Netflix Movie Recommendation

Team Members:
Jalpan Randeri
Madhav Khosla
Sumit Nair
Sweta Patel

INTRODUCTION

The global box office for all films released in each country around the world reached to $35.9 billion in 2013/2014. There are around 4 million movie titles on IMDB. Using this big data, statistical analysis can be performed to improve customer experience for streaming websites. Netflix is using this technique to help users find movies and display movies, which are best, match for them. This can help Netflix to improve user satisfaction that can lead to long-term subscription from users.

The Netflix Prize was an open competition for the best collaborative filtering algorithm to predict user ratings for films, based on previous ratings without any other information about the users or films. Netflix began their contest to find a more accurate movie recommendation system to replace their current one.

- Billions of data transfer task:
  - When identifying class of user (user with similar taste) using K-Nearest Neighbour clustering algorithm, the problem was transferring billions of data from mapper to reducers.
  - There is approximately one billion training data and when finding in which cluster a user belong, there was a need to transfer one billion data from mapper and reducer and then select top k users similar to the test user. As we know that, transferring one billion data will significantly affect running time of the program. We had to choose a clever strategy to do that.
  - Solution: We decided to solve this problem by selecting local top k results by using max heap data structure in the mapper and then emitting local K winners. We also used secondary sorting design pattern to optimize the program. This way we reduced data transfer significantly and which in turn reduced running time of our program to a great extent and improved scalability.

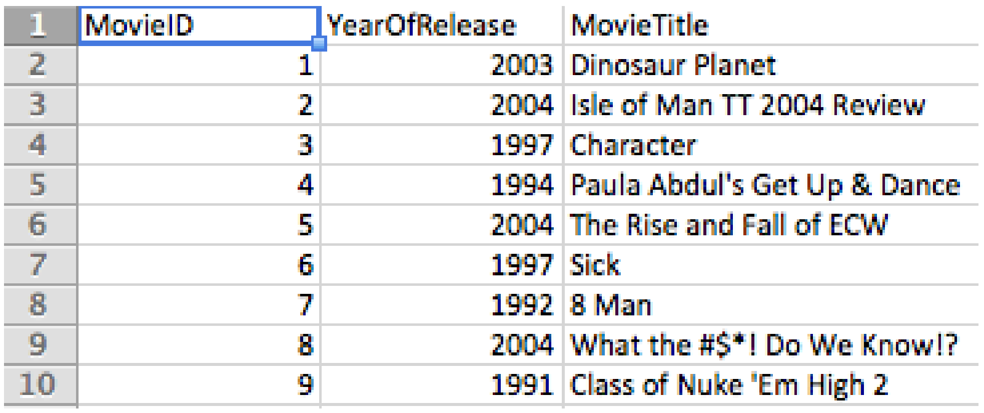
- The major issue with dataset was missing data in movie_titles.txt and dataset.csv file. We use the data imputation techniques to fill this data. We find the median value for the missing columns and impute them.

- One major problem we ran into was that we were trying to clean and organize the data from the dataset into files. But as all the map tasks required these files, there was a lot of overhead for opening these files and reading the large quantity repeatedly. Thus, we finally decided use a HBase as our datastore.

