Cosine Similarity Benchmark

Eric Crockett

May 8, 2025

List of Corrections

1 Introduction

This document describes Shai Halevi's algorithm for cosine similarity. We have a dataset with N (key, value) records. Each key is an ℓ_2-normalized m-dimensional vector. Let A be the $m \times N$ matrix with all of these keys. We assume that each value fits in k slots.

There is a single secret query (row) vector v of dimension m, also ℓ_2-normalized.

There is a threshold $\tau < 1$, and a promise that no more than M records in the dataset are τ-close to v.

The goal is to fetch the values corresponding to the keys that are τ-close to v.

The server has encryptions of the key and value for each record. Shai assumes we are using a ring dimension of size 65K, so there are 32K slots (note that K is never a variable). The encrypted data is batched into blocks of size 32K records, where the keys for block i are encoded into the slots of m ciphertexts, and the values for block i are encoded into the slots of k additional ciphertexts. The number of blocks is $n = \lceil N/32\text{K} \rceil$.

The following list summarizes the notation:

- N the number of records
- m the number of slots per record key
- k the number of slots per record value
- $n := \lceil N/32\text{K} \rceil$, the number of ciphertexts required to hold N slots
- τ the dot product threshold. We return values for keys whose dot product with the query is $\geq \tau$.
- M the maximum number of keys above the dot product.
- C chunk size, used in the compaction step.
- $r := \lceil \log_{\frac{M \cdot C}{32\text{K}}}(2^{-32}) \rceil$, the maximum number of 1s in a slice of size $C \cdot n$, with probability 2^{-32}.
- $R := \lfloor 32\text{K}/C \rfloor$, the number of output ciphertexts

We require $C \cdot n \leq 32\text{K}$, and $r \cdot k \leq 32\text{K}$. Eric: Feel free to improve the variable names Eric!

The procedure proceeds as follows:

1. **Dot Products** Compute $u = v \cdot A$. We can isolate and duplicate each coefficient of v using a multiplication by a (public) mask, followed by 16 shifts and 15 additions. Then we multiply each coefficient ciphertext by a packed row of A, and add the products together (for each block). This process has depth 2. Eric: I suspect Shai had a different algorithm in mind since he said it may take up to depth 8 for this step. See DotProds in Algorithm 1. Eric!

2. **Thresholding** Compare the slots of u against the threshold τ. Shai suggests using a polynomial approximation for this step, consuming 6 or 7 levels.

3. **Compacting** The algorithm requires a promise that at most M dot products will be larger than τ, so the comparison vector has at most M 1s in it. The output is R ciphertexts, each with at most $r \cdot k$ non-zero left-justified slots. Roughly, the algorithm involves "slicing" the threshold vector and value matrix by taking C slots from each of the n ciphertexts (in the input, or in one row of the value matrix). We compact these slots into a single ciphertext, then compute a running sum. With high probability, the sum will be $\leq r$, by design. For $i \in [1, r]$, we do an equality test on the running sum and multiply by the threshold vector to obtain an indicator vector with a one in the position of the i^{th} one in the threshold vector. We can then extract the corresponding value coefficient from the value matrix, and compactly store it in an output ciphertext. See Algorithm 2 for details.

Algorithm 1 Encrypted vector/matrix multiplication. v is a single CT with data in first m non-zero slots. A is a matrix with m rows, where each row is composed of n ciphertexts. The j^{th} ciphertext in the i^{th} row is $A_{i,j}$.

```
procedure REPLICATE(x)                         ▷ x has a value in one slot and 0s elsewhere
                                               ▷ Output has this value in all 32K slots
    for i in range(15) do                      ▷ Hard-coded for 32K slots
        x ← x + (x ≫ 2^i)
    return x

procedure EXTRACTCOEFF(v, i)                   ▷ Return a CT with v_i in all slots
    mask ← Indicator(i)
    w ← mask · v_0
    return Replicate(w)

procedure DOTPRODS(v, A)
    prods← [0] ∗ n
    for j in range(m) do
        v_i ← ExtractCoeff(v, j)
        for i in range(n) do
            prods[i]← prods[i] + v_i · A_{i,j}
    return prods
```

Algorithm 2 Compaction. us is the output of the thresholding step, a vector with n ciphertexts. ps is a $k \times n$ matrix where each row ps_i is n packed ciphertexts. $ps_{i,j}$ means the j^{th} ciphertext in the i^{th} row

```
procedure RUNNINGSUM(v)
    rs ← v · Indicator(0)
    acc ← v
    for i in range(1, 32K) do                  ▷ Extracts the i^th component of the running sum
        acc ← v + (acc ≫ 1)
        rs ← rs + (acc · Indicator(i))
    return rs

procedure EXTRACTSLICE(xs, t)                  ▷ xs is a list of n CTs, t ∈ [0, R)
    mask ← [1]*C+[0]*(32768 − C)
    x ← 0
    for i in range(n) do
        x ← x + (((xs_i · (mask ≫ C · t)) ≪ C · t) ≫ C · i)
    return x

procedure COMPACTIFY(us, ps)
    accs ← [0]*R
    for t in range(R) do                       ▷ Loop over each slice
        u ← ExtractSlice(us, t)
        v ← RunningSum(u)
        xs ←[0]*r
        for i in range(1, r + 1) do
            w_i ← map((== i), v)               ▷ 1 where the running-sum is i, 0 elsewhere
            xs_i ← w_i · u                     ▷ Extracts the i^th 1 from u
        for i in range(k) do                   ▷ k is the number of slots per value
            p ← ExtractSlice(ps_i, t)
            for j in range(1, r + 1) do        ▷ We assume that there are at most r 1s in u
                x ← p · xs_j                   ▷ The value corresponding to the j^th 1 from u
                y ← Replicate(x)               ▷ The value is now in all slots
                z ← y · Indicator(j · k + i − 1) ▷ The value is now in the appropriate output slot
                accs[t] ← accs[t]+z
    return accs
```

2 Failure Modes

2.1 Broken Promise

If there are in fact $> M$ matches, this does not necessarily invalidate the results. The effect is more indirect in that it increases the probability that more than r keys match in a given slice.

2.2 Unlucky Distribution

The algorithm returns up to r results per slice, and therefore up to $r \cdot R$ results total. If more than r keys match for a given slice (either because we got *very* unlucky, or because there are more than M matches and we are only slightly unlucky), only up to r results are returned per slice.

2.3 High-Precision Threshold

One problem that Shai identified with the thresholding step is that the polynomial approximation will return values ≈ 1 if the dot product is larger than τ and ≈ 0 if the dot product is smaller than τ. However, for values very close to τ, the polynomial approximation will return values near $1/2$, which results in inaccurate results. One way to prevent this is to require a "promise" that the dot products are sufficiently far away from τ. Eric: Is it possible to use scheme-switching to do a bit-by-bit comparison?

Eric!