МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

ЗАГАЄЦЬКА АННА МИКОЛАЇВНА

Допускається до захисту:
в.о. завідувача кафедри
інформаційних технологій
канд. техн. наук, доцент
_____ О. В. Зелінська
«____»_____ 20__ р.

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ АВТОМАТИЧНОЇ ПЕРЕВІРКИ ТЕКСТІВ НА СХОЖІСТЬ

Спеціальність 122 Комп’ютерні науки

Кваліфікаційна (бакалаврська) робота

Керівник:
О. В. Зелінська, доцент кафедри
інформаційних технологій,
к. т. н., доцент
_____

Оцінка: _____ / _____ / _____
(бали/за шкалою ЄКТС/за національною шкалою)
Голова ЕК: _____

Вінниця – 2025

АНОТАЦІЯ

Загаєцька А.М. Програмне забезпечення для автоматичної перевірки текстів на схожість. Спеціальність 122 «Комп’ютерні науки», освітня програма «Комп’ютерні науки». Донецький національний університет імені Василя Стуса, Вінниця, 2025.

Кваліфікаційна (бакалаврська) робота присвячена розробці програмного забезпечення для автоматичного виявлення текстової схожості. У роботі проаналізовано сучасні алгоритми перевірки унікальності документів (TF-IDF, cosine similarity, нейромережі) та визначено їх ефективність у різних контекстах. Запропоноване програмне рішення реалізовано у вигляді вебдодатку з використанням HTML, CSS, JavaScript і бібліотек React. Додаток містить інтуїтивно зрозумілий інтерфейс із підтримкою світлої/темної теми та можливістю роботи з форматами DOCX, PDF, TXT. Виконано тестування точності алгоритмів і юзабіліті системи, а також розроблено методи візуалізації результатів для зручного аналізу.

Ключові слова: перевірка текстів, текстова схожість, алгоритми обробки тексту, TF-IDF, cosine similarity, UX/UI-дизайн, HTML, CSS, JavaScript, React.

77 стор., 20 рис., 2 дод., 40 джерел.

ABSTRACT

Zahaietska A.M. Software for Automatic Text Similarity Detection. Specialty 122 «Computer Science», educational program «Computer Science». Vasyl' Stus Donetsk National University, Vinnytsia, 2025.

The bachelor's thesis is dedicated to the development of software for automatic text similarity detection. The study analyzes modern algorithms for text uniqueness verification (TF-IDF, cosine similarity, neural networks) and evaluates their efficiency in various contexts. The proposed software solution is implemented as a web application using HTML, CSS, JavaScript, and React libraries. The application features an intuitive user interface with support for light/dark themes and the ability to process DOCX, PDF, and TXT formats. Accuracy testing of the algorithms and usability evaluation of the system were conducted, as well as methods for result visualization to facilitate analysis.

Keywords: text verification, text similarity, text processing algorithms, TF-IDF, cosine similarity, UX/UI design, HTML, CSS, JavaScript, React.

77 pages, 20 figures, 2 appendices, 40 references.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ І ТЕРМІНІВ

ПЗ - програмне забезпечення

IDE (Integrated development environment) - інтегроване середовище розробки

IDF (Inverse Document Frequency) - міра рідкості слова в наборі документів

IR (Information retrieval) - інформаційний пошук

NLP - обробка природної мови

TF (Term Frequency) - кількість згадок певного слова в тексті.

TF-IDF (Term Frequency-Inverse Document Frequency) - статистичний показник, що використовується для оцінки важливості слів у документі, що є частиною колекції корпусу.

ВСТУП

У сучасному інформаційному просторі текстовий контент залишається головним носієм знань, ідей та комунікацій. Щодня ми працюємо з ним - у навчанні, роботі, повсякденному житті. Однак доступність цифрових ресурсів створює ризики, зокрема поширення плагіату й копіювання чужих матеріалів без належного посилання. Це підкреслює потребу у надійних засобах контролю авторства. У багатьох закладах вищої освіти такі перевірки вже стали обов’язковими під час захисту академічних робіт.

Ефективність подібного інструменту залежить не лише від точності виявлення збігів, а й від простоти його використання. Складні у користуванні програми нерідко залишаються поза увагою, навіть якщо демонструють високі технічні показники. Тому підтримка різних тем оформлення (світлої та темної), а також сумісність із форматами DOCX, PDF, TXT розглядаються як необхідні функції.

Метою дослідження є створення програмного продукту для виявлення текстової схожості, який об'єднує сучасні алгоритми й інтуїтивно зрозумілий інтерфейс. Адаптивний дизайн сприятиме зменшенню візуального навантаження, що особливо важливо для користувачів, які працюють із великими обсягами даних.

Об’єктом дослідження є процеси створення програмного забезпечення для автоматичної перевірки текстів на схожість

Предметом дослідження є методи, моделі, технології розробки програмного забезпечення та алгоритми виявлення схожих фрагментів.

Завдання роботи охоплюють: аналіз існуючих методів виявлення схожості; розробку модульної архітектури ПЗ; розробка адаптивного дизайну, реалізація функції імпорту з різних форматів файлів, а також тестування точності алгоритмів і зручності інтерфейсу. Наприклад, досвід роботи з тестовими користувачами показав, що чітка візуалізація результатів (відсоток схожості, кількість слів, графіки) значно полегшує сприйняття інформації.

Наукова новизна полягає в інтеграції класичних методів обробки текстів із принципами сучасного UX/UI-дизайну. Запропоновано реалізувати зміну тем безпосередньо в системі перевірки та забезпечити обробку популярних форматів без додаткової конвертації, що покращує користувацький досвід і заощаджує час.

РОЗДІЛ 1
АНАЛІЗ ПРОБЛЕМИ ПЕРЕВІРКИ ТЕКСТІВ НА СХОЖІСТЬ

1.1 Поняття та сутність перевірки текстів на схожість

Виявлення подібності є процесом оцінювання ступеня схожості між двома або більше документами, що має значення для виявлення плагіату чи ідентичних фрагментів у різних документах. Зазвичай для цього застосовуються спеціалізовані програми, які аналізують уривки за допомогою різноманітних методів [3].

У сучасному інформаційному середовищі, де обсяги текстових даних постійно зростають, подібність між матеріалами розглядається як кількісна та якісна характеристика. Вона визначається через лексичні, семантичні, синтаксичні та структурні збіги. У межах комп'ютерної лінгвістики подібність текстів аналізується з кількох основних аспектів:

Лексичну схожість – це збіги окремих слів та словосполучень. Вона свідчить про використання однакових лексичних одиниць, що може бути результатом прямого копіювання чи спільної термінології. Однак значна лексична подібність не обов'язково свідчить про змістову ідентичність.

Семантичну схожість – це вже глибший рівень аналізу, коли документи мають подібний зміст, навіть якщо формулювання різні. Два тексти можуть бути семантично схожими, навіть використовуючи різні слова (синоніми) або маючи різну будову речень, але передаючи збіг ідеї. Виявлення семантичної подібності є ключовим для ідентифікації прихованих форм плагіату, таких як перефразування зі збереженням суті. Інструменти для перевірки на плагіат, що використовують технології обробки природної мови (NLP), спрямовані на аналіз саме семантичної подібності [19].

Структурну схожість – аналогічність композиції вмісту (наприклад, порядок розділів у наукових роботах). Вона відображає подібність у побудові та організації вмісту, включаючи однакову послідовність розділів, підрозділів, використання схожих аргументаційних схем та аналогічну організацію вступу,

основної частини та висновків. Структурна схожість особливо важлива у формалізованих текстах (наукові статті, звіти, юридичні документи) і може свідчити про запозичення не лише ідей, але й способу їхнього викладу. У поєднанні з лексичною та семантичною подібністю це може бути вагомим доказом плагіату. Проте в деяких випадках, особливо в офіційних документах, структурна подібність може бути зумовлена галузевими або жанровими стандартами [14].

Плагіат як соціально-правове явище має складну природу, охоплюючи не лише пряме копіювання, а й складніші форми неправомірного використання інтелектуальної власності. Виділяють кілька основних видів плагіату:

Прямий (дослівний) плагіат – дослівне копіювання фрагментів без цитування. Це найочевидніша та груба форма академічної нечесності, що полягає у копіюванні текстових уривків (одного чи кількох речень, абзаців або навіть цілих сторінок) з оригінального джерела та їх вставці у власну роботу без змін та належного цитування.

Характерні ознаки:

Повна ідентичність: скопійований документ повністю збігається з оригіналом, слово в слово.

Відсутність лапок: запозичений вміст не береться в лапки, що вказувало б на пряме цитування.

Відсутність посилання: не надається жодного посилання на оригінальне джерело, що створює враження, ніби ці ідеї та формулювання є власними.

Прямий плагіат вважається дуже серйозним порушенням академічної етики, оскільки він є прямим присвоєнням чужої інтелектуальної власності.

Мозаїчний плагіат – поєднання запозичених частин з власним текстом. Він є більш прихованою формою, що включає комбінування скопійованих фрагментів з оригінального джерела з оригінальними словами та реченнями, причому запозичені частини часто залишаються без належного цитування.

Характерні ознаки:

Переплетення: робота містить як оригінальні, так і скопійовані з різних джерел елементи.

Незначні зміни: запозичені фрагменти можуть бути дещо модефіковані (наприклад, перестановка слів, заміна окремих слів синонімами), але основна структура та ідеї залишаються незмінними.

Невідповідне цитування: навіть при згадці деяких джерел, конкретні запозичені фрагменти можуть бути не взяті в лапки або не мати чіткого посилання. Мозаїчний плагіат складніше виявити, ніж прямий, оскільки він маскується під оригінальний вміст. Однак детальний порівняльний аналіз вмісту з джерелами часто дозволяє його виявити.

Парафразний плагіат – зміна поверхневих елементів документу зі збереженням оригінальної структури думки. Виникає, коли автор перефразовує вміст з оригінального джерела, змінюючи слова та будову речень, але зберігає основні ідеї, аргументи, організацію та послідовність думок оригінального автора без належного цитування.

Характерні ознаки:

Перефразування: використовуються синоніми, змінюється порядок слів, активні конструкції замінюються на пасивні та навпаки.

Збереження суті: основні ідеї, висновки та структура аргументації оригінального джерела залишаються незмінними.

Відсутність або недостатнє цитування: джерело може згадуватися загально, але конкретні перефразовані ідеї не мають чітких посилань, створюючи враження їхньої оригінальності. Парафразний плагіат є порушенням академічної етики, оскільки він присвоює не лише формулювання, але й інтелектуальний внесок оригінального автора.

Автоплагіат – повторне використання власних попередніх робіт без вказівки джерела. Полягає у повторному використанні власної раніше опублікованої роботи (повністю або частково) в новій роботі без належного цитування попередньої публікації. Хоча автор має права на свої попередні

роботи, їх повторне представлення як нового оригінального матеріалу є академічно некоректним.

Характерні ознаки:

Повторне подання: той самий матеріал або його значні частини використовуються в новій роботі.

Відсутність посилання на попередню роботу: робота не містить посилання на попередню публікацію, створюючи враження новизни матеріалу.

Різні контексти: автоплагіат може траплятися при поданні однієї й тієї ж роботи на різні курси, у різні видання чи при включенні значних фрагментів старої роботи в дисертацію без згадки. Автоплагіат може бути порушенням авторських прав видавництв, а також створює хибне уявлення про новизну наукового внеску автора та може спричинити подвійне врахування однієї й тієї ж роботи [2].

Сутність перевірки на схожість полягає у тому, щоб гарантувати їхню оригінальність і дотримання авторських прав. Це особливо важливо у науці, журналістиці, освіті та інших сферах, де цінується чесність. Сьогодні, у часи масового використання Інтернету, випадків плагіату стало значно більше. Раніше перевірки на плагіат здебільшого здійснювали вручну, що займало багато часу і не завжди давало точні результати. Зараз активно використовуються автоматичні системи пошуку текстових збігів - так звані програми виявлення плагіату (TMS), набуло широкого поширення як у вигляді комерційних продуктів, так і рішень з відкритим кодом. Важливо зазначити, що TMS не є інструментом для визначення плагіату, а лише виявляє збіги між документами. Комп'ютерне виявлення плагіату (CaPD) є задачею інформаційного пошуку (IR), що реалізується спеціалізованими системами, відомими як системи виявлення плагіату (PDS) або системи ідентифікації подібності документів[35].

Однак деякі теоретики піддають сумніву абсолютність поняття оригінальності, що впливає на визначення плагіату: «Якщо немає оригінальності, немає підґрунтя для літературної власності.» Соціальний та культурний зміст також відіграє важливу роль у визначенні норм використання

документів, як це видно на прикладі Мартіна Лютера Кінга-молодшого, де усні традиції спільноти впливали на його письмо.

Університетські правила щодо плагіату, часто засновані на ідеалі «незалежного творця», можуть спрощувати складність студентського письма. Навіть «уривкове письмо» може бути важливим етапом у навчанні. Враховуючи ці складнощі, існує потреба в «перегляді встановлених правил» для кращого врахування різноманітних контекстів авторства [40].

Таким чином, перевірка на схожість є важливим інструментом, але її сутність та застосування потребують постійного критичного осмислення з урахуванням сучасних теоретичних розробок та різноманітних умов створення документів.

1.2 Огляд існуючих рішень для перевірки текстів

Перевірка вмісту на схожість сьогодні має велике значення у найрізноманітніших сферах: від наукових досліджень і юридичних перевірок до боротьби з плагіатом в інтернеті. Завдяки швидкому розвитку технологій з'явилася велика кількість інструментів, які допомагають автоматично визначати ступінь подібності між різними текстами. Вивчення особливостей цих програм дозволяє краще зорієнтуватися і вибрати той варіант, який найкраще підходить під конкретні потреби. Попри різноманіття наявних інструментів, кожен з них вирізняється власними особливостями, сильними та слабкими сторонами [29].

Сьогодні програми для перевірки документів стали справжніми помічниками для авторів, редакторів і всіх, хто працює з великими обсягами інформації. Вони значно полегшують процес порівняння документів, допомагають швидко виявляти відмінності й забезпечують точність у роботі. Серед найбільш популярних варто виділити кілька сервісів.

DiffChecker.net – це безкоштовний в онлайн-сервіс, який використовує спеціальні алгоритми для знаходження відмінностей між двома текстами. Він наочно демонструє цю різницю у зручному форматі. Завдяки застосуванню векторного представлення змісту та різних методів порівняння, цей інструмент

забезпечує точне виявлення розбіжностей між документами. Багато користувачів відзначають простоту використання, його ефективність у знаходженні відмінностей, а також високу надійність та продуктивність.

Основні характеристики:

1. Онлайн-інструмент не потребує встановлення на комп'ютер.
2. Швидке та просте порівняння текстових даних.
3. Візуальне виділення відмінностей поруч один з одним.
4. Зручний інтерфейс.

Переваги:

1. Інтуїтивно зрозумілий інтерфейс;
2. Працює з різними форматами (JSON, HTML, DOC тощо);
3. Можливість порівняння збережених файлів;
4. Колірне виділення змін для зручного аналізу.

Недоліки:

1. Обмежений функціонал без підписки;
2. Залежність від інтернет-з’єднання;
3. Немає глибокого аналізу семантики.

Ще одним популярним сервісом є *Text Compare*, який дозволяє легко знаходити різницю між документами, що значно спрощує аналіз. Користувачі високо цінують за його зручність та продуктивність, відзначаючи його надійність та ефективність.

Основні характеристики:

1. Виділення відмінностей між матеріалами.
2. Спрощення процесу виявлення відмінностей.
3. Висока надійність.
4. Значна ефективність у роботі.

Переваги:

1. Не потребує реєстрації;
2. Миттєве порівняння навіть великих фрагментів;
3. Підходить для початківців.

Недоліки:

1. Відсутність додаткових функцій (збереження, аналіз структури тощо);
2. Мінімалістичний інтерфейс без глибших аналітичних опцій.

