Odkrivanje kode ChatGPT: Tehnike za odkrivanje vira kode

Alen Petek
alen.petek2@student.um.si
Fakulteta za elektrotehniko, računalništvo in informatiko, Univerza v Mariboru
Maribor, Slovenija

Nea Nikolić
nea.nikolic@student.um.si
Fakulteta za elektrotehniko, računalništvo in informatiko, Univerza v Mariboru
Maribor, Slovenija

Andraž Podpečan
andraz.podpecan1@student.um.si
Fakulteta za elektrotehniko, računalništvo in informatiko, Univerza v Mariboru
Maribor, Slovenija

Matevž Sladič
matevz.sladic@student.um.si
Fakulteta za elektrotehniko, računalništvo in informatiko, Univerza v Mariboru
Maribor, Slovenija

Rok Žerdoner
rok.zerdoner@student.um.si
Fakulteta za elektrotehniko, računalništvo in informatiko, Univerza v Mariboru
Maribor, Slovenija

POVZETEK

V tem delu smo analizirali možnosti samodejnega prepoznavanja kode, ustvarjene z jezikovnim modelom ChatGPT. Na osnovi odprtokodne rešitve smo rekonstruirali eksperiment, v katerem strojni učni model razvršča odseke programske kode glede na to, ali so bili ustvarjeni s strani človeka ali umetne inteligence. Sledili smo metodologiji, predstavljeni v članku ChatGPT Code Detection: Techniques for Uncovering the Source of Code [1], in uporabili njihovo implementacijo za ponovno izvedbo ter evalvacijo klasifikacijskih modelov. Cilj naloge je bil preveriti zanesljivost detekcije, ovrednotiti učinkovitost uporabljenih metod, razumeti njihove omejitve ter implementirati lastno rešitev in nadgradnjo obstoječe kode. Rezultati kažejo, da je z ustreznimi pristopi mogoče zaznati AI-generirano kodo z visoko natančnostjo, a obstajajo izzivi pri stilno izenačenih ali kompleksnejših primerih.

KLJUČNE BESEDE

umetna inteligenca, generirana programska koda, prepoznavanje izvora, ChatGPT, strojno učenje, TF-IDF, ADA, embeddings

1 UVOD

Veliki jezikovni modeli (LLM)[2], kot je ChatGPT, so v zadnjih letih postali izjemno zmogljivi pri generiranju naravnega jezika in s tem tudi programsko kodo[3]. Čeprav je osnovna funkcionalnost teh modelov zasnovana kot pomoč programerjem, se vse pogosteje uporabljajo tudi za avtomatsko generiranje celotnih rešitev programerskih nalog. Zaradi tega je prepoznavanje izvora kode – torej ali jo je napisal človek ali umetna inteligenca – postalo pomembno vprašanje na področjih izobraževanja, programerskih tekmovanj in celo v industrijskih okoljih[4].

V tej nalogi preučujemo problem razločevanja med človeško in AI-generirano programsko kodo. Kot izhodišče uporabljamo raziskavo Marca Oedingena in sodelavcev [1], ki predlagajo metodologijo za klasifikacijo izvorne kode na podlagi vektorskih predstavitev (embeddings), uporabe kodnih značilnosti in nadzorovanega učenja. Ponovno smo implementirali in validirali njihove postopke na lastnem eksperimentalnem naboru ter preverili delovanje različnih klasifikatorjev, kot so XGBoost, Random Forest, DNN in Gaussian Mixture Models, s kombinacijo klasičnih in kontekstualnih vektorjev, kot so TF-IDF, Word2Vec in OpenAI ADA.

S tem delom želimo prispevati k razumevanju, kako zanesljivo lahko ločimo umetno generirano kodo od človeške, ter nadgraditi obstoječo rešitev z uporabo večih učnih modelov in težišč modelov.

