ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Дослідження моделі системи пошуку та підбору людей за визначеними критеріями»

на здобуття освітнього ступеня магістра
зі спеціальності 122 Комп'ютерні науки
(код, найменування спеціальності)
освітньо-професійної програми Комп'ютерні науки
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання
на відповідне джерело

_____ Іван ДАБІЖА
(підпис) *(Ім'я, ПРІЗВИЩЕ здобувача)*

Виконав:
здобувач вищої освіти групи КНДМ-61 Іван ДАБІЖА

Керівник:
науковий ступінь, вчене звання Віктор ВИШНІВСЬКИЙ
д.т.н., професор

Рецензент:
науковий ступінь, вчене звання
_____
(Ім'я, ПРІЗВИЩЕ)

Київ 2023

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Комп'ютерних наук

Ступінь вищої освіти Магістр

Спеціальність Комп'ютерні науки

Освітньо-професійна програма Комп'ютерні науки

ЗАТВЕРДЖУЮ
Завідувач кафедрою Комп'ютерних наук

_____ Віктор ВИШНІВСЬКИЙ
«_____» _____ 2023 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Дабіжі Івану Олеговичу
(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи: Дослідження моделі системи пошуку та підбору людей за визначеними критеріями

керівник кваліфікаційної роботи Віктор ВИШНІВСЬКИЙ д.т.н., професор,
(Ім'я, ПРІЗВИЩЕ науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «19» 10.2023р. №145

2. Строк подання кваліфікаційної роботи «29» грудня 2023 р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
Аналіз технологій пошуку
Дослідження можливостей машинного навчання для TYPESCRIPT
Аналіз існуючих технологій для машинного навчання
Побудова фронтової та бекендної частин застосунку
5. Перелік графічного матеріалу: *презентація*

6. Дата видачі завдання «19» жовтня 2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	19.10-05.11.23	
2	Вивчення матеріалів для аналізу особливостей роботи із машинним навчанням у TS	05.11-12.11.23	
3	Дослідження наявних технологій для кросплатформеної розробки	13.11-19.11.23	
4	Дослідження можливостей машинного навчання для TS	20.11-25.11.23	
5	Аналіз існуючих технологій для машинного навчання	27.11-03.12.23	
6	Побудова фронтової та бекендної частин застосунку	04.12-10.12.23	
7	Оформлення роботи: вступ, висновки, реферат	11.12-20.12.23	
8	Розробка демонстраційних матеріалів	21.12-29.12.23	

Здобувач вищої освіти _____ Іван ДАБІЖА
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Керівник
кваліфікаційної роботи _____ Віктор ВИШНІВСЬКИЙ
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 79 стор., 17 рис., 10 джерел.

Мета роботи – підвищити кількість збігів по критеріям через пошук суміжних критеріїв побудованого за допомогою сучасної мови програмування та можливостей ШІ.

Об`єкт дослідження – процес пошуку та підбору людей на основі TYPESCRIPT

Предмет дослідження – модель системи пошуку та підбору людей за визначеними критеріями

Сфера застосування – соціальні знайомства

Актуальність роботи – широке використання датінг-сервісів та пошук оптимальних способів пріоритизації людей із суміжними інтересами.

Короткий зміст роботи: За результатами аналізу знайдено оптимальне рішення щодо бібліотеки ШІ, а після розробки було отримано прототип програмного забезпечення, який надає можливість використовувати алгоритм пошуку нових людей подібних за інтересами.

Також створено список удосконалень і нововведень для майбутніх версій.

Впровадження розробленої системи дозволяє будь кому встановити додаток та швидко знайти людей подібних до себе. Також, додана можливість генерування повідомлення людині на основі його інтересів для швидкого початку спілкування.

КЛЮЧОВІ СЛОВА: TYPESCRIPT, MACHINE LEARNING, АЛГОРИТМ, SEARCH ENGINE, REACT, RTK, NODEJS

ABSTRACT

Text part of the master's qualification work: 79 pages, 17 pictures, 1 table, 10 sources.

The purpose of the work – increase the number of matches by criteria through the search for adjacent criteria built with the help of a modern programming language and AI capabilities.

Object of research –the process of searching and selecting people based on TYPESCRIPT

Subject of research – a model of the search and selection system for people based on defined criteria

Summary of the work: Based on the results of the analysis, the optimal solution for the AI library was found, and after development, a software prototype was obtained, which provides an opportunity to use the algorithm for finding new people with similar interests.

A list of improvements and innovations for future versions has also been created.

The implementation of the developed system allows anyone to install the application and quickly find people similar to themselves. Also, the ability to generate a message to a person based on their interests has been added to quickly start communication.

KEYWORDS: TYPESCRIPT, MACHINE LEARNING, ALGORITHM, SEARCH ENGINE, REACT, RTK, NODEJS

ЗМІСТ

ВСТУП

У динамічному гобелені нашого взаємопов’язаного світу спосіб налагодження стосунків зазнав глибокої еволюції. Зараз, як ніколи, цифрова сфера відіграє ключову роль у формуванні зв’язків. Давайте заглибимося в розповідь, яка позиціонує програми для знайомств не просто як технологічні новинки, а як стратегічні активи в нашому сучасному ландшафті.

У суєті сучасного життя час став безцінним товаром. Традиційні способи знайомства з потенційними партнерами затьмарені ефективністю та доступністю, які пропонують програми для знайомств. Вони стали провідниками, через які люди, пов’язані напруженим графіком і різноманітними зобов’язаннями, перетинаються та досліджують можливість значущих зв’язків.

Ці програми — це не просто платформи для пошуку партнерів; вони є справжньою скарбницею даних. Кожен рух, кожна взаємодія та кожен успішний збіг сприяють надходженню величезної кількості інформації.

Однак важливість додатків для знайомств виходить за межі прибутку. Успішні зв’язки, створені через ці платформи, сприяють позитивному суспільному наративу. В епоху, коли прославляється інклюзивність і різноманітність, програми для знайомств стали каталізаторами для встановлення зв’язків, які виходять за межі традиційних рамок. Цей соціальний вплив є не просто побічним продуктом; це стратегічне узгодження з сучасними цінностями та свідчення корпоративної відповідальності.

Оскільки ми орієнтуємось у течіях технологічної еволюції, програми для знайомств стоять на передньому краї інновацій. Їх здатність адаптуватися до нових технологій, таких як штучний інтелект, доповнена реальність і віртуальна реальність, позиціонує їх не просто як продукти сьогодення, але як динамічні організації, готові прийняти виклики майбутнього.

1 АНАЛІЗ ТЕХНОЛОГІЇ ПОШУКУ ЛЮДЕЙ ЗІ СПІЛЬНИМИ ІНТЕРЕСАМИ

1.1 Аналітика даних у програмах для знайомств: відкриття статистичних даних для значущих збігів

Аналітика даних відстежує кожну дію користувача, від свайпів до повідомлень, розгадуючи тонкощі моделей взаємодії. Це розуміння виходить за межі поверхні, надаючи детальне розуміння того, як користувачі взаємодіють із додатком і які функції найбільше сприймають аудиторію.

Досліджуючи заявлені вподобання та інтереси, аналітика даних створює багаті профілі користувачів, які виходять за рамки кліше. Це відображення стає основою для припущення потенційних збігів із спільними хобі чи пристрастями, підвищуючи ймовірність зв’язків, які виходять за межі поверхневих.

Аналітика даних служить компасом, направляючи додатки для знайомств у постійно мінливий ландшафт активності користувачів. Виявляючи тенденції та сезонні коливання, ці платформи адаптують свої стратегії — коригуючи функції чи вдосконалюючи маркетингові ініціативи — щоб забезпечити постійну релевантність і залучення.

У динамічному середовищі додатків для знайомств функції потребують розвитку. Аналітика даних полегшує A/B-тестування, де перевіряються варіації функцій, щоб оцінити реакцію користувача. Цей ітеративний процес забезпечує безперервне вдосконалення, адаптацію програми до вподобань користувача та випередження.

Інтеграція аналізу даних – це не просто доповнення; це трансформація всього процесу пошуку партнерів. Від алгоритмів динамічного зіставлення, які реагують на зміну поведінки користувачів, до зіставлення інтересів у реальному

часі на основі поточних дій, аналітика даних сприяє взаємодії з користувачем, яка полягає не лише в пошуку когось, а в пошуку значущих зв’язків.

Підсумовуючи, аналітика даних не є інструментом, що працює у фоновому режимі; це серцебиття сучасних програм для знайомств. Це рушійна сила кожного матчу, історії успіху та еволюції програми. Оскільки платформи для знайомств продовжують використовувати потужність аналітики даних, користувачі можуть очікувати досвіду, який виходить за межі поверхні, де зв’язки не просто сприяють, а й справді мають значення.

1.2 Навігація підключень: суть пошукових алгоритмів у програмах для знайомств

На динамічній арені сучасних програм для знайомств ключовою детермінантою успіху є стратегічне впровадження складних алгоритмів пошуку. Ці алгоритми, бездоганно інтегровані в ядро цих платформ, служать архітекторами цілеспрямованих зв’язків, складно переміщаючись між уподобаннями, інтересами та профілями користувачів, щоб організовувати збіги, які виходять за межі поверхневого.

За своєю суттю, алгоритми пошуку ретельно розроблені, щоб бути орієнтованими на користувача, дозволяючи людям вводити конкретні критерії, такі як демографічні дані та нюанси інтересів. Це гарантує, що результати пошуку точно узгоджуються з параметрами, визначеними користувачем, створюючи персоналізований та ефективний досвід користувача.

Динамічність цих алгоритмів є вирішальним аспектом. Вони не статичні; вони розвиваються та адаптуються на основі взаємодії користувачів і нових тенденцій. Постійно вивчаючи поведінку користувачів, вони динамічно вдосконалюють процес пошуку партнерів. Якщо користувач постійно взаємодіє з

профілями, які мають спільні інтереси, алгоритм адаптується, повторно калібруючи рекомендації для покращеної персоналізації.

Окрім базових параметрів, включено вдосконалені механізми фільтрації. Це дозволяє користувачам фільтрувати збіги на основі таких складних факторів, як спосіб життя, цілі стосунків і конкретні інтереси. Деталізація цих функцій покращує процес підбору, задовольняючи різноманітні та нюансовані вподобання користувачів.

Оцінка сумісності додає рівень складності процесу підбору. Виходячи за межі основного збігу інтересів, ці алгоритми призначають бали сумісності потенційним збігам. Користувачам пропонуються збіги, які супроводжуються оцінкою сумісності, пропонуючи швидку та обґрунтовану оцінку того, наскільки добре дві особи узгоджуються з точки зору спільних інтересів — функція, яка покращує прийняття рішень.

Інтеграція обробки природної мови (NLP) покращує взаємодію з користувачем, дозволяючи користувачам вводити свої інтереси в розмовному форматі. Відступаючи від жорстких прапорців, алгоритм обробляє та інтерпретує вхідні дані природною мовою, забезпечуючи плавний і схожий на людину пошук.

Постійне вдосконалення через відгуки користувачів формує суть постійного вдосконалення. Програми для знайомств активно шукають інформацію від користувачів щодо збігів, розвиваючи симбіотичні стосунки з пошуковими алгоритмами. Цей ітеративний цикл зворотного зв’язку гарантує, що алгоритми розвиваються на основі реального досвіду користувача, узгоджуючи платформу з уподобаннями користувача.

Підсумовуючи, інтеграція пошукових алгоритмів у програми для знайомств є стратегічним імперативом. Їх орієнтований на користувача дизайн, адаптивність і розширені функції орієнтуються в сучасному ландшафті відносин, що розвивається, пропонуючи людям стратегічний шлях до зв’язків, які виходять за

межі поверхні — бізнес-підхід, який сприяє ефективності та користувацькому досвіду, який сприяє значущим залученням.

1.3 Перелік основних алгоритмів з описом, які використовуються в процесі пошуку

Спільна фільтрація:

Опис. Спільна фільтрація використовує матричну факторізацію та показники подібності. Він передбачає створення матриці користувацьких елементів, де записи представляють взаємодії користувача (лайки, свайпи). Потім алгоритм прогнозує перевагу користувача щодо елемента на основі схожих користувачів.

Приклад: нехай U — матриця користувача, I — матриця предметів, а R — матриця взаємодії між користувачем і елементом. Мета полягає в тому, щоб розкласти R на × U × I. Якщо користувачеві A сподобалися профілі X і Y, а користувачеві B сподобалися профілі Y і Z, спільна фільтрація може рекомендувати профіль Z користувачеві A.

Фільтрування на основі вмісту:

Опис: фільтрація на основі вмісту використовує вектори ознак для представлення елементів і користувачів. Подібність вимірюється за допомогою таких методів, як косинусна подібність. Він рекомендує елементи на основі подібності між уподобаннями користувача та характеристиками елементів.

Приклад: якщо користувач A висловив значний інтерес до активного відпочинку на свіжому повітрі, а профіль X має такі функції, як «похід», «кемпінг» і «природа», фільтрація на основі вмісту може рекомендувати профіль X користувачеві A через подібність уподобань. .

Геопросторові алгоритми:

Опис: геопросторові алгоритми використовують показники відстані (наприклад, формулу Гаверсинуса), щоб обчислити відстань між користувачами на основі їхніх географічних координат.

Приклад: якщо користувач А знаходиться за координатами (lat1, lon1) і віддає перевагу локальним з'єднанням, алгоритм може запропонувати профілі на певній відстані, гарантуючи, що збіги знаходяться в безпосередній географічній близькості.

Прогнозні моделі на основі машинного навчання:

Опис. Прогнозні моделі часто включають регресію або алгоритми класифікації. Вони вчаться з історії взаємодії користувачів, щоб передбачити переваги для невидимих елементів.

Приклад: за допомогою логістичної регресії модель вчиться передбачати ймовірність того, що користувачеві сподобається профіль на основі таких ознак, як спільні інтереси, попередні оцінки "подобається" та демографічна інформація. Чим вища ймовірність, тим більша ймовірність того, що профіль буде рекомендовано.

Обробка природної мови (NLP):

Опис: НЛП включає токенізацію, семантичний аналіз і вбудовування слів. Це дозволяє алгоритмам розуміти контекст і значення, що лежить в основі введених користувачами природної мови.

Приклад: якщо користувач вводить такі слова: «Мені подобається читати таємничі романи», NLP аналізує структуру речень і визначає такі ключові слова, як «читання» та «таємничі романи». Потім алгоритм пов'язує ці ключові слова з відповідними функціями профілю для кращого підбору відповідності.

Ці алгоритми, збагачені математичними основами, дозволяють додаткам для знайомств надавати користувачам персоналізовані та точні рекомендації, покращуючи загальну взаємодію з користувачем і сприяючи значущим зв'язкам на основі різноманітних критеріїв.

1.4 Аналіз алгоритму пошуку людей зі схожими інтересами на основі ШІ та Typescript

Побудова системи для об’єднання людей зі спільними інтересами передбачає багатогранний підхід, поєднання алгоритмів ШІ з TypeScript для ефективного виконання. Щоб запустити процес, ми повинні спочатку зрозуміти суть проблеми. Ми прагнемо створити механізм, який вміло об’єднує людей на основі їхніх уподобань, хобі чи будь-яких інших важливих факторів.

У TypeScript наша подорож починається з визначення надійних структур даних для представлення профілів користувачів та їхніх відповідних інтересів. Ці структури, часто виражені через інтерфейси або класи, служать основою, на якій працюватиме наш алгоритм. Користувачі, інкапсульовані унікальними ідентифікаторами, іменами та списком інтересів, стають центрами нашої роботи зі збігу.

Далі ми заглибимося в сферу штучного інтелекту, щоб полегшити процес підбору партнерів. Спільна фільтрація, наріжний метод, займає центральне місце. Цей метод передбачає порівняння користувачів на основі їхніх уподобань чи поведінки. TypeScript дає нам змогу реалізувати ці алгоритми за допомогою функцій, плавно порівнюючи користувачів і надаючи підібраний список людей зі схожими інтересами.

У міру зміцнення логіки відповідності на основі штучного інтелекту інтеграція в інтерфейс користувача стає першорядною. Інтерфейс, імовірно, розроблений з використанням таких фреймворків, як React або Angular, служить порталом користувача для введення своїх інтересів і вивчення запропонованих збігів. Така синергія між алгоритмом та інтерфейсом забезпечує зручність використання.

