How To Build a Real-Time Product Recommendation System Using Redis and DocArray

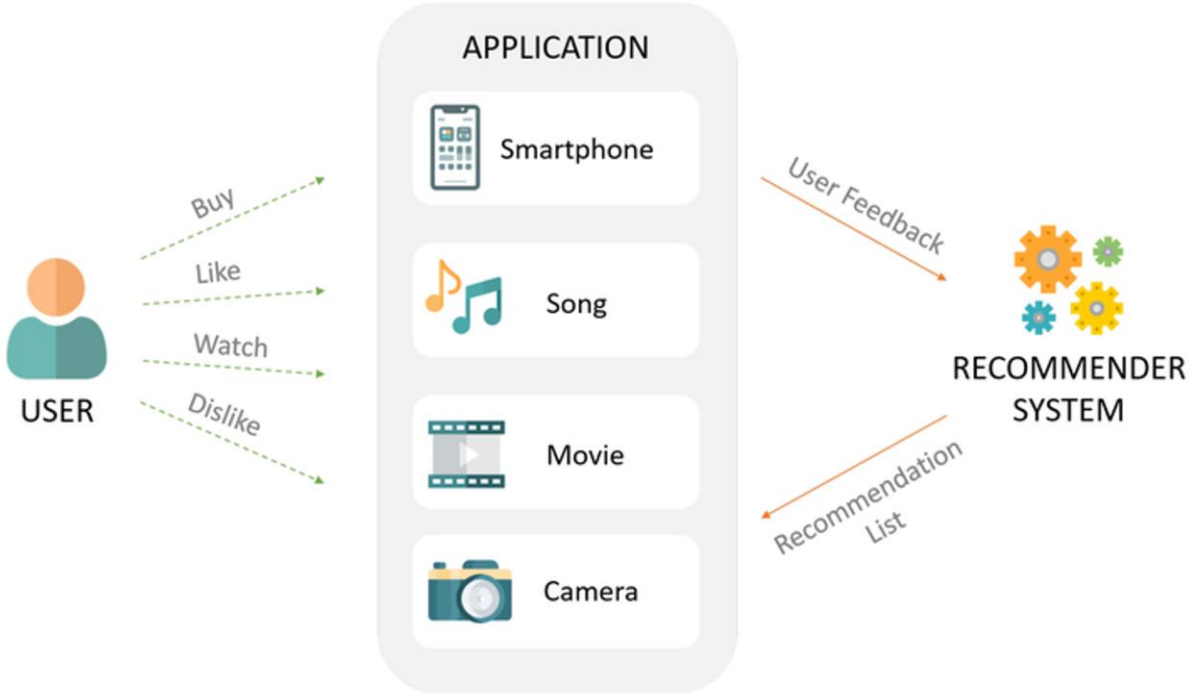
November 08, 20228 minute read

Alaeddine Abdessalem

This tutorial helps you build a real-time product recommendation system for an e-commerce system using content-based filtering and vector similarity search. Follow along to learn the essential steps and how it works.

Recommendation systems are an important technology for most online businesses and for e-commerce sites in particular. They're an essential element in generating good conversion and maintaining customer loyalty.

A recommendation system typically shows items to users based on their profiles and preferences and by observing their actions (such as buying, liking, or viewing items).



Consider the challenges involved in building a recommendation system for a modern e-commerce site. This is just a subset of the issues to consider:

- **Customization: Customers want to filter results, such as by price range, brand, and size. Our system should recommend products only within these parameters.**
- **Multiple modalities: A product listing is more than a text description. It can also contain images, video, audio, and 3D mesh, for example. All available data modalities should be exploited when making recommendations.**
- **Latency: Customers expect recommendations to appear quickly. If the system's recommendations aren't returned immediately, they're irrelevant.**

- **Data volume: The more products and customers the site has, the harder it is to recommend products efficiently. Computing recommendations should stay fast even with big datasets.**

As these requirements evolve, approaches to building recommendation systems need to evolve as well. In this blog post, we show you how to build a real-time product recommendation system with respect to user-defined filters using the latest vector search technology. The tool suite includes Redis and DocArray, but the methodology is relevant no matter which tools you employ.

Recommendation systems basics

As with any other computing problem, there are multiple approaches to building recommendation systems and tools to support each effort. They include:

- **Collaborative filtering: The system predicts items relevant to the user based on the preferences of similar users.**
- **Content-based filtering: The system models the user's interests as feature vectors and predicts relevant items based on the similarity between vectors.**
- **Hybrid approaches: The system combines collaborative filtering, content-based filtering, and other approaches.**

This blog post focuses on improving the content-based filtering approach. If you're new to this topic, it may help to read Google's overview of content-based filtering first.

