**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**
**«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»**

Навчально-науковий інститут атомної та теплової енергетики

Кафедра інженерії програмного забезпечення в енергетиці

ДО ЗАХИСТУ ДОПУЩЕНО
В.о. завідувача кафедри
_____Олександр КОВАЛЬ
«_____»_____2023 р.

# Дипломна робота

**на здобуття ступеня бакалавра**
**за освітньо-професійною програмою «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем і веб-технологій»**
**спеціальності 121 Інженерія програмного забезпечення**
**на тему: «Програмний веб-додаток інформаційно-семантичного пошуку з використанням онтології RDF та OWL»**

Виконав:
студент IV курсу, групи ТВ-91
Продан Андрій Григорович _____
(прізвище, ім'я, по батькові) (підпис)

Керівник:
професор кафедри ІПЗЕ, професор, д.т.н. Коваль О. В. _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент:
Старший науковий співробітник, к. т. н. В'ячеслав Сенченко _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2023

**Національний технічний університет України**

**«Київський політехнічний інститут імені Ігоря Сікорського»**

Навчально-науковий інститут атомної та теплової енергетики
Кафедра інженерії програмного забезпечення в енергетиці
Рівень вищої освіти перший (бакалаврський)
Спеціальність 121 Інженерія програмного забезпечення
Освітньо-професійна програма «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем і веб-технологій»

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Олександр КОВАЛЬ
(підпис)
«___»_____ 2023 р.

**ЗАВДАННЯ**
**на дипломну роботу студенту**

Продан Андрій Григорович
(прізвище, ім'я, по батькові)

1. Тема роботи
Програмний веб-додаток інформаційно-семантичного пошуку з використанням онтології RDF та OWL
керівник роботи Коваль О. В.
( прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від "__"_____ 2023 року №_____
2. Строк подання студентом роботи _____
3. Вихідні дані до роботи_____
_____
_____
_____
_____
4. Зміст (дипломної роботи) пояснювальної записки (перелік завдань, які потрібно розробити) реалізувати програму для пошуку проіндексованих в базі даних веб-ресурсів за допомогою моделей машинного навчання. Додати використання RDF графів для більш детального зв'язування статей між собою та для надавання цим статтям більшого контексту. Розробити систему для редагування зв'язків в онтології.

5. Перелік ілюстративного матеріалу робота містить 13 ілюстрацій.

6. Дата видачі завдання «30» вересня 2022 р.

КАЛЕНДАРНИЙ ПЛАН

| № з/п | Назва етапів виконання дипломної роботи | Строки виконання етапів роботи | Примітка |
|---|---|---|---|
| 1 | Отримання завдання | 30.09.22 | Виконано |
| 2 | Дослідження предметної області | 02.10.22 - 01.02.23 | Виконано |
| 3 | Постановка вимог до проєктування системи | 02.02.23-02.03.23 | Виконано |
| 4 | Дослідження існуючих рішень | 3.03.23-16.04.23 | Виконано |
| 5 | Розробка програмного продукту | 17.04.23-17.05.23 | Виконано |
| 6 | Тестування | 17-18.05.23 | Виконано |
| 7 | Захист програмного продукту | 19.05.23 | Виконано |
| 8 | Оформлення дипломної роботи | 22.05.23-04.06.23 | Виконано |
| 9 | Передзахист | 06.06.23 | Виконано |
| 10 | Захист | 19.06.2023 | Виконано |

Студент _____ Продан Андрій Григорович
(підпис) (ім'я, прізвище)

Керівник роботи _____ Коваль Олександр Васильович
(підпис) (ім'я, прізвище)

# РЕФЕРАТ

**Структура та обсяг дипломної роботи.** Робота містить 57 сторінок, 13 рисунків, 2 додатки та 15 посилань.

**Метою роботи** є розробка та реалізація програмного веб-додатка, який буде здатний забезпечувати ефективний пошук інформації з використанням семантичних методів та технологій.

Для досягнення поставленої мети виконано такі завдання:

- Розроблено архітектуру веб-додатка: Визначено необхідні компоненти та функціональні модулі веб-додатка, які будуть відповідати за обробку запитів користувачів та виконання пошуку інформації.
- Інтегровано sentence-transformers: Використано бібліотеку sentence-transformers для реалізації модуля векторного представлення тексту.
- Імплементовано FAISS: Використано бібліотеку FAISS (Facebook AI Similarity Search) для реалізації пошукового двигуна, який забезпечить ефективний пошук схожих векторів. Цей модуль буде відповідати за швидкий і точний пошук інформації на основі векторів.
- Розроблено онтології RDF та OWL: Створено та імплементовано онтології RDF та OWL (Web Ontology Language), які дозволять структурувати та семантично описувати інформацію, що зберігається у веб-додатку.

**Практичне значення одержаних результатів** полягає в тому, що розроблений веб-додаток забезпечує покращення ефективності пошуку інформації. Застосування методів векторного представлення тексту, онтологій RDF та OWL, а також пошукового двигура FAISS дозволяє користувачам швидко та точно знаходити релевантну інформацію.

**Ключові слова:** інформаційно-семантичний пошук, sentence-transformers, FAISS, онтології RDF, OWL, ефективність пошуку, векторне представлення тексту.

# ABSTRACT

**Structure and scope of the thesis.** The work contains 58 pages, 13 figures, 2 appendices and 15 references.

**The purpose of the work** is to develop and implement a web application that will be able to provide effective information retrieval using semantic methods and technologies.

To achieve this goal, the following tasks were performed:

- The architecture of the web application is developed: The necessary components and functional modules of the web application that will be responsible for processing user requests and performing information retrieval have been identified.

- Integrated sentence-transformers: The sentence-transformers library was used to implement the vector text representation module.

- Implemented FAISS: The FAISS (Facebook AI Similarity Search) library was used to implement a search engine that will provide an efficient search for similar vectors. This module will be responsible for fast and accurate search for information based on vectors.

- RDF and OWL ontologies were developed: The RDF and OWL (Web Ontology Language) ontologies have been created and implemented, which will allow structuring and semantically describing the information stored in a web application.

**The practical significance** of the results obtained is that the developed web application provides an improvement in the efficiency of information retrieval. The use of vector text representation methods, RDF and OWL ontologies, and the FAISS search engine allows users to quickly and accurately find relevant information.

**Keywords**: information and semantic search, sentence-transformers, FAISS, RDF, OWL ontologies, search efficiency, vector text representation.

# ЗМІСТ

# ВСТУП

Сьогоднішній розвиток інформаційних технологій та швидкість доступу до мережі Інтернет змінює спосіб, яким ми збираємо, обробляємо та шукаємо інформацію. Семантичний веб є однією з найбільш обіцяючих технологій, яка дає можливість знайти відповідні результати швидко та точно.

У даній дипломній роботі досліджується тема "Семантичний веб пошук сайтів". Основна мета роботи полягає в розробці веб-застосунку, який дозволяє використовувати семантичну пошукову систему для пошуку сайтів або, наприклад, наукових матеріалів.

Актуальність теми обумовлена швидким розвитком Інтернету та необхідністю ефективного та точного пошуку інформації. На сьогоднішній день багато пошукових систем працюють на основі ключових слів та не можуть ефективно знаходити інформацію за контекстом запиту користувача. Розроблений у цій роботі веб-застосунок дає можливість здійснювати семантичний пошук, що значно покращує якість пошуку та забезпечує користувачам більш точні та відповідні результати.

# 1 ЗАДАЧА СЕМАНТИЧНОГО ПОШУКУ

Однією з основних задач семантичного пошуку є забезпечення користувачів інформацією, яка відповідає їх потребам та інтересам, навіть якщо їх запит не містить точної фрази або ключових слів, що пов'язані з цією інформацією. Такий пошук базується на аналізі контексту та семантики запиту, а не просто на відповідності окремих слів.

## 1.1 Перевага над синтаксичним пошуком

Синтаксичний пошук - це пошук інформації на основі синтаксичної відповідності між запитом користувача та документами в базі даних. У цьому випадку, результати пошуку пов'язані з точністю відповідності синтаксичних конструкцій запиту. Наприклад, якщо користувач шукає фразу "мобільний телефон Samsung", синтаксичний пошук буде шукати документи, що містять ці самі слова. В той час, коли на релевантність результатів семантичного пошуку впливає порядок слів та їх зв’язок між собою.  Синтаксичний пошук не враховує семантичного змісту запиту та його контексту. У деяких випадках, синтаксичний пошук може бути корисним, коли користувач точно знає, яку інформацію він шукає та які слова чи конструкції повинні міститися в результаті. Однак, в більш складних запитах, синтаксичний пошук може приводити до надмірної чи недостатньої кількості результатів, що не відповідають потребам користувача.

Однією з основних задач семантичного пошуку є покращення ефективності та точності пошуку в порівнянні з синтаксичним пошуком. Синтаксичний пошук зазвичай ґрунтується на ключових словах та певних правилах граматики, і він може бути неефективним, коли користувач шукає конкретну інформацію, але використовує різні форми тих самих слів або відповідність частинам запиту не точна.

Семантичний пошук використовує технології машинного навчання та обробки природної мови, щоб зрозуміти не тільки конкретні слова, але і їх контекст, і знайти релевантну інформацію, що задовольняє потреби користувача. Це дозволяє покращити якість пошуку та точність результатів, а також забезпечує можливість розуміти відносини між словами та допомагає у вирішенні проблем, пов'язаних з синонімічністю та полісемією.

Задача семантичного пошуку полягає в розробці алгоритмів, що забезпечують більш точний та ефективний пошук, використовуючи технології машинного навчання та обробки природної мови, а також розробці відповідних моделей та архітектур, що можуть забезпечити найкращу продуктивність та точність результатів.

## 1.2 Принципи семантичного пошуку

Семантичний пошук базується на декількох принципах, які допомагають забезпечити точність та релевантність результатів пошуку. Основні принципи семантичного пошуку включають:

- аналіз тексту: семантичний пошук аналізує текст запиту та документів, що містяться в базі даних, для виявлення семантичних зв'язків та контексту;
- використання семантичних моделей: для розуміння смислу тексту та виявлення відносин між словами та поняттями, в семантичному пошуку використовуються семантичні моделі, такі як глибинні нейронні мережі;
- застосування машинного навчання: семантичний пошук може використовувати методи машинного навчання для покращення релевантності результатів пошуку;
- ранжування результатів: семантичний пошук використовує алгоритми ранжування результатів, щоб показувати найбільш відповідні результати першими;

- контекстуальний аналіз: в семантичному пошуку враховується контекст запиту та документу, що дозволяє забезпечити більш точні результати пошуку;
- обробка природної мови: для зручності користувачів, які використовують звичну мову для формулювання запитів, семантичний пошук може використовувати технології обробки природної мови (Natural Language Processing - NLP);
- робота з великими об'ємами даних: для обробки великих об'ємів даних, що містяться в базі даних, в семантичному пошуку можуть використовуватися розподілені системи зберігання та обробки даних.

## 1.3 Щільні та розріджені вектори

Напевно, немає більш значущого внеску у сферу обробки природної мови (NLP) ніж використання векторного представлення мови. Початок NLP революції насправді почався з випуску word2vec у 2013 році [1].

Word2vec є одним з найбільш визначних та ранніх прикладів щільного векторного представлення тексту. Але з того часу, розвиток методів представлення мови швидко прогресував.

Перша запитання, яке виникає, це чому нам потрібно використовувати векторне представлення тексту? Просто кажучи, це необхідно для того, щоб комп'ютер міг розуміти текст, який зрозумілий для людини. Нам потрібно перетворити текст у формат, який машина може обробити.

Мова є дуже складною та інформаційно насиченою, тому нам потрібно велику кількість даних для представлення навіть невеликого обсягу тексту. Вектори є потенційним кандидатом для цього формату.

У нас також є два варіанти представлення векторів: розріджені вектори та щільні вектори.

