Міністерство освіти і науки України
Криворізький національний університет
Кафедра моделювання та програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня вищої освіти магістра

зі спеціальності 121 – Інженерія програмного забезпечення

На тему: Розробка програмного модуля семантичної обробки даних OSINT

Засвідчую, що в цій кваліфікаційній роботі немає запозичень із праць інших авторів без відповідних посилань.

Студент гр. ІПЗ-23-1м

_____ /І. О. Берест /

Керівник кваліфікаційної роботи _____ / Н. Н. Шаповалова /

Завідувач кафедри _____ / А. М. Стрюк /

Кривий Ріг

2024

Криворізький національний університет

Факультет: Інформаційних технологій

Кафедра: Моделювання та програмного забезпечення

Ступінь вищої освіти: магістр

Спеціальність: 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:

Зав. кафедри

_____ А. М. Стрюк

«___» _____ 2024 р.

ЗАВДАННЯ

на кваліфікаційну роботу

студенту групи ІПЗ-23-1м Бересту Ігорю Олеговичу

1. Тема: Розробка програмного модуля семантичної обробки даних OSINT затверджено наказом по КНУ №272с від «18» квітня 2024.
2. Термін подання студентом закінченої роботи: «15» червня 2024.
3. Вихідні дані по роботі: розроблюване програмне забезпечення повинне обробляти дані.
4. Зміст пояснювальної записки (перелік питань, що їх треба розробити): проаналізувати існуючі модулі семантичної обробки даних, дослідити методології взаємодії з …, розробити алгоритми для …, спроектувати програмне забезпечення у вигляді веб сайту, провести тестування програмного забезпечення.
5. Перелік ілюстративного матеріалу: функціональна діаграма, діаграма класів, блок-схема розроблених алгоритмів та роботи програми, знімки графічного інтерфейсу.

Календарний план:

№	Найменування етапів кваліфікаційної роботи	Термін виконання етапів роботи
1	Огляд літературних джерел за темою	22.03.2024 – 27.03.2024
2	Аналіз існуючих рекомендаційних систем для пошуку та організації заходів	28.03.2024 – 31.03.2024
3	Аналіз методології взаємодії з …	01.04.2024 – 10.04.2024
4	Аналіз предметної області, формулювання вимог до системи	11.04.2024 – 14.04.2024
5	Підготовка матеріалів першого розділу роботи	15.04.2024 – 20.04.2024
6	Розробка блок-схем та діаграм розроблюваного програмного забезпечення	21.04.2024 – 26.04.2024
7	Підготовка матеріалів другого розділу роботи	27.04.2024 – 01.05.2024
8	Реалізація програмного модуля для взаємодії з …	02.05.2024 – 08.05.2024
9	Реалізація мобільного додатку для користування системою	09.05.2024 – 16.05.2024
10	Підготовка матеріалів третього розділу роботи	17.05.2024 – 20.05.2024
11	Оформлення пояснювальної записки	21.05.2024 – 29.05.2024

Дата видачі завдання: «**» квітня 2024 р.

Студент _____ / І. О. Берест /

Керівник роботи _____ / Н. Н. Шаповалова /

РЕФЕРАТ

СЕМАНТИКА, ВІДКРИТІ ДЖЕРЕЛА, АНАЛІЗ, ПОШУК, РОЗРОБКА СИСТЕМ.

Пояснювальна записка: 55 с., 13 рис., 1 дод., 9 джерел.

Метою кваліфікаційної роботи є розробка програмного модуля для семантичної обробки OSINT даних, що дозволяє ефективно аналізувати й групувати інформацію з відкритих джерел, забезпечуючи її структуроване представлення на основі семантичного змісту.

У ході виконання роботи було проведено аналіз сучасних підходів до обробки текстових даних, зокрема методів векторизації, таких як TF-IDF, та алгоритмів кластеризації, зокрема DBSCAN. Особливу увагу було приділено дослідженню існуючих OSINT-систем, їхніх можливостей, обмежень та способів взаємодії. На основі проведеного аналізу було визначено вимоги до програмного модуля, які охоплюють точність семантичної кластеризації, швидкість обробки запитів та зручність користування.

У результаті роботи було розроблено програмний модуль для семантичного аналізу, який групує дані за їх змістом і шукає інформацію у відкритих джерелах. Розроблена система забезпечує зручний інтерфейс для введення запиту та надання користувачеві структурованих даних у вигляді, придатному для подальшого аналізу.

Результати роботи демонструють підвищену ефективність обробки великих обсягів відкритої інформації, що значно спрощує процес прийняття рішень у контексті аналізу OSINT даних.

ABSTRACT

SEMANTIC, OSINT, ANALYSIS, SEARCH, SYSTEMS DEVELOPEMENT.

Thesis in: 55 p., 13 fig., 1 app., 9 references.

The purpose of qualification work is to develop a software module for semantic processing of OSINT data, enabling efficient analysis and grouping of information from open sources while providing a structured representation based on semantic content.

During the course of the work, an analysis of modern approaches to text data processing was conducted, focusing on vectorization methods such as TF-IDF and clustering algorithms, particularly DBSCAN. Special attention was given to studying existing OSINT systems, their capabilities, limitations, and methods of interaction. Based on the analysis, requirements for the software module were defined, encompassing the accuracy of semantic clustering, query processing speed, and user convenience.

As a result, a semantic analysis software module was developed, which groups data by their content and searches for information in open sources. The developed system provides a user-friendly interface for entering queries and delivering structured data suitable for further analysis.

The outcomes of this work demonstrate increased efficiency in processing large volumes of open information, significantly simplifying decision-making processes in the context of OSINT data analysis.

ЗМІСТ

ВСТУП

У сучасному світі зростаючих інформаційних потоків, OSINT (Open Source Intelligence) відіграє ключову роль у зборі, аналізі та використанні інформації з відкритих джерел. OSINT означає розвідку на основі відкритих джерел і охоплює всі доступні публічні джерела інформації, які можуть бути використані для аналітики. До таких джерел належать інтернет-сайти, соціальні мережі, новини, блоги, офіційні звіти, патенти, наукові публікації, дані зі збору статистики, форуми, а також дані з аудіо- та відеоматеріалів. Важливою особливістю OSINT є його законність та етичність, оскільки вся інформація збирається лише з відкритих та доступних джерел.

Застосування OSINT охоплює широке коло сфер, включаючи безпеку, бізнес-аналітику, конкуренційний аналіз, управління ризиками, журналістику, розвідку та військові операції. У бізнесі, зокрема, OSINT використовується для відстеження ринкових тенденцій, аналізу діяльності конкурентів та моніторингу репутації бренду. У сфері кібербезпеки – для виявлення потенційних загроз і збирання інформації про можливі вразливості в IT-інфраструктурі.

OSINT дозволяє організаціям отримувати важливу інформацію без значних фінансових витрат, оскільки більшість ресурсів доступні безкоштовно або за відносно низьку ціну. При цьому обсяги таких даних є надзвичайно великими, що робить їх ручну обробку складною і повільною. Тому розробка ефективних інструментів для автоматизованої семантичної обробки є надзвичайно актуальною. Це дозволяє оптимізувати роботу з великими масивами даних, знаходити приховані закономірності та виділяти корисну інформацію для подальшого використання.

Метою даної роботи є створення програмного модуля, що зможе не лише збирати дані з публічних джерел, але й здійснювати їхню семантичну обробку – виділяти ключові концепції, оцінювати взаємозв’язки між ними та забезпечувати якісну фільтрацію шуму. Завдяки цьому можна буде значно

підвищити якість інформації, що використовується для аналізу, а також швидкість прийняття рішень на основі отриманих даних.

Таким чином, тема дипломної роботи «Аналіз і розробка програмного модуля семантичної обробки даних OSINT» є актуальною та важливою для розвитку сучасних систем збору та обробки інформації. Ефективне використання відкритих джерел за допомогою автоматизованих методів аналізу може забезпечити критично важливу інформацію, що допомагає у вирішенні завдань безпеки, управління та аналітики у багатьох сферах діяльності.

1 СУЧАСНИЙ СТАН І АКТУАЛЬНІСТЬ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1.1 Критичний аналіз літературних джерел за темою

OSINT (Open Source Intelligence) – це процес збору та аналізу інформації з відкритих джерел, яка є загальнодоступною. Ця інформація може бути використана для отримання важливих знань у різних сферах діяльності, таких як кібербезпека, бізнес-аналітика, національна безпека, журналістика та кримінальні розслідування.

Алгоритм дій при пошуку інформації OSINT можна поділити на три простих етапи: вибір джерел інформації, збір даних та аналіз зібраних даних.

Джерелами інформації в залежності від завдання дослідження можуть виступати наступні:

- соціальні мережі – які зазвичай містять велику кількість персональної інформації, які люди надають самі, що може використовуватися при дослідженні певної особи чи групи осіб, часто використовується у кримінальних дослідженнях;
- новинні ресурси, блоги – місять матеріали, статті на різні теми і публічну реакцію на них, що допомагає у контексті політичних, кримінальний чи економічних досліджень;
- державні чи публічні записи – це публічні бази даних, такі як EDGAR (US SEC база даних), які надають доступ до великої кількості різноманітної інформації, такої як: патенти, записи судових рішень, реєстраційні данні компаній і тому подібне;
- наукові або академічні публікації – сервісі як Google Schoolar дають можливість шукати наукові матеріали на різні теми, а також авторів та їх співавторів, що дозволяє будувати зв’язки між різними науковими діячами;

- інформація про домен чи IP адресу – лише знаючи домен або адресу вузла у мережі інтернет можна дізнатися багато технічної інформації, такої як: власника домена, геолокацію, інформацію про інтернет провайдера, відкриті порти і так далі;
- пошукові системи та інтернет архіви – інструменти як Google дозволяють знайти безліч веб сайтів, документів, фото та відео на різноманітні теми, а сервіс як Wayback Machine дозволяє додатково дізнатися стан специфічної сторінки у певний момент у часі, що надає ще більше інформації для OSINT розслідування;
- dark web – це більш ризикове джерело інформації, яке натомість надає доступ до такої інформації як чорні ринки або дані про злочинців.

Наступним етапом під час проведенні OSINT розслідування є збір інформації, який можна поділити на мануальний та автоматизований. Мануальний збір це найпростіший спосіб але і найбільш працеміский оскільки потребує велику кількість людського часу.

Автоматизований збір дозволяє зменшити участь людини у процес збору та полегшити його. До методів такого способу входять наступні:

- веб скрапери, API пошук
- виділення мета інформації (геолокації, часу, і т.д.) з медіа ресурсів
- сканування мережі

Веб скрапери – це інструменти або програми, які автоматизовано отримують інформацію з вебсайтів шляхом зчитування та обробки HTML-коду сторінок. Вони дозволяють користувачам витягати дані, які можуть бути недоступні через стандартні методи доступу, наприклад, через API. Веб скрапінг використовується для збору інформації з різноманітних джерел, включаючи новини, соціальні мережі, магазини, блоги та форуми. Основний процес веб скрапінгу полягає у відправці HTTP-запиту до сторінки, отриманні

її вихідного коду, а потім аналізі та вилученні необхідної інформації за допомогою парсера.

Робота веб скраперів полягає в тому, щоб знаходити елементи HTML-коду (наприклад, теги, класи, ID), які відповідають певним структурам даних (заголовки, ціни, текстові блоки), і витягати ці дані для подальшого використання. Важливо відзначити, що веб скрапери повинні адаптуватися до змін на вебсайтах, оскільки оновлення їхньої структури може призвести до некоректного збору інформації. Також веб скрапінг часто потребує обходу обмежень, встановлених власниками вебсайтів, таких як блокування IP-адрес або використання CAPTCHA.

