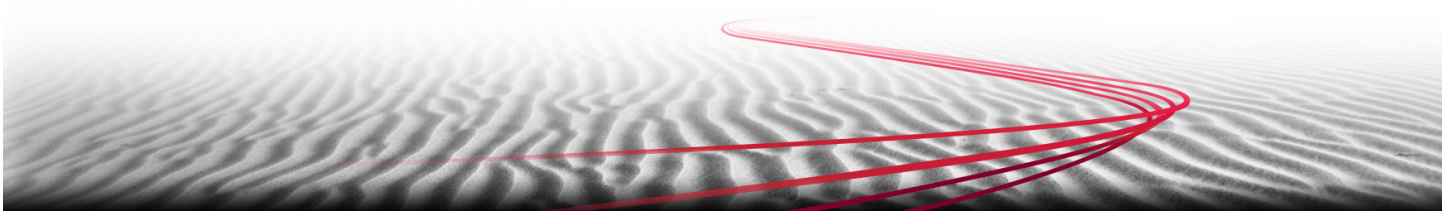


Recommender System with Mahout and Elasticsearch

This tutorial will describe how a surprisingly small amount of code can be used to build a recommendation engine using the MapR Sandbox for Hadoop (https://www.mapr.com/products/mapr-sandbox-hadoop) with Apache Mahout (http://mahout.apache.org/) and Elasticsearch (http://www.elastic.co/guide/en/elasticsearch/guide/current/index.html).

This tutorial will give step-by-step instructions on how to:

- Use sample movie ratings data from http://grouplens.org/datasets/movielens/ (http://grouplens.org/datasets/movielens/)
- Use a collaborative filtering algorithm from Apache Mahout to build and train a machine learning model
- Use the search technology from Elasticsearch to simplify deployment of the recommender

Step by Step Instructions

Software

This tutorial will run on the MapR Sandbox. The tutorial also requires Elasticsearch and Mahout to be installed on the sandbox.

- Download the 10M MovieLens dataset from the sample movie ratings data from http://grouplens.org/datasets/movielens/ (http://grouplens.org/datasets/movielens/)
- Install Mahout
- Install Elasticsearch

Step 1: Indexing the movie meta data in Elasticsearch

In Elasticsearch, documents contain fields which are, by default, all indexed. Typically documents are written as a single-level JSON structure. Documents are contained in an index and have an associated type that tells Elasticsearch how to interpret the fields in documents.

For our purposes, we will have a single **film** document type. The JSON film document has the following fields: id, title, year, genre, indicators, numFields:

```
{
 "id": "65006",
 "title": "Impulse",
 "year": "2008",
 "genre": ["Mystery","Thriller"],
 "indicators": ["154","272",”154","308", "535", "583", "593", "668", "6
 "numFields": 12
}
```

You can communicate with Elasticsearch over port 9200 using a RESTful API, accessible with a web client or from the command line by using the curl command. Here is an example command that accesses the Elasticsearch REST interface (http://www.elastic.co/guide/en/elasticsearch/guide/current/_talking_to_elasticsearch.html#_restful_api_with_json_over_http).

```
curl -X<VERB> 'http://<HOST>/<PATH>?<QUERY_STRING>' -d '<BODY>'
```

For a good explanation of how to fill in the parts surrounded by <>'s", see the Elasticsearch 101 tutorial (http://joelabrahamsson.com/elasticsearch-101/).