Розріджені вектори можуть бути збережені більш ефективно, і вони дозволяють порівнювати синтаксичні аспекти двох послідовностей. Наприклад, якщо у нас є два речення: "Олег пересів з коня в машину" і "Олег пересів з машини на коня", ми отримуємо ідеальний або практично ідеальний збіг. Незважаючи на різницю у змісті речень, вони мають однакову синтаксичну структуру (наприклад, однакові слова). Тому розріджені вектори будуть майже або повністю збігатися (залежно від підходу до побудови). [1]

Якщо розріджені вектори використовуються для представлення синтаксичної структури тексту, то щільні вектори можна розглядати як числове відображення семантичного значення. Зазвичай ми беремо слова і перетворюємо їх у високорозмірні щільні вектори. Абстрактне значення і взаємозв'язок між словами кодуються у вигляді послідовності чисел.

Розріджені вектори отримали свою назву через те, що вони містять незначну кількість інформації. Зазвичай, ми бачимо тисячі нулів, перш ніж зустріти кілька одиниць, які несуть значиму інформацію. Такі вектори можуть мати велику кількість вимірів, часто десятки тисяч.

Щільні вектори також мають високу розмірність (зазвичай мають 784 виміри, але ця кількість може бути більшою або меншою). Кожен вимір містить відповідну інформацію, яка була визначена нейронною мережею. Стискання таких векторів є складнішим завданням, тому вони зазвичай вимагають більше пам'яті. На рисунку 1.1 зображено приблизну різницю між цими типами векторів.[1]

Уявімо, що ми генеруємо щільні вектори для кожного слова у книзі, зменшуємо їх розмірність, а потім відображаємо їх у тривимірному просторі. Завдяки цьому, ми зможемо виявити взаємозв'язки між словами. На рисунку 1.2

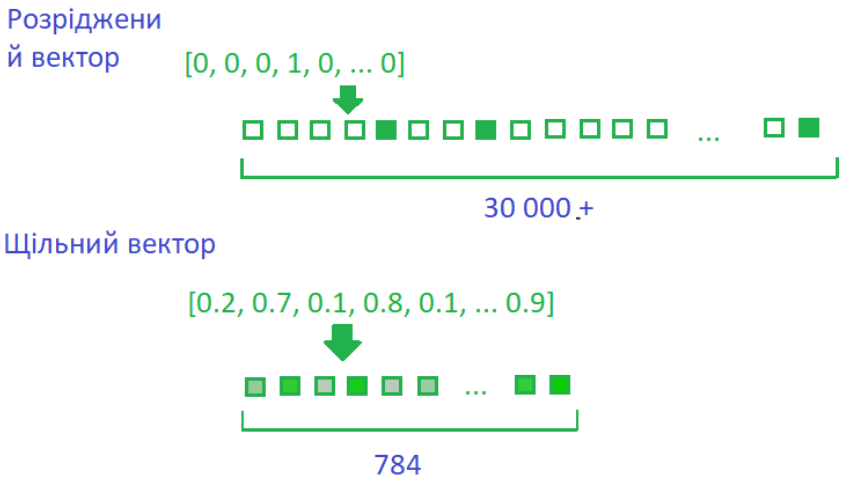


Рисунок 1.1 - Різниця між розрідженим та щільним векторами

зображено, як слова, що пов’язані з терміном “історія” зображені близько в спроектованому просторі.

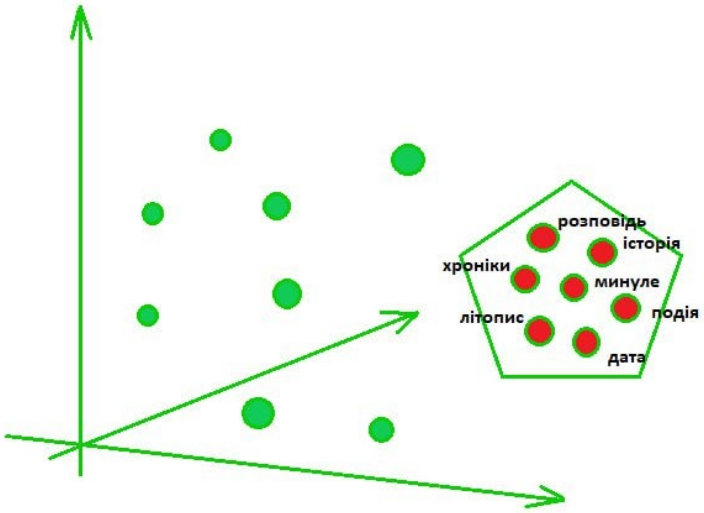


Рисунок 1.2 – Схожість векторів у тривимірному просторі

Це досягається завдяки складним нейронним мережам, які аналізують великі обсяги текстових даних, виявляють патерни і перетворюють їх у щільні вектори.

Отже, можна розглядати розріджені і щільні вектори як представлення синтаксису і семантики мови відповідно.

## 1.4 Генерування щільних векторів

Існує широкий спектр технологій для створення щільних векторів, серед них декілька особливо цікавих та цінних:

1. Методи "2vec": Ці методи, зокрема word2vec та doc2vec, використовують нейронні мережі для створення щільних векторів слів та документів відповідно.
2. Трансформери для речень: Трансформери, такі як BERT (Bidirectional Encoder Representations from Transformers), дозволяють отримати щільні вектори для речень, використовуючи контекстуальну інформацію.
3. Пошук щільних уривків (DPR): Цей метод використовує поєднання нейронних мереж та пошуку, щоб створити щільні вектори для коротких уривків тексту, які мають високу релевантність до запитів користувачів.
4. Трансформери для бачення (ViT): Ці трансформери застосовуються до задач обробки зображень і дозволяють створювати щільні вектори для зображень, що містять семантичну інформацію про їх зміст.

За допомогою цих технологій можна отримати якісні щільні вектори, які використовуються в різних областях, включаючи обробку природної мови та комп'ютерне зорове сприйняття.

### 1.4.1 Word2Vec

Незважаючи на наявність більш розширених технологій для створення вбудовувань, неможливо проігнорувати word2vec при розгляді щільних векторів. Він може не бути першим, але є першим широко використовуваним інструментом для створення щільних вбудовувань. Це стало можливим завдяки двом факторам: по-перше, модель word2vec була дуже ефективною, і по-друге, був розроблений відповідний набір інструментів, що дозволяє легко навчати або використовувати передньо натреновані вбудовування word2vec.

У моделі word2vec вбудовування слів створюються на основі контексту речення. Конкретне слово (перетворене на вектор з однократним кодуванням) порівнюється з навколишніми словами за допомогою нейронної мережі кодера-декодера для створення векторного представлення слова.

Skip-gram модель word2vec, намагається передбачити навколишні слова (контекст) для заданого слова. Після навчання ми відкидаємо ліву і праву частини, залишаючи лише середній щільний вектор. Цей вектор представляє слово зліва від діаграми і може бути використаний для вбудовування цього слова в наступні мовні моделі.

Крім того, є модель Continuous Bag of Words (CBOW), яка працює у зворотному напрямку і намагається передбачити слово на основі його контексту.

Як і CBOW, так і skip-gram побудовані на основі мережі кодерів-декодерів та генерують щільний вектор вбудовування з середнього прихованого шару.

На основі цього принципу Міколов та інші створили відомий приклад векторної арифметики "Король - Чоловік + Жінка = Королева", що демонструє потужність мовного векторного представлення.[1]

Word2vec був революційним у досягненнях у сфері NLP. Однак, коли мова йшла про представлення довших текстових фрагментів за допомогою окремих

векторів, word2vec став недостатнім. Він міг кодувати окремі слова або n-грами, але не більше того, що унеможливлювало представлення довгих текстових уривків одним вектором.

Для ефективного порівняння довгих фрагментів тексту нам потрібно мати їх представлення у вигляді одного вектора. Ця обмеженість спонукала до появи розширених методів вбудовування, таких як sentence2vec і doc2vec.

Зараз є значно кращі технології для побудови цих щільних векторів. Таким чином, хоча "2vec" був початковим кроком, його рідко використовують в сучасних дослідженнях

### 1.4.2 Подібність речень

Трансформерні моделі стали домінуючими сучасними мовними моделями завдяки своїм інформаційно насиченим щільним векторам.

Один з найвідоміших трансформерних архітектур - BERT, продовжує побудову вбудовування на рівні слова (або лексеми), аналогічно до word2vec. Але завдяки глибшим мережам і механізму уваги, вбудовування в BERT стають багатшими та здатними враховувати контекст слів.[2]

Механізм уваги дозволяє BERT приділяти пріоритетність словам контексту, що мають найбільший вплив на конкретне вбудовування. Це досягається шляхом звернення уваги BERT на конкретні слова, залежно від їх контексту.

Проте виникає проблема, оскільки ми хочемо порівнювати речення, а не окремі слова. Вбудовування BERT створюються для кожного токена, а не для цілих речень. Тому нам потрібен єдиний вектор, який може представляти ці речення або абзаци, наприклад, як у sentence2vec.

Перша трансформерна модель, спеціально розроблена для цієї мети, називається Sentence-BERT (SBERT) і є модифікацією BERT [3].

BERT (і SBERT) використовують токенізатор WordPiece, де кожне слово може бути розбите на один або кілька токенів. SBERT дозволяє створювати єдине векторне вбудовування для послідовностей, що містять до 128 токенів, відкидаючи решту.

Хоча це обмеження може не бути ідеальним для довгих текстів, воно вистачає для порівняння невеликих речень або абзаців. Також варто зазначити, що багато сучасних моделей можуть працювати з більшими послідовностями.

## 1.5 Можливості моделей BERT і GPT

Трансформатори спричинили повну перебудову в області обробки природної мови (NLP). Попередньо, для перекладу та класифікації мови використовувалися рекурентні нейронні мережі (RNN), але їхнє розуміння мови було обмеженим і часто призводило до незначних помилок, а також ускладнювало зв'язність великих текстових блоків.

Починаючи з представлення першої моделі-трансформера в статті "Увага - це все, що вам потрібно" в 2017 році, NLP перейшло від використання RNN до використання нових моделей, таких як BERT і GPT. Ці нові моделі можуть відповідати на питання, генерувати статті, здійснювати надзвичайно інтуїтивний семантичний пошук та виконувати багато інших завдань.[4]

Цікаво, що для багатьох завдань останні шари цих моделей такі ж самі, як у RNN - часто це пари нейронних мереж зі зворотнім зв'язком, що генерують прогнози моделі.

Змінилися лише вхідні дані для цих шарів. Щільні вбудовування, створені трансформаторними моделями, містять велику кількість інформації, що

дозволяє отримати значну продуктивність, навіть застосовуючи ті ж самі зовнішні шари.

Ці вбудовування речень, що стають все багатшими, можна використовувати для швидкого порівняння схожості речень у різних сценаріях використання. Наприклад:

- Семантична текстова схожість (STS): порівняння пар речень. Це може бути корисним для виявлення закономірностей у наборах даних і, найчастіше, використовується як бенчмарк.
- Семантичний пошук: пошук інформації на основі семантики. Задаючи запит у вигляді речення, ми можемо знайти найбільш схожі записи, що дозволяє здійснювати пошук за концепціями, а не конкретними словами.
- Кластеризація: речення можна кластеризувати для моделювання тем, що є корисним.

### 1.5.1 Мережі кодер-декодер

Трансформатори є непрямими нащадками попередніх моделей RNN. Ці старі рекурентні моделі зазвичай будувалися з багатьох рекурентних одиниць, таких як LSTM або GRU.

У машинному перекладі ми зустрічаємо мережі кодерів-декодерів. Перша модель кодує мову оригіналу у вектор контексту, а друга - декодує його в мову перекладу.

Проблема полягає в тому, що ми створюємо обмежену точку передачі інформації між двома моделями. Ми генеруємо велику кількість інформації протягом декількох часових кроків і намагаємося передати її через одне з'єднання. Це обмежує продуктивність кодера-декодера, оскільки значна

частина інформації, створена кодером, втрачається, перш ніж досягне декодера.[5]

Механізм уваги був запропонований для вирішення проблеми обмеженої точки передачі. Він пропонує альтернативний шлях для проходження інформації. Однак, він не перевантажує процес, оскільки зосереджується лише на найбільш важливій і релевантній інформації.

