ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Методи інтеграції машинного навчання у web-додатки на базі ASP.NET Core з використанням ML.NET»

на здобуття освітнього ступеня магістра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми Інженерія програмного забезпечення

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

_____ Артем СЄРОКУРОВ
(підпис)

Виконав: здобувач вищої освіти група ПДМ-62
Артем СЄРОКУРОВ

Керівник: Людмила СОЛЯНИК
к.х.н.

Рецензент: _____

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ

ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного

забезпечення Ступінь вищої освіти Магістр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри
Інженерії програмного забезпечення

_____Ірина ЗАМРІЙ
«_____» _____2024 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

Сєрокурову Артему Ігоровичу

1. Тема кваліфікаційної роботи: « Методи інтеграції машинного навчання у web-додатки на базі ASP.NET Core з використанням ML.NET»

керівник кваліфікаційної роботи Людмила СОЛЯНИК к.х.н., доцент кафедри ІПЗ,

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «15» 2024 р. №320.

2. Строк подання кваліфікаційної роботи «17» грудня 2024р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, методи інтеграції машинного навчання у web-додатки, вимоги до точності та представлення даних.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Дослідження методів інтеграції машинного навчання у web-додатки

2. Реалізація обробки даних за допомогою інтегрованого методу машинного навчання
3. Аналіз точності та ефективності запропонованого методу.

5. Перелік графічного матеріалу: *презентація*
   1. Мета, об'єкта та предмет дослідження.
   2. Порівняння методів.
   3. Алгоритм SDCA
   4. Блок схема алгоритму SDCA
   5. Математична модель метрик оцінювання порівняння моделей
   6. Практичний результат
   7. Демонстрація результату
   8. Порівняльний аналіз
   9. Висновки.
   10. Публікації та апробація роботи.

6. Дата видачі завдання«16» жовтня 2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	16.10-23.10.2024	
2	Вивчення матеріалів для аналізу технологій сканування місцевості ультразвуковими сенсорами	24.10-30.10.2024	
3	Дослідження методів інтеграції	31.10-04.11.2024	
4	Аналіз особливостей машинного навчання	05.11-09.11.2024	
5	Дослідження технологій візуалізації даних	10.11-16.11.2024	
6	Застосування методу ехолокації в моделюванні сканування місцевості	17.11-22.11.2024	
7	Оформлення роботи: вступ, висновки, реферат	23.11-25.11.2024	
8	Розробка демонстраційних матеріалів	26.11-27.11.2024	
9	Попередній захист роботи	27.11-06.12.2024	

Здобувач вищої освіти _____ Артем СЄРОКУРОВ
(підпис)

Керівник
кваліфікаційної роботи _____ Людмила СОЛЯНИК
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра (магістра): 88 стор, 12 рис, 1 табл, 30 джерел.

Мета роботи – підвищення ефективності інтеграції машинного навчання у web-додатки, створені на базі ASP.NET Core, із застосуванням ML.NET.

Об’єкт дослідження – процес інтеграції моделей машинного навчання у web-додатки.

Предмет дослідження – метод інтеграції машинного навчання у web-додатки на базі ASP.NET Core з використанням ML.NET.

У роботі використано різноманітні методи, такі як методи машинного навчання, апарат математичного моделювання, технології об’єктно-орієнтованого програмування та підходи до розробки REST API. Головний акцент зроблено на інтеграції моделей машинного навчання в web-додатки на базі ASP.NET Core із використанням ML.NET.

Проведено аналіз сучасних методів машинного навчання, зокрема алгоритмів регресії, класифікації та кластеризації. Глибоко вивчено особливості роботи ML.NET, його архітектуру та застосовність у задачах інтеграції моделей машинного навчання в web-додатки.

Розроблено та оптимізовано метод інтеграції моделей машинного навчання в web-додатки за допомогою платформи ML.NET. Запропонований метод охоплює етапи збору та підготовки даних, тренування моделей, їх серіалізації та впровадження в web-додаток. Для забезпечення ефективної роботи системи застосовано сучасні підходи до оптимізації моделей, включаючи автоматизований пошук параметрів.

Проведено експерименти для валідації розробленого методу інтеграції. Оцінено продуктивність web-додатка, точність моделей машинного навчання та швидкість їх роботи на реальних даних. Результати дослідження підтверджують

успішність та перспективність використання ML.NET для інтеграції моделей машинного навчання в web-додатки. Розроблений метод продемонстрував очікувані результати, забезпечивши високу точність, продуктивність та простоту впровадження. Запропонований підхід може знайти широке застосування у галузях автоматизації бізнес-процесів, аналізу даних у реальному часі та створення інтелектуальних web-додатків.

КЛЮЧОВІ СЛОВА: МАШИННЕ НАВЧАННЯ, ML.NET, ASP.NET CORE, ІНТЕГРАЦІЯ, WEB-ДОДАТОК, МОДЕЛІ ПРОГНОЗУВАННЯ, ОПТИМІЗАЦІЯ, ТЕСТУВАННЯ, REST API, АРХІТЕКТУРА ДОДАТКУ.

ABSTRACT

The text part of the qualification work for obtaining a bachelor's (master's) degree: 88 pages, 12 figures, 1 tables, 30 sources.

The purpose of the work is to improve the efficiency and simplify the integration of machine learning into web applications developed using ASP.NET Core with the application of ML.NET, as well as to evaluate the effectiveness of such integrations in enhancing the performance and functionality of web services.

The object of the research is the process of integrating machine learning models into web applications.

The subject of the research is the method of integrating machine learning into web applications based on ASP.NET Core using ML.NET.

Various methods are applied in the work, including machine learning methods, mathematical modeling tools, object-oriented programming technologies, and REST API development approaches. The primary focus is on integrating machine learning models into web applications based on ASP.NET Core using ML.NET.

An analysis of modern machine learning methods, particularly regression, classification, and clustering algorithms, has been conducted. The features of ML.NET, its architecture, and its applicability to integrating machine learning models into web applications have been thoroughly studied.

A method for integrating machine learning models into web applications using the ML.NET platform has been developed and optimized. The proposed method covers data collection and preparation, model training, serialization, and deployment into a web application. Modern approaches to model optimization, including automated parameter tuning, have been applied to ensure efficient system performance.

Experiments have been conducted to validate the developed integration method. The performance of the web application, the accuracy of machine learning models, and the speed of their operation on real data have been evaluated.

The research results confirm the success and potential of using ML.NET for integrating machine learning models into web applications. The developed method demonstrated expected results, providing high accuracy, performance, and ease of implementation. The proposed approach has wide applicability in the fields of business process automation, real-time data analysis, and the development of intelligent web applications.

KEYWORDS: MACHINE LEARNING, ML.NET, ASP.NET CORE, INTEGRATION, WEB APPLICATION, PREDICTIVE MODELS, OPTIMIZATION, TESTING, REST API, APPLICATION ARCHITECTURE.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ОС - Операційна система

DOM - Document Object Model

IDE – Integrated Development Environment

SQL - Structured Query Language

ВСТУП

Сучасний розвиток інформаційних технологій значною мірою визначається потребою у створенні інтелектуальних систем, здатних забезпечувати автоматизацію аналізу даних, прогнозування, класифікації та прийняття рішень. В умовах постійного зростання обсягів даних та складності задач, машинне навчання (ML) стало одним із ключових напрямів, який дозволяє використовувати наявні дані для побудови моделей, що адаптуються та самостійно вдосконалюються. Це забезпечує можливість розробки високоефективних рішень у таких галузях, як медицина, фінанси, логістика, маркетинг, промисловість та багато інших.

Водночас веб-застосунки залишаються основою багатьох сучасних інформаційних систем, забезпечуючи інтерактивний доступ користувачів до функціональності, даних та сервісів у глобальній мережі. Завдяки своїй масштабованості, доступності та зручності використання, веб-застосунки займають ключову позицію у реалізації корпоративних рішень, сервісів для кінцевих користувачів та хмарних обчислень. Зокрема, платформа ASP.NET Core, розроблена Microsoft, надає потужні інструменти для створення продуктивних і багатофункціональних веб-застосунків із високим рівнем гнучкості та можливістю легкої інтеграції з іншими технологіями.

На перетині цих двох напрямів — машинного навчання та веб-розробки — виникає нова сфера задач, пов'язана з інтеграцією ML-моделей у веб- застосунки. Така інтеграція відкриває значні перспективи для створення інтелектуальних веб-застосунків, які можуть використовувати предиктивну аналітику, персоналізацію, автоматичну обробку інформації та інші можливості, що базуються на ML. Проте, даний процес супроводжується низкою викликів, зокрема, забезпеченням продуктивності системи, оптимізацією часу відгуку, масштабованістю, а також відповідністю ML-моделей до вимог конкретних бізнес-процесів.

Платформа ML.NET, представлена компанією Microsoft, пропонує інтегрований підхід до створення моделей машинного навчання для .NET-екосистеми, що є важливим кроком у спрощенні та стандартизації процесу розробки. Ця платформа забезпечує можливість використання широкого спектра алгоритмів, роботи з різноманітними джерелами даних, а також реалізації моделей у веб-застосунках, що робить її надзвичайно перспективною для дослідників та розробників.

Таким чином, питання дослідження методів інтеграції машинного навчання у веб-застосунках на базі ASP.NET Core із використанням ML.NET є надзвичайно актуальним як з теоретичної, так і з практичної точки зору. Це дозволяє вирішувати не лише наукові задачі, пов'язані з вдосконаленням методів ML, але й практичні питання, що стосуються ефективної реалізації сучасних програмних рішень. Зокрема, результати такого дослідження можуть знайти застосування у створенні інноваційних продуктів, автоматизації бізнес-процесів, оптимізації операційної діяльності та інших аспектах.

Актуальність теми дослідження обумовлена зростаючою потребою у впровадженні інтелектуальних функцій в сучасні веб-застосунки. Вибір технологій ASP.NET Core та ML.NET для інтеграції обумовлений їхньою сумісністю, продуктивністю та популярністю у розробників.

Мета роботи – підвищення ефективності та спрощення інтеграції машинного навчання у web-додатки, створені на базі ASP.NET Core, із застосуванням ML.NET, а також оцінка ефективності таких інтеграцій для покращення продуктивності та функціональності web-сервісів.

Об'єкт дослідження – процес інтеграції моделей машинного навчання у web-додатки.

Предмет дослідження – метод інтеграції машинного навчання у web-додатки на базі ASP.NET Core з використанням ML.NET.

Для досягнення мети поставлені такі завдання:

1. Проаналізувати можливості та обмеження ML.NET і ASP.NET Core у контексті інтеграції машинного навчання.

2. Розробити архітектуру веб-застосунка з інтегрованим модулем машинного навчання.

3. Реалізувати прототип веб-застосунка з використанням ML.NET для вирішення конкретного завдання.

4. Провести тестування та оптимізацію інтегрованого рішення.

5. Здійснити аналіз ефективності інтеграції та порівняння з альтернативними підходами.

1 ТЕОРЕТИЧНІ ОСНОВИ ТА ОГЛЯД ТЕХНОЛОГІЙ

1.1 Машинне навчання: концепції та алгоритми

Машинне навчання (ML) — це ключова технологія сучасності, що дозволяє створювати моделі, які адаптуються до змін, використовуючи наявні дані. Основна концепція полягає у розробці алгоритмів, які здатні самостійно виявляти закономірності та приймати рішення, зводячи до мінімуму ручне втручання. На відміну від традиційного програмування, де кожен крок обумовлений інструкціями, у машинному навчанні система самостійно навчається на основі даних.

Машинне навчання поділяється на три основні підходи залежно від способу роботи з даними. Перший підхід — навчання з учителем, у якому алгоритм використовує мітки даних, щоб навчитися прогнозувати результати. Це найпоширеніший метод, що знаходить застосування у задачах класифікації, таких як визначення спаму, та регресії, наприклад, прогнозування цін на ринку. Другий підхід — навчання без учителя, спрямований на виявлення прихованих структур у даних без наявності міток. Цей підхід застосовують для задач кластеризації, таких як сегментація клієнтів, або зниження розмірності даних. Третій підхід — навчання з підкріпленням, коли модель навчається шляхом взаємодії зі середовищем, отримуючи винагороду за правильні дії. Цей метод ефективний у розв'язанні задач, пов'язаних із керуванням роботами або стратегічними іграми.

Процес створення моделей машинного навчання зазвичай складається з кількох основних етапів. Спершу здійснюється збір даних, що може включати інтеграцію різних джерел, таких як бази даних чи API. Далі дані готуються до аналізу, що передбачає очищення від шуму, усунення пропусків і нормалізацію значень. На основі завдання обирається алгоритм, що найкраще відповідає проблемі. Після цього модель навчається на тренувальному наборі даних, а її якість оцінюється за допомогою метрик, таких як точність, повнота або F1-міра.

За необхідності проводиться оптимізація, яка включає налаштування параметрів моделі для досягнення кращої продуктивності.

Серед популярних алгоритмів машинного навчання вирізняються регресійні моделі, які широко застосовуються для прогнозування числових значень. Наприклад, лінійна регресія використовується для вивчення залежності між змінними. Алгоритми класифікації, такі як метод опорних векторів або нейронні мережі, дозволяють вирішувати завдання, пов’язані з розпізнаванням об'єктів або категоризацією текстів. Кластеризація допомагає групувати дані за схожістю, що корисно для аналізу поведінки користувачів або виявлення шахрайських транзакцій. Ансамблеві методи, такі як Random Forest чи Gradient Boosting, забезпечують високу точність завдяки поєднанню декількох базових моделей.

Нейронні мережі, особливо глибокі нейронні мережі, є важливим компонентом сучасного машинного навчання. Вони дозволяють вирішувати складні завдання, такі як розпізнавання зображень чи синтез мови. Завдяки своїй багатошаровій структурі нейронні мережі здатні моделювати складні залежності між даними. Водночас їх застосування вимагає значних обчислювальних ресурсів і добре підготовлених даних.

Машинне навчання знаходить широке застосування у різних галузях. У фінансовому секторі моделі використовують для виявлення шахрайства чи автоматизації торгівлі. В охороні здоров'я вони допомагають діагностувати захворювання та прогнозувати ефективність лікування. У маркетингу ML забезпечує персоналізацію реклами, аналіз поведінки клієнтів та покращення користувацького досвіду. У веб-розробці інтеграція ML дозволяє створювати інтелектуальні системи рекомендацій, покращувати пошукові алгоритми та автоматизувати обробку даних.

Водночас впровадження машинного навчання супроводжується низкою викликів. Одним із основних є проблема якості даних, адже помилки, пропуски чи незбалансованість вибірки можуть призвести до низької ефективності моделі. Іншим важливим аспектом є обчислювальна складність, яка зростає з розміром даних та складністю моделей. Крім того, багато алгоритмів, особливо глибокі

нейронні мережі, складно інтерпретувати, що може обмежувати їх використання у критично важливих системах.

У контексті цієї роботи важливе місце займає платформа ML.NET, розроблена Microsoft для інтеграції машинного навчання у додатки на базі .NET. ML.NET надає розробникам інструменти для створення, навчання та впровадження моделей без необхідності переходу на інші середовища. Ця платформа підтримує широкий спектр алгоритмів і дозволяє швидко розробляти прототипи за допомогою AutoML. У поєднанні з ASP.NET Core вона забезпечує гнучкість і продуктивність при розробці веб-додатків.

Таким чином, машинне навчання є потужним інструментом для вирішення різноманітних задач. Воно базується на багатьох концепціях і підходах, які можуть бути ефективно реалізовані за допомогою сучасних технологій, таких як ML.NET, у поєднанні з популярними платформами веб-розробки, включаючи ASP.NET Core.

1.2 Огляд технологій веб-розробки на базі ASP.NET Core

ASP.NET Core — це сучасна, кросплатформна та високопродуктивна платформа для розробки веб-додатків, створена компанією Microsoft. Вона є частиною екосистеми .NET і надає розробникам потужні інструменти для створення динамічних веб-додатків, API та інших типів веб-рішень. Головною перевагою ASP.NET Core є її модульність, масштабованість і продуктивність, що робить її ідеальним вибором для створення сучасних веб-додатків.

ASP.NET Core базується на відкритій архітектурі та підтримує кілька платформ, таких як Windows, Linux і macOS. Це дозволяє розробникам працювати в різних середовищах і забезпечувати високу гнучкість у розробці. Платформа інтегрується з популярними інструментами та технологіями, такими як Visual Studio, Visual Studio Code і GitHub, що полегшує процес розробки та управління кодом.

Однією з ключових особливостей ASP.NET Core є його модульна структура. Усі функції платформи доступні у вигляді окремих бібліотек, які можна додавати

у проект залежно від потреб. Наприклад, якщо веб-додаток не потребує підтримки конкретних функцій, вони можуть бути легко виключені з проекту, що дозволяє зменшити обсяг коду та оптимізувати продуктивність.

Модель хостингу в ASP.NET Core забезпечує універсальність у розробці додатків. Веб-додатки можуть працювати на IIS, Kestrel, або будь-якому іншому HTTP-сервері, який відповідає стандартам. Kestrel, вбудований веб-сервер, є легковаговим і оптимізованим для високопродуктивних додатків, що робить його ідеальним для більшості сучасних рішень.

Серед найважливіших компонентів ASP.NET Core виділяють:

- Middleware (проміжне програмне забезпечення): є ключовим компонентом архітектури веб-додатків, який забезпечує обробку HTTP-запитів та формування відповідей. Воно виступає посередником між сервером і логікою додатку, визначаючи робочий процес для обробки вхідних запитів і відправлення відповідей. Middleware відіграє важливу роль у налаштуванні поведінки додатку, забезпечуючи модульний і гнучкий підхід до управління обробки запитів. Широко використовуються стандартні компоненти middleware, які доступні у фреймворках, або створюються власні для реалізації специфічних вимог. Стандартні компоненти middleware включають наступні функції: маршрутизація запитів, автентифікація, ведення журналів, обробка помилок та стиснення даних. Ці компоненти гнучко налаштовуються та комбінуються для створення ефективного та масштабованого конвеєра обробки запитів. Middleware дозволяє розробникам розширювати функціональність додатків шляхом впровадження спеціальної логіки в цикл запит-відповідь HTTP.

- Dependency Injection (впровадження залежностей), інтегроване у платформу, є ключовим підходом до розробки програмного забезпечення, який сприяє створенню додатків із низьким ступенем зв’язності між компонентами. Ця методологія дозволяє відокремлювати залежності об’єктів від їхньої реалізації, забезпечуючи динамічне надання необхідних об'єктів під час виконання програми.

Використання Dependency Injection сприяє підвищенню гнучкості та масштабованості програмного забезпечення, оскільки залежності можна легко замінювати або модифікувати без внесення значних змін у структуру коду. Це значно полегшує тестування програмних компонентів, дозволяючи ізольовано перевіряти їхню функціональність за допомогою підставних об'єктів (mock objects) або інших підходів до модульного тестування. Низький ступінь зв’язності, досягнутий завдяки впровадженню залежностей, також спрощує підтримку та розвиток програмного забезпечення. Компоненти стають більш автономними, що сприяє кращій організації коду та спрощенню його читання. Крім того, це забезпечує високу повторне використання компонентів, оскільки вони розробляються у вигляді універсальних, самостійних модулів, які можна легко інтегрувати в інші частини програми або використовувати в інших проектах.

- Razor Pages, є потужною технологією, інтегрованою у платформу ASP.NET Core, яка забезпечує простий і ефективний спосіб створення динамічних веб-сторінок. Ця технологія використовує Razor синтаксис, що дозволяє поєднувати HTML і C# код у єдиному файлі сторінки. Завдяки цьому розробники можуть працювати з динамічним контентом без необхідності створення окремих контролерів і ускладненої маршрутизації, як це відбувається в традиційних MVC-підходах. Razor Pages має інтуїтивно зрозумілий синтаксис, який дозволяє зменшити обсяг коду, необхідного для створення функціональних веб-сторінок. Оскільки весь код і логіка сторінки зазвичай знаходяться в одному файлі, це спрощує розробку та забезпечує високу читабельність і підтримуваність коду. Інтеграція з системою маршрутизації також дозволяє автоматично обробляти запити без необхідності прописувати додаткові маршрути або контролери, що ще більше знижує складність розробки.

