Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп’ютерної інженерії та управління
(повна назва)
Кафедра Комп'ютерних інтелектуальних технологій та систем
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

Інтелектуальна система нечіткого пошуку адрес у базі даних

Виконав:
здобувач IV року навчання,
групи КІУКІ-21-10
Денис ПОСТОЛЬНИЙ
(власне ім'я, прізвище)
Спеціальність 123 Комп’ютерна інженерія
(код і повна назва спеціальності)
Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)
Освітня програма Комп'ютерна інженерія
(повна назва освітньої програми)
Керівник проф. каф. КІТС Наталія АКСАК
(посада, власне ім'я, прізвище)

Допускається до захисту

Зав. кафедри Олег РУДЕНКО

2025 р.

Харківський національний університет радіоелектроніки

Факультет	Комп'ютерної інженерії та управління
Кафедра	Комп'ютерних інтелектуальних технологій та систем
Рівень вищої освіти	перший (бакалаврський)
Спеціальність	123 Комп'ютерна інженерія
Тип програми	освітньо-професійна
Освітня програма	Комп'ютерна інженерія

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«_____»_____202_ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Постольному Денису Олексійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Інтелектуальна система нечіткого пошуку адрес у базі даних

затверджена наказом по університету від " 21 " травня 2025 р. № 399СТ

2. Термін подання здобувачем роботи до екзаменаційної комісії 14.06.2025

3. Вхідні дані до роботи

Адресні довідники Укрпошти
Користувацькі запити з помилками
Нормативні та технічні вимоги
Гібридна метрика подібності
Технологічне середовище

4. Перелік питань, що потрібно опрацювати в роботі

Підходи до нечіткого пошуку адрес і в чому їхні обмеження
Фільтрування великого обсягу адрес
Модель семантичного зіставлення адрес
Фільтр і семантичний пошук
Тестування системи на реальних прикладах

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) 11 слайдів___
_____
_____
_____
_____
_____

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Огляд і аналіз сучасного стану розглянутої проблеми, а також існуючих методів і засобів вирішення задач кваліфікаційної роботи	26.05.2025 – 31.05.2025	Виконано
2	Проектування архітектури додатку	01.06.2025 – 03.06.2025	Виконано
3	Створення додатку	04.06.2025 – 07.06.2025	Виконано
4	Аналіз та налагодження роботи додатку	08.06.2025 – 09.06.2025	Виконано
5	Оформлення матеріалів кваліфікаційної роботи	10.06.2025 – 14.06.2025	Виконано

Дата видачі завдання 26.05.2025

Здобувач _____
(підпис)

Керівник роботи _____ Проф. Аксак Н.Г.
(підпис) (посада, ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 62 с., 14 рис., 1 табл., 1 дод., 10 джерел.

НЕЧІТКИЙ ПОШУК, АДРЕСА, JARO-WINKLER, СЕМАНТИЧНИЙ ПОШУК, WEAVIATE, REST API, АВТОЗАПОВНЕННЯ, НОРМАЛІЗАЦІЯ, ВЕКТОРНА БАЗА ДАНИХ, FASTAPI

Кваліфікаційну роботу присвячено розробці інтелектуальної системи нечіткого пошуку адрес з використанням гібридного підходу: лексичних метрик Jaro-Winkler та векторного семантичного пошуку на базі Weaviate. Актуальність теми зумовлена потребою в підвищенні якості та швидкодії автозаповнення адресних форм у CRM-, ERP-, логістичних та державних системах, особливо в умовах воєнного стану, зростання внутрішньої міграції та цифровізації державних сервісів.

Розроблена система дозволяє попередньо фільтрувати результати в реляційній базі даних, використовує семантичне зіставлення векторних представлень адрес, реалізує REST API з повним циклом обробки запитів. Проведено експериментальне оцінювання точності та латентності на реальних користувацьких запитах.

Запропоновано структуру, що поєднує SQL Server, Weaviate, Python FastAPI та модуль імпорту даних Укрпошти. Результати показали точність зіставлення понад 95 % та середню затримку обробки запиту < 150 мс.

Робота має наукову новизну в інтеграції класичних та сучасних методів обробки адрес та може бути впроваджена як сервіс у сучасні інформаційні системи

ABSTRACT

Explanatory note of qualification work 62 pages, 14 figures, 1 tables, 1 appendices, 10 sources.

FUZZY SEARCH, ADDRESS, JARO-WINKLER, SEMANTIC SEARCH, WEAVIATE, REST API, AUTOCOMPLETE, NORMALIZATION, VECTOR DATABASE, FASTAPI

The thesis focuses on the development of an intelligent fuzzy address search system using a hybrid approach that combines lexical similarity metrics (Jaro-Winkler) and vector-based semantic search in Weaviate. The relevance of the topic lies in the increasing need for accurate and fast address autocomplete in CRM, ERP, logistics, and e-government systems, especially under conditions of wartime migration and digital transformation.

The proposed solution applies pre-filtering using SQL-based similarity functions, followed by semantic ranking using sentence embeddings. A RESTful API handles the end-to-end query lifecycle. The architecture integrates SQL Server, Weaviate, Python FastAPI, and address data import from Ukrposhta.

Experimental results confirm high accuracy (>95%) and low latency (<150 ms) on real-world noisy address queries. The system demonstrates the effectiveness of combining traditional and semantic models and is suitable for deployment in commercial and public digital services.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

API – Application Programming Interface – програмний інтерфейс застосунків

REST – Representational State Transfer – архітектурний стиль веб-сервісів

SQL – Structured Query Language – мова структурованих запитів

CRM – Customer Relationship Management – система керування взаєминами з клієнтами

ERP – Enterprise Resource Planning – система планування ресурсів підприємства

CO_2 – Вуглекислий газ

PII – Personally Identifiable Information – персонально-ідентифікована інформація

GDPR – General Data Protection Regulation – Загальний регламент захисту даних (ЄС)

ВСТУП

У сучасному світі база адрес відіграє ключову роль у багатьох сферах діяльності. Незалежно від того, чи йдеться про доставку товарів, надання державних послуг, або маркетингові кампанії, точне і ефективне опрацювання адрес є критично важливим. У базах даних адреси зазвичай зберігаються в певному форматі, який забезпечує їх однозначне ідентифікування. Проте при введенні адрес користувачами можуть виникати численні проблеми, пов'язані з варіаціями у написанні, орфографічними помилками, неповнотою або некоректністю даних.

Коли користувач вводить адресу, вона може значно відрізнятися від стандартного формату, збереженого в базі даних. Наприклад, одна і та сама адреса може бути написана по-різному, як "вул. Лесі Українки" і "вулиця Лесі Українки", або мати різні орфографічні помилки. Такі відмінності значно ускладнюють процес пошуку та зіставлення адрес в базі даних, що призводить до зниження ефективності та точності роботи систем.

Для вирішення цієї проблеми можна використовувати штучні нейронні мережі, які здатні навчатися на прикладах і знаходити відповідності між різними варіантами написання адрес. Використання штучних нейронних мереж у цій сфері дозволяє автоматизувати процес пошуку і співставлення адрес, знижуючи ризик помилок і підвищуючи ефективність роботи систем. Це забезпечує більш високу якість послуг для користувачів і сприяє економії ресурсів для організацій, що використовують ці дані.

Об’єкт дослідження — процес інтелектуального пошуку та зіставлення адрес у реляційних і векторних базах даних.

Предмет дослідження — алгоритми й архітектурні рішення гібридної системи, що поєднує лексичні метрики (Jaro-Winkler), векторні представлення та мовні моделі для нечіткого порівняння записів.

Метою роботи є розроблення та експериментальна перевірка інтелектуальної системи нечіткого пошуку адрес, здатної у режимі реального часу:

– нормалізувати користувацьке введення;

– відсіювати нерелевантні кандидати за допомогою швидких лексичних метрик;

– проводити семантичний пошук у векторній БД Weaviate;

– повертати єдиний, найбільш ймовірний результат із точністю ≥ 95 % при латентності ≤ 150 мс.

Для досягнення мети розв’язано такі основні завдання:

– провести огляд існуючих методів адресного зіставлення та визначити їхні обмеження;

– обґрунтувати вибір гібридного підходу та побудувати загальну архітектуру системи;

– створити реляційну БД міст/вулиць і підготувати імпорт відкритих даних Укрпошти;

– реалізувати функції Jaro-Winkler у SQL Server для попереднього фільтру;

– розгорнути Weaviate у Docker-середовищі та налаштувати мовну модель для створення ембеддингів;

– спроєктувати REST API, що інкапсулює повний конвеєр пошуку;

– провести експериментальне оцінювання точності та часу відповіді в реальній вибірці помилкових запитів.

Наукова новизна полягає у синергетичному поєднанні класичної лексичної метрики й семантичного пошуку у векторному просторі, що забезпечує значне скорочення латентності без втрати якості порівняно з «чистими» LLM-підходами. Запропоновано також евристичну схему вагового ранжування, яка враховує регіональний контекст (місто/район) та частоту використання адресних шаблонів.

Практичне значення роботи підтверджується можливістю інтеграції запропонованої служби як мікросервісу у типові CRM-, ERP- та логістичні платформи: це знижує витрати на ручну валідацію записів до 70 % і підвищує коефіцієнт успішного автозаповнення форм на 15–20 %.

Методи дослідження. У роботі використано методи статистичного аналізу текстових даних, алгоритм Jaro-Winkler для лексичної схожості, техніку векторного

семантичного пошуку (Hybrid-Search) у Weaviate, а також експериментальне оцінювання продуктивності на вибірках реальних помилок користувачів.

Структура кваліфікаційної роботи. Пояснювальна записка складається зі вступу, чотирьох розділів, висновків, списку з NN джерел і додатків із графічним матеріалом.

У розділі 1 обґрунтовано актуальність та проаналізовано існуючі підходи.

У розділі 2 наведено вибір і теоретичне обґрунтування архітектури моделі.

У розділі 3 описано розробку інтелектуальної системи та взаємодію її компонентів.

У розділі 4 подано деталі реалізації, результати випробувань і REST-інтерфейсу.

Таким чином, виконана робота спрямована на підвищення якісних і економічних показників пошуку адрес та має перспективи подальшого розвитку в напрямах мультимовної підтримки й автоматичного виправлення поширених помилок користувачів.

1 АКТУАЛЬНІСТЬ ЗАДАЧІ ТА АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Актуальність теми

Точний пошук адрес в базах даних є надзвичайно важливим для багатьох галузей. У сучасному світі, де інформаційні технології швидко розвиваються, бази даних часто містять величезні обсяги адресної інформації, яку потрібно ефективно обробляти. Це особливо актуально для таких сфер, як логістика, маркетинг, державні служби та інші.

У логістиці точний пошук адрес є критичним для забезпечення своєчасної та правильної доставки товарів. Невірні або некоректно введені адреси можуть призвести до затримок, додаткових витрат на транспортування та незадоволення клієнтів. Компанії витрачають значні ресурси на виправлення помилок у адресах, що могло б бути уникнено за допомогою автоматизованих систем перевірки та корекції адрес.

Для маркетингових кампаній точні адреси є основою для сегментації аудиторії та таргетованої реклами. Якщо адреси клієнтів введені неправильно, це може призвести до того, що рекламні матеріали будуть відправлені не за тією адресою, що знижує ефективність кампаній та збільшує витрати. Крім того, точні адресні дані дозволяють проводити більш глибокий аналіз ринку та розробляти стратегії розвитку бізнесу.

У державних службах точний облік адрес є необхідним для надання якісних адміністративних послуг. Від правильності введення адрес залежить ефективність роботи служб, таких як реєстрація місця проживання, видача паспортів та інші. Крім того, екстрені служби, такі як швидка допомога, пожежна служба та поліція, потребують точних адрес для швидкого реагування на виклики.

Проте, при ручному введенні адрес виникають численні проблеми. Помилки введення, такі як орфографічні помилки, неповні або некоректні дані, можуть значно ускладнити пошук потрібної адреси. Різні стандарти написання адрес, використання

скорочень та варіацій у написанні можуть призвести до плутанини і неправильного зіставлення адрес. Людський фактор також відіграє роль, оскільки різні користувачі можуть вводити одну й ту саму адресу по-різному.

