Algoritmos de vizinhos mais próximos aproximados

Disciplina: TCC00288
Professora: Luiz André Portes Paes Leme

Barbara Keren Nascimento C. Guarino
Bruna de Assunção Santos
Gabriel Gavazzi Felix
Gabriel Ramalho Braga
Tatiana Machado Brito dos Santos
Vitor Balestro Dias da Silva

1 Descrição do problema

Informalmente, o problema dos k vizinhos mais próximos consiste em encontrar os k vetores de um dado conjunto S que estão mais próximos de um dado vetor v. Formalmente, sejam $k, n \in \mathbb{N}$, $S \subseteq \mathbb{R}^n$ e $v \in \mathbb{R}^n$. Assuma que S é um conjunto finito com m elementos, onde $m \geq k$, e seja d uma distância em $\mathbb{R}^n$. Considere a ordenação

$$S = \{v_1, \ldots, v_m\}$$

onde $d(v_j, v) \geq d(v_i, v)$ sempre que $j > i$ (isto é, S está ordenado pela distâncias a v). Desejamos encontrar o conjunto

$$\mathrm{NN}_k(v, S, d) := \{v_1, \ldots, v_k\}.$$

Não é difícil implementar um algoritmo computacional que forneça uma solução exata para este problema. Entretanto, a depender do tamanho de S, tal solução seria computacionalmente cara. De fato, é preciso calcular as distâncias dos elementos de S a v e, em seguida, ordená-las.

Desta forma, o desafio é desenvolver e implementar algoritmos que encontrem *aproximadamente* os k vizinhos mais próximos. Em linhas gerais, o objetivo é obter *boas* aproximações com *baixo* custo de execução. Note que os adjetivos em itálico referem-se a propriedades que são, ao menos em alguma medida, subjetivas.

Em uma descrição de alto nível, um *algoritmo de vizinhos mais próximos aproximados* (vamos chamá-lo de π) consiste me duas etapas:

(i) uma fase de *pré-processamento*, em que são construídas estruturas de dados para S;

(ii) a fase de processamento em que, dados um número $k \in \mathbb{N}$ e um vetor v, é retornado um conjunto $\mathrm{ANN}_k(v, S, d) = \{w_1, \ldots, w_{k^*}\} \subseteq S$ (com $k \leq k^*$) de vizinhos mais próximos aproximados de v.

Em linhas gerais, a qualidade da aproximação obtida é mensurada comparando o conjunto aproximado obtido $\mathrm{ANN}_k(\pi, v, S, d)$ com a solução exata (ordenada) $\mathrm{NN}(v, S, d) = \{v_1, \ldots, v_k\}$. Formalmente, definimos o *recall* da solução aproximada como

$$\mathrm{recall}(\pi) = \frac{\left|\left\{w \in \mathrm{ANN}_k(\pi, v, S, d) \colon d(w, v) \leq d(w, v_k)\right\}\right|}{k},$$

onde denotamos por $|A|$ a quantidade de elementos do conjunto A. Nesta definição, estamos contando quantos vetores da solução aproximada têm distância para v mais próxima do que v_k, que é o último vetor da solução exata ordenada.

Note que para calcular o recall, precisamos apenas de $d(w, v_k)$. Assim, esta distância será chamada *distância de referência* a partir de agora.

2 Tabelas

Consideramos o problema sobre um conjunto S de 1000000 de vetores em $\mathbb{R}^{128}$ com $k = 100$. No banco de dados, temos as seguintes tabelas:

(i) object: contém os vetores do conjunto S. Portanto, é uma tabela com 1000000 de linhas, em que cada linha é um vetor de 128 coordenadas;

(ii) tquery: contém um conjunto de 10000 casos de teste, para os quais a solução exata é conhecida ;

(iii) neighbor: contém as soluções exatas para os vetores da tabela **tquery**. Assim, é uma tabela de 10000 linhas em que cada linha é um vetor de 100 coordenadas.

Para o que segue, denote por tabela[j] a j-ésima linha de uma tabela, e por vetor[j] a j-ésima coordenada de um vetor. Para cada $1 \leq j \leq 10000$, o vetor neighbor[j] contém os *índices* dos 100 vetores da tabela object que estão (ordenadamente) mais próximos o vetor tquery[j]. Estes índices se referem à tabela object. Por exemplo, se

$$\text{neighbor}[10] = (20, 12, 14, \ldots),$$

então os vetores de object mais próximos do vetor tquery[10] são

$$\text{object}[20],\ \text{object}[12],\ \text{object}[14],\ \ldots$$

3 Pré-processamento

Na fase de pré-processamento, primeiro executamos um algoritmo de *clusterização* de S. Para este fim, usamos o algoritmo *KMeans* da biblioteca *scikit* da linguagem *Python*. Informalmente, este algoritmo agrupa os vetores de S em k conjuntos de vetores próximos entre si, e retorna os *centróides* destes conjuntos (pense no centróide como um vetor que é, de alguma forma, central dentro do conjunto). Consideramos $k = 128$, e isso significa que o algoritmo retornará uma tabela com 128 vetores de S (os *centróides*). Estes vetores estão armazenados na tabela **sight**. Esta tabela tem duas colunas: **id**, que é a chave primária (inteiros sequenciais) e **centroid**, que armazena o centróide correspondente (vetor de inteiros).

O próximo passo é criar uma tabela que informe o centróide mais próximo de cada vetor da tabela **object** (e a distância do vetor ao seu centróide mais próximo). Assim, dado um vetor **query**, podemos, por exemplo, restringir a busca aos vetores cujo centróide mais próximo é o centróide mais próximo de **query**. A tabela que guarda estas informações é chamada **closer_centroids**, e o código para gerá-la é exibido abaixo. Vale notar que esta etapa é custosa. Note que a chamada da função recebe como parâmetros a quantidade de vetores de **object** para os quais o centróide mais próximo será calculado e um valor de *offset*. Para evitar esgotamento de memória principal, dividimos a tabela rodando a função para cada uma de suas partes. Em um notebook de categoria intermediária, cada execução para 100000 vetores levou em torno de 4 minutos.

```
1 CREATE TYPE tuple AS (ind int, dist double precision);
2
3 CREATE TABLE closer_centroids (id INT NOT NULL, closer_centroid_ind INT, dist
     double precision);
4
5 CREATE OR REPLACE FUNCTION get_closer_centroids(qtd int,offs int) RETURNS void AS
      $$
6 DECLARE
7    object_line record;
8    sight_line record;
9    distances_vector tuple[];
10   current_tuple tuple;
11   dist double precision;
12   min_dist double precision;
13   i int;
14   ind int;
15 BEGIN
16   FOR object_line IN SELECT * FROM object ORDER BY id LIMIT qtd OFFSET offs LOOP
17      FOR sight_line IN SELECT * FROM sight_ LOOP
18         dist = euclidean_distance(object_line.features,sight_line.centroid);
19         current_tuple = (sight_line.id,dist);
20         distances_vector[sight_line.id] = current_tuple;
21      END LOOP;
22      min_dist = distances_vector[1].dist;
23      ind = 1;
24      FOR i in 1..128 LOOP
25         IF (distances_vector[i].dist < min_dist) THEN
26            min_dist = distances_vector[i].dist;
27            ind = distances_vector[i].ind;
28         END IF;
29      END LOOP;
30      INSERT INTO closer_centroids VALUES (object_line.id,ind, min_dist);
31   END LOOP;
32 END;
33 $$ LANGUAGE plpgsql;
34
```

Listing 1: Obtendo a tabela de centróides mais próximos

4 Calculando o recall

Para calcular o *recall* de uma determinada execução, primeiro obtemos a distância referência correspondente (através da tabela **neighbor**). Usamos a função explicitada abaixo para calcular a distância de referência do j-ésimo vetor da tabela **object** (isto é, o vetor cujo id é j).

```
1 CREATE OR REPLACE FUNCTION get_reference_distance(j int) RETURNS double precision
      AS
2 $$
3 DECLARE
4    query_vector int[];
5    neighbors_vector int[];
6    index_ int;
7    reference_vector double precision [];
8
9 BEGIN
10   query_vector := (SELECT query FROM tquery LIMIT 1 OFFSET j-1);
11   SELECT neighbors INTO neighbors_vector FROM neighbors WHERE id = j;
12   index_ := neighbors_vector[100];
13   SELECT features INTO reference_vector FROM object WHERE id = index_;
14
15   RETURN euclidean_distance(query_vector,reference_vector);
16
17 END;
18 $$ LANGUAGE plpgsql;
19
```

Listing 2: Calculando a distância de referência.

Agora, basta calcular as distâncias do conjunto de vizinhos aproximados obtidos para o vetor de *query* e comparar com a distância de referência deste vetor. Os vetores obtidos como resultado estão armazenados na tabela **result_table**.

```
1 CREATE OR REPLACE FUNCTION get_recall(query_vector_index int) RETURNS double
     precision AS
2 $$
3 DECLARE
4    i int;
5    query_vector double precision[];
6    result_vector double precision[];
7    reference_distance double precision;
8    hit_count int;
9    dist double precision;
10 BEGIN
11   hit_count := 0;
12   reference_distance := get_reference_distance(query_vector_index);
13   query_vector := (SELECT query FROM tquery WHERE id = j);
14   FOR i IN 1..100 LOOP
15      result_vector := (SELECT vec FROM result_table LIMIT 1 OFFSET i-1);
16      dist := euclidean_distance(query_vector,result_vector);
17      IF (dist <= reference_distance) THEN
18         hit_count := hit_count + 1;
19      END IF;
20   END LOOP;
21   RETURN hit_count::double precision / 100;
22 END;
23 $$ LANGUAGE plpgsql;
24
```

Listing 3: Calculando o *recall*.

5 Método 1: busca por regiões parametrizadas

Este método de busca dos 100 vetores mais próximos de um vetor $\mathbf{q}$ é como segue:

• **Passo 1:** encontramos o centróide $\mathbf{c}$ mais próximo de $\mathbf{q}$;

• **Passo 2:** computamos a distância d de $\mathbf{q}$ para $\mathbf{c}$;

• **Passo 3:** escolhemos parâmetros $0 < \alpha < 1$, $\beta > 1$ e $0 < \gamma < 1$ e, dentre os vetores $v \in S$ (tabela **object**) cujo centróide mais próximo é $\mathbf{c}$, encontramos aqueles que satisfazem:

(i) $\alpha d < d(v, \mathbf{c}) < \beta d$, e

(ii) $\gamma < \cos(v - \mathbf{c}, \mathbf{q} - \mathbf{c}) < 1$.

Isto é, consideramos a interseção entre a coroa circular com centro em $\mathbf{c}$ e raios αd e βd (interno e externo, respectivamente) com o cone de vetores w tais que o cosseno do ângulo entre $w - \mathbf{c}$ e $\mathbf{q} - \mathbf{c}$ é menor do que γ. Veja a Figura 5.1 para uma ilustração da região considerada.

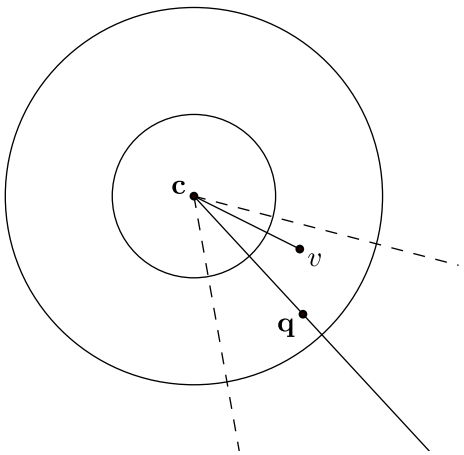

Figura 5.1: Ilustração em duas dimensões do método 1.

Para fins de simplicar a notação, a partir de agora vamos denotar por $S_{\mathbf{c}}$ o subconjunto dos vetores de S que têm $\mathbf{c}$ como centróide mais próximo.

Aqui, vale um comentário sobre a eficiência deste algoritmo. A tabela **closer_centroids** já fornece a distância entre cada vetor $v \in S_{\mathbf{c}}$ cujo centróide $\mathbf{c}$. Assim, nenhuma distância será calculada no passo 3. Mais ainda, lembre-se de que o valor do cosseno entre $v - \mathbf{c}$ e $\mathbf{q} - \mathbf{c}$ é computado pela fórmula

$$\cos(v - \mathbf{c}, \mathbf{q} - \mathbf{c}) = \frac{\langle v - \mathbf{c}, \mathbf{q} - \mathbf{c} \rangle}{||v - \mathbf{c}||\,||\mathbf{q} - \mathbf{c}||} = \frac{\langle v - \mathbf{c}, \mathbf{q} - \mathbf{c} \rangle}{d(v, c) \cdot d}$$

A norma de $v - \mathbf{c}$ é igual à distância entre v e $\mathbf{c}$ e, portanto, não precisará ser calculada. A norma de $\mathbf{q} - \mathbf{c}$ é igual à d, que foi calculado no passo 1. Daí, apenas o produto interno deverá ser computado. Ou seja, para cada $v \in S_{\mathbf{c}}$, o produto interno $\langle v - \mathbf{c}, \mathbf{q} - \mathbf{c} \rangle$ será calculado. Para calcular cada um destes produtos internos, são realizadas 128 multiplicações dentro de um *loop* que atualiza uma soma.

Observação 5.1. Note que para encontrar o conjunto $S_{\mathbf{c}}$, teremos que executar a seguinte consulta:

```
1
2 SELECT * FROM closer_centroids WHERE closer_centroid_ind = id(c);
3
```

Listing 4: Encontrando $S_{\mathbf{c}}$

onde id(c) é a chave do centróide **c** (na tabela **sight_**, é claro). Para otimizar estas consultas, criamos um índice na tabela **closer_centróides**:

```
1
2 CREATE INDEX closer_centroid_idx ON closer_centroids (closer_centroid_ind);
3
```

Listing 5: Índice na chave do centróide mais próximo

Esta consulta, que é feita originalmente sobre uma tabela com 1000000 de entradas, leva 60 milisegundos, em média (novamente, em um notebook pessoal de nível intermediário).

6 Referências

[1] M. Aumüller, E. Bernhardsson, A. Faithfull: *ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms*. Disponível em: https://arxiv.org/pdf/1807.05614.pdf