За допомогою механізму уваги передається вектор контексту з кожного часового кроку (створюючи вектори анотацій), що дозволяє уникнути обмеженої точки передачі інформації, а також краще зберігати інформацію в довгих послідовностях.

Під час процесу декодування модель послідовно декодує слова по одному кроці. На кожному кроці обчислюється відповідність (наприклад, схожість) між поточним словом та всіма анотаціями кодера.

Це вирівнювання призводить до надання більшого значення кодувальній анотації під час декодування. Іншими словами, механізм визначає, на які слова кодера слід звернути увагу. Усі найефективніші кодери-декодери RNN використовували цей механізм уваги.

### 1.5.2 Attention is all you need

У 2017 році була опублікована стаття з назвою "Attention Is All You Need", що стала проривом у сфері обробки природної мови (НЛП). Автори цієї статті продемонстрували, що можна здобути вражаючі результати без використання рекурентних нейронних мереж (RNN), просто застосувавши механізм уваги з невеликими модифікаціями.[4]

Ця нова модель, що базується на механізмі уваги, отримала назву "трансформер". Відтоді у сфері NLP відбулася повна зміна парадигми з

використання RNN на трансформери, завдяки їх високій продуктивності та надзвичайній здатності до узагальнення.

Перший трансформер вирішив проблему, яку раніше вирішували RNN, шляхом впровадження трьох ключових компонентів:

1. Позиційне кодування: Замість використання рекурентних механізмів для збереження послідовності, трансформер використовує позиційне кодування. Це досягається шляхом додавання синусоїдальних хвиль змінної амплітуди до вбудованого входу для кожної позиції у послідовності. Це дозволяє моделі зберігати інформацію про порядок слів у тексті.

2. Самоувага: Механізм самоуваги дозволяє моделі зосереджуватись на різних частинах власного контексту під час обчислення ваги для кожного слова. Вона відрізняється від традиційного механізму уваги, який зосереджується на зв'язках між кодерами та декодерами. Завдяки самоувазі, трансформер може виявляти важливі залежності між словами в контексті речення або абзацу.

3. Багатоголова увага: Можна уявляти багатоголову увагу як декілька паралельних механізмів уваги, які працюють одночасно. Використання декількох голів уваги дозволяє моделі представляти кілька наборів зв'язків між словами, замість одного набору. Це допомагає моделі краще моделювати відносини і виконувати більш складні завдання у вивченні природної мови.

### 1.5.3 Натреновані моделі

Нові трансформер-моделі узагальнюються значно краще, ніж попередні RNN-моделі, які часто створювалися спеціально для конкретних випадків використання.

З використанням трансформер-моделей можна застосовувати те саме "ядро" моделі та просто змінювати останні кілька шарів для різних випадків використання, без необхідності повторного навчання основної моделі.

Ця нова можливість призвела до розробки попередньо підготовлених моделей для обробки природної мови. Попередньо підготовлені моделі трансформерів навчаються на великій кількості навчальних даних, часто за високу вартість, наприклад, компаніями Google або OpenAI, і потім стають доступними для безкоштовного використання.

Одна з найпопулярніших попередньо підготовлених моделей це BERT (Bidirectional Encoder Representations from Transformers) або двонаправлений кодувальний представник від Transformers, розроблений Google AI.

BERT послужив основою для розробки цілої низки інших моделей та їх варіацій, таких як distilBERT, RoBERTa та ALBERT. Ці моделі застосовуються для різноманітних завдань, включаючи класифікацію тексту та відповіді на запитання.

## Висновки до розділу 1

В розділі, досліджені основні аспекти цього напрямку семантичного пошуку. Семантичний пошук має переваги над синтаксичним підходом, оскільки він здатний розуміти семантику тексту та забезпечувати більш точні результати. Розглянуто принципи семантичного пошуку, включаючи використання щільних та розріджених векторів для представлення тексту. З'ясовано, що щільні вектори можуть бути ефективно згенеровані за допомогою моделей, таких як Word2Vec, а також застосуванням алгоритмів подібності речень.

Досліджено можливості використання моделей BERT і GPT, які забезпечують високу якість представлення тексту та розуміння його семантики. З'ясовано, що щільні вектори можуть бути ефективно згенеровані за допомогою моделей, таких як Word2Vec, а також застосуванням алгоритмів подібності речень.

# 2 ІСНУЮЧІ СИСТЕМИ СЕМАНТИЧНОГО ПОШУКУ

Існує багато систем семантичного пошуку. Нижче перераховані деякі з них.

1. Google - найбільш популярна пошукова система, яка використовує семантичний пошук для виведення корисних результатів.
2. Wolfram Alpha - система, яка надає відповіді на запитання, користуючись зібраними знаннями з багатьох галузей науки та техніки.
3. IBM Watson - платформа штучного інтелекту, яка забезпечує доступ до різних сервісів, включаючи семантичний пошук.
4. Microsoft Bing - пошукова система, яка використовує технології штучного інтелекту, в тому числі семантичний пошук.
5. Semantria - платформа аналізу тексту, яка використовує семантичний аналіз для отримання детальної інформації про вміст документів.
6. Algolia - сервіс пошуку, який використовує семантичний пошук для розуміння запитів користувачів та швидкого виведення результатів.
7. Sinequa - платформа аналізу даних, яка використовує семантичний аналіз для розуміння великих обсягів текстових даних.

## 2.1 Пошукова система Google

Найбільш популярною серед цих систем є пошуковик Google. Семантичний пошук Google базується на використанні різних алгоритмів та методів, що допомагають зрозуміти не тільки самі запити користувачів, але й їхній контекст та інтенції. Компанія використовує штучний інтелект, щоб збирати та аналізувати великі обсяги даних, які допомагають зрозуміти семантику запитів користувачів та підібрати до них найбільш відповідні результати пошуку.

Один з основних алгоритмів, який використовує Google, - це PageRank, який визначає важливість сторінок в інтернеті на основі кількості посилань на них з інших веб-сайтів. Крім того, Google використовує різні алгоритми, такі як алгоритми машинного навчання та нейронні мережі, для розуміння семантики запитів та контексту.

Наприклад, якщо користувач шукає "ресторани навколо мене", то Google буде використовувати геолокацію користувача, щоб знайти ресторани в її регіоні. Крім того, Google може враховувати контекст запиту, наприклад, якщо користувач шукає "будинок", то можна зрозуміти, що вона шукає "будинок для житла", а не "будинок для продажу".Також Google використовує семантичні зв'язки між словами, щоб зрозуміти сутність запиту. Наприклад, якщо користувач шукає "Сітібанк", Google зрозуміє, що вона шукає інформацію про банк, незалежно від того, як вона написала запит.

Google використовує семантичний пошук для покращення релевантності і точності пошукових результатів. Заснований на штучному інтелекті та машинному навчанні, семантичний пошук спрямований на розуміння смислу та зв'язків між словами, фразами та поняттями, замість простого співставлення ключових слів.

Один з ключових компонентів семантичного пошуку в Google - це Knowledge Graph (Граф знань). Knowledge Graph - це велика база даних, яка містить мільярди сутностей (людей, місць, організацій тощо) та їх взаємозв'язки. Google використовує цей граф, щоб розуміти, як сутності пов'язані між собою та які зв'язки існують між ними.

## 2.2 Платформа аналізу тексту Semantria

Semantria - це платформа аналізу тексту, яка використовує методи обробки природної мови (NLP) для розуміння та аналізу текстових даних. Ця платформа надає

розширені засоби для витягування інформації, категоризації, аналізу настроїв, семантичного розпізнавання та інших завдань обробки тексту.

Одна з ключових особливостей Semantria - це його здатність до глибокого семантичного аналізу тексту. Використовуючи високопродуктивні алгоритми машинного навчання, Semantria розпізнає не тільки слова та фрази, але й їхнє значення та смисл. Це дозволяє виявляти в тексті настрої, емоції, ключові поняття, стилістику та інші семантичні аспекти.

Платформа Semantria пропонує API, яке дозволяє розробникам та дослідникам інтегрувати функціональність обробки тексту в свої додатки або системи. Використовуючи API Semantria, користувачі можуть передавати текстові дані для аналізу та отримувати повернені результати, які містять інформацію про виявлені сутності, настрої, ключові фрази та інші семантичні елементи.

Semantria також надає інтерфейс користувача, який дозволяє вручну завантажувати та аналізувати текстові дані, а також виконувати розширений аналіз та візуалізацію результатів. Користувачі можуть налаштовувати параметри аналізу, створювати словники, використовувати фільтри та робити інші налаштування для досягнення бажаних результатів.

Застосування Semantria розподіляються на різні галузі, включаючи соціальні медіа моніторинг, огляди товарів та послуг, аналіз відгуків клієнтів, стеження за репутацією брендів, аналіз ринку та багато іншого. Він може бути використаний як самостійний інструмент аналізу тексту або як компонент більших систем обробки даних та інтелектуального аналізу.

## 2.3 Платформа аналізу даних Sinequa

Sinequa - це платформа для пошуку та аналізу інформації, яка використовується в корпоративних середовищах для розуміння та ефективного

управління великими обсягами структурованих і неструктурованих даних. Ця платформа пропонує розширені можливості пошуку, аналізу та витягування знань з різноманітних джерел, таких як бази даних, документи, електронні листи, веб-сторінки, соціальні медіа та інші.

Одна з ключових особливостей Sinequa - це його здатність до розуміння семантики тексту та контекстуального розуміння. Використовуючи технології обробки природної мови (NLP), машинного навчання та штучного інтелекту (AI), Sinequa здатна аналізувати та інтерпретувати текст, розпізнавати сутності, відносини між ними та витягувати семантичну інформацію.

Платформа Sinequa надає широкий набір функціональних можливостей, включаючи:

1. Пошук та ранжування: Sinequa забезпечує потужний пошуковий двигун, який дозволяє користувачам швидко знаходити релевантну інформацію з великого обсягу даних. Він використовує розуміння контексту та семантичних зв'язків для покращення точності пошуку та ранжування результатів.

2. Аналіз тексту: Sinequa використовує NLP для аналізу та розуміння текстових документів. Він може автоматично виділяти ключові терміни, категоризувати документи за темами, виявляти настрої та емоції, розпізнавати назви організацій, людей, місць тощо. Це дозволяє здійснювати більш глибокий та докладний аналіз тексту.

3. Розумний пошук і рекомендації: Sinequa забезпечує функціонал розумного пошуку, який враховує контекст та індивідуальні потреби користувача. Він може враховувати контекст роботи, історію пошуку та інші параметри для надання персоналізованих рекомендацій та підказок.

4. Аналіз та витягування знань: Sinequa може використовувати машинне навчання та AI для автоматичного аналізу та витягування знань з текстових

даних. Він може виявляти тенденції, залежності, асоціації та інші важливі відносини в даних, що допомагає зробити інформовані рішення та прогнози.

5. Інтеграція та розширення: Sinequa може бути легко інтегрований з існуючими системами, базами даних та додатками. Він надає API для забезпечення взаємодії з іншими додатками та інфраструктурою. Крім того, Sinequa може бути розширений за допомогою модулів та настроюваних компонентів, що дозволяє підлаштовувати платформу під специфічні потреби організації.

Sinequa використовується в різних сферах, включаючи фінанси, фармацевтику, медіа, виробництво, технології та інші. Вона допомагає організаціям ефективно керувати та аналізувати великі обсяги даних, поліпшувати роботу з інформацією та приймати обґрунтовані рішення.

## Висновки до розділу 2

У цьому розділі були розглянуті існуючі системи семантичного пошуку, зокрема пошукова система Google, платформа аналізу тексту Semantria та платформа аналізу даних Sinequa. Ці системи використовують семантичні методи та технології для поліпшення пошуку інформації та аналізу даних.

Пошукова система Google використовує широкий набір алгоритмів та методів, включаючи семантичне розуміння тексту та контекстуальний аналіз. Платформа Semantria пропонує розширений набір інструментів для аналізу тексту з використанням семантичних методів. Вона здатна розпізнавати емоції, оцінки та тематики текстових документів, що дозволяє проводити глибокий аналіз великого обсягу даних.

