CS757 FINAL PROJECT

AHMED ALABDULLAH

AARON LEE

Large-Scale Clustering

with Canopy and Hierarchical Clustering

**1.** **SUMMARY:**

Our project is an implementation of clustering, specifically the hierarchical clustering algorithm with canopy clustering as a preprocessing step.

Following initial data cleansing, the algorithm operates in two major steps. The first step is the creation of canopies, which are loosely-defined clusters whose members may be part of several canopies at once. Those canopies are then sub-clustered by hierarchical clustering into the final clusters.

Canopies provide a major improvement over direct hierarchical clustering, as it speeds up the processing for large datasets. Quickly drawing out the initial shape of clusters before hierarchical clustering is employed with a more rigorous distance metric to outline the final clusters.

**2.** **DATASET:**

We used the MovieLens dataset, we utilized all three variations of the dataset but our primary focus was the 10M dataset.

**2.1 MUNGING:**

Some preprocessing was required to massage the data into a format suitable for the algorithm’s requirements.

As a first preprocessing step, movies that rarely feature in the ratings dataset were thrown out as their effect on the results was negligible.

**2.1.1: Remove Rarely Occurring Data**

After examining the distribution of ratings across movies. We determined a good cutoff point would be around the 35-45 percentile mark for the dataset (Figure 1).

**10K data set**

percentile=5, counts[=1](http://piratepad.net/ep/search?query=1)

percentile=10, counts[=2](http://piratepad.net/ep/search?query=2)

percentile=15, counts[=3](http://piratepad.net/ep/search?query=3)

percentile=20, counts[=5](http://piratepad.net/ep/search?query=5)

percentile=25, counts[=6](http://piratepad.net/ep/search?query=6)

percentile=30, counts[=9](http://piratepad.net/ep/search?query=9)

percentile=35, counts[=12](http://piratepad.net/ep/search?query=12)

percentile=40, counts[=16](http://piratepad.net/ep/search?query=16)

percentile=45, counts[=21](http://piratepad.net/ep/search?query=21)

percentile=50, counts[=27](http://piratepad.net/ep/search?query=27)

percentile=55, counts[=34](http://piratepad.net/ep/search?query=34)

percentile=60, counts[=43](http://piratepad.net/ep/search?query=43)

percentile=65, counts[=51](http://piratepad.net/ep/search?query=51)

percentile=70, counts[=65](http://piratepad.net/ep/search?query=65)

percentile=75, counts[=80](http://piratepad.net/ep/search?query=80)

percentile=80, counts[=100](http://piratepad.net/ep/search?query=100)

percentile=85, counts[=127](http://piratepad.net/ep/search?query=127)

percentile=90, counts[=169](http://piratepad.net/ep/search?query=169)

percentile=95, counts[=227](http://piratepad.net/ep/search?query=227)

**1M data set**

percentile=5, counts[=2](http://piratepad.net/ep/search?query=2)

percentile=10, counts[=7](http://piratepad.net/ep/search?query=7)

percentile=15, counts[=14](http://piratepad.net/ep/search?query=14)

percentile=20, counts[=23](http://piratepad.net/ep/search?query=23)

percentile=25, counts[=33](http://piratepad.net/ep/search?query=33)

percentile=30, counts[=44](http://piratepad.net/ep/search?query=44)

percentile=35, counts[=58](http://piratepad.net/ep/search?query=58)

percentile=40, counts[=74](http://piratepad.net/ep/search?query=74)

percentile=45, counts[=97](http://piratepad.net/ep/search?query=97) <----

percentile=50, counts[=123](http://piratepad.net/ep/search?query=123)

percentile=55, counts[=153](http://piratepad.net/ep/search?query=153)

percentile=60, counts[=187](http://piratepad.net/ep/search?query=187)

percentile=65, counts[=227](http://piratepad.net/ep/search?query=227)

percentile=70, counts[=278](http://piratepad.net/ep/search?query=278)

percentile=75, counts[=350](http://piratepad.net/ep/search?query=350)

percentile=80, counts[=427](http://piratepad.net/ep/search?query=427)

percentile=85, counts[=549](http://piratepad.net/ep/search?query=549)

percentile=90, counts[=722](http://piratepad.net/ep/search?query=722)

percentile=95, counts[=1045](http://piratepad.net/ep/search?query=1045)

