Machine Learning at Scale

Author: Bc. Ondřej Fiedler
Mentor: Ing. Tomáš Bartoň

18. 9. 2014

Chapter 1

Introduction and approach

1.1 Introduction

My task during the *eClub Summer Camp 2014* was to implement a recommendation system in *Scala* using *Apache Spark* framework. A good recommender can be a key advantage for any company, which offers some products on the Internet as clients will tend to use more of its products and they will be even happier about it.

We chose Spark framework because it introduces primitives for in-memory cluster computing, that should provide great scalability and computation speed (authors of Spark claims it to be up to 100 times faster than *Hadoop MapReduce* for certain applications[1]).

Distributed computing on a cluster brings many advantages, but also some disadvantages – mainly time overhead due to communication between nodes. Its impact is decreasing with bigger data, so we analyzed the dependency of data size and used time, and compared our system with the sequential solution.

1.2 Approach

1.2.1 Collaborative filtering

Our recommendations are based on previously gathered ratings from users. The assumption is that if person A has similar taste as B on common rated products, than A would probably like also B's other favourite products. This technique of recommending favorite products of *similar* users is called *collaborative filtering*.

1.2.2 Ratings and distance metric

We represent each user as one vector, which contains user's rating for every rated product in the past and some default value for other products. The ratings can be collected in two ways:

- *explicitly collected* – users gave products some ratings (which can vary from binary (like/dislike) to integer or floating point)
- *implicitly collected* – ratings are made from product pageviews in an online store or from purchased products

Now let's say, that we want to recommend some products to user A. As mentioned above, we need to find the most similar users to user A in terms of common product preferences. For that we need a distance metric between two vectors, that represent the users. We use these two metrics in our project:

- *Euclidean distance*[*]

$$d(a,b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \cdots + (a_n - b_n)^2} = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2} \tag{1.1}$$

- *Modified cosine distance*

$$d(a,b) = -\frac{A \cdot B}{\|A\|\|B\|} \tag{1.2}$$

where $a = (a_1, a_2, \ldots, a_n)$ and $b = (b_1, b_2, \ldots, b_n)$ are vectors with ratings corresponding to users A and B, and $d(a,b)$ is the distance between them.

1.2.3 *k*-nearest neighbors algorithm

k-nearest neighbors algorithm (k-NN) is one of the most common machine learning algorithms and does exactly what was proposed in previous section:

1. Find the k most similar users to user A (according to a metric)
2. Compute average rating for every product from ratings of similar users
3. Recommend products with the highest ratings, excluding the products already rated/purchased by A

[*]We don't take the square root in the real implementation, as it is not necessary for comparing two distances and square root is time consuming operation. Also only products that were rated by both users were included in computation.

1.2.4 *k*-NN with clustering

As you can see k-NN algorithm is extremely simple, but it can still give reasonable recommendations. The downside of this algorithm is the computational cost – it has to iterate over every user vector to compute the distances. In many business usecases is the number of users and ratings very high and at the same time the recommendations must be computed very quickly. Therefore the basic k-NN algorithm must be changed to favor the speed of computation.

One of the possible ways is to use clustering. In the simplest scenario the vectors are clustered for example by k-Means algorithm[5] on startup. Then when we recommend some products to user A, we firstly find the cluster that A belongs to (or few nearest clusters), and then we run the k-NN algorithm on vectors in that cluster. This approach should be much faster then regular k-NN, because the number of vectors in cluster is many times smaller than the number of all vectors. Determining the nearest cluster is also computationally easy, because the number of those centroids is by an order of magnitude smaller than number of all vectors. This approach can of course harm the quality of the recommendation, so finding the right trade-off between speed and quality is the key question.

1.2.5 Tree of clusters

Combination of k-NN and clustering is not a new idea – you can read for example papers by Al Mamunur Rashid et al.[6] or by Jerome Kelleher and Derek Bridge[4]. The second mentioned authors improved the idea with an algorithm called RecTree that builds a binary tree of clusters using recursively called k-Means algorithm with $k = 2$. We have used their idea in the following algorithm.

We will create a binary tree in which the actual clusters of user vectors will be stored in the leafs. Each node except root will contain a representative – centroid vector of the *supercluster* that contain all vectors in that node's subtree.

Creating tree with n nodes takes these steps:

1. Create a root node that contains all the user vectors.
2. Take the leaf l with largest associated cluster and split it into two smaller clusters by k-Means algorithm (with $k = 2$). Create two children of l and put the new clusters into them. Centroids of those clusters become the representatives of those two new nodes.
3. If there is already n clusters (leafs) then quit, otherwise proceed with step 2.

Recommendation for user represented by vector v then works as follows:

1. *node* = root of the tree
2. If *node* is a leaf, then recommend using k-NN algorithm computed on the vectors of the cluster associated with this leaf and quit.
3. *node* is the inner node. Compute distance between v and representative of the *node*'s children. *node* = child with representative nearer to v.
4. Proceed with step 2.