2 SORODNI ČLANKI

Raziskave na področju prepoznavanja umetno generirane kode so v porastu, saj postaja umetna inteligenca čedalje bolj vpeta v avtomatsko generiranje programske kode. Osrednji izziv teh raziskav je razviti zanesljive metode, ki lahko zanesljivo razločijo med kodo, ki jo je napisal človek, in tisto, ki jo je ustvaril jezikovni model, kot je v našem primeru ChatGPT. Preučili smo tudi možnosti, da bi s pomočjo umetne inteligence zaznavali generirano programsko kodo, saj v sklopu našega dela želimo s pomočjo LLM zaznavati kodo, ki je bila generirana s pomočjo umetne inteligence, ki sama temelji na LLM. To strategijo smo raziskali v članku Fighting fire with fire: can ChatGPT detect AI-generated text? [5].

2.1 Ali si to ti, LLM?

2.1.1 Prepoznavanje ChatGPT-generirane kode. Ena izmed osrednjih referenc za naše delo je članek ChatGPT Code Detection: Techniques for Uncovering the Source of Code [1], kjer avtorji predstavljajo različne pristope k detekciji izvora kode. Njihova metodologija temelji na raznolikem naboru predstavitev kode — od klasičnih pristopov, kot je termin frekvenca–inverzna frekvenca dokumenta (TF-IDF), do naprednejših semantičnih vektorjev, kot je OpenAI ADA embeddings [6]. Te vektorske reprezentacije nato uporabijo kot vhod za različne klasifikatorje.

Pomembna prednost njihovega pristopa je uporaba tako sintaktičnih kot semantičnih informacij o kodi. Ugotovili so, da kombinacija obeh vrst značilk močno poveča natančnost klasifikacije. Vendar pa rezultati kažejo tudi na meje teh tehnik — zlasti kadar je koda stilistično preurejena ali dopolnjena z ročnimi spremembami, se zanesljivost prepoznave zmanjša.

2.1.2 Vloga rekonstrukcije v strojno učnih modelih. Čeprav se osrednji članek ne osredotoča neposredno na rekonstrukcijo, pa je

ta koncept ključnega pomena v številnih drugih modelih, ki obravnavajo obnavljanje oziroma interpretacijo latentne strukture podatkov. Poudarjen primer tega pristopa je raziskava Diffusion probabilistic model made slim [7], ki rekonstrukcijo postavlja v središče modeliranja dinamičnih procesov.

V tej študiji avtorji obravnavajo problem napovedovanja gibanja delcev na osnovi redkih ali šumnih opazovanj. Z uporabo denoising difuzijskega modela (SDDPM) rekonstruirajo verjetne trajektorije gibanja tako, da iz šumnih začetnih signalov generirajo gladke, fizikalno smiselne poti. Ključno pri tem je, da rekonstrukcija ni ločen korak pred učno fazo, temveč integriran del modela, kar omogoča boljšo generalizacijo.

Prenos te zamisli na področje detekcije AI-generirane kode je konceptualno zelo zanimiv. Namesto da bi klasifikator deloval neposredno na vhodni kodi, bi lahko rekonstrukcijski model najprej poskušal obnoviti "čist" latentni vzorec — bodisi človekovega stila kodiranja ali značilnega sloga AI. Tak pristop bi lahko pripomogel k večji robustnosti klasifikacije, saj bi se klasifikator osredotočal na bolj stabilne značilnosti, ki niso neposredno občutljive na stil ali obliko.

2.1.3 Možnosti integracije rekonstrukcije v prihodnje detektorje kode. Na osnovi omenjenih raziskav je mogoče domnevati, da bi se prihodnji sistemi za zaznavo AI-generirane kode lahko gibali v smeri dvostopenjskih arhitektur. Prva stopnja bi izvajala rekonstrukcijo, bodisi preko difuzijskih modelov, autoencoderjev ali drugih generativnih pristopov. Druga pa bi izvajala klasifikacijo na osnovi rekonstruktiranih značilnosti. Tovrsten pristop bi omogočal ne samo višjo natančnost, temveč tudi večjo interpretabilnost rezultatov.

2.2 Detekcija kode s kontrastnim učenjem

2.2.1 Detekcija s klasično klasifikacijo. Študija ChatGPT Code Detection [1] postavlja temelje za klasični pristop k detekciji AI-generirane kode. Avtorji uporabljajo različne predstavitve kode, med drugim:

- TF-IDF kot bag-of-words model, ki temelji na frekvenci žetonov;
- OpenAI Ada embeddings kot semantični vektorski prikaz kode;
- Statistične značilke (dolžina, število komentarjev, razmerje med vrsticami itd.).

