

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «магістр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)
на тему:

**«Генерація тестових даних за допомогою штучного
інтелекту для Restful API»**

Виконав студент групи ЗПЗм-22-1
ВОЛЬНИХ Ірина Вікторівна

Керівник роботи:
КОНЦИДАЙЛО Андрій Михайлович

Рецензент:
ЄФРЕМОВ Юрій Миколайович

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

зав. кафедри інженерії
програмного забезпечення

Тетяна ВАКАЛЮК

«18» вересня 2023 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: ВОЛЬНИХ Ірина Вікторівна

Керівник роботи: КОНЦИДАЙЛО Андрій Михайлович

Тема роботи: «Генерація тестових даних за допомогою штучного інтелекту для Restful API», затверджена наказом закладу вищої освіти від **18.09.2023 р. № 481/с**

Вихідні дані для роботи: набір даних користувачів, отримані з відкритих джерел; методи генерації, обробка та аналіз текстових.

Консультанти з магістерської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Концидайло А.М.	22.09.2023р.	22.09.2023р.
2	Концидайло А.М.	22.09.2023р.	22.09.2023р.
3	Концидайло А.М.	22.09.2023р.	22.09.2023р.

РЕФЕРАТ

Магістерська атестаційна робота, що представлена на розгляд, присвячена темі «Генерація тестових даних за допомогою штучного інтелекту для Restful API». Робота включає вступ, три основні розділи, висновки, список використаних джерел із 33 найменувань, 6 рисунків та 23 лістингів. Загальний обсяг роботи складає 64 сторінки, з яких 49 припадає на основний текст.

Магістерська робота присвячена вивченню методів використання штучного інтелекту для генерації тестових даних, з особливим акцентом на генерацію даних для RESTful API. Робота складається з вступу, чотирьох основних розділів, висновків, а також включає 20 рисунків, 10 лістингів коду, і охоплює 62 сторінки з бібліографічним списком, що містить 30 найменувань.

Основна мета дослідження полягає у розробці та аналізі стратегій застосування моделей штучного інтелекту, зокрема GPT, для ефективної генерації даних. У роботі проведено порівняльний аналіз цих методів із традиційними підходами до тестування, оцінено їх ефективність та потенціал застосування. Отримані результати включають розробку методів легкої інтерпретації тестових даних та виявлення важливих ознак, що є ключовими для генерації якісних тестових наборів.

Робота може бути використана для покращення процесів тестування в розробці програмного забезпечення, зокрема у сфері веб-сервісів та API. Рекомендації включають застосування розроблених методів для автоматизації тестування та підвищення ефективності процесу перевірки функціональності веб-сервісів.

Ключові слова, що описують зміст роботи: Тестові дані, генерація даних, RNN, GPT, тестування програмного забезпечення, автоматизоване тестування, штучний інтелект.

ABSTRACT

The Master's thesis presented for review is dedicated to the topic "Generating Test Data Using Artificial Intelligence for Restful API." The thesis comprises an introduction, three main sections, conclusions, a list of 33 references, 6 figures, and 23 listings. The total length of the thesis is 64 pages, with 49 pages dedicated to the main text.

The Master's thesis focuses on studying the methods of utilizing artificial intelligence for generating test data, with a special emphasis on data generation for RESTful APIs. The work consists of an introduction, four main sections, conclusions, and includes 20 figures and 10 code listings, covering 62 pages with a bibliography containing 30 references.

The main objective of the research is the development and analysis of strategies for applying artificial intelligence models, particularly GPT, for effective data generation. The thesis conducts a comparative analysis of these methods with traditional testing approaches, evaluates their effectiveness, and assesses their potential applications. The obtained results include the development of methods for easy interpretation of test data and the identification of important features crucial for generating high-quality test datasets.

The work can be used to enhance testing processes in software development, especially in the field of web services and APIs. Recommendations include the application of the developed methods for test automation and improving the efficiency of functionality verification in web services.

Keywords describing the content of the work: Test Data, Data Generation, RNN, GPT, Software Testing, Automated Testing, Artificial Intelligence.

ЗМІСТ

РЕФЕРАТ.....	3
ЗМІСТ.....	5
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ.....	7
ВСТУП.....	8
РОЗДІЛ 1. ДОСЛІДЖЕННЯ МЕТОДІВ ВИКОРИСТАННЯ ШТУЧНОГО ІНТЕЛЕКТУ ДЛЯ ГЕНЕРАЦІЇ ТЕСТОВИХ ДАНИХ.....	11
1.1 Генерація тестових даних для Restful api: переваги штучного інтелекту над ручним тестуванням.....	11
1.2 Породжувальний, попередньо тренований трансформер у процесах тестування.....	12
1.3 Методи генерації тестових даних за допомогою рекурентної нейронної мережі (RNN) та аналіз потенціалу RNN.....	15
1.4 Архітектура RNN, принципи роботи та процес навчання RNN.....	16
Висновки до розділу 1.....	17
РОЗДІЛ 2. РОЗРОБКА МЕТОДІВ ГЕНЕРАЦІЇ ТЕСТОВИХ ДАНИХ З ВИКОРИСТАННЯМ ШІ.....	19
2.1 Генерація тестових даних за допомогою API чат-бота породжувального, попередньо тренованого трансформера.....	19
2.2 Генерація масиву даних за допомогою API GPT-3.5-turbo.....	26
2.3 Реалізація та навчання рекурентної нейронної мережі (RNN) для генерації синтетичних тестових даних.....	28
2.4 Використання мапінгу для перетворення типів даних та JSON-схеми для маркування тестових сценаріїв.....	33
2.5 Використання JSON-схеми для маркування тестових сценаріїв.....	34
Висновки до розділу 2.....	39
РОЗДІЛ 3. РЕАЛІЗАЦІЯ МЕТОДІВ ГЕНЕРАЦІЇ ТЕСТОВИХ ДАНИХ З	

ВИКОРИСТАННЯМ ШТУЧНОГО ІНТЕЛЕКТУ.....	41
3.1 Використання згенерованих даних за допомогою GPT-3.5 Turbo у автоматизованому тестуванні.....	41
3.2 Використання згенерованих даних за допомогою RNN у автоматизованому тестуванні.....	46
3.3 Порівняння можливостей RNN та GTP з GAN (Generative Adversarial Networks).....	56
Висновки до розділу 3.....	57
ВИСНОВКИ.....	58
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	61

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API – Application Programming Interface (Інтерфейс програмування застосунків)

GAN – Generative Adversarial Networks (Конфліктні Генеративні Мережі)

GPT – Generative Pre-trained Transformer (Породжувальний Попередньо Тренований Трансформер)

GPT-turbo 3.5 – Enhanced version of the Generative Pre-trained Transformer

JSON – JavaScript Object Notation (Нотація Об'єктів JavaScript)

RNN – Recurrent Neural Network (Рекурентна Нейронна Мережа)

ШІ – Штучний Інтелект

ВСТУП

Актуальність роботи визначається потребою у покращенні методів автоматизації тестування, особливо з використанням штучного інтелекту для генерації тестових даних для RESTful API. Інтелектуальні методи, які розглядаються у дослідженні, забезпечують не тільки ефективність у створенні реалістичних тестових сценаріїв, але й підвищують загальну продуктивність та точність процесів тестування.

У сучасному науковому та технологічному контексті, де роль та складність веб-сервісів та API постійно зростають, важливість цієї теми є особливо актуальною. Розвиток методів для генерації синтетичних даних, що імітують реальні сценарії використання, є ключовим для забезпечення високої якості та безпеки програмного забезпечення. Такі методи сприяють більш ефективному виявленню помилок і допомагають уникати потенційних вразливостей у програмах.

Мета роботи полягає в аналізі та розробці методів використання штучного інтелекту, включаючи породжувальні попередньо треновані трансформери та рекурентні нейронні мережі (RNN), для ефективної генерації тестових даних. Основний акцент роботи зосереджений на покращенні процесів тестування, зокрема у контексті RESTful API.

Завдання роботи. На основі мети роботи завдання можна сформулювати таким чином:

1. Розробка та впровадження процесу генерації тестових даних для автоматизованого та мануального тестування RESTful API з використанням штучного інтелекту.
2. Аналіз потенціалу рекурентних нейронних мереж (RNN) у контексті генерації тестових даних.
3. Впровадження згенерованих даних у реальні сценарії тестування для оцінки їхньої релевантності та ефективності.

4. Оцінка впливу використання штучного інтелекту на процеси тестування, включаючи аналіз переваг та обмежень цього підходу.

Поставлені завдання спрямовані на досягнення основної мети роботи – розробці методів використання штучного інтелекту, включаючи породжувальні попередньо треновані трансформери та рекурентні нейронні мережі (RNN), для ефективної генерації тестових даних. і їх застосування у процесах тестування.

Об'єктом дослідження у магістерській роботі є процес генерації тестових даних для RESTful API, особливо з акцентом на використання методів штучного інтелекту. Цей об'єкт вибрано через його значущість у контексті швидкого розвитку інформаційних технологій та зростаючої ролі RESTful API в області веб-розробки.

Предмет дослідження є застосування методів штучного інтелекту, зокрема попередньо тренованих трансформерних моделей як Generative Pre-trained Transformer (GPT) та рекурентної нейронної мережі (RNN) для генерації тестових даних, специфічно для використання у тестуванні RESTful API.

Методи дослідження у магістерській роботі включають: використання штучного інтелекту, зокрема рекурентних нейронних мереж (RNN) та породжувальних попередньо тренованих трансформерів.

Інформаційну базу дослідження магістерської роботи складають наукові публікації, статті, та дослідження, пов'язані з машинним навчанням та тестуванням програмного забезпечення. Для створення синтетичних даних використовуються інструменти з бібліотеки Gretel Synthetics, яка містить набори інструментів для генерації даних за допомогою рекурентних нейронних мереж (RNN).

Наукова новизна роботи полягає у побудові та оптимізації моделі рекурентної нейронної мережі (RNN) для генерації синтетичних тестових даних, узагальненні та удосконаленні методів генерації даних з використанням технологій штучного інтелекту, модифікації підходів до автоматизації тестування RESTful API, та розробці інноваційних алгоритмів, що зменшують

час та ресурси процесів тестування. Ці результати вносять вагомий вклад у практику тестування програмного забезпечення.

Практичне значення отриманих результатів магістерської роботи полягає у значному покращенні та оптимізації процесів тестування RESTful API. Це досягається через розробку методів для генерації синтетичних тестових даних, які дозволяють виконувати більш реалістичне тестування зовнішніх систем. Використання таких методів сприяє виявленню та усуненню потенційних вразливостей і помилок у програмному забезпеченні, забезпечуючи високу якість та надійність веб-сервісів.

Публікації. За темою атестаційної магістерської роботи було опубліковано тези [1].

РОЗДІЛ 1. ДОСЛІДЖЕННЯ МЕТОДІВ ВИКОРИСТАННЯ ШТУЧНОГО ІНТЕЛЕКТУ ДЛЯ ГЕНЕРАЦІЇ ТЕСТОВИХ ДАНИХ

1.1 Генерація тестових даних для Restful api: переваги штучного інтелекту над ручним тестуванням

Дослідження методів генерації даних за допомогою штучного інтелекту (ШІ) для RESTful API представляє науковий інтерес у контексті швидкого розвитку інформаційних технологій та зростаючої важливості RESTful API у сучасній веб-розробці. Актуальність цього дослідження визначається декількома науковими аспектами.

Наукове обґрунтування використання штучного інтелекту у генерації тестових даних для RESTful API базується на потребі покращення автоматизації тестування. Інтелектуальні методи можуть забезпечити не тільки ефективність у створенні реалістичних тестових сценаріїв, але й підвищити рівень покриття коду, забезпечуючи високу якість програмного продукту [2]. Дослідження спрямоване на розробку та оптимізацію процесу розробки RESTful API, де використання ШІ для генерації тестових даних виступає як науковий підхід до підвищення продуктивності та швидкості виявлення та усунення помилок.

Вивчення обмежень та особливостей методів генерації тестових даних науково обґрунтовано в контексті уникнення артефактів тестування та покращення реалістичності тестових сценаріїв. Це може включати дослідження впливу різних факторів на результати тестування та розробку стратегій підвищення надійності методів генерації даних.

Також, обґрунтування використання ШІ у забезпеченні безпеки RESTful API базується на потребі виявлення та вирішення потенційних вразливостей за допомогою реалістичних тестових сценаріїв [3]. Це може включати створення та аналіз тестових випадків для виявлення помилок безпеки та розробку стратегій запобігання їх появі в реальних умовах експлуатації API.

Таким чином, науковий підхід до дослідження методів генерації тестових даних для RESTful API полягає в ретельному вивченні та визначенні

ефективних стратегій, які базуються на сучасних технологіях штучного інтелекту та враховують вимоги сучасного програмного забезпечення.

У порівнянні з ручним тестуванням, використання ШІ для генерації тестових даних дозволяє значно підвищити швидкість та ефективність тестування. Інтелектуальні методи можуть автоматизувати та прискорити процес створення реалістичних сценаріїв, у той час як ручне тестування вимагає великої кількості робочого часу та зусиль.

Дослідження ШІ також дозволяє враховувати більшу кількість варіацій та умов для тестування, в порівнянні з обмеженими можливостями ручного тестування. Це дозволяє ефективніше покривати широкий спектр можливих сценаріїв використання та забезпечує більш глибокий аналіз надійності та безпеки RESTful API.

Крім того, результати наукового дослідження можуть включати математичні та статистичні обґрунтування ефективності та надійності використання ШІ. Це дозволяє зробити об'єктивні порівняння з ручним тестуванням на основі конкретних метрик та результатів експериментів [4].

У підсумку, наше дослідження визначається важливістю заміщення ручного тестування інтелектуальними методами генерації тестових даних для RESTful API. Це дозволяє підвищити продуктивність, покращити якість та забезпечити більш широкий охоплення можливих сценаріїв використання, що робить наше дослідження значущим для сучасної практики тестування програмного забезпечення.

1.2 Породжувальний, попередньо тренований трансформер у процесах тестування

Генерація тестових даних на основі попередньо тренованого трансформера GPT (Generative Pre-trained Transformer) є актуальною та інноваційною галуззю в області штучного інтелекту та обробки природної мови (NLP) [5]. У цій главі будуть розглянуті теоретичні аспекти цього процесу, включаючи основні концепції, методи та потенційні застосування.

Трансформер GPT є однією з найважливіших досягнень в галузі NLP. Вперше представлений в 2018 році, цей архітектурний підхід до обробки природної мови став основою для численних завдань, включаючи генерацію тексту, машинний переклад, аналіз тональності і багато інших.

