuje 32 neurónov a druhá 16 neurónov. Posledná (výstupná) vrstva obsahuje 3 neuróny (klasifikujeme do 3 tried) a *softmax* aktivácie. Výstupom siete je teda trojrozmerný pravdepodobnostný vektor, obsahujúci predikciu pre typ zdrojového kódu v nasledovnom poradí: obyčajný zdrojový kód, shebang, spec. Všetky vrstvy (okrem poslednej výstupnej vrstvy) obsahujú *ReLU* aktivácie. Architektúra tejto siete je zobrazená na obrázku 5.3.



Obr. 5.3: *Architektúra modelu neurónovej siete, slúžiacej na klasifikáciu grafov. Na vstupe je graf reprezentovaný jeho maticou susednosti a zakódovanými typmi uzlov. Grafové konvolučné vrstvy (označené žltou) slúžia na extrakciu vlastností tohoto grafu. Dropout vrstvy (označené červenou) pomáhajú predchádzať pretrénovaniu modelu a vrstva mean pooling (označená modrou) redukuje dimenzionalitu dát. Dense vrstvy (označené zelenou) slúžia na samotný proces klasifikácie. Výstupná vrstva je pravdepodobnostný vektor pre všetky triedy.*

5.4 Implementácia

Na samotné generovanie grafov nad celými repozitármi bol použitý existujúci modul *Module Extractor* [14] implementovaný v jazyku *Lua*, ktorý bol vytvorený v predošlých prácach. Ďalšie spracovanie týchto grafov a následné zapísanie grafov pre jednotlivé súbory so zdrojovými kódmi do JSON súborov je implementované v jazyku *Lua* vo verzii 5.1.

Následná manipulácia s vytvorenými JSON súbormi bola vykonávaná v prostredí *Jupyter Notebook* v jazyku *Python* vo verzii 3.8.0.

Pri práci s poliami sme využívali knižnicu *numpy*. Na zakódovanie typov uzlov sme použili metódu *OneHotEncoding* z knižnice *scikit-learn*. Úprava vyváženia datasetu bola implementovaná pomocou triedy *RandomUnderSampler* z knižnice *imbalanced-learn*, ktorou sme zmiernili nevyváženosť datasetu. Na rozdelenie datasetu na trénovací, validačný a testovací sme použili metódu *train_test_split* a na výpočet váh (keďže bol dataset stále nevyrovnaný) metódu *compute_class_weight*. Obe tieto metódy patria do knižnice *scikit-learn*. Grafy boli držané ako inštancie triedy *StellarGraph* z knižnice *stellargraph* vo verzii 1.2.1. Z tejto knižnice sme použili aj model *GCNSupervisedGraphC-*

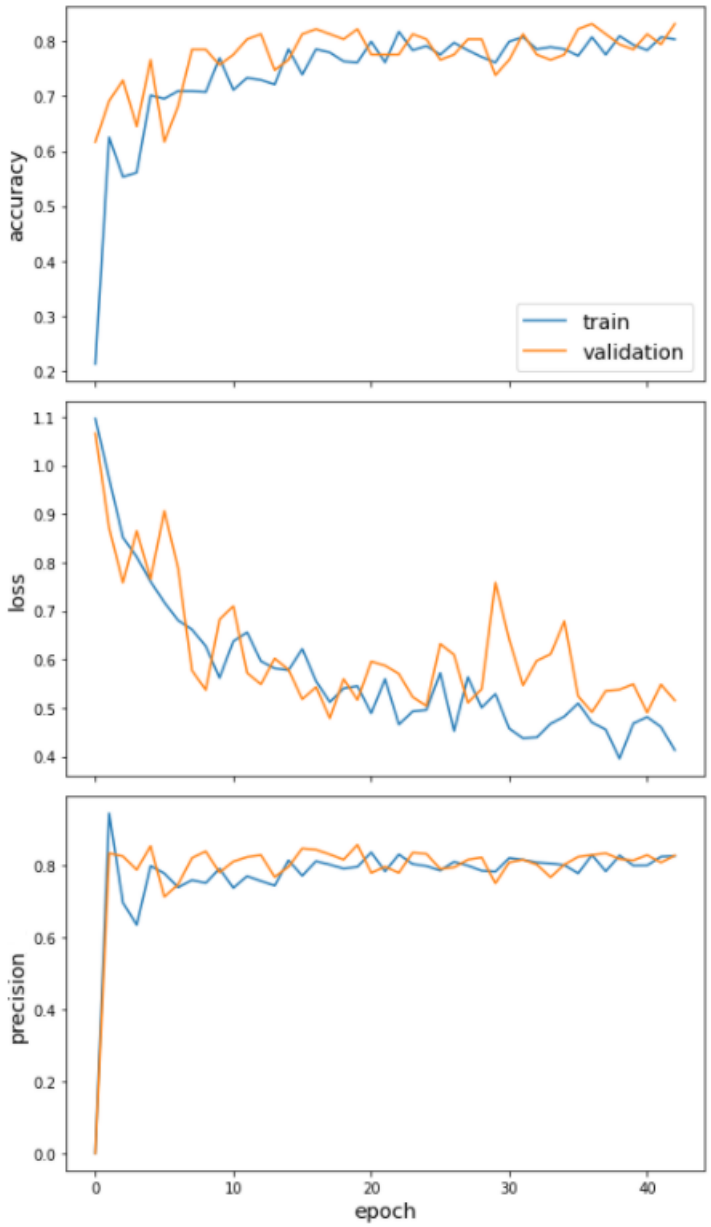
lassification, ktorý obsahuje grafové konvolučné vrstvy a slúži na extrakciu vlastností z grafov. Vytvorenie, trénovanie a evaluácia modelu boli implementované pomocou knižnice *tensorflow* vo verzii 2.4.1 prostredníctvom keras API. Na vizualizácie boli použité knižnice *matplotlib*, *seaborn* a *graphviz*.

