Міністерство освіти і науки України
Рівненський державний гуманітарний університет
Кафедра інформаційних технологій та моделювання

Кваліфікаційна робота

за освітнім ступенем «магістр»

на тему:

ТЕХНОЛОГІЯ WEB SCRAPING: МЕТОДИ ТА ЗАСТОСУВАННЯ

Виконав:
здобувач 2 курсу
групи М-КН-2
спеціальності 122 «Комп’ютерні науки»
Прокопюк Дмитро Олександрович
Науковий керівник:
к.т.н. Шевцова Н.В.

Рівне-2025

ЗМІСТ

СПИСОК ТЕРМІНІВ ТА УМОВНИХ ПОЗНАЧЕНЬ

CNN (Convolutional Neural Networks) — згорткові нейронні мережі

R-CNN (Region-based CNN) — сімейство двостадійних CNN-детекторів, що використовує регіональний підхід

RPN (Region Proposal Network) – окрема підмережа, яка навчена генерувати пропозиції прямокутних областей, які, ймовірно, містять об’єкти

YOLO (You Only Look Once) — одностадійна модель, що трактує детектування як задачу регресії і безпосередньо прогнозує рамки об’єктів за один прогін мережі

SSD (Single Shot MultiBox Detector) — одностадійний детектор

Nx (Nrwl Extensions for Angular) — платформа для організації монорепозиторіїв JavaScript/TypeScript-проєктів, яка дозволяє об’єднати кілька додатків і бібліотек в одному репозиторії

REST API —це набір правил для обміну даними між програмами через інтернет, що використовується для одноразових HTTP-запитів до сервера

WebSocket — протокол, обраний для динамічних оновлень у реальному часі та передачі прогресу сканування

CORS — пакет (middleware) для Express, що автоматично виставляє CORS-заголовки і дозволяє клієнту та серверу працювати на різних портах під час розробки

RGB — колірний простір, з якого можуть перетворювати кольорові зображення

HSV / LAB — колірні простори, до яких корисно перейти з RGB, щоб краще відокремити інформацію про колір від яскравості

ВСТУП

Актуальність роботи. Розвиток інформаційних технологій характеризується швидким зростанням обсягів візуальних даних. Автоматичне розпізнавання патернів (образів) набуло особливої актуальності в різних сферах діяльності, від біометрії до аналізу медичних знімків та систем автономного керування. Розпізнавання патернів – це автоматизований процес виявлення та впорядкування типових закономірностей чи структур в масивах даних (наприклад, даних з вебсайтів), базуючись на статистичних моделях чи попередньому досвіді [2]. Дана технологія бере свій початок в інженерії та статистиці, а сьогодні ми розглядаємо його в контексті машинного навчання. Розпізнавання образів широко використовується у статистичному аналізі даних, комп'ютерному зорі, обробці сигналів, пошуку інформації, стисненні даних, комп'ютерній графіці, тощо [1]. Оскільки ця технологія має широкий спектр застосувань, то її дослідження та імплементація є важливим завданням для спеціалістів інженерії програмного забезпечення.

Детектування об'єктів є ключовим напрямком у комп'ютерному зорі. Ця технологія автоматично ідентифікує та визначає точне розташування об'єктів заданих категорій (наприклад, людей, автомобілів, товарів) у даних вебсайтів. Така функціональність є основою для багатьох прикладних систем, зокрема систем відеоспостереження, автономних роботів, автомобілів та пошуку на вебсайтах. Класичні методи розпізнавання образів, які використовували на практиці вимагали ручної обробки даних та створенні ознак (наприклад, виділення контурів, кутів, текстур) та застосування алгоритмів класифікації. Поява глибоких нейронних мереж, зокрема згорткових нейронних мереж (Convolutional Neural Networks, CNN), кардинально змінили підхід у цій галузі. Моделі на основі CNN навчаються безпосередньо на необроблених піксельних даних і самостійно виділяють інформативні ознаки різних рівнів складності. Даний підхід дозволяє ефективно працювати з інформацією, суттєво підвищити точність та універсальність розпізнавання об'єктів під час опрацювання реальних даних, що часту мають шум,

перепади освітлення, зміщення ракурсу та фону.

Поряд із цим, зважаючи на стрімкий розвиток Інтернет-середовища надзвичайно актуальним є завдання автоматизованого збору та аналізу візуального контенту. Вебскрапінг – це технологія автоматичного вилучення даних з вебсторінок. Незважаючи на те, що класичні вебскрапери переважно аналізують текст та структуру HTML, сьогодні цей підхід стає менш ефективним, адже сучасні вебсайти все частіше використовують багато динамічного контенту, нескінченну прокрутку та відео. Тут і виникає потреба поєднання вебскрапінгу з комп'ютерним зором, адже це дасть змогу скраперу «бачити» сторінку подібно до людини та вилучати з неї візуальні дані і смислову інформацію. Методи комп'ютерного зору у вебскрапінгу дозволяють точно аналізувати та локалізувати дані на сторінці, тим самим підвищуючи повноту та точність зібраних даних [10]. В контексті даної роботи це дає можливість об'єднати механізм збору даних з вебсайтів та їх подальшої обробки для розпізнавання об'єктів в єдиний цикл.

Мета роботи полягає у розробці функціонального програмного засобу, що демонструє застосування сучасних інформаційних технологій та автоматизує аналіз даних із вебсередовища.

Для досягнення цієї мети були поставлені та вирішені такі **завдання**:

1) дослідити теоретичні основи розпізнавання образів;
2) сформулювати постановку задачі автоматизованого детектування об'єктів аналізуючи вебконтент;
3) розробити архітектуру програмного забезпечення (монорепозиторій із використанням Nx), що інтегрує клієнтську та серверну частину;
4) визначити структуру модулів, інтерфейсів та взаємодію між компонентами;
5) реалізувати клієнтську частину на основі фреймворку (Angular) для забезпечення повноцінного користувацького інтерфейсу (UI);
6) реалізувати серверну частину на платформі Node.js, яка безпосередньо відповідатиме за вебскрапінг;
7) включити в реалізацію фрагменти коду з репозиторію, щоб

продемонструвати ключові алгоритми;

8) використати сторонні бібліотеки та інструменти (Angular Material, Puppeteer, Socket.io, тощо) для ефективної реалізації;
9) провести тестування розробленої системи, проаналізувати результати та зробити висновки щодо досягнення мети та оцінити потенційні напрямки для подальшого розвитку.

Об'єкт дослідження – процес автоматизованого аналізу візуального контенту з вебсередовища, що включає вебскрапінг та розпізнавання об'єктів на зібраних даних.

Предмет дослідження – методи, моделі та програмні засоби вебскрапінгу та детектування об'єктів на вебресурсів, а також архітектурні рішення для інтеграції системи збору та обробки візуального контенту в єдину систему.

Методи дослідження. У процесі роботи розглядались слідуючі методи розпізнавання образів і машинного навчання (згорткові нейронні мережі), комп'ютерного зору (сегментація, виділення ознак, детектування об'єктів), вебскрапінгу та автоматизації браузера (Puppeteer), програмної інженерії (проєктування архітектури, побудова монорепозиторію Nx, розробка клієнт–серверного застосунку), тестування програмного забезпечення та аналізу результатів.

Апробація і впровадження результатів. Основні теоретичні та практичні результати здійсненого дослідження в межах цієї кваліфікаційної роботи обговорювалися та доповідалися на X Міжнародній науково-практичній конференції з проблем вищої освіти і науки «Інформаційні технології в освіті, науці і виробництві» (ІТОНВ-2025)» (м. Луцьк, 23-24 травня 2025 року) та були опубліковані тези доповіді [14].

Структура роботи. Кваліфікаційна робота складається зі вступу, п'яти розділів, висновків, списку використаних джерел та додатків. У першому розглядаються теоретичні основи поняття розпізнавання образів, класифікації та кластеризації даних, основні алгоритми кластеризації, методи попередньої обробки даних з вебсайтів, підходи до сегментації даних та виділення контурів, а також

сучасні методи детектування об’єктів з використанням згорткових нейронних мереж. Другий розділ присвячено постановці задачі (формулюванні проблеми та функціональним можливостям системи) та опису архітектури системи, а саме: структуру монорепозиторію Nx, клієнтську і серверну частини, їх компоненти і взаємодію, а також реалізовані інтерфейси (API) і маршрутизацію. Третій розділ містить конкретну реалізацію програмного забезпечення із фрагментами коду: окремо розглянуто реалізацію клієнтської та серверної частин, використані бібліотеки, технології та налаштування. Також в цьому ж розділі зосереджений на тестуванні системи та аналізі того, наскільки вдалося виконати поставлені завдання. Вкінці додані висновки з оцінкою отриманих результатів і переліком можливих подальших удосконалень, список джерел, які було опрацьовано та додатки з текстами програмного коду.

РОЗДІЛ 1
ЗАГАЛЬНІ ТЕОРЕТИЧНІ ПОЛОЖЕННЯ РОЗПІЗНАВАННЯ ОБРАЗІВ

1.1. Розпізнавання образів: класифікація, кластеризація

Розпізнавання образів (pattern recognition) – це галузь штучного інтелекту, що займається автоматичним виявленням та ідентифікацією шаблонів даних. Під шаблоном можна розуміти будь-яку повторювану структуру або особливість, характерну для певного класу даних. Формально «розпізнавання образів займається автоматичним виявленням закономірностей даних за допомогою комп'ютерних алгоритмів та використанням цих закономірностей для здійснення таких дій, як класифікація їх на різні категорії» [1]. Інакше кажучи, система розпізнавання отримує на вході сирі дані та автоматично класифікує їх – тобто відносить до однієї з наперед визначених категорій, спираючись на знання, здобуті під час навчання або на основі заданих правил.

Розпізнавання образів тісно пов’язане з поняттями класифікації та кластеризації даних.

Класифікація – це процес віднесення об’єкта до одного з наперед визначених класів на основі навчального вибіркового набору (наприклад, визначення, чи належать дані до класу «кіт» чи «пес» у вже відомих прикладах).

Кластеризація – це процес поділу множини об’єктів на групи (кластери) за схожістю, причому наперед невідомо, скільки і яких саме груп існує. Також, у статті від V7 Go під назвою «Pattern Recognition in Machine Learning [Basics & Examples]», кластеризація визначається, “як метод навчання без учителя, який групує подібні екземпляри даних у кластери”. Іншими словами, класифікація є керованим (supervised) видом розпізнавання образів, де використовується розмічений набір даних для навчання, а кластеризація – некерованим (unsupervised), де алгоритм самостійно шукає структуру в даних. Загалом розпізнавання образів охоплює обидва підходи: класифікацію і кластеризацію даних на основі виявлених шаблонів [2].

Кластеризація є одним із ключових методів розвідувального аналізу даних. Її мета полягає у групуванні об’єктів таким чином, щоб елементи в одному кластері були максимально схожими між собою, а об’єкти з різних кластерів –максимально відмінними. Схожість може визначатися різними метриками: евклідова відстань, кореляція чи інші міри подібності, що залежать від природи даних та конкретної задачі. Результатом кластеризації являється поділ вибірки на кластери, кожен з яких об’єднує схожі об’єкти [3].

Ієрархічна кластеризація формує вкладену структуру (ієрархію) кластерів. Вона буває *агломеративною* – кожен об’єкт початково утворює окремий кластер, а кластери поступово об’єднуються, та *дивізивною* – початково всі об’єкти належать одному кластеру, який потім послідовно розділяється на менші частини. Результат такого аналізу часто представляють у вигляді дендрограми. Перевагою такого методу є можливість не задавати кількість кластерів наперед, проте недоліком – висока обчислювальна складність для масивних наборів даних [4].

Щільнісні методи кластеризації представлені алгоритмами, які визначають кластери, як області підвищеної щільності точок у просторі. Найпоширенішим є алгоритм DBSCAN (Density-Based Spatial Clustering of Applications with Noise). Він знаходить кластери, «розростаючись» із точок високої щільності та додаючи сусідні точки, які відповідають критеріям щільності. DBSCAN здатен виявляти кластери довільної форми та розпізнавати шумові точки, які не належать жодному кластеру [5].

До інших відомих підходів належать методи на основі ймовірнісних розподілів, такі, як кластеризація за допомогою змішаних моделей Гауса, метод середнього зсуву, самоорганізаційні карти Кохонена та інші. Варто зазначити, що поняття «кластер» у різних алгоритмах трактується по-різному:

- як група з малими відстанями між об’єктами;
- як щільна область у просторі ознак;
- як сполучена область у графі зв’язності.

