Movie Recommender system using Spark and ElasticSearch

Pranav Kumar Sivakumar, Shemal Somil Lalaji

Introduction:

Recommendation engines are one of the most commonly used machine-learning based application that can be easily implemented. But the issue of huge datasets makes it harder to develop one of them. However we can make use of data center techniques to solve this hard problem.

In this project, we propose an approach to provide real-time recommendations, by constructing a large-scale recommender engine using big-data technologies like Apache Spark and ElasticSearch. This approach is highly scalable as it involves frameworks based on distributed systems and cluster computing. So we will be utilizing the related distributed computing services provided by Amazon Web Services (AWS) to accelerate our implementation. By making use of technologies like AWS, ElasticSearch, Spark we can easily manipulate the problem to satisfy the project requirements.

The motivation behind the project is that no big data system has been developed on this topic and so we were inclined to choose this project. This project is very useful for movie goers to choose the movie that satisfy their preferences and so one that is of very high utility.

The data we use in this dataset is the MovieLens dataset. This dataset is very common for projects involving movie recommendation. One of the very useful property of this is there are two different data sources, a small one of size 1 MB and large one of size around 2 GB. So the approach taken by us in this was to first build the model on the smaller dataset and evaluate it and then use AWS for the larger dataset.

The output of this project gives the user a list of movies that he/she will want to see. Say we want to see movies similar to “Men in Black”. The system will recommend sequels of Men in Black such as “Men in Black 2” or even similar action genre movies like “Spider Man”.

Related Work:

Some of the work in this area has been on recommendations based on rank of user who are giving ratings. One of such projects has been by Doug TurnBull who has developed a system using Relevant Search and applied it to MovieLens dataset[3].

Another idea was also to use Elastic Graph for the same task as described above. However none of these systems was a large scale recommender model and thus the list of movies recommended is restricted by the size of the data set under consideration [2].

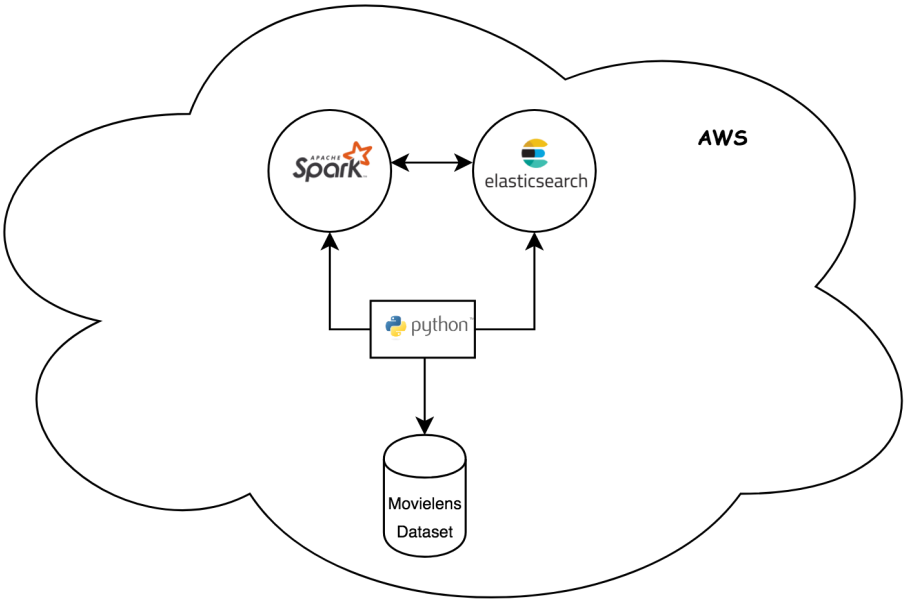
Text

Other work has been making use of ElasticSearch and react.js by Raj Meghpara[1].The project involved use of Reactive Search, a UI components library written in React for building search UIs. The project involved 8 components and there was a nice MovieSearch UI created. The project was also augmented with SSR.

what is SSR?

System Design:

The architecture of the system is best visualized by the following figure:



The architecture of the system tells us all the major components of the project. They are the Spark, ElasticSearch, Python and AWS. We use the MovieLens dataset. For preprocessing and training the data we make use of Spark and load it into the proper format in ElasticSearch for querying. AS we move the vectors, data and model regularly from Spark to ElasticSearch there is a bidirectional link between the two crucial components. We use Python as the language for developing our system. Finally,we use AWS for using our project on larger dataset. The issues or challenges in this project was integrating all of these components in the project.