Na ladenie hyperparametrov modelu sme použili knižnicu *keras-tuner*, avšak tejto časti sme sa už nestihli plne venovať.

5.5 Dosiahnuté výsledky

Model pri klasifikácii do 3 tried dosahuje pomerne vysokú presnosť, viď obrázok 5.5, čo však pri nevyrovnanom datasete môže byť veľmi zavádzajúca informácia. Použili sme preto aj metriku *precíznosť* a následne sme ešte vizualizovali predikcie prostredníctvom matice zámen.

Na obrázku 5.4 sú zobrazené metriky siete počas tréningu. Môžeme vidieť, že hodnota *loss* pomerne skákala. Je to zapríčinené zvýšenou citlivosťou na minoritné triedy nachádzajúce sa v datasete zabezpečenú prostredníctvom váh tried, ktoré sme vypočítali na základe počtu jednotlivých označení v datasete. Na tréning bol pôvodne nastavený počet epôch na hodnotu 100, avšak funkcia *EarlyStop* monitorujúca klesanie hodnoty *loss* tréning zastavila už pri 43. epoche, aby sa zabránilo pretrénovaniu siete.



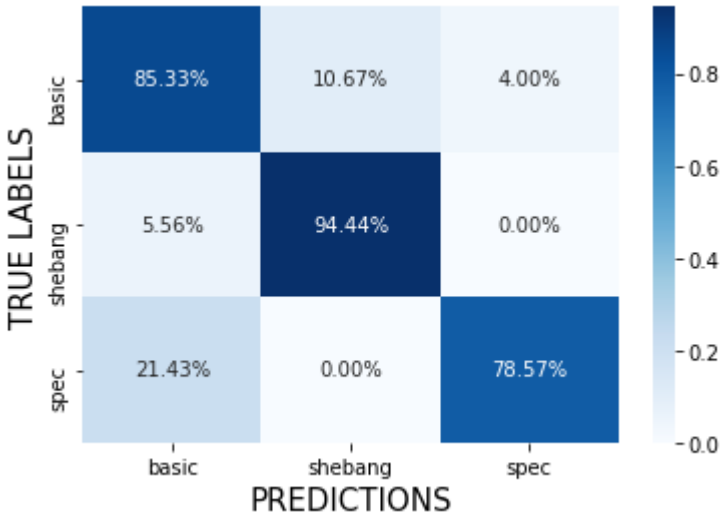
Obr. 5.4: *Hodnoty metrík počas trénovania modelu*

Evaluáciu modelu sme vykonávali na testovacom datasete, ktorý nebol použitý pri trénovaní. Výsledky tejto evaluácie sú zobrazené na obrázku 5.5.

```
107/107 [==============================] - 0s 2ms/step - loss: 0.3978
                                                       - accuracy: 0.8598
                                                       - precision: 0.8835
Test Set Metrics:
        loss: 0.3978
        accuracy: 0.8598
        precision: 0.8835
```