Трансформер GPT базується на архітектурі з увагою, що дозволяє ефективно обробляти послідовності даних різної довжини. Основною ідеєю є використання механізмів уваги для взаємодії між словами (або токенами) у вхідних даних. Це дозволяє моделі розуміти контекст і структуру тексту.

Архітектура трансформера є ключовою для розуміння принципів роботи моделей, які використовують цей підхід. Трансформер був запропонований в статті "Attention Is All You Need" [6] в 2017 році і став популярним в галузі обробки природної мови та інших областях штучного інтелекту. Нижче розглянуті основні компоненти архітектури трансформера:

- позиційне кодування (Positional Encoding): оскільки трансформер обробляє послідовності даних без урахування порядку, потрібно додатково передати моделі інформацію про позицію кожного токена в послідовності. Для цього використовуються позиційні кодування, які додаються до векторного представлення кожного токена;

- блоки уваги (Attention Mechanism): основний компонент трансформера - це механізм уваги. Увага дозволяє моделі фокусуватися на певних частинах вхідної послідовності, залежно від їх важливості для завдання. Модель може визначити, які токени важливі для генерації кожного вихідного токена;

- багатоголовий механізм уваги (Multi-Head Attention): трансформер використовує кілька головок уваги паралельно, щоб вивчати різні типи залежностей у даних. Кожна головка генерує власні ваги для tokenів, і результати їхньої роботи комбінуються;

- повнозв'язані шари (Feed-Forward Layers): після кожного блоку уваги в трансформері є повнозв'язані шари, які виконують операції лінійного перетворення та активації, щоб обробити вихідні дані від блоку уваги;

- маскування (Masking): маскування використовується для приховання деяких входів від блоків уваги, щоб модель не "бачила" майбутні токени під час навчання, що важливо для генерації тексту послідовностями;
- стекування шарів (Layer Stacking): Трансформери зазвичай мають багато шарів, які стекуються один на одному. Це дозволяє моделі вивчати різні рівні абстракції та складність завдань;
- функція втрат (Loss Function): Під час тренування моделі використовують функцію втрат, яка вимірює різницю між прогнозованими і справжніми вихідними даними. Найпоширенішою функцією втрат для генерації тексту є перехресна ентропія (Cross-Entropy Loss) [7]. Ці компоненти разом утворюють архітектуру трансформера, яка здатна ефективно вирішувати завдання обробки послідовностей та генерації тексту. Трансформери виявилися дуже потужними для багатьох NLP-завдань і знайшли широке застосування в сучасних системах штучного інтелекту.

Однією з ключових особливостей трансформера GPT є здатність до автопрогнозування. Модель може послідовно генерувати текст, прогножуючи наступні токени на основі вже згенерованого контексту. Ця можливість дозволяє створювати тексти, які мають логічний зв'язок та граматичну правильність [8]. Генерація тексту на основі GPT може бути використана для багатьох завдань, включаючи створення автоматичних описів, генерацію мовленнєвого контенту, розробку діалогових систем та інше [9]. Застосування таких моделей розширюється на різні сфери життя, від медицини до медіа та маркетингу.

Використання архітектури GPT для генерації тестових даних для RESTful API є вигідним завдяки її здатності розуміти та генерувати природну мову, що забезпечує реалістичність тестів. Гнучкість у форматах даних, швидка адаптація до змін у специфікаціях API [10], та здатність генерувати велику кількість різноманітних даних значно економить час і підвищує якість тестування.

1.3 Методи генерації тестових даних за допомогою рекурентної нейронної мережі (RNN) та аналіз потенціалу RNN

Рекурентні нейронні мережі (RNN) відіграють ключову роль у дослідженні обробки природної мови, особливо у контексті структурування та генерації послідовних даних. Їхня основна перевага полягає у здатності зберігати інформацію з попередніх кроків обчислення, що дозволяє здійснювати аналіз залежностей у довгих текстових послідовностях. Ця характеристика робить RNN надзвичайно ефективними в моделюванні мовних патернів та залежностей, що є важливим для точної генерації тексту [11].

Використання RNN у генерації тексту демонструє їх здатність до відтворення складних мовних структур. Завдяки рекурентній архітектурі, ці мережі можуть моделювати зв'язки між різними частинами тексту, що дозволяє генерувати зв'язний і змістовно насичений вивід. Ця здатність робить RNN важливим інструментом у задачах, де потрібно створювати реалістичні та переконливі текстові послідовності.

Додатково, важливим аспектом є можливість RNN обробляти і генерувати текст на основі контексту. Це означає, що мережа може враховувати не тільки вхідні дані, але й загальний контекст тексту, що забезпечує глибше розуміння та здатність до генерації більш складних і різноманітних текстових форм.

Одним з методів полягає в тому, щоб використовувати RNN для генерації тексту, який відповідає заданому цільовому відгуку. Наприклад, RNN можна використовувати для генерації запитів для REST API, які відповідають певному набору критеріїв [12].

Інший метод полягає в тому, щоб використовувати RNN для генерації тексту, який відповідає певним статистичним характеристикам. Наприклад, RNN можна використовувати для генерації випадкових запитів для REST API, які мають певну довжину і складність.

Ще один метод полягає в тому, щоб використовувати RNN для генерації тексту, який є схожим на текст, що був використаний для навчання мережі. Наприклад, RNN можна використовувати для генерації запитів для REST API, які схожі на запити, які були використані для навчання мережі.

При виборі методу генерації тестових даних за допомогою RNN необхідно враховувати такі фактори:

- тип завдання: різні методи генерації тестових даних підходять для різних завдань. Наприклад, метод цільового відгуку підходить для завдань, де необхідно згенерувати текст, який відповідає певним вимогам. Метод генеративної моделі підходить для завдань, де необхідно згенерувати текст, який відповідає певним статистичним характеристикам. Метод генеративного навчання підходить для завдань, де необхідно згенерувати текст, який схожий на текст, що був використаний для навчання мережі;
- обчислювальні ресурси: генерація тестових даних за допомогою RNN може вимагати значних обчислювальних ресурсів [13].

1.4 Архітектура RNN, принципи роботи та процес навчання RNN

RNN - це клас нейронних мереж, який ефективно обробляє послідовні дані. Відрізняючись від звичайних глибоких нейронних мереж, RNN має петлі у своїй архітектурі, дозволяючи інформації передаватися від одного кроку мережі до наступного. Ця властивість робить RNN ідеальною для завдань, де необхідно враховувати контекст, таких як обробка мови або генерація тексту [14].

Архітектура рекурентних нейронних мереж (RNN) представляє собою важливий клас глибоких нейронних мереж, який широко використовується в обробці послідовних даних, включаючи мовлення, текст, часові ряди та інші сфери. Однією з головних характеристик RNN є їхній здатність до моделювання залежностей в послідовних даних шляхом використання зв'язків між попередніми станами мережі та поточними вхідними даними.

Принцип роботи RNN полягає в тому, що кожен часовий крок обчислює вихідний стан, який потім передається наступному кроку разом з новими

вхідними даними. Ця зворотна зв'язок дозволяє мережі зберігати та використовувати інформацію про попередні стани для вирішення завдань, пов'язаних із послідовністю.

Процес навчання RNN полягає в підлаштуванні вагових коефіцієнтів мережі, щоб вона вивчала корисні залежності в даних. Це робиться за допомогою алгоритмів оптимізації, таких як зворотнє розповсюдження помилки (backpropagation), який коригує ваги так, щоб мінімізувати помилку передбачення мережі. Після завершення процесу навчання RNN може використовуватися для генерації послідовностей, класифікації тексту, прогнозування часових рядів та інших завдань [15].

У контексті дослідження можна використовувати RNN для створення великої кількості різних запитів до API для тестування його функціональності та надійності. Мережа може навчитися генерувати різні запити з різними параметрами, щоб перевірити, чи API правильно обробляє різні вхідні дані. Такий підхід дозволяє автоматизувати тестування RESTful API та забезпечити високу якість програмного забезпечення, що використовує це API.

Висновки до розділу 1

У цьому розділі було докладно розглянуто різні інструменти та методи, які використовуються для оптимізації процесів тестування з метою уникнення ручного тестування та підвищення реалістичності тестових сценаріїв.

Перш за все, було розглянуто використання ШІ для генерації тестових даних та розглянуто як такий підхід підвищує ефективність, швидкість, та якість процесу тестування, зокрема за рахунок кращого покриття коду та реалістичності тестових сценаріїв.

Додатково, дослідження включає розробку та оптимізацію ефективних стратегій, які базуються на сучасних технологіях ШІ та враховують вимоги сучасного програмного забезпечення а також порівняння ефективності та надійності ШІ з ручним тестуванням на основі специфічних метрик і результатів дослідження..

Дослідження підкреслює важливість заміщення ручного тестування інтелектуальними методами, що дозволяє підвищити продуктивність, поліпшити якість та розширити охоплення можливих сценаріїв використання, роблячи дослідження значущим для сучасної практики тестування програмного забезпечення.

РОЗДІЛ 2. РОЗРОБКА МЕТОДІВ ГЕНЕРАЦІЇ ТЕСТОВИХ ДАНИХ З ВИКОРИСТАННЯМ ШІ

2.1 Генерація тестових даних за допомогою API чат-бота породжувального, попередньо тренованого трансформера

Попередньо треновані трансформери стали важливим інструментом для обробки природної мови та генерації текстів. Розроблені для вирішення складних завдань завдяки великому обсягу даних, ці моделі виявляються неабияким ресурсом у різноманітних дисциплінах. Досягненням особливого інтересу стає застосування попередньо тренованих трансформерів у контексті їхнього породжувального використання.

Трансформерна архітектура - це тип архітектури глибоких нейронних мереж, яка була розроблена для обробки послідовних даних, зокрема тексту. Вона була вперше представлена в роботі "Attention Is All You Need" в 2017 році та з тих пір стала основною архітектурою для багатьох завдань обробки природної мови та інших послідовних даних [16].

Основним елементом трансформера є механізм уваги (attention mechanism), який дозволяє мережі "концентруватися" на різних частинах вхідного тексту при обробці кожного вихідного символу. Це дозволяє моделі враховувати довгі залежності та створювати більш контекстно-залежні представлення тексту.

Математично, механізм уваги включає в себе обчислення ваги для кожного слова в послідовності відносно поточного слова, і ці ваги використовуються для обчислення вагового контекстного представлення слова. У формулі, це можна представити як:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V,$$

де Q, K, V - це матриці запитів (query), ключів (keys), і значень (values);

d, k - розмірність ключів;

softmax - операція, яка обчислює розподіл ваг за допомогою функції *softmax*.

Трансформерна архітектура також включає в себе додаткові шари, такі як шари feedforward та нормалізації, що допомагають вдосконалити представлення даних. Загальна ідея трансформера полягає в тому, щоб використовувати механізм уваги для обробки кожного елемента в послідовності залежно від всіх інших елементів, що дозволяє моделі добре узагальнювати залежності в даних та досягати вражаючих результатів у завданнях обробки тексту [17].

У рамках завдання, зосередимося на розробці та впровадженні процесу генерації тестових даних для застосування у автоматизованому та мануальному тестуванні RESTful API з використанням сучасних технологій штучного інтелекту. Основною метою є створення ефективної та гнучкої системи, що здатна генерувати високоякісні тестові дані, спрямовані на оцінку різних аспектів функціональності та безпеки API. У цьому контексті, ключовим інструментом є використання моделі GPT (Generative Pre-trained Transformer), яка дозволяє автоматизувати процес генерації даних і значно підвищити охоплення тестів.

Використання GPT для генерації даних включає кілька основних кроків, що починаються з формулювання вхідного запиту і закінчуються інтеграцією згенерованих даних у процес тестування API [18]. Кожен крок має свої особливості і важливість у загальному контексті завдання, а розуміння цих аспектів допомагає у досягненні оптимальних результатів тестування.

Нижче ми детально розглянемо кожен з цих кроків, починаючи з формулювання вхідного запиту, обробки запиту через механізм уваги в GPT, генерації відповіді, оптимізації вибору токенів, формування JSON відповіді, і, нарешті, використання згенерованих даних у тестуванні RESTful API.

Визначення вхідного запиту є ключовим кроком у процесі використання GPT для генерації тестових даних. Вхідний запит, який ми визначаємо як C , слугує як основа для інструкцій, які модель GPT буде використовувати для створення даних. Він складається з набору компонентів $C = \{c_1, c_2, \dots, c_n\}$,

кожен з яких вносить певний аспект інформації або вимогу до генерованих даних [19].

Важливість визначення вхідного запиту можна розкрити в наступних аспектах:

- визначення вхідного запиту дозволяє точно передати контекст та специфікації, які повинні бути враховані моделлю при генерації даних. Це особливо важливо, коли мова йде про створення структурних даних, як-от JSON, що вимагають дотримання певного формату та структури;
- чітко визначений запит гарантує, що згенеровані дані відповідають конкретним вимогам тестування, наприклад, включення валідних та невалідних значень для перевірки роботи API [20];
- з точно визначеним запитом, GPT може більш ефективно генерувати відповідні тестові дані, що зменшує потребу в ручному втручанні та підвищує швидкість тестування.

У підсумку, адекватне визначення вхідного запиту є фундаментальним для успішного використання GPT у генерації тестових даних, забезпечуючи точність, відповідність специфікаціям і ефективність у процесі тестування.

Ключовою частиною моделі GPT є механізм уваги [21], який дозволяє моделі визначити, які частини вхідного запиту найбільш важливі для генерації відповіді. У контексті вашого дослідження, де ми використовуємо GPT для створення тестових даних у форматі JSON для RESTful API, механізм уваги аналізує і розуміє важливість різних аспектів вашого запиту.

Механізм уваги в моделі GPT визначається як:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V .$$

У цьому випадку, Q, K, V визначаються відповідно до контексту C , та модель зосереджується на релевантних аспектах для генерації даних.

Механізм уваги працює шляхом вагового розподілу уваги серед різних частин вхідного тексту. Він використовує вектори запитів Q , ключів K , та значень V , які витягуються з вхідного тексту. В моделі GPT ці вектори

визначають, які елементи вхідного запиту найбільш релевантні для поточної задачі генерації даних.

Коли модель GPT отримує вхідний запит для створення JSON структури, механізм уваги використовується для визначення, які інструкції та вимоги в запиті є ключовими. Наприклад, якщо запит включає вимоги до валідності та формату даних, механізм уваги допоможе моделі зосередитись на цих аспектах, щоб згенерувати відповідні тестові дані.

Завдяки механізму уваги, GPT здатна більш точно та ефективно реагувати на складні запити, створюючи високоякісні та релевантні тестові дані. Це забезпечує, що згенеровані дані не тільки відповідають формальним критеріям, але й ефективно виконують свою роль у тестуванні RESTful API [22].