Оновлення в реальному часі та постійна оптимізація складають основу динамічності нашої системи. Користувачам із змінними інтересами потрібен алгоритм відповідності, який адаптується в режимі реального часу. Регулярна

оптимізація як з точки зору алгоритмічної ефективності, так і з точки зору взаємодії з користувачем гарантує, що система залишається чутливою та точною в міру розширення бази користувачів.

Забезпечення цілісності даних користувача є обов'язковим. Система повинна відповідати суворим стандартам безпеки та конфіденційності. З інформацією про користувачів слід поводитися дуже обережно, а будь-які моделі ШІ слід навчати на анонімних наборах даних, зберігаючи довіру користувачів.

Тестування, безперервний процес, підтверджує ефективність системи. Різноманітні набори даних, що відображають різноманітність користувачів, стрес-тестування алгоритму. Відгуки користувачів стають компасом, що спрямовує ітераційні вдосконалення, прокладаючи шлях для постійно розвиваючої та орієнтованої на користувача системи підбору партнерів.

Документування кодової бази та алгоритмів є найкращою практикою, яка допомагає у розумінні та майбутньому розвитку. Масштабованість, життєво важлива умова, вплетена в структуру системи. Архітектура розроблена для того, щоб витончено обробляти зростаючу базу користувачів, забезпечуючи бездоганну функціональність у міру розширення платформи.

Цей уніфікований підхід, що забезпечує плавний перехід від представлення даних до інтеграції штучного інтелекту, реалізації інтерфейсу користувача, адаптації в реальному часі тощо, закладає основу для привабливої та ефективної системи зіставлення. У ландшафті алгоритмів підбору людей інтеграція штучного інтелекту та TypeScript формує системний підхід, який усуває складні зв'язки в цифровому середовищі.

Уявіть собі основу цієї технологічної структури – систематичне представлення профілів користувачів. TypeScript, відомий своєю точністю та структурованістю, стає оркестром, що керує даними. Визначення інтерфейсів або класів UserProfile логічно інкапсулює дані користувача. Кожен користувач, який має унікальний ідентифікатор, конкретне ім'я та чітко визначений список

інтересів, вносить свій внесок у структурований набір даних, за яким наш алгоритм буде систематично орієнтуватися.

Далі алгоритм зіставлення на основі ШІ займає центральне місце, де наукові принципи керують процесом. Спільна фільтрація, яка ефективно підтримується функціями TypeScript, стає ключовим аспектом. Користувачі порівнюються на основі логічного аналізу інтересів, при цьому алгоритм оцінює перетин уподобань користувачів. Він розпізнає закономірності, пропонуючи зв’язки, які відповідають динамічній еволюції інтересів користувачів з часом.

У міру розгортання сюжету інтеграція оновлень у реальному часі стає ключовим аспектом нашої алгоритмічної системи. Алгоритм, налаштований на динамічну природу мінливих інтересів користувачів, адаптується в режимі реального часу. Користувачі стають свідками миттєвих коригувань системи, коли вони змінюють свої профілі, гарантуючи, що пропозиції залишаються актуальними відповідно до динаміки розвитку спільноти.

Об’єднання TypeScript із зовнішніми фреймворками, такими як React або Angular, стає практичним інтерфейсом, що полегшує взаємодію користувачів. Цей інтерфейс користувача, структурований портал, дозволяє користувачам логічно вводити свої інтереси. Він діє як провідник, бездоганно з’єднуючи алгоритмічну логіку з взаємодією користувачів, створюючи прагматичний досвід, який одночасно є складним і зручним для користувача.

І чому прагматичному розробнику слід вибрати таку комбінацію штучного інтелекту та TypeScript?

Алгоритм зіставлення, керований ШІ, використовує логічну точність. Йдеться не про чарівні зв’язки, а про системний підхід до об’єднання користувачів на основі їхніх чітких уподобань, сприяючи відчуттю зв’язку спільноти.

Адаптивність системи забезпечує динамічність процесу. Зі зміною інтересів користувачів алгоритм миттєво коригується, постійно надаючи пропозиції, які відповідають поточному стану вподобань користувачів.

Надійна типізація та надійні структури даних TypeScript сприяють захисту даних користувача науково обґрунтованим способом. Система дотримується стандартів конфіденційності, безпечно обробляючи інформацію користувача відповідно до встановлених протоколів.

Масштабованість TypeScript у поєднанні з продуманим дизайном алгоритму забезпечує оптимальну продуктивність із зростанням спільноти користувачів. Це ефективна система, здатна працювати з розширеною базою користувачів із логічною масштабованістю.

Повна інтеграція TypeScript із зовнішніми фреймворками спрощує розробку, сприяючи реалізації системи, яка працює логічно та захоплює користувачів завдяки інтуїтивно зрозумілому та зручному інтерфейсу.

По суті, алгоритм підбору людей на основі ШІ та TypeScript є систематичним застосуванням логічних принципів. Це не містичний процес, а свідчення майстерності створення динамічних і персоналізованих вражень, які резонують із користувачами, зміцнюючи зв’язки, засновані на наукових принципах.

2 ДОСЛІДЖЕННЯ МОЖЛИВОСТЕЙ TS ДЛЯ МАШИННОГО НАВЧАННЯ

2.1 Розуміння зв’язку між ШІ та машинним навчанням

Штучний інтелект (AI) і машинне навчання (ML) — це терміни, які часто використовуються як синоніми, але вони представляють різні, але взаємопов’язані концепції в сфері інформатики. Коли ми заглиблюємося в тонкощі цих сфер, дуже важливо зрозуміти головні цілі та методології, які визначають кожну з них.

2.1.1 Штучний інтелект (AI): цілісний підхід до інтелектуальних систем

За своєю суттю штучний інтелект — це широке й експансивне поле, яке прагне наповнити машини здатністю виконувати завдання, які зазвичай потребують людського інтелекту. Ці завдання охоплюють спектр діяльності, починаючи від міркувань і вирішення проблем до сприйняття, розуміння мови та прийняття рішень.

Системи штучного інтелекту створені для імітації людського інтелекту, і вони бувають різних форм. Традиційні підходи включають системи на основі правил та експертні системи, де заздалегідь визначені правила та бази знань керують процесами прийняття рішень. Символічне міркування, ще один аспект ШІ, передбачає маніпулювання символами та правилами для отримання висновків.

Однак однією з найбільш трансформуючих гілок штучного інтелекту є машинне навчання.

2.1.2 Машинне навчання (ML): динамічна підмножина ШІ

Машинне навчання — це підмножина штучного інтелекту, яка зосереджена на створенні алгоритмів і статистичних моделей, які дозволяють комп'ютерам покращувати свою продуктивність у виконанні певного завдання з часом без явного програмування. На відміну від традиційних систем, заснованих на правилах, системи ML навчаються на даних і досвіді, адаптуються та розвиваються для більш ефективного виконання завдань.

Існує три основні типи машинного навчання:

Контрольоване навчання: у цьому підході алгоритм навчається на позначеному наборі даних, де вхідні дані поєднуються з відповідними бажаними виходами. Модель вчиться робити прогнози або приймати рішення на основі даних навчання.

Неконтрольоване навчання: тут алгоритм представлений з даними без чітких інструкцій. Система намагається розпізнати закономірності та структури в даних, виявляючи ідеї без попередньо визначених результатів.

Навчання з підкріпленням: цей метод передбачає взаємодію алгоритму з навколишнім середовищем. Система отримує зворотній зв'язок у вигляді винагород або штрафів на основі своїх дій, що дозволяє їй коригувати свою поведінку з часом, щоб максимізувати сукупні винагороди.

Важливо визнати, що хоча все машинне навчання є підмножиною ШІ, не всі системи ШІ включають машинне навчання. Машинне навчання є потужним інструментом у ширшій сфері штучного інтелекту, пропонуючи адаптивні можливості, які відрізняють його від більш жорстких, заснованих на правилах підходів.

Підсумовуючи, штучний інтелект і машинне навчання представляють взаємопов'язані аспекти інформатики, кожен з яких вносить унікальний внесок у розвиток інтелектуальних систем. У той час як штучний інтелект охоплює спектр технік і методологій для створення інтелектуальних машин, машинне навчання

виділяється як динамічна підмножина, яка дає змогу системам навчатися, адаптуватися та вдосконалюватися на основі досвіду. Разом вони сприяють еволюції технологій у напрямку все більш складних і автономних можливостей, змінюючи спосіб взаємодії з інтелектуальними системами та отримання від них переваг.

2.2 Аналіз бібліотек машинного навчання для Typescript

2.2.1 TensorFlow.js

TensorFlow.js — це бібліотека JavaScript, розроблена Google для навчання та розгортання моделей машинного навчання в браузері та на Node.js. Ось кілька переваг:

1. Запуск у браузері: TensorFlow.js дозволяє запускати вже існуючі моделі машинного навчання безпосередньо у браузері, не потребуючи сервера. Це може значно зменшити витрати на сервер і затримку.
2. Навчання в браузері: TensorFlow.js також може навчати нові моделі безпосередньо в браузері. Це дозволяє використовувати пристрій користувача для обчислень, що може бути корисним для програм із збереженням конфіденційності, де ви не хочете надсилати дані на сервер.
3. Проста інтеграція з веб-технологіями: Оскільки TensorFlow.js є бібліотекою JavaScript, її можна легко інтегрувати з іншими веб-технологіями, такими як HTML і CSS, для створення інтерактивних програм машинного навчання.
4. Навчання передачі: TensorFlow.js підтримує навчання передачі, що дозволяє взяти попередньо навчену модель і налаштувати її для конкретної програми з невеликою кількістю нових даних.

5. Підтримка різних моделей і шарів: TensorFlow.js підтримує широкий спектр шарів, функцій втрати, оптимізаторів тощо. Він також підтримує імпорт моделей, навчених у Python TensorFlow і Keras.

6. Спільнота та підтримка: TensorFlow.js підтримується Google і має велику спільноту, що означає, що ви можете знайти багато навчальних посібників і попередньо навчених моделей в Інтернеті.

Великі компанії і організацій використовують TensorFlow.js для різних програм:

Google: Google використовує TensorFlow.js у кількох своїх продуктах. Наприклад, у веб-версії Google Teachable Machine користувачі можуть створювати моделі у своєму браузері без написання коду.

IBM: IBM використала TensorFlow.js для створення моделі, яка може класифікувати текст за різними категоріями безпосередньо в браузері.

Uber: Uber розробив інструмент із відкритим вихідним кодом під назвою Ludwig, який використовує TensorFlow.js, щоб дозволити користувачам тренувати та тестувати моделі глибокого навчання без написання коду.

Airbnb: Airbnb використав TensorFlow.js для створення прототипу моделі машинного навчання для категоризації фотографій за типом кімнати.

TensorFlow.js підтримує широкий спектр алгоритмів машинного навчання. Ось деякі з найпопулярніших, які можна реалізувати за допомогою TensorFlow.js:

Лінійна регресія: це базовий алгоритм, який використовується для прогнозування кількісної реакції. Він широко використовується в усіх галузях, від бізнесу до академічних кіл.

Логістична регресія: використовується, коли залежна змінна є категоричною. Він часто використовується для задач двійкової класифікації.

Нейронні мережі: TensorFlow.js підтримує широкий спектр архітектур нейронних мереж, включаючи згорткові нейронні мережі (CNN), повторювані нейронні мережі (RNN) і мережі довгострокової короткочасної пам’яті (LSTM).

Глибоке навчання: TensorFlow.js особливо добре підходить для глибокого навчання, підмножини машинного навчання, яке використовує багаторівневі нейронні мережі.

Трансферне навчання: це техніка, за якої попередньо навчена модель адаптується для іншої, але пов'язаної проблеми. TensorFlow.js дозволяє вам імпортувати попередньо навчені моделі та точно налаштовувати їх на ваших власних даних.

K-Means Clustering: це неконтрольований алгоритм навчання, який використовується для поділу набору даних на K різних кластерів.

Машини опорних векторів (SVM): це контрольована модель навчання, яка використовується для класифікації та регресійного аналізу.

Хоча TensorFlow.js є потужним інструментом, він не завжди є найкращим інструментом для кожної роботи. Якщо ми працюємо з дуже великими наборами даних або складними моделями, можливо, вам краще використовувати TensorFlow від Python і запускати свої моделі на сервері або хмарній платформі.

2.2.2 Brain.js

Brain.js — це бібліотека JavaScript для нейронних мереж із прискоренням GPU. Він досить простий і зручний для початківців, що робить його хорошим вибором для тих, хто тільки починає працювати з машинним навчанням. Ось кілька причин, чому ви можете використовувати Brain.js:

Простота: Brain.js створено для зручності використання. Він забезпечує інтуїтивно зрозумілий API високого рівня для створення та навчання нейронних мереж. Це робить його хорошим вибором для початківців або для простих проектів.

Продуктивність: Brain.js підтримує навчання з прискоренням GPU через WebGL, якщо він доступний, і повертається до чистого JavaScript, якщо ні.

Підтримка браузера та Node.js: як і TensorFlow.js, Brain.js може працювати як у браузері, так і на Node.js, що робить його універсальним для різних середовищ.

Типи нейронних мереж: Brain.js підтримує багаторівневі нейронні мережі прямого зв'язку та рекурентні нейронні мережі для роботи з послідовними даними.

Порівняння Brain.js із TensorFlow.js:

Складність: TensorFlow.js є більш складним і універсальним, ніж Brain.js. Він підтримує ширший діапазон типів нейронних мереж і має більше функцій, але також має крутішу криву навчання.

Продуктивність: TensorFlow.js також підтримує навчання з прискоренням GPU і може бути більш ефективним для великомасштабних моделей або даних.

Спільнота та підтримка: TensorFlow.js, будучи частиною більшого проекту TensorFlow за підтримки Google, має більшу спільноту та більше ресурсів, доступних онлайн.

Таким чином, для новачка або людини, яка працює над простим проектом, Brain.js може бути хорошим вибором через його простоту. Однак для більш складних проектів або великомасштабних даних TensorFlow.js може бути більш придатним завдяки своїй ефективності та широкому набору функцій.

Хоча Brain.js є популярною бібліотекою для нейронних мереж у JavaScript, зазвичай вона використовується більше окремими розробниками, малими підприємствами чи в освітніх цілях через її простоту та легкість використання. Рідше можна побачити великі компанії, які використовують Brain.js для додатків виробничого рівня, оскільки їм часто потрібні більш складні та масштабовані рішення, надані такими бібліотеками, як TensorFlow.js. Brain.js є чудовим інструментом для створення прототипів, вивчення нейронних мереж або створення невеликих програм. Він також має відкритий вихідний код, тому будь-хто може зробити свій внесок у його розвиток і вдосконалення. Також він в основному зосереджений на нейронних мережах і надає простий та інтуїтивно

зрозумілий API для створення, навчання та роботи цих мереж. Ось кілька типів нейронних мереж, які можна реалізувати за допомогою Brain.js:

Нейронні мережі прямого зв'язку: це найпростіший тип штучної нейронної мережі. У мережі прямого зв'язку інформація рухається лише в одному напрямку — вперед — від вхідних вузлів, через приховані вузли (якщо такі є) і до вихідних вузлів.

Повторювані нейронні мережі (RNN): RNN — це тип штучної нейронної мережі, призначеної для розпізнавання шаблонів у послідовностях даних, таких як текст, геноми, почерк або вимовлені слова. За допомогою Brain.js ви можете створювати прості RNN.

Мережі довготривалої короткочасної пам'яті (LSTM): LSTM – це особливий тип RNN, який здатний вивчати довготривалі залежності, що робить їх ідеальними для роботи з послідовностями даних.

Згорткові нейронні мережі (CNN): хоча Brain.js не підтримує CNN, які зазвичай використовуються для обробки зображень, можна реалізувати просту CNN за допомогою API нижчого рівня, наданого бібліотекою.

2.2.3 Synaptic

Synaptic.js — це потужна бібліотека нейронних мереж JavaScript, яка дозволяє створювати та навчати будь-які архітектури нейронних мереж першого або навіть другого порядку. Він не має архітектури, що означає, що ви можете створити та навчити будь-який тип нейронної мережі першого чи другого порядку.

