Tuning vector recall while generating query embeddings in the database

author: Montana Low

April 28, 2023

PostgresML makes it easy to generate embeddings using open source models and perform complex queries with vector indexes unlike any other database. The full expressive power of SQL as a query language is available to seamlessly combine semantic, geospatial, and full text search, along with filtering, boosting, aggregation, and ML reranking in low latency use cases. You can do all of this faster, simpler and with higher quality compared to applications built on disjoint APIs like OpenAI + Pinecone. Prove the results in this series to your own satisfaction, for free, by signing up for a GPU accelerated database.

Introduction

This article is the second in a multipart series that will show you how to build a post-modern semantic search and recommendation engine, including personalization, using open source models. The previous article discussed how to generate embeddings that perform better than OpenAI's `text-embedding-ada-002` and save them in a table with a vector index. In this article, we'll show you how to query those embeddings effectively.

Embeddings show us the relationships between rows in the database, using natural language.

Our example data is based on 5 million DVD reviews from Amazon customers submitted over a decade. For reference, that's more data than fits in a Pinecone Pod at the time of writing. Webscale: check. Let's start with a quick refresher on the data in our `pgml.amazon_us_reviews` table:

- Query
- Result

```
SELECT  *
FROM  pgml.amazon_us_reviews
LIMIT 5;
```

priority_highNote

You may have noticed it took more than 100ms to retrieve those 5 rows with their embeddings. Scroll the code block over to see how much numeric data there is. *Fetching an embedding over the wire takes about as long as generating it from scratch with a state-of-the-art model.* 🤯

Many benchmarks completely ignore the costs of data transfer and (de)serialization but in practice, it happens multiple times and becomes the largely dominant cost in typical complex systems.

Sorry, that was supposed to be a refresher, but it set me off. At PostgresML we're concerned about microseconds. 107.207 milliseconds better be spent doing something *really* useful, not just fetching 5 rows. Bear with me while I belabor this point, because it reveals the source of most latency in machine learning microservice architectures that separate the database from the model, or worse, put the model behind an HTTP API in a different datacenter.

It's especially harmful because, in a mature organization, the models are often owned by one team and the database by anothern. Both teams (let's assume the best) may be using efficient implementations and purpose-built tech, but the latency problem lies in the gap between them while communicating over a wire, and it's impossible to solve due to Conway's Law.

Eliminating this gap, with it's cost and organizational misalignment is central to the design of PostgresML.

One query. One system. One team. Simple, fast, and efficient.

Rather than shipping the entire vector back to an application like a normal vector database, PostgresML includes all the algorithms needed to compute results internally. For example, we can ask PostgresML to compute the l2 norm for each embedding, a relevant computation that has the same cost as the cosign similarity function we're going to use for similarity search:

- Query

- Result

```
SELECT pgml.norm_l2(review_embedding_e5_large)
FROM  pgml.amazon_us_reviews
LIMIT 5;
```

Most people would assume that "complex ML functions" with `O(n * m)` runtime will increase load on the database compared to a "simple" `SELECT *`, but in fact, *moving the function to the database reduced the latency 50 times over*, and now our application doesn't need to do the "ML function" at all. This isn't just a problem with Postgres or databases in general, it's a problem with all programs that have to ship vectors over a wire, aka microservice architectures full of "feature stores" and "vector databases".

> Shuffling the data between programs is often more expensive than the actual computations the programs perform.

This is what should convince you of PostgresML's approach to bring the algorithms to the data is the right one, rather than shipping data all over the place.

We're not the only ones who think so. Initiatives like Apache Arrow prove the ML community is aware of this issue, but Arrow and Google's Protobuf are not a solution to this problem, they're excellently crafted band-aids spanning the festering wounds in complex ML systems.

> For legacy ML systems, it's time for surgery to cut out the necrotic tissue and stitch the wounds closed.

Some systems start simple enough, or deal with little enough data, that these inefficiencies don't matter. Over time however, they will increase financial costs by orders of magnitude. If you're building new systems, rather than dealing with legacy data pipelines, you can avoid learning these painful lessons yourself, and build on top of 40 years of solid database engineering instead.

Similarity Search

I hope my rant convinced you it's worth wrapping your head around some advanced SQL to handle this task more efficiently. If you're still skeptical, there are more

benchmarks to come. Let's go back to our 5 million movie reviews.

