1. Uvod u Retrieval-Augmented Generation (RAG)

Pri istraživanju primjene velikih jezičnih modela (*eng. Large Language Model, LLM*) i umjetne inteligencije u visokoškolskom obrazovanju, neizbježna je tema RAG sustava. RAG (*eng. Retrieval-Augmented Generation*) predstavlja pristup u obradi prirodnog jezika (*eng. Natural Language Processing, NLP*) koji spaja sposobnosti velikih jezičnih modela i sustava za pretraživanje dokumenata.

Jedan od uobičajenih izazova u procesu učenja jest obilje dostupnih informacija, zbog čega je često teško identificirati točno one podatke koji su relevantni za određeno pitanje ili problem. RAG sustav rješava taj problem omogućujući interakciju s bazom znanja; korisnik postavlja pitanje, a prije generiranja odgovora, model prethodno pretražuje relevantne informacije u unaprijed definiranim izvorima, poput dokumenata ili baza znanja. Ovakav pristup značajno povećava točnost i relevantnost odgovora te smanjuje rizik od tzv. halucinacija, odnosno netočnih ili izmišljenih informacija koje se ponekad pojavljuju kod samostalnog generiranja teksta.

Pojam RAG razvio se u kontekstu napretka obrade prirodnog jezika, kada je uočeno da modeli poput GPT-a (*eng. Generative Pretrained Transformer*) uspješno generiraju tekst, ali često ne raspolažu specifičnim podacima iz stvarnog svijeta. Integracijom generativnih modela i *retrieval* sustava, RAG omogućuje razvoj interaktivnih sustava i *chatbotova* koji odgovaraju na korisnička pitanja temeljeći se na stvarnim, provjerenim dokumentima i izvorima informacija.

2. Osnovni princip rada RAG sustava

RAG sustav funkcionira kroz dva osnovna koraka; dobavljanje konteksta pretraživanjem baze znanja (*eng. retrieval*) i generiranje odgovora korištenjem pribavljenog konteksta (*eng. generation*).

U koraku dobavljanja konteksta, korisnički upit uspoređuje se s dokumentima pohranjenima u vektorskoj bazi podataka (*eng. vector store*). Dokumenti su prethodno razdvojeni na manje cjeline (tzv. *chunks*) i pretvoreni u numeričke vektore. Sustav vraća one dijelove dokumenata koji su najrelevantniji za postavljeni upit.

U koraku generiranja odgovora odabrani dijelovi dokumenata prosljeđuju se jezičnom modelu, koji na temelju njih generira kontekstualno točan i informativan odgovor.

Ovaj pristup omogućuje korisnicima postavljanje pitanja o dostupnim dokumentima i informacijama, dok *chatbot* pruža precizne odgovore bez potrebe za ručnim pretraživanjem sadržaja. Važno je napomenuti da je opisani primjer osnovni, jednostavni RAG sustav. Današnji sustavi često uključuju dodatne jezične modele, kako bi se poboljšala formulacija pitanja, procijenila kvaliteta odgovora ili optimizirao cijeli proces generiranja. Procesi pripreme i pohrane dokumenata u vektorsku bazu podataka, pretraživanje znanja i generiranje odgovora bit će detaljnije objašnjeni u nastavku.

3. Pregled izrade RAG sustava

Za izradu RAG sustava korišten je *LangChain*, *open-source* biblioteka koja omogućava lako korištenje i povezivanje različitih komponenti za obradu prirodnog jezika, rad s velikim

jezičnim modelima, dohvat, obradu, pohranu teksta i izgradnju RAG sustava. U *Jupyter* bilježnici isprobane su različite metode koje *LangChain* nudi, kako bi se na kraju sastavio funkcionalan RAG sustav.

U nastavku su prikazani sljedeći koraci: učitavanje dokumenata (*eng. Document Loading)*, razdvajanje dokumenata na dijelove (*eng. Document Splitting*), tokenizacija (*eng. Token Splitting*), izrada vektorskih reprezentacija i pohrana u vektorsku bazu (*eng. Vectorstores & Embeddings*), metode dohvaćanja relevantnih dokumenata (*eng. Retrieval*), metode generiranja odgovora (*eng. Generation*), interaktivni RAG obogaćen s poviješću razgovora, te utjecaj dizajna *prompta* na ponašanje jezičnog modela.

3.1 Učitavanje dokumenata

Za učitavanje PDF dokumenata koristi se *PyPDFLoader*. Svaka stranica PDF-a se učitava kao zaseban dokument s tekstom i pripadajućim metapodacima. *LangChain* može učitavati i druge formate, poput URL-ova.