Онлайн-сервіс *DiffNow* допомагає порівнювати зміст, відображаючи відмінності за допомогою кольорових позначок, що дозволяє швидко знаходити зміни між двома текстами завдяки візуальному представленню різниці. Користувачі позитивно відгукуються про зручний інтерфейс та ефективність у проведенні паралелей документів, підкреслюючи його надійність та продуктивність.

Основні характеристики:

1. Відображення відмінностей за допомогою кольорових позначень
2. Швидке знаходження змін у текстах
3. Зрозуміле візуальне відображення різниці
4. Зручний користувацький інтерфейс
5. Висока ефективність

Переваги:

1. Можливість порівнювати різні формати (PDF, DOCX, Excel);
2. Працює як із локальними, так і з онлайн-файлами;
3. Можна зберігати результати.

Недоліки:

1. Обмеження кількості порівнянь без облікового запису;
2. Безкоштовна версія має обмежений функціонал;
3. Може бути повільнішим при порівнянні важких документів.

Безкоштовна програма для Windows, *WinMerge*, дозволяє зручно перевіряти не лише тексти, але й цілі папки. Завдяки широкому набору функцій і підсвічуванню синтаксису вона підходить як для повсякденної роботи з вмістом, так і для програмістів.

Основні характеристики:

1. Перегляд порівнюваних документів поруч.

2. Візуальне виділення відмінностей.
3. Функціональність об’єднання для комбінування змін.
4. Підтримка підсвічування синтаксису.
5. Порівняння та синхронізація папок.

Переваги:

1. Потужна інтеграція з операційною системою;
2. Можливість об’єднання текстів в один;
3. Підтримка численних форматів;
4. Ідеальний для роботи з кодом.

Недоліки:

1. Працює лише на Windows;
2. Відносно складний інтерфейс для новачків;
3. Не підтримує онлайн-режим.

Ще один безкоштовний додаток, *Meld*, працює на різних операційних системах. Він забезпечує зручне порівняння та об'єднання текстових файлів, що робить його корисним інструментом для роботи. Користувачі відзначають за його інтуїтивно зрозумілий інтерфейс, підкреслюючи його надійність та продуктивність.

Основні характеристики:

1. Візуальне порівняння документів та папок.
2. Підтримка порівняння трьох версій файлу.
3. Вбудований інструмент для об’єднання та вирішення конфліктів.
4. Підсвічування синтаксису для багатьох мов програмування.
5. Опція ігнорування змін у пробілах.

Переваги:

1. Кросплатформеність;
2. Гнучкість налаштувань;
3. Підтримка SCM-систем (Git, Bazaar, Mercurial).

Недоліки:

1. Вимагає встановлення;

2. Іноді складний для користувачів без технічного досвіду;
3. Відсутність вебверсії[38].

Загалом, за критеріями надійності, ефективності й відгуками користувачів, *DiffChecker* і *Text Compare* заслужено вважаються одними з найкращих інструментів для швидкого та якісного порівняння текстів. Обидві програми демонструють високу точність у виявленні відмінностей, мають простий інтерфейс і отримали чимало позитивних відгуків за зручність у роботі [9].

1.3 Аналіз існуючих методів

Для аналізу та визначення ступеня подібності між матеріалами існує чимало алгоритмів, кожен з яких має свої сильні та слабкі сторони. Одним з найпоширеніших є TF-IDF. Це, по суті, статистичний метод, що визначає важливість слів у тексті. Він робить це, беручи до уваги, як часто слово використовується, і наскільки воно унікальне порівняно з іншими документами в певній колекції. TF обчислюється як відношення кількості входжень терміна ω_i у документі $j(n_{ij})$ до загальної кількості термінів у ньому $(\sum_k n_{kj})$, що математично виражається як:

$$tf(\omega_i) = \frac{\Sigma_k\, n_{kj}}{n_{ij}}, \tag{1.1}$$

де n_{ij} - кількість входжень терміна ω_i в документі d_j, а $\sum_k n_{kj}$ - загальна кількість термінів у документі d_j.

TF-IDF поєднує два основні показники:

TF (Term Frequency) - кількість згадок певного слова в тексті.

IDF (Inverse Document Frequency) - міра рідкості слова в наборі документів. Чим рідше слово зустрічається в інших документах, тим більшу вагу воно набуває в контексті конкретного документа. IDF, зі свого боку, обчислюється як логарифм частки від ділення загальної кількості документів (N) на кількість документів, де зустрічається даний термін ω_i ($|\{j: \omega_i \in d_j\}|$):

$$idf(\omega_i) = log\ \frac{N}{|\{j:\omega_i \in d_j\}|} \tag{1.2}$$

Значення TF-IDF для терміна ω_i є добутком цих двох величин:

$$TFIDF(\omega_i) = tf(\omega_i) \times idf(\omega_i) \quad (1.3)$$

Переваги:

1. Простота реалізації та використання.
2. Ефективність у випадку перевірки на схожість між великими обсягами документів.

Недоліки:

1. Не враховує семантичний зв'язки між словами (однакові слова можуть мати різний зміст).
2. Не може розпізнати синоніми та не враховує граматичні зміни слів [17].

Переходячи до іншого методу, варто згадати косинусну схожість. Це спосіб вимірювання подібності між двома вбудовуваннями на основі кута між ними. Якщо вектори спрямовані в одному напрямку, кут між ними мінімальний (0°), і значення косинусної схожості дорівнює 1. Це означає високу схожість. Якщо ж вектори перпендикулярні (кут 90°), схожість дорівнює 0. І, нарешті, якщо вектори протилежні (кут 180°), значення стає -1, що вказує на максимальну несхожість. Отже, косинусна схожість коливається в діапазоні від -1 до 1. (Рисунок 1.1).

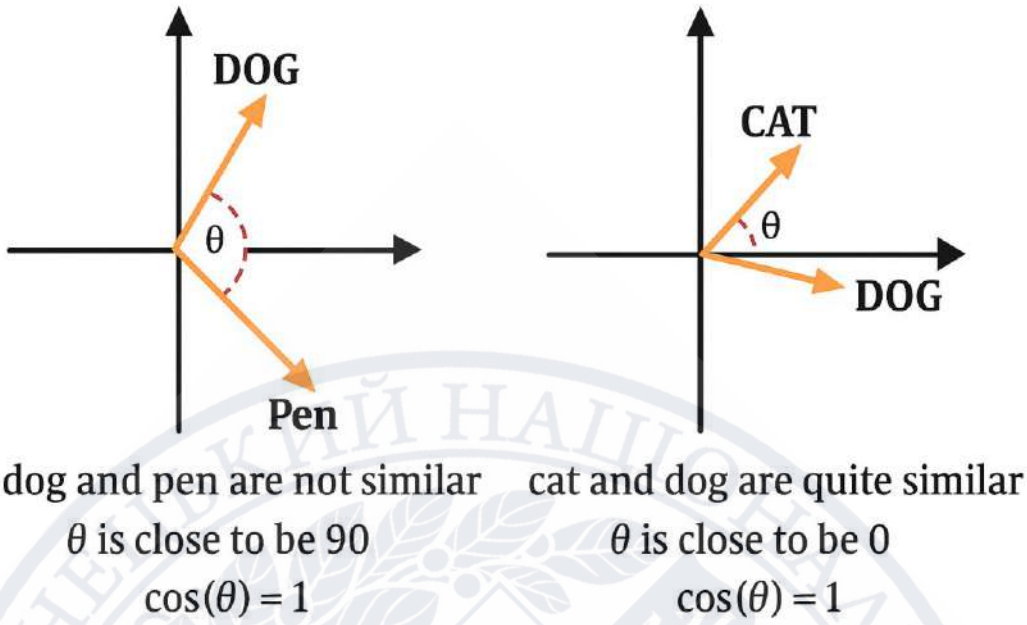

Рисунок 1.1 – Косинусна схожість між векторними представленнями слів

Принцип дії цього методу полягає в тому, що тексти перетворюються на вектори. Кожна складова вектора відображає частоту певного терміна. Схожість між двома матеріалами визначається за допомогою косинусної міри, яка обчислює косинус кута між двома векторами A та B:

$$< 0 > \cos(\theta) = \frac{A \times B}{||A|| \times ||B||} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \times \sqrt{\sum_{i=1}^{n} B_i^2}}, \quad (1.4)$$

де:

- $A \cdot B$ - скалярний добуток векторів A та B.
- $||\ A\ ||$ та $||\ B\ ||$ - евклідові норми (довжини) векторів A та B.
- n - розмірність векторів (кількість унікальних термінів).
- Ai та Bi - значення і-го елемента у векторах A та B відповідно.

Переваги цього методу полягають у тому, що він враховує відносне розташування слів у текстах і є досить легким в обчисленні та впровадженні. Однак, варто зауважити, що він певним недоліком є те, що цей метод ігнорує структуру речення та не розпізнає синоніми [21].

Метод n-грам полягає в аналізі послідовностей з n слів (або символів) у вмісті та їхньому використанні для перевірки подібності між документами. Розбиття змісту на n-грами дозволяє порівнювати частини документу на рівні окремих слів або словосполучень.

Переваги цього методу в тому, що він забезпечує кращий аналіз змісту порівняно з методами, що ґрунтуються лише на окремих словах. Він також враховує локальні структури документів. З іншого боку, він не звертає увагу на значення слів (лише лексичну схожість) і може генерувати значну кількість непотрібних даних для довгих текстів.

З розвитком глибокого навчання та нейронних мереж з'явились нові підходи до аналізу схожості фрагментів, які використовують більш складні моделі, здатні розуміти контекст, семантику і навіть тональність вмісту.

BERT є однією з передових моделей NLP, що використовує архітектуру трансформерів для двонаправленого аналізу документу, завдяки чому досягає глибокого розуміння аоля значень та семантичних зв'язків. Навчена на великих текстових корпусах, BERT може бути донавчена для вирішення конкретних завдань, зокрема визначення схожості документів, на спеціалізованих датасетах.

Переваги BERT очевидні: він враховує глибокий контекст і семантику, а також може ефективно розпізнавати схожість навіть при використанні синонімів. Проте, слід визнати, що він має високу обчислювальну складність і потребує значних ресурсів. Крім того, для його навчання потрібен великий обсяг даних.

Sentence-BERT - це, по суті, варіант BERT, оптимізований спеціально для порівняння подібності між цілими реченнями або документами. Він використовує технології BERT для створення векторних представлень для кожного речення, які можна порівнювати для виявлення схожості. SBERT перетворює речення в багатовимірні вектори і використовує схожість векторів для порівняння схожості.

Переваги SBERT полягають у тому, що він дозволяє порівнювати речення та тексти в цілому, а також забезпечує високу точність при виявленні семантичної схожості. Однак, він також має значні вимоги до обчислювальних

потужностей і може бути складним в інтеграції в практичні додатки без відповідної адаптації.

GloVe є моделлю для створення векторних представлень слів на основі статистичних характеристик. Її можна використовувати для аналізу подібності матеріалів, оскільки вона дозволяє оцінити схожість за допомогою їхніх векторів.

Переваги GloVe - проста інтеграція в різні системи та використання змісту слів у ширшому розумінні. Але він не настільки потужний, як трансформери (BERT або SBERT), і не може повністю обробляти складні семантичні зв'язки.

Для подолання обмежень TF-IDF у врахуванні семантики пропонується гібридний метод, що поєднує статистичну силу TF-IDF. Використовує (TSWT) та семантичну інформацію з HowNet. Процес включає попередню обробку тексту (сегментація, лематизація, видалення стоп-слів, обробка спеціальних термінів, об'єднання синонімів), а потім вибір ключових термінів на основі їхніх високих значень TF-IDF. Експерименти показали, що гібридний метод перевершує як чистий TF-IDF, так і методи, що базуються лише на семантичному розумінні, за показниками точності, повноти та F1-міри при кластеризації. Ефективність оцінювалася, зокрема, шляхом обчислення середніх показників по класах[30]. Для цього було застосовано макроусереднення - підхід, при якому метрики (точність, повнота, F1-міра) обчислюються окремо для кожного класу, а потім усереднюються арифметично. Такий метод дозволяє об’єктивно оцінити якість кластеризації незалежно від кількості прикладів у кожному класі, надаючи однакову вагу кожній категорії. Це особливо важливо у випадках, коли кластери є нерівномірними за обсягом, як це часто трапляється в реальних текстових даних.

Цей підхід є досить перспективним, оскільки він поєднує статистичну важливість слів з їхнім семантичним значенням. Це, ймовірно, підвищує точність і зменшує розмірність векторного представлення порівняно з чисто семантичними методами. Розробка ефективного програмного забезпечення для перевірки текстів базується на кількох ключових теоретичних підходах.

Лінгвістичні аспекти: перевірка вмісту повинна враховувати граматику, синтаксис, стилістику та семантику. Для виявлення граматичних помилок використовуються граматичні правила і методи статистичного аналізу.

Алгоритми та штучний інтелект: машинне навчання та обробка природної мови є основними для розробки програм для перевірки текстів. Вони дозволяють покращити точність перевірки шляхом навчання на прикладах правильних і неправильних документів.

Контекстний аналіз: аналіз змісту тексту допомагає точно визначити правильність використання слів та фраз, враховуючи різні мовленнєві ситуації.

Реалізація інтерфейсу користувача: інтерфейс має бути зручним і інтуїтивно зрозумілим, щоб користувачі могли швидко взаємодіяти з програмою і отримувати зрозумілі результати.

Тестування та оцінка якості: програмне забезпечення (ПЗ) повинно бути протестоване на різноманітних матеріалах для перевірки точності і ефективності. Порівняння результатів з іншими програмами також допомагає оцінити якість.

Розпізнавання структури тексту: важливо правильно розпізнавати структуру вмісту (абзаци, речення, фрази), що допомагає зберегти логічний порядок та забезпечити якісні результати перевірки.

Врахування мовних особливостей: ПЗ повинно враховувати лінгвістичні особливості різних мов та їхніх варіантів, включаючи складні граматичні правила та використання специфічних частин мови.

Виявлення стилістичних аспектів: для точнішої перевірки важливо враховувати стилістичні особливості різних типів текстів (наприклад, офіційний лист vs. блог-пост).

Лексичний та семантичний аналіз: для виявлення граматичних помилок та неоднозначності значень слів, програми повинні проводити аналіз лексики та семантики документу.

Розробка інтерактивних функцій: включення таких функцій, як автоматичні підказки або виправлення помилок, покращує досвід користувачів і робить перевірку тексту ефективнішою.

Інтеграція штучного інтелекту та машинного навчання є критично важливою для забезпечення високої якості перевірки, оскільки ці технології допомагають системам навчатися та адаптуватися до нових даних.

Існуючі рішення для порівняння використовують широкий спектр підходів – від простих статистичних методів, таких як TF-IDF та косинусна схожість, до складних моделей глибокого навчання, таких як BERT та SBERT, а також різноманітне програмне забезпечення для перевірки на схожість. Вибір методу та інструменту залежить від вимог до точності, швидкості та доступних обчислювальних ресурсів, а також від конкретних потреб користувача у порівнянні та аналізі контенту. У майбутньому очікується, що з розвитком технологій перевірка схожості документів стане ще більш точною та ефективною завдяки використанню передових моделей на основі нейронних мереж та удосконаленню існуючих програмних рішень[26].

1.4 Постановка задачі дослідження

Створення ефективного програмного забезпечення для виявлення подібностей між текстами є процесом, що потребує ретельного опрацювання кожного кроку, починаючи від отримання вихідних даних і закінчуючи представленням результатів користувачеві. У цьому розділі подано докладний огляд основних етапів налаштування та розробки такого програмного забезпечення з акцентом на сучасних підходах та потребах користувацького досвіду.

Першим і надзвичайно важливим кроком є збір вихідних матеріалів, які будуть порівнюватися на предмет схожості. Джерела цих документів можуть бути різноманітними: вебсторінки, локальні файли різних форматів, бази даних, а також безпосередньо введений користувачем текст. На цьому етапі головним є забезпечення належного представлення вмісту у форматі, зручному для подальшої обробки, наприклад, у вигляді словесних рядків або структурованих об'єктів.

