4. Clustering the Sessions

The second challenge problem is to cluster the user sessions based on features in the data. The process of clustering informs you which groups of sessions are notably more similar to each than to other sessions, based on the set of features you decide to use. The clustering process reveals both the number of natural groupings given the data and features, and which sessions belong to which grouping. You can then examine those groupings to look for notable behavior groupings, such as a large group of sessions where the user searches unsuccessfully several times and then watches a video from the home page. You can also flag sessions that are outliers in the grouping, as these sessions may represent anomalies of interest, such as bots, fraud, system errors, etc.

Sources of information about clustering abound. From a practical perspective, one of the more useful ones is chapter 7 of Mining of Massive Data Sets. For more of the theory, see http://projecteuclid.org/download/pdf_1/euclid.bsmsp/1200512992 and http://ilpubs.stanford.edu:8090/778/1/2006–13.pdf

The log data captures user behavior. As such, your clustering should center around user behavior. In these exercises, you'll use Cloudera ML (https://github.com/cloudera/ml/wiki) as your clustering tool because of the useful statistics it gives about our clusters. The general approach is to create a list of features and try them out, using the statistics from Cloudera ML to evaluate the quality of the features. When you find the set of features that gives you optimal statistics, Cloudera ML will also tell you the optimal number of clusters.

Step 1: Determine features for testing

Start by seeing what kind of features you can pull out of the data. Look at a line from the cleaned data for a reference:

```
$ hadoop fs -cat clean/part\* | head -1
{"session": "2b5846cb-9cbf-4f92-a1e7-b5349ff08662", "hover": ["16177", "10286", "8565", "10596",
"29609", "13338"], "end": "1368189995", "played": {"16316": "4990"}, "browsed": [], "recommendations":
["13338", "10759", "39122", "26996", "10002", "25224", "6891", "16361", "7489", "16316", "12023",
"25803", "4286e89", "1565", "20435", "10596", "29609", "14528", "6723", "35792e23", "25450",
"10143e155", "10286", "25668", "37307"], "actions": ["login"], "reviewed": {}, "start": "1368189205",
"recommended": ["8565", "10759", "10002", "25803", "10286"], "rated": {}, "user": "10108881",
"searched": [], "popular": ["16177", "26365", "14969", "38420", "7097"], "kid": null, "queued":
["10286", "13338"], "recent": ["18392e39"]}
cat: Unable to write to output stream.
```

From this example, you can directly pull the following features:

- Actions (a feature for each, except login and logout)
- Number of items hovered over
- Session duration
- Number of items played
- Number of items browsed
- Number of items reviewed
- Number of items rated
- Number of items searched
- Number of recommendations that were reviewed
- Kid (parental controls)
- Number of items queued

Some obvious features you can safely ignore because they clearly aren't going to carry any signal, such as whether the user logged in and out – this is unlikely to relate to the user's behavior during the session.

There are also some less direct features you can extract:

- Mean play time
- Shortest play time

- Longest play time
- Total play time
- Total play time as fraction of session duration
- Longest play: less than 5 minutes, between 5 and 60 minutes, more than 60 minutes
- Shortest play: less than 5 minutes, between 5 and 60 minutes, more than 60 minutes
- Number of items played less than 5 minutes
- Number of items played more than 60 minutes
- Number of items hovered over that were played
- Number of browsed items that were played
- Number of reviewed items that were played
- Number of recommended items that were played
- Number of rated items that were played
- Number of searched items that were played
- Number of popular items that were played
- Number of queued items that were played
- Number of recent items that were played
- Number of recent items that were reviewed
- Number of recent items that were rated

There are other possible features, but none of them seem likely to have any signal in them.

Our method for testing features will be to add them into the model in chunks and then back them out incrementally when the clusters start to become less distinct. As a place to start, add in all of the action features plus the number of items played. Start with those features because they seem like fairly safe choices.