Актуальність дослідження значно виходить за рамки «класичного» кейсу електронної комерції й охоплює економічні, соціальні, безпекові та екологічні виклики.

По-перше, український e-commerce демонструє вибухове зростання: лише у 2024 році споживачі витратили в онлайні ≈ 239 млрд ₴, що на 25 % більше, ніж торік. Водночас до чверті посилок не доходять із першої спроби через неточні або неуніфіковані адреси — це визнають 74 % опитаних компаній-ретейлерів, і кожен такий збій прямо конвертується у додаткові витрати на повторну доставку та втрату лояльності клієнтів.

По-друге, цифрові держсервіси («Дія», Єдині центри надання адмінпослуг) покладаються на коректність адресних атрибутів для верифікації особи та формування довідок. Масові міграції всередині країни (≈ 3,6 млн внутрішньо переміщених осіб станом на кінець 2024 р.) потребують швидкого перенесення й актуалізації даних, аби громадяни без затримки отримували соціальні виплати і публічні послуги.

По-третє, з позиції кібербезпеки проблема має критичний масштаб: 46 % порушень, зафіксованих у 2024 р., містили адресну PII, а середня вартість такої утрати даних сягнула $4,88 млн. Якісна автоматична валідація та анонімізація адрес стає обов’язковою для дотримання GDPR та українського закону «Про захист персональних даних».

По-четверте, тема має чіткий екологічний вимір: кожен повторний виїзд кур’єра збільшує вуглецевий слід ланцюжка доставки; дослідження останньої милі показують, що навіть додаткові 0,01 кг CO_2 на посилку множаться до сотень тонн за рік для великих маркетплейсів. Усунення адресних помилок напряму сприяє ESG-цілям компаній.

По-п’яте, розумні міста та IoT-інфраструктура — від лічильників до дронів-кур’єрів — потребують точних геоідентифікаторів і низьколатентних API

автодоповнення адрес у реальному часі, адже помилка в кілька символів може призвести до збоїв автономної навігації.

Нарешті, Україна дедалі активніше переходить на латинку у міжнародних документах, тому система має коректно зіставляти «вул. Шевченка, 10» і «10 Shevchenka St.», що підсилює потребу в гібридних лексичних і семантичних алгоритмах.

Таким чином, вирішення задачі нечіткого пошуку й нормалізації адрес одночасно відповідає економічним, соціальним, безпековим, екологічним та технологічним потребам сучасного суспільства й робить тему дослідження надзвичайно актуальною.

Ці проблеми роблять актуальним використання сучасних технологій для автоматизації процесу пошуку та корекції адрес. Штучні нейронні мережі, які здатні навчатися на прикладах і розпізнавати патерни в даних, можуть значно підвищити точність і ефективність обробки адресної інформації.

1.2 Огляд існуючих рішень. Їх переваги та недоліки

Розглянемо основні варіанти рішень для реалізації пошуку адрес у базі даних:

- Штучні нейронні мережі (ШНМ).
- Векторні представлення.
- Алгоритми машинного навчання.
- Гібридні підходи.

1.1.1 Штучні нейронні мережі (ШНМ)

Штучні нейронні мережі (ШНМ) є моделями машинного навчання, натхненими біологічними нейронними мережами мозку. Вони складаються з множини штучних нейронів, організованих у шари: вхідний шар, приховані шари та вихідний шар. Кожен нейрон обробляє вхідні дані і передає результати далі через ваги, що коригуються під час навчання. Завдяки здатності навчатися на великих обсягах даних,

ШНМ можуть розпізнавати складні патерни і взаємозв'язки в даних, що робить їх потужним інструментом для різних завдань, включаючи пошук і обробку адрес.

Серед переваг штучних нейронних мереж можна відмітити їх високу точність у задачах розпізнавання патернів, що досягається завдяки здатності навчатися на великих обсягах даних. Вони є гнучкими та універсальними, оскільки можуть застосовуватися до різноманітних типів задач — від класифікації та регресії до обробки зображень і природної мови. Важливою особливістю є також здатність ШНМ автоматично виділяти найбільш інформативні ознаки з даних, що зменшує потребу в ручному створенні ознакових просторов. Крім того, ці моделі добре масштабуються, що дозволяє ефективно використовувати їх у промислових умовах з великими базами даних.

Серед недоліків штучних нейронних мереж варто зазначити їхні високі обчислювальні витрати, оскільки навчання, особливо глибоких моделей із великою кількістю параметрів, потребує значних ресурсів і часу. Крім того, для досягнення високої точності такі моделі зазвичай вимагають великих обсягів навчальних даних, що не завжди можливо забезпечити. Ще однією складністю є чутливість до налаштувань: оптимізація гіперпараметрів є непростим завданням і вимагає досвіду, адже некоректні налаштування можуть значно знизити ефективність моделі. Також, ШНМ мають обмежену інтерпретованість — часто складно пояснити, як саме система приймає рішення, що є критичним у задачах, де потрібна прозорість і довіра до моделі.

1.2.2 Векторні представлення

Векторні представлення — це спосіб перетворення тексту в числові формати, які можуть бути оброблені алгоритмами машинного навчання. Векторизація тексту дозволяє кодувати семантичну інформацію про слова та їх контекст у вигляді векторів фіксованої довжини. Одні з найпопулярніших методів векторизації включають Word2Vec, GloVe, FastText та сучасні контекстні векторні представлення, такі як BERT і GPT. Ці методи навчаються на великих текстових корпусах і створюють

вектори, де семантично схожі слова мають близькі вектори.

Серед переваг векторних представлень варто відзначити їхню здатність відображати семантичну близькість між словами, що дозволяє ефективно зіставляти різні варіанти написання однієї й тієї ж адреси або виправляти орфографічні помилки. Векторизація також забезпечує зменшення розмірності, перетворюючи текст у вектори фіксованої довжини, що спрощує обробку та зберігання даних. Завдяки своїй універсальності такі представлення застосовуються в різних задачах обробки тексту, зокрема класифікації, кластеризації, пошуку та зіставленні. Крім того, після навчання модель векторизації може оперативно кодувати нові слова або фрази, що забезпечує високу швидкість і ефективність при роботі з великими обсягами даних.

Серед недоліків векторних представлень варто зазначити їхню обмежену здатність враховувати контекст: більшість класичних підходів, як-от Word2Vec чи GloVe, не розрізняють значення полісемантичних слів у різних ситуаціях, що може призводити до неоднозначностей. Крім того, створення якісних векторних моделей вимагає попереднього навчання на великих текстових корпусах, що є тривалим і ресурсомістким процесом. Ще однією проблемою є складність у роботі з рідкісними або новими словами, які можуть бути відсутні в навчальних даних. Нарешті, хоча такі представлення і фіксують семантичну схожість, самі вектори залишаються числовими абстракціями, що ускладнює їхню інтерпретацію.

1.2.3 Алгоритми машинного навчання

Алгоритми машинного навчання (ML) дозволяють комп'ютерним системам навчатися і робити прогнози або приймати рішення без явного програмування. Вони використовують історичні дані для побудови моделей, які можуть передбачати результати на нових, невідомих даних. До основних категорій алгоритмів машинного навчання належать методи наглядуваного навчання (наприклад, регресія, дерева рішень, SVM, нейронні мережі), ненаглядного навчання (кластеризація, метод головних компонент) і методи навчання з підкріпленням.

Серед переваг алгоритмів машинного навчання можна відзначити їх здатність

автоматизувати обробку великих обсягів даних, що суттєво знижує потребу в ручній роботі та підвищує загальну ефективність. Важливою особливістю є адаптивність: моделі можуть навчатися на нових даних і таким чином покращувати точність своїх передбачень з часом. До того ж, ці алгоритми мають широкий спектр застосувань — від класифікації і регресії до кластеризації, виявлення аномалій та інших складних завдань. Завдяки своїй здатності виявляти приховані закономірності й взаємозв’язки у даних, машинне навчання відкриває нові можливості для глибшого аналізу й прийняття обґрунтованих рішень.

Серед недоліків алгоритмів машинного навчання варто відзначити їхню потребу у великих обсягах даних для досягнення високої точності та надійності, що не завжди можливо забезпечити. Процес навчання моделей часто вимагає значних обчислювальних ресурсів і часу, що може стати серйозною перепоною для впровадження у практичних умовах. Крім того, налаштування гіперпараметрів і вибір оптимальних алгоритмів є складним завданням, що потребує глибоких експертних знань. Ще однією проблемою є перенавчання, коли модель демонструє високі результати на тренувальних даних, але не здатна узагальнювати знання на нових, невідомих прикладах.

1.2.4 Гібридні підходи

Гібридні підходи в машинному навчанні комбінують різні методи та алгоритми для досягнення кращих результатів. Це може включати поєднання традиційних алгоритмів машинного навчання, векторних представлень і штучних нейронних мереж. Основна ідея полягає в тому, щоб використовувати сильні сторони кожного з методів, компенсуючи їхні недоліки. Наприклад, можна комбінувати векторні представлення тексту з моделями глибокого навчання для покращення точності обробки текстових даних.

Серед переваг гібридних моделей можна відзначити їхню здатність покращувати точність завдяки поєднанню переваг кількох підходів: векторні представлення надають семантичну інформацію, а нейронні мережі ефективно

враховують контекст. Такий синергетичний ефект дозволяє отримати кращі результати, ніж при використанні кожного методу окремо. Крім того, гібридні підходи вирізняються гнучкістю і універсальністю, що дає змогу застосовувати їх до різноманітних задач і типів даних. Вони також здатні компенсувати слабкі сторони окремих методів, наприклад, подолати обмеження контекстної нечутливості векторних представлень або зменшити обчислювальні витрати, характерні для нейронних мереж.

Серед недоліків гібридних моделей варто відзначити їхню складність в інтеграції, оскільки поєднання різних методів вимагає ретельного налаштування і скоординованої роботи компонентів. Такий підхід часто супроводжується підвищеними обчислювальними витратами, адже обробка даних і навчання моделі потребують більше ресурсів і часу. Крім того, розробка гібридних рішень вимагає глибоких експертних знань у кількох напрямах машинного навчання, що може бути складно реалізувати в невеликих командах або для початківців. Додаткову складність становить налаштування гіперпараметрів, оскільки кожен компонент такої моделі має власні параметри, які потребують оптимізації.

Усі розглянуті методи мають свої сильні і слабкі сторони, і їх вибір залежить від конкретних потреб і обмежень задачі. Штучні нейронні мережі забезпечують високу точність і автоматизацію, але потребують значних ресурсів. Векторні представлення дозволяють ефективно кодувати семантичну інформацію, але можуть мати проблеми з контекстом. Алгоритми машинного навчання забезпечують широку адаптивність і автоматизацію, але вимагають великих обсягів даних. Гібридні підходи, поєднуючи різні методи, пропонують найбільш гнучкі і точні рішення, але складність їх реалізації може бути значною.

2 ВИБІР СТРУКТУРИ МОДЕЛІ

2.1 Фактори вибору структури та архітектури моделі

Вибір структури та архітектури моделі є ключовим етапом у розробці системи для точного пошуку адрес у базах даних. Це рішення визначає, наскільки ефективною та точною буде модель у виконанні своїх завдань. Основними факторами, що впливають на вибір структури та архітектури моделі, є такі:

1) природа даних:

- тип даних: характер даних, що використовуються для навчання моделі, суттєво впливає на вибір архітектури. Для текстових даних, таких як адреси, підходять моделі, здатні обробляти послідовності символів або слів;

- обсяг даних: великий обсяг даних дозволяє використовувати складні моделі, тоді як для невеликих наборів даних можуть бути ефективні простіші підходи;

2) цілі задачі:

- точність пошуку: висока точність є критичною для задачі пошуку адрес, оскільки навіть невеликі помилки можуть призвести до значних проблем у логістиці, маркетингу або державних службах;

- швидкість обробки: для великих баз даних важлива також швидкість обробки запитів, що може вимагати оптимізації моделей;

3) обчислювальні ресурси:

- доступні ресурси: вибір моделі залежить від наявних обчислювальних ресурсів. Глибокі нейронні мережі можуть вимагати потужних графічних процесорів (GPU) або навіть спеціалізованих апаратних рішень;

- масштабованість: модель повинна бути здатною масштабуватися з ростом обсягів даних та запитів, що обробляються;

4) особливості завданнянаявність шуму:

- модель повинна бути стійкою до шуму в даних, такого як орфографічні помилки або варіації у форматуванні адрес;

- прив'язка до населеного пункту: для коректного пошуку адреси важливо враховувати населений пункт, оскільки однакові або схожі назви вулиць можуть існувати в різних містах або селах;

Ретельний аналіз цих факторів дозволяє вибрати оптимальну модель, яка забезпечить високу точність, швидкість і надійність системи пошуку адрес.