As you can see the tree may not be (and on most dataset won't be) balanced, because we always split the largest existing cluster. We do this because we want the clusters to be roughly similar size (optimal would be if all the clusters had the size of $\frac{\text{number of all vectors}}{\text{number of clusters}}$), so the average recommendation time will be small.

Chapter 2

Implementation

2.1 Algorithms

I have implemented k-NN, clustered k-NN, tree of clusters and also adapted Alternating Least Squares method (ALS) from Spark's MLlib.[7]

2.1.1 *k*-NN

Implementation of k-NN with Spark is pretty straight forward. The first thing we need to do is to find the nearest neighbors – that is done by computing a distance to every other vector.

```
val vectorsWithDistances = vectorsRDD.map(v => (distanceMetric.getDistance(x, v), v))
```

Then we choose k nearest vectors.

```
val kNearestVectors = vectorsWithDistances.takeOrdered(k).map(pair => pair._2)
```

And from these vectors we can compute average rating for every product and recommend those with best rating.

As you can see, operating with Spark's RDDs is easy with functional methods like *map* or *filter*. Spark RDDs also include handy methods like *takeOrdered*.

2.1.2 Clustered *k*-NN

I have taken advantage of having k-Means algorithm already implemented in Spark's MLlib. It is also very easy to use:

```
val kmeans = new KMeans().setK(numberOfClusters).setMaxIterations(numberOfKMeansIterations)
val model = kmeans.run(mllibVectorsRDD)
```

The *model* (instance of *KMeansModel*) has function *predict*, which assigns vectors to clusters and we can also get centroids from its value *clusterCenters*. The distance metric used by this algorithm is the squared Euclidean distance.

After clustering a basic k-NN recommender is created for each cluster. Recommendation then works as follows: At the beginning we choose the nearest centroid and then compute distances only to vectors in the nearest cluster.

2.1.3 Tree of clusters

The implementation follows exactly the steps of pseudocode in 1.2.5. The clustering method is the same as in the basic clustered k-NN – MLlib's k-Means. Every leaf in cluster tree has a k-NN recommender that operates with vectors in the leaf's cluster.

2.1.4 ALS

Using ALS from MLlib is as easy as using the k-Means algorithm. First step is training:

```
val model = ALS.train(trainingData, rank, numIters, lambda)
```

Model has the *predict* method, so we can predict rating for every product and then recommend the best.

```
val recommendations = model.predict(candidates.map((0, _)))
  .collect
  .sortBy(-_.rating)
  .take(numberOfRecommendedProducts)
```

2.2 Support tools

In order to verify my implementations I wrote an API using Spray framework[3] and a command line interface using Scallop library[2]. For documentation of API and the command line interface look at A.3 and A.2.

The system was designed with extensibility on mind. It should be easy to add new algorithms for recommendation, distance metrics or use this system with a different dataset. See A.4 for more information.

Chapter 3

Experimental results

3.1 Used datasets and methodology

3.1.1 Datasets

We benchmarked the implemented algorithms on two datasets – 10M MovieLens dataset and Netflix Prize dataset.

MovieLens datasets

These datasets contain movie ratings collected from users by MovieLens Site, which is ran by GroupLens Research at the University of Minnesota (`http://grouplens.org/datasets/movielens/`). You can choose from three sizes of the dataset:

- *100k* – 100,000 ratings by 1000 users on 1700 movies.
- *1M* – 1 million ratings by 6000 users on 4000 movies.
- *10M* – 10 million ratings on 10,000 movies by 72,000 users.

We used only the largest one with 10 million ratings. All the ratings are explicit on scale from one (worst) to five (best) stars, with half star increment.

Netflix prize dataset

This dataset was used in the famous Netflix Prize that took place from 2006 to 2009 (`http://www.netflixprize.com/`) and it's much larger than any MovieLens dataset. It contains 100,480,507 ratings that 480,189 users gave to 17,770 movies with file size more than 2 GB. The ratings are split into training set with 99,072,112 ratings and probe set with 1,408,395 ratings. All the ratings are also explicit with integer scale from one to five.

3.1.2 Used evaluation metrics

Time measurement

The time of one recommendation was measured by an external script and it's time from posting a request to the API till getting an answer. We have also made some measurements inside the application, to get to know which are the worst performing parts of code (for example how long does it take to find the vector that corresponds to the given user ID or how long does it take to find the nearest neighbors).

Every configuration (algorithm with given parameters) was tested by 200 requests. Every request demanded a recommendation for a user. In case of Netflix dataset, those users were chosen as a subset of users from probe set (chosen randomly before the entire measurement and then used for every experiment). The recommendation system gave 50 recommended products with predicted ratings in response to every request.

Quality measurement

These measures were applied in experiments where the Netflix dataset was used, because it was possible to determine the correct answers due to existence of the probe set.