Step 2: Merge the data

First, you'll need is a tool to parse the clean data and generate the feature vectors. Again, as before, using a Python Hadoop streaming job is the best choice because Python is good at handling JSON. Remember that in cleaning the data, we had you split session with parental controls into two separate sessions. You need to merge those before you can properly cluster the sessions.

Start with a job that aggregates the records by session ID and then merges records for the same ID. You could reuse the same job that you used to create the clean data, minus the part that splits the sessions, but since the raw data is so much larger than the cleaned data, you'll get better scalability out of a quick merge job on the clean data.

1. Write a mapper

Create a new file called `merge_map.py` with the following contents:

```
#!/usr/bin/python

import re
import sys

def main():
   p = re.compile('.*"session": "([^"]+)".*')

# Read all the lines from stdin
   for line in sys.stdin:
  m = p.match(line)

  if m:
     print "*%s*\t*%s*" % (m.group(1), line.strip())
  else:
     sys.strerr.write("Failed to find ID in line: *%s*" % line)

if __name__ == '__main__':
   main()
```

The script uses a regular expression instead of parsing the JSON because the regular expression match is a much less expensive operation.

2. Write a reducer

For the reducer, you have to be a little careful how you implement the merge because of the nested structure of the clean data; you'll have to handle the various nested types on a case by case basis.

Create a new file called `merge_reduce.py` with the following contents:

```
#!/usr/bin/python

import json
import sys

def main():
   last = None

   # Read all the lines from stdin
   for line in sys.stdin:
      id, str = line.strip().split('\t', 1)

      if id != last:
         if last != None:
            print json.dumps(data)

         last = id
         data = None

      data = merge(data, json.loads(str))

   print json.dumps(data)

"""
Merge the contents of new into data being mindful of types
"""
def merge(data, new):
   if data:
      for key in new:
         if type(new[key]) == dict:
            data[key].update(new[key])
         elif type(new[key]) == list:
            data[key].extend(new[key])
         elif type(new[key]) == bool:
            data[key] |= new[key]
         elif data[key] != new[key] and key not in ['start', 'end', 'kid']:
            sys.stderr.write("Mismatch for %s on %s: %s != %s\n" % (data['session'], key,
str(data[key]), str(new[key])))
   else:
      data = dict(new)

   return data

if __name__ == '__main__':
   main()
```

3. Run the job

Run the job with the following commands:

```
$ hadoop jar $STREAMING -mapper merge_map.py -file merge_map.py -reducer merge_reduce.py -file
merge_reduce.py -input clean -output merged
```

Step 3. Generate feature vectors

Now, you can use the merged data to generate the features.

1. Write a mapper

Create a new file called `features_map.py` with the following contents:

```
#!/usr/bin/env python

import sys
import json

def main():
   # Read all lines from stdin
    for line in sys.stdin:
        session = json.loads(line)
        fields = []

        fields.append(session['session'])
        fields.append('updatePassword' in session['actions'])
        fields.append('updatePaymentInfo' in session['actions'])
        fields.append('verifiedPassword' in session['actions'])
        fields.append('reviewedQueue' in session['actions'])
        fields.append(session['kid'])

        played = set(session['played'].keys())
        fields.append(len(played))
        print ','.join(map(str, fields))
if __name__ == '__main__':
    main()
```

2. Run as map-only job

You can now run the job as map-only and gather the results. You have to be careful with the treatment of the separators: because these columns are going to be interpreted as numeric values, you don't want to have an extraneous tab character at the end of each line. The way to avoid it is to tell Hadoop that the key and value are comma delimited and to also use commas to delimit the final output:

```
$ hadoop jar $STREAMING -D mapred.reduce.tasks=0 -D mapred.textoutputformat.separator=, -D
stream.map.output.field.separator=, -mapper features_map.py -file features_map.py -input merged -
output features0
...
$ hadoop fs -getmerge features0 features.csv
$ hadoop fs -put features.csv
```

Step 4. Build Cloudera ML workflow

1. Build header file for Cloudera ML

Next, Cloudera ML requires a header file that describes all the features. Create a new file called `header.csv` with the following contents:

```
session_id,identifier
updatePassword,categorical
updatePaymentInfo,categorical
verifiedPassword,categorical
reviewedQueue,categorical
kid,categorical
num_plays
```

The keyword in the second column tells Cloudera ML the feature class. If no class is given, Cloudera ML assumes the feature is numeric. The boolean features are all categorical because there can only be two values, and doing things like finding the average and standard deviation wouldn't make sense.

