Санкт-Петербургский государственный университет

Группа 23.Б16-пу

Лабораторная работа по дисциплине
Алгоритмы и структуры данных

«Генерация синтетических данных о покупках в магазинах»

Выполнил студент:
Зайнуллин Мансур Альбертович

Преподаватель:
Дик Александр Геннадьевич
ассистент кафедры компьютерного моделирования
и многопоточных систем

Санкт-Петербург
2024

Оглавление

1 Цель работы

Цель работы — создание синтетических данных о покупках в магазинах для анализа и тестирования алгоритмов. Генерация данных включает информацию о магазинах, категориях товаров, брендах и транзакциях.

2 Описание задачи

2.1 Требования к датасету

Сгенерировать датасет, в котором будут следующие свойства:

- **Название магазина**: М.Видео.
- **Координаты** (дата и время, долгота и широта): 2020-01-22T08:30+03:00 и 59.881653, 29.830170.
- **Категория**: ноутбук.
- **Бренд**: Lenovo.
- **Номер карточки**: “1234 5678 1234 5678”.
- **Количество товаров**: 1 шт.
- **Стоимость**: 50 000 руб.

2.2 Дополнительные параметры для настройки

Дополнительная информация по каждому свойству (для магазинов в Санкт-Петербурге):

- **Название магазина**: генерируется по ”Словарю”, состоящему из магазинов, представленных на данной территории.
- **Координаты и время**: определяются согласно реальному местоположению магазина.
- **Категории**: есть возможность настройки категорий товаров, которые соответствуют тематике магазина.
- **Бренд**: генерируется настраиваемый набор брендов, которые соответствуют категориям товаров в магазине.

- **Номер карточки**: генерируется набор номеров с учётом вероятности банка (Сбербанк, Газпромбанк и т.д.) и платёжной системы (Visa, MasterCard и т.д.). Оплата может осуществляться несколько раз с одной карты.
- **Количество товаров**: можно задавать произвольно в процессе генерации данных.
- **Стоимость**: определяется согласно средней стоимости товаров в категории.

2.3 Ограничения для датасета

Ограничения для датасета:

- **Минимум строк**: датасет должен содержать не менее 50 000 строк.
- **Название магазина**: ”Словарь” должен включать не менее 30 магазинов.
- **Координаты**: долгота и широта округляются до 10 знаков после запятой. Дата и время покупок должны быть реальными и находиться в пределах времени работы магазина. Если магазин работает с 10:00 до 22:00, то покупки возможны только в этот интервал.
- **Категории**: ”Словарь” должен включать не менее 50 категорий товаров.
- **Бренды**: ”Словарь” должен содержать не менее 500 брендов.
- **Номер карточки**: один и тот же номер может использоваться не более 5 раз.
- **Количество товаров**: минимум 5 товаров в каждой покупке.
- **Стоимость**: все товары должны иметь ненулевую стоимость. Если появляется ошибка — это должно быть зафиксировано.

3 Теоретическая часть

3.1 Требования и ограничения для датасета

Для создания синтетического датасета о покупках в магазинах необходимо учесть ряд требований и ограничений. Эти ограничения направлены на то, чтобы полученные данные были реалистичными, разнообразными и соответствовали условиям задачи. Основные требования к датасету включают следующие аспекты: тематические магазины, реальные координаты, категории товаров, бренды, время работы магазинов, количество товаров и стоимость.

3.1.1 Генерация категорий магазинов и товаров

Магазины в датасете будут разделены на несколько тематик, таких как продуктовые магазины, аптеки, магазины электроники, строительные магазины, магазины одежды и магазины автозапчастей. Для каждой тематики создаются категории товаров, которые могут быть представлены в этих магазинах. Например, в магазинах электроники можно встретить категории товаров, такие как смартфоны, ноутбуки, мониторы и периферийные устройства.

Генерация категорий товаров для каждого типа магазинов происходит на основе запроса к модели GPT, которая возвращает список категорий, подходящих для определённой тематики. Для каждой категории товара также генерируется набор брендов. Бренды и товары связаны через категорию, что позволяет обеспечить соответствие между типом магазина, категориями и продаваемыми брендами.

3.1.2 Использование реальных координат магазинов

Для того чтобы географические данные были реалистичными, в датасете используются реальные координаты магазинов. Эти координаты получаются с помощью Yandex ”API Поиска по организациям”. Сервис позволяет искать магазины по заданной тематике в определённом регионе (например,

в Санкт-Петербурге). После получения данных о магазинах их координаты сохраняются и включаются в датасет.

Каждая точка координат в датасете округляется до 8 знаков после запятой, что обеспечивает точность местоположения.

3.1.3 Настройка распределений времени работы магазинов

Магазины в датасете работают в разные часы. Для моделирования времени работы магазинов используется вероятностное распределение. Время открытия магазина выбирается случайным образом из заранее заданного списка (например, 7:00, 8:00, 9:00, 10:00, 11:00). Аналогично, время закрытия магазина выбирается из списка (например, 20:00, 21:00, 22:00, 23:00, 24:00). Некоторые типы магазинов, такие как продуктовые и аптеки, могут работать круглосуточно, для них устанавливается вероятность работы в круглосуточно.

Генерация времени работы магазина происходит для каждой транзакции, что позволяет учесть реальные временные ограничения для покупок. Если магазин не работает круглосуточно, то время покупки выбирается в пределах заданного интервала работы магазина.

3.1.4 Генерация количества товаров и цен

Количество товаров в каждой покупке определяется с использованием нормального распределения. Среднее значение и стандартное отклонение задаются в настройках.

Если мы сгенерировали значение < 5, то мы его задаём равным 5.

Цены на товары также привязываются к брендам, которые производят эти товары. Генерация цен происходит на основе шаблонов, предоставленных моделью GPT. Это позволяет задать адекватные ценовые диапазоны для каждой категории товаров. Например, смартфоны будут иметь более высокие цены, чем продукты питания, что соответствует реальной картине.

3.1.5 Генерация номеров карт и их использование

Каждая покупка привязывается к номеру карты, который генерируется на основе BIN-кодов (Bank Identification Number). BIN-коды представляют собой первые шесть цифр номера карты, которые указывают на банк-эмитент и платёжную систему (например, Visa, Mastercard). Для генерации номеров карт используется база данных BIN-кодов, которая включает в себя информацию о банках и платёжных системах.

Номер карты генерируется случайным образом, используя первые шесть цифр BIN, а оставшиеся цифры заполняются случайными значениями. Один и тот же номер карты может использоваться не более 5 раз, что контролируется в ходе генерации.

3.2 Структура settings.json

Файл `settings.json` служит конфигурационным файлом, который содержит параметры для генерации синтетического датасета. Этот файл структурирован и включает в себя несколько ключевых разделов, каждый из которых отвечает за определённый аспект процесса генерации данных.

Основные разделы `settings.json` включают:

3.2.1 Категории магазинов и товары

Этот раздел определяет типы магазинов, участвующих в генерации данных, а также доступные категории товаров для каждого магазина. Для каждой категории товаров задаётся список брендов. Например, в категории «электроника» могут быть указаны товары, такие как смартфоны, ноутбуки, и бренды, которые их производят.

3.2.2 Местоположения магазинов

Каждая торговая сеть, указанная в разделе категорий, имеет привязанные координаты местоположений. Эти координаты (долгота и широта) получаются с помощью API Yandex и сохраняются с точностью до 8 знаков

после запятой, чтобы обеспечить точное расположение магазинов. Эти данные важны для анализа транзакций с привязкой к реальной географии.

