Zbornik radova Fakulteta tehničkih nauka, Novi Sad

UDK: 004.85
DOI: https://doi.org/10.24867/11BE03Prokic

AUTOMATSKA DETEKCIJA INDIKATORA LOŠE DIZAJNIRANOG KODA BAZIRANA NA ISTORIJI PROMENA KODA

AUTOMATIC DETECTION OF CODE SMELLS BASED ON CODE CHANGE HISTORY

Simona Prokić, *Fakultet tehničkih nauka, Novi Sad*

Oblast – ELEKTROTEHNIKA I RAČUNARSTVO

Kratak sadržaj – *Kod niskog kvaliteta sadrži strukture (code smells) koje otežavaju održavanje i dalji razvoj softvera. U ovom radu predstavljen je model zasnovan na mašinskom učenju za automatsku detekciju indikatora loše dizajniranog koda (code smell-ova) baziranu na istoriji promena koda. Ulaz modela su vrednosti metrika softverskog koda, izračunate u n revizija za posmatrani isečak koda. Izlaz iz modela je labela koja označava da li posmatrani isečak koda sadrži indikator loše dizajniranog koda ili ne. Studija slučaja izvršena je na detekciji klasa sa mnogo odgovornosti (God Class). Predloženi su koraci za poboljšanje i dalji razvoj arhitekture.*

Ključne reči: *mašinsko učenje, transformer, indikatori loše dizajniranog koda*

Abstract – *This paper presents a machine learning model for automatic detection of code smells based on code change history. The model's inputs are the source code metrics' values in n revisions for the observed code snippet. The model's output is a label that indicates whether the observed code snippet contains a code smell or not. We test the model on the case study of detecting classes with many responsibilities (God Classes). Steps for further architecture improvement are discussed.*

Keywords: *machine learning, transformer, code smells*

1. UVOD

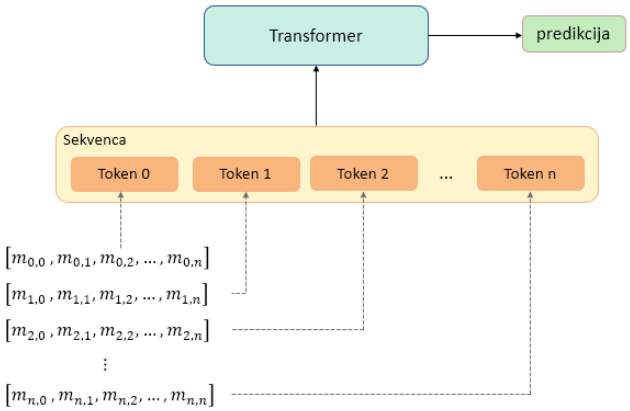
Softver niskog kvaliteta je podložan greškama i teško se menja [1], što rezultuje dodatnim troškovima i otežanom održavanju, a u ekstremnim slučajevima i potpunom nemogućnošću izmene i daljeg razvoja softvera. Izvorni kod niskog kvaliteta sadrži strukture koje se nazivaju *code smells* [2] i nastaju kada programeri pišu kod koji obavlja posao, ali nije čitljiv i lako promenljiv [1]. Ključna aktivnost u razvoju softvera je refaktorisanje koda, što podrazumeva uklanjanje *code smell*-ova bez promene ponašanja softvera, čime se osigurava duži vek trajanja softvera [2].

Većina postojećih alata i tehnika za detekciju *smell*-ova je zasnovana na heuristikama i vrši detekciju izračunavanjem vrednosti metrika koda i primenom predefinisanih pragova [4]. Međutim, takvi alati se retko koriste, jer nisu dovoljno precizni i laki za korišćenje, i ne pružaju informacije kako refaktorisati kod [3].

NAPOMENA:
Ovaj rad proistekao je iz master rada čiji mentor je bila dr Jelena Slivka, vanredni profesor.

Tehnike za detekciju *code smell*-ova zasnovane na mašinskom učenju (*Machine Learning*, ML) imaju bolje performanse od tehnika zasnovanih na heuristikama [4].