- *Recommendation hitrate*

$$\text{hitrate} = \frac{\begin{array}{c}\text{number of responses which contain at least one product} \\ \text{that was rated by given user in probe set}\end{array}}{\text{number of requests}} \tag{3.1}$$

- *Root-mean-square error on hits*

$$\text{RMSE} = \sqrt{\frac{\sum_{t=1}^{n} (\hat{y}_t - y_t)^2}{n}} \tag{3.2}$$

  where the sum iterates over all products that were rated by a user in probe set and were recommended at the same time. y_t is the rating given by a user to the product number t (from probe set) and $\hat{y}_t$ is predicted rating of this product (from recommendation).

3.1.3 Hardware and software configuration

The recommendation system was tested on cluster Storm at Faculty of Information Technology, Czech Technical University in Prague. It consists of four executors – every of them contains twelve AMD Opteron^{TM} 6344 (2.6 GHz, 2 MB cache) cores. Eight of those 48 cores are reserved for operating system, system processes etc. The cluster has 60 GB RAM.

Used software include:

- *Apache Spark* at version 1.0.2 from 5th August 2014
- *Scala* at version 2.10.2 from 6th June 2013
- *Java^{TM} SE Runtime Environment* build 1.7.0_67-b01
- *sbt* at version 0.13.0

3.2 Results

3.2.1 *k*-NN

Number of neighbors and distance metrics

Measuring average recommendation time for Netflix dataset when using the whole cluster showed, that the computation is faster when Euclidean distance is used. The Euclidean distance is also less influenced by number of neighbors, but as you can see in figure 3.1 it has significant drawback when it comes to the quality of recommendation. The best hitrate of 36.6 % was achieved when the recommmendations were based on 10 nearest neighbors according to the cosine distance.

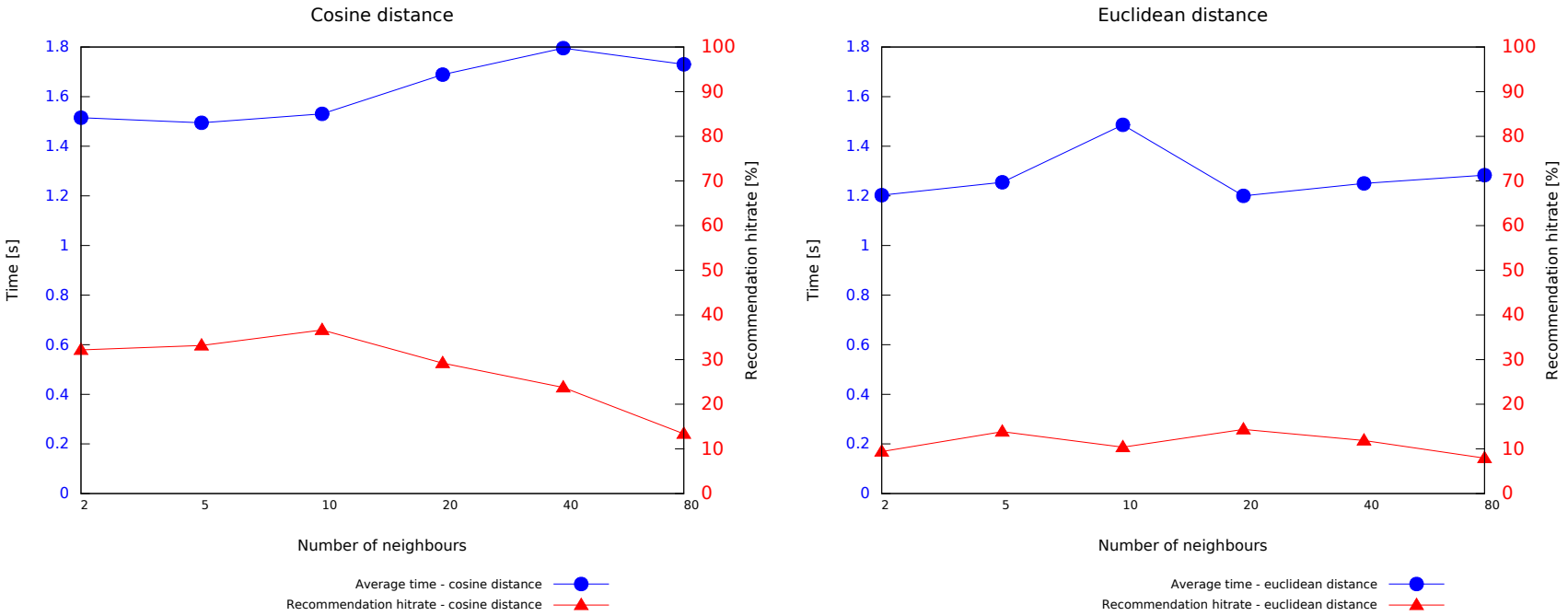

Figure 3.1: Dependency between number of neighbors and recommendation hitrate and time on Netflix dataset

Scalability

- *MovieLens 10M dataset*
  The k-NN algorithm scales with increasing number of cores sublinearly, except for difference between using just one core and two cores, where the recommendation are 2.78 times faster.