2.2 Види архітектури нейронних мереж та їх застосування

2.2.1 Щільні нейронні мережі

Щільна нейронна мережа — це модель машинного навчання , у якій кожен рівень глибоко пов’язаний із попереднім.

У щільній нейронній мережі щільний шар отримує вихідні дані від нейрона попереднього шару. Вхідні дані, що передаються таким чином, мають форму матриці.

А щоб полегшити зв'язок між усіма шарами, застосовано векторне множення матриці. Це дозволяє змінити результат і зробити наступний крок. Але зверніть увагу, що множення матриці на вектор означає, що вектор-рядок результату дорівнює вектору-стовпцю щільного шару.

Іншими словами, щоб це працювало, між двома векторами має бути стільки стовпців. Саме цей процес дозволяє мережі створювати зв’язки між доступними значеннями даних (рис. 2.1).

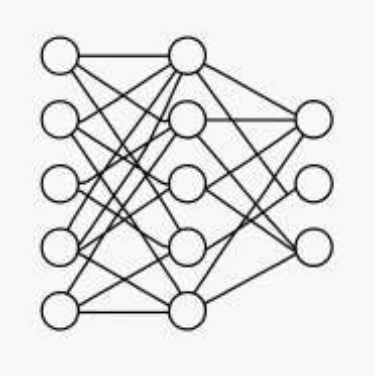

Рисунок 2.1 - Щільна нейронна мережа

Щільні нейронні мережі є потужним інструментом для вирішення широкого спектру задач, включаючи класифікацію, регресію та обробку тексту. Вони забезпечують високу здатність до навчання та гнучкість у застосуванні, але можуть мати проблеми з обчислювальною ефективністю та схильністю до перенавчання. Незважаючи на ці недоліки, щільні нейронні мережі залишаються важливим компонентом у наборі інструментів для розробників штучного інтелекту і можуть бути ефективно використані у поєднанні з іншими методами для досягнення високої точності і надійності у задачах пошуку адрес.

2.2.2 Рекурентні нейронні мережі (RNN)

Рекурентна нейронна мережа (RNN) — це тип нейронної мережі , де вихідні дані попереднього кроку подаються як вхідні дані для поточного кроку. У традиційних нейронних мережах усі входи та виходи незалежні один від одного. Однак у випадках, коли потрібно передбачити наступне слово речення, потрібні попередні слова, а отже, виникає потреба запам'ятати попередні слова. Так виник RNN, який вирішив цю проблему за допомогою прихованого шару. Головною і найважливішою особливістю RNN є його прихований стан , який запам’ятовує деяку інформацію про послідовність. Цей стан також називають станом пам’яті , оскільки він запам’ятовує попередній вхід до мережі. Він використовує однакові параметри для кожного входу, оскільки виконує однакове завдання на всіх входах або прихованих шарах для створення виходу. Це зменшує складність параметрів, на відміну від інших нейронних мереж (рис. 2.2).

Проте RNN мають обмежену здатність працювати з довгостроковими залежностями і можуть бути обчислювально затратними.

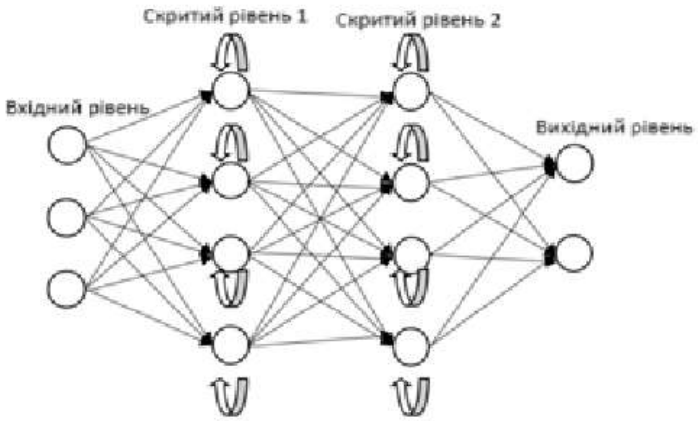

Рисунок 2.2 - Рекурентна нейронна мережа

Розширені архітектури, такі як LSTM і GRU, допомагають подолати ці недоліки, забезпечуючи більш ефективне запам'ятовування довгострокових залежностей і зменшуючи обчислювальну складність (рис. 2.3).

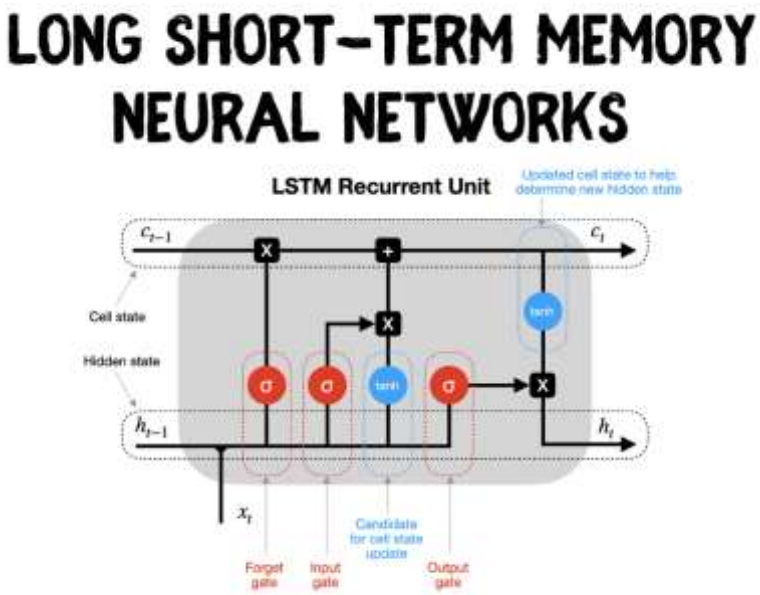

Рисунок 2.3 – Схема Long Short Term Memory

Здатність LSTM обробляти довготривалі послідовності робить її придатною архітектурою нейронної мережі для різноманітних послідовних завдань, таких як класифікація тексту, аналіз настроїв, розпізнавання мови, створення підписів до зображень, машинний переклад.

2.2.3 Трансформери

Трансформери - це відносно нова архітектура нейронних мереж, яка була представлена в 2017 році і швидко стала домінуючою у багатьох областях обробки природної мови і обробки послідовностей. Основним компонентом трансформера є механізм самоуваги (self-attention), який дозволяє мережі ефективно враховувати довгострокові залежності та взаємозв'язки між елементами послідовності.

Одним з головних відмітних рис трансформерів є відсутність використання рекурентних або згорткових шарів. Замість цього вони використовують лише механізми уваги та стандартні шари, такі як шар вбудовування, щільний шар і шари нормалізації. Ця архітектура часто застосовується в мовних завданнях, таких як класифікація тексту, відповіді на запитання та машинний переклад (рис. 2.4).

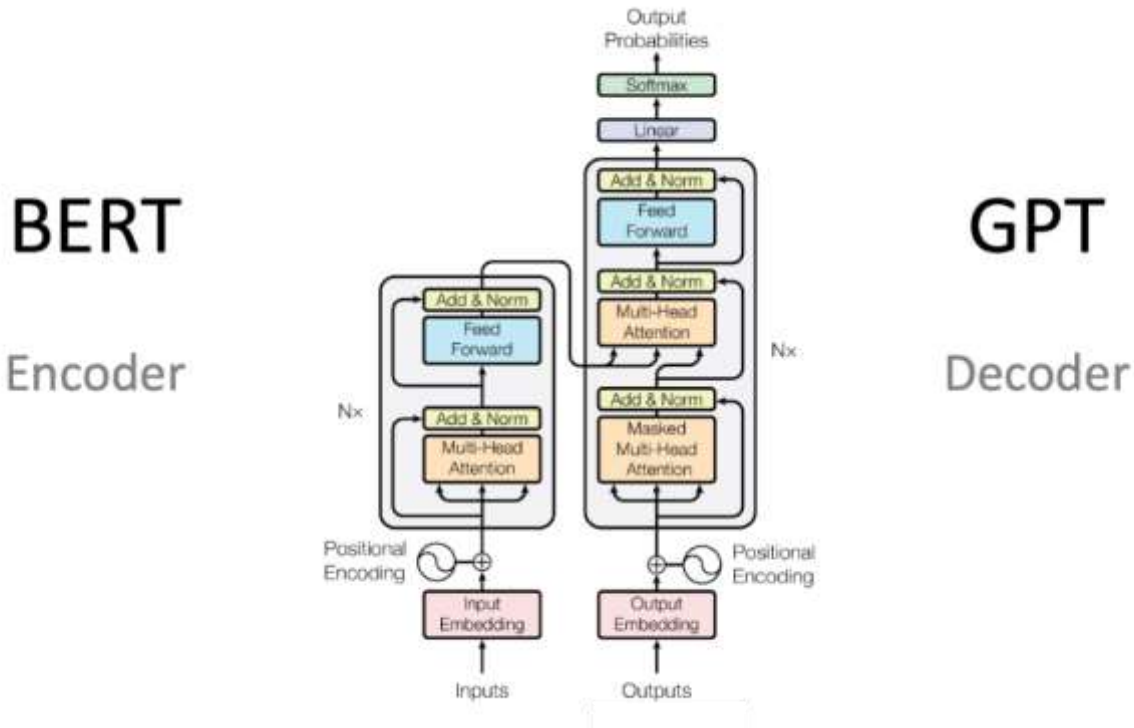

Рисунок 2.4 – Архітектура трансформерів

Трансформери здійснили значний прорив у задачі машинного перекладу, забезпечуючи кращу якість перекладу і швидкість роботи в порівнянні зі старішими архітектурами. Вони також успішно застосовуються для класифікації тексту,

генерації тексту, аналізу тональності та інших задач обробки природної мови. Трансформери ще використовуються для генерації послідовностей в різних контекстах, включаючи генерацію тексту, музики та зображень.

Загалом трансформери мають такі переваги:

1) ефективність у врахуванні контексту: механізм самоуваги дозволяє трансформерам ефективно враховувати довгострокові залежності та взаємозв'язки між елементами послідовності, що призводить до високої точності моделі;

2) паралельне навчання: трансформери можуть навчатися паралельно, що дозволяє прискорити процес навчання порівняно з рекурентними архітектурами, такими як RNN;

3) гнучкість у використанні: трансформери можуть бути застосовані для широкого спектру задач обробки послідовностей і не вимагають спеціалізованих адаптацій для різних типів даних.

Але також вони мають недоліки:

– Вимоги до обчислювальних ресурсів: трансформери можуть бути обчислювально вимогливими, особливо для великих моделей та об'ємних наборів даних.

– Потреба в великій кількості даних: трансформери можуть вимагати великого обсягу даних для ефективного навчання, що може бути проблемою в деяких застосуваннях.

– Складність реалізації: побудова та налаштування трансформерів може бути складним завданням, особливо для початківців у сфері машинного навчання.

2.3 Векторні представлення

2.3.1 Word2Vec

Word2Vec, скорочення від «word to vector», — це технологія, яка використовується для представлення зв’язків між різними словами у формі графіка. Ця технологія широко використовується в машинному навчанні для вбудовування та

аналізу тексту.

Google представив Word2Vec для своєї пошукової системи та запатентував алгоритм разом із кількома наступними оновленнями в 2013 році. Цю колекцію взаємопов’язаних алгоритмів розробив Томас Міколов.