Після того, як механізм уваги в моделі GPT аналізує та визначає важливі частини вхідного запиту, наступний крок - генерація відповіді. В цьому кроці модель використовує зібрану інформацію та контекст для створення релевантного виводу - у нашому випадку, тестових даних у форматі JSON.

Згенерована відповідь R визначається як послідовність токенів $\{r_1, r_2, \dots, r_m\}$, де кожен r_j є частиною згенерованих даних у форматі JSON. Ймовірність кожного токена в R залежить від попередніх токенів та контексту R і визначається як послідовність токенів C , тобто

$$P(r_j | r_{<j}, C; \theta),$$

де θ - параметри моделі.

Результатом цього процесу є згенерований вивід, який не просто формально відповідає запиту, але й враховує нюанси та специфікації, які можуть бути важливими для тестування. Наприклад, якщо потрібно генерувати тестові дані, що включають як валідні, так і невалідні JSON об'єкти, GPT врахує це, створюючи відповідний набір даних.

Такий підхід до генерації відповіді робить GPT оптимальним інструментом для створення тестових даних. Вона не тільки автоматизує процес генерації, але й забезпечує високий рівень адаптивності та релевантності даних,

що є важливим для ефективного тестування RESTful API.

Під час генерації відповіді, модель GPT досягає точки, де вона повинна вибрати наступний токен. Оптимізація вибору токенів полягає у виборі найбільш підходящого токена, який продовжує послідовність відповіді. Це вирішальний момент, оскільки кожен вибраний токен впливає на зміст і якість кінцевого виводу.

Вибір токена відбувається шляхом оцінки умовних ймовірностей на основі попередньо згенерованих токенів та вхідного контексту. Модель використовує свої внутрішні параметри, набуті під час попереднього навчання, щоб визначити, який токен має найвищу ймовірність бути правильним у даному контексті.

Модель вибирає кожен токен r_j , максимізуючи умовну ймовірність з урахуванням контексту:

$$r_j = \arg \max P(r_j | r_{<j}, C; \theta).$$

Цей процес оптимізації є критичним для забезпечення якості та релевантності згенерованих тестових даних. Правильний вибір токенів гарантує, що вихідні дані точно відповідають заданим вимогам та контексту, що є особливо важливим при створенні даних для тестування RESTful API.

Таким чином, оптимізація вибору токенів забезпечує, що згенеровані дані будуть не тільки формально правильними, але й функціонально ефективними для використання в тестових сценаріях.

Після того, як модель GPT вибрала відповідні токени, наступний крок – це трансформація цих токенів у структуровану відповідь у форматі JSON. Цей етап є критичним для перетворення сирогої послідовності токенів на корисні тестові дані.

Остаточний вивід R трансформується у структурований JSON формат для використання в тестуванні API [23].

Процес трансформації включає організацію токенів у валідну структуру JSON, що відповідає всім вимогам до формату та включає необхідні атрибути та значення. Це означає розміщення токенів у правильному порядку, врахування

необхідної ієрархії та забезпечення відповідності встановленим специфікаціям.

Формування JSON відповіді є ключовим для тестування RESTful API, оскільки забезпечує, що тестові дані можуть бути легко інтегровані у тестові процедури. Правильно структуровані JSON дані дозволяють ефективно перевіряти функціональність та відмовостійкість API, а також виявляти потенційні помилки та вразливості.

Таким чином, формування JSON відповіді забезпечує, що результати роботи моделі GPT є не тільки технічно коректними, але й практично придатними для реальних сценаріїв тестування.

На цьому етапі, згенеровані та структуровані у форматі JSON дані використовуються безпосередньо для тестування RESTful API. Ці дані відправляються як частина запитів до API, щоб оцінити, як API обробляє різні вхідні сценарії, включаючи як валідні, так і не валідні варіанти [24].

Використання синтетичних тестових даних дозволяє ефективно оцінити функціональність, відмовостійкість та безпеку API. Це включає перевірку правильності обробки вхідних даних API, виявлення помилок у обробці запитів та перевірку захисту від потенційно шкідливих вхідних даних.

Використання автоматично згенерованих даних значно підвищує ефективність процесу тестування. Замість ручного створення тестових сценаріїв, можна швидко отримати різноманітний набір даних, що охоплює широкий спектр можливих вхідних умов для API.

Таким чином, інтеграція згенерованих даних у тестування RESTful API є важливим кроком, який не тільки спрощує та автоматизує процес тестування, але й забезпечує глибоке та всебічне випробування функціональності API.

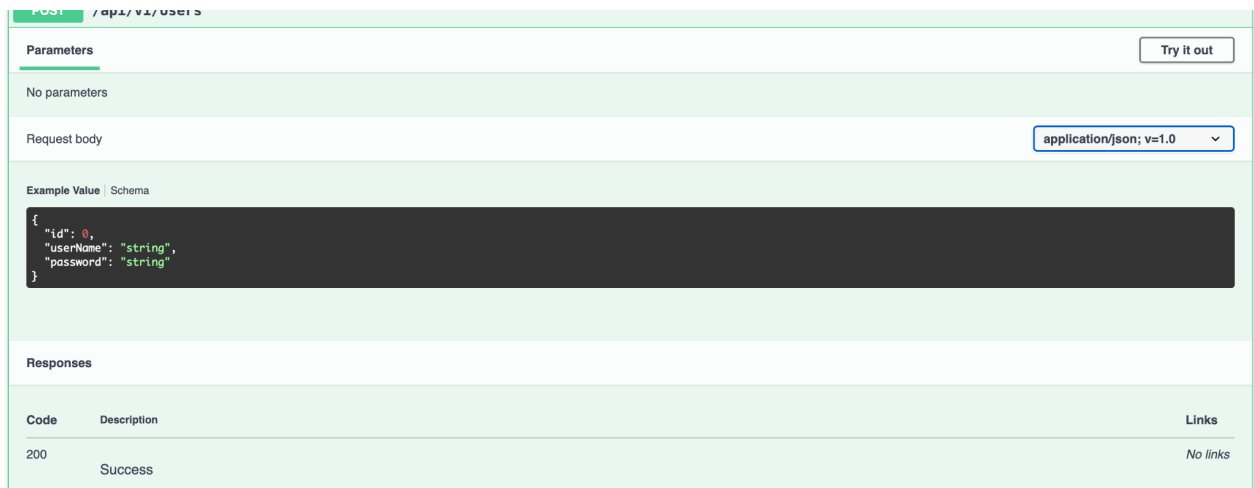
Генерація та використання синтетичних тестових даних може бути корисною в контексті тестування RESTful API з різних причин. Декілька прикладів практичних застосувань включають [25]:

- тестування зовнішніх систем - важливо мати тестові дані, які відображають реальні сценарії використання для взаємодії з іншими системами;
- стрес-тестування - створення великої кількості синтетичних даних

для перевірки стійкості API під час інтенсивного використання;

- тестування безпеки - перевірка реакції API на спроби атак та забезпечення безпеки перед впровадженням;
- тестування відновлення після збоїв - створення сценаріїв для перевірки ефективності відновлення системи після непередбачуваних ситуацій.

Розглянемо один з прикладів такого запиту:



The screenshot shows an API client interface for a POST request to the endpoint `/api/v1/user`. The interface includes a 'Parameters' section with 'No parameters', a 'Request body' section with a dropdown menu set to 'application/json; v=1.0', and an 'Example Value' section displaying a JSON object: `{ "id": 0, "userName": "string", "password": "string" }`. Below this is a 'Responses' section with a table showing a 200 status code and a 'Success' description.

Code	Description	Links
200	Success	No links

Рис 2.1 Запит POST /user

Рисунок 2.1 показує основні компоненти та опис запиту для створення користувача та дані які ми будуть згенеровані.

Лістинг 2.1

JSON об'єкт для створення користувача в системі

```
{
  "id": 0,
  "userName": "string",
  "password": "string"
}
```

Визначення параметрів:

1. `id` - унікальний ідентифікатор користувача;
2. `userName` - унікальне ім'я користувача;
3. `password` - пароль користувача.

На прикладі даного запиту ми будемо використовувати API GPT для генерації тестових даних для кожного з параметрів.

2.2 Генерація масиву даних за допомогою API GPT-3.5-turbo

Генерація тестових даних за допомогою чат-бота, що базується на породжувальному попередньо тренованому трансформері, використовує математичні принципи глибокого навчання, а зокрема, архітектуру трансформера. Генерація тестових даних заснована на здатності попередньо тренованого трансформера "вивчати" структуру та стилістику тексту під час тренування та використовувати ці знання для генерації нового тексту.

Вибір між різними моделями для конкретного завдання може залежати від різних факторів, таких як точність, швидкодія, обсяг генерації тексту, вартість та інші. Однак ось кілька причин, чому в даній роботі була вибрана GPT-3.5-turbo:

1. GPT-3.5-turbo є однією з найшвидших моделей у сімействі GPT. Вона забезпечує хорошу продуктивність при генерації тексту, що може бути важливим фактором для реального часу або інтерактивних застосувань.

2. Зазвичай, GPT-3.5-turbo може надати точні та змістовні відповіді. Для багатьох завдань вона може бути досить ефективною без необхідності використання більш великих моделей, таких як оригінальний GPT-3..

3. GPT-3.5-turbo може бути більш ефективним з точки зору витрат ресурсів порівняно з більшими моделями. Це може бути важливим фактором, особливо при обмежених обчислювальних ресурсах або обмеженому бюджеті.

Створення API запиту до GPT-3.5-turbo відкриває двері до потужної можливості автоматичної генерації тексту та використання цих даних у автоматизованому тестуванні. Цей процес дозволяє взаємодіяти з мовним моделюванням GPT-3.5-turbo від OpenAI [26], використовуючи простий та зручний інтерфейс програмування застосунків (API).

Лістинг 2.2

Конфігурація запиту до GPT-3.5-turbo

```
POST https://api.openai.com/v1/chat/completions
-H "Content-Type: application/json"
-H "Authorization: Bearer $OPENAI_API_KEY"
```

```
-d '{
  "model": "gpt-3.5-turbo-1106",
  "messages": [
    {
      "role": "system",
      "content": "You are an assistant for QA engineer and you
need to provide the test data for API request. Important rule is
return ONLY values, no parameters in JSON objects in content."
    },
    {
      "role": "user",
      "content": "generate the array with positive values for the
name which is string. List of valid Names: could contain single,
double names with -, double names with space, names could be 3 types
of languages - cyrillic, latin, with or without diacritics."
    }
  ]
}
```

На лістингу представлено HTTP запит типу POST до API OpenAI для отримання результату від моделі GPT-3.5 Turbo. Нижче розглянемо параметри цього запиту:

- URL адреса: <https://api.openai.com/v1/chat/completions>;
- Заголовки: -H "Content-Type: application/json": Цей заголовок вказує, що дані у форматі JSON. -H "Authorization: Bearer \$OPENAI_API_KEY": Заголовок авторизації, \$OPENAI_API_KEY повинен бути замінений реальним ключем API для доступу до OpenAI сервісу.
- Тіло запиту (Request Body): -d '{...}': Це тіло запиту, яке містить дані у форматі JSON. Ми передаємо об'єкт з двома повідомленнями: системним та користувачем.

Цей запит створює діалог між системним повідомленням, яке визначає роль та правила для асистента, та повідомленням користувача, яке містить вимогу до асистента щодо генерації масиву з позитивними значеннями для параметра "name" з різними можливими форматами і мовами. У вказаному запиті "role": "system" вказує на роль повідомлення в діалозі, а саме на системне

повідомлення. У діалозі для генерації тексту за допомогою OpenAI GPT-3.5 Turbo визначення ролей дозволяє встановлювати контекст та вказувати різні "ролі" для кожного повідомлення, щоб сценарій був зрозуміліший для моделі.

Системне повідомлення встановлює роль асистента для QA інженера та визначає важливе правило: повертати лише значення, а не параметри у JSON об'єктах в контенті. Щоб надати контекст для правильного генерування даних, системні повідомлення можна використовувати для передачі інструкцій асистентові чи визначення умов для завдання. У даному випадку, асистент повинен згенерувати масив з позитивними значеннями для параметра "name", при цьому дотримуючись конкретних вимог до формату та мови. Контекст системного повідомлення допомагає моделі краще розуміти завдання і виконувати його відповідно до визначених вимог.

2.3 Реалізація та навчання рекурентної нейронної мережі (RNN) для генерації синтетичних тестових даних

У цьому розділі ми розглянемо використання рекурентної нейронної мережі (RNN) для генерації синтетичних тестових даних для тіла запиту для RESTful API. RNN - це тип нейронної мережі, яка може обробляти послідовності даних [27]. Це робить їх ідеальними для генерації синтетичних даних, які відповідають певному шаблону.

Для генерації синтетичних тестових даних для тіла запиту для POST методу створення нового користувача ми будемо використовувати бібліотеки TensorFlow і Gretel Synthetics.

TensorFlow - це бібліотека машинного навчання, яка використовується для навчання та використання рекурентної нейронної мережі (RNN) для генерації синтетичних даних.

Gretel Synthetics - це бібліотека, яка надає інструменти для створення синтетичних даних для різних цілей, таких як тестування програмного забезпечення, навчання машинного навчання та захист даних. Бібліотека Gretel

Synthetics містить набори інструментів для створення синтетичних даних, які використовують рекурентні нейронні мережі (RNN).

Ми використовуємо ці бібліотеки для того, щоб створити модель RNN, яка може генерувати синтетичні тіла запитів, які відповідають певному шаблону, а саме, шаблону необхідному для тестування методу createUser. Ось кілька конкретних причин, чому ми можемо використовувати ці бібліотеки:

- полегшити тестування програмного забезпечення. Синтетичні тестові дані можуть бути використані для створення широкого спектру тестових сценаріїв, які може бути важко або неможливо реалізувати за допомогою реальних даних. Це може допомогти нам знайти помилки у нашому програмному забезпеченні, які ми могли б пропустити, використовуючи лише реальні дані;

- захистити конфіденційність даних, оскільки синтетичні тестові дані можуть бути використані для створення даних, які не містять особистої інформації. Це може допомогти нам захистити конфіденційність даних під час тестування програмного забезпечення;

- синтетичні тестові дані можуть бути використані для збільшення розміру набору даних, який ми використовуємо для навчання моделі машинного навчання. Це може допомогти нам покращити продуктивність моделі.

Розробка проводиться у Google Collab - безкоштовному, доступному у хмарі і простому у використанні інструменті. Отже, виконуємо наступні команди для встановлення бібліотек TensorFlow і Gretel Synthetics [28].

Лістинг 2.3

Встановлення бібліотек TensorFlow та Gretel Synthetics

```
pip install tensorflow==2.8.0
pip install gretel-synthetics --upgrade
```

Розглянемо налаштування процесу тренування нейронної мережі. Цей фрагмент коду використовує бібліотеку gretel-synthetics для тренування синтетичного генератора даних.