U ovom radu opisana je automatska detekcija indikatora loše dizajniranog koda zasnovana na tehnici mašinskog učenja, pri čemu ulaz u ML model predstavlja fragment koda, a izlaz oznaka da li taj fragment sadrži indikatore loše dizajniranog koda ili ne. Detekcija je zasnovana na istoriji promena koda, što se pokazalo kao uspešan pristup [5][6][7]. Model koji je korišćen za implementaciju automatske detekcije je Transformer [11]. Ulaz u Transformer je sekvenca koja se sastoji od više tokena, gde jedan token sadrži niz vrednosti strukturnih metrika koda u jednoj reviziji (odnosno *commit*-u). Ovo znači da sekvenca sadrži nizove vrednosti metrika u više *commit*-ova i omogućava posmatranje istorije promene koda kroz metrike (Slika 1).



Slika 1. *Ulaz u Transformer je sekvenca od više tokena, gde jedan token* $\boldsymbol{t_i}$ *sadrži niz vrednosti metrika koda* $\boldsymbol{m_{i,j}}$ *(**i** predstavlja broj revizije tj. commit-a, a **j** predstavlja vrstu metrike). Izlaz iz Transformera je predikcija da li postoji code smell na posmatranom fragmentu koda ili ne.*

Studija slučaja je izvršena na primeru detekcije klase sa više odgovornosti (*God Class*). Skup podataka koji se koristi je predstavljen u radu [8]. Model je evaluiran unakrsnom evaluacijom, pri čemu se vodilo računa da instance (klase) iz jednog projekta budu isključivo u istom delu (*fold*-u).

Postignuti rezultati pokazuju da je rad sa Transformerima na klasifikaciji vremenski zavisnih numeričkih vrednosti kompleksniji u odnosu na klasične NLP (*Natural Language Processing*) zadatke, pri čemu su istaknuti problemi sa tokenima i sekvencama u skupu podataka prilikom rada sa numeričkim podacima.

U narednom poglavlju dat je pregled prethodnih rešenja i poređenje postojećih pristupa sa pristupom opisanim u ovom radu. Specifikacija i implementacija sistema opisana je u poglavlju 3, a poglavlje 4 sadrži verifikaciju i analizu dobijenih rezultata. Poglavlje 5 predstavlja sumarizaciju celog rada.

2. PRETHODNA REŠENJA

U ovom poglavlju predstavljena su prethodna rešenja koja su se bavila detekcijom indikatora loše dizajniranog koda zasnovana na mašinskom učenju. Pošto ne postoji mnogo rešenja koja koriste istoriju promena koda za detekciju *code smell*-ova, razmotrena su i rešenja koja koriste sličan pristup u cilju detekcije drugih vrsta nedostataka u kodu.

U radu [9] su autori predstavili alat za prevenciju *code smell*-ova COSP (***CO**de **S**mell **P**redictor*) koji za datu klasu predviđa da li će sadržati neki tip *code smell*-a u narednih *t* dana. Za implementaciju COSP alata su koristili *Weka* [1] implementaciju *Random Forest* klasifikatora. Obeležja koja se ekstrahuju iz koda su strukturne metrike i istorijske metrike (vrednosti strukturnih metrika u *n* prethodnih *commit*-ova), što predstavlja sličnost sa pristupom opisanim u ovom radu.

Autori rada [7] predstavili su CAME [2] (***C**onvolutional **A**nalysis of code **M**etrics **E**volution)*, pristup zasnovan na dubokom učenju (*deep-learning*) koji se oslanja na strukturne i istorijske vrednosti metrika u cilju detektovanja *code smell*-ova. Za merenje performansi je upotrebljena F-mera, kao i u ovom radu. Autori su istakli pozitivan uticaj istorijskih metrika na detekciju. Za ekstrahovanje vrednosti metrika koda koristili su *RepositoryMiner*[3] alat, koji prolazi kroz istoriju revizija (*commit*-ova) i računa vrednosti metrika u svakoj reviziji. *RepositoryMiner* je upotrebljen i u ovom radu.

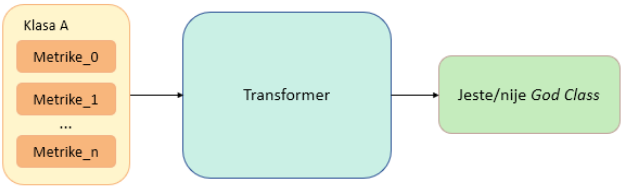
Autori rada [6] su predstavili pristup za predviđanje defekata u kodu na osnovu istorije promena, koristeći rekurentnu neuronsku mrežu (*Recurrent Neural Network*, RNN). Promene koda se posmatraju na nivou jedne datoteke, tako da dužina sekvence odgovara broju promena nad datotekom. Za svaku promenu se računaju vrednosti metrika, dakle jedna promena je vektor vrednosti metrika i predstavlja reč u sekvenci. Sličan pristup je primenjen i u ovom radu, gde se za svaku reviziju (*commit*) ekstrahuju vrednosti metrika i taj vektor vrednosti metrika predstavlja reč u sekvenci. Dužina sekvence zavisi od broja promena i samim tim nije fiksne dužine.

