OpenSearch\_Search

داکیومنت: سند واحدی است که اطلاعات (متن یا داده های ساخت یافته) را ذخیره می کند. در OpenSearch، اسناد در قالب JSON ذخیره می شوند.

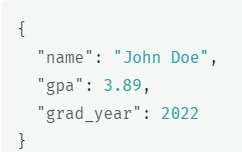
ما می توانیم یک سند را به چند روش در نظر بگیریم:

در پایگاه داده دانشجویان، یک سند ممکن است نشان دهنده یک دانش آموز باشد.

وقتی اطلاعاتی را جستجو می کنید، OpenSearch اسناد مربوط به جستجوی شما را برمی گرداند. یک سند نشان دهنده یک ردیف در یک پایگاه داده سنتی است. به عنوان مثال، در یک پایگاه داده مدرسه، یک سند ممکن است یک دانش آموز را نشان دهد و حاوی داده های زیر باشد.

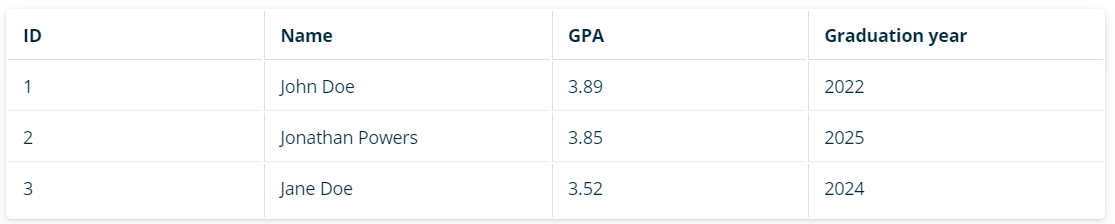


در اینجا شکل این سند در قالب JSON است:



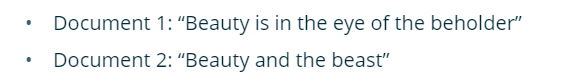
اینکدس: نمایه مجموعه‌ای از اسناد است.

در پایگاه داده دانشجویان، یک نمایه نشان دهنده همه دانش آموزان در پایگاه داده است. وقتی اطلاعاتی را جستجو می کنید، داده های موجود در یک فهرست را پرس و جو می کنید.ایندکس یک جدول پایگاه داده را در یک پایگاه داده سنتی نشان می دهد.به عنوان مثال، در یک پایگاه داده مدرسه، یک نمایه ممکن است شامل همه دانش آموزان مدرسه باشد.

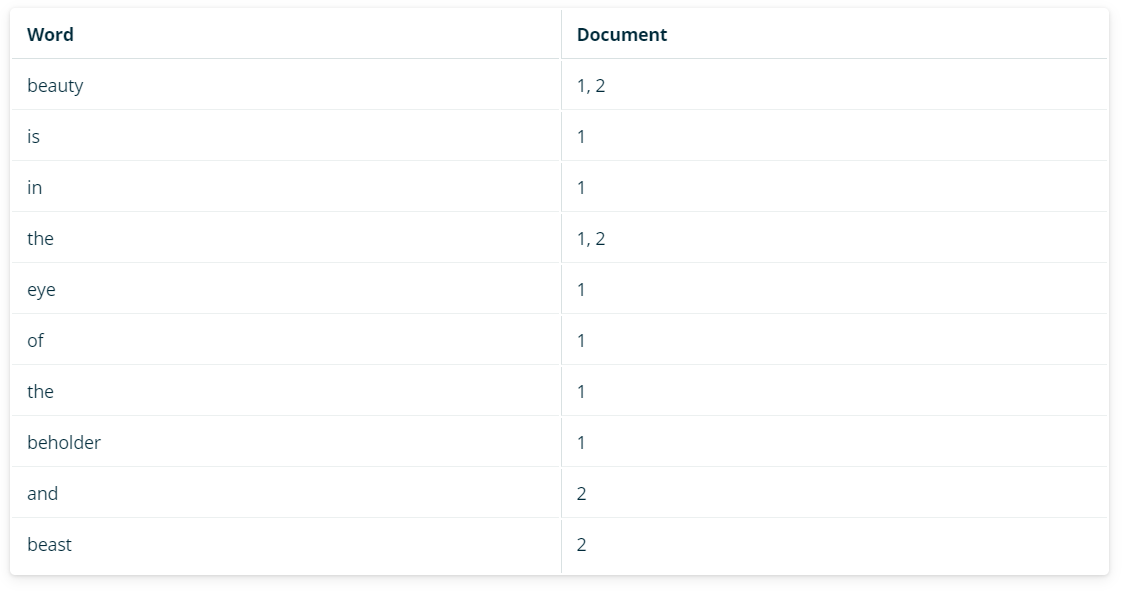


جدول معکوس:

یک نمایه OpenSearch از ساختار داده ای به نام شاخص معکوس استفاده می کند. یک نمایه معکوس کلمات را به اسنادی که در آنها وجود دارند نگاشت می‌کند. به عنوان مثال، یک شاخص شامل دو سند زیر را در نظر بگیرید:



یک نمایه معکوس برای چنین نمایه‌ای، کلمات را به اسنادی که در آنها قرار دارند نگاشت می‌کند:



علاوه بر شناسه سند، OpenSearch موقعیت کلمه را در سند برای اجرای عبارت عبارت ذخیره می کند، جایی که کلمات باید در کنار یکدیگر ظاهر شوند.

ارتباط:

هنگامی که یک سند را جستجو می کنید، OpenSearch کلمات موجود در پرس و جو را با کلمات موجود در اسناد مطابقت می دهد. به عنوان مثال، اگر فهرست توصیف شده در بخش قبل را برای کلمه beauty جستجو کنید، OpenSearch اسناد 1 و 2 را برمی‌گرداند. به هر سند یک امتیاز مرتبط اختصاص داده می‌شود که به شما می‌گوید چقدر سند با درخواست مطابقت دارد.

کلمات منفرد در یک عبارت جستجو را اصطلاحات جستجو می گویند. هر عبارت جستجو بر اساس قوانین زیر امتیازدهی می شود:

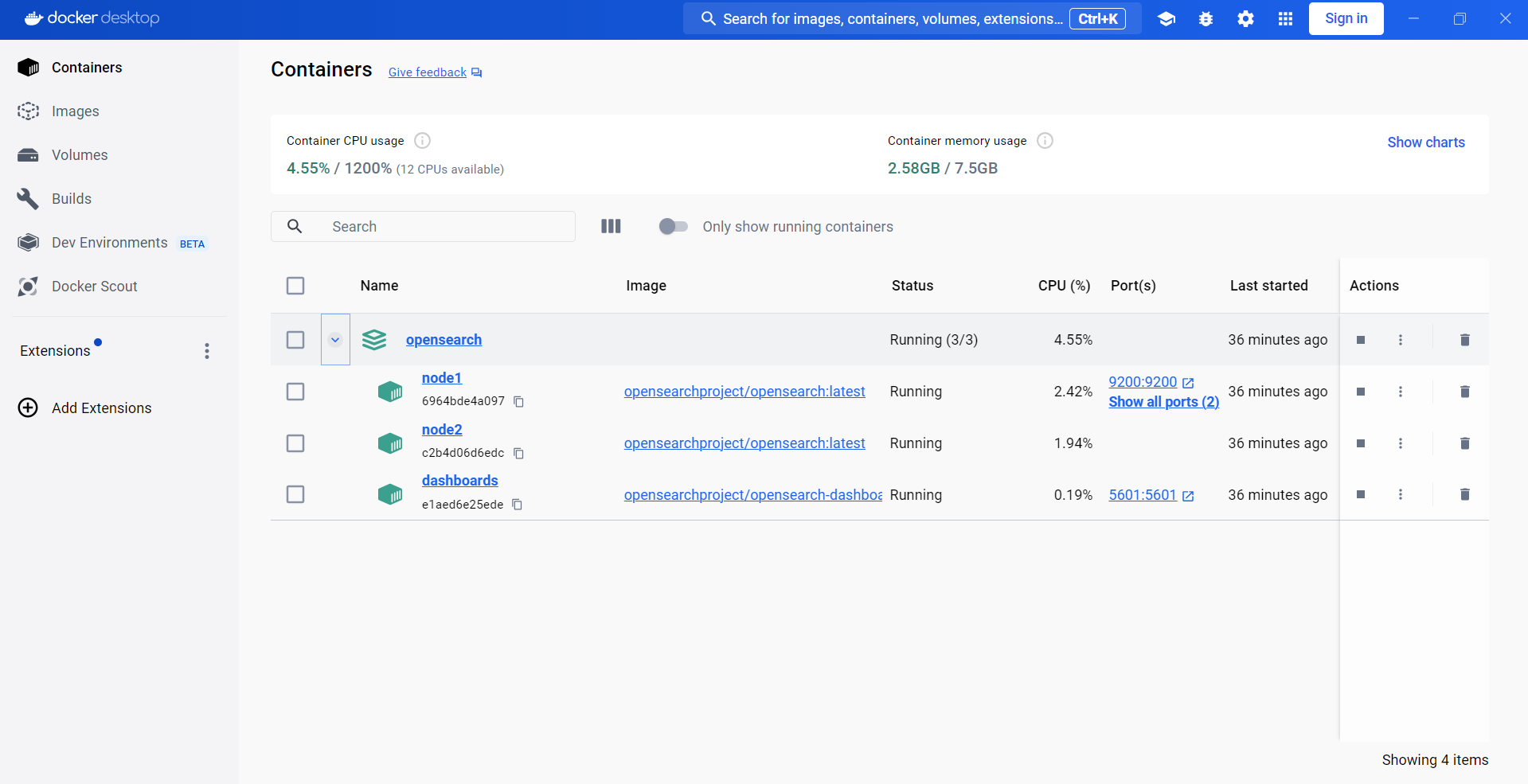
1. عبارت جستجویی که بیشتر در یک سند وجود دارد، امتیاز بیشتری کسب می کند. سندی در مورد سگ‌ها که بارها از کلمه سگ استفاده می‌کند، احتمالاً مرتبط‌تر از سندی است که تعداد کمتری کلمه سگ را در خود دارد. این عبارت جزء فرکانس امتیاز است.
2. عبارت جستجویی که در اسناد بیشتری وجود دارد، امتیاز کمتری دریافت می کند. پرس و جو برای عبارات آبی و axolotl باید اسنادی را که حاوی axolotl هستند بر کلمه احتمالاً رایج تر آبی ترجیح دهد. این جزء معکوس فرکانس سند نمره است.
3. یک تطابق در یک سند طولانی‌تر باید امتیاز کمتری نسبت به یک سند کوتاه‌تر داشته باشد. سندی که حاوی یک فرهنگ لغت کامل باشد، با هر کلمه ای مطابقت دارد، اما به کلمه خاصی مرتبط نیست. این با مولفه نرمال سازی طول امتیاز مطابقت دارد.

این سیستم از الگوریتم رتبه بندی BM25 برای محاسبه امتیازات مربوط به اسناد استفاده می‌کند و سپس نتایج مرتب شده بر اساس ارتباط را برمی گرداند.

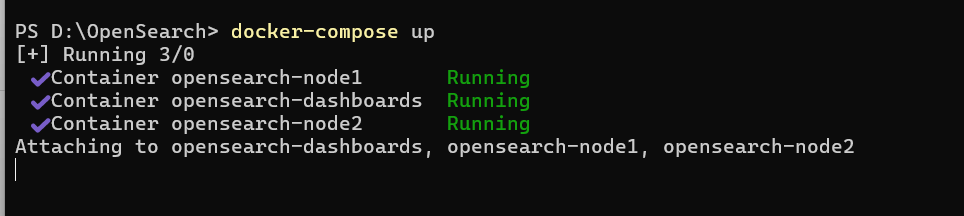
بخش عملی ارتباط با open search

برای اتصال به این سیستم، باید از داکر استفاده نمود. ما به یک فایل خاص به نام فایل Compose نیاز داریم که Docker Compose از آن برای تعریف و ایجاد کانتینرها در کلاستر ما استفاده می‌کند. پروژه OpenSearch یک نمونه فایل Compose را ارائه می دهد که می توانیم برای شروع از آن استفاده کنیم.





با وارد کردن دستور زیر سرویس‌ها فعال و اجرا می‌شوند.