Ось кілька причин, чому можна використовувати Synaptic.js:

Без архітектури: Synaptic.js не залежить від архітектури, що означає, що ви можете створювати будь-які типи нейронних мереж, включаючи, але не обмежуючись ними: багатошарові персептрони, багаторівневі мережі довго-короткочасної пам'яті (LSTM), рідкі автомати або мережі Хопфілда.

Вбудовані архітектури: незважаючи на відсутність архітектури, Synaptic.js надає кілька вбудованих архітектур, як-от Perceptron і LSTM, щоб швидко розпочати роботу.

Тренажер у комплекті: Synaptic.js містить вбудований тренер, який спрощує навчання вашої мережі за допомогою параметрів швидкості, ітерацій, помилок, перемішування, журналу, вартості, розкладу та спеціальних наборів тренувань.

Автономність: Synaptic.js підтримує автономне навчання, що означає, що ви можете навчити свою мережу, перетворити її на окрему функцію, а потім зберегти/експортувати цю функцію для використання пізніше.

Ось кілька випадків використання Synaptic.js:

Прогнозне моделювання: ви можете використовувати Synaptic.js для створення нейронної мережі, яка може передбачати майбутні результати на основі минулих даних.

Розпізнавання шаблонів: Synaptic.js можна використовувати для створення мереж, які розпізнають шаблони або особливості у вхідних даних.

Виявлення аномалій. Нейронні мережі можна використовувати для виявлення аномалій або викидів у складних наборах даних.

Обробка природної мови: з правильною архітектурою ви можете використовувати Synaptic.js для завдань обробки природної мови. *Розпізнавання зображень:* Хоча JavaScript може не бути першим вибором для завдань розпізнавання зображень через міркування про продуктивність, можна використовувати Synaptic.js для простих завдань розпізнавання зображень.

2.2.4 Compromise

Compromise — це бібліотека JavaScript, яка забезпечує швидкий практичний спосіб обробки та аналізу тексту англійською мовою. Він створений як малий, швидкий і гнучкий.

Переваги Compromise:

Швидкість: компроміс створений як швидкий і ефективний, що робить його придатним для обробки тексту в реальному часі.

Розмір: бібліотека відносно невелика та легка, що робить її гарним вибором для використання на стороні клієнта у веб-браузерах.

Простота: Compromise надає простий та інтуїтивно зрозумілий API, що робить його легким у використанні навіть для новачків.

Універсальність: Compromise може обробляти широкий спектр завдань обробки природної мови, включаючи токенізацію, виділення іменників, тегування частин мови тощо.

Недоліки Compromise:

Підтримка мови: наразі Compromise підтримує лише англійську мову. Якщо вам потрібно обробити текст іншими мовами, вам потрібно буде використовувати іншу бібліотеку.

Точність. Незважаючи на те, що Compromise швидкий і простий у використанні, він не такий точний і повний, як деякі інші бібліотеки обробки природної мови. Він розроблений для практичних випадків, коли швидкість і простота важливіші за ідеальну точність.

Розширені функції: Compromise не підтримує деякі з більш просунутих завдань обробки природної мови, як-от розпізнавання іменованих об’єктів або аналіз залежностей. Якщо вам потрібні ці функції, вам знадобиться використовувати розширенішу бібліотеку.

Підсумовуючи, компроміс є хорошим вибором для простих практичних завдань обробки природної мови англійською, особливо коли важливі швидкість і простота. Однак для більш складних завдань або завдань, що включають інші мови, вам може знадобитися використовувати іншу бібліотеку.

2.2.5 Natural

Natural — це загальна бібліотека обробки природної мови (NLP) для Node.js. Він надає широкий спектр функцій для обробки та аналізу тексту. Ось деякі його особливості:

Токенізація: Natural може ділити текст на слова або речення.

Створення коренів і лемматизация: Natural може скорочувати слова до кореневої форми.

Позначення частин мови: Natural може позначати слова як іменники, дієслова, прикметники тощо.

Розпізнавання іменованих сутностей: Natural може ідентифікувати імена, місця, дати та інші сутності в тексті.

Аналіз настрою: Natural може визначити настрій фрагмента тексту, чи є він позитивним, негативним чи нейтральним.

Відстань і подібність рядків: Natural може порівнювати рядки та визначати, наскільки вони схожі.

Класифікація: Natural може класифікувати текст на різні групи.

Фонетика: Natural може працювати з фонетичними представленнями слів.

Флексія: Natural може використовувати слова у множині та в однині, а також відмінювати дієслова.

TF-IDF: Natural може виконувати частоту термінів, зворотну частоті документа, статистику, яка відображає, наскільки важливим є слово для документа в колекції чи корпусі.

Інтеграція WordNet: Natural може взаємодіяти з WordNet, великою лексичною базою даних англійської мови. Natural — це потужний інструмент для виконання завдань NLP у Node.js, але варто зазначити, що він переважно підтримує англійську мову.

Так як я новачок у машинному навчанні, Brain.js може бути більш підходящим вибором. Він розроблений таким чином, щоб бути простим і зручним

для початківців, але водночас дозволяє створювати нейронні мережі та системи рекомендацій.

Brain.js надає інтуїтивно зрозумілий API високого рівня для створення та навчання нейронних мереж. Це робить його хорошим вибором для початківців або для простих проектів. Він має хорошу документацію та багато посібників, доступних онлайн, які можуть допомогти вам розпочати машинне навчання. Також, я можу використовувати наявні знання JavaScript, щоб розпочати машинне навчання. Як і TensorFlow.js, Brain.js може працювати як у браузері, так і на Node.js, що робить його універсальним для різних середовищ.

2.3 Аналіз алгоритму K-найближчих сусідів (k-NN)

Алгоритм K-Nearest Neighbors (KNN) є свідченням незмінної сили простоти в складному світі машинного навчання. За своєю суттю KNN — це модель, яка уникає навороtів складніших алгоритмів, віддаючи перевагу простому підходу, який суперечить його потужним можливостям у задачах класифікації та регресії. KNN — це непараметричний алгоритм відкладеного навчання. Це означає, що він не робить жодних базових припущень щодо розподілу даних і відкладає всі обчислення, доки не буде запитано передбачення. Його універсальність дозволяє використовувати його для широкого кола проблем, від виявлення рукописного тексту до розрахунку кредитного рейтингу, а його логіку легко зрозуміти: подібні речі існують поруч. Під час фази навчання KNN є тихим спостерігачем, зберігаючи всі наявні дані без їх обробки. Це етап підготовки, на якому алгоритм готується до майбутніх прогнозів без формування будь-якої моделі. Подумайте про це як про запам’ятовування кожного обличчя в натовпі, щоб ви могли їх пізніше впізнати.

Коли приходить час зробити прогноз, KNN починає діяти. Він обчислює відстань між новою точкою даних і будь-якою іншою точкою, яку він бачив раніше, використовуючи такий показник відстані, як евклідова відстань. Це схоже

на вимірювання фізичної відстані між двома людьми в кімнаті, щоб визначити, хто стоїть ближче до вас.

У класифікації KNN дивиться на своїх K найближчих сусідів, подібно до людини, яка шукає найпопулярнішу думку серед групи друзів. Потім алгоритм приймає більшість голосів або найпоширеніший клас серед цих сусідів як свій прогноз. У регресії KNN використовує більш спільний підхід. Замість того, щоб шукати думку більшості, він усереднює думки або, в даному випадку, числові значення K найближчих сусідів. Іноді навіть враховує важливість думки кожного сусіда, надаючи більшої ваги тим, хто ближче. Вибір правильної кількості сусідів, K, подібний до вибору кількості думок, які слід враховувати під час прийняття рішення: правильна кількість може змінити все. Занадто мало, і на вас можуть вплинути викиди; занадто багато, і ви ризикуєте зменшити релевантність думок ваших сусідів. Вибір метрики відстані може змінити ландшафт сусідів. Незалежно від того, чи вимірюєте ви відстань прямими лініями (евклідова система) чи шляхом сітки (Манхеттен), результуючі сусіди можуть бути дуже різними, що впливає на остаточний прогноз. Простота KNN є його відмінною рисою, що робить його чудовою точкою входу для новачків у машинному навчанні. Він також неймовірно універсальний, здатний з легкістю виконувати різноманітні завдання. Його логіка інтуїтивно зрозуміла, подібно до процесів прийняття людських рішень, які часто розглядають найближчі приклади чи досвід. Однак KNN не позбавлений проблем. Його ефективність залежить від правильного вибору K і метрики відстані. Крім того, із зростанням наборів даних KNN може стати млявим, оскільки кожен прогноз потребує порівняння з кожною точкою даних у наборі даних.

Удосконалення KNN включає такі методи, як масштабування функцій, щоб гарантувати, що всі точки даних знаходяться в рівних умовах, і зважену KNN, щоб надати більшої важливості окремим сусідам над іншими. Ці коригування можуть допомогти KNN приймати кращі рішення, подібно до того, як людина розглядає поради як від близьких друзів, так і від мудрих знайомих.

2.3.1 Як працює алгоритм

Розглянемо простий сценарій, щоб зрозуміти роботу цього алгоритму. Нижче наведено розподіл червоних кіл (RC) і зелених квадратів (GS): (рис. 2.1)

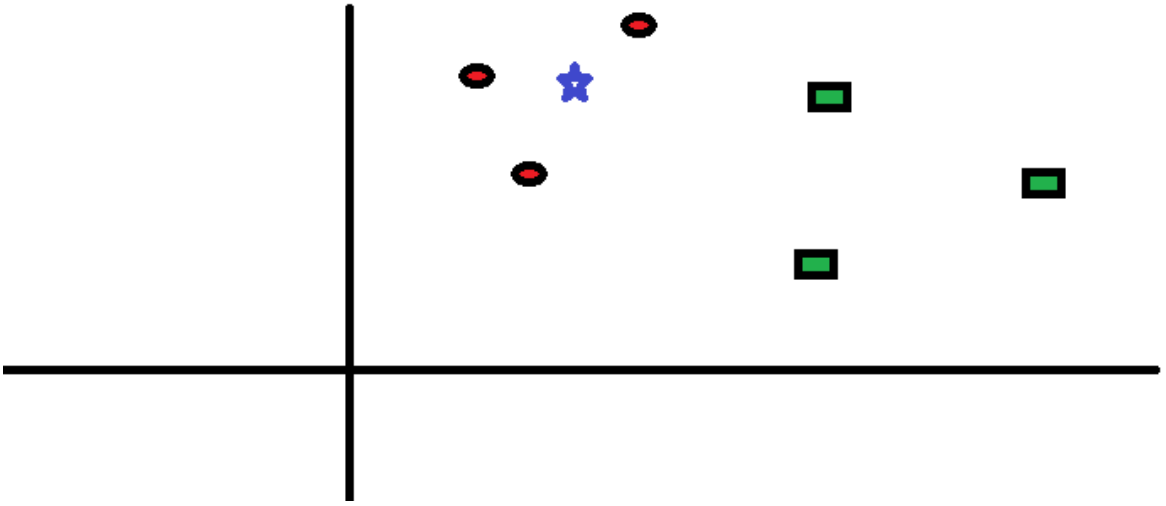

Рисунок 2.1 - Розподіл кіл

Ваша мета визначити категорію блакитної зірки (БЗ). BS може належати виключно до червоного кола (RC) або зеленого квадрата (GS), без інших можливостей. В алгоритмі K-найближчих сусідів (KNN) «K» представляє кількість найближчих сусідів, чиї голоси ми враховуємо. Припустимо, що K встановлено рівним 3. Отже, ми намалюємо коло з центром BS, розміром якого буде охоплювати лише три точки даних на площині. Зверніться до діаграми нижче для більш детальної ілюстрації. (рис. 2.2)

Усі три найближчі до BS точки належать до класу RC. Тому з високим рівнем впевненості можна стверджувати, що БС слід віднести до категорії РК. У цьому випадку рішення очевидне, враховуючи, що всі три голоси від найближчих сусідів збігаються з RC. Важливе значення в цьому алгоритмі має вибір параметра K. Згодом ми заглибимося в фактори, які необхідно враховувати, щоб визначити оптимальне значення K.

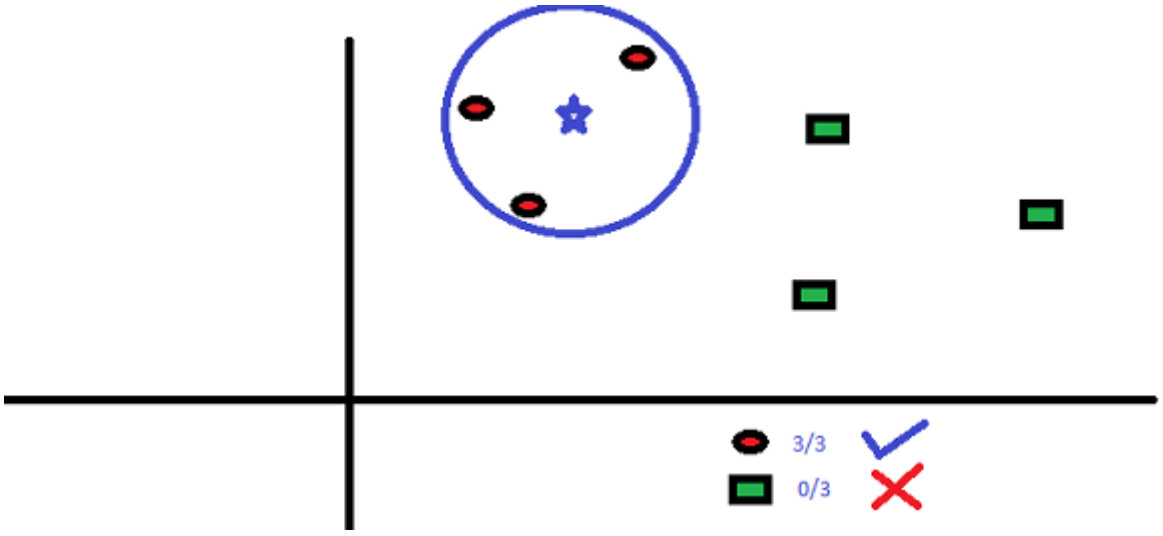

Рисунок 2.2 - Коло з центром BS

2.3.2 Як ми вибираємо фактор K?

Спочатку спробуємо зрозуміти точний вплив параметра K в алгоритмі. Посилаючись на вищезгаданий приклад, де всі шість навчальних спостережень залишаються постійними, вибране значення K дозволяє окреслити межі для кожного класу. Ці межі рішень фактично відокремлюють RC від GS. Аналогічно розглянемо вплив змінної «K» на демаркацію меж класів. Нижче наведено чіткі межі, які розділяють два класи за різних значень K. (рис. 2.3)

При уважному спостереженні стає очевидним, що межа досягає більш плавного переходу зі збільшенням значення K. Коли K наближається до нескінченності, межа зрештою стає повністю синьою або повністю червоною, залежно від переважної більшості. Оцінка частоти помилок навчання та частоти помилок перевірки є важливими параметрами для оцінки різних значень K.