Obr. 5.5: *Evaluácia modelu na testovacom datasete*

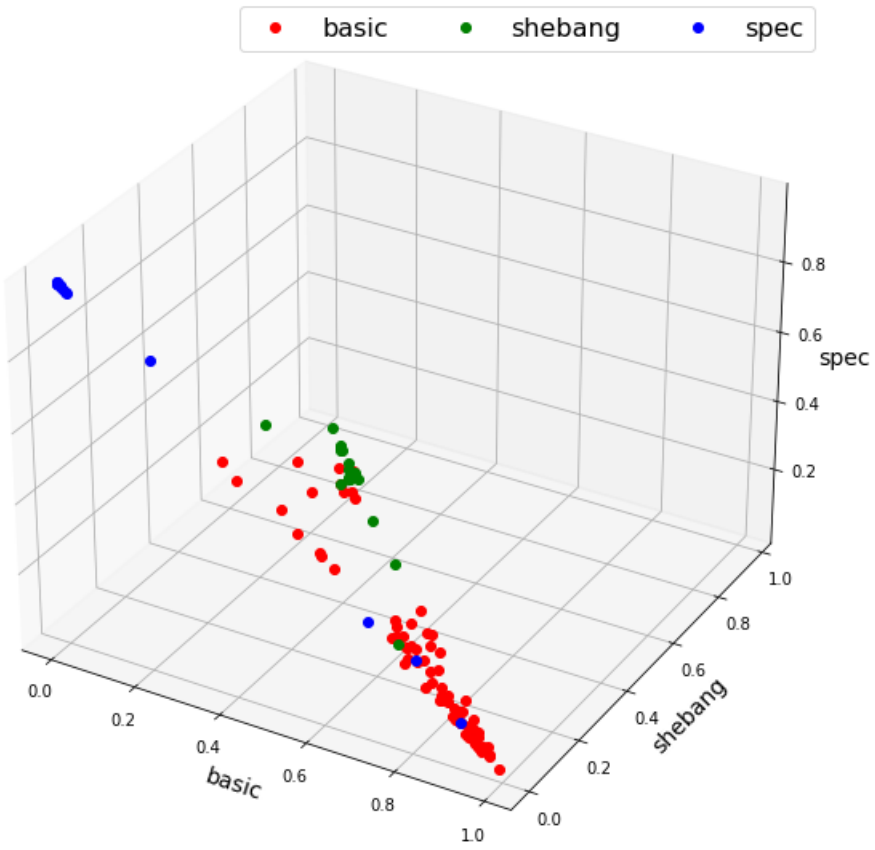
Normalizovaná matica zámen na obrázku 5.6 zobrazuje skutočnú úspešnosť natrénovaného modelu pre klasifikáciu do 3 tried: obyčajný zdrojový kód, shebang, spec.



Obr. 5.6: *Zobrazenie normalizovanej matice zámen pri klasifikácii do 3 tried. Riadky predstavujú skutočné označenia a v stĺpcoch sa nachádzajú predikcie modelu. Jednotlivé polia obsahujú počet percent predikcií pre grafy s daným označením na základe riadku.*

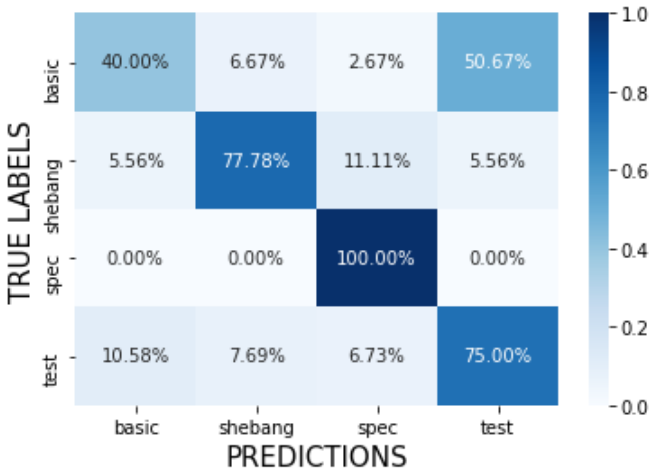
Na obrázku 5.7 je zobrazený 3-rozmerný bodový diagram, vytvorený z predikcií modelu na testovacom datasete. Jednotlivé body predstavujú grafy, pre ktoré model predikoval ich triedy. Zafarbenie bodov predstavuje ich skutočné označenie: červenou sú grafy vytvorené z obyčajných zdrojových

kódov, zelenou shebang a modrou spec grafy. Body sú rozmiestnené podľa ich pravdepodobnostného vektora z modelu.



