# Document Similarity Analyzer

Source Code Listing - Rust Implementation

## Table of Contents



## Entry Points

### 1. src/main.rs

```rust
//! Document Similarity Analyzer - Main Entry Point
//!
//! A backend service for analyzing document similarity using TF-IDF
//! and Cosine Similarity with parallel processing.

use tracing_subscriber::{layer::SubscriberExt, util::SubscriberInitExt};

use document_similarity_analyzer::api::run_server;

#[tokio::main]
async fn main() -> anyhow::Result<()> {
    // Initialize tracing/logging
    tracing_subscriber::registry()
        .with(
            tracing_subscriber::EnvFilter::try_from_default_env()
                .unwrap_or_else(|_| "document_similarity_analyzer=debug,tower_http=debug".into()),
        )
        .with(tracing_subscriber::fmt::layer())
        .init();

    // Get port from environment or use default
    let port: u16 = std::env::var("PORT")
        .ok()
        .and_then(|p| p.parse().ok())
        .unwrap_or(3000);

    // Run the server
    run_server(port).await
}
```

### 2. src/lib.rs

```rust
//! Document Similarity Analyzer
//!
//! A backend service for analyzing document similarity using TF-IDF and Cosine Similarity
//! with parallel processing powered by Rayon.
//!
//! ## Architecture
//! - `api` - HTTP API handlers and server configuration
//! - `core` - Pure functions for text processing and similarity computation
//! - `models` - Immutable data structures

pub mod api;
pub mod core;
pub mod models;
```

## API Module

### 3. src/api/mod.rs

```rust
//! HTTP API module

mod error;
pub mod handlers;
mod server;

pub use error::AppError;
pub use handlers::{analyze_handler, health_handler, validate_request};
pub use server::{create_router, run_server};


```

### 4. src/api/server.rs

```rust
//! HTTP server configuration

use axum::{
    routing::{get, post},
    Router,
};
use std::net::SocketAddr;
use tower_http::cors::{Any, CorsLayer};
use tracing::info;

use super::handlers::{analyze_handler, health_handler};

/// Creates the Axum router with all routes configured
pub fn create_router() -> Router {
    // Configure CORS
    let cors = CorsLayer::new()
        .allow_origin(Any)
        .allow_methods(Any)
        .allow_headers(Any);

    Router::new()
        .route("/health", get(health_handler))
        .route("/analyze", post(analyze_handler))
        .layer(cors)
}

/// Runs the HTTP server
pub async fn run_server(port: u16) -> anyhow::Result<()> {
    let app = create_router();
    let addr = SocketAddr::from(([0, 0, 0, 0], port));

    info!("🚀 Server starting on http://{}", addr);
    info!("📊 POST /analyze - Analyze document similarity");
    info!("❤️  GET /health  - Health check");

    let listener = tokio::net::TcpListener::bind(addr).await?;
    axum::serve(listener, app).await?;

    Ok(())
}


```

## 5. src/api/handlers.rs

```rust
//! API request handlers

use axum::Json;

use crate::core::analyze_documents;
use crate::models::{AnalyzeRequest, AnalyzeResponse};
use super::AppError;

/// Maximum number of documents allowed per request
const MAX_DOCUMENTS: usize = 100;

/// Minimum number of documents required for comparison
const MIN_DOCUMENTS: usize = 2;

/// Maximum length of a single document in characters
const MAX_DOCUMENT_LENGTH: usize = 50_000;

/// Handler for POST /analyze endpoint
///
/// Receives documents and returns their similarity matrix.
pub async fn analyze_handler(
    Json(payload): Json<AnalyzeRequest>,
) -> Result<Json<AnalyzeResponse>, AppError> {
    // Validate input
    validate_request(&payload)?;

    // Process documents through the pipeline
    let result = analyze_documents(&payload.documents);

    // Convert to response format
    Ok(Json(AnalyzeResponse::from(result)))
}

/// Validates the analyze request
pub fn validate_request(request: &AnalyzeRequest) -> Result<(), AppError> {
    // Check if documents array is empty
    if request.documents.is_empty() {
        return Err(AppError::NoDocuments);
    }

    // Check minimum documents
    if request.documents.len() < MIN_DOCUMENTS {
        return Err(AppError::NotEnoughDocuments(request.documents.len()));
    }

    // Check maximum documents
    if request.documents.len() > MAX_DOCUMENTS {
        return Err(AppError::TooManyDocuments(request.documents.len()));
    }

    // Check each document
    for (i, doc) in request.documents.iter().enumerate() {
        if doc.trim().is_empty() {
            return Err(AppError::EmptyDocument(i));
        }
        if doc.len() > MAX_DOCUMENT_LENGTH {
            return Err(AppError::DocumentTooLong(i, MAX_DOCUMENT_LENGTH));
        }
    }

    Ok(())
}
```

```rust

/// Handler for GET /health endpoint
pub async fn health_handler() -> &'static str {
    "OK"
}