Ti prikazi so nato podani kot vhod v klasifikatorje (XGBoost, DNN, Random Forest), ki napovedujejo, ali je koda nastala s ChatGPT. Avtorji eksperimentirajo tudi z fine-tuningom LLM modelov, vendar ugotovijo, da enostavnejši modeli pogosto dosegajo primerljive rezultate.

Glavna pomanjkljivost pristopa je njegova občutljivost na površinske spremembe v kodi, kot so komentarji ali zamenjave imen spremenljivk, kar pomeni, da modeli večinoma prepoznavajo stil in ne globoko semantično strukturo.

2.2.2 Detekcija s kontrastnim učenjem. Tretji pomemben pristop k detekciji je kontrastno učenje, kot ga predstavijo Xu in sod. v študiji Distinguishing LLM-generated from Human-written Code by Contrastive Learning[8] . Glavni prispevki te študije so:

- Zbirka podatkov HMCorp: 550.000 parov funkcij, generiranih v Pythonu in Javi, kjer je vsak par sestavljen iz človeško in LLM-generirane implementacije iste naloge.
- Model CodeGPTSensor: Temelji na enkoderju UniXcoder, ki kodo pretvori v semantične vektorje.
- Kontrastni okvir: Model se uči razlikovati med človeško in AI-generirano kodo tako, da zmanjšuje razdaljo med podobnimi primeri (npr. človeška-človeška) in povečuje razdaljo med različnimi (človeška-AI). vrsticami itd.).

Ključno je, da se model ne uči le prepoznavati značilnosti ene vrste kode, ampak neposredno razliko med vrstama. To pomeni, da je poudarek na relacijskem znanju, ne absolutnem.

Ta pristop implicitno rekonstruira pomembne ločevalne lastnosti med obema vrstama kode, brez da bi moral neposredno generirati novo kodo. Lahko bi rekli, da kontrastno učenje nadomešča eksplicitno rekonstrukcijo z diskriminativno "rekonstruirano razliko" med obema izvoroma [9].

2.3 Zaključek pregleda literature

Vsak od treh pristopov: klasifikacija, rekonstrukcija in kontrastno učenje, prinaša edinstven pogled na problem detekcije AI-generirane kode. Medtem ko klasične metode izkoriščajo površinske značilnosti in so enostavne za implementacijo, rekonstrukcijski pristopi ponujajo globlje semantično razumevanje in večjo odpornost na preoblikovanje. Kontrastno učenje pa uspešno združuje semantično razlikovanje in visoko klasifikacijsko zmogljivost brez nujne potrebe po generativnem modelu.

To delo gradi predvsem na metodologiji Oedingen in sod. [1], a se usmerja v smer rekonstrukcije ter razmišlja o prihodnji integraciji konceptov kontrastnega učenja. Cilj je izboljšati zanesljivost detekcije s prehodom iz površinskih značilnosti na globlje, latentne vzorce pisanja kode.

3 PONOVITEV EKSPERIMENTA

V okviru prvega dela našega eksperimenta smo si zadali cilj rekreacije originalnega dela[1]. Avtorji so v svoji študiji predstavili več metod za klasifikacijo izvorne kode kot človeško narejeno ali umetno generirane, pri čemer so uporabili različne kombinacije vektorizacijskih pristopov (npr. TF-IDF, embeddingi) in klasifikacijskih modelov (od klasičnih do DNN). Pomemben doprinos njihovega dela, k našemu eksperimentu je javna dostopnost kode in korpus generirane in človeško narejene programske kode, ki jih ponujajo v svojem repozitoriju GitHub – ChatGPT Code Detection, kar smo uporabili kot osnovo za našo rekonstrukcijo.