Після отримання документів необхідно провести їхню попередню обробку, що є критично важливим для якості майбутнього аналізу. Цей етап включає ряд дій, спрямованих на очищення, нормалізацію та підготовку текстових даних. У випадку багатомовних або уривок з різним кодуванням, важливо їх уніфікувати шляхом визначення мови та перекодування. Ефективна їх обробка на цьому етапі є основою для подальшого якісного аналізу схожості.

Ключовим етапом розробки програмного забезпечення є вибір та застосування алгоритмів і методів для визначення ступеня подібності між підготовленими текстами. Існує кілька основних підходів до вирішення цієї задачі, включаючи порівняння векторних моделей слів (наприклад, на основі TF-IDF з використанням косинусної схожості або з використанням векторних представлень слів), застосування статистичних метрик (таких як коефіцієнт Жаккара або редакторська відстань), а також застосування методів NLP (семантичний аналіз та нейронні мережі). Вибір конкретного методу або їхньої комбінації залежить від вимог до точності, обсягу даних, наявних обчислювальних ресурсів та специфіки завдань, для яких створюється ПЗ.

Завершальним кроком є надання користувачеві зрозумілих та інформативних результатів аналізу схожості. Ефективне відображення результатів є запорукою зручності використання програмного забезпечення. Це може включати числову оцінку подібності (наприклад, відсоток схожості), візуальне порівняння з виділенням подібних та відмінних частин, порівняльні графіки та діаграми, а також детальні звіти. Важливо розробити інтуїтивно зрозумілий інтерфейс, який дозволить користувачеві легко інтерпретувати результати аналізу та приймати на їхній основі обґрунтовані рішення.

Окрім основних етапів, при розробці програмного забезпечення для пошуку схожості текстів необхідно враховувати ряд додаткових аспектів, таких як ретельний вибір методу виявлення схожості, управління великими обсягами даних, підвищення точності та надійності аналізу, забезпечення захисту даних, автоматизація процесів, відстеження та аналіз результатів роботи програми, забезпечення сумісності та інтеграції з іншими системами, а також орієнтація на

потреби користувача при створенні інтерфейсу. Врахування всіх цих факторів дозволить створити потужне, ефективне та зручне у використанні програмне забезпечення для автоматичної перевірки подібності, яке відповідатиме сучасним вимогам [12].

РОЗДІЛ 2
ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Вимоги до програмного забезпечення

На етапі, коли ми визначаємо, що ж саме має робити наша система, формулюємо чіткі, вимірювані, перевірювані та відстежувані вимоги. Вимоги до системи являють собою формалізований перелік критеріїв, яким повинне відповідати розроблюване програмне забезпечення. Вони є фундаментальною основою для подальшого процесу проектування та слугують індикатори оцінки якості кінцевого продукту. Етап визначення вимог передбачає комплексний аналіз потреб усіх зацікавлених сторін, включаючи кінцевих користувачів, замовників, команду розробників та персонал, відповідальний за подальшу експлуатацію та супровід системи (Рисунок 2.1).

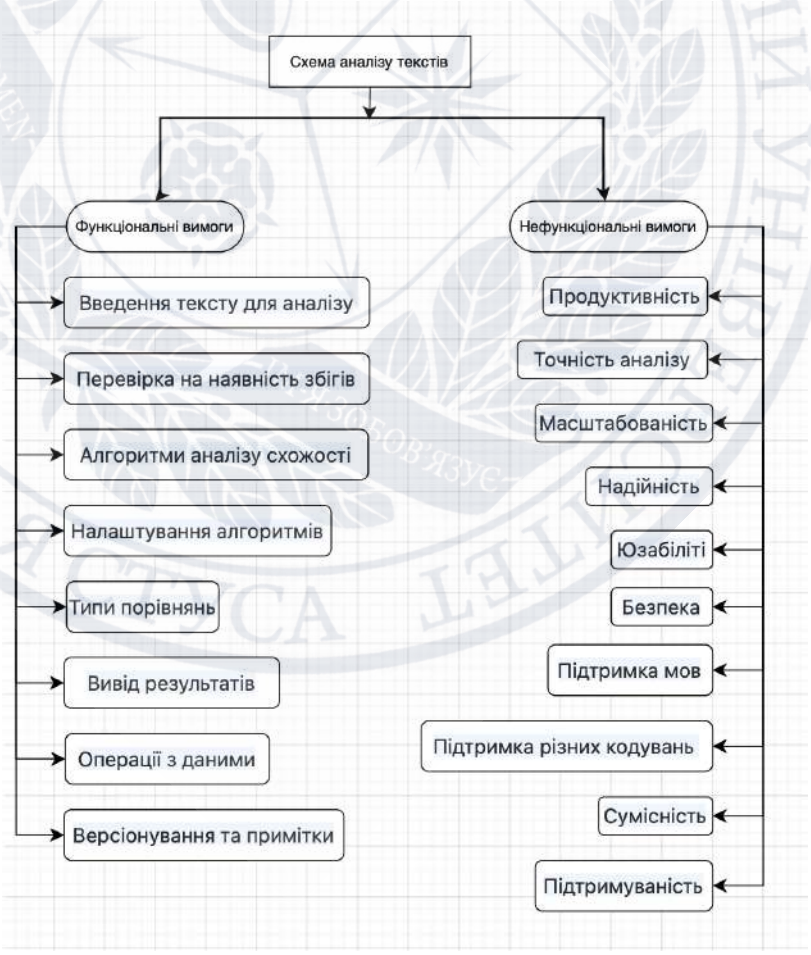

Рисунок 2.1 Схема чітких вимог

Функціональні вимоги є наріжним каменем у процесі проектування програмного забезпечення, адже саме вони визначають конкретні операції та послуги, які надаватиме майбутня система. Ці вимоги детально описують її поведінку у відповідь на вхідну інформацію, перелік виконуваних завдань та очікувані результати функціонування, формуючи тим самим чітке розуміння можливостей програмного продукту.

Для забезпечення зручної взаємодії з системою, однією з функціональних вимог є введення тексту. Система повинна надавати різноманітні та інтуїтивно зрозумілі способи внесення уривків для подальшого аналізу та порівняння. Це включає в себе пряме введення вмісту через зручне текстове поле інтерфейсу користувача, що є найбільш простим та швидким способом для невеликих обсягів інформації. Крім того, передбачається можливість завантаження файлів у найбільш поширених форматах, таких як .txt, та .pdf, з коректною обробкою різних кодувань символів, включаючи UTF-8 та Windows-1251, що забезпечує універсальність роботи з різними джерелами [16].

Основною метою системи є перевірка на наявність збігів. Ця функціональність передбачає виявлення фрагментів схожості між введеними текстами та їх порівняння з базою даних індексованих документів або відкритими джерелами інформації для виявлення плагіату або дублювання контенту.

З метою забезпечення гнучкості аналізу, користувачеві слід надати змогу обирати один або комбінацію кількох алгоритмів для аналізу ступеня схожості документів, беручи до уваги особливості завдання. Доступними можуть бути як класичні статистичні підходи, такі як TF-IDF (Term Frequency-Inverse Document Frequency) з подальшим обчисленням косинусної схожості, що ефективні для виявлення лексичної подібності, так і методи на основі n-грамного аналізу для виявлення послідовностей ідентичних слів. Для більш глибокого аналізу, система повинна підтримувати сучасні семантичні методи аналізу, що використовують векторні представлення слів та документів, отримані за допомогою моделей Word2Vec, GloVe, FastText або трансформерних архітектур

(наприклад, BERT, SBERT), для ідентифікації смислової подібності, враховуючи контекст та синонімію [25].

Для пристосованості системи до різних сценаріїв використання, користувачеві необхідно надати інструменти для конфігурації основних параметрів обраних алгоритмів. Приміром: для n-грамного аналізу – розмір n-грами, що впливає на чутливість до коротких або довгих збігів; для TF-IDF – схема зважування термінів, що визначає важливість різних слів; для семантичного аналізу – поріг схожості векторів, що регулює рівень необхідної смислової близькості. Крім того, важливо передбачити налаштування обробки стоп-слів (вилучення загальновживаних службових слів), що підвищує точність аналізу, та застосування лематизації (зведення слів до початкової форми), що дозволяє ігнорувати граматичні варіації слів.

Система повинна забезпечувати можливість порівняння двох текстів, що є базовою функціональністю, яка дозволяє користувачеві завантажити або ввести два окремі документи та отримати кількісну та якісну оцінку їхньої схожості. Для більш масштабних завдань, передбачається порівняння одного тексту з колекцією, що надає можливість виявляти потенційні плагіатні фрагменти або схожі за змістом матеріали шляхом порівняння введеного вмісту з набором попередньо завантажених, індексованих або підключених документів. Крім того, реалізується функціонал порівняння кількох матеріалів між собою, що дозволяє проводити попарне порівняння у групі з кількох завантажених текстів, що може бути корисним для виявлення взаємних збігів у наборі документів.

Результати аналізу повинні бути відображені у чіткому та інформативному вигляді, охоплюючи кількісну оцінку ступеня схожості (наприклад, у відсотках), що дозволяє швидко оцінити рівень подібності. Для детального аналізу, передбачається візуальна ідентифікація схожих та відмінних фрагментів безпосередньо у порівнюваних (наприклад, за допомогою кольорової індикації), що полегшує розуміння знайдених збігів. Для документування результатів, система повинна генерувати узагальнені порівняльні звіти, що детально відображають знайдені збіги, їхнє розташування у текстах та ступінь подібності.

Для ефективного оперування даними передбачено їхнє упорядкування. Система повинна забезпечити функціонал для занесення введених документів, результатів аналізу та інших релевантних відомостей до бази даних, а також можливості їхнього перегляду, редагування та оперативного пошуку за різними критеріями, що полегшить роботу з великими обсягами інформації.

Задля забезпечення можливості відстеження змін та повернення до попередніх версій, реалізується функціональність версіонування текстів, що дозволяє користувачам зберігати історію змін введених уривків. Для покращення спільної роботи та документування знайдених збігів, передбачається можливість додавання приміток та коментарів до результатів аналізу.

Якість програмного забезпечення визначається через призму нефункціональних вимог, що безпосередньо впливають на його ефективність, зручність експлуатації та загальну цінність для користувачів. Ці вимоги встановлюють критерії, за якими оцінюється робота системи, і є не менш важливими за функціональні, оскільки визначають досвід користувача та стабільність функціонування продукту.

Однією з ключових нефункціональних вимог є продуктивність. Система повинна демонструвати швидке виконання аналізу схожості навіть при обробці значних обсягів текстової інформації та одночасній роботі великої кількості користувачів. Час відгуку системи на ключові операції, такі як завантаження сторінки, ініціація аналізу та відображення результатів, не повинен перевищувати встановлених часових меж, забезпечуючи комфортну та ефективну роботу користувачів без неприємних затримок.

Іншою вкрай важливою вимогою є висока точність виявлення схожості. Алгоритми, що лежать в основі аналізу, повинні забезпечувати високий рівень точності не лише у виявленні прямих текстових збігів, але й у розпізнаванні більш складних випадків плагіату, включаючи перефразування, використання синонімічних замін та зміну порядку слів. Це дозволить забезпечити достовірність результатів та ефективно виявляти навіть завуальовані спроби копіювання.

Здатність системи адаптуватися до зростаючих потреб, тобто масштабованість, є критично важливою. Тому її архітектура має підтримувати ефективне горизонтальне та вертикальне розширення для обробки більших обсягів даних, зростаючої кількості користувачів та нового функціоналу без суттєвої втрати продуктивності та стабільності.

Стабільність та безперебійність роботи системи визначаються вимогою надійності. Система повинна працювати стабільно та без помилок протягом тривалого часу, забезпечуючи високу доступність функціоналу та достовірність результатів аналізу. Для досягнення цієї вимоги необхідно передбачити дієві механізми моніторингу стану системи, опрацювання потенційних помилок, регулярного резервування даних та їхнього відновлення у разі непередбачуваних ситуацій.

Комфорт та зручність взаємодії з системою забезпечуються вимогою зручності використання (юзабіліті). Інтерфейс користувача має бути інтуїтивно зрозумілим, забезпечувати легку навігацію між розділами та функціями, а також сприяти ефективній взаємодії з системою користувачами різного рівня технічної підготовки. Дизайн інтерфейсу повинен відповідати сучасним стандартам UI/UX, забезпечуючи зручність введення даних, інтуїтивне налаштування параметрів аналізу та зрозумілий перегляд отриманих результатів.

Пріоритетним аспектом при розробці системи є безпека. Забезпечення конфіденційності та цілісності даних користувачів є ключовою нефункціональною вимогою. Це вимагає використання надійних механізмів аутентифікації та авторизації для контролю доступу до функцій і даних, а також шифрування конфіденційної інформації як під час передачі (наприклад, через HTTPS), так і при зберіганні в базі даних, що зменшує ризики несанкціонованого доступу та витоку інформації.

Зважаючи на глобальний характер інформаційного обміну, важливою є підтримка різних мов. Система повинна мати можливість обробки та порівняння різними мовами, що залежить від обраних алгоритмів аналізу та наявності

відповідних лінгвістичних ресурсів, таких як словники стоп-слів та моделі для лематизації та семантичного аналізу для кожної підтримуваної мови.

Для забезпечення коректної роботи з документами з різних джерел, необхідною є підтримка різних кодувань. Система повинна забезпечувати коректну обробку текстів, представлених у різних кодуваннях символів (наприклад, UTF-8, Windows-1251), щоб уникнути проблем з відображенням символів та спотворенням результатів аналізу.

Залежно від потреб користувачів та існуючої інфраструктури, система повинна відповідати вимозі працювати на різних платформах, зокрема підтримувати веббраузери. Система буде веб-додатком, доступним через будь-який сучасний веббраузер незалежно від операційної системи.

Нарешті, для забезпечення довготривалої ефективності та актуальності системи, важливою є підтримуваність. Система повинна бути розроблена для легкої підтримки, оновлення та розширення в майбутньому. Це включає використання чіткої та зрозумілої архітектури, добре документованого коду та застосування стандартних підходів до розробки, що полегшить внесення змін, виправлення помилок та додавання нових функціональних можливостей [36].

2.2 Вибір технологій для розробки

Вибір відповідного стеку технологій є одним з найважливіших етапів розробки програмного забезпечення, оскільки він безпосередньо впливає на його функціональність, продуктивність, зручність використання, масштабованість, підтримку та стабільність. У межах створення клієнтського застосунку для аналізу текстової схожості, що має бути придатним як для вебплатформи, так і для нативного запуску на операційних системах Windows, macOS та Linux, основна увага була зосереджена на JavaScript-орієнтованих технологіях, які забезпечують крос-платформність, гнучкість розробки та сучасний рівень інтерактивності інтерфейсу.

Основу системи становить фреймворк React, який забезпечує компонентну архітектуру, дозволяє повторно використовувати елементи інтерфейсу та

забезпечує високу швидкість рендерингу завдяки віртуальному DOM. Цей фреймворк ідеально підходить для реалізації SPA-додатків (Single Page Applications), які потребують миттєвої реакції на дії користувача без повного перезавантаження сторінки. До основних переваг React також належать велика спільнота, широка екосистема готових бібліотек (наприклад, react-router, react-hook-form) та підтримка TypeScript при потребі у типізації.

Оскільки система повинна бути доступною не лише через браузер, а й як самостійний застосунок, що не залежить від сторонніх серверів чи бекендів, до складу проєкту включено Electron. Цей фреймворк надає можливість компілювати вебзастосунки у вигляді повноцінних десктопних програм. Electron використовує Chromium для візуалізації інтерфейсу та Node.js для взаємодії з операційною системою, що дозволяє створювати крос-платформні рішення з єдиною кодовою базою. Таким чином, React забезпечує фронтенд-функціональність, а Electron обгортає цей інтерфейс у нативний додаток.