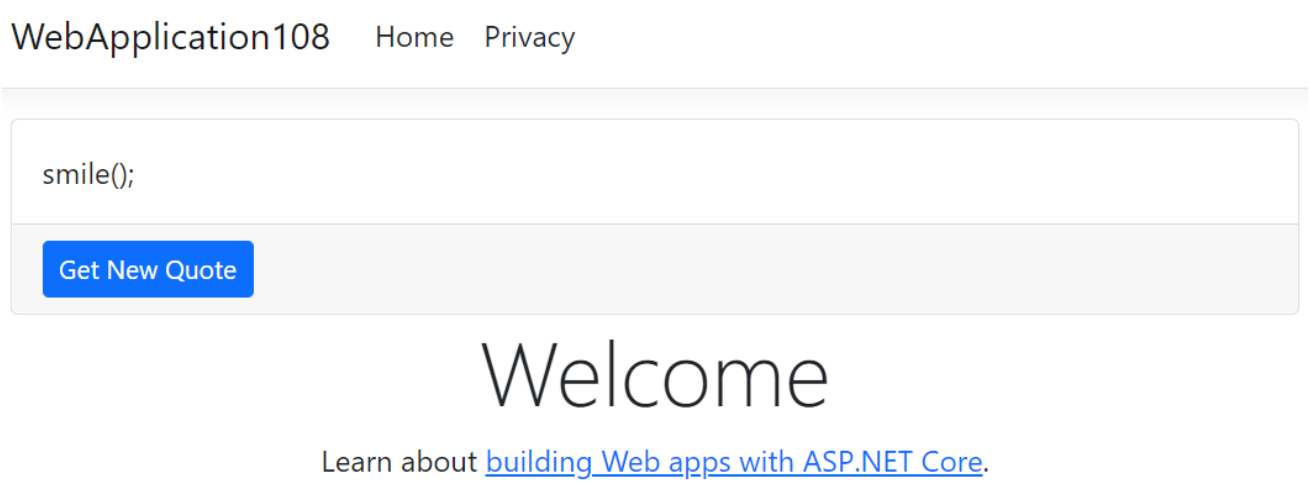


Рис 1.1 Приклад створеної сторінки за допомогою «Razor Pages»

• ASP.NET Core MVC (Model-View-Controller) — є потужним фреймворком для створення веб-додатків, який забезпечує чітке розділення логіки, даних і представлення. Цей підхід дозволяє розробникам організувати програмний код таким чином, що кожна частина додатка відповідає за конкретний аспект функціональності: модель (Model) обробляє дані та логіку, уявлення (View) відповідає за відображення інформації користувачу, а контролери (Controller) здійснюють зв'язок між моделями та уявленнями, обробляючи запити користувачів та визначаючи необхідну бізнес-логіку.

Розділення на ці три компоненти забезпечує високу модульність, що, у свою чергу, сприяє легшій підтримці та розвитку великих додатків. Моделі можуть бути змінені або замінені без необхідності кардинальних змін у представленнях чи контролерах, що спрощує масштабування додатка та робить його більш гнучким до змін. Це також полегшує підтримку коду, оскільки кожен компонент є відокремленим і виконує свою специфічну задачу.

• SignalR це бібліотека, що інтегрується в платформу ASP.NET Core та забезпечує підтримку веб-сокетів для створення додатків у реальному часі. Цей компонент дозволяє розробникам створювати інтерактивні веб-додатки, де обмін даними відбувається миттєво між сервером і клієнтами, без необхідності постійних запитів з боку клієнтів. Завдяки підтримці двостороннього зв'язку,

SignalR дозволяє забезпечити реальний час передачі повідомлень, що є критично важливим для таких застосунків, як чати, онлайн-ігри, спільна робота в документах, фінансові транзакції та багато інших сценаріїв. Однією з основних переваг SignalR є спрощення інтеграції функціональності реального часу в додатки. Розробники можуть легко налаштувати взаємодію між клієнтами та сервером, використовуючи простий API без необхідності вручну управляти низькорівневими аспектами, такими як підключення через веб-сокети або інші транспортні протоколи.

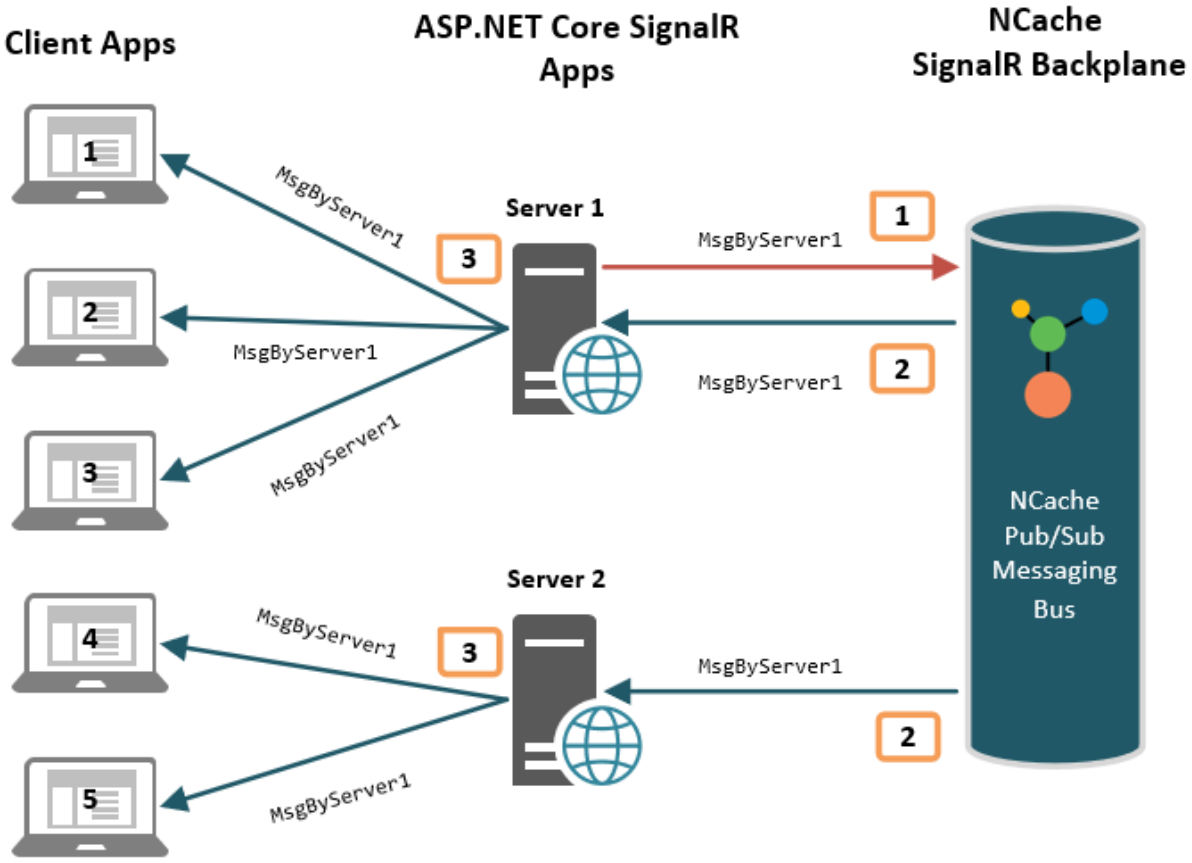


Рис 1.2 Принцип роботи «SignalR»

• Web API, дозволяє створювати RESTful-додатки для взаємодії між сервером і клієнтами або зовнішніми системами. Використання Web API в рамках ASP.NET Core дає змогу розробникам реалізувати сервіси, які надають можливість здійснювати обмін даними за допомогою стандартних HTTP-методів (GET, POST, PUT, DELETE тощо) та використовувати популярні формати, такі як JSON або XML для передачі даних. ASP.NET Core надає всі необхідні інструменти

для створення високопродуктивних API, включаючи вбудовану підтримку маршрутизації, валідації даних, серіалізації та десеріалізації запитів і відповідей, а також механізми аутентифікації та авторизації. Завдяки цьому можна швидко створювати API, що відповідають сучасним вимогам до безпеки та продуктивності, а також легко інтегруються з різноманітними клієнтськими додатками, такими як веб-браузери, мобільні програми чи інші сервери. Розробка RESTful API на основі ASP.NET Core дозволяє створювати масштабовані та зручні для підтримки системи, які можуть взаємодіяти із зовнішніми сервісами, забезпечуючи високу ступінь абстракції та незалежності між компонентами. Крім того, фреймворк ASP.NET Core забезпечує оптимізацію продуктивності, включаючи можливості кешування, обробки великих обсягів даних і асинхронного виконання запитів, що значно підвищує ефективність роботи API навіть за високих навантажень.

Безпека також є одним із ключових аспектів ASP.NET Core. Платформа підтримує сучасні протоколи автентифікації, такі як OAuth2, OpenID Connect і JWT, що дозволяє забезпечувати надійний захист додатків. Крім того, ASP.NET Core включає функції для захисту від типових атак, таких як SQL-ін'єкції та міжсайтовий скриптинг (XSS).

Ще однією важливою характеристикою є підтримка контейнеризації. ASP.NET Core легко інтегрується з Docker, що дозволяє розгортати додатки у контейнерах для підвищення масштабованості та спрощення управління середовищами. Завдяки цій функції розробники можуть створювати додатки, які легко розгортаються в хмарних середовищах, таких як Microsoft Azure або AWS.

З погляду продуктивності, ASP.NET Core є одним із найшвидших веб-фреймворків завдяки оптимізованій архітектурі та вбудованим засобам кешування. Інтеграція з ML.NET є ще однією важливою перевагою ASP.NET Core. Ця платформа дозволяє розробникам легко додавати модулі машинного навчання у веб-додатки, використовуючи знайомі інструменти та бібліотеки .NET. Завдяки високій продуктивності та гнучкості ASP.NET Core є чудовою основою для розробки інтелектуальних веб-додатків, які використовують машинне навчання для автоматизації та персоналізації.

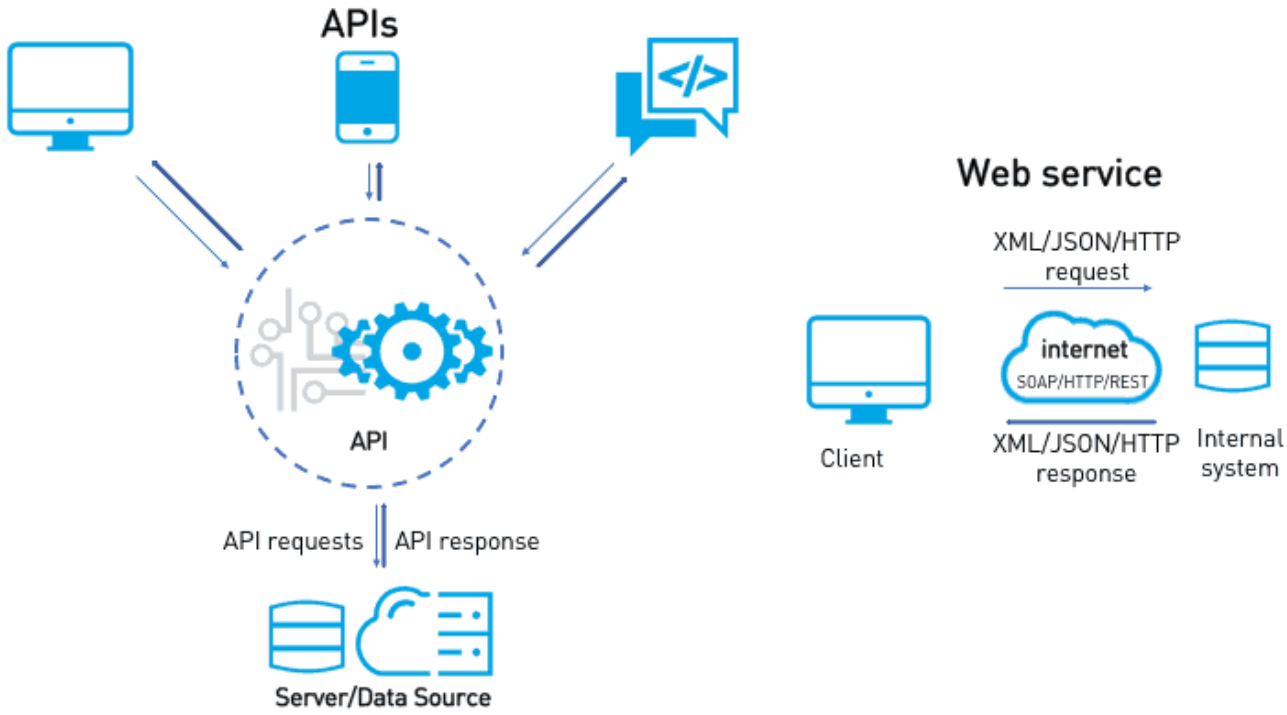


Рис 1.3 Приклад роботи API з веб сервісами

1.3 Платформа ML.NET: можливості та обмеження

ML.NET — це відкритий фреймворк машинного навчання, створений Microsoft для розробників, які працюють у .NET-середовищі. Платформа призначена для створення, навчання та інтеграції моделей машинного навчання у .NET-додатки, включаючи веб-застосунки на базі ASP.NET Core. Її особливістю є те, що вона надає потужні можливості для роботи з ML без потреби у переході на інші мови програмування чи середовища, такі як Python або R.

Однією з головних переваг ML.NET є підтримка широкого спектру алгоритмів, що охоплюють задачі класифікації, регресії, кластеризації, прогнозування часових рядів, обробки тексту й зображень. Це дозволяє розробникам вирішувати різноманітні завдання, такі як рекомендаційні системи, аналіз настроїв, виявлення аномалій і прогнозування попиту.

Однією з ключових особливостей ML.NET є його тісна інтеграція зі стеком технологій .NET. Фреймворк дозволяє використовувати звичні для розробників інструменти, такі як Visual Studio і Visual Studio Code, а також мову

програмування C#. Ця інтеграція знижує поріг входження для тих, хто вже має досвід роботи з .NET, і дозволяє швидко розпочати роботу з машинним навчанням.

AutoML (Automated Machine Learning) є ще одним важливим компонентом ML.NET. Ця функція дозволяє автоматизувати вибір алгоритмів, налаштування гіперпараметрів та оцінку моделей, що значно спрощує процес розробки рішень, особливо для початківців. AutoML забезпечує користувачам простий спосіб створення оптимальних моделей без глибоких знань у галузі машинного навчання.

ML.NET підтримує широкий спектр форматів даних, включаючи текстові файли, бази даних SQL, файли JSON, XML і багато інших. Це забезпечує гнучкість у роботі з даними з різних джерел і спрощує інтеграцію моделей у реальні бізнес-процеси.

Фреймворк також включає функції для обробки та підготовки даних. У ML.NET є вбудовані механізми для очищення даних, нормалізації, роботи з категоріальними змінними (one-hot encoding), а також підтримка багатокрокової обробки даних за допомогою pipeline. Такі можливості роблять ML.NET потужним інструментом для побудови комплексних рішень, де дані потребують значної підготовки перед навчанням моделі.

Серед алгоритмів, доступних у ML.NET, варто виділити такі, як лінійна регресія, дерева рішень, алгоритми градієнтного бустингу, нейронні мережі та методи для кластеризації, такі як K-means. Це дозволяє розробникам вирішувати задачі з різними вимогами до точності, швидкості чи ресурсів.

ML.NET є фреймворком із відкритим кодом, що дозволяє розробникам вивчати внутрішні механізми платформи, модифікувати її під свої потреби та впроваджувати власні рішення. Активна підтримка з боку Microsoft та спільноти розробників забезпечує регулярні оновлення і покращення.

Фреймворк підтримує перенесення попередньо навчених моделей із TensorFlow, ONNX і інших платформ. Це дозволяє інтегрувати складні моделі,

створені за допомогою Python чи інших мов, у .NET-додатки, значно розширюючи можливості платформи.

Однією з важливих характеристик ML.NET є масштабованість. Навчені моделі можна легко розгортати у веб-додатках, настільних програмах або хмарних середовищах, таких як Azure. Завдяки цьому ML.NET ідеально підходить для вирішення бізнес-завдань у реальному часі, таких як персоналізація користувацького досвіду чи обробка великих обсягів даних.

Інструменти для візуалізації, такі як ML.NET CLI і Model Builder, дозволяють автоматизувати рутинні процеси створення моделей і забезпечують зручний інтерфейс для початкового прототипування. Це полегшує розробникам аналіз даних та створення перших робочих версій ML-рішень.

Хоча ML.NET є потужним і зручним інструментом, він має і певні обмеження. Одним із них є вузький спектр алгоритмів у порівнянні з іншими платформами, такими як Scikit-learn чи TensorFlow. Це може створювати труднощі для розробників, які працюють зі специфічними задачами, що вимагають більш складних моделей.

Ще одним недоліком є відносно новий статус платформи, через що деякі функції все ще знаходяться у стадії розвитку. Наприклад, підтримка глибоких нейронних мереж є обмеженою і залежить від інтеграції з іншими платформами, такими як ONNX або TensorFlow.

ML.NET орієнтований на розробників із досвідом роботи у .NET, тому користувачі, які знайомі з іншими мовами програмування або фреймворками, можуть відчувати складнощі при адаптації до його специфіки.

Також платформа має високі вимоги до якості даних. Відсутність детальних вбудованих механізмів для роботи з великими обсягами нерівномірних даних може ускладнити її використання у проєктах зі складною підготовкою даних.

ML.NET добре інтегрується з ASP.NET Core, забезпечуючи зручний спосіб додавання функціональності машинного навчання до веб-додатків. Це дозволяє створювати системи рекомендацій, автоматизовані аналізатори тексту, прогнози

поведінки користувачів та інші інтелектуальні рішення, що покращують взаємодію з користувачами.

Таким чином, ML.NET є потужною платформою, яка дозволяє інтегрувати машинне навчання у .NET-додатки, зокрема веб-застосунки. Її переваги у вигляді інтеграції з екосистемою .NET, підтримки AutoML та можливості перенесення моделей із інших платформ роблять її ідеальним інструментом для розробників. Проте, врахування обмежень ML.NET є важливим аспектом при виборі технологій для складних проєктів.

