Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мультимедійних систем факультету інформатики

Засоби автоматизованої побудова онтології на основі великих даних

Текстова частина до курсової роботи
за спеціальністю „Інженерія програмного забезпечення” 121

Керівник курсової роботи
к.т.н., доц. _____
(прізвище та ініціали)
_____
(підпис)
“____” _____ 2021 р.

Виконав студент
_____
(прізвище та ініціали)
“____” _____ 2021 р.

Київ 2021

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мультимедійних систем факультету інформатики

ЗАТВЕРДЖУЮ
Зав.кафедри мультимедійних систем,
Доц., кандидат. ф.-м. н. _____О. П. Жежерун
(підпис)
„____”_____2021 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

студенту факультету інформатики МП “Інженерія програмного забезпечення” 1 року навчання
Рєпкіну Максиму Сергійовичу
ТЕМА Автоматизована побудова онтології на основі великих даних

Зміст ТЧ до курсової роботи:

1. Індивідуальне завдання
2. Календарний план
3. Анотація
4. Вступ
5. Загальний огляд системи
6. Збір даних
7. Обробка даних
8. Побудова онтології
9. Висновки
10. Список використаної літератури
11. Додатки

Дата видачі „___” _____ 2021 р. Керівник _____
(підпис)
Завдання отримав _____

(підпис)

Календарний план виконання роботи:

№ п/п	Назва етапу курсового проекту	Термін виконання етапу
1.	Отримання завдання на курсову роботу	04.10.2020
2.	Ознайомлення з існуючими роботами на цю тему	01.12.2020
3.	Побудова системи отримання та обробки даних	01.01.2021
4.	Побудова онтології	15.03.2021
5.	Написання теоретичної частини	30.03.2021

Зміст

Анотація

У роботі розглянуто особливості автоматизованої побудови онтології на основі великих даних. Описані основні методи обробки великих даних, методи обробки тексту. Представлено систему, яка обробляє великий об’єм неструктурованих даних з української частини Вікіпедії.

ВСТУП

Світ зараз стрімко розвивається. Разом із цим розвитком з’являються нові практичні задачі, багато з яких вже було описано теоретично вченими минулого. Серед таких галузей можна виокремити штучний інтелект. Елементи цієї галузі математики зараз використовуються в багатьох сучасних проектах. Проте, не так багато уваги приділяється новій проблемі людства - збору та перетворенню даних, що є запорукою ефективного використання штучного інтелекту.

У цій роботі розглянуто задачу збору великої кількості неструктурованих даних, їх перетворення та консолідація за допомогою онтології. Щодня у мережі Інтернет з’являються нові знання з різних галузей. Людині вже не під силу дивитись і, тим паче, аналізувати відношення нових знань до вже існуючих. Через це було прийнято рішення побудувати автоматизовану систему, де виконуються всі зазначені кроки, а людині залишається лише аналізувати вже інтегровані дані до онтології.

Робота складається з п’яти розділів. Перший дає визначення великим даним та описує проблеми, пов’язані з ними. Другий дає визначення найбільш поширеним методам аналізу тексту. В третьому описується архітектура нейронних мереж LSTM, яка використовувалась для аналізу тексту в цій системі. Четвертий дає визначення онтології та описує їх представлення на низькому рівні. П’ятий складається з детального опису побудованої системи, її архітектури та можливостей розширення.

РОЗДІЛ 1: Великі дані

1.1 Поняття великих даних

Зараз великі дані прийнято характеризувати літерами “V”. Вперше таку класифікацію великим даним дав Дуг Лейні, аналітик компанії “Meta Group Inc.” в 2001 році:

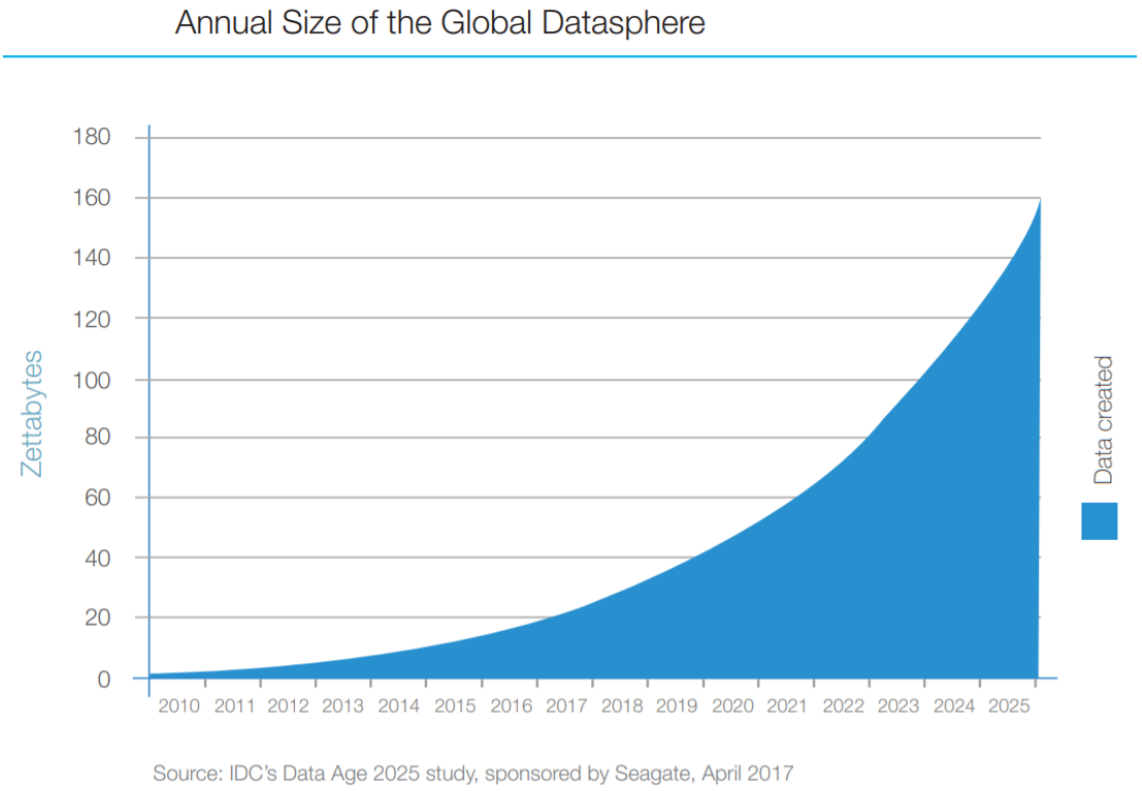
- Volume (об’єм) - перша літера “V” описує великий об’єм даних. Вчені спорять з приводу межі, з якої починається великий об’єм. Більшість зійшлись на тому, що це така кількість даних, яку не може зберегти один комп’ютер.
- Velocity (швидкість) - друга літера описує швидкість постачання нових даних до системи. Межа - можливості обробити дані для одного комп’ютеру.
- Variety (різноманітність) - остання основна літера описує різноманітну кількість структур, в яких ці дані поступають до системи. [1]

Характеристики великих даних постійно оновлюються, тому що галузь є дуже молодою та наразі стрімко розвивається на практиці. Таким чином, окрім основних, існують також додаткові характеристики, які відрізняються від компанії до компанії:

- Veracity (достовірність) - характеристика даних, що описує достовірність, була впроваджена IBM. [2]
- Value (цінність) та Viability (життєздатність) - також важливі характеристики, що описують цінність даних, в першу чергу для бізнесу. [3]

1.2 Історія появи

У 1960-х роках вчені почали створювати перші моделі сховищ для великих даних. Відтоді змінилося все, від об’ємів до методів обробки та зберігання. Як ми бачимо на малюнку 1.1., збільшення кількості даних нагадує експоненційний. У 2010 році в світі було 0.1 зетабайт даних, а зараз ми маємо близько 60. Проте, вже у 2000-х з’явився термін “великі дані”, але був популярним виключно в академічному середовищі.