برای اینکه مطمعن شویم سرویس‌ها به درستی اجرا شده است دستور زیر را در مرورگر وارد می‌کنیم:

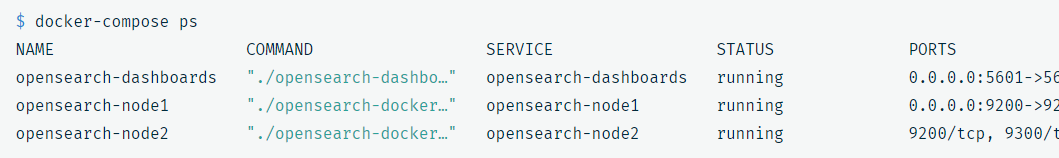
دقت شود پورت 9200 و 9600 برای نود 1و 2 و پورت 5601 برای داشبورد می‌باشد.

Localhost:9200



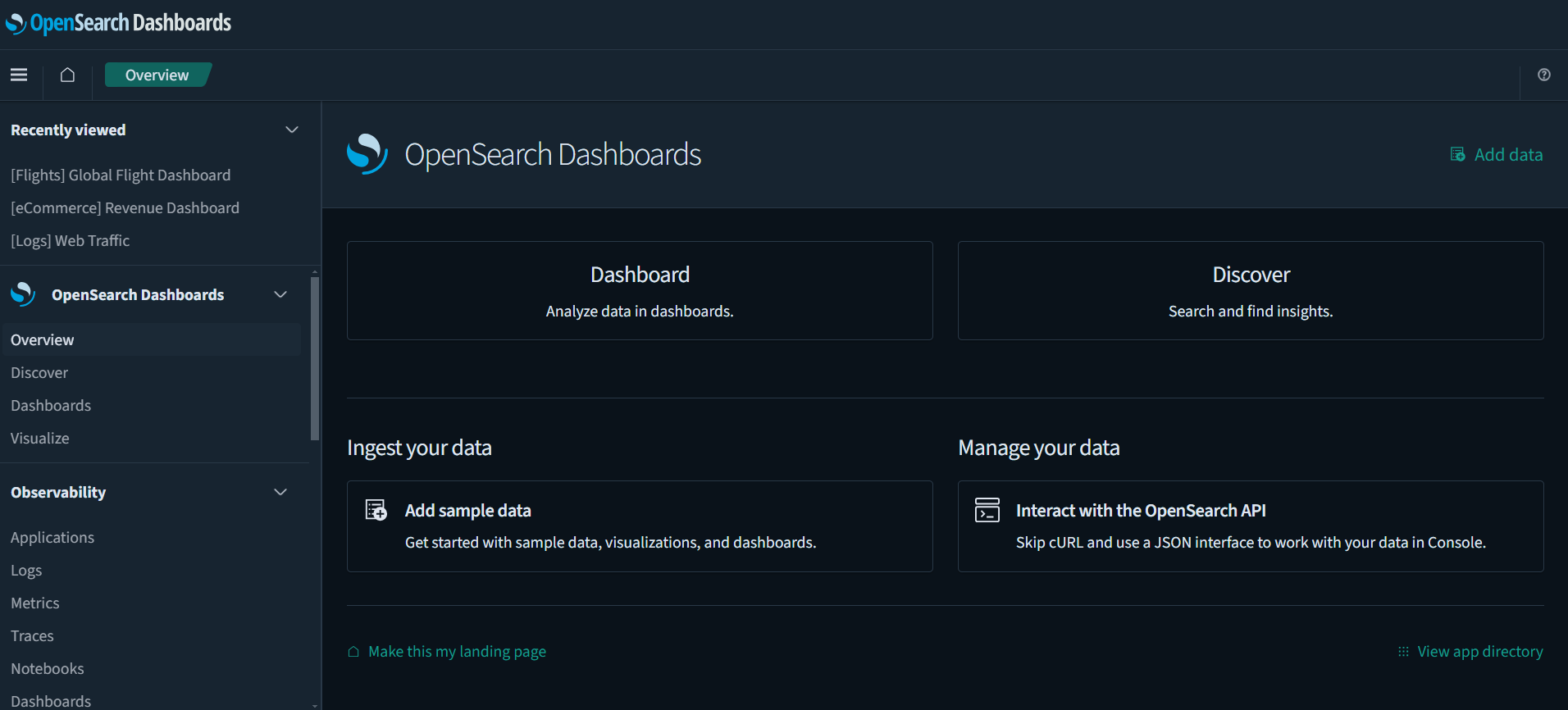
و یا با وارد کردن دستور زیر در ترمینال:

docker-compose up -d

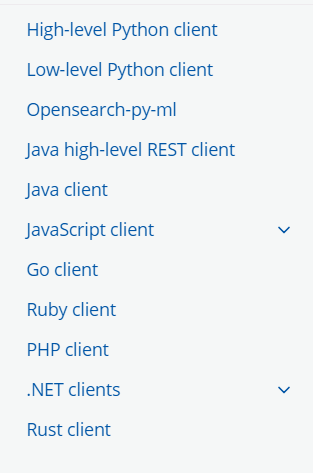


با وارد کردن دستور زیر به داشبورد سیستم متصل می‌شویم:

Localhost:5601



Open search از زبان‌های زیر برای اتصال پشتیبانی می‌کند:



در این پروژه تمامی کدنویسی‌ها با استفاده از زبان پایتون )سطح بالا(می‌باشد:

pip install opensearch-dsl

بعد از نصب می‌توانیم از کتابخانه‌ها برای پروژه استفاده کنیم:

from opensearchpy import OpenSearch

from opensearch\_dsl import Search

برای اتصال به میزبان OpenSearch پیش فرض، اگر از افزونه Security استفاده می‌شود ، یک شی کلاینت با SSL فعال ایجاد کنید.

host = 'localhost'

port = 9200

auth = ('admin', 'admin')

ca\_certs\_path = '/full/path/to/root-ca.pem' # Provide a CA bundle if you use intermediate CAs with your root CA.

# Create the client with SSL/TLS enabled, but hostname verification disabled.

client = OpenSearch(

hosts = [{'host': host, 'port': port}],

http\_compress = True, # enables gzip compression for request bodies

http\_auth = auth,

use\_ssl = True,

verify\_certs = True,

ssl\_assert\_hostname = False,

ssl\_show\_warn = False,

ca\_certs = ca\_certs\_path

)

اگر از افزونه Security استفاده نمی‌کنید، یک شی کلاینت با SSL غیرفعال ایجاد کنید:

نکته: ما در ابتدای کانفیگ قسمت Security را غیر فعال کردیم.

host = 'localhost'

port = 9200

# Create the client with SSL/TLS and hostname verification disabled.

client = OpenSearch(

hosts = [{'host': host, 'port': port}],

http\_compress = True, # enables gzip compression for request bodies

use\_ssl = False,

verify\_certs = False,

ssl\_assert\_hostname = False,

ssl\_show\_warn = False

)

ایجاد شاخص

برای ایجاد یک فهرست OpenSearch، از متد ()client.indices.create استفاده کنید. می‌توانید از کد زیر برای ساخت یک شی JSON با تنظیمات سفارشی استفاده کنید:

index\_name = 'test'

index\_body = {

'settings': {

'index': {

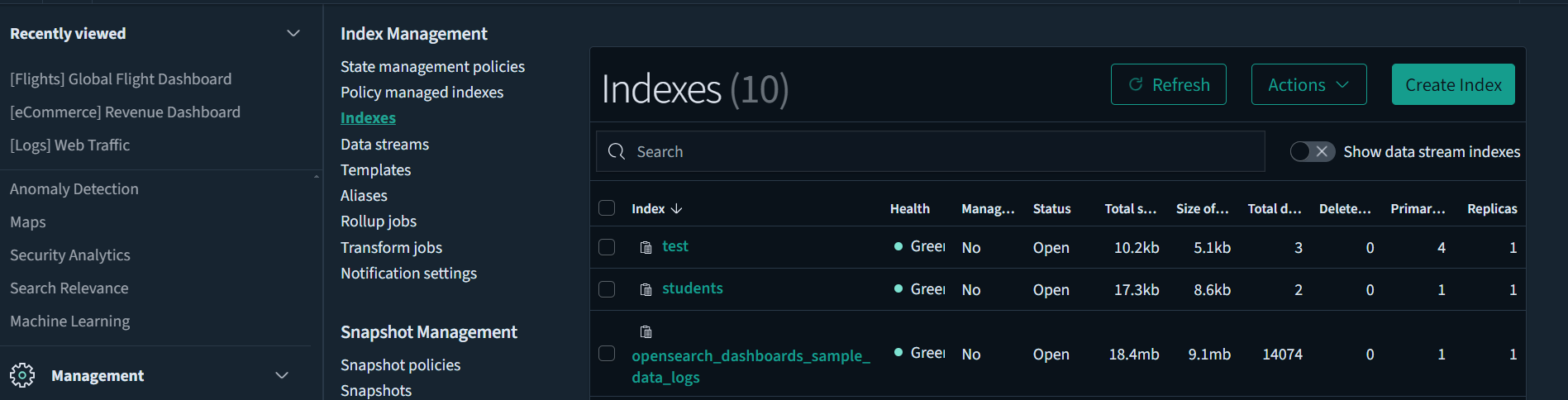
'number\_of\_shards': 4

}

}

}

response = client.indices.create(index\_name, body=index\_body)



می توانید با گسترش کلاس Document یک کلاس برای نشان دادن اسنادی که در OpenSearch ایندکس می کنید ایجاد کنید:

class Movie(Document):

title = Text(fields={'raw': Keyword()})

director = Text()

year = Text()

class Index:

name = index\_name

def save(self, \*\* kwargs):

return super(Movie, self).save(\*\* kwargs)

برای ایندکس کردن یک سند، یک شی از کلاس جدید ایجاد کنید و متد save() آن را فراخوانی کنید:

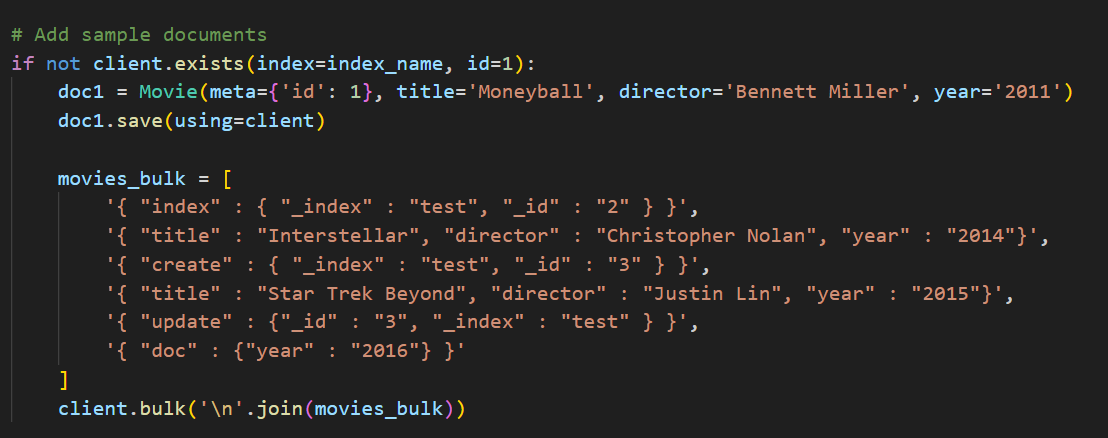
# Set up the opensearch-py version of the document

Movie.init(using=client)

doc = Movie(meta={'id': 1}, title='Moneyball', director='Bennett Miller', year='2011')

response = doc.save(using=client)

با استفاده از متد bulkکلاینت می‌توانید چندین عملیات را همزمان انجام دهید. عملیات ممکن است از یک نوع یا از انواع مختلف باشد. توجه داشته باشید که عملیات باید با \n از هم جدا شوند و کل رشته باید یک خط باشد:



شما می توانید از کلاس Search برای ساخت یک پرس و جو استفاده کنید. کد زیر یک کوئری بولی با فیلتر ایجاد می‌کند:

s = Search(using=client, index=index\_name) \

.filter("term", year="2011") \

.query("match", title="Moneyball")

response = s.execute()

می توانید یک سند را با استفاده از متد ()client.delete حذف کنید:

response = client.delete(

index = 'my-dsl-index',

id = '1'

)

