Scalable Computing Report - NewsForYou

Remi Brandt (2509644) Timo Smit (2337789) Timon Back (3218147)

March 30, 2017

1 Introduction

NewsForYou is a news article recommendation engine for everyone. Based on the likes of each specific user, personal recommendation for other interesting articles are made. All news articles are provided by the New York Times API.

This application is fully scalable to more users, more articles and more user-created likes by using Apache Spark. The limit is not what the a single machine can offer on computational power, but rather how many nodes in the cluster are available. Also, the system is fault-tolerant, since the restarting of failed services is taken over by the Docker swarm technology.

The main focus of this system is to be scalable with the self-implemented version of the Alternating Least Squares (ALS) recommendation algorithm that was improved to run in a distributed fashion.

2 Recommendation generation

In order to recommend articles, the Alternating Least Squares Method for Collaborative Filtering algorithm (ALS) was implemented. In essence, the ALS algorithm is used to compute two matrices: $\mathbf{U}$ and $\mathbf{A}$, such that $\mathbf{U} \cdot \mathbf{A} = \mathbf{R}$, where matrix $\mathbf{R}$ is filled with predicted rating values. In matrix $\mathbf{R}$, rows correspond to users and columns correspond to articles. $\mathbf{U}$ and $\mathbf{A}$ are computed from known `Rating` tuples which contain a user id, article id and rating. These tuples correspond to all the ratings given by users to articles and are stored in a MongoDB collection which contains one rating per row.

ALS is an iterative algorithm of which the number of iterations can be set manually. Other parameters which are set manually are the number of latent factors (the number of expected underlying factors which will be inferred by ALS) of $\mathbf{U}$ and $\mathbf{A}$, the regularization parameter (used to prevent over-fitting) and how many recommendations should be made for each user.

2.1 Batch processing

We have parallelized ALS using Spark. First $\mathbf{A}$ is initialized with random data. Subsequently, in each iteration $\mathbf{A}$ is fixed and $\mathbf{U}$ is updated and then $\mathbf{U}$ is fixed and $\mathbf{A}$ is updated.

In order to update $\mathbf{U}$,

$$\mathbf{U}_u = (\mathbf{A}\mathbf{W}_u\mathbf{A}^T + \lambda\mathbf{I})^{-1}\mathbf{A}\mathbf{W}_u\mathbf{R}_u \tag{1}$$

is computed for each user. Subsequently, to update $\mathbf{A}$,

$$\mathbf{A}_i = (\mathbf{U}^T\mathbf{W}_i\mathbf{U} + \lambda\mathbf{I})^{-1}\mathbf{U}^T\mathbf{W}_i\mathbf{R}_i \tag{2}$$

is computed for each article [1]. The resulting vectors are when combined (obviously) equal to $\mathbf{U}$ and $\mathbf{A}$ respectively. In eq. (1) and eq. (2), $\mathbf{I}$ is the identity matrix, λ is the regularization parameter, $\mathbf{R}$ is a vector of ratings given by a user to an item, of ratings of an item given by users, or a 0 if

[1] https://bugra.github.io/work/notes/2014-04-19/alternating-least-squares-method-for-collaborative-filtering/

no rating exist, $\mathbf{Z}^{-1}$ is the matrix inverse of $\mathbf{Z}$, and $\mathbf{W}$ are diagonal matrices corresponding to a user or article indicating whether a user has rated an article or whether an article has been rated by a user. A value of 1 on the diagonal indicates that a rating exists, a value of 0 that a rating does not exist.

We have decided to store both matrix $\mathbf{A}$ and $\mathbf{U}$ in an RDD which has it's dimension which contains the most elements as RDD rows. We have done this because this will allow for the most fair and scalable parallelization since Spark distributes RDD rows among Spark nodes. During matrix multiplication, the transpose of $\mathbf{A}$ is taken to correct for the fact that the transpose of $\mathbf{A}$ is stored in memory.

In order to parallelize the update computation of $\mathbf{U}$ and $\mathbf{A}$, a number of mathematical theorems where considered:

1. $\mathbf{Z}\mathbf{W}_u\mathbf{Z}^T = \mathbf{X}\mathbf{X}^T$, where $\mathbf{X}$ is equal to $\mathbf{Z}$ where columns which correspond to a zero in $\mathbf{W}_u$ are filled with zeros.

2. $\mathbf{Z}\mathbf{W}_u = \mathbf{X}$, where $\mathbf{X}$ is equal to $\mathbf{Z}$ where columns which correspond to a zero in $\mathbf{W}_u$ are filled with zeros.

3. $\mathbf{Z}\mathbf{Z}^T = \mathbf{X}$, where $\mathbf{X}$ is the sum of the column vectors of $\mathbf{Z}$ times their transpose.