3.2.3 Распределение времени работы магазинов

Время открытия и закрытия магазинов моделируется через вероятностное распределение. В этом разделе указываются возможные времена открытия и закрытия для каждого магазина или группы магазинов. Например, для продуктовых магазинов может быть настроена возможность круглосуточной работы.

3.2.4 Распределение количества покупок

Этот раздел определяет параметры для генерации количества товаров в каждой транзакции. Настройки включают среднее значение и стандартное отклонение для нормального распределения. Это обеспечивает реалистичное распределение количества покупок для каждой транзакции.

3.2.5 Платёжные системы и банки

Раздел платёжных систем включает информацию о платёжных системах (например, Visa, Mastercard) и банках, которые могут использоваться при генерации данных. Для каждой платёжной системы и банка задаётся вероятность их использования. Также указывается связь с BIN-кодами, которые используются для генерации уникальных номеров карт.

3.2.6 Пути к файлам

Отдельный раздел файла `settings.json` отвечает за указание путей к другим необходимым файлам, таким как база данных BIN-кодов, которые используются при генерации платёжных данных. Это позволяет скриптам получать доступ к необходимым данным во время выполнения генерации.

3.3 Использование внешних API

Для генерации данных о покупках в магазинах в синтетическом датасете используются два внешних API: API Поиска по организациям Yandex и ChatGPT API. Каждый из них решает специфические задачи, необходимые для выполнения требований задания.

3.3.1 API Поиска по организациям Yandex

API Yandex используется для получения географических координат реальных магазинов, что необходимо для выполнения требования по обеспечению точных данных о местоположении магазинов в Санкт-Петербурге. В запросе к API указываются параметры, такие как название торговой сети, центральная точка поиска (долгота и широта) и радиус поиска. На основе ответа API возвращаются координаты магазинов, которые затем сохраняются в файл `settings.json`.

Координаты округляются до 8 знаков после запятой для достижения точности, что позволяет привязать каждую транзакцию к реальному месту на карте. Данный подход используется для всех категорий магазинов, что обеспечивает реалистичность датасета.

3.3.2 ChatGPT API

ChatGPT API используется для генерации контента, связанного с ассортиментом товаров и брендов. При помощи специально настроенных запросов к модели GPT генерируются списки категорий товаров, которые могут продаваться в каждом типе магазина. Также с помощью API генерируются списки брендов, выпускающих эти товары, и примерные цены на них.

Запросы к ChatGPT строятся таким образом, чтобы модель возвращала только необходимые данные без лишней информации. Например, для магазинов электроники API может вернуть категории товаров, такие как смартфоны, ноутбуки, и бренды, такие как ”Apple” и ”Samsung”. Также API помогает сгенерировать цены на товары, что позволяет создавать реалистичные данные о покупках.

3.4 База данных BIN-кодов

BIN-коды (Bank Identification Number) — это первые шесть цифр номера банковской карты, которые указывают на платёжную систему и банк-эмитент. BIN-коды важны для генерации номеров карт, так как они позволяют указать, через какую платёжную систему и банк прошла транзакция.

3.4.1 Структура базы BIN-кодов

База данных BIN-кодов содержит несколько ключевых полей: сам BIN-код, платёжная система (`brand`) и банк-эмитент (`issuer`). Эти данные используются для создания номеров карт, что позволяет в дальнейшем генерировать реалистичные транзакции, привязанные к конкретным банкам и платёжным системам.

3.4.2 Связь BIN-кодов с платёжными системами и банками

В проекте используется база данных, содержащая BIN-коды для российских банков, что позволяет генерировать номера карт, соответствующие реальным платёжным системам и банкам. BIN-коды используются для генерации уникальных номеров карт, что обеспечивает разнообразие данных в датасете.

3.5 Блок-схемы алгоритма

В работе используются два скрипта: `settings_generator.py` для создания файла `settings.json` и `data_generator.py` для генерации базы данных покупок.

Скрипт `settings_generator.py` собирает данные о категориях магазинов, товарах, брендах и времени работы, формируя файл `settings.json`.

Скрипт `data_generator.py` использует `settings.json` для создания синтетической базы данных, включающей информацию о транзакциях, номерах карт и стоимости товаров.

Подробные блок-схемы алгоритмов работы этих скриптов доступны по ссылке: https://github.com/MansurYa/labs-for-algorithms-and-data-structures/blob/main/Lab1/code-flowchart-lab1.pdf.

4 Описание программы

4.1 Архитектура программы

Программа состоит из двух основных компонентов: скрипта `settings_generator.py` и скрипта `data_generator.py`.

Скрипт `settings_generator.py` предназначен для создания файла настроек `settings.json`, который содержит информацию о магазинах, товарах, брендах, времени работы и платёжных системах. Этот файл используется вторым скриптом.

Скрипт `data_generator.py` загружает файл настроек `settings.json` и на его основе создаёт синтетический набор данных о покупках в магазинах. Данные записываются в файл `purchases_data.xlsx` в формате Excel.

Структура проекта включает следующие каталоги и файлы:

- `src/`: содержит скрипты `settings_generator.py` и `data_generator.py`.
- `BINs/`: содержит данные для генерации номеров карт.
- `settings.json`: файл настроек, создаваемый первым скриптом и используемый вторым.
- `purchases_data.xlsx`: файл с результатами генерации данных.

4.2 Описание скриптов

В данном разделе описаны два ключевых скрипта проекта: `settings_generator.py` и `data_generator.py`. Оба скрипта выполняют разные задачи в процессе генерации синтетических данных, взаимодействуя с внешними API, а также с файлами, такими как `settings.json` и `purchases_data.xlsx`.

4.2.1 Скрипт settings_generator.py

Скрипт `settings_generator.py` отвечает за создание конфигурационного файла `settings.json`, который используется

для генерации базы данных покупок. Программа взаимодействует с пользователем через командную строку, запрашивая различные параметры (например, количество магазинов, их расположение, категории товаров). После получения данных от пользователя, скрипт обращается к API Yandex и ChatGPT для получения реальных координат магазинов и ассортимента товаров.

Ключевые функции скрипта:

- `chat_GPT_response`: получает списки товаров и брендов с помощью ChatGPT API, основываясь на переданном запросе.
- `get_organizations_locations`: использует Yandex API для поиска координат магазинов по заданным параметрам, таким как категория магазина и местоположение.
- `get_input`: получает ввод от пользователя, преобразует его в требуемый формат (например, строка, целое число, или число с плавающей запятой) и проверяет корректность данных.
- `is_valid_file_path`: проверяет правильность указанного пользователем пути к файлу и его расширения.
- `main`: основная функция программы, собирающая все данные и сохраняющая их в `settings.json` для последующего использования в генерации данных.