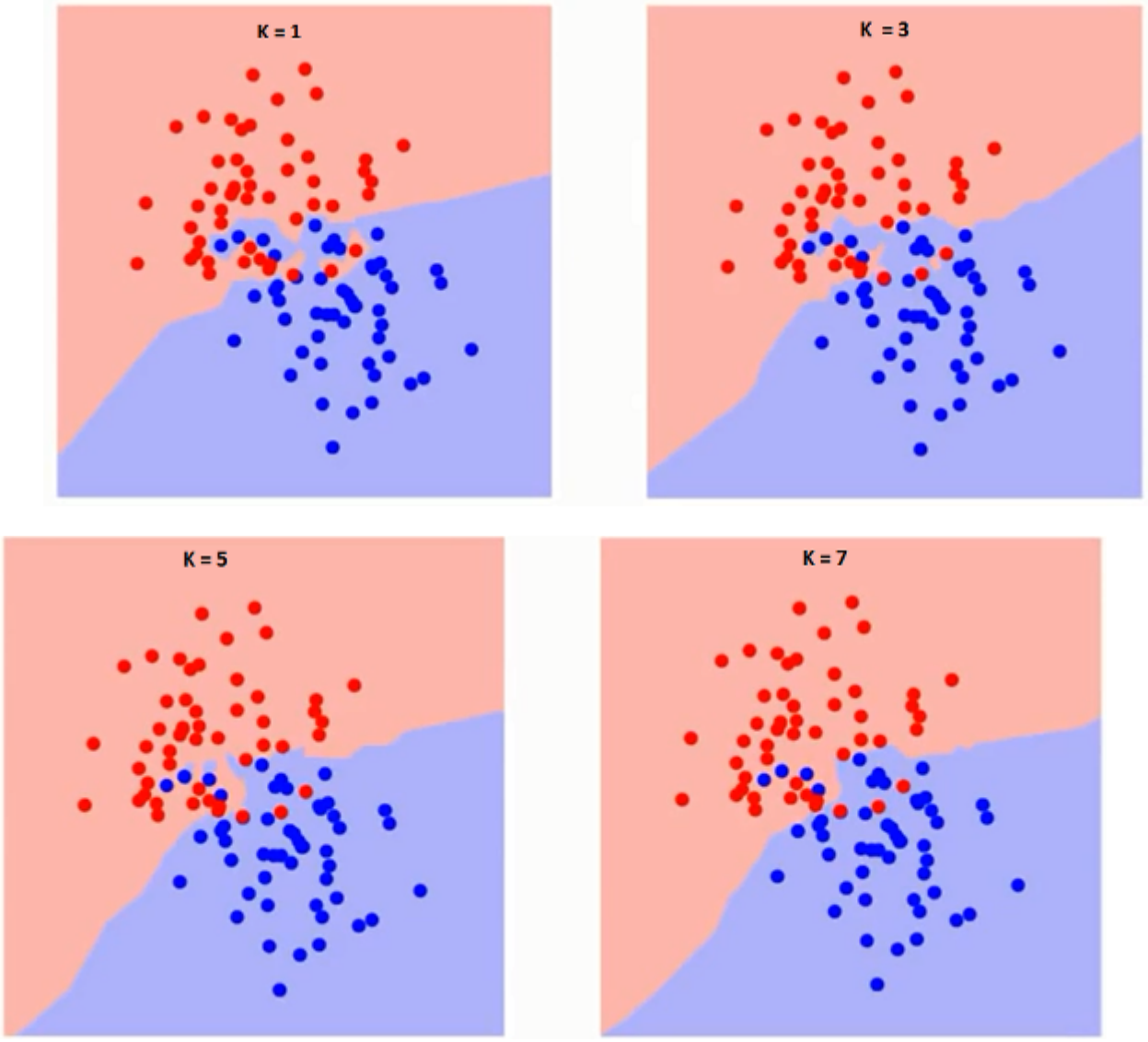

Рисунок 2.3 - Вплив змінної «K» на демаркацію між класами

Нижче представлена крива, що ілюструє частоту помилок навчання, оскільки вона змінюється залежно від різних значень K. (рис 2.4)

Як очевидно, частота помилок постійно залишається нульовою для навчальної вибірки, коли K дорівнює 1. Це пояснюється тим фактом, що найближчою точкою до будь-якої заданої точки навчальних даних є вона сама, що призводить до незмінно точних прогнозів із значенням K, рівним 1. Якби крива помилки перевірки показала порівнянну картину, оптимальним вибором для K справді було б 1.

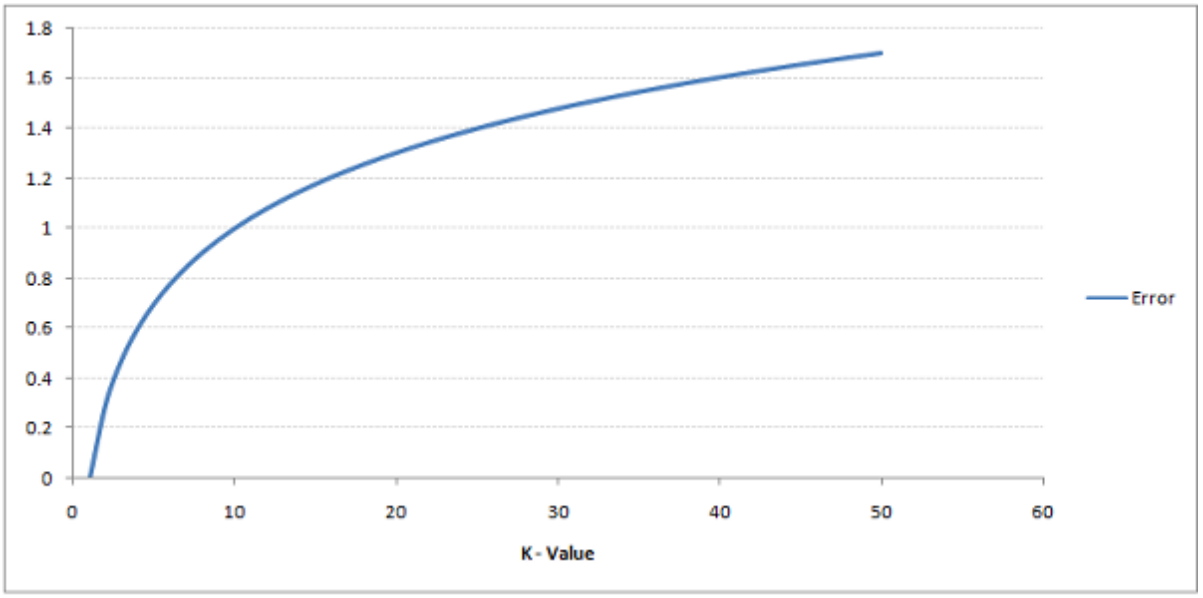

Рисунок 2.4 - Частота помилок навчання

Нижче наведена крива помилки перевірки, що зображує зміни частоти помилок з різними значеннями K (рис 2.5)

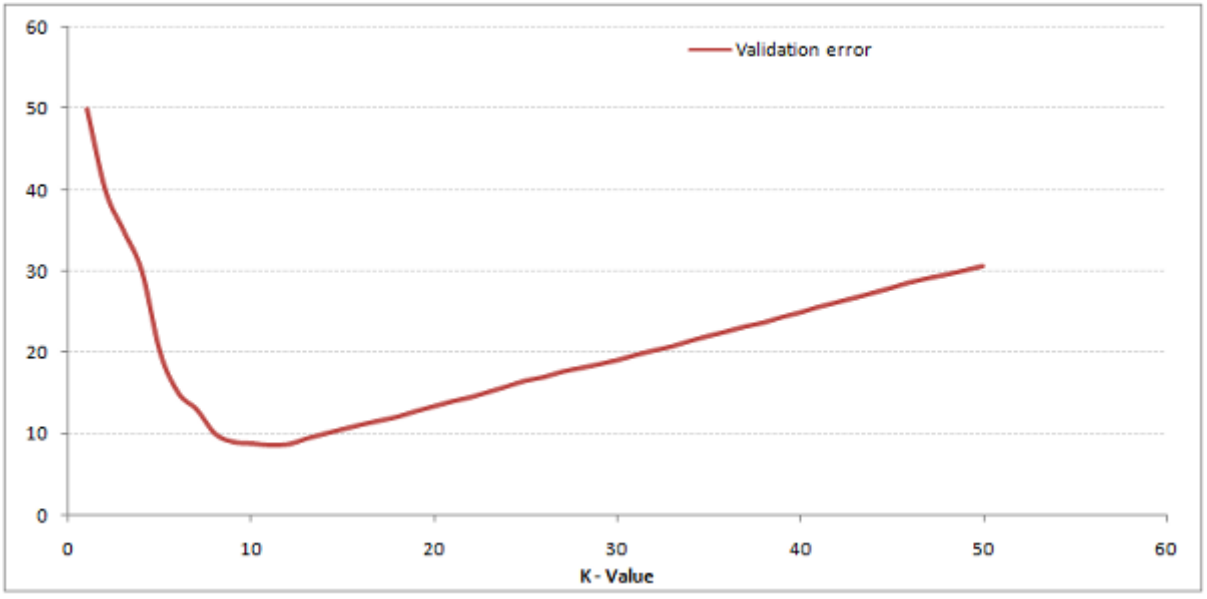

Рисунок 2.5 - Зміни частоти помилок з різними значеннями K

Це уточнення посилює оповідь. При K=1 відбулося переобладнання меж, що призвело до зниження рівня помилок до досягнення мінімуму. За межами цієї

точки частота помилок почала зростати з подальшим збільшенням K. Щоб визначити оптимальне значення K, доцільно розділити початковий набір даних на окремі набори для навчання та перевірки. Згодом побудова кривої помилок перевірки полегшує ідентифікацію оптимального значення K, яке слід використовувати для всіх наступних прогнозів. Псевдокод цього алгоритму можна записати всього в кілька рядків (рис. 2.6)

Algorithm The k-nearest neighbors classification algorithm

Input:

D: a set of training samples $\{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)\}$
k: the number of nearest neighbors
$d(\mathbf{x}, \mathbf{y})$: a distance metric
$\mathbf{x}$: a test sample

1: **for each** training sample $(\mathbf{x}_i, y_i) \in D$ **do**
2: Compute $d(\mathbf{x}, \mathbf{x}_i)$, the distance between $\mathbf{x}$ and $\mathbf{x}_i$
3: Let $N \subseteq D$ be the set of training samples with the k smallest distances $d(\mathbf{x}, \mathbf{x}_i)$
4: **return** the majority label of the samples in N

Рисунок 2.6 - k-nn алгоритм

3 РОЗРОБКА ЗАСТОСУНКУ

Перш за все, потрібно оприділити ключові технології, та проаналізувати їх переваги та недоліки. Отже - обрана мова - TYPESCRIPT. Питання полягає у тому, чому не JS, але відповідь не буде складною: TYPESCRIPT строго типізована мова програмування, у той час, коли JS динамічно типізований. Так, для розгортання середовища розробки для TYPESCRIPT складніше, бо він не працює із коробки в браузері, але це не буде проблемою, згодом опишу чому. Пройдемося по недоліках обраної мови: через те, що TYPESCRIPT базується на мові програмування JS ми маємо ті ж самі проблеми, що із JS. Перш за все, це однопоточна мова програмування, тобто ми забуваємо про багатопоточність, що трохи б'є по перфомансу самого застосунку. Тепер перейдемо до плюсів, бо їх більше. По-перше, це швидко розгорнути, та легко маштабувати. По-друге, на базі TYPESCRIPT можна працювати на різних платформах за допомогою суміжних технологій і бібліотек. Тому перейдемо до них.

3.1 Додаткові бібліотеки

NodeJS - буде використовуватися для бекендної частини застосунку.

React Native - додаткова бібліотека, яка дає можливість створити апку на базі мобільного девайсу, саме цю платформу я обрав для своєї дипломної роботи

Expo - замість того, щоб довго розбиратися з бандлерами, було прийнято рішення скористатися цією ітілітою. Так, це також вплине на швидкість роботи, але у майбутньому це можна буде зробити.

Brain.js - бібліотека для машинного навчання. Опис якої та порівняння із альтернативними бібліотеками було надано вище.

Redux Toolkit - для загального зберігання даних, без прокидання його через children

React Native Paper - вьюшна бібліотека для швидкого використання компонентів без подальшого стилізування.

3.2 Детальніше про обрані технології

3.2.1 NodeJS

Node.js є популярним вибором для багатьох розробників завдяки його характеристикам продуктивності та іншим функціям. Ось кілька причин, чому ви можете вибрати Node.js:

Неблокуючий ввід-вивід: Node.js використовує неблокуючу модель введення-виведення, керовану подіями, що робить його легким і ефективним. Це ідеально підходить для додатків реального часу з великим об’ємом даних, які працюють на розподілених пристроях.

Однопотоковий: Node.js є однопоточним, що означає, що він використовує одне ядро ЦП. Це може бути недоліком для завдань із інтенсивним використанням процесора, але це не проблема для завдань, пов’язаних із введенням/виведенням, які частіше зустрічаються на веб-серверах.

V8 JavaScript Engine: Node.js працює на V8 JavaScript engine, який компілює JavaScript безпосередньо до машинного коду, що робить його дуже швидким.

Масштабованість: Node.js розроблений таким чином, щоб бути масштабованим і може обробляти велику кількість одночасних з’єднань із високою пропускною здатністю, що робить його хорошим вибором для архітектур мікросервісів і програм реального часу.

JavaScript всюди: оскільки Node.js використовує JavaScript, ви можете використовувати ту саму мову на стороні сервера та стороні клієнта. Це може зробити розвиток більш ефективним і послідовним.

Велика екосистема: Node.js має велику та активну екосистему з величезною кількістю відкритих бібліотек, доступних для використання. Це може пришвидшити розробку та зменшити кількість коду, який потрібно написати.

Щодо статистики продуктивності, згідно з різними контрольними показниками та тестами, Node.js працює дуже добре порівняно з традиційними веб-серверами, такими як Apache, під час обробки багатьох одночасних з'єднань завдяки своїй керованій подіями архітектурі. Однак продуктивність може змінюватися залежно від конкретного робочого навантаження, тому завжди варто проводити власний порівняльний аналіз.

3.2.2 React Native

React Native виділяється як видатна структура для створення мобільних додатків за допомогою використання JavaScript і React. Кілька вагомих причин підтримують прийняття React Native:

Кросплатформна розробка: React Native полегшує створення мобільних додатків, сумісних як з платформами Android, так і з iOS, використовуючи єдину кодову базу. Такий підхід істотно скорочує час і витрати на розробку.

Продуктивність: Фреймворк перевершує інші гібридні аналоги завдяки використанню нативних компонентів замість веб-переглядів. Незважаючи на те, що React Native не такий швидкий, як нативний код, продуктивність React Native

загалом є більш ніж задовільною для більшості програм із постійними удосконаленнями.

Гаряче перезавантаження: React Native містить функцію гарячого перезавантаження, що дозволяє візуалізувати зміни в реальному часі під час розробки без необхідності перескладання всієї програми. Ця функція значно прискорює процес розробки.

JavaScript і React: Розробники, знайомі з JavaScript і React, можуть використати свій наявний досвід для створення мобільних додатків, спрощуючи процес навчання та підвищуючи ефективність розробки.

Спільнота та екосистема: React Native може похвалитися надійною та динамічною спільнотою, що забезпечує доступність обширних ресурсів, бібліотек і сторонніх плагінів, які прискорюють процес розробки.

Незважаючи на ці переваги, важливо визнати, що React Native, хоча потенційно швидший за альтернативні гібридні фреймворки, загалом не поступається нативному коду з точки зору швидкості. Однак ця невідповідність продуктивності часто непомітна для кінцевих користувачів. Згідно з опитуванням 2020 року, проведеним компанією Statista, 42% розробників програмного забезпечення вибрали React Native у кросплатформний мобільній розробці, підтвердивши її позицію як найбільш улюбленої міжплатформної мобільної системи.

Недоліки:

Обмежений доступ до нативних функцій: React Native може мати обмеження в доступі до певних нативних функцій порівняно з повністю нативною розробкою.

Крива навчання для нативних модулів: Інтеграція нативних модулів може потребувати додаткового навчання розробників.

Залежність від модулів сторонніх розробників: використання модулів сторонніх розробників може становити ризики, якщо їх не обслуговувати чи оновлювати належним чином.

Приклади нативного використання React:

Facebook: React Native виник у Facebook і широко використовується для розробки мобільних додатків.

Instagram: Instagram використовує React Native для частини своєї програми, демонструючи її універсальність у великомасштабних програмах.

UberEATS: програма UberEATS використовує React Native для своїх крос-платформних можливостей.

Ці випадки підкреслюють широке впровадження та застосовність React Native у різноманітних сферах, підтверджуючи його статус кращого вибору для розробки мобільних додатків.

3.2.3 Expo

Розробка мобільних додатків стала свідком сплеску фреймворків та інструментів, спрямованих на спрощення процесу розробки та забезпечення сумісності між платформами. Одним із таких помітних гравців у цій сфері є Expo. У цьому дослідженні ми заглибимося в обґрунтування використання Expo, враховуючи його переваги, характеристики продуктивності, потенційні недоліки та реальні приклади його застосування.

