Documentation du Projet en Rust

Votre Nom

16 janvier 2025

Introduction

Ce projet en Rust est une application permettant de manipuler et de rechercher des vecteurs dans une base de données. Il utilise des concepts comme les embeddings vectoriels, la similarité cosinus, et une base de données interne basée sur un `HashMap`. Le projet génère des embeddings aléatoires et recherche les vecteurs les plus similaires en termes de similarité cosinus.

Structure du Code

Le projet est divisé en plusieurs parties :

- Une fonction pour générer des embeddings aléatoires.
- Une fonction pour calculer la similarité cosinus entre deux vecteurs.
- Une structure `Db` encapsulant un `HashMap` pour stocker les vecteurs.
- Une méthode pour rechercher les vecteurs les plus similaires dans la base de données.
- Une fonction `main` pour orchestrer l'exécution.

Code Source

Fonction pour générer un embedding aléatoire

```
fn generer_embedding(dimension: usize) -> Vec<f32> {
    let mut rng = rand::thread_rng();
    (0..dimension).map(|_| rng.gen_range(0.0..1.0)).collect()
}
```

Cette fonction génère un vecteur de taille `dimension` avec des valeurs aléatoires comprises entre 0 et 1.

Fonction pour calculer la similarité cosinus

```rust
fn calcul_similarite(vec1: &[f32], vec2: &[f32]) -> f32 {
    let produit: f32 = vec1.iter().zip(vec2).map(|(a, b)| a * b).sum();
    let vec1: f32 = vec1.iter().map(|x| x * x).sum::<f32>().sqrt();
    let vec2: f32 = vec2.iter().map(|x| x * x).sum::<f32>().sqrt();

    if vec1 == 0.0 || vec2 == 0.0 {
        0.0
    } else {
        produit / (vec1 * vec2)
    }
}
```

Cette fonction calcule la similarité cosinus entre deux vecteurs en évitant la division par zéro.

Structure `Db` et ses méthodes

```rust
struct Db {
    inner: HashMap<Uuid, Vec<f32>>,
}

impl Db {
    fn new() -> Self {
        Db {
            inner: HashMap::new(),
        }
    }

    fn insert(&mut self, uuid: Uuid, embedding: Vec<f32>) {
        self.inner.insert(uuid, embedding);
    }

    fn trouver_les_plus_similaires(&self, query: &[f32], n: usize) -> Vec<(Uuid, f32)> {
        let mut similarities: Vec<(Uuid, f32)> = self
            .inner
            .iter()
            .map(|(uuid, embedding)| (*uuid, calcul_similarite(query, embedding)))
            .collect();

        similarities.sort_by(|a, b| b.1.partial_cmp(&a.1).unwrap_or(std::cmp::Ordering::Equa

        similarities.into_iter().take(n).collect()
    }
}
```

La structure `Db` contient une base de données interne sous forme de

`HashMap` et permet d'insérer des vecteurs et de trouver les vecteurs les plus similaires.

Fonction `main`

```rust
fn main() {
    let mut db = Db::new();
    let embedding_dimension = 768;

    for _ in 0..10 {
        let uuid = Uuid::new_v4();
        let embedding = generer_embedding(embedding_dimension);
        db.insert(uuid, embedding);
    }

    let query_embedding = generer_embedding(embedding_dimension);
    let plus_similaire = db.trouver_les_plus_similaires(&query_embedding, 3);

    println!("Les 3 vecteurs les plus similaires :");
    for (uuid, similarity) in plus_similaire {
        println!("UUID: {}, Similarité: {:.4}", uuid, similarity);
    }
}
```

La fonction `main` crée une base de données, y insère des vecteurs aléatoires, génère un vecteur de requête, et affiche les trois vecteurs les plus similaires.

Conclusion

Ce projet démontre l'utilisation efficace de Rust pour manipuler des données vectorielles, calculer des similarités, et effectuer des recherches performantes dans une base de données interne. Il peut être étendu pour intégrer des fonctionnalités supplémentaires comme la persistance des données ou l'optimisation des calculs.