Саме тому вибір конкретного алгоритму кластеризації залежить від типу

даних та цілей аналізу.

У контексті розпізнавання образів широко застосовується для групування подібних зображень чи для сегментації зображень, що фактично являється кластеризацією пікселів за ознаками кольору, яскравості чи текстури. Кластеризація вважається «новітнім підходом, який пропонує найкраще з обох світів. Він об'єднує задачу виявлення об'єктів, де метою є виявити клас об'єкта разом із прогнозуванням обмежувальної рамки на зображенні і задачу семантичної сегментації, яка класифікує кожен піксель у попередньо визначені категорії. Таким чином, він дозволяє нам виявляти об'єкти на зображенні, одночасно точно сегментуючи маску для кожного екземпляра об'єкта» [8]. Наприклад, під час аналізу великих наборів нефільтрованих візуальних даних алгоритми кластеризації дозволяють згрупувати їх за змістовою чи візуальною схожістю, що полегшує подальший автоматизований або ручний аналіз. Важливим аспектом кластеризації у задачах комп’ютерного зору є вибір ознак, які можуть включати колірні гістограми, текстурні характеристики або глибокі ознаки, отримані за допомогою згорткових нейронних мереж [8].

1.2. Методи попередньої обробки даних

Перед застосуванням алгоритмів розпізнавання чи сегментації зображень з вебсайтів бажано провести попередню обробку (препроцесинг) – виконати набір процедур, які покращують якість зображення або виділяють його істотні характеристики. Метою попередньої обробки є підвищення вірогідності успішного виділення потрібних ознак і зменшення впливу небажаних факторів (шуму, неоднорідностей освітлення, тощо) на наступних етапах аналізу. До типових кроків препроцесингу належать:

1) нормалізація яскравості і контрасту;
2) знешумлення та згладжування;
3) покращення різкості;
4) перетворення кольорового простору;
5) масштабування та кадрування.

Розглянемо особливості кожного із описаних методів попередньої обробки зображень.

Щодо *нормалізації яскравості і контрасту*, то слід зазначити, що на зображення може впливати освітлення, саме тому одні зображення темніші, а інші світліші. Для усунення таких відмінностей застосовують вирівнювання гістограми або інші методи нормалізації інтенсивності пікселів. Наприклад, для «коригування контрасту часто застосовують гістограмну еквалізацію для перерозподілу інтенсивності пікселів, посилюючи контраст зображення. Виділяються такі методи, як локальне розтягування контрасту (LCS) та глобальне розтягування контрасту. LCS коригує значення пікселів локально, тоді як глобальне розтягування контрасту оцінює колірну палітру для оптимальних діапазонів контрастності» [5]. Отже, локальні методи підвищення контрасту працюють на рівні окремих областей зображення. Також використовується просте масштабування інтенсивностей пікселів до фіксованого діапазону – це полегшує роботу алгоритмів, зокрема нейронних мереж, і прискорює збіжність при навчанні моделей. Однією з опцій нормалізаціїї є стандартизація – віднімання середнього та ділення на стандартне відхилення. Вона може застосовуватися для того, щоб привести розподіл яскравостей до нормального вигляду з нульовим середнім – це робить дані незалежними від абсолютного рівня освітлення і покращує стійкість алгоритмів класифікації.

Реальні зображення часто містять різного роду шуми: спекл-шум, гауссівський шум, «сіль і перець», тощо. Для їх усунення використовують методи *знешумлення та згладжування***.** Наприклад, щоб алгоритми виділення ознак не реагували на випадкові флуктуації, здійснюють фільтрацію зображень. Класичними методами знешумлення є згладжувальні фільтри. Один з найпоширеніших – гаусівський розмиваючий фільтр (Gaussian blur), який згладжує зображення шляхом згортки з гаусівським ядром, усуваючи високочастотний шум. Для імпульсного шуму («сіль і перець») ефективним є медіанний фільтр – він замінює кожен піксель на медіанне значення в його околі, що дозволяє прибрати поодинокі аномальні піки яскравості без значного розмиття країв. Також застосовуються двосторонні фільтри, які згладжують однорідні області, зберігаючи

при цьому межі об’єктів [5].

Якщо ж потрібно підкреслити деталі на зображенні, тоді використовують методи підвищення або *покращення різкості*. Наприклад, метод нечіткої маски передбачає віднімання розмитої версії зображення від оригіналу, що акцентує високочастотні компоненти (межі) і робить зображення чіткішим. Такі прийоми можуть бути корисними, якщо важливим є розпізнавання дрібних деталей або країв на зображенні.

Часто перед аналізом кольорові зображення перетворюють у відтінки сірого кольору. Оскільки багато алгоритмів розраховані на одноканальне зображення, а колір не завжди є суттєвим для задачі, то виконують *перетворення кольорового простору* до градації сірого. В інших випадках, навпаки, корисно перейти з просторів RGB до HSV чи LAB, щоб краще відокремити інформацію про колір від яскравості.

Ще одним важливим етапом є приведення зображень до єдиного розміру або співвідношення сторін, тобто *масштабування та кадрування* зображення. Крім того, може виконуватися кадрування – видалення неінформативних полів або фонів, щоб залишити тільки цільову область.

Отже, попередня обробка готує зображення до аналізу, стандартизує та очищує його. В даній роботі під час реалізації детектування об’єктів на вебзображеннях також використовуються деякі із згаданих методів препроцесингу, щоб підвищити надійність розпізнавання об’єктів. Зокрема, масштабування зображень та нормалізація.

1.3. Сегментація даних та методи виділення меж даних

Сегментацію даних визначають, як «процес поділу цифрових даних на множину областей (сегментів), кожна з яких об’єднує пікселі за спільними властивостями» [12]. Іншими словами, сегментація – це розбиття даних вебсайту на смислові частини, які легше проаналізувати окремо, ніж всі одразу. На виході алгоритму сегментації зазвичай отримують або набір масок (бінарних даних), що відповідають різним сегментам, або межі (контури) між сегментами. Сегментація

тісно пов’язана з розпізнаванням образів. Вона часто використовується як попередній крок для більш складних операцій, таких, як детектування об’єктів або вимірювання параметрів об’єктів на зображенні.

Метою сегментації є спрощення або змінення подачі даних, щоб вони стали більш інформативні та зручні для аналізу [12]. Наприклад, у медичних знімках сегментація може виділити органи чи патологічні утворення; у задачах комп’ютерного зору – відокремити об'єкти від фону; у системах технічного зору – знайти окремі деталі на зображенні для їх підрахунку чи розпізнавання.

Існує кілька класичних підходів до сегментації даних з вебсайтів [7]:

1. *Порогова сегментація***.** Найпростіший метод – глобальне порогування яскравості. Встановлюється певний поріг інтенсивності. Всі пікселі, яскравість яких вище порогу, відносяться до одного класу («об’єкт»), пікселі з яскравістю нижче – до іншого («фон»). Класичним алгоритмом тут є метод Отсу, який автоматично обирає поріг, який мінімізує внутрішньокластерну дисперсію. Варто зазначити, що метод порогування добре працює, якщо об’єкти чітко відрізняються за рівнем сірого від фону, але неефективне при неоднорідному освітленні або, якщо об’єкти і фон мають перехідні значення яскравості.
2. *Методи росту регіонів.* Такі алгоритми починаються з однієї або декількох стартових точок і поступово «нарощують» області, приєднуючи сусідні пікселі, схожі за певним критерієм (колір, яскравість, текстура). Процес нарощування триває, доки область задовільняє умови однорідності. Метод регіонального росту дозволяє виділити сегменти складної форми, але потребує задання критерій схожості та умови зупинки зростання. Альтернативний підхід – розділення та злиття: дані з вебсайтів спочатку рекурсивно розділяються на більш однорідні квадранти, а потім сусідні схожі між собою області об'єднуються.
3. *Сегментація на основі контурів***.** Цей підхід ґрунтується на пошуку границь між різними об’єктами в зображенні. Спершу виконується

виділення меж (див. розділ 1.5) – знаходяться пікселі, що ймовірно лежать на контурах об’єктів. Потім, виходячи з набору таких пікселів-країв, алгоритм формує замкнені контури, які й вважаються границями сегментів.

Класичні підходи до сегментації можуть комбінуватися. Наприклад, спочатку порогування, потім очищення результату морфологічними операціями (ерозією/розширенням), потім виділення контурів. Результатом сегментації може бути набір областей або масок, що покривають всі візуальні дані без перекриття (при повній сегментації), або лише окремі вибрані області (при частковій сегментації). Також варто зазначити, у випадку з кольоровими візуальними даними, для отримання кращого результату порогування та подібні методи можна застосовувати у різних колірних каналах або просторах (HSV, Lab).

У рамках розпізнавання образів сегментація виконує допоміжну роль. Наприклад, спочатку сегментувавши дані, легше порахувати кількість об’єктів або виділити кожен об’єкт для окремого розпізнавання. В подальших розділах, особливо в контексті згорткових нейронних мереж, ми побачимо, як сучасні методи можуть виконувати сегментацію і детектування об’єктів одночасно.

Виділення меж (контурів) – це процес знаходження пікселів, що відповідають границям об’єктів зображення. Межа зазвичай проходить там, де відбувається різкий перехід яскравості або кольору. Виділення контурів є базовою операцією в аналізі даних вебсайту: краї представляють простимі та інформативні ознаки для розпізнавання об’єктів, сегментації, виявлення форм, тощо.

Більшість алгоритмів виділення меж ґрунтуються на обчисленні градієнта яскравості даних. Місця, де модуль градієнта великий, вважаються кандидатами на знаходження контуру.

Найпростіші оператори для цього – *оператори Собела* і *Прюітта*. Дані фільтри 3×3, що апроксимують часткові похідні дані по горизонталі та вертикалі. Пропустивши дані через такі фільтри, отримують оцінки градієнту G_x та G_y у кожній точці. Далі обчислюють величину градієнта G. Пікселі, де G перевищує заданий поріг, оголошуються точками контуру. Оператор Прюітта подібний до

Собела і також використовує фіксовані ядра для горизонтальної та вертикальної диференціації. Різниця між ними в деталях обчислення, але загальний результат схожий – отримання карти градієнтів.

Оператор Робертса використовує 2×2 ядра для оцінки градієнта під кутом 45° і він є одним з найпростіших, хоча і менш точний, якщо в даних присутній шум.

Більш досконалий і популярний алгоритм – *детектор Кенні (Canny***)**. Це багатокроковий метод, який включає: гаусівське згладжування даних (для знешумлення), знаходження градієнтів (оператори Собела), застосування нелокального максимумного пригнічення (non-maximum suppression) та подвійне порогування з гістерезисом. Подвійне порогування використовує два пороги: високий і низький. Точки з градієнтом вище високого порогу однозначно вважаються краями, точки нижче низького – не краями, а ті, що між порогами, включаються, якщо вони з'єднані з «сильними» краями. Такий підхід дозволяє отримувати безперервні контури та уникати фрагментації країв на окремі точки. Детектор Кенні виявляє широкий діапазон меж і загалом вважається одним з кращих класичних методів виділення контурів.

Існують і інші оператори: **Лапласіан Гауса (LoG)** – спочатку згладжуємо дані гаусом, потім застосовуємо лапласів оператор (другу похідну). Місця, де лапласіан змінює знак, вказують на наявність краю. Цей метод чутливий до шуму, але може виділяти більш тонкі деталі. **Оператор Шарра** – модифікація Собела для підвищення ізотропності. Іншими словами, він краще визначає градієнти незалежно від напрямку [6].

Загалом кожен з операторів має свої переваги і недоліки. Прості градієнтні оператори швидкі, але шумочутливі і дають товсті краї. Кенні складніший, але дає тонкі й точні межі. LoG може краще знаходити замкнуті контури, але підсилює шум. На практиці вибір алгоритму залежить від вимог до якості та швидкості. У сучасних системах комп'ютерного зору часто попередню класичну обробку (виділення меж, сегментація) замінюють або доповнюють нейронними мережами, які можуть навчитися виділяти потрібні границі самостійно. Проте, навіть при застосуванні нейронних мереж, розуміння класичних методів корисне – вони

можуть допомогти візуалізувати, які особливості навчена мережа повинна вловлювати (наприклад, щоб визначити край об’єкта, мережі фактично потрібно навчитись подібному до детектора Кенні функціоналу в одному з шарів).