3.2 Razdvajanje dokumenata, segmentacija

Dokumenti se razdvajaju u semantičke cjeline (*chunks*), kako bi LLM-ovi mogli raditi s tekstom u manjim i povezanijim segmentima. *LangChain* nudi različite *text splittere*, npr. *CharacterTextSplitter* i *RecursiveCharacterTextSplitter*.

Dok *CharacterTextSplitter* razdvaja dokumente fiksno po samom broju znakova, *RecursiveCharacterTextSplitter* bolje očuva kontekst i strukturu paragrafa definicijom različitih prioritetnih separatora. Za RAG sustav ovog rada korišten je rekurzivni *splitter* sa separatorima (separators="\n\n", "\n", "(?<=\.)", " ", ""), što znači da će prvo podijeliti tekst pomoću "\n\n", a ako je duljina bloka i dalje prevelika, podijelit će se na temelju sljedećeg separatora iz niza. Ovaj pristup osigurava da odlomci ostanu netaknuti i dijele se samo ako premaše određenu duljinu.

U rekurzivnom *splitteru*, parametar *chunk size* definira maksimalnu veličinu pojedinog tekstualnog isječka izraženu u broju znakova. Korištena vrijednost od 2000 znakova osigurava da svaki *chunk* sadrži dovoljno konteksta za smisleno razumijevanje sadržaja, ali istovremeno ne prelazi ograničenja kontekstnog prozora jezičnog modela.

Parametar *chunk overlap* određuje broj znakova koji se preklapaju između susjednih isječaka. Preklapanje omogućuje očuvanje kontinuiteta informacije na granicama *chunkova*, čime se smanjuje rizik gubitka važnog konteksta pri razdvajanju teksta. Na taj način model može razumjeti sadržaj koji se prostire preko više isječaka, što je osobito važno kod složenih pojmova i dužih objašnjenja. Korištena je vrijednost 200.

3.3 Segmentacija u tokene

Za modele s ograničenjem konteksta u tokenima, korisno je razdvajati tekst po tokenima. Tako *TokenTextSplitter* osigurava da *chunkovi* odgovaraju kontekstu modela i ne prekorače maksimalnu duljinu.

3.4 Izrada vektorskih reprezentacija i pohrana u vektorsku bazu

Kako bi RAG sustav mogao učinkovito pretraživati velike količine tekstualnih podataka, tekst se pretvara u numerički oblik koji omogućuje usporedbu značenja. U tu svrhu koriste se vektorske reprezentacije teksta, odnosno *embeddings*. Embeddings predstavljaju numeričke vektore koji kodiraju semantičko značenje teksta, pri čemu tekstualni segmenti sličnog sadržaja imaju slične vektorske reprezentacije. Nakon što se dokumenti razdijele na manje dijelove (*chunks*), svaki od njih se pretvara u *embedding* i pohranjuje u vektorsku bazu podataka.

U ovom radu korištena je *sentence-transformers*, *open-source* biblioteka *Hugging Face* transformera, koja nudi niz unaprijed treniranih modela za generiranje *embeddinga*. Korišten je model *all-mpnet-base-v2*, koji se pokazao vrlo učinkovitim za semantičko pretraživanje.

Kako bi se intuitivno prikazao način rada *embeddinga*, razmotren je jednostavan primjer s tri kratke rečenice: *"I like piano"*, *"I like guitar"* i *"Entschuldigung, wo ist der Kellner?"*. Prve dvije rečenice su semantički slične jer izražavaju sličnu preferenciju, dok je treća rečenica na drugom jeziku i potpuno drugačijeg značenja. Nakon generiranja *embeddinga*, sličnost između vektora mjeri se skalarnim umnošcima, koji su pokazali visoku sličnost između prve dvije rečenice, dok je sličnost s trećom rečenicom gotovo zanemariva. Ovaj primjer ilustrira glavnu prednost *embeddinga*; semantičku bliskost teksta mogu kvantitativno izmjeriti, neovisno o točnom izboru riječi ili jeziku.

Nakon generiranja *embeddinga*, oni se pohranjuju u vektorske baze podataka (*eng*. *vectorstores*), koje su optimizirane za brzo pronalaženje najsličnijih vektora. *LangChain* podržava integraciju s velikim brojem vektorskih baza, a u ovom radu korištena je *Chroma*, lagana i jednostavna vektorska baza koja se može koristiti u memoriji ili s trajnom pohranom na disku. Prilikom upita, tekst pitanja se također pretvara u *embedding*, nakon čega se u bazi pronalaze oni dokumenti čiji su vektori najsličniji vektoru upita.