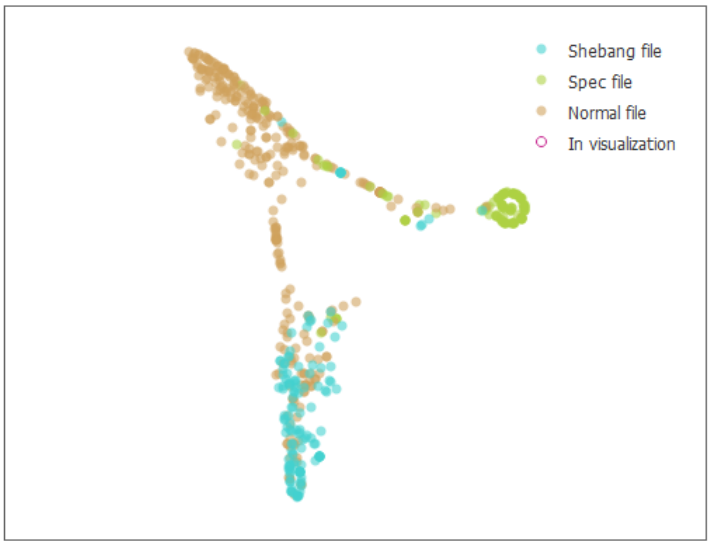
Obr. 5.7: *Zobrazenie 3-rozmerného bodové diagramu. Jednotlivé body predstavujú grafy a sú ofarbené na základe ich skutočného označenia. V priestore sú body rozmiestnené na základe ich pravdepodobnostného vektora.*

Na obrázku 5.8 je zobrazená normalizovaná matica zámen predikcií modelu pre klasifikáciu do 4 tried: obyčajný zdrojový kód, shebang, spec, test. Na matici je vidno, že model nevedel dobre rozoznať obyčajné zdrojové kódy a zdrojové kódy označené ako test. Kvôli tomuto faktu sme sa nakoniec rozhodli klasifikovať kódy len do 3 kategórií, a zdrojové kódy označené ako test sme brali ako obyčajné zdrojové kódy.



Obr. 5.8: *Zobrazenie normalizovanej matice zámen pri klasifikácii do 4 tried. Riadky predstavujú skutočné označenia a v stĺpcoch sa nachádzajú predikcie modelu. Jednotlivé polia obsahujú počet percent predikcií pre grafy s daným označením na základe riadku.*

Na obrázku 5.9 je zobrazený diagram zhlukov vytvorený pomocou algoritmu t-sne po vytvorení predikcií modelu pre celý dataset z práce kolegu Andreja Cikraia [8].



Obr. 5.9: *Diagram zobrazujúci rozdelenie tried do 3 farebne odlíšených kategórií. Jednotlivé farby predstavujú skutočné označenia grafov.*

Počas práce sme tiež vytvorili triedu *GraphPredictor*, ktorá slúži na manipuláciu s už natrénovaným modelom neurónovej siete. Pri konštrukcii inštancie tejto triedy sa ako parameter použije cesta k natrénovanému modelu v lokálnom úložisku alebo samotná inštancia tohoto modelu. Pri predikcii sa jednoducho použije funkcia tejto triedy *predict*, ktorá berie ako parameter list ciest k extrahovaným grafom v lokálnom úložisku, viď obrázok 5.10 alebo list inštancií už načítaných grafov a vypíše jednotlivé predikcie pre každý graf.

```
graph_paths = ['../data/30log/30log.lua.json',
               '../data/bencode/dumptorrent.lua.json',
               '../data/busted/cl_error_messages.lua.json']

graph_predictor.predict(graph_paths)
```

```
30log.lua:
        basic 89.4%
        shebang 5.5%
        spec 5.1%
dumptorrent.lua:
        basic 22.34%
        shebang 74.91%
        spec 2.76%
cl_error_messages.lua:
        basic 1.06%
        shebang 1.69%
        spec 97.24%
```

Obr. 5.10: *Zobrazenie jednoduchého použitia inštancie triedy GraphPredictor na predikciu extrahovaných grafov.*

Na obrázku 5.11 je zobrazené použitie inštancie triedy *GraphPredictor* na evaluáciu modelu na zvolených grafoch na vstupe (podobne ako na obrázku 5.10.

```
test_metrics = graph_predictor.evaluate(graph_paths)

print("\nTest Set Metrics:")
for name, val in zip(graph_predictor.model.metrics_names, test_metrics):
    print("\t{}: {:0.4f}".format(name, val))
```

```
3/3 [==============================] - 0s 3ms/step - loss: 0.1430
                                                    - accuracy: 1.0000
                                                    - precision: 1.0000
Test Set Metrics:
        loss: 0.1430
        accuracy: 1.0000
        precision: 1.0000
```

Obr. 5.11: *Zobrazenie jednoduchého použitia inštancie triedy GraphPredictor na evaluáciu modelu na zvolených extrahovaných grafoch.*

6 Zhodnotenie

Na začiatku analytickej časti tejto práce bol opísaný základný koncept učenia sa reprezentácií a fungovania neurónových sietí. Následne sa analýza venuje hlbšie problematike grafových neurónových sietí a ich využitiu. Posledná časť analýzy sa venuje konkrétnej oblasti spracovania zdrojových kódov a opisom už existujúcich riešení v tejto oblasti.