Переваги Expo:

Спрощений процес розробки: Expo виділяється своєю здатністю оптимізувати розробку мобільних додатків. Надаючи повний набір попередньо

зібраних компонентів і бібліотек, Expo прискорює цикл розробки, особливо для проектів із простими вимогами.

Кросплатформна сумісність: Розробники високо оцінюють здатність Expo сприяти міжплатформній розробці як для iOS, так і для Android. Це дозволяє створювати програми з використанням уніфікованої кодової бази, сприяючи ефективності та узгодженості на різних платформах.

Оновлення по повітрю: Expo підтримує оновлення по повітрю, функцію, яка забезпечує безперебійне розгортання оновлень для користувачів без необхідності завантажувати їх із магазину програм. Це не тільки покращує гнучкість циклу розробки, але й гарантує, що користувачі швидко отримають найновіші функції та виправлення помилок.

Вбудовані служби: Expo об’єднує такі вбудовані служби, як push-сповіщення, автентифікація та аналітика. Ця інтеграція зменшує потребу розробників окремо налаштовувати ці служби та керувати ними, пропонуючи більш узгоджений досвід розробки.

Легкість входу: Особливо корисно для тих, хто тільки починає розробляти мобільні пристрої, Expo забезпечує доступну точку входу. Вимагаючи мінімального налаштування та конфігурації, Expo дозволяє розробникам швидко розпочати роботу, не заглиблюючись у тонкощі нативної розробки.

Зауваження щодо продуктивності: Однак важливо визнати компроміс, який приходить із зручністю Expo. Хоча це прискорює розробку, Expo може пожертвувати деяким рівнем продуктивності порівняно з повністю нативними рішеннями. Цей компроміс дуже важливо враховувати, особливо для програм із критично важливими вимогами до продуктивності.

Потенційні недоліки: Обмежений доступ до рідного модуля: Expo накладає обмеження на доступ до певних нативних модулів. Це обмеження може вплинути на проекти, які вимагають широкої інтеграції з рідними функціями, і розробникам слід ретельно оцінити, чи Expo відповідає конкретним потребам їх проекту.

Залежність від Expo Services: Додатки Expo залежать від послуг Expo. Будь-які обмеження або збої в цих службах можуть потенційно вплинути на функціональність розгорнутих програм, що потребує розгляду потенційної залежності від зовнішніх служб.

Реальні приклади використання Expo:

Додаток *Khan Academy* використовує Expo для розробки мобільних пристроїв, демонструючи простоту Expo та крос-платформні можливості в освітньому секторі.

Artsy, онлайн-платформа для відкриття та колекціонування мистецтва, включає Expo для розробки своїх мобільних додатків. Цей приклад підкреслює універсальність Expo у сфері мистецтва та культури.

Гаманець Celo: Wallet, криптовалютний гаманець, використовує Expo для кросплатформної розробки. Ця програма підкреслює актуальність Expo у секторі фінансових технологій, демонструючи її адаптивність до різних областей.

Підсумовуючи, Expo постає як переконливе рішення для певних сценаріїв розробки мобільних додатків. Пропонуючи простоту, крос-платформну сумісність і вбудовані служби, розробники повинні ретельно оцінювати компроміси, особливо у випадках, коли детальний контроль або оптимальна продуктивність мають першочергове значення. Реальні приклади Expo свідчать про його універсальність і демонструють його потенціал у різних секторах мобільних додатків.

3.2.4 Redux Toolkit

Redux Toolkit — це набір інструментів і утиліт, призначених для спрощення та оптимізації використання Redux у програмах React. Подібно до Expo, він має на меті покращити досвід розробки, надаючи набір умовностей і абстракцій. Тут ми досліджуємо причини вибору Redux Toolkit, його переваги, характеристики продуктивності, потенційні недоліки та реальні приклади його застосування.

Привабливість Redux Toolkit:

Спрощена конфігурація Redux: Однією з головних переваг Redux Toolkit є його здатність спрощувати часто багатослівну та шаблонну конфігурацію Redux у програмі React. Він представляє стислий синтаксис, що робить його більш доступним для розробників і зменшує кількість коду, необхідного для керування станом.

Вбудоване проміжне ПЗ: Redux Toolkit постачається з вбудованим проміжним програмним забезпеченням, включаючи проміжне програмне забезпечення реєстратора та thunk. Це позбавляє розробників від необхідності вручну налаштовувати проміжне програмне забезпечення, пропонуючи більш інтегрований і спрощений досвід розробки.

Помічники незмінності: Незмінні оновлення є фундаментальним аспектом Redux, і Redux Toolkit містить службові функції для спрощення процесу. Функція createSlice, наприклад, генерує незмінні редуктори без необхідності розробникам явно обробляти незмінність.

Інтеграція DevTools: Redux Toolkit легко інтегрується з Redux DevTools, надаючи розробникам потужні можливості налагодження. Ця інтеграція полегшує

перевірку дій, змін стану та налагодження подорожей у часі для більш ефективного робочого процесу розробки.

Покращена продуктивність: Хоча Redux Toolkit зосереджений на ергономіці розробника, його умовності та абстракції часто призводять до більш ефективного та продуктивного коду. Обробляючи загальні шаблони та оптимізуючи під капотом, це сприяє більш плавній роботі програм.

Зауваження щодо продуктивності: Redux Toolkit пропонує кілька оптимізацій для підвищення продуктивності, але розробники повинні пам’ятати про особливі потреби своїх програм. Хоча це загалом призводить до ефективного управління станом, належна архітектура програми та методи оптимізації є важливими для оптимальної продуктивності у великих програмах.

Потенційні недоліки:

Крива навчання: Для розробників, які не знайомі з Redux, умовності, запроваджені Redux Toolkit, спочатку можуть стати кривою навчання. Розуміння абстракцій і архітектури Redux має вирішальне значення для ефективного використання набору інструментів.

Сумісність з існуючим кодом: Інтеграція Redux Toolkit в існуючу кодову базу Redux може потребувати деяких коригувань. Розробникам слід ретельно оцінити вплив на наявну функціональність і за потреби відредагувати код.

Підсумовуючи, Redux Toolkit надає набір інструментів, які значно спрощують використання Redux у програмах React. Хоча він пропонує переваги з точки зору досвіду розробника та оптимізації продуктивності, ретельний аналіз кривої навчання та сумісності з існуючим кодом є важливим. Реальні приклади демонструють його універсальність у різних областях застосування,

демонструючи його застосовність у сценаріях управління державою в різних секторах.

3.2.5 React Native Paper

React Native Paper — це популярна бібліотека інтерфейсу користувача для додатків React Native, яка пропонує попередньо розроблені компоненти та стилі для оптимізації розробки естетично привабливих і чуйних інтерфейсів користувача. Давайте дослідимо причини вибору React Native Paper, його переваги, міркування щодо продуктивності, потенційні недоліки та реальні приклади його застосування.

Привабливість React Native Paper:

Послідовний дизайн матеріалів: React Native Paper розроблено з урахуванням принципів матеріального дизайну, що забезпечує послідовний і візуально привабливий вигляд і відчуття на різних платформах. Це спрощує процес створення відточеного та професійного інтерфейсу користувача.

Настроювані компоненти: Надаючи набір готових до використання компонентів, React Native Paper також легко налаштовується. Розробники можуть легко налаштувати зовнішній вигляд і поведінку компонентів відповідно до конкретних вимог до дизайну своїх програм.

Підтримка тем: React Native Paper має підтримку тем, що дозволяє розробникам визначати узгоджений візуальний стиль для своїх програм. Ця можливість створення тем забезпечує послідовну мову дизайну в усьому додатку та полегшує обслуговування.

Спеціальні можливості: Доступність є критично важливим аспектом розробки мобільних додатків, і React Native Paper містить у своїх компонентах функції доступності. Це гарантує, що програми, створені за допомогою React Native Paper, придатні для використання різноманітною аудиторією, включаючи користувачів з обмеженими можливостями.

Інтеграція з React Navigation: React Native Paper легко інтегрується з React Navigation, популярною бібліотекою навігації для React Native. Ця інтеграція спрощує реалізацію шаблонів навігації в програмі.

Міркування щодо продуктивності: React Native Paper розроблено для забезпечення продуктивної та плавної взаємодії з користувачем. Однак загальна продуктивність програми залежить від різних факторів, включаючи складність інтерфейсу користувача, специфікації пристрою та ефективність інших бібліотек і компонентів, які використовуються разом з React Native Paper.

Потенційні недоліки:

Обмежене налаштування в деяких випадках: Хоча React Native Paper дуже легко налаштовується, можуть виникнути ситуації, коли розробникам потрібен більш детальний контроль над зовнішнім виглядом або поведінкою певних компонентів. У таких випадках може знадобитися додаткове налаштування або використання альтернативних рішень.

Вплив розміру комплекту: Інтеграція бібліотеки інтерфейсу користувача, як-от React Native Paper, може сприяти збільшенню розміру набору програм. Розробники повинні пам’ятати про вплив на час завантаження, особливо для користувачів з обмеженою пропускною здатністю мережі.

Підсумовуючи, React Native Paper є переконливим вибором для розробників, які прагнуть прискорити процес розробки інтерфейсу користувача в програмах React Native. Його послідовність у дизайні, можливості налаштування, підтримка тем та інтеграція з React Navigation сприяють його популярності. Однак розробники повинні пам’ятати про можливі недоліки та оцінювати, чи відповідає

React Native Paper конкретним потребам і вимогам до налаштування їхніх проектів. Приклади з реального світу демонструють його універсальність у різних сферах застосування, демонструючи його ефективність у створенні візуально привабливих і зручних інтерфейсів.

3.3 Архітектура проекту

Структура папок буде виглядати майже звично, щодо фронтової частини реалізації це буде модульна структура (рис. 3.1)

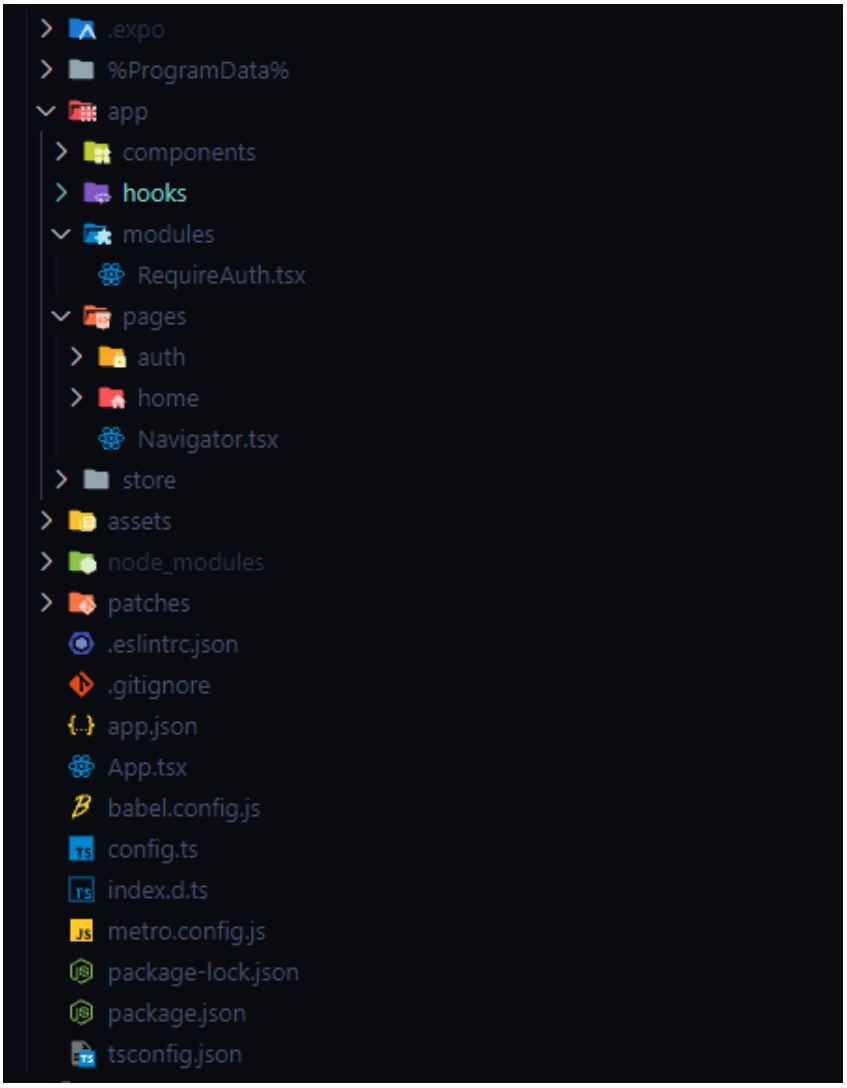

Рисунок 3.1 - Модульна структура

Вхідним файлом по дефолту буде App.jsx, бо React Native не працює із html. Також проведемо налаштування App.tsx (рис. 3.2)

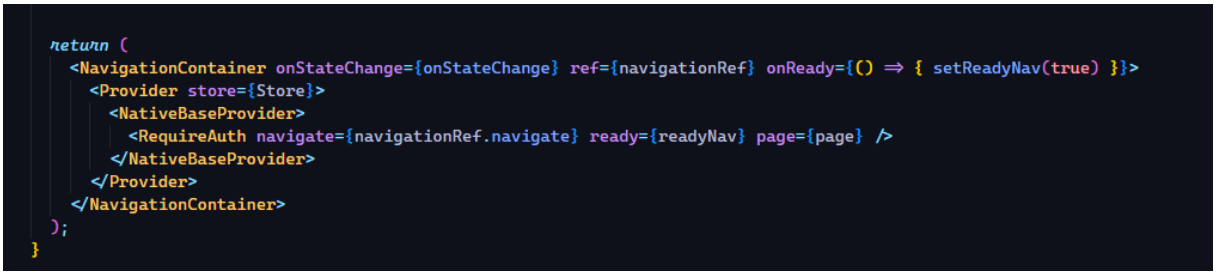

Рисунок 3.2 - Налаштування App.tsx

та головну конфігурацію для Expo, без якої не буде працювати застосунок (рис. 3.3). Найперше, що потрібно зробити - роутинг між сторінками. Компонент NavigationContainer — це контейнер для всіх навігаційних функцій вашої програми. Ви повинні відтворити лише один NavigationContainer у своїй програмі.

Для більш складних структур навігації вам також потрібно буде встановити та використовувати одну з бібліотек навігатора, наприклад @react-navigation/stack для навігації стеком або @react-navigation/bottom-tabs для навігації вкладками. Ці бібліотеки містять такі компоненти, як StackNavigator і BottomTabNavigator, які можна використовувати для визначення різних екранів і шляхів навігації у вашій програмі. До них ми повернемось трохи згодом.

Ця функція TypeScript, performLoadingTasks, є асинхронною функцією, яка виконує деякі початкові завдання завантаження для програми React Native. Ось розбивка того, що він робить:

SplashScreen.preventAutoHideAsync(): цей рядок коду запобігає автоматичному приховуванню екрану-заставки. У програмах React Native під час початкового завантаження програми зазвичай відображається екран-заставка. Запобігши автоматичному приховуванню, ви можете контролювати, коли воно зникне відповідно до потреб вашої програми.

```
{
  "expo": {
    "name": "messenger",
    "slug": "messenger",
    "version": "1.0.0",
    "orientation": "portrait",
    "icon": "./assets/icon.png",
    "userInterfaceStyle": "light",
    "splash": {
      "image": "./assets/splash.png",
      "resizeMode": "contain",
      "backgroundColor": "#ffffff"
    },
    "assetBundlePatterns": [
      "**/*"
    ],
    "ios": {
      "supportsTablet": true
    },
    "android": {
      "adaptiveIcon": {
        "foregroundImage": "./assets/adaptive-icon.png",
        "backgroundColor": "#ffffff"
      }
    },
    "web": {
      "favicon": "./assets/favicon.png"
    }
  }
}
```

Рисунок 3.3 - Головна конфігурація Expo

Font.loadAsync(): Ця функція використовується для завантаження файлів шрифтів у вашу програму. У цьому випадку завантажуються два шрифти: «OpenSans-Regular» і «OpenSans-Bold». Функція require використовується для включення файлів шрифтів із локальної файлової системи. Це асинхронна операція, тобто її виконання може зайняти деякий час, особливо якщо файли шрифтів великі. Ключове слово await використовується для призупинення виконання функції до завантаження шрифтів.