  When we compare k-NN to the MLlib's ALS with default parameters (see Spark documentation for details), then k-NN is faster on every number of cores except one core, when the ALS is faster by 0.018 second. Unlike ALS, there is also at least small improvement between using one executor and the whole cluster (by 11.4 %) when k-NN is used.

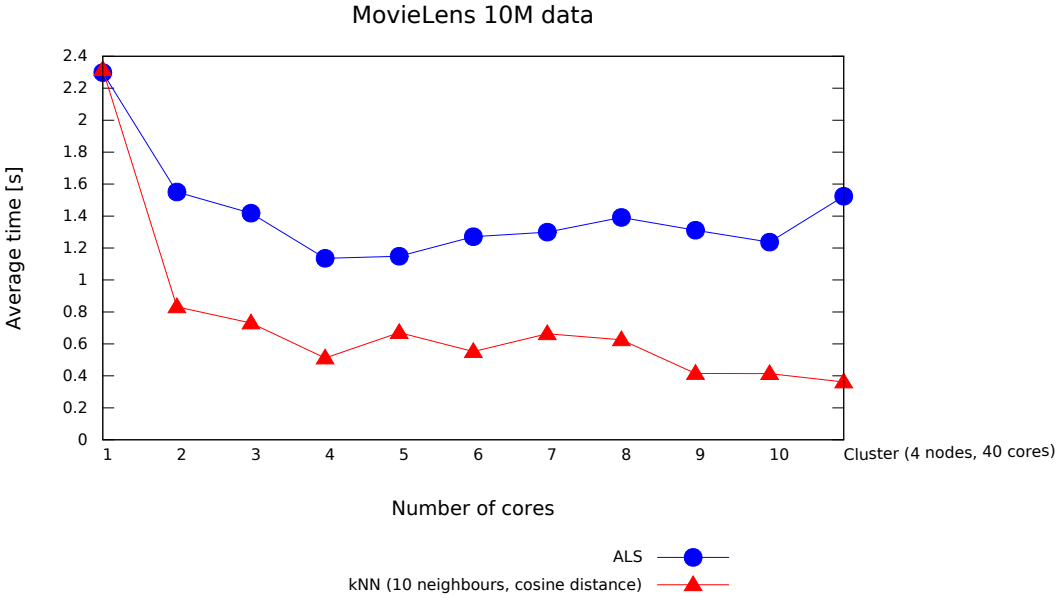

Figure 3.2: Scalability of kNN and ALS on 10M MovieLens dataset

- *Netflix dataset*
  The scalability of both k-NN and ALS is better (although still very sublinear) with larger dataset. With one exception the average time of recommendation by k-NN always drops with added core. There is also larger difference (of 67 %) between computation using 10 cores and the whole cluster. ALS fails again to speed up when the whole cluster is used. The average recommendation time of k-NN using the whole

cluster is 1.53 seconds and 87 % of this time is spent in computing distances to every user.

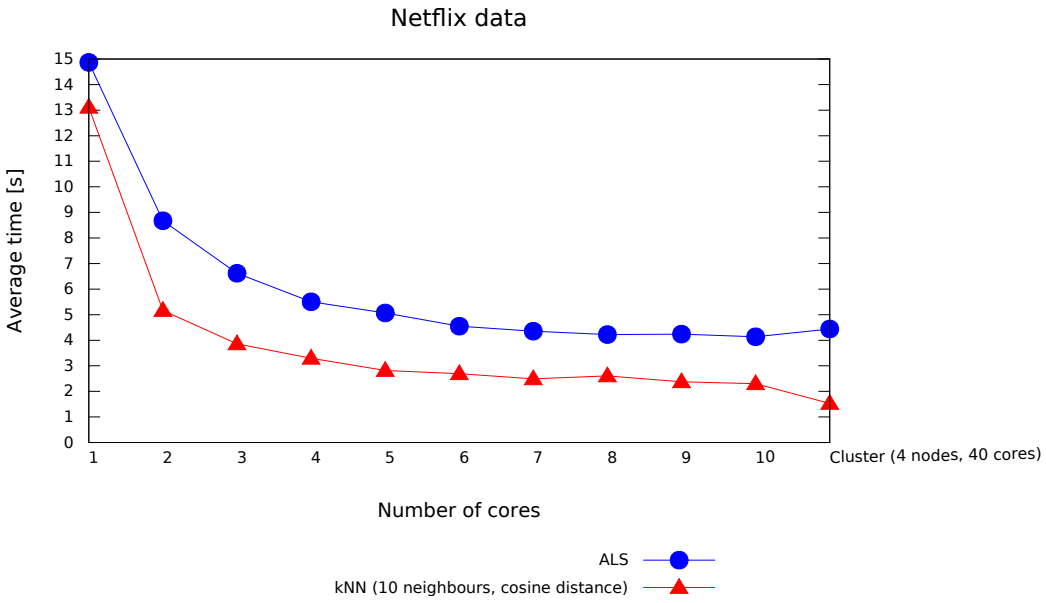

Figure 3.3: Scalability of k-NN and ALS on Netflix dataset

The recommendations given by ALS are different from those given by k-NN. It is interesting that despite being slower, ALS has very low hitrate, 4.45 %, when compared to k-NN (36.6 %). On those hits has ALS better RMSE (0.42) than k-NN (1.4).