V praktickej časti bola vytvorená reprezentácia zdrojového kódu prostredníctvom grafu. Následne sa tieto grafy ďalej spracúvali až pokiaľ sme nezískali finálny dataset, ktorý sme použili na natrénovanie modelu. Časť z tohoto finálneho datasetu sme pri trénovaní nepoužili, aby sme ju mohli použiť pri testovaní natrénovaného modelu.

Aj napriek nie príliš vhodnej reprezentácii dát (zdrojových kódov), ktorá nezachytáva všetky vlastnosti kódu a nedostatku času na ladenie hyperparametrov, model dosahoval pomerne uspokojivé výsledky.

6.1 Možné vylepšenia

Ako hlavný nedostatok považujeme nedostatočnú reprezentáciu dát, pretože graf extrahovaný zo zdrojových kódov nezachytáva rôzne vnorenia funkcií a používanie anonymných funkcií v parametroch.

Za ďalší z nedostatkov môžeme považovať nie príliš najšťastnejšie označenie grafov v datasete. Spoliehali sme sa na konvenciu, že spec súbory sa nachádzajú len v "spec" adresároch, avšak toto nemusí vždy platiť. Okrem toho bol dataset veľmi nevyvážený a bolo by vhodné mať k dispozícii viac vzoriek z minoritných zdrojových kódov spec a shebang.

V závere práce nám bohužial nezostal dostatok času na ledenie hyperparametrov modelu, takže aj tu sa ponúka možné vylepšenie.

Literatúra

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[3] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[4] Yoshua Bengio, Ian Goodfellow, and Aaron Courville. *Deep learning*, volume 1. MIT press Massachusetts, USA:, 2017.

[5] Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.

[6] Zimin Chen and Martin Monperrus. A literature study of embeddings on source code. *arXiv preprint arXiv:1904.03061*, 2019.

[7] KR Chowdhary. Natural language processing. In *Fundamentals of Artificial Intelligence*, pages 603–649. Springer, 2020.

[8] Andrej Cikrai. Vizualizácia a vysvetliteľnost hlbokého účenia, 2021.

[9] Li Deng and Dong Yu. Deep learning: methods and applications. *Foundations and trends in signal processing*, 7, 2014.

[10] Rohan N Dhamdhere. Meta learning for graph neural networks. 2018.

[11] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. Graph neural networks for social recommendation. In *The World Wide Web Conference*, pages 417–426, 2019.

[12] Shlok Gilda. Source code classification using neural networks. In *2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pages 1–6. IEEE, 2017.

[13] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 200–20010. IEEE, 2018.

[14] Denis Illes. Vizualizácia softvérových systémov v 3D priestore. Master's thesis, 2018.

[15] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, 2016.

[16] Shaoxiong Ji, Shirui Pan, Erik Cambria, Pekka Marttinen, and Philip S Yu. A survey on knowledge graphs: Representation, acquisition and applications. *arXiv preprint arXiv:2002.00388*, 2020.

[17] Justin Johnson, Agrim Gupta, and Li Fei-Fei. Image generation from scene graphs. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1219–1228, 2018.

[18] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[19] Lawtomated. Machine vs deep learning, 2019.

[20] Shuai Li, Wanqing Li, Chris Cook, Ce Zhu, and Yanbo Gao. Independently recurrent neural network (indrnn): Building a longer and deeper rnn. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5457–5466, 2018.

[21] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[22] Z. Liu and J. Zhou. *Introduction to Graph Neural Networks*. 2020.

[23] T. Mitchell et al. Machine learning. 1997. *Burr Ridge, IL: McGraw Hill*, 45(37), 1997.

[24] Federico Monti, Fabrizio Frasca, Davide Eynard, Damon Mannion, and Michael M Bronstein. Fake news detection on social media using geometric deep learning. *arXiv preprint arXiv:1902.06673*, 2019.

[25] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[26] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. Learning convolutional neural networks for graphs. In *International conference on machine learning*, pages 2014–2023, 2016.

[27] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

[28] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.

[29] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[30] Amarnag Subramanya and Partha Pratim Talukdar. Graph-based semi-supervised learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 8(4):1–125, 2014.

[31] Heung-Il Suk. An introduction to neural networks and deep learning. In *Deep Learning for Medical Image Analysis*, pages 3–24. Elsevier, 2017.

[32] Xiaobing Sun, Xiangyue Liu, Jiajun Hu, and Junwu Zhu. Empirical studies on the nlp techniques for source code data preprocessing. In *Proceedings of the 2014 3rd International Workshop on Evidential Assessment of Software Technologies*, pages 32–39, 2014.

[33] Sahil Suneja, Yunhui Zheng, Yufan Zhuang, Jim Laredo, and Alessandro Morari. Learning to map source code to software vulnerability using code-as-a-graph. *arXiv preprint arXiv:2006.08614*, 2020.

[34] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[35] E. Tjoa and C. Guan. A survey on explainable artificial intelligence (xai): Toward medical xai. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–21, 2020.