Обробка помилок: структура try/catch/finally використовується для обробки будь-яких помилок, які можуть виникнути під час виконання функції. Якщо виникає помилка під час запобігання приховуванню або завантаженню шрифтів екрану-заставки, вона буде перехоплена та зареєстрована на консолі за допомогою console.warn(e). Незалежно від того, чи сталася помилка, блок finally завжди виконуватиметься.

setAppIsReady(true): цей рядок коду виконується в блоці finally, що означає, що він запуститься після завершення завантаження шрифту, незалежно від того, було воно успішним чи ні. setAppIsReady — це функція, яка передається як аргумент для виконанняЗавдань завантаження. Ця функція, ймовірно, є установником стану з хука useState у компоненті React. Викликаючи setAppIsReady(true), функція сигналізує, що програма завершила свої початкові завдання завантаження та готова до відтворення. (рис. 3.4)

```
const performLoadingTasks = async (setAppIsReady) => {
  try {
    // Keep the splash screen visible while we fetch resources
    await SplashScreen.preventAutoHideAsync();
      // Preload fonts
    await Font.loadAsync({
      'OpenSans-Regular': require('./assets/fonts/OpenSans-Regular.ttf'),
      'OpenSans-Bold': require('./assets/fonts/OpenSans-Bold.ttf')
    });
  } catch (e) {
    console.warn(e);
  } finally {
    // Tell the application to render
    setAppIsReady(true);
  }
};
```

Рисунок 3.4 - performLoadingTasks функція

Далі - компонента авторізації під назвою RequireAuth, який використовується для обробки автентифікації в програмі React Native. Ось розбивка того, що він робить:

isLoggined(): Ця функція імпортована з іншого модуля та, імовірно, використовується для перевірки, чи ввійшов користувач. Результат зберігається в константі authed. хук useEffect: цей хук використовується для виконання побічних ефектів у функціональних компонентах. У цьому випадку він перевіряє, чи користувач не автентифікований (!authed) і чи готовий проп є істинним. Якщо обидві умови виконуються і поточна сторінка не є «Вхід» або «Реєстрація», відбувається перехід на сторінку «Вхід». Цей ефект спрацьовує щоразу, коли змінюється сторінка, аутентифікація або готові атрибути. <Navigator />: це компонент, який обробляє навігацію між різними екранами в програмі. Він відображається незалежно від статусу автентифікації.

Компонент RequireAuth, використовується як оболонка для частин програми, які вимагають автентифікації користувача. Він перевіряє статус автентифікації та перенаправляє неавтентифікованих користувачів на сторінку «Вхід», якщо вони вже не перебувають на сторінці «Вхід» або «Реєстрація». Готовий prop, використовується для очікування завершення певної ініціалізації перед перевіркою статусу автентифікації.

Наступна по значимості структура навігації (рис. 3.5) для програми React Native за допомогою пакета @react-navigation/native-stack. Почнемо із операторів імпорту: необхідні компоненти та функції імпортуються з відповідних модулів. createNativeStackNavigator імпортується з @react-navigation/native-stack, а компоненти Home, Login, Registration та Interests імпортуються з відповідних файлів. Далі - createNativeStackNavigator, ця функція використовується для створення нового стекового навігатора. Навігатор стека — це тип навігатора, який представляє екрани таким чином, що кожен новий екран розміщується на вершині

стека. Компонент функції Navigator - це компонент функції React, який повертає компонент Stack.Navigator. Цей компонент є контейнером для компонентів екрана.

Stack.Navigator: цей компонент використовується для визначення навігаційного стеку. Для атрибута initialRouteName встановлено значення «Home», що означає, що головний екран буде першим екраном, який відображатиметься під час запуску програми. Опис screenOptions використовується для налаштування зовнішнього вигляду та поведінки екранів у навігаторі. У цьому випадку встановлено значення { headerShown: false }, що означає, що панель заголовка не відображатиметься на жодному екрані.

Stack.Screen: цей компонент використовується для визначення екрана в стеку навігації. Кожен компонент Stack.Screen має властивість імені, яка використовується для посилання на екран, і властивість компонента, яка є компонентом React, який відображатиметься, коли цей екран активний. У цьому випадку визначено чотири екрани: «Вхід», «Реєстрація», «Домашня сторінка» та «Інтереси».

```tsx
import { createNativeStackNavigator } from '@react-navigation/native-stack';
//
import Home from './home/Home';
import { Login } from './auth/Login';
import { Registration } from './auth/Registration';
import { Interests } from './auth/Interests';

const Stack = createNativeStackNavigator();

export function Navigator() {
  //
  return (
  <Stack.Navigator initialRouteName="Home" screenOptions={ { headerShown: false } }>
    <Stack.Screen name='Login' component={Login} />
    <Stack.Screen name='Registration' component={Registration} />
    <Stack.Screen name="Home" component={Home} />
    <Stack.Screen name="Interests" component={Interests} />
  </Stack.Navigator>
  )
}
```

Рисунок 3.5 - Компонента Navigator

Таким чином, цей код визначає навігаційну структуру стека з чотирма екранами. Спочатку відображається головний екран, а панель заголовка прихована на всіх екранах.

Далі дефолтні сторінки логіну, реєстрації та головної сторінки, яка, у свою чергу, ділится на 3 секції, завдяки @react-navigation/bottom-tabs (рис. 3.6), що дає нам можливість додати нижнє меню, через яке можна працювати одразу із трьома секціями: сторінкою пошуку людей, людей які хотіли б поспілкуватися та моя інформація як користувача. Також, сторінка авторизації ділится на 2 частини:

```
essenger > app > pages > home > Home.tsx > ...
import { createBottomTabNavigator } from '@react-navigation/bottom-tabs';

import Cards from './Cards';
import Likes from './Likes';
import User from './User';

import CardsSvg from '../../../assets/icons/cards-outline.svg';
import UserSvg from '../../../assets/icons/account-outline.svg';
import LikeSvg from '../../../assets/icons/heart-multiple-outline.svg';

const Tab = createBottomTabNavigator();

export default function Home() {
  return (
    <Tab.Navigator
      screenOptions={({ route }) => ({
        tabBarIcon: ({ focused, color, size }) => {
          if (route.name === 'Cards') return <CardsSvg width={15} height={15} />;
          if (route.name === 'Likes') return <LikeSvg width={15} height={15} />;
          if (route.name === 'User Info') return <UserSvg width={15} height={15} />;
        },
      })}
      backBehavior='none'
    >
      <Tab.Screen
        name="Cards"
        component={Cards}
        options={{ headerShown: false }}
      />
      <Tab.Screen
        name="Likes"
        component={Likes}
        options={{ headerShown: false }}
      />
```

Рисунок 3.6 - 3 секції головної сторінки

головні дані, ПІБ, номер телефону, дані соц мереж, та сторінку інтересів, де ти обираєш починаючи з верхнього рівня, наприклад, книги, а закінчуєш найнижчим, наприклад, “Потік”. Таким чином, маємо структуру: Книги -> Психологія -> Чин Сек Михай -> Потік. Кожен рівень має свою пріоритизацію, де найвищий - 1, а найнижчий - 4, якщо ми кажемо про структуру із 4 елементів. Принцип додавання нових інтересів (рис 3.7) - random Constant: цей рядок коду фільтрує масив selectedInterests за допомогою Math.random() - 0,5. Це буде випадковим чином включати або виключати кожен елемент у масиві selectedInterests, у результаті чого буде створено новий масив із випадковою підмножиною вихідних елементів. Константа randomsub: цей рядок коду відображає випадковий масив і для кожного інтересу фільтрує свій масив підінтересів (якщо він існує) таким же випадковим чином, як і раніше. Потім метод flat() використовується для зведення отриманого масиву масивів в єдиний масив. Повернене значення – це новий масив, який включає всі елементи з масиву інтересів і масиву randomsub. Проте будь-які елементи, включені до масиву selectedInterests, відфільтровуються. Це означає, що showedInterests міститиме всі інтереси, які наразі не вибрано, а також випадкову підмножину підінтересів вибраних інтересів.

```
const showedInterests = useMemo(() => {
  const random = selectedInterests.filter(() => Math.random() - 0.5);
  const randomsub = random.map((interest) => (interest.subinterests || []).filter(() => Math.random() - 0.5)).flat();

  return [...interests, ...randomsub].filter((interest) => !selectedInterests.includes(interest));
}, [interests, selectedInterests]);
```

Рисунок 3.7 - Принцип додавання нових інтересів

Таким чином, цей код використовується для створення списку інтересів для показу користувачеві, який включає всі інтереси, які наразі не вибрано, а також випадкову підмножину підінтересів обраних інтересів. Список перераховується щоразу, коли змінюються інтереси або вибрані інтереси.

У цьому фрагменті коду TypeScript описано створення сховища Redux для програми React за допомогою Redux Toolkit. Початковий розділ містить необхідні оператори імпорту, у яких імпортуються такі функції та модулі, як `configureStore` з Redux Toolkit, `devToolsEnhancer` з `remote-redux-devtools` та `authSlice` зі сховища авторизації. Центральна точка коду полягає у функції `configureStore`, яка відповідає за створення сховища Redux. У межах цієї функції сховище ретельно налаштовано з редуктором, інструментами розробника, покращувачами та проміжним програмним забезпеченням. Редуктор, об’єкт, що відображає зрізи стану на їхні відповідні редуктори, переважно містить окремий зріз — auth — який обробляється `authSlice`. Опція `devTools`, залежно від режиму розробки (process.env.NODE_ENV !== 'production'), вмикає розширення Redux DevTools. Масив `enhancers` охоплює розширювачі магазину, включаючи `devToolsEnhancer` з опцією `realtime`, встановленою на true, що означає представлення змін стану в реальному часі.

Конфігуруємо store (рис 3.8) Розділ `middleware`, по суті, функція, яка дає проміжне програмне забезпечення, застосоване до сховища, використовує проміжне програмне забезпечення за замовчуванням, хоча для параметра `serializableCheck` встановлено значення false. Це навмисне вимкнення перевірки гарантує, що всі дії та стан не потребують суворої серіалізації.

Результуюча сутність, магазин Redux, постає як суть цього процесу. Він створюється функцією `configureStore` і згодом експортується як стандартний експорт модуля. Примітно, що визначення типів `RootState` і `AppDispatch` походять зі сховища, представляючи все дерево стану Redux і тип методу відправлення сховища відповідно. Ці типи служать невід’ємними компонентами для підтримки цілісності типів під час взаємодії зі станом Redux або диспетчеризації дій у всій програмі.

```
import { configureStore } from '@reduxjs/toolkit';  23.9k (gzipped: 8.3k)
import devToolsEnhancer from 'remote-redux-devtools';  161.9k (gzipped: 47.3k)

import { authSlice } from '../pages/auth/store';

const store = configureStore({
  reducer: {
    auth: authSlice
  },
  devTools: process.env.NODE_ENV !== 'production',
  enhancers: [devToolsEnhancer({ realtime: true })],
  middleware: getDefaultMiddleware => getDefaultMiddleware({
    serializableCheck: false
  }),
});

export default store;

export type RootState = ReturnType<typeof store.getState>
export type AppDispatch = typeof store.dispatch
```

Рисунок 3.8

Більш детально по home сторінці. Приклад головної сторінки та нижнього меню (рис 3.9). У цьому коді TypeScript/React Native компонент під назвою «Cards» ретельно визначено, щоб представити візуально привабливу карусель із зображеннями разом із додатковою інформацією, такою як ім’я, список інтересів і опис. Реалізація розгортається наступним чином:

Початковий розділ містить основні оператори імпорту, представляючи необхідні компоненти, хуки та модулі. Примітно, що компоненти «Carousel» і «Pagination» імпортуються з «react-native-snap-carousel», а спеціальний хук «useAppSelector» використовується для доступу до стану зі сховища Redux.

Суть коду полягає у функціональному компоненті «Cards», де хук «useAppSelector» використовується для отримання інтересів зі стану «auth» у сховищі Redux.

Згодом додається масив «carouselItems», який складається з об’єктів, кожен з яких інкапсулює URI для зображення. Ці зображення разом утворюють зміст каруселі.

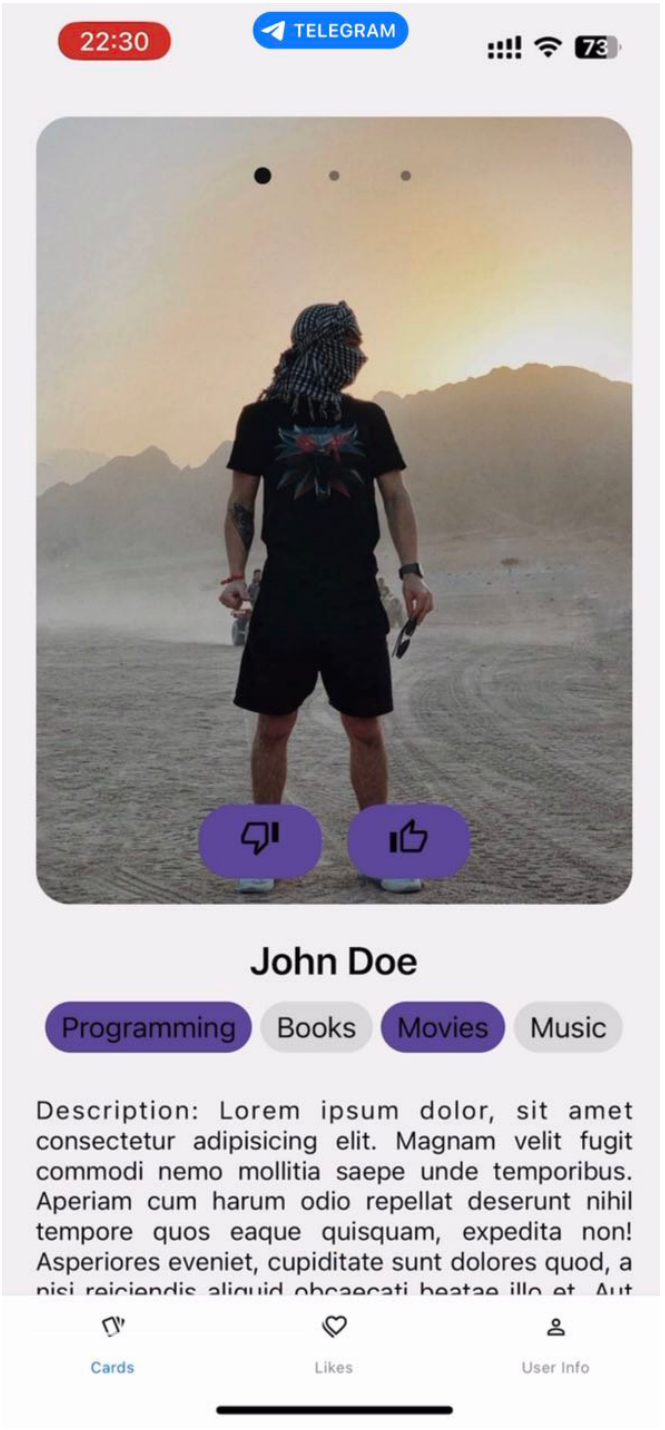

Рисунок 3.9 - Головна сторінка

Важливим аспектом коду є функція "renderItem", призначена для відтворення кожного елемента в каруселі. Ця функція приймає об’єкт із властивостями «item» і «index» і повертає компонент «View», що містить

компонент «Image». Джерело «Зображення» динамічно встановлюється на URI відповідного елемента.

Інструкція повернення компонента інкапсулює "ScrollView", що містить "SafeAreaView". У цій структурі організовано кілька компонентів:

Компонент «Пагінація», відповідальний за представлення індикатора сторінки для каруселі.

Компонент «Карусель», який керує відображенням каруселі зображень. Він використовує функцію «renderItem» для візуалізації елементів і використовує властивість «onSnapToItem» для оновлення стану «activeSlide», коли користувачі переходять до різних елементів.