```

## 6. src/api/error.rs

```rust
//! Application error types

use axum::{
    http::StatusCode,
    response::{IntoResponse, Response},
    Json,
};
use serde::Serialize;
use thiserror::Error;

/// Application error types
#[derive(Debug, Error)]
pub enum AppError {
    #[error("Too many documents: {0}, maximum allowed is 100")]
    TooManyDocuments(usize),

    #[error("Empty document at index {0}")]
    EmptyDocument(usize),

    #[error("No documents provided")]
    NoDocuments,

    #[error("Not enough documents: minimum 2 required for comparison, got {0}")]
    NotEnoughDocuments(usize),

    #[error("Document at index {0} exceeds maximum length of {1} characters")]
    DocumentTooLong(usize, usize),

    #[error("Internal server error: {0}")]
    Internal(#[from] anyhow::Error),
}

/// Error response body
#[derive(Debug, Serialize)]
struct ErrorResponse {
    error: String,
    code: String,
}

impl IntoResponse for AppError {
    fn into_response(self) -> Response {
        let (status, code) = match &self {
            AppError::TooManyDocuments(_) => (StatusCode::BAD_REQUEST, "TOO_MANY_DOCUMENTS"),
            AppError::EmptyDocument(_) => (StatusCode::BAD_REQUEST, "EMPTY_DOCUMENT"),
            AppError::NoDocuments => (StatusCode::BAD_REQUEST, "NO_DOCUMENTS"),
            AppError::NotEnoughDocuments(_) => (StatusCode::BAD_REQUEST, "NOT_ENOUGH_DOCUMENTS"),
            AppError::DocumentTooLong(_, _) => (StatusCode::BAD_REQUEST, "DOCUMENT_TOO_LONG"),
            AppError::Internal(_) => (StatusCode::INTERNAL_SERVER_ERROR, "INTERNAL_ERROR"),
        };

        let body = ErrorResponse {
            error: self.to_string(),
            code: code.to_string(),
        };

        (status, Json(body)).into_response()
    }
}

```

# Core Module

### 7. src/core/mod.rs

```rust
//! Core processing functions - all pure functions with no side effects

mod normalize;
mod tokenize;
mod tf;
mod idf;
mod vectorize;
mod similarity;
mod matrix;
mod pipeline;

pub use normalize::normalize_text;
pub use tokenize::tokenize;
pub use tf::compute_tf;
pub use idf::compute_idf;
pub use vectorize::vectorize;
pub use similarity::cosine_similarity;
pub use matrix::compute_similarity_matrix;
pub use pipeline::analyze_documents;