3.5 Dohvat relevantnih dokumenata

Nakon što su dokumenti pohranjeni u vektorskoj bazi, sljedeći korak u RAG sustavu jest *retrieval*, odnosno dohvat onih tekstualnih isječaka (*chunks*) koji su najrelevantniji za korisnički upit. U nastavku su opisane neke od testiranih metoda koje nudi *LangChain*.

Maximum Marginal Relevance (MMR)

Najjednostavniji pristup dohvatu relevantnih dokumenata temelji se na pronalaženju najsličnijih dokumenata prema semantičkoj sličnosti. Međutim, takav pristup često dovodi do rezultata koji su međusobno vrlo slični ili čak duplicirani. Kako bi se izbjegla redundantnost i dobio informativniji skup dokumenata, koristi se metoda *Maximum Marginal Relevance* (MMR). MMR kombinira dva kriterija; relevantnost dokumenata u odnosu na upit i raznolikost dokumenata međusobno. Iz vektorske baze dohvaća se *k* dokumenata koji su semantički najbliži upitu. Iz tog skupa odabire se konačnih *n* dokumenata tako da se maksimizira njihova raznolikost. Korištenjem MMR metode dobiveni su raznovrsniji dokumenti, bez dupliciranog sadržaja, što rezultira bogatijim kontekstom za generiranje odgovora.

SelfQuery (dohvat uz pomoć LLM-a)

U nekim slučajevima korisnički upit sadrži implicitne informacije o metapodacima dokumenata (npr. o izvoru ili vremenskom kontekstu). Kako bi se takve informacije automatski iskoristile, koristi se *SelfQueryRetriever*, koji u proces dohvata dokumenata uključuje jezični model. Pomoću njega izdvaja semantički dio upita za vektorsko pretraživanje i generira filtre nad metapodacima (npr. izvor dokumenta, broj stranice). *SelfQueryRetriever* omogućuje precizniji dohvat dokumenata koristeći i sadržaj i strukturu podataka, bez potrebe za ručnim definiranjem filtera.

Kontekstualna kompresija dokumenata

Dohvaćeni relevantni dokumenti često sadrže više informacija nego što je potrebno za konkretno pitanje. Kako bi se smanjila količina teksta i povećala fokusiranost odgovora, koristi se kompresija konteksta. Ova metoda svaki dohvaćeni dokument kompresira kroz kompresijski LLM; iz dokumenta se izdvajaju samo najrelevantniji dijelovi i skraćeni kontekst prosljeđuje se glavnom LLM-u za generiranje odgovora. Kompresija povećava preciznost odgovora i smanjuje količinu nepotrebnog konteksta, uz cijenu dodatnih poziva LLM-u.

3.6 Generiranje odgovora

Metode poput *RetrievalQA, Map-reduce* i *Refine* pripadaju generativnoj fazi RAG sustava, jer uključuju LLM koji proizvodi konačan odgovor.

RetrievalQA

RetrievalQA lanac kombinira dohvat relevantnih dokumenata i generiranje odgovora u jednom koraku. Nakon dohvaćanja relevantnih dokumenata, oni se prosljeđuju LLM-u koji generira odgovor.

Map-reduce

Map-reduce metoda svaki dokument zasebno prosljeđuje LLM-u (*map faza*) i dobivene odgovore spaja u jedan konačni odgovor (*reduce faza*). Prednost ovog pristupa je paralelizacija, ali nedostatak je gubitak konteksta kada je informacija raspršena kroz više dokumenata. Testiranjem metoda pokazalo se da ovaj pristup može rezultirati slabijim odgovorima, osobito kada mnogi dokumenti ne sadrže relevantne informacije.

Refine

Refine metoda koristi sekvencijalno generiranje. Prvi LLM generira inicijalni odgovor, a svaki sljedeći LLM nadograđuje postojeći odgovor i stvara novi; informacije se prenose i poboljšavaju kroz cijeli lanac. *Refine* metoda daje potpunije odgovore jer omogućuje akumulaciju znanja iz više dokumenata.

3.7 Interaktivni RAG obogaćen s poviješću razgovora

Kako bi *chatbot* mogao voditi kontekstualno ispravan dijalog s korisnikom, nužno je omogućiti pamćenje prethodnih poruka unutar jedne sesije. U tu svrhu korištena je kombinacija *ConversationBufferMemory* i *ConversationalRetrievalChain*, čime se ostvaruje interaktivna komunikacija u kojoj se raniji upiti i odgovori koriste kao kontekst za

generiranje novih odgovora. Prilikom svakog novog upita, lanac najprije uzima u obzir povijest razgovora kako bi pravilno interpretirao značenje pitanja, posebno u slučajevima kada se korisnik referira na prethodne odgovore. Nakon toga se, na temelju obogaćenog upita, dohvaćaju relevantni dokumenti iz vektorske baze, koji se zajedno s kontekstom razgovora prosljeđuju jezičnom modelu.