3.2.2 Clustered *k*-NN variants

Time

Clustering decrease the needed amount of time for recommendation, but the decrease is not linear with increasing number of clusters. That's especially the case of basic clustered variant, which does not always recommend faster with increased number of clusters (see figure 3.6). That's because the cluster sizes are very unequal. For example when the users are clustered into 20 clusters, than almost half of them (42 %) lie in one big cluster (see figure 3.5), and there is one very large cluster even if the number of cluster increases. The time of recommendation is strongly affected by size of the cluster that contains given user, and because the most of the users are clustered in few big

clusters that might not change very much with increasing number of clusters (instead some very small clusters are created), the overall time doesn't have to decrease.

Unlike the recommending using basic clustering, k-NN with clusters in cluster tree always recommend faster with increased number of clusters. The speed-up is also not linear with number of clusters, because even if the largest cluster is splitted in every iteration of the algorithms, the cluster sizes are still different and there is also some overhead that is not reduced by clustering.

We have also tried algorithm that always split the largest cluster into two clusters, but doesn't form a tree. When compared to the variant with tree, then the version without tree was slower for every number of clusters (for example with 10 clusters by 0.12 seconds on average). That's because the centroids created by k-Means are dense vectors, and therefore it's computationaly difficult to get distances to those vectors, so the reduced number of those computations improve the time even in that small tree.

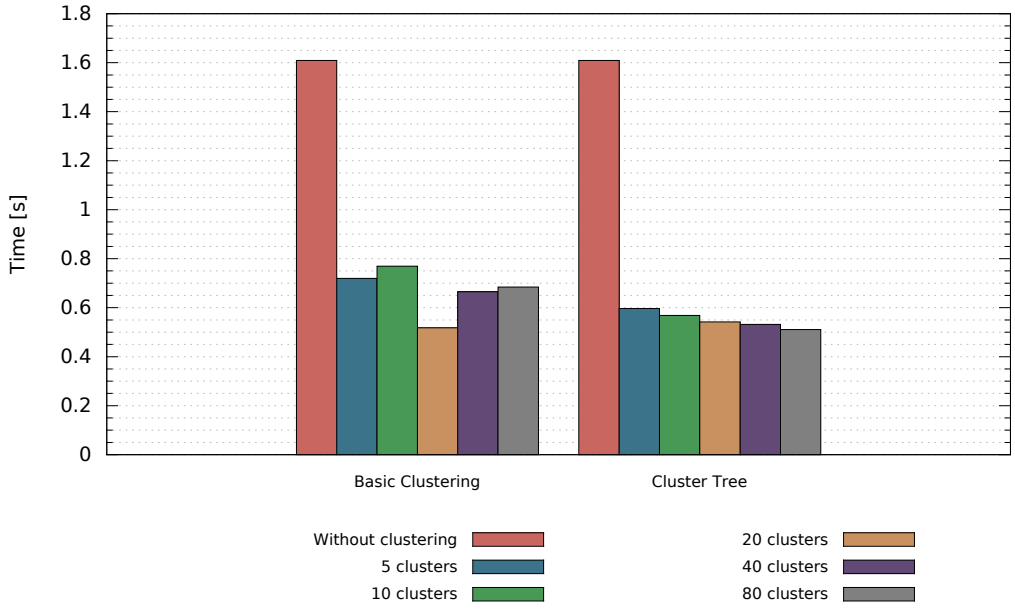

Figure 3.4: Average recommendation time for clustered variants of k-NN on Netflix dataset. For both variants number of neighbors was 10 and cosine distance was used.