U radu [5] je korišćena RNN za detekciju defekata u kodu na osnovu istorije promena koda. Dužina sekvence je određena brojem verzija datoteke i nije fiksne dužine (neke datoteke postoje duže od drugih, te imaju dužu sekvencu). Kada se konstruišu sekvence, gledaju se samo datoteke koje su aktuelne („žive“) u posmatranoj verziji, a ostale se zanemaruju. Jedna reč (token) u sekvenci je vektor vrednosti metrika izračunatih za datoteku u nekoj verziji. U radu je rečeno da je prosečna dužina sekvence između 3 i 5.

U svim navedenim radovima, sem u radu [7] gde je korišćen *cross-project*[4], korišćen je *within-project*[5] pristup evaluacije modela. Pored toga, u svim radovima je korišćena F-mera za evaluaciju modela.

3. SPECIFIKACIJA I IMPLEMENTACIJA

Tema ovog rada je detekcija indikatora loše dizajniranog koda (*code smell*) bazirana na istoriji promena. Studija slučaja rađena je na jednoj vrsti *code smell*-a koja predstavlja klasu sa puno odgovornosti (*God Class*). Osnovna ideja je upotreba ML modela koji će za neku instancu prediktovati da li je posmatrana instanca zahvaćena *code smell*-om ili ne. U našem slučaju, instanca koju posmatramo je klasa, a *code smell* je *God Class*. Podaci koji su potrebni ML modelu za predikciju su vrednosti strukturnih metrika za datu klasu. Pošto smo rekli da je detekcija zasnovana na istoriji promena, vrednosti strukturnih metrika se gledaju u više revizija softvera (*commit*-ova). Dakle, ulaz predstavljaju vrednosti metrika u *n* revizija za klasu A. Izlaz ML modela je binarna labela, odnosno pokazatelj da li je klasa zahvaćena *God Class code smell*-om ili nije. Za ML model odabran je Transformer. Slika 2 prikazuje ulaz i izlaz iz ML modela.



Slika 2. *Prikaz ulaza (sa vrednostima metrika za klasu A u n revizija) i izlaza iz Transformera.*

U potpoglavlju 3.1. se nalazi opis prikupljanja podataka potrebnih za obučavanje i evaluaciju modela, dok se u potpoglavlju 3.2. nalazi opis pretprocesiranja podataka. Potpoglavlje 3.3. sadrži opis sekvence i tokena, dok se arhitekturom modela bavimo u potpoglavlju 3.4. U potpoglavlju 3.5. se nalazi opis implementacije sistema, sa korišćenim bibliotekama i eksternim softverima.

3.1. Metrike i prikupljanje podataka

Za prikupljanje podataka korišćen je *RepositoryMiner*[6] alat koji je “iskopao” vrednosti 7 metrika za projekte iz skupa podataka [8], počevši od *commit*-a poslednjeg anotiranog izdanja, unazad 1000 *commit*-ova. Time su dobijene vrednosti metrika za svaku instancu (klasu) u svakom projektu u prethodnih 1000 *commit*-ova (ili manje ukoliko projekti nemaju toliko revizija). Metrike koje su korišćene su LOC (*Lines of Code*), NMD (*Number of Methods Declared*), NAD (*Number of Attributes Declared*), LCOM5 (*Lack of COhesion in Methods*), NADC (*Number of Associated Data Classes*), ATFD (*Access To Foreign Data*) i WMC (*Weighted Method Count*).

