Практикум по разарботке ML

Анисимов Я.О и ко

итмо

07 ноября 2023

Section 1

Общее сведения о курсе

О чем курс

- Разработка бекенд сервиса в котором есть ML модельки
- Отработка конкретной бизес-логики приложения
- Знакомство с технологиями: FastAPI, SQL, Docker
- Общие сведения о Clean Architecture

Стурктура курса

- Лекциионый блок(ноябрь)
- Серия ментор сессий/семинаров(декабрь)
- Консультации(январь)
- Зачет(январь)

Темы лекционного блока

- Постановка задачи, общее сведения про API
- FastAPI и чистая архитектура
- Базы данных и sqlalhcemy
- Воркеры для тяжелых задач
- WebUI средствами Python
- Инфраструктура

Section 2

Бриф заадачи Разработка ML-сервиса с подсистемой биллинга

Описание проекта

Цель проекта - разработать ML-сервис с подсистемой биллинга, который будет осуществлять предсказания на основе ML-моделей и списывать кредиты с личного счета пользователя за успешное выполнение предсказания. Сервис должен быть надежны и готовым для использования в продакшн-окружении.

Основные требования

1. Возможность загрузки и использования ML-моделей: сервис должен иметь возможность загружать и использовать различные ML-модели для выполнения предсказаний. Входные данные для моделей должны подаваться в сервис с использованием удобного API (Application Programming Interface).
2. Биллинговая подсистема: сервис должен поддерживать функциональность биллинга, где пользователь хранит определенное количество кредитов на своем личном счете. При успешном выполнении предсказания, счет пользователя должен быть списан за использованные кредиты.
3. Пользовательская система: сервис должен иметь пользовательский интерфейс, позволяющий пользователям регистрироваться, входить в систему и управлять своим личным счетом.
4. Мониторинг и аналитика: сервис должен предоставлять возможность мониторинга и аналитики, включая отчеты о выполненных предсказаниях, использованных кредитах и другой статистике.

Технические требования

1. Язык программирования: разработка сервиса должна быть выполнена с использованием языка программирования, который наилучшим образом соответствует требованиям проекта (например, Python).
2. ML-фреймворк: для загрузки и использования ML-моделей рекомендуется использовать Scikit-learn.
3. База данных: для хранения пользовательских данных, моделей и биллинговой информации можно использовать реляционную базу данных (например, PostgreSQL или Sqlite).
4. API: сервис должен предоставлять удобное и документированное API для загрузки моделей, выполнения предсказаний и управления пользовательскими данными.
5. Инфраструктура: необходимо использовать технологии контейнерезации.

План работы

1. Анализ требований: уточнение и детализация требований проекта, создание документации.
2. Проектирование архитектуры: разработка общей архитектуры сервиса, определение компонентов и API.
3. Разработка ML-функциональности: загрузка и использование ML-моделей, реализация функций предсказания.
4. Разработка биллинговой подсистемы: создание механизма учета кредитов и списывания с личного счета пользователя.
5. Разработка пользовательской системы: регистрация, аутентификация и управление личным счетом пользователей.
6. Внедрение и документация: установка сервиса в продакшн-окружение, создание документации для пользователей и администраторов.

Ожидаемые результаты

1. Функционирующий ML-сервис с подсистемой биллинга, способный загружать и использовать ML-модели для выполнения предсказаний.
2. Биллинговая система, позволяющая управлять пользовательскими счетами и списывать кредиты за успешное выполнение предсказания.
3. Пользовательская система, позволяющая пользователям регистрироваться, входить в систему и управлять своим личным счетом.
4. Масштабируемая инфраструктура, способная обрабатывать большое количество запросов и обеспечивать высокую доступность.
5. Документация, описывающая работу сервиса, API и рекомендации по развертыванию и использованию.

ML задача курса

TBD

Коммуникация на курсе

TBD

Секция QnA

Вопросы?

