Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Távközlési és Mesterséges Intelligencia Tanszék

Czotter Benedek

Klaszterezési stratégiák alkalmazása RAG rendszerek hatékonyságának növelésére

Tudományos Diákköri Konferencia Dolgozat

Konzulens

Dr. Szűcs Gábor

BUDAPEST, 2025

Tartalomjegyzék

Összefoglaló

A Retrieval-Augmented Generation (RAG) modellek az utóbbi években kulcsszerepet kaptak a nagynyelvi modellek (LLM-ek) tudásbővítésében és naprakész információkkal való ellátásában. E rendszerek lényege, hogy a nyelvi modell válaszadási folyamata előtt releváns dokumentumokat vagy dokumentumrészleteket („chunkokat”) keresnek ki egy nagyméretű tudásbázisból embedding-alapú hasonlóságmérés segítségével. A jelenlegi gyakorlatban a legtöbb RAG-megoldás a teljes embedding-halmazt közvetlenül használja a hasonlósági kereséshez, ami azonban pontossági, redundancia- és futásidőbeli korlátokat eredményezhet. A dolgozat célja annak vizsgálata, hogy klaszterezési technikák integrálásával miként növelhető a RAG-rendszerek hatékonysága, pontossága és robusztussága; hiszen hogyha a dokumentumok embeddingjeit először klaszterekbe rendezzük, majd a felhasználói lekérdezés embeddingjét először klaszterszinten vetjük össze velük, a keresés jelentősen gyorsabbá és relevánsabbá válhat. A dolgozat bemutatja a különböző klaszterezési algoritmusok teljesítményét és alkalmazhatóságát eltérő embedding-reprezentációkon, valamint azt, hogy hogyan lehet lehetővé tenni új dokumentumok folyamatos integrálását a tudásbázisba a teljes újraklaszterezés költsége nélkül. A kutatás eredménye egy klaszterezésen alapuló, adaptív retrieval-pipeline kialakítása, amely a hagyományos RAG-architektúrákhoz képest magasabb hatékonyságot, jobb pontosságot és fokozott robusztusságot biztosít. A kutatás eredményei nemcsak elméleti szempontból járulnak hozzá a RAG-rendszerek fejlődéséhez, hanem gyakorlati alkalmazásokban is közvetlenül hasznosíthatók lesznek.

Abstract

Retrieval-Augmented Generation (RAG) models have become key components in recent years for enhancing the knowledge base of large language models (LLMs) and providing them with up-to-date information. The core idea behind these systems is that, before the language model generates a response, they retrieve relevant documents or document segments (“chunks”) from a large knowledge base using embedding-based similarity search. In current practice, most RAG solutions directly use the entire embedding set for similarity search, which can lead to limitations in accuracy, redundancy, and runtime performance. The aim of this thesis is to investigate how the integration of clustering techniques can improve the efficiency, accuracy, and robustness of RAG systems. By first organizing document embeddings into clusters and then comparing the query embedding at the cluster level, the retrieval process can become significantly faster and more relevant. The thesis explores the performance and applicability of various clustering algorithms on different embedding representations and examines how new documents can be continuously integrated into the knowledge base without the cost of full re-clustering. The outcome of the research is the design of a clustering-based adaptive retrieval pipeline that provides higher efficiency, better accuracy, and increased robustness compared to traditional RAG architectures. The results of this research contribute not only to the theoretical advancement of RAG systems but are also directly applicable in practical implementations.

1 Bevezetés

1.1 A kutatás célja

A mesterséges intelligencia fejlődésének egyik legmeghatározóbb területe az utóbbi években a természetes nyelvfeldolgozás (Natural Language Processing, NLP) volt. A nagyméretű nyelvi modellek (Large Language Models, LLM-ek) képesek emberi szintű szövegértésre és generálásra, azonban működésük korlátozott, ha a feladatukhoz szükséges információ nem szerepel a modell paramétereiben. A Retrieval-Augmented Generation (RAG) [11] megközelítés ezt a problémát kezeli azáltal, hogy a szövegalkotás előtt a modell külső dokumentumokból keres releváns kontextust, és azt integrálja a válasz generálásába.

Jelen kutatás célja a retrieval-lépés hatékonyságának növelése. Nagyméretű dokumentumkorpuszok (pl. Wikipédia, tudományos cikkgyűjtemények vagy QA-adathalmazok) esetén a releváns információ visszakeresése rendkívül számításigényes feladat, hiszen minden lekérdezéshez több tízezer, sőt akár milliónyi szövegrész (chunk) vektoros reprezentációját kell összehasonlítani. Ez a folyamat a RAG-rendszerek egyik legszűkebb keresztmetszete, amely korlátozza azok valós idejű vagy online alkalmazását.

A dolgozat célja ezért a RAG-rendszerek hatékonyabbá tétele online klaszterezési módszerek integrálásával, különösen a centroid-alapú keresés alkalmazásán keresztül. A dolgozatban bemutatásra kerül, hogyan csökkenthető a visszakeresési idő anélkül, hogy a pontosság jelentősen romlana, és milyen kompromisszum érhető el a gyorsaság és információvesztés között. A munka kísérleti platformja a SQuAD (Stanford Question Answering Dataset) [12] adathalmaz, amely jól mérhető alapot biztosít a retrieval-modulok összehasonlításához.

1.2 RAG-rendszerek szerepe a modern NLP-ben

A modern NLP-ben a RAG-rendszerek kulcsszerepet töltenek be az adatvezérelt tudás-hozzáférés, a forráshivatkozással alátámasztott szövegalkotás, valamint a megbízhatóbb generatív mesterséges intelligencia kialakításában. Széles körben alkalmazzák őket keresés-alapú chatbotokban, tudásmenedzsment rendszerekben, valamint olyan feladatokban, ahol a pontosság és az aktualitás kiemelt fontosságú. A RAG tehát nem csupán technológiai fejlesztés, hanem egyben szemléletváltás is, mivel az intelligens nyelvi rendszerek egyre inkább a „tudásalapú”, és nem pusztán a „nyelvre tanított” modellek felé fejlődnek.

Míg korábban a modellek statikus tudásra támaszkodtak, a RAG megközelítés lehetővé teszi, hogy a rendszerek folyamatosan bővítsék és frissítsék tudásukat anélkül, hogy újratanítást igényelnének. Ez különösen fontos a gyorsan változó információs környezetekben, például a tudományos vagy üzleti szférában, ahol a relevancia és az aktualitás kulcstényezők. A modern NLP kutatásában a RAG így

nemcsak egy hatékony technikai megoldás, hanem a mesterséges intelligencia átláthatóbbá és megbízhatóbbá tételének egyik legfontosabb irányvonala.

1.3 Hatékonysági kérdések nagy dokumentumkorpuszok esetén

A RAG-rendszerek egyik legkritikusabb gyakorlati problémája a retrieval-lépés skálázhatósága. Nagy dokumentumkorpuszok esetén, ahol több millió szövegrész (chunk) kerül beágyazásba (embedding formába), a keresés időigénye drámaian megnő. Egyetlen lekérdezés esetében akár több százezer koszinusz-távolság számítás is történhet, ami valós idejű alkalmazásokban (pl. kérdés–válasz rendszerekben) nem megengedhető [20].

A keresés gyorsítása több irányból közelíthető meg. Az egyik megoldás a vektorindexelés, például a FAISS (Facebook AI Similarity Search) [8] vagy a HNSW (Hierarchical Navigable Small World graphs) [27] algoritmusok alkalmazásával, amelyek különböző approximációs módszerekkel (pl. kvantizáció, gráf-alapú közelítés) csökkentik a szükséges összehasonlítások számát. A FAISS indexek jellemzően statikus szerkezetűek, és bár több típusuk támogatja új vektorok hozzáadását, ezek a módosítások nem feltétlenül hatékonyak nagyobb adatmennyiség vagy gyakori frissítés esetén. A HNSW ezzel szemben egy dinamikus gráfstruktúra, amely hatékonyan kezeli az új pontok hozzáadását, ugyanakkor a törlés és a tömeges frissítés itt is korlátozottan megoldható.

A közelmúltban megjelent megoldások, mint a Databricks Vector Search [3], már state-of-the-art megközelítést képviselnek, hiszen ezek a rendszerek natívan támogatják a valós idejű, konzisztens és skálázható indexfrissítést, valamint integráltan működnek elosztott adatplatformokkal, ami lehetővé teszi a vektoralapú keresés és az adatfeldolgozás egyesítését vállalati környezetben.

Munkám során arra keresek megoldást, hogyan adnak a klaszterezésen alapuló megközelítések lehetőséget a keresési tér intelligens felosztására. Az adathalmaz vektorjaiból képzett klaszterek centroidjai egyfajta reprezentatív középpontot képeznek, amelyek alapján előszűrés végezhető, így a keresés először csak a centroidok között történik, majd a legközelebbi klaszter(ek)en belül folytatódik a részletesebb keresés. Ez a kétlépcsős folyamat jelentősen csökkenti az összehasonlítások számát, miközben a releváns találatok többsége továbbra is megtalálható marad.

A hatékonyság kulcsa a pontosság és sebesség közötti kompromisszum megtalálása. Ha túl kevés klasztert használunk, a keresés továbbra is lassú marad, ha viszont túl sokat, nő a hibás kizárások aránya. A jelen kutatás egyik fő célja ezért a paramétertér (klaszterszám, top-k klaszterek száma, top-n chunk visszaadása) grid search alapú optimalizálása, különböző embedding-méretek mellett.

További kérdést vet fel az online klaszterezés lehetősége. Míg az klasszikus KMeans vagy MiniBatchKMeans [18] csak teljes újratanítással tud reagálni az új adatokra, az online klaszterezési megoldások adaptív modelljei folyamatosan képesek frissíteni a centroidokat a beérkező minták alapján [2][7]. Ez a megközelítés elméletileg lehetővé teszi a RAG-rendszerek folyamatosan tanuló retriever

komponensének megvalósítását, olyat, amely idővel egyre jobban illeszkedik a beérkező kérdések és dokumentumok eloszlásához.

A dolgozat későbbi fejezeteiben bemutatásra kerül, hogy az ilyen online klaszterezési módszerek hogyan illeszthetők a RAG-rendszerek retrieval folyamatába, és milyen teljesítményt nyújtanak a hagyományos, minden lekérdezéskor a teljes vektortérben végzett kereséssel szemben. A cél egy olyan robusztus, adaptív és gyors retrieval-architektúra megalkotása, amely a valós idejű RAG-rendszerek egyik kulcskomponensévé válhat.

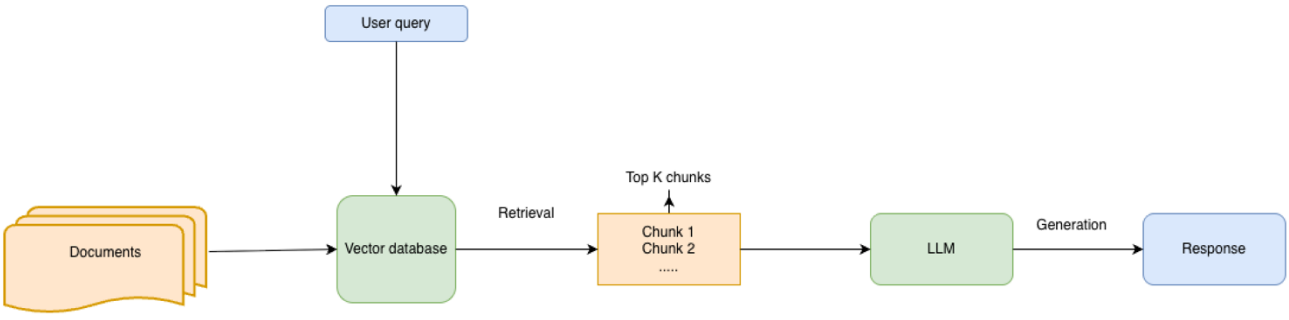
2 Elméleti háttér

A modern természetes nyelvfeldolgozás és információkinyerés területén az utóbbi évek egyik jelentős előrelépése a Retrieval-Augmented Generation [11] modell alkalmazása. A RAG lényege, hogy a generatív modellek nem csupán a tanult mintákból próbálnak választ adni, hanem aktívan hivatkozásokat és információkat is keresnek a rendelkezésre álló dokumentumtárakban. Ez a megközelítés különösen hasznos olyan helyzetekben, ahol a pontos és friss információ kritikus, mivel a generált válaszok pontossága és relevanciája jelentősen növelhető a visszakeresett dokumentumok felhasználásával. A következő alfejezetekben részletesen bemutatom a RAG architektúráját, a dokumentum-visszakeresés elméleti alapjait, az embeddingek és vektorterek szerepét, valamint a klaszterezés lehetséges alkalmazását az információkinyerésben.

2.1 A Retrieval-Augmented Generation elmélete

A RAG modell a mesterséges intelligencia két meghatározó komponensét, a dokumentum-visszakeresést (retrieval) és a szöveggenerálást (generation), egyesíti egy egységes keretrendszerben. A modell elsődleges célja, hogy a felhasználói lekérdezésekre adott válaszokat ne kizárólag a nyelvi modell előzetesen betanított paramétereiből származó tudás alapján állítsa elő, hanem azokat kiegészítse külső forrásokból, például dokumentumtárakból vagy tudásbázisokból származó, aktuális és releváns információkkal.

A RAG működése egy kétfázisú folyamatként értelmezhető: az első szakaszban a rendszer a bemeneti lekérdezés alapján releváns dokumentumokat azonosít a rendelkezésre álló adatforrásokban, majd a második szakaszban ezen visszakeresett szövegeket kontextuális inputként felhasználva generál egy koherens, tartalmilag megalapozott választ. Ennek eredményeként a modell képes egyesíteni a retrieval-alapú tudásbővítés előnyeit a generatív nyelvi modellek rugalmas válaszképességével.