На відміну від веб скраперів, отримання інформації через API є більш структурованим і надійним процесом. API (Application Programming Interface) – це набір правил, що дозволяє одній програмі взаємодіяти з іншою. API надають доступ до певної інформації чи функцій вебсайту через стандартизовані запити, зазвичай у форматі JSON або XML. Взаємодія через API зазвичай є легальною і передбаченою самим розробником ресурсу. API дають змогу отримувати дані без необхідності аналізувати HTML-код і долати захисні механізми.

Процес виділення інформації з медіа-даних полягає в аналізі метаданих, вмісту медіа-файлів (зображень, відео, аудіо), а також аналізі контексту їх публікації. Медіа-дані, такі як фотографії або відео, містять приховану інформацію, зокрема час і місце створення файлу, які зберігаються в метаданих, відомих як EXIF (Exchangeable Image File Format). Ці метадані можуть містити інформацію про модель камери або пристрою, налаштування камери, координати GPS, дату та час зйомки. Ці дані дозволяють визначити, коли і де був зроблений медіа-файл, що є критично важливим для аналізу подій або виявлення джерела інформації.

Визначення локації зазвичай здійснюється шляхом аналізу координат GPS, якщо вони включені в метадані. Також використовуються методи аналізу контексту зображення або відео, такі як розпізнавання об'єктів, пейзажів або

будівель. Це дозволяє визначити географічне розташування навіть у випадках, коли GPS-дані відсутні. У таких випадках алгоритми розпізнавання можуть порівнювати елементи на зображенні з існуючими базами даних супутникових знімків або іншими джерелами візуальної інформації.

Сканування мережі для пошуку інформації в контексті OSINT (відкритої розвідки) є ключовим етапом збору технічних даних про цільову мережу або систему. Цей процес включає аналіз публічно доступних IP-адрес і портів з метою виявлення активних хостів, служб та вразливостей, які можуть бути використані для подальшого дослідження або атак. Основна мета сканування мережі – отримати якомога більше інформації про структуру мережі та стан її компонентів без безпосереднього проникнення всередину систем.

Port-сканування є більш детальним процесом, який дозволяє виявляти, які порти на знайдених пристроях відкриті та які служби працюють через ці порти. Порти – це точки входу до різних сервісів на сервері або іншому мережевому пристрої, через які відбувається передача даних. Наприклад, веб-сервер зазвичай працює через порт 80 або 443, тоді як FTP – через порт 21. Відкриті порти можуть надати важливу інформацію про типи сервісів, що працюють на системі, і можливі вразливості. Наприклад, виявлення відкритого порту, через який працює застарілий сервіс, може свідчити про можливі вразливості, які можуть бути використані для проникнення.

Такі методи сканування використовуються як у легітимних цілях, так і для потенційних кібератак. У контексті OSINT вони можуть використовуватися для збору даних про потенційно небезпечні системи або для дослідження мереж з метою аналізу ризиків. Сканування портів і IP допомагає виявити вразливі служби, недоліки в конфігурації або застарілі програми, що можуть представляти загрозу для безпеки мережі. Такі методи також застосовуються в рамках тестування на проникнення (penetration testing), коли кіберфахівці легально досліджують мережі для виявлення потенційних слабких місць з метою їх усунення.

Для виконання IP та порт-сканування використовуються різні спеціалізовані інструменти. Наприклад, Nmap є одним із найпопулярніших інструментів для сканування мереж, який дозволяє виконувати як базове IP-сканування, так і детальніше сканування портів і служб. Інший відомий сервіс – Shodan, який дозволяє знаходити підключені до інтернету пристрої і отримувати інформацію про відкриті порти, служби та навіть дані про вразливості. Ці інструменти є потужними інструментами для збору інформації, що можуть бути використані як у легальних цілях безпеки, так і для більш загального аналізу мережевої інфраструктури в рамках OSINT-розвідки.

Для деяких технічних даних неможливо провести семантичний аналіз. Ці дані зазвичай мають окреме чітке значення, що робить даний аналіз надлишковим. До них відносять: геолокацію, IP інформацію, дані про домени, вразливості інтернет ресурсів. Тому надалі акцентуватися увага на подібному аспекту OSINT не буде.

Аналіз зібраних даних – це критичний етап, на якому необроблена інформація перетворюється на корисну аналітику для прийняття рішень. Одним із важливих інструментів цього процесу є семантичний аналіз, який допомагає автоматично визначити емоційне забарвлення тексту або коментарів у соціальних мережах, новинах, блогах тощо. Це може бути позитивна, негативна або нейтральна тональність, що дозволяє зрозуміти загальний настрій стосовно певної події, організації, бренду або особи.

Семантичний аналіз працює на основі машинного навчання та алгоритмів обробки природної мови (NLP). Процес аналізу включає такі етапи, як вилучення тексту з джерела (наприклад, коментарів у соцмережах або статей), очищення даних (усунення шумів, таких як емодзі, помилки або нерелевантна інформація), і класифікація тексту за тональністю. За допомогою спеціальних алгоритмів система аналізує семантичну структуру тексту, контекст слів і ключові фрази, щоб зробити висновок про загальний емоційний відтінок тексту.

Один з головних висновків, який можна зробити за допомогою семантичний аналіз, – це розуміння ставлення громадськості до конкретної теми або об'єкта. Наприклад, аналізуючи сотні або тисячі коментарів про продукт, можна оцінити загальне сприйняття цього продукту споживачами, визначити, чи є настрої переважно позитивними або негативними, і відповідно скоригувати маркетингову стратегію. В OSINT розвідці семантичний аналіз може бути використаний для вивчення настроїв щодо політичних подій, соціальних рухів або публічних фігур. Це допомагає зрозуміти потенційні ризики або можливості, виявити точки напруги у суспільстві або передбачити можливі конфлікти.

Крім того, семантичний аналіз дозволяє ідентифікувати ключових впливових осіб, що формують настрій аудиторії. Якщо певний користувач або група поширюють негативні чи позитивні коментарі, аналіз їхньої тональності дозволяє виділити, хто є основними генераторами думок. Це може бути корисно як у бізнес-аналітиці, так і в політичному моніторингу.

Наприклад, у контексті кризового моніторингу семантичний аналіз може показати динаміку змін у тональності повідомлень щодо організації або події. Якщо до цього була стабільна нейтральна чи позитивна тональність, але раптово зростає кількість негативних відгуків, це може свідчити про наближення кризи, яку необхідно вирішувати.

1.2 Актуальність теми кваліфікаційної роботи

У сучасних умовах інформаційного суспільства та стрімкого розвитку технологій, важливість збору й аналізу інформації з відкритих джерел (OSINT) постійно зростає. Інформація, яка публічно доступна через вебсайти, соціальні мережі, наукові публікації, блоги та форуми, є цінним ресурсом для різних сфер, таких як кібербезпека, бізнес-аналітика, національна безпека, журналістика та кримінальні розслідування.

Оскільки обсяги доступної інформації продовжують збільшуватися, зростає необхідність у розробці ефективних методів її обробки. Проте,

основною проблемою є те, що більшість цих даних є неструктурованими або слабо структурованими, що ускладнює їх аналіз та інтерпретацію. Семантична обробка даних OSINT дозволяє перетворювати ці неструктуровані дані у зрозумілу та аналітично цінну інформацію, забезпечуючи виявлення сутностей, визначення зв’язків, категоризацію даних та фільтрацію інформаційного шуму.

Розробка програмного модуля, який автоматизує процес семантичної обробки OSINT даних, дозволить значно покращити ефективність і швидкість аналізу інформації, що є критично важливим для прийняття рішень в різних галузях. Семантична обробка також відкриває нові можливості для аналізу тональності текстів, що є ключовим аспектом для маркетингових, політичних, медійних та фінансових аналітичних процесів.

Таким чином, розробка програмного модуля семантичної обробки даних OSINT є надзвичайно актуальною та перспективною темою для кваліфікаційної роботи, яка сприятиме підвищенню ефективності збору та аналізу інформації в сучасному світі.

1.3 Аналіз існуючих аналогів

Наразі існує велика кількість різних інструментів для пошуку інформації з відкритих джерел (OSINT). Розглянемо кілька популярних варіантів.

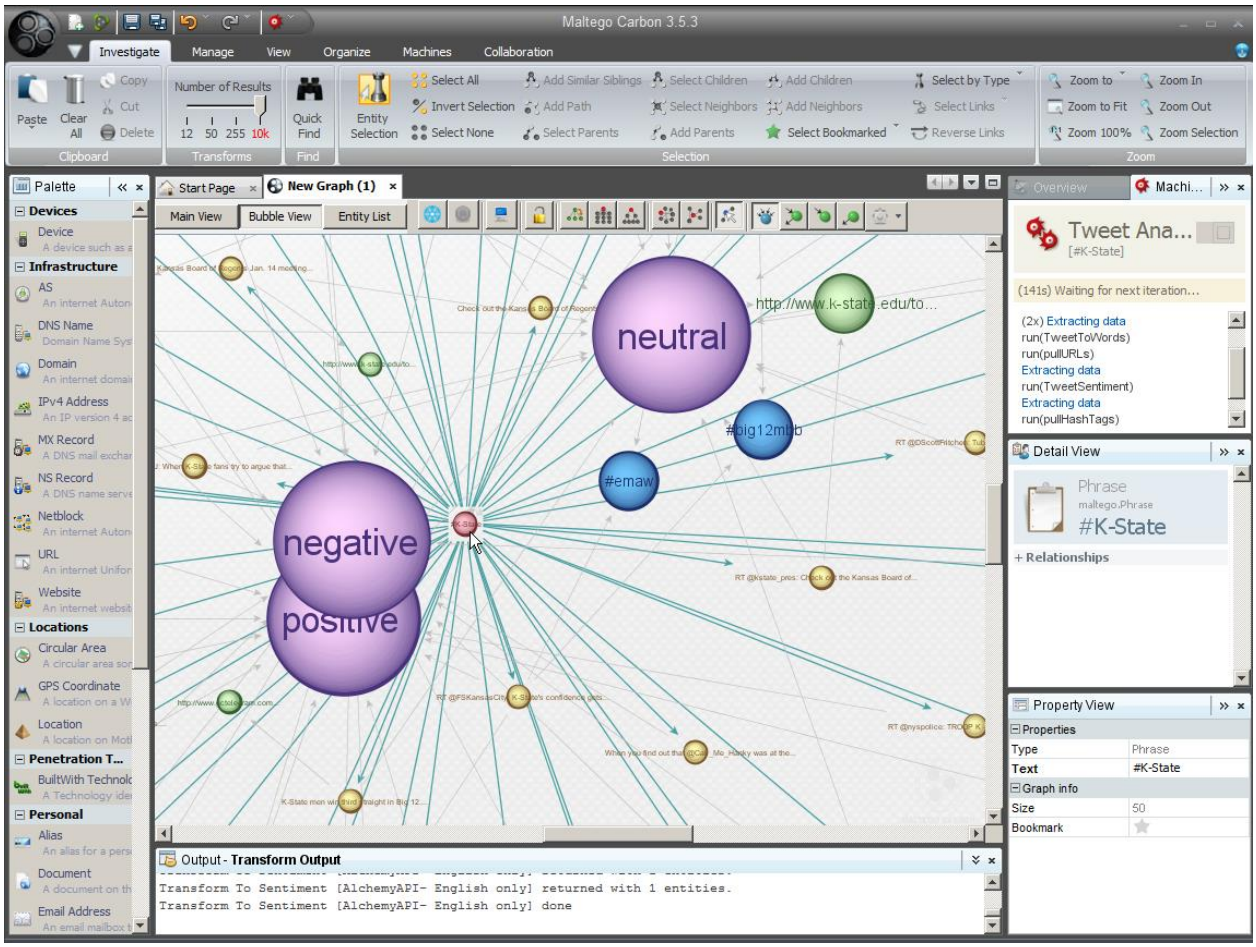

Рисунок 1.1 – Інтерфейс системи Maltego

Maltego [4] – це потужна платформа для збору, обробки та візуалізації відкритих даних (OSINT). Вона дозволяє користувачам збирати інформацію з різних джерел, включаючи соціальні медіа, вебсайти, бази даних та інші ресурси, і візуально відображати зв’язки між цими даними. Maltego особливо корисна для розслідувань у сфері кібербезпеки, кримінальних розслідувань, аналізу мережевих зв’язків та виявлення загроз. Інструмент підтримує використання семантичного аналізу через інтеграції з додатковими сервісами або через розширення можливостей збору даних. Однією з найсильніших сторін Maltego є візуалізація даних. Платформа створює графи, які відображають зв'язки між об'єктами, такими як особи, організації, адреси електронної пошти, домени, IP-адреси тощо. Це дозволяє дослідникам візуально розпізнавати приховані зв’язки та робити висновки.