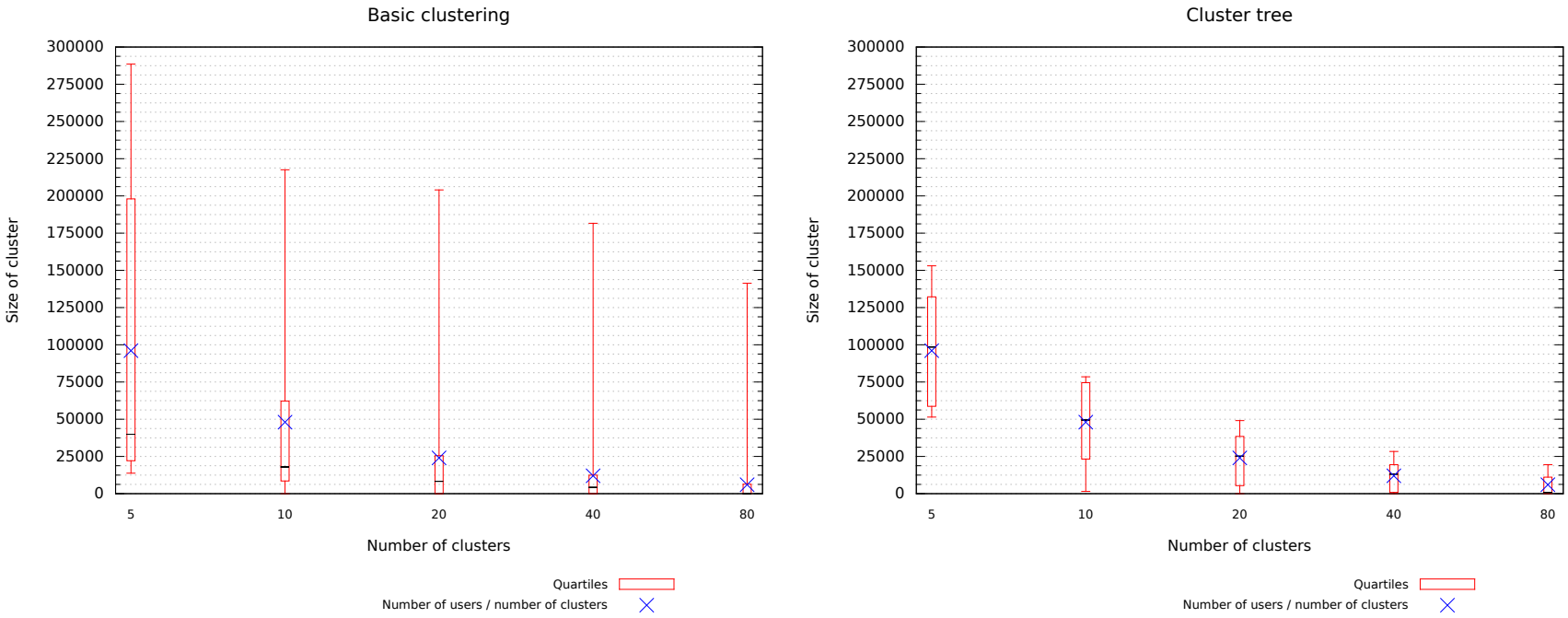

Figure 3.5: Cluster sizes on Netflix dataset. Both methods use implementation of k-Means from MLlib.

Quality

The basic clustering doesn't harm the quality of the recommendation, because the similar users are kept together – it even makes it a little bit better. The algorithm that creates the cluster tree doesn't always respect those relations in data, because it splits the biggest cluster into two smaller on every iteration, no matter what is the structure of data in that cluster. This resulted into a small decrease of recommendation hitrate. Both variants have similar RMSE on hits ranging from 1.15 to 1.44, which doesn't seem to be in relation with number of clusters.

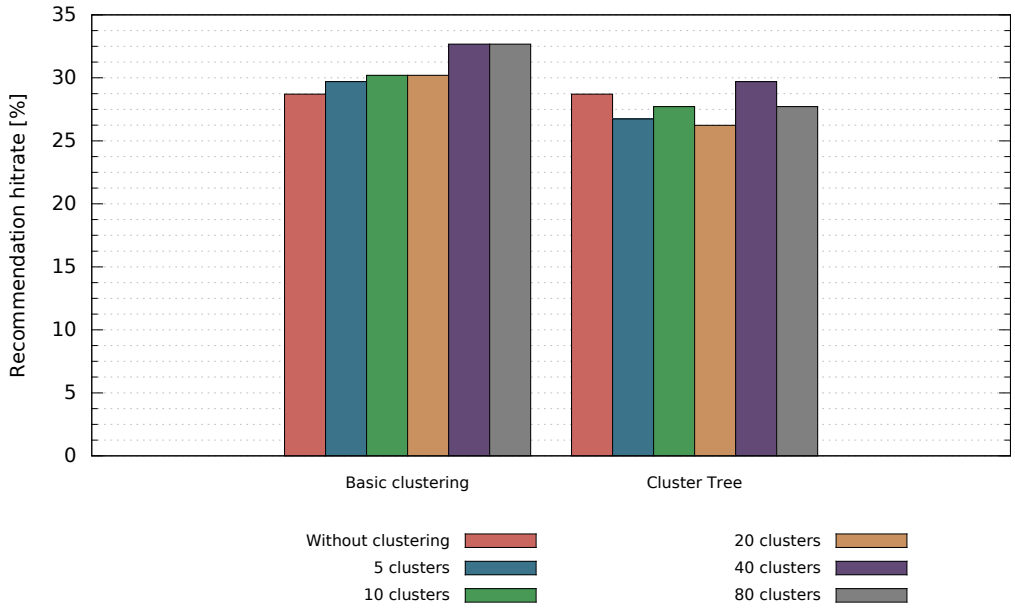

Figure 3.6: Recommendation hitrate for clustered variants of k-NN on Netflix dataset. For both variants number of neighbors was 10 and cosine distance was used.