Obviously, $\mathbf{W}_u$ and $\mathbf{W}_i$ are easily derived from `Rating` tuples: $\mathbf{W}_u$ contains in each of the elements of the diagonal matrix a 1 if user u has rated the corresponding article, otherwise 0. $\mathbf{W}_i$ contains in each of the elements of the diagonal matrix a 1 if article i has been rated by the corresponding user, otherwise 0. We have implemented $\mathbf{W}_u$ and $\mathbf{W}_i$ by not updating for rows and columns which correspond to a 0. In order to find rows and columns which correspond to a 0, we join ratings with either the article factors or user factors based on the user id or article id. When no rating exists for an element of the factors, it will not be in the joined set.

To update $\mathbf{U}$ or $\mathbf{A}$, ratings are first joined with factors $\mathbf{U}$ or $\mathbf{A}$ based on their corresponding key (user id or item id) which results in accordance with theorem (1) and (3) in $\mathbf{A}\mathbf{W}_u\mathbf{A}^T$ or $\mathbf{U}^T\mathbf{W}_i\mathbf{U} = \mathbf{P}_1$. Likewise ratings are joined with factors $\mathbf{U}$ or $\mathbf{A}$ based on their corresponding key to compute $\mathbf{U}^T\mathbf{W}_i\mathbf{R}_i$ or $\mathbf{A}\mathbf{W}_u\mathbf{R}_u = \mathbf{P}_2$ in accordance with theorem (2) and (3). The previous operation is done in parallel where the work is distributed on a users or articles basis.

The remaining task to compute $(\mathbf{P}_1+\lambda\mathbf{I})^{-1}\mathbf{P}_2$ is again done in parallel where the work is distributed on a users or articles basis. Matrix inversion is done locally on each Spark node using breeze, because inverting a single matrix is not resource expensive. The reason for this is that the matrix is a two dimensional square matrix with in both dimensions as many elements as there are latent factors (commonly around 10).

In order to recommend articles, $\mathbf{U}$ and $\mathbf{A}$ are multiplied using the Spark multiplication function for distributed block matrices resulting in matrix $\mathbf{R}$. Subsequently a CoordinateMatrix $\mathbf{C}$ is constructed which contains positive infinity at each location (userId, itemId) where a user has rated an article. $\mathbf{C}$ is subtracted from $\mathbf{R}$ in order to remove already rated articles from recommended articles resulting in matrix $\mathbf{Q}$. For each row of $\mathbf{Q}$, which corresponds to each user, a set number of highest ratings and the corresponding column id is determined. This task is distributed between Spark nodes based on rows. The resulting recommendations are subsequently stored in a MongoDB collection which contains one row per user and in each row a user id and array of article ids which where recommended to the user.

2.2 Online processing

When the batch processing is done with computing matrix $\mathbf{A}$ and $\mathbf{U}$, matrix $\mathbf{A}$ is stored in a MongoDB collection which contains in each row an article id and the corresponding latent factor values. Matrix $\mathbf{A}$ is subsequently read by the streaming algorithm. The streaming algorithm has two new rating processing tasks: remove recommended articles after they have been rated by a user and to recommend new articles when no articles are recommended to a user any more.

Kafka is used to stream messages to a number of consumers which share the same consumer group. In that way new rating processing is load balanced between the different consumers.

The streaming recommender removes, whenever a rating from a user comes in, the rated article from the recommendations for the user if it is in said recommendations list.

The streaming algorithm recommends articles based on ratings made both after and before the model was learned. In order to do this, the default ALS algorithm is run, but matrix $\mathbf{A}$ is kept constant and $\mathbf{U}$ is recomputed based on the ratings of only one user. Because $\mathbf{A}$ is kept constant, ALS is run only with one iteration. In this way, new recommendations can be generated in a fraction of the time it would take the batch processing algorithm.

3 Infrastructure

The complete infrastructure is shown in fig. 1. The basic data flow starts with the importer querying the archive API of the New York Times. This component runs the importing task locally in its own Apache Spark Context and publishes its results to the MongoDB.

The streamer reads the imported articles from the database and then generates automatically some ratings based on that, which can then be used by the recommender.

The recommender consists of two parts: the batch recommender and the streaming recommender. The batch recommender generates based on the available ratings that it reads from the database the recommendation matrix. The streaming recommender updates those recommendations based on new incoming ratings - as it is described in section 2.2. Both of these tasks are run in the distributed Apache Spark cluster with multiple worker nodes.

Finally, the visualization is done with the visualization node, which is a simple Play framework (Scala) application.