Із мінусів даної платформи можна виділити велику вартість повноцінної версії, яка надає можливість використовувати модуль семантичного аналізу, використання великої кількості обчислювальних ресурсів при дослідженні та залежність від сторонніх інтеграцій.

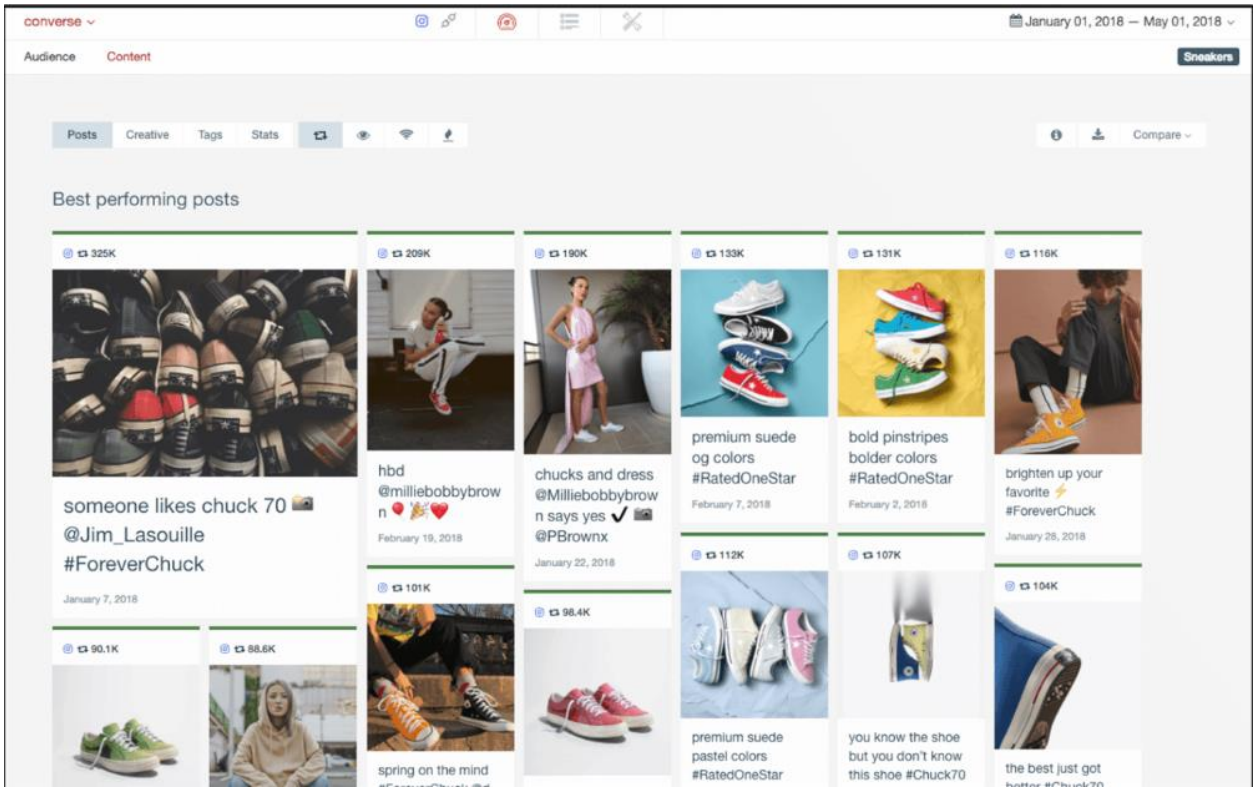

Рисунок 1.2 – Інтерфейс системи Pulsar Platform

Pulsar Platform [5] – це багатофункціональна платформа для моніторингу та аналізу соціальних медіа, створена для виявлення та аналізу громадських настроїв, тенденцій, та взаємодій у цифровому просторі. Платформа поєднує методи збору даних з різних джерел (соціальні медіа, новини, блоги, форуми) та аналітичні інструменти, зокрема семантичний аналіз, щоб допомогти користувачам глибше зрозуміти настрої аудиторії, ринкові тенденції та загальну реакцію громадськості.

Pulsar особливо корисна для маркетологів, PR-фахівців, дослідників та аналітиків, оскільки дозволяє відстежувати реакції на кампанії, бренди, ключові теми та інші важливі соціальні події. Pulsar дозволяє налаштовувати

сповіщення для відстеження нових трендів, згадок або змін у тональності. Це допомагає користувачам оперативно реагувати на зміни в громадській думці або раптові інформаційні кризи.

З недоліків можна знову виділити велику вартість застосунку, що робить його недоступним для малих та середніх підприємств, а також вузько направленість засобу.

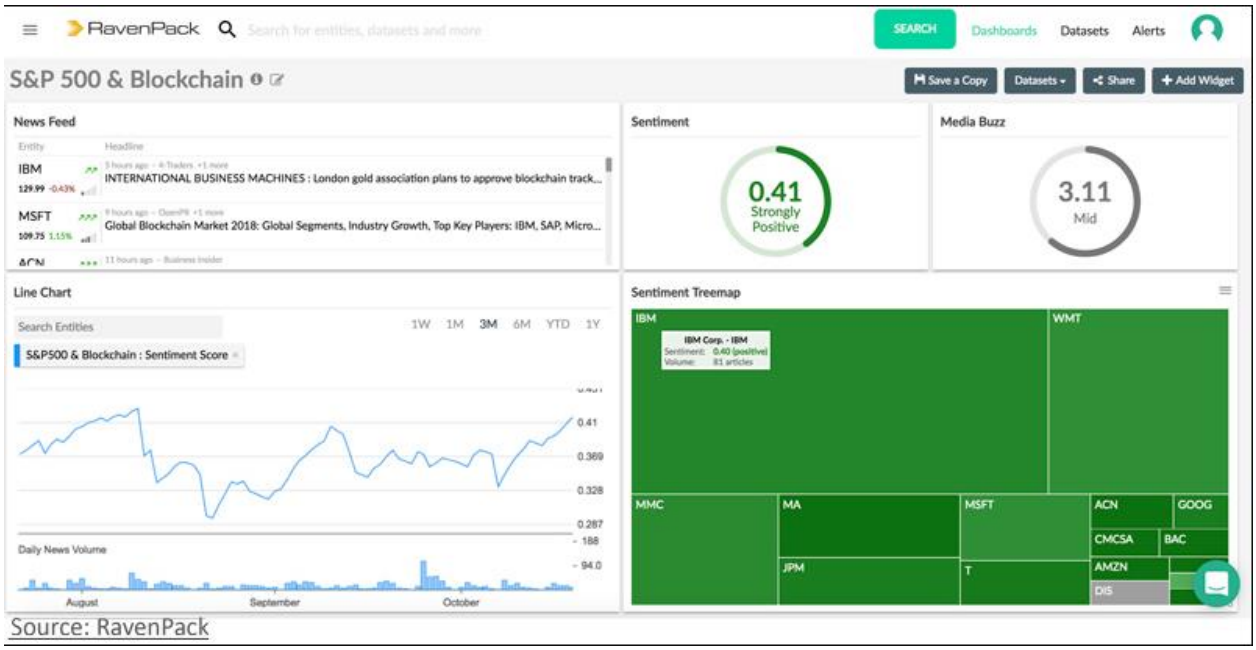

Рисунок 1.3 – Інтерфейс системи RavenPack

RavenPack [6] є потужним інструментом для аналізу великих обсягів даних яка спеціалізується на обробці новин, соціальних медіа, звітів, та інших текстових джерел що робить її незамінною для фінансового сектору. Платформа застосовує штучний інтелект та алгоритми обробки природної мови (NLP). Вона дозволяє фінансовим аналітикам і інвесторам отримувати актуальні дані про ринкові події, настрої та ризики, що допомагає приймати більш обґрунтовані рішення. RavenPack дозволяє автоматично виявляти важливі події в новинах і соціальних медіа, такі як злиття та поглинання, корпоративні події, випуски звітів, політичні події тощо. Система може класифікувати події за ступенем їхньої важливості та впливу на ринок.

Платформа переважно орієнтована на фінансовий сектор, що обмежує її корисність для інших галузей. І хоч даний застосунок пропонує потужні можливості, вона може вимагати певного часу для освоєння, особливо для нових користувачів.

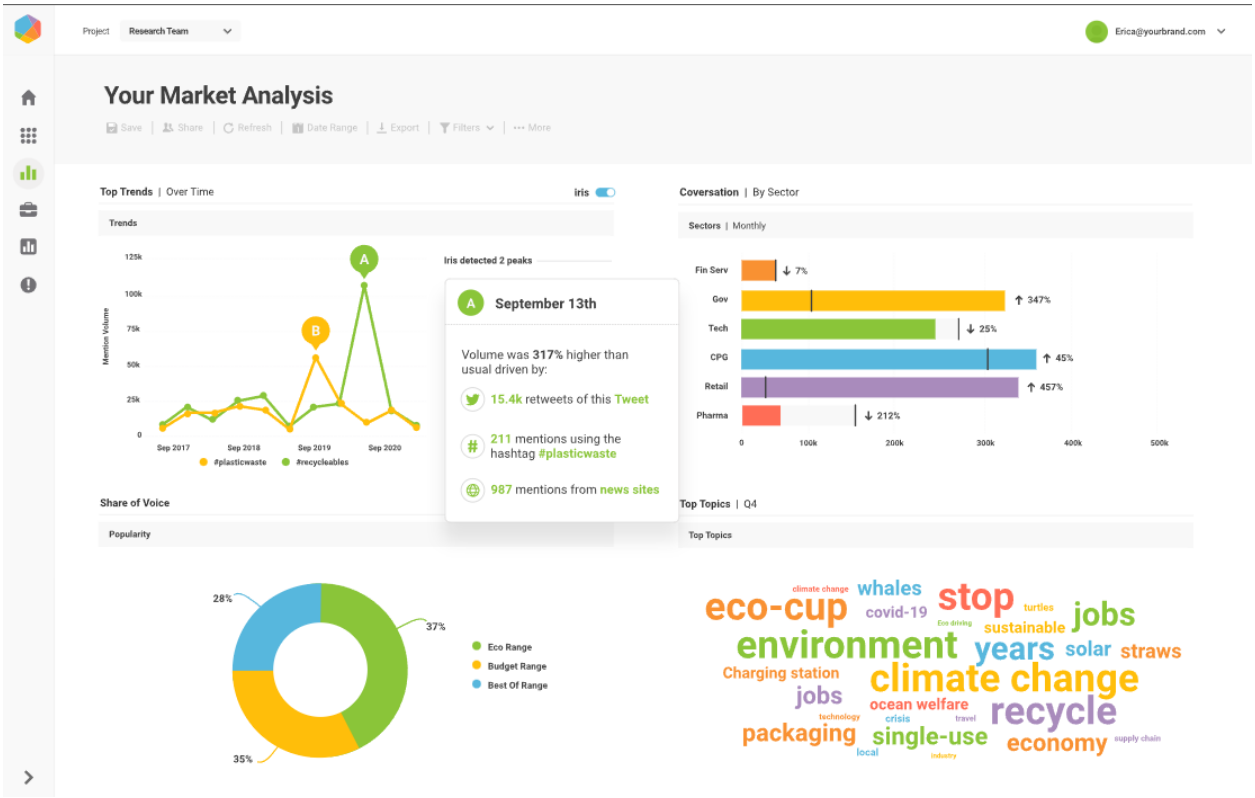

Рисунок 1.4 – Інтерфейс системи Brandwatch Analytics

Brandwatch Analytics [7] – платформа для моніторингу соціальних медіа та аналізу відкритих даних, яка дозволяє компаніям відстежувати згадки про свої бренди, продукти або ключові теми. Brandwatch Analytics допомагає компаніям приймати обґрунтовані рішення на основі громадської думки та реакцій у соціальних мережах, новинах, форумах та інших джерелах. Вона збирає дані з багатьох джерел, включаючи Twitter, Facebook, Instagram, YouTube, новинні сайти, блоги та форуми і автоматично визначає тональність кожного згадування (позитивна, негативна або нейтральна), допомагаючи брендам зрозуміти, як до них ставляться користувачі.