There are two important considerations when implementing content-based filtering:

First, when you model the user and items as a feature vector, it's important to exploit all modalities of the data. Simply relying on keywords or a set of engineered features might not efficiently represent complex data.

That's why state-of-the-art AI models are important, as they represent complex, multimodal data as vector embeddings.

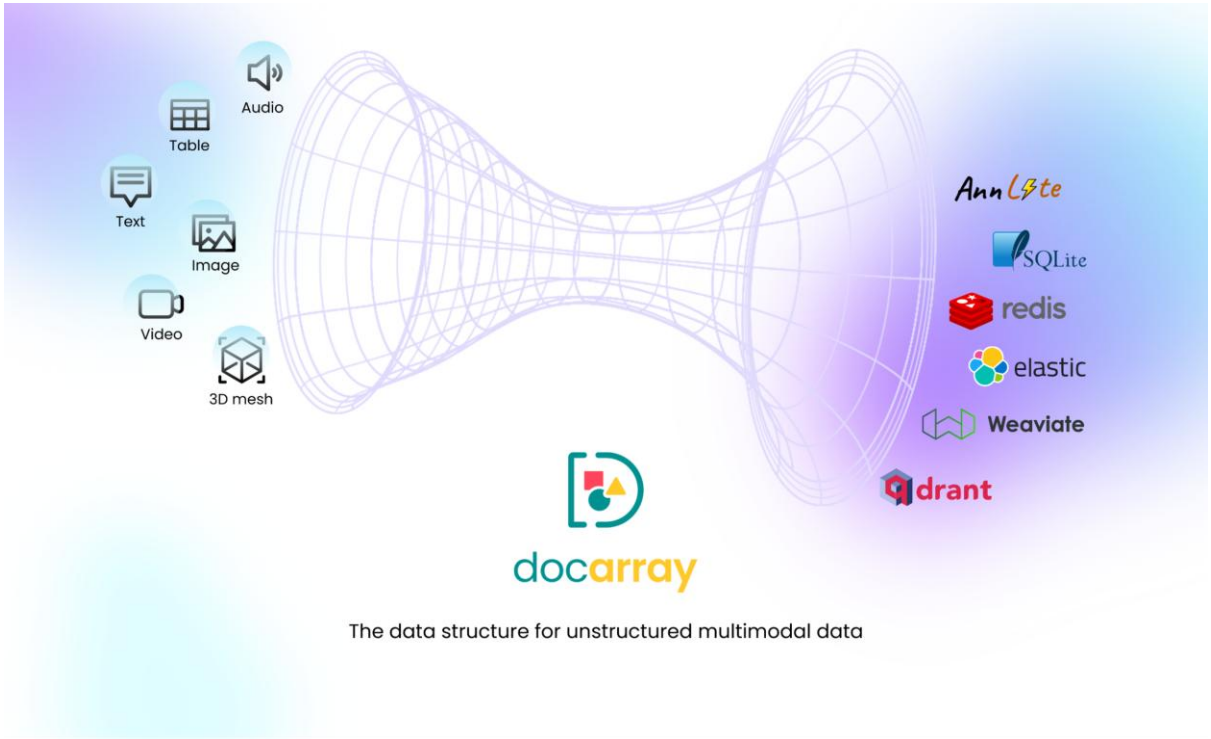
One of the best-known models to represent both text and image data is CLIP. Therefore, in this tutorial, we use CLIP-as-service as the inference engine that powers our recommendations.

Also, computing vector similarity can be slow and costly if not performed efficiently. Our application requirements (to respect user filters and deliver low-latency recommendations) make it impractical to pre-compute similarity between items and user profiles in batch jobs. That's why it's crucial to compute vector similarity in real-time, using efficient techniques such as Hierarchical Navigable Small World (HNSW).

These techniques are implemented in vector databases. Redis offers vector search capabilities in RediSearch 2.4. And since Redis is an in-memory database, recommending items is both fast and performed in a real-time context.

With feature representation and computing vector similarity covered, we still need a data structure to bridge the gap between our multimodal data and the vector database. For that, we use DocArray. Think of DocArray as a universal vector database client with support for

multimodal data. It has a Pythonic interface that makes it easy to build a recommendation system in just a few lines of code.