Мал. 1.1 Кількість даних в сучасному інтернеті [4]

Широке введення терміну “Великі дані” пов’язують із Кліфордом Лінчем, який у 2008 році підготував статтю “Як можуть вплинути на майбутнє науки технології, які відкривають можливості працювати з великими обсягами даних?” [5] Проте, в ті роки, цей термін вважався суто теоретичним і був популярним лише в академічних колах. Тільки у 2014 році Gartner перестав випускати окремий цикл зрілості великих даних,

мотивуючи тим, що ця технологія перейшла з етапу обговорення до практичного застосування.

1.3 Основні методи їх зберігання та обробки

Великі дані, за визначенням, не можуть зберігатись та оброблятись потужностями одного комп'ютера, через те, що не буде вистачати обчислювальної потужності. Існує два види підвищення потужності системи: горизонтальне та вертикальне масштабування. Основна ідея вертикального масштабування полягає в тому, що збільшення навантаження на систему потребує більш потужних комп’ютерів (з потужнішим процесором, оперативною пам’яттю та іншим). Основний мінус такого підходу - великі організації будуть потребувати супер-комп’ютери. А що робити, якщо він зламається?

Через це зазвичай використовується горизонтальне масштабування. Основна ідея полягає в додаванні нової машини до системи і змушення її виконувати частину роботи. Сукупність таких машин будує кластер. Кластер - це різновид паралельної чи розподіленої системи, що має наступні характеристики:

- Складається з декількох пов’язаних між собою комп’ютерів.
- Використовується як єдиний, уніфікований комп’ютерний ресурс.[6]

Таким чином не обов’язково покупати супер-комп’ютер, щоб встигати обробляти великі дані, достатньо додавати нові машини до кластеру.

1.3.1 Hadoop

Основна технологія, яка використовується для зберігання даних - Apache Hadoop. Він передбачався як система, в якій рекомендовано використовувати кластер з серійних, дешевих комп'ютерів, які з відносно великою ймовірністю можуть зламатись. Такий підхід дозволяє сильно зменшити витрати на професіональне залізо. Дані при відмові машин не губляться через вбудований механізм реплікації (копія зберігається на декількох фізичних серверах, за замовчуванням кількість копій - 3). Також, код цієї системи є у відкритому доступі, що ще більше зумовило популярність. Hadoop складається з трьох основних компонентів:

- HDFS (Hadoop Distributed File System) - є головним елементом зберігання даних.
- MapReduce - алгоритм, який є основою обробки та отримання даних з системи.
- Yarn - менеджер MapReduce задач.

Hadoop - це ідеальна система для зберігання великих, слабо структурованих файлів. Наступні компанії використовують його для збереження петабайтів даних: Facebook, eBay, LinkedIn та інші.

РОЗДІЛ 2: Алгоритми обробки та аналізу тексту

2.1 Стеммінг

Стемінг - це відомий базовий алгоритм попередньої обробки тексту. Він використовується в інформаційному пошуку практично завжди. Принцип роботи полягає в тому, що для кожного слова витягується його основа, яка може не бути однакова з корнем слова, і в такому вигляді зберігається у сховище. Це дає можливість зменшити розмір даних для зберігання, тому що практично завжди основа слова є коротшою. Ще один плюс - це можливість здійснювати пошук без прив'язки до роду чи

відмінка слова. Стеммінг вважається “молодшим братом” більш потужного алгоритму під назвою “лематизація”. Проте, він не вимагає готового словнику основ слів, як лематизація, що дає можливість використовувати його в пошуковому двигуні Google. На відміну від лематизації, стеммінг потребує менших обчислювальних витрат, але і дає гірші результати, тому існує потреба балансувати між цими алгоритмами.

2.2 POS-tagging

POS - це алгоритм, який видає кожному слову його приналежність до частин мови. В роботи було використано нейронну мережу “Bidirectional LSTM”[7], яка аналізує текст та видає наступні теги:

- ADJ (adjective)
- ADP (adposition)
- ADV (adverb)
- AUX (auxiliary)
- CCONJ (coordinating conjunction)
- DET (determiner)
- INTJ (interjection)
- NOUN (noun)
- NUM (numeral)
- PART (particle)
- PRON (pronoun)
- PROPN (proper noun)
- PUNCT (punctuation)
- SCONJ (subordinating conjunction)
- SYM (symbol)
- VERB (verb)
- X (other)

Приклад аналізованого тексту:

```
Орден/NOUN Досконалості/NOUN –/PUNCT державна/ADJ нагорода/NOUN Грузії,/PROPN
заснована/ADJ рішенням/NOUN Парламенту/NOUN Грузії/PROPN
№/NOUN 1553/PROPN від/ADP 31/ADJ липня/NOUN 2009/ADJ
для/ADP нагородження/NOUN видатних/ADJ діячів/NOUN культури,/PROPN
освіти,/PROPN науки,/PROPN мистецтва,/ADJ
спорту/NOUN та/CCONJ інших/DET сфер/NOUN за/ADP видатні/ADJ заслуги./NOUN
```

Мал. 2.1 Приклад pos-tagged тексту

РОЗДІЛ 3: Нейронна мережа LSTM

3.1 Визначення нейронних мереж

Нейронна мережа - це математична модель, а також її програмне чи апаратне втілення, побудована по принципу організації та функціонування біологічних нейронних мереж - мереж нервових клітин живого організму. Це поняття виникло при вивченні процесів, що відбуваються у мозку, та при спробі змоделювати ці процеси. Першою спробою були нейронні мережі вчених Маккалока і Піттса [8]. Нейронні мережі не програмуються, вони вчаться. Можливість вчитися - одна з найголовніших переваг нейронних мереж над традиційними алгоритмами. Технічно навчання зводиться до знаходження коефіцієнтів зв’язків між нейронами. У процесі навчання нейронна мережа здатна виявляти складні залежності між вхідними та вихідними даними, а також робити узагальнення. Це означає, що правильно навчена мережа здатна повернути правильну відповідь, яка не була присутня у виборці для навчання. [9]

Дані для навчання моделі мають обиратись дуже ретельно, тому що від цього сильно залежить точність майбутньої моделі. Ці дані мають відповідати наступним критеріям:

- Репрезентативність, тобто дані повинні висвітлювати повну картину в предметній області.

- Несуперечливість, тобто дані не мають взаємовиключати один одного.

Після обрання навчальної вибірки зазвичай дані готують до навчання. Підготовка скзадається з декількох етапів:

- Дослідження аутлаєрів - знаходження даних, які сильно відрізняються від вибірки за деякими параметрами. Є декілька рішень, які можуть бути прийняті з приводу цих даних:
  - Перевірити кількість даних, що є аутлаєрами. Якщо вона занадто велика, то слід дослідити ці дані більш детально.
  - Дослідити дані на помилковість. Якщо вони містять помилку і це можна виправити, то слід виправляти.
  - Якщо нічого зробити не можна, то слід видалити ці дані.
- Видалення пустих даних.
- Нормування даних - якщо дані лежать в різних діапазонах. Наприклад, якщо ми аналізуємо дані про будинки, де ціна будинку у виборці лежить в проміжку [10000; 1000000], а площа лежить у проміжку [30; 1000], то ці дані необхідно привести до одного проміжку. Зазвичай цей проміжок - [0;1]. Це робиться, щоб всі дані були в рівних умовах при навчанні моделі. Інакше, в цьому випадку, ціна б повністю перекривала площу.

Нейронні мережі класифікуються по принципу навчання. Існують наступні методи навчання:

- Навчання з вчителем - відповіді, яка повинна надати мережа відомі заздалегідь при навчанні. Основні задачі, де цей підхід використовується - передбачення подій на основі минулого, класифікація та інші.

- Навчання без вчителя - мережа сама формує результати на основі вхідних даних. Основні задачі використання - кластеризація. Кластеризація - це класифікація даних, де нічого про класи невідомо заздалегідь. Модель сама формує кластери з навчальної вибірки.
- Навчання з підкріпленням - найскладніший для реалізації підхід. Це система штрафів за неправильні дії та заохочення за правильні. Основне задача використання - це автопілоти, навігація з уникненням заторів та інше.