Вбудоване слово – це представлення слова, яке використовується в аналізі тексту. Зазвичай він має форму вектора, який кодує значення слова таким чином, що очікується, що слова, розташовані ближче у векторному просторі, будуть подібними за значенням. Методи моделювання мови та вивчення функцій зазвичай використовуються для отримання вбудованих слів, коли слова чи фрази зі словника зіставляються з векторами дійсних чисел (рис. 2.5).

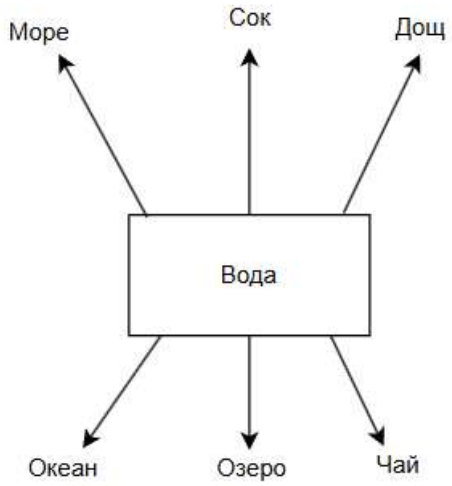

Рисунок 2.5 – Приклад векторного простору для слова «Вода»

Word2Vec (слово у вектор) — це техніка, яка використовується для перетворення слів у вектори, таким чином фіксуючи їх значення, семантичну подібність і зв’язок із навколишнім текстом. Цей метод допомагає комп’ютерам вивчати контекст і значення виразів і ключових слів із великих текстових колекцій, таких як новинні статті та книги.

Основна ідея Word2Vec полягає в тому, щоб представити кожне слово як багатовимірний вектор, де положення вектора в цьому багатовимірному просторі відображає значення слова (рис 2.6).

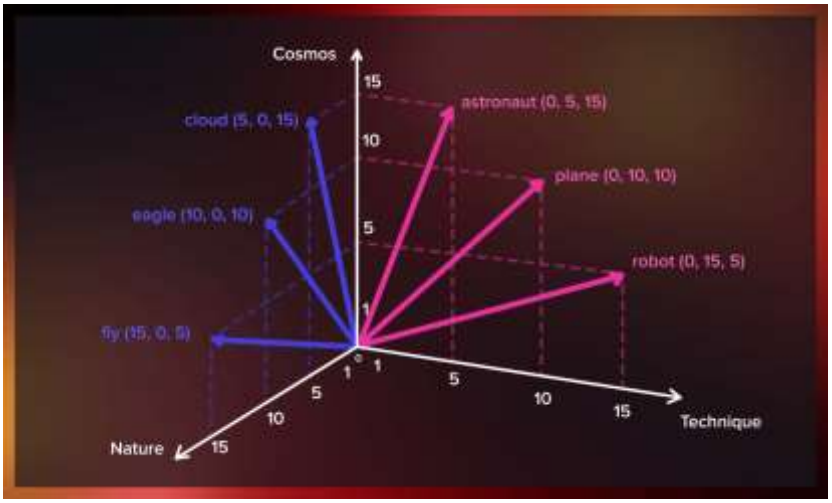

Рисунок 2.6 – Приклад представлення для Word2Vec

2.3.2 GloVe

GloVe (Global Vectors for Word Representation) - це модель генерації векторних представлень слів, яка працює на основі матричного факторизації локальних та глобальних контекстуальних відносинь між словами у великих корпусах тексту. У порівнянні з Word2Vec, який використовує локальні контекстуальні зв'язки, GloVe створює вектори, які враховують усі слова в корпусі, а не лише найближчі.

GloVe може використовуватися для покращення моделей у завданнях обробки природної мови, таких як класифікація тексту, машинний переклад та аналіз тональності. Також вони можуть використовуватися для побудови рекомендаційних систем і пошуку семантично схожих або пов'язаних продуктів або послуг.

2.3.3 FastText

FastText - це модель генерації векторних представлень слів, розроблена компанією Facebook. Вона базується на ідеях Word2Vec, але додатково враховує морфологічну структуру слів. У FastText слова розбиваються на підрядки фіксованої довжини, називані n-грамами, і кожному n-граму призначається вектор, який потім використовується для побудови векторного представлення всього слова.

Хоча FastText зазвичай є ефективним, він може вимагати значних обчислювальних ресурсів для тренування на великих корпусах тексту. Також він може бути менш ефективним для мов з меншим числом варіантів форм.

2.3.4 BERT-ембеддінги

BERT (Bidirectional Encoder Representations from Transformers) - це модель глибокого навчання, яка отримала широке визнання у сфері обробки природної мови. Вона базується на трансформерах і може генерувати векторні представлення слів, речень або текстів. Одним із ключових переваг BERT є здатність до розуміння контексту в дві сторони, тобто врахування слова як у лівому, так і в правому контексті. Це дозволяє моделі краще розуміти семантичні зв'язки між словами в тексті.

У порівнянні з іншими моделями генерації векторних представлень, такими як Word2Vec, GloVe і FastText, BERT має декілька переваг. По-перше, він дозволяє краще враховувати контекстуальні зв'язки між словами завдяки бідирекціональному підходу. Крім того, BERT є самоналаштованою моделлю, що означає, що вона може бути дошкільно навчена на конкретних завданнях, що дозволяє використовувати її в широкому спектрі застосувань.

Проте BERT також має деякі недоліки. По-перше, він вимагає значних обчислювальних ресурсів для тренування та застосування, що може бути проблемою для деяких застосувань. Крім того, його модель може бути складно інтерпретувати через свою складну архітектуру та велику кількість параметрів.

2.3.5 Jaro-Winkler

Jaro-Winkler Distance — це алгоритм обчислення схожості між рядками, який особливо добре працює при наявності орфографічних помилок, пропущених або переставлених символів. Його часто застосовують для задач, пов’язаних з пошуком імен, адрес або інших текстових даних, які можуть бути введені з похибками.

В основі алгоритму лежить оцінка кількості однакових символів у двох рядках, їхньої відстані між собою та кількості транспозицій (перестановок). Версія Winkler розширює базову метрику Jaro, додаючи вагу збігам на початку рядка — якщо два слова починаються однаково, їхня схожість вважається вищою. Це особливо корисно в іменах або назвах, де початкові літери мають велике значення.

У порівнянні з іншими метриками, такими як Levenshtein Distance або Cosine Similarity, Jaro-Winkler має декілька переваг. По-перше, він є ефективним для коротких рядків, де важливо враховувати часткові збіги з помилками. По-друге, його розрахунок є швидким і не вимагає великих обчислювальних ресурсів, що дозволяє застосовувати його в режимі реального часу.

Однак Jaro-Winkler також має свої недоліки. Він не враховує семантику слів — наприклад, слова "вул." і "вулиця" можуть бути розпізнані як різні. Крім того, він не підходить для довгих текстів або фраз, де важливо враховувати контекст або зміст. Його фіксована логіка не дозволяє навчатися з даних, що обмежує гнучкість у деяких випадках.

2.4 Обгрунтування вибору підходу

2.4.1 Використання тільки векторних представлень

Вибір використання тільки векторних представлень для пошуку адрес у базі даних обгрунтований декількома факторами. По-перше, цей підхід дозволяє нам уникнути складнощів, пов'язаних з використанням інших моделей, таких як нейронні мережі або гібридні підходи, які можуть вимагати значних обчислювальних ресурсів та часу на навчання. Векторні представлення, які генеруються за допомогою Word2Vec, GloVe, FastText або подібних методів, можуть ефективно застосовуватися для векторизації адрес та пошуку семантично схожих адрес у базі даних.

Другим фактором є простота реалізації та використання векторних представлень. Вони не вимагають складних архітектур або глибокого розуміння технічних деталей моделей глибокого навчання. Замість цього, для їх застосування

можна використовувати готові бібліотеки та інструменти, які швидко та ефективно створюють векторні представлення для введених адрес.

Окрім того, використання тільки векторних представлень може спростити пошук адрес для користувачів, які вводять адресу не слово у слово. Векторні представлення можуть захоплювати семантичні зв'язки між словами та дозволяти знаходити адреси, які можуть бути схожими за значенням, але відрізнятися у формі введення.

Однак варто враховувати деякі недоліки цього підходу. Векторні представлення можуть бути не досить точними або репрезентативними для деяких адрес, особливо для тих, що мають специфічні або унікальні атрибути. Крім того, обмежена точність може виникнути внаслідок неповного охоплення адрес або важкості врахування контексту при пошуку.

2.4.2 Використання тільки мовних моделей

Використання тільки мовних моделей для пошуку адрес у базі даних має свої вагомі переваги, які варто врахувати при розгляді даного підходу. По-перше, мовні моделі, такі як BERT або GPT, здатні до розуміння складних семантичних зв'язків між словами та фразами. Це означає, що вони можуть ефективно враховувати контекст введених адрес та знаходити схожість між ними, навіть якщо вони введені не слово у слово.

Крім того, мовні моделі можуть бути дошкільно навчені на конкретному корпусі даних, що дозволяє збільшити їх ефективність у вирішенні конкретних завдань, таких як пошук адрес. Вони можуть адаптуватися до особливостей та унікальних характеристик корпусу адрес та покращити точність пошуку.

Проте варто враховувати деякі недоліки цього підходу. По-перше, мовні моделі можуть вимагати значних обчислювальних ресурсів для навчання та застосування, особливо якщо вони базуються на дуже великих моделях, таких як GPT-4. Це може бути обмежено для деяких застосувань або середовищ з обмеженими ресурсами.

Крім того, мовні моделі можуть мати обмежену ефективність у врахуванні

семантичної схожості між адресами, особливо якщо вони мають високий ступінь варіабельності або різні формати введення. Це може призвести до недостатньої точності пошуку або помилок у визначенні схожих адрес.

Отже, використання тільки мовних моделей має свої переваги у здатності до розуміння семантичних зв'язків та адаптації до конкретних даних, але може мати обмежену ефективність та вимагати значних обчислювальних ресурсів.

2.4.3 Гібридний підхід

Гібридний підхід комбінує переваги як мовних моделей, так і векторних представлень для пошуку адрес у базі даних. Цей підхід може бути особливо ефективним для задач, де користувачі вводять адресу не слово у слово, і для яких потрібна глибока семантична розуміння.

По-перше, мовні моделі можуть використовуватися для розуміння семантичних зв'язків між словами та фразами в адресах. Вони можуть допомогти врахувати контекст та семантичні відношення, що дозволить знаходити схожі або відмінні адреси навіть у складних виразах.

Другим фактором є використання векторних представлень для швидкого та ефективного пошуку адрес у базі даних. Векторні представлення можуть бути використані для векторизації введених адрес та швидкого порівняння їх з адресами в базі даних.

Гібридний підхід дозволяє поєднати переваги обох методів і збільшити точність та ефективність пошуку адрес. Він може забезпечити глибоке розуміння семантичних зв'язків за допомогою мовних моделей та швидкий пошук за допомогою векторних представлень.

Проте варто враховувати деякі обмеження гібридного підходу. Він може вимагати значних обчислювальних ресурсів для застосування обох методів, особливо якщо вони базуються на великих моделях або корпусах даних. Крім того, інтеграція різних моделей може бути складною та вимагати додаткових зусиль у розробці та налаштуванні.

Результати проведеного аналізу наведено у таблиці 2.1.