Для забезпечення основного функціоналу порівняння текстів використовуються бібліотеки на зразок «text-diff», яка дозволяє здійснювати символьну або рядкову диференціацію вхідних даних. Візуалізація схожих фрагментів реалізована за допомогою стандартних можливостей CSS та JavaScript, що дає змогу позначати спільні або відмінні частини у зручному для користувача форматі. Також до складу системи включено бібліотеку «pdfjs-dist» для можливості обробки PDF-документів як одного з варіантів джерела вхідного тексту.

Усе програмне забезпечення виконується повністю на клієнтському боці, не потребує зовнішнього підключення до серверів або баз даних, і працює автономно на пристрої користувача. Це забезпечує високий рівень безпеки, оскільки дані не залишають межі локального середовища, а також гарантує доступність інструменту навіть в умовах обмеженого або відсутнього підключення до Інтернету.

Загалом обраний стек технологій, який поєднує JavaScript, React та Electron, дає змогу створити масштабований, швидкий, зручний та функціональний застосунок, здатний задовольнити потреби користувачів у кросплатформному аналізі текстової схожості без потреби у зовнішніх ресурсах чи серверній логіці [9].

Семантична подібність є засадничим поняттям у сфері NLP, що надає машинам здатність осягати глибинний зміст тексту. Її важливість охоплює численні застосування, починаючи від підвищення точності пошукових систем і закінчуючи вдосконаленням взаємодії чат-ботів та оптимізацією систем рекомендацій. Зі зростанням потреби в прецизійних інструментах, усвідомлення та застосування семантичної подібності стає ключовим чинником при розробці інтелектуальних, адаптивних систем [39].

Семантична подібність являє собою метрику, що використовується для оцінки ступеня смислової близькості між двома текстовими фрагментами на основі їхнього значення або семантичного наповнення [32]. На відміну від простого лексичного зіставлення, яке концентрується на ідентичності використаних слів, семантична подібність аналізує глибинний контекст та концепції, що передаються вмістом. Цей підхід забезпечує більш глибоке розуміння мови, дозволяючи машинам обробляти та інтерпретувати людську мову з більшою ефективністю [27].

З технічної точки зору, семантичну подібність можна кількісно оцінити за допомогою різноманітних математичних моделей та алгоритмів, які аналізують зв’язки між словами, фразами чи цілими текстами. В основі таких моделей зазвичай лежить векторне представлення слів (word embeddings), а також методи глибокого навчання, що дозволяють передавати змістові відношення. Семантична подібність є ключовою складовою в обробці природної мови (NLP), оскільки забезпечує більш глибоке розуміння мови машиною. Саме завдяки аналізу змістовних зв’язків між текстами сучасні інтерфейси можуть виконувати складні лінгвістичні завдання. Основні переваги, які надає використання семантичної подібності у вебзастосунках, такі:

• дозволяє точно інтерпретувати наміри користувача, забезпечуючи змістовно релевантні відповіді та покращуючи загальну взаємодію з системою;

• забезпечує точний пошук і систематизацію даних у межах локального застосунку, без необхідності підключення до серверів чи баз даних;

• підвищує зручність користування завдяки контекстно орієнтованим результатам аналізу, адаптованим під зміст, а не лише ключові слова.

У випадку вебзастосунків, що працюють у середовищі Electron, семантична подібність може бути інтегрована без потреби у зовнішньому сервері. Наприклад, коли користувач вводить запит на кшталт «найкращі ідеї для літнього відпочинку», додаток може аналізувати не лише наявність слів «відпочинок» чи «літо», а й загальний зміст, надаючи релевантні тексти про туристичні напрямки. Це досягається завдяки використанню локальних моделей для векторизації та зіставлення текстів, таких як USE або MiniLM, які можна інтегрувати напряму у фронтенд.

React у поєднанні з Electron дозволяє створювати гнучкі кросплатформні додатки, де обробка даних здійснюється повністю на боці клієнта. Це усуває потребу у класичних архітектурах із сервером та базами даних. При роботі з великими текстовими масивами, обчислення векторної подібності можуть виконуватись за допомогою WebAssembly або оптимізованих JS-бібліотек, що забезпечує високу продуктивність навіть у середовищі браузера чи настільного застосунку.

Одним із перспективних рішень є використання векторних сховищ прямо у клієнтському додатку. Такі бібліотеки, як annoy-lite або faiss-js, дозволяють створювати локальні індекси для швидкого пошуку схожих фрагментів тексту. Це особливо корисно для реалізації таких функцій, як семантичний пошук у великих документах або системи рекомендацій у офлайн-режимі. Перевага локального зберігання та обробки полягає в захисті даних користувача, відсутності залежності від інтернет-з’єднання та зменшенні затримки у взаємодії.

Інтеграція з моделями машинного навчання можлива через використання готових вебсумісних моделей, таких як DistilBERT або SimCSE, які можуть бути завантажені у форматі ONNX або TF.js. Це дозволяє запускати потужні алгоритми прямо в браузері або в середовищі Electron. Наприклад, модель SimCSE дає змогу точно визначати семантичну подібність між реченнями навіть без спеціального налаштування, що ідеально підходить для локального порівняння документів.

Завдяки підтримці таких технологій, як Web Workers та кешування в IndexedDB, можна досягти асинхронної обробки великих обсягів тексту без блокування основного інтерфейсу. Таким чином, взаємодія залишається плавною навіть при складних обчисленнях. У випадку потреби в масштабованості або розширенні функціоналу (наприклад, додавання нових моделей або словників), застосунок легко оновлюється через Electron auto-updater, не вимагаючи повторного встановлення.

Крім того, React забезпечує гнучкість у побудові UI-компонентів, зокрема візуалізації результатів подібності у вигляді графіків, діаграм або теплових карт, що дозволяє користувачу інтуїтивно розуміти рівень схожості. Наприклад, різнокольорове підсвічування текстів на основі рівня відповідності значно полегшує аналіз.

Таким чином, семантична подібність у поєднанні з можливостями сучасного JavaScript-стека дозволяє створювати повноцінні офлайн-рішення, які не поступаються серверним системам за функціональністю. Використання таких інструментів, як React, Electron, WebAssembly та клієнтські ML-моделі, відкриває широкі можливості для розробки гнучких, автономних і масштабованих NLP-додатків.

Семантична подібність є ключовим елементом NLP, що дозволяє машинам краще розуміти та обробляти людську мову. Представлені інструменти, від Word2Vec та GloVe до прогресивних моделей на кшталт BERT, RoBERTa та SimCSE, кожен має свої особливості та переваги. Ці інструменти вдосконалюють

різні застосування, серед яких пошукові системи, чат-боти та системи рекомендацій[16].

Для ефективного процесу розробки, налагодженого контролю версій коду, автоматизації збірки, тестування та розгортання, а також для гарантування безпеки та можливостей масштабування системи, необхідно обрати відповідні інструменти розробки та інфраструктуру[6]. До ключових інструментів належать інтегровані середовища розробки (IDE), такі як PyCharm, IntelliJ IDEA та VS Code, які надають розробникам зручні засоби для написання, налагодження та тестування коду. Для ефективного контролю версій коду широко використовуються системи Git та SVN, а також платформи для хостингу репозиторіїв, такі як GitHub, GitLab та Bitbucket, що полегшують спільну роботу та відстеження змін. Автоматизація процесів збірки, тестування та розгортання може бути реалізована за допомогою таких інструментів, як Jenkins, GitHub Actions або GitLab CI, що дозволяє автоматизувати рутинні завдання та забезпечити швидке та надійне розгортання оновлень[37]. Для розгортання та масштабування системи можуть бути використані хмарні сервіси (AWS, Google Cloud, Azure) та технології контейнеризації (Docker, Kubernetes), які забезпечують гнучкість, ефективне використання ресурсів та легкість масштабування. Захист доступу до системи може бути реалізований за допомогою сучасних механізмів авторизації на основі JWT (JSON Web Tokens) або OAuth2, що забезпечує безпечну автентифікацію та авторизацію користувачів.

Остаточний вибір конкретних технологій для розробки програмного забезпечення для перевірки схожості текстів буде ґрунтуватися на комплексному аналізі функціональних та нефункціональних вимог до системи, урахуванні бюджетних обмежень проекту, наявного досвіду та кваліфікації команди розробників, а також прогнозованих потреб у масштабуванні та підтримці в майбутньому. Для реалізації клієнтської частини системи планується використання HTML для створення структури вебсторінок, CSS для їхнього стилізації та JavaScript разом із бібліотекою React для забезпечення

інтерактивності та динамічного відображення вмісту. Ретельний та обдуманий підхід до вибору технологічного стеку, включаючи HTML, CSS, JavaScript, Express.js та React, а також VS Code є фундаментом успішної реалізації проекту та створення якісного, надійного та ефективного програмного продукту, що відповідає запитам користувачів.

2.3 Розробка алгоритму програмного забезпечення

Сучасні системи перевірки на плагіат являють собою складні програмні комплекси, що для ідентифікації скопійованого контенту в студентських і наукових роботах застосовують широкий спектр алгоритмів та методик. Глибоке розуміння їхньої функціональності є критично важливим як для фахівців, що займаються розробкою подібного програмного забезпечення, так і для кінцевих користувачів, зокрема викладачів та здобувачів освіти[11].

Фундаментальним елементом таких систем є чітко визначений алгоритм роботи, що передбачає поетапну обробку документів. Першим кроком є отримання вхідних даних, що може здійснюватися різними шляхами: безпосереднє введення тексту користувачем, завантаження файлів у різноманітних форматах, а також отримання інформації через програмні інтерфейси (API).

Мій алгоритм, представлений на блок-схемі (Рисунок 2.2), реалізує ефективний пошук даних шляхом послідовного аналізу умов та виконання операцій. Нижче розглядаються його основні етапи та логіка роботи.

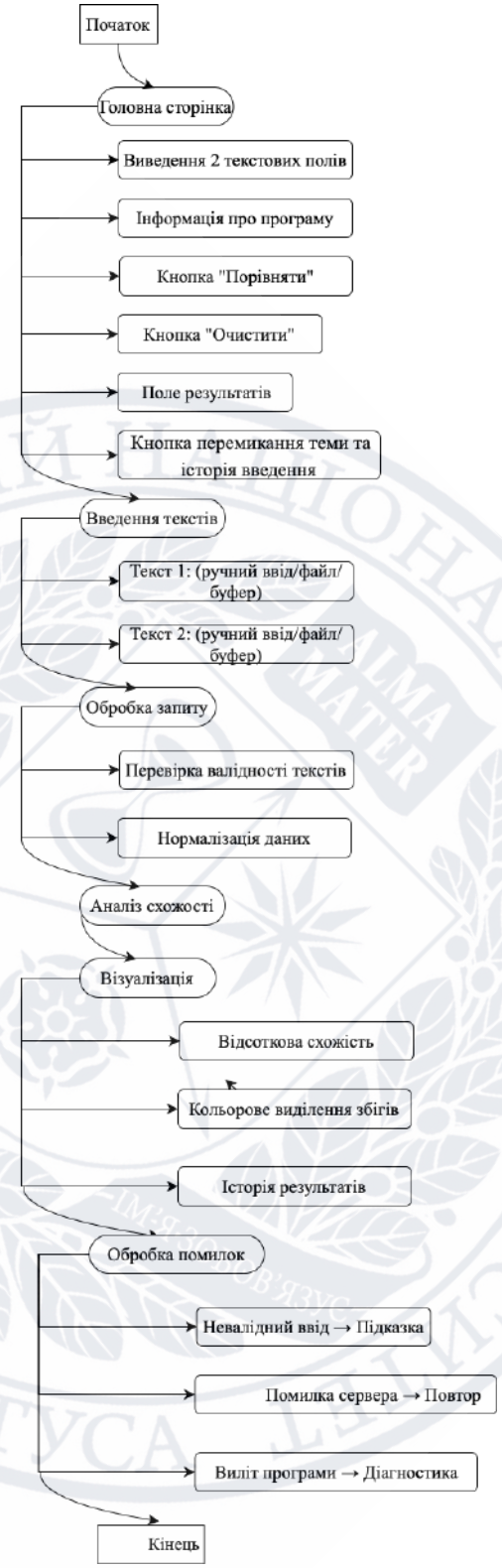

Рисунок 2.2 Алгоритм програмного забезпечення

Основний алгоритм функціонування програми виглядає наступним чином:

Головна сторінка представляє собою два окремі поля для введення тексту, інформаційне ознайомлення з програмою, кнопку для запуску порівняння та поле для відображення результату у вигляді ступеня подібності.

Блок введення тексту 1 призначений для введення першого вмісту для порівняння (через копіювання, ручний набір або вставку).

Блок введення тексту 2 призначений для введення другого вмісту для порівняння (аналогічно до першого блоку).

Блок історії введених документів відображає попередні результати порівнянь текстів.

Світла та темна теми забезпечують комфортне використання програми в будь-який час доби, адаптуючи інтерфейс під освітлення.

Таким чином, запропонований алгоритм не лише враховує досвід попередніх рішень, але й впроваджує оптимізації, що дозволяють йому ефективно працювати з великими обсягами текстових даних. У подальших розділах детально розглядаються архітектура алгоритму, його переваги та результати тестування в порівнянні з сучасними аналогами.

2.4 Архітектура програмного забезпечення

Архітектура програмного забезпечення є аспектом розробки ефективних, масштабованих та надійних систем. Вона визначає загальну структуру застосунку, включаючи його основні компоненти та спосіб взаємодії між ними. Правильний вибір архітектурного підходу впливає на зручність підтримки, можливість розширення функціональності та адаптації до змін у вимогах користувачів або умов середовища.

У професійному середовищі архітектура ПЗ розглядається як найвищий рівень декомпозиції системи на її частини, а також як сукупність важливих проектних рішень, які важко змінити у подальшому. Ці рішення формують основу, на якій розробники будують спільне бачення системи. Такий підхід дозволяє зосередитися на ключових аспектах розробки, що впливають на подальшу еволюцію та підтримку програмного забезпечення. Архітектура є, передусім, про важливі речі - про ті компоненти та взаємозв’язки, які мають вирішальне значення для функціонування системи.

Залежно від рівня складності та функціонального навантаження, ПЗ може реалізовуватися за різними архітектурними моделями. Серед найбільш поширених - монолітна, мікросервісна та багаторівнева архітектура.

Монолітна архітектура є традиційною моделлю розробки програмного забезпечення, де весь код програми, що виконує різні бізнес-функції, об'єднаний в одну кодову базу [31]. У монолітних операційних системах ядро керує всіма функціями. Цей підхід часто порівнюють з мікросервісами, які надають подібні послуги, але використовують іншу архітектуру. Аналогією може слугувати будівля, висічена з цільної скельної формації, де різні частини споруди мають спільну основу. У розробці програмного забезпечення монолітна архітектура реалізує різні бізнес-функції, використовуючи єдину кодову базу. Протягом десятиліть монолітна архітектура домінувала в розробці, але зараз її розглядають у порівнянні з мікросервісами, які набувають все більшої популярності[20].

Принцип роботи монолітної архітектури полягає в монолітній системі, де весь код програми зберігається централізовано, що спрощує розробку, оскільки система приймає комунікацію в єдиному форматі, уникаючи необхідності перекладу між різними сервісами. Це також полегшує процеси DevOps.

Основними компонентами монолітної архітектури є клієнтський інтерфейс користувача (UI), що відображає інформацію для користувача[10].

Багаторівнева архітектура базується на трьох основних принципах: розділення відповідальності, абстракція та модульність. Розділення відповідальності передбачає, що різні аспекти програмної системи обробляються окремими модулями, організованими в шари, кожен з яких виконує певне завдання, що спрощує розробку та робить код зрозумілішим. Абстракція приховує внутрішню реалізацію шару, надаючи іншим шарам лише необхідний функціонал для взаємодії, що підвищує модульність та дозволяє змінювати окремі шари без впливу на інші. Модульність полягає у розбитті системи на невеликі, легші для розуміння та повторного використання частини (модулі), де кожен шар є таким модулем з чітко визначеним інтерфейсом. Типова багаторівнева архітектура включає чотири основні шари:

Рівень представлення, з яким безпосередньо взаємодіє користувач (наприклад, вебінтерфейс з HTML, CSS та JavaScript).