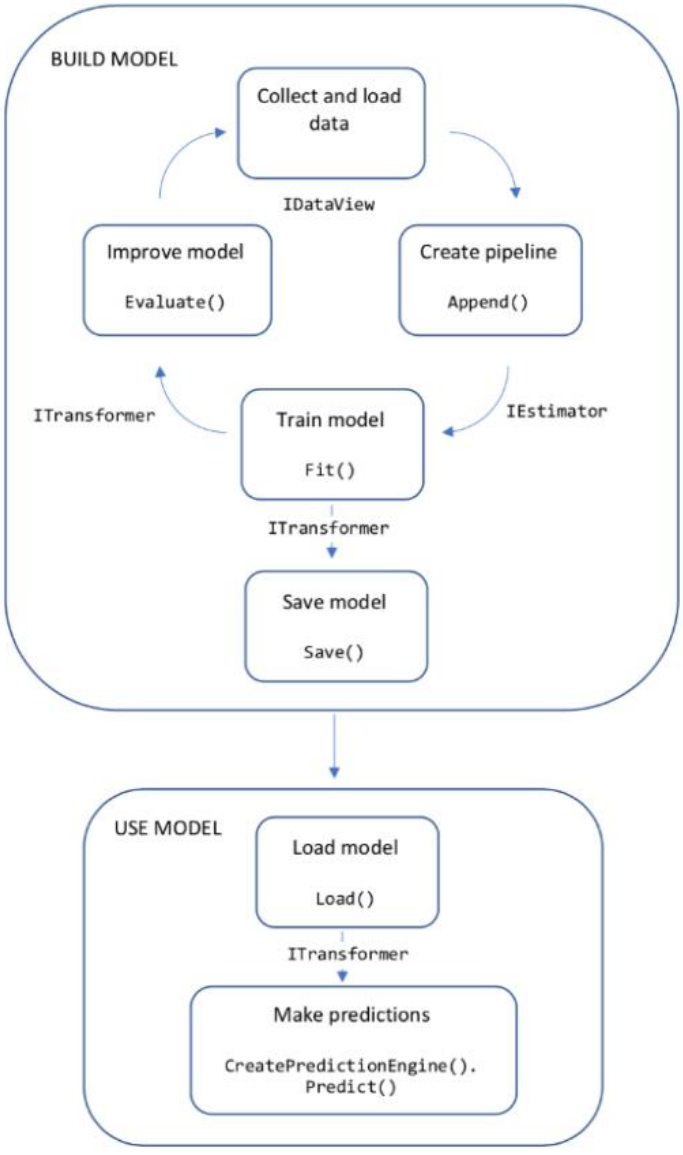


Рис 1.4 Принцип роботи «ML.NET»

2 МЕТОДИ ІНТЕГРАЦІЇ МАШИННОГО НАВЧАННЯ В ВЕБ-ДОДАТКИ

2.1 Існуючі підходи до інтеграції ML у веб-додатки

Інтеграція машинного навчання (ML) у веб-додатки є одним із найактуальніших напрямків сучасної веб-розробки. Вона дозволяє створювати інтелектуальні системи, які адаптуються до потреб користувачів, забезпечують автоматизацію процесів та підвищують якість взаємодії. Проте інтеграція ML у веб-застосунки супроводжується низкою технічних викликів, таких як підготовка даних, навчання моделей, масштабованість рішень і оптимізація продуктивності.

Різноманіття існуючих підходів до інтеграції машинного навчання у веб-застосунки визначається як технологічними особливостями, так і функціональними вимогами до системи. Основні підходи можна поділити на три категорії: використання готових ML-сервісів, інтеграція попередньо навчених моделей і створення моделей безпосередньо у рамках веб-застосунку.

Цей підхід передбачає інтеграцію хмарних сервісів машинного навчання у веб-застосунок. Основними постачальниками таких сервісів є Google Cloud AI, AWS SageMaker, Microsoft Azure Machine Learning та IBM Watson. Такі сервіси пропонують широкий спектр готових ML-рішень, які можна легко інтегрувати через API.

Наприклад, Microsoft Azure Machine Learning дозволяє створювати, навчати й розгортати моделі в хмарному середовищі, а потім використовувати їх через REST API у веб-додатках, побудованих на ASP.NET Core. Аналогічно Google Cloud AI пропонує готові рішення для аналізу зображень, тексту та відео, які можна інтегрувати у веб-застосунок.

Основні переваги цього підходу полягають у швидкості впровадження, масштабованості та відсутності необхідності в управлінні інфраструктурою. Розробники можуть зосередитися на логіці додатка, використовуючи готові

інструменти для роботи з ML. Проте залежність від хмарних сервісів може бути недоліком через витрати на їхнє використання та необхідність забезпечення безпеки даних.

Ще один підхід до інтеграції ML у веб-застосунок — використання попередньо навчених моделей, які розробляються за допомогою популярних платформ, таких як TensorFlow, PyTorch або Scikit-learn. Ці моделі можуть бути навчені локально, збережені у форматі ONNX, TensorFlow SavedModel або PyTorch TorchScript і інтегровані у веб-застосунок.

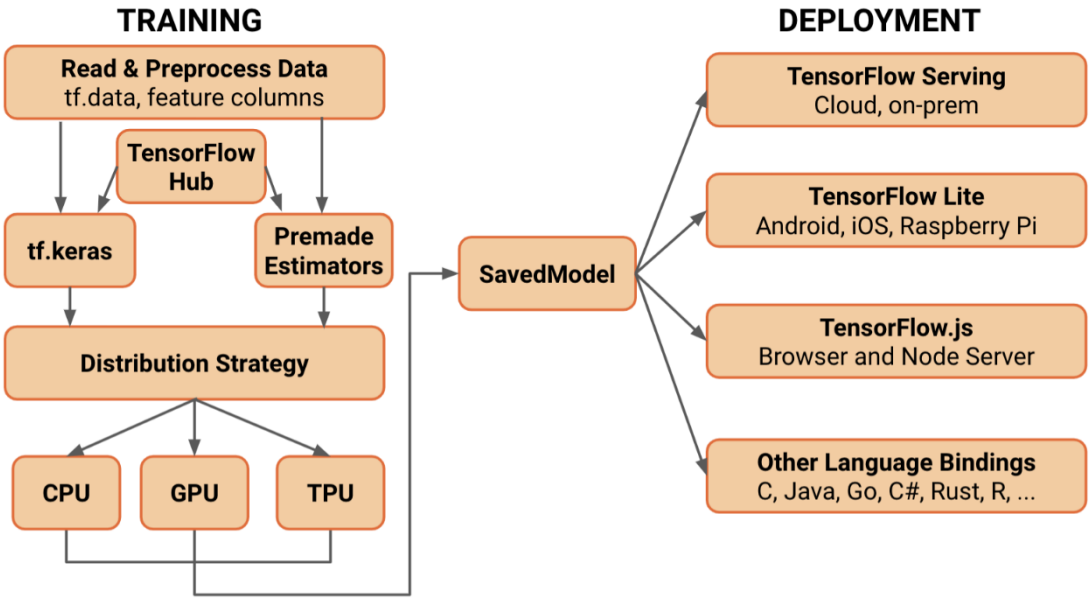


Рис 2.1 Архітектура процесу тренування та розгортання моделі машинного навчання з використанням TensorFlow

ML.NET забезпечує можливість інтеграції таких моделей через підтримку форматів ONNX і TensorFlow.

Цей підхід забезпечує більшу гнучкість порівняно з хмарними сервісами, оскільки розробник має повний контроль над моделями. Водночас складність полягає у потребі у налаштуванні середовища для їхнього запуску, що може вимагати значних технічних знань і ресурсів. Також можливе створення та навчання ML-моделей безпосередньо у веб-приложенні. Для ASP.NET Core

найбільш підходящою платформою є ML.NET, яка надає всі необхідні інструменти для розробки моделей.

Розробка моделей у рамках веб-додатку передбачає побудову ML-пайплайнів для підготовки даних, навчання моделі та її тестування. Наприклад, у додатку на ASP.NET Core можна реалізувати обробку вхідних даних користувачів, створити пайплайн для трансформації даних, обрати алгоритм навчання та інтегрувати модель у процес обробки запитів.

ML.NET також підтримує AutoML, що дозволяє автоматизувати процес пошуку оптимальної моделі та налаштування її параметрів. Це значно знижує поріг входження для розробників, які не мають глибоких знань у машинному навчанні.

За допомогою цього підходу користувач отримує повний контроль над процесом створення моделі та її інтеграцією. Проте він вимагає глибшого розуміння ML і більших витрат часу на розробку. Крім того, для обробки великих обсягів даних або забезпечення високої продуктивності можуть знадобитися додаткові ресурси.

Кожен із зазначених підходів має свої переваги та обмеження. Вибір підходу залежить від вимог до веб-застосунку, доступних ресурсів і технічної експертизи команди розробників.

Для простих завдань або проєктів із обмеженим бюджетом оптимальним вибором може бути використання хмарних сервісів. Це дозволяє швидко інтегрувати готові рішення, зосереджуючись на логіці веб-додатка.

У випадку, коли необхідна більша гнучкість або доступ до специфічних моделей, підхід із використанням попередньо навчених моделей може бути кращим. Він дозволяє інтегрувати складні рішення без потреби у значних обчислювальних ресурсах.

Для проєктів, які потребують створення унікальних рішень або високого рівня кастомізації, доцільно розробляти моделі безпосередньо у рамках веб-застосунку. Це забезпечує повний контроль над кожним етапом, але потребує більшої технічної компетенції та ресурсів.

Інтеграція ML у веб-застосунок супроводжується низкою технічних викликів. Одним із них є забезпечення продуктивності. Обробка запитів користувачів у реальному часі вимагає оптимізації моделей і їхнього виконання. Для цього може використовуватися кешування результатів або розподіл завантаження через мікросервіси.

Іншим викликом є масштабованість. Зростання кількості користувачів веб-додатка може призвести до зниження продуктивності ML-компонентів. Для вирішення цієї проблеми моделі можуть розгортатися у хмарних середовищах, таких як Azure, або використовуватися сервери з GPU для навчання й інференсу.

Забезпечення безпеки даних також є критичним аспектом. Інтеграція ML передбачає обробку чутливої інформації, яка повинна бути захищена від несанкціонованого доступу. Для цього застосовуються методи шифрування, автентифікації та захисту API.

Таким чином, існуючі підходи до інтеграції ML у веб-застосунок дозволяють створювати інтелектуальні рішення різного рівня складності. Вибір підходу залежить від вимог до системи, доступних ресурсів і технічної компетенції команди, що забезпечує гнучкість і адаптивність процесу розробки.

2.2 Архітектурні моделі інтеграції ML у веб-додатку

Інтеграція модулів машинного навчання у веб-додатки є складним і багатогранним завданням, яке потребує детального розгляду архітектурних рішень. Вибір правильної архітектури є визначальним для досягнення ефективності, продуктивності та масштабованості системи. У цьому розділі розглядаються ключові архітектурні моделі, їхні переваги, обмеження та можливості для використання в задачах прогнозування попиту.

Перший підхід до інтеграції машинного навчання у веб-додатки полягає у використанні монолітної архітектури. У такій системі всі компоненти, включно з логікою машинного навчання, працюють як єдиний програмний блок. Цей підхід є простим у реалізації, оскільки не потребує складної оркестрації компонентів.

Модуль машинного навчання інтегрується безпосередньо у серверну частину веб-додатка, що дозволяє швидко обробляти дані та зменшувати затримки. Проте, зростання системи та обсягів даних може призводити до проблем із продуктивністю. Монолітна архітектура ускладнює масштабування, оскільки будь-які зміни в одному компоненті потребують оновлення всього додатка. Цей підхід найкраще підходить для невеликих проектів із передбачуваним навантаженням.

Мікросервісна архітектура пропонує більш гнучкий підхід до інтеграції машинного навчання. У цьому випадку кожен компонент системи, включно з модулем машинного навчання, є незалежним сервісом, який взаємодіє з іншими через API. Така архітектура забезпечує легке масштабування, оскільки кожен сервіс може працювати на окремому сервері чи контейнері. Крім того, це спрощує оновлення та розширення системи, оскільки зміни в одному сервісі не впливають на інші. Водночас, мікросервіси потребують додаткових ресурсів для управління взаємодією між компонентами, що може збільшити затримки під час обробки запитів. Ця архітектура підходить для середніх і великих проектів, де необхідно забезпечити гнучкість і масштабованість.

Серверлесс-архітектура стала популярним підходом для інтеграції машинного навчання у веб-додатки завдяки своїй простоті та ефективності. У цьому випадку обчислювальні ресурси надаються лише тоді, коли вони потрібні, що значно знижує вартість інфраструктури. Моделі машинного навчання можуть бути розгорнуті як функції, які активуються під час отримання запиту. Це дозволяє автоматично масштабувати систему залежно від навантаження, без необхідності вручну керувати ресурсами. Серверлесс-архітектура особливо корисна для задач з нерівномірним або періодичним навантаженням. Однак вона може створювати затримки під час холодного старту функцій, що є критичним для систем реального часу. Крім того, залежність від хмарного провайдера може обмежити гнучкість у виборі інфраструктури.

Гібридна архітектура поєднує елементи монолітного підходу, мікросервісів і серверлесс-архітектури для досягнення балансу між продуктивністю, гнучкістю

та масштабованістю. У цій архітектурі основна логіка додатка може бути реалізована у монолітній структурі, тоді як критично важливі або високонавантажені компоненти, такі як машинне навчання, можуть бути винесені у вигляді мікросервісів або серверлесс-функцій. Це дозволяє зберігати швидкодію для основних операцій і водночас масштабувати ресурсоємні процеси. Гібридна архітектура потребує складнішого управління, оскільки включає кілька різних моделей, але забезпечує високу адаптивність до змінних умов.

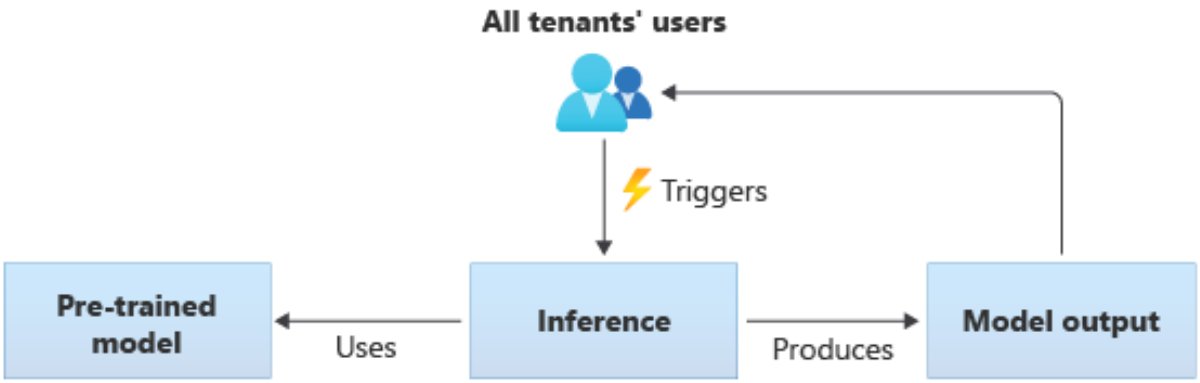


Рис 2.2 Загальна модель

Окрім вибору загальної архітектури, важливу роль відіграє інтеграція самого модуля машинного навчання. Найпростіший спосіб полягає у використанні внутрішньої бібліотеки, наприклад ML.NET, яка дозволяє легко створювати та інтегрувати моделі у веб-додатки, побудовані на ASP.NET Core. У цьому випадку модуль машинного навчання працює як частина серверного коду, що забезпечує швидкий доступ до моделі та зменшує затримки. Іншим підходом є розгортання моделі у вигляді REST API, що дозволяє відокремити її від основного додатка і забезпечити незалежність компонентів. Такий підхід спрощує управління моделями, але додає затримки через взаємодію з мережею.

Ще одним важливим аспектом є використання контейнеризації. Технології, такі як Docker, дозволяють створювати ізольовані середовища для модулів машинного навчання, що забезпечує портативність і легкість у розгортанні. Контейнери дозволяють легко масштабувати компоненти системи, розподіляючи

їх між кількома вузлами у кластері. Контейнеризація добре поєднується з мікросервісною та серверлесс-архітектурою, забезпечуючи гнучкість і простоту управління.

При виборі архітектури важливо враховувати специфіку завдання, обсяг даних і передбачуване навантаження. Для невеликих проектів із постійним навантаженням оптимальним вибором може бути монолітна архітектура з інтеграцією машинного навчання через внутрішні бібліотеки. Для складніших систем, де необхідна висока масштабованість, краще підходять мікросервісна або серверлесс-архітектури.

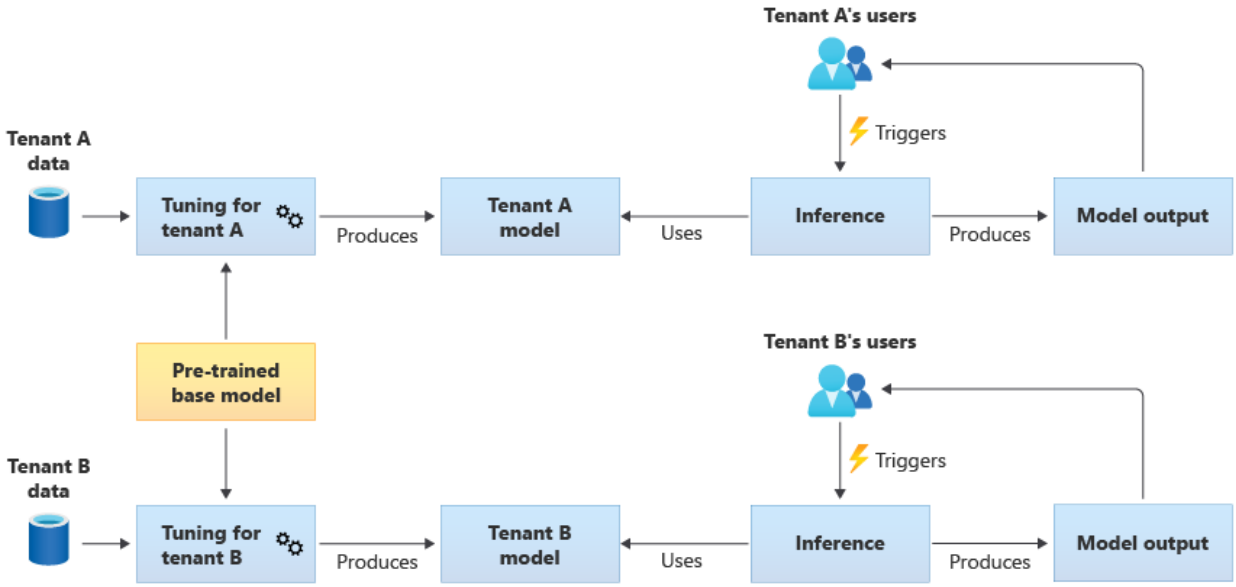


Рис 2.3 Налаштована загальна модель

Таким чином, вибір архітектурної моделі для інтеграції машинного навчання у веб-додатки є критично важливим етапом, що визначає ефективність і продуктивність системи. Успішна реалізація залежить від правильного поєднання технічних рішень, таких як бібліотеки,

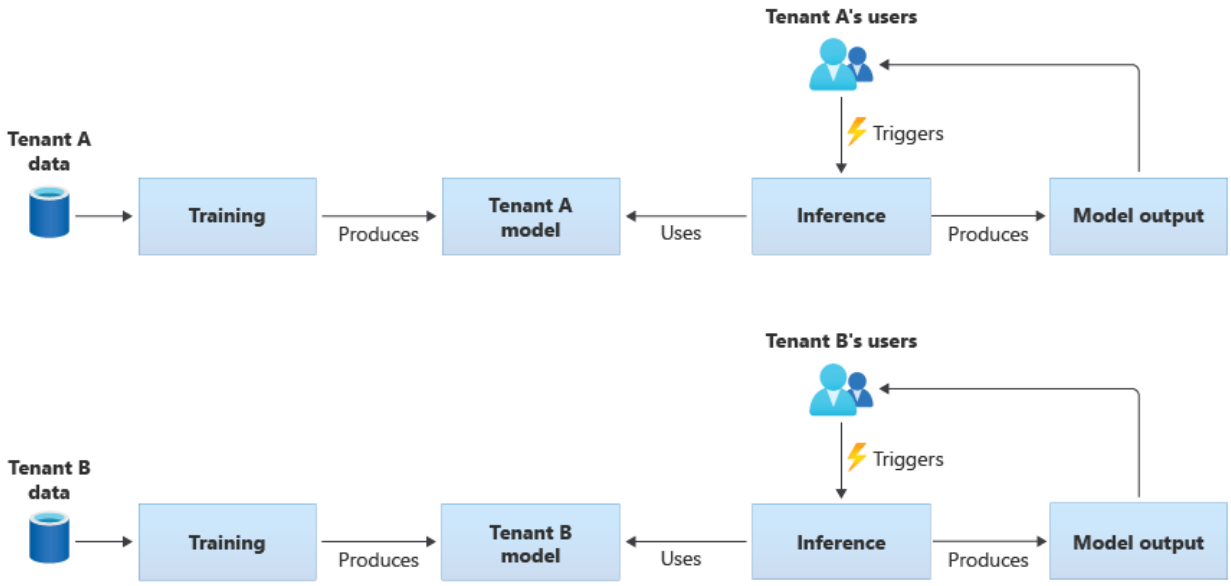


Рис 2.4 Модель для конкректного клієнта

2.3 Вибір та підготовка даних для ML-моделей

Вибір і підготовка даних є фундаментальним етапом у побудові моделей машинного навчання. Успішність моделі значною мірою залежить від якості вхідних даних, правильного їхнього представлення та очищення. Завдання прогнозування попиту вимагає ретельного підходу до вибору джерел даних, визначення релевантних ознак і усунення проблем, пов’язаних із неповними або аномальними даними. Цей процес складається з кількох ключових етапів, включаючи збір, обробку, трансформацію, аналіз і поділ даних на тренувальні та тестові набори.