می توانید با استفاده از متد ()client.indices.delete یک فهرست را حذف کنید:

response = client.indices.delete(

index = 'my-dsl-index'

)

رابط کاربری:

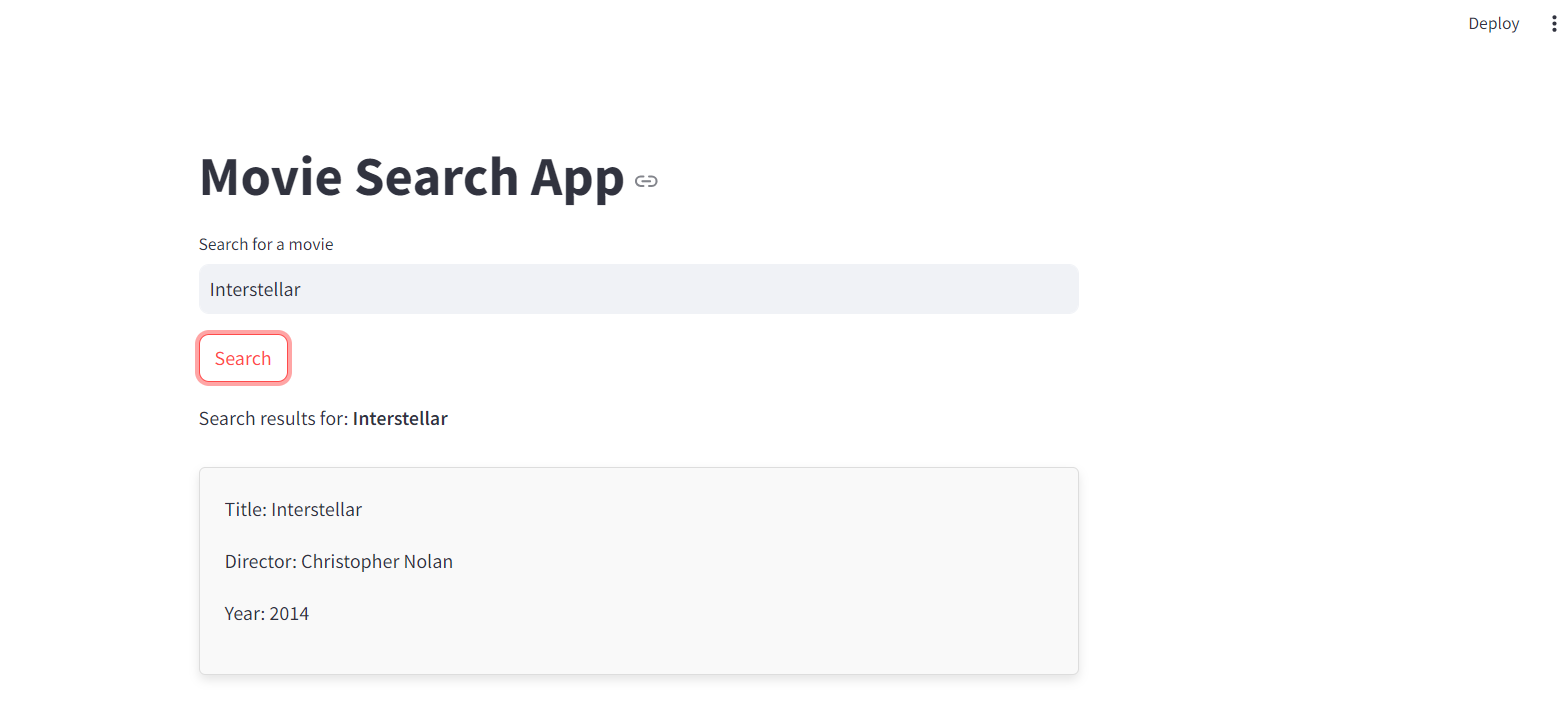
در این پروژه تصمیم گرفته شد رابط کاربری با استفاده از streamlit انجام شود.

streamlit run app.py

خروجی:

http://localhost:8501

با توضیحات بالا یک مثال ساده ببینیم:



در ادامه قدرت سرچ و بازیابی این سیستم به وضوح نشان داده خواهد شد. برای بازیابی اطلاعات ما از یک دیتاست آماده با این [لینک](https://www.kaggle.com/datasets/harshitshankhdhar/imdb-dataset-of-top-1000-movies-and-tv-shows) استفاده کردیم.

بازیابی اطلاعات:

OpenSearch از روش‌های جستجوی زیر پشتیبانی می‌کند:

1. جستجوی واژگانی سنتی

* جستجوی کلمه کلیدی (BM25): مجموعه سند را برای کلماتی که در پرس و جو ظاهر می‌شوند جستجو می‌کند.

1. جستجوی مبتنی بر یادگیری ماشینی (ML).

* جستجوی برداری
* جستجوی k-NN: k-نزدیکترین همسایه یک عبارت جستجو را در میان شاخصی از بردارها جستجو می‌کند.
* جستجوی عصبی: جستجوی عصبی تولید جاسازی‌های برداری را در زمان مصرف و جستجوی آنها در زمان جستجو را تسهیل می‌کند. جستجوی عصبی به ما امکان می‌دهد مدل های ML را در جستجوی خود ادغام کنیم و به عنوان چارچوبی برای اجرای سایر روش های جستجو عمل می کند. روش‌های جستجوی زیر بر روی جستجوی عصبی ساخته شده‌اند:
* جستجوی معنایی: معنای کلمات را در زمینه جستجو در نظر می گیرد. از بازیابی متراکم بر اساس مدل های جاسازی متن برای جستجوی داده های متنی استفاده می کند.
* جستجوی چندوجهی: از مدل‌های تعبیه چند وجهی برای جستجوی داده‌های متن و تصویر استفاده می‌کند.
* جستجوی پراکنده عصبی: از بازیابی پراکنده بر اساس مدل‌های جاسازی پراکنده برای جستجوی داده‌های متنی استفاده می‌کند.
* جستجوی ترکیبی: جستجوی سنتی و جستجوی برداری را برای بهبود ارتباط جستجو ترکیب می‌کند.
* جستجوی مکالمه: یک جستجوی مولد تقویت شده با بازیابی را پیاده سازی می کند.

در OpenSearch، می‌توانید از زبان‌های جستجوی زیر برای جستجوی داده‌های خود استفاده کنید:

* زبان خاص دامنه پرس و جو (DSL): زبان پرس و جو اولیه OpenSearch که از ایجاد پرس و جوهای پیچیده و کاملاً قابل تنظیم پشتیبانی می‌کند.
* Query string query language: یک زبان پرس و جو کوچک شده که می توانید در پارامتر پرس و جو درخواست جستجو یا در داشبوردهای OpenSearch استفاده کنید.
* SQL: یک زبان پرس و جو سنتی که شکاف بین مفاهیم پایگاه داده سنتی رابطه‌ای و انعطاف‌پذیری ذخیره‌سازی اسناد مداربسته OpenSearch را پر می‌کند.
* زبان پردازش لوله ای (PPL): زبان اصلی مورد استفاده با قابلیت مشاهده در OpenSearch. PPL از یک نحو لوله ای استفاده می کند که دستورات را به یک پرس و جو زنجیره می دهد.
* Dashboards Query Language (DQL): یک زبان پرس و جوی مبتنی بر متن ساده برای فیلتر کردن داده ها در داشبوردهای OpenSearch.

عملکرد جستجو

OpenSearch چندین راه برای بهبود عملکرد جستجو ارائه می دهد:

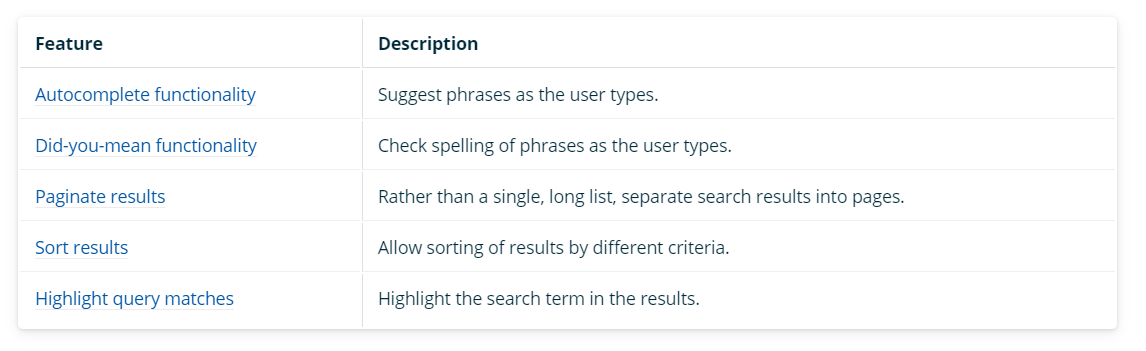
* جستجوی ناهمزمان: جستارهای پرمصرف منابع را به صورت ناهمزمان اجرا می کند.
* جستجوی بخش همزمان: بخش ها را همزمان جستجو می‌کند.

OpenSearch از عملیات رایج زیر در نتایج جستجو پشتیبانی می‌کند:

* Paginate
* Sort
* Highlight search terms
* Autocomplete
* Did-you-mean

**Searching data**

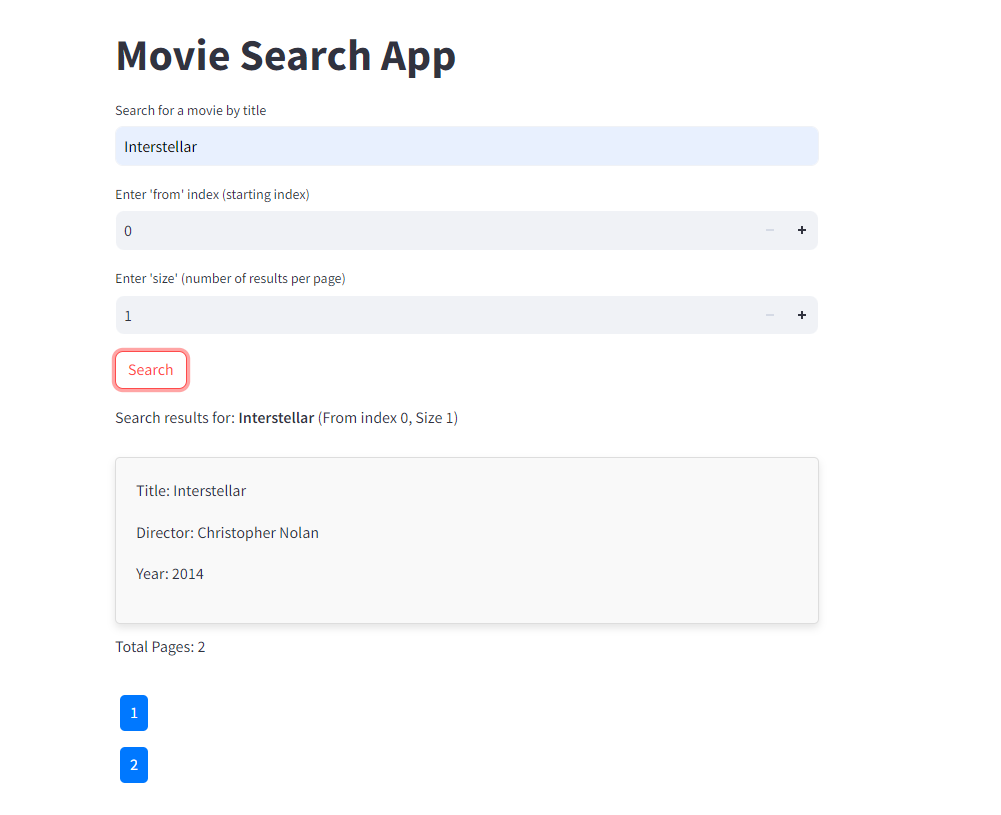
آنچه کاربران از موتورهای جستجو انتظار دارند در طول سال ها تکامل یافته است. فقط بازگرداندن سریع نتایج مرتبط دیگر برای اکثر کاربران کافی نیست. اکنون کاربران به دنبال روش هایی هستند که به آنها امکان می دهد نتایج مرتبط تری را دریافت کنند، نتایج را مرتب و سازماندهی کنند و پرس و جوهای خود را برجسته کنند. OpenSearch شامل ویژگی های زیادی است که در جدول زیر توضیح داده شده است که تجربه جستجو را بهبود می بخشد.