Now that you have your inputs, go through the Cloudera ML workflow.

2. Create summary file

Create the summary file in plain text format using the header and features files:

```
$ ml summary --summary-file summary.json --header-file header.csv --format text --input-paths
features.csv
```

```
…
```

3. Normalize the features

Here, you will normalize the features using the header and features files and the summary file you just generated. You have two choices for normalization: unit normal or range. For this particular problem, you'll find it most practical to make all numeric features unit normal. Set the transform to Z. Since you'll only be accessing the normalized data programmatically, set the output format to Avro.

```
$ ml normalize --summary-file summary.json --format text --id-column 0 --transform Z --input-paths
features.csv --output-path part2normalized --output-type avro
…
```

Step 5. Create a *k*-means++ sketch

1. Create the *k*-means++ sketch

Create the *k*-means++ sketch using the normalized data. Enter the following commands:

```
$ ml ksketch --format avro --input-paths part2normalized --output-file part2sketch.avro --points-per-
iteration 1000 --iterations 10 --seed 1729
…
```

2. Specify the seed

The Cloudera ML command line tool lets you specify a seed to use so that the results across multiple runs are comparable. Without specifying a seed, a different set of random centroids will be selected each run, producing results that may be arbitrarily better or worse than previous runs.

Specify the seed so that your results are comparable and run the *k*-means parallel algorithm on the sketch data with the following command:

```
$ ml kmeans --input-file part2sketch.avro --centers-file part2centers.avro --clusters
40,60,80,100,120,140,160,180,200 --best-of 3 --seed 1729 --num-threads 1 --eval-details-file
part2evaldetails.csv --eval-stats-file part2evalstats.csv
…
ID,NumClusters,TestCost,TrainCost,PredStrength,StableClusters,StablePoints
ID,NumClusters,TestCost,TrainCost,PredStrength,StableClusters,StablePoints
0,40,8.13,33.48,0.5148,0.88,0.9693
1,40,9.94,34.05,0.8075,0.93,0.9818
2,40,12.64,34.92,0.5148,0.83,0.9572
3,52,0.00,27.48,1.0000,1.00,1.0000
4,52,0.00,27.48,1.0000,1.00,1.0000
5,52,0.00,27.48,1.0000,1.00,1.0000
6,52,0.00,27.48,1.0000,1.00,1.0000
7,52,0.00,27.48,1.0000,1.00,1.0000
8,52,0.00,27.48,1.0000,1.00,1.0000
9,52,0.00,27.48,1.0000,1.00,1.0000
10,52,0.00,27.48,1.0000,1.00,1.0000
11,52,0.00,27.48,1.0000,1.00,1.0000
12,52,0.00,27.48,1.0000,1.00,1.0000
13,52,0.00,27.48,1.0000,1.00,1.0000
14,52,0.00,27.48,1.0000,1.00,1.0000
15,52,0.00,27.48,1.0000,1.00,1.0000
16,52,0.00,27.48,1.0000,1.00,1.0000
17,52,0.00,27.48,1.0000,1.00,1.0000
18,52,0.00,27.48,1.0000,1.00,1.0000
19,52,0.00,27.48,1.0000,1.00,1.0000
20,52,0.00,27.48,1.0000,1.00,1.0000
21,52,0.00,27.48,1.0000,1.00,1.0000
22,52,0.00,27.48,1.0000,1.00,1.0000
23,52,0.00,27.48,1.0000,1.00,1.0000
24,52,0.00,27.48,1.0000,1.00,1.0000
25,52,0.00,27.48,1.0000,1.00,1.0000
26,52,0.00,27.48,1.0000,1.00,1.0000
```

The list of cluster sizes are the sizes that will be tried by the algorithm. The “best of” parameter tells Cloudera ML to run the algorithm three times for each cluster size. The number of threads is set to 1 since that’s how many processors you have in your local machine by default. If you set the number of processors higher, you may use a larger number an the number of threads in the command.