The technological components are:

Apache Spark: An open-source, fast and general-purpose cluster computing system

ElasticSearch: Open-source search and analytics engine

Python: Python is a programming language that lets you work more quickly and integrate your systems more effectively.

Now coming to the relevant work done on this aspect, we can notice that even though fancier technologies like React.js and Elastic Graph was used none of them will eventually give good results. This is due to the fact that the data used on this was around the size of 1 MB and hence the recommendations were generated from a small sample space of movies.

We have to also consider the model used for generating recommendation in this project was more complicated than the methods used in the subsequent projects. This was one of the biggest challenge of this project. Also this project exploits the power of ElasticSearch and spark MLlib. So it uses a combination of topics from different domains to arrive at the best possible output.

First let us consider the dataset used by us for the project.The dataset is the Movielens 20M dataset that consists of ratings given by a set of users to movies, combined with their metadata information. It was collected and made available by the GroupLens Research.

It is a standard benchmark dataset with 20 million ratings. It includes tag genome data with 12 million relevance scores across 1,100 tags. The dataset describes 5-star ratings and other tags. It contains 20000263 ratings and 465564 tag applications across 27278 movies

The data are contained in six files: genome-scores.csv, genome-tags.csv, links.csv, movies.csv, ratings.csv, and tags.csv.

For data collection, the users were selected at random for inclusion and they had to rate at least 20 movies. User ids are consistent between ratings.csv and tags.csv (i.e., the same id refers to the same user across the two files).

Considering movies column, only those movies with at least one rating or tag are included in the dataset. Movie ids are also consistent between ratings.csv, tags.csv, movies.csv, and links.csv (i.e., the same id refers to the same movie across these four data files).

The data is a static and open source dataset, so there is no need for a developer account as it can be easily accessed. This dataset, as well as the others, are available for free download at http://grouplens.org/datasets/.

We have built a system that gives the recommendation of movies to a user based on the movies that he/she prefers to watch. The following are some of the steps that we have done to finish the desired system:

1. The first part is the data preprocessing which involved some basic text manipulation.

2. After preprocessing the dataset that contained the movie ratings, we trained a collaborative filtering model in Spark using RDD operations and MLlib.

3. We loaded the data in ElasticSearch after designing a schema for the database. We used the Python wrapper for ElasticSearch and Spark ElasticSearch connector for Spark and ElasticSearch. We indexed for mappings for users, movies, and ratings.

4. After successfully loading it, we have trained the Collaborative filtering model on that data. We chose the collaborative filtering method as it is a sort of technique that enables to model the similarity between the items(movies in our project) based on other similar user's preferences.
5. We loaded the trained model's factors into ElasticSearch and performed some Dataframe operations for indexing.

6. The last step consisted of generating the recommendations. To do this, we created some utility functions that handle different tasks such as finding similar movies based on another movie, or based on user queries, fetching movie metadata etc. After getting the results from the queries, we plan to display them effectively.

Let us consider each of the work that we done in more detail:
A.) Data Preprocessing:

The pre-processing of data that we have done in spark is listed below:

1. We have separated the genre of each movie into a list so that we can use them to query in our ElasticSearch more efficiently.
2. Another feature of our data included the property that the titles of the movie also contained the year in which they were released. We have modified this property by separating the year and the title so that we can use them discreetly in the search queries.

B.) Indexing and Loading Data in ElasticSearch:

In ElasticSearch, an "index" is roughly similar to a "database", while a "document type" is roughly similar to a "table" in that database. The schema for a document type is our index mapping. While ElasticSearch supports dynamic mapping, since in our project we know the structure of the data we specify the mapping explicitly.

Also in our project it is imperative to specify a custom analyzer for the field that will hold the recommendation model(that is, the factor vectors). This will ensure the vector-scoring plugin will work correctly.

Later we have written the data from the data set into the ElasticSearch using Spark ElasticSearch connector.

C.) Train Recommender Model on the Data:

For training the data set, we have made use of collaborative filtering and separated the user ratings and movies ratings with a type of technique called ALS (Alternating Least Squares). This type of filtering is very rarely used and is only used here as it is suitable for the MovieLens dataset.The core idea used here is Matrix Factorization (MF) of ratings into users and movies and using the vectors obtained to query for ElasticSearch.

Collaborative filtering is a recommendation approach. It makes the assumption that, if two people share similar preferences, then the things that one of them prefers could be good recommendations to make to the other. In other words, if user A tends to like certain movies, and user B shares some of these preferences with user A, then the movies that user A likes, that user B has not yet seen, may well be movies that user B will also like.In a similar manner , one can

watch grammar mistakes…

think about items as being similar if they tend to be rated highly by the same people, on average.