Za pohranu povijesti razgovora korišten je *Redis*, brza *in-memory* baza podataka, koja je u ovom sustavu pokrenuta unutar *Docker* kontejnera. Ovakav pristup omogućuje izolirano i pouzdano izvršavanje servisa, te jednostavno pokretanje sustava. Povijest razgovora organizirana je po korisničkim sesijama, pri čemu je svaka sesija identificirana jedinstvenim *session_id*. Time se osigurava da svaki korisnik ima vlastiti, neovisan kontekst razgovora, bez miješanja podataka između različitih sesija.

3.8 Utjecaj dizajna prompta na ponašanje jezičnog modela

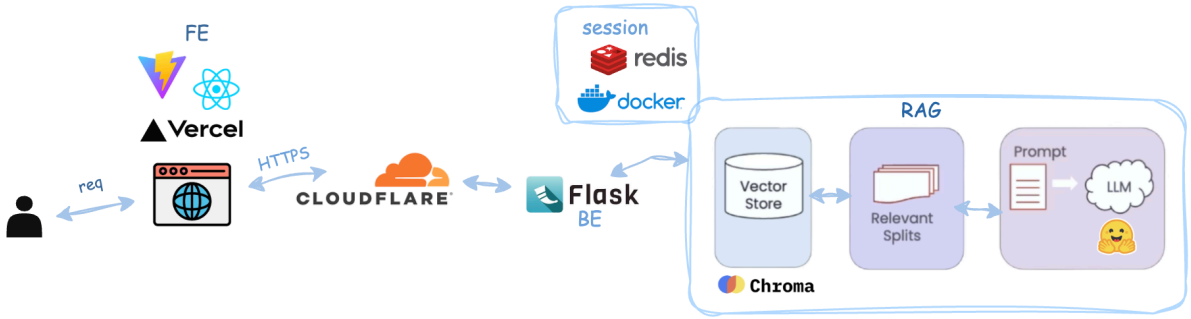
Tijekom razvoja sustava posebna je pažnja posvećena oblikovanju *prompta*, budući da način na koji su upute formulirane značajno utječe na ponašanje jezičnog modela i kvalitetu generiranih odgovora. U početnoj fazi sustav je koristio samo pitanje korisnika, bez jasno definiranih ograničenja u pogledu izvora informacija. U tom slučaju model je često odgovarao i na pitanja za koja odgovor nije bio prisutan u dohvaćenim dokumentima iz baze znanja, oslanjajući se na vlastito opće znanje. Takvo ponašanje rezultiralo je pojavom tzv. halucinacija; uvjerljivih, ali netočnih odgovora.

Kako bi se smanjila mogućnost generiranja odgovora koji nisu potkrijepljeni znanjem dokumenata RAG sustava, uveden je stroži *prompt*, kojim je modelu izričito zabranjeno odgovaranje na pitanja ako odgovor nije prisutan u dohvaćenom kontekstu. Iako je ovakav pristup uspješno eliminirao halucinacije, pokazao se previše restriktivnim. Model je postao izrazito nesiguran te je u velikom broju slučajeva odgovarao s “ne znam”, čak i onda kada su relevantne informacije bile prisutne u dokumentima, ali nisu bile formulirane na identičan način kao u pitanju.

Konačno, razvijen je uravnoteženiji prompt koji modelu daje jasnu, ali ne pretjerano restriktivnu uputu: odgovor treba temeljiti isključivo na kontekstu, a u slučaju da informacija nije pronađena u dokumentima, model treba navesti da ne zna odgovor. Ovaj pristup omogućio je postizanje zadovoljavajuće ravnoteže između točnosti i korisnosti odgovora.

4. Arhitektura sustava

Razvijeni sustav prikazan na slici 1 temelji se na klijent-poslužitelj arhitekturi. Frontend *Vite* aplikacija implementirana je u *Reactu* i služi kao korisničko sučelje za interakciju s *chatbotom*, dok backend aplikacija, temeljena na *Flask* okviru, implementira RAG sustav i upravlja obradom korisničkih upita. Memorija razgovora realizirana je korištenjem *Redis* baze podataka pokrenute unutar *Docker* kontejnera, čime je omogućeno učinkovito upravljanje korisničkim sesijama. Backend aplikacija izložena je putem *Cloudflared* tunela, dok je frontend aplikacija *deployana* na platformi *Vercel*.



