# Locality-sensitive Hashing

Discente: Thiago Oliveira França

MATA 54 - ESTRUTURAS DE DADOS E ALGORITMOS II

# Introdução

Como plataformas de serviços, como o YouTube, sabem o que recomendar com base no que estamos consumindo?

# Motivação

Por exemplo, dado um vídeo no YouTube, podemos comparar esse vídeo com outros vídeos e verificar se há semelhanças. No entanto, esse processo de comparação entre todos os vídeos pode ter um custo de $O(n^2)$, o que se torna inviável em bancos de dados grandes como o do YouTube.

# Locality-sensitive hashing

O algoritmo Locality-Sensitive Hashing (LSH) surge para reduzir o número de comparações necessárias, tornando mais eficiente o processo de encontrar itens semelhantes.

O objetivo do LSH é agrupar itens semelhantes em um mesmo "bucket" (balde), facilitando a busca por semelhanças sem a necessidade de realizar comparações exaustivas entre todos os itens.

O LSH é composto por três etapas principais:

1. Shingling
2. MinHashing
3. LSH (Locality-Sensitive Hashing)

# Shingling

O primeiro passo do LSH consiste em dividir uma string em substrings de tamanho fixo, chamadas "shingles". Essas substrings representam características da palavra ou documento.

Para cada i na palavra
 singles += palavra[i] + palavra[i+1]

O processo de lsh será simulando a comparação de semelhança de palavras

# Exemplo

“algoritmo”

lista_shingles = [”al”]

# Exemplo

“algoritmo”

lista_shingles = [”al”, “lg”]

# Exemplo

“algoritmo”

lista_shingles = [”al”, “lg”, “go”]

# Exemplo

“algoritmo”

lista_shingles = [”al”, “lg”, “go”, “or“]

# Exemplo

“algoritmo”

lista_shingles = [”al”, “lg”, “go”, “or“, ”ri”]

# Exemplo

“algoritmo”

lista_shingles = [”al”, “lg”, “go”, “or“, ”ri”, “it”]

# Exemplo

“algoritmo”

lista_shingles = [”al”, “lg”, “go”, “or“, ”ri”, “it,”, “tm”]

# Exemplo

“algoritmo”

lista_shingles = [”al”, “lg”, “go”, “or“, ”ri”, “it,”,
“tm”, “mo”]

# MinHashing

Após realizar o processo de Shingling para uma palavra *k* e seja *m* um vocabulário de substrings possíveis, marcamos as substrings presentes em *k* em uma lista de tamanho igual ao comprimento do vocabulário.

# Exemplo

Vocabulario = {

“ab”: 0, “ca”:1, “sa”: 2, “va”: 3, “lo”: 4, “as”: 5

}

String “casa”

lista_shingles = [”ca”, “as”, “sa”]

hash_bitmap = [0,1,0,0,0,0]

# Exemplo

Vocabulario = {

“ab”: 0, “ca”:1, “sa”: 2, “va”: 3, “lo”: 4, “as”: 5

}

String “casa”

lista_shingles = [”ca”, “as”, “sa”]

hash_bitmap = [0,1,0,0,0,1]

# Exemplo

Vocabulario = {

“ab”: 0, “ca”:1, “sa”: 2, “va”: 3, “lo”: 4, “as”: 5

}

String “casa”

lista_shingles = [”ca”, “as”, “sa”]

hash_bitmap = [0,1,1,0,0,1]

# MinHashing

Em seguida, geramos várias funções hash para criar permutações dessa lista

Buscamos, nessas novas listas permutadas, a primeira ocorrência do valor 1 e criamos uma lista de assinatura, que é usada para comparar palavras de forma eficiente.

hash_bitmap = [0,1,1,0,0,1]

# Geramos listas permutadas
permutacoes = [[0,0,1,1,1,0], [0,1,0,1,0,1], [1,0,0,0,1,1]]

# Procuramos a primeira ocorrência de 1 em cada permutação
assinatura = [2, 1, 0]

# Lsh

O último passo é dividir a assinatura gerada pelo MinHashing em "bandas" de tamanho *b*.