Лістинг 2.4

Налаштування процесу тренування нейронної мережі

```

from pathlib import Path
from gretel_synthetics.config import LocalConfig
config = LocalConfig(

    max_line_len=2048,
    vocab_size=20000,
    field_delimiter="," ,
    overwrite=True,
    checkpoint_dir=(Path.cwd() / 'checkpoints').as_posix(),

    input_data_path="https://gretel-public-website.s3-us-west-2.amazonaws.
com/datasets/uber_scooter_rides_1day.csv"
)

```

Максимальна довжина рядка `max_line_len` визначає максимальну кількість символів у вхідних рядках тренувальних даних. Більші значення можуть допомогти моделі засвоювати більше інформації, але при цьому може збільшитися обсяг пам'яті, необхідний для тренування.

Роздільник полів `field_delimiter` вказує, якщо тренувальний текст є структурованим, це полегшує розпізнавання полів у вхідних даних. У нашому випадку це необхідно оскільки файл з тренувальними даними має формат `.csv` та має роздільник полів. Визначення, чи слід перезаписувати раніше збережені моделі виконується завдяки налаштуванню `overwrite: true`.

Шлях до директорії контрольних точок `checkpoint_dir` вказує шлях до директорії, де зберігатимуться контрольні точки під час тренування [29].

Кількість епох (`epochs`) визначає, скільки разів тренувальний алгоритм буде працювати з повним набором даних. Зазвичай більше епох може покращити точність моделі, але при цьому може збільшити час тренування.

Шлях до файлу (`input_data_path`) визначає місцеположення з тренувальними даними, які використовуються для тренування моделі. Для прикладу API запиту який ми розглянули раніше, ми будемо використовувати файл з 10000 різних об'єктів з параметрами, необхідними для створення користувача в системі. Значення параметрів включають як валідні так і не валідні дані, розподілені за принципами технік тест дизайну, а саме:

еквівалентного розділення на класи по типам даних та граничних значень цих еквівалентних класів.

Лістинг 2.5

Функція тренування моделі синтетичного генератора

```
from gretel_synthetics.train import train_rnn  
train_rnn(config)
```

Цей код викликає функцію `train_rnn` з аргументом `config`, яка в свою чергу використовує конфігурацію `config` для тренування рекурентної нейронної мережі (RNN) з використанням бібліотеки `gretel_synthetics`. Давайте розглянемо кожен етап цього процесу: `from gretel_synthetics.train import train_rnn` імпортує функцію `train_rnn` з модуля `gretel_synthetics.train`. Ця функція відповідає за тренування моделі синтетичного генератора даних на основі рекурентної нейронної мережі. Виклик функції `train_rnn(config)` передає об'єкт конфігурації `config` як аргумент для тренування моделі. Функція `train_rnn` використовує цей конфіг для налаштування параметрів тренування, таких як максимальна довжина рядка, розмір словника, роздільник полів тощо.

Загальною метою цього коду є тренування моделі, яка може генерувати синтетичні дані на основі вхідних тренувальних даних. Процес тренування може займати час, особливо при великому обсязі даних та складних моделях. Після тренування згенерована модель може бути використана для створення нових синтетичних даних, які мають подібні характеристики до вихідних тренувальних даних.

Результати виконання коду під час тренування рекурентної нейронної мережі (RNN) для синтетичного генератора даних представлені на рис.2.1

```
100%|██████████| 10001/10001 [00:00<00:00, 28473.11it/s]
Model: "sequential"
Layer (type) Output Shape Param #
-----
embedding (Embedding) (64, None, 256) 5120000
dropout (Dropout) (64, None, 256) 0
lstm (LSTM) (64, None, 256) 525312
dropout_1 (Dropout) (64, None, 256) 0
lstm_1 (LSTM) (64, None, 256) 525312
dropout_2 (Dropout) (64, None, 256) 0
dense (Dense) (64, None, 20000) 5140000
-----
Total params: 11,310,624
Trainable params: 11,310,624
Non-trainable params: 0

Epoch 1/100
45/46 [=====] - 10s 121ms/step - loss: 6.2203 - accuracy: 0.8975WARNING:tensorflow:Callback method 'on_train_batch_end' is slow compared to the batch time (batch time: 0.0519s vs 'on_train_batch_end' time: 0.0729s). Check your callbacks.
46/46 [=====] - 19s 206ms/step - loss: 6.2553 - accuracy: 0.1777 - val_loss: 5.4336 - val_accuracy: 0.1867
Epoch 2/100
45/46 [=====] - 9s 159ms/step - loss: 4.2689 - accuracy: 0.3558 - val_loss: 3.6658 - val_accuracy: 0.4236
46/46 [=====] - 9s 186ms/step - loss: 3.6178 - accuracy: 0.4233 - val_loss: 3.4837 - val_accuracy: 0.4388
Epoch 3/100
45/46 [=====] - 8s 159ms/step - loss: 3.4249 - accuracy: 0.4319 - val_loss: 3.2442 - val_accuracy: 0.4365
46/46 [=====] - 8s 175ms/step - loss: 3.2969 - accuracy: 0.4366 - val_loss: 3.1823 - val_accuracy: 0.4417
Epoch 4/100
45/46 [=====] - 8s 162ms/step - loss: 3.2892 - accuracy: 0.4432 - val_loss: 3.0751 - val_accuracy: 0.4457
46/46 [=====] - 9s 160ms/step - loss: 3.1428 - accuracy: 0.4448 - val_loss: 3.0219 - val_accuracy: 0.4512
Epoch 5/100
45/46 [=====] - 9s 187ms/step - loss: 3.0943 - accuracy: 0.4479 - val_loss: 2.9748 - val_accuracy: 0.4557
46/46 [=====] - 9s 161ms/step - loss: 3.0476 - accuracy: 0.4509 - val_loss: 2.9818 - val_accuracy: 0.4619
Epoch 6/100
45/46 [=====] - 9s 180ms/step - loss: 3.0040 - accuracy: 0.4553 - val_loss: 2.8626 - val_accuracy: 0.4629
46/46 [=====] - 9s 162ms/step - loss: 2.9612 - accuracy: 0.4589 - val_loss: 2.8181 - val_accuracy: 0.4714
Epoch 7/100
45/46 [=====] - 9s 178ms/step - loss: 2.9226 - accuracy: 0.4625 - val_loss: 2.7735 - val_accuracy: 0.4774
46/46 [=====] - 9s 187ms/step - loss: 2.8805 - accuracy: 0.4661 - val_loss: 2.7181 - val_accuracy: 0.4797
Epoch 8/100
45/46 [=====] - 9s 178ms/step - loss: 2.8472 - accuracy: 0.4698 - val_loss: 2.6745 - val_accuracy: 0.4846
46/46 [=====] - 9s 189ms/step - loss: 2.8098 - accuracy: 0.4727 - val_loss: 2.6102 - val_accuracy: 0.4985
Epoch 9/100
45/46 [=====] - 10s 190ms/step - loss: 2.7752 - accuracy: 0.4758 - val_loss: 2.5858 - val_accuracy: 0.4945
46/46 [=====] - 10s 190ms/step - loss: 2.7752 - accuracy: 0.4758 - val_loss: 2.5858 - val_accuracy: 0.4945
```

Рис 2.2 Логування тренування рекурентної нейронної мережі

Розберемо основні етапи та інформацію, в результаті виконання передніх функцій:

1. Прогрес тренування: "00%|██████████| 10001/10001 [00:00<00:00, 28473.11it/s]". Це відображає прогрес тренування у вигляді прогрес-смуги. 10001 ітерація вказує на кількість оброблених пакетів даних (батчів), а швидкість виконання вимірюється в ітераціях за секунду.

2. Опис моделі: Знаходиться між результатом "Model: "sequential"" та "Total params:". Це опис моделі RNN. Модель має шар Embedding, Dropout, LSTM (Long Short-Term Memory), Dense, і вона складається з загальної кількості параметрів 11,310,624. Цей опис структури моделі може бути корисним для розуміння архітектури, використаної для генерації синтетичних даних.

3. Інформація про тренування: Наприклад, "Epoch 1/100", "loss: 6.2553 - accuracy: 0.1777 - val_loss: 5.4336 - val_accuracy: 0.1867". Це виведена інформація про кожну епоху тренування. Зокрема, вказується номер епохи, значення функції втрат (loss) та точності (accuracy) на тренувальних та валідаційних даних. Ця інформація дозволяє вам відслідковувати покращення моделі під час тренування.

4. Інша інформація про тренування: Наприклад, час тренування на кожній епосі, виведення попередження про те, що одна з callback-функцій повільна, та інша інформація, пов'язана з ходом тренування.

Ці дані корисні для відстеження прогресу тренування, ефективності моделі та виявлення будь-яких проблем під час тренування RNN [30].

2.4 Використання мапінгу для перетворення типів даних та JSON-схеми для маркування тестових сценаріїв

У наступному фрагменті коду використовується функція `generate_text` для генерації текстових рядків з використанням конфігурації `config`. Кожен згенерований рядок розбивається на окремі поля, які потім мапляться за допомогою функції `gen_mapped_result`. Оскільки результатом роботи натренованої моделі нейронної мережі є тест, то і всі дані отримані будуть мати символічний тип даних. Для того щоб мати можливість використовувати дані як в позитивному так і в негативному тестуванні, результати генерації, мапінгу та валідації виводяться у вигляді JSON-об'єкту з міткою 'valid' або 'invalid'.

Лістинг 2.6

Реалізація функції “generate_text”

```
for gen_text_obj in generate_text(config, line_validator=None,
num_lines=100):
    line = gen_text_obj.text
    record = dict(zip(["id", "username", "firstName", "lastName",
"email", "password", "phone", "userStatus"], line.split(",")))
    mapped_record = gen_mapped_result(record)
    result = {'label': validate_record(mapped_record), 'data':
mapped_record}
    print(json.dumps(result))
```

Для того щоб забезпечити відповідність типам даних, який визначений у Restful API використовується функція `gen_mapped_result`, яка виконує перетворення отриманих даних. Наприклад, у цьому випадку, перевіряється, чи дані 'id' та 'userStatus' мають правильний тип (ціле число) перед використанням

у тілі запиту. В разі невідповідності визначеним у JSON-схемі вимогам, кожен об'єкт буде визначений як позитивний або негативний тест, що дозволить маркувати тестові сценарії. Маппінг гарантує, що згенеровані дані відповідають вимогам API та можуть бути відправлені в тілі запиту без проблем або ж, якщо у згенерованих даних 'id' та 'userStatus' мають неправильний тип даних, наприклад символьний тип даних тоді об'єкт буде визначений як негативний тест.

Лістинг 2.7

Реалізація функції “gen_mapped_result”

```
def gen_mapped_result(obj):
    mapped_result = copy.deepcopy(obj)

    if 'id' in mapped_result and isinstance(mapped_result['id'], str):
        try:
            mapped_result['id'] = int(mapped_result['id'])
        except ValueError:
            mapped_result['id'] = 0

    if 'userStatus' in mapped_result and
isinstance(mapped_result['userStatus'], str):
        try:
            mapped_result['userStatus'] =
int(mapped_result['userStatus'])
        except ValueError:
            mapped_result['userStatus'] = 0

    return mapped_result
```

2.5 Використання JSON-схеми для маркування тестових сценаріїв

У цьому розділі визначається JSON-схема (json_schema), яка описує очікувану структуру та формат генерованих даних.

JSON-схема (JSON Schema) - це специфікація, яка використовується для опису структури та обмежень даних в форматі JSON [31]. JSON (JavaScript Object Notation) - це легкий формат обміну даними, який широко

використовується в веб-розробці для передачі даних між різними програмами і системами. JSON-схема дозволяє визначити правила для структури та типів даних у JSON-об'єктах, щоб забезпечити їхню валідацію і коректність обробки.

JSON-схема може бути використана для таких цілей:

- валідація даних: JSON-схема дозволяє перевірити, чи відповідають дані у форматі JSON заданим правилам. Наприклад, можна визначити, що поле "ім'я" має бути рядком, а поле "вік" - цілим числом;
- документація: JSON-схема може служити як документація для структури даних, що допомагає розробникам іншими користувачами легше розуміти, як повинні виглядати дані;
- генерація коду: JSON-схема може бути використана для автоматичної генерації коду для обробки даних у відповідності до заданих правил.

JSON-схеми можуть бути складні та включати різні правила для об'єктів, масивів, типів даних, обов'язкових та необов'язкових полів тощо. Вони використовуються в багатьох сучасних веб-сервісах та програмах для забезпечення консистентності та надійності обміну даними.

Використання JSON-схеми у даній роботі може бути обумовлене кількома ключовими причинами, які стосуються ефективності, структуризації даних, та інтеграції з сучасними технологіями:

- структурування даних: JSON (JavaScript Object Notation) є легким форматом обміну даними, який легко читається як людьми, так і машинами. Використання JSON-схеми дозволяє чітко структурувати та визначати формат даних, що спрощує їх обробку та аналіз;
- гнучкість і сумісність: JSON є широко прийнятим стандартом у веб-розробці та багатьох інших галузях ІТ. Це забезпечує високий рівень сумісності з різними мовами програмування та платформами.
- валідація даних: JSON-схема дозволяє визначити правила або обмеження для даних, що забезпечує їх валідацію при обробці. Це особливо

важливо при роботі з даними, що приходять з зовнішніх джерел, або при інтеграції систем.

- організація тестових даних: У контексті генерації тестових даних, JSON-схема може використовуватися для створення структурованих тестових випадків, що полегшує автоматизацію тестування та аналіз результатів.
- інтеграція з іншими системами: JSON-схеми часто використовуються для опису структури даних у RESTful API, що робить їх ідеальними для інтеграції різних веб-сервісів та додатків.
- легкість оновлення та розширення: JSON-схеми легко оновлювати та розширювати, що є важливим для динамічних проєктів, де вимоги можуть змінюватися або розширюватися з часом.

Загалом, використання JSON-схеми в рамках даної реалізації може значно спростити обробку та аналіз даних, а також поліпшити інтеграцію та взаємодію з іншими системами та технологіями.

Функція `validate_record` використовує бібліотеку `jsonschema` для валідації згенерованих та замаплених даних на відповідність визначеній JSON-схемі. В результаті отримується мітка `'valid'` або `'invalid'` в залежності від валідації.

Мета використання даної функції - перевірка згенерованих даних на відповідність певним правилам або стандартам, визначеним у JSON-схемі. Валідація кожного запису (об'єкту JSON) відбувається шляхом порівняння його структури та вмісту з правилами, заданими в JSON-схемі.