```

## 8. src/core/normalize.rs

```rust
//! Text normalization - pure function

/// Normalizes text by converting to lowercase, removing punctuation,
/// and collapsing multiple whitespace into single space.
///
/// # Arguments
/// * `text` - The input text to normalize
///
/// # Returns
/// A new String with normalized text
///
/// # Example
/// ```
/// use document_similarity_analyzer::core::normalize_text;
///
/// let result = normalize_text("Hello, World!");
/// assert_eq!(result, "hello world");
/// ```
pub fn normalize_text(text: &str) -> String {
    text.chars()
        .map(|c| {
            if c.is_ascii_punctuation() {
                ' '
            } else {
                c.to_ascii_lowercase()
            }
        })
        .collect::<String>()
        .split_whitespace()
        .collect::<Vec<&str>>()
        .join(" ")
}

```

## 9. src/core/tokenize.rs

```rust
//! Tokenization - pure function

/// Tokenizes text into a vector of words by splitting on whitespace.
///
/// # Arguments
/// * `text` - The input text to tokenize (should be pre-normalized)
///
/// # Returns
/// A Vec<String> containing individual tokens
///
/// # Example
/// ```
/// use document_similarity_analyzer::core::tokenize;
///
/// let tokens = tokenize("hello world");
/// assert_eq!(tokens, vec!["hello", "world"]);
/// ```
pub fn tokenize(text: &str) -> Vec<String> {
    text.split_whitespace()
        .filter(|s| !s.is_empty())
        .map(|s| s.to_string())
        .collect()
}
```

## 10. src/core/tf.rs

```rust
//! Term Frequency calculation - pure function

use std::collections::HashMap;

/// Computes Term Frequency (TF) for a list of tokens.
/// TF = (number of times term appears) / (total number of terms)
///
/// # Arguments
/// * `tokens` - Slice of tokens from a document
///
/// # Returns
/// A HashMap mapping each term to its frequency (0.0 to 1.0)
///
/// # Example
/// ```
/// use document_similarity_analyzer::core::compute_tf;
///
/// let tokens = vec!["a".to_string(), "b".to_string(), "a".to_string()];
/// let tf = compute_tf(&tokens);
/// assert!((tf["a"] - 0.667).abs() < 0.01);
/// assert!((tf["b"] - 0.333).abs() < 0.01);
/// ```
pub fn compute_tf(tokens: &[String]) -> HashMap<String, f32> {
    if tokens.is_empty() {
        return HashMap::new();
    }

    let total = tokens.len() as f32;
    let mut counts: HashMap<String, usize> = HashMap::new();

    for token in tokens {
        *counts.entry(token.clone()).or_insert(0) += 1;
    }

    counts
        .into_iter()
        .map(|(term, count)| (term, count as f32 / total))
        .collect()
}


```

## 11. src/core/idf.rs

```rust
//! Inverse Document Frequency calculation - pure function

use std::collections::{HashMap, HashSet};

/// Computes Inverse Document Frequency (IDF) across all documents.
/// Uses smoothed IDF: IDF = log((N + 1) / (df + 1)) + 1
/// This prevents division by zero and ensures non-zero IDF values.
///
/// # Arguments
/// * `tfs` - Slice of Term Frequency maps, one per document
///
/// # Returns
/// A HashMap mapping each term to its IDF value
///
/// # Example
/// ```
/// use document_similarity_analyzer::core::compute_idf;
/// use std::collections::HashMap;
///
/// let tf1: HashMap<String, f32> = [("hello".to_string(), 0.5)].into_iter().collect();
/// let tf2: HashMap<String, f32> = [("world".to_string(), 0.5)].into_iter().collect();
/// let idf = compute_idf(&[tf1, tf2]);
/// ```
pub fn compute_idf(tfs: &[HashMap<String, f32>]) -> HashMap<String, f32> {
    if tfs.is_empty() {
        return HashMap::new();
    }

    let n = tfs.len() as f32;
    let mut document_frequency: HashMap<String, usize> = HashMap::new();

    // Count in how many documents each term appears
    for tf in tfs {
        let terms: HashSet<&String> = tf.keys().collect();
        for term in terms {
            *document_frequency.entry(term.clone()).or_insert(0) += 1;
        }
    }

    // Calculate smoothed IDF: log((N + 1) / (df + 1)) + 1
    document_frequency
        .into_iter()
        .map(|(term, df)| {
            let idf = ((n + 1.0) / (df as f32 + 1.0)).ln() + 1.0;
            (term, idf)
        })
        .collect()
}