**10M data set**

percentile=5, counts[=5](http://piratepad.net/ep/search?query=5)

percentile=10, counts[=11](http://piratepad.net/ep/search?query=11)

percentile=15, counts[=17](http://piratepad.net/ep/search?query=17)

percentile=20, counts[=26](http://piratepad.net/ep/search?query=26)

percentile=25, counts[=34](http://piratepad.net/ep/search?query=34)

percentile=30, counts[=44](http://piratepad.net/ep/search?query=44)

percentile=35, counts[=59](http://piratepad.net/ep/search?query=59)

percentile=40, counts[=78](http://piratepad.net/ep/search?query=78)

percentile=45, counts[=101](http://piratepad.net/ep/search?query=101)

percentile=50, counts[=135](http://piratepad.net/ep/search?query=135)

percentile=55, counts[=177](http://piratepad.net/ep/search?query=177)

percentile=60, counts[=232](http://piratepad.net/ep/search?query=232)

percentile=65, counts[=320](http://piratepad.net/ep/search?query=320)

percentile=70, counts[=440](http://piratepad.net/ep/search?query=440)

percentile=75, counts[=622](http://piratepad.net/ep/search?query=622)

percentile=80, counts[=920](http://piratepad.net/ep/search?query=920)

percentile=85, counts[=1439](http://piratepad.net/ep/search?query=1439)

percentile=90, counts[=2353](http://piratepad.net/ep/search?query=2353)

percentile=95, counts[=4378](http://piratepad.net/ep/search?query=4378)

*Figure 1: Distribution of movies in the ratings across the MovieLens dataset*

*example: in the 10KData set, 40% of movies have at most 16 ratings.*

This produced a significant decrease in the number of movies, across all three datasets at least 35% of movies were discarded, but this was a mere blip as far as the ratings data is concerned as the 10M set lost less than 1% of its ratings, the loss for the 1M and 100K datasets was approximately 4%.

In the accompanying code description, this result is achieved by the file RemoveRareRatings.java, which does not run as a Map-Reduce job but as a simple Java program.

**2.1.2: DATA MASSAGING**

Next, the data was formatted into a structure that would prove useful for the algorithm.

So far, the data is still in the original format of the MovieLens dataset:

*userID::movieID::rating::timestamp*

Figure 2.1: MovieLens dataset format

We convert this format using the following process:

We omit all timestamp information as it’s not useful to us.

We aggregate the ratings of one user into a single line, this will be a single “data point” in our data set.

*userID (tab separator) ratings-vector*

Figure 2.2: Massaged input

The ratings vector for each user has the following structure:

*movieID1:rating1, movieID2:rating2, movieID3:rating3*

We also normalize the data from a 5-star scale into a 10-star scale.

The code responsible for this part of our program is the Munger.java class. Which runs as a Map-Reduce job.

**2.1.3** **DATA NORMALIZATION:**

In the MovieLens dataset, the ratings use a 5-star scale, while the 10K and 1M dataset use increments of 1-star, the 10M dataset uses 0.5 increments. Because we are going to employ Jaccard Distance on bags as our first distance metric, which expects whole numbers. The 5-star scale had to be normalized into a 10-star scale. Every rating is simply multiplied by 2 to get the normalized output. This process is done in the Munger.java class.

3. **CLUSTERING**

In additional to smaller pre- and post- processing steps, the clustering algorithm operates in two major steps:

1. Canopy Clustering: An initial quick clustering of the datasets, utilizing a cheaper distance metric.

2. Hierarchical Clustering: The canopies are then clustered into the final clusters using a more expensive metric.

**3.1** **CANOPY CLUSTERING:**

In large datasets where it may not be feasible to process the data at once using a distance metric that is slow and expensive, canopy clustering is used to speed up the clustering process by determining an initial outline of the final clusters using a cheaper distance metric.

The difference between a canopy and a cluster is that data points may belong to different canopies at the same time. That is, canopies may overlap. Canopy clustering is useful in cases where the number of initial cluster may be large, the dataset is highly dimensional in its feature set, and the dataset is large. [1]

The way our distributed implementation of the algorithm works is as follows:

Given a subset of the data:

Pick a point, *c,* at random as a center for a canopy

Calculate similarity using Jaccard distance on bags with all other points

If the similarity exceeds the threshold **T1,** or the loose similarity, then the point is included in the canopy centered at *c. It is still considered for inclusion in future canopies.*

Only if the similarity exceeds the tighter threshold **T2,** or the tight similarity, then the point is removed from the original set and is no longer considered in other canopies.

This effectively means that canopies may overlap.

Once a canopy is formed, its centroid is then calculated as the average of all the points in the canopy.

**3.1.1 CANOPY DISTANCE METRIC (JACCARD DISTANCE ON BAGS)**

We considered several metrics that may work for our high-dimensional dataset in order to avert falling under the curse of dimensionality, after considering Euclidean and Cosine distances, we decided to go with the *Jaccard Distance on Bags* as a cheap metric (we still use cosine distance as a distance metric for hierarchical clustering):

Jaccard Distance:

Jaccard Similarity between sets is the size of their intersection divided by the size of their union, and the Jaccard distance is 1-SIM where SIM is the Jaccard similarity between two sets.

This will work well if our feature set is of a binary nature (0/1, true/false) but since each feature may have one of several values (10, after normalization) we opted to employ Jaccard bags instead.

Suppose a user x has given a rating of 3.5 for movie y in the MovieLens dataset.

3.5 becomes 7 post normalization, and so for the Jaccard set of user x, we include the movie ID y 7 times.