1. ábra - RAG rendszerek általános felépítése [28]

A Retrieval-Augmented Generation rendszerek [11], amint az az 1. ábrán is szemléltetésre kerül, három alapvető komponensből épülnek fel. Az első elem a dokumentumtár, amely a modell

számára releváns szöveges források gyűjteményét tartalmazza. Ezen források körébe tartozhatnak tudományos cikkek, termékdokumentációk, adatbázis-bejegyzések, valamint különféle webes tartalmak is.

A második fő komponens a visszakereső modul (retrieval module), amelynek feladata a felhasználói lekérdezés alapján a legrelevánsabb dokumentumok azonosítása. A folyamat több, egymásra épülő lépésből áll. Elsőként a dokumentumokat és a lekérdezést tokenizálják, normalizálják és chunkolják, azaz kisebb egységekre bontják. Ezt követően a szövegeket numerikus embeddingekké alakítják, amelyek lehetővé teszik a gépi hasonlóságmérést. A lekérdezés embeddingjét a dokumentumtár embeddingjeivel összehasonlítva meghatározzák a legrelevánsabb dokumentumokat, gyakran a koszinusz hasonlóság vagy az euklideszi távolság mérőszámai alapján. Végül a kiválasztott dokumentumokat relevancia szerint rangsorolják, biztosítva, hogy a generatív modell a lekérdezés szempontjából legértékesebb információkhoz férjen hozzá.

A harmadik komponens a generatív modul (generation module), amely a visszakeresett dokumentumokból származó kontextuális információ felhasználásával állítja elő a végső választ. Ezáltal a modell nem csupán a saját neurális súlyaiban kódolt, előzetesen tanult mintázatokra támaszkodik, hanem az aktuálisan elérhető és releváns tudáselemeket is integrálja a válaszgenerálás folyamatába. Ennek eredményeként a RAG rendszerek válaszai nagyobb pontosságot, kontextuális koherenciát és információs megbízhatóságot mutatnak a hagyományos, kizárólag generatív alapú nyelvi modellekhez képest.

Jelen kutatás célja a RAG rendszerek visszakereső moduljának fejlesztése, különös tekintettel a klaszterezési technikák alkalmazására a dokumentum-visszakeresés pontosságának és hatékonyságának javítása érdekében. A kutatás további célja, hogy feltárja, miként használhatóak fel nem felügyelt tanulási módszerek, különösen a dokumentum-embeddingek klaszterezése, a releváns információk strukturáltabb és gyorsabb előhívásának támogatására.

2.2 Embeddingek és vektorterek

2.2.1 Sentence-transformerek működése és hatásuk a reprezentációra

Az embeddingek a természetes nyelvi szövegek numerikus, vektoralapú reprezentációi, amelyek lehetővé teszik a gépi rendszerek számára a szövegek közötti hasonlóságok kvantitatív mérését és a hatékony vektoralapú keresést. Ezen ábrázolások nem csupán a szavak szerinti egyezést képesek megragadni, hanem a szövegek jelentésbeli és kontextuális kapcsolatait is képesek modellezni. Az embeddingek így olyan dimenziókat hoznak létre, amelyek a nyelvi mintázatokat, szemantikai összefüggéseket és a kontextuális információkat kódolják, lehetővé téve a gépek számára, hogy a jelentésbeli hasonlóságokat is figyelembe véve rangsorolják vagy csoportosítsák a dokumentumokat.

A sentence-transformerek [14] kifejezetten mondat- és dokumentumszintű embeddingek előállítására szolgálnak. Ezek a modellek a bemeneti szöveget tokenizálják, majd a tokeneket belső reprezentációkká alakítják, amelyeket pooling műveletek segítségével egy fix hosszúságú vektorrá egyesítenek. Ennek eredményeként a jelentésükben hasonló mondatok embeddingjei a vektortérben közel helyezkednek el egymáshoz, ami elősegíti a dokumentum-visszakeresés, a klaszterezés és egyéb jelentésalapú elemzések pontosságát, valamint minimalizálja a releváns információk kihagyásának kockázatát. A sentence-transformerek így közvetlenül támogatják a gazdag, szemantikai információt hordozó reprezentációk létrehozását a természetes nyelvi adatok feldolgozásához.

2.2.2 Távolságmértékek

Az embedding vektorok közötti hasonlóság vagy távolság mérésére több módszer is létezik. A koszinusz hasonlóság (cosine similarity) a két vektor közötti szöget méri, ami különösen jól működik a nagyméretű, normalizált embeddingeknél. Az euklideszi távolság (euclidean distance) a vektorok geometriai távolságát adja meg, míg a Manhattan távolság (Manhattan distance) [26] az egyes koordináták abszolút különbségeinek összegét használja. A skaláris szorzat (dot product) lineáris kapcsolatot mér, amelyet gyakran neurális hálózatokban alkalmaznak.

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}}$$

1. egyenlet - Koszinusz hasonlóság számítási módja [25]

A kutatás során az embedding vektorok összehasonlításához a koszinusz-hasonlóságot alkalmaztam, mivel magas dimenziós térben a vektorok irányának hasonlósága informatívabb, mint azok távolsága. Az 1. egyenlet tehát a koszinusz hasonlóság számításának módját adja meg, ahol A és B két, n-dimenziós vektort jelöl, ahol n az embedding tér dimenziószáma. Az egyenletben A_i és B_i pedig az A vektor és a B vektor i-edik komponensét jelöli. A teljesítmény növelése érdekében minden vektort L2-normalizáltam, vagyis úgy skáláztam, hogy a hosszúságuk 1 legyen. A koszinusz hasonlóság a vektorok közötti szöget méri, nem a nagyságukat; ha a vektorok különböző hosszúságúak, a nagyobb vektorok torzíthatják az eredményt. L2-normalizálás után a koszinusz hasonlóság két embedding között egyszerűen a skaláris szorzatra redukálódik, így a számítás is egyszerűsödik.

2.2.3 Vektorindexelési módszerek áttekintése

A nagy mennyiségű embedding hatékony és gyors keresése alapvető kihívást jelent a modern információ-visszakeresési rendszerekben. A hagyományos, ún. brute-force keresés során minden dokumentum embeddingjét összehasonlítják a lekérdezés embeddingjével, ami ugyan garantálja a

pontos találatokat, de nagyméretű adatbázisok esetén rendkívül idő- és erőforrás-igényes megoldást jelent.

A gyakorlatban ezért gyakran alkalmaznak közelítő legközelebbi szomszéd (Approximate Nearest Neighbor, ANN [1]) algoritmusokat, amelyek jelentősen felgyorsítják a keresési folyamatot, miközben a találatok pontossága közelítő módon megőrződik. Ezek közé tartozik többek között a state-of-the-art FAISS [8] és a HNSW [27], melyek hatékony adattárolási és keresési struktúrákat biztosítanak a nagyméretű vektorterekben.

2.3 Klaszterezés az információkinyerésben

2.3.1 Klaszterezés célja

A klaszterezés alapvető célja, hogy a dokumentumokat vagy azok embedding-reprezentációit tematikus, szemantikai szempontból koherens csoportokba rendezzük. A klaszterezés során olyan csoportokat alakítunk ki, amelyekben a belső hasonlóság maximális, míg a különböző klaszterek közötti eltérés jelentős. Ennek eredményeként a tartalmilag hasonló dokumentumok a vektortérben közel helyezkednek el egymáshoz, míg a különböző témájú csoportok jól elkülönülnek egymástól.

Ez a struktúra több szempontból is előnyös a dokumentum-visszakeresési rendszerek számára. Egyrészt lehetővé teszi, hogy a keresési folyamat a releváns klaszterekre koncentrálódjon, így jelentősen csökkentve a szükséges számítási időt és növelve a keresés hatékonyságát. Másrészt támogatja a relevancia szerinti rangsorolást, mivel a klasztereken belüli dokumentumok tartalmilag egységesebbek. Továbbá a klaszterezés hozzájárul a redundancia minimalizálásához is, mivel a hasonló információkat tartalmazó dokumentumok egy csoportba kerülnek, így a felhasználó számára a bemutatott találatok változatosabb és informatívabb képet adnak a keresett témáról.

2.3.2 Offline és online klaszterezés

A klaszterezési folyamatok két alapvető módon valósíthatóak meg: offline és online módszerekkel [22]. Az offline klaszterezés esetén a klaszterek a teljes rendelkezésre álló adatállomány alapján előre létrejönnek. Ez a megközelítés hatékony, ha az adatok viszonylag statikusak, mivel lehetővé teszi a számításigényes algoritmusok alkalmazását és a klaszterek alapos optimalizálását. Hátránya azonban, hogy nem képes gyorsan reagálni az új, dinamikusan érkező adatokra, így a klaszterek idővel elavulhatnak, és a keresési relevancia csökkenhet.

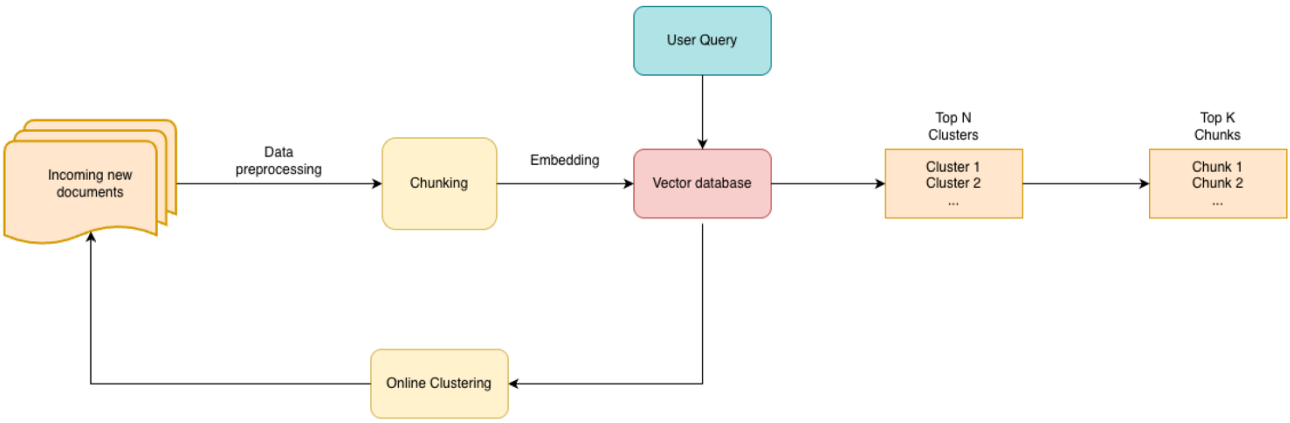
Ezzel szemben az online klaszterezés folyamatosan alkalmazkodik az új adatokhoz, lehetővé téve a klaszterek dinamikus frissítését valós időben. Ez különösen fontos olyan környezetekben, ahol az adatforrások gyorsan változnak, például hírek, közösségi média bejegyzések, chat-adatok vagy valós idejű dokumentumáramlások esetén. Az online klaszterezés lehetővé teszi, hogy a vektoralapú

embeddingeken alapuló reprezentációk folyamatosan tükrözzék az aktuális szemantikai szerkezetet, így a releváns dokumentumok könnyebben és gyorsabban azonosíthatók.

A Retrieval-Augmented Generation rendszerekben az online klaszterezés különösen hasznos lehet azokban az esetekben, ahol a dokumentumállomány folyamatosan bővül vagy változik. Dinamikus adatforrások, például hírek, közösségi média bejegyzések vagy valós idejű chat-adatok esetén a klaszterek hagyományos, előre definiált struktúrái gyorsan elavulhatnak, és nem tükrözik megfelelően a friss tartalmak jelentését. Az online klaszterezés lehetővé teszi, hogy a klaszterek folyamatosan frissüljenek az új dokumentumokkal, így a hasonló tartalmak mindig egy csoportban jelenjenek meg, és a keresési folyamat a releváns klaszterekre koncentrálódhasson. Ennek eredményeként a rendszerek képesek valós időben alkalmazkodni a változó dokumentumhalmazokhoz, fenntartva a tartalmilag koherens és jól strukturált klasztereket.

3 Klaszterezési módszerek implementációja és integrálása a retrieval folyamatba

A retrieval-alapú rendszerek, különösen a RAG architektúrák esetében, az egyik legfontosabb kérdés, hogy hogyan lehet hatékonyan és pontosan visszakeresni a releváns dokumentumokat vagy szövegrészeket a tudásbázisból. A hagyományos megközelítés szerint minden lekérdezés esetén a teljes adatbázist (vagy annak embeddingjeit) kell átvizsgálni, ami nagy adatméretek esetén számottevően lelassítja a válaszidőt. Ennek a problémának a kezelésére vezettem be a klaszterezés-alapú retrieval módszert, amelynek célja a keresési tér redukálása a reprezentatív centroidok segítségével. A módszer az offline és az online tanulási folyamatokban is kulcsszerepet játszik, különösen olyan környezetben, ahol az adatok folyamatosan frissülnek. A továbbiakban egy olyan módszert mutatok be, mellyel nagy mértékben felgyorsítható az információ visszakeresés sebessége a pontosság minimális csökkenése mellett, illetve bemutatásra kerül az is, hogyan lehet ezt a klaszterezés-alapú retrieval módszert dinamikusan változó dokumentum halmazok esetén használni, biztosítva ezzel, hogy a rendszer mindig a legrelevánsabb és legaktuálisabb információt szolgáltassa a lekérdezésekhez.



2. ábra - Az általam készített retrieval pipeline felépítése

A 2. ábrán látható architektúra dinamikusan változó dokumentum halmazok kezelésére épül. A folyamatosan érkező új dokumentumok áthaladnak a korábban említett adatelőkészítési lépéseken, majd elkészülnek a feldolgozott chunkokból a szövegbeágyazások. Ezt az online klaszterező algoritmus azonnal egy már meglévő vagy új klaszterhez rendeli, majd ezen klaszterek ismeretében indul a RAG alapját képező, hasonlóság alapú információ visszatérítés, melynek végeredményében a felhasználói bemenethez leginkább „közel” elhelyezkedő dokumentum részletek kerülnek meghatározásra. A továbbiakban részletesen bemutatom a centroid alapú és az online klaszterező megközelítéseket.