3.1 Tehnična izvedba

Pri rekonstrukciji eksperimenta smo sledili dokumentaciji objavljeni na repozitoriju ter poskušali v celoti reproducirati okolje in postopke. Vendar pa je rekonstrukcija naletela na več ovir, predvsem na področju strojne opreme in nekompatibilnosti določenih knjižnic:

- Avtorji priporočajo uporabo sistema z vsaj 32 GB RAM, saj je procesiranje embeddingov (posebej pri modelih, kot je Ada v kombinaciji z DNN) pomnilniško zelo zahtevno.

- V našem primeru smo uporabljali sistem z omejenimi viri (cca. 16 GB RAM), zaradi česar nekateri deli kode niso delovali brez prilagoditev.
- Poleg tega so bile nekatere knjižnice zastarele ali v neskladju z novejšimi verzijami Pythona, kar je zahtevalo dodatne nadgradnje in zamenjave.

Kljub omenjenim izzivom nam je uspelo rekonstruirati večino eksperimenta, največje težave nam je povzročilo formatiranje kode.

V celotnem postopku smo uporabljali tri različne predstavitve podatkov (ang. feature extraction methods):

(1) Ročno izbrani atributi (features) – metrika iz same strukture kode (npr. število zank, funkcij, dolžina kode).
(2) TF-IDF – vektorizacija izvorne kode kot tekstovnega dokumenta.
(3) Ada Embedding – vnaprej naučeni besedilni embeddingi, pridobljeni z uporabo modela iz družine OpenAI Ada[10].

Na teh predstavitvah smo preizkusili več klasifikatorjev:

- Logistična regresija (LR)
- Odločitvena drevesa (DT)
- Naključni gozd (RF)
- Boosting metode
- Nevronske mreže (DNN)
- Gaussov model mešanice (GMM)
- Optimalno stroškovno občutljivo obrezano drevo (OPCT)
- Bayesov klasifikator

Za vsako kombinacijo predstavitve in modela smo izvedli več ponovitev (5-kratno navzkrižno validacijo), na podlagi katerih smo izračunali povprečje in standardni odklon osnovnih metrik: natančnost (accuracy), preciznost (precision), priklic (recall), F1-mera in Area Under Curve (AUC).

3.2 Rezultati

Rezultati so pokazali, da je mogoče tudi z omejenimi računalniškimi viri doseči primerljive uspešnosti kot v originalnem članku. Najuspešnejše kombinacije smo dosegli z uporabo TF-IDF in Ada embeddingov, skupaj z naprednimi klasifikatorji, kot sta XGBoost in DNN.

Iz sledečih tabel rezultatov, lahko razberemo, da je ponovitev eksperimenta pokazala zmanjšanje natančnosti v primerjavi z izvirno implementacijo. Prav tako smo zaznali povečanje standardnega odklona med posameznimi preizkusi, kar nakazuje na večjo nestabilnost modela. Ti rezultati kažejo, da kljub omejitvam strojne opreme model ohranja funkcionalnost zaznavanja generirane kode, vendar je manj zanesljiv in občutljivejši na spremembe v podatkih.

Izpostavimo lahko naslednje najpomembnejše rezultate:

features - original

	Accuracy	Precision	Recall	F1	AUC
DT	82,54±0,23	82,70±0,36	82.29±0.57	82.49±0.26	82.54±0.23
DNN	85.37±0.80	84.58±2.73	86.84±4.77	85.54±1.21	93.73±0.42
GMM	79.68±0.50	74.74±0.66	89.74±0.58	81.55±0.39	89.05±0.25
LR	76.69±0.63	74.80±0.64	80.50±0.94	77.54±0.63	84.57±0.41
OPCT	84.12±0.60	80.89±1.95	89.52±2.37	84.94±0.43	89.60±0.81
RF	88.10±0.40	87.29±0.41	89.19±0.66	88.23±0.41	95.34±0.22
XGB	**88.48±0.26**	**87.39±0.46**	**89.93±0.42**	**88.64±0.24**	**95.59±0.20**