Slika 1 Arhitektura sustava

4.1 Frontend aplikcacija

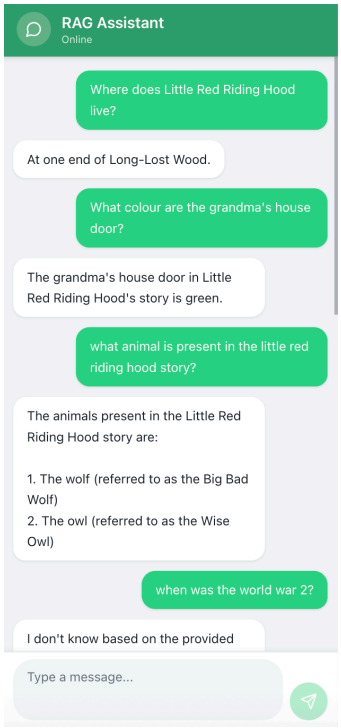
Frontend aplikacija razvijena je korištenjem *React* biblioteke, koja omogućuje izgradnju modularnih i responzivnih korisničkih sučelja temeljenih na komponentama. Aplikacija predstavlja glavno sučelje između korisnika i sustava te omogućuje postavljanje pitanja *chatbotu*. Svaki korisnički upit šalje se backend aplikaciji putem HTTP POST zahtjeva.

Za razvoj aplikacije korišten je *TypeScript*, tipizirani *JavaScript*, koji povećava pouzdanost koda, te smanjuje mogućnost grešaka tijekom razvoja. Stilizacija korisničkog sučelja ostvarena je pomoću *Tailwind CSS* okvira, koji omogućuje brzo oblikovanje sučelja korištenjem unaprijed definiranih *utility* klasa.

Projekt je inicijaliziran i razvijan uz pomoć alata za izgradnju frontend aplikacija *Vite*, koji omogućuje brzo pokretanje razvojnog okruženja i optimizirani *build* proces.

Frontend aplikacija je *deployana* na platformi *Vercel*, koja pruža automatizirano upravljanje *build* i *deploy* procesima, visoku dostupnost aplikacije te jednostavno povezivanje s vanjskim API servisima.

Korisničko sučelje prikazano je na slici 2.



Slika 2 Korisničko sučelje

4.2 Backend aplikacija

Backend aplikacija implementirana je pomoću *Flask* web okvira. *Flask* je lagani i fleksibilni *Python* web okvir koji omogućuje brzu izradu REST API-ja i jednostavnu integraciju s drugim bibliotekama. Glavne zadaće backend aplikacije uključuju obradu korisničkih upita, dohvat relevantnih dokumenata iz vektorske baze, generiranje odgovora pomoću RAG arhitekture i upravljanje memorijom razgovora.

Prilikom pokretanja aplikacije, RAG *pipeline* se inicijalizira jednom, čime se izbjegava ponovno učitavanje dokumenata i njihova segmentacija za svaki pojedini upit. Sustav najprije učitava PDF dokumente iz baze znanja, zatim ih dijeli na manje dijelove, te generira vektorske reprezentacije koristeći model *all-mpnet-base-v2*. Za generiranje *embeddinga* i obradu prirodnog jezika korišteni su besplatni modeli dostupni putem *Hugging Face* ekosustava. Za pohranu *embeddinga* koristi se *Chroma* vektorska baza, koja se trajno pohranjuje na disk, što omogućuje ponovno korištenje već izgrađenog indeksa prilikom ponovnog pokretanja aplikacije i značajno smanjuje vrijeme inicijalizacije sustava.

Kako bi se omogućio interaktivni chat s očuvanim kontekstom, sustav koristi *Redis* bazu podataka, pokrenutu unutar *Docker* kontejnera. *Redis* služi kao brza *in-memory* pohrana povijesti razgovora, pri čemu je svaka sesija povezana s jedinstvenim *session_id*.

Za obradu korisničkih upita koristi se *ConversationalRetrievalChain*, koji kombinira povijest razgovora, semantičko pretraživanje vektorske baze i generiranje odgovora pomoću jezičnog modela. Na taj način svaki novi upit interpretira se u kontekstu prethodnih poruka, što omogućuje prirodniji dijalog s korisnikom.

Budući da je frontend aplikacija *deployana* na *Vercelu*, a backend se izvršava lokalno, korišten je *Cloudflared tunnel* za sigurno izlaganje backend API-ja prema internetu. Time je omogućena komunikacija između frontend i backend dijela sustava.