3.1 Centroid-alapú retrieval logika

A centroid-alapú retrieval egyik legfontosabb lépése a klaszterezés utáni centroidokhoz tartozó vektorok kiszámítása, amelyek a klaszterek középpontjait reprezentálják a nagy dimenziós embedding térben. Ezek a centroidok a klaszterben található dokumentum- vagy szövegembeddingek átlagát képezik, így minden klaszterhez egyetlen, reprezentatív vektor rendelhető. Matematikailag ez a folyamat úgy írható le, hogy az adott n darab embedding $e_1, e_2, ..., e_n$ alapján a centroid vektor c az alábbi képlettel számolható:

$$c = \frac{1}{n} \cdot \sum_{i=1}^{n} e_i$$

Ez az egyszerű átlagolás ugyan lineáris művelet, de szemantikailag nagy jelentőséggel bír, hiszen a centroid a klaszter elemeinek közös irányát mutatja a vektortérben, azaz azt a pontot, amely a legjobban jellemzi a csoport tartalmi témáját. Másképpen fogalmazva: a centroid egy absztrakt reprezentációja annak a fogalmi mezőnek, amelyet a klaszter elemei körülírnak.

Amikor egy új lekérdezés érkezik, a hagyományos, minden dokumentumot átvizsgáló kereséssel szemben itt nem az összes embeddinggel, hanem csak a centroidokkal történik az első hasonlóság-számítás, amihez koszinusz hasonlóságot használtam.

A lekérdezés embeddingje tehát minden centroidhoz viszonyítva kap egy hasonlósági értéket. Ebből a rendszer kiválasztja a legmagasabb értéket mutató néhány klasztert, majd csak ezekben a klaszterekben végez el részletes keresést. Ez a megközelítés drasztikusan csökkenti a számítási igényt, hiszen a centroidok száma tipikusan több nagyságrenddel kisebb, mint az eredeti chunkoké. Például, ha az adatbázis egymillió chunkból áll, de ezeket ezer klaszterre osztjuk, akkor a rendszer az első körben mindössze ezer hasonlóság-számítást végez, nem pedig egymilliót.

Ezzel a módszerrel egy kétszintű visszakeresési folyamat valósul meg. Az első szint a klaszterszintű szűrés, amely gyors, közelítő keresést biztosít. Itt a cél nem az, hogy a legpontosabb találatokat kapjuk, hanem hogy a keresési tér méretét jelentősen leszűkítsük. A második szint a chunk-szintű szűrés, amely az előző lépésben kiválasztott releváns klasztereken belül történik. Ebben a fázisban már minden chunk embeddinget közvetlenül összevetünk a lekérdezés embeddingjével, így ez a keresés sokkal pontosabb, ugyanakkor a csökkentett adathalmaz miatt továbbra is gyors marad.

Azután, hogy a rendszer azonosította a legjobb embeddingeket a legrelevánsabb klaszterekben a koszinusz hasonlóság alapján, a végső chunkok visszatérítése előtt csak azokat a chunkokat adjuk vissza, ahol a koszinusz hasonlóságuk meghalad egy előre definiált küszöbértéket. Ez a megközelítés biztosítja, hogy a felhasználónak csak a legrelevánsabb és legmegbízhatóbb információk érkezzenek, minimalizálva a zajt és a pontatlan találatokat. Erre azért van szükség, hogy abban az esetben, ha olyan felhasználói kérdés érkezik, amelyre a tudásbázisban nincs megfelelő válasz, ne

térítsünk vissza olyat, amivel esetleg a későbbiekben félrevezetjük a retrieval architektúra felett elhelyezkedő intelligens rendszereket.

A kétszintű struktúra egyensúlyt teremt a sebesség és a pontosság között. A klaszterszintű előszűrés hatékonyan csökkenti a számítási komplexitást, míg a chunk-szintű finomítás garantálja, hogy a végső találatok valóban relevánsak maradjanak. A centroid-alapú retrieval további előnye, hogy könnyen skálázható. Új dokumentumok beérkezése esetén elegendő a hozzájuk tartozó embeddinget kiszámítani és a megfelelő klaszterhez rendelni, illetve a centroidot újraszámolni. Ezzel elkerülhető a teljes embeddingtér újratanítása.

3.2 Javasolt módszer: Online KMeans

3.2.1 A pipeline technikai felépítése

A hagyományos, statikus klaszterezési eljárásokkal szemben, amelyek egy rögzített adathalmazon futnak le egyszer, az általam kidolgozott Online KMeans olyan dinamikus megközelítést valósít meg, amely képes a rendszerbe folyamatosan érkező új adatpontokat, jelen esetben új szöveges chunkokat vagy dokumentumokat, valós időben integrálni a meglévő klaszterstruktúrába. Ennek az adaptív viselkedésnek különösen nagy jelentősége van olyan retrieval pipeline-okban, amelyek élő adatfolyamokat, gyakran frissülő dokumentumkorpuszt vagy felhasználói generált tartalmat kezelnek. Ennek megoldására a pipeline-ba egy online klaszterezési réteget integráltam, amely képes az új embeddingeket folyamatosan beilleszteni a meglévő klaszterstruktúrába, miközben adaptívan frissíti a klaszterek centroidjait. Ez az integráció teszi lehetővé a rendszer valós idejű tanulását és alkalmazkodását a folyamatosan változó adatkörnyezethez.

A beérkező dokumentumokat először előfeldolgoztam, majd pedig az aktuálisan használt sentence transformer modell segítségével elkészítettem a chunkok beágyazásait. Az online klaszterezés központi célja, hogy minden újonnan érkező embeddinget hatékonyan beillesszen a megfelelő klaszterbe, vagy új klasztert hozzon létre, ha a hasonlósági feltételek ezt indokolják.

Az algoritmus működésének alapja a centroidalapú klaszterezés, ahol minden embeddinget a legközelebbi klaszterközépponthoz rendelünk egy választott távolságmérték alapján. Az implementáció támogatja az euklideszi és a koszinuszos metrikát, így alkalmazható mind abszolút értékbeli, mind irányfüggő hasonlóságokra épülő feladatokban. A modell a tanulási folyamat során minden beérkező embedding esetén kiszámítja a vektor és a jelenlegi centroidok közötti távolságot, majd az embeddinget ahhoz a klaszterhez rendeli, amelyhez a legkisebb távolság tartozik. A hozzárendelés során euklideszi távolság metrikát feltételezve minden x_i beágyazási vektort ahhoz a klaszterhez rendelünk, amelynek középpontja r_i a legközelebb esik hozzá:

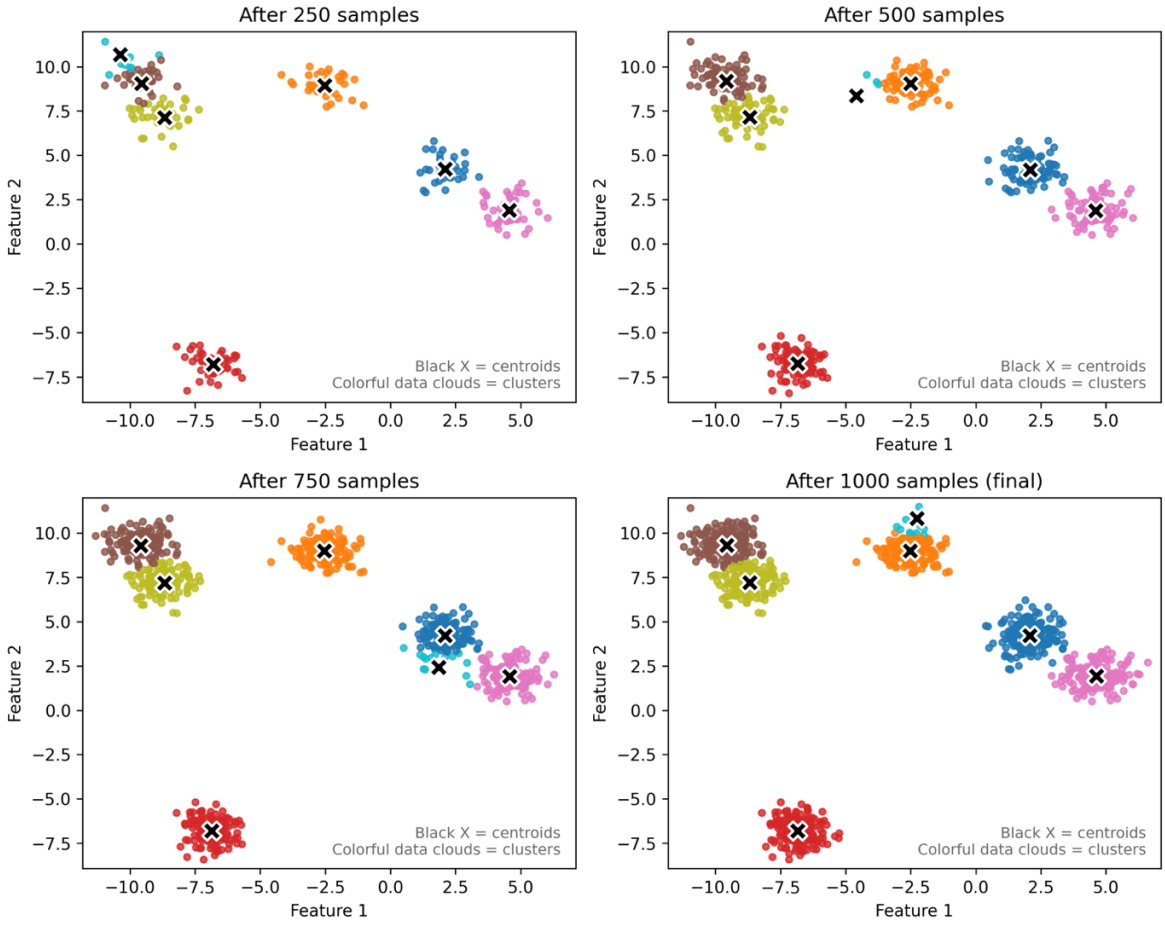
$$C_i = \arg min_j |x_i - r_i|$$

Ahol C_i jelöli az embeddinghez tartozó klaszter indexét, melyet az embedding vektor és a legközelebb eső klaszter középpont euklideszi távolsága alapján határozunk meg. Ezzel szemben koszinuszos metrika esetén a hasonlóságot az egységnormára hozott vektorok skalárszorzatából számítjuk, ahogy azt a korábban található 1. egyenlet is bemutatta.

A pipeline az induláskor a kezdeti centroidokat a kmeans++ [6] inicializálási eljárással határozza meg, amely véletlenszerű, de távolságalapú szelekciót alkalmaz a kezdeti klaszterközéppontok eloszlásának optimalizálására. Ez a megoldás biztosítja, hogy a centroidok már a kezdeti fázisban reprezentálják az adathalmaz különböző régióit, így gyorsabb és stabilabb konvergenciát eredményez.

3.2.2 Dinamikus klaszterszám-kezelés

Az algoritmus egyik legfontosabb tulajdonsága a dinamikus klaszterszám-kezelés. Amennyiben egy új embedding vektor távolsága minden meglévő centroidtól meghalad egy előre definiált küszöbértéket, az algoritmus új klasztert hoz létre abból a pontból. Ez a mechanizmus lehetővé teszi, hogy a modell alkalmazkodjon az adatstruktúra változásaihoz, és új mintázatok megjelenésekor automatikusan bővítse a klaszterkészletét. Ugyanakkor beállítható egy maximális klaszterszám, amely megakadályozza a klaszterek korlátlan növekedését.



3. ábra - Minta példa az online klaszterező algoritmus működésének folyamatára

A klaszterközéppontok frissítése inkrementális statisztikai frissítési képlettel történik, amely egyensúlyt teremt a korábban látott és az újonnan beérkező adatok hatása között. Erre látható egy minta példa a 3. ábrán. Itt az látható, hogy egy szintetikus, 2-dimenziós adathalmazon, hogyan viselkedik az Online KMeans algoritmus. Összesen 1000 adatpont érkezett az adatfolyamban, melyeket az Online KMeans 25-ös batchekben rendelt új klaszterhez. Az ábra ebből a klaszterezési folyamatból mutat pillanatképeket, hogyan néztek ki a klaszterek és a centroidok 250, 500, 750 és 1000 beérkezett adatpont után.

Legyen n_{old} a klaszter korábbi elemszáma, n_{new} az új pontok száma, r_{old} a korábbi centroid, és r_{batch} az új pontok átlaga. A frissített klaszterközéppont a következőképpen számítható:

$$r_{new} = \frac{n_{old} \cdot r_{old} + n_{new} \cdot r_{batch}}{n_{old} + n_{new}}$$

Tehát minden klaszter esetében az algoritmus kiszámítja az adott batch-hez tartozó pontok átlagát és varianciáját, majd ezeket az értékeket a korábbi klaszterparaméterekkel kombinálva pontosítja a centroid helyét és a klaszter szórását. Bár az algoritmus aktuális implementációja a klaszterek varianciáit közvetlenül nem használja fel, ezen statisztikai jellemzők kiszámítása megalapozza a jövőbeli fejlesztések és kiegészítő elemzések lehetőségét. Ez a megközelítés megőrzi a korábbi információt, ugyanakkor fokozatosan beépíti az új adatokat, így a rendszer folyamatosan tanul anélkül, hogy az előző állapotot elfelejtené.

Az online klaszterezés előnye, hogy nem igényli az egész adathalmaz memóriában tartását, és alkalmas folyamatosan változó környezetekhez. Azonban a korábban említettek alapján egyértelműen látszik az is, hogy ez a megközelítés is, mint minden más online klaszterező algoritmus, függ az adatok érkezési sorrendjétől, amely egy hátrányukat képezi. Ez a sorrendfüggőség különösen akkor válik kritikussá, ha az adatok nem véletlenszerű sorrendben, hanem például időben rendezve érkeznek. Ilyen esetekben a korai minták nagyobb hatással lehetnek a klaszterek kezdeti pozícióira, ami torzításhoz vezethet.