Платформа Sinequa спрямована на аналіз та інтеграцію даних з різних джерел, включаючи структуровані та неструктуровані джерела. Вона використовує семантичні технології для розуміння зв'язків між різними елементами інформації та забезпечує швидкий та точний пошук даних.

# 3 ЗАСОБИ РОЗРОБКИ СИСТЕМИ СЕМАНТИЧНОГО ПОШУКУ

## 3.1 Bidirectional Encoder Representations from Transformers (BERT)

Bidirectional Encoder Representations from Transformers (BERT) — це сімейство моделей із замаскованою мовою, представлене в 2018 році дослідниками Google. Опитування літератури 2020 року дійшло висновку, що «трохи більше ніж за рік BERT став повсюдною базою в експериментах з обробки природної мови (NLP).

Мовна модель - це розподіл ймовірностей по послідовності слів. Коли мовна модель навчається на текстових корпусах, вона генерує ймовірності для будь-якої послідовності слів довжиною m, що відображає ймовірність $P(w_1, \ldots, w_m)$ усій послідовності. Оскільки мова може використовуватися для вираження нескінченної різноманітності речень, мовне моделювання стикається з проблемою присвоєння ненульових ймовірностей лінгвістично дійсним послідовностям, які можуть ніколи не зустрітися в навчальних даних. Для подолання цієї проблеми було розроблено кілька підходів до моделювання, таких як застосування припущення Маркова або використання нейронних архітектур, таких як рекурентні нейронні мережі або трансформатори.[6]

Мовні моделі корисні для різноманітних проблем комп'ютерної лінгвістики, включаючи розпізнавання мовлення, машинний переклад, генерацію природної мови, позначення частин мови, синтаксичний аналіз, оптичне розпізнавання символів, розпізнавання рукописного тексту, індукцію граматики, пошук інформації та інші програми.

У інформаційному пошуку мовні моделі використовуються для оцінки правдоподібності запиту. Кожен документ у колекції пов'язаний з окремою мовною

моделлю. Документи ранжуються на основі ймовірності запиту Q у мовній моделі документа $P(Q \mid M_d)$. Уніграмна модель мови зазвичай використовується для цієї мети.

BERT використовує механізм уваги під назвою Transformer, який дозволяє вивчати зв'язки між словами або підсловами в тексті, що допомагає створювати мовну модель. Transformer складається з кодера, який зчитує введений текст, і декодера, який створює прогноз для завдання. Однак BERT використовує тільки механізм кодування. Transformer кодує всю послідовність слів одночасно, що дозволяє моделі вивчати контекст слова на основі всього його оточення (ліворуч і праворуч від слова). Наведена нижче таблиця описує високорівневий кодер Transformer, який використовується в BERT. На вході - послідовність токенів, які вбудовуються у вектори, а потім обробляються в нейронній мережі. Результат - послідовність векторів розміру H, кожен з яких відповідає вхідному маркеру з таким же індексом. [7]

Перед подачею послідовностей слів у BERT 15% слів у кожній послідовності замінюються маркером [MASK]. Потім модель намагається передбачити початкове значення замаскованих слів на основі контексту, наданого іншими незамаскованими словами в послідовності (Рис. 3.1).

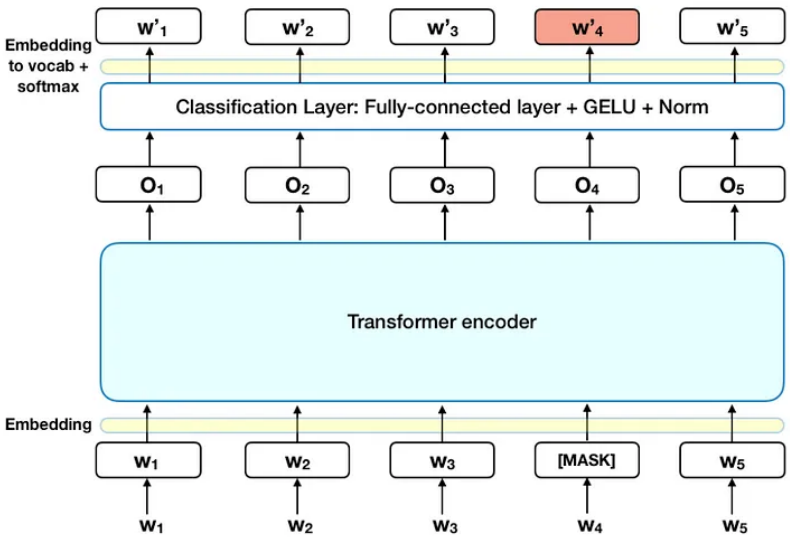


Рисунок 3.1 - Masked LM (MLM)

З технічної точки зору, передбачення вихідних слів вимагає:

1) додавання шару класифікації поверх виводу кодера;

2) множення вихідних векторів на матрицю вбудовування, перетворення їх у вимір словника;

3) обчислення ймовірності кожного слова в словнику за допомогою softmax.

Функція втрат BERT враховує тільки передбачення для замаскованих значень, і не звертає увагу на передбачення для незамаскованих слів. Це призводить до того, що модель збігається повільніше, ніж спрямовані моделі. Проте, ця особливість в компенсується підвищеною здатністю моделі розуміти контекст.

## 3.2 Бібліотека SentenceTransformers

SentenceTransformers - це фреймворк на Python для сучасних вбудовувань речень, тексту та зображень.

### 3.2.1 Sentence-BERT: Вбудовані речення з використанням BERT-мереж

Sentence-BERT (SBERT) — це модифікована версія мережі BERT, яка використовує сіамську та триплетну нейронні архітектури для отримання семантично значущих вбудованих речень. Це дозволяє використовувати BERT для нових завдань, які раніше були неможливими, наприклад, великомасштабне порівняння семантичної подібності, кластеризація та пошук інформації за допомогою семантичного пошуку.

Архітектура сіамської мережі дозволяє отримати вектори фіксованого розміру для вхідних речень. За допомогою мір подібності, таких як косинусна подібність та Манхеттенська/Евклідова відстань, можна знайти семантично схожі речення. SBERT дозволяє виконувати знаходження подібності надзвичайно ефективно на сучасному апаратному забезпеченні, що дозволяє використовувати SBERT для пошуку

семантичної подібності та кластеризації. Наприклад, з використанням SBERT складність пошуку найбільш схожої пари речень в колекції з 10 000 речень скорочується з 65 годин до 5 секунд, а обчислення косинусної подібності займає лише 0,01 секунди. З оптимізованими структурами індексів, пошук найбільш схожого запитання Quora можна скоротити з 50 годин до кількох мілісекунд. [8]

### 3.2.2 Можливості бібліотеки SentenceTransformers

SentenceTransformers можна використовувати для обчислення вбудовувань речень/текстів для більш ніж 100 мов. Потім ці вставки можна порівняти, наприклад, з косинусоїдальною подібністю, щоб знайти речення зі схожим значенням. Це може бути корисно для семантичної схожості текстів, семантичного пошуку або перефразування.[9]

Фреймворк базується на PyTorch та Transformers і пропонує велику колекцію попередньо навчених моделей, налаштованих для різних завдань. Крім того, можна легко налаштувати власні моделі.

### 3.3.3 Семантичний пошук

Семантичний пошук спрямований на покращення точності пошуку шляхом розуміння семантики пошукового запиту. Замість традиційних пошукових систем, які використовують тільки лексичні збіги для знаходження документів, семантичний пошук може враховувати також синоніми та інші зв'язки між словами.

Ідея семантичного пошуку полягає в тому, що всі записи у вашому корпусі, будь то речення, абзаци або документи, вбудовуються у векторний простір.

Під час пошуку запит також вбудовується в той же векторний простір, і знаходяться найближчі записи з вашого корпусу, які мають високу семантичну

відповідність запиту. Відмінність від традиційних пошукових систем полягає у тому, що семантичний пошук враховує синоніми та інші семантичні зв'язки між словами, а не лише лексичні збіги. Приблизну логіку знаходження вектору зображено на рисунку 3.2. Тут для наочності використано двовимірний простір, але насправді використовується простір з 784 вимірами.

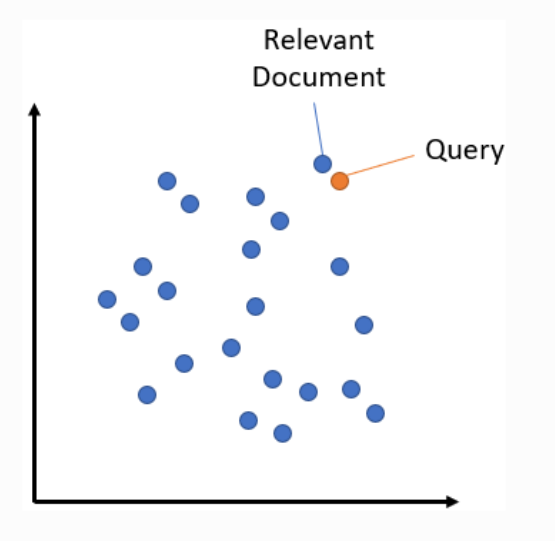


Рисунок 3.2 - Подібність запиту і документу у векторному просторі

Один з важливих аспектів, який варто враховувати при налаштуванні семантичного пошуку, це розрізнення між симетричним та асиметричним підходом.

У симетричному семантичному пошуку запит та записи в корпусі мають близьку довжину та обсяг контенту. Наприклад, якщо ви шукаєте "Як вивчити Python онлайн?", то вам може підійти запис "Як вивчити Python в Інтернеті?". В цьому випадку ви можете порівняти запит та записи з корпусу, і відшукати ті, які мають схожий зміст.

У випадку з асиметричним семантичним пошуком запит зазвичай є коротким, наприклад, запитання або кілька ключових слів. Ви шукаєте довший абзац або документ, який містить відповідь на цей запит. Наприклад, якщо запит звучить як "Що таке Python?", то ви шукаєте абзац або документ, що містить відповідь на це запитання.

В цьому випадку порівняння запиту та записів з корпусу має більш складну структуру, і потребує використання відповідної моделі для знаходження схожих записів.

Важливо правильно вибрати модель для вашого типу задачі. Наприклад, для симетричного семантичного пошуку можна використовувати попередньо навчені моделі вбудовування речень, а для асиметричного семантичного пошуку - моделі, навчені на даних MS MARCO. Для корпусів невеликих розмірів (приблизно до 1 мільйона записів) можна обчислити косинусну подібність між запитом і всіма записами в корпусі. У наступному прикладі ми створюємо невеликий корпус, який складається з кількох прикладів речень, і обчислюємо векторне представлення для корпусу і для нашого запиту. Далі ми використовуємо функцію util.cos_sim() для обчислення косинусної подібності між запитом і всіма записами корпусу. Для великих корпусів сортування всіх оцінок може зайняти занадто багато часу, тому ми використовуємо функцію torch.topk для отримання лише найкращих k записів.