Дві кнопки, одна для вподобання, інша для відміни поточного елемента каруселі. Ці кнопки мають піктограми SVG і викликають повідомлення журналу консолі після натискання.

Компонент "Текст", що відображає ім'я "Джон Доу".

Комбінація компонентів «Перегляд» і «Текст», ретельно організованих для демонстрації списку інтересів. Кожен інтерес міститься в "Виданні" з вбудованим "Текстом", а стиль "Вигляду" визначається динамічно.

Інший компонент "Текст", який відображає попередньо визначений опис.

Крім того, компонент «StatusBar» використовується для демонстрації рядка стану зі стилем “auto”.

По суті, компонент «Картки» складно збирає візуально привабливу карусель із зображеннями, інтерактивними кнопками та інформативним вмістом, таким чином пропонуючи комплексний інтерфейс користувача. Інтеграція пошуку стану Redux і динамічного рендерингу додатково збагачує функціональність компонента.

Опишемо реєстрацію, щоб розуміти, як працює внесення даних (рис 3.10)

```
  }

  return (
    <TouchableWithoutFeedback onPress={Keyboard.dismiss}>
      <SafeAreaView style={styles.container}>
        <Text style={styles.title}> Register </Text>
          <FormControl isRequired>
            <Stack mx="4">
              <FormControl.Label>Email</FormControl.Label>
              <Input type="text" placeholder="mail" onChangeText={onInput('email')} />

              <FormControl.Label>Username</FormControl.Label>
              <Input type="text" placeholder="username" onChangeText={onInput('username')} />

              <FormControl.Label>Password</FormControl.Label>
              <Input type="password" placeholder="password" onChangeText={onInput('pass')} />

              <FormControl.Label>Confirm password</FormControl.Label>
              <Input type="password" placeholder="confirm password" onChangeText={onInput('conPass')} />
              <FormControl.HelperText>
                Must be atleast 6 characters.
              </FormControl.HelperText>

              <FormControl.ErrorMessage leftIcon={<WarningOutlineIcon size="xs" />}>
                Atleast 6 characters are required.
              </FormControl.ErrorMessage>
              <Text
                style={styles.link}
                onPress={onLogin}
              >
                Already registered? Log in here.
              </Text>
              <Button mode="contained" style={styles.button} onPress={onSubmit}> To next page </Button>
              <Text style={styles.error}> {error} </Text>
            </Stack>
          </FormControl>
        {/* </Box> */}
      </SafeAreaView>
    </TouchableWithoutFeedback>
  );
}
```

Рисунок 3.10 - Компонента реєстрації

У цьому коді TypeScript React ретельно розроблено компонент `Registration`, щоб полегшити реєстрацію користувачів у фреймворку мобільних додатків на основі React Native. Давайте приступимо до поглибленого вивчення його складних функцій:

Починаємо з серії інструкцій імпорту, які містять різноманітний набір компонентів і хуків із різних бібліотек. Вони охоплюють компоненти інтерфейсу користувача, отримані з `native-base` і `react-native-paper`, користувацькі хуки, витягнуті зі спеціального файлу `useState`, дії, що походять із центрального файлу `store`, і стилістичні елементи, ретельно створені в окремому ` файл стилів.

В основі цього коду лежить визначення компонента функції ``Реєстрація". Цей компонент, параметризований `props`, який, зокрема, включає незамінний об’єкт `navigation` з React Navigation, служить основою для робочих процесів реєстрації користувачів.

У пошуках ефективності керування станом код елегантно використовує хуки useAppSelector і useAppDispatch. Ці хуки, бездоганно інтегровані з Redux, дають компоненту можливість як отримувати доступ до глобального стану, так і відправляти дії, коли це необхідно.

Навігаційна хореографія займає центральне місце з чіткими обробниками подій. Функція onLogin під час виклику плавно спрямовує користувача до екрана входу, втілюючи плавні переходи в програмі.

Обробник події `onSubmit` тонко організовує процес реєстрації користувача. Він спритно відправляє дію `onRegisterFirstStep`, сповіщаючи про початок шляху реєстрації. Після тріумфального вирішення обіцянки система навігації спрямовує користувача до екрана «Інтереси», сповіщаючи про успішний прогрес реєстрації.

Функція onInput додатково прикрашає ансамбль обробників подій. Ця універсальна функція після виклику динамічно відправляє дію `setInput` із наданими параметрами `name` і `value`. Ця стратегічна взаємодія лежить в основі динамічного оновлення стану компонента на основі введених користувачем даних.

Фінальне рішення розгортається в методі візуалізації, де формується ретельно створений елемент JSX. Цей елемент містить реєстраційну форму користувача з полями електронної пошти, імені користувача та підтвердження пароля. Навігаційні містки майстерно вплетені в тканину форми, надаючи користувачам посилання на сторінку входу. Кнопка «Надіслати» є шлюзом для

передачі даних користувача в конвеєр реєстрації. У разі помилок, розумно розміщений елемент відображення внизу форми забезпечує інформативний зворотний зв’язок.

По суті, цей наратив коду показує складний збір компонентів, хуків і дій, відтворюючи реєстрацію користувачів у мобільному додатку React Native. Повна інтеграція Redux для управління станом і React Navigation для навігації демонструє симфонію технологічної майстерності. Форма з продумано розробленими полями та навігаційними підказками є свідченням продуманого дизайну взаємодії з користувачем у екосистемі мобільної реєстрації.

Наступна ступінь - сторінка вибору інтересів (рис 3.11) У цьому коді TypeScript React компонент `Interests` ретельно створено для мобільного додатку в рамках експансивного середовища React Native framework. Цей компонент бере на себе ключову роль в організації складного танцю вибору інтересів користувача, пропонуючи нюанси та захоплюючий досвід користувача.

У міру розгортання оповіді оператори імпорту створюють основу, започатковуючи набір компонентів і хуків, отриманих із різноманітних бібліотек. Компоненти інтерфейсу користувача з `react-native` і `react-native-paper` займають свої позиції поряд із хуками, що виникають із спеціального файлу `useState`, діями, що виходять із спеціального файлу `store`, і визначенням типу для `Interest`.

В основі цієї мобільної симфонії лежить визначення функціонального компонента «Інтереси» — самодостатньої сутності, яка, на відміну від багатьох своїх аналогів, не покладається на жодні вхідні реквізити. Це є свідченням інкапсуляції та модульності в парадигмі React Native.

Складний танець продовжується стратегічним розгортанням гаків. Хук `useAppSelector` майстерно надає доступ до скарбів, що зберігаються в стані

```
return (
  <SafeAreaView style={styles.container}>
    <Text style={styles.title}> Interests </Text>

    <View style={styles.interests}>
      {showedInterests.map((interest: Interest) => (
        <TouchableOpacity
          key={interest.id}
          style={styles.interest}
          onPress={() => { onSelect(interest)}}
        >
          <Text style={styles.interestText}> {interest.name} </Text>
        </TouchableOpacity>
      ))}
    </View>

    <View style={{ flex: 1 }} />

    <Text style={styles.title}> Selected Interests </Text>
    <View style={styles.interests}>
      {selectedInterests.map((interest: Interest) => (
        <TouchableOpacity
          key={interest.id}
          style={styles.interest}
          onPress={() => { onRemove(interest)}}
        >
          <Text style={styles.interestText}> {interest.name} </Text>
        </TouchableOpacity>
      ))}
    </View>

    <View style={{ flex: 1 }} />

    <Button
      mode="contained"
      onPress={handleContinue}
      style={styles.button}
      disabled={!isFilled}
    >
      Continue
    </Button>

    <View style={styles.progress}>
      <ProgressBar progress={progress} color={'blue'} />
    </View>
    <Text style={styles.buttonText}> To continue, you need to select at least 3 interests. </Text>
  </SafeAreaView>
```

Рисунок 3.11 - Сторінка вибору інтересів

Redux, а саме до `interests` і `selectedInterests`. Між тим, гачок useAppDispatch займає центральне місце, володіючи владою для диспетчеризації дій і оркестрування переходів станів, які вдихають життя в програму.

Примітним видовищем у цій композиції є мемоізоване значення, `showedInterests`, ретельно створене за допомогою хука `useMemo`. Цей підібраний список відображає інтереси, які очікують на увагу користувача,

стратегічно поєднуючи випадкові інтереси та їхні підінтереси, які ще потрібно вибрати.

Розповідь досягає своєї кульмінації з відкриттям обробників подій, де функція `handleContinue` наразі чекає свого моменту в центрі уваги, готова бути викликана до дії, коли користувач натискає кнопку «Продовжити» резонує в мобільному ландшафті. Навпаки, функція `onSelect` бере на себе активну роль, витончено відправляючи дію `addInterest` з вибраним інтересом як емісаром. Тим часом функція `onRemove` хореографує видалення інтересу, відправляючи дію `removeInterest` з ідентифікатором видаленого інтересу як месенджером.

Розв’язка проявляється в методі візуалізації, де оживає ретельно створений елемент JSX. Розгортається візуальна картина, що містить гобелен інтересів і вибраних інтересів, кожен з яких проявляється у вигляді кнопки, що чекає спритного дотику користувача. З кожним натисканням інтереси плавно переходять між сферами, від пулу потенційних виборів до сфери вибраних інтересів. Кнопка «Продовжити», подібна до воротаря, залишається неактивною, доки принаймні 3 інтереси не прикрасять вибрану когорту, забезпечуючи уважний прогрес. Сцена додатково прикрашена індикатором прогресу, який візуально розповідає про подорож користувача до бажаної мети вибору 3 інтересів.

Таким чином, цей код не є просто утилітарним сценарієм; це розповідь, сплетена з компонентів, гачків і дій — історія, де Redux оркеструє симфонію управління державою, а React Navigation веде користувачів по розділах, які цікавлять. У цій розповіді користувачі беруть участь у тактильному танці вибору та видалення інтересів, а програма, постійно пильна, чекає моменту, коли обрані користувачем інтереси прокладуть шлях до грандіозного відкриття — кнопки «Продовжити», яка спонукає до подорожі вперед.

Тепер поговоримо про стилювання компонентів а саме про `StyleSheet.create`. За його невибагливим фасадом ховається складний механізм, який впливає на візуальне представлення вашої мобільної програми. У цьому великому дослідженні ми розгадуємо тонкощі `StyleSheet.create`, відшаровуючи шари, щоб розкрити його внутрішню роботу та висвітлити його значення в екосистемі React Native.

Перш ніж заглиблюватися в глибини `StyleSheet.create`, важливо зрозуміти основоположну роль стилів у React Native. Стилі — це не просто косметичні додатки; вони є архітекторами візуального ландшафту, диктуючи, як компоненти мають виглядати, вирівнювати та взаємодіяти. На відміну від традиційної веб-розробки, де стилі можуть застосовуватися за допомогою простого CSS, React Native представляє спеціалізований підхід до стилів, свідомо розроблений для задоволення особливостей мобільної розробки.

Мобільні програми – це інша порода, яка працює в межах різних розмірів екрана, роздільної здатності та можливостей пристрою. Таким чином, парадигма стилю в React Native має бути адаптивною, чуйною та продуктивною. Саме тут `StyleSheet.create` займає центральне місце, пропонуючи ефективне рішення проблем, пов’язаних із мобільним стилем.

При стилізації компонентів React Native виникає спокуса вдатися до вбудованих стилів, що нагадує практики веб-розробки. Однак такий підхід може призвести до вузьких місць продуктивності. Введіть `StyleSheet.create`, утиліту, призначену для вирішення цієї проблеми.

`StyleSheet.create` працює, перетворюючи ваші стилі в більш ефективний, оптимізований формат. Замість того, щоб використовувати звичайні об’єкти JavaScript для визначення стилів, ця утиліта перетворює їх на числовий ідентифікатор. Потім цей ідентифікатор використовується як посилання для отримання стилів, коли це необхідно. Переходячи до цього числового представлення, React Native отримує можливість виконувати оптимізацію, що призводить до покращення продуктивності під час операцій стилю.

Під капотом `StyleSheet.create` використовує розумну техніку, що включає числові ідентифікатори. Кожному визначенню стилю присвоюється унікальний ідентифікатор під час процесу створення. Цей ідентифікатор служить ключем пошуку, що дозволяє React Native швидко отримувати відповідні стилі під час відтворення компонентів. Ця чисельна оптимізація кардинально змінює правила гри, особливо в сценаріях, де в численних компонентах застосовується безліч стилів.

Розуміння життєвого циклу стилів у `StyleSheet.create` має ключове значення для розуміння його ефективності. Коли ви визначаєте стилі за допомогою цієї утиліти, вони піддаються одноразовій трансформації під час ініціалізації програми. Стилі обробляються, їм призначаються унікальні цифрові ідентифікатори та кешуються для подальшого використання. Як наслідок, витрати на виконання стилю значно зменшуються, що сприяє більш плавній та чутливій взаємодії з користувачем.

Незважаючи на те, що основна увага `StyleSheet.create` зосереджена на продуктивності, вона також приносить другорядну перевагу — покращену зручність обслуговування та читабельність. Централізуючи визначення стилів, ця утиліта сприяє більш організованій кодовій базі. Він заохочує розробників інкапсулювати стилі в межах виділеного об’єкта, сприяючи структурованому підходу до стилів, який узгоджується з найкращими практиками React Native.

Можна поставити під сумнів придатність `StyleSheet.create` для динамічних компонентів, які потребують адаптації стилів відповідно до умов виконання. Не бійся; React Native пропонує безпроблемне рішення. Хоча `StyleSheet.create` чудово оптимізує статичні стилі, сценарії динамічного стилю можуть використовувати метод `StyleSheet.flatten`. Цей метод дозволяє об’єднати динамічні стилі з попередньо оптимізованими статичними стилями, забезпечуючи гармонійне співіснування.

У грандіозній симфонії стилю React Native `StyleSheet.create` постає як віртуоз, організовуючи заплутаний танець між оптимізацією продуктивності та

можливістю обслуговування кодової бази. Перетворюючи стилі на цифрові ідентифікатори, ця утиліта забезпечує підвищення ефективності, яке резонує в усій програмі, особливо в сценаріях із великою кількістю стилів. Числові ідентифікатори не тільки підвищують продуктивність, але й сприяють більш чистій і читабельній кодовій базі.

Коли ви починаєте свою подорож React Native, використання потужності `StyleSheet.create` схоже на відкриття таємної кімнати оптимізації. Скористайтеся цією утилітою, оцініть її роль у ширшій історії стилю та спостерігайте гармонійну взаємодію продуктивності та елегантності коду у вашому мобільному додатку.

У сфері розробки React Native, де управління станом є критично важливим аспектом, також важливе правильне зберігання даних, а саме - Redux Toolkit (RTK) стає потужним союзником. RTK надає спрощений і впевнений набір інструментів для керування станом вашої програми. У цьому розширеному дослідженні ми розгадаємо тонкощі Redux Toolkit і заглибимося в його бездоганну комунікацію з React Native.

За своєю суттю Redux Toolkit — це впевнений набір утиліт, призначених для спрощення та оптимізації робочого процесу керування станом Redux. Він містить кілька потужних інструментів, у тому числі функцію `createSlice` для стислого визначення редукторів і дій, функцію `configureStore` для створення сховища Redux із вбудованими передовими методами та `createAsyncThunk` для легкого керування асинхронними діями.

В екосистемі React Native, де ефективність компонентів має першочергове значення, `createSlice` виділяється як чемпіон у скороченні стандартної форми, пов’язаної з визначенням редукторів і дій. Він підтримує модульну та читабельну структуру коду, інкапсулюючи логіку редуктора та творців дій в одному фрагменті. Це не тільки покращує зручність обслуговування коду, але й ідеально

узгоджується з філософією React Native щодо створення масштабованого та читабельного коду.

Функція configureStore в RTK виводить процес створення магазину Redux на новий рівень. У контексті React Native, де продуктивність є ключовою, `configureStore` за замовчуванням містить важливе проміжне програмне забезпечення та параметри конфігурації. Це включає проміжне програмне забезпечення «redux-thunk» для обробки асинхронних дій, що забезпечує плавну інтеграцію з компонентами React Native, які часто вимагають асинхронного отримання даних.