We then calculate the Jaccard distance accordingly (1 –Jaccard Similarity between Bags)

[1]: Andrew McCallum, Kamal Nigam, Lyle H. Ungar: Efficient Clustering of High Dimensional Data Sets With Applications to Reference Matching

**3.1.2 Canopy Implementation:**

**3.1.2.1 Initial Attempt:**

A first attempt at implementing the canopy clustering algorithm is found in the Step1WithReducer.java class.

After randomly picking a point u, then performing canopy clustering with T1 and T2 as described above. The centroid (which is here simply the randomly chosen point u) and the corresponding members (output as string IDs) are packaged in a custom key Canopy.javawhich in itself contains a custom key User.java(which packages the centroid full feature information), whereas all other members are expressed as their String IDs.

In the map phase, each mapper operates on a subset of the data, finding canopies local in scope only to that subset. All the map outputs are then funneled into a single reducer, this is to account for the fact that two centroids incoming from separate map tasks may calculate centroids that are too close to each other. In this case one of them simply absorbs the members of the other centroid.

As an example of output, this implementation yielded in 1142 canopies, after the reducer combined similar centroids this was whittled down to 112.

Doing some tuning with the values of T1 and T2 to decrease the number of canopies even further did not introduce any significant advantage and this another approach was attempted.

**3.1.2.2 Second Attempt: Hierarchical Clustering in the Reducer**

The second implementation resides in Step1.java in our source code.

At the map phase, an in-memory structure is built to store an associative array of key-value pairs where the key is the userID and the value is its detailed feature vector. The canopy formation process is similar to the one outlined above utilizing Jaccard Bag distance as a metric.

Once a canopy is formed, its centroid is then calculated as the average of all the points comprising the canopy. The centroid/canopy pair is then sent to a singular reducer from all the mappers, to account for too-close centroids.

Another thing that gets sent to the reducer is the number of canopies found, this will come in handy to calculate *k,* the number of clusters in the reducer.

The reduce phase attempts to merge centroids that are too close together by implementing agglomerative hierarchical clustering on the centroids. First, mapper outputs are aggregated into in-memory data structures, with each mapper reporting the number of canopies it has found. This is combined into the total number of canopies and then it’s divided over the number of map partitions (we have determined the number of map tasks by splitting at a certain file size, so this number will be relative to the size of the input) to produce the “average” number of canopies. This number will be the maximum upper bound for *k,* the number of clusters.

While the number of centroids (clusters) is greater than k, we find the two most similar ones and combine them together.

Originally the implementation was a naive hierarchical clustering algorithm that compared each centroid with all the other remaining centroids in order to find the two of the most similar centroids to combine. This proved infeasible as even with a small number of centroids, n < 400, calculation took a very long time. This was due to the fact that since each centroid was the aggregate of all the points of a canopy, the centroid representation was a very dense and high dimensional vector. So, comparing any two centroid was computationally costly. We optimized this algorithm by making sure that similarity between any two centroids were only calculated once. After each computation, two centroid's similarity measure were cached and recalled later when needed. When two centroids were combined, similarity measures associated with them were removed from the cache as well.

**Output:**

The output from step 1 looks as follows, this is the description of canopy centroid, we have decided on 6 as the number of clusters, So there are 6 lines in the output, with each line being a complete feature vector of the centroid and the feature set has the format of [movieID : average rating in canopy….]

1000:2.0,1003:6.0,1007:8.0,1023:10.0,1044:8.0,1050:8.0,1052:8.0,1058:8.0,1066:8.0,108:6.0,1085:6.0,1087:8.0,1106:10.0,1107:8.0,1136:7.333333333333333,1143:8.0,1178:6.0,1179:6.666666666666667,1181:8.0,1185:6.0,119:6.0,1190:6.0,1191:6.0,1194:8.0,1197:10.0,1206:9.333333333333334,1263:6.0,1273:4.0,1284:6.0,1285:8.0,1317:6.0,1331:10.0,1337:10.0,1366:8.0,1373:6.0,140:10.0,1411:2.0,1424:10.0,1435:10.0,1444:6.0,1489:8.0,1495:6.0,1528:10.0,1532:8.0,1556:6.0,1579:8.0,1609:8.0,1614:8.666666666666666,1638:6.0,1646:10.0,1647:8.0,1653:8.0,1670:8.0,1696:10.0,1697:7.333333333333333,1698:6.0,1724:6.0,1729:4.0,1730:6.0,1742:6.0,1766:8.0,177:7.0,1781:4.0,1786:8.5,180:10.0,1810:7.333333333333333,1812:6.0,1833:6.0,1834:6.0,1837:6.0,1843:8.0,1920:10.0,1931:8.0,196:8.0,197:8.0,1972:8.0,198:4.0,1995:6.0,2005:6.0,202:6.0,2022:8.0,2024:4.0,2043:10.0,2079….