Бібліотека `jsonschema` - це бібліотека Python, яка дозволяє валідувати JSON об'єкти, використовуючи схеми JSON. Вона перевіряє, чи відповідають дані вимогам схеми, включаючи типи даних, формати, обмеження розміру тощо. Механізм Валідації у `jsonschema` - це аналіз структури JSON об'єкта та перевірка, чи всі його елементи відповідають визначенням у схемі.

Лістинг 2.8

Опис JSON схеми

```
json_schema = {  
    "type": "object",  
    "properties": {
```

```

    "id": {
      "type": "integer",
      "minimum": 1,
      "maximum": 9999
    },
    "username": {
      "type": "string"
    },
    "firstName": {
      "type": "string"
    },
    "lastName": {
      "type": "string"
    },
    "email": {
      "type": "string",
      "format": "email"
    },
    "password": {
      "type": "string"
    },
    "phone": {
      "type": "string",
      "pattern": "^\\+\\d{1,3} \\d{1,}-\\d{1,}-\\d{1,}$"
    },
    "userStatus": {
      "type": "integer",
      "enum": [0, 1]
    }
  },
  "required": [
    "id",
    "username",
    "firstName",
    "lastName",
    "email",
    "password",
    "phone",
    "userStatus"
  ]
}

```

```
]
}
```

Схема визначає тип об'єкта ("type": "object"), а також властивості цього об'єкта (наприклад, id, username, email тощо):

1. Властивості об'єкта:

- id: Властивість, яка визначається як ціле число ("type": "integer"), з обмеженнями, що значення має бути мінімум 1 і максимум 9999;
- username: Рядкова властивість ("type": "string"), без додаткових обмежень;
- firstName і lastName: Обидві ці властивості визначені як рядки, що відповідають іменам осіб;
- email: Властивість, яка також є рядком, але з додатковим обмеженням, що вона має відповідати формату електронної пошти ("format": "email");
- password: Ще одна рядкова властивість, яка, як передбачається, використовується для зберігання паролів.

2. Додаткові властивості:

- phone: Визначається як рядок, але з додатковим вимогами до формату, використовуючи регулярний вираз для визначення структури телефонного номера ("pattern": "^\\+\\d{1,3} \\d{1,}-\\d{1,}-\\d{1,}\$");
- userStatus: Цілочисельна властивість, яка має відповідати одному з перерахованих значень ("enum": [0, 1]), що може відображати, наприклад, статус користувача.

3. Обов'язкові властивості:

- в схемі також вказано, які властивості є обов'язковими для кожного об'єкта: id, username, firstName, lastName, email, password, phone, і userStatus.

Мітки 'valid' або 'invalid': Після валідації кожен запис отримує мітку 'valid', якщо він відповідає схемі, або 'invalid', якщо він їй не відповідає. Це

дозволяє легко ідентифікувати дані, які можуть бути використані в подальших процесах, та ті, які потребують перегляду або корекції. Ця система валідації забезпечує надійність та точність даних, які використовуються у даній роботі, і є важливою частиною процесу обробки та аналізу даних.

JSON-схема включає обмеження типів, мінімальні та максимальні значення, обмеження формату та перерахування допустимих значень. В результаті ми отримуємо набір з фіксованою кількістю об'єктів, яку можна змінити в залежності від потреб та можливостей тестувальника:

```
{
  "label": "valid", "data": {
    "id": 1588, "username": "Vergie Ondricka", "firstName": "Savanna", "lastName": "Bosinski", "email": "lana13cristi@yahoo.com", "password": "7m4d4-181w9jlr", "phone": "+3 326-345-3487", "userStatus": 0},
    "label": "valid", "data": {
      "id": 7986, "username": "Pamela Mueller", "firstName": "Crawford", "lastName": "Schaefer", "email": "wim1free3schaefer@yahoo.com", "password": "8d1v5idshh", "phone": "+3 326-345-3922", "userStatus": 0},
    "label": "invalid", "data": {
      "id": 1374, "username": "Domenick Glover", "firstName": "Magdalena", "lastName": "Ortiz", "email": "lajson19@yahoo.com", "password": "y4hxytghwq", "phone": "+3 347-456-316-0148", "userStatus": 0},
    "label": "valid", "data": {
      "id": 8592, "username": "Robert Satterfield", "firstName": "Jo", "lastName": "Muelten", "email": "elody1muelten@hotmail.com", "password": "bkainfgq3j", "phone": "+3 889-689-9945", "userStatus": 1},
    "label": "valid", "data": {
      "id": 1556, "username": "Cleve Krist", "firstName": "Vergie", "lastName": "Klein", "email": "vmo4k1eismotmail.com", "password": "hgdtrm45d", "phone": "+3 584-317-2177", "userStatus": 1},
    "label": "valid", "data": {
      "id": 4386, "username": "Terrill Green", "firstName": "Arvid", "lastName": "Cronin", "email": "wman.cronin@gmail.com", "password": "8ra5f312ca", "phone": "+3 224-897-3784", "userStatus": 0},
    "label": "valid", "data": {
      "id": 8149, "username": "Pansy Schum", "firstName": "Alexander", "lastName": "Hoppe", "email": "hagerna.hoppe@yahoo.com", "password": "fmhmadfsc", "phone": "+3 314-118-4488", "userStatus": 0},
    "label": "valid", "data": {
      "id": 4764, "username": "Junior Crist", "firstName": "Jonathan", "lastName": "Koutkowsk", "email": "wrlm3koutkowsk@hotmail.com", "password": "w5yll13agz", "phone": "+3 965-324-8837", "userStatus": 1},
    "label": "invalid", "data": {
      "id": 1570, "username": "Brook Morissette", "firstName": "Kara", "lastName": "Keeling", "email": "roosevelt.keeling@yahoo.com", "password": "h56xz34@gmail.com", "phone": "ctab2f1e6", "userStatus": 0},
    "label": "invalid", "data": {
      "id": 3559, "username": "Cassandre Senger", "firstName": "Richie", "lastName": "Pourros", "email": "roosevelt.pourros2@yahoo.com", "password": "Flatley", "phone": "emaline.flatley@yahoo.com", "userStatus": 0},
    "label": "invalid", "data": {
      "id": 4709, "username": "Manuela Windler", "firstName": "Zora", "lastName": "Hermann", "email": "lmanuel.hermann7@gmail.com", "password": "Schmidt", "phone": "emell1schmidt@yahoo.com", "userStatus": 0},
    "label": "invalid", "data": {
      "id": 5934, "username": "Vivien Marks", "firstName": "Eldridge", "lastName": "Nolan", "email": "vita.nolan8@hotmail.com", "password": "yid4xlc.Rodriguez", "phone": "Miller", "userStatus": 0},
    "label": "invalid", "data": {
      "id": 3722, "username": "Robert Weissnat", "firstName": "Blanca", "lastName": "Rintz", "email": "l1j"},
    "label": "valid", "data": {
      "id": 4770, "username": "Levon Carter", "firstName": "Barrett", "lastName": "Hudson", "email": "mardita.hudson88@hotmail.com", "password": "9e7xydtgb2", "phone": "+3 461-309-8199", "userStatus": 1},
    "label": "valid", "data": {
      "id": 8530, "username": "Sandra Krajcik", "firstName": "Hank", "lastName": "Mlenke", "email": "lajaythe.1enke@hotmail.com", "password": "819725-659kd3", "phone": "+3 895-996-8896", "userStatus": 1},
    "label": "valid", "data": {
      "id": 7392, "username": "Cullen Turner", "firstName": "Rosanna", "lastName": "Grimes", "email": "nik143m@hotmail.com", "password": "c3cwa19du6", "phone": "+3 259-580-2628", "userStatus": 0},
    "label": "valid", "data": {
      "id": 7252, "username": "Zander Konopolski", "firstName": "Emory", "lastName": "Nolan", "email": "maurice.howe@yahoo.com", "password": "480m5918t", "phone": "+3 241-845-5191", "userStatus": 1},
    "label": "valid", "data": {
      "id": 5255, "username": "Lourdes Ondricka", "firstName": "Norval", "lastName": "Murazik", "email": "maceira45@gmail.com", "password": "mb8omcha5", "phone": "+3 221-831-8384", "userStatus": 0},
    "label": "valid", "data": {
      "id": 5167, "username": "Candace Ruecker", "firstName": "Winifred", "lastName": "Ernsner", "email": "c1cercr.ernser@hotmail.com", "password": "4k3nqbh27", "phone": "+3 642-384-8726", "userStatus": 1},
    "label": "valid", "data": {
      "id": 9414, "username": "Julius Howell", "firstName": "Leola", "lastName": "Koch", "email": "d4dota.koch@yahoo.com", "password": "zuzst4d4rm", "phone": "+3 581-121-5029", "userStatus": 1},
    "label": "valid", "data": {
      "id": 3342, "username": "Nash Deckow", "firstName": "Kelisi", "lastName": "Bins", "email": "maxwell32@yahoo.com", "password": "k4f3fip12l", "phone": "+3 944-855-8368", "userStatus": 1},
    "label": "valid", "data": {
      "id": 8857, "username": "Selmer Hauck", "firstName": "Quinn", "lastName": "Okunev", "email": "krystal.okunev@gmail.com", "password": "rajbrud2em", "phone": "+3 845-429-1950", "userStatus": 0},
    "label": "valid", "data": {
      "id": 3163, "username": "Maricella Reynolds", "firstName": "Arianna", "lastName": "Hinz", "email": "danny.hinz10@gmail.com", "password": "brad82ygt", "phone": "+3 685-860-8318", "userStatus": 0},
    "label": "invalid", "data": {
      "id": 3451, "username": "Marcelle Kiehn", "firstName": "Marlan", "lastName": "Kunde", "email": "zachariah31@yahoo.com", "password": "019zscbhc", "phone": "31@gmail.com", "userStatus": 0},
    "label": "invalid", "data": {
      "id": 5938, "username": "Ricardo Strawn", "firstName": "Rene", "lastName": "Kuhlman", "email": "r1ccrercr.suhlman@gmail.com", "password": "1189y9hoo.com", "phone": "qynr3zaqz", "userStatus": 0},
    "label": "valid", "data": {
      "id": 5939, "username": "Orpin O'kon", "firstName": "Aurelie", "lastName": "Sperry", "email": "hastive.sperry@gmail.com", "password": "r8p4d4e4z", "phone": "+3 255-346-0843", "userStatus": 1},
    "label": "invalid", "data": {
      "id": 1787, "username": "Jannie Kreiger", "firstName": "Regan", "lastName": "Barrows", "email": "maba.barrows@yahoo.com", "password": "59d912126v", "phone": "+3 191- Folkman", "userStatus": 0},
    "label": "invalid", "data": {
      "id": 182, "username": "Mackenzie Murphy", "firstName": "Osborne", "lastName": "Heaney", "email": "l0mbay3.heaney@gmail.com", "password": "gmyahoo.com", "phone": "d6j9ym4cip", "userStatus": 0},
    "label": "invalid", "data": {
      "id": 419, "username": "Lessie", "firstName": "Zlame", "lastName": "Hcobal334", "email": "+3 894-759-7784", "password": "v"},
    "label": "invalid", "data": {
      "id": 8288, "username": "Shany Cumerata", "firstName": "Johann", "lastName": "Toy", "email": "totc2m81jd", "password": "+3 987-633-7277", "phone": "11"},
    "label": "valid", "data": {
      "id": 1844, "username": "Eden Komer", "firstName": "Deanna", "lastName": "Corvin", "email": "lajaythe104@gmail.com", "password": "5t1drdcv", "phone": "+3 885-295-6848", "userStatus": 0},
    "label": "valid", "data": {
      "id": 591, "username": "Dion Flatley", "firstName": "Lana", "lastName": "Burton", "email": "cgrist88@hotmail.com", "password": "Vukxod8qc", "phone": "+3 335-718-6663", "userStatus": 0},
    "label": "valid", "data": {
      "id": 2172, "username": "Wilbert Towne", "firstName": "Lemuel", "lastName": "Runte", "email": "alaysia29@hotmail.com", "password": "hcrpnbpx74", "phone": "+3 898-783-6561", "userStatus": 0},
    "label": "valid", "data": {
      "id": 5737, "username": "Dallin Grady", "firstName": "Elvo", "lastName": "Lubowitz", "email": "gussie6390@gmail.com", "password": "lthux7w4k", "phone": "+3 181-387-2698", "userStatus": 1},
    "label": "valid", "data": {
      "id": 4159, "username": "Mohamed Olson", "firstName": "Coty", "lastName": "Wilpert", "email": "cathleen36@gmail.com", "password": "169y1w48r", "phone": "+3 217-363-5124", "userStatus": 1},
    "label": "invalid", "data": {
      "id": 0, "username": "Molly", "firstName": "Schaefer", "lastName": "leahlab.schaefer18@gmail.com", "email": "01a88n3cus", "password": "+3 788-918-1593", "phone": "n"}},
    "label": "valid", "data": {
      "id": 5885, "username": "Garnett Champlin", "firstName": "Oleta", "lastName": "Ward", "email": "mdu1.ward@yahoo.com", "password": "zokx0lbu4d", "phone": "+3 399-354-5959", "userStatus": 1},
    "label": "valid", "data": {
      "id": 569, "username": "Hilbert Emerich", "firstName": "Kathrine", "lastName": "Howell", "email": "kathrine.howell1@yahoo.com", "password": "hrrt733jz", "phone": "+3 868-368-3477", "userStatus": 0}
  }
}
```

Рис 2.3. Результат генерації синтетичних тестових даних у форматі JSON

Висновки до розділу 2

Отже, у цьому розділі було розглянуто застосування попередньо тренованих трансформерів як важливого інструмента для обробки природної мови та генерації текстів а також писано трансформерну архітектуру - тип глибоких нейронних мереж, розроблений для обробки послідовних даних, зокрема тексту. Особливу увагу було приділено механізму уваги (attention mechanism), який дозволяє моделі враховувати довгі залежності та створювати більш контекстно-залежні представлення тексту.

Додатково було розглянуто математичне підґрунтя механізму уваги, включаючи обчислення ваги для кожного слова в послідовності та використання цих ваг для створення контекстних представлень а також описано процес розробки та впровадження системи для генерації тестових даних з використанням моделі GPT (Generative Pre-trained Transformer). Ця модель була

вибрана через її здатність автоматизувати процес генерації та підвищити охоплення тестів. Детально описано кроки використання GPT для генерації даних, починаючи від формулювання вхідного запиту і закінчуючи інтеграцією згенерованих даних у процес тестування API. Кожен крок має свою специфіку і важливість у контексті загального завдання.