3.2 Рекурентні нейронні мережі

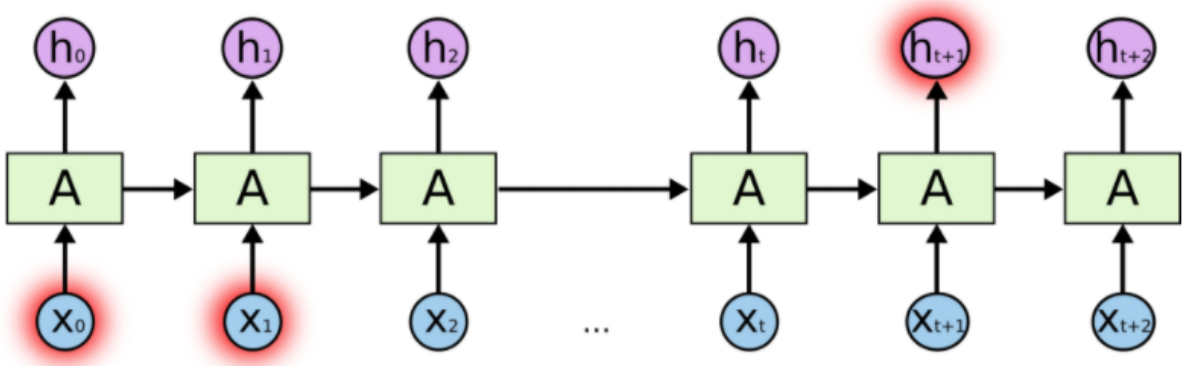
Рекурентні нейронні мережі - це клас нейронних мереж, де лінки між нейронами формують направлений граф у часі. Це дозволяє добитись динамічної поведінки. Цей клас нейронних мереж використовує внутрішню пам’ять, щоб обробляти довгі вхідні дані. Це дозволяє використовувати їх у мовному аналізі. [10] Ці мережі діляться на два наступні класи:

- Фільтр з кінцевою імпульсною характеристикою - представлено у виді направленого ациклічного графу.
- Фільтр з безкінечною імпульсною характеристикою - представлено у виді направленого графу.

Обидва види мають додаткові стани, що зберігаються у мережі. Сховище станів може бути замінено на іншу нейронну мережу або граф.[11]

Одна з ідей застосування рекурентної нейронної мережі - те, що вони можуть зв’язувати попередню інформацію з поточною задачею, щоб минуле допомогало нам в розумінні теперішнього. Проте, якщо між

теперішнім та пояснюючим моментом в минулому пройшло достатньо часу, тоді рекурентні мережі втрачають можливість зв’язувати дані.

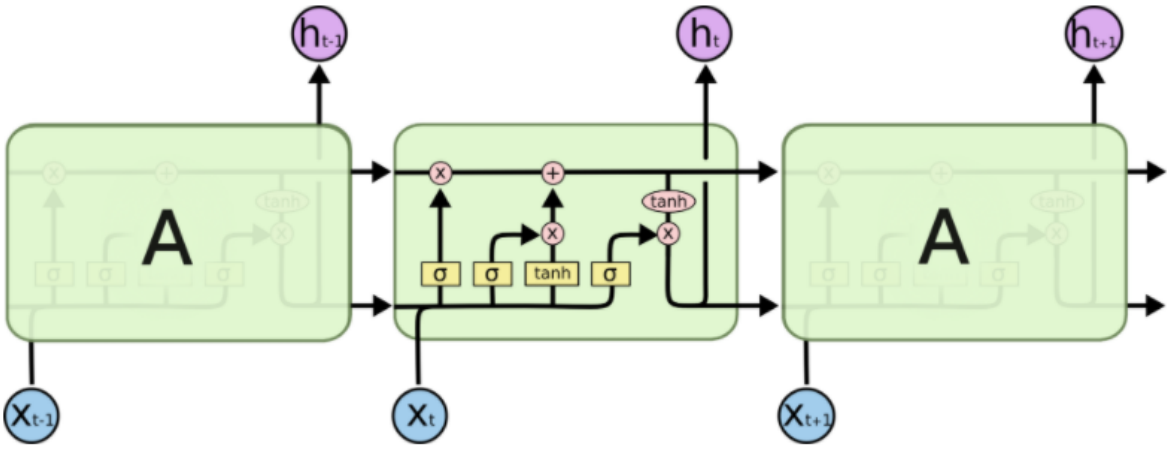


Мал.3.1 Рекурентна нейронна мережа[12]

Через це на допомогу приходить мережа LSTM.

3.3 Огляд нейронної мережі LSTM

LSTM (Long short-term memory) - архітектура рекурентних нейронних мереж, яка створена для запам’ятовування інформації на тривалий час. Вона була вперше представлена Зепом Хохрайтером та Юргеном Шмідхубером у 1997 році.



Мал. 3.2 LSTM нейронна мережа[12]

Покроковий розбір роботи мережі:

- Визначити які дані можна викинути. Це вирішує сигмоїдальна функція.
- Вирішити нову інформацію, яка буде зберігатись. Це вирішує функція tanh.
- Потім здійснюється заміна старих і нових даних для збереження в комірці.
- Останній крок - визначення інформації для виводу. Це вирішує сигмоїдальна та tanh функції.

РОЗДІЛ 4: Онтології

4.1 Визначення онтології

Поняття онтології народилося в античні часи та йде з філософії. Воно описує “теорію існування природи”. [13] Аристотель навіть запропонував свою онтологію з примітивними категоріями, такими як сутність та якість. В комп’ютерній сфері онтологія означає річ, яка була зроблена для моделювання знань з обраної предметної області. Термін було впроваджено дослідниками штучного інтелекту, які визнали працездатність математичної логіки. У 1980-х спільнота прийшла до висновку, що онтологія відноситься як до теорії змодельованого світу, так і виступає одним з модулів систем знань.[14]

Онтологія схожа на графову базу даних. Деякі вчені навіть відносять цю технологію до розділу комп’ютерних наук про організацію баз даних.

	Графова база	Онтологія
Типи даних	NodeType	Class
Сутності	Node	Individual

Типи зв’язків	Pointed	Functional, inverse functional, symmetric, asymmetric, reflexive, irreflexive
Data generation	No	Yes

Порівняння графової бази з онтологією

Як ми бачимо, онтологія дозволяє описувати більш складні структури предметних областей. Також, за допомогою класів в онтології будується ієрархія, чого не можна зробити в графовій базі. Зв’язки в онтології також більш складні. Це зроблено для більш точного опису зв’язку сутностей, що приведе до більш точної генерації нових даних на основі математичної логіки. Різонер - це алгоритм, який генерує нові дані на основі існуючих сутностей та їх зв’язків. Існує досить багато імплементацій різонерів, кожен відрізняється за швидкістю та призначенням. Ось короткий список наявних різонерів в редакторі онтологій Protege:

- FaCT
- ELK
- HermiT
- Mastro DL-Lite Reasoner
- Ontop
- Pellet
- Jcel

В роботи було використано “FaCT” через наступні характеристики:

- Він є різонером загального застосування.

- Написаний на C++, що робить його швидшим за конкурентів, тому що дозволяє не витрачати час на зайві маніпуляції пам’яттю. В описаній системі швидкість обробки є критичною, через великі дані.
- Він є найбільш популярним, що робить пошук проблем з ним легшим.