Figure 3 Sample from Step 1 output

**4. Step 2: Hierarchical Clustering**

Once canopies are formed, we are ready to perform hierarchical clustering on them.

The output of step1 is distributed to all mappers (using Hadoop’s distributed cache), before the map phase begins. An in-memory data structure contains the info for each centroid.

The points from the massaged inputs file are compared to those centroids but this time using cosine distance. If a point does not belong to any cluster, it is put into a special “unclustered” cluster.

The map phase maps the points in a cluster, emitting <centroid, member> key-value pairs, the reducer simply aggregates each cluster and outputs it.

We attempted to try out Jaccard bags again in addition to cosine distances, but found its output to be redundant.

The final output is of the form: <centroid feature> <tab separator> <member1 features, member2 features…., memberN features>

1:7.5,10:6.7,1020:2.0,103:7.117647058823529,104:4.666666666666667,1078:9.0,11:9.2,112:10.0,1121:1.0,1167:1.0,1171:1.0,121:4.0,122:8.0,126:8.0,127:8.666666666666666,129:10.0,131:8.0,134:8.347826086956522,135:8.666666666666666,137:6.222222222222222,1371:1.0,14:4.666666666666667,142:6.8,144:5.666666666666667,1444:2.0,145:8.0,149:6.769230769230769,15:6.0,152:7.25,157:4.0,166:8.0,169:7.032258064516129,17:10.0,170:4.8,1700:2.0,1745:6.0,179:4.0,186:6.0,19:5.75,190:8.0,191:5.615384615384615,1927:5.0,2:7.6,20:6.0,201:10.0,204:8.0,206:2.0,207:7.2,208:5.333333333333333,2097:10.0,21:6.666666666666667,210:4.0,211:10.0,212:5.5,213:8.0,214:4.0,2158:1.0,216:8.0,217:7.333333333333333,218:8.0,219:8.0,2227:1.0,230:4.0,233:7.5,234:7.166666666666667,237:8.0,24:4.0,2401:5.0,242:10.0,243:6.0,247:6.333333333333333,252:8.0,253:6.0,255:10.0,257:7.333333333333333,258:6.0,263:8.0,267:6.285714285714286,2673:1.0,271:7.153846153846154,272:7.5,275:6.75,278:8.0,279:6.333333333333333,293:7.0,294:6.580645161290323,295:7.777777777777778,296:9.15625,297:8.0,3:6.0,3009:5.0,305:4.0,306:6.853658536585366,314:8.0,316:6.866666666666666,317:10.0,319:6.0,32:4.666666666666667,321:6.068965517241379,3236:9.0,326:7.515151515151516,33:8.285714285714286,332:6.0,333:8.518518518518519,339:8.0,341:7.625,344:7.2,347:7.2,348:6.0,354:8.272727272727273,357:7.05,358:6.0,37:6.571428571428571,376:6.0,381:8.0,386:7.777777777777778,387:8.0,397:8.0,398:6.857142857142857,399:10.0,400:7.03030303030303,401:5.142857142857143,403:6.0,405:9.0,407:8.0,419:7.5,422:8.666666666666666,439:10.0,442:6.0,445:7.923076923076923,45:5.818181818181818,450:8.0,456:8.0,465:8.0,48:7.0,480:8.0,489:2.0,492:8.0,503:7.2,516:7.6,518:6.0,538:10.0,541:7.0,542:7.875,543:7.6571428571428575,544:7.363636363636363,545:7.230769230769231,546:6.15,547:8.275,548:7.333333333333333,549:7.470588235294118,550:8.0,551:8.5,56:6.0,560:8.142857142857142,568:8.0,58:8.5,588:6.666666666666667,591:8.0,595:5.333333333333333,6:8.0,604:10.0,621:6.0,640:10.0,642:10.0,643:10.0,648:9.2,649:10.0,650:7.5,657:8.0,669:4.0,672:10.0,673:10.0,679:9.0,696:8.0,698:6.0,7:8.666666666666666,701:8.0,719:6.0,733:6.0,741:10.0,78:10.0,845:10.0,847:10.0,854:10.0,856:10.0,857:10.0,858:10.0,888:10.0,892:10.0,896:10.0,897:10.0,898:10.0,899:10.0,900:10.0,921:10.0,926:8.0,940:10.0,946:10.0,969:10.0,98:10.0 10781,10633,9494,10165,10645,10786,10652,9447,10790,9545,9717,10420,10552,10754,10554,10842,10660,10661,10413,9707,10666,10794,10014,10813,10003,10757,9524,10316,9622,10321,10570,10323,10329,9420,10277,10276,10262,10206,9678,9467,10359,9881,10746,9657,10245,10226,9520,9866,10225,9972,9864,10496,10029,9859,10113,10823,10120,10064,10126,9964,10083,10593,10493,10596,10015,9822,9821,9935,10138,9499,9930,9911,9769,9765,9761,10838,10025,9608,8830,9238,8061,9066,8211,8823,8807,9117,8782,8604,8778,8759,8150,8758,8696,8972,9298,9248,8726,8981,8491,8986,8062,8176,8490,9316,9371,8459,8454,8136,9321,8406,9162,8391,8382,9282,8196,9170,8371,8356,9180,8349,8058,9040,8549,7970,8019,9198,8521,8115,8514,9206,8084,9116,8588,8887,8873,9346,8599,8208,8277,8270,8846,8926,8046,8603,8935,57281,57025,57280,56103,56557,56558,56592,56990,57269,55940,56606,56969,56652,56654,56119,57262,56131,56695,56138,56760,56913,56783,56149,56791,56793,56814,56835,57239,56156,56852,56161,56876,55961,56164,56175,55970,57331,56176,56195,57366,56196,57207,57205,56200,55983,56207,57199,57198,57193,57182,57179,57318,55921,56256,56273,56276,56277,56283,56293,56296,55895,56322,56336,57117,56052,57115,57111,56378,56059,57099,56392,56064,56406,57083,56069,56439,56075,56453,57060,57058,57055,56092,56095,56490,56516,56535,57028,44651,44884,43737,43537,44149,43738,44181,44197,44210,44211,44229,43582,44243,44599,44595,44278,44864,44285,43593,44295,44303,44859,44978,43541,44305,44856,44568,44306,43752,44848,44381,44536,44844,43602,43765,44529,4441