In total 9 different services are used:

- MongoDB
- Kafka
- Spark (master node)
- Spark (worker node)
- Importer
- Streamer
- Batch recommender (via a Spark task submitter)
- Streaming recommender (via a Spark task submitter)
- Visualization

To administrate all of the services, all components are encapsulated in a Docker[2] container. Moreover, all services are managed in a Docker swarm cluster. This allows for easy scaling of the components even over multiple machines.

Also, the load balancing of requests is taken over by the Docker technology. However, the control over the request redirection is not transparent to the outside. A request made to a service running on the same machine may get redirected to a different node due to the load balancing policies. This also applies to request to a specific IP address - which is a specific instance. So, it can never be assumed that a specific instance responses to a request.

[2]Docker is a virtualization technology to run processes in an own system environment without requiring a virtual machine around it, which makes it lightweight.

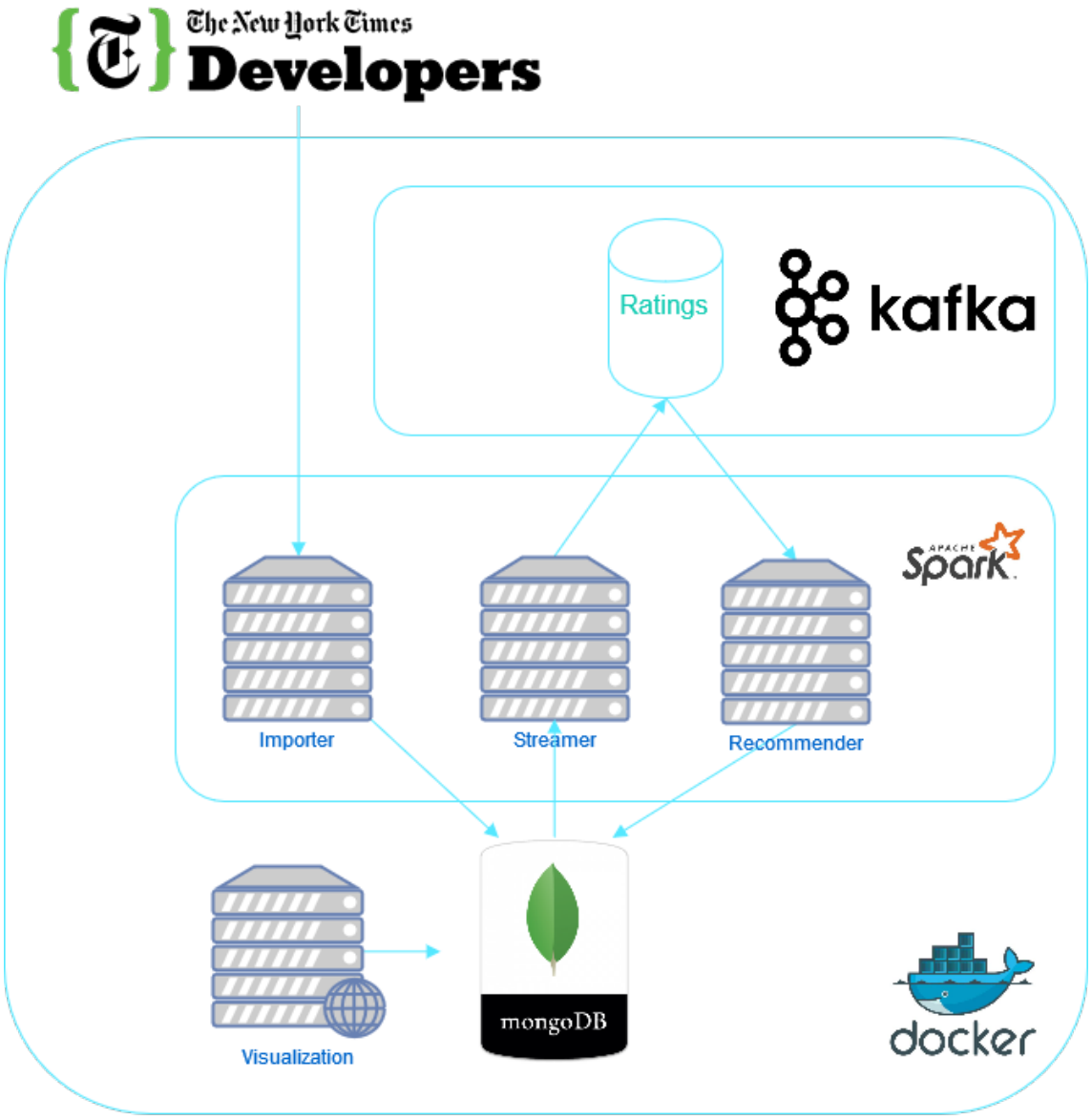

Figure 1: Overview of the infrastructure of NewsForYou

3.1 Database

As a permanent data storage MongoDB is used. Unfortunately, MongoDB does not offer automatic configuration of replica sets, so that manual configuration during runtime is necessary. This is also due to the fact that MongoDB has no static master node, but can dynamically elect a new master node, which then becomes the new administration node to which new nodes have to connect to.