4.2 Триплети

Використаний різонер працює нативно з форматом онтологій “RDF”. RDF(Resource Description Framework) - модель опису зв’язаних даних, яка дозволяє технології Семантичного веба інтерпретувати інформацію, що наявна у вебі. Основу моделі складає: Суб’єкт - Предикат (або властивість) - Об’єкт (значення властивості). Триплет можна співставити з простим реченням типу: Підмет - Присудок - Доповнення. При цьому кожен об’єкт може бути суб’єктом в іншому відношенні, що дозволяє зберігати досить складні ієрархії. Для того, щоб машина розуміла всі вирази правильно, необхідно мати систему унікальних ідентифікаторів, щоб позначати кожного учасника триплетів. В RDF використовується URI(Uniform Resource Identifier), що складається з частини URI та опціональном ідентифікатором в кінці. Для запису URI прийнято використовувати http протокол, але він у вебі нікуди не веде, а використовується виключно для внутрішнього використовування.

```
<http://www.rusmarc.ru/index.html> <http://purl.org/dc/elements/1.1/creator>
            <http://www.nlr.ru/nlr/contacts.htm#npp>
<http://www.rusmarc.ru/index.html> <http://www.example.org/terms/creation-
            date > "20 ноября 2001"
<http://www.rusmarc.ru/index.html> <http://purl.org/dc/elements/1.1/language>
            "RU"
```

Мал 4.1 Приклад URI

RDF не надає механізму для опису класів сутностей та їх відносин. Таку задачу виконує словник. Словник - це набір URI, в якому зберігаються всі терміни. Якщо вони пов’язані, то, для більшої наглядності, використовують однакові префікси. Таким чином можна швидко знайти пов’язані терміни. Проте, це не є вимогою, а залишається на розсуд оператора словника.

Для розміщення RDF у вебі його необхідно серіалізувати. Існують наступні формати серіалізації:

- RDF/XML - репрезентація RDF у вигляді XML. Використовується у цій роботи. Є стандартом W3C.
- RDFa - у виді HTML. Є стандартом W3C.
- N3 (Notation 3) - позбавлений надлишкового синтаксису, у порівнянні з XML.

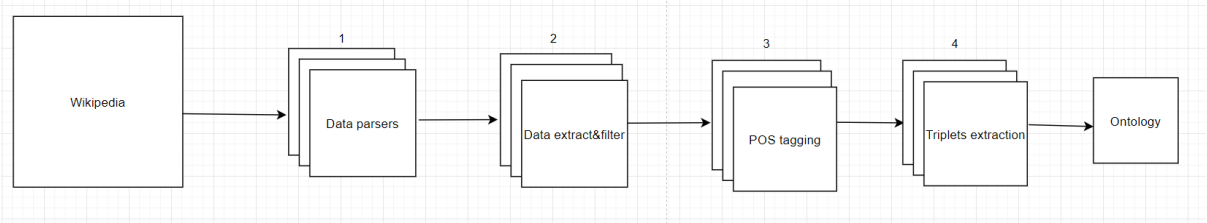
```
1.<?xml version="1.0"?>
2.<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3.    xmlns:exterms="http://www.example.org/terms/"
4.    xmlns:nlr="http://www.nlr.ru/nlr/contacts.htm#"
5.    xmlns:dc="http://purl.org/dc/elements/1.1/">

6.    <rdf:Description rdf:about="http://www.rusmarc.ru/index.html">
7.      <dc:language>RU</dc:language>
8.      <exterms:creation-date>20 ноября 2001 г.</exterms:creation-
      date>
9.      <dc:creator
      rdf:resource="http://www.nlr.ru/nlr/contacts.htm#npp"/>
10.   </rdf:Description>

11.   </rdf:RDF>
```

Мал. 4.2 Приклад RDF/XML формату[20]

Розділ 5: Розробка та аналіз системи автоматизованої побудови онтології на великих даних

5.1 Огляд архітектури системи

В якості практичної частини було побудовано систему з автоматизованого отримання, очищення даних та збереження їх до онтології.



Мал. 5.1 Архітектура побудованої системи

Як показано на малюнку 5.1, система має чотири кроки, які включають в себе отримання, обробку та збереження неструктурованої інформації до онтології. Отже, далі представлено стислий опис дій, що відбуваються на кожному кроці:

1. Здійснення http запиту до Wikipedia та отримання html сторінки для подальшого аналізу.
2. Витягування тексту з html, його фільтрація.
3. Тегування тексту за частинами мови.
4. Розбиття на речення, побудова синтаксичного дерева, виокремлення триплетів з нього та запис до онтології.

Всі елементи пайплайну створені в декількох екземплярах для забезпечення більшої швидкості обробки інформації та відмовостійкості.
У випадку цієї роботи використовується один потужний комп’ютер для всіх цих елементів. У разі виведення в реальні умови необхідно розділити зберігання та обробку даних по окремим комп’ютерам, задля забезпечення безпеки та коректній роботі при відмові серверів, мережі або при будь яких інших ситуаціях.

5.2 Огляд використаних технологій

5.2.1 Python

Python - це інтерпретована мова програмування загального застосування, яка була запропонована в у 1991 році датським програмістом Гуідо ван Руссумом в Нідерландах[15]. Мова підтримує парадигми:

- Об’єктно орієнтованого програмування. У мові наявні класи. Є можливість наслідувати класи, перевизначати їх поведінку, скривати повний інтерфейс від користувача, використовуючи ідентифікатори доступу.
- Структурного програмування. Всі структури контролю виконання наявні в мові (if then, else, do until, рекурсія).
- Функціонального програмування. В мові наявні наступні операції з функціональної парадигми: filter, map, reduce, вирази над списками, сети, генератори та інші.
- Аспектно-орієнтоване програмування. В мові є можливість втілити метапрограмування.[16]

Python має динамічну типізацію, що дозволяє не писати типи змінних при їх визначенні. Це дозволяє програмувати не під конкретні типи, а під інтерфейси, що робить код програми більш гнучким. Проте, у великих проектах гарним тоном є анотація типів. Це не впливає на виконання програми, наприклад, якщо функція має анотований вхідний параметр “int”, а ми передамо “string”, то помилки не станеться. Таке анотування - лише підказка програмістам про інтерфейс змінних. Також, динамічна типізація зумовила константність більшості типів даних в мові.

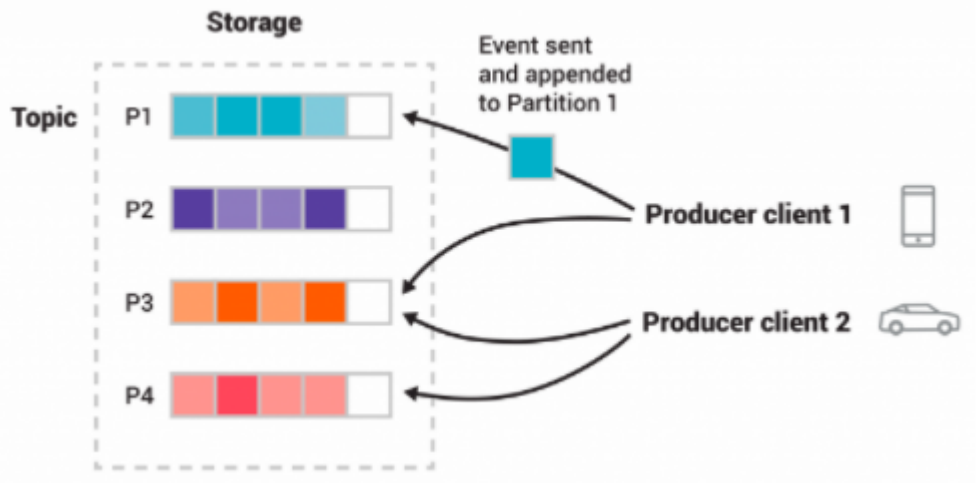
Отже, основною мовою програмування в цій системі є Python. Таке рішення було прийнято через наступні причини:

- Широкий вибір бібліотек, які необхідно використати тут.
- Виразність мови, що дозволяє швидше писати код, випробовуючи нові підходи.
- Можливість машинного навчання

5.2.2 Apache Kafka

Apache Kafka - розподілена система передачі повідомлень, яка написана на мовах Scala та Java. Розроблений компанією LinkedIn для внутрішнього використання. Згодом, у 2011 році код був переданий фонду Apache, який відтоді займався підтримкою системи.[17]

Один з ключових термінів в цій технології - це черга повідомлень, або топік. Топік складається з партицій, що дозволяє зберігати дані з однієї черги на різних фізичних комп'ютерах. Це дозволяє балансувати навантаження на машини, тим самим збільшувати загальну пропускну спроможність системи.