Система використовується маркетологами для оцінки ефективності своїх кампаній, аналізу конкурентів та власного бренду.

Із недоліків даного застосунку можна знову виділити велику вартість для малих компаній а також сфокусованість на соціальних мережах, що означає меншу підтримку звичайних новин, форумів та ін.

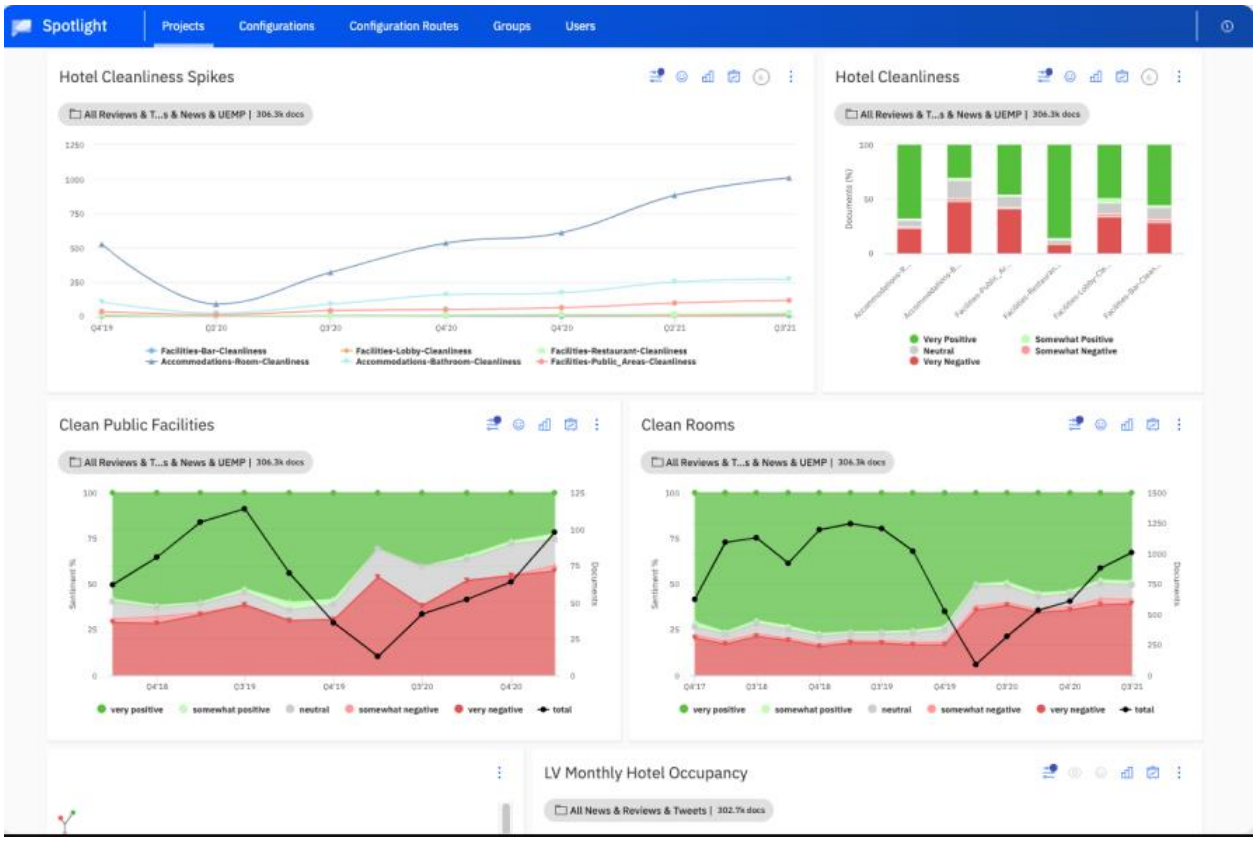

Рисунок 1.5 – Інтерфейс системи Lexalytics

Lexalytics [8] – це провідна платформа для аналізу текстових даних, яка спеціалізується на обробці природної мови (NLP) та автоматизації аналізу тексту з широкого спектра джерел. Платформа була створена для того, щоб допомогти організаціям витягувати корисну інформацію з різних типів текстових даних, таких як публікації в соціальних медіа, відгуки клієнтів, новинні статті, опитування та інші текстові документи. Це дозволяє компаніям швидше та ефективніше аналізувати великі обсяги інформації, надаючи інструменти для автоматизованої обробки та аналізу змісту.

Однією з ключових переваг Lexalytics є її здатність автоматично витягати цінні дані з тексту, використовуючи методи обробки природної мови. Платформа дозволяє виконувати аналіз тональності, що допомагає виявляти емоційне забарвлення тексту, визначати позитивну, негативну або нейтральну реакцію аудиторії на певні події чи продукти. Крім того, Lexalytics підтримує тематичну класифікацію, яка дозволяє автоматично розподіляти текст на різні категорії залежно від його змісту. Інші важливі можливості платформи включають виявлення емоцій, що дозволяє вивчати більш тонкі аспекти людських почуттів у текстах, та екстракцію ключових термінів, що допомагає виділяти важливі концепти та терміни з великих обсягів інформації.

Lexalytics також надає можливість для бізнес-аналітики, де дані, витягнуті з текстів, використовуються для прийняття важливих бізнес-рішень. Платформа допомагає компаніям покращувати обслуговування клієнтів, аналізуючи їх відгуки та коментарі, що дає можливість швидко реагувати на їхні потреби та підвищувати рівень задоволеності. Крім того, Lexalytics застосовується для аналізу репутації бренду, де компанії можуть відстежувати, як їх продукція або послуги сприймаються аудиторією, що дозволяє оперативно реагувати на потенційні кризи або посилювати позитивний імідж.

Дана платформа є однією з найкращих для семантичного аналізу даних, незважаючи на факт, що вона не може автоматично збирати дані для аналізу, їх потрібно надавати окремо. Також вона потребує чи не малих технічних знань для використання та інтеграції.

Проаналізувавши наявні системи збору та семантичного аналізу даних OSINT можна виділити їх наступні ключові моменти:

- збір даних проводиться переважно у соціальних мережах через їх популярність та кількість наявної інформації;
- аналіз новин та блогів не у всіх системах присутній, але він також розширює можливості системи та покращує точність аналізу;

- більшість систем є дорого вартісними, що зменшує кількість їх потенційних користувачів.

1.4 Цілі та завдання кваліфікаційної роботи

Тема кваліфікаційної роботи "Аналіз і розробка програмного модуля семантичної обробки даних OSINT" є актуальною у зв'язку з постійним зростанням обсягів відкритих джерел інформації, таких як соціальні медіа, новини, наукові публікації та інші публічно доступні дані. У сучасних умовах аналітична обробка цих даних має критичне значення для різних галузей, включаючи бізнес-розвідку, кібербезпеку, журналістику, кримінальні розслідування та національну безпеку. Для ефективної роботи з великими масивами неструктурованої інформації необхідно застосовувати методи семантичної обробки, що дозволяють автоматизувати збір, аналіз і структурування даних.

Метою кваліфікаційної роботи є аналіз існуючих технологій семантичної обробки даних OSINT та розробка програмного модуля, який автоматизує процес аналізу великих обсягів даних з відкритих джерел. Предметом роботи є процеси збору, обробки та аналізу неструктурованих даних з відкритих джерел за допомогою методів семантичної обробки.

Для досягнення поставленої мети необхідно виконати наступні завдання:

- провести огляд існуючих методів і технологій семантичної обробки даних OSINT;
- визначити ключові завдання для обробки неструктурованих даних, такі як вилучення сутностей, виявлення зв'язків між даними та категоризація інформації;
- розробити алгоритм для автоматизації семантичного аналізу та обробки OSINT даних;
- розробити програмний модуль, який інтегрує зазначені алгоритми для автоматизації процесів збору та аналізу даних;

- провести тестування розробленого модуля на реальних даних з відкритих джерел та оцінити його ефективність.

В результаті виконання кваліфікаційної роботи буде створений програмний модуль, який дозволить підвищити ефективність роботи з даними OSINT шляхом автоматизації їхнього семантичного аналізу та обробки.

2 РОЗРОБКА ФУНКЦІОНАЛЬНОЇ СХЕМИ ТА АЛГОРИТМІВ

Функціонал розроблюваного програмного модуля для семантичної обробки OSINT даних передбачає можливість користувачу або окремій системі ефективно аналізувати та кластеризувати текстову інформацію з використанням навченої математичної моделі. Це включає автоматичне розбиття тексту на семантичні частини, аналіз їх змісту та пошук відповідної інформації у відкритих джерелах.

З урахуванням поставлених цілей і задач, проектування програмного забезпечення модуля включає розробку структурної та функціональної діаграм, діаграми прецедентів, UML-діаграми для відображення структури бази даних, а також блок-схем алгоритмів роботи модуля. Ці елементи дозволяють відобразити архітектуру модуля, його основні функції та логіку роботи, забезпечуючи прозорість розробки та можливість подальшого масштабування системи.

2.1 Розробка та опис функціональної схеми системи

Основна ідея системи полягає в створенні модуля семантичної обробки даних OSINT, але для демонстрації його роботи було також розроблено модуль пошуку даних у відкритих джерелах. Для такої системи добре підходить клієнт-серверна архітектура, де клієнтами виступають користувачі, а сервер надає API, тому може бути інтегрований як незалежний сервіс в мікро сервісну інфраструктуру.

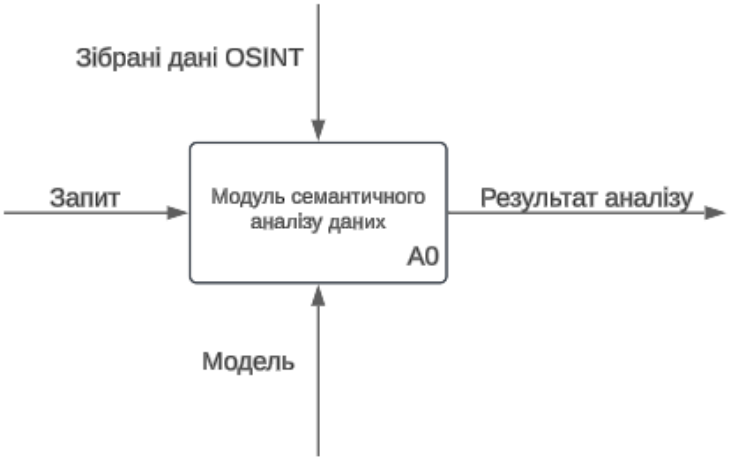

Рисунок 2.1 – Функціональна діаграма системи першого рівня деталізації

На рисунку 2.1 зображена діаграма першого рівня деталізації для модуля семантичного аналізу даних.