Para cada banda, utilizamos uma função hash para escolher uma posição na lista de baldes, agrupando palavras semelhantes em um mesmo balde. Isso reduz o número de comparações necessárias, pois só comparamos palavras dentro do mesmo balde.

assinatura = [2,1,0,...]

#Podemos dividir em duas bandas
banda1 = [2,1,.., n]
banda2 = [n+1,...len(assinatura)]

#Geramos hashs para cada banda e colocamos a palalvra no balde cujo indice é o valor da hash, ou seja uma palavra pode estar em mais de um balde ou nenhum balde

# Algoritmo (Python)

```python
tabelaHashingBaldes = [[] for _ in range(51)]
vocabulario = {
    "aa": 0, "ab": 1, "ac": 2,
    "ba": 3, "bb": 4, "bc": 5,
    "ca": 7, "cb": 8, "cc": 9,
    }

n = len(vocabulario)

#Shingles

def shingles (palavra):
    return (palavra[i:i+2] for i in range(len(palavra) - 1))
```

```python
#Funcao Principal
def lsh(palavra):

    shingles_lista = []
    shingles_lista.append(shingles(palavra)) #[aa, ab, ba, ac, ca, aa, ac, ca, aa]

    #minHashing
    hash_bitmap = [0] * (n)
    min_hash_bitmap = [0] * n
    assinatura = []

    for i in range(len(shingles_lista)):
        if (shingles_lista[i] in vocabulario):
            hash_bitmap[vocabulario[shingles_lista[i]]] = 1 #marcar na lista os shingles que estão presentes na palavra

    for k in range(len(hash_bitmap)):
        if hash_bitmap[k] == 1:
            min_hash_bitmap[(random.randint(0, 51)) % n] = 1 #gerar um valor aleatório para cada shingle presente na palavra

    menores = [float('inf')] * 20 #inicializa a lista de menores com infinito
```

```
    for i in range(len(min_hash_bitmap)):
        if min_hash_bitmap[i] == 1:
            hashes = [(random.randrange(0, 100) * i) % n for _ in range(20)] #Gera funções aleatórias para cada shingle

            for j in range(20):
                if hashes[j] < menores[j]: #se o valor gerado for menor que o valor atual, substitui
                    menores[j] = hashes[j]
    assinatura = menores
```

```
76      #lsh
77
78      bandas = []
79      banda_size = 2  # Cada banda contém 2 valores da assinatura
80      for i in range(0, len(assinatura), banda_size):
81          bandas.append(assinatura[i:i + banda_size])
82
83      hash_bandas = {(random.randint(0, 100)) % 51 for _ in range(len(bandas))}
84
85      for hash_banda in hash_bandas:
86          tabelaHashingBaldes[hash_banda].append(palavra)
```

```
88    lsh("abacabacaa")
89    lsh("abaaacacab")
90    lsh("bbacbacaac")
91    lsh("bbbcacaaba")
92    lsh("ccaccbbbbb")
93    lsh("ccbcaaaacc")
94
95    print(tabelaHashingBaldes)
```

```
[[], ['bbbcacaaba'], [], [], [], ['abaaacacab', 'bbbcacaaba'], ['ccaccbbbbb', 'ccbcaaaacc'], ['bbacbaca
ac'], ['bbacbacaac'], ['abacabacaa', 'bbacbacaac', 'bbbcacaaba'], ['bbbcacaaba'], [], [], ['ccbcaaaacc'
], [], ['ccaccbbbbb'], [], ['abaaacacab', 'ccbcaaaacc'], [], ['abacabacaa', 'abaaacacab'], ['ccaccbbbbb
'], ['abaaacacab', 'ccaccbbbbb'], ['abacabacaa', 'bbacbacaac', 'bbbcacaaba', 'ccaccbbbbb'], [], ['abaca
bacaa', 'abaaacacab', 'bbacbacaac', 'bbbcacaaba', 'ccbcaaaacc'], [], ['bbacbacaac'], [], ['ccbcaaaacc']
, ['bbacbacaac', 'ccbcaaaacc'], ['abaaacacab', 'ccaccbbbbb'], ['bbacbacaac'], ['abaaacacab', 'bbbcacaab
a'], [], ['abacabacaa'], ['bbbcacaaba', 'ccbcaaaacc'], ['abacabacaa'], ['abaaacacab'], ['abacabacaa'],
['ccaccbbbbb'], ['bbacbacaac', 'ccbcaaaacc'], [], [], [], ['ccaccbbbbb'], ['bbbcacaaba'], ['abacabacaa'
], ['abaaacacab', 'ccaccbbbbb'], ['ccbcaaaacc'], ['bbacbacaac'], ['abacabacaa']]
```

```python
#Calcula a similaridade de Jaccard entre duas palavras
def calcular_jaccard(palavra1, palavra2):
    shingles1 = set(shingles(palavra1))
    shingles2 = set(shingles(palavra2))

    intersecao = shingles1 & shingles2
    uniao = shingles1 | shingles2

    similaridade = len(intersecao) / len(uniao)
    return similaridade

# Função para comparar todas as palavras em um balde usando a similaridade de Jaccard
def comparar_balde_lsh(balde):
    n = len(balde)

    if n == 1:
        print("O balde contém apenas uma palavra!")
        return
    for i in range(n):
        for j in range(i + 1, n):
            palavra1 = balde[i]
            palavra2 = balde[j]
            similaridade = calcular_jaccard(palavra1, palavra2)
            print(f"Similaridade entre '{palavra1}' e '{palavra2}': {similaridade:.2f}")
```

```python
# Testando a função com o balde específico
for i, balde in enumerate(tabelaHashingBaldes):
    if balde:
        print(f"\nComparando palavras no balde {i}:")
        comparar balde lsh(balde)
```

```
Similaridade entre 'abaaacacab' e 'bbbcacaaba': 0.71

Comparando palavras no balde 19:
Similaridade entre 'abacabacaa' e 'bbbcacaaba': 0.71

Comparando palavras no balde 20:
O balde contém apenas uma palavra!

Comparando palavras no balde 21:
Similaridade entre 'abacabacaa' e 'bbacbacaac': 0.57

Comparando palavras no balde 22:
O balde contém apenas uma palavra!

Comparando palavras no balde 23:
Similaridade entre 'abacabacaa' e 'bbbcacaaba': 0.71

Comparando palavras no balde 25:
Similaridade entre 'ccaccbbbbb' e 'ccbcaaaacc': 0.57
```