features - rekreacija

	Accuracy	Precision	Recall	F1	AUC
DT	83.36±0.24	83.73±0.26	82.81±0.66	83.26±0.30	83.36±0.24
DNN	88.36±0.31	86.32±1.31	91.22±1.59	88.68±0.29	95.56±0.18
GMM	86.96±0.35	87.27±0.43	86.53±0.57	86.9±0.36	94.12±0.2
LR	78.68±0.69	77.13±0.86	81.56±0.83	79.28±0.64	85.93±0.5
OPCT	85.27±0.64	81.12±1.24	**91.99±1.39**	86.2±0.53	91.11±0.3
RF	88.39±0.37	**87.35±0.44**	90.31±0.77	**88.8±0.38**	**95.71±0.15**
XGB	**88.61±0.36**	86.78±0.34	90.59±0.6	88.64±0.38	95.68±0.17

Table 1: Primerjava - features

ADA - original

	Accuracy	Precision	Recall	F1	AUC
DT	80.82±0.50	80.63 ±0.55	81.46 ±0.67	79.83±1.03	80.82±0.50
DNN	**97.79±0.36**	**97.40±0.77**	**98.22±0.60**	**97.80±0.35**	**99.76±0.05**
GMM	92.71±0.39	95.10±0.48	90.06±0.66	92.51±0.42	97.37±0.21
LR	95.87±0.13	95.93±0.13	94.60±0.24	97.30±0.25	99.06±0.077
OPCT	95.53±0.27	95.59±0.25	94.35±0.81	96.87±0.65	97.24±0.466
RF	96.87±0.65	97.24±0.46	90.67 ±0.63	94.52±0.55	97.85±0.145
XGB	95.05±0.23	95.09±0.22	94.30±0.42	95.89±0.30	98.94±0.09

ADA - rekreacija

	Accuracy	Precision	Recall	F1	AUC
DT	79.92±0.61	80.72±0.39	78.61±1.18	79.65±0.73	79.92±0.61
DNN	**97.95±0.25**	**97.86±0.7**	**98.06±0.67**	**97.95±0.25**	**99.79±0.04**
GMM	93.29±0.43	96.05±0.33	90.31±0.95	93.09±0.47	97.84±0.2
LR	96.34±0.24	95.27±0.34	97.52±0.34	96.38±0.24	99.15±0.1
OPCT	95.51±0.28	94.69±0.56	96.45±0.93	95.55±0.3	96.99±0.37
RF	91.48±0.27	91.02±0.38	92.05±0.39	91.53±0.27	97.4±0.14
XGB	94.66±0.36	94.35±0.47	95.0±0.39	94.68±0.35	98.8±0.143

Table 2: Primerjava - ADA

4 LASTNA IMPLEMENTACIJA

Za detekcijo generirane programske kode smo implementirali metodo, ki temelji na analizi različnih komponent programske kode z uporabo znakovnih in sintaktičnih n-gramov. Ključna predpostavka našega pristopa je, da so generirane in človeško napisane kode ločljive na osnovi statističnih vzorcev v treh komponentah:

- imena spremenljivk, funkcij in razredov,
- komentarji ter
- sintaktična struktura kode.

Po analizi sorodnih člankov, zlasti Distinguishing LLM-generated from Human-written Code by Contrastive Learning [8], smo ugotovili, da ti trije elementi najpogosteje odražajo stil pisanja in s tem tudi razlike med človeškim in avtomatsko generiranim programiranjem.

4.1 Potek analize

4.1.1 Tokenizacija in predobdelava. Nad vhodnim primerom izvedemo sintaktično tokenizacijo, s katero izluščimo tri vrste podatkov:

(1) zaporedje števil (tokenov), ki predstavlja sintaktično strukturo kode,
(2) seznam imen spremenljivk, funkcij in razredov,
(3) seznam komentarjev.