[1] https://www.cs.waikato.ac.nz/ml/weka/
[2] Implementacija CAME se može pronaći na https://github.com/antoineBarbez/CAME/
[3] Alat je *open-source* i može se pronaći na https://github.com/antoineBarbez/RepositoryMiner/

[4] Model se obučava na skupu podataka koji pripada jednom softverskom projektu, a evaluira na skupu podataka koji pripada drugom softverskom projektu
[5] Model se i obučava i evaluira na skupu podataka koji pripada istom softverskom projektu
[6] https://github.com/antoineBarbez/RepositoryMiner/

3.2. Pretprocesiranje podataka

Procenat koda koji je zahvaćen *God Class code smell*-om je veoma mali u odnosu na ukupan broj klasa u projektu, što rezultuje nebalansiranim skupom podataka. U cilju rešavanja ovog problema, korišćena je SMOTE tehnika [10] kojom su kreirane instance one klase koja je manje zastupljena (ovde je to oznaka da je klasa zahvaćena *code smell*-om). Izvršena je normalizacija podataka kako bi se vrednosti različitih metrika iz različitih opsega svele na opseg od 0 do 1. Diskretizacija podataka je primenjena kako bi se kontinualne vrednosti metrika svele na diskretne vrednosti. Diskretizacija je izvršena pristupom *equal-frequency* sa 50 grupa, po ugledu na [6]. Svi elementi su sortirani i podeljeni u 50 grupa, tako da svaka grupa sadrži približno sličan broj elemenata.

3.3. Sekvenca i token

Ulazna sekvenca sadrži vrednosti metrika za posmatrane instance u različitim revizijama. Dakle, za jednu instancu (klasu), potrebno je obezbediti vrednosti metrika u *n* revizija (*commit*-ova) softvera. Vrednosti metrika za jednu reviziju se smeštaju u vektor i predstavljaju jedan token u ulaznoj sekvenci. Sekvenca sadrži više tokena i njena dužina zavisi od broja revizija softvera u kojima posmatrana instanca postoji.

3.4. Arhitektura modela

Arhitektura Transformera je izmenjena kako bi se prilagodila numeričkim podacima koji predstavljaju vrednosti metrika u *commit*-ovima. S obzirom na to da ne radimo sa prirodnim jezicima, već sa sekvencama koje sadrže vektore numeričkih vrednosti, uklonjen je sloj za *word embedding*. Pošto se vrši binarna klasifikacija, na izlazu iz Transformera se nalazi *sigmoid* funkcija koja daje vrednosti između 0 i 1. Rezultat ove funkcije predstavlja verovatnoću, odnosno, ako je vrednost iznad 0.5, izlaz iz modela će biti "jeste *God Class*".

3.5. Implementacija sistema

Za implementaciju sistema korišćen je programski jezik *Python* i biblioteke *NumPy*[7], *Scikit-learn*[8] i *PyTorch*[9], koje su služile za pretprocesiranje podataka, računanje F-mere, kreiranje, obučavanje i evaluaciju modela. Korišćena je *imbalanced-learn*[10] implementacija SMOTE tehnike za balansiranje podataka. *RepositoryMiner*[11] alat korišćen je za ekstrakciju vrednosti metrika koda iz istorije *commit*-ova.

4. VERIFIKACIJA I REZULTATI

U ovom radu je izvršena studija slučaja na detekciji klasa sa mnogo odgovornosti (*God Class*), te je u potpoglavlju 4.1. opisan ovaj *code smell*. Potpoglavlje 4.2. sadrži opis skupa podataka, a opis eksperimenta i analiza rezultata se nalaze u potpoglavlju 4.3.

4.1. Klasa sa mnogo odgovornosti (*God Class*)

God Class predstavlja klasu koja implementira veliki broj odgovornosti i prisvaja logiku čitavog sistema. Ovakve klase najčešće odlikuje veliki broj atributa i metoda koje međusobno nisu povezane, tj., klasa ima nizak nivo kohezije[12]. Pored toga, *God Class* je klasa koja je uvezana sa velikim brojem ostalih komponenti u sistemu, što ukazuje na visoku spregnutost[13]. Pošto ove klase implementiraju veći broj odgovornosti, one mogu postati poprilično velike, čime se otežava razumevanje i održavanje sistema. Zbog više funkcionalnosti koje implementiraju, klase su podložne većem broju izmena, što povećava rizik od nastajanja grešaka. Izmene koje je potrebno izvršiti ne predstavljaju trivijalan zadatak, jer klase imaju slabu koheziju i jaku spregnutost, čime je otežana reorganizacija i refaktorisanje softvera.

