Codelab 4
K-Means Clustering of Netflix Data (Hadoop version 0.17.0)

Goals:

1. Understand the K-means and canopy clustering algorithms and their relationship
2. Implement these algorithms in MapReduce
3. Analyze the effect of running these algorithms on a large data set

Clustering Algorithms And Netflix:

Canopy Clustering:
http://net.pku.edu.cn/~course/cs402/reading/canopy-kdd00.pdf
K-means Clustering: http://en.wikipedia.org/wiki/K-means_algorithm
Netflix is a company that allows users to rent DVDs by mail. Users select movies on the Netflix website, and their selections are mailed to them. Based on your viewing history, Netflix will recommend other movies to you. The algorithm Netflix uses is based on what other people watched and liked, after watching the same movies you have. So if you watch movie A, and others who watch movie A also watch (and like!) movie B, Netflix will recommend that you watch movie B.

Netflix has made a large amount of movie recommendation data available on the Internet. The data files are a set of lines of the form:
"movieId, userId, rating, dateRated"

K-means clustering is one of the simplest clustering algorithms. It is called k-means because it iteratively improves our partition of the data into *k* sets. We choose *k* initial points and mark each as a center point for one of the *k* sets. Then for every item in the total data set we mark which of the *k* sets it is closest to. We then find the average center of each set, by averaging the points which are closest to the set. With the new set of centers we repeat the algorithm.

The problem with this algorithm is that it is not scalable to large sizes: as k gets large, each of the (large number of) data points must be compared to k different possible centers. So we use another, faster, process to partition the data set into reasonable subsets: Canopy clustering.

Canopy clustering works as follows: We create a set of overlapping "canopies" (regions) wherein each data item is a member of at least one canopy. This can be evaluated in one pass. We then ensure that at least one k-center is in each canopy. We only match data points with possible k-centers found in the same canopies as the data points. This greatly reduces the amount of computation involved. The trade-off is that once a data point has been assigned to canopies, it can never be bound to k-centers in different canopies, even if they would have been better global

choices.

MapReduce Steps:

This presents the high-level requirements of what each MapReduce step of the program should do. This is the most involved process seen yet; make sure you understand the requirements of each step of the dataflow before you begin programming.

Step 1: Aggregate Data

Given the input data format described above, turn this into a key consisting of the movieId, and a value consisting of a list of <userId, rating> pairs. The Mapper should parse each line of input data and map each movieId to a userId & rating. The reducer should create a list of all the <userId, rating> pairs it receives for a common movie. The input and output will all be represented by Text objects: pick a format for your textual data that you can easily parse in future steps.

You may wish to create a class that represents a movie & its associated data. Create methods which read in a textual description of a movie, and write one back out. You can then use this class to better organize the following data processing steps.

Step 2: Choose Canopies

This step determines where in the space canopy centers should be located. It does not assign individual points to canopies -- that is the next step; it simply establishes where the canopies are to start with.

Each mapper receives some portion of the input movies. The mapper maintains a list in memory of canopies it has generated so far. If the current movie being mapped is within some "near" threshold of an existing canopy it has generated, then do nothing. Otherwise, the current movie is a new canopy center: emit it as an intermediate value and add it to the already-created list.

The reducer does the same thing: We take all the canopy center candidates generated by all of the mappers and then perform the same process to ensure that we do not generate two canopy centers that are basically on top of one another. For this to work, we will need to set the number of reducers to 1. (JobConf.setNumReduceTasks())

The metric we use to determine distance for canopies is the number of userIds two rated movies have in common. For the "too close" threshold, try using 10 or more userId's in common.

Step 3: Assign Movies to Canopies

Given the set of canopies, map over your movie data assembled in step 1; each value should now include a set of canopies under which the movie falls. Use the same distance metric as in step 2.

Each mapper will need to load the set of canopies generated in step 2. Use a SequenceFile.Reader to read the file generated by step 2 in the configure() method of your mapper class for this step.