Chapter 4

Conclusion

I have implemented a recommendation system, which include k-nearest neighbors algorithm, k-NN with clustering by k-Means and k-NN with a tree of clusters. It includes also Alternating Least Squares method (ALS) from Spark's MLlib which was used for comparsion.

The experiments using Netflix Prize dataset showed, that recommending by k-NN is faster than recommending by ALS on any number of cores. When more than one core is used, k-NN is faster at least by 40 %. k-NN has also better scalability, but it is very sublinear though. Recommendations of ALS and k-NN are different – k-NN has better hitrate but ALS has better RMSE on hits.

Clustering can decrease the needed time for a recommendation. It works even for small number of clusters – with five clusters the average time of recommendation decreased to less than half of the time that was needed by version without clustering. Unfortunately with increasing number of clusters the time didn't decrease much, because the k-Means algorithm created a lot of small clusters, but most of users kept in few largest clusters. This problem was partly solved by using a tree of clusters, which was created by recursive splitting of the biggest cluster. With this improvement time decreased with every increase of number of clusters, but it is still far from being proportional to $\frac{\text{time without clustering}}{\text{number of clusters}}$. It would be interesting to try different clustering algorithm in a future version.

The basic clustering didn't harm the quality of the recommendation, because the similar users stayed together – it even made the recommendation hitrate a little bit better. The recommendation hitrate of the version with cluster tree was lower by few percents, because the tree is created by splitting the biggest cluster into two smaller, no matter what is the structure of data in that cluster.

The average time of a recommendation on Storm cluster (which consists

of 40 CPUs, 60 GB RAM) with Netflix Prize dataset was 1.53 seconds when k-NN with cosine distance and 10 neighbors was used, and 0.51 second using k-NN with users clustered into 80 clusters in leafs of a tree and the same distance metric and number of neighbors. This amount of time is still too high for online recommendations, but it is sufficient for recommending using following scenario: When the first request for recommendation for a given user is received, system returns a default non-personalized list of products (for example current best-sellers) and compute the real recommendation. This recommendation will be returned when next request for this user arrives (which will be usually soon – for example when the user reloads the page or change the category in an e-shop).

The Spark framework is very easy to use (for example when compared to OpenMP) and the code has good readability, but the speed and scalability of algorithms built on this system is not as good as if a lower level solution was used. Therefore I would recommend to use Spark for projects that require parallel solution but the scalability can be sublinear, and very important characteristics are readability, easy deployment on many operating systems and mantainability. It is also good for creating proofs of concepts, because creating a parallel program with Spark is nearly as easy as writing a sequential one.

Appendix A

Application usage

A.1 Building and running

A.1.1 Building

The application is built with Simple Build Tool (SBT). Run command `sbt assembly` in recommender's root directory. It creates the *jar* file in directory target/scala-2.10/.

A.1.2 Running

The application can then be run using the *spark-submit* script.

```
cd target/scala-2.10/

‘$SPARK_HOME‘/bin/spark-submit --master local --driver-memory 2G \
--executor-memory 6G SparkRecommender-assembly-0.1.jar --class Boot
```

(+ parameters of the recommender)

See documentation of Spark for information about parameters of *spark-submit*. Parameters of the recommender are described in A.2.

A.2 Command line interface

A.2.1 Arguments

1. Setting up API
   - `--interface <arg>`
     Interface for setting up API (default = localhost)
   - `--port <arg>`
     Port of interface for setting up API (default = 8080)
2. Setting the dataset
   - `--data <arg>`
     Type of dataset
   - `--dir <arg>`
     Directory containing files of dataset
3. Setting the algorithm
   - `--method <arg>`
     Algorithm
   - `-pkey=value [key=value]...`
     Parameters for algorithm
4. Other
   - `--products <arg>`
     Maximal number of recommended products (default = 10)
   - `--help`
     Shows help
   - `--version`
     Shows version

A.2.2 Provided algorithms, datasets and distance metrics

Argument `method` accept one of these algorithms:

- kNN
  Parameters:
  - numberOfNeighbors = <Int> (default: 10)

  - distanceMetric = <distanceMetric> (default: cosine)

- kMeansClusteredKnn
  Parameters:
  - numberOfNeighbors = <Int> (default: 10)
  - distanceMetric = <distanceMetric> (default: cosine)
  - numberOfClusters = <Int> (default: 10)
  - numberOfClusteringIterations = <Int> (default: 10)

- clusterTreeKnn
  Parameteres are the same as in KMeansClusteredKnn.

- als
  Parameters:
  - rank = <Int> (default: 12)
  - lambda = <Double> (default: 0.01)
  - numberOfIterations = <Int> (default: 10)

  See `https://spark.apache.org/docs/latest/mllib-collaborative-filtering.html` for more information about these parameters.