Ennek elkerülése végett bevezettem a modell egy további sajátosságát: a klaszterek összeolvasztása. Ez a mechanizmus akkor aktiválódik, ha két centroid közötti távolság egy előre megadott határérték alá csökken. Ilyen esetben a két klasztert súlyozott átlagolással egyesíti, miközben frissíti a varianciaértékeket. Ez a módszer segít elkerülni a redundáns vagy egymáshoz túl közeli klaszterek kialakulását, valamint elősegíti a stabilabb és értelmezhetőbb klaszterstruktúra fenntartását. A jövőben az eltárolt variancia értékek alapján bevezethető egy további mechanizmusa az algoritmusnak, amely a túl nagy szórással rendelkező klaszterek esetleges szétválasztása.

Az implementáció mindemellett egy state management mechanizmust is tartalmaz, amely lehetővé teszi a modell aktuális állapotának (centroidok, számlálók, varianciák, feldolgozott adatok száma) kinyerését és mentését. Ez különösen hasznos online tanulási környezetben, ahol a folyamat bármikor megszakítható és újraindítható anélkül, hogy az addigi tudás elveszne.

Ez a módszer különösképp memóriahatékony is, hiszen futás közben csak az adott batchben érkező embeddingek, az aktuális klaszterközéppontokat tartalmazó tömb, egy vektor, amely azt tartja nyilván, hogy az egyes klaszterekhez eddig mennyi adat tartozott, a klaszterekhez tartozó pontok futó összege, ami minden klaszter esetén összegzi az oda tartozó pontok koordinátáit, a klaszterekhez egyetlen számként nyilvántartott variancia és egy skalár, ami azt mutatja meg, hogy összesen mennyi pontot dolgozott fel a modell idáig, van a memóriában.

3.2.3 Idő- és memóriakomplexitás

	Online KMeans	**MiniBatchKMeans**
Időkomplexitás	O(n·k·d)	O(n·k·d·t)
Memóriakomplexitás	O(k·d)	O(k·d)

1. táblázat – Idő- és memóriakomplexitás összehasonlítása

A 1. táblázatban látható az Online KMeans algoritmusom idő- és memóriakomplexitása összehasonlítva a MiniBatchKMeans [18] algoritmussal. A táblázatban *n* az adott batchben érkező adatok számát, *k* a klaszterek számát, *d* az adatok dimenzióját és *t* pedig az iterációk számát jelöli. A MiniBatchKMeans időkomplexitását [4] cikk adta meg, a memóriakomplexitásáról hivatalos tanulmányt nem találtam, viszont a scikit-learn hivatalos dokumentációja alapján [18] a memóriahasználat főként a klaszterközéppontok és a batch adatok tárolásából, illetve további a klaszterekhez tartozó metainformációk tárolásából adódik, ezutóbbi viszont nem jelent jelentős memóriaigényt. Emiatt memóriakomplexitás tekintetében a két algoritmus közel azonosan viselkedik. Időkomplexitás szempontjából ugyanakkor az Online KMeans algoritmus gyorsabb, mivel egyetlen iteráció során frissíti a klaszterközéppontokat, így az iterációk száma t = 1. Ezzel szemben a MiniBatchKMeans algoritmus több iterációt igényel a konvergencia eléréséhez, ezért a futásideje arányosan növekszik az iterációk számával.

3.2.4 Online KMeans algoritmus

```
1. KMeans++ inicializálás
2. FOR x_adat IN batch:
     distance_adat = min(distance(x_adat, centroid)) minden centroidra
     IF distance_adat > new_cluster_threshold:
          x_adat új klaszterbe kerül
     ELSE:
          x_adat a legközelebbi klaszterbe
3. FOR minden centroidra:
     centroid_i = klaszter pontjainak kombinált átlaga (lásd feljebb)
     Klaszterhez tartozó pontok számának, koordinátáinak összegének és klaszter
     szórásának frissítése
4. Klaszterek összevonása
     IF két centroid távolsága (i és j) < merge_threshold:
          centroid_i = két klaszter pontjainak átlaga
          Klaszter (i)-hez tartozó pontok számának, koordinátáinak összegének és
          klaszter szórásának frissítése
          Klaszter (j) törlése
```

4. ábra - Az Online KMeans pszeudokódja

A 4. ábrán található pszeudokód áttekinthető formában foglalja össze az Online KMeans algoritmusom működését. Tömör módon jeleníti meg a klaszterek inicializálását, az adatpontok hozzárendelését, a centroidok frissítését, az új klaszterek létrehozását és a közel eső klaszterek összevonását, így gyors képet ad az algoritmus logikájáról.

A centroidok frissítése és új embeddingek klaszterekbe sorolását követően elkezdődhet a retrieval fázis a korábban bemutatott módszer alapján. Először a felhasználói kérdést klaszter-szinten hasonlítjuk össze, majd csak ezt követően végzünk részletes keresést a legrelevánsabb klaszterekben szereplő embeddingek között. Végül pedig visszatérítjük az így kapott leghasonlóbb embedding vektorokat, melyek alapján a későbbiekben egy nagy nyelvi modell vagy más intelligens rendszer képes lehet a felhasználói kérdésre egy pontosított választ adni.

4 Adathalmaz és előfeldolgozás

4.1 A SQuAD adathalmaz

A jelenlegi munkában a SQuAD (Stanford Question Answering Dataset) [12] adathalmazt alkalmazom a RAG rendszer fejlesztéséhez és kiértékeléséhez. A SQuAD egy széles körben használt, nyílt forráskódú kérdés-válasz adathalmaz, amely tartalmazza a Wikipédia-szövegekből származó kontextusokat és a hozzájuk tartozó kérdéseket, valamint a helyes válaszokat. Az adathalmaz két fő részből áll: a train és a validation szettből, amelyek segítségével a modell betanítható és validálható. A train és validációs adatokat a datasets könyvtár segítségével töltöttem be, majd Pandas DataFrame-be konvertáltam a könnyebb feldolgozás érdekében. A kontextusokhoz egyedi context_id azonosítót rendeltem, ezzel biztosítva, hogy a kérdések és a hozzájuk tartozó szövegek könnyen összekapcsolhatók legyenek. A válaszokat a SQuAD struktúrából kinyertem, külön mezőkbe helyeztem (answer_text, answer_start), ezzel megkönnyítve a későbbi összehasonlítást és kiértékelést.

A SQuAD adathalmaz elsődlegesen arra szolgál, hogy a modellek képesek legyenek a kontextusból pontosan kiválasztani a kérdésre adott választ. A kontextusok hosszúsága változó, és gyakran több bekezdést tartalmaznak, ami kihívást jelent a nagy nyelvi modellek számára, különösen akkor, ha az egész dokumentumot egyszerre kellene feldolgozni. Emiatt a chunking és a szövegfeldolgozás kulcsfontosságú előfeldolgozási lépések, amelyek lehetővé teszik a nagyobb szövegek kezelhetőségét és a releváns információk kiemelését.

4.2 Szövegfeldolgozás és chunking

A szövegfeldolgozás első lépéseként a SQuAD adathalmazban található kontextusokat megtisztítottam a nem kívánt karakterektől és a felesleges szóközöktől. Ez a lépés kulcsszerepet játszik a folyamatban, mivel az adathalmazban gyakran előfordulnak HTML-entitásokból származó szimbólumok, speciális karakterek, illetve többszörös sortörések és tabulátorok, amelyek zavarhatják a későbbi nyelvi modellek működését. A tisztításhoz reguláris kifejezéseket (regex) alkalmaztam, amelyek segítségével kiszűrtem a nem alfanumerikus karaktereket, valamint egységesítettem a whitespace karakterek használatát. A szövegeket először megtisztítottam az idegen szimbólumoktól, majd egyetlen szóközre redukáltam az egymás után következő üres karaktereket, végül eltávolítottam a szöveg elején és végén található felesleges szóközöket.

A tisztított szövegeket ezt követően chunkolási eljárásoknak vetettem alá, amelyek célja a hosszú kontextusok kisebb, jól kezelhető egységekre bontása volt. A chunkolás azért szükséges, mert a nyelvi modellek, különösen a transformer alapú architektúrák, csak korlátozott hosszúságú bemenetet képesek feldolgozni, jellemzően 512–1024 token között [24]. Ezért a dokumentumokat több, egymást

részben átfedő szövegszegmensre bontottam, hogy minden releváns információ elérhető maradjon a későbbi visszakeresés során.

A chunkolási eljárásnak többféle megközelítése is létezik [21], mint például mozgó ablakos chunkolás (sliding window chunking), szemantikus chunkolás (semantic chunking) vagy agentic chunking. A sliding window chunking esetében a szöveget fix hosszúságú, egymással átfedésben lévő szeletekre bontjuk. Ennek célja, hogy a modellek kontextusablakának korlátozott mérete mellett is megmaradjon a szövegrészek közötti folytonosság. Az agentic chunking során a chunkolást egy nagy nyelvi modell végzi, amely a szöveg tartalmát és szerkezetét elemezve, adaptívan határozza meg a szeletek határait. Ez lehetővé teszi, hogy a chunkok a szemantikai összefüggéseket és a logikai egységeket természetes módon kövessék.

A munkám során a szemantikus chunkolást alkalmaztam, amely nem fix hosszúságú ablakokra épült, hanem a mondatok közötti hasonlóság alapján választotta el az egységeket. Ehhez a szöveget először mondatokra bontottam az NLTK (Natural Language Toolkit) könyvtár *sent_tokenize()* függvényével [9], majd minden mondatot vektorra képeztem egy SentenceTransformer modell segítségével, konkrétan az „all-MiniLM-L6-v2” [14][23] architektúrával. A modell a mondatokat nagy dimenziós vektortérbe ágyazza, ahol a hasonló jelentésű mondatok közel kerülnek egymáshoz. Ezt kihasználva kiszámítottam az egymást követő mondatok közötti koszinusz hasonlóságot, és ha az érték meghaladta a 0,6-os küszöböt, a mondatokat egy közös chunkba sorolta. Amikor a hasonlóság ez alá a küszöbérték alá esett, új chunk kezdődött. A módszer így képes volt dinamikusan, a tartalmi folytonosság alapján meghatározni az optimális szeleteket, miközben figyelembe vette a maximális 500 karakteres hosszkorlátot is.

A chunkolási eljárás kimenetét egy egységes struktúrában tároltam. Iteratívan végighaladtam az adathalmazon, és minden dokumentumból több chunkot generáltam. Minden chunkhoz hozzárendeltem a forrás dokumentum azonosítóját, a chunk pozícióját, valamint egyedi azonosítóját.

A chunkolás eredményeként egy strukturált, egységesített és tisztított szöveges adathalmaz jött létre, amelyben minden dokumentum több kisebb egységre bomlott. Ez a feldolgozott forma már közvetlenül alkalmas volt a chunkszintű embeddingek előállítására, illetve a dokumentum-visszakeresési modul elkészítésére.

4.3 Embedding generálás

A chunkok előállítása után a következő lépést az embeddingek (szövegbeágyazások) elkészítése jelentette, amelynek célja, hogy a szöveges tartalmakat numerikus vektortérbeli reprezentációkká alakítsuk, amely lépés alapvető fontosságú a későbbi szemantikus visszakeresési és klaszterezési folyamatok szempontjából.

A beágyazások előállításához a SentenceTransformer keretrendszert alkalmaztam, amely kifejezetten alkalmas természetes nyelvű szövegek kompakt és informatív vektorreprezentációinak

előállítására. Annak érdekében, hogy a modell robosztusságát és általánosíthatóságát is vizsgálni lehessen, több különböző architektúrával is elvégeztem a beágyazásokat. A választott modellek között szerepelt az „all-MiniLM-L6-v2” [23], amely 384 dimenziós vektorokat generál, valamint a „Snowflake/snowflake-arctic-embed-l-v2.0” [13] modell, amely a chunkokat 1024 dimenziós térbe képezte le. Ez lehetővé tette, hogy a későbbi kísérletekben összehasonlítsam a különböző dimenziójú embeddingek teljesítményét, mind a visszakeresési, mind a klaszterezési feladatokban.

Az embeddingek generálása során batch-szerű feldolgozást alkalmaztam a memóriahasználat optimalizálása és a feldolgozási sebesség növelése érdekében. Minden feldolgozott chunk beágyazását elmentettem egy DataFrame struktúrában, illetve NumPy tömbként is, ami lehetővé teszi a hatékony, vektoralapú műveletek (mint például normalizálás, klaszterezés vagy valós idejű keresés) végrehajtását. Az így kapott vektorokat L2 normalizálással előfeldolgoztam, hogy a koszinusz hasonlóságon alapuló keresési és klaszterezési eljárások gyorsabb eredményeket adjanak.

5 Eredmények és összehasonlítás

5.1 Klaszterezési módszerek hatékonyságának vizsgálata

Az eredmények bemutatását és a kiértékelési módszerek ismertetését a különböző klaszterezési algoritmusok hatékonyságának összehasonlító vizsgálatával kezdem. A kísérletek célja annak feltárása volt, hogy az egyes megközelítések milyen mértékben képesek hatékonyan és stabilan kezelni a dinamikusan változó, nagyméretű adatbeáramlást. Ennek érdekében három különböző klaszterezési módszert elemeztem és értékeltem kvantitatív módon.

Az első vizsgált módszer az előző fejezetben részletesen bemutatott Online KMeans algoritmusom, amely az adatok folyamatos érkezéséhez adaptálódva frissíti a klaszterközéppontokat, és képes a korábbi tanulási eredmények megőrzése mellett az új információk integrálására.

A második vizsgált megközelítés a MiniBatchKMeans [18] algoritmus, amely a klasszikus KMeans továbbfejlesztett, részben online működésre képes változata. Ez a módszer szintén batch-alapú feldolgozást alkalmaz, azonban nem támogatja a klaszterszám dinamikus módosítását a beérkező adatok függvényében. Ennek következtében a modell merevebb, és kevésbé alkalmas az olyan környezetekben történő alkalmazásra, ahol az adatok eloszlása időben változhat, vagy új minták jelenhetnek meg.