Section 3

Основы дизайна API

Протокол HTTP

- Протокол передачи данных, используемый веб-серверами и клиентами.
- Основные методы HTTP:
  - GET: получение данных
  - POST: отправка данных на сервер
  - PUT: обновление данных на сервере м - DELETE: удаление данных на сервере
- Коды состояния HTTP (status codes):
  - 200 OK: успешный запрос
  - 400 Bad Request: некорректный запрос
  - 404 Not Found: запрошенный ресурс не найден

Паттерны проектирования API

1. RESTful API:
   - Основан на принципах REST.
   - Ресурсы представлены в формате URL.
   - Использует верблюжью нотацию для именования ресурсов.
2. GraphQL API:
   - Модернизированный подход к созданию API.
   - Клиенты выбирают, какие данные им нужны.
   - Единый запрос для получения нескольких ресурсов.

Инструменты проектирования API

1. Swagger:
   - Фреймворк для разработки, проектирования и документирования API.
   - Позволяет создавать спецификацию API в формате JSON или YAML.
   - Генерирует интерактивную документацию и клиентские библиотеки.
2. API Blueprint:
   - Язык для описания API в формате Markdown.
   - Позволяет создавать простую и читабельную документацию.
   - Поддерживает генерацию кода и автоматическую проверку API.
3. RAML:
   - YAML-ориентированный язык описания API.
   - Позволяет задавать макет данных и примеры.
   - Поддерживает генерацию кода для различных языков.

Принципы проектирования API

1. Единообразие:
   - Устанавливайте согласованные стандарты и используйте их повсюду.
   - Имена ресурсов, методы HTTP и параметры запросов должны быть последовательными и понятными.
2. Понятность:
   - Легко понять, как использовать API и что ожидать в ответе.
   - Правильно документируйте API, предоставляя примеры запросов и ответов.
3. Безопасность:
   - Используйте соответствующие механизмы аутентификации и авторизации.
   - Защитите свои эндпоинты от нежелательного доступа и злоумышленников.

Типичные ошибки в проектировании API и способы их исправления

1. Нестабильность API:
   - Избегайте изменений внутренней реализации, которые приводят к частым изменениям в API.
   - Создайте стабильные версии API и поддерживайте их долгое время.
2. Неправильная обработка ошибок:
   - Возвращайте адекватные коды состояния и сообщения об ошибках.
   - Предлагайте разработчикам способы понять и исправить ошибки.
3. Неподходящая структура данных:
   - Определите наиболее подходящую структуру в соответствии с потребностями клиентов.
   - Используйте запросы с параметрами, чтобы фильтровать и сортировать данные.
4. Недостаточная документация:
   - Создайте полную и понятную документацию для вашего API.
   - Обновляйте документацию с каждым изменением API.

Пример хорошего RESTful API

GET /items

- Запрос на получение списка всех айтемов.
- Бизнес-значимость: клиенты могут получить полный список доступных айтемов.

GET /items/{id}

- Запрос на получение конкретного айтема по его идентификатору.
- Бизнес-значимость: клиенты могут получить информацию о конкретном айтеме, используя его идентификатор.

POST /items

- Запрос на создание нового айтема.
- Бизнес-значимость: клиенты могут добавлять новые айтемы в систему.

PUT /items/{id}

- Запрос на обновление информации о существующем айтеме.
- Бизнес-значимость: клиенты могут изменять информацию о существующем айтеме, используя его идентификатор.

Пример плохого RESTful API

GET /getAllItems

- Запрос на получение списка всех айтемов.
- Бизнес-значимость: в названии эндпоинта повторяется “все”, что является лишним, так как нет другой альтернативы.

GET /getItemById/{id}

- Запрос на получение конкретного айтема по его идентификатору.
- Бизнес-значимость: параметр “ById” в названии эндпоинта излишен, так как уже понятно, что идентификатор используется.

POST /addItemToInventory

- Запрос на создание нового айтема в инвентаре.
- Бизнес-значимость: в названии эндпоинта присутствует уточнение о добавлении айтема в инвентарь, что не является необходимым.

