Scalable Methods of Data Analysis

First assignment - MongoDB

David Kozak
dkozak94@gmail.com

Assignment to the subject Scalable Methods of Data Analysis

This document describes the assignment for the subjest Scalable Methods of Data Analysis. The goal is to implement 10 operations in mongodb.

Source

As a data set, I choose employee reviews of six tech companies, which I found on kaggle. The reviews include Google, Facebook, Amazon, Microsoft, Apple and Netlix.

The structure of one review is the following.

```
{
    "_id" : ObjectId("5cb8472aef58eac9129de127"),
    "id" : 1,
    "company" : "google",
    "location" : "none",
    "dates" : " Dec 11, 2018",
    "job-title" : "Current Employee - Anonymous Employee",
    "summary" : "Best Company to work for",
    "pros" : "People are smart and friendly",
    "cons" : "Bureaucracy is slowing things down",
    "advice-to-mgmt" : "none",
    "overall-ratings" : 5,
    "work-balance-stars" : 4,
    "culture-values-stars" : 5,
    "carrer-opportunities-stars" : 5,
    "comp-benefit-stars" : 4,
    "senior-mangemnet-stars" : 5,
    "helpful-count" : 0,
    "link" : "https://www.glassdoor.com/Reviews/Google-Reviews-E9079_P1.htm"
}
```

Import into mongo

The first step is to import the dataset into Mongo. It can be downloaded as a csv file. Since the first column did not have a name in the csv file, which resulted in a document property with empty name, it is necessary to add the "id" to it before import. The command for importing the modified csv file is the following.

```
mongoimport -d reviews -c reviews --type csv --file employee_reviews.csv --headerline
```

It will import the csv file into database reviews as a collection reviews.

Run the code

To run the code, you must select a mongo database server on which you want to run it and also specify the review database in the database server. The source code expects reviews collection to be there and have the structure described above.

```
mongo localhost:27017/reviews  fileToRun.js
```

1) Lookup table

The goal of this task was to create a lookup collection. I decided to create a new collection companies, which maps integer ids to company names. It can be done using the following command.

```
db.reviews.aggregate(
   [
      {
         $group:
            {
               "_id": "$company", // group based on the company
               index: {
                  $min: "$id" // as a unique id, use the smallest review id
               }
            }
```

```
    },
    {
      $project: {
        name: "$_id", // extract only wanted fields, which are id and name
        id: "$index",
        _id: 0 // remove the _id field
      }
    },
    {
      $out: "companies" // save into companies collection
    }

  ]
)
```

Then it is necessary to create a modified version of the reviews. First, the review and company are merged using the lookup operation. Then the company string is replaced with corresponding id from the companies collection.

```
db.reviews.aggregate(
  [
    {
      $lookup: {                // merge the collections together
        from: "companies",
        localField: "company",
        foreignField: "name",
        as: "others" // the documents from company collection will be included as field named others
      }
    },
    {
      $addFields:
        {
          company: { // replace the company field using the id value
            $arrayElemAt: ["$others.id", 0] // others is an array, therefore the arrayElemAt is used
          }
        }
    },
    {
      $project:{
        others:0 // remove the joined table
      }
    },
    {
      $out: "reviewsForLookup" // save into new collection
    }
  ]
)
```

2) Oversampling

For this task I decided to perform oversampling on the companies, because the distribution of reviews per company is uneven. This can be easily determined by executing the following query.

```
db.reviews.aggregate([
  {
    $group: {
      "_id": "$company",
      count: {
        $sum: 1
      }
    }
  }
]);
```

Which results in

```
{ "_id" : "apple", "count" : 12950 }
{ "_id" : "google", "count" : 7819 }
{ "_id" : "microsoft", "count" : 17930 }
{ "_id" : "amazon", "count" : 26430 }
{ "_id" : "facebook", "count" : 1590 }
{ "_id" : "netflix", "count" : 810 }
```