DATA

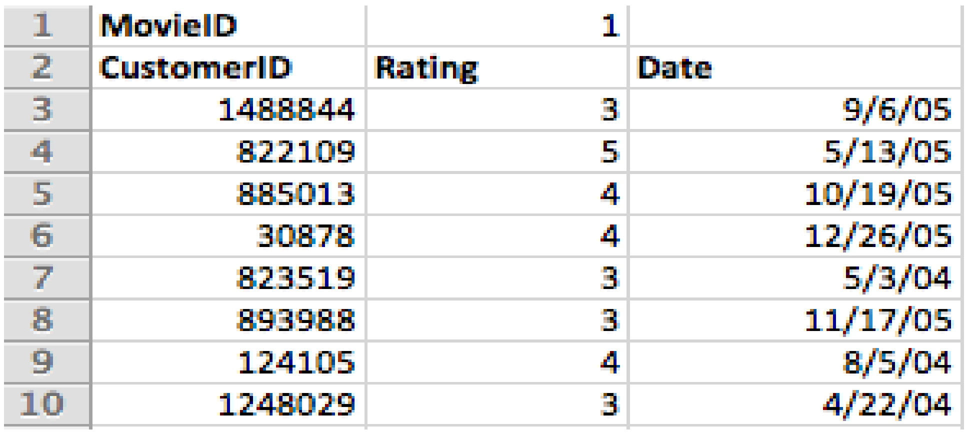
Netflix Prize Dataset, on which we worked, was obtained from the following link. The dataset contains about 100 million ratings from 480 thousand randomly chosen, anonymous Netflix customers over 17 thousand-movie titles. The data were collected in the time period of October 1998 to December 2005 and it also reflects the distribution of all ratings received during this period. Ratings are categorized from 1 to 5 stars.
In order to protect the privacy of customers, no information at all is provided about users. In the training data set, the average user rated over 200 movies and the average movie was rated by over 5000 users. Data set provided wide variety of data in the sense that some movies in the training set have as few as 3 ratings, while one user rated over 17,000 movies.

Dataset provides data in the following format. One file contains information about movies in the tuples as <MovieID, YearOfRelease, MovieTitle>.



1	MovieID	YearOfRelease	MovieTitle
2	1	2003	Dinosaur Planet
3	2	2004	Isle of Man TT 2004 Review
4	3	1997	Character
5	4	1994	Paula Abdul's Get Up & Dance
6	5	2004	The Rise and Fall of ECW
7	6	1997	Sick
8	7	1992	8 Man
9	8	2004	What the #$*! Do We Know!?
10	9	1991	Class of Nuke 'Em High 2

Training dataset contains many files which describes information about movie ratings in the tuples as <CustomerID, Rating, Date>.



1	MovieID	1	
2	CustomerID	Rating	Date
3	1488844	3	9/6/05
4	822109	5	5/13/05
5	885013	4	10/19/05
6	30878	4	12/26/05
7	823519	3	5/3/04
8	893988	3	11/17/05
9	124105	4	8/5/04
10	1248029	3	4/22/04

TECHNICAL DETAILS

1. HBase as Index vs Replicated Join:

a. Purpose of Task:

- The main purpose of this task is to modify the dataset so that we can run the data mining tasks efficiently. In this task we are calculating average watch year and average release year for each user.
- We need to join movie_titles.txt and dataset.csv on movie_id to determine average release year dataset to create the netflix dataset.
- We analysed the performance of Replicated join with HBase as Index and join the two dataset to create netflix dataset.
- In the other approach we are first inserting data from both the files into HTable and then calculating desired result and storing into HTable.
- Performance comparison for both approaches is described in the below section.
- In this task we are using HBase as an index for an equi-join implementation on movie Id.

b. Pseudo Code:

```
map() {
    1. Created composite key consisting of customer id and movie id.
    2. This composite key can be used for secondary sorting based on the movie
       id, this helps in knn and kmeans calculations.
    3. emit([CustomerId, MovieId], [Rating, WatchYear, ReleaseYear])
}


// Composite Key
MovKey {
        public String user;
        public String movie_id;
            1. Custom key used for populating the hbase table consists of
               customerid and mov_id.
            2. overrides methods equal, hashcode, compareTo , for secondary
               sorting
  }


  // Grouping comparator
  UserGroupper(){
```

```
            1.  Grouping comparator which ensures that all the keys with same
                customer_id goes to the same reduce call
               }
      // Key Comparator
  KeyComparator() {
           1.  sorts according to movie id and customer id
  }

  Reducer([CustomerId, *],[Rating, WatchYear, ReleaseYear]) {
      protected void reducehelper() {
              For each in list:
                  1.  Calculate average watch year and average release year
                  2.  Append movie in the list
                  3.  Insert data into HTable(RowKey as customer Id)
      }
  }
```

c. Related File:
Replicated.java, HPopulateMovie.java

d. Final Conclusion

Type	5 large machine
Hbase as Index	3 mins 12 secs
Plain MR	5 mins 46 secs

- From the above table we can clearly conclude that Hbase as Index is faster than Plain MR. This happens because the Plain Map reduce job requires file IO while HBase as index quiers the data from HBase so it can read it in parallel and the data comes in sorted format so we can take advantage of this to boost the computation. This optimization clearly seen in running times.