We'll start with semantic search. Semantic search, given a user query, e.g. "Best 1980's scifi movie", is perfomed over embeddings to find the most similar reviews to a user request. We'll use the `cosine distance` operator `<=>` to compare the request embedding to the review embedding, then sort by the closest match and take the top 5.

Cosine similarily is defined as `1 - cosine distance`. These functions are the reverse of each other, but it's more natural to interpret with the similarity scale from `[-1, 1]`, where -1 is opposite, 0 is neutral, and 1 is identical.

- Query
- Result

```
WITH request AS (
  SELECTpgml.embed(
    'intfloat/e5-large',
    'Best 1980''s scifi movie'
  )::vector(1024) AS embedding
)

SELECT
  review_body,
  product_title,
  star_rating,
  total_votes,
  1 - (
    review_embedding_e5_large <=> (
      SELECT embedding FROM request
    )
  ) AS cosine_similiarity
FROM  pgml.amazon_us_reviews
ORDER BY  review_embedding_e5_large <=> (
    SELECT embedding FROM request
  )
LIMIT 5;
```

tips_and_updatesTip

Common Table Expressions (CTEs) that begin `WITH name AS (...)` can be a nice way to organize complex queries into more modular sections. They also make it easier for Postgres to create a query plan, by introducing an optimization gate and separating the conditions in the CTE from the rest of the query.

Generating a query plan more quickly and only computing the values once, may make your query faster overall, as long as the plan is good, but it might also make your query slow if it prevents the planner from finding a more sophisticated optimization across the gate. It's often worth checking the query plan with and without the CTE to see if it makes a difference. We'll cover query plans and tuning in more detail later.

There's some good stuff happening in those query results, so let's break it down:

- **It's fast** - We're able to generate a request embedding on the fly with a state-of-the-art model, and search 5M reviews in 152ms, including fetching the results back to the client 😍. You can't even generate an embedding from OpenAI's API in that time, much less search 5M reviews in some other database with it.
- **It's good** - The `review_body` results are very similar to the "Best 1980's scifi movie" request text. We're using the `intfloat/e5-large` open source embedding model, which outperforms OpenAI's `text-embedding-ada-002` in most quality benchmarks.
    - Qualitatively: the embeddings understand our request for `scifi` being equivalent to `Sci-Fi`, `sci-fi`, `SciFi`, and `sci fi`, as well as `1980's` matching `80s` and `80's` and is close to `seventies` (last place). We didn't have to configure any of this and the most enthusiastic for "best" is at the top, the least enthusiastic is at the bottom, so the model has appropriately captured "sentiment".
    - Quantitatively: the `cosine_similarity` of all results are high and tight, 0.90-0.95 on a scale from -1:1. We can be confident we recalled very similar results from our 5M candidates, even though it would take 485 times as long to check all of them directly.
- **It's reliable** - The model is stored in the database, so we don't need to worry about managing a separate service. If you repeat this query over and over, the timings will be extremely consistent, because we don't have to deal with things like random network congestion.
- **It's SQL** - `SELECT`, `ORDER BY`, `LIMIT`, and `WITH` are all standard SQL, so you can use them on any data in your database, and further compose queries with standard SQL.

This seems to actually just work out of the box... but, there is some room for improvement.

Yeah, well, that's just like, your opinion, man

1. **It's a single persons opinion** - We're searching individual reviews, not all reviews for a movie. The correct answer to this request is undisputedly "Episode V: The Empire Strikes Back". Ok, maybe "Blade Runner", but I really did like "Back to the Future"... Oh no, someone on the internet is wrong, and we need to fix it!
2. **It's approximate** - There are more than four 80's Sci-Fi movie reviews in this dataset of 5M. It really shouldn't be including results from the 70's. More relevant reviews are not being returned, which is a pretty sneaky optimization for a database to pull, but the disclaimer was in the name.
3. **It's narrow** - We're only searching the review text, not the product title, or incorporating other data like the star rating and total votes. Not to mention this is an intentionally crafted semantic search, rather than a keyword search of people looking for a specific title.

We can fix all of these issues with the tools in PostgresML. First, to address The Dude's point, we'll need to aggregate reviews about movies and then search them.

Aggregating reviews about movies

We'd really like a search for movies, not reviews, so let's create a new movies table out of our reviews table.

We can use SQL aggregates over the reviews to generate some simple stats for each movie, like the number of reviews and average star rating. PostgresML provides aggregate functions for vectors.