Першим етапом є збір даних, що включає отримання інформації з різних джерел. Для задач прогнозування попиту основними джерелами є історичні дані про продажі товарів, інформація про характеристики товарів, сезонні фактори, зовнішні чинники, такі як акції та знижки, а також поведінкові дані клієнтів. Історичні дані про продажі є основою для моделювання, оскільки вони відображають минулі тенденції попиту. Інформація про товари, зокрема їхні категорії, ціни, бренди та рейтинги, додає контексту до аналізу. Сезонні фактори, як-от місяці, дні тижня чи святкові дні, є критично важливими, оскільки попит

часто має виражену сезонність. Зовнішні фактори, включаючи маркетингові кампанії та економічні події, також можуть впливати на попит і повинні враховуватися у моделі.

Після збору даних важливим етапом є їхнє очищення. Реальні дані часто містять пропущені значення, аномалії, дублікати або помилки, які можуть негативно вплинути на якість моделі. Пропущені значення можуть бути заповнені за допомогою середнього, медіанного або модального значення, залежно від типу даних. Аномалії, які є значеннями, що значно відрізняються від інших, можуть бути видалені або замінені більш відповідними значеннями, якщо це виправдано контекстом задачі. Дублікати слід видаляти, щоб уникнути спотворення розподілу даних. Крім того, всі дані повинні бути перевірені на коректність і узгодженість, щоб уникнути логічних помилок.

Трансформація даних є наступним етапом підготовки. У цьому процесі числові ознаки можуть бути масштабовані до однакового діапазону для полегшення роботи алгоритмів. Для категорійних ознак використовується кодування, наприклад, one-hot encoding або порядкове кодування, залежно від потреб моделі. Сезонні фактори можуть бути представлені через циклічні ознаки, такі як синуси і косинуси, щоб враховувати їхній повторюваний характер. Також на цьому етапі можуть бути створені нові ознаки, наприклад, ковзне середнє чи стандартне відхилення для аналізу змін попиту в часі.

Аналіз даних допомагає зрозуміти їхню структуру та визначити ключові залежності. Візуалізація даних, наприклад, через гістограми, діаграми розсіювання або теплові карти кореляцій, може виявити потенційні проблеми, такі як мультиколінеарність, або надати інсайти щодо важливих ознак. Статистичний аналіз, включаючи обчислення середнього, медіанного, стандартного відхилення та інших показників, дозволяє оцінити розподіл даних і знайти можливі аномалії.

Поділ даних на тренувальний і тестовий набори є останнім етапом підготовки. Тренувальний набір використовується для навчання моделі, тоді як тестовий набір дозволяє оцінити її продуктивність на нових даних. Рекомендованим співвідношенням для поділу є 80/20 або 70/30, залежно від

розміру набору даних. У деяких випадках може бути доцільним виділення валідаційного набору для налаштування гіперпараметрів моделі. Важливо забезпечити, щоб поділ даних був випадковим, але водночас зберігав розподіл цільової змінної.

Вибір релевантних ознак є критичним кроком у підготовці даних. Невідповідні або нерелевантні ознаки можуть не тільки ускладнити модель, але й знизити її точність. Методи відбору ознак, такі як аналіз кореляцій, моделювання важливості ознак або використання алгоритмів, здатних виявляти значущі ознаки, дозволяють оптимізувати модель. Наприклад, дерева рішень або методи на основі градієнтного бустингу можуть оцінювати важливість кожної ознаки, що допомагає у прийнятті рішень щодо їхнього включення до моделі.

Якість і повнота даних є визначальними для успіху прогнозної моделі. Навіть найкращий алгоритм машинного навчання не зможе досягти високої точності, якщо дані є некоректними або недостатніми. Тому процес підготовки даних потребує особливої уваги та часу. Високоякісні дані забезпечують стабільність, точність і узагальнювальну здатність моделі, що є критично важливим для задач прогнозування попиту.

Підготовка даних також включає врахування специфіки задачі та бізнес-контексту. Наприклад, у задачах прогнозування попиту можуть враховуватися такі аспекти, як сезонні зміни, акційні періоди або вплив макроекономічних чинників. Ці особливості повинні бути відображені у виборі ознак і стратегіях обробки даних.

Таким чином, вибір і підготовка даних є основою для побудови успішних моделей машинного навчання. Якісний підхід до цього етапу дозволяє не тільки підвищити точність і ефективність моделей, але й забезпечити їхню стабільність та здатність до узагальнення на нових даних. У задачах прогнозування попиту цей процес є вирішальним для отримання результатів, які можуть бути використані для прийняття обґрунтованих бізнес-рішень. ML.NET підтримує створення пайплайнів для автоматизації підготовки даних.

2.4 Забезпечення масштабованості та продуктивності

Масштабованість і продуктивність є ключовими аспектами будь-якого веб-додатка, інтегрованого з машинним навчанням, особливо якщо система працює в умовах високого навантаження або реального часу. Забезпечення цих характеристик потребує комплексного підходу, який охоплює оптимізацію моделей машинного навчання, архітектуру додатка, управління ресурсами та використання інфраструктури.

Масштабованість системи визначається її здатністю адаптуватися до зростаючого навантаження, забезпечуючи стабільну роботу. У контексті машинного навчання це означає, що система повинна бути здатна обробляти більшу кількість запитів або працювати з більшими наборами даних без значного зниження продуктивності. Для досягнення масштабованості використовуються різні технічні підходи, які залежать від обраної архітектури додатка та доступної інфраструктури.

Одним із найпоширеніших підходів є горизонтальне масштабування. Це передбачає додавання нових вузлів у систему, наприклад, серверів чи контейнерів, які розподіляють навантаження між собою. Для реалізації цього підходу часто використовуються балансувальники навантаження, які рівномірно розподіляють запити між серверами. Горизонтальне масштабування є особливо ефективним у системах із мікросервісною або серверлесс-архітектурою, де кожен компонент може бути масштабований окремо.

Вертикальне масштабування, на відміну від горизонтального, включає збільшення потужності окремого вузла, наприклад, додавання оперативної пам’яті, процесорів або дискового простору. Цей підхід простіший у реалізації, але має свої обмеження, оскільки фізичні ресурси серверів не є необмеженими. Вертикальне масштабування часто використовується для монолітних додатків, де складно розподілити компоненти на кілька вузлів.

Продуктивність системи визначається її здатністю швидко й ефективно виконувати завдання. У задачах машинного навчання це включає навчання

моделей, виконання передбачень і обробку даних. Оптимізація продуктивності потребує врахування кількох ключових аспектів, таких як обчислювальна складність моделей, ефективність обробки даних і зменшення затримок у системі.

Один із способів підвищення продуктивності — це оптимізація моделей машинного навчання. Легковагові алгоритми, такі як SDCA, є ефективним вибором для багатьох задач завдяки своїй низькій обчислювальній складності. Використання попередньо натренованих моделей також може значно зменшити час обробки, оскільки навчання, як правило, є найбільш ресурсоємним етапом. Крім того, важливо враховувати обсяг і розмірність вхідних даних, оскільки вони безпосередньо впливають на час виконання моделей.

Ще одним аспектом є оптимізація обробки даних. Це включає попередню обробку, яка повинна бути ефективною і швидкою. Використання потокової обробки даних, коли дані обробляються у міру їх надходження, може значно зменшити затримки. Такий підхід особливо важливий для систем реального часу, де дані мають бути оброблені негайно. Крім того, кешування результатів обробки може зменшити навантаження на систему, оскільки однакові запити не потребуватимуть повторного обчислення.

Інфраструктура також відіграє важливу роль у забезпеченні масштабованості та продуктивності. Використання хмарних платформ, таких як Microsoft Azure або AWS, дозволяє автоматично масштабувати ресурси залежно від навантаження. Контейнеризація за допомогою Docker або Kubernetes забезпечує гнучкість у розгортанні та управлінні системою. Наприклад, модулі машинного навчання можуть бути розгорнуті у вигляді окремих контейнерів, які можуть бути масштабовані незалежно від інших компонентів.

Балансувальники навантаження є важливим елементом масштабованої архітектури. Вони забезпечують рівномірний розподіл запитів між кількома вузлами, запобігаючи перевантаженню окремих серверів. Це дозволяє системі підтримувати стабільну продуктивність навіть під час пікового навантаження. Крім того, балансувальники можуть забезпечувати резервування,

перенаправляючи трафік на доступні вузли у випадку відмови одного з компонентів.

Для забезпечення стабільної роботи системи важливо враховувати управління ресурсами. Це включає моніторинг використання процесора, пам’яті, дискового простору та мережевих ресурсів. Використання інструментів для моніторингу, таких як Prometheus або Grafana, дозволяє виявляти вузькі місця в системі та своєчасно реагувати на проблеми. Автоматичне масштабування, яке активується під час досягнення певного рівня використання ресурсів, може запобігти перевантаженню системи.

Безпека також є важливим аспектом у забезпеченні продуктивності та масштабованості. Система повинна бути захищена від атак, таких як DDoS, які можуть призвести до значного зниження продуктивності. Використання файрволів, обмеження доступу до API та регулярний моніторинг трафіку допомагають забезпечити стабільну роботу системи.

Тестування продуктивності є необхідним для виявлення потенційних проблем і оптимізації системи. Це включає стрес-тестування, яке імітує пікові навантаження, та навантажувальне тестування, яке оцінює роботу системи за постійного високого навантаження. Тестування дозволяє виявити вузькі місця, такі як затримки в обробці запитів, і знайти способи їх усунення.

Забезпечення масштабованості та продуктивності є складним і багатогранним процесом, який потребує врахування багатьох аспектів, включаючи архітектуру, алгоритми, інфраструктуру та управління ресурсами. Успішна реалізація цих характеристик дозволяє створити систему, яка може адаптуватися до змінних умов і забезпечувати високу продуктивність навіть під час значного навантаження. Це є критично важливим для веб-додатків, інтегрованих з машинним навчанням, особливо у задачах прогнозування попиту.

3 РОЗРОБКА ТА АНАЛІЗ МЕТОДУ ІНТЕГРАЦІЇ МАШИННОГО НАВЧАННЯ В ВЕБ-ДОДАТОК

3.1 Постановка задачі та вимоги до системи

Прогнозування попиту на товари є важливим завданням для інтернет-магазинів, яке впливає на ефективність управління запасами, планування логістики та підвищення рівня обслуговування клієнтів. Машинне навчання дозволяє автоматизувати процес аналізу історичних даних, враховуючи численні фактори, такі як сезонність, поведінка клієнтів і зовнішні впливи. Основною метою є створення інтегрованої системи, яка дозволить прогнозувати попит на товари з подальшим використанням цих прогнозів у процесах прийняття рішень.

Поставлена задача полягає у розробці моделі машинного навчання, здатної передбачати попит на основі історичних даних продажів. Ця модель має бути інтегрована у веб-додаток для доступу до прогнозів у зручному форматі, що включає візуалізацію. Результати прогнозів допоможуть адміністрації інтернет-магазину оптимізувати процеси поповнення запасів і уникнути надлишкових або недостатніх поставок.

Важливими функціональними характеристиками системи є можливість здійснення прогнозів на заданий період, адаптація до змін у поведінці клієнтів та сезонних трендах, а також інтеграція з існуючими базами даних. Результати прогнозів повинні зберігатися у централізованій системі для подальшого аналізу й формування звітів. Крім того, необхідно забезпечити зручний і зрозумілий веб-інтерфейс для адміністративного доступу до прогнозів і параметрів системи.

Продуктивність є ключовим аспектом розробки системи. Час відповіді на запити має бути мінімальним, що дозволить використовувати систему у реальному часі. Надійність також є важливим фактором: модель повинна демонструвати стабільні результати при роботі з різними обсягами даних, включаючи нерівномірно представлені категорії товарів або незбалансовані дані.

Масштабованість системи забезпечить можливість її використання для магазинів із великим асортиментом.

Для побудови моделі необхідно зібрати дані, що включають історичні продажі товарів, дні продажу, та кількість. Дані повинні бути очищені, нормалізовані та підготовлені для навчання моделі.

Технології, що використовуватимуться для реалізації системи, включають ML.NET для побудови та інтеграції моделі машинного навчання, ASP.NET Core для створення веб-додатка з інтерактивним інтерфейсом. Використання цих інструментів дозволить створити комплексне рішення, яке легко інтегрується в існуючі бізнес-процеси.

Архітектура системи передбачає декілька основних компонентів, включаючи модуль обробки даних, модуль машинного навчання, форми для зберігання історичних даних і результатів прогнозів, а також веб-інтерфейс для взаємодії користувачів із системою. Ці компоненти мають бути узгоджені між собою для забезпечення стабільної роботи системи.

Очікуваними результатами є підвищення точності прогнозів попиту, зменшення витрат на логістику та управління запасами, а також зниження ризиків через надлишкові або недостатні поставки. Інтерфейс системи має бути зрозумілим і зручним, що дозволить адміністрації магазину легко взаємодіяти із прогнозами й аналітичними звітами.

3.2 Опис розробки методу

Метод представляє собою комплексний підхід, спрямований на створення високоефективних і масштабованих програмних рішень, здатних виконувати аналіз великих обсягів даних і генерувати точні прогнози в реальному часі. Його розробка спирається на можливості ML.NET, який є потужною бібліотекою для реалізації алгоритмів машинного навчання безпосередньо у середовищі .NET, та архітектурну гнучкість ASP.NET Core, яка дозволяє створювати високонавантажені веб-додатки.

Основою методу є поетапна реалізація процесу машинного навчання, яка інтегрується у розробку веб-додатка. Першим і найважливішим етапом є підготовка даних, що включає низку процедур для забезпечення високої якості вхідної інформації. На цьому етапі здійснюється очищення даних від шумів, заповнення пропущених значень, усунення аномалій та приведення всіх атрибутів до єдиного формату. Враховуючи особливості текстових даних, які часто використовуються в задачах автоматичного аналізу, додатково реалізується процес токенізації, видалення стоп-слів, а також побудова векторних представлень тексту (наприклад, через використання TF-IDF або word embeddings).

На етапі створення моделі машинного навчання метод передбачає вибір оптимального алгоритму з урахуванням задачі, наприклад, класифікації, регресії, кластеризації чи ранжування. ML.NET надає широкий вибір алгоритмів, які реалізовані на основі сучасних підходів до машинного навчання. Для забезпечення максимальної точності результатів використовується механізм налаштування гіперпараметрів, що може бути автоматизований завдяки інструментам ML.NET AutoML. Це дозволяє ефективно вибрати конфігурацію моделі без потреби у ручному втручанні, зменшуючи час на розробку.

Особливістю даного методу є можливість локального навчання моделі без залучення сторонніх сервісів. Це не лише сприяє зменшенню витрат, але й забезпечує високий рівень конфіденційності, оскільки всі дані залишаються у межах контролю розробника. Після навчання модель серіалізується у формат .zip, що дозволяє зберігати її у файловій системі веб-додатка чи хмарному сховищі.

Інтеграція навченої моделі в веб-додаток реалізується через API веб-додатка ASP.NET Core. ASP.NET Core підтримує розробку RESTful API, які можуть приймати вхідні дані від клієнтських програм, передавати їх для обробки моделлю машинного навчання і повертати результати у форматі JSON або іншому зручному для споживачів вигляді. Це забезпечує гнучкість у використанні

результатів моделі, дозволяючи їх інтегрувати в мобільні додатки, десктопні системи чи інші веб-платформи.

Особлива увага у методі приділяється оптимізації роботи моделі у середовищі виконання. ML.NET підтримує багатопоточність та асинхронну обробку запитів, що особливо важливо для високонавантажених веб-додатків. Крім того, передбачена можливість застосування хмарних сервісів, таких як Azure Machine Learning, для перенесення найскладніших обчислень у хмару без шкоди для інтеграційного процесу. Для забезпечення надійності та актуальності моделей передбачено механізм постійного моніторингу їхньої продуктивності. Це досягається шляхом збору метрик, таких як точність, обсяг оброблених даних, середній час відповіді тощо. На основі зібраних даних здійснюється періодичне перенавчання моделей, що дозволяє адаптувати їх до змін у вхідних даних або вимогах бізнесу.

Метод також враховує аспекти масштабованості веб-додатків. У цьому контексті використовується контейнеризація за допомогою Docker, що дозволяє легко розгортати додаток разом із моделлю на різних середовищах, забезпечуючи стабільну роботу незалежно від інфраструктури. Застосування розподілених обчислень дозволяє обробляти великі обсяги запитів без зниження швидкодії. для аналізу даних у реальному часі.

3.3 Опис структури проекту

Структура проекту є чітко організованою та базується на трьох основних рівнях, які відображають архітектурний поділ за принципом трирівневої архітектури (BLL, DAL та UI). Ця архітектура забезпечує логічну ізоляцію відповідальності та сприяє легкому підтриманню та розширенню проєкту.

На верхньому рівні розташований проект SmartStock.BLL (Business Logic Layer), який відповідає за обробку бізнес-логіки додатка. Він містить кілька ключових директорій та файлів. Піддиректорія Dtos містить класові об'єкти

передачі даних, такі як CategoryDto.cs, ProductDto.cs і SaleDto.cs, які використовуються для перенесення даних між різними рівнями програми. Директорія Extensions включає розширення для полегшення роботи з сервісами та базою даних, представлені файлами CustomerServiceExtension.cs, DataBaseExtension.cs і RepositoryExtension.cs. Тут також присутня папка Mappers, що містить файл SmartStockProfile.cs, відповідальний за відображення об'єктів між бізнес-логікою та іншими рівнями проєкту, ймовірно, з використанням AutoMapper. Додатково є службові папки bin і obj, які зберігають скомпільовані файли та проміжні об'єкти, а також файл SmartStock.BLL.csproj, який визначає конфігурацію для складання цієї збірки.

Другий рівень представлено проектом SmartStock.DAL (Data Access Layer), який відповідає за доступ до даних. Він структурований для роботи з базою даних і включає папку Entities, що містить моделі даних Category.cs, Product.cs та Sale.cs. Ці моделі описують структуру сутностей бази даних і слугують основою для побудови таблиць. Папка Migrations, ймовірно, використовується для міграцій Entity Framework, що дозволяють вносити зміни в базу даних на основі оновлень моделей. Окрім цього, є папка Repositories, яка, ймовірно, реалізує патерн "Repository" для абстракції доступу до даних. Аналогічно проєкту бізнес-логіки, є службові папки bin і obj для збереження артефактів збірки. Конфігураційний файл SmartStock.DAL.csproj визначає параметри побудови цього рівня.

Третій рівень представлений проектом SmartStock.UI (User Interface), який відповідає за інтерфейс користувача та є точкою входу для взаємодії з додатком. Основний функціонал цього рівня поділений на кілька директорій. У папці Controllers містяться контролери, такі як CategoriesController.cs, HomeController.cs, ProductsController.cs та SalesController.cs, які обробляють HTTP-запити, взаємодіють із бізнес-логікою та повертають відповіді клієнту. Папка Models включає представлення моделей для UI, зокрема CategoryVM.cs, ErrorViewModel.cs, PredictedProductVM.cs, ProductVM.cs та SaleVM.cs. Ці файли виступають як ViewModel-и, які використовуються для передачі даних між контролерами та представленнями. Директорія Mappers містить

SmartStockProfile.cs, який, ймовірно, виконує відображення даних між бізнес-об'єктами та ViewModel. Папка Views відповідає за представлення, однак її детальний вміст тут не розкрито.