Step 4: K-Means Iteration

This Mapper/Reducer pair is iterable, just like the main MapReduce pass of PageRank. The mapper receives a movie, its user/rating pairs, and its canopies (the output of step 3). It determines which of the current set of k-centers it is closest to (from among the k-centers in the same canopy as the movie) and emits a record containing all the movie's data and its chosen k-center. The value sent to the reducer should be keyed by the k-center (which is a movieId).

The reducer receives a k-center and all movies which are bound to that k-center. It should calculate the new position of the k-center. The giant list of movies is not updated by this step: the input from step 3 is always the only data used as input to each k-means iteration step. The purpose of this step is to move the k-centers around. So given the list of movies bound to a current k-center, we determine which movieId is the new center of the cluster. We then emit this movie's record as the new k-center.

Pick a small number of reducers for this (5--8).

Each Mapper should load the current list of k-centers in its configure() phase. It should also load the list of canopies to determine which canopies each k-center falls in. For the first iteration, each canopy center can be considered a k-center. Remember to use the canopy distance metric for this. Also note that the list of k-centers is split across as many files as there were reducers in the previous round. Make sure that you load all the files.

The K-means Distance Metric:

The set of ratings for a movie given by a set of users can be thought of as a vector [user1_score, user2_score, ..., userN_score] (our actual representation of this vector will be sparse). If a movie has not been rated by some set of users, those scores are held at zero. To evaluate the distance between two movies, we will use the *cosine vector similarity* metric.

The cosine vector similarity metric is expressed as follows:

For vectors A and B each of length 'n', similarity(A, B) = sum(A_i * B_i) / (sqrt(sum(A_i^2)) * sqrt(sum(B_i^2)))
where the sum(...) functions retrieve all A_i or B_i for 0 <= i < n

Note that for the numerator, if a given user has not rated one of the movies, the

rating for the other movie is not considered, because the term will be zero. The similarity metric will return a value between 0 and 1. The higher the value, the more similar (nearer) the vectors are to one another. Use double-precision floating point values for this computation.

Finding the New Center:
Finding the new center is a two step process. First, compute the term-wise average vector from all the movies mapped to a given center. Then find the movie most similar to this average vector.

Efficiency note:When the new centers are found by averaging the vectors we found it necessary to minimize the number of features per vector, allowing them to better fit into memory for the next map. To do this we picked the top hundred thousand occurring userIDs per movie and only output the average of those user ratings.

Step 5: Cleanup
The final phase should output results for you to use. The output should list each k-center as a key, with the value being the list of movieIds attached to that center (use the mapper from step 4 to bind movieIds to a k-center). This is again like the cleanup step of PageRank, where we inverted our keys/values to sort by the related data.

CodeLab Exercise:

Implement the clustering algorithm described above. You will need a driver class to run this process, which should run the steps in order, calculate k-means for 10 iterations, and then run the cleanup pass.

Overall advice:

- Our local copy of NetFlix data is stored in "/netflix" on the DFS. Use this as your first input directory. Do NOT use it as your output directory.
- SequenceFiles are a special binary format used by Hadoop for fast intermediate I/O.The Input- and OutputFormats for each of your intermediate steps of the process should all be set to org.apache.hadoop.mapred.SequenceInputFormat and SequenceOutputFormat.
- Test running a single pass of the k-means mapper/reducer before putting it in a loop
- Each pass will require its own input and output directory; one output directory is used as the input directory for the next pass of the algorithm. Set the input and output directory names in the JobConf to values that make sense for this flow.
- Create a new JobClient and JobConf object for each MapReduce pass. main() should call a series of driver methods.
- Remember that you need to remove your intermediate/output directories between executions of your program.

- Use reporter.SetStatus() and incrementCounter() to report basic debugging information, such as the number of k-centers loaded into memory, the number of points mapped so far, features per vector, etc.
- If you need help with particular methods or classes, see the API Javadoc: http://net.pku.edu.cn/~course/cs501/2008/resource/hadoop-0.17.0_docs/api/index.html