Hence these models are based on the combined, collaborative preferences and behavior of all users in aggregate. They tend to be very effective in practice for large data. The data we have is a form of explicit preference data which is perfect for training collaborative filtering models.

Alternating Least Squares (ALS) is a specific algorithm for solving a type of collaborative filtering model known as matrix factorization (MF). The core idea of MF is to represent the ratings as a user-item ratings matrix.

MF methods aim to find two much smaller matrices that, when multiplied together, reconstruct the original ratings matrix as closely as possible. This is known as factorizing the original matrix.

The two smaller matrices are called factor matrices (or latent features).. The idea in our project is that each user factor vector is a compressed representation of the user's preferences and behavior. Likewise, each item factor vector is a compressed representation of the item. Once the model is trained, the factor vectors can be used to make recommendations.

Fortunately, Spark's MLlib machine learning library has a scalable, efficient implementation of matrix factorization built in, which we can use to train our recommendation model.

D.) Export ALS user and item factor vectors to ElasticSearch:

The next step is to export the model factors to ElasticSearch.

In order to store the model in the correct format for the index mappings, we needed to create some utility functions. These functions allowed us to convert the raw vectors (which are equivalent to a Python list in the factor Data Frames above) to the correct delimited string format. This ensures ElasticSearch will parse the vector field in the model correctly using the delimited token filter custom analyzer you configured earlier.

We have also created a function to convert a vector and related metadata (such as the Spark model id and a timestamp) into a DataFrame field that matches the model field in the ElasticSearch index mapping.

In the recommend.py,we have defined utility functions that help us to achieve the recommendations. There are functions that define the similarity of movies for our project. Also along with that, we have defined the cosine similarity metric for our project that we have made use of. The utility function also allows us to convert the model vector back and forth in our implementation. The model vector is then stored in our ElasticSearch.

Also for our project, the most important method is the fn_query method. This method contains the vector plugin which we use to query against the model vector and the query is structured on the user's preferences.

```
def fn_query(query_vec, q="*", cosine=False):

    return {
    "query": {
        "function_score": {
            "query" : {
                "query_string": {
                    "query": q
                }
            },
            "script_score": {
                "script": {
                        "inline": "payload_vector_score",
                        "lang": "native",
                        "params": {
                            "field": "@model.factor",
                            "vector": query_vec,
                            "cosine" : cosine
                        }
                    }
            },
            "boost_mode": "replace"
        }
    }
}
```

Another key aspect in this function is the payload vector scoring. It is a metric which helps us in a custom analyzer to assign a scoring to the raw vectors. This score is useful when we try to query using ElasticSearch as it associates the query which we want with some payload and thereby gives us the flexibility in determining the best recommendations

In predict.py, we have written functions that allow us to store the model in the format that is suitable for the index in the ElasticSearch. They allow us to convert the raw vectors into the proper string format. Unless this is done properly, ElasticSearch would not work on the vector field built using the custom analyzer that we have specified in fn_query method

Also we have created additional functions for mapping all of the data with the model of the ElasticSearch index mappings to ingest the data in the it for querying.

```
model.itemFactors.select("id", vector_struct("features", lit(ver), ts).alias("@model")).write.format("es") \
    .option("es.mapping.id", "id") \
    .option("es.write.operation", "update") \
    .save("demo/movies", mode="append")
```

The above code is used for ingesting data of Items in the elastic search using the ElasticSearch connector with options of updating and appending available with alias as @model used.

good, thorough descriptions. thanks.

E.) Recommend using ElasticSearch:

Now that we have loaded your recommendation model into ElasticSearch, you will generate some recommendations. We created a few utility functions for:

1. Finding all the movies which contains the word in the search query
2. Finding similar movies to a given movie in the query

For finding similar movies we use a similarity score.This similarity score is computed from the model factor vectors for each movie. The ALS model we trained earlier is a collaborative filtering model, so the similarity between movie vectors will be based on the rating co-occurrence of the movies. In other words, two movies that tend to be rated highly by a user will tend to be more similar. We have used the cosine similarity of the movie factor vectors as a measure of the similarity between two movies.

Evaluations/Findings:

The result of the project can be best explained via an example. Say a user wants to search for a movie “Psycho”. He can do it by using the query statement:

search(‘psycho’)

Please note that these are the utility functions which exploit the power given by ElasticSearch and extend it to complete the system.