3.2 Spark

For the highly computational tasks, Apache Spark is used. Apache Spark is used in a cluster mode with at least one administration instance and one or multiple worker nodes. Tasks are submitted to the administrator instance, which then distributes the tasks among the worker nodes and in the end collects the results.

In the NewsForYou project, the complete article recommendation is run over this Apache Spark cluster using the map-reduce pattern that makes Spark so powerful. Also, the importer of the data is run on a Spark instance, however that is a local instance and not connected to the cluster. Although that is of course possible, it does not make as much sense since the importing process does not require a lot of computational power.

It is important to keep in mind that when a task is submitted to the Spark cluster, also all dependencies of the program (Java or Scala) have to be included as well. Apache Spark is not able to do dependency resolving on its own.

3.2.1 Importer

The importer is a rather straightforward component of our system. It is responsible, as the name suggests, for importing the data into the system when it is to be deployed for first use, or after truncating all the documents in the database.

The component connects to the New York Times API, which we preferred over other APIs (e.g. `newsapi.org`), since it allows us to query the archive by month and year and does not have a query limit. In addition to that it gives us much information compared to other services, such as abstracts and also several identifiers that we can use to quickly find an article.

While abstracts and related fields are not used currently in the algorithm, it is good to have the information available, as the importing process takes some time. This is due to the fact that it iteratively queries the archive, starting from the current month up to the moment that the API first refuses to give any results (around mid-twentieth century).

The articles that are collected from the API are split into two sets: one that is readily available for the recommender to create a first model, and a second set that is used in the streamer (see section section 3.2.2). This split is done on a 25%/75% basis. Both sets are stored in the same database, but in different collections.

3.2.2 Streamer

The streamer component is used to create an artificial stream of ratings. This is done to simulate a real-world application of the system, in which articles are continuously rated by users to indicate what they (dis)like.

The component queries the data that has been collected from the API for streaming purposes, as described in section section 3.2.1. For each article that has been collected, it will first add it to the main articles collection (which contains all collected articles) and then add it to the Kafka queue (see section section 3.3).

This process has to be specifically executed in this order, as the recommender needs the articles to be available in order to determine the ratings.

3.2.3 Recommender

Please consult section section 2.1 and section 2.2 for information concerning the use of Spark in the recommender subsystem. In this section some additional design choices regarding Spark will be shared.

Both in the streaming and batch recommender virtually no action functions have been applied to RDD's only transformations, which is beneficial to their scalability. The only exception is a few uses of the `first` function.

In the `alsStep` function, an identity matrix is initialized locally and not in a distributed way. This is done because said matrices will commonly have about 10 elements in both of their dimensions and these matrices can hence easily be stored as a whole on a Spark node.

In the `alsStep` function, matrix inverse, multiplication and addition are computed locally because these are applied to matrices or vectors with around ten elements per dimension. Note that the collection of these small tasks is distributed among Spark nodes, i.e. they are not all computed on the same node.

In order to distribute the work of determining the biggest elements in each row of $\mathbf{R}$, $\mathbf{R}$ is converted to an `IndexedRowMatrix` after which the rows are distributed among Spark nodes. Locally the biggest n elements of each row are determined in linear time using the `getLargestN` function.

3.3 Kafka

To be able to send message and to have a consistent data queue over the whole instance, we use Apache Kafka. Additionally, to saving some data permanently into the MongoDB database, we stream data also to to the message queue. This allows for a better scalability.

Kafka is used in the streaming recommender to allow automatic updates on the recommendation model without having to update the whole model. Instead of queuing for all available data, only the new incoming data is processed.

Apache Kafka works fully transparent and was easily integrated into the overall system.

4 Conclusion

Over the course of the project the main concept of the project has not changed, though several (sub)components had to change to come to a successful scalable and fault-tolerant application.

First of all, we successfully developed a news article recommendation service that is based on artificial user ratings on real world news articles. This service is able to cope with both batches of articles, as well as streaming likes.

Although no article texts are actually parsed in the process and no articles can be added through streaming, this could be done if the algorithm would be changed. However, the current approach fits within the scope of the project and the course and is sufficiently sophisticated.

The recommendation service makes great use of Spark's ability to scale RDDs among several nodes, as well as using an algorithm that can perfectly scale with many users.

For visualizing the results of the recommendation process a Play Framework application is made, which is parallelized using Play's native functionality.

All in all, all of the vital components are run in parallel and are able to scale on demand. This is true except for the importer (and streamer), however these are only used on deployment/initialization of the project.