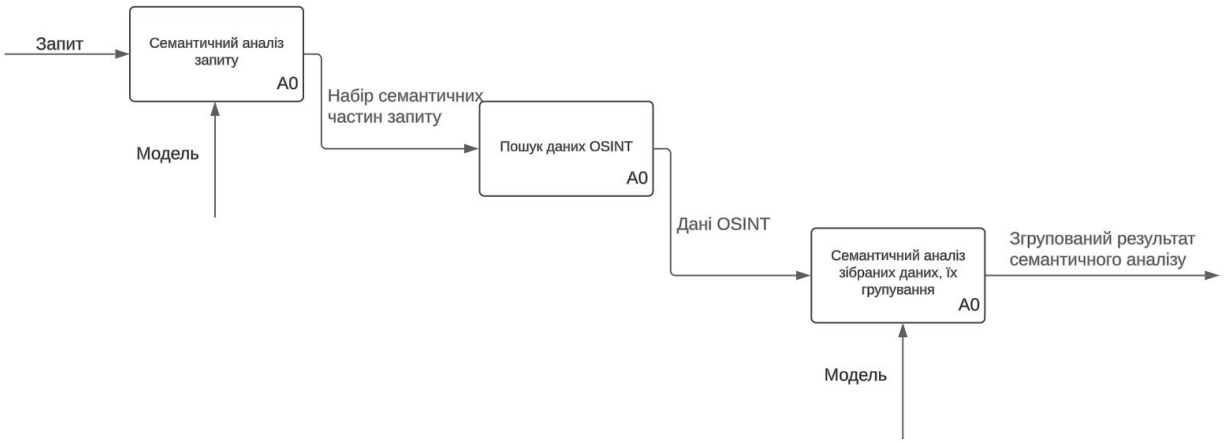

Рисунок 2.2 – Функціональна діаграма системи другого рівня деталізації

На рисунку 2.2 зображена функціональна діаграма розроблюваного модуля другого рівня деталізації. На ній видно що загальний процес обробки даних відбувається у декілька етапів. На початку процесу система очікує запит від користувача. Даний запит за допомогою навченої моделі для семантичного аналізу обробляє запит користувача та намагається поділити запит на відповідні семантичні частини. На цьому етапі виокремлюються імена, власні

назви (країни, компанії), імена користувачів, номера телефонів, електронні адреси та ін.

На наступному етапі на основі даних з попереднього кроку збираються дані з відкритих джерел. Відповідно від типу даних залежить інтернет ресурс, на якому відбувається пошук інформації.

На останньому етапі зібрані дані OSINT додатково обробляються модулем семантичного аналізу для групування даних основуючись на семантичному змісті.

2.2 Розробка та опис діаграми прецедентів

Розглянемо діаграму прецедентів:

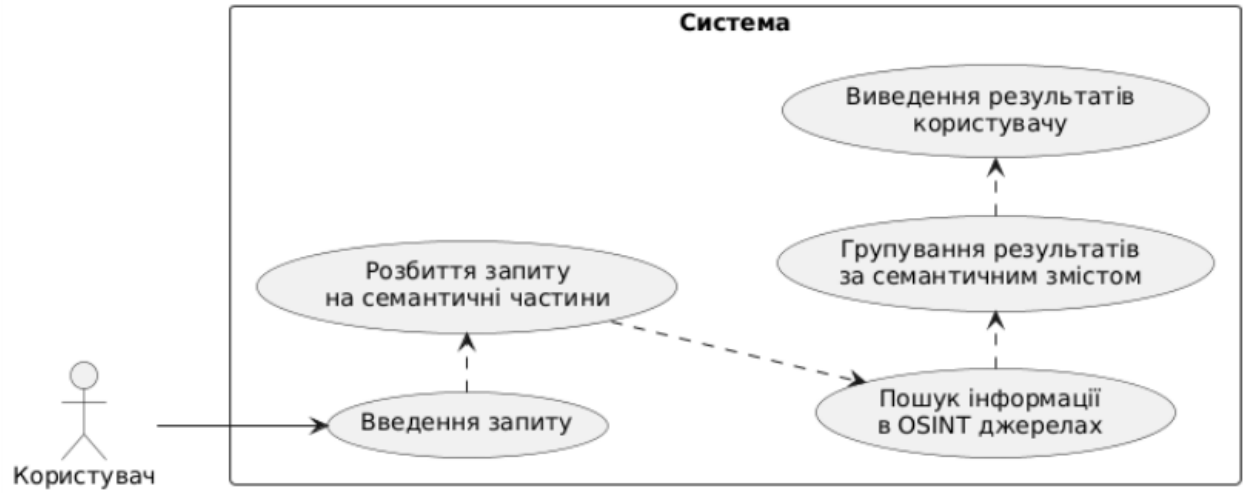

Рисунок 2.3 – Діаграма прецедентів

Користувач має єдину доступну дію: введення запиту для пошуку і аналізу. Після підтвердження введених даних підправляється HTTP запит на сервер для обробки.

2.3 Розробка алгоритму роботи програми

Рисунок 2.4 – Блок-схема алгоритму роботи системи

На рисунку 2.4 зображена блок-схема алгоритму роботи системи семантичного аналізу даних. Спочатку користувач вводить запит для пошуку. Запит аналізується моделлю, яка виокремлює з нього різні семантичні дані. Вона націлена на пошук наступної інформації:

- власних імен, таких як імена людей, назви компаній, населених пунктів, тощо;
- імен користувачів;

- адрес електронної пошти;
- посилань;
- номерів телефонів.

Зібрані та згруповані семантичні далі відправляються до модулю пошуку, який порівнюючи тип інформації обирає групу інтернет ресурсів, на яких треба здійснювати пошук цієї інформації. Наприклад, якщо було знайдено ім’я людини, система спробує знайти наукові роботи з участю даної людини за допомогою ресурсу Google Schoolar [9].

Останнім етапом роботи системи є аналіз зібраних даних за допомогою моделі аналізу семантичних даних, яка перевіряє результати пошуку і групує знову дані за їх семантичним змістом. В результаті роботи отриманий результат повертається користувачеві та відображається у інтерфейсі.

2.4 Розробка інтерфейсу користувача

Враховуючи вимоги до інтерфейсу користувача даної системи було розроблено простий та зручний інтерфейс який цілком задовольняє потреби користувача розроблюваного модуля.

Інтерфейс складається з текстового поля для введення запиту користувача та блоку для відображення результатів роботи програми.

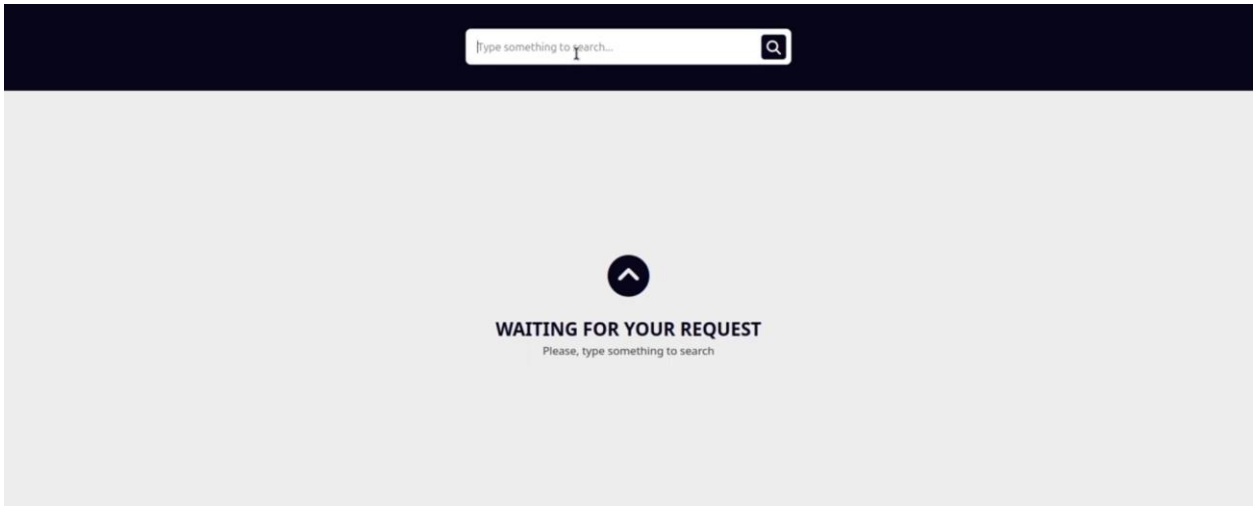

Рисунок 2.5 – Інтерфейс користувача до введення запиту

На рисунку 2.5 зображено інтерфейс користувача до введення запиту для пошуку та аналізу. На даному етапі відображається лише поле для введення запиту.

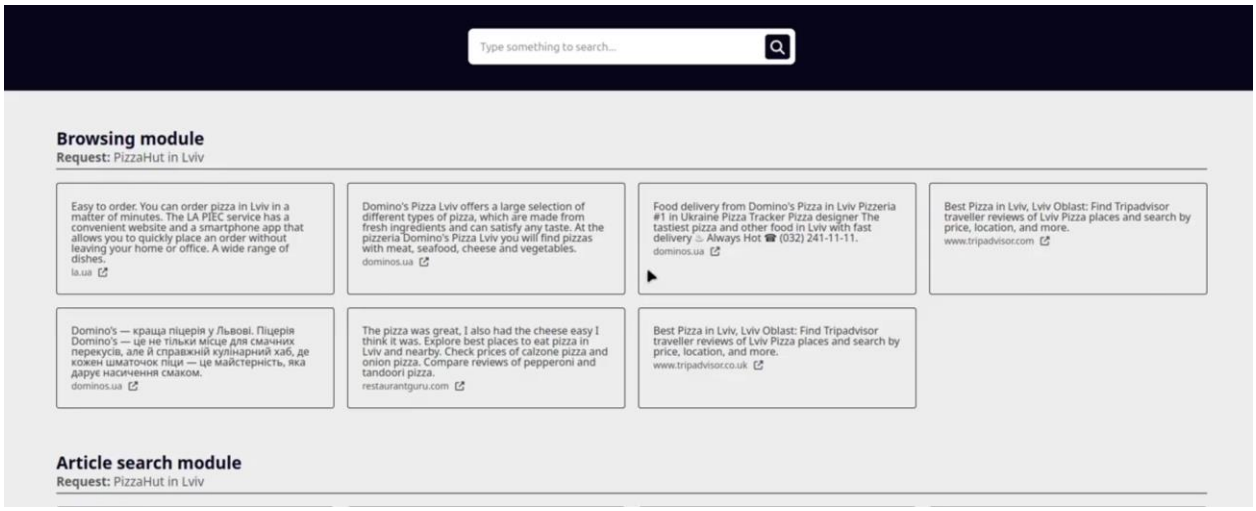

Рисунок 2.6 – Інтерфейс користувача з результатом роботи модуля

На рисунку 2.6 зображено інтерфейс користувача після обробки запиту. На даному етапі з’являється результат роботи згрупований по модулям, які відповідали за пошук даних разом і з даними, які слугували у якості початкових даних для роботи конкретного алгоритму пошуку даних.

