05duplicate_detection

November 9, 2025

1 CAIM Lab Session 5: Duplicate detection using simhash

Guillem Cabré, David Morais

Exercise 1:

As a little exercise, suppose that a document d contains *two* non-zero entries, which correspond to terms `bit` with tf-idf weight 0.4 and `coin` with tf-idf weight 1.2. The md5 hash code for `bit` is 1111 and the md5 code for `coin` is 1001 (so $b = 4$). Compute the binary simhash for this document with $k = 4, m = 1$. (*Hint: result should be 1001*)

Vamos a calcular el simhash de un documento que contiene dos términos: `bit` con peso tf-idf 0.4 y `coin` con peso tf-idf 1.2. Los hashes md5 son 1111 para `bit` y 1001 para `coin`.

Term	peso tf-idf	md5 hash (4 bits)	(h'_t) (reemplazar 0 con -1)
bit	0.4	1111	([1, 1, 1, 1] –> [1, 1, 1, 1])
coin	1.2	1001	([1, 0, 0, 1] –> [1, -1, -1, 1])

Por lo tanto la *weighted sum* es:

$$simhash(d) = 0.4 * [1, 1, 1, 1] + 1.2 * [1, -1, -1, 1] = [1.6, -0.8, -0.8, 1.6]$$

Ahora convertimos esta suma en un vector binario usando el signo de las entradas:

$$simhash(d) = [1, 0, 0, 1]$$

En este ejercicio se muestra manualmente cómo se calcula el vector simhash de un documento a partir de los hashes de sus términos y sus pesos tf-idf. Se obtiene una representación binaria que refleja en qué lado del hiperplano cae el documento respecto a las proyecciones aleatorias. El resultado [1, 0, 0, 1] indica la huella binaria final, que sirve como representación compacta y eficiente para detectar similitud entre documentos.

1.0.1 Cargar el corpus de documentos, y obtención del hash de un término

[1]:
```
import hashlib

def _termhash(x : str, b : int) -> str:
    """returns bitstring of size b based on md5 algorithm"""
    assert b <= 128, 'this encoding scheme supports hashes of length at most␣
 ↪128; try smaller b'
    h = hashlib.md5(x.encode('utf8')).digest()
    return ''.join(format(byte, '08b') for byte in h)[:b]

import pickle
import os

fname = '05corpus.pkl'

with open(fname, 'rb') as f:
    corpus = pickle.load(f)
```

1.1 3. Simhashing scheme and near-duplicate collection and verification

---

Exercise 2:

Write the `simhash` function provided below. Make sure you understand that each bit of the simhash encodes which "side" the tf-idf document representation falls on a (pseudo-)random projection.

---

[2]:
```
from tqdm import tqdm
import numpy as np

## constants
K = 18
M = 3
B = M*K

def _simhash(id, b):
    """
        id is the document id, b is the desired length of the simhash
        it should return a bitstring of length b as explained in the
        first section of the notebook
    """
    doc = corpus[id]
    simhash_vector = np.zeros(b)

    # For each term in the document
    for term, weight in doc.items():
```

```
        hash_bits = _termhash(term, b)
        h_prime = np.array([1 if bit == '1' else -1 for bit in hash_bits])
        simhash_vector += h_prime * weight

    binary_simhash = ''.join(['1' if val > 0 else '0' for val in␣
 ↪simhash_vector])
    return binary_simhash

# simhash stores all simhashes of the whole corpus in a dictionary
simhash = {id: _simhash(id, B) for id in tqdm(corpus)}
```

```
100%|      | 58102/58102 [01:36<00:00, 602.40it/s]
```

Esta función implementa el cálculo automático del vector simhash a partir de las representaciones tf-idf. Para cada término del documento, se genera un hash binario de longitud b que se convierte a una representación de 1 y -1. Luego, se pondera por su peso tf-idf y se acumula la suma por cada dimensión del hash. Finalmente, se aplica la función signo para obtener el vector binario final (1 si el valor es positivo, 0 si es negativo). Este proceso convierte la representación de alta dimensión de un documento en una firma binaria de tamaño fijo que preserva la similitud semántica.