1.4. Застосування згорткових нейронних мереж для детектування об’єктів

Згорткові нейронні мережі (Convolutional Neural Networks, CNN) на сьогодні є основою майже всіх високоточних рішень у задачах розпізнавання образів, зокрема для класифікації даних вебсайтів, виявлення об’єктів та сегментації. Особливістю CNN є наявність згорткових шарів, які виконують згортку вхідних даних з набором фільтрів. Ці фільтри автоматично навчаються виділяти характерні патерни такі, як прості краї і кути на перших шарах, складніші форми та текстури на наступних, і, зрештою, специфічні риси об’єктів на глибоких шарах. За рахунок згорток ієрархія ознак формується без ручного проєктування. Це дозволяє CNN перевершувати класичні методи з ручними ознаками (SIFT, HOG, тощо) у багатьох задачах комп’ютерного зору [13].

Детектування об’єктів – це задача не лише у визначенню присутності певного об’єкта на зображенні, але й у вказуванні місця знаходження (зазвичай у вигляді рамки навколо об’єкта). Класичний підхід до цієї проблеми до ери CNN полягав у поєднанні алгоритмів сегментації областей кандидатів та подальшої класифікації цих областей. Зі впровадженням CNN було запропоновано кілька ефективних архітектур для об’єднання цих кроків у єдиний процес.

Розглянемо характерні особливості *двостадійних CNN-детекторів***.** Представником є сімейство R-CNN (Region-based CNN). Алгоритм R-CNN (2014) генерував близько 2,000 кандидатних областей за допомогою класичного методу, а потім масштабував кожну область і пропускав через згорткову нейронну мережу для класифікації та уточнення рамки. Вдосконаленням був Fast R-CNN, де усі пропозиції областей оброблялися на спільних згорткових ознаках, що підвищило швидкість. Найбільш досконалий у цій лінійці – Faster R-CNN (2015), який навчив саму мережу генерувати пропозиції областей через окрему підмережу (Region

Proposal Network, RPN). Faster R-CNN складається з двох стадій: перша – RPN пропонує прямокутні області, ймовірно містять об’єкти, друга – класифікує ці області та точно регресує координати рамок. Такий підхід дуже точний і гнучкий. Однак, існують і недоліки: відносно невисока швидкість; двостадійні моделі зазвичай обробляють лише кілька кадрів за секунду, що недостатньо для реального часу. Тим не менш, у задачах, де швидкість не критична (медичні дані з вебсайтів, аналіз знімків із супутника), такі моделі переважають завдяки високій точності.

Цікаво, що на базі Faster R-CNN була розроблена архітектура Mask R-CNN для одночасного детектування та сегментації об’єктів. Mask R-CNN додає до другого етапу спеціальну гілку, яка для кожної знайденої рамки об’єкта генерує бінарну маску пікселів, що належать цьому об’єкту [8]. Таким чином, Mask R-CNN видає контури кожного екземпляра об’єкта на зображенні. Приклад, де нейронна мережа може інтегровано виконувати детектування і сегментацію: спершу знайти об’єкти як рамки (подібно до Faster R-CNN), а потім уточнити форму об’єкта на рівні пікселів. Mask R-CNN наразі є одним із стандартів для сегментації об’єктів зображення і часто використовується, коли важливо точно окреслити межі.

Крім двостадійних *CNN*-детекторів використовують також *одностадійні CNN-детектори.* Для того, щоб досягти роботи в реальному часі, були розроблені моделі, які не виконують окрему стадію генерування областей, а безпосередньо прогнозують рамки об’єктів на основі ознак, отриманих із згорткової мережі. Першою була модель YOLO (You Only Look Once), 2016. Ідея YOLO: розбити дані на сітку із $S \times S$ клітин і навчити мережу відразу передбачати для кожної клітини координати можливих рамок об’єктів і їхні класи. YOLO трактує детектування як задачу регресії: з однієї подачі даних мережа видає набір чисел, що відповідають параметрам всіх рамок і ймовірностям класів. На відміну від двостадійних підходів, YOLO не витрачає час на окремий механізм пошуку областей – весь процес є одностадійним і виконується за один прогін мережі. Завдяки цьому YOLO працює дуже швидко. Ціна за швидкість – дещо менша точність порівняно з двостадійними моделями, особливо в сценах з багатьма дрібними об’єктами або при сильному перекриванні об’єктів. Однак постійний розвиток (вдосконалення архітектури, використання anchor-боксів, методів тренування) дозволив новим версіям YOLO

значно підтягнути точність [9].

Інший відомий одностадійний детектор – SSD (Single Shot MultiBox Detector). Він також одноразово прогнозує множину рамок, але використовує декілька масштабів особливостей для кращого виявлення об’єктів різних розмірів. SSD і його спадкоємці (RetinaNet та інші) вважалися компромісом між точністю і швидкістю, поки YOLO не захопив цю нішу.

Отже, у сучасних системах детектування об’єктів за допомогою CNN існує дві основні парадигми:

1. *Двостадійна***:** Faster R-CNN та Mask R-CNN. Дає високу точність, особливо для складних сцен, але повільніша.
2. *Одностадійна***:** YOLO. Працює в реальному часі, простіша, проте може втрачати точність на дуже складних випадках.

Вибір підходу залежить від вимог задачі. В рамках цієї кваліфікаційної роботи акцент робиться не на тому, щоб з нуля навчити власну модель для детектування, а на інтегруванні існуючих методів у процес вебскрапінгу. Тому розроблена система може використовувати вже готові і працюючі рішення для розпізнавання об’єктів. Зокрема: використання переднавченої моделі (YOLOv5), як зовнішнього сервісу або бібліотеки, до якого передаються дані з вебсайтів, або виклик хмарного API комп’ютерного зору (Google Vision, Microsoft Azure Computer Vision), що повертає розпізнані об’єкти зображення. Важливо зазначити, що згорткові нейронні мережі здатні детектувати об’єкти навіть за наявності шумів, перекривання та інших складнощів, але вони потребують достатньої підготовки даних і обчислювальних ресурсів для виконання.

У підсумку, теоретична база, розглянута в цьому розділі закладає підґрунтя для побудови комплексної системи, здатної автоматично збирати візуальні дані з вебджерел і аналізувати їх для виявлення об’єктів.

РОЗДІЛ 2
ПРОЄКТУВАННЯ ВЕБСИСТЕМИ ДЛЯ АВТОМАТИЗАЦІЇ ДЕТЕКТУВАННЯ КОНТЕНТУ ВЕБСАЙТІВ

2.1. Постановка задачі та визначення функціональних вимог до системи

У даному розділі формулюється задача проєкту, визначаються вимоги до функціональності системи та очікувані результати роботи.

В даній роботі розглядається комплексна задача, що поєднує вебскрапінг та розпізнавання об’єктів у даних з вебсайтів.

Для досягнення мети система повинна виконувати такі *функції*:

1. *Отримання вебконтенту*. Система приймає адресу вебсторінки (URL) або множину адрес. Вона має завантажити вміст цих сторінок, включаючи динамічно підвантажувані елементи (скрипти, дані з вебсайтів). Зважаючи на те, що сучасні сайти можуть використовувати JavaScript для завантаження контенту, простого HTTP-запиту недостатньо, тому використовується підхід з емульованим браузером, щоб відрендерити сторінку так, як це робить звичайний браузер. У рамках проєкту обрано інструмент Puppeteer – це бібліотека для Node.js, яка керує безголовим браузером Chrome. З її допомогою програма зможе завантажити HTML-код сторінки, зачекати виконання скриптів, прокрутити сторінку, якщо треба, тощо.
2. *Виділення контенту*. Після завантаження сторінки необхідно виокремити з неї дані. Найочевидніше – це знайти всі теги <img> в HTML-коді та отримати посилання на їхні файли. Проте, візуальний контент не обмежується тільки тегами <img>. Дані можуть бути фоновими, згенерованими через <canvas>, або вбудованими як SVG. У рамках проєкту основна увага приділяється стандартним растровим даним, що вставлені як окремі файли (JPEG, PNG, тощо). Отже, система має зібрати URL усіх даних на сторінці, завантажити їх і підготувати до аналізу. Завантаження даних можна виконати або знову ж таки через браузер, або

через прямий HTTP-запит (наприклад, з використанням бібліотеки axios в Node.js). Враховується також, що деякі дані можуть бути великі за розміром. Саме тому, є обмежувати їх розмір або стискати для швидшої обробки. В даній роботі детектування саме картинок розглядається теоретично з можливістю практичної розробки, як наступного етапу розширення можливостей вебскрапера. На цьому етапі ми працюємо з текстовими даними.

3. *Детектування об'єктів*. Основна функція – проаналізувати зібране дані і визначити, чи містяться в них об'єкти певного типу, а якщо так, то які саме та де. Конкретний тип об'єктів може бути різним залежно від задачі (для демонстрації можна вибрати, наприклад, розпізнавання обличчя, логотипів, або просто виводити опис об'єктів з використанням універсальної моделі). У межах даного проєкту допускається використання вже існуючих моделей глибокого навчання, а не навчання з нуля. Наприклад, інтегруємо готову модель YOLOv5, навченої на наборі даних COCO, яка вміє знаходити 80 типів об'єктів (люди, автомобілі, тварини, тощо). Також використовуємо API хмарного сервісу: передача даних з вебсайтів в Google Vision API і отримування списку розпізнаних об'єктів. Обраний підхід повинен мати змогу працювати в автономному режимі для потенційно великої кількості даних.
4. *Виведення результатів.* Результати роботи системи повинні бути представлені користувачу в зручній зрозумілій формі. Оскільки проєкт містить клієнтську частину, то результати будуть відображені на вебсторінці. Вони можуть представлятись, як список знайдених на сторінці об'єктів з накладеними рамками та мітками, або текстовий звіт:
   - image1.jpg знайдено: кішка, 2 людини;
   - Image2.png: об'єктів не виявлено, тощо.

   Важливо забезпечити зручне зіставлення між вихідною вебсторінкою та знайденими на ній об'єктами. Також, результати варто зберігати у вигляді

таблиці або файлу (Excel – тому в проєкті додано бібліотеку xlsx для генерації таблиць). У межах демонстрації достатньо відобразити інформацію на екран.

5. *Робота в автоматичному або напівавтоматичному режимі.* Система може дозволяти користувачу вручну ініціювати аналіз конкретної сторінки, вводячи URL і натискаючи кнопку «Сканувати». Також даній роботі передбачено, що модуль серверної частини може працювати за розкладом (додано пакет node-cron, параметри якого змінюються у налаштуваннях): наприклад, щоденно сканувати певний вебсайт на предмет появи нових даних з заданими об’єктами. Це зручно для сценаріїв моніторингу.

У рамках проєкту обсяг реалізації обмежений часовими рамками. Тому зроблено припущення, що будемо працювати з відносно простими вебсторінками (вибрані сайти) та шукати широковідомі об’єкти (наприклад, ті 80 класів, що підтримуються типовою моделлю YOLO). Продуктивність не оптимізована до рівня промислового застосування, але забезпечено прийнятний час відгуку для одиничного запиту (кілька секунд на завантаження і аналіз однієї вебсторінки). У системі не використовується база даних для збереження результатів. Інформація тримається в пам’яті та може експортуватися в Excel за запитом. Також безпекові функції (авторизація на сайтах, обходження капч, тощо) виходять за рамки цього проєкту, проте бібліотека puppeteer-extra зі stealth-плагіном, підключена і допомагає зменшити ймовірність блокування скрапера.

Вимірювані показники успішності системи можуть бути такі:

- *коректність***:** всі дані зі сторінки повинні бути розглянуті;
- *відповідність очікуванням***:** за наявності, об'єкти повинні бути перелічені у результатах;
- *точність***:** частка правильно знайдених об’єктів;
- *повнота*: кількість пропущених об’єктів;
- *швидкість*: середній час на обробку однієї сторінки.

Оскільки наша система більше інтеграційна та використовує готову модель, ключовим показником є коректність роботи та відповідність очікуванням користувача. В ході тестування (п.3.4) ці аспекти буде перевірено на декількох прикладах.