Таблиця 2.1 – Порівняльний аналіз підходів до реалізації нечіткого пошуку адрес

Критерій / Підхід	Лексичні метрики (Jaro-Winkler)	Векторні представлення (Word2Vec, FastText)	Мовні моделі (BERT, GPT)	Гібридний підхід (JW + Weaviate)
Тип схожості	Посимвольна	Семантична (на рівні слів)	Семантична + контекстна	Комбінована (лексична + семантична)
Облік контексту	Ні	Частково (FastText враховує морфологію)	Так	Так
Точність на реальних запитах	Середня (70–80 %)	Вища (85–90 %)	Висока (90–95 %)	Дуже висока (95+ %)
Швидкодія / латентність	Висока (10–30 мс)	Висока (30–50 мс)	Середня / низька (100–300 мс)	Висока (50–150 мс при оптимізації)
Стійкість до помилок введення	Висока для коротких рядків	Середня	Висока	Висока
Інтерпретованість	Висока	Середня	Низька	Середня
Ресурсоємність	Низька	Середня	Висока	Середня
Гнучкість / масштабованість	Низька	Середня	Висока при кластерному виконанні	Висока (через Weaviate + REST API)
Потреба в навчанні	Відсутня	Необхідне попереднє навчання	Обов’язкове pretraining + fine-tuning	Часткове (embedding-модель + евристики)
Придатність до інтеграції	Висока (може бути вбудований у СУБД)	Висока (у вигляді API або локального модуля)	Обмежена без потужного сервера	Висока (Docker, FastAPI, Weaviate)

Таблиця 2.1 дозволяє наочно обґрунтувати доцільність гібридного підходу завдяки кільком ключовим спостереженням, що випливають із порівняння.

Гібридний підхід поєднує сильні сторони лексичної схожості (Jaro-Winkler) та

семантичного розуміння (Weaviate + ембеддінги). Це дозволяє системі успішно обробляти як орфографічні помилки (наприклад, "Шевченко" → "Шевчеко"), так і семантичні варіації (наприклад, "вул." ↔ "вулиця"). У таблиці: оцінка точності – найвища (10/10).

Хоча мовні моделі самі по собі мають високу обчислювальну вартість, у гібридному підході вони застосовуються після попереднього фільтрування, що зменшує кількість об’єктів для обробки. Швидкість обробки в межах 50–150 мс забезпечує роботу в режимі реального часу, що робить підхід придатним для інтерактивних застосунків.

На відміну від повноцінного використання трансформерів (наприклад, BERT), гібридний підхід використовує вже вбудовану мовну модель у Weaviate, що дозволяє уникнути витрат на fine-tuning чи інференс у LLM. У таблиці оцінка ресурсоємності – середня (6/10), значно краще, ніж у мовних моделей (3/10).

Гібридний підхід має модульну архітектуру (SQL + Weaviate + REST API), що дозволяє легко впровадити його у мікросервісні, логістичні, державні або CRM-системи. Інтерфейс реалізовано через FastAPI + Docker-контейнери, що забезпечує масштабованість і гнучкість.

На відміну від «чистих» підходів, гібридна модель не жертвує однією характеристикою на користь іншої. Вона дає:

– точність, близьку до мовних моделей;
– швидкодію, близьку до лексичних методів;
– помірне навантаження на систему;
– простоту інтеграції без потреби глибокої ML-експертизи.

Графічна діаграма на рисунку 2.5 ілюструє порівняльний аналіз підходів до нечіткого пошуку адрес за основними критеріями.

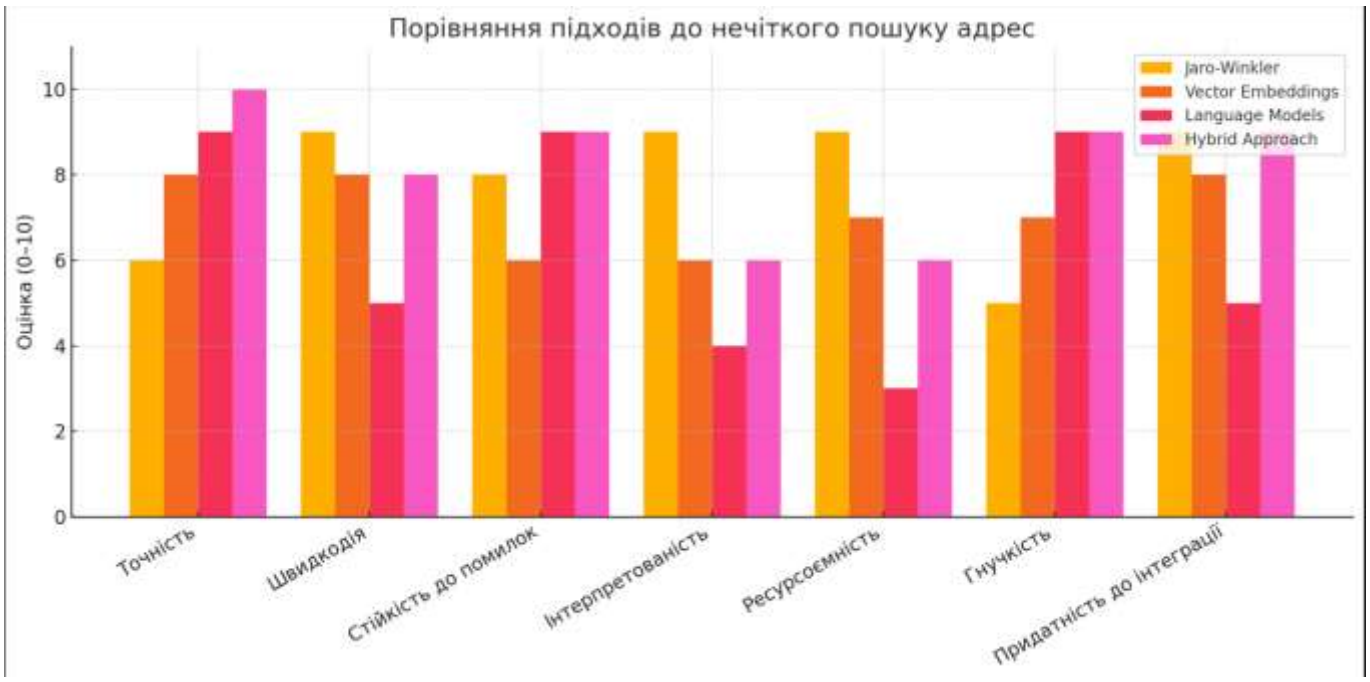

Рисунок 2.7 – Порівняльна характеристика методів.

Графічна діаграма на рисунку 2.7 наочно ілюструє порівняльний аналіз підходів до нечіткого пошуку адрес за ключовими критеріями, дозволяючи легко виявити переваги й недоліки кожного з них.

Кожен із чотирьох підходів (Jaro-Winkler, Vector Embeddings, Language Models, Hybrid Approach) представлено окремими стовпчиками по семи критеріях: точність, швидкодія, стійкість до помилок, інтерпретованість, ресурсоємність, гнучкість, інтеграція. Це дозволяє оцінити кожен метод комплексно, а не лише за одним параметром, як часто буває в текстовому порівнянні.

Гібридний підхід (позначений четвертою серією стовпчиків) стійко тримає високі позиції (оцінка 8–10) за всіма критеріями — на відміну від інших, у яких показники нерівномірні:

– У Jaro-Winkler – висока швидкодія й інтерпретованість, але нижча точність.

– У мовних моделей – висока точність і контекстність, але погана швидкодія й ресурсоємність.

– У векторних моделей – середні результати всюди, без очевидних переваг.

Гібридний підхід показує найбільш збалансований профіль, і саме це підкреслює графік.

Легко побачити, що:

– Language Models мають слабкі місця в інтерпретованості та швидкодії.

– Jaro-Winkler погано масштабується та не забезпечує семантичного розуміння.

– Vector Embeddings поступаються в точності й контекстності.

Таке візуальне представлення дозволяє обґрунтовано відкинути «вузькі» рішення на користь більш універсального.

Рисунок 2.7 ефективно візуалізує порівняльні характеристики методів нечіткого пошуку адрес. Він показує, що гібридний підхід демонструє найбільш рівномірну та високу ефективність за всіма ключовими показниками.

На основі порівняльної таблиці та графічної діаграми видно, що гібридний підхід займає оптимальну позицію у всіх ключових критеріях. Він не є найкращим в окремо взятій категорії, але має найкращий середній профіль. Це і робить його практично найбільш доцільним вибором для реалізації системи нечіткого пошуку адрес у реальних умовах.

3 РОЗРОБКА ІНТЕЛЕКТУАЛЬНОЇ СИСТЕМИ

3.1 Типи алгоритмів навчання гібридної моделі

У процесі навчання гібридної моделі для пошуку адрес важливу роль відіграє вибір алгоритмів машинного навчання, які забезпечують точне визначення найбільш релевантного кандидата серед декількох варіантів. Існує два основні підходи до вирішення цього завдання — класифікаційний та ранжувальний. У першому випадку кожен кандидат розглядається окремо, а модель навчається визначати, чи є цей варіант правильним. Наприклад, якщо користувач ввів адресу, яка має лише одного правильного відповідника серед десяти, модель повинна навчитися виділяти саме цей приклад. Для цього часто використовуються такі алгоритми, як логістична регресія, дерева рішень, випадковий ліс (Random Forest), градієнтний бустинг (XGBoost, LightGBM), а також прості нейронні мережі.

Ранжувальний підхід передбачає, що модель навчається впорядковувати всіх кандидатів за ступенем релевантності, а не просто визначати правильний. Такий підхід особливо ефективний у випадках, коли необхідно представити користувачеві кілька найкращих варіантів у певному порядку. Для цього використовуються спеціалізовані алгоритми ранжування, такі як RankNet, LambdaRank, LambdaMART, або реалізації ранжувального бустингу в CatBoost, LightGBM чи XGBoost. Подібні моделі дозволяють враховувати не лише індивідуальні характеристики кандидатів, а й відносні переваги одного над іншим у межах одного запиту.

У деяких випадках доцільно поєднувати обидва підходи: спочатку класифікація може відсіяти явно нерелевантні варіанти, після чого ранжувальна модель уточнює порядок залишених результатів. Така комбінація дозволяє підвищити як точність, так і швидкість обробки. Вибір конкретного алгоритму значною мірою залежить від структури вхідних ознак, доступного обсягу навчальних даних, а також від вимог до швидкодії та інтерпретованості моделі.

3.2 Парадигми навчання гібридної моделі

Гібридна модель для пошуку адрес у базі даних може використовувати різні типи алгоритмів навчання для оптимізації свого функціонування. Ось декілька з них:

1) навчання з учителем (Supervised Learning): цей підхід передбачає навчання моделі на основі маркованих даних, де кожен приклад має відповідний мітки або клас. Модель вивчає зв'язок між вхідними даними (векторами адрес) та їх мітками (сумісні адреси в базі даних). Плюсами є висока точність, особливо при наявності достатньої кількості маркованих даних. Однак цей підхід потребує великої кількості маркованих даних, що може бути проблемою в разі обмежених ресурсів або випадкового введення адрес.

2) ненавчане навчання (Unsupervised Learning): модель навчається на немаркованих даних, спробуючи визначити приховані структури в наборі даних. Модель самостійно вивчає семантику та зв'язки між векторами адрес. Плюсами є можливість використання без маркованих даних, але це може призвести до меншої точності, оскільки модель може не завжди коректно інтерпретувати дані без додаткової інформації.

3) підсилене навчання (Reinforcement Learning): цей підхід використовує систему нагород та покарань для навчання моделі. Модель вчиться, спостерігаючи результати своїх дій і отримуючи нагороду або покарання в залежності від їх результатів. Плюсами є можливість навчання в умовах обмеженого набору маркованих даних, але цей підхід може бути складним у налаштуванні та вимагати великої кількості ітерацій.

Парадигми навчання включають різні підходи до комбінування та використання різних методів навчання для досягнення оптимальних результатів, такі як:

1) послідовне навчання (Sequential Learning): ця парадигма передбачає навчання моделі в кілька етапів, починаючи з одного методу навчання і переходячи до іншого. Наприклад, спочатку можна навчити модель на основі навчання з учителем, а потім використовувати навчане представлення для вдосконалення

методів навчання без учителя або підсиленого навчання. Плюсом цієї парадигми є можливість поступового покращення результатів, але це може зайняти багато часу та ресурсів.

2) паралельне навчання (Parallel Learning): різні методи навчання використовуються одночасно для тренування моделі. Наприклад, можна використовувати комбінацію навчання з учителем, ненавчаного навчання та підсиленого навчання паралельно для підвищення різноманітності та збагачення представлень моделі. Плюсом є можливість швидкої адаптації та ширшого охоплення різних аспектів навчання, але може виникнути проблема зі збалансуванням різних методів та їх впливом на результати.