Designing the solution

We have assembled the tools for this application: Redis for Vector Similarity Search, CLIP-as-service to encode visual data, and DocArray to represent multimodal documents and connect to Redis. In this tutorial, we apply these technologies to build a content-based filtering recommendation system.

The procedure is as follows:

- **Load the dataset into DocArray format.**
- **Model products by encoding product images using CLIP-as-service.**
- **Model the user profile by computing the weighted average of the last *k* viewed products' embeddings.**
- **Use DocArray to load the product data.**
- **Index the product data in Redis.**
- **Use Redis Vector Similarity Search to recommend the most similar items to the user's view history while also filtering those results to respect user preferences.**

In the instruction that follows, the data we use for product recommendations comes from the Amazon Berkeley Objects Dataset, a dataset of Amazon products with metadata, catalog images, and 3D models.

Setup

The first step is to provision a Redis instance. You can create a local Redis instance using Docker:

docker run -d -p 6379:6379 redis/redis-stack:latest

Copied!

Next, we need to install DocArray, Jina, and clip-client:

pip install docarray[redis] jina clip-client

Copied!

That's all the setup we need. Now we're ready to start exploring with data.

Dataset exploration

The Amazon Berkeley Objects Dataset consists of product items accompanied by images and metadata such as brand, country, and color. It represents the inventory of an e-commerce website.

Name: Nia & Nicole Women's Wallet (Tan)

Category: WALLET

Country: IN

Brand: Nia & Nicole

Width: 2000

Height: 2000

For the purposes of this tutorial, we can download a subset of this dataset from Jina Cloud, pre-processed in DocArray format.

First, authenticate to Jina Cloud using the terminal:

jina auth login

Copied!

Next, download the dataset:

from docarray import DocumentArray, Document

da = DocumentArray.pull('amazon-berkeley-objects-dataset', show_progress=True)

Copied!

This returns a DocumentArray object containing samples from the Amazon Berkeley Objects dataset. We get an overview using the summary() method:

da.summary()

Copied!

```
╭──────────────────── Documents Summary ─────────────────────╮
│                                                            │
│  Type                DocumentArrayInMemory                 │
│  Length              **5809**                              │
│  Homogenous Documents  *True*                              │
│  Common Attributes   **(**'id', 'mime_type', 'uri', 'tags'**)**  │
│  Multimodal dataclass  *False*                             │
│                                                            │
╰────────────────────────────────────────────────────────────╯

╭──────────────────── Attributes Summary ────────────────────╮
│                                                            │
│  **Attribute**  **Data type**  **#Unique values**  **Has empty value**  │
│ ──────────────────────────────────────────────────────────  │
│  id         **(**'str',**)**  **5809**        *False*      │
│  mime_type  **(**'str',**)**  **1**           *False*      │
│  tags       **(**'dict',**)**  **5809**       *False*      │
│  uri        **(**'str',**)**  **4848**        *False*      │
│                                                            │
╰────────────────────────────────────────────────────────────╯
```

Copied!

Or we can display the images of the first items using the [plot_image_sprites()](<https://docarray.jina.ai/api/docarray.array.document/#docarray.array.document.DocumentArray.plot_image_sprites>) method.

```
da[:12].plot_image_sprites()
```

Copied!

Each product contains the metadata information in the tags field.

Let’s take a look at the content of tags:

da[0].tags

Copied!

{'height': '1926',

'country': 'CA',

'width': '1650',

'product_type': 'ACCESSORY',

'color': 'Blue',

'brand': 'Thirty Five Kent',

'item_name': "Thirty Five Kent Men's Cashmere Zig Zag Scarf, Blue"}

Copied!

Later, we use this metadata to filter recommendations according to the user’s preferences.

Adding embeddings

To create the vector embeddings for our dataset, we first need a token for CLIP-as-service inference:

```
# Create a Jina token to be used for CLIP-as-service inference

jina auth token create fashion -e 30
```

Copied!

Then we can start to encode the data. Be sure to pass the created token to the Client object:

```
from clip_client import Client


c = Client(
   'grpcs://api.clip.jina.ai:2096', credential={'Authorization': 'your-auth-token'}
)


encoded_da = c.encode(da, show_progress=True)
```

Copied!

Encoding the dataset takes a few minutes. When it's done, we proceed with the next steps.

Connecting to Redis

At this point, our data is encoded and ready to index.