[36] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 297–308. IEEE, 2016.

[37] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98. IEEE, 2016.

[38] Pengfei Xu, Xiaojun Chang, Ling Guo, Po-Yao Huang, Xiaojiang Chen, and Alexander G Hauptmann. A survey of scene graph: Generation and application. Technical report, EasyChair, 2020.

[39] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE, 2019.

[40] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A

review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.

[41] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. Aligraph: A comprehensive graph neural network platform. *arXiv preprint arXiv:1902.08730*, 2019.

[42] Xiaojin Zhu and Andrew B Goldberg. Introduction to semi-supervised learning. *Synthesis lectures on artificial intelligence and machine learning*, 3(1):1–130, 2009.

A Technická dokumentácia

A.1 Spracovanie dát

Ukážka extrahovaného grafu

Z každého zdrojového kódu je extrahovaný graf, ktorý je následne uložený do súboru vo formáte JSON.


```
{
  "_filename": "cl_error_messages.lua",
  "_isShebang": false,
  "_isSpec": true,
  "_isTest": false,
  "_path": "modules/busted/spec/cl_error_messages.lua",
  "edges": [ {
    "from": 0,
    "label": "implements",
    "to": 1
  },
  ...
  {
    "from": 25,
    "label": "hasArgument",
    "to": 16
  } ],
  "nodes": [ {
    "id": 0,
    "text": "",
    "type": "file"
  },
  ...
  {
    "id": 34,
    "text": "",
    "type": "require container"
  } ]
}
```

Načítanie grafov

Jednotlivé grafy uložené v JSON súboroch sú načítané v s prostredí JupyterNotebook do premenej *graphs_raw*. Funkcia *tnrange* bola použitá na zobrazenie "progressbaru" na vizualizáciu stavu načítavania. Kedže použitá knižnica, ktorá lua tabuľky parsovala do formátu JSON má pravdepodobne zopár bugov, niektoré JSON súbory nie sú valídne. Preto je nutné otvárať ich v *try-except* blokoch.

```
graphs_raw = []

data_path = '../data'
dirs = os.listdir(data_path)
num_dirs = len(dirs)

for i in tnrange(num_dirs, desc='Dataset loading'):
    dirname = dirs[i]
    dir_path = data_path + '/' + dirname
    for filename in os.listdir(dir_path):
        json_file = dir_path + '/' + filename
        with open(json_file) as jf:
            graph = {}
            try:
                json_data = json.load(jf)
                graph['filename'] = json_data['_filename']
                graph['path'] = json_data['_path']
                graph['is_test'] = int(json_data['_isTest'])
                graph['is_spec'] = int(json_data['_isSpec'])
                graph['is_shebang'] = int(json_data['_isShebang'])
                graph['nodes'] = pd.DataFrame(json_data['nodes'], columns=['id', 'type'])
                graph['nodes'] = graph['nodes'].set_index('id')
                graph['edges'] = pd.DataFrame(json_data['edges'], columns=['from', 'to'])
                graphs_raw.append(graph)
            except:
                print(f'Wrong json file: {json_file}')
```

Pripravenie datasetu pre model

Vytvorenie poľa inštancií triedy StellarGraph a poľa rovnakej veľkosti pre označenia daných grafov. Zakomentovaná časť kódu sa používala pri klasifikácii do 4 tried.

```
def createStellarGraph(graphRaw):
    nodes = graphRaw['enc_node_types']
    edges = graphRaw['edges']
    graph = StellarGraph(
            nodes=nodes,
            edges=edges,
            source_column='from',
            target_column='to',
            is_directed=True
        )
    return graph

graphs = np.array([createStellarGraph(gr) for gr in graphs_raw])
#graph_labels = pd.Series(
#        [
#            1 if gr['is_shebang']
#            else 2 if gr['is_spec']
#            else 3 if gr['is_test']
#            else 0 for gr in graphs_raw
#        ],
#        dtype="category",
#        name='label'
#    )
graph_labels = np.array([
        1 if gr['is_shebang']
        else 2 if gr['is_spec']
        else 0 for gr in graphs_raw
    ])
```