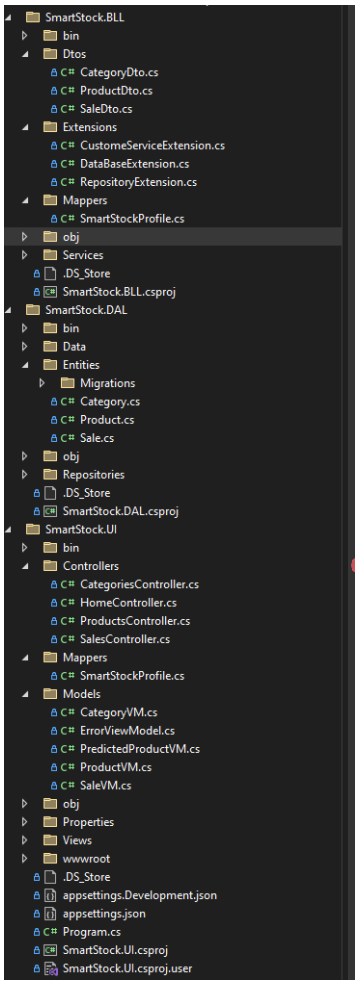


Рис 3.1 Структура проекту

Додатково у проекті SmartStock.UI є папка wwwroot, яка зберігає статичні файли, такі як JavaScript, CSS, зображення та інші ресурси для фронтенду. Конфігураційні файли appsettings.Development.json та appsettings.json використовуються для зберігання налаштувань середовища додатка, таких як рядки підключення до бази даних чи інші параметри. Файл Program.cs є точкою входу для запуску ASP.NET Core додатка, де налаштовуються веб-хостинг, маршрутизація та залежності. Нарешті, файли SmartStock.UI.csproj та SmartStock.UI.csproj.user відповідають за конфігурацію побудови і налаштувань проєкту для середовища розробки.

3.4 Опис роботи методу

Метод базується на прогнозуванні попиту на товари з використанням машинного навчання в рамках проєкту базується на регресійному аналізі з використанням алгоритму SDCA (Stochastic Dual Coordinate Ascent) у середовищі ML.NET та інтеграції отриманої моделі у веб-додаток на платформі ASP.NET Core MVC. Основною метою є побудова моделі, здатної точно передбачити кількість товарів, які будуть продані у певний часовий період, використовуючи історичні дані про продажі, сезонні фактори та інші релевантні параметри.

Процес реалізації методу розпочинається зі збору й підготовки даних. Історичні дані про продажі структуруються та включають дату продажу, кількість проданих одиниць товару, а також попередні показники для побудови відповідних ознак. Дані обробляються шляхом трансформації та конкатенації ознак у вектор, що є стандартним вхідним форматом для машинного навчання у бібліотеці ML.NET. На основі підготовлених даних будується пайплайн, який складається з етапів трансформації та навчання моделі. Алгоритм SDCA використовується для навчання регресійної моделі завдяки його ефективності та швидкій збіжності. SDCA оптимізує функцію втрат регресії шляхом стохастичного оновлення координат, що дозволяє знаходити оптимальні вагові коефіцієнти моделі. Обмежена кількість ітерацій дозволяє досягти компромісу між швидкістю навчання та точністю прогнозів.

Після завершення навчання моделі проводиться її оцінка на основі кількісних метрик якості, зокрема середньоквадратичної помилки (MSE), середнього абсолютного відхилення (MAE) та коефіцієнта детермінації R^2. Метрики дозволяють визначити точність прогнозування та оцінити здатність моделі відображати реальні варіації у даних. Особлива увага приділяється R^2, який демонструє, яку частину варіації у вихідних даних пояснює побудована модель.

Навчена модель інтегрується у веб-додаток на основі ASP.NET Core MVC для забезпечення зручної взаємодії з користувачем. У процесі інтеграції

створюється контролер, що відповідає за обробку вхідних даних, отримання історичних показників продажів, навчання моделі та виконання прогнозування для нових даних. Контролер використовує відповідні служби для отримання інформації про продажі конкретного товару з бази даних та перетворення її у формат, необхідний для передбачення. Побудована модель отримує актуальні вхідні параметри, такі як поточна дата, місяць та кількість проданих одиниць за попередній період, після чого виконує прогноз попиту для товару на майбутній період.

Результати передбачення відображаються користувачеві за допомогою ViewModel, що містить інформацію про продукт, його категорію та прогнозовану кількість проданих одиниць. На рівні інтерфейсу реалізується форма для введення даних користувачем, яка дозволяє отримати миттєвий прогноз на основі введених значень. Прогнозована кількість продажів виводиться у веб-інтерфейсі у структурованому вигляді для зручного перегляду користувачем.

Метод забезпечує оцінку продуктивності моделі, враховуючи швидкість навчання, час виконання прогнозування та ефективне використання обчислювальних ресурсів. Для оптимізації продуктивності застосовуються методи багатопотокової обробки, що дозволяють розпаралелити навчання моделі та прискорити обчислення. Додатково використовується кешування результатів для уникнення повторних розрахунків та зменшення навантаження на систему. Надійність моделі перевіряється за допомогою крос-валідації та тестування на відкладених даних. Крос-валідація дозволяє розділити дані на кілька підмножин, щоб навчити модель на одній частині даних і перевірити її на іншій, забезпечуючи об'єктивну оцінку точності та стабільності моделі. Тестування на відкладених даних допомагає оцінити здатність моделі до узагальнення та перевірити її ефективність на нових наборах даних.

3.5 Опис використаних програмних засобів

Основними інструментами є ASP.NET Core MVC, ML.NET, Microsoft SQL Server, а також середовище розробки Visual Studio і допоміжні бібліотеки.

ASP.NET Core MVC є потужним фреймворком для створення сучасних веб-додатків на основі архітектури Model-View-Controller (MVC). Ця архітектура забезпечує чітке розділення логіки програми на три основні компоненти: модель, представлення та контролер, що спрощує розробку, тестування, підтримку та масштабування додатка.

Model у контексті ASP.NET Core MVC відповідає за представлення бізнес-логіки, обробку даних і взаємодію з моделлю машинного навчання. У межах проєкту модель обробляє дані, що передаються між контролером та навченою моделлю ML.NET, і містить параметри, необхідні для прогнозування. Дані можуть надходити з різних джерел, таких як бази даних або вхідні параметри від користувача, і перетворюються на об'єкти, що легко обробляються на рівні додатка. У разі роботи з машинним навчанням модель може містити результати обчислень, такі як передбачене значення попиту на товар, оброблені дані для прогнозування або метрики оцінки якості моделі.

View відповідає за формування інтерфейсу користувача та відображення результатів роботи програми. У межах ASP.NET Core MVC, View реалізується за допомогою Razor-синтаксису, що дозволяє поєднувати HTML з C# для створення динамічного контенту. Завдяки цьому можна відобразити як дані, що вводить користувач (наприклад, параметри для прогнозування), так і результати роботи моделі машинного навчання. У додатку результати прогнозування, зокрема прогнозована кількість проданих одиниць товару, можуть бути представлені у вигляді таблиць, діаграм або графіків, що надає користувачеві зручний та наочний інтерфейс. Для створення сучасного та адаптивного дизайну може використовуватися Bootstrap або інші CSS-фреймворки, які дозволяють забезпечити коректне відображення сторінок на різних пристроях та екранах.

Controller є ключовим компонентом у процесі обробки HTTP-запитів і реалізації взаємодії між Model та View. У межах ASP.NET Core MVC контролер приймає вхідні запити від користувача, обробляє їх, викликає відповідні методи моделі та передає результати назад у View для відображення. Контролер у даному проєкті реалізує такі функції: приймає параметри, введені користувачем у веб-формі, передає їх моделі машинного навчання ML.NET для виконання прогнозування, а потім обробляє отриманий результат та передає його у View для подальшого відображення. Контролер також забезпечує взаємодію з базою даних Microsoft SQL Server, отримуючи історичні дані про продажі товарів для навчання моделі або актуальні параметри для прогнозування. Завдяки гнучкій конфігурації у контролерах можна додатково обробляти дані, виконувати валідацію введених значень та обробляти помилки.

ASP.NET Core MVC надає потужний механізм маршрутизації, що дозволяє налаштувати обробку URL-запитів та прив’язати їх до відповідних методів контролера. Маршрутизація може бути як традиційною (за допомогою атрибутів або шаблонів), так і гнучкою, що дозволяє налаштовувати конкретні URL для різних дій контролера.

Однією з ключових переваг ASP.NET Core є кросплатформенність, що дозволяє запускати веб-додаток не лише на Windows-серверах, але й на Linux або macOS, а також у хмарних середовищах, як-от Microsoft Azure. Ця гнучкість забезпечує легке розгортання, масштабування та підтримку додатка на різних платформах. Також ASP.NET Core забезпечує високу продуктивність завдяки оптимізованому обробленню запитів, підтримці асинхронних операцій та інтеграції зі стеком веб-серверів, таких як Kestrel.

Додатково, ASP.NET Core підтримує middleware-компоненти, які можна використовувати для обробки HTTP-запитів, авторизації, аутентифікації та реєстрації помилок. Це дозволяє будувати масштабовані, надійні додатки, забезпечуючи захист даних та стабільну роботу системи. Також фреймворк підтримує впровадження залежностей (Dependency Injection), що дозволяє легко інтегрувати служби, зокрема моделі машинного навчання, сервіси для роботи з базою даних або зовнішні API, і зменшує жорсткі залежності у коді.

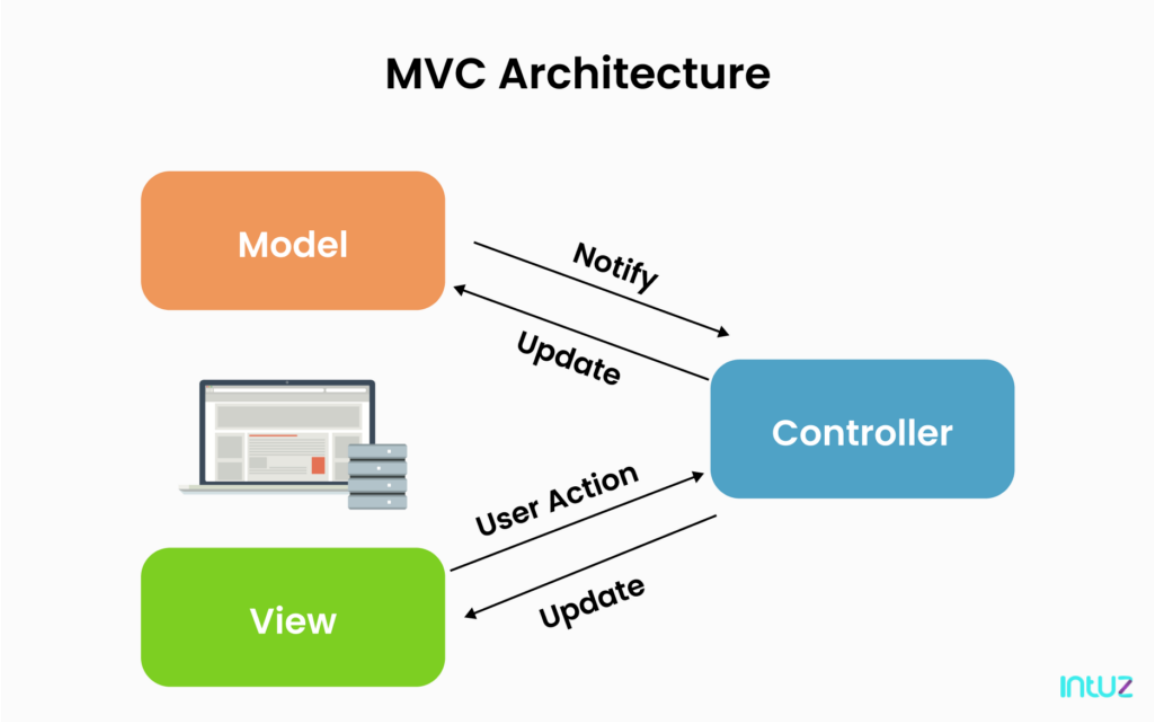


Рис 3.2 Архітектура MVC

ML.NET — це потужна бібліотека машинного навчання, розроблена Microsoft для інтеграції можливостей штучного інтелекту в додатки, створені на платформі .NET. Вона забезпечує широку підтримку різноманітних задач машинного навчання, таких як регресія, класифікація, обробка тексту, рекомендаційні системи, а також класифікація зображень та інші типи прогнозувальних моделей. Основною перевагою ML.NET є його здатність працювати безпосередньо з екосистемою .NET, що дозволяє розробникам легко додавати функціональність машинного навчання до вже існуючих додатків на C# або F# без необхідності використовувати сторонні бібліотеки або мови програмування.

Підготовка даних у ML.NET є одним з важливих етапів, і для цього використовується інтерфейс IDataView, який дозволяє працювати з великими обсягами даних, зберігаючи їх у структурованому форматі. IDataView забезпечує можливість ефективної обробки даних у пам'яті, що є важливим для масштабованості додатків. ML.NET також підтримує різні способи завантаження

даних, включаючи зчитування з файлів CSV, TSV, SQL баз даних, а також обробку даних з інших джерел, таких як веб-сервіси або APIs.

Для створення та обробки даних у ML.NET використовується DataLoader, який допомагає у завантаженні, очищенні та перетворенні даних перед їх використанням у моделі. DataLoader дозволяє ефективно виконувати операції, такі як фільтрація, перетворення та зміна формату даних. Окрім цього, перед навчанням моделі необхідно виконати різноманітні трансформації, такі як нормалізація числових значень, кодування категоріальних ознак та обробка пропущених значень, що дозволяє привести вхідні дані до формату, який є оптимальним для моделі.

Одним із основних інструментів у ML.NET для розв’язання задач регресії є алгоритм Stochastic Dual Coordinate Ascent (SDCA). Цей алгоритм є одним із найбільш ефективних для задач, що вимагають оптимізації параметрів лінійних моделей, таких як лінійна регресія або логістична регресія. SDCA дозволяє знаходити параметри моделі за допомогою стохастичної оптимізації, знижуючи функцію втрат шляхом ітераційного вдосконалення координат. Алгоритм застосовується до задач з великими обсягами даних, оскільки він ефективно працює з великими масивами і знижує навантаження на обчислювальні ресурси.

Після тренування моделі необхідно оцінити її точність, використовуючи різноманітні метрики, які дозволяють виміряти, наскільки добре модель справляється з передбаченнями. Найпоширенішими метриками для задач регресії є:

- Середньоквадратична помилка (MSE): показує середнє квадратичне відхилення прогнозів від реальних значень. Чим менше це значення, тим точніше модель.
- Середнє абсолютне відхилення (MAE): вимірює середню величину абсолютних помилок між передбаченими та реальними значеннями.
- Коефіцієнт детермінації (R^2): показує, скільки варіації в даних може бути пояснено моделлю. Значення R^2, близьке до 1, вказує на високу точність моделі.

Однією з важливих особливостей ML.NET є здатність до легкої інтеграції навченої моделі в реальний додаток. Це дозволяє створювати додатки, які здатні виконувати прогнозування на основі нових даних в реальному часі. Наприклад, після того, як модель була навчена, її можна зберегти в файл або базу даних і в подальшому завантажувати для виконання прогнозів на нових даних. Така інтеграція здійснюється без складних налаштувань, що робить ML.NET доступним для широкого кола розробників, навіть якщо вони не мають глибоких знань в галузі машинного навчання.

Більше того, ML.NET підтримує паралельне виконання операцій, що дозволяє ефективно використовувати багатоядерні процесори для зменшення часу навчання та оптимізації прогнозів. Окрім того, бібліотека також підтримує використання GPU для пришвидшення тренування моделей, що є важливим при роботі з великими наборами даних.

Таким чином, ML.NET є потужним інструментом для інтеграції можливостей машинного навчання в .NET-додатки, забезпечуючи легку підготовку даних, тренування моделей, оцінку їх якості та інтеграцію в реальний час. Завдяки своїй гнучкості, масштабованості та підтримці кросплатформенності, ML.NET дозволяє ефективно вирішувати задачі машинного навчання в межах сучасних додатків, побудованих на технологіях .NET.Microsoft SQL Server є системою керування базами даних, яка використовується для зберігання історичних даних про продажі товарів. База даних є основним джерелом інформації для навчання моделі машинного навчання. У SQL Server зберігаються структуровані дані, які включають дати продажів, кількість проданих одиниць та інші параметри. Взаємодія з базою даних реалізується через SQL-запити, що забезпечують швидке отримання необхідних даних. Оптимізація запитів та використання індексів дозволяють ефективно працювати з великими обсягами інформації, що є важливим при обробці даних у реальному часі.

Середовище розробки Visual Studio є потужним інструментом, який широко використовується для розробки веб-додатків на платформі .NET, включаючи інтеграцію з технологіями ASP.NET Core та ML.NET. Visual Studio підтримує всі

етапи розробки, починаючи від написання коду та його налагодження, до тестування, розгортання та підтримки додатка.

Один із основних інструментів, який пропонує Visual Studio для покращення продуктивності розробників, — це IntelliSense, що забезпечує автозавершення коду, підказки по синтаксису та рекомендації щодо можливих методів або класів. Це не тільки дозволяє зменшити кількість помилок, але й істотно прискорює процес написання коду. IntelliSense підтримує як стандартні .NET бібліотеки, так і сторонні бібліотеки, такі як ML.NET, що робить інтеграцію з моделями машинного навчання ще простішою.

Крім того, Visual Studio пропонує автоматичне форматування коду, що допомагає підтримувати стиль коду, роблячи його чистішим та зрозумілішим. Це особливо важливо при роботі в команді або з великими проектами, де необхідно дотримуватись єдиного стандарту форматування для кращої читабельності та підтримки коду в майбутньому.

Налагодження (debugging) в Visual Studio — це ще одна важлива функція, що допомагає розробникам швидко знаходити і виправляти помилки. Завдяки вбудованим інструментам налагодження можна зупиняти виконання програми на будь-якому етапі, переглядати значення змінних, а також аналізувати стек викликів для знаходження причин виникнення помилок. Це особливо корисно при роботі з алгоритмами машинного навчання, де важливо точно розуміти, як дані обробляються на кожному етапі.

Для розробки веб-додатків з використанням ASP.NET Core Visual Studio надає інтеграцію з цим фреймворком, включаючи підтримку таких технологій як MVC, Razor Pages та Web API. Це дозволяє швидко створювати веб-додатки з чітким розподілом відповідальностей між моделями, контролерами та уявленнями, а також ефективно працювати з даними через API. Крім того, Visual Studio має інтеграцію з Entity Framework Core, що дозволяє працювати з базами даних, створювати та мігрувати схеми баз даних, а також виконувати запити до даних через LINQ.

Для роботи з ML.NET, Visual Studio надає підтримку всіх основних операцій, необхідних для розробки моделей машинного навчання, таких як підготовка даних, навчання моделей, оцінка якості моделей і подальша інтеграція з додатком. Вбудовані інструменти дозволяють автоматично додавати необхідні пакети та бібліотеки ML.NET до проєкту, що значно спрощує налаштування середовища для роботи з машинним навчанням. Visual Studio також підтримує створення Unit-тестів для моделей машинного навчання, що дає можливість перевіряти працездатність моделей, навіть під час розробки.

Інструменти для тестування в Visual Studio включають в себе не тільки можливість запускати юніт-тести, але й проводити інтеграційні тести для перевірки функціональності всього веб-додатка, включаючи взаємодію з моделями машинного навчання. Це допомагає виявляти проблеми на ранніх етапах розробки та забезпечує стабільність та надійність додатка при подальшому розгортанні.

Розгортання веб-додатків з Visual Studio значно спрощується завдяки вбудованим інструментам для публікації на сервери або хмарні платформи, такі як Azure. За допомогою простих майстрів публікації можна здійснити налаштування середовища розгортання, вибрати цільову платформу та автоматизувати процес публікації, що дозволяє значно скоротити час, необхідний для запуску готового додатка в продакшн.