Полный код скрипта `settings_generator.py` приведён ниже:

Примечание: Из-за сложностей с отображением русского текста в блоках кода LaTeX, промпты для ChatGPT, которые содержат текст на русском языке, приведены ниже после основного кода.

```
# -*- coding: utf-8 -*-
import os
import re
import requests
import pandas as pd
from openai import OpenAI
import json
```

```
def chat_GPT_response(prompt: str):
    """
    Get a response from Chat GPT.

    :param prompt: Instruction/message - what needs to be done?
    :return: String - the response from Chat GPT
    """

    OPENAI_API_KEY = os.environ.get("OPENAI_API_KEY")
    OPENAI_API_ORGANIZATION_KEY =
        ↪ os.environ.get("OPENAI_API_ORGANIZATION_KEY")

    if not OPENAI_API_KEY or not OPENAI_API_ORGANIZATION_KEY:
        raise ValueError("API keys not found. Check environment variables.")

    client = OpenAI(
        organization=OPENAI_API_ORGANIZATION_KEY,
        api_key=OPENAI_API_KEY
    )

    response = client.chat.completions.create(
        model="gpt-4o",
        messages=[{"role": "user", "content": prompt}],
        temperature=0.4,
        max_tokens=4096,
        stream=False,
    )

    return response.choices[0].message.content


def get_organizations_locations(organizations_name: str,
    ↪ location_for_search: str, search_center_longitude: float,
                                search_center_latitude: float, search_radius:
                                    ↪ float):
    """
    Using Yandex's Organization Search API,
    finds the coordinates of organizations by their name
```

```
in a specific area, for example, in a specific city.

:param organizations_name: The name of the organization network to
    ↪ search for
:param location_for_search: Location for the search, for example, the
    ↪ city name
:param search_center_longitude: The longitude of the search center in
    ↪ degrees
:param search_center_latitude: The latitude of the search center in
    ↪ degrees
:param search_radius: The radius of the search area around the center
:return: Returns a list of dictionaries. Each dictionary represents the
    ↪ store coordinates in the following format:
            {
                "longitude": store longitude,
                "latitude": store latitude
            }
"""

YANDEX_ORGANIZATION_SEARCH_API_KEY =
    ↪ os.environ.get("YANDEX_ORGANIZATION_SEARCH_API_KEY")

if not YANDEX_ORGANIZATION_SEARCH_API_KEY:
    raise ValueError("API key not found. Set it as an environment
        ↪ variable YANDEX_ORGANIZATION_SEARCH_API_KEY")

base_url = "https://search-maps.yandex.ru/v1/"

params = {
    "text": f"{location_for_search}, {organizations_name}",
    "type": "biz",
    "lang": "ru_RU",
    "ll": f"{search_center_longitude},{search_center_latitude}",
    "spn": f"{search_radius},{search_radius}",
    "results": 50, # Maximum number of results per request
    "apikey": YANDEX_ORGANIZATION_SEARCH_API_KEY
}

response = requests.get(base_url, params=params)
```

```
    if response.status_code != 200:
        raise Exception(f"API request failed with status code
            ↪ {response.status_code}")

    data = response.json()

    organizations_locations = []
    for feature in data.get("features", []):
        coordinates = feature.get("geometry", {}).get("coordinates", [])
        if len(coordinates) == 2:
            longitude, latitude = coordinates
            organizations_locations.append({
                "longitude": longitude,
                "latitude": latitude
            })

    return organizations_locations


def get_input(message_text: str, expected_type: type):
    """
    Displays message_text in the console, gets input from the user,
    converts it to the specified type (str, int, float), and returns the
        ↪ result.
    If the type is incorrect, keeps asking until valid input is provided.

    :param message_text: The message to be displayed to the user
    :param expected_type: The expected data type (str, int, float)
    :return: Value converted to the specified type
    """

    if expected_type not in [str, int, float]:
        raise ValueError("Only types str, int, float are allowed")

    while True:
        user_input = input(message_text)
        try:
            if expected_type == str:
                return user_input
            elif expected_type == int:
```

```
                return int(user_input)
            elif expected_type == float:
                return float(user_input)
        except ValueError:
            print(f"Error: please enter a valid {expected_type.__name__}
                ↪ value.")


def is_valid_file_path(file_path: str, expected_extension: str) -> bool:
    """
    Checks if the file path is valid and if the file extension matches the
        ↪ expected one.

    :param file_path: The file path including the name and extension
    :param expected_extension: The expected file extension (e.g., 'png')
    :return: True if the path is valid and the file has the correct
        ↪ extension, otherwise False
    """

    if not expected_extension.startswith('.'):
        expected_extension = f".{expected_extension}"

    _, file_extension = os.path.splitext(file_path)

    if file_extension.lower() != expected_extension.lower():
        return False

    try:
        if not os.path.isabs(file_path):
            return False

        os.path.normpath(file_path)

        return True
    except Exception:
        return False


def
    ↪ get_list_of_strings_or_ints_from_chat_gpt_response(prompt_for_list_generation
```

```
↪ str, expected_type: type):
"""
Using formatted input from the console based on the GPT response,
returns a list of strings or integers.

:param prompt_for_list_generation: Prompt for GPT to generate a list
:param expected_type: Expected data type (str or int)
:return: A list of strings or integers
"""

list_of_values = []

string_list_regex = r'^\[\s*"([^"]+)"(?:\s*,\s*"([^"]+)")*\s*\]$'
int_list_regex = r'^\[\s*\d+(?:\s*,\s*\d+)*\s*\]$'

while True:
    GPT_response = chat_GPT_response(prompt_for_list_generation)

    str_list_of_values = get_input(f"""

    user: {prompt_for_list_generation}

    chat_GPT: {GPT_response}

    Enter the list in the following format:
    "[\"Name1\", \"Name2\", ...]\" if a list of strings is expected, or
        ↪ \"[1, 2, 3, ...]\" for a list of integers

    Input: """, str)

    if expected_type == str and re.match(string_list_regex,
        ↪ str_list_of_values.strip()):
        list_of_values = re.findall(r'"([^"]*)"', str_list_of_values)

    elif expected_type == int and re.match(int_list_regex,
        ↪ str_list_of_values.strip()):
        list_of_values = [int(x) for x in re.findall(r'\d+',
            ↪ str_list_of_values)]

    else:
```

```
            print(f"Invalid input. Check the list format for
                ↪ {expected_type.__name__} and try again.")
            continue

        if list_of_values:
            print("List successfully obtained!")
            break
        else:
            print("The list is empty, try again.")

    return list_of_values


def main():
    while True:
        path_to_settings_file = get_input("Enter the path to settings.json
            ↪ including the file name and extension: ", str)

        if is_valid_file_path(path_to_settings_file, "json"): break

    if os.path.isfile(path_to_settings_file):
        raise Exception("This file already exists! Enter a different path.")

    directory_path_to_settings_file = os.path.dirname(path_to_settings_file)
    if not os.path.exists(directory_path_to_settings_file):
        os.makedirs(directory_path_to_settings_file)

    settings = {} # This will be saved to settings.json at the end (after
        ↪ populating settings)

    shop_categories_list = [
        "Grocery stores",
        "Electronics stores",
        "Hardware stores",
        "Clothing stores",
        "Auto parts and supplies stores"
    ]

    settings["shop_categories"] = {}
```

```
for shop_category in shop_categories_list:
    settings["shop_categories"][shop_category] = {}

    chain_of_stores_list = []

    while True:
        count_of_stores_to_add = get_input(
            f"Enter the number of stores you want to add for the
                ↪ \"{shop_category}\" category (> 0): ",
            int)

        if count_of_stores_to_add < 1:
            print("You entered a value < 1.\n")
        else:
            break

    for count in range(count_of_stores_to_add):
        chain_of_stores_list.append(get_input(
            f"Enter the name of store network {count + 1} of
                ↪ {count_of_stores_to_add} for the \"{shop_category}\"
                ↪ category: ",
            str))

    settings["shop_categories"][shop_category]["chains_of_stores"] = {}

    for name_of_store in chain_of_stores_list:
        settings["shop_categories"][shop_category]["chains_of_stores"][name_of_st
            ↪ = {}

        settings["shop_categories"][shop_category]["chains_of_stores"][name_of_st
            ↪ \
            = get_organizations_locations(name_of_store, "Saint
                ↪ Petersburg", 59.938784, 30.314997, 0.2)

    print(f"\nsettings: {settings}\n\n")

    settings["shop_categories"][shop_category]["categories"] = {}

    product_category_list =
        ↪ get_list_of_strings_or_ints_from_chat_gpt_response(f"""
```

```
Write a list of product categories that can be found in stores with
    ↪ the theme "{shop_category}".

In the response, do not include any additional information! All
    ↪ categories must be listed with a single example, and there
    ↪ should be no punctuation at the end.
Before listing, write `\start`, and after finishing -`\end`. Each
    ↪ name must be enclosed in double quotes `"name"` and formatted
    ↪ as an array [].
Stick to the template precisely, only changing the category names
    ↪ to fit the store theme.

Example for the "Electronics stores" theme:
```
\start
["Smartphones", "Tablets", "Laptops", "Desktop computers",
    ↪ "Monitors", "Computer peripherals", "Printers", "Network
    ↪ equipment", "Televisions", "Audio equipment", "Cameras",
    ↪ "Gaming consoles", "Smart home devices", "Wearable
    ↪ electronics", "Software", "Storage devices", "Cables",
    ↪ "Batteries", "Accessories", "Drones", "Car electronics",
    ↪ "Office equipment", "Small home appliances", "VR/AR devices",
    ↪ "Security systems", "Programmable devices", "3D printers",
    ↪ "E-book readers", "Landline phones", "Network services"]
\end
```
""", str)

for product_category in product_category_list:
    settings["shop_categories"][shop_category]["categories"][product_category
        ↪ = {}
    settings["shop_categories"][shop_category]["categories"][product_category
        ↪ = {}

    brands_list =
        ↪ get_list_of_strings_or_ints_from_chat_gpt_response(f"""
    Imagine we're in a {shop_category}.
    Write a list of brands that might appear in this store and that
        ↪ produce products in the category {product_category},
        ↪ meaning they have products from the {product_category}
```

```
    ↪ category!

In the response, do not include any additional information! All
    ↪ brands must be listed with a single example, and there
    ↪ should be no punctuation at the end.
Start the list with `\start`, and finish it with `\end`. Each
    ↪ name should be in double quotes and formatted as an array
    ↪ []. Stick exactly to the template, just with different
    ↪ brand names for the given category.

Example for "Electronics stores" and category "Smartphones":
```
\start
["Apple", "Samsung", "Huawei", "Xiaomi", "OPPO", "Vivo", "Sony",
    ↪ "Google", "OnePlus", "Nokia", "Motorola", "Asus",
    ↪ "Lenovo", "Realme", "ZTE", "Nothing", "Honor", "Infinix",
    ↪ "Tecno", "Ulefone", "Prestigio", "UMIDIGI"]
\end
```
""", str)

price_for_brands_list =
    ↪ get_list_of_strings_or_ints_from_chat_gpt_response(f"""
Imagine we are in {shop_category}.
In the {product_category} section!
Here is the list of brands that sell products from the
    ↪ {product_category} category in a store of the
    ↪ {shop_category} type:
{", ".join(brands_list)}.
Estimate how much products from the {product_category} category
    ↪ will cost from each of these brands in {shop_category}.

Do not add any additional information! The example response
    ↪ below shows the format, and do not deviate from it!

Example for {shop_category} and {product_category}:
```
\start""
Apple: 80000""
Samsung: 45000""
```
```

```
        Huawei: 35000""
        Xiaomi: 25000""
        OPPO: 30000""
        Vivo: 28000""
        Sony: 50000""
        Google: 55000""
        OnePlus: 40000""
        Nokia: 20000""
        Motorola: 22000""
        Asus: 35000""
        Lenovo: 18000""
        Realme: 20000""
        ZTE: 16000""
        Nothing: 37000""
        Honor: 27000""
        Infinix: 15000""
        Tecno: 14000""
        Ulefone: 12000""
        Prestigio: 11000""
        UMIDIGI: 13000
        \end
        \start_of_array
        [80000, 45000, 35000, 25000, 30000, 28000, 50000, 55000, 40000,
            ↪ 20000, 22000, 35000, 18000, 20000, 16000, 37000, 27000,
            ↪ 15000, 14000, 12000, 11000, 13000]
        \end_of_array
        ```

        Provide reasonable price estimates! For instance, it's normal
            ↪ for smartphones to cost around 80000, but milk should be
            ↪ around 80, and a new car could cost 5000000!
        """, int)

        for brand, price in zip(brands_list, price_for_brands_list):
            settings["shop_categories"][shop_category]["categories"][product_cate
                ↪ = price

    while True:
        weight = get_input(
            f"Specify the probability that {shop_category} stores are
```
```