2. Clustering of users (K Means algorithm):

a. Purpose of Task:

- The purpose of this task is to cluster users based on their taste of movies. Clustering is performed using average release year and average watch year.
- This clustering is useful in the next step when, we need to identify that new user belongs to which group.
- We have used KMeans clustering algorithm to find all the possible clusters.

b. Pseudo Code:

```
 map() {
    1. for each centroid in centroid_table
    2. find the closest cosine distance from centroid to the
       this given user.
    3. emitt ( centroid, user_id)
}

reduce(centroid, [user_id1, user_id2, ...])  {
        1. Get the median element for the given centroid
        2. Update the centroid with the median element.
        3. Insert the new centroid in the Hbase table
}
public class KMenasUserClustering{
        1. Pick random k initial random users from dataset name it as centroids
        2. Run the job using initial centroids
        3. Get the new centroid at the end of job.
        4. Determine if the new centroid are same as the previous centroid
        5. If both centroids are not same then go to step 2 as set centroid as
        new centroids
}
```

c. Related File:

KMeansUserClustering.java

d. Final Conclusion

Type	5 large machine	6 large machine
KMeans	14 mins 16 secs	10 mins 12 sec

3. K nearest neighbour computation:

a. Purpose of Task:

- After generating clusters of users with similar taste, it is now time to evaluate taste of a new user which is done by finding to which cluster, new user belongs to.
- We have used K nearest neighbour algorithm to find out the taste of new user based on movies users have watched.
- After end of this step, we are able to classify each test user into particular clusters.
- The approach of the KNN implementation is to replicate the test set to all the machines and then we scan the HBase table and find the cosine distance from each user in training set.
- We used multiple approaches to implement K nearest neighbours like the basic naive implementation, then we added secondary sort to speed up the execution time. Finally, we added Max Heap implementation for finding and emitting the local winners.
- Below is the pseudo code for the third approach used :

b. Pseudo Code:

Mapper:

```
public void map extends table mapper()  {
       protected void setup(Context context) {
              1. Initialize Max Heap Data Structure for every test user
        }

       protected void map() {
              1. For each user in test set calculate the distance from the
              current user and add it to Max Heap.
       }

       protected void cleanup(Context context) {
              1. For every user in test file get the corresponding heap
              2. emit([user_id, distance], cluster_membership)
       }

       protected void heap_insert(){
           if(heap.size == K) {
             remove the top element
           }
            insert the element in heap
       }
}
```

```
Secondary Sort : { first sort on user_id and then sort ascending on distance
value)

Grouping : { Group user based on user_id };
```

Reducer:

```
public void reduce([user_id,*],[cluster_membership, ..]) {
            1.  read the first K elements into the list
            2.  get the most occurring membership among this list
            3.  Insert into hbase table (user_id, flag)
}
```

c. Related File:
KnnUserMatcherOptimized.java

d. Final Conclusion

Type	5 large machine	10 large machine
Naive	4 hrs 35 mins	3 hr 16 mins
Secondary Sort	3 hrs 7 mins	3 hr 0 mins
Optimized with Secondary Sort	5 mins	2 mins 17 sec

- The **Naive** Implementation just calculate the distance for each training instance and emit them to reducers. Reducer will sort the distance and take the top K records to determine the resultant class. This is clearly seen in the program spent most of time in sorting and transmitting data from mapper to reducers.
- We come up with **Secondary sort design pattern** to optimize the code. Instead of sorting the data in reducer we take advantage of mapreduce shuffle sorting phase. This gives us only slight improvement of 3 hr 7 min. This is not so scalable.
- After carefully considering the algorithm of K Nearest Neighbour. It comes to us that we actually do not need to send all the distance from mapper to reducer. In map we just need to identify local winner and emit them to reducer. We implemented this technique using **Max Heap**. We also included the secondary sorting. This turn out to be major improvement as clearly seen in running time of only 35 mins. This make KNN scalable.