1. **Paginate results**

* The from and size parameters
* The scroll search operation
* The search\_after parameter
* Point in Time with search\_after
* **The**from**and**size**parameters**

پارامتر from شماره سندی است که می‌خواهید نتایج را از آنجا شروع کنید. پارامتر اندازه تعداد نتایجی است که می خواهید نشان دهید. آنها با هم به شما اجازه می دهند زیرمجموعه ای از نتایج جستجو را برگردانید. به عنوان مثال، اگر مقدار اندازه 10 و مقدار from 0 باشد، 10 نتیجه اول را مشاهده می کنید. اگر مقدار از را به 10 تغییر دهید، 10 نتیجه بعدی را مشاهده خواهید کرد (زیرا نتایج دارای شاخص صفر هستند). بنابراین اگر می خواهید نتایجی را ببینید که از نتیجه 11 شروع می شود، از باید 10 باشد.

****

* **Scroll search**

پارامترهای from و size به شما این امکان را می دهند که نتایج جستجوی خود را صفحه بندی کنید اما با محدودیت 10000 نتیجه در یک زمان. اگر نیاز دارید حجم داده های بزرگتر از 1 PB را از مثلاً یک کار یادگیری ماشینی درخواست کنید، به جای آن از عملیات اسکرول استفاده کنید. عملیات اسکرول به شما امکان می دهد تا تعداد نامحدودی از نتایج را درخواست کنید. برای استفاده از عملیات اسکرول، یک پارامتر اسکرول را به سرفصل درخواست اضافه کنید با یک زمینه جستجو که به OpenSearch می‌گوید چه مدت باید به پیمایش ادامه دهید. این زمینه جستجو باید به اندازه کافی طولانی باشد تا بتواند یک دسته از نتایج را پردازش کند. برای تنظیم تعداد نتایجی که می خواهید برای هر دسته برگردانده شود، از پارامتر اندازه استفاده کنید.

* **The**search\_after**parameter**

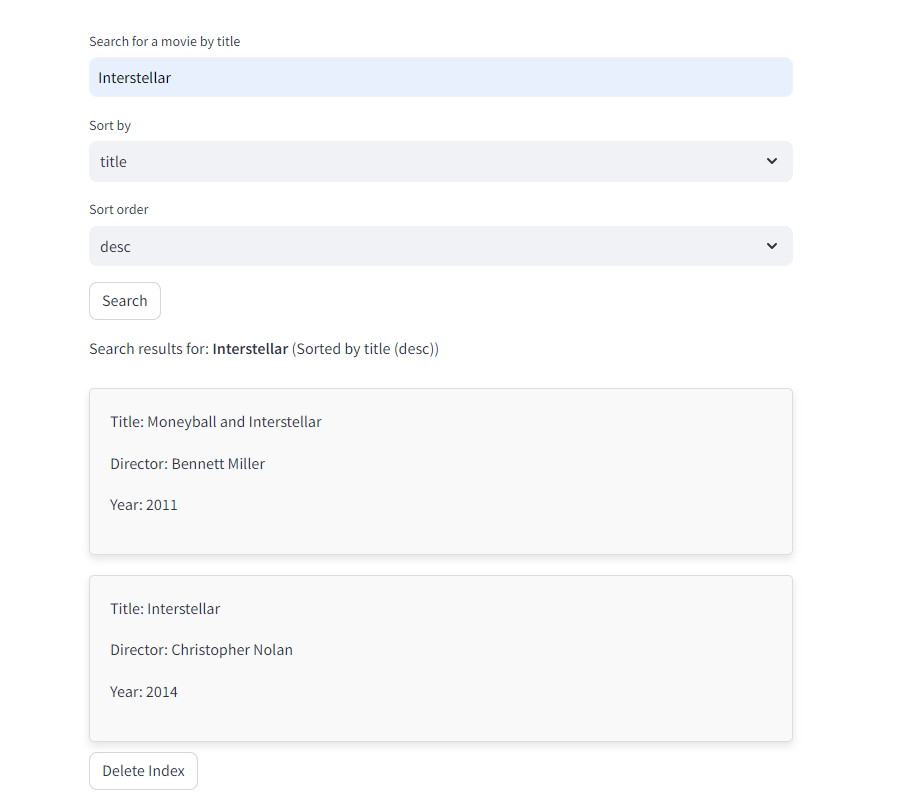
پارامتر search\_after یک مکان‌نمای زنده ارائه می‌کند که از نتایج صفحه قبلی برای به دست آوردن نتایج صفحه بعدی استفاده می‌کند. این شبیه به عملیات اسکرول است که به معنای پیمایش بسیاری از پرس و جوها به صورت موازی است.

* **Point in Time with**search\_after

نقطه در زمان (PIT) با search\_after روش صفحه‌بندی ترجیحی در OpenSearch است، مخصوصاً برای صفحه‌بندی عمیق. محدودیت‌های همه روش‌های دیگر را دور می‌زند زیرا روی مجموعه داده‌ای کار می‌کند که در زمان ثابت است، به یک پرس و جو محدود نمی‌شود، و از صفحه‌بندی ثابت به جلو و عقب پشتیبانی می‌کند.

1. **Sort results**

مرتب سازی به کاربران شما اجازه می دهد تا نتایج را به روشی که برای آنها معنادارتر است مرتب کنند.به طور پیش فرض، پرس و جوهای متن کامل نتایج را بر اساس امتیاز مربوطه مرتب می کنند. شما می توانید با تنظیم پارامتر ترتیب به صعودی یا نزولی، نتایج را بر اساس هر مقدار فیلد به ترتیب صعودی یا نزولی مرتب کنید.

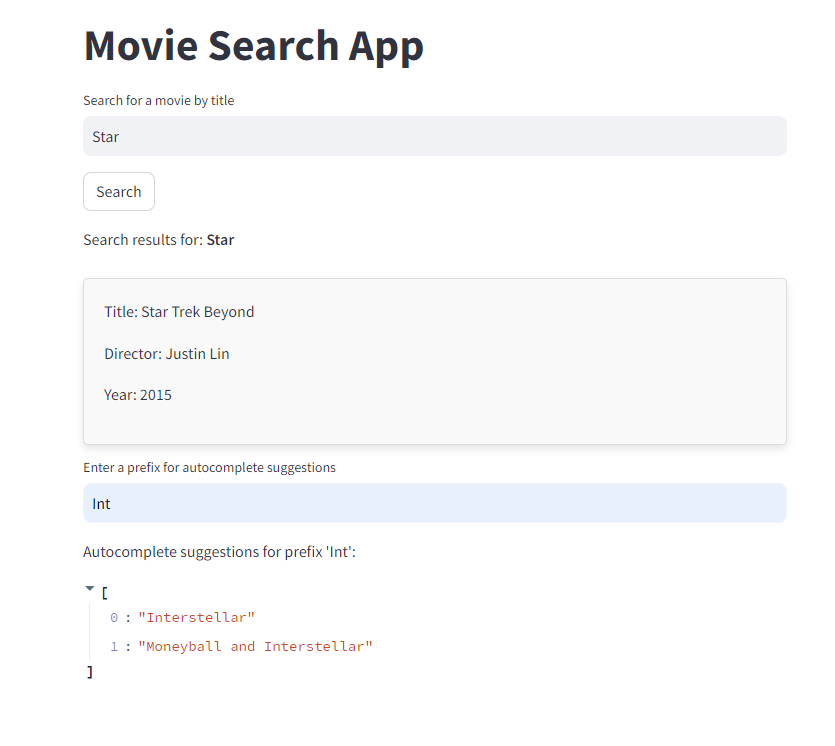


1. **Autocomplete functionality**

تکمیل خودکار پیشنهادها را هنگام تایپ به کاربران نشان می دهد. برای مثال، اگر کاربری «pop» را تایپ کند، OpenSearch پیشنهاداتی مانند «پاپ کورن» یا «popsicles» ارائه می‌کند. این آن‌ها را سریع‌تر به یک عبارت جستجوی احتمالی هدایت می کند.

OpenSearch به شما امکان می دهد تکمیل خودکار طراحی کنید که با هر بار زدن کلید به روز می شود تکمیل خودکار را با استفاده از یکی از روش های زیر پیاده سازی کنید:

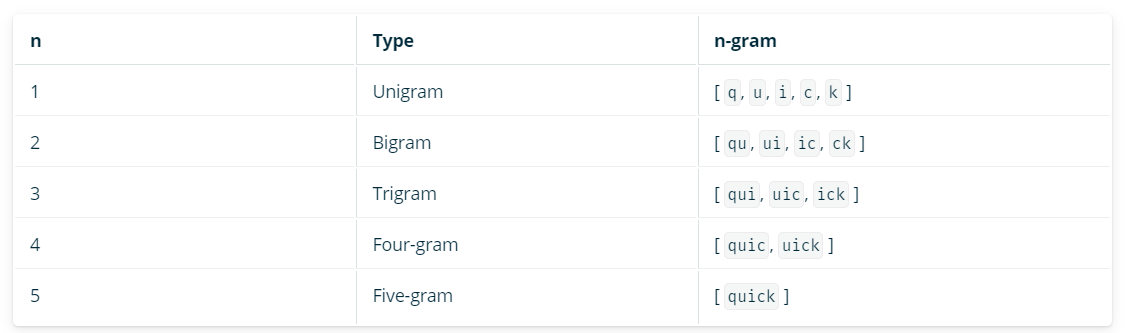
* [Prefix matching](https://opensearch.org/docs/latest/search-plugins/searching-data/autocomplete/#prefix-matching)
* [Edge n-gram matching](https://opensearch.org/docs/latest/search-plugins/searching-data/autocomplete/#edge-n-gram-matching)
* [Search as you type](https://opensearch.org/docs/latest/search-plugins/searching-data/autocomplete/#search-as-you-type)
* [Completion suggesters](https://opensearch.org/docs/latest/search-plugins/searching-data/autocomplete/#completion-suggester)
* **Prefix matching**

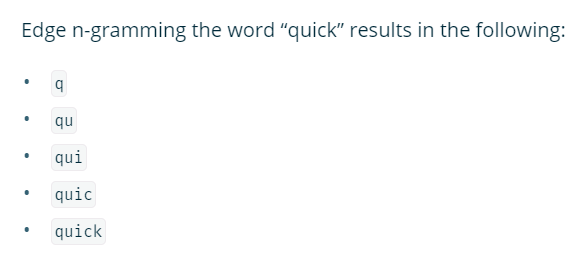


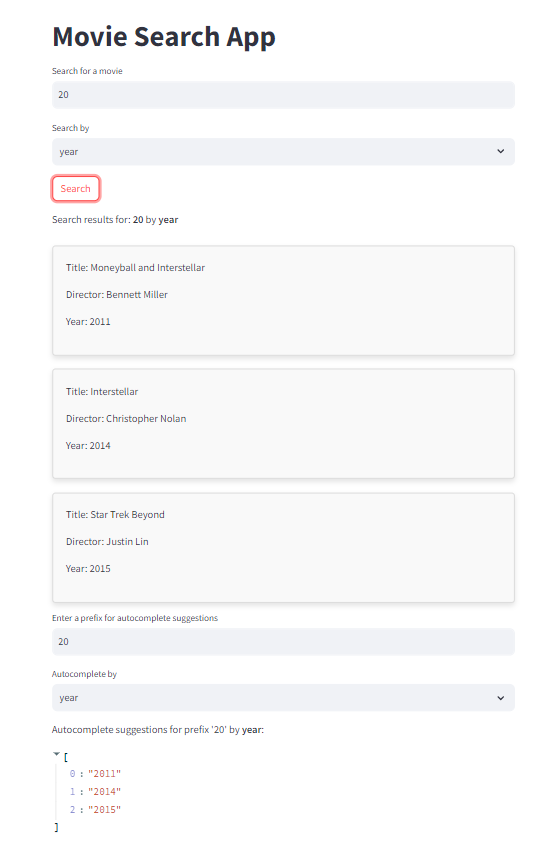
* **Edge n-gram matching**

در طول ایندکس سازی، n-gram یک کلمه را به دنباله ای از n کاراکتر تقسیم می‌کند تا از جستجوی سریع تر عبارات جستجوی جزئی پشتیبانی کند.

If you n-gram the word “quick,” the results depend on the value of n.