## 3.3 Бібліотека Facebook AI Similarity Search (Faiss)

Faiss - це бібліотека Facebook AI Similarity Search, яка дозволяє ефективно шукати мультимедійні документи, що мають схожі характеристики, що важко досягнути для традиційних пошукових систем. Бібліотека використовує метод пошуку найближчих сусідів для наборів даних мільярдів розмірностей, який працює приблизно в 8,5 рази швидше, ніж попередні методи, і має найшвидший алгоритм k-вибору на графічному процесорі, який описаний в літературі. Ці нові досягнення дозволили побити декілька рекордів, зокрема перший граф k-найближчих сусідів, побудований на 1 мільярді векторів високої розмірності.[10]

Звичайні системи баз даних SQL не є практичними, оскільки вони оптимізовані для пошуку на основі хешу або одномірних інтервалів. Можливості пошуку подібності в таких пакетах, як OpenCV, сильно обмежені з точки зору масштабованості, як і інші

бібліотеки пошуку подібності, розроблені для "малих" наборів даних (наприклад, лише 1 мільйон векторів). На зображенні 3.3 показано, як FAISS використовується для

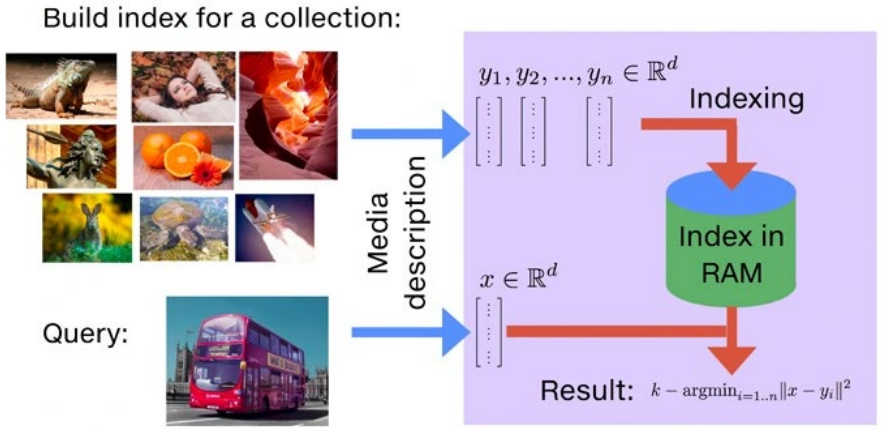


Рисунок 3.3 - Принцип роботи FAISS

знаходження подібних зображень в певній колекції

Плюси використання бібліотеки FAISS:

- Faiss пропонує ряд методів пошуку схожості з різними компромісами з точки зору використання;
- бібліотеку оптимізовано для швидкості та використання пам'яті;
- Faiss забезпечує найсучаснішу реалізацію на GPU для найбільш релевантних методів індексування.

### 3.3.1 Оцінка пошуку подібності

Після того, як вектори були вилучені за допомогою машинного навчання з різних джерел, таких як зображення, відео та текстові документи, їх можна завантажити в бібліотеку пошуку схожості.

Як еталон використовується алгоритм грубої сили, який точно і вичерпно обчислює всі подібності і повертає список найбільш схожих елементів. Однак цей алгоритм нелегко ефективно реалізувати, і його продуктивність впливає на інші компоненти.

Для прискорення пошуку схожості виконується індексування набору даних, що передбачає певну попередню обробку набору даних. Цей підхід може пожертвувати деякою точністю заради швидкості, але він може бути набагато швидшим, ніж алгоритм грубого перебору.

Три ключові метрики, що представляють інтерес, - це швидкість, використання пам'яті та точність. Компроміс між швидкістю і точністю зазвичай оцінюється для фіксованого використання пам'яті. FAISS зосереджується на методах, які стискають вихідні вектори для масштабування до наборів даних у мільярди векторів, використовуючи при цьому лише 32 байти пам'яті на вектор.

В той час як багато бібліотек індексування існують для близько 1 мільйона векторів, що ми називаємо малим масштабом, Faiss оптимізовано для використання пам'яті та швидкості і надає найсучаснішу GPU-реалізацію для найбільш актуальних методів індексування. Nmslib - ще один приклад бібліотеки, яка працює швидше за Faiss, але вимагає значно більше пам'яті.[10]

Після вибору типу індексу наступним кроком є виконання індексації. Це передбачає подачу векторів через вибраний алгоритм для створення індексу. Індекс можна або негайно використати, або зберегти на диску. Крім того, до індексу можна додавати та видаляти його під час пошуку.

### 3.3.2 Пошук в індексі

Після створення індексу параметри пошуку можна налаштувати так, щоб оптимізувати компроміс між точністю і часом пошуку, зберігаючи при цьому

фіксоване використання пам'яті. Це передбачає встановлення параметрів, які дають бажаний рівень точності при мінімізації часу пошуку. Faiss надає механізм автоматичного налаштування, який досліджує простір параметрів і визначає найкращі робочі точки для заданих вимог до точності і часу пошуку. На рисунку 3.4 зображено, яким чином кількість елементів в кластері впливає на швидкодію пошукових запитів.

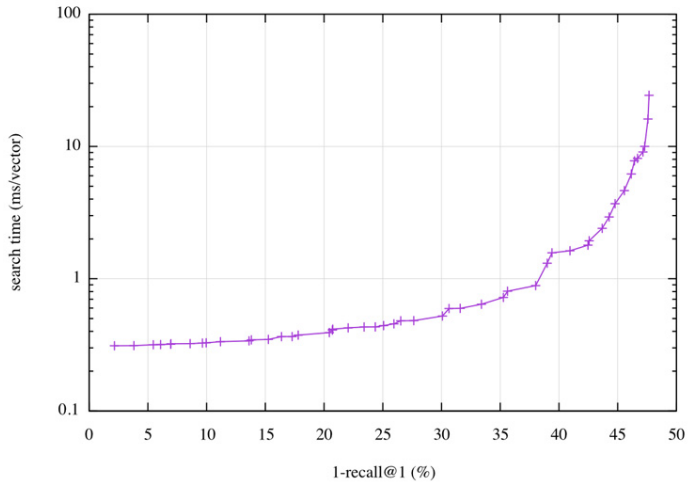


Рисунок 3.4 - Графік залежності часу пошуку від кількості елементів

Графік надає інформацію про компроміс між точністю та часом запиту. Наприклад, для досягнення 1-відповіді@1 у 40% потрібно менше 2 мс на вектор, або при кількості часу 0,5 мс можна досягти 30%-ї відповіді. При часі запиту 2 мс одне ядро може обробляти 500 QPS. [11]

### 3.3.3 Використання FAISS

Faiss (як C++, так і Python) пропонує екземпляри Index, які дозволяють користувачеві реалізувати структуру індексування, до якої можна додавати вектори та здійснювати пошук. Кожен з різних підкласів Index реалізує різну структуру індексування. Наприклад, IndexFlatL2 — це індекс грубої сили, який виконує пошук за допомогою відстаней L2.

Щоб виконати пошук за допомогою екземпляра Index у Faiss, спочатку потрібно додати вектори, які ви хочете шукати, до індексу за допомогою методу add. Після цього ви можете використовувати метод пошуку для пошуку найближчих сусідів вектора запиту. Ось приклад:

Faiss надає екземпляри Index, де кожен підклас Index реалізує структуру індексування для пошуку та додавання векторів. Наприклад, IndexFlatL2 — це індекс, який використовує грубу силу для пошуку відстаней L2. На рисунку 3.5 зображено алгоритми, що використані в реалізації бібліотеки пошуку.

Щоб додати вектори до індексу, ви можете викликати метод `add` в екземплярі індексу, передаючи масив NumPy векторів. Для пошуку векторів, найбільш схожих на набір векторів запиту, ви можете викликати метод `search` в екземплярі індексу, передаючи вектори запиту та кількість результатів, які вам потрібні. Метод повертає два масиви: `D` та `I`. «D» — це матриця квадратів відстаней між векторами запиту та найбільш схожими векторами в індексі, а «I» — це матриця індексів найбільш подібних векторів в індексі. [12]

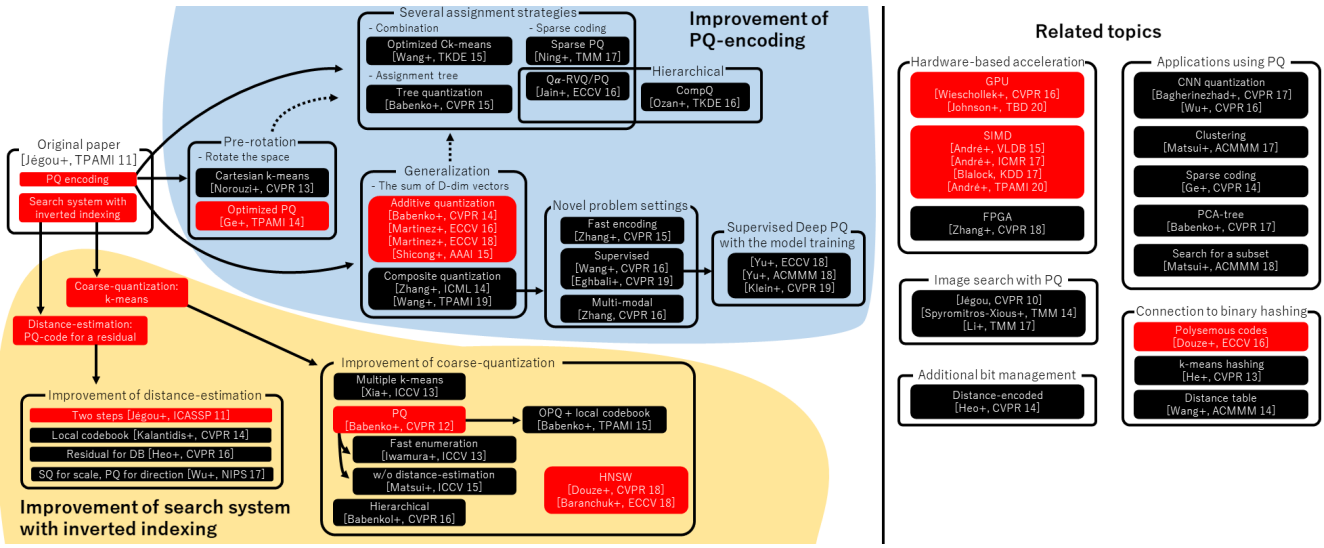


Рисунок 3.5 - Методи, що реалізовані в FAISS

## 3.4 Технологія RDF

RDF (Resource Description Framework) є стандартом, розробленим і прийнятим Консорціумом Всесвітньої павутини (W3C) для опису веб-ресурсів та обміну даними. Цей стандарт надає найпростіший, найпотужніший і найвиразніший спосіб роботи зі зв'язками між даними. Хоча існують різні інструменти для роботи з даними, RDF виділяється як найефективніший та широко прийнятий стандарт на сьогоднішній день.[13]

### 3.4.1 Трипли RDF

RDF використовує триплети (трипозиційні твердження) для зв'язування фрагментів даних між собою.

Просто кажучи, RDF дозволяє описувати факти, зв'язки та дані шляхом пов'язування різних типів ресурсів. За допомогою RDF-запитів можна виражати практично будь-яку інформацію за допомогою єдиної структури, що складається з трьох взаємопов'язаних фрагментів даних. На рисунку 3.6 зображено структуру одного триплу.

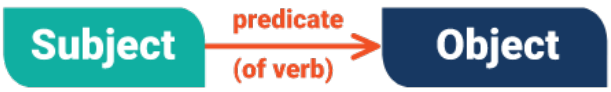


Рисунок 3.6 - Трипл RDF

### 3.4.2 Граф знань RDF

RDF використовується для побудови графів знань - високоякісних, гнучких та тісно пов'язаних інформаційних структур, які володіють потужною виразністю.

Діаграма нижче (рисунок 3.7) демонструє виразність RDF і його здатність до представлення складних зв'язків інформації.

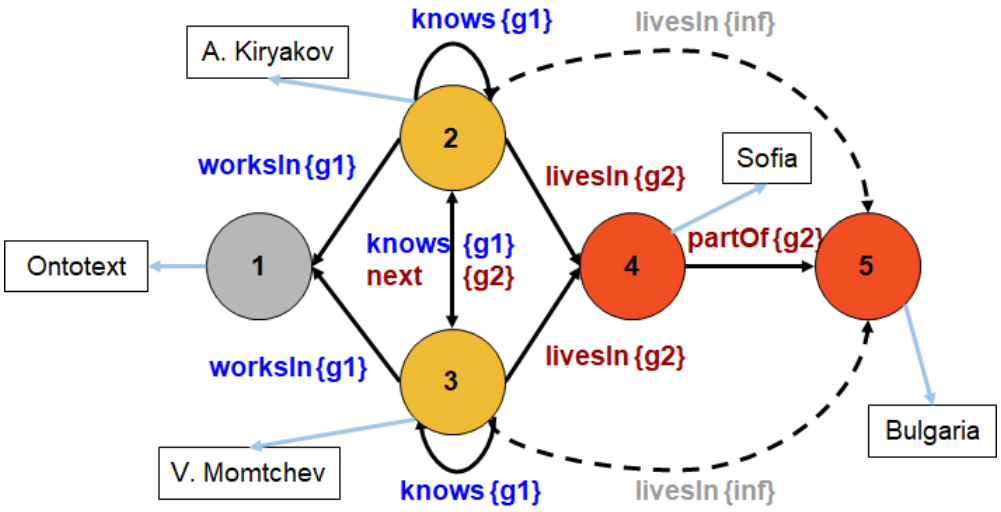


Рисунок 3.7 – Граф знань RDF

RDF використовує моделі даних для передачі інформації про дані, не обмежуючи їх структуру. Це означає широкий спектр можливих інтерпретацій та мінімальну залежність від конкретної стратегії впровадження.