# Conclusão

O algoritmo LSH consegue reduzir a complexidade do problema de compações de $O(n^2)$ para $O(n + k)$ em que k é o numero de comparações diretas dentro de cada balde.

Ferramenta eficiente para lidar com grande quantidade de dados.

Aplicações: Sistema de recomendações (como por exemplo musicas no Spotify, Youtube, amazon, google, facebook), busca de arquivos (como documento ou imagens), corretores ortográficos, compreensao de dados.

# Referências

EFIMOV, Vyacheslav. Similarity Search, Part 5: Locality Sensitive Hashing (LSH)
Explore how similarity information can be incorporated into hash function
**Towards Data Science**, 2023. Disponível em: https://towardsdatascience.com/similarity-search-part-5-locality-sensitive-hashing-lsh-76ae4b388203. Acesso em: 28 jan. 2025.

https://www.pinecone.io/learn/series/faiss/locality-sensitive-hashing/ BRIGGS, James. Locality Sensitive Hashing (LSH): The Illustrated Guide, **Pinecone**, Disponível em: https://www.pinecone.io/learn/series/faiss/locality-sensitive-hashing/. Acesso em: 28 jan. 2025.

RAJARAMAN, Anand , ULLMAN, Jeffrey D..Finding Similar Items, In: Mining of Massive Datasets, 2010. Disponível em: http://infolab.stanford.edu/~ullman/mmds/booka.pdf. Acesso em: 28 jan. 2025.