Exercise 3:

Based on simhashes, write code that places the documents in their corresponding lsh hash tables' buckets across m repetitions. Once, the lsh hash tables have been populated, find all potential near-duplicate candidate pairs. Check among all candidates their real cosine similarity and based on this determine true positives (pairs of documents that are indeed similar and have collided in some table), and false positives (pairs of documents that are not similar but have collided in some table). Compute speed of the method vs. false positives / true positives for different values of m and k. Make sure that k is not tiny (say greater than 10) if you want the method to be fast.

[3]:
```
import pandas as pd
import numpy as np
from collections import defaultdict
from itertools import combinations
import time
from tqdm import tqdm
```

[4]:
```
from collections import defaultdict
from itertools import combinations
import time

def cosine_similarity_fast(id1, id2):
    doc1 = corpus[id1]
    doc2 = corpus[id2]

    common_terms = doc1.keys() & doc2.keys()
    if not common_terms:
```

```
        return 0.0

    dot_product = sum(doc1[term] * doc2[term] for term in common_terms)
    return dot_product / (doc_norms[id1] * doc_norms[id2])

def lsh_buckets(k, m):
    """
    Create LSH hash tables and find candidate pairs
    k: number of bits per chunk
    m: number of hash tables (repetitions)
    """
    b = k * m

    # Create m hash tables, each with buckets indexed by k-bit chunks
    hash_tables = [defaultdict(list) for _ in range(m)]

    # Place documents into buckets
    for doc_id in corpus.keys():
        # Get or compute simhash for this document
        if doc_id in simhash and len(simhash[doc_id]) == b:
            sh = simhash[doc_id]
        else:
            sh = _simhash(doc_id, b)

        # Split into m chunks of k bits
        for table_idx in range(m):
            start = table_idx * k
            end = start + k
            chunk = sh[start:end]

            # Convert chunk to integer for bucket indexing
            bucket_id = int(chunk, 2)
            hash_tables[table_idx][bucket_id].append(doc_id)

    # Find candidate pairs (documents that share at least one bucket)
    candidates = set()
    for table in hash_tables:
        for bucket_id, doc_list in table.items():
            if len(doc_list) > 1:
                # All pairs in this bucket are candidates
                for pair in combinations(sorted(doc_list), 2):
                    candidates.add(pair)

    return candidates, hash_tables
```