A neat thing about embeddings is if you sum a bunch of related vectors up, the common components of the vectors will increase, and the components where there isn't good agreement will cancel out. The `sum` of all the movie review embeddings will give us a representative embedding for the movie, in terms of what people have said about it.

Aggregating embeddings around related tables is a super powerful technique. In the next post, we'll show how to generate a related embedding for each reviewer, and then we can use that to personalize our search results, but one step at a time.

- Query
- Result

```
CREATE TABLE movies ASSELECT  product_id AS id,
  product_title AS title,
  product_parent AS parent,
  product_category AS category,
  count(*) AS total_reviews,
  avg(star_rating) AS star_rating_avg,
pgml.sum(review_embedding_e5_large)::vector(1024) AS review_embedding_e5_large
FROM pgml.amazon_us_reviews
GROUP BY product_id, product_title, product_parent, product_category;
```

We've just aggregated our original 5M reviews (including their embeddings) into 300k unique movies. I like to include the model name used to generate the embeddings in the column name, so that as new models come out, we can just add new columns with new embeddings to compare side by side. Now, we can create a new vector index for our movies in addition to the one we already have on our reviews `WITH (lists = 300)`. `lists` is one of the key parameters for tuning the vector index; we're using a rule of thumb of about 1 list per thousand vectors.

- Query
- Result

```
CREATE INDEX CONCURRENTLY  index_movies_on_review_embedding_e5_large
ON movies
USING ivfflat (review_embedding_e5_large vector_cosine_ops)
WITH (lists = 300);
```

Now we can quickly search for movies by what people have said about them:

- Query
- Result

```
WITH request AS (
  SELECTpgml.embed(
    'intfloat/e5-large',
    'Best 1980''s scifi movie'
  )::vector(1024) AS embedding
)
SELECT  title,
  1 - (
    review_embedding_e5_large <=> (SELECT embedding FROM request)
  ) AS cosine_similiarity
FROM  movies
ORDER BY  review_embedding_e5_large <=> (SELECT embedding FROM request)
LIMIT 10;
```

It's somewhat expected that the movie vectors will have been diluted compared to review vectors during aggregation, but we still have results with pretty high cosine similarity of ~0.85 (compared to ~0.95 for reviews).

It's important to remember that we're doing *Approximate* Nearest Neighbor (ANN) search, so we're not guaranteed to get the exact best results. When we were searching 5M reviews, it was more likely we'd find 5 good matches just because there were more candidates, but now that we have fewer movie candidates, we may want to dig deeper into the dataset to find more high quality matches.

Tuning vector indexes for recall vs speed

Inverted File Indexes (IVF) are built by clustering all the vectors into `lists` using cosine similarity. Once the `lists` are created, their center is computed by summing all the vectors in the list. It's the same thing we did as clustering the reviews around their movies, except these clusters are just some arbitrary number of similar vectors.

When we perform a vector search, we will compare to the center of all `lists` to find the closest ones. The default number of `probes` in a query is 1. In that case, only the closest `list` will be exhaustively searched. This reduces the number of vectors that need to be compared from 300,000 to (300 + 1000) = 1300. That saves a lot of work, but sometimes the best results were just on the edges of the `lists` we skipped.

Most applications have an acceptable latency limit. If we have some latency budget to spare, it may be worth increasing the number of `probes` to check more `lists` for better recall. If we up the number of `probes` to 300, we can exhaustively search all lists and get the best possible results:

```
SET ivfflat.probes = 300;
```

- Query
- Result

```
WITH request AS (
  SELECTpgml.embed(
    'intfloat/e5-large',
    'Best 1980''s scifi movie'
  )::vector(1024) AS embedding
)
SELECT  title,
  1 - (
    review_embedding_e5_large <=> (SELECT embedding FROM request)
  ) AS cosine_similiarity
FROM  movies
ORDER BY  review_embedding_e5_large <=> (SELECT embedding FROM request)
LIMIT 10;
```

There's a big difference in the time it takes to search 300,000 vectors vs 1,300 vectors, almost 20 times as long, although it does find one more vector that was not in the original list:

```
 Big Trouble in Little China [UMD for PSP]                        | 0.8649691870870362
```

This is a weird result. It's not Sci-Fi like all the others and it wasn't clustered with them in the closest list, which makes sense. So why did it rank so highly? Let's dig into the

individual reviews to see if we can tell what's going on.

Digging deeper into recall quality

SQL makes it easy to investigate these sorts of data issues. Let's look at the reviews for `Big Trouble in Little China [UMD for PSP]`, noting it only has 1 review.