Figure 4 Sample from Step 2 output, green text is the centroid feature vector. Gold text are the userIDs comprising the cluster

**5. Step 3: Output Results**

We ended up with 6 clusters and an additional cluster to group together “unclustered” points that do not belong to any cluster.

We performed additional post-processing to extract meaningful information out of the output. Those are linear non Map-Reduce jobs that reside in the postprocessing package in the source code.

One of the programs, MovieIDResolver.java, is responsible for converting movie IDs back into their full names.

The other is responsible for resolving user IDs in each cluster.

PART II

RESULTS

Here is a sample of the highest rated movies per each of the six clusters:[2]



[2]: the full movie per cluster output for the 1M MovieLens dataset is available at:

<https://docs.google.com/spreadsheets/d/1yJMSI5JIrf_1KAxJp4qbPfMlmmC37S4z97uXMYZSX6c/edit?usp=sharing>, it is also included in the project submission.

Here is the analysis of the distributed of user professions per cluster:

everyone

male 0.7170061268421924

academic/educator 0.08726610365954628

artist 0.044212617983109784

clerical/admin 0.028647127007782745

college/grad student 0.12568306010928962

customer service 0.018546116906772644

doctor/health care 0.03907931776784236

executive/managerial 0.11243583374730916

farmer 0.002815035601920848

homemaker 0.01523431031627753

K-12 student 0.03229011425732737

lawyer 0.02136115250869349

programmer 0.06424904785560523

retired 0.02351382679251532

sales/marketing 0.05000827951647624

scientist 0.02384500745156483

self-employed 0.039907269415466136

technician/engineer 0.08312634542142738

tradesman/craftsman 0.011591323066732903

unemployed 0.011922503725782414

writer 0.046530882596456366

Here is the analysis of the user age distribution per cluster:

everyone

Under 18 0.03676105315449578

18-24 0.1826461334658056

25-34 0.3470773306838881

35-44 0.1975492631230336

45-49 0.09107468123861566

50-55 0.08213280344427885

56    0.06275873488988243

Trends Observed Per Cluster:

**Cluster 1 (Females Who Hate Sci-Fi):**

The male-to-female ration is lower than normal, and the tech/engineer profession ratio was also much lower than normal (0.0683) versus (0.083)

Movies that this cluster uniquely liked were movies focusing on females and murder (**To Die For, Serial Mom, Getting Away With Murder, All About Eve)**

Movies that this cluster uniquely disliked were mainstream thrillers (**Mission Impossible, The Specialist)** and Science Fiction (**Star Wars: A New Hope, Tron)**

**AGE DISTRIBUTION**

Under 18 0.04296875

18-24 0.162109375

25-34 0.291015625

35-44 0.19921875

45-49 0.1015625

50-55 0.125

56    0.078125

**GENDER DISTRIBUTION**

**male 0.640625**

**PROFESSION DISTRIBUTION**

academic/educator 0.09375

artist 0.03125

clerical/admin 0.03125

college/grad student 0.111328125

customer service 0.025390625

doctor/health care 0.04296875

executive/managerial 0.1328125

farmer 0.005859375

homemaker 0.01953125

K-12 student 0.037109375

lawyer 0.013671875

programmer 0.052734375

retired 0.03125

sales/marketing 0.060546875

scientist 0.03515625

self-employed 0.046875

technician/engineer 0.068359375**much lower than normal**

tradesman/craftsman 0.013671875

unemployed 0.0

writer 0.03125

**Cluster 2 (Hipsters):**

The male-to-female ratio is slightly higher than normal, the programmer profession has a slight increase than normal, and the percentage of people who are retired is higher than average.