Ці висновки підкреслюють значущість застосування сучасних технологій штучного інтелекту, зокрема попередньо тренованих трансформерів та моделі GPT, для ефективної генерації тестових даних, що може значно покращити процеси автоматизованого та мануального тестування.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ МЕТОДІВ ГЕНЕРАЦІЇ ТЕСТОВИХ ДАНИХ З ВИКОРИСТАННЯМ ШТУЧНОГО ІНТЕЛЕКТУ

3.1 Використання згенерованих даних за допомогою GPT-3.5 Turbo у автоматизованому тестуванні

Сучасні веб-додатки нерозривно пов'язані зі складними API (інтерфейсами програмування застосунків), які грають ключову роль у взаємодії клієнтської та серверної частини системи. Забезпечення правильності та стабільності цих API стає критичним завданням для розробників та QA-інженерів. У цьому розділі ми розглянемо використання інструменту Cypress для автоматизованого тестування API з використанням даних, згенерованих GPT-3.5 Turbo. Cypress - це потужний фреймворк для автоматизованого тестування веб-додатків, який поєднує в собі інструменти для тестування як клієнтської, так і серверної частини додатків [33].

У наступному лістингу визначено тіло запиту для взаємодії з API OpenAI, використовуючи бібліотеку Cypress. Давайте розглянемо кожен блок цього коду:

Лістинг 3.1

Опис тестового випадку

```
describe('GPT API Test', () => {  
  it('Name is positive value', () => {
```

Цей блок починає новий тестовий випадок з назвою "Name is positive value". Тестуємо генерацію позитивних значень для параметра "name", який є рядком.

Лістинг 3.2

Оголошення змінних API-ключа та URL

```
const openAIApiKey = 'key';  
const openAIApiUrl = 'https://api.openai.com/v1/chat/completions';
```

Оголошення констант openAIApiKey та openAIApiUrl для зберігання ключа API та URL для виклику API OpenAI.

Створення тіла запиту

```
const requestBody = {
  model: 'gpt-3.5-turbo-1106',
  messages: [
    {
      role: 'system',
      content: 'You are an assistant for QA engineer and you need
to provide the test data for API requests. Important rule is to return
ONLY values, no parameters in JSON objects in content. DO NOT USE
RUSSIAN.',
    },
    {
      role: 'user',
      content: 'Generate the array with positive values for the
name which is a string. List of valid Names: could contain single,
double names with -, double names with space, names could be 3 types of
languages - cyrillic, latin, with or without diacritics.',
    },
  ],
};
```

Створення об'єкта `requestBody`, який представляє тіло запиту до API. Визначення ролі (`system/user`) та контенту повідомлень для взаємодії з OpenAI API. Також зазначаємо модель, яку будемо використовувати для генерації тексту - `'gpt-3.5-turbo-1106'`. Для передачі вхідного повідомлення для моделі створюється масив повідомлень (`messages`), що містить системне та користувацьке повідомлення.

Системне повідомлення визначається роллю (`'role'`): `'system'` та складається з контенту (`'content'`): "You are an assistant for QA engineer and you need to provide the test data for API requests. Important rule is to return ONLY values, no parameters in JSON objects in content." Системне повідомлення встановлює контекст асистента та важливе правило щодо структури даних у відповіді.

Повідомлення користувача визначається роллю ('role'): 'user' та складається з контенту ('content'): "generate the array with positive values for the name which is a string. List of valid Names: could contain single, double names with -, double names with space, names could be 3 types of languages - cyrillic, latin, with or without diacritics." Користувацьке повідомлення встановлює завдання асистентові щодо генерації масиву з позитивними значеннями для параметра "name", описуючи при цьому різні варіанти даних.

Контент для системного повідомлення та повідомлення користувача може змінюватись в залежності від необхідних даних для тестового випадку. Весь цей контент об'єднується в один об'єкт requestBody.

Лістинг 3.4

POST-запит до API OpenAI

```
cy.request({
  method: 'POST',
  url: openAIApiUrl,
  headers: {
    'Content-Type': 'application/json',
    'Authorization': `Bearer ${openAIApiKey}`,
  },
  body: requestBody,
})
```

Використовуємо команду cy.request() для виклику API OpenAI. Встановлення методу POST, URL, та заголовків, включаючи Content-Type та Authorization.

Лістинг 3.5

Згенерований масив даних для подальшого використання у тестах

```
[
  "John",
  "Maria",
  "José",
  "Александр",
  "André",
  "Иванов-Петров",
  "Anna Maria",
```

```
"Γεώργιος",  
"Márquez"  
]
```

Після успішного виклику API OpenAI, зосередитись на обробці відповіді та подальших діях. Використовуємо метод `.then()` для взаємодії з результатом виклику API та виконання певних дій.

Лістинг 3.6

Обробка відповіді від OpenAI API

```
expect(response.status).to.equal(200);
```

Лістинг 3.7

Отримання згенерованого контенту

```
generatedContent = response.body.choices[0].message.content;  
cy.log('Generated Content:', generatedContent);
```

Використовується змінна `generatedContent` для зберігання згенерованого тексту `response.body.choices[0].message.content`.

Лістинг 3.8

Обробка та використання згенерованого контенту

```
const parsedContent = JSON.parse(generatedContent);
```

Згенерований текст перетворюється з JSON у зручний формат для подальшого використання.

Лістинг 3.9

Використання згенерованих даних для POST-запитів

```
cy.wrap(parsedContent).each((name, index) => {  
  const apiUrl =  
'https://fakerestapi.azurewebsites.net/api/v1/Users';  
  const requestBody = {  
    id: index + 1,  
    userName: name,  
    password: 'string',  
  };
```

Для кожного згенерованого імені виконується POST-запит до іншого API (<https://fakerestapi.azurewebsites.net/api/v1/Users>). Кожен POST-запит містить інформацію, яку ми отримали від OpenAI API.

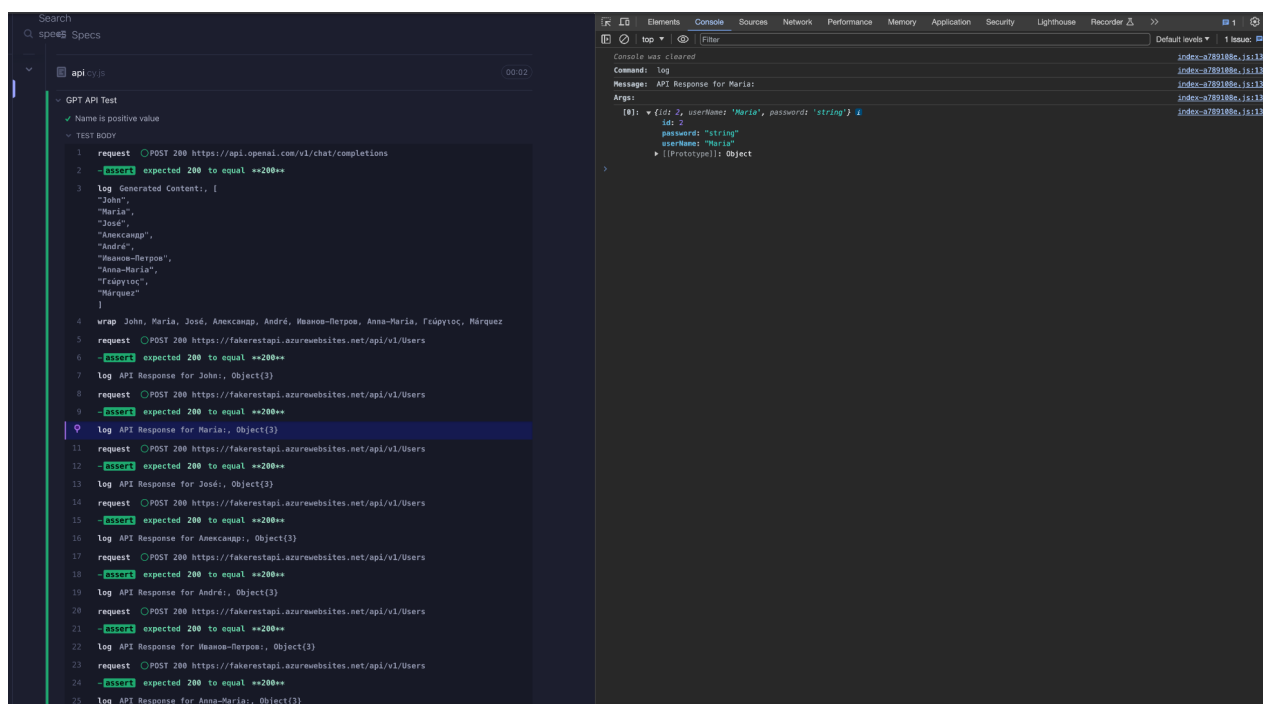
Лістинг 3.10

Виклик POST-запиту та оцінка відповіді

```
cy.request({
  method: 'POST',
  url: apiUrl,
  body: requestBody,
}).then((response) => {
  expect(response.status).to.equal(200);
  cy.log(`API Response for ${name}:`, response.body);
});
```

Здійснюється POST-запит на вказаний API з переданим тілом запиту а також перевіряється, що статус відповіді є 200 (OK). Логується відповідь для подальшого аналізу.

Цей алгоритм дій дозволяє тестувальнику ефективно використовувати результати генерації для взаємодії з іншими системами та проведення динамічних тестів, при цьому уникати ефекту пестициду при запуску тестування.



3.2 Використання згенерованих даних за допомогою RNN у автоматизованому тестуванні

Сьогодні вимоги до якості програмного забезпечення стають все більшими, оскільки вони визначають вплив на функціонування організацій, а також на безпеку та досвід користувачів. Отже, використання автоматизованого тестування є важливою стратегією для забезпечення надійності та функціональності програмного забезпечення.

В цьому контексті, робота ґрунтується на основах сучасних досліджень у галузі тестування програмного забезпечення, використовуючи методи машинного навчання, автоматизованого генерування даних та валідації. Вона спрямована на зниження ризику помилок та витрат, пов'язаних з тестуванням, та підвищення якості та стабільності наших програмних продуктів.

Розглянемо реалізацію методології автоматизованого тестування API, яка ґрунтується на сучасних підходах до забезпечення якості програмного забезпечення. Він був розроблений з метою створення ефективного інструмента для генерації, відправлення та валідації тестових даних до API. Цей процес автоматизованого тестування має велике значення для підтвердження правильності функціонування API, а також для виявлення помилок та проблем у великих обсягах даних.

Маркування позитивних та негативних тестів

Лістинг 3.11

Функція `validate_record(record)`

```
def validate_record(record):  
    try:  
        print(record)  
        validate(instance=record, schema=json_schema)  
        return 'valid'  
    except ValidationError as e:  
        return 'invalid'
```

Ця функція виконує наступні дії:

1. Виконає спробу здійснити валідацію запису (record) за допомогою заданої JSON-схеми (json_schema). Це відбувається за допомогою функції validate з бібліотеки jsonschema.
2. Якщо валідація успішна і запис відповідає схемі, то функція повертає рядок 'valid' для позначення того, що запис є валідним.
3. У випадку, якщо є помилка валідації (запис не відповідає схемі), вона перехоплює виключення ValidationError. Коли це відбувається, функція повертає рядок 'invalid' для позначення невалідного запису.
4. Функція generate_and_validate_records використовує функцію validate_record для валідації згенерованих даних перед відправленням їх на сервер Flask та подальшою обробкою.
5. Функція generate_and_validate_records допомагає автоматизувати процес генерації, валідації та тестування даних для вашого API або програмного забезпечення, забезпечуючи зручний спосіб створення тестових сценаріїв та збереження результатів.

Лістинг 3.12

Функція generate_and_validate_records

```
def generate_and_validate_records(line_validator=None,
num_lines=100):
    results = []
    for gen_text_obj in generate_text(config,
line_validator=line_validator, num_lines=num_lines):
        line = gen_text_obj.text
        record = dict(zip(["id", "username", "firstName",
"lastName", "email", "password", "phone", "userStatus"],
line.split(",")))
        mapped_record = gen_mapped_result(record)
        result = {'label': validate_record(mapped_record), 'data':
mapped_record}
        results.append(result)
    return results
```

Функція generate_and_validate_records виконує наступні дії:

1. `results = []`: Цей рядок ініціалізує порожній список `results`, в якому будуть зберігатися результати операцій.
2. `for gen_text_obj in generate_text(config, line_validator=line_validator, num_lines=num_lines)`: Цикл `for` перебирає об'єкти `gen_text_obj`, які генеруються функцією `generate_text`. Цей цикл генерує та обробляє текстові рядки.
3. `line = gen_text_obj.text`: Для кожного об'єкта `gen_text_obj` отримуємо текстовий рядок, який був згенерований функцією `generate_text`.
4. `record = dict(zip(["id", "username", "firstName", "lastName", "email", "password", "phone", "userStatus"], line.split(",")))`: Рядок `line` розбивається на окремі частини, використовуючи `,` як роздільник, і створюється словник `record`, де ключіми є імена полів (наприклад, `"id"`, `"username"`, тощо), а значеннями - відповідні значення з рядка.
5. `mapped_record = gen_mapped_result(record)`: Функція `gen_mapped_result` використовується для подальшої обробки запису `record`, який містить дані. Це може включати в себе перетворення типів або інші операції, залежно від потреб.
6. `result = {'label': validate_record(mapped_record), 'data': mapped_record}`: Створюється словник `result`, де `'label'` - це результат виклику функції `validate_record(mapped_record)`, яка валідує запис `mapped_record`, а `'data'` - сам `mapped_record`. Важливою частиною тут є валідація запису перед подальшою обробкою.
7. `results.append(result)`: Отриманий результат додається до списку `results`, що дозволяє зберігати всі оброблені результати.
8. `return results`: Після завершення циклу повертається список `results`, який містить всі оброблені та валідовані дані.

Отже, ця функція генерує текстові дані, розбиває їх на окремі поля, виконує валідацію кожного запису та повертає результати цих операцій у вигляді списку словників.

Створення віддаленого серверу для отримання тестових даних

Flask і ngrok - це інструменти, які дозволяють створити API-інтерфейс для отримання тестових даних та надають можливість автоматизованого тестування. Flask - це мікро фреймворк для створення веб-додатків на мові програмування Python, тоді як ngrok - це інструмент для створення публічних ngrok тунелів для локальних серверів.

Ролі компонентів:

- Flask: Flask використовується для створення веб-додатку та визначення API-маршрутів, які дозволяють отримувати тестові дані через HTTP запити. Flask надає зручний інтерфейс для обробки вхідних запитів та відправки відповідей.

- Ngrok: Ngrok дозволяє створювати публічні тунелі, які роблять локальний сервер доступним через Інтернет. Це особливо корисно для тестування, оскільки дозволяє зовнішнім користувачам отримувати доступ до вашого локального сервера, навіть якщо він працює на вашому комп'ютері.

Процес використання:

1. Спочатку створюється веб-додаток за допомогою Flask. Визначаються маршрути, які використовуються для отримання тестових даних через API.