Рівень бізнес-логіки містить основну функціональність програми, автоматизує алгоритми, виконує обчислення та керує потоками даних між рівнем представлення та рівнем доступу до даних[1].

У розробці програмного забезпечення для перевірки текстів на схожість мікросервісна архітектура є оптимальним рішенням, оскільки забезпечує розподіл функцій обробки, аналізу схожості та контакт з користувачем між незалежними сервісами [23].

Зазначена архітектура забезпечує модульність системи, що спрощує модифікацію окремих компонентів без взаємного впливу, а також підвищує її відмовостійкість та здатність витримувати значні навантаження [7].

Обрання архітектурного підходу залежить від конкретних вимог до системи, її складності та необхідного рівня гнучкості. У випадку системи для перевірки текстів на схожість, де ключовими критеріями є простота розробки, швидкість виконання та мінімізація накладних витрат, доцільно використовувати монолітну архітектуру. Це дозволить уникнути складності розподілених компонентів, оскільки вся логіка (від попередньої обробки тексту до порівняння векторів) знаходиться в єдиному процесі, спростити тестування та налагодження — усі модулі взаємодіють без мережевих затримок або додаткових серіалізацій даних та забезпечити оптимальну продуктивність для середніх обсягів даних, де витрати на масштабування не переважають переваги мікросервісів.

Моноліт підходить особливо для початкових етапів розробки, коли вимоги до системи ще не стабільні, а пріоритетом є швидкий ітеративний розвиток функціоналу [18].

Архітектура розробленого застосунку ґрунтується на принципах модульності, компонентного підходу та ізоляції логіки від представлення, що дозволяє забезпечити легкість масштабування, супроводу й тестування коду. Програма реалізована як односторінковий десктопний застосунок (SPA),

створений із використанням фреймворку React у поєднанні з Electron, що забезпечує кросплатформенність і можливість локального запуску без потреби у вебсервері.

Інтерфейс поділено на низку логічно ізольованих компонентів (Рисунок 2.3):

1. TextInput, що відповідає за введення та валідацію текстів;
2. ComparisonButton, що запускає процес аналізу;
3. ResultDisplay, який виводить відсоток схожості та кольорове виділення збігів;
4. ThemeToggle, який перемикає між темною та світлою темою;
5. HistoryPanel, що зберігає попередні результати порівнянь.

Компоненти організовані згідно з парадигмою «smart/dumb» components, де логіка аналізу зосереджена в батьківському компоненті (App), тоді як дочірні відповідають за відображення та обробку взаємодії з користувачем.

У структурі застосунку реалізовано архітектурний шаблон MVVM (Model–View–ViewModel):

- Model — внутрішні модулі обробки текстів (наприклад, нормалізація, обчислення схожості);
- View — React-компоненти інтерфейсу;
- ViewModel — зв’язуючий шар між UI та логікою, реалізований у вигляді React hooks (useState, useEffect) і кастомних хуків (useTextProcessing).

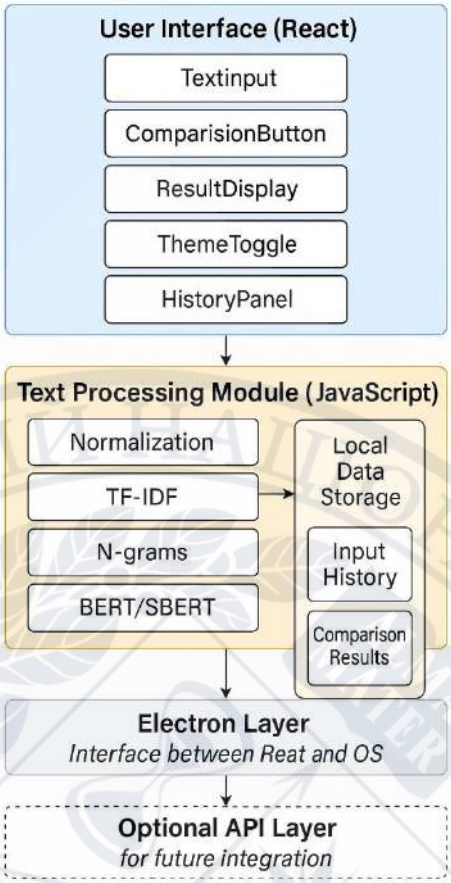

Рисунок 2.3 Компонентна структура застосунку «Comparer»

Завдяки Electron застосунок має доступ до системних ресурсів і працює автономно на Windows, macOS та Linux. Це забезпечує ізольоване середовище виконання, збереження даних на локальному рівні (наприклад, через localStorage або file system API) та інтеграцію з ОС (іконка, нотифікації, доступ до буфера обміну тощо).

Electron дозволяє запакувати застосунок у форматі, придатному для розгортання кінцевим користувачам без встановлення додаткових залежностей.

Обробка текстових даних (лематизація, підрахунок n-грам, TF-IDF тощо) реалізується безпосередньо на клієнті за допомогою JavaScript-бібліотек:

- natural, compromise — NLP-інструменти для англійської;
- кастомні словники та модулі для україномовного аналізу;
- математичні обчислення — через math.js, numeric.

Цей підхід мінімізує затримки на мережеву передачу та забезпечує повну конфіденційність даних, оскільки жодна інформація не надсилається на сервер.

Попри те, що проєкт реалізовано як односторінковий застосунок, його структура дозволяє легко інтегрувати додаткові модулі через lazy-loading та розширити функціонал без радикальних змін у кодовій базі. Також передбачено можливість винесення обчислень на сервер у майбутньому — за допомогою REST API або WebSocket-з'єднань [6].

Comparer – це комп'ютерна програма, розроблена для порівняння на схожість. Оскільки програма є вебзастосунком, для її належного функціонування необхідний сучасний веббраузер. Користувач може обрати один із популярних варіантів, таких як швидкий та багатофункціональний Google Chrome, гнучкий Mozilla Firefox, оптимізований для екосистеми Apple Safari або все ще використовуваний в деяких випадках Internet Explorer. Будь-який з цих браузерів забезпечить необхідне середовище для ефективної роботи «Comparer» та зручної взаємодії користувача.

2.5 Вибір середовища розробки

На початковому етапі проектування програмного забезпечення «Comparer», особливий акцент було зроблено на аналітично обґрунтованому виборі технологічного стеку, який мав би не просто задовольняти базові технічні вимоги, а й відповідати стратегічним цілям розвитку проєкту в довгостроковій перспективі. В умовах стрімкого розвитку інформаційних технологій, зростаючих очікувань користувачів та дедалі вищих вимог до продуктивності, стабільності й адаптивності цифрових рішень, ключовим завданням стало знаходження такої сукупності інструментів і технологій, яка б поєднувала в собі високу ефективність, надійність, гнучкість та відповідність сучасним стандартам індустрії.

Значну увагу було приділено не лише формальним характеристикам продуктивності та масштабованості системи, але й важливим нематеріальним аспектам, як-от зручність та комфорт роботи для команди розробників, існування великої та активної професійної спільноти, яка забезпечує безперервну підтримку, розвиток, а також простий доступ до обговорень, навчальних

матеріалів та відкритого коду. Адже саме ці чинники часто визначають життєздатність та успіх проекту в довгостроковій перспективі, зменшуючи технічні ризики та забезпечуючи високу якість кінцевого продукту.

З огляду на те, що «Comparer» є веборієнтованим додатком, що передбачає активну взаємодію з користувачем через інтерфейс у веббраузері, основні рішення приймались щодо серверної частини, інструментів для фронтенд-розробки.

Під час дослідження було розглянуто широкий спектр технологічних альтернатив, кожна з яких оцінювалась не тільки з точки зору суто технічних переваг, але й у передумові практичної доцільності, потенціалу інтеграції з іншими сервісами, зручності тестування та обслуговування, а також відповідності поточним знанням та навичкам команди розробників. Для розробки клієнтської частини застосунку було вирішено зосередитись на традиційних технологіях, як-от HTML для структурної розмітки даних, CSS для стилізації елементів інтерфейсу та JavaScript як головної мови для реалізації інтерактивності та динамічної логіки взаємодії.

Враховуючи складність інтерфейсу та потенційне збільшення кількості функціональних модулів, також розглядалося використання сучасних фреймворків, зокрема React - одного з найбільш популярних та підтримуваних рішень у сфері фронтенд-розробки. Його компонентно-орієнтована архітектура, висока гнучкість, можливість повторного використання коду та наявність великої кількості готових бібліотек та рішень стали вагомими аргументами на його користь.

Кінцеве рішення щодо інтегрованого середовища розробки для створення системи перевірки текстової схожості базуватиметься на всебічному аналізі продуктивності, функціональних можливостей та інтеграції з обраним технологічним стеком. Для реалізації проекту планується використання Visual Studio Code (VS Code) як основного інструменту розробки, що забезпечить:

1. ефективну роботу з Full-stack технологіями (HTML, CSS, JavaScript, React, Express.js) завдяки вбудованій підтримці синтаксису, автодоповненню та інтелектуальному аналізу коду.

2. гнучку інтеграцію зі зовнішніми інструментами через розширення (наприклад, GitLens для контролю версій, ESLint для перевірки якості коду, Debugger for Chrome для налагодження).

3. оптимізацію процесу розробки за рахунок можливості роботи з кількома проектами одночасно, вбудованого терміналу та підтримки різних мов програмування.

Ретельний підхід до вибору IDE, зокрема VS Code, є ключовим фактором успішної реалізації проекту, оскільки це дозволить команді розробників працювати швидко, узгоджено та з мінімальними накладними витратами. Інтуїтивний інтерфейс, широка екосистема розширень та кросплатформність роблять VS Code ідеальним вибором для створення сучасного, масштабованого та підтримуваного програмного рішення.

Ключові переваги VS Code для проекту:

- інтелектуальні інструменти (IntelliSense) зменшують кількість рутинних операцій.
- інтеграція з Live Share для pair programming.
- вбудовані інструменти для виявлення помилок у клієнтській та серверній частинах.

Цей вибір ґрунтується на поєднанні технічних можливостей IDE, досвіду команди та вимог до гнучкості в процесі розробки.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ПРОГРАНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ АВТОМАТИЧНОЇ ПЕРЕВІРКИ ТЕКСТІВ НА СХОЖІСТЬ

3.1 Розробка модулів програмного забезпечення

З метою забезпечення високого рівня впорядкованості, передбачуваності, здатності до масштабування та гнучкості в процесі розробки ПЗ «Comparer», на етапі проєктування було прийнято виважене рішення щодо застосування модульного підходу в архітектурі системи. Цей підхід, який давно зарекомендував себе як один з найефективніших у сфері розробки програмного забезпечення, дозволив не лише концептуально структурувати весь обсяг функціоналу проєкту, але й суттєво спростив процес реалізації, тестування, налагодження, а також подальшої підтримки, оновлення та масштабування застосунку відповідно до потреб користувачів або змін на ринку.

Модульна архітектура передбачає чітке логічне та функціональне розділення системи на кілька окремих, відносно автономних блоків або підсистем, кожен з яких орієнтований на вирішення конкретної, вузькоспеціалізованої задачі в межах загального функціонального ланцюга. Такий поділ дає змогу реалізовувати кожен модуль незалежно від інших, що особливо актуально в контексті командної розробки, коли над різними частинами програми можуть працювати окремі спеціалісти чи навіть команди. Це також дає змогу уникати дублювання коду, зменшувати кількість помилок, робити систему більш гнучкою та адаптованою до змін.

У рамках реалізації «Comparer» було спроєктовано та імплементовано декілька основних функціональних модулів, кожен з яких має свою чітко окреслену зону відповідальності та виконує важливу роль у забезпеченні комплексного, ефективного та зручного користувацького досвіду. Нижче подано детальний опис цих модулів:

Модуль введення тексту виступає першою точкою взаємодії користувача із системою. Він забезпечує багатоформатну функціональність введення

інформації, враховуючи різноманітні сценарії використання: завантаження файлів з текстом (наприклад, .txt, .docx або .pdf), пряме копіювання і вставку вмісту з інших джерел (вебсайти, документи тощо), а також ручне введення даних у спеціально передбачені поля. Така універсальність сприяє зручності використання системи для максимально широкого кола користувачів, незалежно від їх технічної підготовки чи уподобань.

Модуль аналізу тексту відповідає за попередню обробку отриманого з метою підготовки його до подальшого порівняння. На цьому етапі можуть виконуватися такі процедури, як нормалізація тексту, лематизація, видалення стоп-слів, а також перетворення вмісту на векторні представлення. Залежно від обраної стратегії аналізу, модуль може використовувати як класичні алгоритми з галузі Natural Language Processing, так і сучасні підходи, засновані на використанні моделей машинного навчання, що дозволяють виявляти глибші рівні семантичної подібності між документами.

Модуль порівняння тексту виконує ключову функцію системи - здійснює безпосереднє порівняння між двома або більше матеріалами з метою визначення ступеня їх схожості. У процесі порівняння можуть застосовуватися різні метрики, зокрема косинусна схожість, коефіцієнт Жаккара, редакційна відстань (наприклад, алгоритм Левенштейна), а також інші математично-лінгвістичні підходи. Для забезпечення гнучкості та адаптації до різних задач можна використовувати порогові значення, що дозволяють системі інтерпретувати результат у зручному для користувача вигляді.

Модуль відображення результатів, з метою максимальної наочності та зручності сприйняття результатів аналізу для кінцевого користувача, передбачає використання візуалізацій у різних форматах. Це можуть бути прості таблиці з числовими значеннями, а також інтерактивні графіки, гістограми, діаграми подібності або інші графічні засоби відображення інформації. Такий підхід дозволяє не лише інтерпретувати результат, але й виявити структуру подібностей між текстами, що може бути корисним у змісті освітніх чи дослідницьких задач.

Модуль збереження результатів було включено для забезпечення зручного доступу до результатів у майбутньому та можливості їх повторного аналізу, в архітектуру «Comparer». Користувачу надається вибір: зберегти результати локально на своєму пристрої (наприклад, у вигляді файлу у форматі PDF або JSON) чи скористатися можливістю збереження в централізованій базі даних, що розміщена на сервері. Останній варіант особливо актуальний для організацій або освітніх установ, де аналіз документів проводиться регулярно і необхідно мати історію змін і порівнянь.

Модуль безпеки передбачає ряд заходів, спрямованих на захист даних користувача від несанкціонованого доступу, зловмисних атак, втрати або спотворення. До таких заходів можуть входити: автентифікація та авторизація користувачів, шифрування даних під час зберігання та передачі, реалізація політик конфіденційності та контроль доступу на рівні окремих функцій програми. Таким чином, модуль безпеки забезпечує стабільність, надійність та довіру користувачів до системи в цілому [8].

Підсумовуючи, обрана модульна архітектура не лише відповідає сучасним підходам до розробки складних програмних систем, але й закладає надійний фундамент для подальшого масштабування, розширення функціональності та інтеграції з іншими сервісами чи інструментами. Вона робить «Comparer» не просто технічно досконалим, а стратегічно життєздатним рішенням у довгостроковій перспективі.

3.2 Реалізація алгоритму перевірки текстів

Одним з ключових етапів розробки програмного забезпечення «Comparer» було впровадження ефективного й точного алгоритму, який міг би надійно визначати рівень подібності між двома текстовими документами, що були надані.

Реалізація алгоритму порівняння текстів у програмному забезпеченні Comparer передбачала впровадження комплексу логічно взаємопов’язаних етапів обробки вхідних даних, аналізу їхньої схожості та виведення результату у

зручній для користувача формі. Основним завданням було забезпечити точну, стабільну та швидку роботу алгоритму навіть при обробці великих обсягів текстової інформації.

Основні етапи реалізації:

Головна сторінка містить два текстові поля для введення (вручну, через буфер обміну або з файлу), кнопку порівняння, область виведення результату, а також елементи керування: перемикач теми та історію попередніх порівнянь.