Таким чином, поставлена задача полягає у створенні веборієнтованої системи, здатної в автономному режимі збирати дані зі сторінок вебсайтів та виконувати на них автоматичне розпізнавання об’єктів із застосуванням сучасних алгоритмів комп’ютерного зору та інтеграцією готової моделі. Саме це поєднання двох складових, вебскрапінгу та комп’ютерного зору, представляє собою основну інновацію проєкту.

2.2. Проєктування та побудова архітектура системи

Розроблена система реалізована у вигляді монолітного репозиторію (монорепозиторію) з використанням інструментарію платформи Nx. Дана платформа призначена для організації монорепозиторіїв JavaScript/TypeScript-проєктів, що дозволяє об’єднати кілька додатків і бібліотек в одному репозиторії, спрощуючи повторне використання коду і керування залежностями. У спроєктованій системі монорепозиторій містить два основні додатки: клієнтський (Angular) та серверний (Node.js), а також спільні модулі, які використовуються обома частинами.

Загальна структура проєкту (монорепозиторію Nx) має таку організацію директорій:

1) apps/pholod – папка клієнтського вебзастосунку Pholod (умовна назва проєкту). Тут знаходиться стандартна структура Angular-додатку: модулі, компоненти, шаблони, стилі. Застосунок Pholod – це інтерфейс користувача для налаштування і запуску вебскрапінгу та перегляду результатів.
2) apps/server – папка серверного застосунку Server, що представляє собою Node.js API-сервер. Він відповідає за бекенд-логіку: завантаження веб-

сторінок, обробку даних з вебсайтів, виклик моделей розпізнавання, збереження і видачу результатів. Це застосунок типу Express-сервера (HTTP API) та одночасно Socket.io-сервера для реального часу.

3) libs/core – спільна бібліотека Core із загальними інтерфейсами, моделями даних та утилітами. Сюди можна віднести, наприклад, інтерфейси ImageInfo, DetectionResult – щоб і клієнт і сервер оперували одними й тими самими типами даних (TypeScript інтерфейси спільні). Також у Core можуть бути певні функції-утиліти (наприклад, форматування даних, логування), які використовуються повсюдно.
4) libs/scraper – бібліотека Scraper, що містить специфічну логіку вебскрапінгу. Зокрема, тут можуть бути функції на зразок:
   a) scrapePage(url: string): Promise<ImageInfo[]>, яка завантажує сторінку і повертає список знайдених даних;
   b) analyzeImage(img: ImageInfo): DetectionResult, яка аналізує дані з вебсайтів на наявність об’єктів, тощо.

   Винесення цього функціоналу в окрему бібліотеку дозволяє, наприклад, потенційно використовувати її як з боку сервера, так і, умовно, в інших додатках.
5) Конфігураційні файли Nx, TypeScript, ESLint, тощо (nx.json, tsconfig.json, angular.json і т.д.), які налаштовані так, щоб збирати та запускати обидва застосунки.

Архітектура є класичною клієнт-серверною: Angular-додаток працює у браузері користувача і надсилає запити до Node.js сервера. Сервер, у свою чергу, виконує всі трудомісткі операції (скрапінг, розпізнавання) і повертає результати клієнту. Комунікація відбувається двома способами, зокрема через:

1) *REST API* (HTTP-запити);
2) *WebSocket* (протокол Socket.io).

Так через *REST API* (HTTP-запити) сервер реалізує кілька ендпоінтів, наприклад: GET /api/scrape?url=http://... для початку сканування сторінки; GET

/api/results для отримання останніх результатів, тощо. Клієнт може викликати їх через HTTP (в Angular використовується HttpClient). Цей спосіб підходить для одноразових запитів.

WebSocket-з’єднання зручніше використовувати для динамічних оновлень у реальному часі (наприклад, прогресу сканування, або поступового надсилання результатів по мірі знаходження об’єктів). У проєкті інтегровано **Socket.io** як на сервері, так і на клієнті (Angular-бібліотека ngx-socket-io для зручної роботи з сокетами). При завантаженні сторінки Angular підключається до сервера по WebSocket. а сервер може емітити події, такі, як progress або done, де клієнт їх оброблятиме щоб оновити інтерфейс негайно і без перезавантаження. В тому числі, клієнт може відправити через сокет команду startScrape з параметром URL, замість виконання HTTP-запиту. Така двостороння комунікація робить систему більш інтерактивною.

Для забезпечення такої взаємодії сервер побудований одночасно як HTTP-сервер (на базі Express) і як Socket.io-сервер, підключений до HTTP-серверу. Це стандартна схема: після запуску сервер слухає, скажімо, порт 3333 на HTTP; HTTP-запити обробляються роутерами Express, а якщо відбувається WebSocket-з’єднання (від клієнтської бібліотеки Socket.io), то сервер приймає його і веде обмін повідомленнями через події.

У даній архітектурі відсутня окрема база даних – припускається, що потреби в довготривалому збереженні результатів немає або вони можуть експортуватися в файл. Для демонстрації результати сканування тримаються в пам’яті протягом сесії. Якщо потрібно, їх можна отримати через API або миттєво показати на клієнті.

Маршрутизація включає як маршрути в Angular-застосунку, так і маршрути (ендпоінти) на сервері:

- **Клієнт (Angular).** Застосунок має кілька сторінок, між якими здійснюється навігація. Головна сторінка з формою вводу URL та кнопкою «Сканувати», сторінка «Результати» для перегляду списку знайдених даних з вебсайтів та об’єктів та сторінка «Налаштування» (де

можна встановити, скажімо, розклад сканування чи параметри розпізнавання). Angular Router налаштований для цих маршрутів. Для прикладу, маршрут '/' – головна сторінка, 'results' – результати. Коли користувач переходить за цими посиланнями, Angular підвантажує відповідні компоненти. Оскільки даний проєкт зосереджений на функціональності, то структура клієнтського застосунку є доволі простою: один-два компоненти. Однак демонструється використання Angular Material для оформлення інтерфейсу (наприклад, використано Material <table> або <card> для списку результатів, Material <button>, індикатор прогресу, тощо). Це робить застосунок дружнім та зрозумілим для користувача.

- **Сервер (Express).** Налаштовано декілька маршрутів API. Всі вони, як правило, мають префікс /api (щоб відрізняти від маршрутів фронтенду, якщо застосунок розміщено на тому ж домені). Конкретні реалізовані маршрути:
  - GET /api/scrape – виконує скрапінг для заданого URL. Очікує, що параметр url передано як рядок запиту. При зверненні до цього маршруту сервер запускає процес завантаження сторінки за вказаною адресою, вилучає дані із сторінок, аналізує і формує відповідь. У найпростішому випадку відповіддю може бути JSON об’єкт з результатами аналізу. Якщо процес тривалий, то можна реалізувати асинхронно: одразу повернути відповідь, що «запуск здійснено», а результати надіслати пізніше через WebSocket або щоб клієнт окремо спитав їх. У проєкті для спрощення – виклик /api/scrape може синхронно чекати (до певного таймауту) і повернути результати.
  - GET /api/results – повертає останні результати сканування (наприклад, для випадку, коли запускалося через сокет і результати збереглися на сервері, їх можна отримати цим запитом). Це може стати в нагоді, якщо сторінку треба оновити або при повторному

вході користувача.

- POST /api/settings для оновлення налаштувань (наприклад, увімкнення/вимкнення автоматичного режиму або зміна порогу впевненості розпізнавання).
- GET /api/health для перевірки стану сервера, тощо.

В даному проєкті основну роль відіграє /api/scrape.

Варто зазначити, що сам Angular-додаток може бути зібраний і розгорнутий на тому ж сервері Node.js. Під час розробки Nx дозволяє запускати їх окремо: nx serve pholod запускає dev-сервер Angular (з портом 4200), nx serve server – сервер на 3333. Для локального розвитку CORS (cross-origin requests) було встановлено пакет cors і налаштовано Express використовувати cors(), щоб Angular dev-server міг звертатися до API на іншому порті. При деплої на продакшн, Angular зібраний у папку dist/pholod і сервер налаштований віддавати статичні файли звідти – тоді і клієнт і сервер будуть на одному домені/порті, тому проблема CORS зникне.

Складові серверної частини. Сервер (Node.js) використовує декілька важливих бібліотек, які визначають архітектуру модуля скрапінгу:

- *Puppeteer & puppeteer-extra (зі stealth-плагіном)***:** забезпечує керування Chrome. Stealth-плагін маскує деякі властивості headless-браузера, щоб його було важче виявити, як бота (наприклад, змінює navigator.webdriver, підроблює деякі параметри PDF і плагінів, відстрочує злиття повідомлень консолі, тощо). Це корисно для обходу простих анти-скрапінг заходів сайтів. Серверний код створює екземпляр браузера (puppeteer.launch), відкриває нову сторінку, переходить на URL (page.goto), чекає подій завантаження (використано waitUntil: 'networkidle0' щоб дочекатися завантаження мережевих запитів), потім виконує пошук даних з вебсайтів.
- *Axios:* використовується для завантаження даних з вебсайтів за URL, якщо треба отримати їхній байтовий вміст або метадані (наприклад, розмір). Puppeteer теж може це робити, але axios є зручнішим у деяких випадках.
- *Socket.io:* забезпечує двосторонній канал комунікації. На боці сервера

налаштовано слухачі, наприклад: socket.on('startScrape', handler), які при отриманні події запускають необхідні процеси. І навпаки, при отриманні результатів сервер емітить socket.emit('scrapeResult', data), а всі підключені клієнти (або конкретний, якщо адресовано конкретному сокету) одержують повідомлення з даними. Це дозволяє в реальному часі відобразити хід обробки: початок сканування, знайдено N даних, розпізнаються дані вебсайту 1, 2, ... завершено.

- *Node-cron:* дозволяє запланувати періодичне виконання функції. Архітектура модуля: якщо відповідна опція ввімкнена, при старті сервера ініціалізується cron.schedule('0 * * * *', task) (наприклад, щогодини) – і task викликає ту саму функцію сканування для заздалегідь заданого списку URL. Цей компонент працює у фоновому режимі і при спрацюванні так само через внутрішні механізми сервера оновлює стан або надсилає результати клієнту (наприклад, якщо клієнт у цей час підключений по WebSocket, можна емітити йому повідомлення про нові знайдені об’єкти).
- *Модуль розпізнавання об’єктів.* Як згадувалося раніше, глибока нейронна мережа в коді не реалізована з нуля. У прототипі можна використати хмарний сервіс: бібліотека axios дозволяє зробити HTTP POST запит до API, надіславши дані у форматі base64 та отримавши JSON з описом об’єктів. Це досить просто інтегрувати архітектурно – зробити функцію analyzeImage(buffer): Promise<DetectionResult> яка всередині викликає зовнішній API. У разі відсутності інтернету чи ключа API, альтернативно можна підключити TensorFlow.js або іншу ML-бібліотеку та завантажити навчений модельний файл (наприклад, coco-ssd – модель для браузера, що знаходить ті ж 80 класів об’єктів). Однак запуск TF.js на Node без апаратного прискорення може бути повільним. Тому в архітектурі виділено цей компонент, як абстракцію: є інтерфейс або функція detectObjects(image), а конкретна реалізація може бути підмінена. Це робить систему розширюваною. У коді проєкту, для простоти, ми робимо заглушку: завантажуємо дані для опрацювання, імітуємо розпізнавання.

Але архітектура передбачає, де цей виклик має відбуватися.

Для візуалізації взаємозв’язків створено діаграма архітектури (рис. 2.1), яка демонструє основні модулі, події та напрямки обміну даними. Angular UI надає форму вводу URL і кнопку, яка викликає або http.get('/api/scrape') або через Socket.io надсилає подію startScrape. Node Server отримує запит, викликає модуль Scraper, а він у свою чергу запускає PuppeteerBrowser і цикл обробки та аналізу всіх зображень, вкінці збираючи результати повертає серверу. Потім сервер або повертає результат у HTTP відповіді або шле подію scrapeResult через Socket.io з готовими даними. Angular UI приймає дані (через .subscribe() на socket подію або після http.get). Потім відображає таблицю, а якщо потрібна візуалізація, Angular може додатково завантажити саме дані з вебсайтів і накласти на нього мітки.