You can see from this output that the features you selected lead to exactly 52 clusters. This clearly overshot on the number of clusters specified, but that’s OK. You’re looking for a predictive strength over 0.8 with good stability numbers. Predictive strength is a measure of how well the clusters describe the data. Stability measures how much the clusters and data points change between the training data and the test data. Since you have 1.0 for all three, this is looking good. Time to add more features.

3. Add features to your mapper

Add in the rest of the first class features. Open up the `features_map.py` file and modify it as follows (changes in bold red):

```
#!/usr/bin/env python

import sys
import json

def main():
   # Read all lines from stdin
    for line in sys.stdin:
        session = json.loads(line)
        fields = []

        fields.append(session['session'])

        fields.append('updatePassword' in session['actions'])
        fields.append('updatePaymentInfo' in session['actions'])
        fields.append('verifiedPassword' in session['actions'])
        fields.append('reviewedQueue' in session['actions'])
        fields.append(session['kid'])

        session_duration = (long(session['end']) - long(session['start']))
        fields.append(session_duration)

        played = set(session['played'].keys())
        browsed = set(session['browsed'])
        hovered = set(session['hover'])
        queued = set(session['queued'])
        recommendations = set(session['recommendations'])
        rated = set(session['rated'].keys())
        reviewed = set(session['reviewed'].keys())
        searched = set(session['searched'])

        fields.append(len(played))
        fields.append(len(browsed))
        fields.append(len(hovered))
        fields.append(len(queued))
        fields.append(len(recommendations))
        fields.append(len(rated))
        fields.append(len(reviewed))
        fields.append(len(searched))

        print ','.join(map(str, fields))

if __name__ == '__main__':
    main()
```

4. Add features to your header

Next, open up the `header.csv` file and modify it as follows (changes in bold red):

```
session_id,identifier
```

```
updatePassword,categorical
updatePaymentInfo,categorical
verifiedPassword,categorical
reviewedQueue,categorical
kid,categorical
session_duration
num_plays
num_browsed
num_hovered
num_queued
num_recommendations
num_rated
num_reviewed
num_searched
```