Unfortunately the aggregation operator sample cannot oversample, therefore it was necessary to perform this task mostly using a javascript code.

```
const necessarySamples = 26430;

const apple = {
    name: "apple",
    reviewCount: 12950,
};
const google = {
    name: "google",
    reviewCount: 7819,
};
const microsoft = {
    name: "microsoft",
    reviewCount: 17930,
};
const facebook = {
    name: "facebook",
    reviewCount: 1590,
};
const netflix = {
    name: "netflix",
    reviewCount: 810,
};

for (let {name, reviewCount} of [apple, google, microsoft, facebook, netflix]) {
    const steps = Math.floor(necessarySamples / reviewCount);
    const lastIterationSamples = necessarySamples % reviewCount;

    print(`Oversampling on ${name}, which requires full ${steps} iterations and ${lastIterationSamples} extra samples in the last round`);
    for (let i = 0; i < steps; i++) {
        const samples = db.reviews.aggregate([
            {
                $match: {
                    "company": name
                }
            }, {
                $project: {
                    "_id": 0
                }
            }
        ]).toArray();
        db.oversampled.insertMany(samples);
    }
    const samples = db.reviews.aggregate([
        {
            $match: {
                "company": name
            }
        },
        {
            $project: {
                "_id": 0
            }
        },
        {
            $sample: {
                size: lastIterationSamples
            }
        }
    ]).toArray();
    db.oversampled.insertMany(samples);
}

// add amazon
db.oversampled.insertMany(db.reviews.aggregate([
    {
        $match:
            {
                company: "amazon"
            }
    },
    {
        $project: {
            "_id":
                0
        }
    }
]).toArray());
```

3) Undersampling

Thanks to the fact that there is a sample aggregation operator, this task could be done in a simple manner as you can see in the code below. I decided to undersample to 810 reviews per company, since it is the lowest amount per company in the original collection.

```
const min = 810;

for (let name of ["apple", "google", "microsoft", "amazon", "facebook", "netflix"]) {
    const samples = db.reviews.aggregate([
        {
            $match: {
                "company": name
            }
        }, {
            $sample: {
                size: min
            }
        },{
            $project:{
                "_id":0
            }
        }
    ]).toArray();
    db.undersampled.insertMany(samples);
}
```

4) Discretizing

For the discretizing tasks I decided to convert the dates property. In the original collection, it contains dates in string format. Since for reviews their date is very important, I decided to discretize this column into values from the set {"CURRENT","OLD"}, where all the reviews that are older than 1.1.2017 are considered old. It is better to read the newer ones, because they can give you a more accurate insight into the current situation.

```
db.reviews.aggregate([{
        $addFields: {
            dates: {
                $dateFromString: {
                    dateString: "$dates", // convert to date object
                    onError: new Date() // use very old date on error
                }
            }
        }
    }, {
        $addFields: {
            dates: {
                $cond: [{$gte: ["$dates", ISODate("2017-01-01")]}, "CURRENT", "OLD"] // split into two categories
            }
        }
    },
        {
            $out: "modifiedReviews"
        }
    ]
);
```

5) Probability analysis

I decided to analyze is what is the probability of each review belonging to a specific company.

```
db.reviews.aggregate(
    [
        {
            $group: {_id: "$company", count: {$sum: 1}} // group based on company, count all reviews
        },
        {
            $project: {
                name: "$_id", _id: 0, probability: {
                    $divide: ["$count", db.reviews.count()] // divide by the total number of reviews
                }
            }
        },
        {
            $out: "probReviews" // save into another collection
        }
    ]
);
```

```
// and of course the total should be one
db.probReviews.aggregate( [ {$group:{ "_id":"name",total: { $sum : "$probability"  }  }}])
```

6) Tf-idf

This was a challenging task. In the end, I ended up using MapReduce to perform it. I decided to calculate the importance of the word 'work'. First, I had to perform a precomputation to determine in how many reviews this word occurs.

```
db.reviews.mapReduce(
    function () {
        emit(null, `${this.summary}`.indexOf(' work ') > 0 ? 1 : 0);
    },
    function (key, values) {
        return Array.sum(values);
    },
    {
        out: "wordStats"
    }
);
```

Than I grouped this information together with total document count and joined them with stats, creating new collection withStats.

```
db.wordStats.aggregate([
    {
        $project: {
            "work": "$value",
        }
    },
    {
        $addFields: {
            "totalDocs": 67529
        }
    },
    {$out: "wordStats"}
]);

db.reviews.aggregate([
    {
        $lookup: {
            from: "wordStats",
            localField: "null",
            foreignField: "null",
            as: "stats"
        }
    },
    {$out: "withStats"}
]);
```

Afterwards, I could calculate the tf-idf using the following query.

```
const map = function () {
    const calcTf = (word, document) => {
        const count = document.filter(elem => word === elem).length;
        return count / document.length;
    };
    const stats = this.stats[0];
    const tf = calcTf("work", `${this.summary}`.split(' '));
    const idf = Math.log10(stats.totalDocs / stats.work);
    const tfidf = tf * idf;
    emit(this.id, {
        summary: this.summary,
        tfidf
    })
};

db.withStats.mapReduce(
    map,
    function (key, values) {
        throw new Error(`Should never be called, was called with key ${key} and values ${values}`)
    },
    {
        out: "tfidf"
    }
);
```