3 РОЗРОБКА СТРУКТУР ДАНИХ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Аналіз обраного середовища програмування

Для написання коду системи було обрано інтегровану середу розробки PyCharm. PyCharm – це інтегроване середовище розробки (IDE) для Python, розроблене компанією JetBrains. Воно надає повний набір інструментів для зручної та ефективної розробки Python-додатків.

Середовище розробки PyCharm має багато функцій, які полегшують процес програмування. Однією з ключових є підтримка автоматичного завершення коду, що допомагає розробникам писати код швидше та зменшує кількість помилок. PyCharm також має розумний аналізатор коду, який виявляє синтаксичні помилки та пропонує рекомендації щодо їх виправлення. Крім того, середовище підтримує перевірку типів змінних і статичний аналіз коду, що підвищує безпеку та надійність програмного забезпечення.

PyCharm інтегрується з популярними системами контролю версій, такими як Git, Mercurial і SVN, що дозволяє розробникам працювати з репозиторіями безпосередньо у середовищі розробки. Це значно спрощує управління версіями коду та сприяє командній співпраці.

PyCharm пропонує зручний вбудований редактор, який підтримує роботу з HTML, CSS і JavaScript. У ньому є підсвічування синтаксису, авто доповнення, інтеграція з інструментами перевірки стандартів та багато інших функцій, що полегшують розробку фронтенду.

Середовище має розширену підтримку Python-фреймворків, таких як Django, Flask, Pyramid та інших. PyCharm включає вбудовані інструменти для роботи з цими фреймворками: генератори коду, дебагери, менеджери середовища і автодоповнення, що дозволяє зосередитися на розробці функціональності без необхідності витрачати час на налаштування середовища.

PyCharm є потужним інструментом для розробки, що забезпечує високу продуктивність, зручність та адаптивність під різні проекти.

3.2 Аналіз обраних технологій

Для реалізації модулю було обрано мову програмування Python через його простоту, великий вибір бібліотек як для розробки бекенд так і фронтенд частин веб-сайту. Також, дана мова програмування відома у сфері машинного навчанні і надає велику кількість інструментів у даній області розробки.

Для бекенд частини було обрано фреймворк FastApi. FastAPI – це сучасний веб-фреймворк для створення API, що використовує можливості Python 3.6+, зокрема типи підказок. Він забезпечує зручність і високу ефективність розробки. Завдяки інтеграції з OpenAPI, FastAPI автоматично генерує інтерактивну документацію, яка значно полегшує тестування і використання API. Крім того, цей фреймворк побудований на базі ASGI-сервера Starlette і HTTP-обробника Uvicorn, що забезпечує високу продуктивність і асинхронність. Типи підказок Python спрощують читання коду, його обслуговування та знижують кількість помилок під час розробки. FastAPI також має вбудовані механізми для автентифікації й авторизації, що робить його універсальним інструментом для створення API.

У порівнянні з Django, FastAPI має кілька важливих переваг. По-перше, його продуктивність значно вища завдяки нативній підтримці асинхронного коду, тоді як Django лише частково підтримує асинхронність і робить це складніше. По-друге, FastAPI фокусується на створенні API і забезпечує автоматичну генерацію документації, що є вагомою перевагою для швидкої розробки RESTful API. Django, у свою чергу, більшою мірою орієнтований на створення повноцінних веб-додатків із комплексною логікою та інтеграцією різних модулів. FastAPI також дозволяє працювати з типами даних на рівні Python, що полегшує валідацію та перевірку, тоді як Django для цього використовує форми та моделі, які менш інтегровані з типізацією.

FastAPI зазвичай обирають для проектів, що потребують швидкодії, роботи з асинхронними запитами та інтеграції з машинним навчанням. Натомість Django залишається кращим вибором для великих, багатофункціональних проектів, які потребують потужного інструментарію для управління базами даних, адміністрування та інтеграції з фронтендом.

Для фронтенд частини було обрано ще одну бібліотеку мови програмування Python, Jinja2. Jinja2 – це потужна шаблонізаторна бібліотека для Python, яка використовується для створення динамічного HTML, XML або інших текстових форматів. Вона дозволяє розділяти бізнес-логіку і представлення, що є важливим аспектом веб-розробки. Jinja2 інтегрується з популярними веб-фреймворками, такими як Flask і Django, але може бути використана і автономно.

Переваги Jinja2:

- Простий і зрозумілий синтаксис, що нагадує HTML, робить її зручною навіть для новачків.
- Широкий набір фільтрів і функцій, які спрощують обробку даних у шаблонах.
- Підтримка циклів і умов у шаблонах дозволяє створювати складні динамічні документи.
- Можливість створення макросів для пере використання коду та спрощення структури шаблонів.
- Безпека за рахунок автоматичного екранування HTML, що запобігає XSS-атакам.
- Гнучка система налаштувань і можливість розширення за допомогою власних фільтрів і функцій.
- Висока продуктивність завдяки використанню компіляції шаблонів у байт-код.

Jinja2 є ідеальним вибором для проектів, де потрібно створювати чистий, структурований і безпечний HTML-код із даних, що змінюються динамічно.

Pandas — це одна з найпопулярніших бібліотек у Python, яка забезпечує зручну структуру даних DataFrame для ефективної роботи з даними та їх обробки.

PyTorch — це сучасний і широко використовуваний фреймворк для глибокого навчання, створений дослідницькою лабораторією Facebook AI Research (FAIR). Його розроблено з метою забезпечення високої гнучкості та швидкості, що особливо важливо в дослідницькій діяльності.

PyTorch має кілька ключових особливостей, які відрізняють його від інших фреймворків для глибокого навчання. Однією з таких переваг є використання динамічного графа обчислень, який на відміну від статичних графів TensorFlow (до версії 2.0), дозволяє змінювати граф під час виконання, що робить PyTorch більш гнучким для роботи з рекурентними нейронними мережами. Синтаксис PyTorch наближений до Python, що робить його інтуїтивно зрозумілим і зручним для розробників. Бібліотека також забезпечує високу швидкість обчислень і демонструє відмінну продуктивність, особливо при роботі з великими обсягами даних. Гнучкість PyTorch зробила його улюбленим інструментом серед дослідників у галузі машинного навчання.

Порівнюючи з TensorFlow, можна зазначити, що обидва фреймворки є потужними, але орієнтованими на різні задачі. TensorFlow більше підходить для виробничих рішень, завдяки підтримці розподілених систем і оптимізації для виробництва, тоді як PyTorch краще підходить для прототипування та досліджень. Останні версії TensorFlow з Keras API значно спростили його використання, але PyTorch все ще вважається більш «pythonic».

Keras, у свою чергу, є високорівневим API, створеним для спрощення роботи з TensorFlow. Попри його простоту і зручність, він менш гнучкий у дослідницьких завданнях, ніж PyTorch.

3.4 Програмна реалізація основних функцій

Однією з найважливіших частин системи є семантичні процесори. Вони відповідають за пошук та аналіз певних семантичних даних, таких як імена, власні назви, номери телефонів і таке інше.

```
class Processor(ABC):
    @property
    @abstractmethod
    def supported_semantic(self) -> list:
        pass

    @abstractmethod
    def process(self, semantic: str, data: str) -> dict|None:
        pass

    @abstractmethod
    def _is_empty(self, result: dict) -> bool:
        pass

    def supports(self, semantic: str) -> bool:
        return semantic.lower() in self.supported_semantic

    def _finalize_process_result(self, semantic: str, module:
str, data: str, result: dict|list) -> dict|None:
        if self._is_empty(result):
            print('EMPTY', semantic, module , result)
            return None

        return {
            'semantic': semantic,
            'module': module,
            'formatted_module': format_module_name(module) +
' module',
            'input': data,
```

```
                'result': result
            }

    class UsernameProcessor(Processor):
        def _is_empty(self, result: dict) -> bool:
            if result == {}:
                return True
            if 'running_errors' in result and
result['running_errors'] != []:
                return True
            return False

        @property
        def supported_semantic(self) -> list:
            return ['full_name', 'username']

        def process(self, semantic:str, data: str) -> dict:
            result = execute_maryam('username_search', ['-q',
rf"{data}"])

            return self._finalize_process_result(semantic,
'username_search', data, result)

    class PhoneNumberProcessor(Processor):
        def _is_empty(self, result: dict) -> bool:
            if result == {}:
                return True
            if 'running_errors' in result and
result['running_errors'] != []:
                return True
            if 'success' in result and result['success'] is
False:
                return True

            if 'valid' not in result and result['valid'] is False:
```

```
                return True

            return False

        @property
        def supported_semantic(self) -> list:
            return ['phone']

        def process(self, semantic:str, data: str) -> dict:
            result = execute_maryam('phone_number_search', ['-
n', rf"{data}"])

            return      self._finalize_process_result(semantic,
'phone_number_search', data, result)

    class DocsProcessor(Processor):
        def _is_empty(self, result: dict) -> bool:
            if type(result) is dict:
                if result == {}:
                    return True
                if    'running_errors'    in    result    and
result['running_errors'] != []:
                    return True
                if 'links' not in result and 'data' not in result:
                    return True

                if 'data' not in result or result['data'] == []:
                    return True
            elif type(result) is list:
                if result == []:
                    return True

            return False

        @property
```

```
        def supported_semantic(self) -> list:
            return ['docs']

        def process(self, semantic:str, data: str) -> dict:
            result  =  execute_maryam('article_search',  ['-q',
rf"{data}"])

            for res_line in result['results']:
                res_line['title'] = res_line['t']
                res_line['link'] = res_line['a']
                res_line['authors'] = res_line['c']
                res_line['description'] = res_line['d']
                res_line['domain']                                =
urlparse(res_line['a']).netloc

                del res_line['t']
                del res_line['a']
                del res_line['c']
                del res_line['d']

            return      self._finalize_process_result(semantic,
'article_search', data, result['results'])

    class WikipediaProcessor(Processor):
        def _is_empty(self, result: dict) -> bool:
            if type(result) is dict:
                if result == {}:
                    return True
                if    'running_errors'    in    result    and
result['running_errors'] != []:
                    return True
                if 'links' not in result and 'data' not in result:
                    return True

                if 'data' not in result or result['data'] == []:
```

```
                return True
            elif type(result) is list:
                if result == []:
                    return True

            return False

        @property
        def supported_semantic(self) -> list:
            return ['full_name', 'occupation', 'country', 'city',
'company']

        def process(self, semantic: str, data: str) -> dict:
            result    =    execute_maryam('wikipedia',    ['-q',
rf"{data}"])

            if 'titles' not in result or 'titles' not in result:
                return   self._finalize_process_result(semantic,
'wikipedia', data, result)

            if result['titles'] is None or result['links'] is
None:
                return   self._finalize_process_result(semantic,
'wikipedia', data, result)

            formated_data = []
            for    title,    link    in    zip(result['titles'],
result['links']):
                title = title[:title.find('[')]
                formated_data.append({'title':  title,  'link':
link})

            return       self._finalize_process_result(semantic,
'wikipedia', data, formated_data)
```

```
    class BrowsingProcessor(Processor):
        def _is_empty(self, result: dict) -> bool:
            if type(result) is dict:
                if result == {}:
                    return True

                if 'results' not in result or result['results']
== []:
                    return True
                if    'running_errors'    in    result    and
result['running_errors'] != []:
                    return True
                if all([v == [] for v in result.values()]):
                    return False
            elif type(result) is list:
                if result == []:
                    return True

            return False

        BROWSERS = [
            'duckduckgo',
            'bing',
            'google',
        ]


        @property
        def supported_semantic(self) -> list:
            return ['full_name', 'occupation', 'country', 'city',
'company']

        def process(self, semantic:str, data: str) -> dict:
            submodule_res   =   execute_maryam('iris',   ['-q',
rf"{data}"])
```

```
            for res_line in submodule_res['results']:
                res_line['link'] = res_line['a']
                res_line['description'] = res_line['d']
                res_line['domain']                                  =
urlparse(res_line['a']).netloc

                del res_line['t']
                del res_line['a']
                del res_line['c']
                del res_line['d']

            return       self._finalize_process_result(semantic,
'browsing', data, submodule_res['results'])

    class GitHubProcessor(Processor):
        def process(self, semantic: str, data: str) -> dict |
None:
            result = execute_maryam('github', ['-q', rf"{data}"])

            return       self._finalize_process_result(semantic,
'github', data, result)

        def _is_empty(self, result: dict) -> bool:
            if result == {}:
                return True
            if     'running_errors'     in     result     and
result['running_errors'] != []:
                return True
            if all([v == [] for v in result.values()]):
                return False

            return False

        @property
```

```
        def supported_semantic(self) -> list:
            return ['full_name', 'occupation', 'country', 'city',
'company']

    class TwitterProcessor(Processor):
        def process(self, semantic: str, data: str) -> dict |
None:
            result = execute_maryam('twitter', ['-q',
rf"{data}"])

            return self._finalize_process_result(semantic,
'twitter', data, result)

        def _is_empty(self, result: dict) -> bool:
            if result == {}:
                return True
            if 'running_errors' in result and
result['running_errors'] != []:
                return True
            if all([v == [] for v in result.values()]):
                return False

            return False

        @property
        def supported_semantic(self) -> list:
            return ['full_name', 'occupation', 'country', 'city',
'company']
```