4.2. Skup podataka

U ovom radu je korišćen skup podataka sastavljen od strane autora rada [8]. Skup obuhvata 17 350 instanci iz 395 izdanja 30 projekata napisanih u programskom jeziku *Java*. Projekti su raznovrsni po pitanju veličine, životnog veka, domena i zajednice koja ga razvija. Skup podataka sadrži anotacije za 13 tipova *code smell*-ova, među kojima je i *God Class*. U nekoliko projekata nije zabeležena nijedna pojava *God Class code smell*-a, dok u nekim projektima broj detektovanih instanci iznosi više od 20 (maksimalan broj pojavljivanja je 26 u jednom od projekata).

4.3. Eksperiment i rezultati

U ovom radu je korišćen metod unakrsne validacije (*cross-validation*) za estimaciju ML modela. Skup podataka je podeljen na 5 delova (*folds*), pri čemu se vodilo računa da sve instance jednog projekta budu u istom delu. Obučavanje i evaluacija modela je izvršena u 5 iteracija, a u svakoj iteraciji je jedan deo bio korišćen za testiranje, dok su preostala 4 služila za obučavanje modela. Mera koja je odabrana za evaluaciju modela u ovom radu je F-mera. Računa se makro F-mera, odnosno, vrši se unakrsna validacija sa 5 iteracija, te se računa F-mera u svakoj iteraciji, a konačan rezultat je prosek svih dobijenih F-mera.

Prilikom testiranja modela nad testnim podacima dobili smo veoma nisku F-meru u svih 5 iteracija eksperimenta. Tražeći uzrok loših rezultata, analizirali smo tokene i sekvence koje se nalaze u trening i test delu skupa podataka. Za svaki unikatan token i unikatnu sekvencu iz trening skupa podataka izračunato je koliko puta se taj token javlja u trening i test podacima. Za svaki token i sekvencu je izračunat maksimalan, minimalan i prosečan broj ponavljanja u trening i test delu skupa podataka. Iz dobijenih podataka (Tabela 1 i Tabela 2) se može zaključiti da je problem u tome što model nije dovoljan broj puta video tokene iz test skupa, odnosno, prosečan broj ponavljanja tokena u skupu podataka je veoma mali. Pored toga, prosečan broj ponavljanja sekvenci u skupu podataka je vrlo mali, što negativno utiče na sposobnost modela da nauči i prediktuje odgovarajuće vrednosti.

[7] https://numpy.org/
[8] https://scikit-learn.org/stable/
[9] https://pytorch.org/
[10] https://imbalanced-learn.org/stable/
[11] https://github.com/antoineBarbez/RepositoryMiner/

[12] Metode i atributi u klasi su međusobno povezani ako većina metoda koristi većinu atributa. U najboljem slučaju, sve metode jedne klase treba da koriste sve atribute te klase.
[13] Spregnutost se odnosi na međuzavisnost između modula.

Tokom analize tokena i sekvenci smo menjali broj grupa (binova) koji se koriste prilikom diskretizacije, u cilju pronalaska potencijalnog rešenja u problemu koji smo uvideli. Smanjivanjem broja binova povećava se prosečan broj pojavljivanja tokena u trening i test delu skupa. Međutim, ni u slučaju najmanjeg isprobanog broja binova (5), rezultati nisu bili značajno bolji.

Tabela 1. *Analiza tokena iz trening skupa podataka*

Broj binova u diskretizaciji	50	40	30	20	10	5
Unikatni tokeni u treningu	26 645	25 032	22 230	17 306	6279	1038
Tokeni samo u treningu	26 037	24 298	21 218	15 407	4084	444
Tokeni u treningu i testu	608	734	1012	1899	2195	594
Max pojavljivanje tokena u testu	193	200	202	207	235	376
Min pojavljivanje tokena u testu	1	1	1	1	1	1
Prosečno pojavljivanje tokena u testu	2	2	2	2	3	15
Max pojavljivanje tokena u treningu	1173	1187	1295	1187	2032	2484
Min pojavljivanje tokena u treningu	1	1	1	1	1	1
Prosečno pojavljivanje tokena u treningu	2	2	3	4	11	69