## 3.5 Веб-фреймворк Django

Django - це веб-фреймворк на основі мови програмування Python, який пропонує простий і потужний спосіб для створення веб-додатків. Він має велику популярність завдяки своїй простоті використання, широким можливостям розширення і великій активній спільноті розробників.

### 3.5.1 Основні характеристики Django

Основні характеристики Django включають:[14]

1. Модель-представлення-контролер (MVC) або модель-представлення-шаблон (MVT) архітектура: Django використовує цю архітектурну модель, що дозволяє розділити логіку додатку на компоненти моделі (дані), представлення (відображення даних) та контролер або шаблон (логіка взаємодії з користувачем).

2. Вбудована адміністративна панель: Django надає готову адміністративну панель, що дозволяє зручно управляти базою даних, моделями та іншими аспектами веб-додатку без необхідності написання власного адміністративного інтерфейсу.

3. ORM (Object-Relational Mapping): Django має потужну ORM-систему, яка дозволяє працювати з базами даних на вищому рівні абстракції, використовуючи об'єктно-орієнтований підхід. Це спрощує роботу з базами даних і забезпечує переносимість додатків між різними системами управління базами даних.

4. Розширюваність: Django надає гнучкість та можливості для розширення функціональності додатків за допомогою сторонніх пакетів (бібліотек) та власних розширень. Велика спільнота розробників Django активно підтримує та розширює екосистему фреймворку.

5. Безпека: Django має вбудовані механізми безпеки, які допомагають уникнути загроз, таких як SQL-ін'єкції, скриптові атаки та перехоплення сесій.

6. Велика активна спільнота: Django має широку та активну спільноту розробників, яка надає допомогу, документацію, статті, приклади та пакети, що допомагають у веб-розробці з використанням Django.

Загальна характеристика Django полягає в його простоті використання, готових компонентах і широких можливостях для швидкого створення веб-додатків різного рівня складності.

### 3.5.2 Модель-представлення-шаблон

Модель-представлення-шаблон (MVT) є архітектурним шаблоном, який використовується в фреймворку Django для структурування веб-додатків. Цей шаблон дозволяє розділити різні аспекти додатку, такі як логіка, дані і представлення, для забезпечення простоти розробки, підтримки і тестування.

Основні компоненти MVT:

1. Модель (Model): Модель представляє дані додатку і відповідає за доступ до них. Вона визначає структуру даних, включаючи таблиці бази даних, поля та методи для роботи з цими даними. Модель зазвичай використовує об'єктно-реляційне відображення (ORM) Django для зручної взаємодії з базою даних.

2. Представлення (View): Представлення обробляє логіку додатку і взаємодію з користувачем. Воно отримує запити від користувача, виконує потрібні дії (такі як отримання даних з моделі, збереження або видалення даних) і генерує відповідь, яка може бути HTML-сторінкою, JSON-даними або іншим форматом відповіді.

3. Шаблон (Template): Шаблон визначає вигляд сторінок додатку. Він містить HTML-код зі спеціальними тегами та синтаксисом Django, які дозволяють вставляти дані з моделі і відображати їх на сторінці. Шаблони розділені від логіки додатку, що дозволяє змінювати вигляд сторінок без необхідності змінювати код представлення.

В процесі роботи з MVT, користувач взаємодіє з додатком, відправляючи запити до сервера Django. Представлення обробляє ці запити, виконує необхідні дії і звертається до моделі для отримання або збереження даних. Потім, використовуючи шаблон, представлення генерує відповідь, яка надсилається користувачеві.[15]

Переваги використання MVT:

1. Розділення відповідальностей: MVT дозволяє чітко розділити логіку додатку, дані та вигляд сторінок, що полегшує розробку, підтримку і тестування.

2. Перевикористання компонентів: Компоненти MVT можуть бути перевикористані в різних частинах додатку або навіть в інших проектах, що дозволяє зекономити час і зусилля при розробці.

3. Гнучкість і розширюваність: Django надає широкі можливості для розширення і налаштування MVT, що дозволяє пристосовувати фреймворк до конкретних потреб проекту.

4. Швидкість розробки: Завдяки чіткому поділу обов'язків і використанню готових компонентів, розробка веб-додатків з використанням Django і MVT може бути швидкою і ефективною.

## 3.6. Фреймворк для розробки API Django REST Framework

Django REST Framework (DRF) є потужним фреймворком для розробки API веб-додатків на основі Django. DRF надає розширений набір інструментів та функціональності для швидкої та ефективної розробки API.

Основні особливості Django REST Framework:

1. Серіалізація даних: DRF надає потужний механізм серіалізації та десеріалізації даних між моделями Django і форматами даних, такими як JSON або XML. Це дозволяє зручно передавати дані між клієнтами та сервером.

2. Маршрутизація та URL-шаблони: DRF надає простий спосіб визначення маршрутів для різних ендпоінтів API. Це дозволяє легко налаштовувати URL-шаблони для обробки різних запитів HTTP, таких як GET, POST, PUT або DELETE.

3. Аутентифікація та авторизація: DRF забезпечує широкі можливості для налаштування механізмів аутентифікації та авторизації в API. Він підтримує різні методи аутентифікації, такі як токени, сесії або JWT (JSON Web Tokens), а також дозволяє контролювати доступ до ресурсів залежно від прав користувачів.

4. Підтримка форматування відповідей: DRF автоматично перетворює відповіді API в різні формати даних, такі як JSON, XML або HTML, в залежності від запиту клієнта. Це дозволяє зручно працювати з різними клієнтськими додатками.

5. Розширення та настроювання: DRF має гнучку архітектуру, яка дозволяє розширювати та настроювати його функціональність згідно з потребами проекту. Він надає можливість створювати власні класи перегляду, серіалізатори та інші компоненти для розробки API згідно з унікальними вимогами проекту.

Django REST Framework є потужним і простим у використанні фреймворком, який допомагає розробникам швидко створювати стабільні та масштабовані API веб-додатків з використанням Django.

## Висновки до розділу 3

У цьому розділі були розглянуті різні засоби розробки системи семантичного пошуку, зокрема Bidirectional Encoder Representations from Transformers (BERT), бібліотека SentenceTransformers, бібліотека Facebook AI Similarity Search (FAISS), технологія RDF та веб-фреймворк Django.BERT є потужною моделлю для розуміння семантики тексту і використовується для генерації щільних векторів, які можуть бути використані для пошуку схожих текстових документів.Бібліотека SentenceTransformers надає зручні інструменти

для використання BERT-мереж для створення векторних представлень речень. Вона дозволяє легко реалізувати семантичний пошук та забезпечує широкі можливості для роботи з текстовими даними.FAISS є ефективною бібліотекою для пошуку подібності векторів. Вона надає інструменти для оцінки подібності, пошуку в індексі та використання FAISS для швидкого і точного пошуку інформації на основі векторів.

# 4 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ ВЕБ-ДОДАТКУ

## 4.1 Парсинг RSS фіда

Для індексації даних сайту використовується команда "index_rss", яка відповідає за парсинг та індексацію RSS-стрічки сайту. Ця команда дозволяє отримати дані з RSS-фіда та зберегти їх у внутрішній системі індексації.

Крім стандартного індексування RSS-стрічки, існує можливість надіслати запит на індексацію власного сайту. У такому випадку, система зберігає посилання на непроіндексовані сайти та здійснює їх індексацію пізніше, коли настане відповідний час.

Під час процесу індексації, створюються щільні вектори для зберігання та ефективного пошуку великого обсягу даних. Ці вектори зберігаються в нереляційній базі даних ElasticSearch, яка забезпечує швидкий та ефективний доступ до інформації.

Кожному вектору в базі даних ElasticSearch ставиться у відповідність посилання, що дозволяє легко знаходити та отримувати вектори для подальшої обробки.

Після процесу індексації, проводиться перетренування індексу FAISS. FAISS - це бібліотека для швидкого пошуку векторів, яка дозволяє покращити результати пошуку та зменшити час його виконання. Цей процес сприяє покращенню ефективності системи та точності результатів.

Важливою складовою процесу індексації є оптимізація часу та використання пам'яті. Це досягається шляхом вдосконалення алгоритмів та налаштування системних параметрів для досягнення найбільш оптимальної продуктивності.

Для зменшення навантаження на систему та запобігання перенавантаженню ресурсів, індексація проводиться в певний час доби, який не співпадає з активним використанням системи. Це дозволяє забезпечити стабільну роботу системи та зберегти її продуктивність під час індексаційного процесу.

## 4.2 Поле, по якому проводиться пошук

Анотація є коротким текстовим описом, який використовується для опису документу, такого як стаття, зображення або відео. В контексті індексації документів, анотація може бути створена зі структурних елементів документа, щоб відображати його важливі ознаки.

У даному випадку, для створення анотації використовуються два поля - "item.title" та "item.full-text". Поле "item.title" містить заголовок статті, а поле "item.full-text" містить повний текст статті. Комбінування цих полів дозволяє отримати короткий текстовий опис, який містить важливі ознаки статті.

Після створення анотації, її можна конвертувати в щільний вектор. Щільний вектор - це числовий вектор фіксованої довжини, який відображає всі важливі ознаки документа. Конвертування анотації в щільний вектор дозволяє зберігати та обробляти дані більш ефективно.

Під час пошуку документів, система використовує вектор анотації для порівняння з векторами інших документів у базі даних. Використовуючи індексування та пошук FAISS (Facebook AI Similarity Search), система знаходить найбільш подібні документи за їх векторними представленнями. Це дозволяє швидко та ефективно здійснювати пошук документів з подібними ознаками до вихідної анотації.

## 4.3 Ініціалізація індексу FAISS

Індекс FAISS - структура даних, що використовується для ефективного пошуку подібності. Індекс будується на основі набору вставок, що зберігаються в data['embeddings'], який вважається масивом NumPy форми (n, d), де n - кількість вставок, а d - їхній розмір. Вкладення вважаються нормалізованими до одиничної довжини.

Для його ініціалізації використовуються наступні функції:

- create_faiss_index(): Ця функція створює новий індекс FAISS, тренує його на наборі включень і додає їх до індексу. Індекс використовує плаский внутрішній квантор добутків і пласку індексну структуру IVF (інвертований файл) з кількістю кластерів n_clusters. Параметр nprobe дорівнює 3, що контролює кількість кластерів, відвіданих під час пошуку. Функція повертає створений індекс;
- save_index_to_file(): Ця функція кодує аннотації, що зберігаються в data['abstracts'], за допомогою Bi-Encoder і зберігає отримані вставки до індексного файлу FAISS під назвою 'faiss.index'. Кожне вбудовування асоціюється з ідентифікатором, що відповідає його позиції у data['abstracts'];
- get_faiss_index(): Ця функція намагається завантажити індекс FAISS з файлу 'faiss.index'. Якщо файл не існує або не може бути прочитаний, вона викликає create_faiss_index() для створення нового індексу.

В кінці коду змінній index присвоюється результат виклику get_faiss_index(), який або завантажує існуючий індекс, або створює новий. Індекс потім використовується для пошуку схожості. Індекс ініціалізується при кожному запуску застосунку Django

## 4.4 Функції пошуку

Ок, розпишу детальніше про код, що відповідає за функцію пошуку в індексі FAISS за запитом користувача.

У програмній реалізації, функція `search` приймає на вхід запит користувача, кількість результатів, об'єкт індексу FAISS та модель позначення речень.

Спочатку відбувається кодування запиту користувача моделлю позначення речень для отримання вектора запиту. Це здійснюється за допомогою моделі

позначення речень, яка перетворює текст запиту в числовий вектор, який представляє його значення.

