Analiza średniej liczby powtórzonych zapytań przez użytkownika na podstawie sumulowanego strumienia danych

Mateusz Kacpura

15 listopada 2024

Spis treści

1 Wstęp

Celem niniejszego sprawozdania jest analiza średniej liczby powtórzonych zapytań przez użytkownika w ciągu tego samego dnia. Ze względu na ograniczenia pamięciowe (możemy przechowywać jedynie 10% strumienia danych), zastosowano próbkowanie proporcjonalne w dwóch implementacjach:

- implementacja naiwna,
- implementacja z hashowaniem dla klucza *użytkownik*.

Przeprowadzono testy na rzeczywistych danych, które zostały wczytane z pliku tekstowego. W sprawozdaniu przedstawiono kod w języku Python, omówiono metody oraz zaprezentowano otrzymane wyniki.

2 Dane

Analizowane dane pochodzą z pliku `log.txt` i zawierają wiersze w formacie:

```
Adres_IP Zapytanie
```

Przykładowa linia z pliku:

```
89.113.36.135 http://www.gingiva.coop
```

Dla celów analizy przyjęto, że dane te reprezentują strumień krotek `(użytkownik, zapytanie)`.

3 Metody

3.1 Opis funkcji

W celu przeprowadzenia analizy zaimplementowano następujące funkcje w języku Python:

3.1.1 `load_data_from_file`

Funkcja `load_data_from_file` wczytuje dane z pliku tekstowego i zwraca listę krotek `(użytkownik, zapytanie)`.

```python
import random
from collections import defaultdict

# Funkcja wczytująca dane z pliku
def load_data_from_file(filename):
    sample_data = []
    with open(filename, 'r') as file:
        for line in file:
            # Podział linii na użytkownika i zapytanie
            parts = line.strip().split()
            if len(parts) == 2:
                user, query = parts
                sample_data.append((user, query))
    return sample_data
```

Listing 1: Funkcja `load_data_from_file`

3.1.2 simulate_data

Funkcja `simulate_data` symuluje strumień zapytań na podstawie wczytanych danych. Dla każdego użytkownika generuje losową liczbę zapytań (od 1 do 10), które mogą się powtarzać.

```
# Symulacja strumienia zapytań na podstawie przykładowych danych
def simulate_data(sample_data):
    simulated_stream = []
    for user, query in sample_data:
        num_queries = random.randint(1, 10)  # Liczba zapytań od 1 do 10
        queries = [random.choice(sample_data)[1] for _ in range(num_queries)]
        for q in queries:
            simulated_stream.append((user, q))
    return simulated_stream
```

Listing 2: Funkcja `simulate_data`

3.1.3 naive_sampling

Implementacja naiwnego próbkowania proporcjonalnego, gdzie każdy rekord jest wybierany z prawdopodobieństwem p.

```
# Naiwne próbkowanie: losowe próbkowanie par użytkownik-zapytanie z
    prawdopodobieństwem p
def naive_sampling(stream, p=0.1):
    sample = [entry for entry in stream if random.random() < p]
    return sample
```

Listing 3: Funkcja `naive_sampling`

3.1.4 hash_sampling

Implementacja próbkowania z hashowaniem klucza *użytkownik*. Dzięki temu wszystkie zapytania wybranych użytkowników są uwzględniane.

```
# Hashowe próbkowanie: próbkowanie par użytkownik-zapytanie na podstawie
    shashowanego ID użytkownika
def hash_sampling(stream, p=0.1):
    sample = []
    users = set()
    for user, _ in stream:
        users.add(user)

    # Próbkowanie użytkowników na podstawie funkcji hash
    sampled_users = {user for user in users if (hash(user) % 1000) < p * 1000}
    sample = [(user, query) for user, query in stream if user in sampled_users]

    return sample
```

Listing 4: Funkcja `hash_sampling`

3.1.5 compute_average_repeats

Funkcja oblicza średnią liczbę powtórzonych zapytań na użytkownika.

```
# Obliczanie średniej liczby powtórzonych zapytań na użytkownika
def compute_average_repeats(sample):
    user_queries = defaultdict(list)

    # Grupowanie zapytań według użytkownika
    for user, query in sample:
        user_queries[user].append(query)

    total_repeats = 0
    total_users = len(user_queries)

    # Zliczanie powtórzonych zapytań dla każdego użytkownika
    for queries in user_queries.values():
        total_queries = len(queries)
        unique_queries = len(set(queries))
        repeats = total_queries - unique_queries
        total_repeats += repeats

    # Obliczenie średniej
    average_repeats_per_user = total_repeats / total_users if total_users > 0
        else 0
    return average_repeats_per_user
```