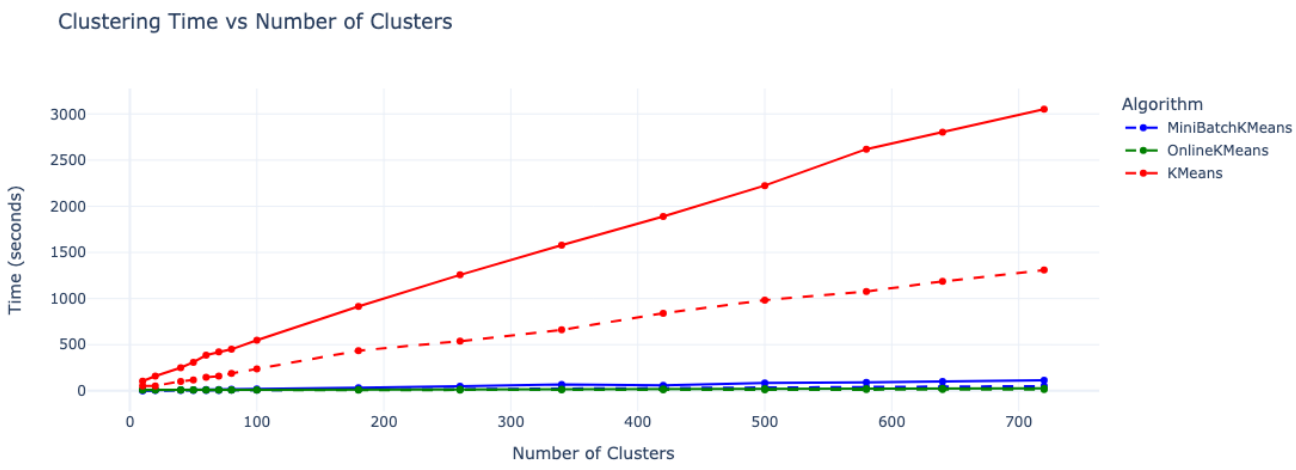
A harmadik elemzett módszer az alap KMeans [16] algoritmus, amely a legelterjedtebb klaszterezési eljárások közé tartozik. Bár egyszerű és hatékony statikus környezetben, hátránya, hogy teljes újratanítást igényel, amennyiben új adatok érkeznek, így valós idejű vagy folyamatosan frissülő rendszerek esetén kevéssé alkalmazható.

A vizsgálatok során alkalmazni szerettem volna egy negyedik módszert is, a River keretrendszerben található KMeans algoritmust [5], amely kifejezetten online, streaming jellegű adatok feldolgozására lett tervezve. Ez az algoritmus a learn_one() függvény segítségével képes új adatpontok egyenkénti megtanulására, ami előnyös lehet folyamatosan érkező, valós idejű adatfolyam esetén. Ugyanakkor ez a megközelítés a jelen munka szempontjából nem bizonyult célszerűnek, mivel nem teszi lehetővé az adatok batchenkénti feldolgozását, illetve nem támogatja egy nagy, előre rendelkezésre álló adathalmazról történő inicializálást.

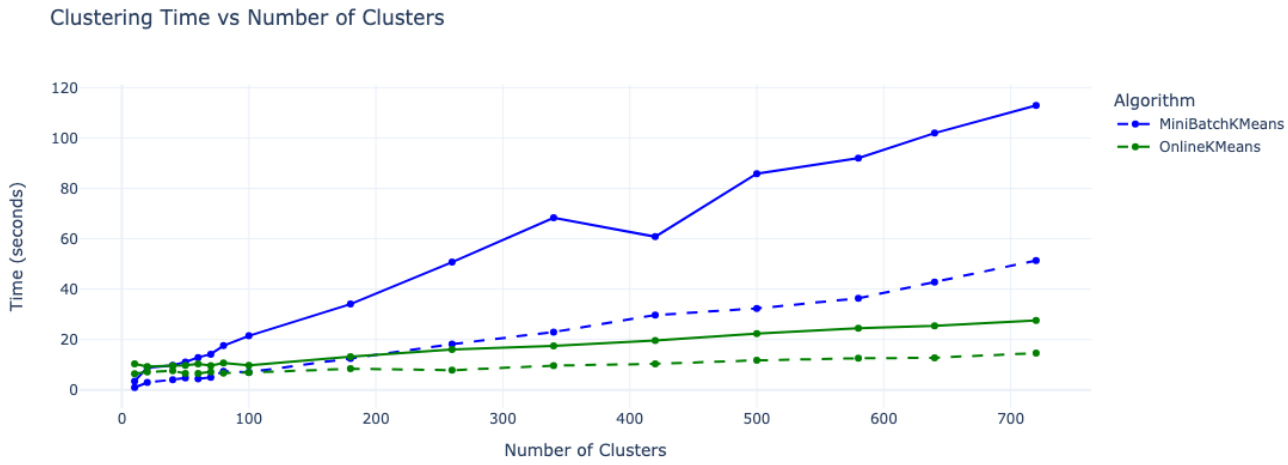
Mivel kutatásom egyik központi célja egy olyan klaszterezési megoldás kialakítása volt, amely nagy méretű, előre ismert tudásbázisra épül, és képes annak dinamikus, folyamatos frissítésére a későbbiekben beérkező adatok alapján, ezért elengedhetetlen követelmény volt a batch alapú feldolgozás és az előzetes inicializálhatóság. Ennek megfelelően a River könyvtár KMeans algoritmusa a további elemzésekből kizárásra került.

Először vizsgáljuk meg a különböző algoritmusokat futásidő szempontjából. Ehhez a már bemutatott SQuAD adathalmazt használtam, amiből a chunkolást követően 384 és 1024 dimenziós

szövegbeágyazásokat is készítettem. Összesen 84007 darab embedding készült 18891 egyedi dokumentumból, amelyeket ezt követően klaszterezési eljárásnak vetettem alá.



5. ábra - KMeans, MiniBatchKMeans és Online KMeans futásideje 384 és 1024 dimenziós embeddingeken különböző számú klasztert feltételezve.



6. ábra - MiniBatchKMeans és Online KMeans futásideje 384 és 1024 dimenziós embeddingeken a maximális indulási klaszterszám függvényében. (5. ábra alsó része nagyítva)

Az 5. ábra az Online KMeans, a klasszikus KMeans és a MiniBatchKMeans algoritmusok futási idejét mutatja be az embedding halmazokon végzett kísérletek során. Szaggatott vonal jelöli a futásidők alakulását a 384-dimenziós, folytonos vonal pedig az 1024-dimenziós embedding vektorok esetén. Jól látható, hogy a KMeans algoritmus sokkal lassabban képes elvégezni a klaszterezést, mint az Online KMeans vagy a MiniBatchKMeans. Ez nem meglepő eredmény, hiszen a klasszikus KMeans a működése során minden iterációban az összes adatpontot újraértékeli, hogy kiszámolja a távolságokat a centroidokhoz, majd újra frissítse azokat. Ez azt jelenti, hogy az egész adathalmaznak egyszerre kell elérhetőnek lennie a memóriában, ami nagy adatok esetén komoly korlátozó tényező. Vizsgáljuk meg most az 6. ábra alapján a MiniBatchKMeans és az Online KMeans algoritmusokat futásidő

szempontjából. Itt az látható, hogy egészen addig, amíg nem érünk el nagyobb kiindulási klaszterszámot, a két algoritmus ugyanolyan eredményt ér el. Azonban, amint a klaszterszám megnövekszik a MiniBatchKMeans futásideje elkezd megnőni. Ezzel szemben az Online KMeans algoritmus még nagy klaszterszámok esetén is kiegyensúlyozott teljesítményt nyújt futásidőben.

Annak érdekében, hogy a különböző klaszterezési eljárásokat még jobban össze tudjuk hasonlítani egymással, készítettem több szintetikus, magas dimenziós adathalmazt a scikit-learn datasets könyvtárának make_blobs() függvényével [17]. A választott dimenziószám 500 volt, és 200, 300, 400, 500 és 800 klaszterből álló, adathalmazokat készítettem, majd ezen adathalmazok mindegyikén kiértékeltem a három klaszterező algoritmust ARI (Adjusted Rand Index) [15] és NMI (Normalized Mutual Information) [19] metrikákkal. Mivel a make_blobs() függvény normál eloszlású adathalmazokat generál, ezért mindegyik klaszterszámhoz többféle, eltérő szórású változatban (σ = 2, 4, 6, 8 és 10) is elkészítettem az adathalmazokat, hogy átfogóbb és robusztusabb képet kapjak az algoritmusok teljesítményéről.

Az ARI a klaszterezés és a valós címkék közötti hasonlóságot méri, figyelembe véve a véletlenszerű egyezéseket is. Az ARI értéke -1 és 1 között mozog. Az 1 tökéletes egyezést jelent, a 0 azt, hogy a klaszterezés véletlenszerű, a negatív értékek pedig a rosszabb, mint véletlenszerű egyezést jelzik. Az ARI számításánál minden pár adatpontot figyelembe veszünk, és megnézzük, hogy az adott pár ugyanabban a klaszterben van-e mindkét felosztásban, vagy sem.

Az NMI a klaszterezés és a valós címkék közötti információmegoszlást méri, normalizálva az egyes felosztások entrópiájával, így az értéke 0 és 1 között van. A 0 azt jelzi, hogy nincs információs egyezés a klaszterek és a valós osztályok között, az 1 pedig a teljes egyezést. Az NMI érzékeny az információs tartalomra, nemcsak a klaszterek pontos egyezésére, ezért jól használható nagy klaszterszámú, vagy részben átfedő klaszterezési feladatok értékelésére.

Klaszter szám	KMeans	MiniBatchKMeans	OnlineKMeans
200	0,969	0,897	0,964
300	0,957	0,664	0,963
400	0,96	0,5	0,962
500	0,951	0,498	0,964
800	0,87	0,014	0,952

2. táblázat - Átlagos ARI értékek a különböző algoritmusokra.

Klaszter szám	KMeans	MiniBatchKMeans	OnlineKMeans
200	0,995	0,99	0,988
300	0,995	0,957	0,989
400	0,995	0,716	0,989
500	0,995	0,816	0,99
800	0,992	0,704	0,989

3. táblázat- Átlagos NMI értékek a különböző algoritmusokra.

Ahogy az 2. és 3. táblázat adatai is mutatják, az Online KMeans algoritmus az ARI és az NMI metrikák tekintetében szinte azonos teljesítményt nyújt a hagyományos, teljes adathalmazt feldolgozó KMeans algoritmussal. Ezzel szemben a MiniBatchKMeans jelentősen gyengébb eredményeket produkál mindkét értékelési mutató alapján.

A MiniBatchKMeans teljesítményének csökkenése részben abból adódhat, hogy az algoritmus a klaszterközéppontokat batch-enként frissíti. Ha az egyes batch-ek túl kevés adatpontot tartalmaznak, az iterációk során számított centroidok pontatlanok lehetnek, különösen szoros vagy egyenlőtlen méretű klaszterek esetén. Ezzel szemben az Online KMeans az adatok beérkezésével folyamatosan frissíti a klasztereket, így jobban alkalmazkodik a klaszterstruktúrákhoz, különösen nagy dimenziós térben. Jelen esetben is ez történt, mivel a MiniBatchKMeans és Online KMeans esetén is 1000 adatponton inicializáltam az algoritmust, ezután pedig ugyanekkora batchekben érkeztek az új adatok, összesen 10000. Az Online KMeans alkalmazkodni tudott az új típusú adatokhoz, és képes volt létrehozni az új klasztereket. A MiniBatchKMeans azonban az 1000 ponton történő inicializálást követően nem tudta kezelni az egyébként új klaszterekbe sorolandó bejövő adatpontokat, emiatt ért el ilyen alacsony ARI értékeket, és teljesített gyengébben NMI mutatóban is.

További előnyt jelent, hogy az Online KMeans a klaszterek definiálásához koszinusz távolságot alkalmaz az euklideszi helyett. Magas dimenziós embedding terekben a koszinusz hasonlóság jobban tükrözi a vektorok közötti szemantikai viszonyokat, mivel a nagydimenziós vektorok közötti „irány” gyakran informatívabb, mint a tényleges euklideszi távolság. Emellett az Online KMeans lehetőséget ad új klaszterek dinamikus létrehozására és meglévők összevonására is. Ennek eredményeként az Online KMeans gyakran képes javítani az ARI és NMI értékeket, még akkor is, ha a végső klaszterszám nem pontosan egyezik a paraméterként megadott klaszterszámmal.

5.2 Klaszterezés-alapú és teljes vektorkeresés összehasonlítása

Az elért eredmények bemutatását és a kiértékelést a klaszterezés-alapú, a teljes vektorkeresés, valamint a state-of-the-art FAISS index alapú megoldás [8] összehasonlításával folytatom. A klaszterezés-alapú megközelítésben tehát a felhasználói kérdést először a klaszter-középpontokkal vetjük össze, ezt követően pedig a legrelevánsabb klaszterekben végzünk részletesebb keresést, és térítjük vissza az így talált legrelevánsabb chunkokat. A teljes vektorkeresés esetében pedig a vektor adatbázis minden elemét összehasonlítjuk a felhasználó által feltett kérdésből képzett embeddinggel. A FAISS index esetében az előállított embeddingekből létrehoztam az index struktúrát, amely a korábban ismertetett, HNSW-alapú (Hierarchical Navigable Small World) [27] megközelítésre épül, amely egy gráfalapú approximációs keresési algoritmust használ, aminek a célja gyorsan megtalálni a legközelebbi vektorokat nagy dimenziós térben, anélkül, hogy minden pontot végig kellene vizsgálni.