We can use the put mapping

(http://www.elastic.co/guide/en/elasticsearch/reference/current/indices-put-mapping.html) command from Elasticsearch's REST API to define a document type. The following request creates a mapping called film within the bigmovie index. The mapping defines that the numFields field is of type integer. All fields are stored and indexed by default, and integers are treated specially.

```
curl -XPUT 'http://localhost:9200/bigmovie' -d '
{
  "mappings": {
    "film" : {
      "properties" : {
        "numFields" : { "type" :   "integer" }
      }
    }
  }
}'
```

Movie information is contained in the file movies.dat. Each line of this file represents one movie, and has the following fields:

```
MovieID::Title::Genres
```

For example:

```
65006::Impulse (2008)::Mystery|Thriller
```

This Python script converts the movies.dat file data into the JSON format for bulk importing into Elasticsearch:

```
import re
import json
count=0
with open('movies.dat','rb') as csv_file:
   content = csv_file.readlines()
   for line in content:
        fixed = re.sub("::", "\t", line).rstrip().split("\t")
   if len(fixed)==3:
          title = re.sub(" \(.*\)$", "", re.sub('"','', fixed[1]))
          genre = fixed[2].split('|')
          print '{ "create" : { "_index" : "bigmovie", "_type" : "film",
          "_id" : "%s" } }' %  fixed[0]
          print '{ "id": "%s", "title" : "%s", "year":"%s" , "genre":%s
          % (fixed[0],title, fixed[1][-5:-1], json.dumps(genre))
```

This command runs the Python script index.py and puts the output into the file index.json:

```
$ python index.py > index.json
```

The Python script generates the Elasticsearch create requests in this format:

```
{ "create" : { "_index" : "bigmovie", "_type" : "film", "_id" : "1" } }
{ "id": "1", "title" : "Toy Story", "year":"1995" , "genre":["Adventure"
{ "create" : { "_index" : "bigmovie", "_type" : "film", "_id" : "2" } }
{ "id": "2", "title" : "Jumanji", "year":"1995" , "genre":["Adventure",
```

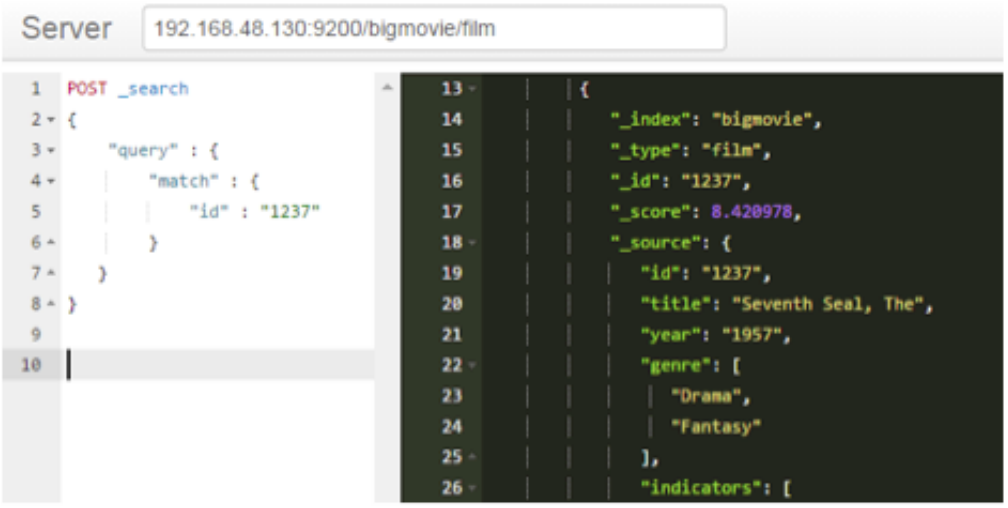
Each pair of lines creates a document and then specifies the fields in the document.

The bulk API makes it possible to perform many index/delete operations in a single API call, which can greatly increase the indexing speed. The following command bulk loads the file **index.json** into elasticsearch:

```
curl -s -XPOST localhost:9200/_bulk --data-binary @index.json; echo
```

After loading the movie data into Elasticsearch, you can use the REST API to query for documents. You can also use the Chrome plugin for Elasticsearch called Sense (https://github.com/bleskes/sense) to make REST queries easy to execute in a browser. The example below searches for movies within the drama genre, which returns "Five Days One Summer" among others:

The example below searches for a movie with the id 1237:



Step 2: Using Mahout to create Movie indicators from user rating data

Ratings are contained in the file ratings.dat. Each line of this file represents one rating of one movie by one user, and has the following format:

```
UserID::MovieID::Rating::Timestamp
```

For example:

```
71567::2294::5::912577968
71567::2338::2::912578016
```

The ratings.data file has "::" as a delimiter, which needs to be converted to a tab for the Mahout input. This sed command replaces the :: with a tab:

```
sed -i  's/::/\t/g' ratings.dat
```

Note that the sed -i option specifies that files are to be edited in-place (http://www.gnu.org/software/sed//manual/sed.html), and this command will open the file, replace the :: with \t and save the file again. Updates are only supported with MapR NFS and thus this command probably won't work on other NFS-on-Hadoop implementations. MapR Direct Access NFS allows files to be modified (supports random reads and writes) and accessed via mounting the Hadoop cluster over NFS.

The sed command above produces the format required for the Mahout input, **item1 item2 rating timestamp** (timestamp is not used):

```
71567    2294     5        912580553
71567    2338     2        912580553
```

Run the Mahout itemsimilarity job with the command line:

```
mahout itemsimilarity \
 --input /user/user01/mlinput/ratings.dat \
 --output /user/user01/mloutput \
 --similarityClassname SIMILARITY_LOGLIKELIHOOD \
 --booleanData TRUE \
 --tempDir /user/user01/temp
```

The argument "-s SIMILARITY_LOGLIKELIHOOD" tells the recommender to use the Log Likelihood Ratio (LLR) method for determining which items co-occur anomalously often and thus which co-occurrences can be used as indicators of preference. The default for similarity is .9; this can be adjusted based on the use case with the --threshold parameter, which will discard pairs with lower similarity (the default is a fine choice). Mahout computes the recommendations by running several Hadoop MapReduce jobs, the final product of which will be an output file in the `/user/user01/mloutput` directory. The output file has the following format (**item1id item2id similarity**):

```
64957   64997   0.9604835425701245
64957   65126   0.919355104432831
64957   65133   0.9580439772229588
```

Step 3: Adding Movie indicators to the Movie documents in Elasticsearch

Next, we need to add the indicators from the output file above, to the film documents in Elasticsearch. For example, we want to put all the indicators for a movie in the indicators field:

```
{
  "id": "65006",
  "title": "Impulse",
  "year": "2008",
  "genre": ["Mystery","Thriller"],
  "indicators": ["1076", "1936", "2057", "2204"],
  "numFields": 4
}
```

The table on the left shows how the document is put into Elasticsearch, the table on the right shows how an inverted index is used to search for a document corresponding to the search field.

Film document

Document: id	Field: indicator
8298	1237, 551
3	551
64418	1237

Inverted Index

Field: indicator	Document id
1237	64418
1237	8298
551	8298
551	3

We will be searching for documents corresponding to the indicator field. If we were searching for movies with the indicators **1237 and 551**, this example would return document (film) id 8298. If we were searching for movies with the indicators **1237 or 551**, this example would return document (film) id 8298, 3, and 64418.

Inverted Index

field	Document id
1237	64118
1237	8298
551	8298
551	3

This Python script will read the Mahout output file part-r-00000, create an array of indicators for each movie id, and then output the JSON Elasticsearch request to update the film document with the list of indicators.

```
import fileinput
from string import join
import json
import csv
import json
### read the output from MAHOUT and collect into hash ###
with open('/user/user01/mloutput/part-r-00000','rb') as csv_file:
    csv_reader = csv.reader(csv_file,delimiter='\t')
    old_id = ""
    indicators = []
    update = {"update" : {"_id":""}}
    doc = {"doc" : {"indicators":[], "numFields":0}}
    for row in csv_reader:
        id = row[0]
        if (id != old_id and old_id != ""):
            update["update"]["_id"] = old_id
            doc["doc"]["indicators"] = indicators
            doc["doc"]["numFields"] = len(indicators)
            print(json.dumps(update))
            print(json.dumps(doc))
            indicators = [row[1]]
        else:
```

```
            indicators.append(row[1])
        old_id = id
```