TF-IDF - original

	Accuracy	Precision	Recall	F1	AUC
DT	95.82±0.31	95.82±0.31	95.93±0.26	95.71±0.52	95.82±0.31
DNN	97.61±0.30	97.79±0.79	97.45±1.17	97.61±0.32	99.68±0.05
GMM	94.60±0.32	95.24±0.26	93.89±0.60	94.56±0.34	96.18±0.26
LR	97.06±0.24	97.09±0.24	96.11±0.28	98.09±0.31	99.46±0.05
OPCT	96.72±0.26	96.72±0.26	96.78±0.50	96.66±0.50	97.62±0.41
RF	98.04±0.11	98.04±0.11	97.77±0.18	98.31±0.23	99.72±0.05
XGB	**98.28±0.09**	**97.87±0.15**	**98.70±0.22**	**98.28±0.09**	**99.84±0.02**

TF-IDF - rekreacija

	Accuracy	Precision	Recall	F1	AUC
DT	95.73±0.23	95.91±0.39	95.54±0.36	95.72±0.23	95.73±0.23
DNN	97.74±0.28	97.66±0.66	97.84±0.92	97.74±0.29	99.66±0.06
GMM	94.46±0.29	95.47±0.39	93.36±0.51	94.4±0.29	96.32±0.24
LR	96.99±0.18	95.92±0.24	98.15±0.26	97.02±0.18	99.41±0.06
OPCT	96.62±0.27	96.55±0.67	96.69±0.58	96.62±0.26	97.53±0.24
RF	98.07±0.16	97.62±0.21	98.54±0.26	98.07±0.16	99.74±0.05
XGB	**98.34±0.2**	**97.93±0.26**	**98.77±0.3**	**98.34±0.2**	**99.87±0.03**

Table 3: Primerjava - TF-IDF

Pri predprocesiranju imen in komentarjev črke v nizih pretvorimo na male, skrajšamo zaporedja "praznih" znakov ter odstranimo vse ostale nečrkovne znake.

4.2 Generiranje profilov in značilk

Za vsako izmed komponent (komentarji, imena spremenljivk, sintaktične komponente) generiramo n-gram profile:

- Znakovni n-grami (n = 3 ali 4): uporabljeni za komentarje in imena spremenljivk.
- Tokenizirani n-grami (n = 2 ali 3): uporabljeni za sintakso, kjer analiziramo zaporedja ključnih besed in operatorjev.

Izračunamo frekvenčne profile in jih razvrstimo po frekvenci pojavljanja. Profil je predstavljen kot urejen seznam najpogostejših n-gramov.

Vhodni korpusi se razdelijo na dva referenčna profila:

- enega za programsko kodo generirano z umetno inteligenco
- drugega za človeško napisano programsko kodo.

4.3 Klasifikacija z metodo primerjave profilov

Za vsak nov primer programa izračunamo n-gram profil in ga primerjamo z obema referenčnima profiloma. Uporabimo rangirano razdaljo (Rank-biased Overlap oz. matriko podobnosti), ki meri, koliko so n-grami na podobnih pozicijah v obeh seznamih.

Za vsako komponento (komentarji, imena, sintaktičnih komponent) dobimo odločitev, ali je primer bolj podoben človeškemu ali generiranemu profilu. Uporabimo večinsko glasovanje (2 od 3 komponent), da določimo končno klasifikacijo. Ko preverimo podamo še v nevronsko mrežo, ki se poskuša naučiti koliko kateri model doprinese h končni rešitvi. Zaradi slabe prisotnosti komentarjev v učnem korpusu, samo cca. 10%, so ti pogosto bili manjšinski "glas" pri odločitvenemu modelu.

4.4 Rezultati in analiza

Rezultati klasifikacije so pokazali nižjo natančnost, kot smo sprva pričakovali, vendar se je še gibala v rangu sprejemljivosti. Glavna razloga za to sta verjetno izguba informacij pri sintaktični tokenizaciji, kjer se odstrani kontekst in struktura višjega reda ter manjša diskriminativna moč n-gramov nad imeni spremenljivk in komentarji, kjer razlike med generirano in človeško kodo niso tako izrazite.