4. Generate Recommendation:

a. Purpose of Task:

- Now we have clusters and a field indicating to which cluster a test user belongs. Now we are giving recommendation of movies to a test user.

- We are doing this by performing join on two tables and retrieving list of all movies watched by all users who belongs to the same cluster.

b. Pseudo Code:

```
Mapper() {

    setup(){
       Cache the test users, cluster_membership_id, seen_movie_id
    }

    map(Cluster_id, Movie_id) {
       for each cached user {
            if( test_user.cluster_membership_id belongs == Cluster_id){
                if(test_user.seen_movie_id != movie_id){
                      emmit ( test_user_id , movie_id)
                }
            }
       }

    }
}

Combiner(){
   reduce(){
      combine similar test_user_id, movie_ids
   }
}

Reducer(Cluster_id,[movie_id1, movie_id2, ...]){
   setup(){
     cache movie_table into hashmap (movie_id, movie_name)
   }

   reduce(user_id, List<Movie_id> values){
      for(each movie_id : values){
        ans = ans + getMovieName(movie_id);
      }

      emmit (user_id, ans)
   }
}
```

c. Related File:
MoviePredictor.java , prediction.hiverc

d. Final Conclusion

Type	5 large machine	10 large machine
Plain Map Reduce Job	12 min	3 min
Hive	20 min	5 min

- Hive provides the sql query like support to implement joins. And Hive also have hive optimizer. so we expected the hive program will outperform the plain map reduce job. But as It turns out hive optimizer only optimize certain task however the well tuned Java program for mapreduce will outperform the hive query. As clearly seen in pseudocode the plain java code will take advantages of Map side join as well reduced side join with the optimization of in mapper combining techniques.
- This optimization helps the plain map reduce job to outperform the hive optimizer. This is backed by the running time of these two programs.

SETUP CHALLENGES (HIVE)

1. Understanding the execution patterns of Hive queries and the consequences for resource consumption
2. Lack of certain SQL elements and functions in hive made multi table join tasks difficult
3. Hive required exporting auxiliary jars which required lot of effort in zeroing down on hive setup.

CONCLUSION

The major extension would be to implement the Ensemble method to boost the accuracy. As the single Collaborative filtering we implemented can be improved with this technique.

Issue with Skewed Data (Load Balancing in K Means Algorithm)

During the implementation of K Means Algorithm the major problem we faced was to come up with the proper distance formula to group the user into proper clusters. We initially used the **cosine similarity** to identify the users but as explained in the class the final matrix distribution of data with respect to this distance measure was very skewed. It is very difficult to split the dataset among the different nodes as explained in the load balancer topic. We tried different techniques and we came up with the **euclidean distance** measure between two users using the **average_watch year** of movies users have seen and **average release year** of those movies and **average rating**. This measure have equally divide the data into **sparse matrix**. This give us an advantage to make the program scalable. We have implemented the fully parallel K Means algorithm. We can further improve this algorithm by making this algorithm pseudo parallel to remove the redundant data transfer between each iterations. To slightly improve upon this we used HBase as temporary solution.

Knn Algorithm Optimization.

We implemented the The KNN algorithm from the Data Mining 3rd edition book. This algorithm was designed for sequential machine. so we faced an issue with the limitation in scalability as well as runtime performance. The Naive solution could not able to use the Mapreduce Framework efficiently . As the naive version creates lots of data transfer between mapper to reducers. **This eventually flooded one reducer task with entire dataset**. We identified this issue and tried to think out of some clever data structure to solve this. We implemented the **Secondary sort** and instead of emitting the entire data from mapper to reducers we emitted only local winner and to do that we used **Max Heap.** This enables us a huge runtime performance and scalability.

<u>Replicated Join vs HBase as Index</u>

One of the major issues we faced was that the HTable that we created was not being connected. We went through the source code and found that that the getTable method used in the Interface was deprecated in the new version which was running on the AWS cluster. We used the create method of HConfiguration interface instead to interface with the table.

Another issue that we faced overall throughout the project was the multiple logger binding issue and we had to go through the Slf4j logger library implementation to understand the multiple binding issue.