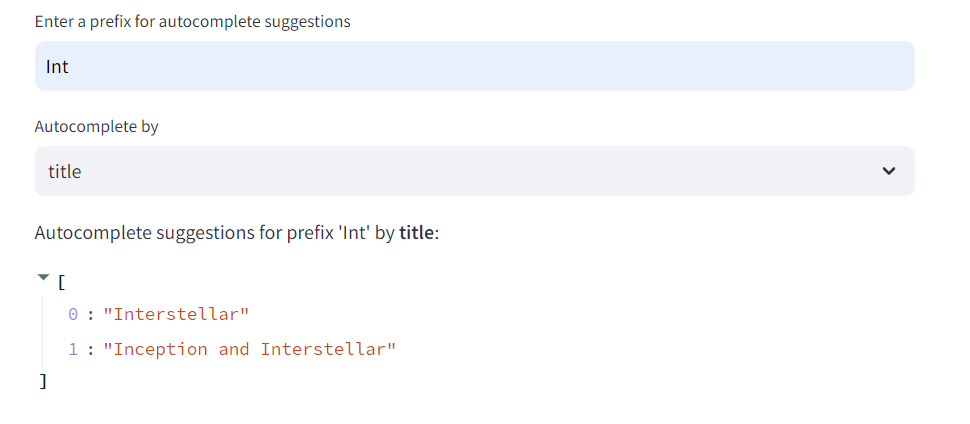




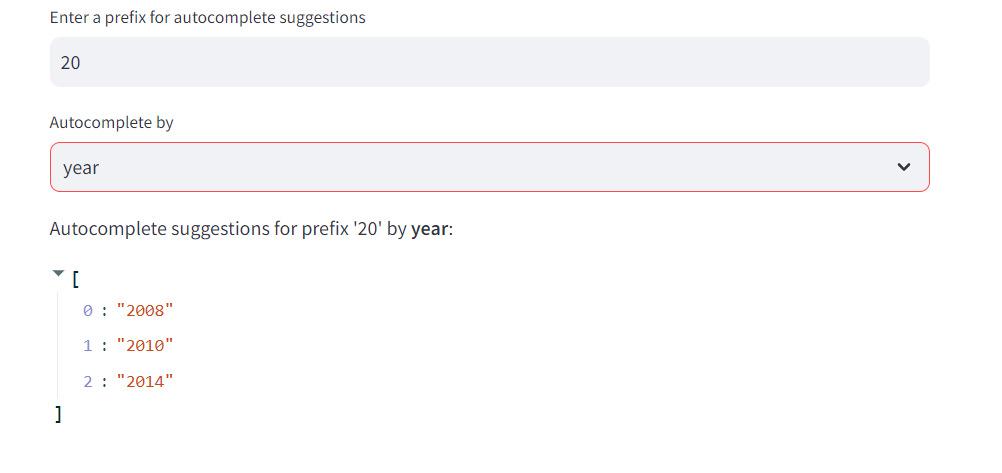
* **Completion suggester**

پیشنهاد تکمیل: لیستی از پیشنهادات را می‌پذیرد و آنها را در یک مبدل حالت محدود (FST) می‌سازد، یک ساختار داده بهینه‌شده که اساساً یک نمودار است. این ساختار داده در حافظه زندگی می کند و برای جستجوی سریع پیشوندها بهینه شده است. برای کسب اطلاعات بیشتر در مورد FST ها، به ویکی پدیا مراجعه کنید.

همانطور که کاربر تایپ می کند، پیشنهاد دهنده تکمیل در نمودار FST هر بار یک کاراکتر در طول مسیر منطبق حرکت می کند. پس از تمام شدن ورودی کاربر، انتهای باقی مانده را بررسی می کند تا لیستی از پیشنهادات را تهیه کند.



* **Search as you type**



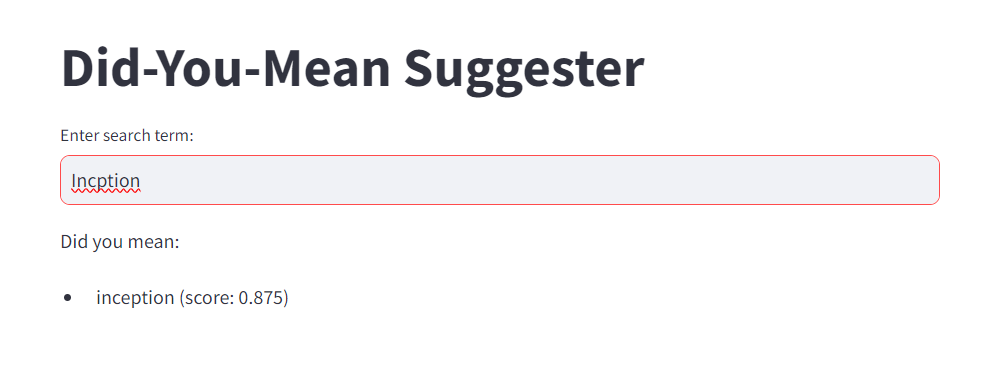
1. Did-you-mean

پیشنهاد دهنده آیا منظور شما اصلاحات پیشنهادی برای عبارت های جستجوی غلط املایی را نشان می دهد.

به عنوان مثال، اگر کاربری "fliud" را تایپ کند، OpenSearch یک عبارت جستجوی اصلاح شده مانند "fluid" را پیشنهاد میکند. در بحث درسی ما Miss\_Spelling است.

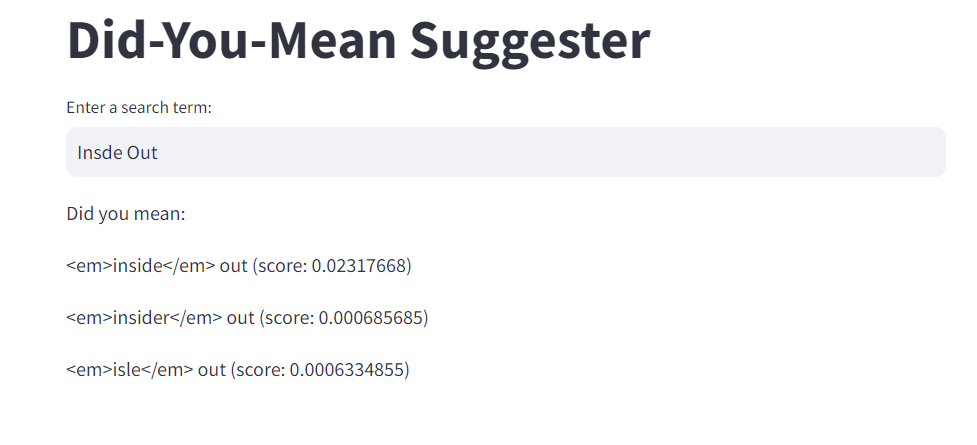
ما می توانیم پیشنهاد دهنده را با استفاده از یکی از روش های زیر پیاده سازی کنیم:

* برای پیشنهاد اصلاحات برای تک تک کلمات از پیشنهاد دهنده اصطلاح استفاده کنیم.
* از پیشنهاد دهنده عبارت برای پیشنهاد اصلاحات برای عبارات استفاده کنیم.
* پیشنهاد اصلاحات برای تک تک کلمات



* پیشنهاد اصلاحات برای عبارات

عبارت پیشنهاد دهنده شبیه اصطلاح پیشنهادگر است، با این تفاوت که از مدل های زبان n-gram برای پیشنهاد عبارات کامل به جای کلمات جداگانه استفاده می کند.

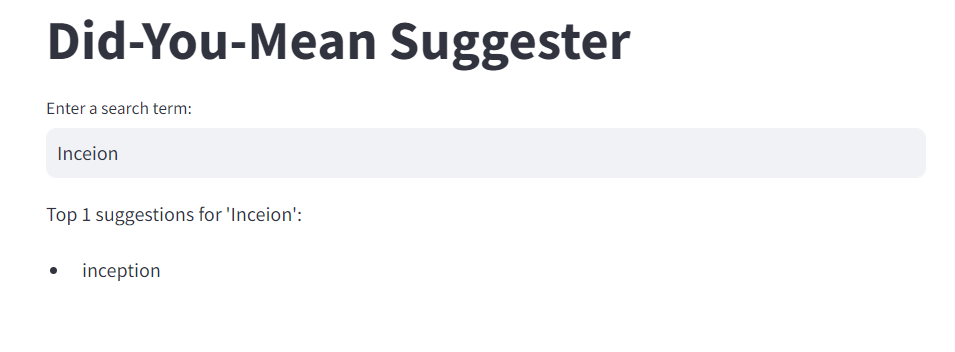


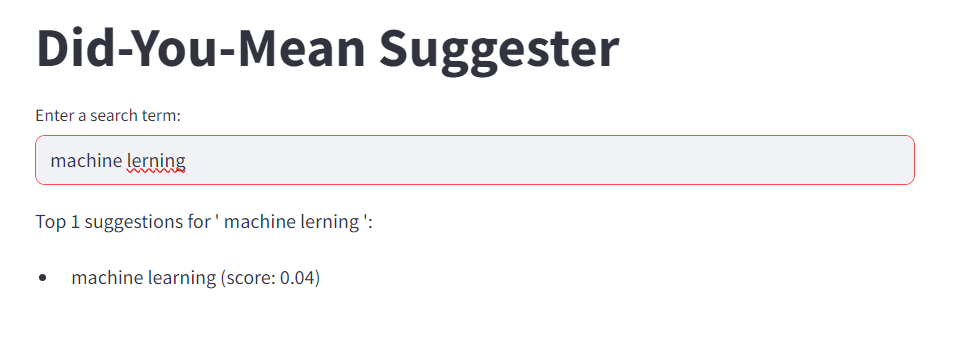
**Collate field**

فرض کنید که شما دارید از این سیستم برای پیشنهاد دادن عناوین فیلم استفاده می‌کنید. اگر کاربر عبارت "هری پاتر" را وارد کند، سیستم با استفاده از phrase suggester از OpenSearch برای پیشنهاد دادن عبارات مشابه استفاده می‌کند. به عنوان مثال، اگر کاربر "هری پتر" را وارد کند، سیستم پیشنهاداتی مانند "هری پاتر و فرش فرنگی"، "هری پاتر و جام آتش" و "هری پاتر و سنگ جادو" را می‌تواند ارائه دهد. برای هر یک از این پیشنهادها، سیستم یک پرسمان match\_phrase بر روی عنوان فیلم اجرا می‌کند تا اطمینان حاصل کند که واقعاً فیلمی با همین عنوان در دیتابیس موجود است و نتیجه ای دارد. در اینجا، فیچر collate به کمک می‌آید تا اطمینان حاصل شود که هر پیشنهاد ارائه شده حداقل یک نتیجه در جستجوی اصلی دارد، به این ترتیب که فقط پیشنهادهایی را نمایش می‌دهد که نتیجه‌ای دارند.