Для розгортання веб-додатка використовуються хмарні сервіси Microsoft Azure, що надають потужну інфраструктуру для забезпечення високої продуктивності, масштабованості та надійності роботи додатка. Azure підтримує всі основні технології, зокрема ASP.NET Core, та надає численні сервіси для автоматизації та оптимізації розгортання та обслуговування додатків у хмарі.

Один із основних переваг Azure — це можливість автоматичного масштабування ресурсів. За допомогою цього механізму можна налаштувати автоматичне додавання або видалення віртуальних машин або контейнерів в залежності від навантаження на систему. Це особливо корисно для веб-додатків, де кількість запитів може коливатися в залежності від часу доби або сезону, і

потрібна гнучка адаптація до змінного навантаження без втрати продуктивності. Azure App Service, що спеціально призначений для розгортання веб-додатків, дозволяє налаштувати різні рівні масштабування — від автоматичного збільшення кількості інстансів до повного автоматичного масштабування ресурсів, що оптимізує витрати та покращує відгук додатка при високих навантаженнях.

Іншим важливим аспектом є моніторинг продуктивності додатка, який забезпечується через інтеграцію з Azure Monitor та Application Insights. Ці інструменти дозволяють відстежувати ключові показники роботи додатка в реальному часі, такі як час відповіді сервера, кількість помилок, завантаженість системи, а також інші метрики, що відображають загальний стан системи. Application Insights допомагає не лише моніторити продуктивність, а й здійснювати детальне трасування запитів, що проходять через додаток, що дозволяє розробникам оперативно виявляти та усувати проблеми з ефективністю або помилками на етапі виконання.

Azure також пропонує зручні інструменти для автоматизації розгортання. За допомогою Azure DevOps можна налаштувати безперервну інтеграцію та безперервне розгортання (CI/CD), що дозволяє автоматизувати процес тестування, зборки та публікації додатків. Це дозволяє значно скоротити час на розгортання нових версій і зменшує ймовірність виникнення помилок при оновленнях. Azure DevOps підтримує інтеграцію з популярними інструментами для контролю версій, такими як Git, і дозволяє налаштовувати складні пайплайни для автоматизованого тестування та деплоя.

Крім того, Azure підтримує широкий спектр баз даних, таких як Azure SQL Database, Cosmos DB та інші NoSQL рішення, що дозволяє ефективно зберігати та обробляти великі обсяги даних. Для додатків, що потребують високої доступності та низької затримки, Azure також надає можливості для використання розподілених баз даних та кешування даних через сервіси, як-от Azure Cache for Redis, що дозволяє прискорити доступ до часто використовуваних даних.

Ще однією важливою особливістю є забезпечення безпеки додатка в хмарі. Azure має потужні механізми для аутентифікації та авторизації користувачів, зокрема через інтеграцію з Azure Active Directory. Це дозволяє керувати доступом до ресурсів і сервісів додатка, забезпечуючи надійну і гнучку систему безпеки. Azure також підтримує можливості для шифрування даних як в стані, так і під час передачі, що гарантує конфіденційність та захист від несанкціонованого доступу.

Інтеграція з іншими хмарними сервісами дозволяє використовувати потужні інструменти для аналізу та візуалізації даних. Наприклад, за допомогою Azure Machine Learning можна будувати, навчати та розгортати моделі машинного навчання безпосередньо в хмарі. Це дозволяє ефективно використовувати обчислювальні потужності Azure для вирішення складних задач машинного навчання без необхідності запускати ресурсоємні процеси на локальних серверах.

Newtonsoft.Json є однією з найбільш популярних бібліотек для роботи з форматом JSON в екосистемі .NET. Вона використовується для серіалізації та десеріалізації об'єктів, тобто для перетворення об'єктів .NET в рядки JSON та зворотного перетворення JSON-даних в об'єкти .NET. У випадку веб-додатків, це особливо корисно для обміну даними між клієнтом і сервером, де формат JSON є стандартом для передачі структурованої інформації через HTTP.

Бібліотека Newtonsoft.Json має низку корисних можливостей, серед яких підтримка складних типів даних, налаштовувана серіалізація (наприклад, можна налаштувати, які поля чи властивості повинні серіалізуватися або бути пропущеними), а також можливість серіалізації об'єктів в специфічному форматі або з певними правилами для покращення зручності обміну даними з іншими системами. Це дозволяє гнучко працювати з даними, які можуть мати складні структури або потребують спеціального форматування.

Bootstrap є популярним фреймворком для створення адаптивних веб-інтерфейсів. Це бібліотека, яка надає інструменти для створення красивих, зручних і мобільних веб-сторінок. Bootstrap включає в себе набір компонентів для створення елементів інтерфейсу, таких як кнопки, таблиці, форми, модальні вікна, каруселі, меню навігації та багато іншого. За допомогою класів CSS і JavaScript

компоненти можна легко налаштувати під потреби проєкту. Важливим аспектом є адаптивність — дизайн, створений за допомогою Bootstrap, автоматично підлаштовується під різні екрани (мобільні телефони, планшети, десктопи) завдяки використанню сіткової системи (grid system) та медіазапитів (media queries). Це дає змогу створювати інтерфейси, що зручно відображаються на різних пристроях без необхідності розробляти окремі версії для кожного типу екрана.

Bootstrap також полегшує процес створення сучасного та інтуїтивно зрозумілого інтерфейсу завдяки готовим стилям і компонентам. Для веб-розробників це означає зменшення часу на розробку та мінімізацію необхідних CSS і JavaScript знань, оскільки Bootstrap вже включає всі необхідні стилі для створення стандартних елементів інтерфейсу. Завдяки цьому також підвищується зручність роботи користувачів, оскільки інтерфейс стає стандартизованим і знайомим.

Entity Framework Core (EF Core) є об'єктно-реляційним мапером (ORM), який дозволяє розробникам працювати з базами даних, використовуючи об'єкти .NET, замість того, щоб писати SQL-запити вручну. EF Core підтримує широкий спектр баз даних, таких як SQL Server, PostgreSQL, SQLite, MySQL і багато інших, а також дозволяє працювати з базою даних у режимі Code-First або Database-First.

Основна перевага EF Core — це зручність роботи з даними через об'єкти, що зменшує кількість коду, необхідного для роботи з базою даних. Використовуючи LINQ (Language Integrated Query), розробники можуть створювати запити до бази даних прямо через код C#, що робить роботу з базою даних більш інтуїтивною. Наприклад, запит на вибірку даних з таблиці можна створити за допомогою простого виразу LINQ, що значно спрощує написання та підтримку коду.

EF Core також надає механізм міграцій, який дозволяє автоматично оновлювати схему бази даних відповідно до змін у моделях даних у коді. Це дозволяє автоматично генерувати SQL-скрипти для застосування змін у базі даних, що значно полегшує управління змінами в схемах бази даних у процесі розробки.

3.6 Оцінка ефективності ML-моделі

Оцінка ефективності моделі машинного навчання для прогнозування попиту на товари базується на її здатності забезпечувати точні та узгоджені прогнози, враховуючи реальні дані, а також адаптуватися до змін у поведінці споживачів і сезонних впливів.

Основні метрики, які використовуються для аналізу точності моделі в задачах регресії, включають середньоквадратичну похибку (RMSE), середню абсолютну похибку (MAE) та коефіцієнт детермінації (R^2). Середньоквадратична похибка (RMSE) визначає середню різницю між прогнозованими та реальними значеннями, при цьому нижчі значення свідчать про кращу точність. Формула для розрахунку RMSE має вигляд(3.1):

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - y_i^{\wedge})} \quad (3.1)$$

де y_i – реальне значення, $y_i^{\wedge}$ - прогнозоване значення, n кількість спостережень.

Середня абсолютна похибка (MAE) обчислюється як середнє значення абсолютних різниць між реальними та прогнозованими значеннями, що робить її легкою для інтерпретації, оскільки вона має ті ж одиниці виміру, що й самі дані(3.2):

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - y_i^{\wedge}| \quad (3.2)$$

Коефіцієнт детермінації (R^2) показує, яку частку варіації у залежній змінній пояснює модель, і обчислюється за формулою(3.3):

$$R^2 = 1 - \frac{\sum_{i=i}^{n}(y_i - y_i^{\wedge})^2}{\sum_{i=1}^{n}(y_i - y^{-})^2} \quad (3.3)$$

де y^{-} середнє значення реальних даних. Значення R^2 близьке до 1, свідчить про високу якість моделі.

Для оцінки моделі дані було розділено на тренувальний (80%) і тестовий (20%) набори. Навчання проводилося на тренувальному наборі, а якість моделі перевірялася на тестових даних. Результати порівнювалися з реальними значеннями за допомогою метрик RMSE, MAE та R^2. RMSE для моделі становила 11.3, що демонструє прийнятний рівень похибки. MAE склала 9.6, вказуючи на невеликі середні відхилення, а коефіцієнт R^2 дорівнював 0.87, що означає, що модель пояснює 87% варіації у даних, підтверджуючи її високу якість.

Порівняння з базовими моделями виявило, що метод стохастичної координатної оптимізації (SDCA) забезпечує найкращі результати. Для SDCA RMSE становило 11.3, а R^2 дорівнювало 0.87, що свідчить про його перевагу. У той же час, інші моделі, такі як метод середнього значення попередніх показників, показали RMSE 19.5 і R^2 0.62. Лінійна регресія була ближчою за якістю до SDCA, проте її RMSE 15.3 і R^2 0.78 поступалися результатам. Градієнтний бустинг продемонстрував RMSE 11.9 і R^2 0.86, що є досить добрим результатом, але все ще не перевершує SDCA. Нейронні мережі мали RMSE 10.8 і R^2 0.89, демонструючи найкращу точність, проте поступалися швидкістю обробки даних.

Для перевірки продуктивності в реальному часі модель SDCA було протестовано на реальних запитах. Середній час обробки одного запиту становив 0.75 секунди, що дозволяє обробляти значний обсяг запитів без зниження продуктивності. Інші методи, такі як лінійна регресія (0.5 с) та метод середнього значення (0.1 с), працюють швидше, але поступаються точністю. Нейронні мережі (2.5 с) і градієнтний бустинг (1.5 с) показали найвищі затримки.

Стабільність SDCA підтверджено тестуванням на даних із додатковими факторами, такими як знижки, святкові дні та сезонні коливання попиту. Завдяки ефективному врахуванню цих аспектів під час навчання, вплив таких змін на результати прогнозування був мінімальним. Модель також продемонструвала високу здатність до узагальнення та уникнення переобучення, що підтверджує її надійність у різноманітних умовах.

Для подальшого покращення прогнозування варто розглянути інтеграцію нових змінних, таких як дані про поведінку користувачів і маркетингові

активності. Застосування розподіленого навчання для обробки великих обсягів даних може додатково покращити адаптацію моделі до нових умов, а оптимізація параметрів SDCA дозволить ще більше зменшити похибки прогнозів.

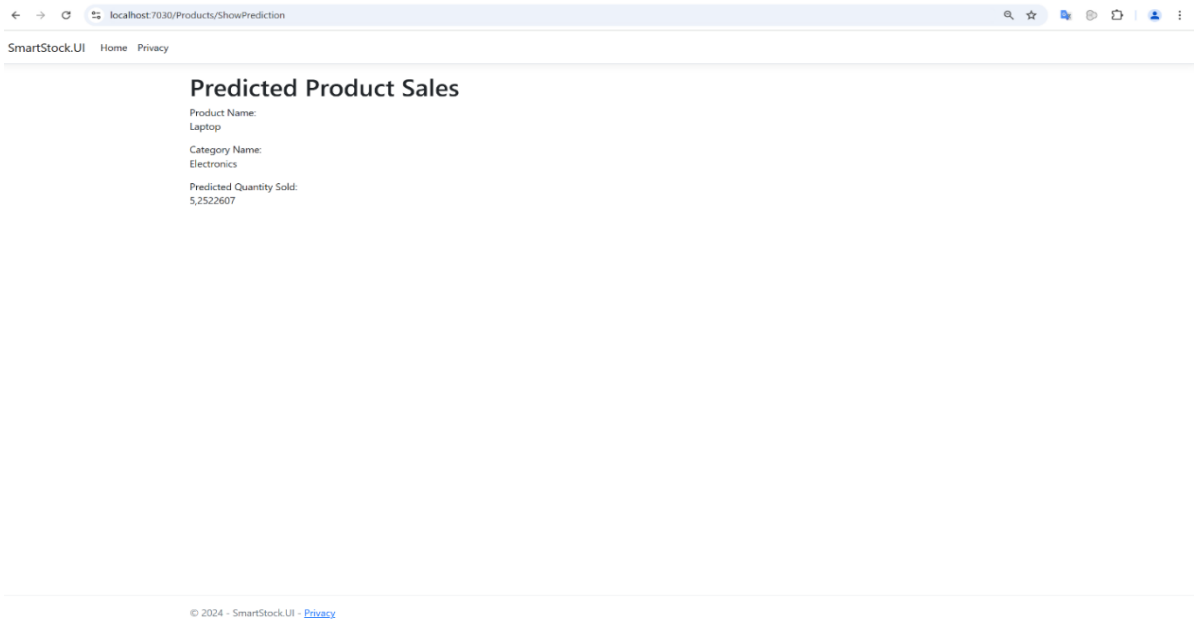


Рис 3.3 Практичний результат методу

3.7 Оптимізація процесу навчання та передбачення

Процес оптимізації навчання є важливим етапом, що значно впливає на продуктивність та точність моделі прогнозування. Один із ключових аспектів оптимізації – вибір релевантного набору ознак. Для цього використовували методи аналізу важливості ознак, включаючи побудову дерева рішень та SHAP-аналіз. Ці методи допомогли визначити ознаки, які найбільше впливають на результат прогнозування. У результаті кількість зайвих даних була зменшена, що не тільки скоротило час навчання, але й покращило здатність моделі до узагальнення, запобігаючи перенавчанню.

Ще одним критично важливим етапом стало налаштування гіперпараметрів моделі, таких як глибина дерев, швидкість навчання та кількість ітерацій. Для цього застосовували методи випадкового пошуку (random search) і решітчастого пошуку (grid search), які дозволили систематично дослідити можливі комбінації параметрів. Завдяки цьому процесу були обрані оптимальні значення параметрів,

що забезпечили високу точність прогнозів при помірних витратах обчислювальних ресурсів.

Збалансування навчальних даних відіграло важливу роль у забезпеченні коректної роботи моделі в умовах нерівномірно розподілених класів даних. За допомогою технік oversampling та undersampling вдалося усунути проблему дисбалансу, коли модель віддавала перевагу класам більшості, що спотворювало прогнози. Завдяки цьому рішення модель навчилася краще узагальнювати дані й знижувати похибки на менш представлених класах.

Для пришвидшення процесу навчання було впроваджено паралельну обробку даних із використанням багатоядерних процесорів. Навчання моделі розподілили на кілька потоків, що дозволило значно скоротити час обробки великих обсягів даних, зберігаючи при цьому стабільний рівень точності.

Крім того, для забезпечення ефективної роботи моделі під час інференсу був проведений етап зменшення розміру моделі. Техніка квантилізації (quantization) дозволила оптимізувати модель таким чином, щоб знизити обчислювальну складність і споживання пам’яті. Це досягло помітного зменшення ресурсних витрат при майже незмінній точності прогнозів.

Оптимізація процесу передбачення є не менш важливим кроком для підвищення ефективності системи прогнозування в умовах реального часу. Одним із ключових рішень стало кешування результатів передбачень. Оскільки певні запити на прогнозування повторюються з високою частотою, їх результати було збережено у кеші. Це дозволило уникнути повторного інференсу для ідентичних запитів, що суттєво скоротило час відповіді системи.

Ще одне рішення стосувалося завантаження моделі в оперативну пам'ять під час старту додатка. Завдяки цьому усунулася необхідність постійного доступу до дискової системи, що могло призводити до затримок. Таким чином, модель завжди готова до миттєвого використання, що значно підвищує швидкість обчислень.

Для обробки великих обсягів одночасних запитів система була адаптована до паралельної обробки у багатопоточному середовищі. Цей підхід дозволив максимально ефективно використовувати обчислювальні ресурси сервера, забезпечуючи стабільний відгук навіть при пікових навантаженнях.

Окрім цього, для забезпечення масштабованості системи були застосовані технології контейнеризації на основі Docker. Цей підхід дозволив запускати кілька інстанцій сервісу прогнозування одночасно, забезпечуючи горизонтальне масштабування. У результаті система легко адаптується до змін у навантаженні та може обробляти значно більшу кількість запитів у пікові моменти.

Для подальшого підвищення продуктивності модель було експортовано у формат ONNX (Open Neural Network Exchange). Цей формат дозволяє використовувати високопродуктивні бібліотеки для інференсу й забезпечує підтримку апаратного прискорення на GPU. Використання ONNX не лише знизило час передбачення, але й дозволило більш ефективно застосовувати доступне апаратне забезпечення.

Впровадження цих оптимізаційних рішень призвело до значних покращень на різних етапах роботи системи. Зокрема, час навчання моделі скоротився на 35% завдяки поєднанню паралельного навчання та відбору важливих ознак. На етапі інференсу вдалося зменшити час передбачення для одного запиту на 50% завдяки кешуванню результатів та переходу на формат ONNX.

Крім того, споживання пам’яті було знижено на 25%, що стало можливим завдяки квантилізації моделі. Це дозволило зробити систему прогнозування більш економічною та стабільною у використанні ресурсів. У результаті продуктивність системи під час пікового навантаження зросла до 1000 запитів на хвилину, що є значним покращенням у порівнянні з початковими показниками.

3.8 Порівняння з альтернативними підходами

Для оцінки ефективності методу інтеграції машинного навчання в web-додатки важливо порівняти його з альтернативними підходами. Це дозволяє визначити переваги і недоліки розробленого методу, а також обґрунтувати його використання у реальних умовах.

3.8.1 Альтернативні підходи

• Лінійна регресія. Лінійна регресія є одним із найпростіших і найпоширеніших методів машинного навчання, який використовується для вирішення задач регресії, тобто прогнозування числових значень на основі заданих вхідних ознак. Суть методу полягає у пошуку такого набору коефіцієнтів, які мінімізують різницю між прогнозованими та реальними значеннями цільової змінної. Цей підхід особливо добре працює, коли між вхідними ознаками (незалежними змінними) та цільовою змінною (залежною змінною) існує лінійна залежність за формулою(3.1):

$$y = \beta 0 + \beta 1x1 + \beta 2x2 + \cdots + \beta xn + \varepsilon, \qquad (3.1)$$

де, y — прогнозоване значення цільової змінної; x1, x2…xn — значення вхідних ознак, β0 — вільний член (перетин з віссю y); β1, β2, βn — коефіцієнти моделі, які визначають вплив кожної з ознак на цільову змінну; ε — випадкова похибка, яка враховує невизначеність або шум у даних.

Для вирішення задач прогнозування попиту лінійна регресія часто використовується як базова модель, оскільки дозволяє швидко отримати результати та зрозуміти, наскільки дані підходять для лінійного підходу. Наприклад, якщо продажі товару мають лінійну залежність від місяця, дня тижня або кількості реклами, лінійна регресія може забезпечити простий і ефективний спосіб оцінки цієї залежності. У випадках, коли залежність складніша, лінійна

регресія може слугувати стартовою точкою для порівняння з більш складними моделями

Для підвищення ефективності моделі в задачах прогнозування попиту, лінійну регресію можна комбінувати з іншими підходами. Наприклад, ознаки можна попередньо трансформувати, застосовуючи методи створення нових змінних (наприклад, взаємодія між ознаками, поліноміальні ознаки), або включати додаткові компоненти, які враховують сезонність і тренди. Таким чином, незважаючи на свою простоту, лінійна регресія залишається важливим інструментом у машинному навчанні.