5. Rerun the workflow

Rerun the workflow with the following commands:

```
$ hadoop jar $STREAMING -D mapred.reduce.tasks=0 -D mapred.textoutputformat.separator=, -D
stream.map.output.field.separator=, -mapper features_map.py -file features_map.py -input merged -
output features1
...
$ hadoop fs -getmerge features1 features.csv
$ hadoop fs -rm features.csv
Deleted features.csv
$ hadoop fs -put features.csv
$ ml summary --summary-file summary.json --header-file header.csv --format text --input-paths
features.csv
...
$ ml normalize --summary-file summary.json --format text --id-column 0 --transform Z --input-paths
features.csv --output-path part2normalized --output-type avro
...
$ ml ksketch --format avro --input-paths part2normalized --output-file part2sketch.avro --points-per-
iteration 1000 --iterations 10 --seed 1729
...
$ ml kmeans --input-file part2sketch.avro --centers-file part2centers.avro --clusters
40,60,80,100,120,140,160,180,200 --best-of 3 --seed 1729 --num-threads 1 --eval-details-file
part2evaldetails.csv --eval-stats-file part2evalstats.csv
…
ID,NumClusters,TestCost,TrainCost,PredStrength,StableClusters,StablePoints
0,40,3451.05,3676.64,0.2611,0.23,0.1132
1,40,3483.86,3818.16,0.3079,0.43,0.3270
2,40,3410.42,3653.56,0.4049,0.28,0.2672
3,60,2709.64,3037.88,0.2524,0.33,0.2517
4,60,2710.45,2959.93,0.2922,0.40,0.3441
5,60,2618.62,3018.66,0.2451,0.35,0.3834
6,80,2212.84,2532.17,0.3738,0.31,0.2135
7,80,2142.80,2528.75,0.3160,0.31,0.2002
8,80,2203.07,2622.36,0.3499,0.33,0.2702
9,100,1870.79,2252.79,Infinity,0.37,0.3649
10,100,1815.35,2296.86,0.3993,0.35,0.2252
11,100,1799.80,2302.34,Infinity,0.29,0.2070
12,120,1596.28,2072.50,Infinity,0.37,0.3615
13,120,1647.65,2016.03,Infinity,0.37,0.2986
14,120,1574.30,2017.44,Infinity,0.35,0.2983
15,140,1384.97,1814.12,Infinity,0.33,0.1877
16,140,1386.61,1886.10,Infinity,0.36,0.2952
17,140,1424.59,1972.93,Infinity,0.39,0.2604
18,160,1286.27,1761.47,Infinity,0.40,0.3293
19,160,1285.12,1791.66,Infinity,0.37,0.2691
20,160,1278.53,1737.79,Infinity,0.35,0.1942
21,180,1159.88,1669.53,Infinity,0.39,0.2790
22,180,1158.47,1748.95,Infinity,0.36,0.3017
23,180,1089.22,1648.59,Infinity,0.41,0.3047
```

```
24,200,1014.00,1596.33,Infinity,0.36,0.2267
25,200,1015.62,1605.15,Infinity,0.45,0.3217
26,200,1050.43,1606.78,Infinity,0.42,0.2820
```

You can see that these results have pretty poor predictive strength, and the clusters and points aren't stable. (A predictive strength of "Infinity" is bad.) That instability is a sign that the clusters aren't distinct enough. Try backing out half of the features you just added and see if things get better.

6. Modify your mapper

To begin backing out features, open up `features_map.py` and modify it as follows (changes in bold red):

```
#!/usr/bin/env python
import sys
import json
def main():
   # Read all lines from stdin
    for line in sys.stdin:
        session = json.loads(line)
        fields = []
        fields.append(session['session'])
        fields.append('updatePassword' in session['actions'])
        fields.append('updatePaymentInfo' in session['actions'])
        fields.append('verifiedPassword' in session['actions'])
        fields.append('reviewedQueue' in session['actions'])
        fields.append(session['kid'])
#       session_duration = (long(session['end']) - long(session['start']))
#       fields.append(session_duration)

        played = set(session['played'].keys())
#        browsed = set(session['browsed'])
#        hovered = set(session['hover'])
#        queued = set(session['queued'])
      recommendations = set(session['recommendations'])
      rated = set(session['rated'].keys())
      reviewed = set(session['reviewed'].keys())
      searched = set(session['searched'])

      fields.append(len(played))
#        fields.append(len(browsed))
#        fields.append(len(hovered))
#        fields.append(len(queued))
      fields.append(len(recommendations))
        fields.append(len(rated))
        fields.append(len(reviewed))
        fields.append(len(searched))

        print ','.join(map(str, fields))

if __name__ == '__main__':
    main()
```

7. Delete features from your header

Now open up the `header.csv` file and delete the lines that list those features. Your file should only contain:

```
session_id,identifier
updatePassword,categorical
updatePaymentInfo,categorical
verifiedPassword,categorical
reviewedQueue,categorical
kid,categorical
num_plays
num_recommendations
num_rated
```

```
num_reviewed
num_searched
```

Make sure that you have 11 lines in your `header.csv` file, which is the number of features produced by the Hadoop job. If you have more or fewer, it will cause the following steps to fail.