2. Потім запускається ngrok для створення публічного тунелю. Цей тунель дозволяє зовнішнім користувачам отримувати доступ до локального сервера через Інтернет. Ngrok надає публічну URL-адресу для цього тунелю.

3. Веб-додаток Flask стає доступним за публічною ngrok URL-адресою. За допомогою цієї URL-адреси зовнішні користувачі можуть здійснювати запити до API та отримувати тестові дані.

4. Це API може бути використано для автоматизованого тестування програмного забезпечення. Скрипти тесту можуть виконувати запити до API, отримувати тестові дані та перевіряти правильність їх обробки.

5. В результаті отримується зручний спосіб для створення та тестування API з використанням тестових даних, що генеруються, що сприяє розвитку та підтримці програмного продукту.

Запуску Flask сервера та створення ngrok тунелю

```
import os
import threading
from flask import Flask, request
from pyngrok import ngrok

app = Flask(__name__)
port = "5000"

public_url = ngrok.connect(port).public_url
print(" * ngrok tunnel \"{}\" ->
      \"http://127.0.0.1:{}\".format(public_url, port))

app.config["BASE_URL"] = public_url

@app.route('/', methods=['GET'])
def index():
    if 'qty' in request.args:
        qty = request.args['qty']
        try:
            qty = float(qty)
            return generate_and_validate_records(None, qty)
        except ValueError:
            return 'Param should be a number', 400
    else:
        return generate_and_validate_records(None, 10)

app.run(use_reloader=False)
```

Скрипт починається з імпорту необхідних модулів і ініціалізації додатку Flask. Встановлюється порт для сервера Flask, який призначається номер 5000. Далі використовується ngrok для створення публічної URL-адреси, яка спрямовується на локальний сервер Flask. Це дає можливість іншим користувачам отримувати доступ до сервера через Інтернет.

Публічна URL-адреса ngrok виводиться на консоль і також зберігається у конфігурації додатку Flask під ключем "BASE_URL". Скрипт визначає маршрут для додатку Flask, доступний за шляхом '/'. Цей маршрут обробляє GET-запити.

Коли до цього маршруту надходить GET-запит, скрипт перевіряє, чи присутній параметр 'qty' у запиті. Якщо він є, скрипт намагається перетворити

його значення у десяткове число (float). Якщо перетворення вдається, він викликає функцію `generate_and_validate_records` з отриманим значенням 'qty'. Якщо перетворення не вдається, він повертає статус 400 з повідомленням про те, що 'qty' повинно бути числом.

Якщо параметр 'qty' відсутній у запиті, скрипт також викликає функцію `generate_and_validate_records` зі значенням за замовчуванням. Нарешті, скрипт запускає сервер Flask. Аргумент `use_reloader` встановлюється на `False`, щоб запобігти перезапуску сервера при виявленні змін у вихідних файлах.

Для проведення тестування ендпоінту веб сервера використовується програма Postman. Postman - це універсальний інструмент для розробників, який надає зручний інтерфейс для взаємодії з веб серверами та тестування їхніх API.

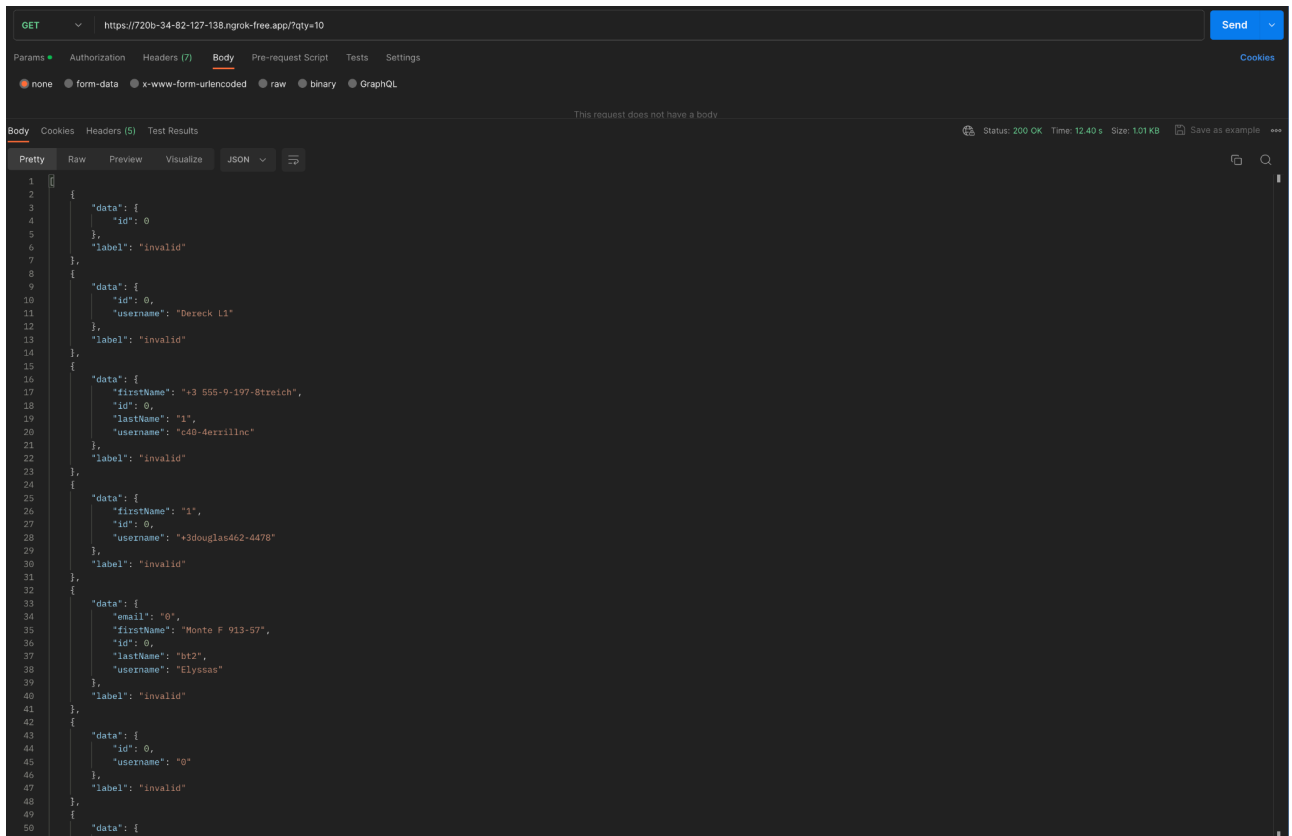


Рис 3.2. Результат виконання тестів

Використання тестових даних в автоматизованих тестах

Розглянемо тестовий випадок, написаний на JavaScript з використанням тестового фреймворку Cypress. Цей тест перевіряє поведінку GET-запиту до певного URL. Важливо відзначити, що дані, які були згенеровані раніше,

використовуються в цьому тестовому випадку для взаємодії з сервером за допомогою GET-запиту.

Лістинг 3.14

Тестовий випадок 'Synthetic data test'

```
const itemsQty = 3;

it.only('Synthetic data test', () => {
  const syntheticDataGeneratorURL =
`https://ff0b-34-125-142-240.ngrok-free.app?qty=${itemsQty}`

  cy.request({
    method: 'GET',
    url: syntheticDataGeneratorURL
  }).then((response) => {
    expect(response.status).to.equal(200);
    expect(response.body).to.have.lengthOf.above(0);
    console.log(response.body)
    Cypress.env('responseData', response.body);
  });
})
```

Давайте розглянемо тестовий випадок “Synthetic data test” детальніше:

1. `it.only('Name is negative value', () => {...})` - цей рядок визначає тестовий випадок з описом 'Name is negative value'. Цей тестовий випадок використовується для перевірки поведінки GET-запиту до сервера.

2. `const syntheticDataGeneratorURL = https://ff0b-34-125-142-240.ngrok-free.app?qty=${itemsQty}`;` - цей рядок визначає константу `syntheticDataGeneratorURL`, яка представляє URL, на який буде відправлено GET-запит. Змінна `itemsQty` використовується для вставки кількості елементів як параметра запиту.

3. `cy.request({...})` - ця функція використовується для відправки HTTP-запиту. У даному випадку метод встановлюється на 'GET', а URL - на `syntheticDataGeneratorURL`.

4. `.then((response) => {...})` - ця функція викликається після завершення запиту і отримує відповідь як аргумент.

5. `expect(response.status).to.equal(200);` - цей рядок перевіряє, що статус відповіді є 200, що вказує на успішну відповідь сервера.

6. `expect(response.body).toHaveLength.above(0);` - цей рядок перевіряє, що тіло відповіді має довжину більше 0, тобто не є порожнім.

7. `console.log(response.body)` - цей рядок виводить тіло відповіді в консоль, що може бути корисним для відлагодження.

8. `Cypress.env('responseData', response.body);` - цей рядок зберігає тіло відповіді як глобальну змінну з іменем 'responseData' - це дозволяє іншим тестовим випадкам використовувати дані відповіді для подальших перевірок і взаємодії з сервером.

Розглянемо фрагмент коду, який представляє собою тестовий випадок, призначений для взаємодії з API і перевірки правильності відповідей сервера на конкретні запити.

Лістинг 3.15

Цикл відправки запитів з використанням згенерованих тестових даних

```
for(let i = 0; i < itemsQty; i++) {  
  it.only(`Test ${i}`, () => {  
    let currentItem = Cypress.env('responseData')[i]  
    cy.request({  
      url: 'https://fakerestapi.azurewebsites.net/api/v1/Users',  
      method: 'POST',  
      body: currentItem.data,  
    }).then((response) => {  
      console.log(response)  
      if (currentItem.label === 'valid') {  
        expect(response.status).to.eq(200);  
      } else if (currentItem.label === 'invalid') {  
        expect(response.status).to.eq(400);  
      }  
    });  
  });  
}
```

На лістингу представлено цикл, який виконує серію тестів. Кожен тест відправляє POST-запит на певний URL і перевіряє статус відповіді. Метою цих тестів є перевірка правильності відповідей сервера на POST-запити та впевненість, що відповіді мають відповідні статуси в залежності від типу

згенерованих даних маркованих мітками "valid" чи "invalid", які була збережені в попередньому тестовому випадку.

Розглянемо тестовий випадок детальніше:

- `for(let i = 0; i < itemsQty; i++) {...}` - цей цикл виконує серію тестів.

Кількість тестів визначається змінною `itemsQty`.

- `it.only(Test ${i}, () => {...})` - цей рядок визначає тестовий випадок.

Функція `it.only` використовується для визначення тестового випадку, який повинен виконуватися ізольовано, що означає, що всі інші тестові випадки будуть пропущені. Опис тестового випадку генерується динамічно на основі поточної ітерації циклу.

- `let currentItem = Cypress.env('responseData')[i]` - цей рядок отримує *i*-й елемент з `'responseData'`, яке зберігається в середовищі `Cypress`. Ці дані, ймовірно, були встановлені в попередньому тестовому випадку.

- `cy.request({...})` - ця функція використовується для відправлення HTTP-запиту. Метод встановлюється на `'POST'`, URL встановлюється на `'https://fakereapi.azurewebsites.net/api/v1/Users'`, а тіло запиту встановлюється на `currentItem.data`.

- `.then((response) => {...})` - ця функція викликається, коли запит завершено, і отримує відповідь у вигляді аргумента.

- `if (currentItem.label === 'valid') {...} else if (currentItem.label === 'invalid') {...}` - ця умовна інструкція перевіряє мітку поточного елемента. Якщо мітка дорівнює `'valid'`, вона перевіряє, що статус відповіді дорівнює 200. Якщо мітка дорівнює `'invalid'`, вона перевіряє, що статус відповіді дорівнює 400.

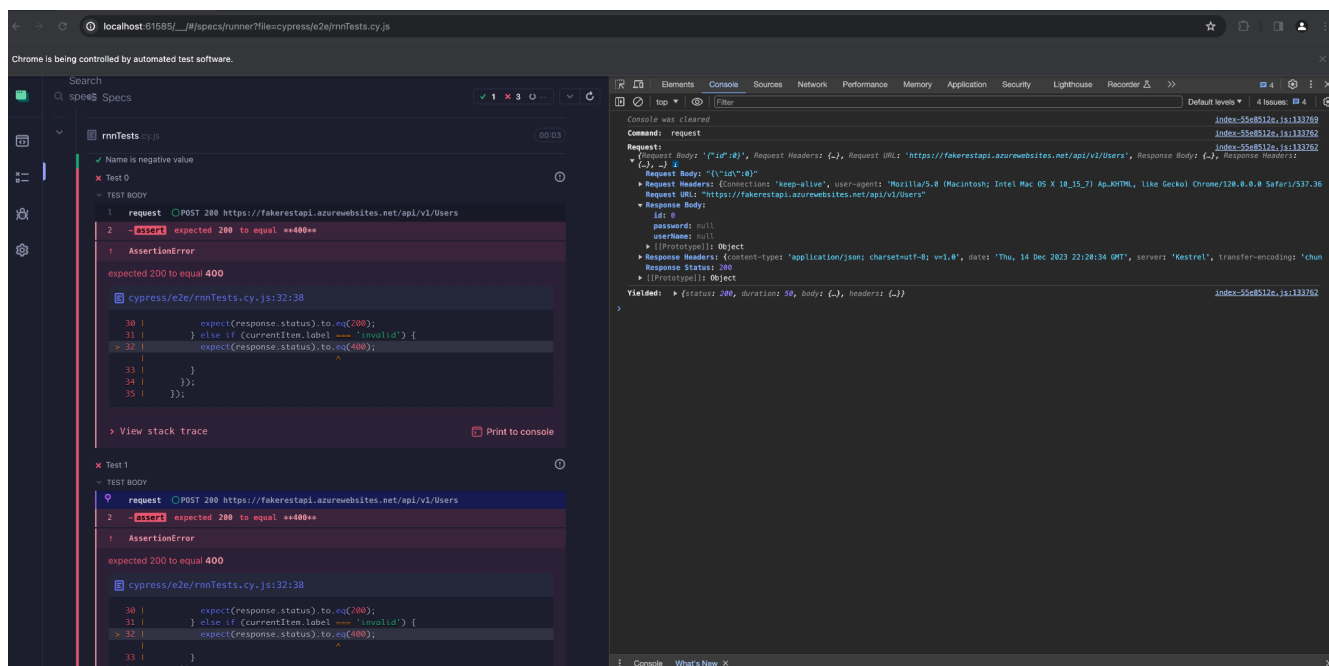


Рис 3.3. Результат виконання тестів

На рисунку можна побачити вікно браузера з відкритим інструментом для автоматизації тестування Cypress. Тести автоматизації визначені як Failed, що видно з червоного тексту та символів "!" поруч з назвами тестів. Це вказує на те, що під час тестування очікувані результати не відповідають фактичним. На рисунку 3.3 показано два тести, обидва з помилками (failed tests). Кожен тест намагається зробити HTTP POST запит до API за адресою `https://fakereapi.azurewebsites.net/api/v1/Users`. Тести містять перевірку (assertion), де очікується, що статус-код відповіді буде 400, що зазвичай вказує на помилку клієнта, наприклад невірні дані запиту. В обох тестах фактичний статус-код відповіді є 200, який є стандартним кодом успіху для HTTP запитів, що означає, що запит був успішно оброблений сервером. У вікні консолі (праворуч внизу) ми бачимо деталі виконання запиту, де 'status: 200' підтверджує, що сервер повернув статус-код 200. Ця невідповідність між очікуваним (400) та отриманим (200) статус-кодами вказує на те, що API не поводить себе так, як очікувалося в ситуаціях, коли повинна бути повернута помилка (можливо через невалідні дані у тілі запиту).