```
                ↪ open 24 hours, where 0.0 means they never operate
                ↪ 24/7, and 1.0 means they always operate 24/7",
            float)
        if weight < 0.0 or weight > 1.0:
            print(f"\nYou entered an invalid value - {weight}\n")
        else:
            break
    settings["shop_categories"][shop_category]["is_open_24_hours"] =
        ↪ weight

    print(settings)

settings["opening_time_distribution"] = {}
for opening_time in ["7:00", "8:00", "9:00", "10:00", "11:00"]:
    while True:
        weight = get_input(
            f"Enter the weight (a positive integer or 0) for the
                ↪ probability that the store opens at {opening_time}: ",
            int)
        if weight < 0:
            print(f"\nYou entered a negative value - {weight}\n")
        else:
            break

    settings["opening_time_distribution"][opening_time] = weight

settings["closing_time_distribution"] = {}
for closing_time in ["20:00", "21:00", "22:00", "23:00", "24:00"]:
    while True:
        weight = get_input(
            f"Enter the weight (a positive integer or 0) for the
                ↪ probability that the store closes at {closing_time}: ",
            int)
        if weight < 0:
            print(f"\nYou entered a negative value - {weight}\n")
        else:
            break

    settings["closing_time_distribution"][closing_time] = weight
```

```
settings["purchase_quantity_distribution"] = {}

while True:
    mean = get_input(
        f"Enter the \"mean\" value (a positive integer >= 5) to define
            ↪ the normal probability distribution function for the
            ↪ number of purchases in a single receipt.",
        int)
    if mean < 5:
        print(f"\nYou entered a value < 5 - {mean}\n")
    else:
        break
settings["purchase_quantity_distribution"]["mean"] = mean

while True:
    standard_deviation = get_input(
        f"Enter the \"standard_deviation\" value (a positive integer) to
            ↪ define the normal probability distribution function for
            ↪ the number of purchases in a single receipt.",
        int)
    if standard_deviation < 1:
        print(f"\nYou entered a value < 1 - {standard_deviation}\n")
    else:
        break
settings["purchase_quantity_distribution"]["standard_deviation"] =
    ↪ standard_deviation

while True:
    bin_list_path = get_input("Enter the path to the BIN code file
        ↪ (.csv): ", str)
    if os.path.exists(bin_list_path) and bin_list_path.endswith('.csv'):
        break
    else:
        print("The file does not exist or the extension is incorrect.
            ↪ Please try again.")
bin_list_path = "../BINs/binlist-data-narrower-and-only-russians.csv"

bin_list = pd.read_csv(bin_list_path, sep=';')

required_columns = ['bin', 'brand', 'issuer']
```

```
if not all(col in bin_list.columns for col in required_columns):
    raise ValueError("The file does not contain all the necessary
        ↪ data.")