A.2 Model grafovej neurónovej siete

Vytvorenie modelu

Model neurónovej siete je opísaný v sekcii 5.3. Funkciu pre vytvorenie modelu je možné použiť aj počas ladenia hyperparametrov pomocou knižnice *keras-tuner*. Ak sa model vytvára mimo procesu ladenia hyperparametrov, použijú sa ich prednastavené hodnoty.

```
def model_builder(hp=None):
    # default hyperparameters
    convo1 = 64
    convo2 = 64
    dropout = 0.5
    dense1 = 32
    dense2 = 16
    learning_rate = 0.005

    # hyperparameter tuning
    if (hp):
        convo1 = hp.Int('convo1', min_value=32, max_value=320, step=32)
        convo2 = hp.Int('convo2', min_value=32, max_value=320, step=32)
        dropout = hp.Float('dropout', min_value=0.1, max_value=0.5, step=0.1)
        dense1 = hp.Int('dense1', min_value=32, max_value=320, step=32)
        dense2 = hp.Int('dense2', min_value=16, max_value=160, step=16)
        learning_rate = hp.Choice('learning_rate', values=[0.0001, 0.001, 0.005, 0.01])

    gc_model = GCNSupervisedGraphClassification(
            layer_sizes=[convo1, convo2],
            activations=["relu", "relu"],
            generator=generator,
            dropout=dropout,
        )

    x_inp, x_out = gc_model.in_out_tensors()
    predictions = Dense(units=dense1, activation="relu")(x_out)
    predictions = Dense(units=dense2, activation="relu")(predictions)
    predictions = Dense(num_classes, activation="softmax")(predictions)

    model = Model(inputs=x_inp, outputs=predictions)
    model.compile(
        optimizer=Adam(learning_rate=learning_rate),
        loss=categorical_crossentropy,
        metrics=['accuracy', Precision()]
    )

    return model
```

Trieda na jednoduchú manipuláciu s natrénovaným modelom

Použitie inštancie triedy je zobrazené na obrázkoch 5.10 a 5.11. Konštruktor berie na vstupe iba jeden parameter a to cestu k natrénovanému modelu neurónovej siete v lokálnom úložisku alebo jeho inštanciu.

```
class GraphPredictor:

    NODE_TYPES = ['require container', 'function', 'local variable', 'file',
                  'tableconstructor', 'module', 'statement:keyword',
                  'variable container', 'statement:numericfor', 'statement:genericfor',
                  'statement:if', 'global function', 'statement:functioncall',
                  'statement:assign', 'function container', 'statement:localassign',
                  'global variable', 'statement:globalfunction', 'statement:while',
                  'statement:localfunction', 'interface container', 'interface',
                  'require local variable', 'statement:do', 'statement:repeat']


    def __init__(self, model):
      # if model is path
      if (isinstance(model, str)):
        self.model = keras.models.load_model(model)
      else:
        self.model = model


    def create_StellarGraph(self, graph):
      nodes = graph['encNodeTypes']
      edges = graph['edges']
      stellGraph = StellarGraph(
          nodes=nodes,
          edges=edges,
          source_column='from',
          target_column='to',
          is_directed=True
        )

      return stellGraph


    def create_label(self, graph):
      label = [1, 0, 0]
      if graph['isShebang']:
        label = [0, 1, 0]
      elif graph['isSpec']:
        label = [0, 0, 1]

      return label
```

```
def encode_node_types(self, graph):
  enc = OneHotEncoder(handle_unknown='ignore')
  X = np.array(self.NODE_TYPES).reshape(-1,1)
  enc.fit(X)
  encoded = enc.transform(graph['nodes']['type'].values.reshape(-1,1))
  graph['encNodeTypes'] = pd.DataFrame(encoded.toarray())


def read_graph(self, graph_path):
  graph = {}

  with open(graph_path) as json_file:
      json_data = json.load(json_file)
      graph['filename'] = json_data['_filename']
      graph['path'] = json_data['_path']
      graph['is_test'] = int(json_data['_isTest'])
      graph['is_spec'] = int(json_data['_isSpec'])
      graph['is_shebang'] = int(json_data['_isShebang'])
      graph['nodes'] = pd.DataFrame(json_data['nodes'], columns=['id', 'type', 'text'])
      graph['nodes'] = graph['nodes'].set_index('id')
      graph['edges'] = pd.DataFrame(json_data['edges'], columns=['from', 'to'])

  self.encode_node_types(graph)

  return graph


def show_graph(self, graph):
  graphRaw = graph

  # if graph is path
  if (isinstance(graph, str)):
    graphRaw = self.read_graph(graph_path)

  nodes_data = graphRaw['nodes']
  edges_data = graphRaw['edges']

  dot = Digraph(format='png')

  for idx, row in nodes_data.iterrows():
    dot.node(str(idx), str(row['type'] + '\n' + str(row['text'])), shape='box')

  for idx, row in edges_data.iterrows():
    dot.edge(str(row['from']), str(row['to']))

  return dot


def predict(self, graphs, print_predictions=True):
```

```
    graphsRaw = graphs

    # if graphs are paths to graphs
    if (isinstance(graphs[0], str)):
      graphsRaw = [self.read_graph(path) for path in graphs]

    stellGraphs = [self.create_StellarGraph(g) for g in graphsRaw]
    labels = [self.create_label(g) for g in graphsRaw]

    generator = PaddedGraphGenerator(graphs=stellGraphs)
    X = generator.flow(graphs=stellGraphs)
    predicts = self.model.predict(X)

    if (print_predictions):
      for i, p in enumerate(predicts):
        filename = graphsRaw[i]['filename']
        basic = round(p[0]*100, 2)
        shebang = round(p[1]*100, 2)
        spec = round(p[2]*100, 2)
        print(f'{filename}:\n\tbasic {basic}%\n\tshebang {shebang}%\n\tspec {spec}%')

      return

    return predicts


  def evaluate(self, graphs):
    graphsRaw = graphs

    # if graphs are paths to graphs
    if (isinstance(graphs[0], str)):
      graphsRaw = [self.read_graph(path) for path in graphs]

    stellGraphs = [self.create_StellarGraph(g) for g in graphsRaw]
    labels = [self.create_label(g) for g in graphsRaw]

    generator = PaddedGraphGenerator(graphs=stellGraphs)
    test_gen = generator.flow(graphs=stellGraphs, targets=labels)
    return self.model.evaluate(test_gen, verbose=1)
```