This cluster had the highest number of tech/engineers professions, but it was not much higher than average.

Movies uniquely favored by this cluster were more left-of-center than others: **Pi, The Phantom, Richie Rich**

Movies uniquely disliked by this cluster were classics: **The Godfather, Gone With The Wind**

**AGE DISTRIBUTION**

Under 18 0.027510316368638238

18-24 0.1856946354883081

25-34 0.31911966987620355

35-44 0.20495185694635487

45-49 0.08528198074277854

50-55 0.08803301237964237

56    0.08940852819807428

**GENDER DISTRIBUTION**

male 0.7634112792297112

**PROFESSION DISTRIBUTION**

academic/educator 0.06327372764786796

artist 0.05364511691884457

clerical/admin 0.023383768913342505

college/grad student 0.13204951856946354

customer service 0.017881705639614855

doctor/health care 0.030261348005502064

executive/managerial 0.1155433287482806

farmer 0.0041265474552957355

homemaker 0.0041265474552957355

K-12 student 0.027510316368638238

lawyer 0.03163686382393398

programmer 0.07840440165061899 **slightly higher**

retired 0.04126547455295736 **higher the average**

sales/marketing 0.04676753782668501

scientist 0.023383768913342505

self-employed 0.03851444291609354

technician/engineer 0.08940852819807428 **not much higher but highest among all the cluster**

tradesman/craftsman 0.009628610729023384

unemployed 0.013755158184319119

writer 0.05639614855570839

**Cluster 3 (Seniors/Godzilla):**

This cluster is noticeably older than the rest, has a much lower male-to-female ratio and as a result has a lot less college students, and a higher percentage of teachers.

Movies uniquely liked by this cluster: Godzilla (3 different versions), Star Wars: Empire Strikes Back

Movies uniquely disliked by this cluster: Ben-Hur, The Maltese Falcon,

**AGE DISTRIBUTION**

Under 18 0.046413502109704644

18-24 0.08227848101265822  **much lower, less than half the rate**

25-34 0.2468354430379747**10 absolute % lower**

35-44 0.23628691983122363  **slightly higher**

45-49 0.1518987341772152**much higher than normal**

50-55 0.12236286919831224  **higher**

56    0.11392405063291139  **much higher**

**GENDER DISTRIBUTION**

male 0.6265822784810127  **much lower than normal**

**PROFESSION DISTRIBUTION**

academic/educator 0.1371308016877637  **much higher than normal**

artist 0.046413502109704644

clerical/admin 0.04008438818565401

**college/grad student 0.06962025316455696 much lower than normal, almost half**

customer service 0.010548523206751054

doctor/health care 0.02531645569620253

executive/managerial 0.11181434599156118

farmer 0.002109704641350211

homemaker 0.0189873417721519

K-12 student 0.03375527426160337

lawyer 0.023206751054852322

programmer 0.06751054852320675

retired 0.04008438818565401

sales/marketing 0.04852320675105485

scientist 0.02109704641350211

self-employed 0.04430379746835443

technician/engineer 0.0759493670886076

tradesman/craftsman 0.004219409282700422

unemployed 0.012658227848101266

writer 0.052742616033755275

**Cluster 4 (Foreign Films):**

The male-to-female ratio is lower than normal in this cluster.

Movies uniquely liked by this cluster: L’Associe, Autoportrait de Decembre, L bambini Ci guardano

Movies uniquely disliked by this cluster: Beauty and the Beast, Peter Pan

**AGE DISTRIBUTION**

Under 18 0.030395136778115502

18-24 0.1833839918946302

25-34 0.3262411347517731

35-44 0.20466058763931105

45-49 0.08611955420466058

50-55 0.08814589665653495

56    0.08105369807497467

**GENDER DISTRIBUTION**

male 0.6524822695035462

**PROFESSION DISTRIBUTION**

academic/educator 0.09321175278622088

artist 0.05065856129685917

clerical/admin 0.029381965552178316

college/grad student 0.12462006079027356

customer service 0.010131712259371834

doctor/health care 0.04660587639311044

executive/managerial 0.10536980749746708

farmer 0.002026342451874367

homemaker 0.034447821681864235

K-12 student 0.019250253292806486

lawyer 0.0243161094224924

programmer 0.0547112462006079

retired 0.030395136778115502

sales/marketing 0.04154002026342452

scientist 0.019250253292806486

self-employed 0.03951367781155015

technician/engineer 0.07598784194528875

tradesman/craftsman 0.016210739614994935

unemployed 0.014184397163120567

writer 0.0486322188449848

**Cluster 5 (Horror):**

No particularly divergent statistics observed in this cluster.

Movies uniquely liked by this cluster: Cape Fear, Poltergeist, Nightmare On Elm Street, Edward Scissorhands

Movies uniquely disliked by this cluster: Philadelphia, a lot of obscure films.

