Visual Analytics – Assignment 2 – Part 1

Claudio Maggioni

Indexing

The first step of indexing is to convert the given CSV dataset (stored in `data/restaurants.csv`) into a JSON-lines file which can be directly used as the HTTP request body of Elasticsearch document insertion requests.

The conversion is performed by the script `./convert.sh`. The converted file is stored in `data/restaurants.jsonl`.

The sources of `./convert.sh` are listed below:

```
#!/bin/sh
set -e

SCRIPT_DIR=$(cd -- "$( dirname -- "${BASH_SOURCE[0]}" )" &> /dev/null && pwd)

input="$SCRIPT_DIR/data/restaurants.csv"
output="$SCRIPT_DIR/data/restaurants.jsonl"


# In order:
# - Convert CSV to JSON
# - Convert JSON array in JSON lines notation
# - Remove last line (which is all `null`)
cat "$input" | jq -s --raw-input --raw-output \
  'split("\n") | .[1:-1] | map(split(",")) |
    map({
      "id": .[0],
      "name": .[1],
      "city": .[2],
      "location": {
          "lon": .[8] | sub("^\"\\["; "") | sub("\\s*"; "") | tonumber,
          "lat": .[9] | sub("\\]\"$"; "") | sub("\\s*"; "") | tonumber,
      },
      "averageCostForTwo": .[3],
      "aggregateRating": .[4],
      "ratingText": .[5],
      "votes": .[6],
      "date": .[7]
    })' "$input" | \
        jq -c '.[]' > "$output"
```

The gist of the conversion script is the following invocation of the *jq* tool. Here the CSV file is read as raw text, splitted into lines, has its first and last line discarded (as they are respectively the CSV header and a terminating blank line), splitted into columns by the `,` (comma) delimiter character, and each line is converted into a JSON object by *jq*. Note that *jq* is invoked in `slurp` mode so that the output is elaborated in one go.

Location coordinate strings are represented in the CSV with the pattern:

"[{longitude}, {latitude}]"

(with {longitude} and {latitude} being two JSON formatted floats). Therefore, the comma split performed by *jq* divides each cell value in two pieces. I exploit this side effect by simply removing the spurious non-numeric characters (like []" and space), converting the obtained strings into floats and storing them in the lon and lat properties of location.

After the conversion, the JSON-lines dataset is uploaded as an *Elasticsearch* index named restaurants by the script upload.sh. The script assumes *Elasticsearch* is deployed locally, uses HTTPS authentication and has HTTP basic authentication turned on. Installation parameters for my machine are hardcoded in variables at the start of the script and may be adapted to the local installation to run it.

The upload script, in order:

- Tries to DELETE (ignoring failures, e.g. if the index does not exist) and POSTs the /restaurants index, which will be used to store the documents.
- Field mappings are POSTed at the URI /restaurants/_mappings/. Mappings are defined in the mappings.json file.
- The lines of the dataset are read one-by-one, and then the correspoding document is POSTed at the URI /restaurants/_doc/{id} where {id} is the value of the id field for the document/line.

The sources of the upload script are listen below:

```
#!/bin/bash

set -e

SCRIPT_DIR=$( cd -- "$( dirname -- "${BASH_SOURCE[0]}" )" &> /dev/null && pwd )

elastic_dir="$HOME/bin/elasticsearch-8.6.2"
elastic_url="https://localhost:9200"
crt="$elastic_dir/config/certs/http_ca.crt"

input="$SCRIPT_DIR/data/restaurants.jsonl"
password="GZH*wqNTvQOWRdrPrpHm"

# Create index
curl --cacert "$crt" -u "elastic:$password" \
    -X DELETE "$elastic_url/restaurants" | jq . || true
curl --cacert "$crt" -u "elastic:$password" \
    -X PUT "$elastic_url/restaurants" | jq .

# Upload mappings
cat mappings.json | curl --cacert "$crt" -u "elastic:$password" -X POST \
    --data-binary @- "$elastic_url/restaurants/_mappings/" \
    -H "Content-Type: application/json" | jq .

# Upload documents one by one
while IFS= read -r line
do
    id=$(echo "$line" | jq '.id | tonumber')
    echo $line | curl -k --cacert "$crt" -u "elastic:$password" -X PUT \
        --data-binary @- "$elastic_url/restaurants/_doc/$id" \
        -H "Content-Type: application/json" | jq ._id &
done < "$input"
```

The mappings map the id field to type long, all other numeric fields to type float, the location field to type geo_point, and the date field to type date by using non-strict ISO 8601 with optional time as a parsing format.