- Query
- Result

```
SELECT  review_body
FROM  pgml.amazon_us_reviews
WHERE  product_title = 'Big Trouble in Little China [UMD for PSP]';
```

This confirms our model has picked up on lingo like "flick" = "movie", and it seems it must have strongly associated "cult" flicks with the "scifi" genre. But, with only 1 review, there hasn't been any generalization in the movie embedding. It's a relatively strong match for a movie, even if it's not the best for a single review match (0.86 vs 0.95).

Overall, our movie results look better to me than the titles pulled just from single reviews, but we haven't completely addressed The Dudes point as evidenced by this movie having a single review and being out of the requested genre. Embeddings often have fuzzy boundaries that we may need to firm up.

Adding a filter to the request

To prevent noise in the data from leaking into our results, we can add a filter to the request to only consider movies with a minimum number of reviews. We can also add a filter to only consider movies with a minimum average review score with a `WHERE` clause.

```
SET ivfflat.probes = 1;
```

- Query
- Result

```
WITH request AS (
  SELECTpgml.embed(
```

```
    'intfloat/e5-large',
    'Best 1980''s scifi movie'
  )::vector(1024) AS embedding
)

SELECT  title,
  total_reviews,
  1 - (
    review_embedding_e5_large <=> (SELECT embedding FROM request)
  ) AS cosine_similiarity
FROM  movies
WHEREtotal_reviews > 10ORDER BY  review_embedding_e5_large <=> (SELECT embedding FROM requ
est)
LIMIT 10;
```

There we go. We've filtered out the noise, and now we're getting a list of movies that are all Sci-Fi. As we play with this dataset a bit, I'm getting the feeling that some of these are legit (Alien), but most of these are a bit too out on the fringe for my interests. I'd like to see more popular movies as well. Let's influence these rankings to take an additional popularity score into account.

Boosting and Reranking

There are a few simple examples where NoSQL vector databases facilitate a killer app, like recalling text chunks to build a prompt to feed an LLM chatbot, but in most cases, it requires more context to create good search results from a user's perspective.

As the Product Manager for this blog post search engine, I have an expectation that results should favor the movies that have more `total_reviews`, so that we can rely on an established consensus. Movies with higher `star_rating_avg` should also be boosted, because people very explicitly like those results. We can add boosts directly to our query to achieve this.

SQL is a very expressive language that can handle a lot of complexity. To keep things clean, we'll move our current query into a second CTE that will provide a first-pass ranking for our initial semantic search candidates. Then, we'll re-score and rerank those first round candidates to refine the final result with a boost to the `ORDER BY` clause for movies with a higher `star_rating_avg`:

- Query
- Result

```
WITH request AS (
  SELECTpgml.embed(
    'intfloat/e5-large',
    'Best 1980''s scifi movie'
  )::vector(1024) AS embedding
),

-- We moved our original query into a CTE called "first_pass"
first_pass AS (
    SELECT      title,
      total_reviews,
      star_rating_avg,
      1 - (
        review_embedding_e5_large <=> (SELECT embedding FROM request)
      ) AS cosine_similiarity,
      star_rating_avg / 5 AS star_rating_score
    FROM movies
    WHERE total_reviews > 10
    ORDER BY review_embedding_e5_large <=> (SELECT embedding FROM request)
    LIMIT 1000
)

SELECT  title,
  total_reviews,
  round(star_rating_avg, 2) as star_rating_avg,
  star_rating_score,
  cosine_similiarity
  cosine_similiarity + star_rating_score AS final_score,
FROM first_pass
ORDER BY final_score DESCLIMIT 10;
```

This is starting to look pretty good! True confessions: I'm really surprised "Empire Strikes Back" is not on this list. What is wrong with people these days?! I'm glad I called "Blade Runner" and "Back to the Future" though. Now, that I've got a list that is catering to my own sensibilities, I need to stop writing code and blog posts and watch some of these! In the next article, we'll look at incorporating more of ~~my preferences~~ a customer's preferences into the search results for effective personalization.

P.S. I'm a little disappointed I didn't recall Aliens, because yeah, it's perfect 80's Sci-Fi, but that series has gone on so long I had associated it all with "vague timeframe". No one is perfect... right? I should probably watch "Plan 9 From Outer Space" & "Forbidden Planet", even though they are both 3 decades too early. I'm sure they are great!

Have Questions?

Join our Discord and ask us anything! We're friendly and would love to talk about PostgresML.