B Používateľská príručka

B.1 Inštalácia

Systémové požiadavky

Na extrakciu grafov zo zdrojových kódov do JSON súborov sú potrebné:

- operačný systém Linux
- Lua - verzia 5.1
- gitlab projekt luadb - vetva develop

Na trénovanie a používanie modelu sú potrebné nasledovné komponenty:

- ipython - verzia 7.22.0
- Python - verzia 3.8.0
- knižnica pandas - verzia 1.2.3
- knižnica tqdm - verzia 4.60.0
- knižnica scikit-learn - verzia 0.24.1
- knižnica numpy - verzia 1.19.5
- knižnica stellargraph - verzia 1.2.1
- knižnica imbalanced-learn - verzia 0.8.0
- knižnica tensorflow - verzia 2.4.1
- knižnica keras-tuner - verzia 1.0.2
- knižnica matplotlib - verzia 3.4.1
- knižnica seaborn - verzia 0.11.1
- knižnica graphviz - verzia 0.16
- modul json

B.2 Spustenie

Po nainštalovaní vyžadovaných knižníc a stiahnutí extrahovaných grafov uložených v JSON súboroch alebo ich vytvorení prostredníctvom extraktoru je možne program spustiť v prostredí *Jupyter Notebook*.

C Digitálna časť práce

Evidenčné číslo práce v informačnom systéme: FIIT-5212-96898

/Application

- implementácia opisovaného riešenia

/Documentation

- bakalárska práca spolu s anotáciami v slovenskom a anglickom jazyku

/Documentation/Latex

- latex zdrojové súbory dokumentácie

/Documentation/BibTeX

- BibTeX súbor s použitými referenciami

/Resources

- vstupné dáta opisované v dokumente

read.me - popis obsahu média v slovenskom jazyku

D Plán práce

D.1 Zimný semester

- 1. týždeň - oboznámenie sa s témou bakalárskej práce
- 2. až 6. týždeň - hľadanie a štúdium materiálov k téme práce
- 7. týždeň - vytvorenie kostry práce
- 8. až 11. týždeň - písanie analýzy a doštudovanie materiálov k téme práce
- 12. týždeň - finalizácia a odovzdanie 1. časti práce

Plán práce som zozačiatku dodržoval, avšak termín odovzdania sa počas semestra predĺžil. Mohol som tak štúdiu materiálov venovať viacej času. Štúdium materiálov mi kvôli komplexnosti literatúry trvalo dlhšie ako som plánoval, avšak počas tohoto štúdia som si už robil veľa poznámok, vďaka ktorým som počas písania práce dobehol stratený čas.

D.2 Letný semester

- 1. až 2. týždeň - implementácia extrakcie grafov zo zdrojových kódov
- 3. týždeň - skúmanie a testovanie existujúcich implementácii modelov neurónových sietí slúžiacich na klasifikáciu grafov
- 4. až 6. týždeň - spracúvanie grafov v datasete
- 7. - vytvorenie modelu grafovej neurónovej siete
- 8. až 10. týždeň - ďalšie spracúvanie grafov a ladenie modelu
- 11 až 12. týždeň - spísanie, finalizácia a odovzdanie riešenia

Harmonogram sa mi už od začiatku nedarilo dodržiavať, keď mi implementácia extrakcie grafov zabrala dvojnásobný čas, aký som predpokladal. V ďalších krokoch som našťastie pomaly dobiehal stratený čas a navyše sa, rovnako ako v zimnom semestri, predĺžil termín odovzdania. Naskytol sa mi tak priestor vyskúšať viacero rôznych modelov sietí, čo mi pri práci veľmi pomohlo.