All string fields are stored as type `text`, while also defining a `.keyword` alias for each to allow exact match queries on each field.

9499 documents are imported.

Queries

We would like to get those restaurants that have 'pizza' in the name and not 'pasta'. Get only the restaurants that have been reviewed at least as 'Very Good'.

This query is implemented with the following Dev Tools command:

```
GET /restaurants/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "name": "pizza"
          }
        }
      ],
      "should": [
        {
          "match": {
            "ratingText.keyword": "Very Good"
          }
        },
        {
          "match": {
            "ratingText.keyword": "Excellent"
          }
        }
      ],
      "must_not": [
        {
          "match": {
            "name": "pasta"
          }
        }
      ]
    }
  }
}
```

The query searches for all documents with the word "pizza" in the bag-of-words of each `name` text field, that either have the exact string "Very Good" or "Excellent" (only rating above "Very Good") in their `ratingText` field, and that whose name does not match with "pasta" in text search.

239 hits are returned.

Which are the 5 most expensive restaurants whose reviews were done in 2018? We are interested in reviews which refer only to places within 20 km from Athens (33.9259, -83.3389) and would like to look at the 5 most expensive.

The following query answers the question:

```
GET /restaurants/_search
{
  "from" : 0,
  "size" : 5,
  "sort" : [
    { "averageCostForTwo" : "desc" }
  ],
  "query": {
    "bool": {
      "must": [
        {
          "range": {
            "date": {
              "gte": "2018-01-01",
              "lte": "2018-12-31"
            }
          }
        },
        {
          "geo_distance": {
            "distance": "20km",
            "location": {
              "lat": 33.9259,
              "lon": -83.3389
            }
          }
        }
      ]
    }
  }
}
```

The 5 most expensive restaurants are in descending order:

- "Five & Ten" (id 108)
- "The National" (id 114)
- "DePalma's Italian Cafe - East Side" (id 107)
- "Shokitini" (id 112)
- "Choo Choo Eastside" (id 102).

Get all restaurants which contain the substring 'pizz' in the restaurant name but that do not contain neither 'pizza' nor 'pizzeria'.

The following query answers the question:

```
GET /restaurants/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "regexp": {
            "name": {
              "value": ".*pizz.*",
              "flags": "ALL",
              "case_insensitive": true
            }
          }
        }
      ],
      "must_not": [
        {
          "regexp": {
            "name": {
              "value": ".*pizza.*",
              "flags": "ALL",
              "case_insensitive": true
            }
          }
        },
        {
          "regexp": {
            "name": {
              "value": ".*pizzeria.*",
              "flags": "ALL",
              "case_insensitive": true
            }
          }
        }
      ]
    }
  }
}
```

The query specifies the required constraints using regular expressions instead of a plain `match` constraints in order to search for letter sequences within words, instead of only searching in the bag-of-words.

Only one restaurant is returned: “Pizzoccheri” (id 2237).

Aggregations

Show the number of restaurants reviewed as ‘Good’ aggregated by number of votes. Please consider the following ranges: from 0 to 250, from 250 to 500, from 500 to 750, from 750 to 1000. For each bucket we would like to know the minimum and maximum value of the average cost per 2.