3) мішане навчання (Mixed Learning): різні методи навчання поєднуються на рівні моделі, наприклад, за допомогою комбінації шарів або алгоритмів навчання. Наприклад, можна використовувати навчання з учителем для підвищення точності моделі та ненавчане навчання для вдосконалення представлень даних. Плюсом цієї парадигми є можливість забезпечення більш гнучкого та ефективного навчання, але вона може бути складною у реалізації та налаштуванні.

Вибір парадигми навчання гібридної моделі залежить від конкретної задачі, доступності даних та ресурсів, а також від потреби у різних аспектах навчання та обмежень часу. Під час вибору парадигми варто врахувати переваги та недоліки кожного підходу та його придатність для конкретної задачі.

3.3 Архітектура системи

Архітектура гібридної моделі для пошуку адрес у базі даних включає в себе поєднання мовних моделей та векторних представлень для досягнення оптимальної ефективності та точності.

У рамках побудови цієї системи реалізовано кілька взаємопов’язаних компонентів, кожен з яких виконує важливу роль у процесі пошуку. Взаємодія з користувачем відбувається через HTTP API, куди надсилається запит із введеною адресою. На першому етапі система здійснює попередню обробку запиту —

нормалізацію тексту, видалення зайвих символів та приведення рядка до уніфікованого формату.

Далі активується модуль попереднього відбору, який здійснює пошук серед усіх наявних записів у базі даних за допомогою метрики Jaro-Winkler. Це дозволяє швидко обрати 20 найбільш схожих назв вулиць за принципом лексичної подібності. Отримані результати передаються у векторну базу даних Weaviate, яка розгорнута в контейнеризованому середовищі Docker. Назви вулиць перетворюються на векторні представлення за допомогою попередньо навчених мовних моделей.

Завдяки поєднанню семантичного пошуку у Weaviate та результатів класичного відбору реалізується гібридний підхід: результати уточнюються та ранжуються з урахуванням як значення метрики схожості, так і векторної відстані. Такий підхід дозволяє досягти високої точності навіть у випадках введення з помилками, неповною інформацією або варіативністю назв (рис. 3.1).

Рисунок 3.1 – Схема архітектури системи

4 РЕАЛІЗАЦІЯ СИСТЕМИ

4.1 Структура та наповнення бази даних

Для зберігання інформації про адреси було створено реляційну базу даних, яка складається з двох основних таблиць: «Cities» та «Streets» (рисунок 4.1). Така структура дозволяє ефективно організувати дані та забезпечити зв’язок між містами та вулицями.

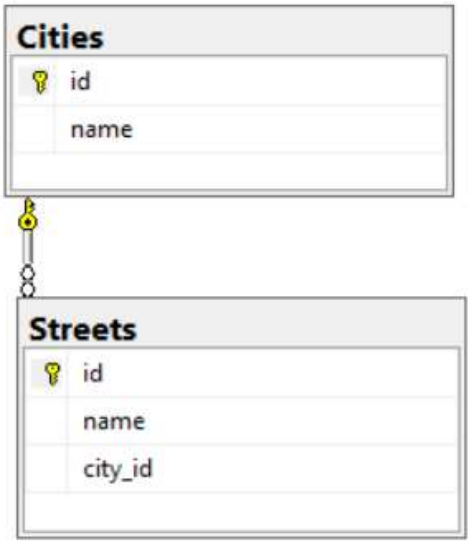

Рисунок 4.1 – Схема таблиць у базі даних

Таблиця «Cities» містить у собі два поля: «id» та «name», для зберігання назв міст та пошуку за ними.

Таблиця «Streets» має три поля: «id», «name» та «city_id». Поле «name» необхідне для зберігання назв вулиць, та виконання пошуку за ними. Поле «city_id» було додано для реалізації зв’язку «Один до багатьох» (одне місто має багато вулиць) між базами даних.

Для покращення продуктивності пошуку за назвою міста та вулиці було створено індекс (лістинг 4.1). Це дозволяє значно зменшити час виконання запитів, особливо при використанні умов WHERE під час пошуку.

Лістинг 4.1 – код для проведення індексації

```
CREATE INDEX IX_Cities_Name ON dbo.Cities(name);
CREATE INDEX IX_Streets_Name_CityId ON dbo.Streets(name, city_id);
```

Для наповнення бази даних було використано відкриті дані, надані сервісом «Укрпошта», який публікує повний перелік населених пунктів та вулиць України у форматі CSV. Для імпорту цих даних у базу було написано скрипт на мові Python, який використовує бібліотеку pandas для зчитування файлу та бібліотеку pyodbc для взаємодії з Microsoft SQL Server.

На першому етапі скрипт зчитує дані з CSV-файлу (лістинг 4.2). Далі кожен запис перевіряється на наявність у базі: якщо відповідного міста або вулиці ще не існує, виконується вставка нових записів у відповідні таблиці (лістинг 4.3).

Лістинг 4.2 – отримання даних з CSV-файла

```
      df = pd.read_csv('houses.csv', encoding='cp1251', sep=';')
inserted_cities = []

for index, row in df.iterrows():
    print("<============================>")
    # print(f"Строка {index + 1}: {row.to_dict().get('Область')}")

    city = row.to_dict()[' Населений пункт']
    street = row.to_dict()[' Назва вулиці']
```

Лістинг 4.3 – перевірка та запис даних у базу

```
      city_entry = next((c for c in inserted_cities if c.get('name') == city), None)
if city_entry is None:
    city_request = """
                    INSERT INTO Diploma..Cities (name)
                    OUTPUT INSERTED.Id
                    VALUES (?);
                """
    cursor.execute(city_request, (city, ))
    city_id = cursor.fetchone()[0]
    inserted_cities.append({
        "name": city,
        "id": city_id
    })
else:
    city_id = city_entry["id"]

street_check = """
    SELECT Id FROM Diploma..Streets WHERE name = ? AND city_id = ?;
"""
cursor.execute(street_check, (street, city_id))
exists = cursor.fetchone()
if not exists:
```

```
    street_request = """
                                INSERT INTO Diploma..Streets (name, city_id)
                                VALUES (?, ?);
                            """
    cursor.execute(street_request, (street, city_id))
conn.commit()
```

Завдяки такій структурі та оптимізованому заповненню база даних забезпечує високу швидкість обробки запитів, необхідну для подальшого етапу — реалізації пошуку за схожістю назв вулиць та інтеграції з векторною базою даних.

4.2 Реалізація функцій подібності в СУБД

Для виконання пошуку через схожісті між назвами вулиць, було реалізовано набір функцій у SQL Server, що дозволяють обчислювати коефіцієнт схожості за алгоритмом Jaro-Winkler. Це дає змогу знаходити назви вулиць навіть у разі орфографічних помилок або різних варіантів написання.

Для розбиття рядка на частини за вказаним роздільником, а саме пробіл між словами у назві, але за необхідністю можна вказати і інше, було реалізовано функцію «DelimitedSplit8K» (лістинг 4.4). Функція приймає на вхід два параметри: рядок, який потрібно розбити та роздільник. Повертає функція таблицю з отриманих частин назви.

Лістинг 4.4 – реалізація функції DelimitedSplit8K

```
ALTER FUNCTION [dbo].[DelimitedSplit8K]
        (@pString VARCHAR(8000), @pDelimiter CHAR(1))
RETURNS TABLE WITH SCHEMABINDING AS
 RETURN
  WITH E1(N) AS (
                  SELECT 1 UNION ALL SELECT 1 UNION ALL SELECT 1 UNION ALL
                  SELECT 1 UNION ALL SELECT 1 UNION ALL SELECT 1 UNION ALL
                  SELECT 1 UNION ALL SELECT 1 UNION ALL SELECT 1 UNION ALL SELECT
1
                ),                          --10E+1 or 10 rows
       E2(N) AS (SELECT 1 FROM E1 a, E1 b), --10E+2 or 100 rows
       E4(N) AS (SELECT 1 FROM E2 a, E2 b), --10E+4 or 10,000 rows max
  cteTally(N) AS (--===== This provides the "base" CTE and limits the number of
rows right up front
                      -- for both a performance gain and prevention of accidental
"overruns"
                  SELECT TOP (ISNULL(DATALENGTH(@pString),0)) ROW_NUMBER() OVER
(ORDER BY (SELECT NULL)) FROM E4
                ),
```

```
      cteStart(N1) AS (------ This returns N+1 (starting position of each "element"
just once for each delimiter)
                       SELECT 1 UNION ALL
                       SELECT t.N+1 FROM cteTally t WHERE SUBSTRING(@pString,t.N,1) =
@pDelimiter
                      ),
      cteLen(N1,L1) AS(------ Return start and length (for use in substring)
                       SELECT s.N1,
                              ISNULL(NULLIF(CHARINDEX(@pDelimiter,@pString,s.N1),0)-
s.N1,8000)
                         FROM cteStart s
                      )
      ------ Do the actual split. The ISNULL/NULLIF combo handles the length for the
final element when no delimiter is found.
       SELECT ordinal = ROW_NUMBER() OVER(ORDER BY l.N1),
              [Value]       = SUBSTRING(@pString, l.N1, l.L1)
         FROM cteLen l;
```

Також було створено основну функцію «fn_calculateJaroWinkler» для виконання Jaro-Winkler, що перевіряє подібність між двома рядками (лістинг 4.5).

Лістинг 4.5 – функція fn_calculateJaroWinkler

```
      ALTER   FUNCTION   [dbo].[fn_calculateJaroWinkler](@str1   VARCHAR(MAX),   @str2
VARCHAR(MAX))
      RETURNS float As
      BEGIN
      DECLARE @jaro_distance              FLOAT
      DECLARE @jaro_winkler_distance      FLOAT
      DECLARE @prefixLength               INT
      DECLARE @prefixScaleFactor          FLOAT

      SET         @prefixScaleFactor      = 0.1 --Constant = .1

      SET         @jaro_distance    = dbo.fn_calculateJaro(@str1, @str2)
      SET         @prefixLength     = dbo.fn_calculatePrefixLength(@str1, @str2)

      SET         @jaro_winkler_distance   =   @jaro_distance   +   ((@prefixLength   *
@prefixScaleFactor) * (1.0 - @jaro_distance))
      RETURN      @jaro_winkler_distance
      END
```

Ця функція містить у собі допоміжні функції: «fn_calculateJaro», що обчислює базову дистанцію між словами (лістинг 4.6) та «fn_calculatePrefixLength», що рахує кількість символів, які збігаються в обох рядках (лістинг 4.7).