Listing 5: Funkcja compute_average_repeats

3.1.6 Główna część programu

Poniższy kod stanowi główną część programu, który wykorzystuje powyższe funkcje do analizy danych z pliku log.txt.

```
# Wczytanie danych z pliku log.txt
sample_data = load_data_from_file('log.txt')

# Symulacja strumienia zapytań
simulated_stream = simulate_data(sample_data)

# Naiwne próbkowanie
naive_sample = naive_sampling(simulated_stream, p=0.1)
naive_average_repeats = compute_average_repeats(naive_sample)

# Hashowe próbkowanie
hash_sample = hash_sampling(simulated_stream, p=0.1)
hash_average_repeats = compute_average_repeats(hash_sample)

# Wyświetlenie wyników
print(f" rednia  liczba powtórzonych zapytań na użytkownika (naiwne próbkowanie):
    {naive_average_repeats}")
print(f" rednia  liczba powtórzonych zapytań na użytkownika (hashowe
    próbkowanie): {hash_average_repeats}")
```

Listing 6: Główna część programu

4 Wyniki

Uruchomienie powyższego kodu na rzeczywistych danych z pliku `log.txt` dało następujące wyniki:

```
Średnia liczba powtórzonych zapytań na użytkownika (naiwne próbkowanie): 142.761
Średnia liczba powtórzonych zapytań na użytkownika (hashowe próbkowanie): 9077.836
```

5 Dyskusja

Z otrzymanych wyników wynika, że średnia liczba powtórzonych zapytań na użytkownika jest znacznie wyższa w przypadku hashowego próbkowania niż w przypadku naiwnego próbkowania. Przyczyną tego jest sposób, w jaki obie metody traktują dane użytkowników:

- **Naiwne próbkowanie**: Każda para `(użytkownik, zapytanie)` jest próbkowana niezależnie z prawdopodobieństwem $p = 0.1$. W rezultacie, dla danego użytkownika w próbie mogą znaleźć się tylko niektóre z jego zapytań, co prowadzi do mniejszej liczby powtórzeń i potencjalnie zaniżonej średniej liczby powtórzonych zapytań.
- **Hashowe próbkowanie**: Użytkownicy są próbkowani na podstawie wartości funkcji hash ich identyfikatora. Jeśli użytkownik zostanie wybrany, to wszystkie jego zapytania zostają uwzględnione w próbie. Dzięki temu zachowana jest kompletność danych dla wybranych użytkowników, co prowadzi do większej liczby powtórzonych zapytań i wyższej średniej.

Istotne jest również zwrócenie uwagi na generowanie danych w funkcji `simulate_data`. W tej funkcji dla każdego użytkownika przypisujemy losową liczbę zapytań, które są losowo wybierane z puli wszystkich zapytań. Może to prowadzić do powtórzeń, zwłaszcza przy ograniczonej liczbie unikalnych zapytań.

6 Wnioski

Przeprowadzona analiza wykazała, że metoda próbkowania z hashowaniem klucza *użytkownik* dostarcza dokładniejsze oszacowanie średniej liczby powtórzonych zapytań na użytkownika w ciągu dnia w porównaniu z naiwnym próbkowaniem. Dzieje się tak, ponieważ hashowe próbkowanie zachowuje pełne informacje o aktywności wybranych użytkowników, co jest kluczowe przy analizie wzorców zachowań.

W praktycznych zastosowaniach, gdzie istotne jest dokładne modelowanie aktywności użytkowników przy ograniczonych zasobach pamięciowych, zaleca się wykorzystanie metod opartych na hashowaniu kluczy identyfikujących użytkowników.

7 Bibliografia

Literatura

[1] Muthukrishnan, S. (2005). *Data streams: Algorithms and applications*. Now Publishers Inc.

[2] Leskovec, J., Rajaraman, A., & Ullman, J. D. (2014). *Mining of Massive Datasets*. Cambridge University Press.