Після отримання вектора запиту, викликається функція `search` з переданим індексом FAISS та вектором запиту. Ця функція здійснює пошук в індексі FAISS і повертає кілька найбільш схожих векторів даних.

Найбільш схожі вектори повертаються разом з додатковою інформацією про статті, які відповідають цим векторам. Інформація про статті зберігається окремо від векторів індексу і знаходиться у змінній `data`. Для отримання інформації про статті, витягується з `data` за індексом топ-K статей, знайдених в індексі FAISS. Ця інформація про статті повертається у вигляді списку словників, де кожен словник представляє одну статтю з її деталізованою інформацією.

З боку Django використовується клас `Search`, який наслідує від класу `TemplateView`. Цей клас відповідає за пошук результатів по запиту користувача та відображення їх у вигляді HTML-сторінки.

Атрибут `template_name` визначає шаблон, який буде використовуватися для відображення сторінки результатів пошуку.

Метод `get_context_data` викликається при запиті GET. Він отримує контекст для шаблону, передаючи його як словник параметрів. У цьому методі звертається до рядка запиту, переданого у GET-запиті, та передає його до функції `search`, яка повертає список результатів, відфільтрованих за допомогою переданого запиту. Якщо запит не заданий, то список результатів порожній.

Після отримання результатів пошуку, вони зберігаються у словник `context` з ключем `results`. Словник `context` передається в шаблон, який буде відображений на веб-сторінці користувачу, де результати пошуку будуть відображені у вигляді HTML-елементів.

## 4.5 Функціонал RDF графів

RDF графи в програмному додатку використовуються для збагачення інформації про статті, знайдені за допомогою пошуку з використанням моделі машинного навчання. Ці графи містять додаткові зв'язки і характеристики, які допомагають краще розуміти взаємозв'язки між статтями. Наприклад, можуть бути вказані зв'язки типу "цитує", "відноситься до теми", "має автора" тощо. Крім того, літерали можуть бути використані для представлення додаткових характеристик статей, таких як рік публікації, ключові слова, рейтинг тощо.

Такий підхід з використанням RDF графів дозволяє покращити якість пошуку та надати користувачам більше інформації про знайдені статті, створюючи більш повну та змістовну зв'язану мережу даних.

Цей підхід до використання RDF графів у програмному додатку принесе безліч переваг. Зокрема, він дозволяє зберігати структуровані дані про статті та їх зв'язки в одному місці, що спрощує управління та пошук інформації. Крім того, за допомогою RDF графів можна здійснювати складні запити та аналізувати зв'язки між статтями, що дозволяє знайти нові залежності та отримати цінні інсайти.

Крім функціоналу, пов'язаного з RDF графами, в програмному додатку можуть бути реалізовані й інші корисні можливості. Наприклад, використання алгоритмів машинного навчання для рекомендаційних систем, що базуються на аналізі зв'язків між статтями. Також, можливо впровадити інтерактивні візуалізації RDF графів, що дозволять користувачам детальніше досліджувати та взаємодіяти зі знайденою інформацією.

Загалом, використання RDF графів в програмному додатку відкриває широкі можливості для покращення функціональності, розширення аналітичних можливостей та підвищення якості пошуку. Це створює потужний інструмент для

забезпечення зручності та задоволення потреб користувачів, які використовують програмний додаток для пошуку та аналізу статей.

### 4.5.1 Додавання та редагування графів для окремих статтей

У програмному додатку, використовуючи Django admin, забезпечується можливість додавання та редагування зв'язків для конкретної статті в RDF графі. Це дозволяє адміністраторам змінювати та встановлювати нові зв'язки між статтями, а також редагувати характеристики, представлені у вигляді літералів. На рисунку 4.1 показано, як відбувається редагування та додавання триплів в RDF графі для конкретного веб-ресурсу.

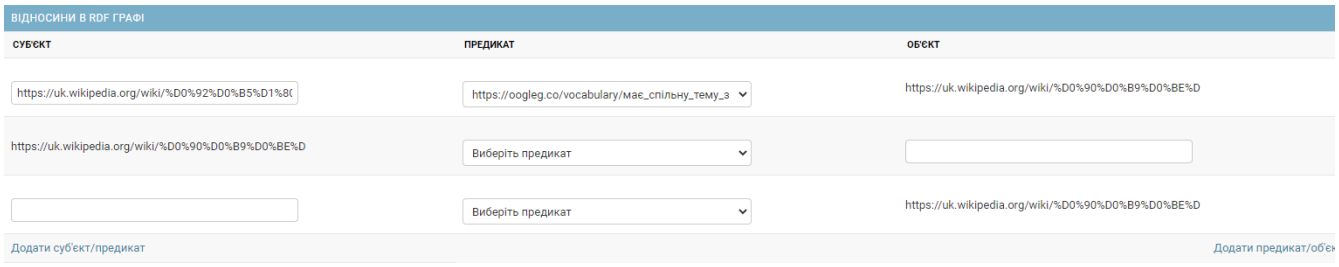


Рисунок 4.1 - Редагування зв'язків статті

### 4.5.2 Відображення взаємозв’язків для статті в результатах пошуку

Під час виведення результатів пошуку в програмному додатку, додаткова інформація про знайдену статтю з RDF графу відображається під посиланням на результат. Ця інформація може включати зв'язки з іншими статтями, їх характеристики, а також будь-які інші дані, які були додані до графу. Це надає користувачам додаткову інформацію та контекст про знайдену статтю.  На рисунку 4.2 зображено зв’язки між однією статтею та літералами в графі.

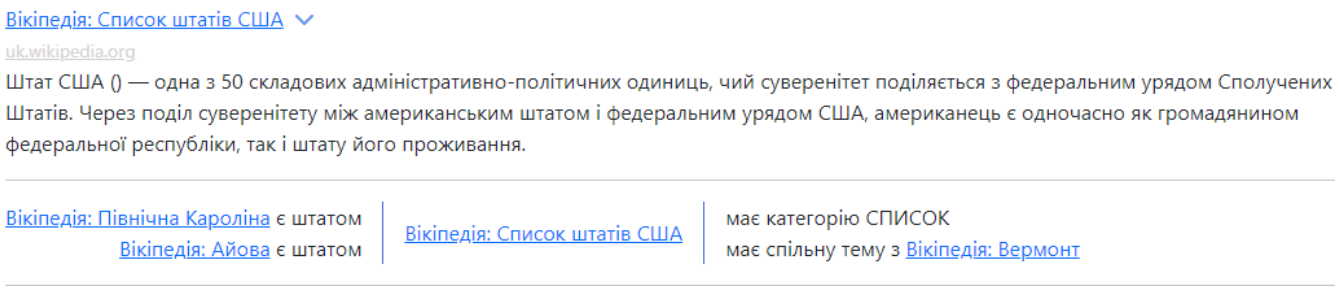


Рисунок 4.2 - Граф RDF під результатом пошуку

## 4.6 Знаходження тегів веб-ресурсів за допомогою нейромережі

В програмі реалізовано функціонал автоматичного знаходження тегів для веб-ресурсів з метою їх подальшого використання в онтології RDF. Ці теги є ключовими словами або поняттями, що ідентифікують тему або зміст статті і використовуються для створення зв'язків між схожими статтями.

Для знаходження тегів була використана нейромережа, яка спроможна групувати статті за наборами тем. Кожна тема може містити різні частини мови, окрім того, статті можуть містити орфографічні помилки. Щоб уникнути цього для отриманих тегів було проводиться фільтрація з метою відбору лише іменників у називному відмінку. Це дозволяє зосередитись на ключових сутностях або поняттях, які найбільш точно визначають тему статті. Крім того,

виправляються орфографічні помилки для забезпечення точності і однозначності тегів.

Крім автоматичного знаходження тегів, також існує можливість внесення в список тегів-виключень. Ці теги не мають використовуватися в онтології і можуть бути виключені з розгляду під час формування зв'язків між статтями.

Цей функціонал дозволяє автоматизувати і полегшити процес створення та управління онтологією RDF, забезпечуючи більш точну та зручну роботу зі зв'язками між веб-ресурсами.

На рисунку 4.3 зображено теги для статті з вікіпедії про європейську монаршу династію Ягеллонів (https://uk.wikipedia.org/wiki/%D0%AF%D0%B3%D0%B5%D0%BB%D0%BB%D0%BE%D0%BD%D0%B8)

```
<https://uk.wikipedia.org/wiki/%D0%AF%D0%B3%D0%B5%D0%BB%D0%BB%D0%BE%D0%BD%D0%B8> voc:має_тег <https://oogleg.co/vocabulary/tag/королівство>,
        <https://oogleg.co/vocabulary/tag/монарх>,
        <https://oogleg.co/vocabulary/tag/монархія>,
        <https://oogleg.co/vocabulary/tag/імператор>,
        <https://oogleg.co/vocabulary/tag/імперія> .
```

Рисунок 4.3 — автоматично згенеровані теги для веб-ресурсу

## 4.7 Виправлення орфографічних помилок

У програмі реалізований допоміжний функціонал, спрямований на виправлення орфографічних помилок в запитах користувачів. Ця функція дозволяє поліпшити користувацький досвід шляхом пропозиції виправленого запиту, якщо виявлено орфографічну помилку в початковому запиті.

При введенні запиту користувачем система автоматично перевіряє правильність написання слів і виявляє можливі орфографічні помилки. Якщо такі помилки виявлені, система пропонує варіанти виправлення, що базуються

на лексичних правилах. Користувачу відображається виправлений варіант запиту разом з оригінальним.

Якщо користувач погоджується з виправленим запитом, відбувається автоматичне виконання пошуку за виправленим запитом. Це дозволяє забезпечити точніші результати пошуку та знизити ймовірність неправильних або невідповідних результатів.

Реалізація цього допоміжного функціоналу покращує користувацький досвід, дозволяючи користувачам швидше та ефективніше здійснювати пошук, навіть у випадках, коли їхні запити містять орфографічні помилки. Крім того, виправлення орфографічних помилок сприяє підвищенню точності пошукових результатів.

## 4.8 Розширений пошук по онтології

В рамках програми було реалізовано функціонал пошуку по зв'язаним статтям, що дозволяє користувачеві отримувати більш розгорнуті та специфічні результати пошуку. Після введення користувачем початкового запиту, система пропонує можливість виконати розширений пошук.

Користувач має можливість натиснути на кнопку розширеного пошуку, що запускає процес пошуку веб-ресурсів, що зв'язані по онтології з результатами, які були знайдені раніше. Це означає, що система буде шукати статті та веб-ресурси, які мають семантичні зв'язки зі статтями, що були знайдені в результаті початкового запиту користувача.

Цей розширений пошук дозволяє знайти більше пов'язаних матеріалів та розширити контекст інформації, доступної користувачу. Він допомагає забезпечити більш глибоке дослідження теми, виявити додаткові ресурси, що

мають відношення до початкового запиту користувача. Це дозволяє забезпечити більш широку та докладну перегляду інформації, що відповідає потребам користувача.

## Висновки до розділу 4

У цьому розділі надано детальна інформація щодо програмної реалізації веб-додатку інформаційно-семантичного пошуку. Реалізовано функціонал для отримання та обробки даних з RSS фідів. Це дозволяє веб-додатку отримувати актуальну інформацію з різних джерел. Реалізовано процес ініціалізації індексу FAISS, який забезпечує ефективний пошук схожих векторів. Це дозволяє швидко знаходити інформацію на основі векторних представлень. Розроблені функції пошуку, які використовують семантичні методи та алгоритми FAISS для забезпечення ефективного пошуку інформації. Реалізовано функціонал для створення, редагування та відображення взаємозв’язків у графах RDF. Це дозволяє структурувати та семантично описувати інформацію у веб-додатку. Був реалізований функціонал для виправлення орфографічних помилок у запитах користувачів.Також було розроблено можливість розширеного пошуку, який використовує онтології RDF та OWL для забезпечення більш глибокого аналізу та пошуку інформації.