A kiértékelés során annak érdekében, hogy minél pontosabb képet kapjak az architektúra működéséről. A dokumentumok 15%-át eltávolítottam a szövegbeágyazások közül, viszont a hozzájuk

tartozó kérdéseket megtartottam. Ezzel azt szeretném szimulálni, hogy az algoritmus hogyan teljesít olyan kérdéseken, amikhez nem szerepel válasz a dokumentumok között, ezáltal átfogóbb képet kapva a megoldásom robusztusságáról. Ennek érdekében annyival egészítettem ki az algoritmust, hogy a legrelevánsabb chunkok visszatérítése után még egy vizsgálatnak vetem alá az embeddingeket. Amennyiben a legmagasabb hasonlósági mutatóval visszatérített chunk nem éri el a 0,6-os küszöbértéket, úgy az algoritmus nem adja vissza a korábban visszatérített chunkok egyikét sem, mert úgy ítéli meg, hogy egyik válasz sem kellően releváns a feltett kérdéshez, így valószínűleg a rendelkezésre álló dokumentumok között nem is szerepel a kérdésre válasz.

A kiértékelés során többféle metrikát is alaposan vizsgáltam, mint például pontosság (accuracy) vagy fedés (recall). Az accuracy azt mutatja meg, hogy az összes előrejelzés hány százaléka volt helyes, a recall pedig azt írja le, hogy az összes valóban pozitív eset közül hányat talált meg a modell. Ezeket az értékeket mind dokumentum-szinten és chunk-szinten is külön kiszámoltam, hogy egy teljesebb képet kapjak az algoritmusok működéséről. Dokumentum-szinten azt vizsgáltam, hogy abban az esetben, ha van válasz a felhasználó által feltett kérdésre, akkor a modell által visszatérített chunkok között megtalálható-e annak a dokumentumnak az azonosítója, amelyikre a feltett kérdés vonatkozik. A helyesen eltalált és helytelen eseteket összegeztem, majd ebből számoltam pontosságot és fedést. Chunk-szintű kiértékelést csak abban az esetben volt értelme végezni, ha a visszatérített chunkok között volt olyan, amelyik abból a dokumentumból származott, amelyikre a feltett kérdés vonatkozott. Tehát, ha volt ilyen chunk a visszatérítettek között, akkor először ezeket kigyűjtöttem. A SQuAD adathalmaz sajátossága az, hogy a kérdésekhez megtalálható az is, hogy karakterre pontosan honnan kezdődik rá a válasz a dokumentumban. Emiatt a chunkolást már úgy végeztem el korábban, hogy minden chunkhoz eltároltam, hogy a dokumentum hányadik karakterénél kezdődik és melyiknél végződik, így könnyen meg tudtam határozni, hogy a visszatérített chunkok melyike tartalmazta a kérdésre a választ. Ezután összegeztem azt, hogy mennyi olyan alkalom volt, amikor vissza lett térítve a releváns chunk is, és mennyi, amikor nem. Ezekből az értékekből pedig tudtam pontosságot és fedést számolni. A kiértékelés során onnan tudtam hamis negatív és hamis pozitív eredményeket előállítani, hogy beállítottam egy küszöbértéket 0,6-ra, amivel azt vizsgáltam, hogy egyáltalán van-e olyan chunk, ami kellően releváns a kérdésre, mert ha egyik sem érte el a 0,6-os szintet, akkor az algoritmus úgy döntött, hogy nincs válasz az embeddingek között a kérdésre. A küszöbértéket azért 0,6-nak választottam meg, mert korábban a chunkolási eljárás során is 0,6-os threshold jelentette a határt két chunk között. Ennek a feltevésnek a hibáiból keletkeztek fals pozitív és fals negatív találatok.

A következőkben az alábbi módszereket hasonlítom össze: először azt az esetet, amikor minden kérdéshez az összes embeddinget közvetlenül összehasonlítjuk (brute force); másodszor azt, hogyan változik az eredmény, ha a keresés során a state-of-the-art FAISS indexeket használjuk; továbbá pedig vizsgálom a kétlépcsős keresési módszert, amely először a centroidok közötti, majd a legjobb klasztereken belüli kimerítő keresést alkalmazza. Az összehasonlítást accuracy és recall mutatók alapján

végzem. A kétlépcsős módszer esetén a klaszterszám 500 volt, ami úgy került meghatározásra, hogy korábbi feltáró adatelemzések során KMeans algoritmusra megpróbáltam megtalálni az ideális klaszterszámot Silhouette-pontszám alapján, aminek lokális maximuma 500 klaszternél volt, így az embeddingeket klasszikus KMeans-sel 500 klaszterbe soroltam.

Top k best chunks returned	Document accuracy	Chunk accuracy	Document recall	Chunk recall
3	0,75	0,66	0,71	0,60
5	0,78	0,70	0,74	0,64
12	0,81	0,74	0,78	0,69
25	0,83	0,77	0,80	0,73

4. táblázat - Brute force megoldás által elért eredmények retrieval során

Top k best chunks returned	Document accuracy	Chunk accuracy	Document recall	Chunk recall
3	0,74	0,66	0,70	0,60
5	0,77	0,69	0,73	0,64
12	0,80	0,73	0,77	0,68
25	0,82	0,76	0,79	0,71

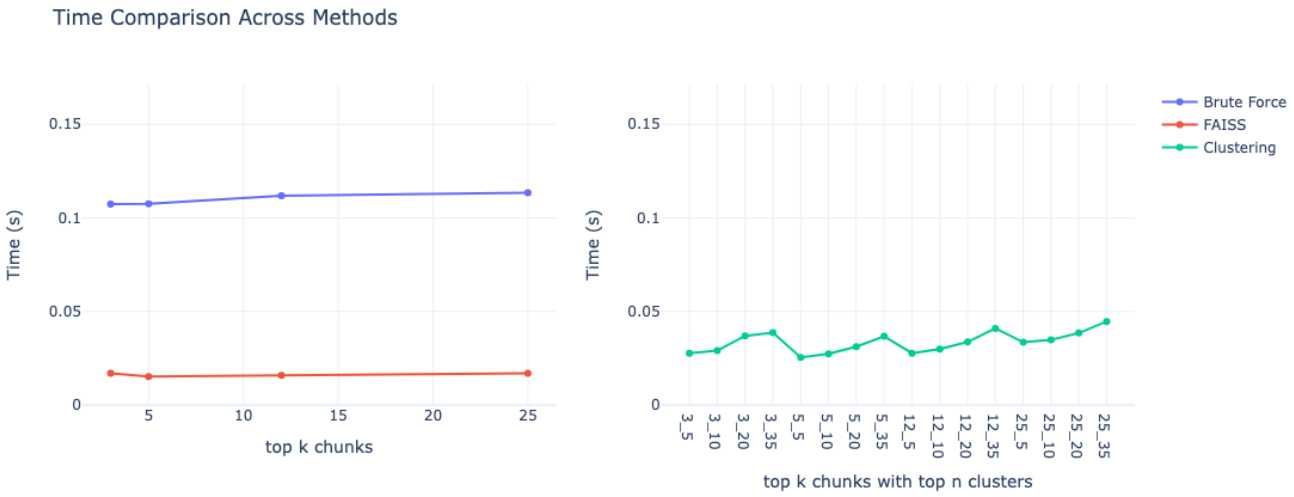
5. táblázat – FAISS indexszel elért eredmények retrieval során

Top k best chunks returned	Top n best clusters returned	Document accuracy	Chunk accuracy	Document recall	Chunk recall
3	5	0,67	0,60	0,62	0,52
3	10	0,70	0,62	0,65	0,55
3	20	0,72	0,66	0,67	0,57
3	35	0,73	0,65	0,69	0,58
5	5	0,70	0,62	0,65	0,55
5	10	0,73	0,65	0,68	0,58
5	20	0,75	0,67	0,70	0,61
5	35	0,76	0,68	0,72	0,62
12	5	0,73	0,65	0,68	0,59
12	10	0,76	0,68	0,71	0,63
12	20	0,78	0,70	0,74	0,65
12	35	0,79	0,72	0,75	0,67
25	5	0,74	0,67	0,70	0,61
25	10	0,77	0,70	0,73	0,65
25	20	0,79	0,73	0,76	0,68
25	35	0,81	0,74	0,77	0,70

6. táblázat – Kétlépcsős (klaszter-szintű majd klasztereken belüli) keresés által elért eredmények retrieval során

A 4., 5. és 6. táblázat tartalmazza a különböző algoritmusok által elért eredményeket. Mindhárom esetben dokumentum és chunk szinten is pontosságot és fedést számítottam ki. A *top k best chunks returned* jelenti azt, hogy a k darab legrelevánsabb chunkot térítette vissza az algoritmus. A *top n best clusters returned* pedig azt jelöli a kétlépcsős megközelítés esetén, hogy az n darab legjobb klaszterben végzünk teljes keresést.

Ahogy az látható az elért eredményekből is a brute force megoldás adja a legmagasabb accuracy és recall értékeket is, ami várható volt, hiszen ebben az esetben minden egyes embedding vektorral összehasonlítjuk a query-t. A FAISS index esetén átlagosan 1% értékkel érünk el gyengébb eredményt, mint a brute force megoldás, a kétlépcsős módszer pedig további 1%-kal teljesít gyengébben. A kétlépcsős módszer esetén fontos azt is megvizsgálni, hogyan változik az accuracy és recall értéke a k (visszatérített chunkok száma) és n (legrelevánsabb klaszterek száma) függvényében. Ebből az látható, hogy minél több klaszterben végzünk teljes keresést, annál jobb eredményt érünk el. Ezenkívül minél több chunkot térítünk vissza, szintén annál magasabb accuracy és recall értékek keletkeznek. Az, hogy k és n paramétereket mi alapján választjuk meg, több mindentől függ. Minél nagyobb elemszámú klasztereink vannak, annál számításigényesebb egy klaszterben elvégezni a teljes keresést. Ezenkívül a k paraméter értékét is körültekintően kell megválasztani. Ha túl alacsony, túl nagy valószínűséggel hagyjuk ki a releváns chunkokat. Ellenben, ha túl nagynak választjuk, túl sok olyan chunk is vissza lesz térítve, ami a kérdéshez nem tartalmaz releváns információt. Azt lehet megállapítani, hogy az általam használt adathalmaz esetén n=10 és k≈12 értékekkel érhetjük el a továbbiakban a leghatékonyabb eredményt, így további munkám során k értékét 10-nek választottam. A k=25 esetén hiába magasabb a pontosság, a nem releváns chunkok is túl nagy mértékben szerepelnének az eredmények között, mivel jelen adathalmaz esetén biztosan tudni, hogy egy dokumentum átlagosan 4-5 chunkból áll.



7. ábra - Az egyes algoritmusok által elért retrieval idők összehasonlítása.

A 7. ábrán az látható, hogy hogyan változik az egyes módszerek futásideje annak függvényében, hogy k és n paraméterek milyen értékeket vesznek fel. Az itt látható másodperc értékek

azt mutatják meg, hogy mennyi időbe telik az egyes algoritmusoknak egy query-hez visszatéríteni a releváns chunkokat. A brute force megoldás jóval lassabb retrieval sebesség elérésére képes, mint a FAISS vagy a kétlépcsős, klaszterezést használó megközelítések. Ez amiatt történik, hogy a vektor adatbázis összes vektorán hajtjuk végre a hasonlóság-számításokat. A kétlépcsős megoldás esetén az vehető észre, hogy nagyban függ a retrieval time k és n paraméterek értékétől. Minél több klaszterben végzünk teljes, kimerítő keresést, annál lassabb retrieval időket érhetünk el, de így is csak kevéssel lesz lassabb a futásidő, mint FAISS indexek esetén. Ahogy az látható is a FAISS-ra épülő megoldás nagyon gyors chunk visszatérítésre képes függetlenül attól, hogy mi k paraméter értéke. Ez a FAISS által használt adatszerkezet miatt van így, hiszen a FAISS optimalizált, vektoralapú indexstruktúrákat alkalmaz, jelen esetben HNSW-t, amely lehetővé teszi a lekérdezések hatékony, közel valós idejű végrehajtását még nagyméretű beágyazástér esetén is.

Vizsgáljuk meg általános esetben az egyes megközelítések futásidejét. A brute force módszer esetében minden lekérdezésvektort az adatbázisban szereplő összes embeddinggel összevetünk. Ennek az eljárásnak az időkomplexitása O(N·d), ahol N az adatbázisban található vektorok száma, d pedig az embeddingek dimenziója. A megközelítés előnye, hogy 100%-ban pontos eredményt kapunk, tehát mindig a leghasonlóbb vektort adja vissza, hátránya viszont a skálázhatatlanság. A klaszterszintű, kétlépcsős megoldás időkomplexitása ezzel szemben O(K·d) + O(n·(N/K)·d), ahol K a klaszterek száma, n pedig annak a száma, hogy a legjobb hány klaszterben keresünk. Ezzel a módszerrel nagyságrendekkel gyorsítható a keresés, és továbbra is 90-95% feletti pontosság érhető el. A FAISS HNSW egy közelítő keresést végez, amelynek az időkomplexitása átlagos esetben O(log N) [10], így lényegesen gyorsabb tud lenni, mint a brute force, és 90-99%-os pontosságot képes elérni.