PUT /updateItem/{id}

- Запрос на обновление информации о существующем айтеме.
- Бизнес-значимость: использование глагола “update” в названии эндпоинта не

Section 4

Документирование API

Пример документирования

Методы API

- POST /users - создание нового пользователя
- GET /users/{userId} - получение информации о пользователе по идентификатору
- PUT /users/{userId} - обновление информации о пользователе
- DELETE /users/{userId} - удаление пользователя по идентификатору
- GET /items/{itemId} - получение информации об айтеме по идентификатору
- POST /items - создание нового айтема
- PUT /items/{itemId} - обновление информации об айтеме
- DELETE /items/{itemId} - удаление айтема по идентификатору

Пример документирования. Пример файла

Пример документирования. Польза от документирования API на Swagger

- Ясность и простота использования для разработчиков, которые используют API
- Быстрая интеграция между различными системами, так как Swagger предоставляет полное описание API и его возможностей
- Улучшение коммуникации между разработчиками, тестировщиками и другими участниками проекта
- Уменьшение затрат на разработку и поддержку API, так как Swagger предоставляет шаблоны и инструкции для генерации документации и кода
- Повышение безопасности путем использования встроенной валидации данных в Swagger

Пример документирования. Заключение

- Пример простого API для регистрации пользователя и работы с айтемом был представлен
- Swagger позволяет документировать API, описывая его методы, параметры, коды ответа и схемы данных
- Важными принципами документирования API на Swagger являются читаемость, описательность и поддержка семантики
- Документирование API на Swagger обладает рядом преимуществ, таких как ясность использования, быстрая интеграция и улучшение коммуникации

Section 5

Паттерны проектирования stateful API

Stateful API

- Сохранение состояния сервера между запросами клиента.
- Использует токены или данные в cookie для идентификации и аутентификации клиента.

Cookie-based подход

- Идентификатор сессии хранится в cookie.
- Сервер проверяет и обновляет сессию при каждом запросе клиента.

Пример

1. Клиент отправляет запрос на аутентификацию с логином и паролем.
2. Сервер проверяет и создает уникальный идентификатор сессии.
3. Сервер возвращает идентификатор сессии в виде cookie.
4. Клиент отправляет запросы с cookie в каждом последующем запросе.
5. Сервер считывает идентификатор сессии из cookie и обрабатывает запрос.

Плюсы и минусы

Плюсы

- Простая реализация.
- Сервер может хранить дополнительную информацию о сессии.
- Удобство использования для клиентов.

Минусы

- Не подходит для мобильных приложений и клиентов без поддержки cookie.

JWT (JSON Web Token) подход

- Токен JWT содержит информацию о клиенте и подписывается сервером.
- Токен передается через заголовок Authorization или параметр запроса.

Пример

1. Клиент отправляет запрос на аутентификацию с логином и паролем.
2. Сервер создает JWT с информацией о клиенте и подписывает его секретным ключом.
3. Сервер возвращает JWT клиенту.
4. Клиент отправляет JWT в заголовке Authorization или параметре запроса.
5. Сервер проверяет подпись и расшифровывает JWT для аутентификации и авторизации.

Плюсы и минусы

Плюсы

- Независимость от сессии и состояния на сервере.
- Поддержка мобильных приложений и клиентов без поддержки cookie.
- Возможность передавать дополнительные данные внутри токена.

Минусы

- Больший размер токена в сравнении с cookie.

Section 6

Примеры использования cookie и JWT для RESTful API по заказу айтемов

Cookie

Структура запроса

Пример запроса с использованием cookie:

```
GET /items HTTP/1.1
Host: example.com
Cookie: sessionId=abcd1234
```

Данные аутентификации

Cookie может содержать данные аутентификации, такие как токен доступа или идентификатор сессии. В примере выше, sessionId является идентификатором сессии.

Содержимое сессии

Cookie может использоваться для хранения информации о сессии пользователя. В сессии может содержаться информация, такая как идентификатор пользователя, предпочтения, корзина с выбранными айтемами и т.д.
Пример содержимого сессии:

```
{
  "userId": "12345",
```

JWT (JSON Web Token)

Структура запроса

Пример запроса с использованием JWT:

```
GET /items HTTP/1.1
Host: example.com
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NT
```

Данные аутентификации

JWT представляет собой токен, который содержит информацию о пользователе или сессии и подписывается с помощью секретного ключа. В примере выше, eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9 представляет собой заголовок токена, а SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c представляет собой подпись.

Содержимое токена

JWT может содержать информацию о пользователе или сессии в виде полезной нагрузки (payload). В примере выше, полезная нагрузка содержит следующую информацию:

```
{
  "sub": "1234567890",
```