**AGE DISTRIBUTION**

Under 18 0.036896877956480605

18-24 0.18543046357615894

25-34 0.3547776726584674

35-44 0.19110690633869443

45-49 0.09176915799432356

50-55 0.07757805108798486

56    0.06244087038789026

**GENDER DISTRIBUTION**

male 0.7369914853358562

**PROFESSION DISTRIBUTION**

academic/educator 0.08893093661305582

artist 0.03216650898770104

clerical/admin 0.026490066225165563

college/grad student 0.13623462630085148

customer service 0.017029328287606435

doctor/health care 0.048249763481551564

executive/managerial 0.13339640491958374

farmer 0.002838221381267739

homemaker 0.014191106906338695

K-12 student 0.030274361400189215

lawyer 0.020813623462630087

programmer 0.07000946073793755

retired 0.017029328287606435

sales/marketing 0.06244087038789026

scientist 0.020813623462630087

self-employed 0.03216650898770104

technician/engineer 0.0879848628192999

tradesman/craftsman 0.010406811731315043

unemployed 0.00946073793755913

writer 0.02838221381267739

**Cluster 6 (Young People):**

This cluster has a much younger population and a slightly higher male-to-female ratio.

Movies uniquely liked (very diverse): They Bite, Raging Bull

Movies uniquely disliked: The Big Lebowski, Forrest Gump, Monty Python’s Life of Brian (witty films)

**AGE DISTRIBUTION**

Under 18 0.039000876424189306

18-24 0.20552147239263804 **slightly higher**

25-34 0.394829097283085 **slightly higher**

35-44 0.1866783523225241

45-49 0.07975460122699386

50-55 0.061787905346187555 **slightly lower**

56    0.03242769500438212 **much lower, halfed**

**GENDER DISTRIBUTION**

male 0.7567922874671341 **slightly higher than normal**

**PROFESSION DISTRIBUTION**

academic/educator 0.07975460122699386

artist 0.04645048203330412

clerical/admin 0.028045574057843997

college/grad student 0.13409290096406662

customer service 0.02322524101665206

doctor/health care 0.036371603856266435

executive/managerial 0.10035056967572305

farmer 0.0021910604732690623

homemaker 0.009202453987730062

K-12 student 0.039000876424189306

lawyer 0.018404907975460124

programmer 0.06310254163014899

retired 0.012708150744960562

sales/marketing 0.04688869412795793

scientist 0.025416301489921123

self-employed 0.04163014899211218

technician/engineer 0.08676599474145487

tradesman/craftsman 0.011831726555652936

unemployed 0.014022787028921999

writer 0.053023663453111304

PART III

**CODE STRUCTURE**

The tar file submitted in this project includes the following folders:

/**documentation**

**/output**

**/src**

**/documentation**: includes this document, and a copy of the slides for the final presentation.

**/output**: sample output files

Step1output: contains the output from our Step1.java program for the 1M dataset

Step10Moutput: contains the output from our Step1.java program for the 10M dataset

Step2\_output.txt: contains the output from our Step2.java program for the 10M dataset.

Cluster Output 1M.xslx: an Excel style sheet which contains the movie distribution for our six clusters output for the 1M dataset.

**/src**: the source code

The /src folder is part of the regular Maven build structure which we used for this project, and it contains:

**/main**

**/test**

**/test** includes a single cs757/project/Tests.java file which includes all of our unit tests.

/**main/java/cs757** is where the majority of our project’s code is, and is divided in five packages:

**/driver** : contains a ProjectDriver class which is a simple launcher for the project, and a JobFactory which configures a job based on the arguments given at the command line:

hadoop jar [jar-name] [fully-qualified-name-of-ProjectDriver] [args]

where args is: [input] [output] [jobType]

JobFactory will create one of five job types based on the jobType argument:

* Preprocess Job: this runs the Munger.java class on the input.
* Step1WithReducer job: this must be followed by the step1PostProcessing job, combined they provided our initial attempt to conduct Step1, which resulted in too many canopies.
* Step1PostProcessing job: this follows the Step1WithReducer job, which outputs a <centroid, members> lines but the members are only represented by their string IDs, this job converts them back to fully-featured vectors.
* Step1 job, this is our final job which creates the canopy centroids.
* Step2 job, this is the final step which outputs the cluster centroids and their members.

**/preprocessor:**

This package include our pre-processors, which consist of two classes:

RemoveRareMovies.java:

This is a non Map-Reduce program.

This is the first step in our program.

We take the ratings input file, read its lines into a map structure, then distribute the movies into percentiles based on the number of ratings they receive. We found out that a good cutoff that applies across datasets is 35, this is set in line 41:

int percentile = 35;  
int cutoff = ordered.get((ordered.size()\*percentile)/100);

In lines 46-57 we loop through the inputs file by line again and this time determine if we need to drop it if the number of ratings it receives is below the cutoff.

