CS410 – Text Information Systems Fall 2020
Technology Review
sithira2@illinois.edu

Amazon Elasticsearch Service

Introduction

Elasticsearch is a search engine based on Apache Lucene. Amazon's Elasticsearch Service (AES for short) offering provides a scalable, easy-to-deploy, secure and cost-effective way to integrate search capabilities into applications. Users can ingest data such as log files, messages, metrics and documents into AES and perform operations such as indexing,

This tech review will mainly focus on the following topics:

- KNN for Elasticsearch
- Learning to Rank API
- Cross Cluster Search

The outcome of this review will be useful to determine which of the above three areas to select when integrating ES for personalized search in applications.

KNN for Elasticsearch

K-nearest neighbours algorithm lets you find nearest neighbours for a given point in a vector space by using similarity measures such as Cosine similarity or Euclidean distance. In the context of AES, one can simply integrate KNN for ranking searching results by using the Open Distro for Elasticsearch. We could create a new index with KNN with the following configuration.

```
PUT my-index
{
  "settings": {
    "index": {
      "knn": true,
      "knn.space_type": "cosinesimil"
    }
  },
  "mappings": {
    "properties": {
      "my_vector": {
        "type": "knn_vector",
        "dimension": 2
      }
    }
  }
}
```

`knn.space_type` defines the similarity function to be used, which could be either `cosinesimil` (Cosine similarity) or `l2` (Euclidean distance).

Under the `mappings` category, we can define multiple vectors of our choice ("my_vector" in our case). The `type` property defines the vectors we want to represent in a document as an array of floating-point numbers. The `dimension` property can take up to 10K.

For demonstration purposes, we have used Kibana's Dev Tools console to create an index and used the following query to retrieve results.

An example of a query to search in a given index would be similar to following,

```
GET my-index/_search
{
  "size": 2,
  "query": {
    "knn": {
      "my_vector2": {
        "vector": [2, 3, 5, 6],
        "k": 2
      }
    }
  }
}
```

The `k` parameter defines the number of closest neighbours (top k results) we want the ranker to return us. This value can take up to an integer of 10K. As you can see from the results below, it will retrieve the closest two neighbors with prices 5.5 and 8.9 respectively ranked according to their similarity scores.

The resulting response contains the following fields,

```
"hits" : [
  {
    "_id" : "7",
    "_score" : 0.4625572,
    "_source" : {
      "my_vector" : [
        2.5,
        3.5,
        5.6,
        6.7
      ],
      "price" : 5.5
    }
  },
  {
    "_id" : "9",
    "_score" : 0.27558172,
    "_source" : {
      "my_vector" : [
        1.5,
        5.5,
        4.5,
        6.4
      ],
      "price" : 8.9
    }
  }
]
```

Learning to Rank API

Elasticsearch uses BM25 to calculate the relevance scores as the default algorithm under the covers. This, however, does not use any feedback mechanisms. Learning to Rank is an API that lets developers fine-tune the relevance of documents which uses Machine Learning and behavioral data such as click-through data.

To get started with LTR API, we need to provide judgment list, a training dataset and train the model outside of AES. Once this initial setup is in place, we can start building out a feature set.

An example of a feature set is shown below,

```
POST _ltr/_featureset/movie_features
{
  "featureset" : {
      "name" : "movie_features",
      "features" : [
        {
          "name" : "1",
          "params" : [
            "keywords"
          ],
          "template_language" : "mustache",
          "template" : {
            "match" : {
              "title" : "{{keywords}}"
            }
          }
        },
        {
          "name" : "2",
          "params" : [
            "keywords"
          ],
          "template_language" : "mustache",
          "template" : {
            "match" : {
              "overview" : "{{keywords}}"
            }
          }
        }
      ]
    }
}
```

Once we have built the initial feature set, we could combine it with the judgement list we provided to AES. This is known as *Feature Values*. These feature values are essentially relevance scores calculated by BM-25.

To train the dataset, we need to come up with a training dataset. This step should be performed outside AES. Upon creating the training dataset, we could then select the algorithm to build the build. LTR API provides us with the following well-known algorithms.

