

# Developer Guide

## Document Similarity Analyzer - Panduan untuk Presentasi

### Daftar Isi

1. [Tech Stack & Fungsinya](#)
2. [Algoritma: TF-IDF dan Cosine Similarity](#)
3. [Contoh Perhitungan Manual](#)
4. [Functional Programming yang Diterapkan](#)
5. [Kode Utama & Penjelasan](#)
6. [FAQ - Pertanyaan yang Mungkin Ditanya](#)

## 1. Tech Stack & Fungsinya

Berikut adalah teknologi yang digunakan dalam project ini beserta fungsi masing-masing:

Teknologi	Versi	Fungsi
Rust	2021 Edition	Bahasa pemrograman utama. Dipilih karena cepat, aman (memory safe), dan mendukung functional programming.
Axum	0.7	Framework web untuk membuat REST API. Menangani HTTP request/response.

<b>Tokio</b>	1.0	Async runtime. Memungkinkan server menangani banyak request secara bersamaan tanpa blocking.
<b>Rayon</b>	1.8	Library untuk parallel processing. Mempercepat perhitungan dengan memanfaatkan semua CPU core.
<b>Serde</b>	1.0	Serialization/Deserialization. Mengubah data Rust $\leftrightarrow$ JSON.
<b>thiserror</b>	1.0	Untuk membuat custom error types yang lebih mudah dibaca.

### **Kenapa Rust?**

Rust dipilih karena: (1) Performa setara C/C++, (2) Memory safety tanpa garbage collector, (3) Mendukung paradigma functional programming dengan baik.

## 2. Algoritma: TF-IDF dan Cosine Similarity

### **Tujuan Utama**

Mengukur seberapa mirip dua dokumen teks. Misalnya, apakah dokumen A dan dokumen B membahas topik yang sama?

### **Alur Kerja**



## Hubungan TF-IDF dan Cosine Similarity:

**TF-IDF** = Mengubah teks menjadi *vektor angka* (representasi matematis)

**Cosine Similarity** = Mengukur *kemiripan antar vektor*

Jadi: TF-IDF adalah *cara merepresentasikan* dokumen, sedangkan Cosine Similarity adalah *cara mengukur* kemiripannya.

## Term Frequency (TF)

TF mengukur **seberapa sering** sebuah kata muncul dalam satu dokumen.

$$TF(t, d) = \frac{\text{Jumlah kemunculan kata } t \text{ dalam dokumen } d}{\text{Total kata dalam dokumen } d}$$

**Contoh:** Dokumen "kucing suka kucing" memiliki 3 kata. Kata "kucing" muncul 2 kali.

$$TF(\text{"kucing"}) = 2/3 = 0.67$$

## Inverse Document Frequency (IDF)

IDF mengukur **seberapa unik** sebuah kata di seluruh dokumen. Kata yang muncul di mana-mana (seperti "dan", "yang") akan memiliki IDF rendah.

$$IDF(t) = \ln \left( \frac{N + 1}{df(t) + 1} \right) + 1$$

Dimana:

- **N** = Total jumlah dokumen
- **df(t)** = Jumlah dokumen yang mengandung kata t

- **+1** = Smoothing, agar tidak terjadi pembagian dengan nol

### **Intuisi IDF:**

Kata yang muncul di *semua* dokumen → IDF rendah (kata tidak penting)

Kata yang muncul di *sedikit* dokumen → IDF tinggi (kata unik/penting)

## **TF-IDF**

TF-IDF adalah hasil perkalian TF dan IDF:

$$\text{TF-IDF}(t, d) = \text{TF}(t, d) \times \text{IDF}(t)$$

Hasilnya adalah vektor angka yang merepresentasikan setiap dokumen.

## **Cosine Similarity**

Setelah dokumen menjadi vektor, kita hitung kemiripannya menggunakan Cosine Similarity:

$$\text{Cosine Similarity}(A, B) = \frac{A \cdot B}{\|A\| \times \|B\|}$$

Dimana:

- **A · B** = Dot product (perkalian elemen per elemen, lalu dijumlahkan)
- **$\|A\|$**  = Magnitude/panjang vektor A

### **Interpretasi Hasil:**

**1.0** = Dokumen identik (100% sama)

**0.0** = Dokumen tidak ada kesamaan sama sekali

**0.5** = Dokumen cukup mirip (50% kesamaan)

### 3. Contoh Perhitungan Manual

#### Contoh Sederhana

Misalkan kita punya 2 dokumen:

- **Dokumen 0:** "kucing suka ikan"
- **Dokumen 1:** "anjing suka tulang"

#### Langkah 1: Hitung TF

Kata	TF (Doc 0)	TF (Doc 1)
kucing	$1/3 = 0.33$	0
suka	$1/3 = 0.33$	$1/3 = 0.33$
ikan	$1/3 = 0.33$	0
anjing	0	$1/3 = 0.33$
tulang	0	$1/3 = 0.33$

#### Langkah 2: Hitung IDF

N = 2 dokumen

Kata	df (muncul di berapa dok)	$IDF = \ln((2+1)/(df+1)) + 1$
kucing	1	$\ln(3/2) + 1 = 1.41$
suka	2	$\ln(3/3) + 1 = 1.00$
ikan	1	$\ln(3/2) + 1 = 1.41$

anjing	1	$\ln(3/2) + 1 = 1.41$
tulang	1	$\ln(3/2) + 1 = 1.41$

Perhatikan: "suka" punya IDF paling rendah karena muncul di kedua dokumen!