Асинхронні дії, поширений сценарій у мобільних програмах, елегантно обробляються `createAsyncThunk`. Ця утиліта спрощує створення асинхронних дій, таких як запити API, інкапсулюючи дії запиту, успіху та помилки в одній функції. У середовищі React Native, де вибірка даних невід'ємна, `createAsyncThunk` сприяє чистому та прямому підходу до керування асинхронною поведінкою.

Поєднання Redux Toolkit і React Native схоже на симбіотичні відносини, де кожен доповнює сильні сторони іншого. Компоненти React Native, керовані необхідністю ефективного управління станом, знаходять розраду в спрощеному синтаксисі RTK. Дії та редуктори, складно визначені за допомогою `createSlice` та організовані в добре налаштованому сховищі, безперебійно взаємодіють із компонентами React Native, створюючи гармонійний потік даних.

У додатку React Native на основі RTK потік даних організований ефективно. Компоненти React Native відправляють дії, створені `createSlice`, запускаючи редуктори для оновлення глобального стану. Оновлений стан, яким ефективно керує сховище Redux, налаштоване за допомогою `configureStore`, плавно повертається назад у компоненти React Native. Цей спрощений потік даних гарантує, що мобільні програми, створені за допомогою React Native і Redux Toolkit, демонструють оптимальну продуктивність і швидкість реагування.

Одним із аспектів Redux Toolkit у розробці React Native, який часто недооцінюють, є його акцент на можливості тестування та налагодження. Модульна природа зрізів і впевнена структура, надана RTK, полегшують просте модульне тестування та налагодження. Розробники React Native можуть впевнено забезпечити стабільність і надійність своїх програм за допомогою чітко визначених тестів і спрощених процесів налагодження.

У грандіозному гобелені розробки React Native Redux Toolkit постає як ткацька гобеленів, поєднуючи складності управління державою в цілісну та ефективну розповідь. Завдяки таким утилітам, як `createSlice`, `configureStore` і `createAsyncThunk`, RTK привносить нотку простоти та елегантності в екосистему керування станом React Native. Мобільні програми процвітають у цьому симбіотичному зв'язку, де ефективність Redux Toolkit бездоганно інтегрується з плавністю компонентів React Native.

Перейдемо до бекендної частини. Перш за все ініціалізуємо коннект до БД (рис. 3.12). Цей код JavaScript керує встановленням з'єднання з базою даних PostgreSQL у програмі Node.js, використовуючи бібліотеку pg. Розбивка його функціональності виглядає наступним чином:

Початковий оператор імпорту додає об'єкт Pool з бібліотеки pg. Цей об'єкт служить ключовим інструментом для створення пулу підключень до бази даних PostgreSQL, оптимізуючі продуктивність шляхом повторного використання існуючих підключень.

Наступний розділ передбачає створення нового об'єкта Pool, оснащеного необхідними параметрами конфігурації для бази даних PostgreSQL. Конфігурація включає такі важливі деталі, як ім'я користувача, хост, ім'я бази даних, пароль і номер порту.

Рисунок 3.12 - Ініціалізація коннекту до БД

Після створення екземпляра пулу код переходить до підключення до бази даних за допомогою методу підключення об’єкта пулу. Цей метод використовує функцію зворотного виклику, яка викликається після успішного підключення або у випадку помилки. У разі помилки на консоль записується відповідне повідомлення. І навпаки, якщо з’єднання встановлено успішно, у журналі реєструється повідомлення про підтвердження.

Важливо, що об’єкт пулу експортується з модуля, що дозволяє використовувати його в інших сегментах програми. Цей експорт дозволяє різним частинам програми спільно використовувати та використовувати той самий пул підключень для взаємодії з базою даних.

Наразі курсор знаходиться на рядку 18, позначаючи точку експорту об’єкта пулу. Отже, після цього рядка інші модулі в межах програми можуть імпортувати об’єкт пулу для взаємодії з базою даних. Приклад оператора імпорту

проілюстровано таким чином: `const pool = require('./index.js');`— припускаючи, що файл має назву index.js.

По суті, цей фрагмент коду інкапсулює основні кроки для налаштування та встановлення з'єднання з базою даних PostgreSQL, полегшуючи подальшу взаємодію в програмі Node.js. Експортований об'єкт пулу стає основним ресурсом для інших компонентів, забезпечуючи уніфікований та ефективний механізм взаємодії бази даних у всій програмі.

Далі - створимо базовий express сервер (рис. 3.13). Цей код JavaScript налаштовує простий веб-сервер за допомогою фреймворку Express.js. Розбивка його функціональних можливостей описана нижче.

Рисунок 3.13 - Базовий express сервер

Початковий оператор імпорту містить експрес-модуль, мінімальну та гнучку структуру веб-додатків Node.js, яка відома своїми надійними функціями, що обслуговують як веб-, так і мобільні програми.

Наступний розділ містить виклик функції express(), створюючи нову програму Express, яка зберігається у змінній під назвою "app". Визначається змінна порту та встановлюється значення 3000, що представляє номер порту, на якому сервер активно прослуховуватиме вхідні запити.

Код продовжує визначення обробника маршруту для запитів HTTP GET за допомогою функції app.get(). Цей обробник асоціюється з кореневою URL-адресою ("/"). Коли до цієї URL-адреси надходить запит GET, сервер відповідає, надсилаючи рядок "Hello World!" назад до клієнта.

Останнім кроком є запуск сервера за допомогою функції app.listen(), вказуючи порт (3000), на якому сервер буде слухати вхідні запити. Після готовності прийняти запити на консоль записується повідомлення із зазначенням URL-адреси, на якій працює сервер (http://localhost:3000).

По суті, цей стислий сегмент коду створює базовий сервер Express.js, який, отримавши запит GET до кореневої URL-адреси, відповідає простим привітанням «Hello World!» Сервер, у свою чергу, прослуховує вхідні запити на порт 3000, ініціюючи його функціональність після виконання.

Далі, додамо декілька запитів для створення, отримання інтересів (рис 3.14)

Цей код JavaScript створює базовий веб-сервер за допомогою фреймворку Express.js і встановлює з’єднання з базою даних PostgreSQL за допомогою бібліотеки pg-promise. Ось детальна розбивка його функцій:

Заяви імпорту: код починається з імпорту експрес-модуля, визнаного своїми мінімалістичними та адаптованими функціями для веб-додатків Node.js.

Створіть програму Express: Викликається функція express(), яка генерує нову програму Express, що зберігається в змінній "app".

Проміжне програмне забезпечення: рядок «app.use(express.json())» представляє проміжне програмне забезпечення для аналізу тіл JSON із вхідних запитів, покращуючи можливості сервера обробляти дані JSON.

```
nature_land > take_control > server.js > ...
const express = require('express');

const app = express();
app.use(express.json());
const port = 3000;
const pgport = 3000;

const db = pgp({
  host: 'localhost',
  port: pgport,
  database: 'your_database',
  user: 'your_username',
  password: 'your_password'
});

app.get('/interests', async (req, res) => {
  try {
    const interests = await db.any('SELECT * FROM interests');
    res.json(interests);
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: err.message || err });
  }
});

app.post('/interests', async (req, res) => {
  try {
    const { id, name, priority, parent_interest_id, subinterests } = req.body;
    const interest = await db.one(
      'INSERT INTO interests (id, name, priority, parent_interest_id, subinterests) VALUES ($1, $2, $3, $4, $5) RETURNING *',
      [id, name, priority, parent_interest_id, JSON.stringify(subinterests)]
    );
    res.status(201).json(interest);
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: err.message || err });
  }
});

app.listen(port, () => {
  console.log(`Server is running at http://localhost:${port}`);
});
```

Рисунок 3.14 - Запити інтересів

Визначення порту: Змінна «порт» установлюється та має значення 3000, що позначає номер порту, на якому сервер активно прослуховуватиме вхідні запити.

Підключення до бази даних: Викликається функція «pgp», налаштована за допомогою об’єкта для полегшення підключення до бази даних PostgreSQL. Отримана змінна "db" служить каналом для підключення до бази даних.

Визначення маршруту: Функція "app.get('/interests', async (req, res) => {...})" служить обробником асинхронного маршруту для запитів HTTP GET, спрямованих на URL-адресу /interests. Він виконує запит до таблиці «інтереси» в базі даних і передає результати назад клієнту. Функція "app.post('/interests', async (req, res) => {...})" є ще одним асинхронним обробником маршруту, цього разу призначеним для запитів HTTP POST до URL-адреси /interests. Він керує

вставленням нового інтересу в таблицю «інтереси» в базі даних і згодом надсилає новостворений інтерес назад клієнту.

По суті, цей комплексний сегмент коду не лише створює фундаментальний сервер Express.js, але й інтегрує функціональність для підключення до бази даних PostgreSQL через бібліотеку pg-promise. Включення проміжного ПЗ покращує здатність сервера обробляти дані JSON. Визначені маршрути інкапсулюють асинхронні операції, такі як запити та вставлення даних у базу даних, що дозволяє серверу належним чином відповідати на вхідні запити за вказаними маршрутами. Це налаштування створює основу для обробки взаємодії з базою даних у контексті веб-сервера.

Створимо базову нейронну мережу для тренування передбачення пріоритизації (рис 3.15)

```
const brain = require('brain.js');

// Prepare your data
const data = [
  { input: { Books: 1 }, output: { priority: 0.1 } },
  { input: { Novels: 1 }, output: { priority: 0.2 } },
  { input: { 'James Rollins': 1 }, output: { priority: 0.3 } },
  { input: { Iceberg: 1 }, output: { priority: 0.4 } },
  // Add more data here
];

// Create a new neural network
const net = new brain.NeuralNetwork();

// Train the network
net.train(data);

// Test the network
const output = net.run({ 'James Rollins': 1 });  // { priority: 0.3 }
```

Рисунок 3.15 - Нейронна мережу для тренування пріоритизації

Під кінець хочу розповісти про те, як буде зберігатись код. GitHub надійний та безкоштовний варіант, який бездоганно для цього підійде. GitHub виступає як культовий колос, платформа, де рядки коду перетворюються на шедеври спільної роботи. GitHub, народжений на основі відкритого вихідного коду, вийшов за межі свого коріння й став епіцентром співпраці в коді, контролю версій і розробки, керованої спільнотою. Давайте вирушимо в подорож просторами GitHub, досліджуючи його ключову роль у формуванні спільного наративу сучасної інженерії програмного забезпечення.

У центрі GitHub лежить репозиторій, цифрова фортеця, де зберігається, розвивається та співпрацює код. Незалежно від того, чи розміщується персональний проект, кодова база компанії чи внесок із відкритим кодом, репозиторій GitHub — це більше, ніж сховище; це жива сутність, динамічне полотно, де розробники об’єднуються, щоб створювати, вдосконалювати та інновувати.

Механізм запиту на підключення GitHub є квінтесенцією спільної розробки. Це перетворює акт пропозиції змін у ретельний танець, де учасники вносять зміни, ініціюють обговорення та йдуть до досконалості. Запити на витягування — це не просто надсилання коду; вони є епіцентром співпраці, де стикаються ідеї та виникає синергія.

Під час припливів і відпливів розробки програмного забезпечення проблеми та помилки неминучі. Відстеження проблем GitHub виходить за межі традиційного поняття звітування про проблеми; він стає форумом, де розробники спільно обговорюють, діагностують і вирішують проблеми. Проблеми не є перешкодами; вони є сходинками до вдосконаленого коду та покращеного досвіду користувача.

У заплутаному танці контролю версій механізм розгалуження GitHub забезпечує хореографію. Гілки – це не просто розбіжні часові шкали; вони

являють собою наративні нитки, що дозволяють розробникам досліджувати ідеї, безстрашно експериментувати та повертатися до спільної симфонії коду. Можливості розгалуження GitHub дають змогу розробникам створювати складні сюжетні лінії в кодовій базі.

GitHub — це не просто сховище коду; це процвітаюча кузня спільноти. Розгалуження репозиторію не є розходженням; це жест внеску. Зірки – це не просто закладки; це схвалення, підтвердження від спільноти. Соціальна структура GitHub виходить за рамки рядків коду, сприяючи почуттю причетності та співпраці між розробниками по всьому світу.

Дії та робочі процеси GitHub додають рівень автоматизації, перетворюючи інтеграцію коду та розгортання в добре організовану симфонію. Постійна інтеграція не є процесом; це ритм, який пронизує життєвий цикл розвитку. Дії та робочі процеси GitHub дають змогу розробникам автоматизувати повсякденні завдання, забезпечуючи якість коду та ефективність доставки.

У спільному гобелені GitHub документація постає як стовп мудрості. Файли README – це не просто ознайомлення; вони є запрошеннями, які заманюють учасників у світ проекту. Добре оформлена документація – це не розкіш; це прагнення до прозорості, що веде розробників через тонкощі коду та надає їм можливість робити суттєвий внесок.

GitHub — це більше, ніж система контролю версій; це культурний феномен, який переосмислив динаміку спільної розробки програмного забезпечення. Це свідчення принципу відкритого коду, де колективна мудрість розробників об’єднується для створення програмного забезпечення, яке перевершує індивідуальну майстерність. Під час навігації сховищами, запитами на отримання та проблемами пам’ятайте, що GitHub — це не просто платформа; це екосистема спільної роботи, де рядки коду перетворюються на інноваційні рішення, керовані спільною пристрастю світової спільноти розробників.

ВИСНОВКИ

Результатом цієї роботи є працюючий застосунок для пошука нових людей на основі інтересів та пріоритезації цих інтересів. Виконання цієї роботи стало можливим завдяки дослідженню проведеному у сферах штучного інтелекту та сучасних технологій.

Цю роботу може використовувати будь-хто кому потрібні нові знайомства у своїй галузі. Наприклад програмісти, чи рекрутери, що сильно спростовує пошук нових кандидатів на роботу. Завдяки тому, що це мультіплатформений застосунок, маємо можливість швидко масштабуватись та шукати інвесторів.

Під час розробки данного застосунку я зіткнувся із великою кількістю помилок, котрі виправляв під час роботи із продуктом. Хочу відмітити, що частина проблем була вирішена завдяки нейронним мережам, котрі вміють аналізувати код. Але, я досі бачу що можна і треба допрацювати у найближчий час. Перш за все, вдосконалити алгоритм пошуку людей, бо наразі це дуже просте зрівняння “у лоб”, та тренування нейронки відбувається лише для встановлення пріоритизації.

Також, як результат роботи, можна відмітити отримані навики працювання з нейронними мережами та React Native бібліотекою, яка надає можливість створювати кросплатформені застосунки. Навіть при тому, що я був знайомий із React , я відкив для себе іншу сторону цієї бібліотеки та поглибив свої знання у цій галузі, а дослідження нейронних мереж та їх використання під час розробки дало мені розуміння того, що за ШШ майбутнє і що це дуже гарний інструмент для розробників.

ПЕРЕЛІК ПОСИЛАНЬ

1. Стаття про базовий алгоритм КНН [Електронний ресурс]. Режим доступу: https://www.analyticsvidhya.com/blog/2018/03/introduction-k-neighbours-algorithm-clustering/
2. Документація по React Native [Електронний ресурс]. Режим доступу: https://reactnative.dev/
3. Документація по React Toolkit [Електронний ресурс]. Режим доступу: https://redux-toolkit.js.org/rtk-query/overview
4. Стаття про базове використання машинлернінгу із BrainJs [Електронний ресурс]. Режим доступу: https://dev.to/gfish94/brainjs-for-beginners-1g77
5. Документація по Typescript [Електронний ресурс]. Режим доступу: https://www.tutorialsteacher.com/typescript/typescript-overview
6. Документація по tensorflow.js [Електронний ресурс]. Режим доступу: https://www.tensorflow.org/js/tutorials
7. Документація по expo [Електронний ресурс]. Режим доступу: https://docs.expo.dev/
8. Документація по Express.js [Електронний ресурс]. Режим доступу: https://expressjs.com/
9. Introduction to React Native Paper [Електронний ресурс]. Режим доступу: https://www.youtube.com/watch?v=Cr5eXyr6CJ4
10. Machine learning in javascript [Електронний ресурс]. Режим доступу: https://www.youtube.com/playlist?list=PLRqwX-V7Uu6YPSwT06y_AEYTqIwbeam3y