Мал. 5.2 Архітектура топіку в Kafka[18]

Kafka складається з серверів та клієнтів. Сервер - одна з машин кластеру Kafka. Клієнт - програма, яка має змогу читати, писати та

обробляти повідомлення в черзі. Клієнт та сервер комунікують через оптимізований протокол, що базується на TCP.

Для передачі даних між сервісами було використано технологію Apache Kafka. Причиною для такого вибору було декілька факторів:

- Відкритий програмний код, що передбачає безкоштовність.
- Практична відсутність конкуренції. Єдиний відомий конкурент - RabbitMq, проте він не масштабується як Kafka і не зберігає історію повідомлень, тому може втрачати дані.
- Можливість мати декілька слухачів на одну чергу повідомлень, що спрощує реалізацію.

5.2.3 Pipeline архітектура

Термін конвеєру прийшов з промисловості, де використовується такий спосіб обробки сировини для отримання кінцевого продукту. Він полягає в тому, що у нас є декілька етапів обробки сировини. Кожний етап передає результат своєї роботи на початок наступного, поки не завершаться всі етапи і ми не отримуємо вихідний продукт. Такий саме підхід використовується у функціональному програмуванні.

Вчені-математики підхопили такий підхід і стали використовувати її у процесорних обчисленнях. Найпростіша форма виконання інструкцій процесора була реалізована в машині “Z3” Конрада Цузе у 1941 році.[21] Такі маніпуляції допомагають процесору працювати швидше, але не всі задачі гарно вирішуються конвеєрним підходом. Наприклад, задачі, у яких час виконання менший за додану кількість часу на перехід з одного етапу на інший, не мають сенсу вирішуватись таким підходом.

В основі цієї системи знаходиться розподілений конвеєр, який забезпечує відмовостійкість, легке горизонтальне розширення необхідним частинам системи та можливість змінювати внутрішні частини пайплайну,

узгоджуючи лише інтерфейс передачі даних. Час, за який дані доставлено з одного етапу до іншого, значно менший за час виконання етапу. Через це, така архітектура є доречною в даному випадку.

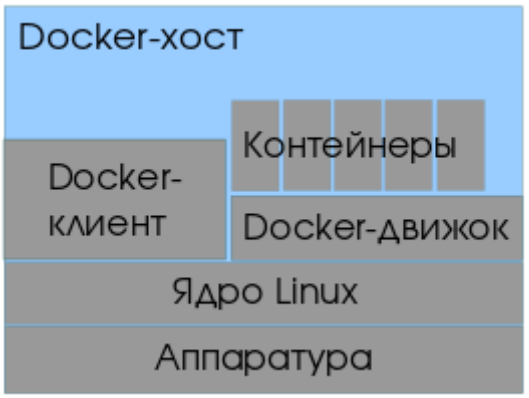
5.2.4 Docker

Docker - програмний продукт, що дозволяє розвертати доданки з підтримкою контейнеризації.

Контейнеризація - це один з методів віртуалізації, де ядро операційної системи підтримує декілька екземплярів просторів для користувача, замість одного. На відміну від апаратної віртуалізації, в контейнері може бути запущено екземпляр операційної системи з однаковим ядром з хостом.[22]

Docker було створено 13 березня 2013 року на мові програмування Golang. Зараз цей програмний продукт знаходиться у стадії активної розробки та належить до продуктів Apache, тобто є відкритим для користування.[23]

Базовою концепцією системи є образ. Образ - це ізольований, скомпільований вихідний код з усіма налаштуваннями, який готовий до запуску. На малюнку 5.3 можна побачити місце контейнерів в архітектурі Docker. Контейнер - це запущений образ. Контейнерів можна запускати необмежену кількість, що означає горизонтальне масштабування.

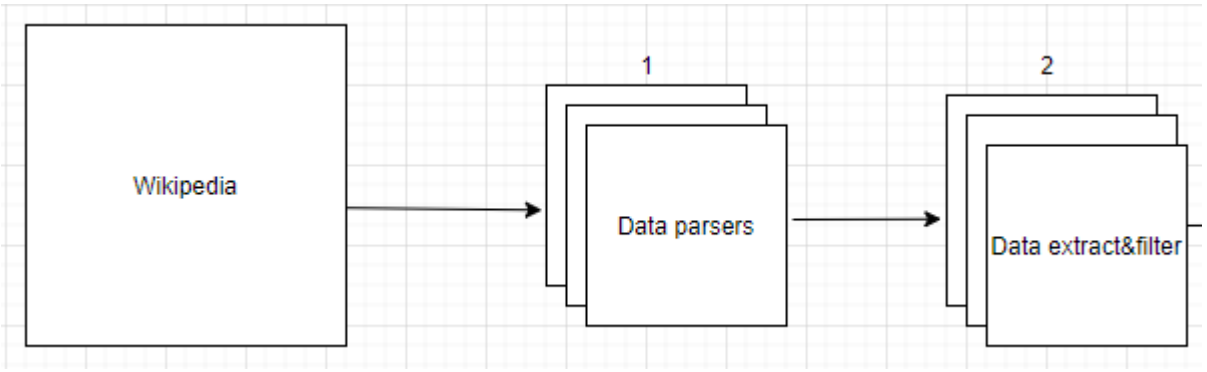


Мал. 5.3 Архітектура Docker

Основна перевага використання цього програмного продукту - можливість побудувати ізольований, працюючий доданок та запускати його на будь-яких комп’ютерах, без прив’язки до операційної системи чи налаштувань цієї машини. Всі необхідні дані про це зберігаються в середині образів, що роблять їх незалежними від зовнішніх параметрів комп’ютера. В розробленій системі такі переваги є критичними, адже через великі дані є необхідність у горизонтальному масштабуванні. Всі комп’ютері у кластері не можуть бути однаковими, а також налаштування кожної машини забирає багато часу, тому було використано такую технологію.

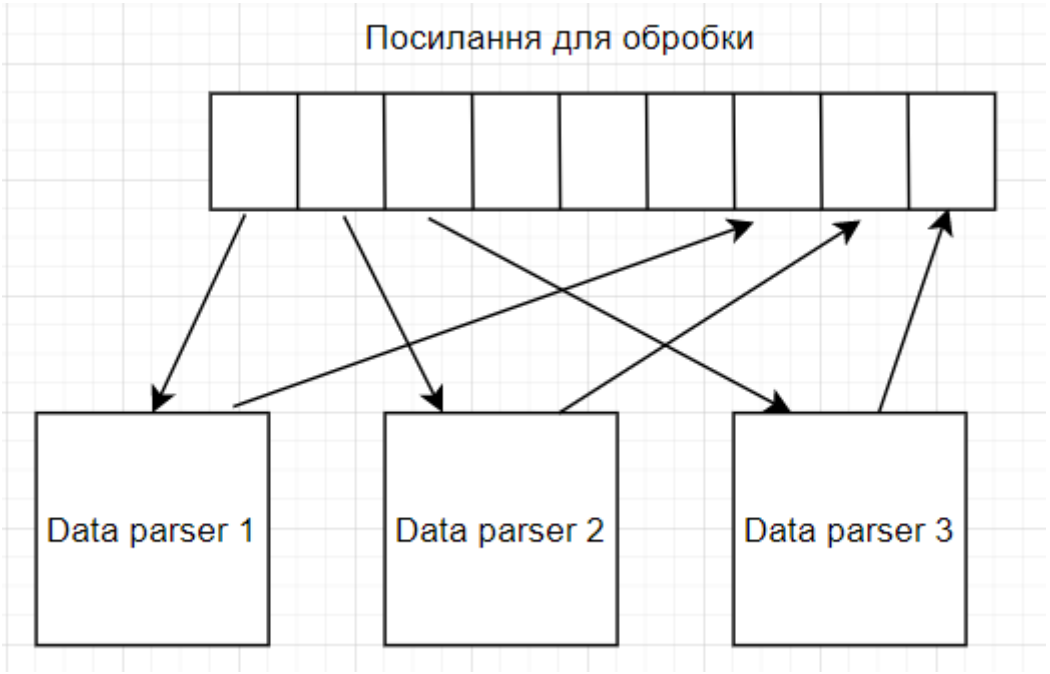
5.2 Детальний огляд частини отримання даних

До частини пайплайну з отримання даних належать перші два кроки:



Мал. 5.4 Етап отримання даних

На першому етапі робиться http запит до Wikipedia. У відповідь програма отримує html сторінку обраної сторінки. За допомогою бібліотеки BeautifulSoup витягуються всі посилання на цій сторінці та додаються до черги посилань, які треба обробити, зроблена дедуплікація.