This query outputs the answer:

```
GET /restaurants/_search
{
  "size": 0,
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "ratingText": "Good"
          }
        }
      ]
    }
  },
  "aggs": {
    "votes_ranges": {
      "range": {
        "field": "votes",
        "ranges": [
          { "from": 0, "to": 250 },
          { "from": 250, "to": 500 },
          { "from": 500, "to": 750 },
          { "from": 750, "to": 1000 }
        ]
      },
      "aggs": {
        "min_cost": {
          "min": {
            "field": "averageCostForTwo"
          }
        },
        "max_cost": {
          "max": { "field": "averageCostForTwo" }
        }
      }
    }
  }
}
```

The query does not face the shard size problem as the types of aggregations used do not face the problem. This is because bucket division combined with computation of a minimum and maximum value is trivially correct without approximation when implemented in a distributed MapReduce-like workflow.

The answer found is:

- Range $[0, 250]$: document count = 2060, minimum cost = 0, maximum cost = 350000
- Range $[250, 500]$: document count = 583, minimum cost = 10, maximum cost = 450000
- Range $[500, 750]$: document count = 217, minimum cost = 10, maximum cost = 5000
- Range $[750, 1000]$: document count = 99, minimum cost = 10, maximum cost = 200000

We are interested in cities which have not less than 10 restaurants and restaurants that have at least 100 votes. Which are the 7 cities with the highest average restaurant price (cost for two)?

This query implements a way to fetch the answer:

```
GET /restaurants/_search
{
  "size": 0,
  "query": {
    "range": {
      "votes": {
        "gte": 100
      }
    }
  },
  "aggs": {
    "city_term": {
      "terms": {
        "field": "city.keyword",
        "size": 10000,
        "shard_size": 10000,
        "min_doc_count": 10,
        "order": { "_count": "desc" }
      },
      "aggs": {
        "avg_price": {
          "avg": {
            "field": "averageCostForTwo"
          }
        },
        "avg_price_bucket_sort": {
          "bucket_sort": {
            "sort": [
              { "avg_price": { "order": "desc" } }
            ],
            "size": 7
          }
        }
      }
    }
  }
}
```

The document count output field `terms` aggregator is vulnerable to the shard size problem, thus to ensure correct counts we specify `shard_size` as an upper bound of the total document count. I can verify the output is correct as the output returns a `doc_count_error_upper_bound` of 0. Additionally, the `shard` parameter of the `terms` aggregator is set to the same value to not limit the number of buckets computed by the aggregator.

The cities found, in decreasing average `averageCostForTwo` are:

- "Jakarta", document count = 16, average price = 308437.5
- "Colombo", document count = 14, average price = 2535.714285714286
- "Hyderabad", document count = 17, average price = 1358.8235294117646
- "Pune", document count = 20, average price = 1337.5
- "Jaipur", document count = 18, average price = 1316.6666666666667
- "Kolkata", document count = 20, average price = 1272.5
- "Bangalore", document count = 20, average price = 1232.5

Show the highest number of votes for different rating types in descending order. You should consider only restaurants that are within 9000 km of New Dehli (28.642449499999998, 77.10684570000001).

This query provides the solution:

```
GET /restaurants/_search
{
  "size": 0,
  "query": {
    "geo_distance": {
      "distance": "9000km",
      "location": {
        "lat": 28.642449499999998,
        "lon": 77.10684570000001
      }
    }
  },
  "aggs": {
    "city_term": {
      "terms": {
        "field": "ratingText.keyword",
        "size": 10000,
        "shard_size": 10000
      },
      "aggs": {
        "max_vote_count": {
          "max": {
            "field": "votes"
          }
        },
        "max_vote_bucket_sort": {
          "bucket_sort": {
            "sort": [
              { "max_vote_count": { "order": "desc" } }
            ]
          }
        }
      }
    }
  }
}
```

The `terms` aggregator in this query is subjected to the same shard size problem described for the previous question.

The results are:

- Rating “Excellent”, document count $= 196”, maximum vote count = 10934
- Rating “Very Good”, document count $= 834”, maximum vote count = 7931
- Rating “Good”, document count = 1902, maximum vote count = 4914
- Rating “Average”, document count = 3683, maxiumum vote count = 2460
- Rating “Poor”, document count = 180, maximum vote count = 2412
- Rating “Not rated”, document count = 2135, maximum vote count = 3