**Candidate generators**



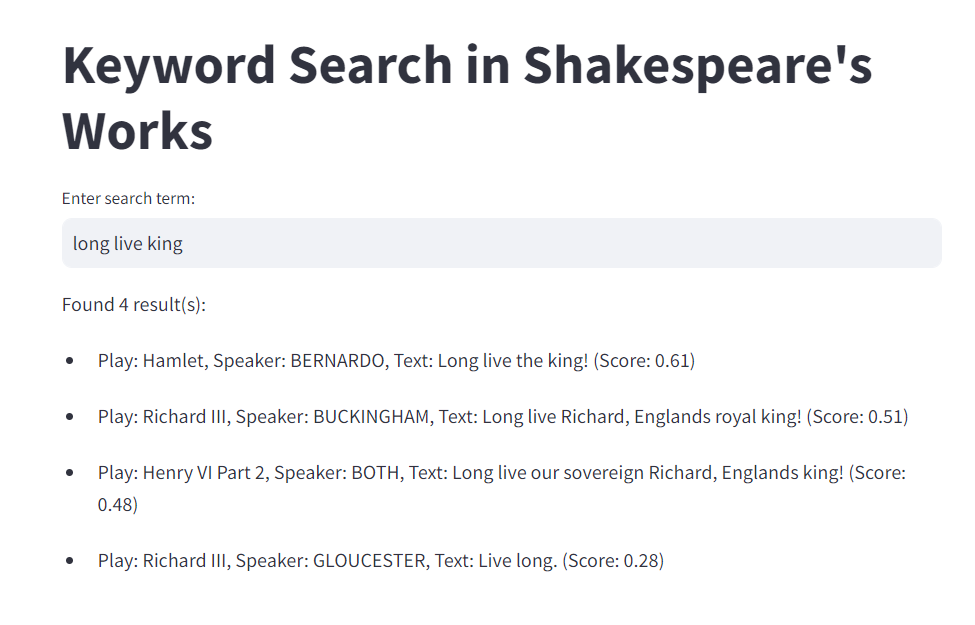


**Keyword search**

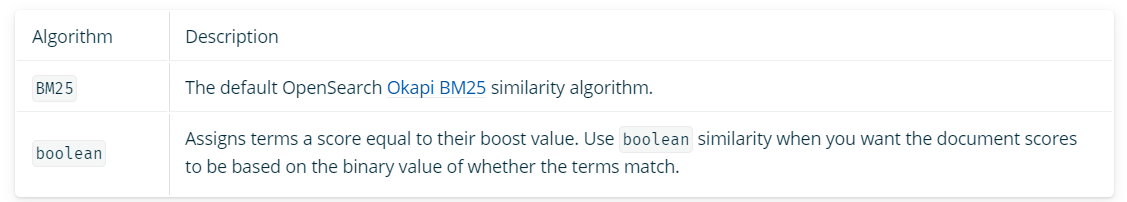
به طور پیش فرض، OpenSearch امتیازات اسناد را با استفاده از الگوریتم Okapi BM25 محاسبه می کند. BM25 یک الگوریتم مبتنی بر کلمه کلیدی است که جستجوی لغوی را برای کلماتی که در پرس و جو ظاهر می شوند انجام می دهد. هنگام تعیین ارتباط یک سند، BM25 فرکانس مدت/فرکانس سند معکوس (TF/IDF) را در نظر می گیرد:

فراوانی اصطلاح تصریح می کند که اسنادی که عبارت جستجو در آنها بیشتر دیده می شود، مرتبط ترند.

بسامد معکوس سند به کلماتی که معمولاً در همه اسناد موجود در مجموعه ظاهر می شوند (به عنوان مثال، مقالاتی مانند "the") وزن کمتری می دهد.



**Similarity algorithms**



**Specifying similarity**

شما می توانید الگوریتم شباهت را در پارامتر شباهت هنگام پیکربندی نگاشتها مشخص کنید.به عنوان مثال، کوئری زیر شباهت بولی را برای boolean\_field مشخص می کند. به bm25\_field شباهت پیش فرض BM25 اختصاص داده شده است:

PUT /testindex

{

"mappings": {

"properties": {

"bm25\_field": {

"type": "text"

},

"boolean\_field": {

"type": "text",

"similarity": "boolean"

}

}

}

}

**Configuring BM25 similarity**

PUT /testindex

{

"settings": {

"index": {

"similarity": {

"custom\_similarity": {

"type": "BM25",

"k1": 1.2,

"b": 0.75,

"discount\_overlaps": "true"

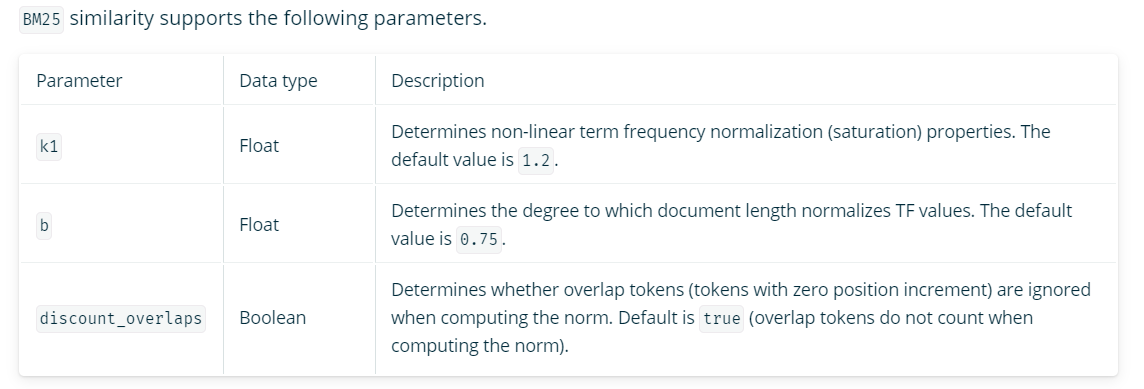
}

}

}

}

}



**جستجوی مبتنی بر یادگیری ماشینی (ML)**

* جستجوی برداری
* جستجوی k-NN: k-نزدیکترین همسایه یک عبارت جستجو را در میان شاخصی از بردارها جستجو می‌کند
* **k-NN search**

افزونه k-NN که مخفف کلمه k-nearest همسایه است، کاربران را قادر می سازد تا k-نزدیک ترین همسایه ها را در یک نقطه پرس و جو در یک شاخص از بردارها جستجو کنند. برای تعیین همسایه‌ها، می‌توانید فاصله (تابع فاصله) را که می‌خواهید برای اندازه‌گیری فاصله بین نقاط استفاده کنید، مشخص کنید.

موارد استفاده شامل توصیه‌ها (به عنوان مثال، ویژگی «آهنگ‌های دیگری که ممکن است دوست داشته باشید» در یک برنامه موسیقی)، تشخیص تصویر، و تشخیص تقلب می‌شود. برای اطلاعات پس زمینه بیشتر در مورد جستجوی k-NN، به ویکی پدیا مراجعه کنید.

این افزونه از سه روش مختلف برای به دست آوردن k نزدیکترین همسایه ها از شاخص بردارها پشتیبانی می کند:

* **Approximate k-NN: روش اول از یک رویکرد تقریبی نزدیکترین همسایه استفاده می کند - از یکی از چندین الگوریتم برای برگرداندن تقریبی k-نزدیک ترین همسایه ها به بردار پرس و جو استفاده می کند. معمولاً این الگوریتم‌ها در ازای مزایای عملکردی مانند تأخیر کمتر، ردپای حافظه کوچک‌تر و جستجوی مقیاس‌پذیرتر، سرعت فهرست‌سازی و دقت جستجو را قربانی می‌کنند. برای کسب اطلاعات بیشتر در مورد الگوریتم ها، به مستندات nmslib و faiss مراجعه کنید. این الگوریتم بهترین انتخاب برای جستجوهای روی شاخص های بزرگ (یعنی صدها هزار بردار یا بیشتر) است که به تأخیر کم نیاز دارند. اگر می‌خواهید قبل از جستجوی k-NN، فیلتری را روی شاخص اعمال کنید، نباید از k-NN تقریبی استفاده کنید، که تعداد بردارهای مورد جستجو را به شدت کاهش می‌دهد. در این حالت باید از روش امتیازدهی اسکریپت یا پسوندهای Painless استفاده کنید. برای جزئیات بیشتر در مورد این روش، از جمله توصیه هایی برای استفاده از موتور، به جستجوی تقریبی k-NN مراجعه کنید.**
* **Script Score k-NN:** **روش دوم عملکرد امتیازدهی اسکریپت OpenSearch را برای اجرای یک نیروی brute force، جستجوی دقیق k-NN در فیلدهای "knn\_vector" یا فیلدی هایی که می توانند اشیاء باینری را نشان دهند، گسترش می دهد. با این رویکرد، می‌توانید جستجوی k-NN را بر روی زیرمجموعه‌ای از بردارها در فهرست خود اجرا کنید (گاهی اوقات به عنوان جستجوی پیش فیلتر نیز شناخته می‌شود). از این رویکرد برای جست‌وجو در بخش‌های کوچک‌تر اسناد یا زمانی که به یک فیلتر اولیه نیاز است، استفاده کنید. استفاده از این رویکرد در شاخص های بزرگ ممکن است منجر به تأخیر زیاد شود. برای جزئیات بیشتر در مورد این روش، دقیق k-NN با اسکریپت امتیازدهی را ببینید.**
* **Painless extensions: روش سوم توابع فاصله را به عنوان اکستنشن های بدون درد اضافه می کند که می توانید در ترکیب های پیچیده تر استفاده کنید. مشابه امتیاز اسکریپت k-NN، می توانید از این روش برای انجام یک جستجوی brute force، k-NN دقیق در یک شاخص استفاده کنید، که از پیش فیلتر کردن نیز پشتیبانی می کند. این رویکرد در مقایسه با امتیاز اسکریپت k-NN عملکرد پرس و جو کمی کندتر دارد. اگر مورد استفاده شما نیاز به سفارشی سازی بیشتری نسبت به امتیاز نهایی دارد، باید از این رویکرد در Script Score k-NN استفاده کنید. برای جزئیات بیشتر در مورد این روش، توابع اسکریپت نویسی بدون درد را ببینید.**

**به طور کلی، برای مجموعه داده‌های بزرگ‌تر، معمولاً باید روش تقریبی نزدیک‌ترین همسایه را انتخاب کنید زیرا مقیاس قابل توجهی بهتر است. برای مجموعه داده های کوچکتر، جایی که ممکن است بخواهید فیلتر اعمال کنید، باید رویکرد امتیازدهی سفارشی را انتخاب کنید. اگر مورد استفاده پیچیده تری دارید که در آن باید از تابع فاصله به عنوان بخشی از روش امتیازدهی آنها استفاده کنید، باید از رویکرد اسکریپت نویسی بدون درد استفاده کنید.**

**Exact k-NN with scoring script**

PUT my-knn-index-1

{

"mappings": {

"properties": {

"my\_vector1": {

"type": "knn\_vector",

"dimension": 2

},

"my\_vector2": {

"type": "knn\_vector",

"dimension": 4

}

}

}

}

POST \_bulk

{ "index": { "\_index": "my-knn-index-1", "\_id": "1" } }

{ "my\_vector1": [1.5, 2.5], "price": 12.2 }

{ "index": { "\_index": "my-knn-index-1", "\_id": "2" } }

{ "my\_vector1": [2.5, 3.5], "price": 7.1 }

{ "index": { "\_index": "my-knn-index-1", "\_id": "3" } }

{ "my\_vector1": [3.5, 4.5], "price": 12.9 }

{ "index": { "\_index": "my-knn-index-1", "\_id": "4" } }

{ "my\_vector1": [5.5, 6.5], "price": 1.2 }

{ "index": { "\_index": "my-knn-index-1", "\_id": "5" } }

{ "my\_vector1": [4.5, 5.5], "price": 3.7 }

{ "index": { "\_index": "my-knn-index-1", "\_id": "6" } }

{ "my\_vector2": [1.5, 5.5, 4.5, 6.4], "price": 10.3 }

{ "index": { "\_index": "my-knn-index-1", "\_id": "7" } }

{ "my\_vector2": [2.5, 3.5, 5.6, 6.7], "price": 5.5 }

{ "index": { "\_index": "my-knn-index-1", "\_id": "8" } }

{ "my\_vector2": [4.5, 5.5, 6.7, 3.7], "price": 4.4 }

{ "index": { "\_index": "my-knn-index-1", "\_id": "9" } }

{ "my\_vector2": [1.5, 5.5, 4.5, 6.4], "price": 8.9 }

GET my-knn-index-1/\_search

{

"size": 4,

"query": {

"script\_score": {

"query": {

"match\_all": {}

},

"script": {

"source": "knn\_score",

"lang": "knn",

"params": {

"field": "my\_vector2",

"query\_value": [2.0, 3.0, 5.0, 6.0],

"space\_type": "cosinesimil"