5.3 Online klaszterezéssel történő retrieval kiértékelése

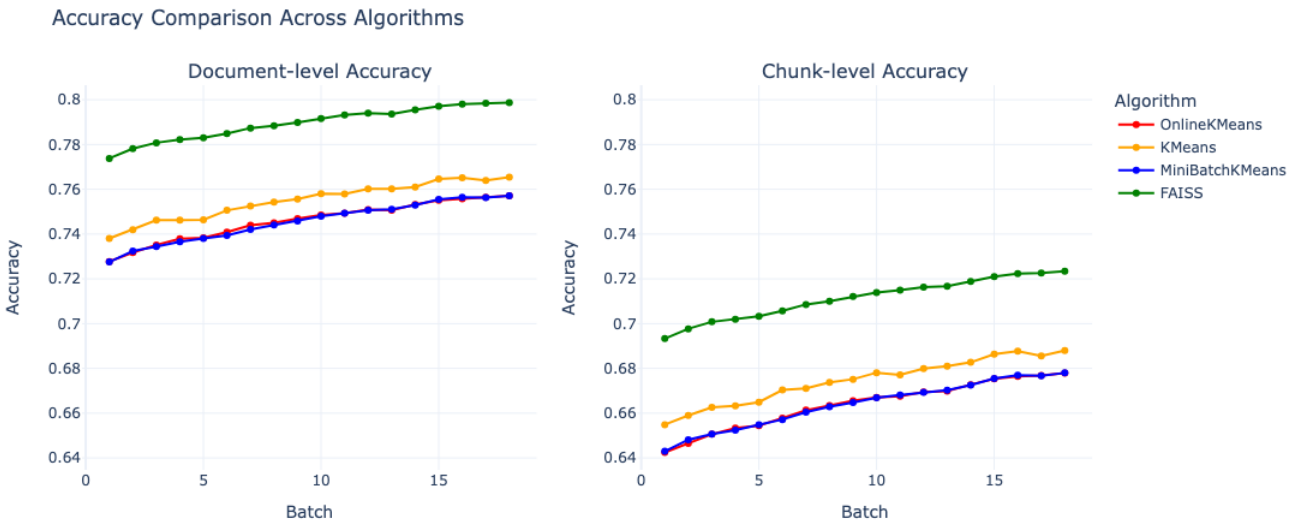
Az alábbi fejezetben az online klaszterezésen alapuló retrieval pipeline teljesítményének eredményeit mutatom be, és összehasonlítom azokat a MiniBatchKMeans, a klasszikus KMeans, illetve a FAISS index alapú algoritmusokkal elért eredményekkel. A cél annak vizsgálata volt, hogy az online klaszterezési módszer mennyiben képes hatékonyan és adaptívan kezelni a folyamatosan beérkező adatokat, illetve hogyan változik a visszakeresés pontossága és megbízhatósága a különböző klaszterezési/indexelési stratégiák alkalmazása mellett.

A kiértékelés során az előző alfejezetben ismertetett, olyan kérdéseket is tartalmazó adathalmazt használtam, amiknek a 15%-ára nincs az embeddingek között válasz. A szimuláció kezdetén a rendszer a teljes adathalmaz 50%-át használta a klaszterek inicializálásához, ezzel biztosítva, hogy mindhárom klaszterezési eljárás indulásakor már egy stabil klaszterszerkezet álljon rendelkezésre, és pontos eredményeket kapjak a MiniBatchKMeans működésére is, hiszen mint az kiderült ez az algoritmus különösen érzékeny arra, ha nem megfelelő mennyiségű adaton inicializálják a klasztereket. A kiindulási klaszterszám a klasszikus KMeans és MiniBatchKMeans esetén szintén 500 volt, míg az

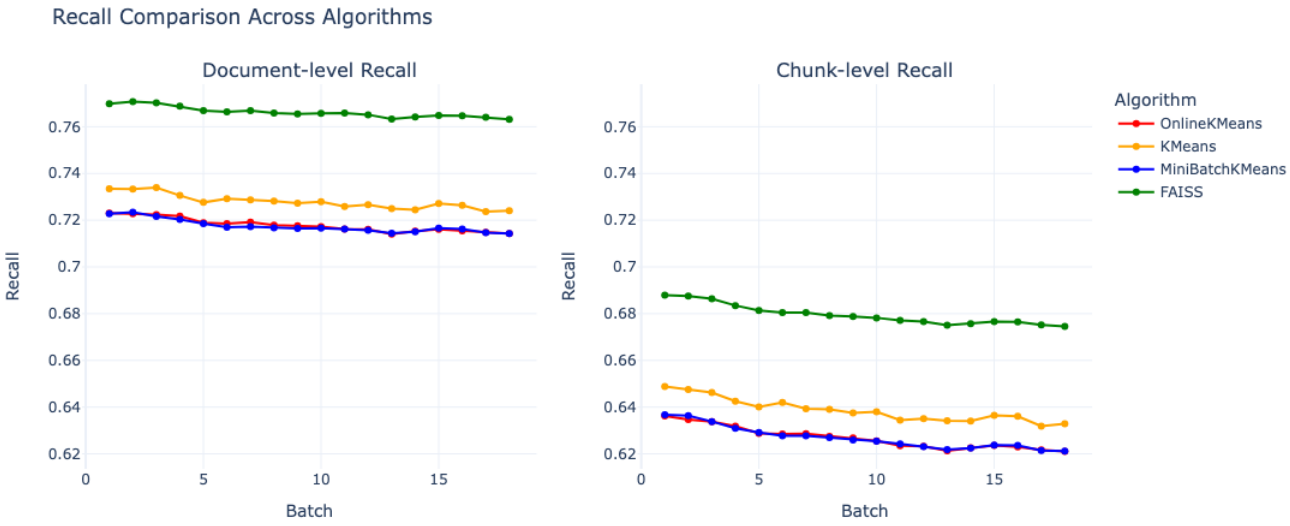
Online KMeans 360 klaszteren inicializáltam, hogy kihasználható legyen az a képessége, hogy létre tud hozni új klasztereket. Ezt követően a fennmaradó adatok 2000 elemű batchekben érkeztek be fokozatosan szimulálva egy valós idejű adatáramlási környezetet. Minden egyes batch beérkezése után lefuttattam a retrieval folyamatot, amelynek során a friss adatok között szerepeltek olyan kérdések, amelyekre biztosan létezett válasz az embeddingek között, valamint arányosan olyan kérdések is, amelyekre nem állt rendelkezésre megfelelő válasz.

Fontos kiemelni azt is, hogy mindhárom klaszterezést használó algoritmus esetében a retrieval a korábban ismertetett kétlépcsős megközelítéssel történt. Tehát először centroid-szinten, majd pedig a legjobb klasztereken belül kimerítően történt a hasonlóság-mérés. A szimuláció során a legjobb 10 klaszteren belül történt a keresés, és a legrelevánsabb 10 chunkot térítette vissza a modell. Ez a megközelítés lehetővé tette a különböző klaszterezési algoritmusok összehasonlítható, valós környezethez közeli értékelését, különös tekintettel az Online KMeans adaptív viselkedésére és az algoritmusok retrieval folyamatra gyakorolt hatására. A FAISS indexet használó megoldás értelemszerűen nem használt további klaszterezést, hanem egyből a legjobb 10 chunkot adta vissza.

Mind a négy algoritmussal lefuttattam a szimulációt, és közben számos metrikát mértem. Ebből a pontosság és fedés az a kettő mutató, ami a leginkább leírja a megoldásom jóságát. Mindkét metrikát dokumentum és chunk szinten is külön mértem, hogy átfogóbb képet kapjak arról, hogyan változnak az értékek az új adatok beérkezését követően. Mind a dokumentum szintű, mind a chunk szintű metrikák ugyanúgy kerültek kiszámításra, mint azt az előző alfejezetben bővebben is kifejtettem.



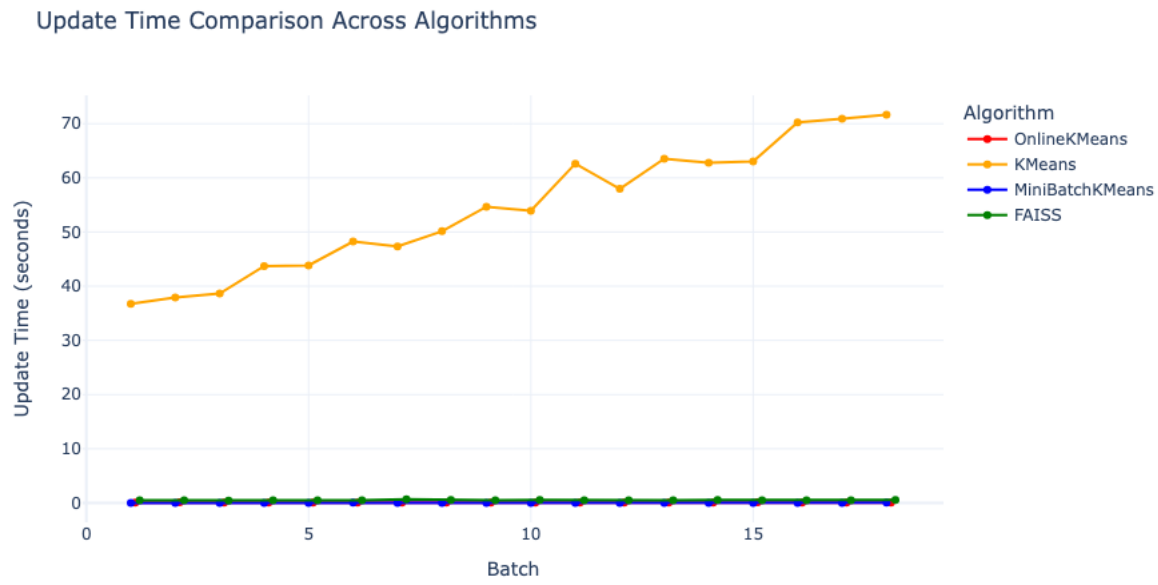
8. ábra - Az egyes algoritmusok által elért pontosság értékek változása a beérkező új batchek függvényében.



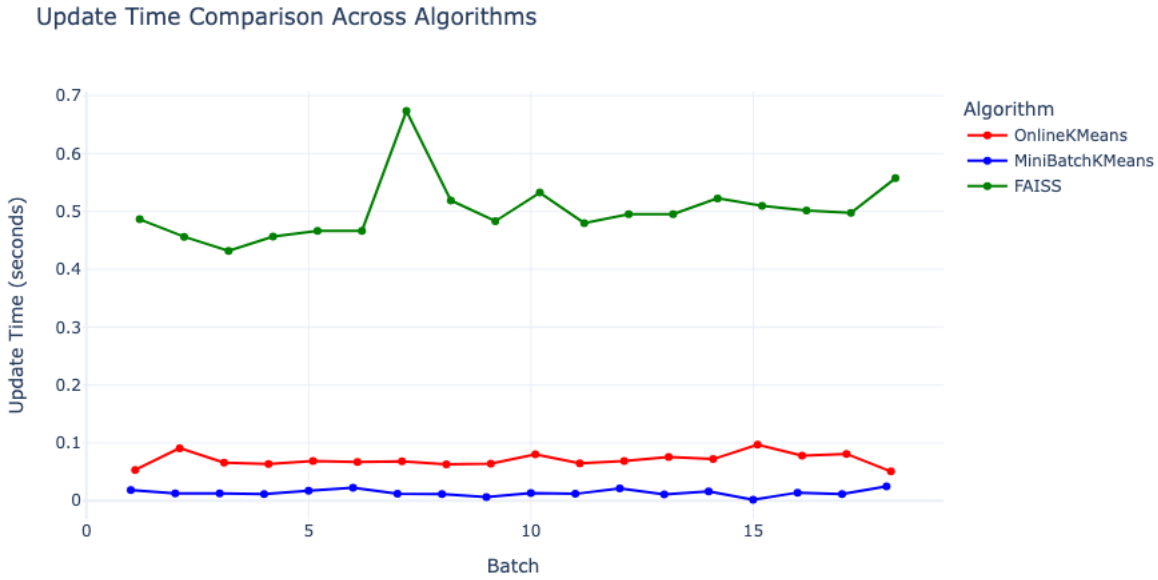
9. ábra - Az egyes algoritmusok által elért fedés értékek változása a beérkező új batchek függvényében.

Ahogy az a 8. és 9. ábrákon is jól látható, mindhárom klaszterező algoritmus nagyon hasonló eredményt ér el pontosság és fedés tekintetében is. A FAISS indexre épülő megoldás azonban mindkét metrikát nézve jóval pontosabb eredményt ér el. Ahogy az látható a recall értéke a beérkező adatok függvényében enyhén csökken, ami azt jelenti, hogy minél több adat van, annál nehezebben választja ki az összes valóban releváns elem közül a megfelelőeket. Viszont, ha azt vizsgáljuk, hogy a kiindulási állapothoz képest a szimuláció végére az adathalmaz mérete több mint a kétszeresére nőtt, a kevesebb mint 2%-os recall csökkenés egyáltalán nem mondható jelentősnek. Pontosság tekintetében ezzel szemben mind a négy algoritmus képes javulni az új adatok beérkezését követően. Ennek több oka is van. Egyfelől a retrieval során a 0,6-os határérték beállítása miatt ki lesz szűrve azon kérdések egyre nagyobb része, melyekre az embeddingek között nincs megfelelő válasz. Másfelől több olyan kérdés is beérkezik, amire az embeddingek között van válasz, és ezeket a válaszokat meg is találják a modellek.

Ahogy az várható volt a FAISS index alapú megoldással történő retrieval nyújtja a legjobb teljesítményt, hiszen ez az algoritmus nagy hatékonysággal képes a vektortérben tárolt dokumentumreprezentációk közötti hasonlóságokat kiszámítani. Ezt követően a KMeans algoritmus alapú megoldás következik, amely az egyik legjobb klaszterező eljárás. Őt pedig szinte azonos eredményekkel a MiniBatchKMeans és az Online KMeans követik, de ahogy az látható is az ábrákon az Online KMeans képes több batch esetén is jobb eredményt elérni, mint a MiniBatchKMeans. Továbbá mind a klasszikus KMeans, mind a MiniBatchKMeans rendelkezik olyan hátrányokkal, amivel az Online KMeans nem.



10. ábra - Az egyes algoritmusok frissítési idejének változása a beérkező új batchek függvényében.



11. ábra - Az egyes algoritmusok frissítési idejének változása a beérkező új batchek függvényében.
(10. ábra alsó része nagyítva)

Ahogy az a 10. és 11. ábrán is látható, a klasszikus KMeans algoritmus a MiniBatchKMeans-hoz, Online KMeans-hoz, illetve a FAISS indexhez képest is jelentősen nagyobb frissítési idővel rendelkezik. A frissítési idő azt mutatja meg, hogy mennyi időre van szüksége az algoritmusnak, hogy a beérkező adatokat besorolja egy klaszterbe és azoknak megfelelően frissítse a centroidokat. Ez a KMeans esetén teljes újraklaszterezést jelent, ami a klaszterszám és adatmennyiség függvényében jelentősen nő. Valós környezetben, ahol azt feltételezhetjük, hogy az adatok 2000-nél kisebb batchekben is érkezhetnek, és az összes chunk száma is nagyságrendekkel magasabb lehet, a klasszikus KMeans

nem jelent jó megoldást, ha valós időben szeretném frissíteni a centroidokat és klaszterbe sorolni az új embeddingeket.