Tabela 2. *Analiza sekvenci iz trening skupa podataka*

Broj binova u diskretizaciji	50	40	30	20	10	5
Unikatne sekvence u treningu	14 132	14 098	14 070	13 997	13 670	12 426
Sekvence samo u treningu	14 098	14 060	14 021	13 928	13 530	12 098
Sekvence u treningu i testu	34	38	49	69	140	328
Max pojavljivanje sekvenci u testu	44	44	44	45	49	76
Min pojavljivanje sekvenci u testu	1	1	1	1	1	1
Prosečno pojavljivanje sekvenci u testu	3	3	3	2	2	2
Max pojavljivanje sekvenci u treningu	702	702	702	702	702	702
Min pojavljivanje sekvenci u treningu	1	1	1	1	1	1
Prosečno pojavljivanje sekvenci u treningu	1	1	1	1	1	1

Transformeri daju odlične rezultate u oblasti NLP-a (*Natural Language Processing*) zbog tokena koji se ponavljaju (*attention* mehanizam u Transformeru reaguje na identične tokene). Broj tokena kod prirodnih jezika je ograničen, međutim, u slučaju numeričkih podataka je potrebno pripremiti i ograničiti rečnik tokena, kako bi Transformeri dali dobre rezultate prilikom klasifikacije vremenski zavisnih numeričkih podataka. Pored toga, u ovom radu nije izvršeno predtreniranje modela, što može biti uzrok loših rezultata. Ideja za dalji razvoj jeste da se odradi predtreniranje modela, kao i da se ograniči i pripremi rečnik tokena tako da skup podataka bude pogodniji za obučavanje Transformera.

5. ZAKLJUČAK

U ovom radu je predstavljen model za automatsku detekciju indikatora loše dizajniranog koda baziranu na istoriji promena koda. Korišćen je skup podataka iz rada [8], a arhitektura modela bazirana je na Transformeru [11]. Za obučavanje i evaluaciju modela korišćen je *cross-project* pristup, a za evaluaciju modela se računa makro F-mera tokom unakrsne validacije sa 5 iteracija.
Rezultati pokazuju da je rad sa Transformerima na klasifikaciji vremenski zavisnih numeričkih vrednosti kompleksniji u odnosu na klasične NLP (*Natural Language Processing*) zadatke. Istaknuti su problemi sa tokenima i sekvencama u skupu podataka prilikom rada sa numeričkim podacima, kao i mogući dalji koraci razvoja i unapređenja.

6. LITERATURA

[1] Martin, R.C., 2002. Agile software development: principles, patterns, and practices. Prentice Hall.

[2] Fowler, M., 2018. Refactoring: improving the design of existing code. Addison-Wesley Professional.

[3] Fernandes, E., Oliveira, J., Vale, G., Paiva, T., and Figueiredo, E., 2016, June. A review-based comparative study of bad smell detection tools. In Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (p. 18.). ACM.

[4] Azeem, M.I., Palomba, F., Shi, L. and Wang, Q., 2019. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. Information and Software Technology.

[5] Lui, Y., Li, Y., Guo, J., Zhou, Y., Xu, B., 2018. Connecting Software Metrics across Versions to Predict Defects. In 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp. 232-243). IEEE.

[6] Wen, M., Wu, R. and Cheung, S.C., 2018. How well do change sequences predict defects? Sequence learning from software changes. IEEE Transactions on Software Engineering.

[7] Barbez, A., Khomh, F. and Guéhéneuc, Y.G., 2019. Deep Learning Anti-patterns from Code Metrics History. IEEE International Conference on Software Maintenance and Evolution (ICSME) pp. 114-124.

[8] Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R. and De Lucia, A., 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empirical Software Engineering, 23(3), pp.1188-1221.

[9] Pantiuchina, J., Bavota, G., Tufano, M. and Poshyvanyk, D., 2018. Towards Just-In-Time Refactoring Recommenders. ACM/IEEE 26th International Conference on Program Comprehension.

[10] Chawla, N. V., Bowyer, K. W., Hall, L. O., Kegelmeyer, W. P., 2002. SMOTE: Synthetic Minority Over-sampling Technique. Journal of Artificial Intelligence Research 16 (2002): 321–357.

[11] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A., Kaiser, L., Polosukhin, I., 2017. Attention Is All You Need.

Kratka biografija:

Simona Prokić rođena je 1996. godine. Osnovne akademske studije završila je 2019. godine na Fakultetu tehničkih nauka, na kom brani i master rad 2020. godine iz oblasti Elektrotehnike i računarstva – Softversko inženjerstvo i informacione tehnologije.
Kontakt: simona.prokic@uns.ac.rs