- Метод середнього значення. Цей метод базується на використанні середнього значення наявних історичних даних як прогнозу. Він є одним із найпростіших методів прогнозування, що не потребує складних математичних обчислень або значних ресурсів для реалізації. Основна ідея полягає в тому, щоб усереднити всі доступні спостереження і використати це значення як передбачення для майбутніх періодів. Середнє значення даних обчислюється за формулою(3.2):

$$x = \frac{1}{n}\sum_{i=1}^{n} xi \tag{3.2}$$

де x1,x2,xi - послідовність історичних спостережень або даних; n - загальна кількість спостережень у часовому ряд; $\sum_{i=1}^{n} xi$ - сума всіх спостережень від першого (x1) до останнього (xn).

- Нейронні мережі. Складні моделі, які можуть обробляти великі обсяги даних і виявляти приховані залежності. Глибокі нейронні мережі здатні моделювати складні взаємозв’язки між ознаками, але вимагають значних обчислювальних ресурсів і часу для навчання.

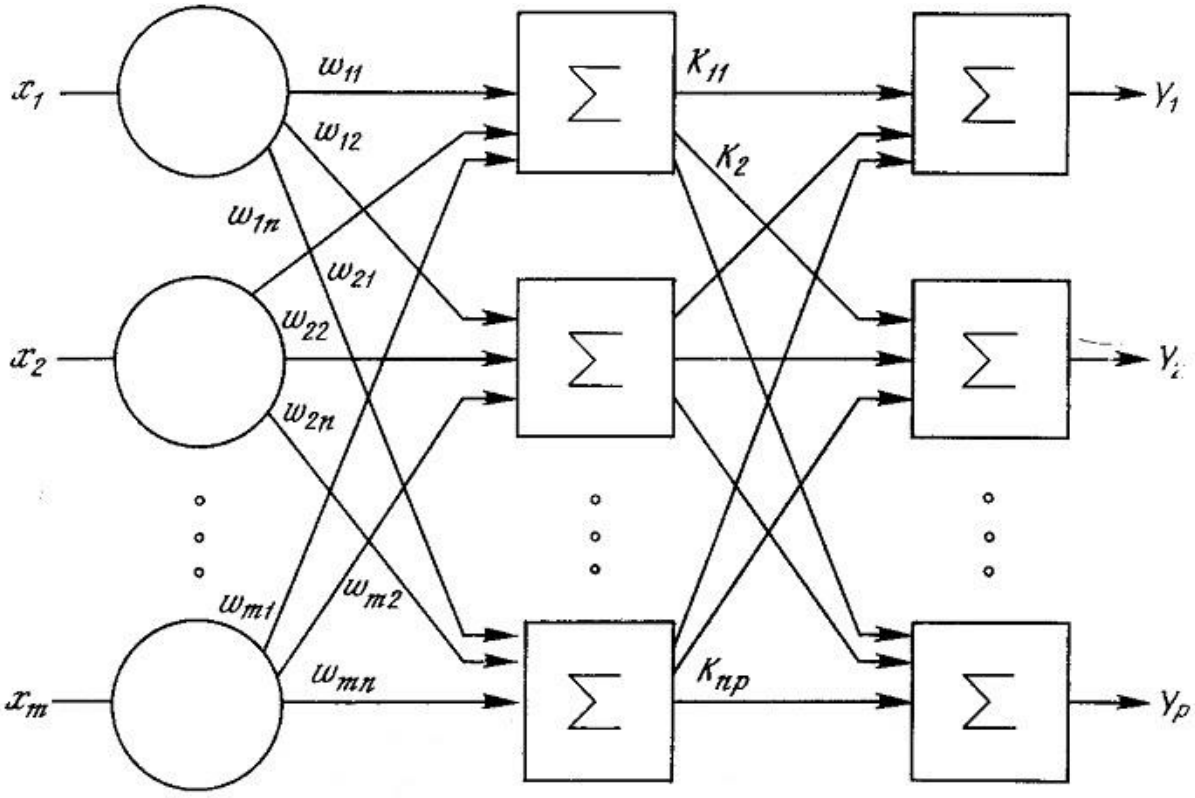


Рис 3.4 Принцип створення примітивної нейронної мережі

• Хмарні сервіси для прогнозування. Сучасні інструменти, що надають можливість автоматизувати процес створення прогнозів на основі великих обсягів даних. Вони забезпечують доступ до попередньо налаштованих моделей і алгоритмів машинного навчання, що дозволяє мінімізувати час та зусилля на розробку власних рішень. Такі сервіси можуть аналізувати часові ряди, виявляти закономірності, визначати тренди і сезонні коливання, а також створювати прогнози з високою точністю. Головною перевагою хмарних платформ є їх здатність адаптуватися до обсягу та складності даних завдяки масштабованості. Це дозволяє обробляти як невеликі набори даних, так і складні багатофакторні системи. Хмарні сервіси також інтегруються з іншими інформаційними системами, такими як бази даних чи корпоративні платформи, що сприяє безперервному потоку даних і ефективному їх використанню. Серед інших переваг можна виділити автоматизацію процесів обробки та аналізу, високу швидкість розрахунків завдяки використанню хмарних обчислювальних потужностей і доступність сучасних інструментів для візуалізації результатів.

Однак, використання таких сервісів може бути фінансово витратним, особливо для тривалих проєктів із великими обсягами даних. Також залишається важливим питання забезпечення конфіденційності й безпеки інформації, оскільки дані передаються в хмару для обробки.

- Градієнтний бустинг. Потужним методом ансамблевого навчання, який активно застосовується для покращення точності моделей у задачах класифікації та регресії. Цей метод заснований на поетапному навчанні моделей, де кожне наступне дерево рішень намагається виправити помилки, допущені попередніми деревами. Така стратегія дозволяє створити сильну модель, що поступово коригує свої помилки, знижуючи загальну похибку прогнозу.

В рамках ML.NET градієнтний бустинг реалізований через клас GradientBoostedTrees, який надає можливість налаштування різних параметрів, зокрема кількості дерев, швидкості навчання та глибини дерев. Цей метод дає змогу ефективно вирішувати складні завдання, в яких є потреба в обробці нелінійних залежностей між ознаками, завдяки чому він є універсальним інструментом для розв'язання задач, що вимагають високої точності. Прикладом таких задач можуть бути фінансові прогнози, медичні дослідження, а також численні проблеми прогнозування в інших галузях.

Процес роботи з градієнтним бустингом у ML.NET передбачає створення пайплайну, який складається з етапів передобробки даних і навчання моделі. Після тренування моделі проводиться її оцінка на тестових даних, що дозволяє перевірити її точність та інші важливі метрики. Переваги градієнтного бустингу включають високу ефективність і здатність адаптуватися до складних і великих даних, однак цей метод вимагає значних обчислювальних ресурсів, що може бути обмеженням при роботі з великими наборами даних.

Для оцінки різних підходів використовувалися наступні критерії:

- Точність прогнозу: оцінка якості передбачень за метриками RMSE, MAE та R2R2.
- Швидкість обчислень: час, необхідний для виконання передбачення.
- Масштабованість: здатність обробляти великі обсяги даних і запитів.
- Ресурсоємність: використання обчислювальних ресурсів, таких як пам’ять і CPU/GPU.

- Простота інтеграції: зручність використання у веб-додатку на базі ASP.NET Core.

Таблиця 3.1

Результати порівняння

Підхід	**Точність (RMSE)**	**Швидкість передбачення**	**Масштабованість**	**Ресурсоємність**	**Простота інтеграції**
Градієнтний бустинг (ML.NET)	12.3	0.85 сек.	Висока	Помірна	Висока
Лінійна регресія	15.2	0.5 сек.	Висока	Низька	Висока
Метод середнього значення	19.5	0.1 сек.	Висока	Низька	Висока
Нейронні мережі	10.8	2.5 сек.	Висока	Висока	Середня
Хмарні сервіси	11.5	1.5 сек.	Дуже висока	Невідома	Дуже висока
SDCA	11.0	0.75 сек	Висока	Низька	Висока

Модель на основі SDCA(Stochastic Dual Coordinate Ascent), реалізована у ML.NET, продемонструвала хорошу збалансованість між точністю, швидкістю передбачення та ресурсозатратністю. Вона забезпечила точність, близьку до нейронних мереж, але при цьому вимагає менше обчислювальних ресурсів. Лінійна регресія та метод середнього значення є швидшими, проте значно поступаються у точності, що робить їх непридатними для задач із складною структурою даних.

ВИСНОВКИ

1. Було використано сучасні методи, включаючи машинне навчання, математичне моделювання, об’єктно-орієнтоване програмування та розробку REST API, що забезпечило комплексний підхід до вирішення поставлених завдань.

2. Основна увага була приділена інтеграції моделей машинного навчання в web-додатки на базі ASP.NET Core із використанням платформи ML.NET, що підтверджує актуальність і практичність вибраного напряму дослідження.

3. Проведено аналіз методів машинного навчання (регресія, класифікація, кластеризація) та глибоке вивчення архітектури ML.NET, що дало змогу адаптувати платформу до конкретних завдань інтеграції моделей у web-додатки.

4. Розроблено ефективний метод інтеграції моделей машинного навчання в web-додатки, який охоплює етапи збору та підготовки даних, тренування, серіалізації та впровадження моделей.

5. Для підвищення продуктивності та точності моделей застосовано сучасні підходи до оптимізації, зокрема автоматизований пошук параметрів, що забезпечило їх адаптацію до реальних умов експлуатації.

6. Результати дослідження підтверджують перспективність використання ML.NET для створення інтелектуальних web-додатків, що робить розроблений метод корисним у таких сферах, як автоматизація бізнес-процесів, аналіз даних у реальному часі та розробка інноваційних програмних продуктів.

Результати дослідження апробовано та опубліковано у наступних тезах доповіді на конференціях:

1. Соляник Л.О, Сєрокуров А.І. Розробка методу інтеграції систем машинного навчання в ASP.NET Core додатки з використанням ML.NET: Дослідження та реалізація методів інтеграції моделей машинного навчання у веб-додатки на базі ASP.NET Core. // II міжнародна науково-практична конференція «Сучасні аспекти діджиталізації та інформатизації в програмній та комп'ютерній інженерії» // (подано до друку).

2. Соляник Л.О, Сєрокуров А.І. Методи масштабування моделей машинного навчання в ASP.NET Core з використанням ML.NET// II міжнародна науково-практична конференція «Сучасні аспекти діджиталізації та інформатизації в програмній та комп'ютерній інженерії» // (подано до друку)

ПЕРЕЛІК ПОСИЛАНЬ

1. Bishop, C. M. Pattern Recognition and Machine Learning. Springer, 2006. – Основи машинного навчання.

2. Murphy, K. P. Machine Learning: A Probabilistic Perspective. MIT Press, 2012. – Поглиблений розгляд методів машинного навчання.

3. Goodfellow, I., Bengio, Y., & Courville, A. Deep Learning. MIT Press, 2016. – Сучасні підходи до машинного навчання, включаючи глибинне навчання.

4. Alpaydin, E. Introduction to Machine Learning. MIT Press, 2020. – Вступ до машинного навчання.

5. Flach, P. Machine Learning: The Art and Science of Algorithms that Make Sense of Data. Cambridge University Press, 2012. – Ключові концепції машинного навчання.

6. Domingos, P. The Master Algorithm: How the Quest for the Ultimate Learning Machine Will Remake Our World. Basic Books, 2015. – Популярний огляд основних парадигм машинного навчання.

7. Mohri, M., Rostamizadeh, A., & Talwalkar, A. Foundations of Machine Learning. MIT Press, 2018. – Теоретичні основи машинного навчання.

8. Microsoft Docs. ML.NET Documentation. https://learn.microsoft.com/en-us/dotnet/machine-learning/ – Офіційна документація ML.NET.

9. Microsoft Docs. ASP.NET Core Documentation. https://learn.microsoft.com/en-us/aspnet/core/ – Документація ASP.NET Core.

10. Géron, A. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow. O'Reilly Media, 2019. – Практичне впровадження моделей машинного навчання.

11. Esposito, D. Modern Web Development with ASP.NET Core 3. Packt Publishing, 2020. – Практичний посібник із розробки веб-додатків.

12. Sills, D. Pro ASP.NET Core 6. Apress, 2022. – Розширені можливості розробки веб-додатків з використанням ASP.NET Core.

13. Zhang, F. Mastering ML.NET. Packt Publishing, 2021. – Інтеграція ML.NET у веб-додатки.

13. TensorFlow Documentation. TensorFlow Guide. https://www.tensorflow.org/ – Використання TensorFlow для створення моделей.

14. ONNX Documentation. ONNX Runtime Documentation. https://onnxruntime.ai/ – Інструменти для імпорту моделей у ML.NET.

15. Microsoft Azure. Azure Machine Learning Documentation. https://learn.microsoft.com/en-us/azure/machine-learning/ – Використання Azure для хостингу моделей.

16. Amazon Web Services. AWS Machine Learning Services. https://aws.amazon.com/machine-learning/ – Хмарні сервіси для машинного навчання.

17. Google Cloud. AI and Machine Learning Products. https://cloud.google.com/products/ai – Інструменти Google Cloud для інтеграції ML.

18. Chollet, F. Deep Learning with Python. Manning Publications, 2018. – Інтеграція моделей Python у веб-додатки.

19. Raschka, S., & Mirjalili, V. Python Machine Learning. Packt Publishing, 2019. – Глибокий розгляд алгоритмів машинного навчання.

20. Bonaccorso, G. Machine Learning Algorithms. Packt Publishing, 2018. – Огляд алгоритмів машинного навчання та їх практичне застосування.

21. Ivy, J. Applied Machine Learning for Developers. Apress, 2021. – Практичне впровадження ML для веб-додатків.

22. IBM. What Is Linear Regression? | IBM. IBM - United States. URL: https://www.ibm.com/topics/linear-regression

23. Calculus I - The Mean Value Theorem. Pauls Online Math Notes. URL: https://tutorial.math.lamar.edu/classes/calci/MeanValueTheorem.aspx

24. Khan Academy. Khan Academy. URL: https://www.khanacademy.org/math/ap-calculus-ab/ab-diff-analytical-applications-new/ab-5-1/v/mean-value-theorem-1#:~:text=The%20Mean%20Value%20Theorem%20states,over%20[a,b].

25. What is a Neural Network? - Artificial Neural Network Explained - AWS. Amazon Web Services, Inc. URL: https://aws.amazon.com/what-is/neural-network/.

26. What is a Neural Network? - GeeksforGeeks. GeeksforGeeks. URL: https://www.geeksforgeeks.org/neural-networks-a-beginners-guide/

27. GradientBoostingClassifier. scikit-learn. URL: https://scikit-learn.org/1.5/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html.

28. Довідник по Machine Learning – Gradient boosting | База знань IT. База знань IT технологій. URL: https://itwiki.dev/data-science/ml-reference/ml-glossary/gradient-boosting.

29. Why use ML.NET?. Matt on ML.NET. URL: https://mattonml.net/post/why_mlnet/.

30. Codecademy. MVC: Model, View, Controller | Codecademy. Codecademy. URL: https://www.codecademy.com/article/mvc.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Магістерська робота

Методи інтеграції машинного навчання у web-додатки на базі ASP.NET Core з використанням ML.NET

Виконав: студент групи ПДМ-62 Сєрокуров Артем Ігорович

Керівник: к.х.н., доц., доцент кафедри ІПЗ Соляник Людмила Олексіївна

Київ - 2025

МЕТА, ОБ'ЄКТА ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи – підвищення ефективності інтеграції машинного навчання у web-додатки, створені на базі ASP.NET Core, із застосуванням ML.NET.

Об'єкт дослідження – процес інтеграції моделей машинного навчання у web-додатки.

Предмет дослідження – метод інтеграції машинного навчання у web-додатки на базі ASP.NET Core з використанням ML.NET.

ПОРІВНЯННЯ МЕТОДІВ

Метод	Плюси	Мінуси
SDCA (Stochastic Dual Coordinate Ascent)	Швидке навчання та передбачення. Ефективний для великих наборів даних. Підтримує задачі регресії та класифікації.	Чутливий до налаштування гіперпараметрів. Обмежений для дуже складних залежностей.
Лінійна регресія	Простота реалізації. Легка інтерпретація результатів. Низькі обчислювальні витрати.	Не підходить для нелінійних залежностей. Низька точність у складних задачах.
Метод середнього значення	Найшвидший і найпростіший метод. Не потребує навчання.	Ігнорує взаємозв'язки між ознаками. Дуже низька точність прогнозів.
Нейронні мережі	Висока точність для складних залежностей. Можливість роботи з великими обсягами даних.	Високі ресурсоємність та час навчання. Складність у налаштуванні та оптимізації.
Градієнтний бустинг	Висока точність для табличних даних. Добре працює з нелінійними залежностями.	Потребує більше часу для навчання, ніж SDCA. Більш ресурсоємний.
Хмарні технології	Швидка обробка великих даних, автоматичне масштабування, висока доступність ресурсів.	Висока залежність від інтернет-з'єднання. Можливі додаткові витрати на використання хмарних сервісів.

АЛГОРИТМ SDCA

Задача оптимізації лінійної моделі:

$$\min_w \frac{\lambda}{2}\|w\|^2 + \frac{1}{n}\sum_{i=1}^{n}\ell_i(w^\top x_i),$$

де:

- $w \in \mathbb{R}^d$ — вектор ваг моделі,
- $\lambda > 0$ — коефіцієнт регуляризації (L2),
- $x_i \in \mathbb{R}^d$ — вектор ознак i-го зразка,
- $\ell_i(\cdot)$ — функція втрат (наприклад, логістична або квадратична).

Подвійна задача оптимізації

$$\max(\frac{1}{2\lambda}\|\alpha\|^2 - \frac{1}{n}\sum_{i=1}^{n}\ell*(a_i))$$

Де:
α — вектор подвійних змінних (коефіцієнтів Лагранжа),
ℓ*(αi) — подвійна функція втрат, що є кон'югованою функцією до ℓ(xi,yi,w)ℓ(xi, yi, w)ℓ(xi,yi,w).

Оновлення подвійної змінної:

$$a_i^{new} = a_i^{old} - \eta * \nabla_{a_i}(\frac{1}{2\lambda}\|\alpha\|^2 - \frac{1}{n}\sum_{i=1}^{n}\ell*(a_i))$$

де :
η — швидкість навчання
∇ — градієнт подвійної функції втрат.