8. Rerun the workflow

Rerun the workflow with the following commands:

```
$ hadoop jar $STREAMING -D mapred.reduce.tasks=0 -D mapred.textoutputformat.separator=, -D
stream.map.output.field.separator=, -mapper features_map.py -file features_map.py -input merged -
output features2
...
$ hadoop fs -getmerge features2 features.csv
$ hadoop fs -rm features.csv
Deleted features.csv
$ hadoop fs -put features.csv
$ ml summary --summary-file summary.json --header-file header.csv --format text --input-paths
features.csv
...
$ ml normalize --summary-file summary.json --format text --id-column 0 --transform Z --input-paths
features.csv --output-path part2normalized --output-type avro
...
$ ml ksketch --format avro --input-paths part2normalized --output-file part2sketch.avro --points-per-
iteration 1000 --iterations 10 --seed 1729
...
$ ml kmeans --input-file part2sketch.avro --centers-file part2centers.avro --clusters
40,60,80,100,120,140,160,180,200 --best-of 3 --seed 1729 --num-threads 1 --eval-details-file
part2evaldetails.csv --eval-stats-file part2evalstats.csv
…
ID,NumClusters,TestCost,TrainCost,PredStrength,StableClusters,StablePoints
0,40,401.01,458.40,0.3236,0.58,0.7574
1,40,372.15,475.52,0.4980,0.60,0.7661
2,40,437.66,487.39,0.4781,0.63,0.7896
3,60,195.93,290.08,0.5132,0.67,0.8270
4,60,184.61,283.48,0.4976,0.70,0.8528
5,60,200.65,271.64,0.4928,0.70,0.8637
6,80,116.12,222.76,1.0000,0.81,0.9489
7,80,99.11,220.22,0.4989,0.74,0.8494
8,80,91.20,206.89,0.4980,0.83,0.8819
9,100,46.35,166.70,0.7908,0.78,0.8668
10,100,46.39,156.36,0.9835,0.81,0.9148
11,100,45.25,159.04,0.9763,0.85,0.9247
12,120,16.59,136.78,0.9763,0.87,0.9364
13,120,15.79,143.98,0.9958,0.88,0.9572
14,120,14.25,145.20,0.9763,0.88,0.9478
15,140,6.40,127.42,1.0000,0.94,0.9375
16,140,4.01,128.81,1.0000,0.92,0.9690
17,140,4.41,127.70,1.0000,0.93,0.9784
18,157,0.66,124.60,1.0000,0.98,0.9966
19,157,0.75,124.60,1.0000,0.99,0.9992
20,157,0.55,124.60,1.0000,0.98,0.9958
21,157,0.00,124.60,1.0000,1.00,1.0000
22,157,0.00,124.60,1.0000,1.00,1.0000
23,157,0.00,124.60,1.0000,1.00,1.0000
24,157,0.00,124.60,1.0000,1.00,1.0000
25,157,0.00,124.60,1.0000,1.00,1.0000
26,157,0.00,124.60,1.0000,1.00,1.0000
```

Those results look good, so try adding back the features you removed one by one.

If your jobs are failing with an `ArrayIndexOutOfBoundsException`, it is most likely because the length of the `header.csv` file does not match the number of features produced by the Hadoop job.

9. Test features

First, add back the session duration and rerun the workflow. The results tell you that feature causes suboptimal clusters, so remove it again. Next, add back the number of items browsed. The results tell us that feature is OK. You should note, however, that the number of clusters found only goes up from 132 to 139, so the feature is not highly discriminating. It doesn't, however, have any negative impact. Next, add the number of items hovered over. The results reveal that number of items hovered over isn't a helpful feature; discard it. Finally, add back the number of items queued, and the results tell us that the feature seems to be OK.