Мал. 4.5 Отримання даних з Вікіпедії

Після цього весь html пишеться в Kafka топік. Та береться наступне посилання для обробки. Приклад отриманого html:

```
<p><b>Орден Досконалості</b> — державна нагорода
<a href="/wiki/%D0%93%D1%80%D1%83%D0%B7%D1%96%D1%8F" title="Грузія">Грузії</a>,
заснована рішенням Парламенту Грузії № 1553 від
<a href="/wiki/31_%D0%BB%D0%B8%D0%BF%D0%BD%D1%8F" title="31 липня">31 липня</a>
<a href="/wiki/2009" title="2009">2009</a> для нагородження
видатних діячів культури, освіти, науки, мистецтва, спорту та інших сфер за видатні заслуги.
</p>
```

Мал. 4.6 Приклад html[19]

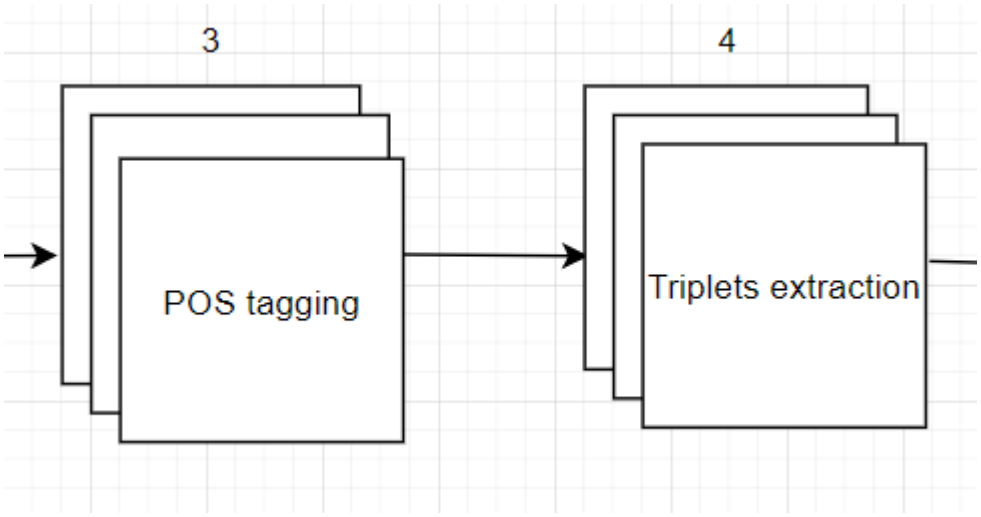
На другому етапі сервіс читає повідомлення з Kafka та за допомогою бібліотеки Beautifulsoup витягується інформація з html тегів “ ... ”, більшість інформації лежить в них. Вся інформація, що була отримана на цьому кроці пишеться в інший топік в Kafka. На цьому етап отримання даних завершується.

```
Орден Досконалості — державна нагорода Грузії,
заснована рішенням Парламенту Грузії № 1553 від 31 липня 2009
для нагородження видатних діячів культури, освіти, науки, мистецтва, спорту
та інших сфер за видатні заслуги.
```

Мал. 4.7 Приклад відфільтрованого html[19]

5.3 Детальних огляд частини обробки даних

Після завершення етапу отримання даних настає етап обробки даних. Він також складається з двох етапів:



Мал. 4.8 Етап обробки даних

На етапі під назвою “POS tagging” очищені дані читають з топіку Kafka для подальшого надання кожному слову тегу частини мови. За слово вважається будь-яка послідовність символів, що оточена пробілами. Словом можуть бути знаки пунктуації, вони виділяють в категорію “PUNCT”. Теги видаються за допомогою моделі машинного навчання з архітектурою “Bidirectional LSTM”. Результат цього етапу виглядає наступним чином:

```
Орден/NOUN Досконалості/NOUN –/PUNCT державна/ADJ нагорода/NOUN Грузії,/PROPN
заснована/ADJ рішенням/NOUN Парламенту/NOUN Грузії/PROPN №/NOUN 1553/PROPN
від/ADP 31/ADJ липня/NOUN 2009/ADJ для/ADP нагородження/NOUN видатних/ADJ
діячів/NOUN культури,/PROPN освіти,/PROPN науки,/PROPN мистецтва,/ADJ
спорту/NOUN та/CCONJ інших/DET сфер/NOUN за/ADP видатні/ADJ заслуги./NOUN
```

Мал. 4.9 Етап POS тегування

Після того, як модель класифікувала слова в тексті, результат записується в Kafka топік. На цьому цей етап завершується.

На етапі “Triplets extraction” отримуються дані з попереднього етапу з топіку. Після цього робиться фільтрація тексту за наступними правилами:

- Видалення знаків пунктуації, крім тих, що розділяють речення (“.”, “!”, “?”, “\n”).
- Фільтрація стоп слів.

Потім текст розбивається на речення і для кожного речення будується Parse Tree на основі тегів з частинами мови, що були надані моделлю. Цей етап допомагає більш точно зрозуміти структуру речення та зробити більш точні припущення про наявність термінів та їх опису в ньому. Після цього здійснюється пошук триплетів, як можна записати до онтології. Пошук здійснюється за наступними правилами:

- Іменники, що стояли поряд, групуються в один термін.
- Здійснюється пошук об’єкту.
    - Пошук об’єкту завершується, коли знайдено перший іменник.
- Здійснюється пошук предикату.
    - Всі прикметники та іменники, що зустрілися до предикату, потенційно характеризують об’єкт.
    - Предикат - перше дієслово в реченні, яке по Parse Tree лежить в одній групі з об’єктом.
- Здійснюється пошук суб’єкту.
    - Всі прикметники, що зустрілися до суб’єкту, потенційно характеризують його.

- Суб’єкт - перший іменник, що лежить після предикату.

Таким чином ми отримуємо триплети, які потенційно можуть бути записані до онтології.

```
object: Орден Досконалості
object_characteristic:
державна нагорода Грузії;
заснована рішенням Парламенту Грузії 1553 від 31 липня 2009 для нагородження видатних діячів культури...;
```

Мал. 4.10 Витягування триплетів

Як ми бачимо з малюнку 4.9, завдяки цим правилам було витягнуто лише об’єкт та його характеристики. Це сталося через те, що в цьому реченні немає предикату. Після цього всі терміни та їх характеристики зберігаються у тимчасове сховище. В моєму випадку це файл. Напроти цих термінів та характеристик ставиться число, яке значить кількість таких термінів у колекції проаналізованих документів. Як тільки таких входжень стає більше п’яти, то термін потрапляє до онтології.

5.4 Можливості розширення

Побудована система має широкий спектр можливостей розширення, що буде зроблено в ході подальшої роботи. По перше, в цій роботі не було предметної області, а досліджувалась можливість імплементації подібної системи. Систему можна розширити класифікацією текстів, що дозволить будувати окремі онтології до кожної предметної області. По друге, можна покращити точність з виокремлення термінів та їх характеристик. Наприклад, зробити тегування не за частинами мови, а за частинами речення, що спростить отримання триплетів, тому що триплет має вигляд: Підмет - Присудок - Додаток. Оптимізація коду є ще одним моментом для покращення. Наразі система може обробити лише 20 мегабайт на секунду.

Більшість часу витрачається на POS тегуванні задопомогою LSTM моделі. Підвищити пропускну здатність можна також за допомогою додавання комп’ютерних ресурсів до системи. Під час роботи все тестувалося на одному комп’ютері з наступними характеристиками: AMD Ryzen 8 3800X 3.9 GHz, 16Gb RAM, 40MB/sec Internet.