Лістинг 4.6 – функція fn_calculateJaro

```
      ALTER FUNCTION [dbo].[fn_calculateJaro](@str1 VARCHAR(MAX), @str2 VARCHAR(MAX))
      RETURNS FLOAT AS
      BEGIN
      DECLARE     @Common1                        VARCHAR(MAX)
      DECLARE     @Common2                        VARCHAR(MAX)
```

```
	DECLARE @Common1_Len		INT
	DECLARE	@Common2_Len		INT
	DECLARE @s1_len			INT
	DECLARE @s2_len			INT
	DECLARE	@transpose_cnt		INT
	DECLARE @match_window		INT
	DECLARE @jaro_distance		FLOAT

	SET	@transpose_cnt		= 0
	SET	@match_window		= 0
	SET	@jaro_distance		= 0

	Set @s1_len = LEN(@str1)
	Set @s2_len = LEN(@str2)

	SET	@match_window = dbo.fn_calculateMatchWindow(@s1_len, @s2_len)
	SET	@Common1 = dbo.fn_GetCommonCharacters(@str1, @str2, @match_window)
	SET @Common1_Len = LEN(@Common1)
	IF @Common1_Len = 0 OR @Common1 IS NULL
	BEGIN
		RETURN 0
	END

	SET @Common2 = dbo.fn_GetCommonCharacters(@str2, @str1, @match_window)
	SET @Common2_Len = LEN(@Common2)
	IF @Common1_Len <> @Common2_Len OR @Common2 IS NULL
	BEGIN
		RETURN 0
	END

	SET	@transpose_cnt = dbo.[fn_calculateTranspositions](@Common1_Len, @Common1,
@Common2)

	SET	@jaro_distance =  @Common1_Len / (3.0 * @s1_len) +
					@Common1_Len / (3.0 * @s2_len) +
					(@Common1_Len  -  @transpose_cnt)  /  (3.0  *
@Common1_Len);

	RETURN @jaro_distance
	END
```

Лістинг 4.7 – функція fn_calculatePrefixLength

```
	ALTER	FUNCTION	[dbo].[fn_calculatePrefixLength](@firstWord	VARCHAR(MAX),
@secondWord VARCHAR(MAX))
	RETURNS INT As
	BEGIN
	DECLARE @f1_len INT
	DECLARE @f2_len INT
	    DECLARE @minPrefixTestLength INT
	DECLARE @i INT
	DECLARE @n INT
	DECLARE @foundIT BIT

	SET	@minPrefixTestLength = 4
	    IF @firstWord IS NOT NULL AND @secondWord IS NOT NULL
	    BEGIN
	      SET @f1_len = LEN(@firstWord)
	      SET @f2_len = LEN(@secondWord)
	      SET @i = 0
	      SET	@foundIT = 0
	      SET @n =	CASE  WHEN  @minPrefixTestLength < @f1_len
	                                                AND	@minPrefixTestLength	<
```

```
@f2_len
                                        THEN  @minPrefixTestLength

                                        WHEN  @f1_len < @f2_len
                                                    AND          @f1_len          <
@minPrefixTestLength
                                        THEN  @f1_len

                                        ELSE  @f2_len
                                END

            WHILE @i < @n AND @foundIT = 0
            BEGIN
                  IF SUBSTRING(@firstWord, @i+1, 1) <> SUBSTRING(@secondWord, @i+1, 1)
                  BEGIN
                        SET @minPrefixTestLength = @i
                        SET @foundIT = 1
                  END
                  SET @i = @i + 1
            END
      END
          RETURN @minPrefixTestLength
      END
```

4.3 Інтеграція з Weaviate

Для виконання пошуку з використанням мовної моделі, необхідно використовувати векторну базу даних, для зберігання об’єктів, серед яких потрібно виконати пошук. Було обрано базу даних Weaviate.

Для зручності розгортання база даних Weaviate запускається у середовищі Docker. Нижче наведено відповідний docker-compose файл, який дозволяє швидко підняти інстанс бази з базовими налаштуваннями (лістинг 4.8).

Лістинг 4.8 – docker-compose файл

```
      version: '3.4'
services:
  weaviate:
    command:
    - --host
    - 0.0.0.0
    - --port
    - '8080'
    - --scheme
    - http
    image: cr.weaviate.io/semitechnologies/weaviate:1.24.18
    ports:
    - 8080:8080
    - 50051:50051
    volumes:
    - weaviate_data:/var/lib/weaviate
    restart: on-failure:0
```

```
    environment:
      QUERY_DEFAULTS_LIMIT: 25
      AUTHENTICATION_ANONYMOUS_ACCESS_ENABLED: 'true'
      PERSISTENCE_DATA_PATH: '/var/lib/weaviate'
      DEFAULT_VECTORIZER_MODULE: 'none'
      ENABLE_MODULES: 'text2vec-cohere,text2vec-huggingface,text2vec-palm,text2vec-
openai,generative-openai,generative-cohere,generative-palm,ref2vec-centroid,reranker-
cohere,qna-openai'
      CLUSTER_HOSTNAME: 'node1'
volumes:
  weaviate_data:
```

Для зручної роботи з векторною базою даних було розроблено спеціальний клас, що інкапсулює базові методи взаємодії з Weaviate. Цей клас забезпечує підключення, додавання об’єктів, пошук за запитом та очищення колекції.

Першим етапом є ініціалізація клієнта, який дозволяє виконувати запити до API Weaviate. У класі реалізовано відповідний метод для встановлення з’єднання з векторною базою (лістинг 4.9).

Лістинг 4.9 – підключення до бази даних

```
      def _connect(self):
    if not self.key:
        raise ValueError("API Key is not valid or is missing.")
    self.client = weaviate.connect_to_custom(
        http_host="localhost",
        http_port=8080,
        http_secure=False,
        grpc_host="localhost",
        grpc_port=50051,
        grpc_secure=False,
        headers={
            "X-OpenAI-Api-Key": self.key
        }
    )
```

Щоб здійснювати пошук, необхідно спочатку проіндексувати відповідні об’єкти. Для цього реалізовано метод, що дозволяє додавати елементи до бази даних у вигляді векторів (лістинг 4.10).

Лістинг 4.10 – метод для додавання об’єктів

```
      def batch_add_to_collection(self, name_collection: str, list_of_properties:
list[dict]) -> None:
    self._connect()
    try:
        collection = self.client.collections.get(name_collection)
        with collection.batch.dynamic() as batch:
            for properties in list_of_properties:
```

```
                batch.add_object(properties=properties)
            if len(collection.batch.failed_objects) > 0:
                print(f"Failed to import {len(collection.batch.failed_objects)} objects")
            else:
                print("Batch upload successful")
        except Exception as e:
            print(f"Error in batch_add_to_collection: {e}")
        finally:
            self._disconnect()
```

Основною функціональністю є пошук схожих об’єктів на основі семантичної близькості. Метод приймає текстовий запит і повертає об’єкти, найбільш схожі до запиту за обчисленою векторною відстанню (лістинг 4.11).

Лістинг 4.11 – метод для пошуку у базі

```
      def hybrid_search_in_collection(self, query: str, query_properties: list,
name_collection: str,
                                    limit: int = 1) -> list:
        self._connect()
        try:
            chunks = self.client.collections.get(name_collection)
            response = chunks.query.hybrid(
                query=query,
                return_metadata=wq.MetadataQuery(score=True, explain_score=True),
                query_properties=query_properties,
                fusion_type=wq.HybridFusion.RELATIVE_SCORE,
                limit=limit,
                alpha=0.4,
            )

            return response.objects
        except Exception as e:
            if "You exceeded your current quota, please check your plan and billing
details." in str(e):
                print("OpenAi API has no payment")
            else:
                print("Error in hybrid_search_in_collection: ", e)
        finally:
            self._disconnect()
```

Для забезпечення коректної роботи при повторному запуску пошуку було реалізовано метод для повного очищення колекції. Це дозволяє уникнути накопичення непотрібних об’єктів між різними сесіями (лістинг 4.12).

Лістинг 4.12 – метод для видалення об’єктів

```
      def batch_delete_all_from_collection(self, name_collection: str) -> None:
        vectors = self.get_all_vectors(name_collection)
        uuids_to_delete = [vector.uuid for vector in vectors]

        self._connect()
```

```
    try:
        if uuids_to_delete:
            collection = self.client.collections.get(Const.COLLECTION_NAME)
            collection.data.delete_many(
                where=Filter.by_id().contains_any(uuids_to_delete)
            )
            print(f"Deleted {len(uuids_to_delete)} objects from the collection.")
        else:
            print("No objects found to delete.")
    except Exception as e:
        print(f"Error in batch_delete_all_from_collection: {e}")
    finally:
        self._disconnect()
```

4.4 Серверна частина та API

Архітектура вебзастосунку побудована з використанням мови програмування Python та асинхронного вебфреймворку FastAPI, який забезпечує високу продуктивність та зручну інтеграцію зі схемами OpenAPI для опису маршрутів. Для взаємодії з базою даних використовується бібліотека pyodbc, що дозволяє виконувати безпосередні SQL-запити до Microsoft SQL Server. Крім того, для реалізації інтелектуального пошуку використовується окремий модуль векторної бази даних.

Застосунок має лише один маршрут, а саме POST запит на кінцеву точку «/search_street/» (лістинг 4.8), який примає назву міста та вулиці, яку потрібно шукати. В результаті повертається вулиця, яку було знайдено, або повідомлення «Вулицю не знайдено», якщо такої намає в базі.

Лістинг 4.8 – кінцева точка search_street

```
    @router.post("/search_street/", tags=["Search Street"])
async def search_street(search_street_request: SearchStreetRequest):
    search_result = search_process(search_street_request)
    return {
        "result": search_result
    }
```

Після отримання запиту, виконується нормалізація назви вулиці, а саме відокремлення типу вулиці, якщо вказано, від назви. Даний формат передається до функції, яка запускає пошук у базі даних, виконавши запит SQL. Спочатку у запиті отримується Id міста, яке вказав користувач (лістинг 4.9).

Лістинг 4.9 – отримання Id міста з бази

```
      SELECT @cityId = c.Id
FROM [Diploma].[dbo].Cities c
WHERE c.Name = @city
```

Далі створюється тимчасова таблиця ##temp, куди записуються вулиці міста з розділенням їх на workstreet — частину без типу, тобто без «вул.», «просп.» тощо. Якщо задано тип вулиці, відбираються лише ті назви, які його містять (лістинг 4.10).

Лістинг 4.10 – отримання списку вулиць міста

```
      drop table if exists ##temp
select s.Id, s.Name as Name, reverse(substring(reverse(s.Name), CHARINDEX(' ',
reverse(s.Name)) + 1, 100)) as workstreet, s.city_id as city_id into ##temp
From [Diploma].[dbo].Streets s
where city_id = @cityId and case when @streettype != '' then CHARINDEX(@streettype,
s.Name) else 1 end > 0
```

Із частини назви workstreet, що потенційно містить старі назви в дужках, отримуються нова назва та стара, що у дужках (лістинг 4.11). Отриманні назви записуються у тимчасову таблицю.

Лістинг 4.11 – відокремлення старої назви від нової

```
      drop table if exists ##temp1
select Id, w.Name, workstreet, trim(replace(value, ')', '')) as PartName, ordinal as
pos, w.city_id as city_id into ##temp1
From ##temp w
cross apply [Diploma].[dbo].DelimitedSplit8K(w.workstreet, '(')
drop table ##temp
      drop table if exists #work
select w1.Id, w1.Name, w1.PartName as namenew, LEN(w1.PartName)-
LEN(REPLACE(w1.PartName, ' ', ''))+1 as wordsnew,  w2.PartName as nameold,
LEN(w2.PartName)-LEN(REPLACE(w2.PartName, ' ', ''))+1 as wordsold, w1.city_id as
city_id into #work
from ##temp1 w1
left join ##temp1 w2 on w2.Id = w1.Id and w2.pos = 2
where w1.pos = 1
drop table ##temp1
```

Наступним кроком виконується обчислення коефіцієнта схожості назв з введеною користувачем вулицею (лістинг 4.12).

Лістинг 4.12 – обчислення схожості

```
      drop table if exists #result
      ;with cte as (
          -- Нова назва
          select w.Id, w.Name, d.value as dVal, d.ordinal as dPos,
```

```
                d1.value as d1Val, d1.ordinal as d1Pos,
                [Diploma].[dbo].[fn_calculateJaroWinkler](d.value,    d1.value)    as
dist,
                ABS(w.wordsnew - @wordslen) as worddiff,
                wordsnew as words, 'new' as tp, w.city_id as city_id
        from #work w
        cross apply DelimitedSplit8K(w.namenew, ' ') d
        cross apply DelimitedSplit8K(@street, ' ') d1
        where w.namenew is not null

        union

        -- Стара назва
        select w.Id, w.Name, d.value as dVal, d.ordinal as dPos,
                d1.value as d1Val, d1.ordinal as d1Pos,
                [Diploma].[dbo].[fn_calculateJaroWinkler](d.value,    d1.value)    as
dist,
                ABS(w.wordsold - @wordslen) as worddiff,
                wordsold as words, 'old' as tp, w.city_id as city_id
        from #work w
        cross apply DelimitedSplit8K(w.nameold, ' ') d
        cross apply DelimitedSplit8K(@street, ' ') d1
        where w.namenew is not null
    )
    select * into #result from cte
```

Знаходимо 20 найбільш схожих за назвою вулиць, на ту що надав користувач (лістинг 4.13). Результати групуються за Id вулиці та словом, обчислюється загальний коефіцієнт схожості, який враховує метрику відстані, і враховується бонус за співпадіння кількості слов у назві.