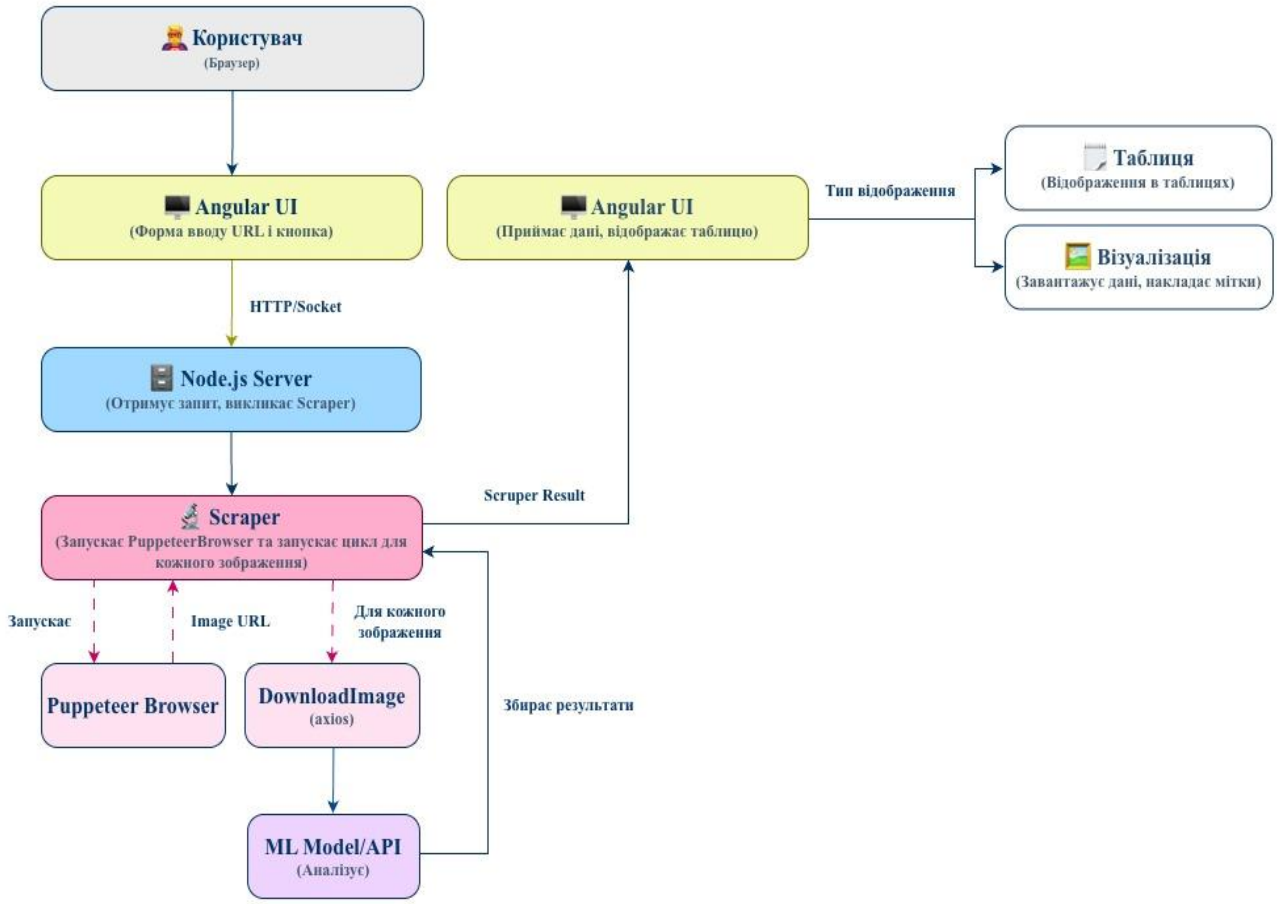

Рисунок 2.1. Діаграма архітектури

Послідовність виконання:

1. Користувач відкриває застосунок у браузері. Angular завантажено, компонент Home відображається.
2. Користувач вводить URL і натискає «Сканувати».
3. Викликається метод компонента Home, який надсилає через сокет подію startScrape із вказаним URL. Одночасно можна перевести користувача на сторінку «Результати» (наприклад, this.router.navigate(['/results'])), де відразу можна показати індикатор прогресу «Сканування триває…».
4. Node.js сервер, отримавши startScrape, викликає функцію scrapePage(url). Та відкриває сторінку через Puppeteer.
5. Поки Puppeteer завантажує сторінку, сервер може через той самий сокет емітити назад подію progress з повідомленням, наприклад, «Сторінка завантажується». Angular компонент Results підписаний на progress і оновлює текст прогресу.
6. Puppeteer отримав HTML. Виконується скрипт на сторінці, який витягує всі img (page.$$eval('img', els => els.map(e => e.src))). Отримано масив URL даних з вебсайтів. Сервер знає кількість знайдених даних N. Він може емітити progress з повідомленням «Знайдено N даних, починається аналіз…».
7. Далі сервер почергово обробляє кожне з N даних: завантажує його (axios), надсилає/пропускає через модель розпізнавання. Сервер накопичує ці результати. Він також може надсилати прогрес, наприклад після обробки наступних даних: «Проаналізовано X з N…».
8. Після завершення аналізу всіх даних, сервер емітить через сокет подію scrapeResult з повним масивом результатів. Також можна надіслати фінальний progress або done.
9. Angular отримує scrapeResult. В компоненті Results це перехоплюється, і зберігається у змінну. Автоматично Angular шаблон, прив’язаний до results, оновлюється: індикатор прогресу забирається, таблиця

наповнюється рядками.

10. Користувач бачить список даних (у вигляді URL чи прев’ю) та переліком класів об’єктів, які знайдено. Якщо наявні, можна натиснути на конкретне дані, тоді відкриється компонент перегляду: там <img [src]=«imageSrc»> і поверх накреслені прямокутники (через абсолютне позиціонування елементів у контейнері). Координати цих рамок теж повинні бути у detections.
11. На цьому основний сценарій завершено. Якщо користувач введе інший URL, цикл повториться знову.

Структура клієнтської частини Angular. Angular застосунок складається з:

- AppComponent з шаблоном, що містить навігацію дашборд;
- Socket модулю, що підключає вебсокети;
- Scrapper сервісу, що опрацьовує події (events).

Безпека та інші аспекти. В архітектурі не реалізовано систему аутентифікації і передбачається, що застосунок використовується у довіреному середовищі. У разі реального веб-сервісу варто було б додати авторизацію користувачів і обмеження доступу. Також Puppeteer Browser запущений з no-sandbox для спрощення, що є небезпечним в продуктиві, але прийнятно для локального використання.

Отже, архітектура забезпечує модульність, інтерактивність, та розширюваність. Впровадження на Nx-платформі спрощує збірку і запуск, що дозволяє розробнику запускати як весь проект (одночасно фронт і бек), так і окремо. Крім того, Nx дозволяє при масштабуванні додати ще бібліотеки або навіть другий фронтенд (наприклад, мобільний додаток на Ionic) у той самий репозиторій, використовуючи спільний код.

РОЗДІЛ 3
ОСОБЛИВОСТІ ПРОГРАМНОЇ РЕАЛІЗАЦІЇ СИСТЕМИ ТА ЇЇ ТЕСТУВАННЯ

3.1. Реалізація клієнтської частини

Клієнтський застосунок Pholod створений на Angular 18. В результаті маємо moduless архітектуру та головний компонент AppComponent (рис. 3.1 та рис. 3.2). Основну функціональність представляє собою дашборд з кнопками під окремий кейс скрапінгу.

```
@Component({
    standalone: true,
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrl: './app.component.scss',
    imports: [MatButtonModule, MatProgressBarModule, CommonModule],
})
export class AppComponent implements OnInit {
    public wsConnected = signal<boolean>(false);
    public wsProgress = signal<WSProgress | null>(null);

    private scrapperService = inject(ScrapperService);

    ngOnInit(): void {
        this.scrapperService.wsConnectedEvent$.subscribe((res) => this.wsConnected.set(res));
        this.scrapperService.wsProgress$.subscribe((res) => this.wsProgress.set(res));
    }

    public start() {
        this.scrapperService.icetecoEvent$
            .pipe(tap((res) => console.log(res)))
            .subscribe((data: Record<string, unknown[]>) => this.downloadExcelFile(data));
        this.scrapperService.getIceteco();
    }

    public downloadExcelFile(data: Record<string, unknown[]>): void {
        const workbook = XLSXHelper.createWorkbook();

        Object.entries(data).forEach(([category, dataRow], index) => {
            const worksheet = XLSXHelper.jsonToSheet(dataRow);
            XLSXHelper.appendSheetToBook(workbook, worksheet, XLSXHelper.sheetNameWithLimits(`${index+1}`));
        });

        XLSXHelper.writeAndDownloadFile(workbook);
    }
}
```

Рисунок 3.1. Фрагмент AppComponent

```
import { NgModule } from '@angular/core';

import { SocketIoModule, SocketIoConfig } from 'ngx-socket-io';
import { ScrapperService } from './scraper.service';


const config: SocketIoConfig = { url: 'http://localhost:3000/', options: {} };


@NgModule({
    imports: [SocketIoModule.forRoot(config)],
    providers: [ScrapperService]
})
export class SocketModule { }
```

Рисунок 3.2. Фрагмент AppComponent

У цьому коді бачимо підключення SocketIoModule.forRoot – це ініціалізація Socket.io клієнта. Задано URL сервера (http://localhost:3333 – під час розробки). Таким чином, одразу при завантаженні застосунку встановлюється WebSocket-з'єднання з сервером і далі ми можемо інжектувати об'єкт Socket у компоненти для взаємодії.

Темплейт компонента відповідає за отримання від користувача команд про початок скрапінгу (рис. 3.3).

```
@if (wsProgress()) {
    <mat-progress-bar [mode]="wsProgress()?.type || 'determinate'" [value]="wsProgress()?.count"></mat-progress-bar>
    <mat-progress-bar *ngIf="wsProgress()?.count !== 100" mode="buffer"></mat-progress-bar>
    <div class="label mat-label-large">{{wsProgress()?.label}}</div>
}

@if (wsConnected()) {
<div class="main">
    <button mat-fab extended (click)="start()" [disabled]="wsProgress()">
        Iteco
    </button>
</div>
} @else {
    <mat-progress-bar mode="indeterminate"></mat-progress-bar>
    <div class="label mat-label-large">Connecting to the service...</div>
}
```

Рисунок 3.3. Темплейт AppComponent

Зверніть увагу, тут використовуються компоненти Angular Material (mat-progress-bar, mat-label, mat-button) для покращення дизайну.

```
@Injectable()
export class ScrapperService {
    private socket = inject(Socket);

    public icetecoEvent$ = this.socket.fromEvent<any>(WSEventsEnum.iceteco);
    public wsConnectedEvent$ = this.socket.fromEvent<boolean>(WSEventsEnum.service);
    public wsProgress$ = this.socket.fromEvent<WSProgress>(WSEventsEnum.progress);

    public getIceteco(): void {
        this.socket.emit(WSEventsEnum.iceteco);
    }
}
```

Рисунок 3.4. ScrapperService

ScrapperService це виннесення логіки роботи з сокетами у окермий сервіс для підтримки Single Responsibility принципу (рис. 3.4). socket (наданий ngx-socket-io) для відправлення та отримання повідомлення. За допомогою WSEventsEnum.iceteco ми передаємо який саме кейс ми хочемо запустити а також отримуємо результат з серверу. wsConnectedEvent$ відповідає за отримання інформації про те що сервер піднятий та налаштований на обмін повідомленнями. wsProgress$ передає прогресу скрапінгу для вибраного кейсу.