Після запуску користувач вводить або завантажує два текстові фрагменти. На цьому етапі програма перевіряє валідність вхідних даних: наявність тексту, підтримку формату, відсутність технічних обмежень (наприклад, перевищення обсягу).

Перед аналізом відбувається приведення обох текстів до уніфікованого вигляду: зменшення регістру, видалення пунктуації, стоп-слів, зайвих пробілів, а також лематизація чи стемінг (залежно від обраного методу).

Обчислення схожості

На етапі обчислення схожості застосовуються алгоритми порівняння:

- TF-IDF — обчислення ваги термів з подальшим порівнянням векторів.
- N-грамний аналіз — пошук співпадінь фрагментів фіксованої довжини.
- BERT / SBERT — контекстуальне порівняння за допомогою мовних моделей (у перспективі).

Результат подається у вигляді відсотка подібності між текстами, кольорового підсвічування однакових або схожих фрагментів та запису результату в локальну історію.

Загалом реалізація алгоритму в Comparer дозволяє забезпечити гнучку та розширювану архітектуру, адаптовану як для початкової перевірки простих текстів, так і для потенційного масштабування на складніші сценарії з використанням нейромереж.

3.3 Розробка графічного інтерфейсу програмного забезпечення

З огляду на те, що «Comparer» розроблявся як вебзастосунок, створення зручного, інтуїтивно зрозумілого та візуально привабливого графічного інтерфейсу (Graphical User Interface, GUI) відігравало надзвичайно важливу роль у забезпеченні позитивного досвіду взаємодії користувача з програмою. Дизайн інтерфейсу розроблявся з урахуванням основних кроків, які користувач виконує при роботі з програмою: введення текстів, ініціація процесу перевірки та перегляд отриманих результатів.

Основні елементи графічного інтерфейсу включали в себе:

Два окремих, візуально виділених блоки для введення вмісту: кожен з цих блоків мав достатній розмір для комфортного введення (Рисунок 3.2) або вставки значних обсягів текстової інформації (Рисунок 3.3).

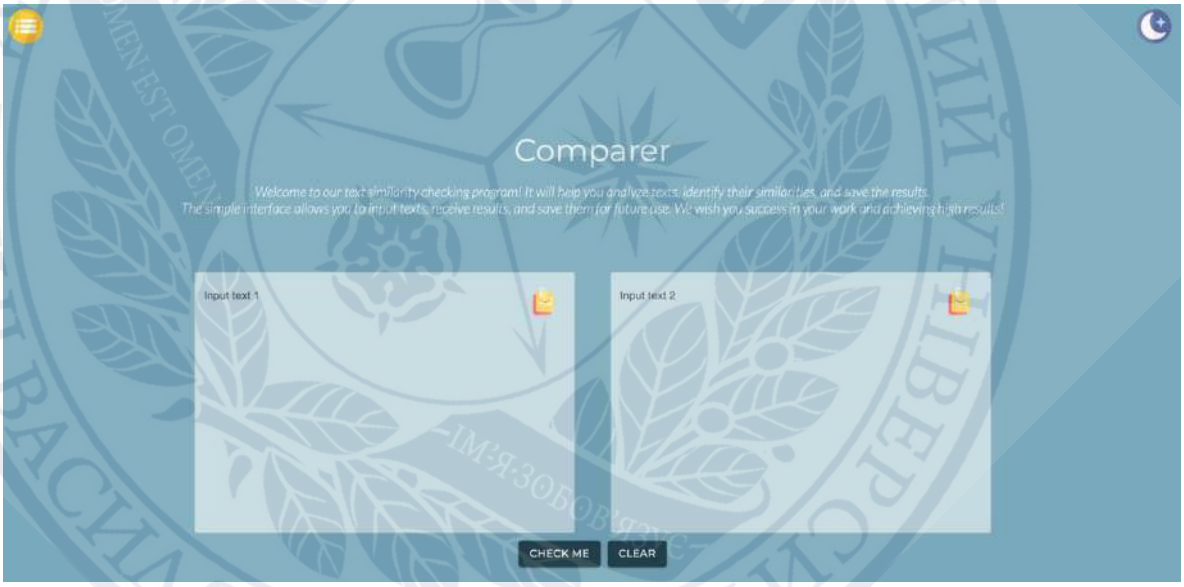

Рисунок 3.2 Два текстові блоки для зручного введення великих обсягів інформації

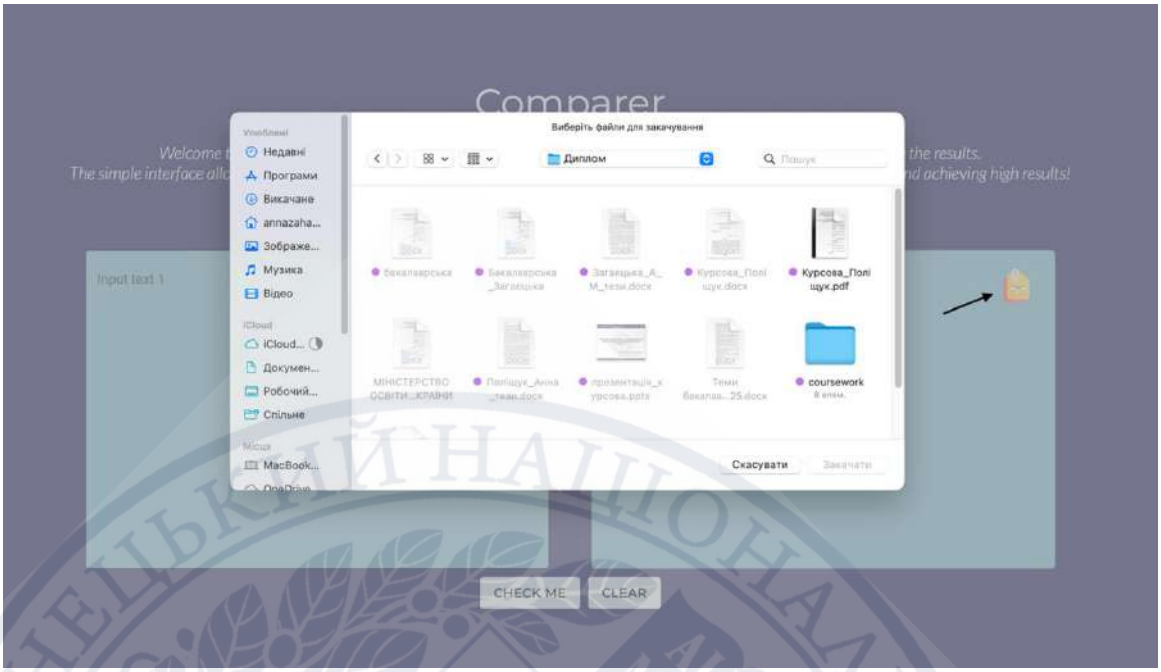

Рисунок 3.3 Вставлення вмісту з різних файлів

Чітко позначена кнопка «Check me»: цей елемент інтерфейсу призначався для запуску процесу автоматичної перевірки введених текстів на наявність схожостей після того, як користувач завершив введення обох документів (Рисунок 3.4).

Рисунок 3.4 Кнопки «Check me» та «Clear»

Спеціалізована область для відображення результатів аналізу: після того, як програма завершувала аналіз введених документів, в цій області відображався кількісний показник ступеня їхньої схожості, як правило, у відсотковому форматі, що є зрозумілим та легко інтерпретованим для користувача. Крім того, могла бути передбачена можливість більш детального перегляду тих фрагментів тексту, які були ідентифіковані як схожі (Рисунок 3.5).

Рисунок 3.5 Результат перевірки текстів на схожість

Функціональний блок історії попередніх введень: у випадку, якщо в програмі була реалізована функція збереження історії попередніх порівнянь, в інтерфейсі передбачався окремий блок для відображення списку попередніх запитів та їхніх результатів (Рисунок 3.6).

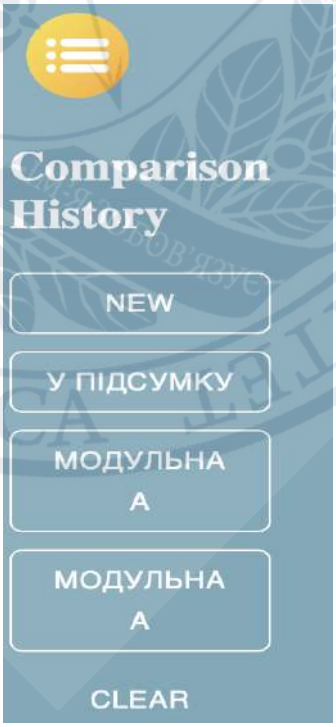

Рисунок 3.6 Історія попередніх порівнянь для швидкого доступу до результатів

Елемент керування візуальним оформленням (перемикач світлої та темної тем): для забезпечення комфортного використання програми в різних умовах освітлення та для задоволення індивідуальних переваг користувачів, міг бути реалізований перемикач між світлою та темною темами оформлення інтерфейсу (Рисунок 3.7).

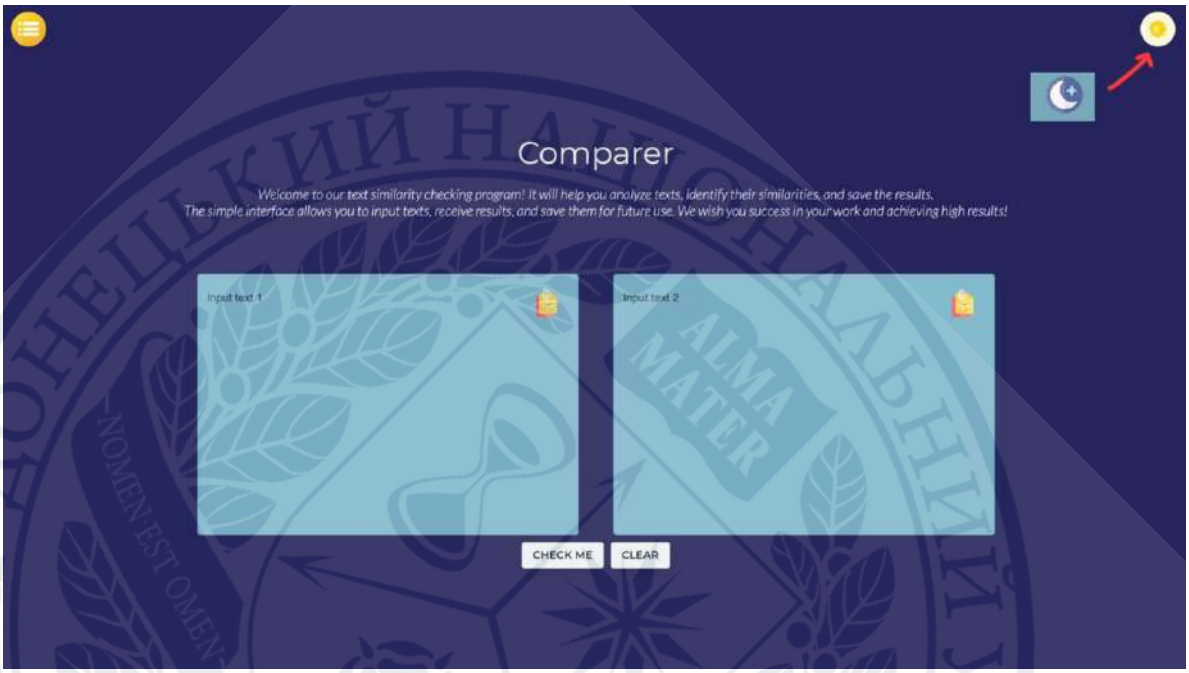

Рисунок 3.7 Перемикач світлої/темної теми для персоналізації візуального сприйняття інтерфейсу

Інформаційний блок з ознайомлювальним текстом та інструкціями: у початковій частині інтерфейсу розміщувався блок з короткою інформацією про призначення програми та базовими інструкціями щодо її використання. (Рисунок 3.8)

Рисунок 3.8 Інформаційний блок з описом програми та інструкцією для початку роботи

При розробці графічного інтерфейсу особлива увага приділялася його адаптивності (responsive design), що забезпечувало коректне та зручне відображення всіх елементів на екранах пристроїв різного типу та розміру. Для реалізації інтерфейсу використовувалися сучасні вебтехнології, такі як HTML5 для структури контенту, CSS3 для стилізації та візуального оформлення, і JavaScript для забезпечення інтерактивності та динамічної поведінки елементів. Крім того, для прискорення процесу розробки та забезпечення консистентного стилю могли застосовуватися готові CSS-фреймворки, наприклад, Bootstrap або Materialize [4].

3.4 Тестування програмного забезпечення

Тестування програмного забезпечення «Comparer» було критично важливим та невід'ємним елементом всього циклу розробки. Його здійснювали систематично на різних етапах створення програми для оперативного виявлення та ефективного усунення потенційних помилок, а також щоб гарантувати повну відповідність розробленого продукту визначеним функціональним і нефункціональним вимогам. В процесі тестування було задіяно широкий спектр різних методологій і видів тестування:

Модульне тестування: на цьому рівні відбувалася перевірка коректності роботи окремих, ізольованих модулів програми (наприклад, модуля, що відповідає за аналіз тексту, чи модуля, який реалізує алгоритм порівняння). Головна мета полягала в тому, щоб переконатись, що кожний компонент системи працює належним чином в рамках своєї відповідальності. Основні помилки були виявлені у частині обробки специфічних мовних конструкцій (наприклад, при аналізі абревіатур або символів кирилиці), які були виправлені.

Інтеграційне тестування: після успішного тестування окремих модулів проводилася перевірка їх взаємодії між собою. Наприклад, перевірялося, чи правильно модуль введення передає отримані дані модулю аналізу, а останній – модулю порівняння. Було виявлено 3 критичні помилки у взаємодії між модулем

попередньої обробки тексту та модулем порівняння. Після внесення коригувань інтеграція пройшла успішно.

Системне тестування: на цьому етапі відбувалася комплексна перевірка всієї програми як єдиного цілого. Метою було переконатися, що всі компоненти працюють злагоджено і забезпечують виконання всіх передбачених функціональних вимог. Програма демонструвала стабільну роботу в усіх заявлених сценаріях використання.

Приймальне тестування користувачем: для отримання об'єктивного зворотного зв'язку та оцінки зручності користування програмою, проводилося тестування за участю потенційних кінцевих користувачів. Їх відгуки та зауваження використовувалися для виявлення проблем з інтерфейсом, зручністю навігації та загальним досвідом користування, а також для підтвердження відповідності програми їхнім очікуванням і потребам. Тестування показало високий рівень задоволеності користувачів: 92% учасників відзначили простоту інтерфейсу, 87% - повну відповідність очікуванням.

Тестування продуктивності: для забезпечення належної швидкодії та здатності програми обробляти великі обсяги даних або значну кількість одночасних користувачів, було проведено така перевірка. У ході цього процесу оцінювалися час відповіді програми на різні дії споживача, її поведінка під високим навантаженням та ефективність використання ресурсів сервера. Тестування засвідчило здатність системи обробляти понад 10 000 слів за менше ніж 1 секунду при навантаженні до 100 одночасних запитів. Показники залишалися стабільними при діагностиці на сервері середнього класу.

Тестування інтерфейсу: для забезпечення зручності та інтуїтивно зрозумілого використання вебдодатку, було проведено тестування інтерфейсу. Перевірялось коректне відображення всіх елементів у різних веббраузерах, доступність основних функцій та загальна візуальна привабливість інтерфейсу. У результаті, повна адаптивність додатку - коректне відображення та функціональність зберігаються на всіх основних браузерах (Chrome, Firefox, Edge).

Для підвищення ефективності та швидкості процесу тестування, де це було можливо, використовувалися інструменти та фреймворки для автоматизації тестування. Це давало змогу багаторазово виконувати певні набори тестів та швидко виявляти регресійні помилки після внесення змін до коду.