- Ranklib training
- XGBoost

Uploading the model is done via LTR's out-of-the-box API method called _createmodel. There we pass in the configuration required

An example query to upload a trained model for Ranklib algorithm could be found below:

```
POST _ltr/_featureset/movie_features/_createmodel
{
    "model": {
        "name": "my_ranklib_model",
        "model": {
            "type": "model/ranklib",
            "definition": "## LambdaMART\n
                            ## No. of trees = 1000
                            ## No. of leaves = 10
                            ## No. of threshold candidates =
256
                            ## Learning rate = 0.1
                            ## Stop early = 100

                            <ensemble>
                                <tree id="1" weight="0.1">
                                    <split>
                                        <feature> 2 </feature>
                                        ...
                        "
        }
    }
}
```

Finally, we could perform a regular ES search by choosing our model as shown below,

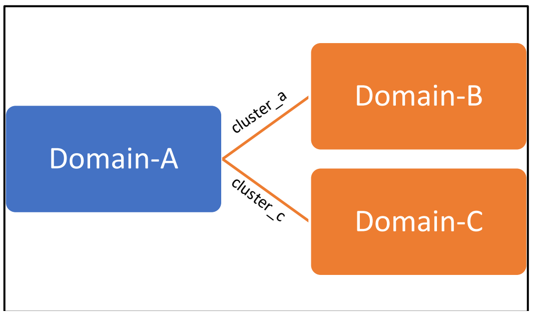
```
POST tmdb/_search
{
  "_source": {
    "includes": ["title", "overview"]
  },
  "query": {
    "multi_match": {
      "query": "rambo",
      "fields": ["title", "overview"]
    }
  },
  "rescore": {
    "query": {
      "rescore_query": {
        "sltr": {
          "params": {
            "keywords": "rambo"
          },
          "model": "my_ranklib_model"
        }
      }
    }
  }
}
```

The resulting response would contain the ranked list of results just along with all the attributes necessary as a default search performed in ES. Overall, Learning to Rank API provides us with great flexibility to train, upload and search by using our models. This could be quite useful for domain-specific applications where default ES search would not yield beneficial results.

Cross Cluster Search

Cross cluster search lets developers to aggregate, search and visualize data scattered across multiple domains. This is performed by setting up a secure connection between the search indexing clusters.

Let us assume that we set up a connection between Domain-A and Domain-B with a connection alias cluster_b and Domain-A to Domain-C an alias cluster_c.



A sample request would be,

```
GET https://src-domain.us-east-
1.es.amazonaws.com/local_index,cluster_b:b_index,cluster_c:c_i
ndex/_search
{
  "query": {
    "match": {
      "user": "domino"
    }
  }
}
```

A sample response would be (some fields removed for brevity),

```
{
  "_clusters": {
    "total": 3,
    "successful": 3,
    "skipped": 0
  },
  "hits": {
    "total": 3,
    "max_score": 1,
    "hits": [
      {
        "_index": "local_index",
        "_type": "_doc",
        "_id": "0",
        "_score": 1,
        "_source": {
          "user": "domino",
          "message": "Let us unite the new mutants",
          "likes": 0
        }
      },
      {
        "_index": "cluster_b:b_index",
        "_type": "_doc",
        "_id": "0",
        "_score": 2,
        "_source": {
          "user": "domino",
          "message": "I'm different",
          "likes": 0
        }
      },
      {
        "_index": "cluster_c:c_index",
        "_type": "_doc",
        "_id": "0",
        "_score": 3,
        "_source": {
          "user": "domino",
          "message": "So am I",
          "likes": 0
        }
      }
    ]
  }
}
```

As we can see, once the initial setup is in place, Elasticsearch service will resolve all the connections and performs the cross-cluster search and aggregate all the results into one JSON object. This lets the developers to saves development time as well as improve the search performance of their applications.

However, this also comes with its fair share of limitations. Developers would not be able to search across managed Elasticsearch instances, not being able to search across multiple AWS regions, versions of Elasticsearch instances being cross-compatible are some of its drawbacks.

Conclusion

In this tech review, we mainly looked at different strategies of integrating Amazon Elasticsearch Service into applications. We first looked at how developers can utilize KNN for their applications by going through a sample use case. Next, we explored Learning to Rank API to understand how we could implement our own model and make use of that to improve the search results returned by AES. Finally, we discussed how cross-cluster search works and improve query performance by combining multiple clusters.

References

1. https://docs.aws.amazon.com/elasticsearch-service/latest/developerguide/learning-to-rank.html

2. https://docs.aws.amazon.com/elasticsearch-service/latest/developerguide/knn.html

3. https://opendistro.github.io/for-elasticsearch/blog/odfe-updates/2020/04/Building-k-Nearest-Neighbor-(k-NN)-Similarity-Search-Engine-with-Elasticsearch/

4. https://docs.aws.amazon.com/elasticsearch-service/latest/developerguide/cross-cluster-search.html

5. https://docs.aws.amazon.com/elasticsearch-service/latest/developerguide/learning-to-rank.html

6. https://aws.amazon.com/about-aws/whats-new/2020/06/announcing-cross-cluster-search-support-for-amazon-elasticsearch-service/

7. https://elasticsearch-learning-to-rank.readthedocs.io/en/latest/training-models.html