7) Index

I decided to create an index over the summary field, since it contains the longest strings in the dataset. This task turned out to be fairly straightforward using MapReduce. It can be achieved using the following code.

```
db.reviews.mapReduce(
    function () {
        for (let word of `${this.summary}`.split(/\s+/)) {
            emit(word, {
                documents: [this.id]
            });
        }
    },
    function (key, values) {
        const documents = [];
        for(let value of values){
            documents.push(...value.documents)
        }
        return {
            documents
        }
    },
    "index"
)
```

8) Cross validation

I decided to split the reviews into 5 datasets. Originally I wanted to use the $sample aggregation pipeline operator, but after some research I found it ill-fitting for two reasons. First of all, $sample does not guarantee to return unique results. On top of that, for generating K datasets, sample has to be run k times and there is no guarantee that there will be unique samples in each run, therefore the sets obtained by that approach won't be disjoint.

That's why I decided to split the dataset using MapReduce in combination with random number generation. In each run of map, I generate a random integer from range [0,k) and I use it as a key. Then in reduce, I merge all the results with same key into one list. This way I obtain a collection with ids from [0,k) and values consisting of the subset of reviews that should be included in given dataset.

```
const map = function () {
    const k = 5;
    const category = Math.floor(Math.random() * k);
    emit(category, {
        reviews: [this]
    })
};

const reduce = function (key, values) {
    const reviews = [];
    for (let item of values) {
        reviews.push(...item.reviews);
    }
    return {
        reviews
    }
}

db.reviews.mapReduce(
    map,
    reduce,
    "cross_validation"
)
```

Afterwards I use aggregation pipeline to split the cross_validation collection into K subcollections, each containing one dataset.

```
for (let i = 0; i < K; i++) {
    db.cross_validation.aggregate([
        {
            $match: {"_id": i}
        },
        {
            $unwind: "$value.reviews"
        },
        {
```

```
            $project: {
                "review": "$value.reviews"
            }
        },
        {
            $project: {
                "value": 0,
                "_id": 0
            }
        },
        {
            $out: `cross_validation_${i}`
        }
    ]);
}
```

9) Normalization

For the normalization task I decided to normalize overall-ratings. In the original dataset, their values are from interval [0,5]. I decided to normalize them to interval [0,1]. This can be achieved using the following query.

```
db.reviews.aggregate([
    {
        $addFields:{
            "overall-ratings":{
                $divide:["$overall-ratings",5]
            }
        }
    },
    {
        $out:"normalized"
    }
]);
```

10) Remove noise

For this task I decided to remove all the reviews that are older than 1.1.2018. The reason is the same as in the discretizing task. Older reviews are not so relevant. This can be done using the following query.

```
db.reviews.aggregate([
    {
        $addFields: {
            dates: {
                $dateFromString: {
                    dateString: "$dates", // convert to date object
                    onError: new Date() // use very old date on error
                }
            }
        }
    },
    {
        $match: {
            dates: {
                $gte: ISODate("2018-01-01")
            }
        }
    },
    {
        $out:"withoutNoise"
    }
]);
```

11) Fill missing values

This task turned out to be a bit problematic, because there were no missing values in the original dataset. Therefore as a first step I had to insert some "artifical nulls". I decided to remove the overall-ratings for every fifth review.

```
db.reviews.aggregate(
    [
        {
            $addFields: {
                "overall-ratings": {
```

```
                $cond: [{$mod: ["$id", 5]}, "$overall-ratings", null] // every fifth review will have null as overall-ratings
            }
         }
      },
      {
         $out: "nulledReviews"
      }
   ]
);
```

Then I computed the average rating.

```
db.reviews.aggregate(
   [
      {
         $group: {
            "_id": null, average:
               {
                  $avg: "$overall-ratings"
               }
         }
      },
      {
         $out: "averageRating"
      }
   ]
);
```

And finally inserted the average value where the overall-ratings is null

```
// extract it
const average = db.averageRating.findOne().average;
db.nulledReviews.aggregate(
   [
      {
         $addFields:
            {"overall-ratings": {$ifNull: ["$overall-ratings", average]}}
      },
      {
         $out: "notNullReviews"
      }
   ]
);
```

12) Pivot table

For this tasks I decided to calculate the average rating per company in * overall-ratings * work-balance-stars * culture-values-stars * carrer-opportunities-stars

This can be achieved using the following aggregation pipeline.

```
db.reviews.aggregate([
   {
      $group: {
         "_id": "$company",
         "overall-ratings": {
            $avg: "$overall-ratings"
         },
         "work-balance-stars": {
            $avg: "$work-balance-stars"
         },
         "culture-values-stars": {
            $avg: "$culture-values-stars"
         },
         "carrer-opportunities-stars": {
            $avg: "$carrer-opportunities-stars"
         }
      }
   },
   {
      $out: "pivot"
   }
]);
```