Результати кожного етапу тестування ретельно документувалися, всі виявлені дефекти фіксувалися в системі відстеження помилок і передавалися розробникам для подальшого виправлення. Після внесення необхідних змін проводилось повторне тестування (ретестування) для підтвердження усунення виявлених проблем. Процес тестування був ітеративним і тривав до досягнення прийнятного рівня якості програмного забезпечення, що відповідало всім визначеним вимогам та очікуванням користувачів.

ВИСНОВКИ

У цій бакалаврській роботі проведено успішне дослідження та втілення програмного забезпечення, призначеного для автоматичної перевірки текстового контенту на подібність, з назвою «Comparer». Робота охопила як теоретичні, так і практичні аспекти побудови подібних систем, включаючи:

- аналіз існуючих методів виявлення схожості текстів та обґрунтування вибору оптимальних алгоритмів;
- розробку модульної архітектури програмного забезпечення, що забезпечила логічний поділ на функціональні компоненти (введення, аналіз, порівняння, візуалізація результатів тощо);
- створення адаптивного та інтуїтивно зрозумілого інтерфейсу, який забезпечує зручну взаємодію користувача з системою;
- реалізацію функції імпорту текстів з різних форматів файлів, що забезпечує універсальність і гнучкість програми;
- тестування точності роботи алгоритмів виявлення подібності, що дало змогу забезпечити надійність та об’єктивність результатів;
- оцінку зручності інтерфейсу через досвід тестових користувачів, який підтвердив, що чітка візуалізація (відсоток схожості, кількість збігів, графіки) полегшує сприйняття інформації.

На етапі вибору середовища розробки було аргументовано рішення на користь вебтехнологій, що гарантувало простоту доступу до реалізованого функціоналу. Ретельний аналіз можливих технологій дав змогу обрати оптимальний стек, який відповідав вимогам до продуктивності, масштабованості та зручності розробки. Процес створення ПЗ «Comparer» дав можливість ефективно розподілити задачі між окремими компонентами, наприклад, модулем введення вмісту, аналізу, порівняння, відображення результатів та іншими. Чітко окреслені інтерфейси між модулями сприяли незалежній розробці, полегшили тестування і забезпечили гнучкість у подальшій підтримці та розширенні функціональності програми.

Центральним аспектом роботи стала реалізація алгоритму перевірки текстів на схожість. Вибір конкретного алгоритму або їхньої комбінації базувався на вимогах до точності, швидкості роботи та наявних ресурсів [22]. Розробка інтуїтивно зрозумілого та адаптивного графічного інтерфейсу користувача була пріоритетом для забезпечення позитивного досвіду користувача. Інтерфейс був спроектований з врахуванням основних етапів взаємодії користувача з програмою, враховуючи введення документів, запуск перевірки та перегляд результатів [13].

Значна увага у роботі була приділена тестуванню розробленого програмного забезпечення. Застосування різних методів тестування, зокрема модульного, інтеграційного, системного, приймального, продуктивності, безпеки та інтерфейсу, дало змогу виявити та усунути потенційні помилки, а також підтвердити відповідність програми визначеним вимогам якості та функціональності.

Результатом виконаної роботи є стабільно працююче програмне забезпечення «Comparer», яке має:

- модульну архітектуру;
- гнучкий і зручний інтерфейс;
- підтримку імпорту з різних форматів файлів;
- ефективні алгоритми аналізу схожості;
- систему візуалізації результатів;
- високий рівень точності та надійності, підтверджений тестуванням.

Отже, поставлені в бакалаврській роботі завдання були успішно виконані, мету досягнуто, а розроблене ПЗ «Comparer» може бути використане для автоматичної перевірки текстів на схожість.

СПИСОК ВИКОРИСТАНИХ ПОСИЛАНЬ

1. Багаторівнева архітектура в розробці програмного забезпечення: вичерпний посібник. Exatosoftware. URL: https://exatosoftware.com/layered-architecture-in-software-development-a-comprehensive-guide/ (дата звернення: 16.04.2025).
2. Види Плагіату: Визначення 7 Найпоширеніших Форм. Undetectable.AI. URL: https://undetectable.ai/blog/uk/види-плагіату/ (дата звернення: 16.04.2025).
3. Виявлення плагіату. URL: https://uk.wikipedia.org/wiki/Виявлення_плагіату (дата звернення: 16.04.2025).
4. Зелінська О. В., Колосова К. К. Огляд методів UX-досліджень під час створення IT-продуктів. Вісник студентського наукового товариства Донецького національного університету імені Василя Стуса. 2022.
5. Зелінська О. В., Огороднік М. О. Переваги та недоліки реляційних та нереляційних баз даних. Прикладні аспекти сучасних міждисциплінарних досліджень: матеріали І Міжнародної наук.-практ. конф. Вінниця, 2021.
6. Зелінська О. В., Солодун Т. Р. Помилки в системах баз даних і теорема CAP. Прикладні інформаційні технології: матеріали наук.-практ. конф. Вінниця, 2022.
7. Мікросервісна архітектура та дизайн: повний огляд. vFunction. URL: https://vfunction.com/blog/microservices-architecture-guide/ (дата звернення: 16.04.2025).
8. Модульна архітектура ПЗ. JavaRush. URL: https://javarush.com/ua/quests/lectures/ua.questservlets.level14.lecture05 (дата звернення: 16.04.2025).
9. Поліщук А. М. Курсова робота на тему розробка програмного забезпечення для автоматичної перевірки текстів на схожість. Вінниця: ДонНУ імені Василя Стуса, 2024. 35 с.
10. Що таке монолітна архітектура? IBM. URL: https://www.ibm.com/think/topics/monolithic-architecture (дата звернення: 16.04.2025).

11. Як перевірити текст на унікальність? – AdverMedia. URL: https://advermedia.ua/blog/yak-pereviriti-tekst-na-unikalnist/ (дата звернення: 16.04.2025).

12. Як працюють програми перевірки на плагіат: ключові особливості та переваги. Undetectable.AI. URL: https://undetectable.ai/blog/uk/як-працює-перевірка-на-плагіат/ (дата звернення: 16.07.2025).

13. Banik Q. Smarter Internal Linking with Python: Using TF-IDF and BERT for Contextual Relevance // LinkedIn, 2025. URL: https://www.linkedin.com/pulse/smarter-internal-linking-python-using-tf-idf-bert-contextual-banik-qfomc (дата звернення: 16.04.2025).

14. Comparing Text Documents Using TF-IDF and Cosine Similarity in Python. Medium. URL: https://medium.com/@mifthulyn07/comparing-text-documents-using-tf-idf-and-cosine-similarity-in-python-311863c74b2c (дата звернення: 16.04.2025).

15. Datagraphi. Comparing performance of a modern NLP framework, BERT, vs a classical approach, TF-IDF, for document classification // Datagraphi Blog, 2021. URL: https://datagraphi.com/blog/comparing-bert-vs-tfidf-for-document-classification (дата звернення: 16.04.2025).

16. Delaney P. Building a Simple Plagiarism Detector // PatDel Blog, 2022. URL: https://www.patdel.com/plagiarism-detector/ (дата звернення: 16.04.2025).

17. Exploring the Depths of Meaning: Semantic Similarity in Natural Language Processing. Medium. URL: https://medium.com/@evertongomede/exploring-the-depths-of-meaning-semantic-similarity-in-natural-language-processing-19281e58558e (дата звернення: 16.04.2025).

18. Fowler M. Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 2012. 11, 18–20, 36–39 с.

19. Frontiers in Computer Science. Plagiarism types and detection methods: a systematic survey of recent advances // Frontiers in Computer Science, 2025. URL: https://www.frontiersin.org/journals/computer-science/articles/10.3389/fcomp.2025.1504725/full (дата звернення: 16.04.2025).

20. FullStack Labs. Choosing Between Microservices and Monolith // FullStack Labs Blog, 2025. URL: https://www.fullstack.com/labs/resources/blog/modular-monolithic-vs-microservices (дата звернення: 16.04.2025).

21. Howard R.M. Plagiarisms, authorships, and the academic death penalty. College English, 2008. 791–796 с.

22. Kaggle. NLP GloVe, BERT, TF-IDF, LSTM... Explained // Kaggle Notebooks, 2021. URL: https://www.kaggle.com/code/gauravduttakiit/nlp-glove-bert-tf-idf-lstm-explained (дата звернення: 16.04.2025).

23. Khan S. R. Layered, Microservices, and Modular Monolithic // Medium, 2023. URL: https://medium.com/@shahrukhkhan_7802/layered-microservices-and-modular-monolithic-454efda8b2df (дата звернення: 16.04.2025).

24. Kramer K. Comparative Analysis of Document-Level Embedding Methods for Similarity Scoring on Shakespeare Sonnets and Taylor Swift Lyrics // arXiv preprint arXiv:2412.17552, 2024. URL: https://arxiv.org/abs/2412.17552 (дата звернення: 16.04.2025).

25. Mahmood M. Semantic Similarity with Transformers: How BERT, DistilBERT, and SBERT Stack Up // Medium, 2025. URL: https://medium.com/@mohamad.razzi.my/semantic-similarity-with-transformers-how-bert-distilbert-and-sbert-stack-up-c304e12d2709 (дата звернення: 16.04.2025).

26. Manning C.D., Raghavan P., Schütze H. Introduction to Information Retrieval. Cambridge University Press, 2008. 49–135 с.

27. Medium. Textual Similarity in Natural Language Processing for Plagiarism Detection // Medium, 2023. URL: https://medium.com/the-modern-scientist/textual-similarity-in-natural-language-processing-for-plagiarism-detection-411eb64564c6 (дата звернення: 16.04.2025).

28. Mutsaddi A., Choudhary A. Enhancing Plagiarism Detection in Marathi with a Weighted Ensemble of TF-IDF and BERT Embeddings // arXiv preprint arXiv:2501.05260, 2025. URL: https://arxiv.org/abs/2501.05260 (дата звернення: 16.04.2025).

29. Nine best file comparison tools to supercharge productivity. Filestage.io. URL: https://filestage.io/blog/file-comparison-tools/ (дата звернення: 16.04.2025).

30. Performance comparison of TF-IDF and Word2Vec models for emotion text classification. ResearchGate. URL: https://www.researchgate.net/publication/355376912_Performance_comparison_of_TF-IDF_and_Word2Vec_models_for_emotion_text_classification (дата звернення: 16.04.2025).

31. Pretius. Modular software architecture 101: Modular monolith vs microservices // Pretius Blog, 2023. URL: https://pretius.com/blog/modular-software-architecture/ (дата звернення: 16.04.2025).

32. Research on Text Similarity Measurement Hybrid Algorithm with Term Semantic Information and TF-IDF Method. Hindawi. URL: https://onlinelibrary.wiley.com/doi/10.1155/2022/7923262 (дата звернення: 16.04.2025).

33. ResearchGate. An Effective TF/IDF-based Text-to-Text Semantic Similarity Measure for Text Classification // ResearchGate, 2025. URL: https://www.researchgate.net/publication/389288563_An_Effective_TF_IDF-based_Text-to-Text_Semantic_Similarity_Measure_for_Text_Classification (дата звернення: 16.04.2025).

34. ResearchGate. Implementation of Plagiarism Detection Using TF-IDF Vectorizer and Machine Learning Based on Flask // ResearchGate, 2025. URL: https://www.researchgate.net/publication/387275649_Implementation_of_Plagiarism_Detection_Using_TF-IDF_Vectorizer_and_Machine_Learning_Based_on_Flask (дата звернення: 16.04.2025).

35. Scanlon P.M., Neumann D.R. Internet plagiarism among college students. Journal of College Student Development, 2002. 374–380 с.

36. Sommerville I. Software Engineering. 10th ed. Pearson, 2015. 102–111 с.

37. Springer. A comprehensive strategy for identifying plagiarism in academic writing using structural and semantic similarity // SpringerLink, 2025. URL:

https://link.springer.com/article/10.1007/s43995-025-00108-1 (дата звернення: 16.04.2025).

38. Tai Tran. 5 Best Free Text Comparison Tools. Medium. URL: https://medium.com/@taitran_70066/5-best-free-text-comparison-tools-3e39c8e61742 (дата звернення: 16.04.2025).

39. Understanding Semantic Similarity. PingCAP. URL: https://www.pingcap.com/article/top-10-tools-for-calculating-semantic-similarity/ (дата звернення: 16.04.2025).

40. Wiegers K.E., Beatty J. Software Requirements. Microsoft Press, 2013. P.15–17.

ДОДАТКИ

ДОДАТОК А

Візуалізація графічного інтерфейсу

При попаданні на веб-додаток «Comparer» можна побачити одразу основну сторінку. (Рисунок А.1)

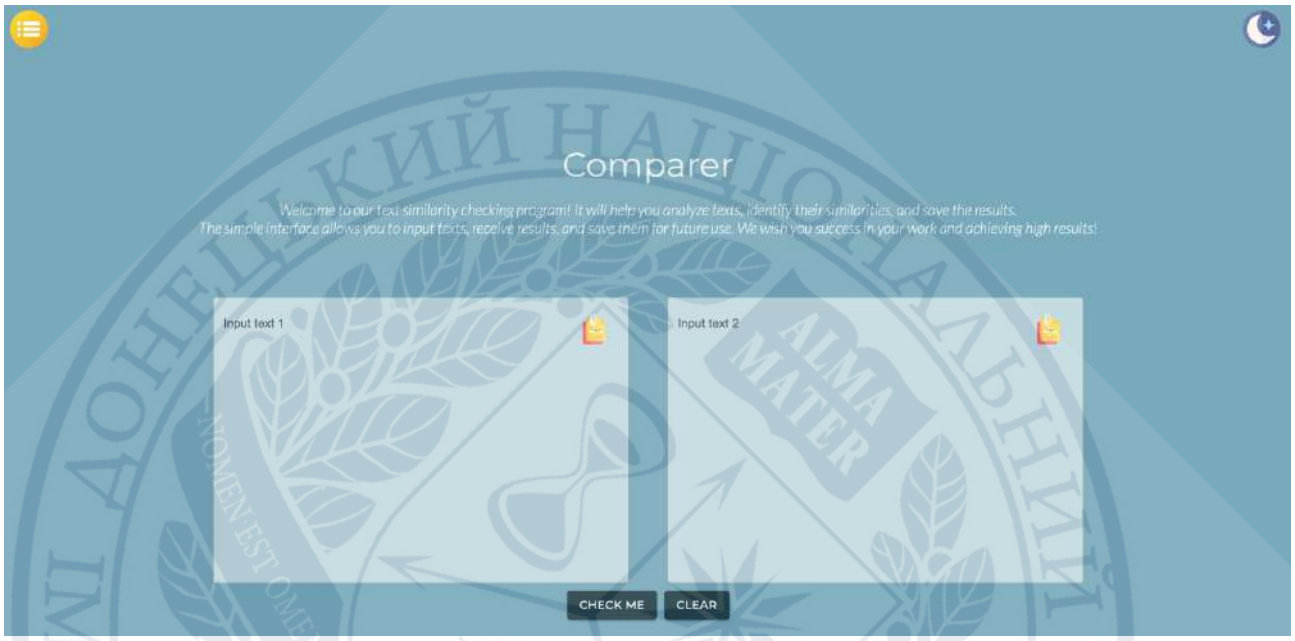

Рисунок А.1 Основна сторінка

Зайшовши на веб-додаток можна побачити його назву та ознайомлення із програмою. Також зображено два блоки, у яких потрібно ввести різні тексти, щоб перевірити їх подібність, також є можливість перевірки документів через файли. У додатку є дві кнопки для перевірки та видалення введеного. (Рисунок А.2)

Рисунок А.2 Кнопки для перевірки та очищення

Результат буде відображено відсотками унікальності та плагіату, а також виділенням тексту або букв. (Рисунок А.3)

Рисунок А.3 Результат перевірки текстів на схожість

Уривки введені однакові, але, трохи змінені. Отже, унікальність сягає 43.41%.