We output a new file, reduced\_ratings.txt, with all ratings that do not match the cutoff taken out.

Munger.java:

This is the second step in our program.

It is a map-reduce job.

The map method:

Takes the reduced\_ratings.txt, normalizes the ratings, then outputs to the reducer.

The reduce method:

Takes input from method and prints it.

**/clustering**

This package includes our clustering code.

There is a child package **/initialattempt** which contains the code for our initial attempt. This will not be explained in detail here.

Step1.java

Following pre-processing, this is the initial step which produces canopies.

This is a Map-Reduce job.

setup method:

this method simply instantiates an in-memory hash map which will hold the details of <userID, userFeatureVector> for its key-value pairs.

Map method:

Puts data from the input file reduced\_ratings.txt into reduced\_ratings.txt

Cleanup method:

Once the hash map is completely built, this cleanup post-map method builds the actual canopies. It calls emitCanopies() which does the heavy-lifting.

emitCanopies:

picks a random point from the canopy

then begins comparing it with every other point in the original input set using the jaccardBag distance measure.

if the similarity is greater than 0.120 (T1), it is added to the canopy

moreover, if it’s greater than 0.135 (T2), it is removed from the original set (added to veryCloseUsers list, which gets removed from the original set later

We put thresholds on the size of canopies (vcMin, canopyMin and limit) to control the size of the canopies that get output, those numbers were determined by trial and error.

The centroid is calculated then by the calcCentroid method.

What gets sent to the reducer is one of two things:

A centroid, denoted by the “::” symbol.

The number of canopies this particular mapper has found, which is indicated by the absence of the “::” symbol.

calcCentroid:

This method takes all the members of the canopy, and averages their details to obtain the centroid.

Movies that feature very little (<=2) are not considered to reduce dimensionality.

Reduce:

There is a single reducer, which loops over all the input it receives, if the value it sees is a centroid (denoted by the “::” symbol) then it adds to the centroid list, if otherwise, it adds it to the list keeping tab on the number of canopies found from each mapper.

Finally, the average number of canopies found by summing all the values in the number of canopies list (total number of canopies) and then dividing by the list size(number of mappers), we have set the number of mappers to be relative to the input size.

Then we attempt to find centroids that are too close to each other and merge them using agglomerative hierarchical clustering.

There are two for loops enclosed in a while loop, the while loop terminates when the number of centroids left is less than the average number of canopies found (an upper bound on k, the number of clusters desired)

Inside the two for loops, we are comparing every centroid to every other centroid using cosine Distance this time, the two that have maximum similarity are merged. This process continues until

The while loop breaks out.

Several attempts were made to optimize those for loops as it is a bottleneck.

The final centroids are then written to disk, one per line.

Step2.java

This is a Map-Reduce job, it is responsible for calculating the final clusters once the canopy centroids are found.

setup():

We put the output of step1 in Hadoop’s distributed cache, the setup method here reads this file into an in-memory ArrayList, centroids.

map():

the input to this file is again the massaged reduced ratings, here we use cosine distance to compare points to each centroid to determine cluster membership, if it is found that the point is not close to any centroid, it is added to the “unclustered” cluster.

What is passed to the reduce method is <centroid, members> where centroid is the fully-featured centroid and members are string IDs of the member of the cluster.

reduce():

this reduce task is a simple aggregation per centroid for all the members it obtains from the several map tasks. What is output to disk is :

<centroid, members>

Where centroid is the fully featured centroid vector

Members is a list of userIDs comprising the cluster.

**/postprocessor**

This package contains our post-processing non Map-Reduce jobs that we run on the output of step 2.

AnalyzeCluster.java:

This program examines clusters and outputs the distribution of users in our clusters by their age, gender and profession.

We read the original user mapping file from the MovieLens dataset into a map structure first

We then read the output of step2, focusing on the users in each cluster, and ignoring all centroids and the entire “unclustered” cluster.

Per cluster, we read the String ID of each user, get the user details from the map structure we built earlier, then add this to both the cluster’s data structure (a list of user attributes) and to another data structure (everyone), which we will use for global statistics.

Once the mapping of the clusters is complete, we take the data structures we have built and extract statistics out of them:

First we map out male-to-female ratio, age and profession distributions in the everyone (global) structure.

Then we repeat this process for every cluster.

We finally print out the averages per cluster and per the global population.

UserAttr.java

This class is a simple value class used by the AnalyzeCluster.java to hold user attributes (age, gender, occupation)

MovieIDResolver.java

This class maps movieIDs from the output of step 2 back into their movie names.

It reads all the lines from the movie ID input file from the dataset using the buildMovieMap() static method

Then iterates over all movie ID per cluster centroid, finally printing out

Cluster Number

Movie ID1 movieName 1

Movie ID2 movieName2

….

**/customkeys**

Those are custom Hadoop composite keys that were used during our initial attempt, they abstract away Canopies and Users into their own classes that are then emitted to the reducer.