Лістинг 4.13 – вибірка найрелевантніших вулиць

```
    drop table if exists #workWithStreetsNew;
    ;with cte as (
        select *, ROW_NUMBER() OVER (partition by Id, dVal order by dist desc) as
pos
        from #result
        where dVal is not null
    )
    select top 20
        c.Id,
        c.Name,
        c.tp,
        SUM(c.dist/(c.worddiff+c.words)) +
        (
            case
                when ABS(c.worddiff) > 0 then MAX(c.dist) * 0.4
                else 0
            end
        ) as koef,
        ci.Name as CityName,
        reverse(substring(reverse(c.Name), 1,
            case when CHARINDEX(' ', reverse(c.Name)) > 0
                 then CHARINDEX(' ', reverse(c.Name)) - 1
                 else LEN(c.Name)
            end
```

```
    )) as StreetType
into #workWithStreetsNew
from cte c
left join [Diploma].[dbo].Cities ci on ci.Id = c.city_id
where c.pos = 1
group by c.Id, c.Name, c.tp, ci.Name, c.worddiff
order by koef desc
```

Якщо користувач ввів назву вулиці та не вказав її тип, може бути таке, що в отриманих назвах буде декілька правильних назв, але з різним типом. Для цього виконується додаткова перевірка та повертається відповідний маркер (лістинг 4.14).

Лістинг 4.14 – перевірка на наявність однакових вулиць з різним типом

```
if @streettype = ''
begin
    select @distinctStreetTypes = COUNT(DISTINCT StreetType)
    from #workWithStreetsNew
    where CHARINDEX(@street, Name) > 0;
end;

if @streettype = '' and @distinctStreetTypes > 1
begin
    select CAST(-1 AS smallint) as Id, '' as Name, '' as CityName;
end
```

Якщо перевірку було пройдено, повертаються отриманні вулиці (лістинг 4.15).

Лістинг 4.15 – повернення результату

```
select top 20 Id, Name, CityName as CityName, koef
from #workWithStreetsNew
where koef > 0.4
order by koef desc;
```

Після отримання результатів пошуку, код Python записує данні класом у масив (лістинг 4.16) та перевіряє результат на наявність помилки однакових вулиць (лістинг 4.17).

Лістинг 4.16 – запис даних у масив

```
  while True:
try:
    rows = self.cursor.fetchall()
    break
except pyodbc.ProgrammingError:
    if not self.cursor.nextset():
        rows = []
        break
```

```
streets = [Street(street_id=row[0], name=row[1], city=row[2]) for row in rows]

return streets
```

Лістинг 4.17 – повернення помилки, при однакових вулицях різних типів

```
       if selected_streets[0].id == -1:
    raise HTTPException(status_code=404, detail="Знайдено декілька вулиць, вкажіть
тип")
```

Після отримання 20 найбільш схожих вулиць, починається друга фаза пошуку, через мовну модель. Для початку, виконується нормалізація об’єктів до JSON формату (лістинг 4.18).

Лістинг 4.18 – нормалізація даних

```
      norm_streets = []
for street in streets:
    norm_streets.append({
        "street": street.name,
        "street_id": str(street.id)
    })
```

Далі додаємо вулиці у векторну базу даних Weaviate та виконуємо пошук через штучний інтелект, передавши на вхід вулицю, яку ввів користувач (лістинг 4.19).

Лістинг 4.19 – пошук вулиці штучним інтелектом

```
      vector_db.batch_add_to_collection(Const.COLLECTION_NAME, norm_streets)
results = vector_db.hybrid_search_in_collection(
    query=request_street.street,
    name_collection=Const.COLLECTION_NAME,
    query_properties=['street'],
    limit=1
)
if not results: raise HTTPException(status_code=404, detail="Вулицю не знайдено")
search_res = results[0].properties
```

Отримана вулиця повертається користувачу, включаючи назву міста, назву вулиці та Id вулиці у базі даних. Якщо вулицю не було знайдено, повертається повідомлення «Вулицю не знайдено» (рис. 4.1.- 4.4).

На рисунку 4.2 показано приклад JSON-запиту, в якому зазначено місто "м. Харків" та назву вулиці з помилкою "просп. Лювіга Свабоди".

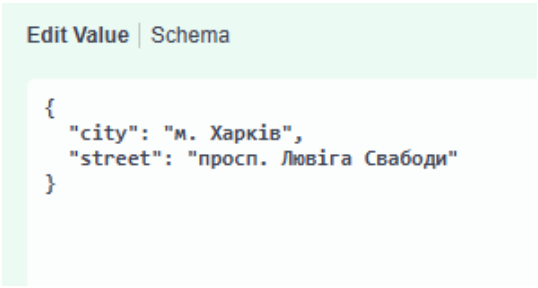

Рисунок 4.2 – Введення корекних даних для пошуку

Рисунок 4.3 – Результат виконання пошуку

На рисунку 4.4 показано приклад JSON-запиту, в якому зазначено місто "м. Харків" та назву вулиці, якої точно немає у базі даних, "string".

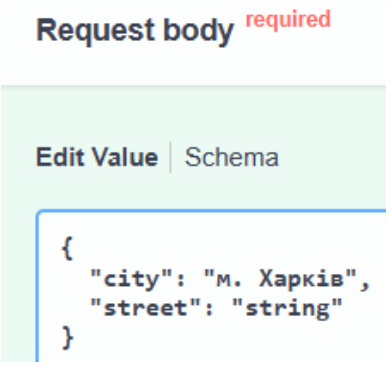

Рисунок 4.4 – Введення не правильного запиту

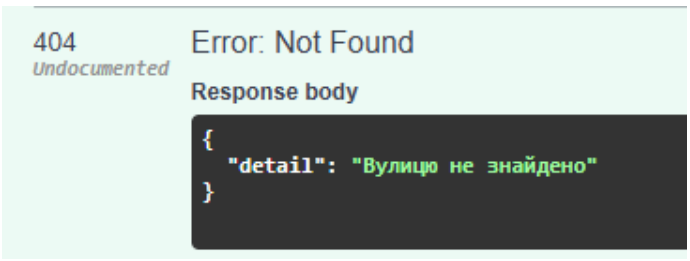

Рисунок 4.5 – Результат виконання пошуку

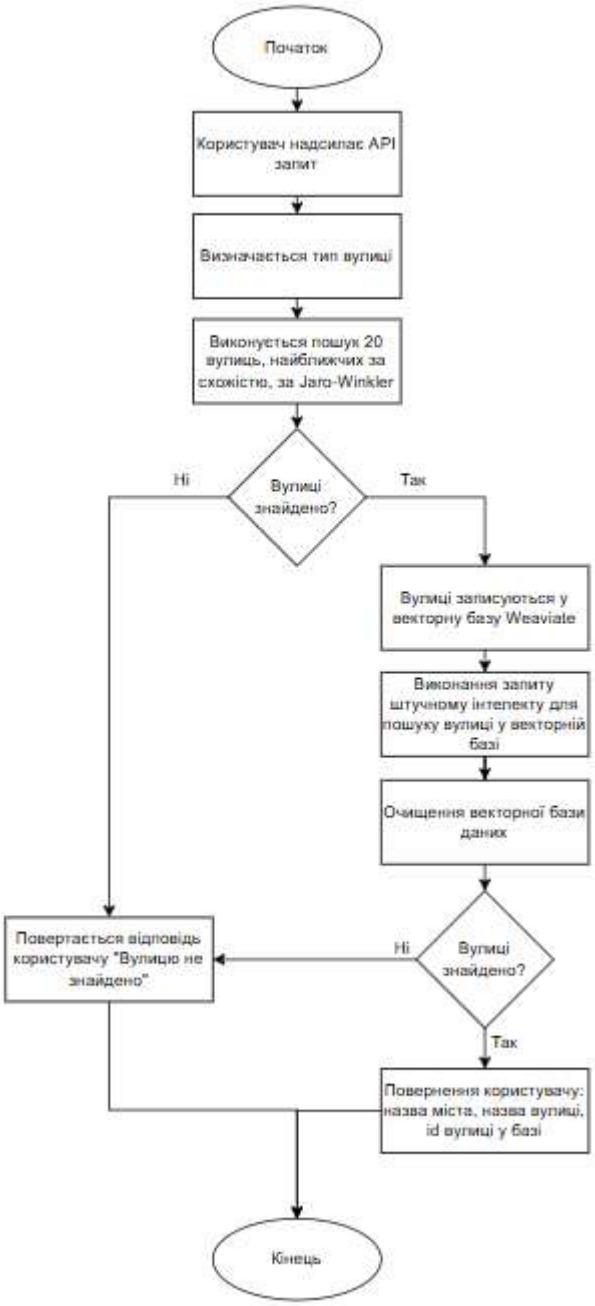

Рисунок 4.6 – Алгоритм виконання

Алгоритм реалізує покрокову логіку обробки користувацького запиту до REST API для нечіткого пошуку вулиці за введеною адресою. Він включає попереднє лексичне фільтрування та семантичне зіставлення у векторному просторі.

Цей алгоритм дозволяє забезпечити: поєднання лексичного і семантичного пошуку; підтримку орфографічних і форматних помилок; швидкодію та розширюваність завдяки використанню Weaviate та Jaro-Winkler; чистоту векторної бази через тимчасове завантаження даних.

ВИСНОВКИ

У ході цієї роботи було досліджено проблему пошуку адрес у базі даних, де введені дані можуть відрізнятися від збережених у базі. Для вирішення цієї проблеми було запропоновано гібридний підхід, який поєднує в собі методи мовних моделей та векторних представлень. Цей підхід дозволяє поєднати точність та семантичне розуміння мовних моделей з ефективністю та швидкістю векторних представлень.

Під час аналізу було розглянуто різні типи моделей та алгоритмів навчання, такі як навчання з учителем, ненавчане навчання, підсилене навчання та їх комбінації. Кожен з цих підходів має свої переваги та недоліки, і вибір конкретного підходу залежить від специфіки задачі, доступності даних та ресурсів.

Також було розглянуто різні види архітектур моделей, включаючи згорткові нейронні мережі, рекурентні нейронні мережі, трансформатори та векторні представлення, такі як Word2Vec, GloVe, FastText, Jaro-Winkler та BERT. Кожен з цих методів має свої особливості та можливості, які можуть бути використані в гібридній моделі для досягнення оптимальних результатів.

На основі проведених досліджень можна зробити висновок, що для ефективного пошуку адрес у базі даних найбільш оптимальним є використання гібридного підходу, який поєднує в собі різні методи та архітектури моделей. Цей підхід дозволяє забезпечити високу точність та швидкість пошуку адрес, що є важливими аспектами для багатьох застосувань, таких як логістика, маркетинг та державні служби.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Artificial Neural Network. URL: https://www.sciencedirect.com/topics/earth-and-planetary-sciences/artificial-neural-network (дата звернення: 03.06.2025).

2. Machine Learning Algorithm. URL: https://www.sciencedirect.com/topics/computer-science/machine-learning-algorithm (дата звернення: 03.06.2025).

3. Hybrid Modeling. URL: https://www.sciencedirect.com/topics/computer-science/hybrid-modeling (дата звернення: 03.06.2025).

4. Artificial Neural Networks Applications and Algorithms. URL: https://www.xenonstack.com/blog/artificial-neural-network-applications (2025)

5. Representation of Vector. URL: https://www.cuemath.com/geometry/representation-of-vector/ (дата звернення: 03.06.2025).

6. What are vector embeddings? URL: https://www.elastic.co/what-is/vector-embedding (дата звернення: 03.06.2025).

7. Dense Neural Networks: Understanding Their Structure and Function. URL: https://datascientest.com/en/dense-neural-networks-understanding-their-structure-and-function (дата звернення: 03.06.2025).

8. Introduction to Recurrent Neural Network. URL: https://www.geeksforgeeks.org/introduction-to-recurrent-neural-network/ (дата звернення: 03.06.2025).

9. Word2Vec. URL: https://serokell.io/blog/word2vec (дата звернення: 03.06.2025).

10. Discovering Trends in BERT Embeddings of Different Levels for the Task of Semantic Context Determining. URL: https://towardsdatascience.com/discovering-trends-in-bert-embeddings-of-different-levels-for-the-task-of-semantic-context-268733fdb17e (дата звернення: 03.06.2025).