FastRoLSH: Высокопроизводительная система приближенного поиска соседей Реализация и архитектура

Аннотация

Данный документ описывает практическую реализацию системы FastRoLSH, объединяющей алгоритмы FastLSH и roLSH для эффективного приближенного поиска ближайших соседей (ANN). В отличие от теоретического описания, акцент здесь сделан на архитектурных решениях, ключевых компонентах кода и их взаимодействии. Система реализована как монолитное серверное приложение на Python с использованием FastAPI, асинхронного доступа к PostgreSQL и GPU-ускорения через PyTorch. Документ отражает фактическую реализацию, включая механизмы индексирования, обработки запросов, семплирования и самооптимизации.

1 Введение

Реализация системы FastRoLSH представляет собой законченный инженерный продукт, готовый к развертыванию и использованию в production-среде. Ядро системы — класс `FastRoLSHsampler` — инкапсулирует всю логику работы алгоритмов. Система спроектирована для обработки потоковых данных, поступающих батчами, что позволяет непрерывно обновлять индекс по мере поступления новых данных. Для взаимодействия с системой реализован полнофункциональный REST API, позволяющий создавать датасеты, загружать данные, выполнять поиск и семплирование.

Архитектура системы сознательно выбрана как «монолит с состоянием в БД». Это упрощает развертывание и эксплуатацию по сравнению с распределенной системой, сохраняя при этом возможность горизонтального масштабирования на уровне API-серверов (за счет балансировщика нагрузки) и использования мощных серверных машин с GPU для вычислений.

2 Архитектура и компоненты системы

2.1 Общая схема работы

Система состоит из трех основных уровней:

1. **Уровень хранения данных (PostgreSQL):** Хранит метаинформацию о датасетах, батчах, а также сериализованное состояние LSH-таблиц. Это обеспечивает персистентность и возможность восстановления состояния сервиса после перезапуска.
2. **Вычислительное ядро (FastRoLSHsampler):** Живет в памяти приложения. Отвечает за инкрементальное обновление хеш-таблиц новыми данными, обработку поисковых запросов и семплирование. Для вычислений использует библиотеку PyTorch, что позволяет автоматически задействовать GPU.

3. **Уровень API (FastAPI):** Предоставляет внешний интерфейс для управления системой и выполнения операций. Запросы, связанные с добавлением данных, ставятся в фоновую очередь (`BackgroundTasks`), чтобы не блокировать отклик API.

2.2 Детали реализации

Класс `FastRoLSHsampler` инициализируется с параметрами, определяющими конфигурацию LSH (`d, m, k, L, w`), и опциональными параметрами для оптимизации roLSH (`initial_radiu` `radius_expansion`).

- **Инициализация хеш-функций:** Метод `_init_hash_functions` инициализирует проекционные векторы и смещения в зависимости от выбранной метрики расстояния. Для евклидовой метрики дополнительно создается тензор `self.indices` размерности `(L, k, m_sampled)` со случайными индексами признаков для реализации семплирования из FastLSH.

- **Обновление состояния (`update`):** Ключевой метод для инкрементального обучения. Для каждого вектора в батче вычисляется его хеш во всех L таблицах. Глобальный индекс точки рассчитывается как `self.total_points + i`. Индексы точек добавляются в соответствующие бакеты (словари `self.tables[l]`). Параллельно обновляется резервуарная выборка (`_update_reservoir_sample`) и статистика по радиусу (`_update_radius_stats`).

- **Поиск запросов (`batched_query`):** Реализует адаптивный поиск roLSH. Для каждого запроса поиск начинается с начального радиуса. На каждой итерации просматриваются бакеты, соответствующие хешу запроса. Если кандидатов недостаточно, радиус расширяется на коэффициент `radius_expansion`. Для евклидовой метрики это приводит к пересчету хеша с новым эффективным `w`. Для косинусной — к генерации соседних хешей путем переворачивания бит.

2.3 Взаимодействие с базой данных

Состояние системы постоянно синхронизируется с PostgreSQL.

- **Метаданные:** Таблицы `datasets` и `batches` хранят параметры и информацию о processed батчах.

- **Состояние LSH:** Таблица `lsh_tables` хранит сериализованные в JSON списки индексов точек для каждого бакета. Метод `_save_lsh_tables` реализует инкрементальное обновление: изменяет только те бакеты, которые были обновлены в памяти. Это критически важно для производительности при работе с большими индексами.

- **Восстановление состояния:** Метод `load_state_from_db` позволяет экземпляру `FastRoLSHsampler` полностью восстановить свое состояние из БД при перезапуске сервиса.

3 Методологические аспекты реализации

3.1 Реализация FastLSH

Принцип случайного семплирования признаков реализован полностью. Для каждой хеш-функции в каждой таблице заранее генерируется свой уникальный набор из `m_sampled =`

int(d * sampling_ratio) случайных индексов. При вычислении хеша вектора X операция X[:, inds] эффективно выбирает только нужные признаки перед скалярным произведением на проекционный вектор self.A[l, k$_i$$dx$].$O(d)$ до $O(m)$ на одну хеш-функцию.

3.2 Реализация roLSH

Адаптивный поиск реализован через итеративное расширение радиуса.

1. Инициализация радиуса: Если радиус не задан пользователем, он вычисляется автомат На основе первых батчей данных методом _update_radius_stats вычисляется среднее расстояние между точками, и начальный радиус устанавливается равным половине этого значения.

2. Механизм расширения: В методе batched_query для каждого запроса цикл выполняется до 5 раз. На каждом шаге радиус умножается на radius_expansion. Для евклидовой метрики новый хеш вычисляется в методе _get_expanded_hashes путем деления ширины бакета w на текущий радиус. Для косинусной метрики в методе _get_cosine_expanded_ генерируются хеши, отличающиеся на 1, 2, ... бита в зависимости от шага расширени

3.3 Семплирование и оптимизация

Система поддерживает стратегии стратифицированного семплирования (sample) на основе LSH-бакетов.

- Пропорциональная стратегия: Из каждого бакета отбирается количество точек, пропорциональное его размеру.

- Сбалансированная стратегия: Из каждого бакета отбирается примерно одинаковое количество точек.

Для обеих стратетий корректно рассчитываются веса для последующего взвешенного обучени компенсирующие смещение выборки.

Метод optimize_parameters реализует самонастройку системы. На основе резервуарной выборки данных строится эмпирическое распределение pairwise-расстояний. Определяется расстояние R (10-й перцентиль) и cR. Путем численного интегрирования функции _p_elast находится значение параметра w, которое минимизирует параметр $\rho = \frac{\log(1/p_1)}{\log(1/p_2)}$, что теоретич гарантирует оптимальное соотношение точности и полноты поиска.

4 Заключение

Представленная реализация системы FastRoLSH является полной, корректной и готовой к применению. Она успешно объединяет теоретические принципы FastLSH и roLSH, адаптируя их для работы в режиме онлайн-обучения на потоковых данных. Ключевыми инженерными преимуществами системы являются:

- Полная персистентность состояния в реляционной БД PostgreSQL.

- Инкрементальное обновление индекса и метаданных.

- Поддержка GPU-ускорения через PyTorch.

- Фоновая обработка данных и асинхронный API.

- Механизмы самооптимизации параметров на основе поступающих данных.