```

## 12. src/core/vectorize.rs

```rust
//! TF-IDF Vectorization - pure function

use std::collections::HashMap;

/// Converts TF and IDF into a TF-IDF vector based on vocabulary order.
///
/// # Arguments
/// * `tf` - Term Frequency map for a document
/// * `idf` - Inverse Document Frequency map (global)
/// * `vocabulary` - Ordered list of all terms (defines vector dimensions)
///
/// # Returns
/// A Vec<f32> representing the document's TF-IDF vector
///
/// # Example
/// ```
/// use document_similarity_analyzer::core::vectorize;
/// use std::collections::HashMap;
///
/// let tf: HashMap<String, f32> = [("hello".to_string(), 0.5)].into_iter().collect();
/// let idf: HashMap<String, f32> = [("hello".to_string(), 0.693)].into_iter().collect();
/// let vocab = vec!["hello".to_string()];
/// let vector = vectorize(&tf, &idf, &vocab);
/// ```
pub fn vectorize(
    tf: &HashMap<String, f32>,
    idf: &HashMap<String, f32>,
    vocabulary: &[String],
) -> Vec<f32> {
    vocabulary
        .iter()
        .map(|term| {
            let tf_value = tf.get(term).copied().unwrap_or(0.0);
            let idf_value = idf.get(term).copied().unwrap_or(0.0);
            tf_value * idf_value
        })
        .collect()
}
```

## 13. src/core/similarity.rs

```rust
//! Cosine Similarity calculation - pure function

/// Computes cosine similarity between two vectors.
/// Formula: (A · B) / (||A|| * ||B||)
///
/// # Arguments
/// * `vec_a` - First vector
/// * `vec_b` - Second vector
///
/// # Returns
/// Cosine similarity value between -1.0 and 1.0
/// Returns 0.0 if either vector has zero magnitude
///
/// # Example
/// ```
/// use document_similarity_analyzer::core::cosine_similarity;
///
/// let a = vec![1.0, 0.0];
/// let b = vec![1.0, 0.0];
/// assert_eq!(cosine_similarity(&a, &b), 1.0);
/// ```
pub fn cosine_similarity(vec_a: &[f32], vec_b: &[f32]) -> f32 {
    if vec_a.len() != vec_b.len() || vec_a.is_empty() {
        return 0.0;
    }

    // Compute dot product: A · B
    let dot_product: f32 = vec_a
        .iter()
        .zip(vec_b.iter())
        .map(|(a, b)| a * b)
        .sum();

    // Compute magnitudes: ||A|| and ||B||
    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();

    // Handle zero magnitude case
    if magnitude_a == 0.0 || magnitude_b == 0.0 {
        return 0.0;
    }

    dot_product / (magnitude_a * magnitude_b)
}


```

## 14. src/core/matrix.rs

```rust
//! Similarity Matrix generation - parallel computation

use rayon::prelude::*;
use super::cosine_similarity;

/// Computes a full NxN similarity matrix from TF-IDF vectors.
/// Utilizes parallel processing for efficiency.
///
/// # Arguments
/// * `vectors` - Slice of TF-IDF vectors, one per document
///
/// # Returns
/// NxN matrix where matrix[i][j] = cosine_similarity(vectors[i], vectors[j])
///
/// # Properties
/// - Diagonal is always 1.0 (document is identical to itself)
/// - Matrix is symmetric: matrix[i][j] == matrix[j][i]
pub fn compute_similarity_matrix(vectors: &[Vec<f32>]) -> Vec<Vec<f32>> {
    let n = vectors.len();

    if n == 0 {
        return vec![];
    }

    // Parallel computation of similarity matrix
    (0..n)
        .into_par_iter()
        .map(|i| {
            (0..n)
                .map(|j| {
                    if i == j {
                        1.0 // Diagonal is always 1.0
                    } else if i < j {
                        // Compute similarity for upper triangle
                        cosine_similarity(&vectors[i], &vectors[j])
                    } else {
                        // For lower triangle, compute it too
                        cosine_similarity(&vectors[i], &vectors[j])
                    }
                })
                .collect()
        })
        .collect()
}
```