Веб-додаток «Comparer» перевіряє будь-яку мову. (Рисунок А.4)

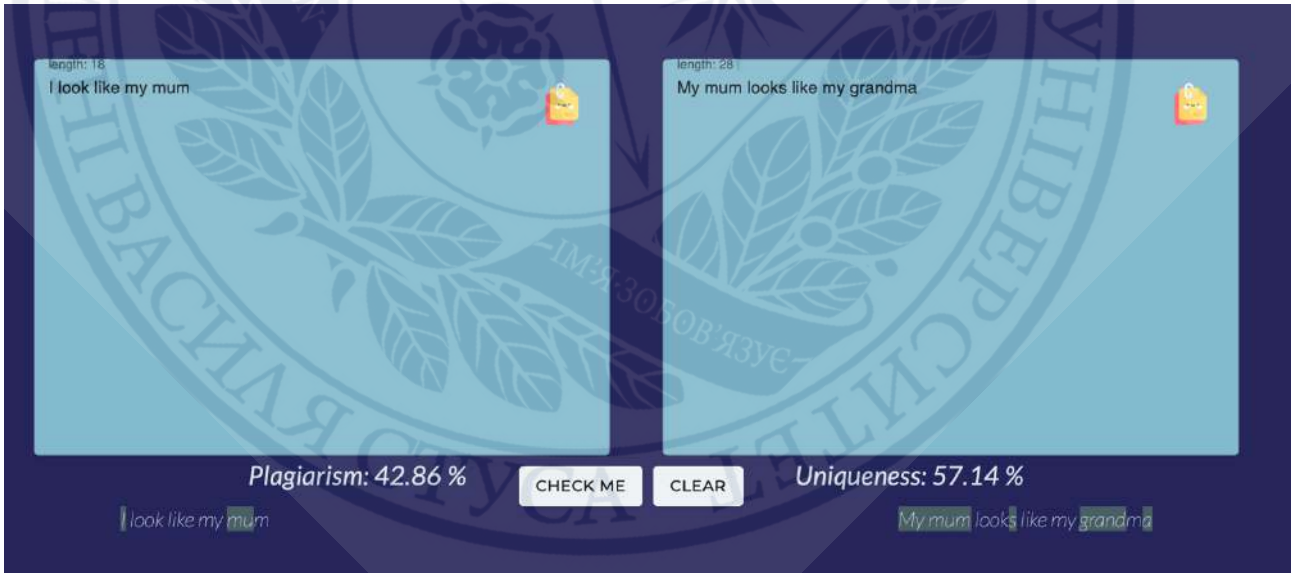

Рисунок А.4 Введення тексту англійською мовою

Щоб дізнатись довжину стрічки, варто просто написати кілька слів. (Рисунок А.5)

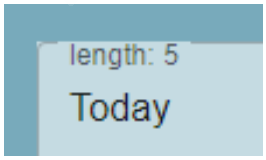

Рисунок А.5 Довжина слова «Today»

Якщо ввести великий за обсягом фрагмент, то з’явиться прокручувач. (Рисунок А.6)

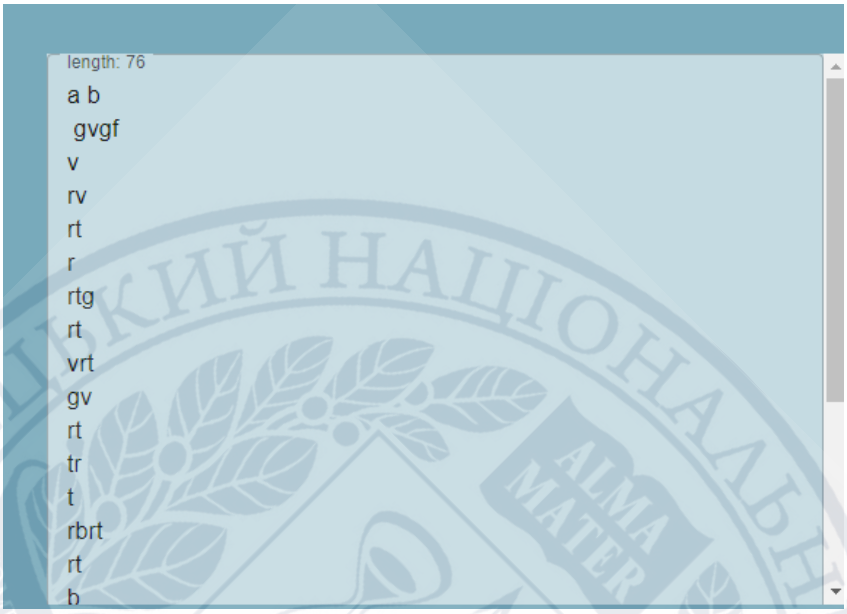

Рисунок А.6 Прокручування тексту

У випадку, якщо випадково почистили результати перевірки, в інтерфейсі передбачався окремий блок для відображення списку попередніх запитів та їхніх результатів. Тут можна перейти знову до минулих введень, створити новий або навіть почистити історію(Рисунок А.7).

Рисунок А.7 Історія минулих порівнянь

При користуванні додатком у темний період часу, перемикач між світлою та темною темами буде дуже корисним та зручним. (Рисунок А.8).

Рисунок А.8 Перемикач світлої/темної теми

ДОДАТОК Б

Лістинг коду

Ініціалізація десктопного додатка

```
const electron = require(«electron»);
const app = electron.app;
const BrowserWindow = electron.BrowserWindow;

const path = require(«path»);
const isDev = require(«electron-is-dev»);

let mainWindow;

require(«update-electron-app»)({
  repo: «kitze/react-electron-example»,
  updateInterval: «1 hour»
});

function createWindow() {
  mainWindow = new BrowserWindow({ width: 900, height: 680,
webPreferences: { nodeIntegration: true }});
  mainWindow.loadURL(
    isDev
      ? «http://localhost:3000»
      : `file://${path.join(__dirname, «../build/index.html»)}`
  );
  mainWindow.on(«closed», () => (mainWindow = null));
}

app.on(«ready», createWindow);

app.on(«window-all-closed», () => {
  if (process.platform !== «darwin») {
    app.quit();
  }
});

app.on(«activate», () => {
  if (mainWindow === null) {
    createWindow();
  }
});
```

Стилі додатку

```
.symbol-added {
  background: #4a5f66;
}

.history-list {
  display: flex;
```

```
  flex-direction: column;
  gap: 10px;
}

.history-btn {
  color: rgb(255, 255, 255) !important;
  border-radius: 6px !important;
  border-color: white !important;
}

.main {
  display: flex;
  height: 100%;
}
.burger-btn {
  position: absolute;
  top: 8px;
  left: 8px;
}
.history {
  padding-right: 8px;
  color: white;
}

body.dark {
  background-color: rgb(37, 37, 92);
  transition: filter 0.3s ease, background-color 0.3s ease;
}

.upload-text-btn {
  align-self: baseline;
}

.attach-file-container {
  display: flex;
  margin-left: 500px;
  justify-content: space-around;
}

body {
  background-color: #78aabb;
}

.theme-switcher {
  width: 100%;
  display: flex;
  justify-content: end;
}

.symbol-removed {
  background: #4a5f66;
}
```

```
.input-container {
  display: flex;
  justify-content: center;
  height: 378px;
  max-height: 100%;
}

.input-container > div {
  background-color: #c5dbe2;
  width: 550px;
  color: #24414a;
  overflow-y: auto;
  border-radius: 5px;
}

body.dark .input-container > div {
  background-color: #85bace;
}

.input-container > div > label {
  padding-top: 8px;
}

.first-input {
  margin-right: 50px !important;
}

.statistic-container {
  display: flex;
  justify-content: center;
}

.plagiarism {
  font-family: «Lato», sans-serif !important;
  font-weight: 400;
  font-style: italic;
  font-size: 25px !important;
  color: #d9e7eb;
}

.unuqueness {
  font-family: «Lato», sans-serif !important;
  font-weight: 400;
  font-style: italic;
  font-size: 25px !important;
  color: #d9e7eb;
}

.result-button {
  margin: 10px 0 0 50px !important;
  font-family: «Montserrat», sans-serif !important;
```

```
  font-optical-sizing: auto;
  font-style: normal;
  font-size: 15px !important;
  color: #d9e7eb;
  background-color: #24414a !important;
}

.clear-button:hover {
  color: #24414a;
  background-color: #d9e7eb !important;
}

.clear-button {
  margin-top: 10px !important;
  margin-left: 10px !important;
  margin-right: 50px !important;
  font-family: «Montserrat», sans-serif !important;
  font-optical-sizing: auto;
  font-style: normal;
  font-size: 15px !important;
  color: #d9e7eb;
  background-color: #24414a !important;
}

body.dark .clear-button {
  background-color: #ecf3f5 !important;
  color: black;
}

body.dark .result-button {
  background-color: #ecf3f5 !important;
  color: black;
}

.result-container {
  display: flex;
  justify-content: center;
}

.result-text-left,
.result-text-right {
  font-family: «Lato», sans-serif !important;
  font-weight: 200;
  font-style: italic;
  font-size: 20px !important;
  color: #d9e7eb;
  display: flex;
}

.result-text-left > div > del,
.result-text-right > div > del,
.result-text-left > div > ins,
```

```
.result-text-right > div > ins {
  text-decoration: none !important;
}

.result-text-left > div > ins,
.result-text-right > div > del {
  display: none;
}

.result-text-left > div > del,
.result-text-right > div > ins {
  background: #4a5f66;
}

.nobutton {
  margin: 0 300px;
}
.result-button:hover {
  color: #24414a;
  background-color: #d9e7eb !important;
}

.header {
  width: 100%;
  justify-content: center;
  display: flex;
  font-family: «Montserrat» sans-serif !important;
  font-optical-sizing: auto;
  font-style: normal;
  font-size: 45px;
  color: #ecf3f5;
}

.description {
  margin: 20px 20px 80px;
  color: #ecf3f5;
  text-align: center;
  font-family: «Lato», sans-serif;
  font-weight: 300;
  font-style: italic;
  font-size: 20px;
}

.main-container {
  margin: 120px 0 10px 0;
  flex-grow: 1;
}

/* npm run react-start */
```

Лістинг головного компонента

```
import React, { useMemo, useState } from «react»;
import { Button, TextField, Typography } from «@mui/material»;
import Diff from «text-diff»;
import «./App.css»;
import ThemeSwitcher from «./ThemeSwitcher/ThemeSwitcher»;
import { handleFileUpload } from «./utils»;

const AcceptFormats = «.txt,.text,.pdf»;

function App() {
  const [text1, setText1] = useState(«»);
  const [text2, setText2] = useState(«»);
  const [plagiarism, setPlagiarism] = useState(null);
  const [uniqueness, setUniqueness] = useState(null);
  const [diff, setDiff] = useState(null);
  const [history, setHistory] = useState([]);
  const [historyOpened, setHistoryOpened] = useState(false);

  const getResult = (textA = text1, textB = text2, withHistory =
true) => {
    const diffInstance = new Diff();
    let difference = diffInstance.main(textA, textB);
    setDiff(difference);
    const changedSymbolsCount = difference.reduce((count, current)
=> {
      return count + (current[0] != 0 ? current[1].length : 0);
    }, 0);
    const biggerTextCount = Math.max(textA.length, textB.length);
    let uniq = (changedSymbolsCount / biggerTextCount) * 100;
    if (uniq > 100) uniq = 100;

    setUniqueness(uniq);
    setPlagiarism(100 - uniq);

    if (withHistory) {
      const plag = 100 - uniq;
      const newHistoryItem = {
        id: Date.now(),
        name: textA.slice(0, 10),
        text1: textA,
        text2: textB,
      };

      setHistory([newHistoryItem, ...history]);
    }
  };
  const diffInstance = useMemo(() => new Diff(), []);
  const fillFromHistory = (data) => {
    setText1(data.text1);
    setText2(data.text2);

    getResult(data.text1, data.text2, false);
```

```
};
return (
  <>
    <div className=«theme-switcher»>
      <ThemeSwitcher />
    </div>
    <div className=«main»>
      <div className=«history»>
        {historyOpened && history.length > 0 && (
          <div>
            <h2>Comparison History</h2>
            <div className=«history-list»>
              <Butto
                className=«history-btn»
                variant=«outlined»
                onClick={() => {
                  setText1(«»);
                  setText2(«»);
                  setDiff(null);
                }}
              >
                New
              </Button>
              {history.map((entry) => (
                <div key={entry.id}>
                  <Button
                    fullWidth
                    className=«history-btn»
                    variant=«outlined»
                    onClick={() => fillFromHistory(entry)}
                  >
                    {entry.name}
                  </Button>
                </div>
              ))}
              <Button
                variant=«text»
                className=«history-btn»
                onClick={() => {
                  setHistory([]);
                  setHistoryOpened(false);
                }}
              >
                Clear
              </Button>
            </div>
          </div>
        )}
      </div>
      <div className=«main-container»>
        <div className=«header»>Comparer</div>
        <div className=«description»>
```

```
            Welcome to our text similarity checking program! It
will help you
            analyze texts, identify their similarities, and save
the results.
            <br />
            The simple interface allows you to input texts,
receive results, and
            save them for future use. We wish you success in your
work and
            achieving high results!
          </div>
          <div>
            <input
              accept={AcceptFormats}
              style={{ display: «none» }}
              id= «upload-text1»
              type= «file»
              onChange={(e) => handleFileUpload(e, setText1)}
            />
            <input
              accept={AcceptFormats}
              style={{ display: «none» }}
              id= «upload-text2»
              type= «file»
              onChange={(e) => handleFileUpload(e, setText2)}
            />
          </div>
          <div className= «input-container»>
            <TextField
              className= «first-input»
              multiline
              minRows={15}
              label={text1.length ? `length: ${text1.length}` :
«Input text 1»}
              value={text1}
              onChange={(e) => setText1(e.target.value)}
              InputProps={{
                endAdornment: (
                  <label htmlFor= «upload-text1» className=
«upload-text-btn»>
                    <Button variant= «text» component= «span»>
                      <img width={40} src= «/free-icon-paperclip-
657119.png» />
                    </Button>
                  </label>
                ),
              }}
            />

            <TextField
              multiline
              minRows={15}
```

```
              label={text2.length ? `length: ${text2.length}` :
«Input text 2»}
              value={text2}
              onChange={(e) => setText2(e.target.value)}
              InputProps={{
                endAdornment: (
                  <label htmlFor= «upload-text2» className=
«upload-text-btn»>
                    <Button variant= «text» component= «span»>
                      <img width={40} src= «/free-icon-paperclip-
657119.png» />
                    </Button>
                  </label>
                ),
              }}
            />
          </div>
          <div className= «statistic-container»>
            {diff && (
              <Typography className= «plagiarism»>
                Plagiarism: {plagiarism && plagiarism.toFixed(2)}
%
              </Typography>
            )}
            <Button
              variant= «contained»
              className= «result-button»
              onClick={() => getResult()}
            >
              Check me
            </Button>
            <Button
              variant= «contained»
              className= «clear-button»
              onClick={() => {
                setText1(«»);
                setText2(«»);
                setDiff(null);
              }}
            >
              Clear
            </Button>
            {diff && (
              <Typography className= «unuqueness»>
                Uniqueness: {uniqueness && uniqueness.toFixed(2)}
%
              </Typography>
            )}
          </div>

          {diff && (
            <div className= «result-container»>
```

```
                <div className= «result-text-left»>
                  <div
                    dangerouslySetInnerHTML={{
                      __html: diffInstance.prettyHtml(diff),
                    }}
                  />
                </div>
                <div className= «nobutton»></div>
                <div className= «result-text-right»>
                  <div
                    dangerouslySetInnerHTML={{
                      __html: diffInstance.prettyHtml(diff),
                    }}
                  />
                </div>
              </div>
            )}
            <div className= «burger-btn»>
              <Button
                variant= «text»
                onClick={() => {
                  setHistoryOpened((x) => !x);
                }}
              >
                <img src= «menu-bar.png» width= «50» className=
«icon» />
              </Button>
            </div>
          </div>
        </div>
      </>
    );
}

export default App;
```