Follow the same process with adding the item-related features, and finally, add in the play-related features. One thing that should become clear, while working in the item-related features, is that the number of items queued interferes with the item-related features. Leaving out the number of items queued returns much better results. Further, adding those features returned statistics that often look like:

```
6,200,985.13,3060.24,Infinity,0.71,0.5322
```

As mentioned above, the "Infinity" in the predictive strength column is generally a bad sign. When you see these kinds of results coupled with low stability scores, you need to remove features to improve the clustering.

The final set of features is:

- Actions (a feature for each, except login and logout)
- Number of items played
- Number of items browsed
- Number of items reviewed
- Number of items rated
- Number of items searched
- Number of recommendations that were reviewed
- Kid (parental controls)
- Total play time as fraction of session duration
- Longest play: less than 5 minutes, between 5 and 60 minutes, more than 60 minutes
- Shortest play: less than 5 minutes, between 5 and 60 minutes, more than 60 minutes
- Number of items played less than 5 minutes
- Number of items played more than 60 minutes
- Number of browsed items that were played
- Number of reviewed items that were played
- Number of recommended items that were played
- Number of rated items that were played
- Number of searched items that were played
- Number of popular items that were played
- Number of queued items that were played
- Number of recent items that were played
- Number of recent items that were reviewed
- Number of recent items that were rated

The final results from `kmeans` are:

```
ID,NumClusters,TestCost,TrainCost,PredStrength,StableClusters,StablePoints
0,160,789.77,2735.04,0.9510,0.75,0.8718
1,160,721.72,2738.69,0.9856,0.75,0.8807
2,160,734.02,2695.29,0.9858,0.74,0.8701
3,180,671.13,2549.84,0.9886,0.74,0.7437
4,180,585.03,2497.42,0.8087,0.78,0.8499
5,180,612.79,2471.32,0.9512,0.79,0.8900
6,200,455.90,2378.06,0.9019,0.78,0.8852
7,200,534.36,2423.53,0.9511,0.79,0.8670
8,200,466.82,2401.94,0.9919,0.78,0.8136
9,220,325.82,2274.35,0.9883,0.79,0.8482
10,220,339.19,2317.40,0.9543,0.80,0.8951
11,220,321.32,2289.31,0.9807,0.83,0.9363
```

```
12,240,254.95,2171.10,0.8186,0.84,0.9178
13,240,264.19,2209.75,0.9807,0.82,0.8941
14,240,281.65,2241.30,0.9758,0.86,0.8924
15,260,218.12,2170.11,0.9807,0.88,0.9483
16,260,208.25,2167.37,0.9953,0.86,0.9143
17,260,222.46,2180.43,0.9883,0.87,0.9198
18,280,130.98,2129.63,0.9543,0.88,0.9140
19,280,142.16,2125.83,0.9883,0.87,0.8920
20,280,146.22,2117.22,0.9497,0.88,0.9184
21,300,97.29,2071.38,1.0000,0.88,0.9304
22,300,90.93,2081.57,1.0000,0.92,0.9620
23,300,93.16,2052.24,0.9542,0.89,0.9198
24,320,54.85,2030.90,0.9729,0.94,0.9575
25,320,56.50,2042.54,0.9729,0.93,0.9493
26,320,61.67,2027.72,1.0000,0.91,0.9486
```

You now have the maximum set of features that return the best cluster statistics. The last step is to let Cloudera ML generate the clustering for you. You can see from the last output that clustering number 22 looks like the best choice. To generate the clustering from the normalized data and the clusters you found, run:

```
$ ml kassign --input-paths part2normalized --format avro --centers-file part2centers.avro --center-ids
22 --output-path part2assigned --output-type csv
…
```

If we wanted to try out more than one clustering, we could have given a comma-separated list of center IDs. To turn the `kassign` output into the final answer format, run the following:

```
$ hadoop fs -cat part2assigned/part\* | cut -d, -f1,3 > Task2Solution.csv
```

Here the `cut` command removes everything but the first and third columns in a comma-delimited file.

This set of features yields 300 clusters which would yield a score of 96.27% in the actual challenge.