Як видно з коду наразі програма має 7 різних процесорів, кожен відповідальний за певне джерело інформації і підтримує різні семантичні дані. Деякі з процесорів оброблюють одну й ту саму семантичну інформацію. Наприклад, ім’я чи назва компанії підтримується WikipediaProcessor та

TwitterProcessor, які відповідно відповідають за пошук інформації на просторах вікіпедії та соціальної мережі «Твіттер».

Можливості модуля можна розширити за допомогою збільшенню кількості процесорів. Додавши, наприклад, для пошуку інформації на мапах, аналіз координат, аналіз IP адрес, та ін.

3.5 Методика роботи користувача

Користувач починає роботу з головної сторінки вводу запиту, зображеної на рисунку 3.1.

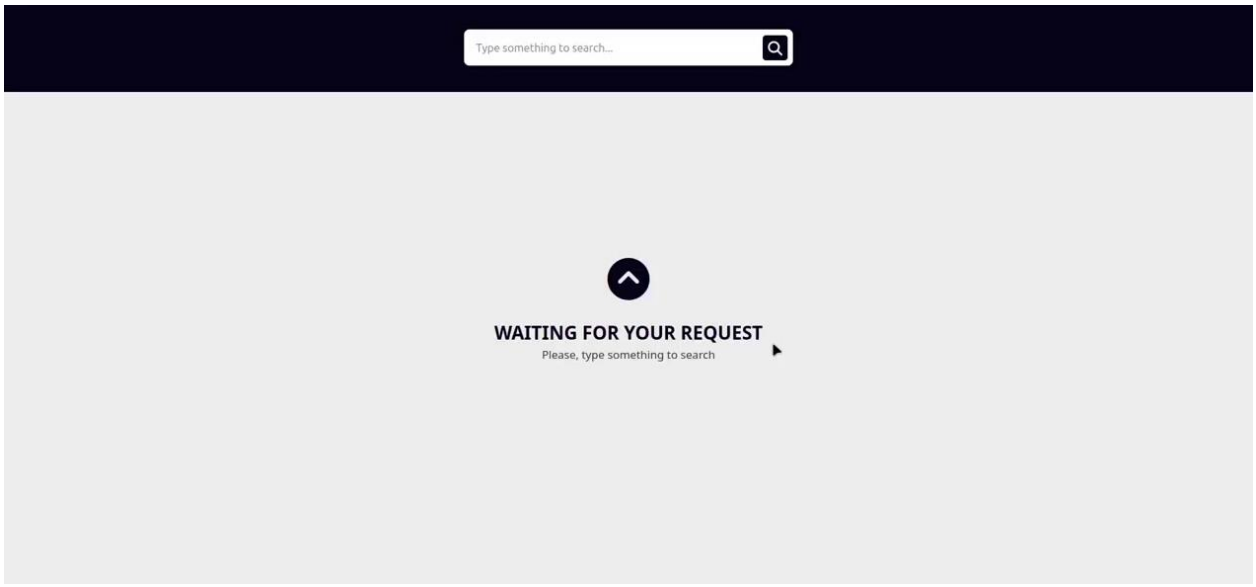

Рисунок 3.1 – Головна сторінка введення запиту користувача

Після введення запиту і його обробки користувачу відображаються результати роботи модуля обробки даних OSINT на сторінці, зображеній на рисунку 3.2. Дані згруповані за семантичним значенням та процесорами які використовувались для обробки певної частини оригінального запиту користувача, або його цілком.

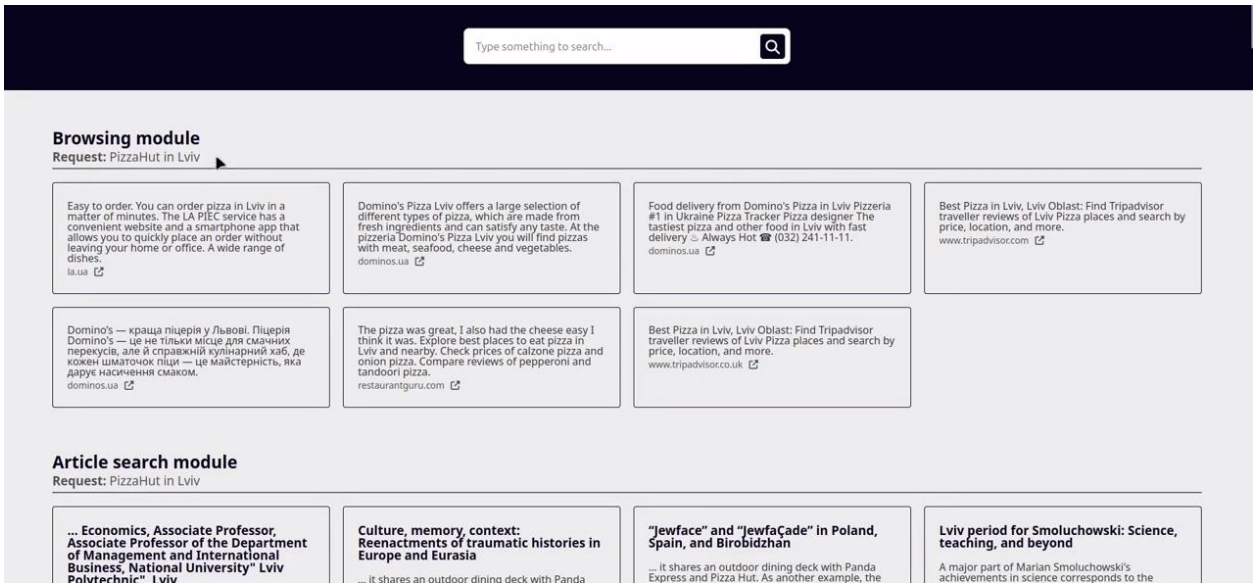

Рисунок 3.2 – Сторінка результатів роботи модуля

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було проведено дослідження та розробку модуля семантичного аналізу даних OSINT. Метою роботи було створення програмного рішення для ефективного аналізу, кластеризації та структурованого представлення інформації з відкритих джерел, що сприяє підвищенню якості прийняття рішень у контексті обробки великих обсягів даних.

У процесі виконання роботи було проаналізовано сучасні підходи до обробки текстових даних, включаючи методи векторизації, такі як TF-IDF, і алгоритми кластеризації, зокрема DBSCAN. Проведено дослідження існуючих систем обробки OSINT-даних, визначено їхні можливості, обмеження та специфіку роботи з різними типами джерел. На основі цього аналізу були сформульовані вимоги до розроблюваного модуля, що включали точність семантичного аналізу, швидкість обробки запитів та адаптивність до різних платформ.

Розроблений програмний модуль забезпечує автоматизовану класифікацію даних за змістом, пошук релевантної інформації та її представлення у зручному структурованому вигляді. Ключовими елементами системи є інтуїтивний інтерфейс для введення запитів, алгоритми виявлення семантичних зв’язків у тексті та механізми для обробки інформації з різних OSINT-джерел.

Подальший розвиток модуля може включати інтеграцію з додатковими джерелами OSINT-інформації, розширення функціональності кластеризації для специфічних галузей та адаптацію до нових форматів даних. Розроблене рішення є особливо корисним у сферах, де обробка відкритих даних має вирішальне значення, таких як інформаційна безпека, аналітика ризиків, розслідування або стратегічне планування.

ПЕРЕЛІК ПОСИЛАНЬ

1. Bazzell M. OSINT TECHNIQUES: RESOURCES FOR UNCOVERING ONLINE INFORMATION / Michael Bazzell., 2023. – 549 с.
2. Hannes H. Natural Language Processing in Action: Understanding, analyzing, and generating text with Python / H. Hannes, H. Cole, L. Hobson., 2019. – 544 p. – (Simon and Schuster).
3. Russell M. Mining the Social Web: Data Mining Facebook, Twitter, LinkedIn, Instagram, GitHub, and More / M. Russell, M. Klassen., 2018. – 432 p. – (O'Reilly Media, Inc).
4. Maltego [Електронний ресурс] – Режим доступу до ресурсу: https://www.maltego.com/.
5. Pulsar Platrofm [Електронний ресурс] – Режим доступу до ресурсу: https://www.pulsarplatform.com/.
6. RavenPack [Електронний ресурс] – Режим доступу до ресурсу: https://www.ravenpack.com/.
7. Brandwatch Analytics [Електронний ресурс] – Режим доступу до ресурсу: https://www.brandwatch.com/.
8. Lexalytics [Електронний ресурс] – Режим доступу до ресурсу: https://www.lexalytics.com/.
9. Google Schoolar [Електронний ресурс] – Режим доступу до ресурсу: https://scholar.google.com/.