settings["payment_systems_distribution"] = {}

payment_systems_stack = bin_list['brand'].unique().tolist()

for payment_system in payment_systems_stack:
    weight = get_input(f"Enter the probability for the payment system
        ↪ {payment_system} (a positive integer or 0): ", int)

    if weight > 0:
        settings["payment_systems_distribution"][payment_system] = weight

print(settings)

settings["banks_distribution"] = {}

bank_list = bin_list['issuer'].unique().tolist()

bank_stack = ["SBER", "VTB", "ALFA", "GAZPROM", "RAIFFEISEN",
    ↪ "UNICREDIT", "TINKOFF", "PROMSVYAZ",
            "RUSSIAN AGRICULTURAL", "ROSBANK", "OTKRITIE", "SOVCOM",
                ↪ "MOSCOW INDUSTRIAL",
            "SAINT PETERSBURG", "RENESANS", "URALSIB", "CREDIT BANK OF
                ↪ MOSCOW", "ZENTI", "BIN"]

for bank in bank_stack:
    weight = get_input(f"Enter the probability for the bank {bank} (a
        ↪ positive integer or 0): ", int)

    if weight > 0:
        active_bank_list = [b for b in bank_list if isinstance(b, str)
            ↪ and bank in b]

        for active_bank in active_bank_list:
            bank_list.remove(active_bank)
            settings["banks_distribution"][active_bank] = weight
```

```
    settings["bin_list_path"] = bin_list_path

    print(f"\nSettings have been configured: \n{settings}")

    try:
        with open(path_to_settings_file, 'w', encoding='utf-8') as f:
            json.dump(settings, f, ensure_ascii=False, indent=4)
        print(f"Settings successfully saved to {path_to_settings_file}")
    except Exception as e:
        print(f"Error saving settings: {e}")


if __name__ == "__main__":
    main()
```

Промпты для генерации категорий товаров и брендов

Генерация категорий товаров

Напиши список категорий, которые могут встречаться в магазинах с тематикой `{shopcategory}`.

Требования:

- В ответном сообщении не указывай никакой дополнительной информации!
- Все категории должны быть перечислены через запятую с одним примером.
- В конце предложения не должно стоять знака окончания предложения.
- Перед началом перечисления нужно написать `/start`, а после завершения — `/end`.
- Каждое наименование должно быть заключено в фигурные кавычки `"name"` и обёрнуто в квадратные скобки (как массив).

- Соблюдай шаблон точно, только с другими названиями категорий для указанной тематики магазина, не добавляй и не убирай никаких символов и слов.

Пример для магазина с тематикой ”Магазины электроники”: /start [”Смартфоны и мобильные телефоны”, ”Планшеты”, ”Ноутбуки”, ”Настольные компьютеры”, ”Мониторы”, ”Компьютерные периферийные устройства”, ”Принтеры и сканеры”, ”Сетевое оборудование”, ”Телевизоры”, ”Аудиооборудование”, ”Фото- и видеокамеры”, ”Игровые консоли и аксессуары”, ”Устройства умного дома”, ”Носимая электроника”, ”Программное обеспечение”, ”Накопители данных”, ”Кабели и адаптеры”, ”Батарейки и зарядные устройства”, ”Аксессуары для мобильных устройств”, ”Дроны и робототехника”, ”Автоэлектроника”, ”Офисная техника”, ”Мелкая бытовая техника”, ”Устройства виртуальной и дополненной реальности”, ”Системы безопасности”, ”Программируемые устройства”, ”3D-принтеры и аксессуары”, ”Электронные книги”, ”Телефоны и оборудование для стационарной связи”, ”Сетевые сервисы и подписки”] /end

Генерация списка брендов

Представь, что мы находимся в `{shopcategory}`. Напиши список брендов, которые могут встречаться в этом магазине и которые ПРОИЗВОДЯТ ТОВАР в категории товаров `{productcategory}`, то есть, у них есть продукция из категории `{productcategory}`.

Требования:

- В ответном сообщении не указывай никакой дополнительной информации!
- Все бренды должны быть перечислены через запятую с одним примером.
- В конце предложения не должно стоять знака окончания предложения.
- Перед началом перечисления нужно написать `/start`, а после завершения — `/end`.

- Каждое наименование должно быть заключено в фигурные кавычки `"name"` и обёрнуто в квадратные скобки (как массив).
- Соблюдай шаблон точно, только с другими названиями брендов для указанной категории товаров, не добавляй и не убирай никаких символов и слов.

Пример для магазина с тематикой ”Магазины электроники” и категорией ”Смартфоны и мобильные телефоны”:

/start [”Apple”, ”Samsung”, ”Huawei”, ”Xiaomi”, ”OPPO”, ”Vivo”, ”Sony”, ”Google”, ”OnePlus”, ”Nokia”, ”Motorola”, ”Asus”, ”Lenovo”, ”Realme”, ”ZTE”, ”Nothing”, ”Honor”, ”Infinix”, ”Tecno”, ”Ulefone”, ”Prestigio”, ”UMIDIGI”] /end

Оценка стоимости товаров

Представь, что мы находимся в `{shopcategory}`. В категории товаров `{productcategory}`!

Шаги для запроса:

- Вот список брендов, которые продают товар из категории `{productcategory}` в магазине типа `{shopcategory}`: `{", ".join(brandslist)}`.
- Оцени, сколько будут стоить продукты категории `{productcategory}` в `{shopcategory}` от каждого из данных брендов.

Пример для магазина с тематикой ”{shopcategory}” и категории {productcategory}: /start ”Apple”: 80000 ”Samsung”: 45000 ”Huawei”: 35000 ”Xiaomi”: 25000 ”OPPO”: 30000 ”Vivo”: 28000 ”Sony”: 50000 ”Google”: 55000 ”OnePlus”: 40000 ”Nokia”: 20000 ”Motorola”: 22000 ”Asus”: 35000 ”Lenovo”: 18000 ”Realme”: 20000 ”ZTE”: 16000 ”Nothing”: 37000 ”Honor”: 27000 ”Infinix”: 15000 ”Tecno”: 14000 ”Ulefone”: 12000 ”Prestigio”: 11000 ”UMIDIGI”: 13000 /end

/startofarray [80000, 45000, 35000, 25000, 30000, 28000, 50000, 55000, 40000, 20000, 22000, 35000, 18000, 20000, 16000, 37000, 27000, 15000, 14000, 12000, 11000, 13000] /endofarray

4.2.2 Скрипт data_generator.py

Скрипт `data_generator.py` использует файл настроек `settings.json`, чтобы сгенерировать синтетические данные о покупках и сохранить их в файл формата Excel. Этот скрипт загружает настройки, проверяет их корректность, а затем создает транзакционные данные, включая номера карт, товары, бренды, а также магазины.

Ключевые функции скрипта:

- `read_settings`: загружает файл `settings.json` и проверяет его структуру на наличие всех необходимых ключевых полей.
- `generate_purchase`: генерирует случайные данные о покупке, включая категорию магазина, товар, бренд, количество товаров и цену на основе настроек.
- `generate_card_number`: создает уникальные номера платёжных карт на основе BIN-кодов, загруженных из CSV-файла.
- `write_to_file`: записывает сгенерированные данные о покупках в файл `purchases_data.xlsx`, используя библиотеку `pandas`, проверяя лимиты по количеству строк на листе Excel.
- `main`: основная логика программы, отвечающая за инициализацию настроек, загрузку BIN-кодов и генерацию данных с отслеживанием прогресса через библиотеку `tqdm`.

Полный код скрипта `data_generator.py` приведён ниже:

```
import os
import json
import random
import pandas as pd
import csv
from datetime import datetime, timedelta
from tqdm import tqdm


def read_settings(settings_path: str) -> dict:
```

```
    """
    Reads and validates settings from the settings.json file.