Висновки

У ході виконання курсової роботи мені вдалося побудувати MVP системи з автоматизованою побудовою онтологій на основі великих даних. Система має необмежений потенціал до горизонтального масштабування. Використана мікросервісна архітектура, що дозволяє легко розширювати систему. Під час проведення експериментів було оброблено та збережено більше 500 гігабайт даних, що свідчить про її потенціал.

Список використаної літератури

1. Laney D. 3D Data Management: Controlling Data Volume, Velocity, and Variety / Laney. // META Group. – 2001.
2. The Four V´s of Big Data. // IBM. – 2011.
3. Biehn N. The Missing V's in Big Data: Viability and Value / Neil Biehn. // Wired. – 2013.
4. The Reality Behind The Enigma That's the Internet [Електронний ресурс] – Режим доступу до ресурсу: https://www.digitalinformationworld.com/2018/06/how-internet-works.html.
5. Черняк Л. Большие Данные — новая теория и практика / Леонид Черняк. // Открытые системы. СУБД. – 2011. – №10.
6. Программное обеспечение повышенной готовности промежуточного уровня в Linux, часть 1: Heartbeat и Web-сервер Apache [Електронний ресурс] – Режим доступу до ресурсу: http://www.interface.ru/home.asp?artId=2620.
7. ukrainian-pos-tagger [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/dutkaD/ukrainian-pos-tagger.
8. Мак-Каллок У. Логическое исчисление идей, относящихся к нервной активности / У. Мак-Каллок, В. Питтс., 1956. – (Автоматы).
9. Винер Н. Кибернетика / Н. Винер., 1961
10. Graves A. A Novel Connectionist System for Improved Unconstrained Handwriting Recognition / A. Graves, M. Liwicki, S. Fernandez. // IEEE Transactions on Pattern Analysis and Machine Intelligence.. – 2009. – №31. – С. 855–868.
11. Miljanovic M. Comparative analysis of Recurrent and Finite Impulse Response Neural Networks in Time Series Prediction / Miljanovic. // Indian Journal of Computer and Engineering. – 2012. – №3.
12. LSTM – сети долгой краткосрочной памяти [Електронний ресурс]. – 2017. – Режим доступу до ресурсу: https://habr.com/ru/company/wunderfund/blog/331310/.
13. Liu L. Encyclopedia of Database Systems / L. Liu, T. Özsu., 2009.
14. Sowa J. Conceptual Structures. Information Processing in Mind and Machine / J. Sowa. // Addison Wesley. – 1984.
15. Van Rossum G. SETL (was: Lukewarm about range literals) [Електронний ресурс] / Guido van Rossum. – 2000. – Режим доступу до ресурсу: https://mail.python.org/pipermail/python-dev/2000-August/008881.html.

16. Special method names [Електронний ресурс] // The Python Language Reference. Python Software Foundation. – 2018. – Режим доступу до ресурсу: https://docs.python.org/3.0/reference/datamodel.html#special-method-names.
17. Kreps J. The Log: What every software engineer should know about real-time data's unifying abstraction [Електронний ресурс] / Kreps. – 2013. – Режим доступу до ресурсу: https://www.google.com/url?q=http://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying&sa=D&source=editors&ust=1617571367762000&usg=AOvVaw0FcPW_SFmDdUj-Kts-O-sc.
18. Kafka 2.7 Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://kafka.apache.org/documentation/.
19. Орден Досконалості [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Орден_Досконалості.
20. Жлобинская О. Как представлять библиографические данные в RDF [Електронний ресурс] / О. Жлобинская – Режим доступу до ресурсу: http://www.rusmarc.ru/publish/bibdateRDF.pdf.
21. Rojas R. The First Computers: History and Architectures / Rojas., 2002. – 472 с.
22. Project crossbow: Crossbow: Network Virtualization and Resource Control [Електронний ресурс] – Режим доступу до ресурсу: https://web.archive.org/web/20100526223328/http://hub.opensolaris.org/bin/view/Project+crossbow/WebHome.
23. Claburn T. Docker looks big biz in the eye: It's not you, it's EE – Enterprise Edition [Електронний ресурс] / Thomas Claburn. – 2017. – Режим доступу до ресурсу: https://www.theregister.com/2017/03/03/docker_enterprise_edition/.