[5]:
```
doc_norms = {}
for doc_id, doc in tqdm(corpus.items(), desc="Computing norms"):
```

```
    doc_norms[doc_id] = np.sqrt(sum(w**2 for w in doc.values()))

results = []
total_comparisons = len(corpus) * (len(corpus) - 1) / 2

for k in range(12, 20, 2):
    for m in range(2, 6):
        b = k * m

        print(f"\n{'='*60}")
        print(f"Running LSH with k={k}, m={m}, b={b}")
        print(f"{'='*60}")

        # Precalcular hashes de términos para este b
        print("Precomputing term hashes...")
        all_terms = set()
        for doc in corpus.values():
            all_terms.update(doc.keys())

        term_hashes = {}
        for term in all_terms:
            hash_bits = _termhash(term, b)
            term_hashes[term] = np.array([1 if bit == '1' else -1 for bit in␣
↪hash_bits])

        # Calcular simhashes
        print("Computing simhashes...")
        simhash_local = {}
        for doc_id in tqdm(corpus.keys()):
            doc = corpus[doc_id]
            simhash_vector = np.zeros(b)
            for term, weight in doc.items():
                simhash_vector += term_hashes[term] * weight
            simhash_local[doc_id] = ''.join(['1' if val > 0 else '0' for val in␣
↪simhash_vector])

        # LSH
        start_time = time.time()
        candidates, _ = lsh_buckets(k, m)
        lsh_time = time.time() - start_time

        print(f"LSH time: {lsh_time:.2f}s, Candidates: {len(candidates)}")

        # Calcular similitudes
        start_time = time.time()
        similarities = {pair: cosine_similarity_fast(*pair) for pair in␣
↪tqdm(candidates, desc="Computing similarities")}
```

```
        similarity_time = time.time() - start_time

        # Análisis
        threshold = 0.8
        true_positives = sum(1 for sim in similarities.values() if sim >=␣
↪threshold)
        false_positives = len(candidates) - true_positives

        precision = true_positives / len(candidates) if len(candidates) > 0␣
↪else 0
        total_time = lsh_time + similarity_time
        speedup = total_comparisons / len(candidates) if len(candidates) > 0␣
↪else float('inf')

        print(f"TP: {true_positives}, FP: {false_positives}, Precision:␣
↪{precision:.4f}")
        print(f"Total time: {total_time:.2f}s, Speedup: {speedup:.2f}x")

        results.append({
            'k': k, 'm': m, 'b': b,
            'candidates': len(candidates),
            'true_positives': true_positives,
            'false_positives': false_positives,
            'precision': precision,
            'time': total_time,
            'speedup': speedup
        })

results_df = pd.DataFrame(results)
print("\n" + "="*60)
print("SUMMARY")
print("="*60)
print(results_df.to_string(index=False))
```

```
Computing norms:   0%|          | 0/58102 [00:00<?, ?it/s]
Computing norms: 100%|     | 58102/58102 [00:01<00:00, 53024.59it/s]


============================================================
Running LSH with k=12, m=2, b=24
============================================================
Precomputing term hashes…
Computing simhashes…
100%|     | 58102/58102 [00:13<00:00, 4382.37it/s]
LSH time: 75.59s, Candidates: 919679
Computing similarities: 100%|     | 919679/919679 [00:11<00:00,
```

```
81499.64it/s]

TP: 16871, FP: 902808, Precision: 0.0183
Total time: 86.87s, Speedup: 1835.31x

============================================================
Running LSH with k=12, m=3, b=36
============================================================
Precomputing term hashes…
Computing simhashes…

100%|      | 58102/58102 [00:13<00:00, 4316.64it/s]

LSH time: 88.84s, Candidates: 1392508

Computing similarities: 100%|      | 1392508/1392508 [00:18<00:00,
73451.99it/s]

TP: 17878, FP: 1374630, Precision: 0.0128
Total time: 107.99s, Speedup: 1212.12x

============================================================
Running LSH with k=12, m=4, b=48
============================================================
Precomputing term hashes…
Computing simhashes…

100%|      | 58102/58102 [00:14<00:00, 3976.93it/s]

LSH time: 92.54s, Candidates: 1899388

Computing similarities: 100%|      | 1899388/1899388 [00:23<00:00,
79520.53it/s]

TP: 18509, FP: 1880879, Precision: 0.0097
Total time: 116.68s, Speedup: 888.65x

============================================================
Running LSH with k=12, m=5, b=60
============================================================
Precomputing term hashes…
Computing simhashes…

100%|      | 58102/58102 [00:14<00:00, 4077.40it/s]

LSH time: 102.76s, Candidates: 2350590

Computing similarities: 100%|      | 2350590/2350590 [00:30<00:00,
76709.84it/s]

TP: 18934, FP: 2331656, Precision: 0.0081
Total time: 133.77s, Speedup: 718.07x

============================================================
```

```
Running LSH with k=14, m=2, b=28
============================================================
Precomputing term hashes…
Computing simhashes…
100%|      | 58102/58102 [00:14<00:00, 4096.02it/s]
LSH time: 79.08s, Candidates: 251097
Computing similarities: 100%|      | 251097/251097 [00:03<00:00,
69878.92it/s]
TP: 16359, FP: 234738, Precision: 0.0652
Total time: 83.21s, Speedup: 6722.07x

============================================================
Running LSH with k=14, m=3, b=42
============================================================
Precomputing term hashes…
Computing simhashes…
100%|      | 58102/58102 [00:13<00:00, 4357.54it/s]
LSH time: 93.07s, Candidates: 373531
Computing similarities: 100%|      | 373531/373531 [00:05<00:00,
73264.58it/s]
TP: 17239, FP: 356292, Precision: 0.0462
Total time: 98.21s, Speedup: 4518.75x

============================================================
Running LSH with k=14, m=4, b=56
============================================================
Precomputing term hashes…
Computing simhashes…
100%|      | 58102/58102 [00:14<00:00, 3948.48it/s]
LSH time: 101.31s, Candidates: 497276