Kljub izgubi natančnosti napovedovanja vira testne programske kode, je prednost naše implementacije optimizacija učinkovitosti in časa učenja, čas izvajanja procesa profiliranja in klasifikacije smo znižali iz več kot ur na le nekaj minut, odvisno od uporabljene strojne opreme. Kar smo dosegli z racionalizacijo podatkovne obdelave, zmanjšanjem števila n-gramov in uporabo hitrejših podatkovnih struktur.

predprocesirano

	dolžina n-gramov		
število n-gramov	3	4	5
300	Accuracy: 0.7147 Loss: 0.5451	Accuracy: 0.7540 Loss: 0.5254	Accuracy: 0.5004 Loss: 0.6851
500	Accuracy: 0.6768 Loss: 0.5637	Accuracy: 0.7388 Loss: 0.5126	Accuracy: 0.7353 Loss: 0.6675
700	Accuracy: 0.6683 Loss: 0.6072	Accuracy: 0.7336 Loss: 0.5006	Accuracy: 0.7350 Loss: 0.5225

Table 4: Rezultati - predprocesirani vhodni podatki

brez predprocesiranja

	dolžina n-gramov		
število n-gramov	3	4	5
300	Accuracy: 0.7303 Loss: 0.5228	Accuracy: 0.7426 Loss: 0.5272	Accuracy: 0.7044 Loss: 0.6445
500	Accuracy: 0.6920 Loss: 0.5460	Accuracy: 0.6477 Loss: 0.5885	Accuracy: 0.7142 Loss: 0.6650
700	Accuracy: 0.6818 Loss: 0.5993	Accuracy: 0.7076 Loss: 0.5579	Accuracy: 0.7182 Loss: 0.5457

Table 5: Rezultati - brez predprocesiranja vhodnih podatkov

4.5 Uporabljena orodja in podatki

- Koda je implementirana v Pythonu.
- Uporabljeni moduli: collections.Counter, re za regularne izraze, difflib za izračun podobnosti.
- Učni in testni korpus je identičen tistemu iz izvirne naloge, kar omogoča neposredno primerjavo.

4.6 Omejitve in pomankljivosti

Kljub uspešni implementaciji in optimizaciji detekcije generirane programske kode ima naš pristop več pomembnih omejitev, ki vplivajo na natančnost in splošno robustnost metode:

- Uporaba znakovnih ali tokeniziranih n-gramov ne zajame globljega semantičnega pomena kode. Dve zelo različni implementaciji lahko imata podobne n-gram vzorce, zlasti pri pogosto uporabljenih strukturah (npr. for, if, return).
- Pri komentarjih in imenih spremenljivk se pogosto pojavljajo podobne fraze ne glede na izvor (npr. „initialize variable“, „set flag“), kar zmanjšuje razlikovalno moč.

- V tokeniziranem nizu ni mogoče razločiti kam je postavljeno katero ime. To mogoče dodatno zabriše slogovne značilnosti pisca.
- To vodi v izgubo informacij, zaradi katere je model manj občutljiv na drobne slogovne razlike, ki so sicer lahko značilne za generirane vsebine.
- Ker se referenčni profili učijo iz konkretnega korpusa, metoda ni popolnoma prenosljiva na drugačne programske jezike ali domene brez ponovnega učenja.
- Če je učna množica premajhna ali neuravnotežena, to vodi v neuravnotežene profile, kar zmanjšuje učinkovitost pri klasifikaciji novih primerov.
- Sodobni modeli, kot je GPT-4 ali Claude, generirajo vedno bolj naravno in raznoliko kodo. To pomeni, da razlike med človeško in generirano kodo postajajo manj izrazite, zlasti na nivoju sintakse in komentarjev.
- Naša metoda tako postaja manj učinkovita pri novejših modelih, saj so ti sposobni posnemati slog in strukturo človeškega pisanja.
- Metoda deluje popolnoma lokalno na posameznih datotekah ali fragmentih kode, brez razumevanja širšega konteksta (npr. organizacije projektne strukture, povezav med datotekami).
- Detekcija generiranosti bi lahko bila učinkovitejša, če bi upoštevali globalne vzorce, kot so struktura imenovanj skozi celoten projekt, dolžina funkcij, konsistentnost komentarjev ipd.