## 15. src/core/pipeline.rs

```rust
//! Document analysis pipeline - parallel processing orchestration

use rayon::prelude::*;
use std::sync::Arc;

use crate::models::SimilarityMatrix;
use super::{
    normalize_text,
    tokenize,
    compute_tf,
    compute_idf,
    vectorize,
    compute_similarity_matrix
};

/// Analyzes multiple documents and computes their similarity matrix.
/// Uses parallel processing for all possible stages.
///
/// # Pipeline
/// 1. Normalize all documents (parallel)
/// 2. Tokenize all documents (parallel)
/// 3. Compute TF for each document (parallel)
/// 4. Compute IDF globally (single-threaded, needs all TFs)
/// 5. Build vocabulary from IDF
/// 6. Vectorize each document (parallel)
/// 7. Compute similarity matrix (parallel)
///
/// # Arguments
/// * `documents` - Slice of document strings to analyze
///
/// # Returns
/// A SimilarityMatrix containing the NxN similarity values and document labels
pub fn analyze_documents(documents: &[String]) -> SimilarityMatrix {
    if documents.is_empty() {
        return SimilarityMatrix::new(vec![], vec![]);
    }

    // Generate document labels
    let labels: Vec<String> = (0..documents.len())
        .map(|i| format!("doc{}", i))
        .collect();

    // Stage 1: Normalize (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: Compute TF for each document (parallel)
    let tfs: Vec<_> = tokenized
        .par_iter()
        .map(|tokens| compute_tf(tokens))
        .collect();

    // Stage 4: Compute IDF (single-threaded, needs all TFs)
    let idf = compute_idf(&tfs);
```

```rust

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

    // Share vocabulary and IDF across threads
    let vocab_arc = Arc::new(vocabulary);
    let idf_arc = Arc::new(idf);

    // Stage 6: Vectorize each document (parallel)
    let vectors: Vec<Vec<f32>> = tfs
        .par_iter()
        .map(|tf| vectorize(tf, &idf_arc, &vocab_arc))
        .collect();

    // Stage 7: Compute similarity matrix (parallel)
    let matrix = compute_similarity_matrix(&vectors);

    SimilarityMatrix::new(matrix, labels)
}
```

## Models Module

### 16. src/models/mod.rs

```rust
//! Immutable data models for document similarity analysis

mod document;
mod request;
mod response;

pub use document::*;
pub use request::*;
pub use response::*;

```

## 17. src/models/document.rs

```rust
//! Document and processing data structures

use serde::{Deserialize, Serialize};
use std::collections::HashMap;

/// Represents a raw document with its ID
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct Document {
    pub id: String,
    pub content: String,
}

impl Document {
    pub fn new(id: impl Into<String>, content: impl Into<String>) -> Self {
        Self {
            id: id.into(),
            content: content.into(),
        }
    }
}

/// Represents a tokenized document
#[derive(Debug, Clone)]
pub struct TokenizedDoc {
    pub id: String,
    pub tokens: Vec<String>,
}

impl TokenizedDoc {
    pub fn new(id: impl Into<String>, tokens: Vec<String>) -> Self {
        Self {
            id: id.into(),
            tokens,
        }
    }
}

/// Term Frequency map for a single document
pub type TermFrequency = HashMap<String, f32>;

/// Inverse Document Frequency map across all documents
pub type InverseDocumentFrequency = HashMap<String, f32>;

/// TF-IDF Vector representation of a document
#[derive(Debug, Clone)]
pub struct TfIdfVector {
    pub id: String,
    pub vector: Vec<f32>,
}

impl TfIdfVector {
    pub fn new(id: impl Into<String>, vector: Vec<f32>) -> Self {
        Self {
            id: id.into(),
            vector,
        }
    }
}

/// Similarity matrix result
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct SimilarityMatrix {
```

```rust
    /// NxN similarity matrix where matrix[i][j] is similarity between doc i and doc j
    pub matrix: Vec<Vec<f32>>,
    /// Document indices/labels
    pub index: Vec<String>,
}

impl SimilarityMatrix {
    pub fn new(matrix: Vec<Vec<f32>>, index: Vec<String>) -> Self {
        Self { matrix, index }
    }
}