Додаток А – Код програми

api.py

```
import json
import os
from copy import deepcopy
from typing import Annotated

from fastapi import FastAPI
from fastapi import Response
from fastapi import Form
from jinja2 import Environment, FileSystemLoader

from integration import get_semantics
from semantic_processor import DocsProcessor
from semantic_processor import BrowsingProcessor,
WikipediaProcessor, GitHubProcessor, get_processors

templateLoader = FileSystemLoader(searchpath=os.getcwd())
jinja_env = Environment(loader=templateLoader)

output_template =
jinja_env.get_template('/public/templates/output.html')

MEDIA_TYPE = 'application/json'
app = FastAPI()

@app.get("/")
def main():
    return Response(content=output_template.render())

@app.post("/")
def search(prompt: Annotated[str, Form()]):
    semantics_response = get_semantics(prompt)
    if 'semantics' not in semantics_response:
        return Response(
            content=output_template.render({'res': "Processing
Error"})
        )

    res = {}
    browsing_processor = BrowsingProcessor()
    docks_processing = DocsProcessor()
    wiki_processor = WikipediaProcessor()
    github_processor = GitHubProcessor()
    if 'original' in semantics_response and 'english' in
semantics_response and semantics_response['original'] ==
semantics_response['english']:
        res = _combine_res(res,
```

```
browsing_processor.process('original',
semantics_response['original']))
        res = _combine_res(res,
docks_processing.process('original',
semantics_response['original']))
        res = _combine_res(res,
wiki_processor.process('original',
semantics_response['original']))
        res = _combine_res(res,
github_processor.process('original',
semantics_response['original']))
    elif 'original' in semantics_response:
        res = _combine_res(res,
browsing_processor.process('original',
semantics_response['original']))
        res = _combine_res(res,
docks_processing.process('original',
semantics_response['original']))
        res = _combine_res(res,
wiki_processor.process('original',
semantics_response['original']))
        res = _combine_res(res,
github_processor.process('original',
semantics_response['original']))
    elif 'english' in semantics_response:
        res = _combine_res(res,
browsing_processor.process('original',
semantics_response['english']))
        res = _combine_res(res,
docks_processing.process('original',
semantics_response['english']))
        res = _combine_res(res,
wiki_processor.process('original',
semantics_response['english']))
        res = _combine_res(res,
github_processor.process('original',
semantics_response['english']))

    for semantic_dict in semantics_response['semantics']:
        semantic = semantic_dict['semantic']
        semantic_processors = get_processors(semantic)

        for processor in semantic_processors:
            if semantic_dict['original_value'] ==
semantic_dict['english_value']:
                res = _combine_res(res,
processor.process(semantic, semantic_dict['original_value']))
            else:
                res = _combine_res(res,
processor.process(semantic, semantic_dict['original_value']))
                res = _combine_res(res,
processor.process(semantic, semantic_dict['english_value']))
```

```
    with open(prompt.strip().lower() + '.json', 'w') as
res_file:
        res_file.write(json.dumps(res))

    return Response(
        content=output_template.render({'res': res})
    )


def _combine_res(res: dict, processor_data: dict|None) -> dict:
    if processor_data is None:
        return res

    processor_semantic = processor_data['semantic']
    processor_data_copy = deepcopy(processor_data)

    if processor_semantic not in res:
        res[processor_semantic] = [processor_data_copy]

        return res

    res[processor_semantic].append(processor_data_copy)

    return res
```

base.py

```
from pathlib import Path

PROJECT_PATH = Path.cwd()
```

osint_wrapper.py

```
import ast
import subprocess

from base import PROJECT_PATH

venv_path = PROJECT_PATH / '.venv'
if venv_path.exists():
    EXECUTABLE = str(venv_path / 'bin' / 'maryam')
else:
    EXECUTABLE = 'maryam'

def execute_maryam(module: str, options: list = None)->dict:
    opts = [EXECUTABLE, '-e', module, '--api', '--output']
    opts.extend(options)
```

```
    result = subprocess.run(opts, capture_output=True,
text=True)
    print(opts)
    print(result.args)
    print(result.stdout)
    print(type(result.stdout))
    try:
        result = ast.literal_eval(result.stdout)
    except Exception as e:
        print("Exception:", e)
        result = {}
    return result
```

semantic_processors.py

```
from abc import ABC, abstractmethod
from enum import Enum
from urllib.parse import urlparse

from osint_wrapper import execute_maryam
from utils import format_module_name

class Semantics(Enum):
    USERNAME = 'username'
    PHONE = 'phone'
    DOCS = 'docs'
    WIKI = 'wiki'
    BROWSING = 'browsing'

class Processor(ABC):
    @property
    @abstractmethod
    def supported_semantic(self) -> list:
        pass

    @abstractmethod
    def process(self, semantic: str, data: str) -> dict|None:
        pass

    @abstractmethod
    def _is_empty(self, result: dict) -> bool:
        pass

    def supports(self, semantic: str) -> bool:
        return semantic.lower() in self.supported_semantic

    def _finalize_process_result(self, semantic: str, module:
str, data: str, result: dict|list) -> dict|None:
        if self._is_empty(result):
            print('EMPTY', semantic, module , result)
            return None
```

```
        return {
            'semantic': semantic,
            'module': module,
            'formatted_module': format_module_name(module) + '
module',
            'input': data,
            'result': result
        }

class UsernameProcessor(Processor):
    def _is_empty(self, result: dict) -> bool:
        if result == {}:
            return True
        if 'running_errors' in result and
result['running_errors'] != []:
            return True
        return False

    @property
    def supported_semantic(self) -> list:
        return ['full_name', 'username']

    def process(self, semantic:str, data: str) -> dict:
        result = execute_maryam('username_search', ['-q',
rf"{data}"])

        return self._finalize_process_result(semantic,
'username_search', data, result)

class PhoneNumberProcessor(Processor):
    def _is_empty(self, result: dict) -> bool:
        if result == {}:
            return True
        if 'running_errors' in result and
result['running_errors'] != []:
            return True
        if 'success' in result and result['success'] is False:
            return True

        if 'valid' not in result and result['valid'] is False:
            return True

        return False

    @property
    def supported_semantic(self) -> list:
        return ['phone']

    def process(self, semantic:str, data: str) -> dict:
        result = execute_maryam('phone_number_search', ['-n',
rf"{data}"])
```

```
        return self._finalize_process_result(semantic,
'phone_number_search', data, result)

class DocsProcessor(Processor):
    def _is_empty(self, result: dict) -> bool:
        if type(result) is dict:
            if result == {}:
                return True
            if 'running_errors' in result and
result['running_errors'] != []:
                return True
            if 'links' not in result and 'data' not in result:
                return True

            if 'data' not in result or result['data'] == []:
                return True
        elif type(result) is list:
            if result == []:
                return True

        return False

    @property
    def supported_semantic(self) -> list:
        return ['docs']

    def process(self, semantic:str, data: str) -> dict:
        result = execute_maryam('article_search', ['-q',
rf"{data}"])

        for res_line in result['results']:
            res_line['title'] = res_line['t']
            res_line['link'] = res_line['a']
            res_line['authors'] = res_line['c']
            res_line['description'] = res_line['d']
            res_line['domain'] = urlparse(res_line['a']).netloc

            del res_line['t']
            del res_line['a']
            del res_line['c']
            del res_line['d']

        return self._finalize_process_result(semantic,
'article_search', data, result['results'])

class WikipediaProcessor(Processor):
    def _is_empty(self, result: dict) -> bool:
        if type(result) is dict:
            if result == {}:
                return True
            if 'running_errors' in result and
```

```
result['running_errors'] != []:
                return True
            if 'links' not in result and 'data' not in result:
                return True

            if 'data' not in result or result['data'] == []:
                return True
        elif type(result) is list:
            if result == []:
                return True

        return False

    @property
    def supported_semantic(self) -> list:
        return ['full_name', 'occupation', 'country', 'city',
'company']

    def process(self, semantic: str, data: str) -> dict:
        result = execute_maryam('wikipedia', ['-q', rf"{data}"])

        if 'titles' not in result or 'titles' not in result:
            return self._finalize_process_result(semantic,
'wikipedia', data, result)

        if result['titles'] is None or result['links'] is None:
            return self._finalize_process_result(semantic,
'wikipedia', data, result)

        formated_data = []
        for title, link in zip(result['titles'],
result['links']):
            title = title[:title.find('[')]
            formated_data.append({'title': title, 'link': link})

        return self._finalize_process_result(semantic,
'wikipedia', data, formated_data)

class BrowsingProcessor(Processor):
    def _is_empty(self, result: dict) -> bool:
        if type(result) is dict:
            if result == {}:
                return True

            if 'results' not in result or result['results'] ==
[]:
                return True
            if 'running_errors' in result and
result['running_errors'] != []:
                return True
            if all([v == [] for v in result.values()]):
                return False
```

```
        elif type(result) is list:
            if result == []:
                return True

        return False

    BROWSERS = [
        'duckduckgo',
        'bing',
        'google',
    ]


    @property
    def supported_semantic(self) -> list:
        return ['full_name', 'occupation', 'country', 'city',
'company']

    def process(self, semantic:str, data: str) -> dict:
        submodule_res = execute_maryam('iris', ['-q',
rf"{data}"])

        for res_line in submodule_res['results']:
            res_line['link'] = res_line['a']
            res_line['description'] = res_line['d']
            res_line['domain'] = urlparse(res_line['a']).netloc

            del res_line['t']
            del res_line['a']
            del res_line['c']
            del res_line['d']

        return self._finalize_process_result(semantic,
'browsing', data, submodule_res['results'])

class GitHubProcessor(Processor):
    def process(self, semantic: str, data: str) -> dict | None:
        result = execute_maryam('github', ['-q', rf"{data}"])

        return self._finalize_process_result(semantic, 'github',
data, result)

    def _is_empty(self, result: dict) -> bool:
        if result == {}:
            return True
        if 'running_errors' in result and
result['running_errors'] != []:
            return True
        if all([v == [] for v in result.values()]):
            return False

        return False
```

```
    @property
    def supported_semantic(self) -> list:
        return ['full_name', 'occupation', 'country', 'city',
'company']

class TwitterProcessor(Processor):
    def process(self, semantic: str, data: str) -> dict | None:
        result = execute_maryam('twitter', ['-q', rf"{data}"])

        return self._finalize_process_result(semantic,
'twitter', data, result)

    def _is_empty(self, result: dict) -> bool:
        if result == {}:
            return True
        if 'running_errors' in result and
result['running_errors'] != []:
            return True
        if all([v == [] for v in result.values()]):
            return False

        return False

    @property
    def supported_semantic(self) -> list:
        return ['full_name', 'occupation', 'country', 'city',
'company']

PROCESSORS = [
    UsernameProcessor(),
    PhoneNumberProcessor(),
    DocsProcessor(),
    WikipediaProcessor(),
    BrowsingProcessor(),
    GitHubProcessor(),
    TwitterProcessor(),
]

def get_processors(semantic: str) -> list:
    result = []
    for processor in PROCESSORS:
        if processor.supports(semantic):
            result.append(processor)

    return result
```

utils.py

```
def format_module_name(module: str)->str:
    module = module.replace('_', ' ')

    return module[0].upper() + module[1:]
```

integration.py

```
import json
import time
from dataclasses import dataclass

import requests
from requests.utils import default_headers

from config import INTEGRATION_BASE_URL, INTEGRATION_API_KEY,
INTEGRATION_HEADERS

BASE_URL = INTEGRATION_BASE_URL

def get_default_headers():
    headers = default_headers()
    headers['Authorization'] = f'Bearer {INTEGRATION_API_KEY}'

    for header, value in INTEGRATION_HEADERS.items():
        headers[header] = value

    return headers

@dataclass
class Response:
    status: int
    rawBody: str
    body: dict

def build_url(module: str) -> str:
    return BASE_URL  + '/' + module

def send_request(method: str, url: str, params: dict = None,
data: dict = None) ->Response :
    request = requests.Request(
        method,
        url,
        get_default_headers(),
        params=params,
        json=data
    ).prepare()
```

```
    print(request.url)

    with requests.Session() as session:
        response = session.send(request)

    return Response(response.status_code, response.content,
response.json())

def get_semantics(prompt: str) -> dict:
    send_request(
        'POST',
        build_url('messages'),
        data={'role': 'user', 'content': prompt}
    )
    runs_response = send_request(
        'POST',
        build_url('runs'),
        data={'assistant_id': 'asst_Oewt7FqP4Sgu8EzraZMhu73d',
'additional_instructions': None, 'tool_choice': None}
    )
    run_id = runs_response.body['id']

    while True:
        result_response = send_request(
            'GET',
            build_url('messages'),
            params={'limit': 1, 'order': 'desc', 'run_id':
run_id}
        )
        print(result_response.body['data'])

        if result_response.body['data'] != [] and
result_response.body['data'][0]['content'] != []:
            break

        time.sleep(1)

    return
json.loads(result_response.body['data'][0]['content'][0]['text']
['value'])
```