This command runs the Python script update.py and puts the output into the file update.json:

```
$ python update.py > update.json
```

The Python script above generates a file like this:

```
{"update": {"_id": "1"}}
{"doc": {"indicators": ["75", "118", "494", "512", "609", "626", "631",
{"update": {"_id": "2"}}
{"doc": {"indicators": ["15", "62", "153", "163", "181", "231", "239", "
```

This Elasticsearch REST bulk (http://www.elastic.co/guide/en/elasticsearch/reference/1.3/docs-bulk.html) request updates the bigmovie film index with the output file, update.json, from the Python script:

```
$ curl -s -XPOST localhost:9200/bigmovie/film/_bulk --data-binary @update.jso
```

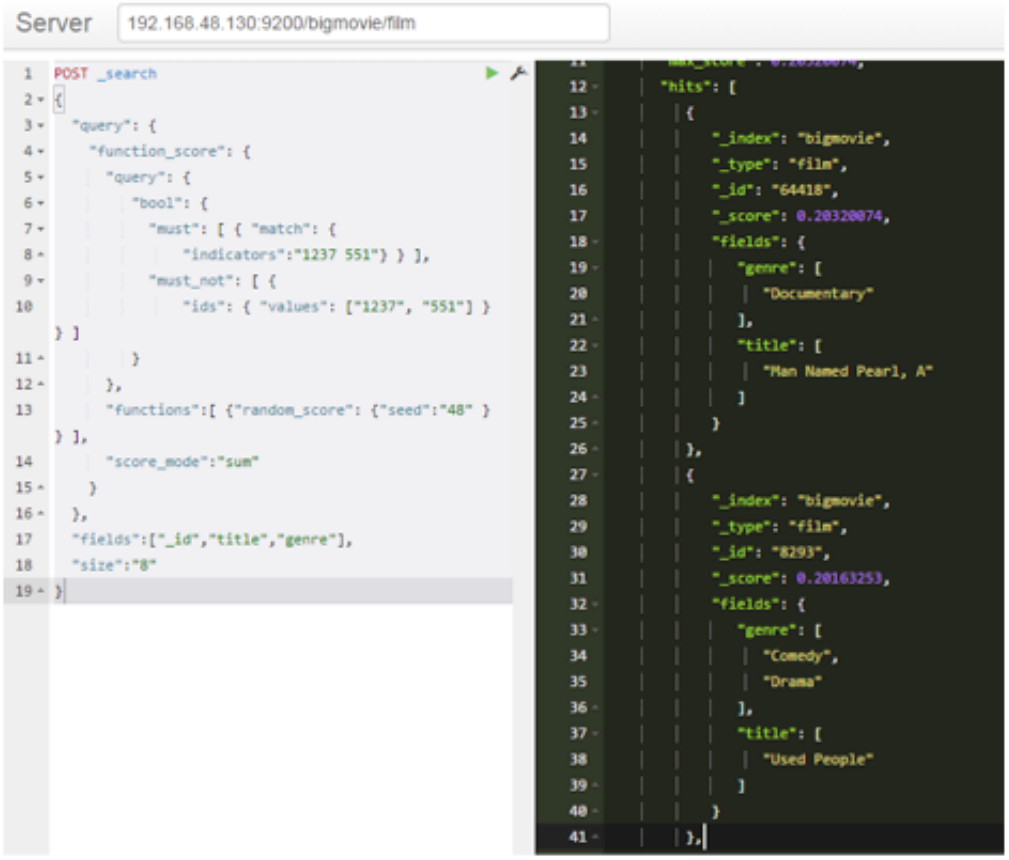
Step 4: Recommend by searching the Film Document Indicators field