5 ZAKLJUČEK

V našem delu smo se osredotočili na izboljšanje detekcije generirane programske kode s pomočjo n-gram analiz in ločene obravnave posameznih komponent programske kode – spremenljivk, komentarjev in sintakse. Uporabili smo obstoječi učni model, priložen učni korpus, ter izvedli pomembne spremembe na področju predobdelave in modeliranja značilk. Naš pristop temelji na predpostavki, ki smo jo zasledili v sledečih člankih [1, 8, 11], da so ravno ti trije elementi – še posebej poimenovanja spremenljivk in način pisanja komentarjev – ključni pokazatelji generirane programske kode.

Čeprav rezultati kažejo nekoliko nižjo natančnost v primerjavi z osnovnim modelom, kar pripisujemo izgubi informacij pri sintaktični tokenizaciji in omejitvam n-gramov, smo dosegli znatno izboljšavo na področju učinkovitosti učenja. Čas učenja modela smo zmanjšali iz več kot devetih ur na nekaj minut, kar pomeni veliko prednost pri eksperimentiranju in nadaljnjem razvoju.

Naš model smo učili na istem korpusu kot izvirni sistem, kar zagotavlja primerljivost rezultatov. Celoten pristop omogoča boljšo modularnost in možnost testiranja različnih kombinacij vhodnih podatkov (velikost n-gramov, števila najpogostejših n-gramov...), kar bi v prihodnje lahko še izboljšali z uvedbo dodatnih značilk ali uporabo bolj sofisticiranih modelov.

Naša implementacija kljub temu dokazuje, da so enostavni pristopi še vedno uporabni pri detekciji generirane kode, še posebej, ko so omejitve časa učenja ali razpoložljivih virov pomemben dejavnik.

KRATICE

- Large Language Model (LLM)
- Term Frequency–Inverse Document Frequency (TF-IDF)
- Speckle Denoising Diffusion Probabilistic Models (SDDPM)
- Deep Neural Network (DNN)
- Giga Byte (GB)
- Random Access Memory (RAM)
- Logistic regresion (LR)
- Decision Tree (DT)
- Random Forest (RF)
- Gaussian Mixture Model (GMM)
- Optimal Cost-sensitive Pruned Tree (OPCT)
- Area Under Curve (AUC)

LITERATURA

[1] M. Oedingen, R. C. Engelhardt, R. Denz, M. Hammer, and W. Konen, "Chatgpt code detection: Techniques for uncovering the source of code," *arXiv preprint arXiv:2405.15512*, 2024.

[2] H. Naveed, A. U. Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, N. Akhtar, N. Barnes, and A. Mian, "A comprehensive overview of large language models," *arXiv preprint arXiv:2307.06435*, 2023.

[3] M. S. I. Sakib, "What is chatgpt," *ResearchGate*, 2023.

[4] S. Cave and K. Dihal, "Hopes and fears for intelligent machines in fiction and reality," *Nature machine intelligence*, vol. 1, no. 2, pp. 74–78, 2019.

[5] A. Bhattacharjee and H. Liu, "Fighting fire with fire: can chatgpt detect ai-generated text?" *ACM SIGKDD Explorations Newsletter*, vol. 25, no. 2, pp. 14–21, 2024.

[6] R. Goel, "Using text embedding models as text classifiers with medical data," *arXiv preprint arXiv:2402.16886*, 2024.

[7] X. Yang, D. Zhou, J. Feng, and X. Wang, "Diffusion probabilistic model made slim," in *Proceedings of the IEEE/CVF Conference on computer vision and pattern recognition*, 2023, pp. 22 552–22 562.

[8] X. Xu, C. Ni, X. Guo, S. Liu, X. Wang, K. Liu, and X. Yang, "Distinguishing llm-generated from human-written code by contrastive learning," *ACM Transactions on Software Engineering and Methodology*, 2024.

[9] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," *arXiv preprint arXiv:2203.03850*, 2022.

[10] R. Aperdannier, M. Koeppel, T. Unger, S. Schacht, and S. K. Barkur, "Systematic evaluation of different approaches on embedding search," in *Future of Information and Communication Conference*. Springer, 2024, pp. 526–536.

[11] A. Gurioli, M. Gabbrielli, and S. Zacchiroli, "Is this you, llm? recognizing ai-written programs with multilingual code stylometry," *arXiv preprint arXiv:2412.14611*, 2024.