To do so, we create a DocumentArray instance connected to our Redis server. It's important to specify the correct embedding dimensions and filter columns:

```
# Configure a new DocumentArray with a Redis document store

redis_da = DocumentArray(storage='redis', config={
   'n_dim': 768,
   'columns': {
      'color': 'str',
      'country': 'str',
      'product_type': 'str',
      'width': 'int',
      'height': 'int',
      'brand': 'str',
   }
})
```

 # Index data

redis_da.extend(encoded_da

Copied!

For more information, see Redis Document Store in DocArray.

Generating recommendations

In order to understand the recommendation logic, consider the following example: Eleanor decided to add a scarf to her wardrobe and looked into several ones in our shop. Her favorite color is navy, and she has to keep the budget under $25.

Our recommendation function should allow specifying those requirements and recommend items based on the view history, with emphasis on the most recently viewed items.

Thus, we combine embeddings of the latest viewed items by giving more weight to the recent items:

Let's implement a function that recommends products based on a weighted average of the embeddings of recently-viewed items, taking user filters into account. That is, in recommended items, we want to emphasize the items the shopper viewed most recently.

Thus, we combine embeddings of the latest viewed items by giving more weight to the recent items:

```
import numpy as np


def recommend(view_history, color=None, country=None):
  embedding = np.average(
    [doc.embedding for doc in view_history],
    weights=range(len(view_history), 0, -1),
    axis=0
  )

  user_filter = ''

  if color:
    user_filter += f'@color:{color} '

  if country:
```

```
    user_filter += f'@country:{country} '


  return redis_da.find(embedding, filter=user_filter)
```

Copied!

Adding recommendations to product views

To show relevant recommendations when a customer views a product, we need three steps:

1. **Show the product's image and description.**
2. **Add the product to the list of last *k* viewed products.**
3. **Show recommendations related to the last *k* viewed products.**

We can achieve that with the following function:

```
k = 5

view_history = []


def view(item: Document, view_history, color=None, country=None):
  print(item.tags['item_name'], ':')
  item.display()
  view_history.insert(0, item)
  view_history = view_history[:k]
  recommendations = recommend(view_history, color=color, country=country)
  recommendations.plot_image_sprites()
  return recommendations
```

Copied!

Viewing the results

Let's try it out a few times. First, let's view the first item in the store and the recommendations for it:

```
recommended = view(redis_da[0], view_history
```

Copied!

That displays an attractive scarf, labeled as " Thirty-Five Kent Men's Cashmere Zig Zag Scarf, Blue":

… and the accompanying recommendations:

How well do they meet the user's filter?

Let's check the third item in the recommendation list and apply a filter color='Navy' to ensure a better match:

```
recommended = view(recommended[2], view_history, color='Navy')
```

Copied!

...which generates a better item display and recommendations:

Thirty-Five Kent Men's Cashmere Zig Zag Scarf, Blue:

Now the recommendation function returns the most visually similar items to scarves that also satisfy the filter color='Navy'.

Success! And, possibly, a new e-commerce sale.

Putting it all together

The instructions above are a brief overview to demonstrate the building blocks for a process to recommend products in an online store.

You're welcome to take it further. We created a GitHub repository with source code for a product store interface with the same dataset and technique we just showed.

This demonstration just showed you how Vector Similarity Search can offer low-latency real-time recommendations that respect user preferences and filter selection.

But how fast is it? Let's look at the latency numbers for recommendation queries. You can find logs in the command line console:

Retrieving products ... Retrieving products takes 0 seconds (0.01s)

Copied!

This means computing recommendations takes about 10 milliseconds!

Of course, there's still room for improvement, especially when it comes to quality. For example, we can come up with more sophisticated ways to model the user's profile and interests. We could also incorporate more types of data, such as 3D mesh and video. That's left as an exercise to the reader.

Shopping for better answers

Vector Similarity Search is an essential technique for implementing recommendations in a real-time context.

Your next steps:

- **Use state-of-the-art AI models to encode multimodal data into vector representations.**
- **Use a vector database to compute vector similarity in a real-time context. For low latency, an in-memory database like Redis is ideal.**
- **Consult the Redis documentation for more information.**

Use DocArray as a convenient data structure for handling multimodal data as well as for interfacing with the vector database. Consult the DocArray documentation to get started and get connected with the Redis AI/ML team for more information on Redis Vector Search.

Get started with the Redis Cloud free tier.

Sections

Share

Get started with Redis today

Speak to a Redis expert and learn more about enterprise-grade Redis today.