Computing similarities: 100%|      | 497276/497276 [00:06<00:00,
73321.49it/s]
TP: 17870, FP: 479406, Precision: 0.0359
Total time: 108.16s, Speedup: 3394.28x

============================================================
Running LSH with k=14, m=5, b=70
============================================================
Precomputing term hashes…
Computing simhashes…
100%|      | 58102/58102 [00:15<00:00, 3814.73it/s]
```

```
LSH time: 111.78s, Candidates: 608925

Computing similarities: 100%|      | 608925/608925 [00:08<00:00,
71991.62it/s]

TP: 18289, FP: 590636, Precision: 0.0300
Total time: 120.34s, Speedup: 2771.92x

============================================================
Running LSH with k=16, m=2, b=32
============================================================
Precomputing term hashes…
Computing simhashes…

100%|      | 58102/58102 [00:15<00:00, 3862.77it/s]

LSH time: 84.10s, Candidates: 75363

Computing similarities: 100%|      | 75363/75363 [00:01<00:00, 56637.71it/s]

TP: 15729, FP: 59634, Precision: 0.2087
Total time: 85.54s, Speedup: 22396.83x

============================================================
Running LSH with k=16, m=3, b=48
============================================================
Precomputing term hashes…
Computing simhashes…

100%|      | 58102/58102 [00:14<00:00, 3916.95it/s]

LSH time: 97.41s, Candidates: 109531

Computing similarities: 100%|      | 109531/109531 [00:01<00:00,
59861.94it/s]

TP: 16685, FP: 92846, Precision: 0.1523
Total time: 99.26s, Speedup: 15410.18x

============================================================
Running LSH with k=16, m=4, b=64
============================================================
Precomputing term hashes…
Computing simhashes…

100%|      | 58102/58102 [00:14<00:00, 3959.27it/s]

LSH time: 103.48s, Candidates: 139005

Computing similarities: 100%|      | 139005/139005 [00:02<00:00,
64323.05it/s]

TP: 17311, FP: 121694, Precision: 0.1245
Total time: 105.67s, Speedup: 12142.67x
```

```
============================================================
Running LSH with k=16, m=5, b=80
============================================================
Precomputing term hashes…
Computing simhashes…
100%|      | 58102/58102 [00:14<00:00, 3896.10it/s]
LSH time: 114.28s, Candidates: 168077
Computing similarities: 100%|      | 168077/168077 [00:02<00:00,
67555.79it/s]
TP: 17716, FP: 150361, Precision: 0.1054
Total time: 116.79s, Speedup: 10042.37x

============================================================
Running LSH with k=18, m=2, b=36
============================================================
Precomputing term hashes…
Computing simhashes…
100%|      | 58102/58102 [00:13<00:00, 4238.37it/s]
LSH time: 84.78s, Candidates: 30853
Computing similarities: 100%|      | 30853/30853 [00:00<00:00, 37773.35it/s]
TP: 15336, FP: 15517, Precision: 0.4971
Total time: 85.64s, Speedup: 54707.55x

============================================================
Running LSH with k=18, m=3, b=54
============================================================
Precomputing term hashes…
Computing simhashes…
100%|      | 58102/58102 [00:15<00:00, 3843.39it/s]
LSH time: 0.38s, Candidates: 40276
Computing similarities: 100%|      | 40276/40276 [00:00<00:00, 44252.54it/s]
TP: 16244, FP: 24032, Precision: 0.4033
Total time: 1.30s, Speedup: 41908.14x

============================================================
Running LSH with k=18, m=4, b=72
============================================================
Precomputing term hashes…
Computing simhashes…
100%|      | 58102/58102 [00:15<00:00, 3784.88it/s]
LSH time: 108.81s, Candidates: 48027
```

```
Computing similarities: 100%|      | 48027/48027 [00:01<00:00, 36739.05it/s]

TP: 16824, FP: 31203, Precision: 0.3503
Total time: 110.13s, Speedup: 35144.65x

============================================================
Running LSH with k=18, m=5, b=90
============================================================
Precomputing term hashes…
Computing simhashes…

100%|      | 58102/58102 [00:16<00:00, 3519.10it/s]

LSH time: 133.64s, Candidates: 56020

Computing similarities: 100%|      | 56020/56020 [00:01<00:00, 46345.10it/s]

TP: 17420, FP: 38600, Precision: 0.3110
Total time: 134.86s, Speedup: 30130.17x

============================================================
SUMMARY
============================================================
 k  m  b  candidates  true_positives  false_positives  precision       time
speedup
12  2 24      919679           16871           902808   0.018344  86.872850
1835.305744
12  3 36     1392508           17878          1374630   0.012839 107.987889
1212.123845
12  4 48     1899388           18509          1880879   0.009745 116.679454
888.650529
12  5 60     2350590           18934          2331656   0.008055 133.766475
718.071697
14  2 28      251097           16359           234738   0.065150  83.207076
6722.072151
14  3 42      373531           17239           356292   0.046151  98.212092
4518.747175
14  4 56      497276           17870           479406   0.035936 108.163835
3394.276319
14  5 70      608925           18289           590636   0.030035 120.335509
2771.921256
16  2 32       75363           15729            59634   0.208710  85.541361
22396.828032
16  3 48      109531           16685            92846   0.152331  99.260442
15410.177493
16  4 64      139005           17311           121694   0.124535 105.665229
12142.672213
16  5 80      168077           17716           150361   0.105404 116.789513
10042.374334
18  2 36       30853           15336            15517   0.497067  85.638391
```

```
54707.553593
18  3 54       40276           16244           24032   0.403317   1.299474
41908.137625
18  4 72       48027           16824           31203   0.350303 110.128650
35144.650946
18  5 90       56020           17420           38600   0.310960 134.858917
30130.170493
```