Once he/she inserts the query our system recommends the following output:

```
In [5]: search('psycho')
```

Out[5]:

	movieId	title	genres
1193	1219	psycho (1960)	Crime|Horror
2305	2389	psycho (1998)	Crime|Horror|Thriller
2817	2902	psycho ii (1983)	Horror|Mystery|Thriller
2818	2903	psycho iii (1986)	Horror|Thriller
3445	3535	american psycho (2000)	Crime|Horror|Mystery|Thriller
3739	3830	psycho beach party (2000)	Comedy|Horror|Thriller
9357	27473	american psycho ii: all american girl (2002)	Comedy|Crime|Horror|Mystery|Thriller
12296	56253	symbiopsychotaxiplasm: take one (1968)	Documentary
13455	66105	psychomania (death wheelers, the) (1973)	Horror
19825	97306	seven psychopaths (2012)	Comedy|Crime
19992	98004	single white female 2: the psycho (2005)	Drama|Thriller
20609	100229	psychopath, the (1966)	Horror|Mystery
21474	103454	batman unmasked: the psychology of the dark kn...	Documentary
21535	103633	making a killing: the untold story of psychotr...	Documentary
23240	109434	psychosis (2010)	Crime|Horror|Mystery

These are the list of all the movies which includes the movie “psycho” as well as it’s sequels and movies similar based on the ratings.

After this the users wants to know list of all the similar movies. For this the user inserts the query:

display_similar(1219,num=5)

The function query takes two parameters under consideration. One of them is the movie_id(1219) obtained from the first output and the second one(num=5) is the number of similar movies that the user wants to see.

This gives the output:

```
In [6]: display_similar(1219, num=5)
```

```
Get similar movies for:
Psycho

People who liked this movie also liked these:

Charm with score: 0.99992234

Death of a Gentleman with score: 0.99992234

How Not to Work & Claim Benefits... (and Other Useful Information for Wasters) with score: 0.9999223

Correcting Christmas with score: 0.9999223

Kaksparsh with score: 0.9999223
```

The above output shows that movie “Charm” is similar to “psycho” by a factor of 0.99 which means that people who liked “psycho” also liked “Charm”

For more examples, consider the video presentation made by us:

https://www.youtube.com/watch?v=oXNdUZcfEGc

thanks, looks great. i love toy story!

Review of Team Member Work:

Since the project was such that all the tasks needed to be completed sequentially, we have worked everything together.All of the work in the project was done by us sitting together. Both of us worked on this every week by sitting around 18 hours and completed all the checkpoints and documentations.

Conclusion:

Thus by developing by this project we have given moviegoers a list of movies that they can enjoy in their free time. The result of similar movies by genre and ratings allow people to access what other people who has similar thinking to theirs enjoyed. This is also used a lot in streaming services such as Netflix for their business. The usage of large data set with fast and efficient ElasticSearch allows users to find the movies that they can visualize and enjoy.

Thus by attempting this project we have achieved the following goals:

1. Ingested and indexed user event data into ElasticSearch using the ElasticSearch Spark connector
2. Loaded event data into Spark DataFrames and use Spark machine learning library (MLlib) to train a collaborative filtering recommender model
3. Exported the trained model into ElasticSearch
4. Using a custom ElasticSearch plugin, computed personalized user and similar item recommendations and combine recommendations with search and content filtering

Future work on this can include:

1. Investigate ways to improve ElasticSearch scoring performance:
    a. Performance for LSH-filtered scoring should be better!
    b. Can we dig deep into ES scoring internals to combine efficiency of matrix-vector math with ES search & filter capabilities?
2. Investigate more complex models:
    a. Scoring performance
    b. Factorization machines & other contextual recommender models
3. Spark Structured Streaming with Kafka, ElasticSearch & Kibana:

a. Continuous recommender application including data, model training, analytics & monitoring

References:

1. https://medium.appbase.io/how-to-build-a-movie-search-app-with-react-and-ElasticSearch-2470f202291c
2. https://opensourceconnections.com/blog/2016/10/05/elastic-graph-recommendor/
3. https://opensourceconnections.com/blog/2016/09/09/better-recsys-ElasticSearch/
4. https://github.com/MLnick/elasticsearch-vector-scoring/
5. https://www.ibm.com/cloud/garage/architectures/dataAnalyticsArchitecture
6. https://developer.ibm.com/technologies/data-science/
7. https://spark.apache.org/sql/
8. https://www.elastic.co/
9. https://www.themoviedb.org/

nice job with the document. project looked really fun. i'm taking off a small amount for grammar mistakes

29.75/30