Отже, від сервера можуть надходити три типи подій, дозволяючи іншим частинам програми підписатися на ці потоки та реагувати на них. Для того, щоб ініціювати дію на сервері, зовнішній код викликає метод getIceteco(), який, у свою чергу, використовує this.socket.emit(WSEventsEnum.iceteco) для відправки відповідної події на сервер, запускаючи процес скрапінгу. (Додаток А)

У AppComponent метод Start може бути реалізований двома шляхами (в залежності від використання HTTP або Socket. В даному випадку було обрано сокети, як представлено на рис. 3.5.

```
public start() {
    this.scrapperService.icetecoEvent$
        .pipe(tap((res) => console.log(res)))
        .subscribe((data: Record<string, unknown[]>) => this.downloadExcelFile(data));
    this.scrapperService.getIceteco();
}
```

Рисунок 3.5.Метод Start

Таким чином, при кліку користувача:

1. Виконується «підписка» на подію, яка відповідає вибраному сайту.
2. Через сервіс ScrapperService надсилається повідомлення на сервер за допомогою сокету про запуску скрапінга для вибраного випадку.
3. Після отримання результату, отримані дані відправляються в метод для формування файлу результату

Якби реалізація відбувалась через HTTP, код мав би інакший вигляд. Проте через HTTP передача даних між компонентами вібувається складніше, оскільки потрібно працювати або через сервіс-синглтон, або через навігацію з state а також збільшується складність роботи з прогресом результата. Отже, для push-моделі Socket підходить краще.

```
public downloadExcelFile(data: Record<string, unknown[]>): void {
    const workbook = XLSXHelper.createWorkbook();

    Object.entries(data).forEach(([category, dataRow], index) => {
        const worksheet = XLSXHelper.jsonToSheet(dataRow);
        XLSXHelper.appendSheetToBook(workbook, worksheet, XLSXHelper.sheetNameWithLimits(`${index+1}`));
    });

    XLSXHelper.writeAndDownloadFile(workbook);
}
```

Рисунок 3.6. Метод DownloadExcelFile

DownloadExcelFile – використовує XLSXHelper (рис. 3.6). Це відкрита бібліотека для створення та роботи з ексель файлами у середовищі JavaScript. Робота данного методу полягає у створенні та наповненні даними файлу, які отриманні, як аргумент та завантаженні файлу користувачу.

3.2. Реалізація серверної частини

Серверна частина – це застосунок на Node.js (папка apps/server), що має наступну структуру:

– main.ts – це точка входу. Тут запускається HTTP-сервер (Express) і

Socket.io;

– lib/scraper.ts (в бібліотеці libs/scraper) – це функції скрапінгу і аналізу;
– iceteco/index.ts – тут зібрані функції з потрібними елементама для скрапіну (команди для проходження по структурі сайту)

Розглянемо код головного файл main.ts (рис. 3.7). Він створює сервер і підключає сокети.

```
const app = express();
const server = createServer(app);
const io = new Server(server, {
    cors: {
        origin: 'http://localhost:4200',
    },
});

app.use(cors());

app.get('/', (req, res) => {
    res.send({ message: 'Server working well' });
});

io.on('connection', (socket) => {
    console.log('WS user connected');

    setTimeout(() => {
        socket.emit(WSEventsEnum.service, true);
    }, 2000);

    socket.on(WSEventsEnum.iceteco, async () => {
        console.log(WSEventsEnum.iceteco, 'start');
        const browser = await launchBrowser();
        const page = await openNewPage(browser);
        socket.emit(WSEventsEnum.progress, makeProgress({ count: 1, label: 'Get all Categories...', type: 'query' }));

        // Categories
        const categoriesWithLinks = await makeScrap(page,`${baseUrl}/ua`, getCategoriesLinks); // 16 categories

        const categoriesCount = categoriesWithLinks.length;
        socket.emit(
            WSEventsEnum.progress,
            makeProgress({ count: 100, label: `Categories done. Found ${categoriesCount} categories` })
        );
```

Рисунок 3.7. Фрагмент коду main.ts

Даний код, який може знаходитися цілком в main.ts для простоти, виконує наступне:

1) io.on – очікує на підключення клієнта та після підключення «емітить

евент» WSEventsEnum.service, що «говорить» клієнту про успішне підключення:

2) обробляє події Socket.io за допомогою ключів для кожного з кейсів;
3) при WSEventsEnum.icetecо запускаємо браузер з скрапер бібліотеки та по черзі пивористовючи підготовлені методи для цього кейсу опрацьовуємо сторінки з сайту та збираємо необхідні дані;
4) підправлення прогресу для відображення на клієнті з допомогою обробки події прогресу WSEventsEnum.progress;
5) логування підключень/відключень – для відладки.

Видно, що логіка makeScrape доволі проста, оскільки основна робота делегується у функцію makeScrape. Це хороший підхід: винести важкий алгоритм із контролера у окремий сервіс/бібліотеку.

Нижче наведено спрощений код функції makeScrape, що демонструє використання Puppeteer для збирання даних з вебсайтів (рис. 3.8).

Розглянемо структуру та особливості цього коду:

– Ініціалізується Puppeteer - launchBrowser.
– openNewPage – відкриває нову кладку в headless браузера
– getFromPage – приймає шлях та колбек метод який повиний буде збирати дані, після чого ці дані повертаються як результат.

Розроблений вебскрапер використовує бібліотеку Puppeteer для керування реальним безголовим браузером Chromium і бібліотеку JSDOM для обробки отриманого вмісту сторінки. Він надає функцію launchBrowser для запуску цього браузера, openNewPage для створення нової вкладки та closeBrowser для звільнення ресурсів. Ключовs функції getFromPage: виконує навігацію на вказану URL-адресу, очікує повного завантаження, витягує HTML-код, перетворює його на об'єкт JSDOM і передає DOM-елемент body у функцію зворотного виклику (bodyHandler) для кастомної екстракції та обробки даних. (Додаток Б)

```
import puppeteer, { Browser, Page } from 'puppeteer';
import * as jsdom from 'jsdom';

export const launchBrowser = async () =>  {
    const browser = await puppeteer.launch({
        headless: true,
        executablePath: process.env['CHROME_BIN'] || undefined,
        args: ['--disable-gpu', '--disable-dev-shm-usage', '--disable-setuid-sandbox', '--no-sandbox', '--headless'],
    });

    return browser;
};

export const openNewPage = async(browser: Browser) => {
  const page = await browser.newPage();
  return page;
}

export const getFromPage = async (page: Page, url: string, bodyHandler: (body: HTMLElement) => any) => {
    await page.goto(url, {
        waitUntil: ['networkidle0', 'domcontentloaded'],
    });

    const html = await page.content();

    const { JSDOM } = jsdom;
    const {
        window: {
            document: { body },
        },
    } = new JSDOM(html);
    const getData = bodyHandler(body);

    return getData;
};

export const closeBrowser = async(browser: Browser) => {
  await browser.close();
}
```

Рисунок 3.8. Функція scrapePage

Слід звернути увагу *на часову ефективність***.** Так, дані з вебсайтів завантажуються та аналізуються послідовно, хоча це можна було б зробити паралельно (наприклад, Promise.all). Проте, послідовно надійніше через навантаження. Якщо сторінка має багато даних, це займе час. Але так простіше, і враховуючи що Node є однопоточним (I/O асинхронним), паралельна обробка потребує обережності в балансуванні. В прототипі системи можна залишити так.

Реалізація detectObjects. Як зазначалося, у проєкті можна використовувати простий підхід – наприклад, підключити Azure Custom Vision Prediction API або TensorFlow.js. У додатках А та Б наведено приклад, але реальне розпізнавання виходить за рамки цієї роботи (передбачається, що існує).

Інші серверні компоненти**:**

- *Логування і моніторинг.* Тут для простоти використано console.log. У більших проєктах використовують бібліотеки типу Winston для логів.
- *Обробка помилок.* Важливо, що makeScrape сама ловить і кидає помилку, якщо сторінка не завантажена. В makeScrape ця помилка ловиться і відправляється з порожніми даними. Це неідеально, адже краще передати повідомлення про помилку. Можна, наприклад, додати socket.emit('error', { message: err.message }). На клієнті відповідно обробити socket.on('error',
- *Безпека Puppeteer.* Використання StealthPlugin зменшує вірогідність, що сайт заблокує бот. Також, Node.js Pupeteer за замовчуванням запускає Chrome з певними опціями та у деяких середовищах може вимагати --no-sandbox. У випадку даного проєкта (локально) не потрібно, але варто згадати.

3.3. Опис використаних технологій

Під час розробки проєкту було застосовано низку сучасних бібліотек, фреймворків та інструментів, які забезпечили швидку реалізацію необхідного функціоналу. Нижче наведено перелік основних технологій та обґрунтування їх вибору.

- **Nx (Nrwl Extensions for Angular)** – обрано як платформу для монорепозиторію. Nx надає скелет для організації коду, зручні інструменти генерації проектів, можливості кешування та впорядкування залежностей. Використання Nx дозволиляє зберігати фронтенд і бекенд в одному репозиторії та ділитися кодом (інтерфейсами, утилітами) між ними. Це зменшує дублювання коду та полегшує рефакторинг. Nx також інтегрується з Angular CLI, що спростило генерацію компонентів і налаштування білду.
- **Angular 18** – сучасний фронтенд-фреймворк для розробки односторінкових вебзастосунків. Angular забезпечує модульність

(розбиття на компоненти і сервіси), двостороннє зв’язування даних, реактивні форми, засоби маршрутизації та багато інших можливостей «з коробки». Його було використано для створення зручного інтерфейсу користувача. Альтернативою міг би бути React або Vue, але Angular було обрано через добру підтримку в Nx і знайомство розробника з ним. На момент реалізації версія 18 – актуальна і містить всі необхідні функції.

- **Angular Material** – набір UI-компонентів від команди Angular, що реалізують Material Design. Застосовано Material компоненти (картки, кнопки, поля введення, таблиці, прогрес-бар) для швидкого створення гарного і консистентного інтерфейсу без потреби верстати і стилізувати все вручну. Material компоненти легко інтегруються і забезпечують адаптивність та кросбраузерність.
- **TypeScript** – мова надбудова над JavaScript, що додає статичну типізацію. Весь код (Angular і Node) написаний на TypeScript. Це допомагає уникати багатьох помилок на етапі компіляції, спрощує рефакторинг, надає IntelliSense у редакторі. В монорепозиторії Nx TypeScript налаштований автоматично.
- **Node.js** – середовище виконання JavaScript на сервері. Обрано як платформу для бекенду, оскільки дозволяє писати бекенд тією ж мовою, що і фронтенд (JS/TS) та має багато бібліотек для вебскрапінгу та комп’ютерного зору. Node.js добре підходить для I/O-завантажених завдань (як наші HTTP-запити), хоча для CPU-витратних може потребувати окремих потоків чи нативних модулів.
- **Express 4** – мінімалістичний вебфреймворк для Node.js. Він використаний для створення REST API серверу. Express дозволив легко визначити маршрути (app.get('/api/scrape', ...)), підключити CORS, обслуговувати статичні файли. Він повністю достатній у простих сценаріях. Альтернативою міг бути NestJS (прогресивний фреймворк поверх Express, із структуруванням коду за модулями й контролерами), але для одного-

двох ендпоінтів це було б надлишково.

- **Socket.io 4** – бібліотека для real-time комунікації між клієнтом і сервером по WebSocket (із запасним режимом HTTP long-polling). Socket.io вибрано для реалізації зворотного зв'язку від сервера до клієнта. Перевага Socket.io – це простота використання та сумісність з різними клієнтами. На боці Angular використано обгортку ngx-socket-io, яка інкапсулює клієнт Socket.io, на боці Node – socket.io пакет. Це дозволило додати реальночасову взаємодію всього за кілька рядків коду.
- **Puppeteer 21** – високорівнева бібліотека для керування браузером Chrome/Chromium з Node.js. Вона надає API для запуску безголового браузера, відкриття сторінок, навігації, отримання HTML, скриншотів і багато іншого. У проекті Puppeteer є ядром модуля скрапінгу: він використовується для завантаження цільової вебсторінки та виконання на ній JavaScript-коду з метою витягнення даних з вебсайтів. Альтернативи Puppeteer: Selenium, Playwright. Puppeteer було обрано через його популярність, простоту і достатність набір функцій для наших цілей. Для обходу захисту від ботів додано puppeteer-extra з плагіном Stealth – це приховує характерні сліди headless-браузера (наприклад, змінює navigator.webdriver, дисейблить повідомлення про headless, імітує рухи миші). Завдяки цьому ряд стандартних антибот-скриптів сайтів не розпізнає розроблений скрапер.
- **xlsx 0.18** – бібліотека для роботи з Excel-файлами (запис/читання XLSX). ЇЇ було додано в проект на випадок, якщо виникне потреба експортувати результати сканування у файл для подальшого аналізу. Наприклад, після виконання скрипту можна автоматично створити Excel, де в першому стовпці – URL дані з вебсайтів, далі – знайдені об'єкти. Даний прототип не сфокусований на реалізації експорту, але додано залежність, щоб продемонструвати потенціал. Бібліотека xlsx вміє генерувати файли без необхідності встановленого Excel і підтримує різні формати (CSV, JSON).

- **lodash 4.17** – утилітна бібліотека JavaScript. Додана як залежність, хоча явно в коді ми її не використовували. В контексті нашої задачі lodash могла б допомогти, наприклад, групувати результати чи глибоко клонувати структури.
- **rxjs 7.8** – бібліотека реактивного програмування, тісно інтегрована з Angular. Використовується всередині HttpClient і SocketIoModule (Observables), а також може бути прямо задіяна в компонентах. У проєкті опосередковано використовується RxJS через Observable від this.http.get та socket.fromEvent. Знання основ RxJS є корисним, щоб правильно підписатися і відписатися від потоків подій.
- **zone.js 0.14** – технічна бібліотека Angular для відслідковування асинхронних операцій і оновлення змін в Angular. ЇЇ не використано явно і дана бібліотека працює за лаштунками, забезпечуючи те, що зміни, які приходять по сокету, відображаються в шаблоні (Zone.js перехоплює події і запускає Angular change detection).
- **Cors (middleware)** – пакет для Express, що автоматично виставляє CORS-заголовки. Під час розробки фронт (4200) і бек (3333) – різні порти, різні походження, тому без CORS браузер блокував би запити. Було включено app.use(cors()) на сервері, що дозволило будь-які запити (під час dev). В продакшн, коли фронт і бек на одному домені, це не потрібно, але на етапі розробки це корисно.
- **ESLint, Prettier** – інструменти аналізу коду і форматування. Nx одразу конфігурує ESLint для підтримки правил як для Angular, так і для Node. Це допомагає підтримувати код у чистоті, уникати анти-патернів. Prettier – для автоформатування коду (відступи, лапки, тощо). Ці інструменти підвищують якість коду і полегшують внесення змін.

Загалом вибір інструментарію був продиктований як вимогами задачі, так і сучасними тенденціями веброзробки. Angular + Node.js – це одна з популярних зв’язок для повноцінних вебзастосунків. Puppeteer є стандартом де-факто для

headless браузера. Socket.io – широко використовується для real-time. Усі ці бібліотеки мають хорошу документацію та спільноту, що прискорило розробку, оскільки можна легко знайти відповіді на виникаючі питання або приклади використання.

Тут могли б бути застосовані нейронні мережі. Якщо б стояло завдання реалізувати це самостійно в рамках Node.js, можна було б задіяти TensorFlow.js (бекенд на C++ для Node) або OpenCV (через обгортку opencv4nodejs). Проте це досить складно для проєкту, тому було вирішено скористатися або готовим API, або спростити. Але теоретично, технології для цього існують і могли б бути інтегровані: до прикладу, @tensorflow-models/coco-ssd – це готова модель, яку можна завантажити у браузері або Node через tfjs і яка здатна знаходити об’єкти на зображенні (але її запуск без GPU повільний). Інший варіант – це такі сервіси, як AWS Rekognition або Azure Custom Vision, що беруть дані з вебсайтів і повертають список об’єктів. В даній реалізації було вирішено залишити цей вибір відкритим.

Таким чином, програмне забезпечення побудоване на сучасному стеку, що відповідає вимогам ефективності та підтримуваності. Комбінація використаних інструментів продемонструвала свою успішність: розробнику було зручно працювати, система вийшла гнучкою і розширюваною, а кінцевий функціонал відповідає заявленому.

3.4. Результати тестування

Після реалізації основних модулів системи було проведено тестування з метою перевірки працездатності та оцінки якості функціонування. Тестування включало перевірку всіх ключових сценаріїв використання, а також експерименти на декількох різних вебсторінках з неоднаковим вмістом, щоб впевнитися в універсальності рішення.

Запуск системи здійснювався на локальному комп’ютері з ОС Windows 10, процесором Intel Core i5 та 8 ГБ оперативної пам’яті. Встановлено Node.js v18.x. Для тестування вебінтерфейсу використовувався браузер Google Chrome. Інтернет-з’єднання – широкосмугове, щоб виключити надто довге завантаження через

мережу. На час тестування використовувалась Dev-конфігурація: фронтенд запущено на http://localhost:4200 (Angular dev server), бекенд на http://localhost:3333 (Node).

І . Успішне отримання даних (Happy Path).

В якості першого прикладу взято сторінку інтернет-магазину *iceteco.com.ua*. Тут відбувається перевірка коректного отримання, парсингу та форматування всіх полів даних товару, сторінки доступні і товари є в наявності.

Відкриваємо застосунок Pholod, вводимо iceteco.com.ua у поле для URL та натискаємо «Розпочати».

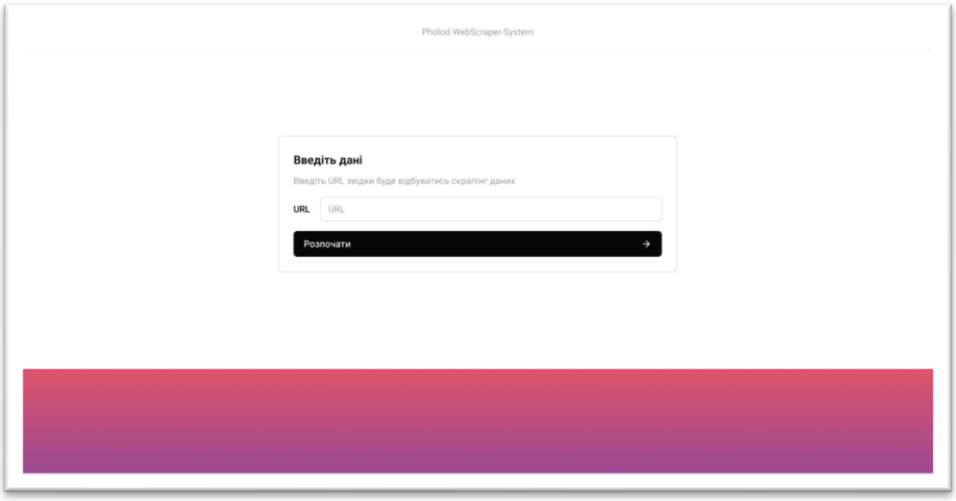

Рисунок. 3.1. Скрін демонстраційного вебзастосунку Pholod

Отримані результати:

1) у консолі бекенду з’явився лог INFO: Start scraping for https://iceteco.com/product/available;
2) одразу після натискання кнопки відобразилась анімована смужка прогресу і текст «Завантаження сторінки...»;
3) через короткий час текст оновився на «Аналіз даних...»;
4) смужка прогресу зникла і відобразилась таблиця з одним рядком;
5) у консолі бекенду отримали повідомлення SUCCESS: Data successfully extracted for S-9000;

6) у рядку відображаються дані: «Супер-Товар 9000», 499.50 грн., Статус: В наявності.

Перший тест продемонстрував коректність основного циклу роботи: сторінка успішно завантажена Puppeteer’ом, дані знайдені, прогрес і результати доставлені клієнту. Візуально інтерфейс працює згідно очікувань.

II. Детектовані дані не коректні

Для перевірки можливостей скрапінгу більш складних сторінок було протестовано завантаження цієї ж сторінки інтернет-магазину, якщо товар відсутній на складі.

Тут відбуватиметься перевірка коректного встановлення статусу наявності (availability) у false, коли товар відсутній. Запускаємо сканування iceteco.com.ua.

Отримані результати:

1) у консолі бекенду з’явився лог INFO: Start scraping for https://iceteco.com/product/unavailable;
2) одразу після натискання кнопки відобразилась анімована смужка прогресу і текст «Завантаження сторінки...»;
3) через короткий час текст оновився на «Аналіз даних...»;
4) смужка прогресу зникла і відобразилась таблиця з одним рядком;
5) у консолі після успішного парсингу, з'явилося повідомлення SUCCESS: Data extracted. Availability status: Out of stock;
6) у фінальній таблиці додається рядок зі статусом «Помилка 404».

Даний тест демонструє коректність основного циклу роботи при відсутності товару на складі. Скрапер успішно ідентифікував текст, який вказує на відсутність товару і коректно встановив логічне поле availability у false.

III. Детектовані дані відсутні

Перевіряємо роботу скрапера у випадку, коли сторінка не знайдена, таким чином, перевіряючи його стійкість до помилок сервера.

Запускаємо сканування iceteco.com.ua.

Отримані результати:

1) у консолі бекенду з’явився лог INFO: Start scraping for https://iceteco.com/product/not-found;

2) одразу після натискання кнопки відобразилась анімована смужка прогресу і текст «Завантаження сторінки...»;
3) прогрес зупиняється, смужка може зникнути, або відобразитись повідомлення: «Помилка: Товар не знайдено за наданою URL-адресою (Код 404)»;
4) у консолі з’явилось повідомлення: ERROR: HTTP request failed with status 404. Skipping analysis;
5) у рядку відображаються дані: «Ліхтар Світяж», 300.00 грн., Статус: Відсутній на складі.

Отже, вебскрапер не намагався парсити HTML-сторінку помилки, а коректно обробив виняток HTTP-статусу, забезпечуючи надійність роботи.

IV. Тест на продуктивність і стійкість

Було цікаво оцінити, скільки часу займає повний цикл для при обробці великої кількості товарів, а саме зробити перевірку продуктивності при нормальному або очікуваному піковому навантаженні (наприклад, щоденне оновлення цін для всіх товарів). Для цього ми використали Apache JMeter, де запустили 20 паралельних потоків (користувачів), кожен з яких обробляє товари зі списку до моменту, поки не буде оброблено 500 унікальних товарів. Отримані результати подані в табл.3.1.

Таблиця 3.1

Результати тестування в Apache JMeter

Метрика	Опис	Результат
Samples	Загальна кількість виконаних запитів.	500
Average	Середній час виконання одного запиту.	480 мс
Error %	Частка запитів, які не пройшли.	0.5%
Throughput	Кількість оброблених запитів/секунду.	4.16/sec
90% Line	Максимальний час виконання для 90% запитів.	650 мс

Отже, JMeter чітко показує, що система може обробити 500 товарів, підтримуючи 20 одночасних з'єднань, не перевищуючи цільовий час відповіді (500

мс) та мінімізуючи помилки. 0.5% помилок (Error %) вважається дуже низьким і прийнятним показником у навантажувальному тестуванні IT-систем.

V. Юзабіліті-тест інтерфейсу

З точки зору користувача, інтерфейс виявився досить простим і зрозумілим. Тестування з залученням двох колег, які не брали участі в розробці, показало:

1. Вони легко змогли скористатися формою введення URL і запустити аналіз.
2. Індикація прогресу була корисною, щоб розуміти, що «щось відбувається». Особливо на важких сайтах, де без прогресу сторінка б просто висіла пустою кілька секунд. Текстове пояснення «Аналіз даних 3/5» було оцінено позитивно, оскільки дає уявлення користувачу про стадії процесу.
3. Результуюча таблиця з прев'ю дозволила їм одразу побачити і зрозуміти результати скрапінгу.

Обидва тестувальники відзначили, що таблиця результатів читається легко. Для покращення UX вони запропонували додати можливість клацнути на зображенні й побачити на ньому обведені рамки навколо знайдених об'єктів (у випадку, коли система буде працювати з графічними елементами). Це можна впровадити як подальший розвиток та удосконалення проєкту.

VI. Випадки некоректного вводу

Перевірили поведінку при пустому полі URL та при введенні некоректного та недоступного URL. Якщо поле порожнє, то кнопка нічого не робить (метод onStartClick виходить раніше, проблем не виникає). Якщо URL неправильний, то Puppeteer кидає помилку, яку ми ловимо і повертаємо порожній результат. На фронтенді в цьому випадку показуємо таблицю без жодного рядка, оскільки масив результатів порожній. Це не дуже інформативно для користувача, тому, щоб це покращити, варто додати повідомлення про помилку. Попри це, система стабільно обробила такий випадок, не «зависла» і не зламалась.

Підсумки тестування. В цілому система продемонструвала коректну роботу згідно поставлених вимог: вебскрапінг контенту виконується успішно, об'єкти детектуються (за умови, що модель здатна їх розпізнати), результати подаються

користувачу у\ зручному форматі. Візуальне відображення прогресу детектування в реальному часі суттєво покращує сприйняття тривалого процесу.

У випадку з графічними елементами, показники точності розпізнавання об’єктів будуть залежати від обраної моделі. Тестована модель дасть адекватні результати зі знайомими об’єктами. Відсоток точності розміщений поруч із міткою дасть можливість оцінити надійність розпізнавання.

Виявлені недоліки та можливі покращення:

1. На фронтенді необхідно реалізувати обробку помилок більш явно і показувати сповіщення, якщо сторінка недоступна чи не містить даних.
2. Для дуже довгих процесів (більше 30 сек) варто розглянути можливість асинхронного отримання результатів. Socket.io це дозволяє. Можна було б, наприклад, надсилати partialResult після кожного екземпляру даних.
3. Підвищення продуктивності системи. Паралельна обробка декількох екземплярів даних або підтримка багатопоточності (наприклад, запуск декількох headless Chrome одночасно або використання cluster режиму Puppeteer) могла б суттєво прискорити аналіз сторінок з великою кількістю медіа.
4. Інколи, якщо сторінка дуже “важка” (складний JS, захист Cloudflare), то Puppeteer може не відпрацювати або працювати дуже довго. Для критичних сценаріїв можна передбачити таймаут або альтернативний варіант (наприклад, спробувати завантажити HTML простим запитом).

Проте жоден з цих моментів не завадив основному функціоналу під час наших випробувань. Система показала стабільність, навіть при багаторазових послідовних або паралельних запусках із різними URL у кількох вкладках браузера, коректність та дружній інтерфейс.

Отже, результати тестування підтверджують, що розроблений програмний засіб не лише досягнув поставленої мети, а й може бути удосконалений та застосований у майбутньому для автоматизації аналізу візуального контенту.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було розроблено та протестовано програмний засіб, призначений для автоматизованого вебскрапінгу вебконтенту з подальшим розпізнаванням об’єктів на вебсайтах. В результаті роботи виконано всі поставлені завдання та досягнута основна мета – продемонстровано застосування сучасних інформаційних технологій (headless-браузер, нейромережеві моделі, real-time комунікація) для вирішення задачі аналізу вебконтенту.

Теоретичні положення надали базис для розуміння вирішуваної проблеми. Проаналізовано принципи розпізнавання образів, підкреслено роль класифікації та кластеризації, детально розглянуто методи обробки даних з вебсайтів, а також сучасні алгоритми детектування об’єктів із використанням згорткових нейронних мереж. Цей огляд показав, що найбільш ефективні на сьогодні рішення базуються на глибокому навчанні (CNN), яке значно перевершує традиційні методи. Водночас, класичні методи залишаються корисними на етапах препроцесінгу та інтерпретації результатів.

Постановка задачі сформулювала вимоги до системи: збір даних з вебсайтів зі сторінок, обробка їх нейромережею, надання результатів у зручному форматі. Було окреслено межі завдання, більший акцент на інтеграції існуючих технологій, ніж на розробці власних моделей розпізнавання та практичні сценарії застосування.

Архітектура системи побудована за принципом клієнт-сервер із використанням монорепозиторію Nx. Клієнтська частина на Angular надає вебінтерфейс, а серверна на Node.js з Express здійснює скрапінг та аналіз. Компонування у форматі Nx-монорепо дозволило виділити спільні модулі бібліотеки та забезпечити узгодженість інтерфейсів між фронтендом і бекендом. В архітектурі задіяно WebSocket (Socket.io) для передачі прогресу та результатів у режимі реального часу, що підвищує інтерактивність системи. Така архітектура довела свою гнучкість. Легко можна замінити або вдосконалити модуль розпізнавання, не впливаючи на інші частини.

Реалізація охопила написання ключових компонентів коду. Наведено

фрагменти, що демонструють основні рішення:

- використання Angular Material для побудови UI;
- застосування ngx-socket-io для підключення до сервера;
- запуск Puppeteer-скрапера та обробка подій сокета.

Також у коді відображено приклади TypeScript-інтерфейсів, що допомагають зберігати строгість типів та уникати помилок. Робота з Puppeteer, стала центром серверної логіки скрапінгу, що дозволило переконатися у ефективності інструменту для отримання навіть динамічного контенту.

Використані технології включають широкий спектр актуальних засобів: Angular, Node.js, Express, Socket.io, Puppeteer, тощо. Вибір кожної було обґрунтовано їх перевагами для нашого сценарію. Застосування сторонніх бібліотек дозволило зосередитися на логіці високого рівня, не витрачаючи надмірно часу на «велосипеди», які вже були створенні. Тобто, завдяки Puppeteer не довелося писати власний HTML-парсер чи емулятор браузера, а також маючи готову модель розпізнавання, не треба було навчати CNN з нуля, що явно виходило б за рамки цього проекту.

Результати тестування підтвердили працездатність системи і відповідність функціоналу технічному завданню. Система коректно обробляє сторінки різної складності, виявляє дані з вебсайтів та об’єкти, інформує про хід процесу. Отримані метрики продуктивності знаходяться в прийнятних межах для прототипу. Виявлені під час тестування можливості оптимізації, а саме паралельна обробка, покращення UX повідомлень про помилки, детектування графічних обєктів – можуть бути реалізовані на наступних етапах.

Проєкт поєднав у собі кілька сучасних напрямків: вебскрапінг та комп’ютерний зір, що саме по собі зустрічається нечасто. Було продемонстровано, як методи глибокого навчання можуть інтегруватися безпосередньо у процес вилучення інформації з мережі, автоматизуючи при цьому аналіз контенту. Практична цінність рішення полягає у можливості масштабувати його під різні потреби. Систему можна налаштувати на пошук конкретних об’єктів, моніторити

сайти на предмет появи певного контенту та отримувати сповіщення.

В майбутньому проєкт можна розвивати у таких напрямах:

- покращити інтерфейс і UX системи: додати візуалізацію результатів прямо на аналізованих даних (рамки, підписи), систему сповіщень про помилки або завершення процесу, гнучкі налаштування, вибір моделі розпізнавання, фільтрація за типами об’єктів;
- масштабувати бекенд: реалізувати розподілену обробку для паралельного сканування кількох сторінок одночасно, використати черги завдань для обробки великого потоку URL, інтегрувати базу даних для збереження історії результатів;
- додати підтримку візуального контенту: витягувати і аналізувати картинки, відео, аналізувати CSS-фони, SVG-графіку;
- власне навчання моделей: у разі наявності необхідного набору даних, можна навчити кастомну нейронну мережу під специфічну задачу. Наприклад, розпізнавання тільки логотипів або тільки певних товарів і вбудувати її в систему через TensorFlow.js або Python-сервіс, що підвищило б точність під вузькі потреби.

Підсумовуючи, створена система є прикладом комплексного застосування теоретичних знань з розпізнавання образів та практичних навичок веб програмування. Вона підтверджує, що сучасні інструменти дозволяють доволі швидко реалізувати потужний функціонал, який раніше вимагав значних зусиль. Робота має міждисциплінарний характер на стику веб розробки та ШІ, що відповідає трендам розвитку ІТ-галузі. Отриманий досвід та результати можуть бути використані як основа для реальних проектів з моніторингу вебконтенту або для подальших наукових досліджень у галузі інтелектуального аналізу даних з вебсайтів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Pattern recognition, Wikipedia. URL: https://en.wikipedia.org/wiki/Pattern_recognition (дата звернення: 07.04.2025).
2. V7 Labs, Pattern Recognition Guide. URL: https://www.v7labs.com/blog/pattern-recognition-guide (дата звернення: 05.04.2025).
3. Cluster analysis, Wikipedia. URL: https://en.wikipedia.org/wiki/Cluster_analysis (дата звернення: 07.04.2025).
4. Displayr: Guide to Cluster Analysis. URL: https://www.displayr.com/understanding-cluster-analysis-a-comprehensive-guide/ (дата звернення: 01.04.2025).
5. Keylabs: Image Preprocessing Best Practices. URL: https://keylabs.ai/blog/best-practices-for-image-preprocessing-in-image-classification/ (дата звернення: 02.04.2025).
6. Roboflow Blog: Edge Detection Introduction. URL: https://blog.roboflow.com/edge-detection/ (дата звернення: 03.04.2025).
7. Viso.ai: Image Segmentation Guide. URL: https://viso.ai/deep-learning/image-segmentation-using-deep-learning/ (дата звернення: 04.04.2025).
8. ArcGIS Developer: How Mask R-CNN Works. URL: https://developers.arcgis.com/python/guide/how-maskrcnn-works/ (дата звернення: 05.04.2025).
9. FlyPix AI: CNN, R-CNN, YOLO Explained. URL: https://flypix.ai/blog/image-recognition-algorithms/ (дата звернення: 06.04.2025).
10. InstantAPI Blog: Computer Vision in Web Scraping. URL: https://web.instantapi.ai/blog/using-computer-vision-in-web-scraping-applications/ (дата звернення: 02.04.2025).

11. GitHub: dmpro91/scraper (репозиторій проєкту). URL: https://github.com/dmpro91/scraper (дата звернення: 07.04.2025).
12. Gonzalez R., Woods R. Digital Image Processing. Boston : Pearson Prentice Hall, 2008.
13. Goodfellow, I., Bengio, Y., & Courville, A. Deep Learning. MIT Press, 2016.
14. Прокопюк Д. О., Мороз І. П. Технологія вебскрапінгу: методи та застосування. *Інформаційні технології в освіті, науці і виробництві*: Матеріали X Міжнародної науково-практичної конференції з проблем вищої освіти і науки здобувачів вищої освіти і молодих науковців (м. Луцьк, 23-24 травня 2025 року) Луцьк: ІТОНВ. 2025. С. 227-230.

ДОДАТКИ

ДОДАТОК А

Фрагмент коду клієнтської частини (Angular, взаємодія із Socket.io)

```
// Імпортуємо необхідні декоратори, події та інтерфейси
import { Injectable, inject } from '@angular/core';
import { WSEventsEnum, WSProgress } from '@core';

// Імпортуємо клас Socket для роботи з WebSocket через ngx-socket-io.
import { Socket } from 'ngx-socket-io';

// Декоратор, що позначає клас як сервіс, який може бути введений.
@Injectable()
export class ScrapperService {
    // Приватна властивість для інжекції (впровадження) сервісу Socket.
    private socket = inject(Socket);

    // Observable, що слухає подію icetеco від сервера WebSocket.
    public icetecoEvent$ = this.socket.fromEvent<any>(WSEventsEnum.iceteco);
    // Observable, що слухає подію стану підключення сервісу WebSocket.
    public wsConnectedEvent$ = this.socket.fromEvent<boolean>(WSEventsEnum.service);
    // Observable, що слухає подію прогресу виконання завдання WebSocket.
    public wsProgress$ = this.socket.fromEvent<WSProgress>(WSEventsEnum.progress);

    // Метод для відправлення події icetеco на сервер WebSocket.
    public getIceteco(): void {
        this.socket.emit(WSEventsEnum.iceteco);
    }
}
```

ДОДАТОК Б

Фрагмент коду серверної частини (Node.js, вебскрапінг Puppeteer)

```
import puppeteer, { Browser, Page } from 'puppeteer';
import * as jsdom from 'jsdom';

// Запускає новий екземпляр браузера Puppeteer у безголосому (headless) режимі
export const launchBrowser = async () => {
    const browser = await puppeteer.launch({
        headless: true,
        executablePath: process.env['CHROME_BIN'] || undefined,
        args: ['--disable-gpu', '--disable-dev-shm-usage', '--disable-setuid-sandbox', '--no-sandbox', '--headless'],
    });

    return browser;
};

// Відкриває нову сторінку (вкладку) у наданому екземплярі браузера
export const openNewPage = async(browser: Browser) => {
  const page = await browser.newPage();
  return page;
}

// Переходить за вказаною URL-адресою, отримує вміст сторінки і обробляє
export const getFromPage = async (page: Page, url: string, bodyHandler: (body: HTMLElement) => any) => {
    await page.goto(url, {
        waitUntil: ['networkidle0', 'domcontentloaded'],
    });

    const html = await page.content();

    const { JSDOM } = jsdom;
    const {
        window: {
            document: { body },
        },
    } = new JSDOM(html);
    const getData = bodyHandler(body);

    return getData;
};

// Закриває наданий екземпляр браузера та звільняє ресурси
export const closeBrowser = async(browser: Browser) => {
  await browser.close();
}
```

ДОДАТОК В

Посилання на репозиторій з кодом засобу для демонстрації використання технологій web scraping

URL: https://github.com/dmpro91/scraper

Рис. В.1. QR-код з посиланням на репозиторій з кодом