Now you can query Elasticsearch for recommendations by searching the film document Indicators field. For example assume someone liked the movies with Ids 1237 and 551 and now you would like to recommend similar movies, running the following Elasticsearch query will get the recommended movies indicated by movie ids **1237 551 (1237=Seventh Seal, 551=Nightmare Before Christmas)**:

```
curl 'http://localhost:9200/bigmovie/film/_search?pretty' -d '
{
  "query": {
    "function_score": {
      "query": {
         "bool": {
           "must": [ { "match": { "indicators":"1237 551"} } ],
           "must_not": [ { "ids": { "values": ["1237", "551"] } } ]
         }
      },
      "functions":[ {"random_score": {"seed":"48" } } ],
      "score_mode":"sum"
    }
  },
```

```
  "fields":["_id","title","genre"],
  "size":"8"
}'
```

This query searches for documents with indicators 1237 or 551, and not with movie ids 1237 or 551. Below is an example running this query with the Sense plugin, with the first couple of results on the right, which shows that “A Man Named Pearl” and “Used People” are recommended movies for the indicators 1237 or 551.



Controlling Relevance

Full-text search engines sort matching documents by relevance, and the Elasticsearch relevance score is represented in the _score field. The **function_score (http://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-function-score-query.html)** allows you to modify the score of documents that are retrieved by a query. The random_score (http://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-function-score-query.html#_random) generates scores using a hash with a seed for variation. In the Elasticsearch query shown below, the random_score function is used to add variation to the result of the query to achieve dithering:

```
"query": {
  "function_score": {
    "query": {
       "bool": {
         "must": [ { "match": { "indicators":"1237 551"} } ],
         "must_not": [ { "ids": { "values": ["1237", "551"] } } ]
       }
    },
    "functions":[ {"random_score": {"seed":"48" } } ],
    "score_mode":"sum"
  }
}
```

Relevance dithering intentionally includes a few items with lower relevance in the top hits to broaden the training data that's fed to the recommendation engine. Recommendation engines select their own training data. Without dithering, tomorrow's training data just teaches what the model already knows today. Adding dithering helps the model by broadening the recommendations a bit, which gives broader range to the training data for the future. If the model is close to an excellent answer, dithering can help it find that answer. Effectively dithering decreases accuracy today in order to improve training data (and thus future performance) tomorrow.

Summary

We showed in this tutorial how to use Apache Mahout and Elasticsearch with the MapR Sandbox to build a basic recommendation engine. You can go beyond a basic recommender and get even better results with a few simple additions to the design to add cross recommendation of items, which leverages a variety of interactions and items for making recommendations. You can find more information about these technologies here:

References

To learn more about the components and logic of a recommendation engine, read "An Inside Look at the Components of a Recommendation Engine" (https://www.mapr.com/blog/inside-look-at-components-of-recommendation-engine#.VSv8o_nF81J) which details the architecture of the recommendation engine, collaborative filtering with Mahout, and the Elasticsearch search engine.

Want to know even more? Here are some additional resources regarding recommendation engines, machine learning, and Elasticsearch.

- Practical Machine Learning: Innovations in Recommendations (https://www.mapr.com/practical-machine-learning-new-look-anomaly-detection)
- An Invitation to Practical Machine Learning (https://www.mapr.com/blog/invitation-practical-machine-learning#.VQnjJI54qVM)
- Building a Simple Recommender (https://www.mapr.com/blog/building-a-simple-recommender#.VSwRAJTF-LS)
- Jump-Start Your Recommendation Engine on Hadoop (https://www.mapr.com/solutions/quickstart/recommendation-engine-on-hadoop-quick-start)
- MapR Quick Start Solution - Recommendation Engine Demo (https://www.youtube.com/watch?v=E597NL0WBJ4)
- Elasticsearch: The Definitive Guide (http://www.elastic.co/guide/en/elasticsearch/guide/current/index.html)
- Introduction to Item-Based Recommendations with Mahout (https://mahout.apache.org/users/recommender/intro-itembased-hadoop.html)

Tutorial Category Reference:
ElasticSearch (/taxonomy/term/2516)
Mahout (/taxonomy/term/2511)