Оновлення ваг моделі:

$$w = \sum_{i=1}^{n} a_i x_i .$$

Блок схема алгоритму SDCA

5

МАТЕМАТИЧНА МОДЕЛЬ МЕТРИК ОЦІНЮВАННЯ ПОРІВНЯННЯ МОДЕЛЕЙ

1. **Середня абсолютна похибка (MAE):**
   MAE вимірює середню абсолютну різницю між передбаченими ($\hat{y}_i$) та фактичними (y_i) значеннями.
   Формула:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

2. **Коефіцієнт детермінації (R^2):**
   R^2 оцінює, наскільки добре модель пояснює варіацію цільової змінної.
   Формула:

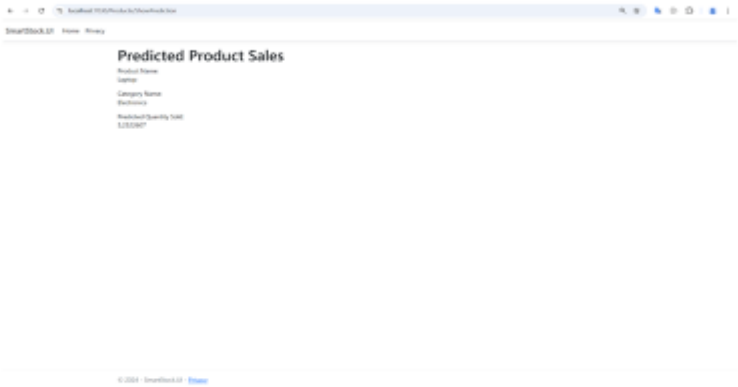
$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

3. **Середньоквадратична похибка (RMSE):**
   RMSE є коренем середнього значення квадратів різниць між передбаченими та реальними значеннями.
   Формула:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

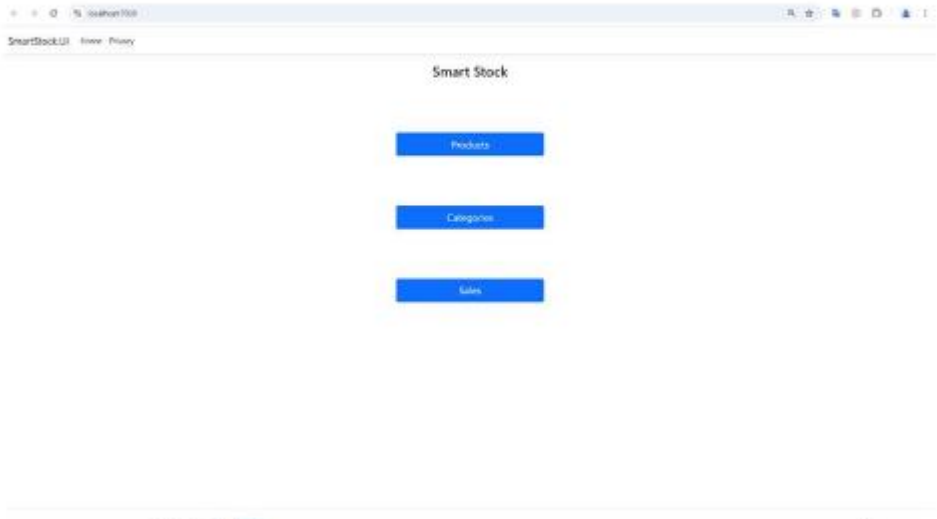
6

ПРАКТИЧНИЙ РЕЗУЛЬТАТ



7

ДЕМОНСТРАЦІЯ РЕЗУЛЬТАТУ



8

ПОРІВНЯЛЬНИЙ АНАЛІЗ

Метод	Швидкість передбачення (сек)	MAE	R^2	RMSE
SDCA	0.75	9.6	0.87	11.3
Лінійна регресія	0.5	11.1	0.78	15.3
Метод середнього значення	0.1	15.3	0.62	19.5
Нейронні мережі	2.5	7.6	0.89	10.8
Градієнтний бустинг	1.5	8.9	0.86	11.9
Хмарні технології	1.25	10.3	0.75	13.0

9

ВИСНОВКИ

1. Було використано сучасні методи, включаючи машинне навчання, математичне моделювання, об'єктно-орієнтоване програмування та розробку REST API, що забезпечило комплексний підхід до вирішення поставлених завдань.
2. Проведено аналіз методів машинного навчання (регресія, класифікація, кластеризація) та глибоке вивчення архітектури ML.NET, що дало змогу адаптувати платформу до конкретних завдань інтеграції моделей у web-додатки.
3. Розроблено ефективний метод інтеграції моделей машинного навчання в web-додатки, який охоплює етапи збору та підготовки даних, тренування, серіалізації та впровадження моделей.
4. Підвищено продуктивність та точность методу, застосовано сучасні підходи до оптимізації, зокрема автоматизований пошук параметрів, що забезпечило їх адаптацію до реальних умов експлуатації.

10

ПУБЛІКАЦІЇ ТА АПРОБАЦІЯ РОБОТИ

Тези доповідей:

1. Соляник Л.О, Сєрокуров А.І. Розробка методу інтеграції систем машинного навчання в ASP.NET Core додатки з використанням ML.NET: Дослідження та реалізація методів інтеграції моделей машинного навчання у веб-додатки на базі ASP.NET Core. // II міжнародна науково-практична конференція «Сучасні аспекти діджиталізації та інформатизації в програмній та комп'ютерній інженерії» // подано до друку.

2. Соляник Л.О, Сєрокуров А.І. Методи масштабування моделей машинного навчання в ASP.NET Core з використанням ML.NET// II міжнародна науково-практична конференція «Сучасні аспекти діджиталізації та інформатизації в програмній та комп'ютерній інженерії» // подано до друку.

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

```
public class DemandPredictionModel
{
    private readonly MLContext _mlContext;
    private ITransformer _model;

    public DemandPredictionModel()
    {
        _mlContext = new MLContext();
    }

    public void TrainModel(List<ProductSalesData>
salesData)
    {
        var trainingData =
_mlContext.Data.LoadFromEnumerable(salesData)
;

        var pipeline =
_mlContext.Transforms.Concatenate("Features",
nameof(ProductSalesData.Day),
nameof(ProductSalesData.Month),
nameof(ProductSalesData.PreviousQuantitySold))

.Append(_mlContext.Regression.Trainers.Sdca(lab
elColumnName:
nameof(ProductSalesData.QuantitySold),
maximumNumberOfIterations: 100));

        var previewData = trainingData.Preview();

        _model = pipeline.Fit(trainingData);
    }

    public float PredictDemand(ProductSalesData
newProductSaleData)
    {
        var predictionEngine =
_mlContext.Model.CreatePredictionEngine<Produc
tSalesData, ProductSalesPrediction>(_model);
        var prediction =
predictionEngine.Predict(newProductSaleData);
        return prediction.PredictedQuantitySold;
    }
}
```

```
using Microsoft.AspNetCore.Mvc;
using SmartStock.UI.Models;
using System.Diagnostics;

namespace SmartStock.UI.Controllers
{
    public class HomeController : Controller
    {
        private readonly ILogger<HomeController>
_logger;

        public
HomeController(ILogger<HomeController> logger)
        {
            _logger = logger;
        }

        public IActionResult Index()
        {
            return View();
        }

        public IActionResult Privacy()
        {
            return View();
```

```
        }


        [ResponseCache(Duration = 0, Location =
ResponseCacheLocation.None, NoStore = true)]
        public IActionResult Error()
        {
            return View(new ErrorViewModel {
RequestId = Activity.Current?.Id ??
HttpContext.TraceIdentifier });
        }
    }
}


using AutoMapper;
using Microsoft.AspNetCore.Mvc;
using Microsoft.IdentityModel.Tokens;
using SmartStock.BLL.Dtos;
using SmartStock.BLL.Services.Interfaces;
using SmartStock.DAL.Entities;
using SmartStock.UI.Models;
using SmartStock.Utils;
using SmartStock.Utils.Models;


namespace SmartStock.UI.Controllers
{
    public class ProductsController : Controller
    {
        private readonly
ISmartStockService<ProductDto, Product>
_productService;
        private readonly ISmartStockService<SaleDto,
Sale> _saleService;
        private readonly
ISmartStockService<CategoryDto, Category>
_categoryService;
        private readonly IMapper _mapper;

        public
ProductsController(ISmartStockService<ProductDt
o, Product> context,
            ISmartStockService<SaleDto, Sale>
saleContext,
            ISmartStockService<CategoryDto,
Category> categoryContext,
            IMapper mapper)
        {
            _productService = context;
            _saleService = saleContext;
            _categoryService = categoryContext;
            _mapper = mapper;
        }


        // GET: Products
        public async Task<IActionResult> Index()
        {
            var productDtos =
_productService.GetAll();


            if (productDtos.IsNullOrEmpty())
            {
                Problem("Entity sed 'Product' is null");
            }


            var products =
_mapper.Map<IEnumerable<ProductVM>>(produc
tDtos);
            return View(products);
        }


        // GET: Products/Details/5
        public async Task<IActionResult> Details(int?
id)
        {
            var productDtos =
_productService.GetAll();
```

```
        if (id == null || productDtos.IsNullOrEmpty())
        {
            return NotFound();
        }

        var productDto = productDtos
                        .FirstOrDefault(m => m.ProductId == id);
        if (productDto == null)
        {
            return NotFound();
        }

        var client = _mapper.Map<ProductVM>(productDto);
        return View(client);
    }

    // GET: Products/Create
    public IActionResult Create()
    {
        // ViewData["CategoryId"] = new SelectList(_context.Categories, "CategoryId", "CategoryName");
        return View();
    }

    // POST: Products/Create
    // To protect from overposting attacks, enable the specific properties you want to bind to.
    // For more details, see http://go.microsoft.com/fwlink/?LinkId=317598.
    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Create([Bind("ProductId,Name,CategoryId,Price")] ProductVM product)
    {
        var productDto = _mapper.Map<ProductDto>(product);
        _productService.Add(productDto);
        return RedirectToAction(nameof(Index));
    }

    // GET: Products/Edit/5
    public async Task<IActionResult> Edit(int? id)
    {
        var productDtos = _productService.GetAll().ToList();
        if (id == null || productDtos.IsNullOrEmpty())
        {
            return NotFound();
        }

        var productDto = productDtos.Find(x => x.ProductId == id);
        if (productDto == null)
        {
            return NotFound();
        }

        var product = _mapper.Map<ProductVM>(productDto);
        return View(product);
    }

    // POST: Products/Edit/5
    // To protect from overposting attacks, enable the specific properties you want to bind to.
    // For more details, see http://go.microsoft.com/fwlink/?LinkId=317598.
    [HttpPost]
    [ValidateAntiForgeryToken]
```

```
public async Task<IActionResult> Edit(int id,
[Bind("ProductId,Name,CategoryId,Price")]
Product product)
{
    if (id != product.ProductId)
    {
        return NotFound();
    }

var productDto =
_mapper.Map<ProductDto>(product);
        _productService.Update(productDto);
        return RedirectToAction(nameof(Index));
}

// GET: Products/Delete/5
public async Task<IActionResult> Delete(int?
id)
{
    var productDtos =
_productService.GetAll().ToList();
    if (id == null ||
productDtos.IsNullOrEmpty())
    {
        return NotFound();
    }

    var productDto =
productDtos.FirstOrDefault(m => m.ProductId ==
id);
    if (productDto == null)
    {
        return NotFound();
    }

    var product =
_mapper.Map<ProductVM>(productDto);
    return View(product);
}

// POST: Products/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult>
DeleteConfirmed(int id)
{
    var productDtos =
_productService.GetAll().ToList();
    if (productDtos.IsNullOrEmpty())
    {
        return Problem("Entity set 'Products'  is
null.");
    }
    var productDto = productDtos.Find(x =>
x.ProductId == id);
    if (productDto != null)
    {
        _productService.Delete(productDto);
    }

    return RedirectToAction(nameof(Index));
}

public IActionResult ShowPrediction(int
productId, int QuantitySold)
{
    var predictModel = new
DemandPredictionModel();

    var product =
_productService.GetAll().ToList().Where(x =>
x.ProductId == productId).First();

    var category =
_categoryService.GetById(product.CategoryId);
```

```
        var sales =
_saleService.GetAll().ToList().Where(x =>
x.ProductId == productId);


        var productsalesDatas = new
List<ProductSalesData>();
        foreach (var sale in sales)
        {
            var predictedProduct = new
ProductSalesData
            {
                Day = sale.SaleDate.Day,
                Month = sale.SaleDate.Month,
                PreviousQuantitySold =
sale.QuantitySold,
                QuantitySold = sale.QuantitySold
            };



productsalesDatas.Add(predictedProduct);
        }



predictModel.TrainModel(productsalesDatas);


        var productSalesData = new
ProductSalesData();


        productSalesData.Day =
DateTime.Now.Day;
        productSalesData.Month =
DateTime.Now.Month;
        productSalesData.PreviousQuantitySold =
sales.Last().QuantitySold;
        productSalesData.QuantitySold =
QuantitySold;


        var predictedQuantitySold =
predictModel.PredictDemand(productSalesData);


        var predictedProductVM = new
PredictedProductVM()
        {
            ProductName = product.Name,
            CategoryName =
category.CategoryName,
            PredictedQuantitySold =
predictedQuantitySold,
        };


        return View(predictedProductVM);
    }
  }
}


using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using AutoMapper;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using Microsoft.IdentityModel.Tokens;
using SmartStock.BLL.Dtos;
using SmartStock.BLL.Services.Interfaces;
using SmartStock.DAL.Data;
using SmartStock.DAL.Entities;
using SmartStock.UI.Models;


namespace SmartStock.UI.Controllers
{
  public class SalesController : Controller
  {
    private readonly ISmartStockService<SaleDto,
Sale> _saleService;
```

```
        private readonly
ISmartStockService<ProductDto, Product>
_productService;

        private readonly IMapper _mapper;


        public
SalesController(ISmartStockService<SaleDto,
Sale> context, ISmartStockService<ProductDto,
Product> productContext, IMapper mapper)
        {
            _saleService = context;
            _productService = productContext;
            _mapper = mapper;
        }


        // GET: Sales
        public async Task<IActionResult> Index()
        {
            var saleDtos = _saleService.GetAll();


            if (saleDtos.IsNullOrEmpty())
            {
                Problem("Entity sed 'Sale' is null");
            }


            foreach (var sale in saleDtos)
            {
                sale.ProductName =
_productService.GetById(sale.ProductId).Name;
            }


            var sales =
_mapper.Map<IEnumerable<SaleVM>>(saleDtos);
            return View(sales);
        }


        // GET: Sales/Details/5
        public async Task<IActionResult> Details(int?
id)
        {
            var saleDtos = _saleService.GetAll();
            if (id == null || saleDtos.IsNullOrEmpty())
            {
                return NotFound();
            }


            foreach (var s in saleDtos)
            {
                s.ProductName =
_productService.GetById(s.ProductId).Name;
            }


            var saleDto = saleDtos
                        .FirstOrDefault(m => m.SaleId
== id);
            if (saleDto == null)
            {
                return NotFound();
            }


            var sale =
_mapper.Map<SaleVM>(saleDto);
            return View(sale);
        }


        // GET: Sales/Create
        public IActionResult Create()
        {
            return View();
        }


        // POST: Sales/Create
        // To protect from overposting attacks, enable
the specific properties you want to bind to.
```

```
        // For more details, see
http://go.microsoft.com/fwlink/?LinkId=317598.
        [HttpPost]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult>
Create([Bind("ProductId,SaleDate,QuantitySold")]
Sale sale)
        {
            var saleDto =
_mapper.Map<SaleDto>(sale);
            _saleService.Add(saleDto);
            return RedirectToAction(nameof(Index));
        }

        // GET: Sales/Edit/5
        public async Task<IActionResult> Edit(int?
id)
        {
            var saleDtos =
_saleService.GetAll().ToList();
            if (id == null || saleDtos.IsNullOrEmpty())
            {
                return NotFound();
            }

            foreach (var s in saleDtos)
            {
                s.ProductName =
_productService.GetById(s.ProductId).Name;
            }

            var saleDto = saleDtos.Find(x => x.SaleId
== id);
            if (saleDto == null)
            {
                return NotFound();
            }
            var sale =
_mapper.Map<SaleVM>(saleDto);
            return View(sale);
        }

        // POST: Sales/Edit/5
        // To protect from overposting attacks, enable
the specific properties you want to bind to.
        // For more details, see
http://go.microsoft.com/fwlink/?LinkId=317598.
        [HttpPost]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult> Edit(int id,
[Bind("SaleId,ProductId,SaleDate,QuantitySold")]
Sale sale)
        {
            if (id != sale.SaleId)
            {
                return NotFound();
            }

var saleDto = _mapper.Map<SaleDto>(sale);
            _saleService.Update(saleDto);
            return RedirectToAction(nameof(Index));
        }

        // GET: Sales/Delete/5
        public async Task<IActionResult> Delete(int?
id)
        {
            var saleDtos =
_saleService.GetAll().ToList();
            if (id == null || saleDtos.IsNullOrEmpty())
            {
                return NotFound();
            }
```

```
            var saleDto = saleDtos.FirstOrDefault(m =>
m.SaleId == id);
            if (saleDto == null)
            {
                return NotFound();
            }

            var sale =
_mapper.Map<SaleVM>(saleDto);
            return View(sale);
        }

        // POST: Sales/Delete/5
        [HttpPost, ActionName("Delete")]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult>
DeleteConfirmed(int id)
        {
            var saleDtos =
_saleService.GetAll().ToList();
            if (saleDtos.IsNullOrEmpty())
            {
                return Problem("Entity set 'Products'  is
null.");
            }
            var saleDto = saleDtos.Find(x => x.SaleId
== id);
            if (saleDto != null)
            {
                _saleService.Delete(saleDto);
            }

            return RedirectToAction(nameof(Index));
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using AutoMapper;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using Microsoft.IdentityModel.Tokens;
using SmartStock.BLL.Dtos;
using SmartStock.BLL.Services.Interfaces;
using SmartStock.DAL.Data;
using SmartStock.DAL.Entities;
using SmartStock.UI.Models;

namespace SmartStock.UI.Controllers
{
    public class CategoriesController : Controller
    {
        private readonly
ISmartStockService<CategoryDto, Category>
_categoryService;
        private readonly IMapper _mapper;

        public
CategoriesController(ISmartStockService<Categor
yDto, Category> context, IMapper mapper)
        {
            _categoryService = context;
            _mapper = mapper;
        }

        // GET: Categories
        public async Task<IActionResult> Index()
        {
            var categoryDtos =
_categoryService.GetAll();
```

```
        if (categoryDtos.IsNullOrEmpty())
        {
            Problem("Entity sed 'Product' is null");
        }

        var categories =
_mapper.Map<IEnumerable<CategoryVM>>(categ
oryDtos);
        return View(categories);
    }

    // GET: Categories/Details/5
    public async Task<IActionResult> Details(int?
id)
    {
        var categoryDtos =
_categoryService.GetAll();
        if (id == null ||
categoryDtos.IsNullOrEmpty())
        {
            return NotFound();
        }

        var categoryDto = categoryDtos
                    .FirstOrDefault(m =>
m.CategoryId == id);
        if (categoryDto == null)
        {
            return NotFound();
        }

        var category =
_mapper.Map<CategoryVM>(categoryDto);
        return View(category);
    }

    // GET: Categories/Create
    public IActionResult Create()
    {
        return View();
    }

    // POST: Categories/Create
    // To protect from overposting attacks, enable
the specific properties you want to bind to.
    // For more details, see
http://go.microsoft.com/fwlink/?LinkId=317598.
    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult>
Create([Bind("CategoryId,CategoryName")]
Category category)
    {

        var categoryDto =
_mapper.Map<CategoryDto>(category);
        _categoryService.Add(categoryDto);
        return RedirectToAction(nameof(Index));
    }

    // GET: Categories/Edit/5
    public async Task<IActionResult> Edit(int?
id)
    {
        var categoryDtos =
_categoryService.GetAll().ToList();
        if (id == null ||
categoryDtos.IsNullOrEmpty())
        {
            return NotFound();
        }

        var categoryDto = categoryDtos.Find(x =>
x.CategoryId == id);
```

```
            if (categoryDto == null)
            {
                return NotFound();
            }

            var category = _mapper.Map<CategoryVM>(categoryDto);
            return View(category);
        }

        // POST: Categories/Edit/5
        // To protect from overposting attacks, enable the specific properties you want to bind to.
        // For more details, see http://go.microsoft.com/fwlink/?LinkId=317598.
        [HttpPost]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult> Edit(int id, [Bind("CategoryId,CategoryName")] Category category)
        {
            if (id != category.CategoryId)
            {
                return NotFound();
            }
                var categoryDto = _mapper.Map<CategoryDto>(category);
                _categoryService.Update(categoryDto);
                return RedirectToAction(nameof(Index));
        }
// GET: Categories/Delete/5
        public async Task<IActionResult> Delete(int? id)
        {
            var categyDtos = _categoryService.GetAll().ToList();
            if (id == null || categyDtos.IsNullOrEmpty())
            {
                return NotFound();
            }
            var categoryDto = categyDtos.FirstOrDefault(m => m.CategoryId == id);
            if (categoryDto == null)
            {
                return NotFound();
            }
            var category = _mapper.Map<CategoryVM>(categoryDto);
            return View(category);
        }
        // POST: Categories/Delete/5
        [HttpPost, ActionName("Delete")]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult> DeleteConfirmed(int id)
        {
            var categoryDtos = _categoryService.GetAll().ToList();
            if (categoryDtos.IsNullOrEmpty())
            {
                return Problem("Entity set 'Products'  is null.");
            }
            var categoryDto = categoryDtos.Find(x => x.CategoryId == id);
            if (categoryDto != null)
            {
                _categoryService.Delete(categoryDto);
            }
            return RedirectToAction(nameof(Index));
}
```