Додаток Г

Текст програми

```
from ai_tagged.sentence_extract import Sentence, Word, TagsToInt
import glob

def get_subtree(pos):
  if pos == TagsToInt.NOUN.value or pos == TagsToInt.PROPN.value:
    return 'NP'
  if pos == TagsToInt.VERB.value or pos == TagsToInt.AUX.value:
    return 'VP'
  if pos == TagsToInt.ADJ.value:
    return 'ADJ'
  return None


def get_start_subtree(sentence: Sentence):
  for word in sentence.words:
    pos = get_subtree(word.pos)
    if pos is not None:
      return pos
  return None


def analyze_sentence(words, tree):
  for i in range(len(words)):
    tree[words[i].pos] = words[i].word
    next_pos = TagsToInt.INVALID.value
    if i + 1 < len(words) - 1:
```

```
            next_pos = words[i + 1].pos
        if next_pos == TagsToInt.NOUN.value or next_pos ==
TagsToInt.PROPN.value:
            tree['NP'] = analyze_sentence(words[i+1:], {})
            return tree
        if next_pos == TagsToInt.VERB.value or next_pos ==
TagsToInt.AUX.value:
            tree['VP'] = analyze_sentence(words[i+1:], {})
            return tree
        if next_pos == TagsToInt.ADJ.value:
            tree['ADJ'] = analyze_sentence(words[i+1:], {})
            return tree
    return tree


def get_triplet(words):
    result = [[]]
    i = 0
    looking_for = 0
    prev = -1
    for word in words:
        if prev == TagsToInt.PROPN.value and looking_for == 1 and word.pos ==
prev:
            result[i].append(word.word)
        if (get_subtree(word.pos) == 'NP') and (looking_for == 0 or looking_for ==
2):
            result[i].append(('object:' if looking_for == 0 else 'subject:') +
word.word)
            looking_for += 1
```

```
            if looking_for == 3:
                looking_for = 0
                i += 1
                result.append([])
        if get_subtree(word.pos) == 'VP' and looking_for == 1:
            result[i].append('predicate:' + word.word)
            looking_for += 1
        if get_subtree(word.pos) == 'ADJ' and (looking_for == 0 or looking_for ==
2):
            result[i].append(('object_class:' if looking_for == 0 else 'subject_class:')
+ word.word)
        prev = word.pos
    return [' '.join(triplet) for triplet in result]

from enum import Enum
import glob
from typing import List



class TagsToInt(Enum):
    INVALID = -1
    ADJ = 0
    ADP = 1
    ADV = 2
    AUX = 3
    CCONJ = 4
    DET = 5
    INTJ = 6
    NOUN = 7
```

```
    NUM = 8
    PART = 9
    PRON = 10
    PROPN = 11
    PUNCT = 12
    SCONJ = 13
    SYM = 14
    VERB = 15
    X = 16


tags = {
    'ADJ': TagsToInt.ADJ,
    'ADP': TagsToInt.ADP,
    'ADV': TagsToInt.ADV,
    'AUX': TagsToInt.AUX,
    'CCONJ': TagsToInt.CCONJ,
    'DET': TagsToInt.DET,
    'INTJ': TagsToInt.INTJ,
    'NOUN': TagsToInt.NOUN,
    'NUM': TagsToInt.NUM,
    'PART': TagsToInt.PART,
    'PRON': TagsToInt.PRON,
    'PROPN': TagsToInt.PROPN,
    'PUNCT': TagsToInt.PUNCT,
    'SCONJ': TagsToInt.SCONJ,
    'SYM': TagsToInt.SYM,
    'VERB': TagsToInt.VERB,
    'X': TagsToInt.X,
```

```
}


class Word:
    __separator = '!#wp#!'

    def __init__(self, word, pos):
        self.word = word
        self.pos = pos

    def to_string(self):
        return Word.__separator.join([self.word, str(self.pos)])

    @staticmethod
    def from_string(word_str):
        word, pos = word_str.split(Word.__separator)
        return Word(word, int(pos))


class Sentence:
    separator = '!#www#!'
    sentence_separator = '!#sss#!'

    def __init__(self):
        self.words: List[Word] = []

    def add_word(self, word):
        self.words.append(word)
```

```
  def to_string(self):
      return Sentence.separator.join([word.to_string() for word in self.words]) +
Sentence.sentence_separator

  @staticmethod
  def from_string(words_str):
      words = [Word.from_string(word) for word in
words_str.split(Sentence.separator)]
      sentence = Sentence()
      for word in words:
          sentence.add_word(word)
      return sentence


def read_text(path):
  with open(path, 'r', encoding='utf-8') as f:
      lines = f.readlines()
      return ''.join(lines)


def write_text(path, text):
  with open(path, 'w', encoding='utf-8') as f:
      f.write(text)


def process_text(text: str):
  words = text.split(' ')
  sentences = []
  sentence = Sentence()
```

```
for word in words:
    is_new_sentence = False
    is_end_sentence = False
    pos = TagsToInt.INVALID
    pos_str = ''
    for tag in tags.keys():
        if tag in word:
            pos = tags[tag]
            pos_str = '/' + tag
            break

    if pos.value == TagsToInt.INVALID.value:
        print(word)
        continue
    word = word.split(pos_str)[0]
    if len(word) == 0:
        continue
    if word[-1] == '.' or word[-1] == '!' or word[-1] == '?' or word[-1] == '\n':
        is_end_sentence = True
    if not word[-1].isalnum():
        word = word[:-1]
    if len(word) == 0:
        continue
    if word[0] == '\n':
        is_new_sentence = True
    word = word.replace('\n', '')
    if is_new_sentence:
        sentences.append(sentence)
        sentence = Sentence()
```

```
            if len(word) > 0:
                sentence.add_word(Word(word, pos.value))
        else:
            if len(word) > 0:
                sentence.add_word(Word(word, pos.value))
            if is_end_sentence and len(sentence.words) > 0:
                sentences.append(sentence)
                sentence = Sentence()
    if len(sentence.words) > 0:
        sentences.append(sentence)
    return sentences

from tree_builder.sentence_extract_pos_tag import SentenceCharacteristics,
MorphClass, WordCharacteristics
from typing import List, Dict
import glob

def get_subtree(morph_class: MorphClass):
    if morph_class.is_noun or morph_class.is_proper_geo or
morph_class.is_proper_name or morph_class.is_proper_surname or
morph_class.is_proper_secname:
        return 'N'
    if morph_class.is_verb:
        return 'V'
    if morph_class.is_adjective:
        return 'A'
    return None

def get_type(morph_class: MorphClass):
```

```
    if morph_class.is_proper_surname:
        return 'LAST_NAME'
    if morph_class.is_proper_name:
        return 'NAME'
    if morph_class.is_proper_secname:
        return 'SEC_NAME'
    if morph_class.is_proper_geo:
        return 'GEO'
    if morph_class.is_adverb:
        return 'ADVB'
    if morph_class.is_verb:
        return 'VERB'
    if morph_class.is_noun:
        return 'NOUN'
    if morph_class.is_adjective:
        return 'ADJ'
    if morph_class.is_conjunction:
        return 'CONJ'
    if morph_class.is_misc:
        return 'MISC'
    if morph_class.is_personal_pronoun:
        return 'PRONOUN'
    return 'SOME'


def is_name(word_type: str):
    return 'LAST_NAME' in word_type or 'SEC_NAME' in word_type or
word_type == 'NAME'
```

```
def is_geo(word_type: str):
    return 'GEO' in word_type


def build_parse_tree(words: List[WordCharacteristics]):
    prev_type = ''
    tree = {}
    i = 0
    result = []
    for word in words:
        if len(word.lemma) == 1 and not word.lemma.isalpha():
            continue
        word_type = get_type(word.morph_class)
        result.append({'word': word.lemma, 'type': word_type})
    return result


def group_items(words: List):
    result = []
    prev_type = ''
    offset = 0
    for i, word in enumerate(words):
        if (is_name(word['type']) and is_name(prev_type)) or (is_geo(word['type'])
and is_geo(prev_type)):
            result[i - 1 - offset]['word'] += ' ' + word['word']
            offset += 1
        else:
            result.append(word)
```

```
        prev_type = word['type']
    return result


def get_triplet(words):
    result = []
    looking_for = 0
    for word in words:
        if (is_name(word['type']) or is_geo(word['type'])) and (looking_for == 0 or
looking_for == 2):
            result.append(word['word'])
            looking_for += 1
        if word['type'] == 'VERB' and looking_for == 1:
            result.append(word['word'])
            looking_for += 1
    return result

from pullenti_wrapper.langs import (
    set_langs,
    UA,
)
from pullenti_wrapper.processor import (
    Processor)
from help_classes import MorphClass
from sentence_extract import SentenceExtract
from typing import List
import glob
```

```
class WordCharacteristics:

  delimiter = ' !word_char_part! '

  def __init__(self,
               morph_class: MorphClass,
               lemma: str):
    self.morph_class = morph_class
    self.lemma = lemma

  def to_string(self):
    return '{}{}{}{}'.format(self.morph_class.to_string(),
WordCharacteristics.delimiter, self.lemma, WordCharacteristics.delimiter)

  @staticmethod
  def from_string(word: str):
    split = word.split(WordCharacteristics.delimiter)
    return WordCharacteristics(MorphClass.from_string(split[0]), split[1])

#word.lemma if hasattr(word, 'lemma') else word.value
class SentenceCharacteristics:

  delimiter = ' !sentence_char_part! '
  word_delimiter = '!word! '

  def __init__(self, sentence_type, words: List[WordCharacteristics]):
    self.sentence_type = sentence_type
    self.words_num = len(words)
    self.words = words
```

```
  def to_string(self):
      return SentenceCharacteristics.delimiter.join([str(self.sentence_type)] +
[SentenceCharacteristics.word_delimiter.join([word.to_string() for word in
self.words])])

  @staticmethod
  def from_string(sentence: str):
      split = sentence.split(SentenceCharacteristics.delimiter)
      sentence_type = int(split[0])
      words = split[1].split(SentenceCharacteristics.word_delimiter)
      words = [WordCharacteristics.from_string(word) for word in words if word
!= '\n' and len(word) > 0]
      return SentenceCharacteristics(sentence_type, words)



class Preprocessor:
  def __init__(self):
      set_langs([UA])
      self.processor = Processor([])

  def __extract_morph_class_characteristics(self, word):
      return MorphClass(
          word.morph.class0_.is_adjective,
          word.morph.class0_.is_adverb,
          word.morph.class0_.is_conjunction,
          word.morph.class0_.is_misc,
          word.morph.class0_.is_noun,
          word.morph.class0_.is_personal_pronoun,
```

```
            word.morph.class0_.is_proper,
            word.morph.class0_.is_proper_geo,
            word.morph.class0_.is_proper_name,
            word.morph.class0_.is_proper_secname,
            word.morph.class0_.is_proper_surname,
            word.morph.class0_.is_undefined,
            word.morph.class0_.is_verb
        )

    def __make_sentence(self, words: List[WordCharacteristics], word):
        if word is None:
            return words
        class_characteristics = self.__extract_morph_class_characteristics(word)
        words.append(WordCharacteristics(class_characteristics, word.lemma if
hasattr(word, 'lemma') else word.value))
        return self.__make_sentence(words, word._m_next)


    def build_sentence_characteristics(self, sentence_type, text):
        processed_words = self.processor(text)
        words = self.__make_sentence([], processed_words.raw.first_token)
        return SentenceCharacteristics(sentence_type, words)


def extract_sentences(lines: List[str]):
    return SentenceExtract.extract_sentences(lines)
```