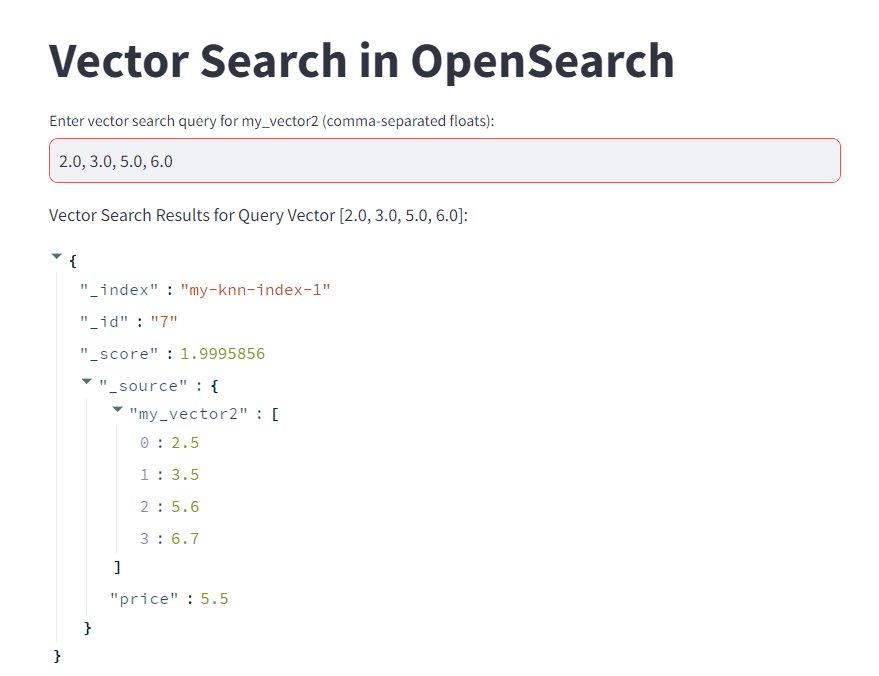
}

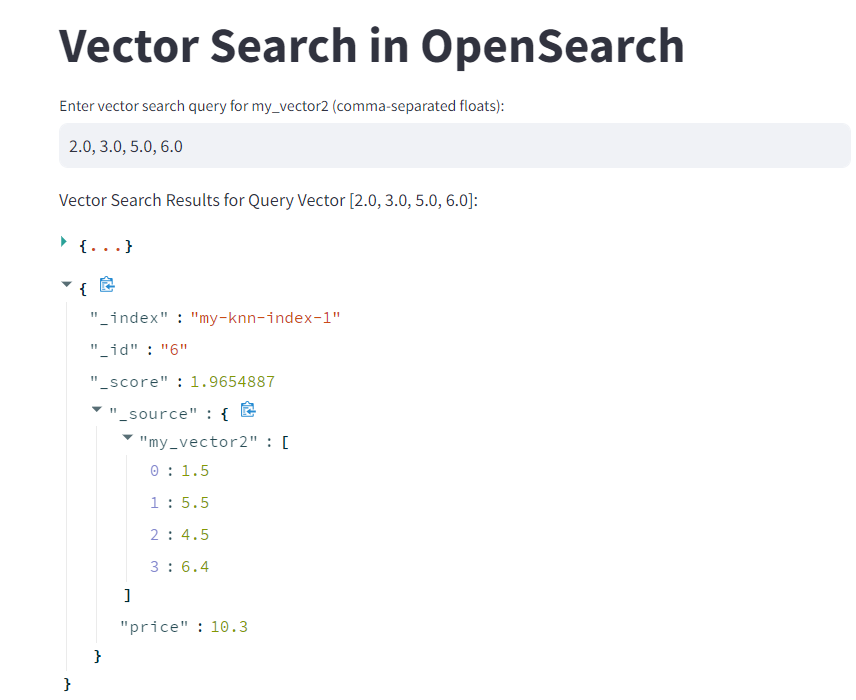
}

}

}}

**خروجی کد:**

****

****

****

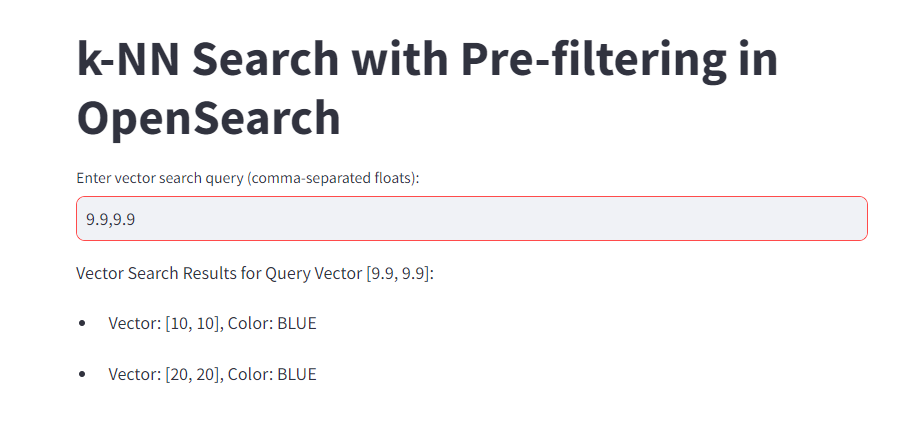
****

All parameters are required.

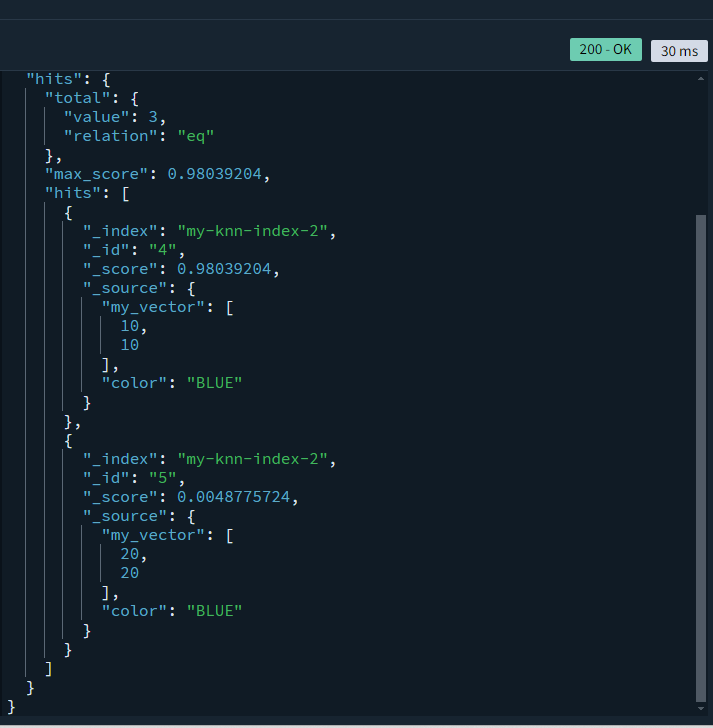
* lang is the script type. This value is usually painless, but here you must specify knn.
* source is the name of the script, knn\_score.

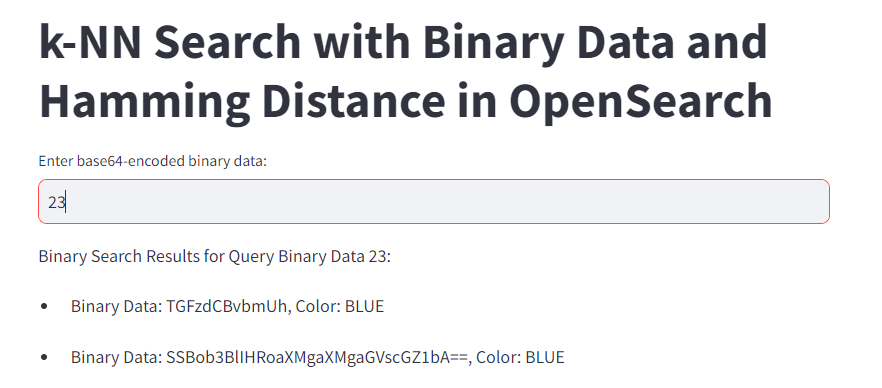
This script is part of the k-NN plugin and isn’t available at the standard \_scripts path. A GET request to \_cluster/state/metadata doesn’t return it, either.

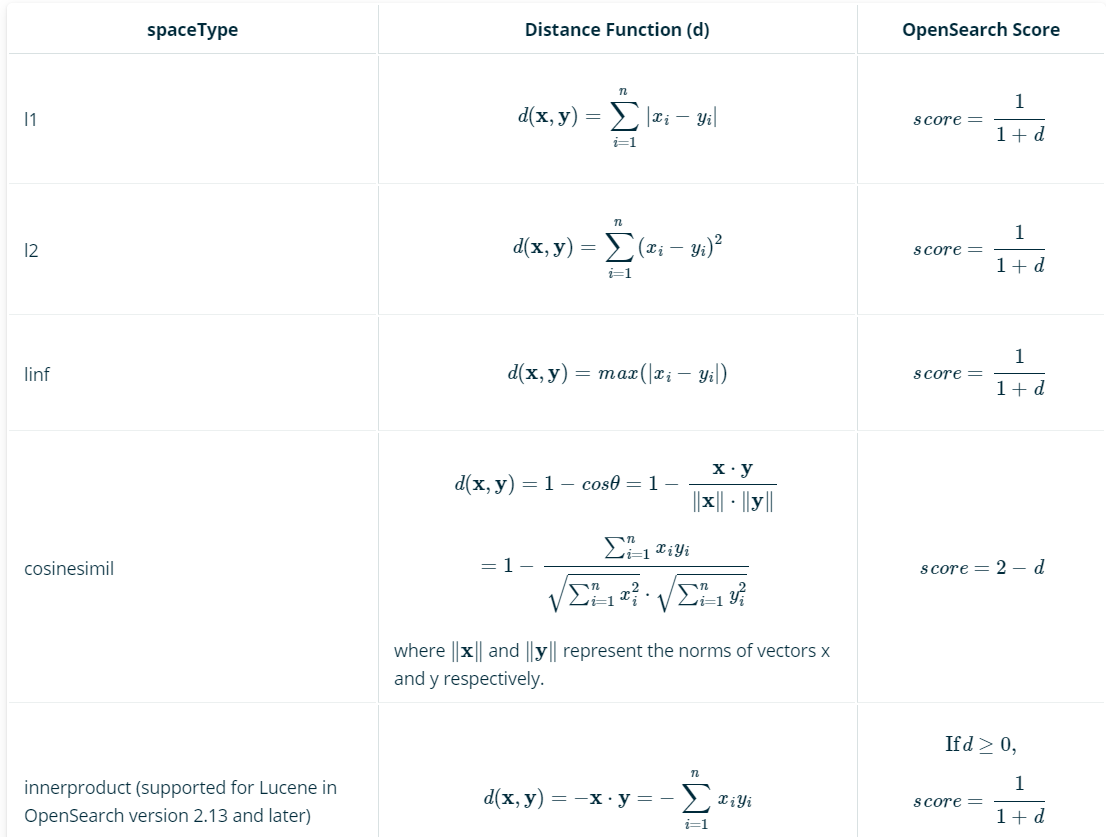
* field is the field that contains your vector data.
* query\_value is the point you want to find the nearest neighbors for. For the Euclidean and cosine similarity spaces, the value must be an array of floats that matches the dimension set in the field’s mapping. For Hamming bit distance, this value can be either of type signed long or a base64-encoded string (for the long and binary field types, respectively).
* space\_type corresponds to the distance function. See the [spaces section](https://opensearch.org/docs/latest/search-plugins/knn/knn-score-script/#spaces).

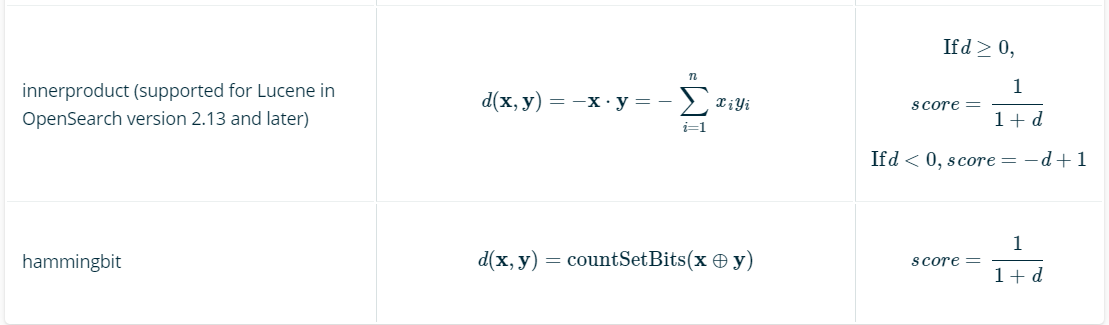


خروجی در open search چک شد.