# ВИСНОВКИ

В даній дипломній роботі було розроблено веб-додаток, в якому є функціонал семантичного пошуку серед великих масивів даних. Для реалізації пошуку використовуються інструменти Facebook AI Similarity Search та Python Sentence-Transformers, що дозволяє ефективно зберігати та швидко знаходити вектори на основі семантичних моделей, побудованих на основі глибокого навчання. Реалізація веб-додатку виконана з використанням фреймворку Django, що дозволило просто та зручно організувати веб-інтерфейс та зв'язати його з іншими компонентами додатку.

У результаті роботи було виявлено, що використання семантичних моделей для пошуку на основі схожості текстів може покращити якість результатів, особливо у випадках, коли пошуковий запит складається зі складних та довгих фраз. Використання Elastic Search для зберігання щільних векторів також є ефективним способом зберігання та пошуку великої кількості даних, що дає можливість розширювати та розвивати веб-додаток з мінімальними затратами на зберігання та обробку даних.

В результаті отримано ефективний та зручний інструмент для семантичного пошуку статей, що може бути використаний в різних сферах діяльності, де необхідно швидко та точно знаходити відповідні тексти за запитами користувачів.

В даному дипломному проекті також було використано функціонал RDF графів для надання додаткової інформації про статті, знайдені за допомогою пошуку з використанням моделі машинного навчання. RDF граф містить інформацію про зв'язки між статтями та їх характеристиками. Завдяки використанню RDF графів, було можливо відображати зв'язки між статтями, включаючи ієрархічні відношення, відношення типу "батько-син" та інші.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. Dense Vectors: Capturing Meaning with Code, URL: https://www.pinecone.io/learn/dense-vector-embeddings-nlp/

2. Devlin, Jacob; Chang, Ming-Wei; Lee, Kenton; Toutanova, Kristina (11 October 2018). "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding".

3. Zhang, Z., Zhao, Z., & Zhang, Z. (2021). Sentence-BERT: A deep sentence embedding method with self-attention mechanism. Journal of Computer Science and Technology, 36(1), 171-185.

4. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin. (2017). Attention Is All You Need. arXiv:1706.03762.

5. Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. In Advances in neural information processing systems. arXiv:1409.3215

6. Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St. John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, Yun-Hsuan Sung, Brian Strope, and Ray Kurzweil. 2018. Universal Sentence Encoder. arXiv preprint arXiv:1803.11175.

7. BERT Explained: State of the art language model for NLP, URL: https://towardsdatascience.com/bert-explained-state-o-the-art-language-model-for-nlp-f8b21a9b6270

8. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks, URL: https://arxiv.org/pdf/1908.10084.pdf

9. SentenceTransformers Documentation, URL: https://www.sbert.net/

10. Faiss: A library for efficient similarity search, URL: https://engineering.fb.com/2017/03/29/data-infrastructure/faiss-a-library-for-efficient-similarity-search/

11. Johnson, J., Douze, M., & Jégou, H. (2017). Billion-scale similarity search with GPUs. arXiv preprint arXiv:1702.08734.

12. Lin, Y., Liu, Z., Sun, M., Liu, Y., & Zhu, X. (2020). Robust entity recognition and disambiguation with context-aware neural language models. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 34, No. 01, pp. 836-843).

13. Grau, B. C., & Horrocks, I. (2008). Modelling ontologies with description logics: theory and practice. In Reasoning web (pp. 1-27). Springer.

14. Holovaty, A., & Kaplan-Moss, J. (2009). The Definitive Guide to Django: Web Development Done Right. Apress.

15. Balogh, P. (2013). Django: Web Development with Python. Packt Publishing Ltd.

# Додаток А. Програмний код

## Модель Веб-ресурсу Django

```
import re
from html import unescape

import unicodedata
from bs4 import BeautifulSoup
from django.db import models



class WebResource(models.Model):
    url = models.URLField(max_length=2000, unique=True, verbose_name='URL')
    title = models.CharField(max_length=2000, verbose_name='Заголовок')
    abstract = models.TextField(verbose_name='Анотація')
    content = models.TextField(verbose_name='Вміст', blank=True)

    class Meta:
        verbose_name = 'Веб-ресурс'
        verbose_name_plural = 'Веб-ресурси'

    def __str__(self):
        return self.title

    @staticmethod
    def remove_html(text: str):
        soup = BeautifulSoup(text, 'html.parser')
        text_string = soup.get_text(separator=' ')
        s = unicodedata.normalize('NFKC', unescape(text_string))
        return re.sub(r'\s+', ' ', s)
    @property
    def content_without_html(self):
        return str(self.remove_html(self.content))
```

```
    def save(self, *args, **kwargs):
        if not self.abstract:
            self.abstract = self.remove_html(self.title)
            if self.content:
                self.abstract += ' ' + self.remove_html(self.content)
                self.abstract = " ".join(self.abstract.split(' ')[:128])

        super().save(*args, **kwargs)
```

### Парсинг RSS фіда

```
import feedparser
from django.core.management import BaseCommand

from articles.models import WebResource
from search.embeddings_storage import write_embeddings


class Command(BaseCommand):
    def add_arguments(self, parser):
        parser.add_argument("url", type=str)

    def handle(self, *args, **options):
        feed = feedparser.parse(options['url'])
        resources = []
        for entry in feed.entries:
            # remove html tags using regex

            wr = WebResource.objects.update_or_create(url=entry['id'],
                                      defaults={'title': entry['title'], 'content': entry['fulltext']})
            resources.append(wr)
```

```
    # machine learning part
    # write dense vectors to file
    write_embeddings(resources)
```

### Запис Векторних вбудовувань у файл

```
import os
import pickle

from django.conf import settings

from search.constants import model_name, max_corpus_size, bi_encoder

embedding_cache_path = settings.BASE_DIR / 'abstract-embeddings-{}-size-{}.pkl'.format(model_name,
max_corpus_size)
images_embeddings_cache_path = settings.BASE_DIR / 'images-abstract-embeddings-{}-size-
{}.pkl'.format(model_name,
                                                                    max_corpus_size)


def write_embeddings(web_resources, mode='wb'):
    pickle_data = {}
    # todo - store the embeddings in a database instead of a file
    # OR todo - don't overwrite the file if it already exists, instead append to it
    print("Encode the corpus. This might take a while")
    abstracts = [wr.abstract for wr in web_resources]
    corpus_embeddings = bi_encoder.encode(abstracts, show_progress_bar=True,
                                          convert_to_numpy=True)
    print("Store file on disc")
    pickle_data['db_ids'] = [wr.id for wr in web_resources]
    pickle_data['embeddings'] = corpus_embeddings
    with open(embedding_cache_path, mode) as fOut:
        pickle.dump(pickle_data, fOut)
```

```
def get_embeddings(cache_path):
    if not os.path.exists(cache_path):
        return None

    print("Load pre-computed embeddings from disc")
    with open(cache_path, "rb") as fIn:
        cache_data = pickle.load(fIn)
        data = {
            'db_ids': cache_data['db_ids'],
            'embeddings': cache_data['embeddings'],
        }
    return data
```

### Ініціалізація індексу FAISS

```
import faiss
import numpy as np

from search.constants import embedding_size, n_clusters, bi_encoder
from search.embeddings_storage import get_embeddings, embedding_cache_path


# not used
def save_index_to_file(index_name, embeddings_data, pandas_data):
    encoded_data = bi_encoder.encode(pandas_data['abstract'].values.tolist())
    encoded_data = np.asarray(encoded_data.astype('float32'))
    index = faiss.IndexIDMap(faiss.IndexFlatIP(768))
    index.add_with_ids(encoded_data, np.array(range(0, len(pandas_data))))
    faiss.write_index(index, f'{index_name}.index')
```

```
def create_faiss_index(index_name, embeddings_data):
    ### Create the FAISS index
    quantizer = faiss.IndexFlatIP(embedding_size)
    index = faiss.IndexIVFFlat(quantizer, embedding_size, n_clusters, faiss.METRIC_INNER_PRODUCT)
    index.nprobe = 3

    print("Start creating FAISS index")
    # First, we need to normalize vectors to unit length
    corpus_embeddings = embeddings_data['embeddings'] /
np.linalg.norm(embeddings_data['embeddings'], axis=1)[:, None]

    # Then we train the index to find a suitable clustering
    # todo - retrain the index every time new data is added (maybe not)
    index.train(corpus_embeddings)

    # Finally we add all embeddings to the index
    index.add(corpus_embeddings)
    # save the index
    faiss.write_index(index, f'{index_name}.index')

    print("Corpus loaded with {} sentences / embeddings".format(len(embeddings_data['db_ids'])))
    return index


def get_faiss_index(index_name, embeddings_cache_path):
    data = get_embeddings(embeddings_cache_path)
    if not data:
        return (None, None)
    try:
        return (faiss.read_index(f'{index_name}.index'), data)
    except:
        return (create_faiss_index(index_name, data), data)
```

```
text_index, ids_data = get_faiss_index('text_index', embedding_cache_path)
```

### Функції пошуку

```
import time

import numpy as np
import torch
from sentence_transformers import util

from articles.models import WebResource
from search.faiss_index import ids_data


def fetch_web_resource_info(dataframe_idx):
    db_id = ids_data['db_ids'][dataframe_idx]

    return WebResource.objects.get(id=db_id)


def search_faiss(query, top_k, index, model):
    t = time.time()
    query_vector = model.encode([query])
    top_k = index.search(query_vector, top_k)

    print('>>>> Results in Total Time: {}'.format(time.time() - t))
    top_k_ids = top_k[1].tolist()[0]
    top_k_ids = list(np.unique(top_k_ids))
    results = [fetch_web_resource_info(idx) for idx in top_k_ids]
    return results
```

```
def search(query, top_k, model):
    t = time.time()
    query_vector = model.encode(query, convert_to_tensor=True).to(model.device)
    tensor_data = torch.tensor(ids_data['embeddings'], device=model.device)
    cos_scores = util.cos_sim(query_vector, tensor_data)[0]
    top_results = torch.topk(cos_scores, k=top_k)
    print('>>>> Results in Total Time: {}'.format(time.time() - t))
    results = [fetch_web_resource_info(idx) for idx in top_results[1]]
    return results
```

### Редагування RDF через Django Admin

```
from django.contrib import admin
from rdflib import URIRef, Literal
from articles.models import WebResource
from oogleg import settings
from rdf_ontologies.RDF_graph import my_graph

@admin.register(WebResource)
class WebResourceAdmin(admin.ModelAdmin):
    list_display = ('title', 'abstract', 'url')
    search_fields = ('url', 'title', 'abstract')

    def changeform_view(self, request, object_id=None, form_url="", extra_context=None):
        extra_context = extra_context or {}
        if object_id and request.method == 'POST':
            url = WebResource.objects.get(pk=object_id).url
            my_graph.remove((URIRef(url), None, None))
            my_graph.remove((None, None, URIRef(url)))
            for key, value in request.POST.items():
                if key.startswith('predicate_objects_predicate_value-'):
                    predicate = value
```

```
            number = key.rsplit('-', 1)[1]
            object = request.POST.get(f'predicate_objects_object_value-{number}')
            if not object:
                continue
            # if object is not URL
            if not object.startswith('http'):
                object = Literal(object)
            else:
                object = URIRef(object)
            if predicate and object:
                my_graph.add((URIRef(url), URIRef(predicate), object))
        elif key.startswith('subject_predicates_subject_value-'):
            subject = value
            number = key.rsplit('-', 1)[1]
            predicate = request.POST.get(f'subject_predicates_predicate_value-{number}')
            if not predicate:
                continue
            if not subject.startswith('http'):
                subject = Literal(subject)
            else:
                subject = URIRef(subject)
            if subject and predicate:
                my_graph.add((URIRef(subject), URIRef(predicate), URIRef(url)))
    my_graph.serialize(destination=str(settings.BASE_DIR / "ontology.ttl"), format="turtle")
if object_id:
    url = WebResource.objects.get(pk=object_id).url
    extra_context['all_predicates'] = my_graph.get_oogleg_properties()
    extra_context['subject_predicates'] = my_graph.subject_predicates(object=URIRef(url))
    extra_context['predicate_objects'] = my_graph.predicate_objects(subject=URIRef(url))

return super().changeform_view(request, object_id, extra_context=extra_context)
```