    :param settings_path: Path to the settings.json file
    :return: Returns a dictionary of settings loaded from settings.json
    :raises FileNotFoundError: If the settings.json file is not found
    :raises ValueError: If the structure of the settings.json file is
        ↪ incorrect
    :raises json.JSONDecodeError: If there is an error while decoding JSON
    """
    try:
        if not os.path.exists(settings_path):
            raise FileNotFoundError(f"File {settings_path} not found.")

        with open(settings_path, 'r', encoding='utf-8') as f:
            settings = json.load(f)

        # Check for required keys
        required_keys = ['shop_categories', 'bin_list_path',
            ↪ 'opening_time_distribution',
                         'closing_time_distribution',
                            ↪ 'purchase_quantity_distribution']
        for key in required_keys:
            if key not in settings:
                raise ValueError(f"Key {key} is missing in settings.json")

        validate_settings(settings) # Additional type checking
        return settings
    except json.JSONDecodeError as e:
        raise ValueError(f"Error while reading JSON file {settings_path}:
            ↪ {e}")


def validate_settings(settings: dict) -> None:
    """
    Validates the structure and data types in settings.json.

    :param settings: Dictionary of settings loaded from settings.json
    :raises ValueError: If the structure or data types in settings.json are
        ↪ incorrect
```

```
    """
    if not isinstance(settings['shop_categories'], dict):
        raise ValueError("shop_categories must be a dictionary.")

    if not isinstance(settings['bin_list_path'], str):
        raise ValueError("bin_list_path must be a string.")

    if not isinstance(settings['opening_time_distribution'], dict) or \
       not all(isinstance(k, str) and isinstance(v, int) for k, v in
           ↪ settings['opening_time_distribution'].items()):
        raise ValueError("opening_time_distribution must be a dictionary
            ↪ with time (string) and weights (integers).")

    if not isinstance(settings['closing_time_distribution'], dict) or \
       not all(isinstance(k, str) and isinstance(v, int) for k, v in
           ↪ settings['closing_time_distribution'].items()):
        raise ValueError("closing_time_distribution must be a dictionary
            ↪ with time (string) and weights (integers).")

    if not isinstance(settings['purchase_quantity_distribution']['mean'],
       ↪ (int, float)) or \
       not
           ↪ isinstance(settings['purchase_quantity_distribution']['standard_deviati
           ↪ (int, float)):
        raise ValueError("purchase_quantity_distribution must contain
            ↪ numeric values for mean and standard_deviation.")


def load_bin_list(bin_list_path: str) -> list:
    """
    Loads BIN codes and related data from a CSV file.

    :param bin_list_path: Path to the CSV file containing BIN codes
    :return: Returns a list of dictionaries with BIN code data
    """
    bin_list = []
    required_columns = ['bin', 'brand', 'issuer']
    try:
        with open(bin_list_path, newline='', encoding='utf-8') as f:
            reader = csv.DictReader(f, delimiter=';')
```

```
            for row in reader:
                if all(col in row for col in required_columns):
                    bin_list.append({
                        'bin': row['bin'],
                        'brand': row['brand'],
                        'issuer': row['issuer']
                    })
                else:
                    raise ValueError("One of the required columns ('bin',
                        ↪ 'issuer', 'brand') is missing in the bin_list_path
                        ↪ file.")
    except FileNotFoundError:
        raise FileNotFoundError(f"File {bin_list_path} not found.")
    except Exception as e:
        raise ValueError(f"Error reading file {bin_list_path}: {e}")

    if not bin_list:
        raise ValueError(f"The BIN code list is empty. Check the
            ↪ bin_list_path file.")

    return bin_list


def initialize_output_file(output_path: str) -> int:
    """
    Checks if the Excel file exists. If yes, counts the rows; if not,
        ↪ creates a new file.
    Returns the number of rows already in the file (if it exists).

    :param output_path: Path to the output Excel file
    :return: The number of rows in the Excel file if it exists, otherwise 0
    """
    if os.path.exists(output_path):
        df_existing = pd.read_excel(output_path)
        existing_row_count = len(df_existing)
        print(f"File {output_path} exists. {existing_row_count} rows
            ↪ found.")
        return existing_row_count
    else:
        # Column names have been adjusted to match the task requirements
```

```
        df_new = pd.DataFrame(columns=['Store Name', 'Date and Time',
            ↪ 'Longitude', 'Latitude',
                                       'Category', 'Brand', 'Card Number',
                                           ↪ 'Quantity', 'Price'])
        df_new.to_excel(output_path, index=False)
        print(f"New file created at {output_path}.")
        return 0


def precompute_bin_codes_and_weights(settings: dict, bin_list: list):
    """
    Precomputes the bin_codes and weights lists based on settings and BIN
        ↪ code data.

    :param settings: Dictionary of settings containing 'banks_distribution'
    :param bin_list: List of dictionaries with BIN code data (each
        ↪ dictionary contains 'bin', 'brand', 'issuer')
    :return: Returns two lists:
            - bin_codes: A list of BIN codes associated with banks in
                ↪ 'banks_distribution'
            - weights: A list of weights corresponding to each BIN code
                ↪ based on the bank distribution
    :raises ValueError: If no BIN codes are available for the banks in
        ↪ 'banks_distribution'
    """
    banks_distribution = settings["banks_distribution"]

    issuer_to_bins = {}
    for bin_entry in bin_list:
        issuer = bin_entry['issuer']
        bin_code = bin_entry['bin']
        if issuer in banks_distribution:
            issuer_to_bins.setdefault(issuer, []).append(bin_code)

    if not issuer_to_bins:
        raise ValueError("No BIN codes available for the given bank
            ↪ distribution.")

    # Create lists of bin_codes and their corresponding weights
    bin_codes = []
```

```
    weights = []
    for issuer, bins in issuer_to_bins.items():
        bank_weight = banks_distribution[issuer]
        num_bins = len(bins)
        weight_per_bin = bank_weight / num_bins
        for bin_code in bins:
            bin_codes.append(bin_code)
            weights.append(weight_per_bin)

    return bin_codes, weights


def generate_bin_code(bin_codes: list, weights: list) -> str:
    """
    Generates a BIN code based on the provided bin_codes and weights lists.

    :param bin_codes: A list of available BIN codes for generation
    :param weights: A list of weights corresponding to each BIN code
    :return: Returns a randomly chosen BIN code based on the weights
    :raises ValueError: If the bin_codes and weights lists are empty or
        ↪ their lengths don't match
    """
    return random.choices(bin_codes, weights=weights, k=1)[0]


def generate_card_number(bin_code: str, set_card_numbers: set) -> str:
    """
    Generates a unique card number based on the BIN code and a random
        ↪ suffix.

    :param bin_code: BIN code (first 6 digits of the card number)
    :param set_card_numbers: A set of already generated card numbers to
        ↪ ensure uniqueness
    :return: Returns a unique card number as a string
    """
    while True:
        card_suffix = ''.join([str(random.randint(0, 9)) for _ in
            ↪ range(10)])
        card_number = bin_code + card_suffix
```

```
        if card_number not in set_card_numbers:
            set_card_numbers.add(card_number)
            return card_number


def generate_datetime(opening_time: str, closing_time: str) -> str:
    """
    Generates a random purchase time within store operating hours.

    :param opening_time: Store opening time (in 'HH:MM' format)
    :param closing_time: Store closing time (in 'HH:MM' format)
    :return: Returns a string with the purchase date and time in the format
        ↪ 'YYYY-MM-DD HH:MM'
    :raises ValueError: If the closing time is earlier than the opening time
    """
    open_hour, open_minute = map(int, opening_time.split(':'))
    close_hour, close_minute = map(int, closing_time.split(':'))

    # Ensure closing time is not earlier than opening time and the store is
        ↪ open for at least 1 hour
    if close_hour < open_hour or (close_hour == open_hour and close_minute
        ↪ <= open_minute):
        raise ValueError("Closing time cannot be earlier than opening
            ↪ time.")

    if close_hour == open_hour and abs(close_minute - open_minute) < 60:
        raise ValueError("The store must be open for at least 1 hour.")