Підсумовуючи, використання згенерованих даних за допомогою рекурентної нейронної мережі для тестування API може бути дуже ефективним

для виявлення помилок та недоліків. Згенеровані дані можуть допомогти ідентифікувати вразливі місця в обробці даних API, такі як неправильна валідація або обробка помилок. Рекурентні нейронні мережі здатні генерувати широкий спектр даних, які можуть імітувати різні випадки використання, включаючи нестандартні та нетривіальні випадки, які можуть не бути враховані при ручному тестуванні.

3.3 Порівняння можливостей RNN та GTP з GAN (Generative Adversarial Networks)

RNN - це клас нейронних мереж, який ефективно обробляє послідовні дані. Відрізняючись від звичайних глибоких нейронних мереж, RNN має петлі у своїй архітектурі, дозволяючи інформації передаватися від одного кроку мережі до наступного. Ця властивість робить RNN ідеальною для завдань, де необхідно враховувати контекст, таких як обробка мови або генерація тексту.

Таблиця 3.1

Порівняння можливостей RNN та GTP з GAN

N	Критерій / Модель ШІ	GPT	RNN	GAN
1.	Основна архітектура	Трансформер	Рекурентна нейронна мережа	Конфліктні мережі (генератор та дискримінатор)
2	Обробка послідовностей	Висока ефективність у великих контекстах	Ефективна у коротших послідовностях	Не призначені для обробки послідовностей
3.	Масштабованість	Висока масштабованість	Обмежена масштабованість	Залежить від комплексності мережі
4	Пам'ять та контекст	Довгострокове збереження контексту	Обмежена пам'ять контексту	Не зберігають контекст
5	Часова складність обчислень	Висока	Середня	Висока (залежить від складності моделі)
6	Застосування	Генерація тексту, переклад, відповіді на питання	Обробка мови, генерація тексту	Обробка мови, генерація тексту Генерація зображень, відео, аудіо Класифікація, регресія

...

Вибір GPT та RNN для порівняння в порівняльній таблиці(див.табл.3.1) визначається кількома ключовими факторами. GPT, як одна з найбільш інноваційних моделей в області штучного інтелекту, представляє собою передову технологію в генерації тексту. Її здатність генерувати змістовний, когерентний та креативний контент робить її ідеальною для аналізу та порівняння з іншими технологіями. GPT демонструє вражаючу гнучкість в різних застосуваннях, починаючи від генерації тексту до складніших завдань, таких як переклад, відповіді на питання та інші.

RNN, з іншого боку, є фундаментальною технологією в обробці послідовних даних, зокрема в областях, де важливий контекст та послідовність інформації. Її особлива здатність до зберігання інформації з попередніх кроків робить її незамінною у таких завданнях, як обробка мови, прогнозування на основі часових рядів та генерація тексту. Особливо це важливо для генерації тестових даних, де потрібно враховувати попередній контекст.

Обидві технології, GPT та RNN, надають унікальні можливості та переваги, що робить їх важливими для порівняльного аналізу. Вивчення та порівняння цих технологій допомагає не лише у розумінні їх потенціалу, але й у визначенні відповідності для конкретних застосувань в галузі штучного інтелекту.

Ця таблиця дає можливість оцінити, як GPT та RNN порівнюються з двома іншими важливими технологіями ШІ. Вона демонструє унікальні переваги кожної технології та підкреслює, що вибір конкретної технології залежить від специфічних потреб та цілей застосування.

Висновки до розділу 3

У цьому розділі було розглянуто використання згенерованих даних у процесі автоматизованого тестування API, з використанням рекурентних нейронних мереж (RNN). Основна мета була спростувати процес створення та використання ефективних інструментів для генерації, відправлення та валідації тестових даних для API. Для досягнення цієї мети була розглянута реалізація

методології автоматизованого тестування, включаючи створення веб-додатків з використанням Flask та ngrok для забезпечення доступу до тестових даних через Інтернет.

Окрема увага була приділена процесу створення та використання функцій для генерації та валідації даних. Ці функції відіграють ключову роль у забезпеченні якості автоматизованих тестів та надійності API.

Загалом, цей розділ надає докладний огляд методології та інструментів, які можна використовувати для автоматизованого тестування API з використанням згенерованих даних. Ця методологія може бути корисною для розробників та тестувальників, які створюють та тестують API з високими вимогами до надійності та якості.

Ось кілька ключових аспектів, що підкреслюють важливість цього підходу:

1. Згенеровані дані дозволяють автоматизовано створювати велику кількість тестових сценаріїв для перевірки різних аспектів API. Це зменшує необхідність у ручному написанні тестів і прискорює процес тестування.
2. Згенеровані дані можуть бути створені таким чином, щоб вони були реалістичними та відображали різні можливі варіанти вхідних даних. Це дозволяє впевнитися, що API взаємодіє з реальними даними належним чином.
3. Рекурентні нейронні мережі можуть створювати послідовності даних, які динамічно змінюються. Це особливо корисно для тестування API, які працюють з динамічними даними, такими як часові ряди або потокові дані.
4. Автоматизоване тестування з використанням згенерованих даних допомагає виявляти помилки та недоліки в роботі API на ранніх етапах розробки. Це дозволяє виправляти проблеми до їх поширення в продукцію.

ВИСНОВКИ

Основні результати, представлені нижче, узагальнюють результати проведеного дослідження, яке було спрямоване на досягнення важливої мети:

розробки та впровадження передових методів генерації тестових даних за допомогою штучного інтелекту для RESTful API. Робота демонструє, що мета була досягнута через ретельний аналіз, експериментальну роботу та інноваційний підхід, що відкриває нові перспективи для ефективності та надійності процесів тестування в сучасному світі веб-розробки. Основні результати цього дослідження включили в себе ряд ключових досягнень:

- розроблено та впроваджено процес генерації тестових даних з використанням штучного інтелекту, що дозволило автоматизувати та удосконалити процес тестування RESTful API. Це забезпечило створення ефективних та реалістичних тестових сценаріїв, знижуючи залежність від мануального тестування.

- розроблено метод використання рекурентних нейронних мереж (RNN) в контексті генерації тестових даних, що підтвердив їх ефективність у створенні високоякісних синтетичних даних, які імітують реальні умови використання API.

- впроваджено застосування синтетичних даних у реальні сценарії тестування, що сприяло забезпеченню більш глибокого та всебічного тестування API.

Для подальших наукових досліджень на основі результатів цієї магістерської роботи рекомендується:

- продовжити розробку та оптимізацію моделей штучного інтелекту для генерації тестових даних, зокрема, досліджуючи новітні підходи в машинному навчанні та їх застосування у тестуванні API.

- провести порівняльні дослідження ефективності та надійності штучного інтелекту порівняно з традиційними методами тестування, використовуючи математичні та статистичні методики для об'єктивного оцінювання.

- розробити детальні рекомендації та керівництва з використання синтетичних даних для різних типів тестування, включаючи стрес-тестування,

безпеку та функціональність, щоб забезпечити широкий спектр покриття в тестуванні..

У підсумку, дана робота розкриває нові можливості для практичного застосування розроблених методів застосування ІІІ і сприяє значному прогресу в сучасних стратегіях тестування програмного забезпечення, особливо у контексті тестування RESTful API.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Вольних І. В. Генерація тестових даних за допомогою штучного інтелекту для RESTful API / Концидайло А.М. // VI Всеукраїнська науково-технічна конференція «Комп'ютерні технології: інновації, проблеми, рішення» 29-30 листопада 2023 р. м. Житомир.
2. Christian, B. (2020). The alignment problem: Machine learning and human values. WW Norton & Company.
3. Maslej, N., Fattorini, L., Brynjolfsson, E., Etchemendy, J., Ligett, K., Lyons, T., ... & Perrault, R. (2023). Artificial intelligence index report 2023. arXiv preprint arXiv:2310.03715.
4. Clapp, J. A. (1995). Software Quality Control, Error, Analysis. William Andrew.
5. Ferrucci, D., Brown, E., Chu-Carroll, J., Fan, J., Gondek, D., Kalyanpur, A. A., ... & Welty, C. (2010). Building Watson: An overview of the DeepQA project. AI magazine, 31(3), 59-79.
6. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. Advances in neural information processing systems, 30.
7. Ketkar, N., Moolayil, J., Ketkar, N., & Moolayil, J. (2021). Introduction to pytorch. Deep Learning with Python: Learn Best Practices of Deep Learning Models with PyTorch, 27-91.
8. Shewalkar, A., Nyavanandi, D., & Ludwig, S. A. (2019). Performance evaluation of deep neural networks applied to speech recognition: RNN, LSTM and GRU. Journal of Artificial Intelligence and Soft Computing Research, 9(4), 235-245.
9. Sohail, S. S., Farhat, F., Himeur, Y., Nadeem, M., Madsen, D. Ø., Singh, Y., ... & Mansoor, W. (2023). Decoding ChatGPT: a taxonomy of existing research, current challenges, and possible future directions. Journal of King Saud University-Computer and Information Sciences, 101675.

10. Martin-Lopez, A. (2020, June). AI-driven web API testing. In Proceedings of the ACM/IEEE 42nd international conference on software engineering: companion proceedings (pp. 202-205).
11. Esnaashari, M., & Damia, A. H. (2021). Automation of software test data generation using genetic algorithm and reinforcement learning. *Expert Systems with Applications*, 183, 115446.
12. Wang, J., & Zhang, C. (2018). Software reliability prediction using a deep learning model based on the RNN encoder–decoder. *Reliability Engineering & System Safety*, 170, 73-82.
13. Rodriguez-Torrealba, R., Garcia-Lopez, E., & Garcia-Cabot, A. (2022). End-to-End generation of Multiple-Choice questions using Text-to-Text transfer Transformer models. *Expert Systems with Applications*, 208, 118258.
14. Wang, J., & Zhang, C. (2018). Software reliability prediction using a deep learning model based on the RNN encoder–decoder. *Reliability Engineering & System Safety*, 170, 73-82.
15. Kolides, A., Nawaz, A., Rathor, A., Beeman, D., Hashmi, M., Fatima, S., ... & Jararweh, Y. (2023). Artificial intelligence foundation and pre-trained models: Fundamentals, applications, opportunities, and social impacts. *Simulation Modelling Practice and Theory*, 126, 102754.
16. Dalalah, D., & Dalalah, O. M. (2023). The false positives and false negatives of generative AI detection tools in education and academic research: The case of ChatGPT. *The International Journal of Management Education*, 21(2), 100822.
17. Xie, Z. (2017). Neural text generation: A practical guide. arXiv preprint arXiv:1711.09534.
18. Єщенко, М. (2020). Автоматизоване тестування мобільних застосунків.
19. Qi, X., Zeng, Y., Xie, T., Chen, P. Y., Jia, R., Mittal, P., & Henderson, P. (2023). Fine-tuning Aligned Language Models Compromises Safety, Even When Users Do Not Intend To!. arXiv preprint arXiv:2310.03693.

20. Andrews, M., & Whittaker, J. A. (2006). How to break web software: Functional and security testing of web applications and web services. Addison-Wesley Professional.
21. Wamba, S. F., Queiroz, M. M., Jabbour, C. J. C., & Shi, C. V. (2023). Are both generative AI and ChatGPT game changers for 21st-Century operations and supply chain excellence?. *International Journal of Production Economics*, 265, 109015.
22. Chen, X., Liu, X., Gales, M. J., & Woodland, P. C. (2015, April). Improving the training and evaluation efficiency of recurrent neural network language models. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (pp. 5401-5405). IEEE.
23. Pelivani, E., & Cico, B. (2021, June). A comparative study of automation testing tools for web applications. In *2021 10th Mediterranean Conference on Embedded Computing (MECO)* (pp. 1-6). IEEE.
24. Moadab, S., & Rashidi, H. (2016). Automatic path-oriented test data generation by boundary hypercuboids. *Journal of King Saud University-Computer and Information Sciences*, 28(1), 82-97.
25. Ong, S. P., Cholia, S., Jain, A., Brafman, M., Gunter, D., Ceder, G., & Persson, K. A. (2015). The Materials Application Programming Interface (API): A simple, flexible and efficient API for materials data based on REpresentational State Transfer (REST) principles. *Computational Materials Science*, 97, 209-215.
26. Casola, S., Lauriola, I., & Lavelli, A. (2022). Pre-trained transformers: an empirical comparison. *Machine Learning with Applications*, 9, 100334.
27. Singh, P., Manure, A., Singh, P., & Manure, A. (2020). Neural networks and deep learning with TensorFlow. *Learn TensorFlow 2.0: Implement Machine Learning and Deep Learning Models with Python*, 53-74.
28. Zafar, I., Tzanidou, G., Burton, R., Patel, N., & Araujo, L. (2018). Hands-on convolutional neural networks with TensorFlow: Solve computer vision problems with modeling in TensorFlow and Python. Packt Publishing Ltd.

29. Sheikh, I., Vincent, E., & Illina, I. (2024). Training RNN language models on uncertain ASR hypotheses in limited data scenarios. *Computer Speech & Language*, 83, 101555.
30. Pang, B., Nijkamp, E., & Wu, Y. N. (2020). Deep learning with tensorflow: A review. *Journal of Educational and Behavioral Statistics*, 45(2), 227-248.
31. Pezoa, F., Reutter, J. L., Suarez, F., Ugarte, M., & Vrgoč, D. (2016, April). Foundations of JSON schema. In *Proceedings of the 25th international conference on World Wide Web* (pp. 263-273).
32. Spillner, A., & Linz, T. (2021). *Software Testing Foundations: A Study Guide for the Certified Tester Exam-Foundation Level-ISTQB® Compliant*. dpunkt.verlag.
33. Al-Ajily, M. (2022). *Automated Testing for React Web Application with Cypress*.