### Langkah 3: Hitung TF-IDF (TF × IDF)

Kata	TF-IDF (Doc 0)	TF-IDF (Doc 1)
kucing	$0.33 \times 1.41 = 0.47$	0
suka	$0.33 \times 1.00 = 0.33$	$0.33 \times 1.00 = 0.33$
ikan	$0.33 \times 1.41 = 0.47$	0
anjing	0	$0.33 \times 1.41 = 0.47$
tulang	0	$0.33 \times 1.41 = 0.47$

### Vektor dokumen:

- Doc 0 = [0.47, 0.33, 0.47, 0, 0]
- Doc 1 = [0, 0.33, 0, 0.47, 0.47]

### Langkah 4: Cosine Similarity

$$\text{Dot Product} = (0.47 \times 0) + (0.33 \times 0.33) + (0.47 \times 0) + (0 \times 0.47) + (0 \times 0.47) = \mathbf{0.11}$$

$$\| \text{Doc 0} \| = \sqrt{(0.47^2 + 0.33^2 + 0.47^2)} = \sqrt{0.55} = \mathbf{0.74}$$

$$\| \text{Doc 1} \| = \sqrt{(0.33^2 + 0.47^2 + 0.47^2)} = \sqrt{0.55} = \mathbf{0.74}$$

$$\text{Cosine Similarity} = 0.11 / (0.74 \times 0.74) = \mathbf{0.20}$$

 **Kesimpulan:** Kedua dokumen memiliki kemiripan 20% karena hanya berbagi kata

"suka".

## 4. Functional Programming yang Diterapkan

Project ini menerapkan prinsip-prinsip Functional Programming (FP) berikut:

### 4.1 Pure Functions

**Definisi:** Fungsi yang outputnya hanya bergantung pada inputnya, tanpa efek samping (side effects).

```
// Pure function - output HANYA bergantung pada input
pub fn normalize_text(text: &str) -> String {
    text.chars()
        .map(|c| if c.is_ascii_punctuation() { ' ' } else { c.to_ascii_lowercase() })
        .collect()
}

// Jika input sama, output PASTI sama (tidak ada database, file, atau random)
```

#### ✓ Keuntungan Pure Functions:

- Mudah di-test (tidak perlu mock)
- Mudah dipahami (tidak ada hidden behavior)
- Thread-safe (aman untuk parallel processing)

### 4.2 Immutable Data

**Definisi:** Data yang tidak bisa diubah setelah dibuat. Jika butuh perubahan, buat data baru.

```
// Di Rust, variable default-nya immutable
let documents = vec!["doc1", "doc2"]; // tidak bisa diubah
```

```
// Untuk membuat mutable, harus eksplisit
let mut documents = vec!["doc1"]; // bisa diubah
```

## 4.3 Higher-Order Functions

**Definisi:** Fungsi yang menerima fungsi lain sebagai parameter, atau mengembalikan fungsi.

```
// .map() adalah higher-order function
// Menerima closure/fungsi sebagai parameter

let normalized: Vec<String> = documents
    .iter()
    .map(|doc| normalize_text(doc)) // map menerima fungsi
    .collect();

// Contoh lain: filter, fold, zip
```

## 4.4 Function Composition (Pipeline)

**Definisi:** Menggabungkan beberapa fungsi kecil menjadi satu alur proses.

```
// Pipeline: output fungsi sebelumnya menjadi input fungsi berikutnya

pub fn analyze_documents(documents: &[String]) -> SimilarityMatrix {
    // Stage 1: Normalize
    let normalized = documents.par_iter().map(|d| normalize_text(d)).collect();

    // Stage 2: Tokenize (input dari stage 1)
    let tokenized = normalized.par_iter().map(|d| tokenize(d)).collect();

    // Stage 3: Compute TF (input dari stage 2)
    let tfs = tokenized.par_iter().map(|t| compute_tf(t)).collect();

    // ... dan seterusnya
}
```

### Pipeline Flow:



## 5. Kode Utama & Penjelasan

### 5.1 Parallel Processing dengan Rayon

**Rayon** adalah library yang membuat parallel processing sangat mudah di Rust. Cukup ganti `.iter()` menjadi `.par_iter()`!

```
use rayon::prelude::*;

// SEQUENTIAL (satu per satu)
let results: Vec<_> = documents
    .iter()                      // proses satu-satu
    .map(|doc| normalize_text(doc))
    .collect();

// PARALLEL (semua CPU core dipakai)
let results: Vec<_> = documents
    .par_iter()                  // proses parallel!
    .map(|doc| normalize_text(doc))
    .collect();
```

#### 💡 Kenapa bisa parallel?

Karena fungsi kita *pure* (tidak ada side effects), setiap dokumen bisa diproses secara independen tanpa khawatir race condition.