    # Generate a random date starting from 2012
    start_date = datetime(2012, 1, 1)
    random_days = random.randint(0, (datetime.now() - start_date).days)
    random_date = start_date + timedelta(days=random_days)

    hour = random.randint(open_hour, close_hour)
    minute = random.randint(0, 59)

    # Add random time
    random_datetime = random_date.replace(hour=hour, minute=minute)
    return random_datetime.strftime("%Y-%m-%d %H:%M")
```

```
def generate_purchase(settings: dict, card_number: str) -> list:
    """
    Generates a purchase record. Selects store, product, time, and price.

    :param settings: Dictionary of settings loaded from settings.json
    :param card_number: Unique card number for this purchase
    :return: Returns a list of purchase data: store chain, purchase time,
        ↪ coordinates, category, brand, quantity, and price
    :raises ValueError: If no available stores, categories, or brands are
        ↪ found for the selected store
    """
    shop_category = random.choice(list(settings['shop_categories'].keys()))
    shop_info = settings['shop_categories'][shop_category]

    if not shop_info['chains_of_stores']:
        raise ValueError(f"No available stores in the category
            ↪ {shop_category}.")

    chain_of_stores =
        ↪ random.choice(list(shop_info['chains_of_stores'].keys()))

    store_locations =
        ↪ shop_info['chains_of_stores'][chain_of_stores]['locations']
    if not store_locations:
        raise ValueError(f"No available locations for the store chain
            ↪ {chain_of_stores}.")

    store_location = random.choice(store_locations)
    store_longitude = round(store_location['longitude'], 8)
    store_latitude = round(store_location['latitude'], 8)

    if random.random() < shop_info['is_open_24_hours']:
        opening_time = "00:00"
        closing_time = "23:59"
    else:
        opening_time =
            ↪ random.choices(list(settings['opening_time_distribution'].keys()),
                                weights=list(settings['opening_time_distribution']
        closing_time =
```

```
            ↪ random.choices(list(settings['closing_time_distribution'].keys()),
                                    weights=list(settings['closing_time_distribution']

        purchase_datetime = generate_datetime(opening_time, closing_time)

        if not shop_info['categories']:
            raise ValueError(f"No available product categories for
                ↪ {chain_of_stores}.")
        product_category = random.choice(list(shop_info['categories'].keys()))
        product_brands = shop_info['categories'][product_category]['brands']

        if not product_brands:
            raise ValueError(f"No available brands for the category
                ↪ {product_category}.")

        product_brand = random.choice(list(product_brands.keys()))
        product_price = product_brands[product_brand]

        number_of_purchases = max(5, min(100,
            ↪ int(random.gauss(settings['purchase_quantity_distribution']['mean'],
                                    settings['purchase_quantity_distribution']['stan

        return [chain_of_stores, purchase_datetime, store_longitude,
            ↪ store_latitude,
                product_category, product_brand, card_number,
                    ↪ number_of_purchases, product_price]


    def write_to_file(output_path: str, rows: list, sheet_name: str =
        ↪ "Sheet1") -> None:
        """
        Writes a list of rows to an Excel file at the given output_path. If the
            ↪ row limit is exceeded, creates a new sheet.

        :param output_path: Path to the output Excel file
        :param rows: List of rows to be written to Excel
        :param sheet_name: Name of the Excel sheet (default is "Sheet1")
        :raises PermissionError: If there are insufficient permissions to write
            ↪ to the file
        :raises IOError: If an error occurs during file writing
```

```
    """
    df_new = pd.DataFrame(rows, columns=['Store Name', 'Date and Time',
        ↪ 'Longitude', 'Latitude',
                                         'Category', 'Brand', 'Card Number',
                                            ↪ 'Quantity', 'Price'])
    max_rows_per_sheet = 1_000_000

    try:
        if os.path.exists(output_path):
            with pd.ExcelWriter(output_path, mode='a', engine='openpyxl',
                ↪ if_sheet_exists='overlay') as writer:
                if sheet_name in writer.sheets:
                    existing_rows = writer.sheets[sheet_name].max_row
                else:
                    existing_rows = 0

                if existing_rows + len(df_new) > max_rows_per_sheet:
                    sheet_name = f"{sheet_name}_part2"

                df_new.to_excel(writer, sheet_name=sheet_name, index=False,
                    ↪ header=False, startrow=existing_rows)
        else:
            df_new.to_excel(output_path, sheet_name=sheet_name, index=False)
    except PermissionError:
        raise PermissionError(f"Insufficient permissions to write to file
            ↪ {output_path}. Check access.")
    except Exception as e:
        raise IOError(f"Error writing to file {output_path}: {e}")


def generate_data(output_path: str, target_row_count: int, settings: dict,
    ↪ bin_list: list) -> None:
    """
    The main data generation process. Manages the number of generated rows
        ↪ and writes them to the file, showing progress using tqdm.

    :param output_path: Path to the output Excel file
    :param target_row_count: Target number of rows to generate
    :param settings: Dictionary of settings loaded from settings.json
    :param bin_list: List of BIN codes for generating card numbers
```

```
    """
    set_card_numbers = set()
    buffer = []
    buffer_size = 10000000

    existing_row_count = initialize_output_file(output_path)
    total_count_of_generated_rows = existing_row_count

    bin_codes, weights = precompute_bin_codes_and_weights(settings,
        ↪ bin_list)

    with tqdm(total=target_row_count, initial=existing_row_count, unit="
        ↪ rows") as pbar:
        while total_count_of_generated_rows < target_row_count:
            bin_code = generate_bin_code(bin_codes, weights)

            card_number = generate_card_number(bin_code, set_card_numbers)

            max_purchases_for_processed_card = random.randint(1, 5)
            for _ in range(max_purchases_for_processed_card):
                purchase = generate_purchase(settings, card_number)
                buffer.append(purchase)
                total_count_of_generated_rows += 1
                pbar.update(1)

                if len(buffer) >= buffer_size:
                    write_to_file(output_path, buffer)
                    buffer.clear()

            if total_count_of_generated_rows >= target_row_count:
                break

        if buffer:
            write_to_file(output_path, buffer)


def main() -> None:
    settings_path = '../settings.json'
    output_path = '../purchases_data_1048570.xlsx'
    target_row_count = 1048570
```

```
    settings = read_settings(settings_path)

    bin_list = load_bin_list(settings['bin_list_path'])

    generate_data(output_path, target_row_count, settings, bin_list)


if __name__ == "__main__":
    main()
```

4.2.3 Заключение

Оба скрипта выполняют критические задачи в проекте: `settings_generator.py` формирует конфигурацию данных для магазинов и товаров, а `data_generator.py` использует эту конфигурацию для генерации транзакционных данных. Благодаря использованию внешних API, таких как Yandex и ChatGPT, программа обеспечивает высокую точность и реалистичность сгенерированных данных.

Полный исходный код и инструкции по установке и запуску проекта доступны в репозитории: `https://github.com/MansurYa/labs-for-algorithms-and-data-structures/tree/main/Lab1`.

5 Рекомендации программиста

Несмотря на успешную работу программы, есть несколько аспектов, которые можно улучшить для повышения её надёжности, производительности и удобства использования:

- **Написание и проведение тестов**: Можно добавить юнит-тесты и интеграционные тесты для проверки основных функций программы. Например, тестирование работы с API, корректной генерации данных и записи их в файл позволит избежать ошибок на ранних этапах.
- **Вынесение промптов для Chat GPT в отдельный файл**: Вынесение текстовых промптов в отдельный файл упростит их редактирование и настройку. Это позволит обновлять запросы без изменения основного кода программы, что улучшит её масштабируемость и поддержку.
- **Добавление функции проверки корректности файла settings.json**: Необходимо реализовать функцию, которая проверяет корректность структуры и содержания файла ‘settings.json‘ перед началом генерации данных. Это поможет избежать ошибок на этапе выполнения программы.
- **Реализация функционала для редактирования settings.json**: Сейчас файл настроек можно только создать с нуля. Добавление функционала для редактирования существующих настроек позволит пользователям изменять параметры программы без необходимости пересоздания файла.
- **Оптимизация буферизации при записи данных**: Стоит побеспокоиться о том, чтобы при увеличении объёма файла, время записи новых сгенерированных строк существенно не возрастало.

6 Описание контрольного примера

В данном разделе представлен пример использования программы с демонстрацией возможности редактирования различных критериев для изменения входных параметров датасета. Основной способ настройки генерации данных осуществляется через создание и редактирование файла настроек `settings.json`.

6.1 Подготовка среды

1. **Установка Python**: Убедитесь, что на вашем компьютере установлена версия Python 3.9. Если нет, скачайте и установите её с официального сайта `https://www.python.org/downloads/`.

2. **Скачивание репозитория**: Клонируйте репозиторий с исходным кодом программы с GitHub по ссылке: `https://github.com/MansurYa/labs-for-algorithms-and-data-structures.git`.

3. **Переход в каталог проекта**: Откройте терминал и перейдите в директорию проекта Lab1:

```
cd labs-for-algorithms-and-data-structures/Lab1
```

4. **Установка зависимостей**: Установите необходимые библиотеки из файла `requirements.txt` командой:

```
pip install -r requirements.txt
```

6.2 Создание и настройка файла `settings.json`

Основным инструментом для редактирования критериев и входных параметров является файл настроек `settings.json`. Именно в этом файле задаются распределения вероятностей, списки категорий, брендов и другие параметры, влияющие на генерацию данных.

5. **Получение API ключей**:

    - **API Яндекс**: Зарегистрируйтесь и получите ключ для API Поиска по организациям на сайте `https://developer.tech.yandex.ru/services`.
    - **API OpenAI**: Зарегистрируйтесь на платформе OpenAI (`https://platform.openai.com`), создайте учётную запись и получите API ключ и идентификатор организации. Обратите внимание, что использование API OpenAI может быть платным.

6. **Настройка переменных окружения**: Сохраните полученные API ключи в переменные окружения вашей системы:

    - `OPENAI_API_KEY`: ваш API ключ OpenAI.
    - `OPENAI_API_ORGANIZATION_KEY`: идентификатор вашей организации OpenAI.
    - `YANDEX_ORGANIZATION_SEARCH_API_KEY`: ваш API ключ Яндекс.

7. **Запуск генератора настроек**: В терминале запустите скрипт `settings_generator.py` командой:

    ```
    python3 settings_generator.py
    ```

8. **Ввод параметров**: Следуйте инструкциям в консоли для ввода различных параметров. Во время работы скрипта вы сможете:

    - Указать путь для сохранения файла `settings.json`.
    - Выбрать категории магазинов и ввести количество сетей магазинов для каждой категории.
    - Ввести названия сетей магазинов и получить их реальные координаты с помощью API Яндекс.
    - Задать вероятность того, что магазины работают круглосуточно.
    - Сгенерировать списки категорий товаров и брендов с помощью ChatGPT API.

- Ввести распределения времени открытия и закрытия магазинов.
- Задать параметры нормального распределения для количества товаров в одной покупке.
- Указать вероятности использования различных платёжных систем и банков.

9. **Редактирование параметров**: Во время ввода данных вы можете задавать свои значения для каждого параметра, тем самым изменяя критерии генерации данных. Например:
    - Изменить список категорий магазинов, добавив или убрав определённые типы.
    - Задать собственные вероятности для времени открытия и закрытия магазинов.
    - Ввести свои списки категорий товаров и брендов, соответствующих тематике магазинов.
    - Настроить распределения платёжных систем и банков по своему усмотрению.
10. **Сохранение настроек**: После ввода всех параметров файл `settings.json` будет сохранён в указанном вами месте и будет содержать все заданные настройки.

6.3 Генерация данных с учётом настроек

11. **Запуск генератора данных**: После создания и настройки файла `settings.json` запустите скрипт `data_generator.py` командой:

```
python3 data_generator.py
```

12. **Настройка параметров генерации (опционально)**: При необходимости вы можете отредактировать файл `data_generator.py`, чтобы изменить следующие параметры:

- `settings_path`: путь к вашему файлу `settings.json`.
- `output_path`: путь для сохранения сгенерированных данных.
- `target_row_count`: желаемое количество строк в итоговом датасете.

Примечание: Если файл, указанный в `output_path`, уже существует, новые данные будут добавлены в конец файла до достижения общего количества строк, равного `target_row_count`. Если текущее количество строк больше или равно `target_row_count`, новые данные генерироваться не будут.

13. **Процесс генерации**: Скрипт использует ваши настройки из `settings.json` для генерации данных. В зависимости от заданных параметров, таких как вероятности, списки категорий и брендов, будут сгенерированы соответствующие данные о покупках.

14. **Получение результата**: По завершении работы скрипта сгенерированный датасет будет сохранён в файле, указанном в `output_path`.

6.4 Пример изменения критериев для генерации данных

Допустим, вы хотите изменить вероятности времени открытия магазинов и добавить новую категорию товаров. Для этого:

1. **Изменение вероятностей времени открытия**:
   - При запуске `settings_generator.py` на шаге ввода распределения времени открытия вам будет предложено ввести веса для каждого времени (например, 7:00, 8:00, 9:00, 10:00, 11:00).
   - Вы можете установить более высокие веса для определённого времени, чтобы увеличить вероятность того, что магазины открываются в это время.

2. **Добавление новой категории товаров**:

- Во время ввода категорий товаров для каждой тематики магазина вы можете добавить свою категорию, введя её название.
- Далее, с помощью промптов к ChatGPT, будут сгенерированы соответствующие бренды и цены для новой категории.

3. **Настройка распределения платёжных систем**:

   - Вы можете изменить вероятности использования различных платёжных систем, введя новые веса для каждой из них.
   - Например, увеличить вероятность использования системы MASTERCARD или уменьшить использование VISA.

После внесения этих изменений и завершения работы `settings_generator.py`, запустите `data_generator.py` для генерации обновлённого датасета с учётом новых критериев.

7 Вывод

В ходе работы была разработана система для генерации синтетических данных о покупках в магазинах. С помощью скриптов `settings_generator.py` и `data_generator.py` был создан датасет, удовлетворяющий заданным требованиям и ограничениям, включая минимальный объём данных и реалистичность информации о магазинах, товарах и транзакциях.

Использование внешних API позволило получить актуальные данные о местоположении магазинов и ассортименте товаров, что повысило достоверность сгенерированных данных. Программа обеспечивает гибкую настройку параметров генерации, что позволяет адаптировать датасет под различные задачи.

Поставленные цели достигнуты: создан инструмент для генерации большого объёма реалистичных данных, пригодных для анализа и тестирования алгоритмов. Рекомендации, приведённые в работе, могут быть использованы для дальнейшего развития и улучшения программы.

8 Полезные ссылки

- Yandex Maps: `https://yandex.ru/maps`
- BIN List Service: `https://binlist.io`
- Блок-схемы проекта: `https://github.com/MansurYa/labs-for-algorithms-and-data-structures/blob/main/Lab1/code-flowchart-lab1.pdf`
- Репозиторий проекта: `https://github.com/MansurYa/labs-for-algorithms-and-data-structures/tree/main/Lab1`
- OpenAI Platform: `https://platform.openai.com`
- Yandex Developer Services: `https://developer.tech.yandex.ru/services`