Ezzel szemben a FAISS index rendkívül gyors és hatékony megoldást kínál az újonnan érkező beágyazási vektorok indexstruktúrához történő hozzárendelésére. Mind a MiniBatchKMeans, mind az Online KMeans algoritmus képes a beérkező adatokat valós időben, jellemzően egy másodpercen belül, a megfelelő klaszterhez rendelni, miközben az Online KMeans a centroidok pozícióját is dinamikusan frissíti. A frissítés időigénye kizárólag a batchben érkező adatok elemszámától függ, ugyanakkor az algoritmus nagy elemszámú batchek esetén is képes megőrizni a valós idejű feldolgozási képességét, és képes gyorsabb eredményt elérni a FAISS indexet használó megoldástól.

A fentebb ismertetett és részletesen bemutatott kiértékelések elkészítése során az adathalmaz előfeldolgozásához szemantikus chunkolási módszert alkalmaztam, továbbá az „all-MiniLM-L6-v2” SentenceTransformer modellt használtam az embeddingek előállítására. A szemantikus chunkolás alkalmazása mérnöki szempontból indokolt, mivel ez a megközelítés lehetővé teszi, hogy a szöveg egységei tartalmilag koherensek maradjanak, és az embeddingek így pontosabban tükrözzék a dokumentumok jelentésbeli szerkezetét. Ezzel szemben a hagyományos, rögzített méretű vagy sliding window alapú szeletelés gyakran figyelmen kívül hagyja a természetes szemantikai határokat, ami a leképezett vektortérben torzulásokhoz vezethet.

Noha jelen kutatás fókusza nem a chunkolási technikák összehasonlítására irányult, a jövőben érdemes lehet megvizsgálni, hogy a sliding window alapú megközelítés milyen hatással van a retrieval folyamat pontosságára, futási idejére és robusztusságára.

A vizsgálataim során kísérleteket végeztem magasabb dimenziójú embeddingekkel is annak érdekében, hogy feltárjam, miként befolyásolja a dimenziószám növelése a retrieval teljesítményét. Az eredmények azt mutatták, hogy a pontosság, fedés, futásidő és frissítési idő tekintetében hasonló tendenciák figyelhetők meg mindhárom megközelítés, a kétlépcsős keresés, a brute force módszer és a FAISS indexet alkalmazó megoldás esetében, mint a 384 dimenziós embeddingekkel, az „all-MiniLM-L6-v2” modellel végzett kísérletek során.

6 Összegzés és kitekintés

A Retrieval-Augmented Generation rendszerek a természetesnyelv-feldolgozás és a tudásalapú mesterséges intelligencia egyik legfontosabb fejlődési irányát képviselik, ahol a generatív és a visszakereső komponensek szoros integrációja valósul meg. A RAG modellek hatékonyságának és pontosságának kulcsa elsősorban a retrieval folyamat minőségében rejlik. A releváns információk azonosítása, a redundáns találatok szűrése és a keresési tér optimalizálása alapvető feltételei annak, hogy a generatív modell tényszerű, kontextushoz illeszkedő válaszokat tudjon előállítani.

A dolgozat a RAG modellek fejlesztésére és optimalizálására fókuszált, különös tekintettel a klaszterezési technikák alkalmazására az információ-visszakeresési folyamat hatékonyságának növelése érdekében. Ez a kutatási irány különösen jelentős, mivel a jelenlegi RAG-implementációk gyakran a teljes embedding-halmaz közvetlen felhasználásával végzik a hasonlósági keresést, ami redundanciát, pontatlanságot és jelentős számítási terhelést eredményezhet. A klaszterezési módszerek integrálása lehetőséget kínál e korlátok áthidalására, az embeddingek előzetes csoportosítása révén a keresés először klaszterszinten, majd azon belül finomítottan történhet, ami nemcsak a futásidőt csökkenti, hanem a visszakeresett találatok relevanciáját is növeli.

A kutatásom első felében bemutatásra került egy innovatív megközelítés embedding vektorok klaszterezésére online módon. Az általam készített Online KMeans képes felvenni a versenyt magas dimenziós embeddingek klaszterezése esetén olyan ismert klaszterező algoritmusokkal, mint a klasszikus KMeans vagy a MiniBatchKMeans. Ezt követően összehasonlítottam, hogyan teljesít a kétlépcsős (klaszter-szintű, majd klasztereken belüli) dokumentum visszakeresés az információ-visszakeresési folyamat során a state-of-the-art FAISS indexszel és a brute force megközelítéssel szemben. A FAISS indexre épülő megoldás bizonyult a vizsgált módszerek közül a leghatékonyabbnak, mivel egy korszerű, jól optimalizált algoritmusról van szó. Ugyanakkor a kétlépcsős algoritmus alkalmazása lehetővé tette a visszakeresési folyamat jelentős gyorsítását, miközben a pontosság csökkenése mindössze minimális mértékű volt a brute force megközelítéshez képest. Ennek eredményeként a kétlépcsős módszer egy olyan kompromisszumos, mégis hatékony alternatívát kínál, amely a számítási erőforrások optimalizálása mellett is képes magas szintű teljesítményt biztosítani a RAG-rendszerek retrieval komponensében.

A kutatásom második felében azt vizsgáltam meg, hogyan lehet effektíven integrálni az újonnan érkező dokumentumokat a retrieval folyamatba a pontosság és hatékonyság jelentős csökkenése nélkül. Ezesetben az Online KMeans algoritmust a klasszikus KMeans, a MiniBatchKMeans és a FAISS indexszel hasonlítottam össze. Ahogy az a kutatás első felében is kiderült, az Online KMeans képes jobb eredményeket elérni a MiniBatchKMeans-szal szemben, és képes sokkal gyorsabban integrálni az új dokumentumokat a klaszterek közé, mint a klasszikus KMeans algoritmus. Ezzel egy olyan alternatívát

nyújt a FAISS index mellé, amely képes hatékonyabbá tenni a dokumentum-visszakeresést, és képes egy robusztus megoldást kínálni arra az esetre, ha a korábbiakhoz képest teljesen eltérő dokumentum típusok kerülnek az embeddingek közé azáltal, hogy dinamikusan képes új klasztereket létrehozni és meglévőket összevonni.

A projekt megvalósítása során nemcsak új eredmények és módszerek születtek, hanem a kutatással kapcsolatban további kérdések is megfogalmazódtak. Ilyen kérdés például a state-of-the-art Databricks Vector Search [3] megoldás alkalmazhatósága dokumentumfolyamok kezelésére, valamint ennek lehetséges integrálása az online klaszterezési megközelítéssel. Emellett az Online KMeans algoritmus további optimalizálása is releváns kutatási irányt jelenthet.

Irodalomjegyzék

[1] Alexandr Andoni, Piotr Indyk, Ilya Razenshteyn, (2018.06.26.), Approximate Nearest Neighbor Search in High Dimensions, ArXiv, arXiv:1806.09823

[2] Ashour, Wesam & Fyfe, Colin. (2008). Online clustering algorithms. International journal of neural systems. 18. 185-94. https://www.doi.org/10.1142/S0129065708001518

[3] Databricks, Vector Search, https://www.databricks.com/product/machine-learning/vector-search, (utolsó megnyitás: 2025.10.23.)

[4] Fahimsadikrashad, (2024.01.08.), Mini Batch K Means : An Approach for Refining Traditional Methods, Medium, https://medium.com/@fahimsadikrashad/mini-batch-k-means-an-approach-for-refining-traditional-methods-bd7716ac8bcb

[5] Hoang-Anh Ngo, (2022 september), ONLINE CLUSTERING: ALGORITHMS, EVALUATION, METRICS, CHALLENGES, APPLICATIONS AND BENCHMARKING WITH RIVER, @Télécom Paris, IP Paris, The University of Waikato, Artificial Intelligence Institiut, https://a3nm.net/work/seminar/slides/20220927-ngo.pdf

[6] Lance Galletti, (2024.02.22.), Kmeans ++ From Scratch, Medium, https://medium.com/@gallettilance/kmeans-from-scratch-24be6bee8021

[7] Li Juanzi, Hu Linmei, Ouyang Tinghui, Alkawsi Gamal Abdulnaser, (2006), Online Clustering, ScienceDirect, https://www.sciencedirect.com/topics/computer-science/online-clustering

[8] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, Hervé Jégou, (2024.01.16.), The Faiss library, ArXiv, arXiv:2401.08281

[9] NLTK Project, 2023, nltk.tokenize.sent_tokenize, https://www.nltk.org/api/nltk.tokenize.sent_tokenize.html, (utolsó megnyitás: 2025.10.26.)

[10] Owen Elliott, (2024.06.04.), Understanding Recall in HNSW Search, marqo, https://www.marqo.ai/blog/understanding-recall-in-hnsw-search

[11] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, Douwe Kiela, (2020.05.22.), Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks, ArXiv, https://arxiv.org/abs/2005.11401

[12] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, Percy Liang, (2016.06.16.),SQuAD: 100,000+ Questions for Machine Comprehension of Text, ArXiv, arXiv:1606.05250

[13] Puxuan Yu, Luke Merrick, Gaurav Nuti, Daniel Campos, (2024.12.04.), Snowflake's Arctic Embed 2.0 Goes Multilingual: Empowering Global-Scale Retrieval with Inference Efficiency and High-Quality Retrieval, snowflake, https://www.snowflake.com/en/engineering-blog/snowflake-arctic-embed-2-multilingual/

[14] Reimers, Nils and Gurevych, Iryna, (2019. november), Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks, Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing, https://arxiv.org/abs/1908.10084

[15] scikit-learn.org, adjusted_rand_score, https://scikit-learn.org/stable/modules/generated/sklearn.metrics.adjusted_rand_score.html, (utolsó megnyitás: 2025.10.25.)

[16] scikit-learn.org, KMeans, https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html, (utolsó megnyitás: 2025. 10. 19.)

[17] scikit-learn.org, make_blobs, https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html, (utolsó megnyitás: 2025.10.24.)

[18] scikit-learn.org, MiniBatchKMeans, https://scikit-learn.org/stable/modules/generated/sklearn.cluster.MiniBatchKMeans.html#sklearn.cluster.MiniBatchKMeans, (utolsó megnyitás: 2025. 10. 19.)

[19] scikit-learn.org, normalized_mutual_info_score, https://scikit-learn.org/stable/modules/generated/sklearn.metrics.normalized_mutual_info_score.html, (utolsó megnyitás: 2025.10.25.)

[20] Shailja Gupta (Carnegie Mellon University, USA) Rajesh Ranjan (Carnegie Mellon University, USA) Surya Narayan Singh (BIT Sindri, India), (2024.10.03.), A Comprehensive Survey of Retrieval-Augmented Generation (RAG): Evolution, Current Landscape and Future Directions, ArXiv, https://arxiv.org/pdf/2410.12837

[21] Shanmukha Ranganath, (2024.10.05.), RAG 101: Chunking Strategies, towardsdatascience.com, https://towardsdatascience.com/rag-101-chunking-strategies-fdc6f6c2aaec/

[22] Silva, J. A., Faria, E. R., Barros, R. C., Hruschka, E. R., de Carvalho, A. C. P. L. F., and Gama, J. 2013. Data stream clustering: A survey. ACM Comput. Surv. 46, 1, Article 13 (October 2013), 31 pages. DOI: http://dx.doi.org/10.1145/2522968.2522981

[23] The Devs from Zilliz, (2025), How do Sentence Transformers relate to large language models like GPT, and are Sentence Transformer models typically smaller or more specialized?, Milvus, https://milvus.io/ai-quick-reference/how-do-sentence-transformers-relate-to-large-language-models-like-gpt-and-are-sentence-transformer-models-typically-smaller-or-more-specialized

[24] The Devs from Zilliz, (2025), How do you handle encoding very long documents with Sentence Transformers (for example, by splitting the text into smaller chunks or using a sliding window approach)?, Milvus, https://milvus.io/ai-quick-reference/how-do-you-handle-encoding-very-long-documents-with-sentence-transformers-for-example-by-splitting-the-text-into-smaller-chunks-or-using-a-sliding-window-approach

[25] Varun, (2020.09.27.), Cosine similarity: How does it measure the similarity, Maths behind and usage in Python, towardsdatascience.com ,https://towardsdatascience.com/cosine-similarity-how-does-it-measure-the-similarity-maths-behind-and-usage-in-python-50ad30aad7db/

[26] Vinod Chugani, (2024.07.17.), What is Manhattan Distance?, datacamp, https://www.datacamp.com/tutorial/manhattan-distance?dc_referrer=https%3A%2F%2Fwww.google.com%2F&utm_source=google&utm_medium=paid_search&utm_campaign=230119_1-sea%7Edsa%7Etofu_2-b2c_3-row-p2_4-prc_5-na_6-na_7-le_8-pdsh-go_9-nb-e_10-na_11-na-oct24

[27] Yu. A. Malkov, D. A. Yashunin, (2018.08.14.), Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs, ArXiv, arXiv:1603.09320

[28] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, Haofen Wang, (2023.12.18.), Retrieval-Augmented Generation for Large Language Models: A Survey, ArXiv, https://arxiv.org/abs/2312.10997

Függelék

A dolgozatban bemutatott kutatáshoz tartozó forráskód és kiegészítő fájlok elérhetőek a következő GitHub-linken: https://github.com/czotterbenedek/szakdoga_tdk

Az embedding tömbök és a nyers adatfájlok azonban nem találhatóak meg a repozitóriumban, mivel méretük meghaladta a GitHub által engedélyezett korlátot.