```

## 18. src/models/request.rs

```rust
//! API request models

use serde::{Deserialize, Serialize};

/// Request payload for document analysis
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct AnalyzeRequest {
    /// List of document texts to analyze
    pub documents: Vec<String>,
}

impl AnalyzeRequest {
    pub fn new(documents: Vec<String>) -> Self {
        Self { documents }
    }
}

```

### 19. src/models/response.rs

```rust
//! API response models

use serde::{Deserialize, Serialize};

/// Response payload for document analysis
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct AnalyzeResponse {
    /// NxN similarity matrix
    pub similarity_matrix: Vec<Vec<f32>>,
    /// Document indices/labels
    pub index: Vec<String>,
}

impl AnalyzeResponse {
    pub fn new(similarity_matrix: Vec<Vec<f32>>, index: Vec<String>) -> Self {
        Self {
            similarity_matrix,
            index,
        }
    }
}

impl From<crate::models::SimilarityMatrix> for AnalyzeResponse {
    fn from(matrix: crate::models::SimilarityMatrix) -> Self {
        Self {
            similarity_matrix: matrix.matrix,
            index: matrix.index,
        }
    }
}


```

## Tests

## 20. tests/integration.rs

```rust
//! Integration tests for Document Similarity Analyzer
//!
//! These tests verify the full pipeline and API endpoints work correctly.

use document_similarity_analyzer::core::analyze_documents;
use document_similarity_analyzer::models::AnalyzeRequest;
use document_similarity_analyzer::api::validate_request;

/// Test full pipeline with known documents
#[test]
fn test_pipeline_with_known_documents() {
    let documents = vec![
        "The quick brown fox jumps over the lazy dog".to_string(),
        "A quick brown dog outpaces a lazy fox".to_string(),
        "Hello world this is a test".to_string(),
    ];

    let result = analyze_documents(&documents);

    // Verify matrix dimensions
    assert_eq!(result.matrix.len(), 3);
    assert_eq!(result.matrix[0].len(), 3);

    // Verify diagonal is 1.0
    for i in 0..3 {
        assert!(
            (result.matrix[i][i] - 1.0).abs() < 0.001,
            "Diagonal element [{i}][{i}] should be 1.0"
        );
    }

    // Verify symmetry
    for i in 0..3 {
        for j in 0..3 {
            assert!(
                (result.matrix[i][j] - result.matrix[j][i]).abs() < 0.001,
                "Matrix should be symmetric at [{i}][{j}]"
            );
        }
    }

    // First two documents should have higher similarity than with the third
    let sim_0_1 = result.matrix[0][1];
    let sim_0_2 = result.matrix[0][2];
    let sim_1_2 = result.matrix[1][2];

    assert!(sim_0_1 > sim_0_2, "Documents 0 and 1 should be more similar than 0 and 2");
    assert!(sim_0_1 > sim_1_2, "Documents 0 and 1 should be more similar than 1 and 2");
}

/// Test deterministic output - same input should give same output
#[test]
fn test_deterministic_output() {
    let documents = vec![
        "machine learning is fascinating".to_string(),
        "deep learning neural networks".to_string(),
        "natural language processing".to_string(),
    ];

    let result1 = analyze_documents(&documents);
    let result2 = analyze_documents(&documents);