Argument `data` accept one of these types of datasets:

- movieLens
  `http://grouplens.org/datasets/movieLens/`
  Ratings are stored in *ratings.dat* and movie titles in *movies.dat*.

- netflix
  `http://www.netflixprize.com/`
  All ratings in file *ratings.txt*. Each line has format:
  `<movieID>,<userID>,<rating>,<date>`

- netflixInManyFiles
  Netflix data in original format. The directory has to contain a sub-directory *training_set* with rating files.

Provided distance metric for parameter *distanceMetric* include:

- euclidean
- cosine

See 1.2.2 for definitions of these metrics.

A.2.3 Examples

Parameters

```
--data netflix --dir /mnt/share/netflix/ \
--method kMeansClusteredKnn -p numberOfClusters=10

--data movieLens --dir /mnt/share/movieLens/ \
--products 20 --method kNN \
-p numberOfNeighbors=80 distanceMetric=euclidean

--interface 127.0.0.2 --port 8070 \
--data movieLens --dir /mnt/share/movieLens/ \
--method als -p rank=11
```

Entire command

```
‘$SPARK_HOME‘/bin/spark-submit --master local --driver-memory 2G \
--executor-memory 6G SparkRecommender-assembly-0.1.jar --class Boot \
--data movieLens --dir /mnt/share/movieLens/ \
--method kNN -p numberOfNeighbors=5
```

A.3 API

API supports two operations:

- Recommend from user ID

  `host:port/recommend/fromuserid/?id=<userID, Int>`

  Example: `http://localhost:8080/recommend/fromuserid/?id=97`

- Recommend from ratings

  `host:port/recommend/fromratings/?rating=<productID, Int>,<rating, Double>`

  Example: `http://localhost:8080/recommend/fromratings/?rating=98,4&rating=176,5&rating=616,5`

The API returns the recommended products in form of array of JSON objects. The JSON object for one recommendation looks like this:

```
{
    "product" : productID
    "rating" : Prediction of rating for this product
    "name" : "Name of product"
}
```

Example of recommendation of three products:


```
{"recommendations":[
{"product":312,"rating":5.0,"name":"High Fidelity (2000)"},
{"product":494,"rating":5.0,"name":"Monty Python's The Meaning of Life: Special Edition (1983)"},
{"product":516,"rating":4.0,"name":"Monsoon Wedding (2001)"}
]}
```

A.4 Extending the application

A.4.1 Add new algorithm

A class of new algorithm for recommendation must implement trait *Recommender*.

If you need just this algorithm, than you can hard-code it to *setUp* method of *MainHolder* object.

If you want to use it via command line interface, then those steps have to be done:

1. Create an object extending *RecommenderFactoryFromConf* trait. This object has to implement method *getRecommender(conf: Conf)* that returns instance of your recommender. Settings for the algorithm can be extracted from *conf*, which contains given command line parameters.

2. Add the object extending the *RecommenderFactoryFromConf* trait to List *recommenderFactories* of *RecommenderFactoryFromConf* object.

Now you can use it from command line interface and it also shows up in help.

A.4.2 Add new dataset

Adding new dataset works like adding a new algorithm. The object describing a dataset must implement the *DataHolder* trait. It can be set in *setUp* method of *MainHolder* object, but for usage with command line interface you have to create an object extending *DataHolderFactoryFromConf* trait (with *getDataHolderInstance* method) and add this object to List *dataHolderFactories* of *DataHolderFactoryFromConf* object.

A.4.3 Add new distance metric

The object for computing distance between vectors must implement the *DistanceMetric* trait. Add this object to List *distanceMetrics* of object *DistanceMetric* for usage via command line interface.

Bibliography

[1] *Apache Spark Homepage* [online] [cit. 2014-08-11]. `https://spark.apache.org/`

[2] *Scallop gitHub* [online] [cit. 2014-09-01]. `https://github.com/scallop/scallop`

[3] *spray Homepage* [online] [cit. 2014-09-01]. `https://http://spray.io/`

[4] Kelleher, J.; Bridge, D.: RecTree Centroid: An accurate, scalable collaborative recommender.

[5] MacQueen, J. B.: Some Methods for Classification and Analysis of Multi-Variate Observations. In *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, editace L. M. L. Cam; J. Neyman, University of California Press, str. 281–297.

[6] Rashid, A.; Lam, S.; LaPitz, A.; aj.: Towards a Scalable kNN CF Algorithm: Exploring Effective Applications of Clustering. 2007, doi:10.1007/978-3-540-77485-3_9. `http://dx.doi.org/10.1007/978-3-540-77485-3_9`

[7] Zhou, Y.; Wilkinson, D.; Schreiber, R.; aj.: Large-Scale Parallel Collaborative Filtering for the Netflix Prize. In *AAIM '08: Proceedings of the 4th international conference on Algorithmic Aspects in Information and Management*, Berlin, Heidelberg: Springer-Verlag, 2008, ISBN 978-3-540-68865-5, str. 337–348, doi:10.1007/978-3-540-68880-8_32.