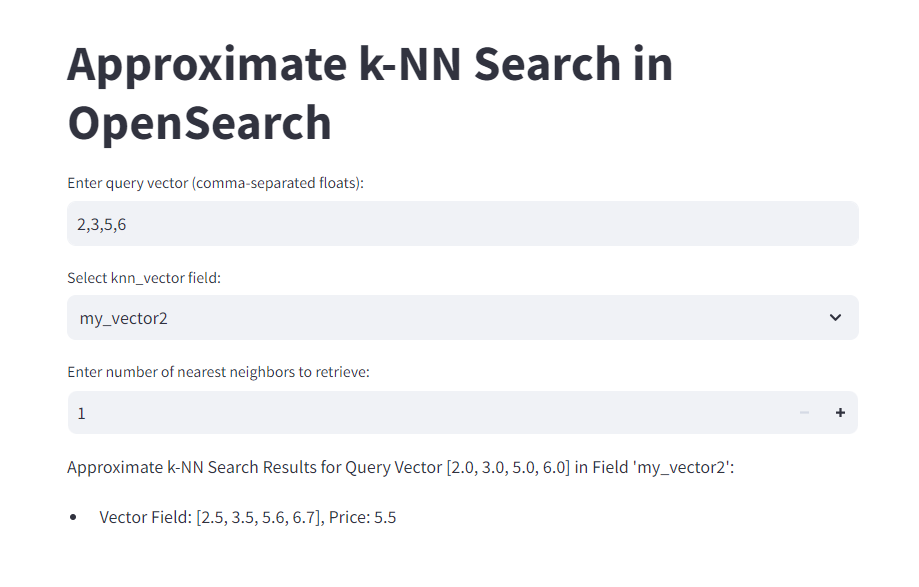


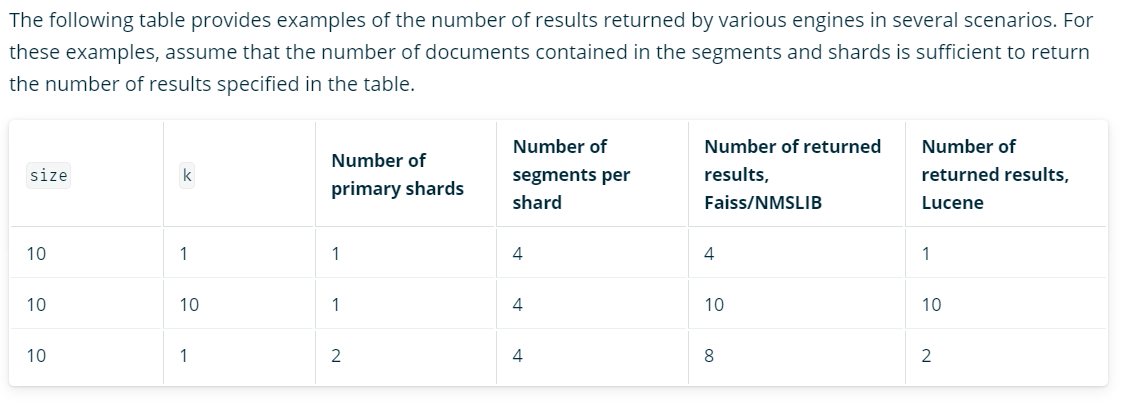




**Approximate k-NN search**

جستجوی تقریبی k-نزدیکترین همسایگان (k-NN) تکنیکی است که در بازیابی اطلاعات و یادگیری ماشینی برای یافتن k نقاط در یک مجموعه داده که نزدیک‌ترین نقطه به یک نقطه پرس و جو هستند، استفاده می‌شود. برخلاف k-NN دقیق، که فاصله ها یا شباهت ها را به طور کامل در تمام نقاط محاسبه می کند، k-NN تقریبی از تکنیک هایی استفاده می کند که دقت را فدای کارایی می کند. این مبادله به زمان‌های پرس و جو سریع‌تر اجازه می‌دهد، به ویژه در مجموعه داده‌های مقیاس بزرگ که جستجوی دقیق k-NN ممکن است از نظر محاسباتی گران شود.



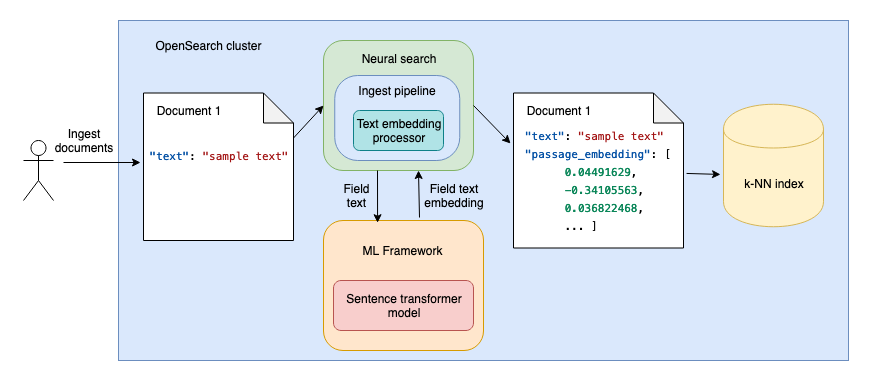


* جستجوی عصبی: جستجوی عصبی تولید جاسازی‌های برداری را در زمان مصرف و جستجوی آنها در زمان جستجو را تسهیل می‌کند. جستجوی عصبی به ما امکان می‌دهد مدل های ML را در جستجوی خود ادغام کنیم و به عنوان چارچوبی برای اجرای سایر روش های جستجو عمل می کند. روش‌های جستجوی زیر بر روی جستجوی عصبی ساخته شده‌اند:
* جستجوی معنایی: معنای کلمات را در زمینه جستجو در نظر می گیرد. از بازیابی متراکم بر اساس مدل های جاسازی متن برای جستجوی داده های متنی استفاده می کند.
* جستجوی چندوجهی: از مدل‌های تعبیه چند وجهی برای جستجوی داده‌های متن و تصویر استفاده می‌کند.
* جستجوی پراکنده عصبی: از بازیابی پراکنده بر اساس مدل‌های جاسازی پراکنده برای جستجوی داده‌های متنی استفاده می‌کند.
* جستجوی ترکیبی: جستجوی سنتی و جستجوی برداری را برای بهبود ارتباط جستجو ترکیب می‌کند.
* جستجوی مکالمه: یک جستجوی مولد تقویت شده با بازیابی را پیاده سازی می کند.

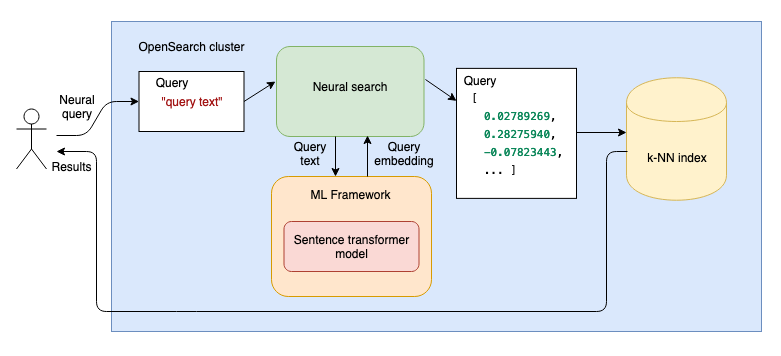
**Terminology**

*Neural search*: Facilitates vector search at ingestion time and at search time:

* At ingestion time, neural search uses language models to generate vector embeddings from the text fields in the document. The documents containing both the original text field and the vector embedding of the field are then indexed in a k-NN index, as shown in the following diagram.



* At search time, when you then use a *neural query*, the query text is passed through a language model, and the resulting vector embeddings are compared with the document text vector embeddings to find the most relevant results, as shown in the following diagram.



* *Semantic search*: Employs neural search in order to determine the intention of the user’s query in the search context, thereby improving search relevance.
* *Hybrid search*: Combines semantic and keyword search to improve search relevance.

در ادامه با اجرای مسیر زیر سرچ عصبی انجام خواهیم داد:

**Step 1: Set up an ML language model**

import pandas as pd

from opensearchpy import OpenSearch

import streamlit as st

from sentence\_transformers import SentenceTransformer

model = SentenceTransformer('intfloat/multilingual-e5-large')

**Adding Documents from Dataset**

df = pd.read\_csv('imdb\_top\_1000.csv')

**Creating vectors**

df["Vector"] = df["Series\_Title"].apply(lambda x: model.encode(x))

**Creating indexing**

client = OpenSearch(

    hosts=[{'host': 'localhost', 'port': 9200}],

    http\_auth=('admin', 'admin')

)

index\_name = 'imdb\_movie'

def create\_index\_with\_knn\_vector(client, index\_name):

    if client.indices.exists(index=index\_name):

        client.indices.delete(index=index\_name)

    index\_body = {

        "settings": {

            "index.knn": True

        },

        "mappings": {

            "properties": {

                "vector": {

                    "type": "knn\_vector",

                    "dimension": 1024

                },

                "Series\_Title": {

                    "type": "text"

                },

                "Released\_Year": {

                    "type": "text"

                },

                "Runtime": {

                    "type": "keyword"

                },

                "Genre": {

                    "type": "text"

                },

                "IMDB\_Rating": {

                    "type": "float"

                },

                "Overview": {

                    "type": "text"

                },

                "Director": {

                    "type": "text"

                },

                "Star1": {

                    "type": "text"

                },

                "Star2": {

                    "type": "text"

                },

                "Star3": {

                    "type": "text"

                },

                "Star4": {

                    "type": "text"

                },

                "No\_of\_Votes": {

                    "type": "integer"

                }

            }

        }

    }

    response = client.indices.create(index=index\_name, body=index\_body)

create\_index\_with\_knn\_vector(client, index\_name)

**indexing Documents**

def index\_documents():

    for idx, row in df.iterrows():

        document = {

            'Series\_Title': row['Series\_Title'],

            'vector': row['Vector'],

            'Released\_Year': row['Released\_Year'],

            'Runtime': row['Runtime'],

            'Genre': row['Genre'],

            'IMDB\_Rating': row['IMDB\_Rating'],

            'Overview': row['Overview'],

            'Director': row['Director'],

            'Star1': row['Star1'],

            'Star2': row['Star2'],

            'Star3': row['Star3'],

            'Star4': row['Star4'],

            'No\_of\_Votes': row['No\_of\_Votes']

        }

        response = client.index(index=index\_name, body=document)

index\_documents()

NLP For Query

from sentence\_transformers import SentenceTransformer

model = SentenceTransformer('intfloat/multilingual-e5-large')

index\_name = 'imdb\_movie'

client = OpenSearch(

    hosts=[{'host': 'localhost', 'port': 9200}],

    http\_auth=('admin', 'admin')

)

def search\_movies(query):

    query\_vector = list(model.encode(query))

    query = {

            "size": 5,

            "query": {

                    "script\_score": {

                        "query": {"match\_all": {}},

                        "script": {

                            "source": "knn\_score",

                            "lang": "knn",

                            "params": {

                                "field": "vector",

                                "query\_value": query\_vector,

                                "space\_type": "cosinesimil"

                            }

                        }

                    }

                }

            }

    response = client.search(index=index\_name, body=query)

    return response

def main():

    st.title('IMDb Movie Search')

    query = st.text\_input('Enter your search query:')

    if st.button('Search'):

        results = search\_movies(query)

        st.subheader('Search Results:')

        st.markdown("""

        <style>

        .card {

            border: 1px solid #ddd;

            border-radius: 10px;

            padding: 10px;

            margin: 10px 0;

        }

        </style>

        """, unsafe\_allow\_html=True)

        for hit in results['hits']['hits']:

            st.markdown(f"""

            <div class="card">

                <p><b>ID:</b> {hit['\_id']}</p>

                <p><b>Title:</b> {hit['\_source']['Series\_Title']}</p>

                <p><b>Overview:</b> {hit['\_source']['Overview']}</p>

                <p><b>IMDB Rating:</b> {hit['\_source']['IMDB\_Rating']}</p>

                <p><b>Director:</b> {hit['\_source']['Director']}</p>

            </div>

            """, unsafe\_allow\_html=True)

if \_\_name\_\_ == '\_\_main\_\_':

    main()

با استفاده از این دو خط کوعری را به بردار تبدیل می‌کنیم.

From sentence\_transformers import SentenceTransformer

model = SentenceTransformer(‘intfloat/multilingual-e5-large’)

با استفاده از cosinesimil درصد شباهت بردار کوعری با داکیومنت را در پلتفرم مورد نظر بدست آورده و مرتبط ترین سرچ برگردانده می‌شود.

خروجی:

