IF775 - Algoritmos para Stream de dados

Projeto 1

Rodrigo de Lima Oliveira - rlo

Gabriela Correia Holanda Freitas - gchf

rlo@cin.ufpe.br

gchf@cin.ufpe.br

25 de Julho de 2021

INTRODUÇÃO

Com o objetivo de consolidar os conhecimentos dos algoritmos vistos em sala, foram implementados 2 algoritmos para processamento de dados.

- HyperLogLog
- Weighted Sampling

O algoritmo HyperLogLog foi implementado e testado por Rodrigo Oliveira e o algoritmo Weighted Sampling foi implementado e testado por Gabriela Freitas. Cada um relatou sua parte na documentação.

Implementação

HLL

O HyperLogLog foi implementado em Python, utilizando apenas 3 libs não nativas no Python puro. Sendo elas **numpy** , a função de Hash adotada nessa implementação **xxhash** e a **scypi**.

A função Hash utilizada é de 32 bits, bem como a precisão.

O range de alpha adotado deve ser entre [4..16].

A implementação do HLL é basicamente dividida em 3 partes:

- Init dos parâmetros do HLL
- Insert
- Estimate

A seguir cada etapa será apresentada de forma separada para que cada detalhe necessário e importante seja explicado da maneira mais simples possível.

Init:

Como especificado no roteiro do projeto, o hll implementado deveria receber como parâmetros:

- **eps** (erro bound) : Especifica o valor do parâmetro **e** do estimador (limite do erro relativo desejado).
- **delta** (error probability) : Especifica o valor do parâmetro δ do estimador (limite para a probabilidade que o erro relativo da estimação seja superior

ao limite especificado).

Nessa etapa valem alguns comentários sobre o processo de implementação. Nos artigos originais de Flajolet não encontrei nenhuma referência direta ao relacionamento desses dois parâmetros. Contudo um trecho do artigo foi decisivo para entender esses parâmetros.

> *Let $\sigma \approx 1.04/\sqrt{m}$ represent the standard error; the estimates provided by* HyperLogLog *are expected to be within σ, 2σ, 3σ of the exact count in respectively 65%, 95%, 99% of all the cases.*

Os valores que estão multiplicando o δ me parecem ser o valor Z da tabela de distribuição normal arredondados que representam respectivamente 65%, 95%, 99%(δ, 2δ, 3δ) de grau de confiança.

Na implementação o valor delta desejado é dividido por 2 para obter o valor Z fornecido pela lib do **Stats (unilateral a esquerda por isso a divisão por 2).**

Esse valor é utilizado para regularizar o **eps** fornecido também como parâmetro de entrada(eps/z). A ideia aqui é: Como queremos um grau de certeza previamente definido devemos garantir que o algoritmo tenha meios de acertar com maior chance. Fazendo o eps aumentar ou diminuir influencia diretamente na quantidade de buckets criados e consequentemente na precisão e quantidade de memória utilizada.

```
def __init__(self, eps, delta):
      prob = 1 - (delta/2)
      z = st.norm.ppf(prob)
      eps /= z
      self.p = int(np.ceil(np.log2((1.04 / eps) ** 2)))
      self.m = 1 << self.p
      self.alpha = self.__get_alpha(self.p)
      self.seed = random.randrange(0, 1 << self.p)
      self.M = np.zeros((self.m,),dtype = np.int8)
```

Insert:

O insert segue basicamente o algoritmo descrito no artigo e apresentado em sala. Aqui utilizei a função xxhash, que tem desempenho melhor que a mais comumente MurmurHash.

https://github.com/Cyan4973/xxHash

```
def insert(self,value):
        x = xxhash.xxh32(bytes(value),seed=self.seed).intdigest()
        j = x & ((1 << self.p) - 1)
        # Remove those p bits
        #Ex: If p = 4
        #11111111
        #00001111
        w = x >> self.p
        # Find the first 0 in the remaining bit pattern
        self.M[j] = max(self.M[j], self.__get_rho(w) - self.p)
```

Estimate:

O estimate também seguiu estritamente o que foi apresentado no artigo e desenvolvido em sala. Fazendo as correções para small range, intermediate e large range.

```
def estimate(self):
       """ Returns the estimate of the cardinality """
       E = self.alpha * float(self.m ** 2) / np.power(2.0, - self.M).sum()
       if E <= 2.5 * self.m:
           V = self.m - np.count_nonzero(self.M)
           if V > 0:
               return self.__linear_counting(V)
           else:
               return int(E)

       elif E <= float(int(1) << 32) / 30.0:
           return int(E)
       else:
           return - self.m * np.log(1.0 - E / self.m)
```

DADOS HLL

Boa parte do tempo do insert acredito que seja causada pelo processo de leitura do arquivo uma vez que o tempo médio do processo de inserção, ou seja, do tempo médio da operação de insert é insignificante.

eps	delta	func	time s	space (M)
0.01	0.02	init	0.002	65,64 kb
0.01	0.02	main(insert + estimate)	561.66	
0.05	0.02	init	0.0023	4,2 kb
0.05	0.02	main(insert +	541.31	

		estimate)		
0.05	0.05	init	0.0022	2,1 kb
0.05	0.05	main(insert + estimate)	541.60	72mb

HLL

WS

O algoritmo Weighted Sampling foi implementado em linguagem c++, com a utilização das bibliotecas <iostream>, <stdlib>, <ctime>, <vector>, <algorithm>, <utility>, <random>, <iterator>, <fstream>, <sstream> e <string>. Foi criada uma classe chamada WS que para a implementação do algoritmo visto em aula, adaptado para a linguagem e para as exigências do projeto. Os valores retirados do arquivo de dados são introduzidos na amostra em pares (*pair <int, int>*), e a amostra está contida em um vetor de pares. A implementação da classe WS seguiu a seguinte sequência:

1 - Declaração de variáveis e vetores iniciais e suas inicialização de variáveis no construtor.

2 - métodos declarados para consultar valores, como os do vetor H (parcela da amostra que, no momento da consulta, é considerada pesada) e do vetor C (parcela da amostra que sabe-se que é leve).

3 - Método de atualização (*update)* da amostra, que é utilizado para introduzir novos valores a esta, e também para decidir que valores devem ser eliminados desta, de acordo com o algoritmo.

Especificidades do Código

- Apesar de diversas tentativas e bastante tempo pesquisando, não obtive êxito na tentativa de criar um script que compilasse o arquivo e o executasse, com as devidas exigências do projeto. Se a implementação tivesse sido feita em python, o êxito com certeza teria sido atingido, porém tentei seguir a preferência da especificação. O programa deve ser compilado e executado, e então as opções e valores devem ser inseridas e o código está funcionando. Para o próximo projeto isso será corrigido e feito devidamente.
- Foi acrescentado ao algoritmo o vetor de pares weight_vector, responsável por guardar os valores de identificação e peso respectivos aos elementos da amostra cujo valor do campo de número *field.no* (como especificado nas diretrizes do projeto) fosse igual ao *field_value.* Para a identificação desses valores foi adicionada a variável booleana *filter_weight* nos argumentos do método *update*. A variável é verdadeira quando os valores são iguais, e falsa quando não são.

```
if (filter_weight == 1)
        weight_vector.push_back(xi);
```

- Ao ser necessário remover algum elemento dos vetores C, ou H, fez-se necessária, também, a consulta do vetor weight_vector. Era checado se o par a ser removido estava ou não presente no weight_vector e, se sim, este par era também removido do último vetor.

```
i = 0;
while (p >= 0 && i < N.size()) {
        p -= (1 - N[i].second / self_tau);
        i++;
}
if (p < 0) {
        remove_weight(weight_vector, N[i-1].first);
        remove_at(N, i - 1);
}
else {
        remove_weight(weight_vector, remove_random(C));
}
```

- Um fato que não foi devidamente compreendido é porque o valor da variável *double* self_s menos o valor do tamanho do vetor H (H.size()) não estava dando o resultado correto. O valor certo foi obtido colocando primeiro o tamanho do vetor H em uma variável H_size inteira, e depois foi feita a subtração.

```
int H_size = H.size();
while ((H.size() > 0) && (W >= ((self_s - H_size) * H[0].second))) {
        aux.first = H[0].first;
        aux.second = H[0].second;
        H.erase(H.begin());
        N.push_back(aux);
        W += wx;
        H_size = H.size(); //por algum motivo nao funcionou o s - H.size() direto
}
```

- O código para o valor de p, aleatório e com distribuição uniforme, foi retirado da documentação online do c++ [referência 6], com as devidas mudanças para que a função possa retornar um intervalo fechado [0, 1].

```
random_device rd;  //Will be used to obtain a seed for the random number engine
mt19937 gen(rd()); //Standard mersenne_twister_engine seeded with rd()
uniform_real_distribution<> dis(0.0, nextafter(1, numeric_limits<double>::max()));
double p = dis(gen);
```

- E esta foi mais uma função útil encontrada em um tutorial [referência 7] de como ordenar vetores de pares pelo segundo argumento pois, por padrão, a ordenação é feita considerando o primeiro, e foi decidido seguir a sequência <id, peso>.

```
bool sortbysec(const pair<int, int>& a,
               const pair<int, int>& b)
{
               return (a.second < b.second);
}
```

- A classe CSVRow foi, também, retirada da internet [referência 5], e ela é utilizada para a leitura do arquivo .csv. O motivo para a cópia é apenas a limitação de linguagem da programadora, que não está completamente familiarizada com a linguagem c++.

```
class CSVRow
{
public:
        string const& operator[](size_t index) const
        {
                return m_data[index];
        }
        size_t size() const
        {
                return m_data.size();
        }
        void readNextRow(istream& str)
        {
                string         line;
                getline(str, line);

                stringstream   lineStream(line);
                string         cell;

                m_data.clear();
                while (getline(lineStream, cell, ','))
                {
                        m_data.push_back(cell);
                }
                // This checks for a trailing comma with no data after it.
                if (!lineStream && cell.empty())
                {
                        // If there was a trailing comma then add an empty element.
                        m_data.push_back("");
                }
        }
private:
        vector<string>    m_data;
};

istream& operator>>(istream& str, CSVRow& data)
{
        data.readNextRow(str);
        return str;
}
```

Testes e Resultados

- Os testes foram realizados sob os dados fornecidos na especificação do projeto do dataset de tráfego de rede, network_flows_unique.csv.
- Para a medição de tempo foi utilizada a biblioteca <time.h> e a utilização de variáveis (clock_t inicio = clock()) e (clock_t fim = clock()). Depois foi feita a subtração (fim - inicio) / CLOCKS_PER_SEC para encontrar o tempo de execução em segundos.
- Foi feito um loop de tamanho 10 na execução do algoritmo, com coluna id = 0, coluna weight = 3, protocolName = GOOGLE e uma amostra de tamanho 100. O resultado da média da soma de pesos da amostra com o filtro de protocolo foi 115.

```
--id 0 --weight 3 --filter 8 GOOGLE --size 100 network_flows_unique.csv
115
```

- Foi repetido o processo individual com 5 amostras, para checar a acurácia da média nas 10 amostras. A média de resultado das 5 amostras foi de 120.

Resultado	Tempo
144	79.820s
78	79.025s
156	79.053s
126	75.300s
96	77.316s

- Sob as mesmas especificações, é possível perceber também que a média também é diretamente proporcional ao tamanho da amostra. Para esta conclusão foi executado um loop de tamanho 10 no algoritmo com tamanhos diferentes de amostras.

Tamanho da amostra	Média
100	115
50	68

REFERÊNCIAS

1. http://cscubs.cs.uni-bonn.de/2016/proceedings/paper-03.pdf
2. http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf
3. https://djhworld.github.io/hyperloglog/adding/
4. https://medium.com/@winwardo/counting-crowds-hyperloglog-in-simple-terms-1d345637db5
5. https://www.ti-enxame.com/pt/c%2B%2B/como-posso-ler-e-analisar-arquivos-csv-em-c/967277856/
6. https://www.cplusplus.com/reference/random/uniform_real_distribution/
7. https://www.geeksforgeeks.org/sorting-vector-of-pairs-in-c-set-1-sort-by-first-and-second/