En este ejercicio se aplica el método de Locality-Sensitive Hashing (LSH) sobre las representaciones Simhash de los documentos para detectar duplicados o casi duplicados. Cada simhash se divide en m fragmentos de k bits que actúan como identificadores de bucket en distintas tablas hash. Los documentos que comparten bucket en al menos una tabla se consideran candidatos a duplicado. Posteriormente, se calcula la similitud coseno real entre los vectores tf-idf para validar la coincidencia.

Este enfoque permite reducir drásticamente la complejidad del proceso de comparación completa, ya que solo se calcula la similitud entre pares que colisionan en al menos una tabla. Finalmente, se evalúa el balance entre número de tablas (m), tamaño de los fragmentos (k), número de verdaderos positivos y falsos positivos, así como el tiempo de ejecución. Un valor de k demasiado pequeño genera muchas colisiones irrelevantes, mientras que uno grande puede perder coincidencias reales.

Podemos ver que el parámetro k (bits por fragmento) tiene un efecto clave sobre la precisión y la cantidad de candidatos: valores bajos de k (como 12 o 14) generan muchos pares falsos positivos y baja precisión, mientras que valores altos (16 o 18) reducen drásticamente las colisiones y mejoran la precisión hasta un 50%. Por otro lado, el parámetro m (número de tablas hash) incrementa el número de verdaderos positivos al aumentar, mejorando el recall, pero a costa de una menor precisión y mayor tiempo de ejecución, ya que introduce más colisiones aleatorias. En conjunto, los mejores resultados se obtienen con k alto y m moderado, logrando un equilibrio entre precisión, cobertura y eficiencia.