### 5.2 Pipeline Utama (analyze\_documents)

Ini adalah fungsi utama yang mengorkestrasi seluruh proses:

```

pub fn analyze_documents(documents: &[String]) -> SimilarityMatrix {
    // Stage 1: Normalize semua dokumen (PARALLEL)
    let normalized: Vec<String> = documents
        .par_iter()
        .map(|doc| normalize_text(doc))
        .collect();

    // Stage 2: Tokenize (PARALLEL)
    let tokenized: Vec<Vec<String>> = normalized
        .par_iter()
        .map(|doc| tokenize(doc))
        .collect();

    // Stage 3: Hitung TF per dokumen (PARALLEL)
    let tfs: Vec<_> = tokenized
        .par_iter()
        .map(|tokens| compute_tf(tokens))
        .collect();

    // Stage 4: Hitung IDF global (SEQUENTIAL - butuh semua TF)
    let idf = compute_idf(&tfs);

    // Stage 5: Build vocabulary
    let vocabulary: Vec<String> = idf.keys().cloned().collect();

    // Stage 6: Vectorize (PARALLEL)
    let vectors: Vec<Vec<f32>> = tfs
        .par_iter()
        .map(|tf| vectorize(tf, &idf, &vocabulary))
        .collect();

    // Stage 7: Hitung similarity matrix (PARALLEL)
    let matrix = compute_similarity_matrix(&vectors);

    SimilarityMatrix::new(matrix, labels)
}

```

### 5.3 Cosine Similarity dalam Kode

```

pub fn cosine_similarity(vec_a: &[f32], vec_b: &[f32]) -> f32 {
    // Hitung dot product: A · B
    let dot_product: f32 = vec_a

```

```

    .iter()
    .zip(vec_b.iter())           // pasangkan elemen
    .map(|(a, b)| a * b)        // kalikan tiap pasangan
    .sum();                      // jumlahkan semua

    // Hitung magnitude (panjang vektor)
    let magnitude_a: f32 = vec_a.iter().map(|x| x * x).sum::<f32>().sqrt();
    let magnitude_b: f32 = vec_b.iter().map(|x| x * x).sum::<f32>().sqrt();

    // Cosine = dot product / (magnitude_a * magnitude_b)
    dot_product / (magnitude_a * magnitude_b)
}

```

## 5.4 Similarity Matrix (Parallel)

```

pub fn compute_similarity_matrix(vectors: &[Vec<f32>]) -> Vec<Vec<f32>> {
    let n = vectors.len();

    // Setiap baris dihitung secara PARALLEL
    (0..n)
        .into_par_iter()           // parallel iterator
        .map(|i| {
            (0..n)
                .map(|j| {
                    if i == j {
                        1.0 // diagonal selalu 1.0 (dokumen sama dengan dirinya)
                    } else {
                        cosine_similarity(&vectors[i], &vectors[j])
                    }
                })
                .collect()
        })
        .collect()
}

```

## 6. FAQ - Pertanyaan yang Mungkin Ditanya

### Q: Apa hubungan TF-IDF dengan Cosine Similarity?

**A:** TF-IDF mengubah teks menjadi vektor angka (representasi matematis). Cosine Similarity mengukur kemiripan antar vektor tersebut. Jadi TF-IDF adalah *cara merepresentasikan*, Cosine Similarity adalah *cara mengukur*.

## Q: Kenapa pakai Cosine, bukan Euclidean Distance?

**A:** Cosine mengukur *arah* vektor, bukan *panjang*. Dokumen panjang dan pendek yang membahas topik sama akan memiliki arah vektor yang sama, meskipun panjangnya berbeda. Euclidean akan menganggap mereka berbeda karena panjang vektornya beda.

## Q: Kenapa IDF pakai logaritma?

**A:** Untuk meredam perbedaan yang terlalu ekstrem. Jika tanpa log, kata yang muncul di 1 dokumen dari 1000 akan punya nilai 1000x lebih tinggi dari kata yang muncul di semua dokumen. Dengan log, perbedaannya lebih moderat.

## Q: Kenapa butuh Rayon/parallel processing?

**A:** Untuk performa. Jika ada 100 dokumen, perhitungan similarity matrix butuh  $100 \times 100 = 10,000$  operasi. Dengan parallel processing, kita bisa memanfaatkan semua CPU core sehingga lebih cepat.

## Q: Apa itu Pure Function dan kenapa penting?

**A:** Pure function adalah fungsi yang outputnya hanya bergantung pada inputnya, tanpa efek samping. Penting karena: (1) mudah di-test, (2) thread-safe untuk parallel processing, (3) mudah dipahami dan di-debug.

## Q: Bagaimana cara kerja .par\_iter()?

**A:** Rayon otomatis membagi pekerjaan ke semua CPU core. Misalnya jika ada 8 core dan 100 dokumen, setiap core akan memproses sekitar 12-13 dokumen secara bersamaan. Programmer tidak perlu mengatur thread secara manual.