```

```rust
    assert_eq!(result1.matrix.len(), result2.matrix.len());

    for i in 0..result1.matrix.len() {
        for j in 0..result1.matrix[i].len() {
            assert!(
                (result1.matrix[i][j] - result2.matrix[i][j]).abs() < 0.0001,
                "Results should be deterministic at [{i}][{j}]"
            );
        }
    }
}

/// Test with identical documents
#[test]
fn test_identical_documents() {
    let documents = vec![
        "hello world".to_string(),
        "hello world".to_string(),
    ];

    let result = analyze_documents(&documents);

    for i in 0..2 {
        for j in 0..2 {
            assert!(
                (result.matrix[i][j] - 1.0).abs() < 0.001,
                "Identical documents should have similarity 1.0"
            );
        }
    }
}

/// Test with completely different documents
#[test]
fn test_completely_different_documents() {
    let documents = vec![
        "apple banana cherry".to_string(),
        "xyz uvw rst".to_string(),
    ];

    let result = analyze_documents(&documents);

    assert!(
        result.matrix[0][1].abs() < 0.001,
        "Documents with no common terms should have ~0 similarity"
    );
}

/// Test with many documents for parallel processing
#[test]
fn test_many_documents_parallel() {
    let documents: Vec<String> = (0..50)
        .map(|i| format!("document number {} with some content here", i))
        .collect();

    let result = analyze_documents(&documents);

    assert_eq!(result.matrix.len(), 50);
    assert_eq!(result.matrix[0].len(), 50);

    for i in 0..50 {
        for j in 0..50 {
            let sim = result.matrix[i][j];
            assert!(sim >= 0.0 && sim <= 1.0);
        }
```

```rust
    }
}

/// Test pipeline with special characters
#[test]
fn test_special_characters() {
    let documents = vec![
        "Hello, World! How are you?".to_string(),
        "HELLO WORLD HOW ARE YOU".to_string(),
    ];

    let result = analyze_documents(&documents);

    assert!(
        (result.matrix[0][1] - 1.0).abs() < 0.001,
        "Documents that normalize to same text should have similarity 1.0"
    );
}

/// Test index labels
#[test]
fn test_index_labels() {
    let documents = vec![
        "doc one".to_string(),
        "doc two".to_string(),
        "doc three".to_string(),
    ];

    let result = analyze_documents(&documents);

    assert_eq!(result.index, vec!["doc0", "doc1", "doc2"]);
}

#[cfg(test)]
mod api_tests {
    use document_similarity_analyzer::models::AnalyzeRequest;
    use document_similarity_analyzer::api::validate_request;

    #[test]
    fn test_validate_no_documents() {
        let request = AnalyzeRequest::new(vec![]);
        let result = validate_request(&request);
        assert!(result.is_err());
    }

    #[test]
    fn test_validate_single_document() {
        let request = AnalyzeRequest::new(vec!["only one".to_string()]);
        let result = validate_request(&request);
        assert!(result.is_err());
    }

    #[test]
    fn test_validate_empty_string() {
        let request = AnalyzeRequest::new(vec!["valid".to_string(), "".to_string()]);
        let result = validate_request(&request);
        assert!(result.is_err());
    }

    #[test]
    fn test_validate_too_many_documents() {
        let documents: Vec<String> = (0..101).map(|i| format!("doc {}", i)).collect();
        let request = AnalyzeRequest::new(documents);
        let result = validate_request(&request);
        assert!(result.is_err());
```

```rust
    }

    #[test]
    fn test_validate_valid_request() {
        let request = AnalyzeRequest::new(vec![
            "first doc".to_string(),
            "second doc".to_string()
        ]);
        let result = validate_request(&request);
        assert!(result.is_ok());
    }

    #[test]
    fn test_validate_exactly_100_documents() {
        let documents: Vec<String> = (0..100).map(|i| format!("doc {}", i)).collect();
        let request = AnalyzeRequest::new(documents);
        let result = validate_request(&request);
        assert!(result.is_ok());
    }
}
```

---

**End of Source Code Listing**

Document Similarity Analyzer - Rust Implementation

Total: 20 source files