IS622 Lustering pt 2 Code

James Quacinella

10/25/2015

Week 8

Data

I could not use my original dataset of tweets, as I could not really think of an example of what data to use for clustering. Using textual features, then it would be non-euclidean, and I could not figure out a good euclidean way of measuring distances between meaningful points.

For Week 8's KMeans via RHadoop, I used a data set I found online from UCI: http://archive.ics.uci.edu/ml/datasets/User+Knowledge+Modeling I chose this data set because it is euclidean, which means I could use KMeans and CURE (I did not want to attempt the other algorithm as it seems above my head to implement right now), and the data looks normalized enough to not worry about pre-processing.

I used the RHadoop tutorial notes for K-means. I did not a get a chance to submit code for week 3 and since I lost points (and don't want to lose time), I am using a pre-made implementation.

```
library(rJava)
library(rhdfs)
```

```
##
## HADOOP_CMD=/home/yarn/hadoop/bin/hadoop
##
## Be sure to run hdfs.init()
```

```
hdfs.init()
library(rmr2)
```

Week 8 - KMeans Hadoop

Lets load the data:

```
# Load Data
#setwd("/home/james/Code/Masters//IS622/Week9")
clusterdata <- read.csv('data.csv')

# Params: I expect 7  clusters
num_clusters = 10;
```

```
# https://github.com/RevolutionAnalytics/rmr2/blob/master/pkg/tests/kmeans.R
kmeans.mr =
  function(
    P,
    num.clusters,
    num.iter,
```

```
    combine,
    in.memory.combine) {
    dist.fun =
      function(C, P) {
        apply(
          C,
          1,
          function(x)
            colSums((t(P) - x)^2))}

    kmeans.map =
      function(., P) {
        nearest = {
          if(is.null(C))
            sample(
              1:num.clusters,
              nrow(P),
              replace = TRUE)
          else {
            D = dist.fun(C, P)
            nearest = max.col(-D)}}
        if(!(combine || in.memory.combine))
          keyval(nearest, P)
        else
          keyval(nearest, cbind(1, P))}

    kmeans.reduce = {
      if (!(combine || in.memory.combine) )
        function(., P)
          t(as.matrix(apply(P, 2, mean)))
      else
        function(k, P)
          keyval(
            k,
            t(as.matrix(apply(P, 2, sum))))}


    C = NULL
    for(i in 1:num.iter ) {
      C =
        values(
          from.dfs(
            mapreduce(
              P,
              map = kmeans.map,
              reduce = kmeans.reduce)))
      if(combine || in.memory.combine)
        C = C[, -1]/C[, 1]


      if(nrow(C) < num.clusters) {
        C =
          rbind(
```

```
            C,
            matrix(
              rnorm(
                (num.clusters -
                   nrow(C)) * nrow(C)),
              ncol = nrow(C)) %*% C) }}
        C}


# Sample runs
out = list()

# For now only do local
for(be in c("local")) {
  # Set RMR backend and set random seed
  rmr.options(backend = be)
  set.seed(0)

#   # Input data (random)
#   P =
#     do.call(
#       rbind,
#       rep(
#         list(
#           matrix(
#             rnorm(10, sd = 10),
#             ncol=2)),
#         20)) +
#     matrix(rnorm(200), ncol =2)

  # Input data is from europe
  P = clusterdata

  # Generate output from Hadoop job
  out[[be]] =
    kmeans.mr(
      to.dfs(P),
      num.clusters = num_clusters,
      num.iter = 5,
      combine = FALSE,
      in.memory.combine = FALSE)
}

# Show output from local run
out$local[order(out$local[,1]), ]
```

```
##              X0      X0.1      X0.2      X0.3      X0.4
##  [1,] 0.2000000 0.2213000 0.6205000 0.4140000 0.1500000
##  [2,] 0.2075926 0.2525926 0.6381481 0.3074074 0.7162963
##  [3,] 0.2568421 0.1776316 0.3415789 0.7515789 0.2442105
##  [4,] 0.2727353 0.5123235 0.3155882 0.5652941 0.2273529
##  [5,] 0.3370250 0.2853250 0.2138750 0.2342500 0.6265000
##  [6,] 0.3735862 0.6717241 0.5751724 0.2427586 0.6697586
```

```
##  [7,] 0.4703478 0.3337391 0.7547826 0.7617391 0.4547826
##  [8,] 0.5276207 0.2507586 0.6758621 0.2179310 0.4800000
##  [9,] 0.5457727 0.2485455 0.2786364 0.4531818 0.2481818
## [10,] 0.6826429 0.6010714 0.3971429 0.8100000 0.6621429
```

```
# Show output from hadoop run
#out$hadoop[order(out$hadoop[,1]), ]
```

Builtin KMeans

```
# https://stat.ethz.ch/R-manual/R-devel/library/stats/html/kmeans.html
library(stats)
model.kmeans.builtin <- kmeans(P, num_clusters)
model.kmeans.builtin$centers[order(model.kmeans.builtin$centers[,1]),]
```

```
##           X0      X0.1      X0.2      X0.3      X0.4
## 3  0.2437778 0.2370370 0.3305556 0.2544444 0.6900000
## 2  0.3100556 0.2659444 0.5611111 0.7369444 0.2163889
## 5  0.3133333 0.2308718 0.2123077 0.3717949 0.2382051
## 10 0.3246190 0.2097619 0.6690476 0.2742857 0.2252381
## 7  0.3273077 0.6169231 0.3834615 0.6088462 0.2284615
## 4  0.3393846 0.6850000 0.6046154 0.2461538 0.6543462
## 6  0.3692857 0.2660571 0.7191429 0.2742857 0.6442857
## 8  0.5413077 0.4258462 0.7507692 0.8238462 0.7023077
## 9  0.5844211 0.4042105 0.2552632 0.2189474 0.7194737
## 1  0.7050000 0.4002000 0.2546667 0.7520000 0.6373333
```

Comparison of KMean Clusters

Deprecated: when using 2-d data from the tutorial, I could grpah cluster centers. Not possible now but including for notes:

```
library(ggplot2)
ggplot(data=as.data.frame(out$local)) +
  geom_point(aes(x=V1, y=V2, colour='blue')) +
  geom_point(data = as.data.frame(model.kmeans.builtin$centers), aes(x=V1, y=V2, colour='red')) +
  scale_color_manual(values = c("red", "black"),
                     labels = c("KMeans Hadoop",
                                "KMeans Stats Module")) +
  ggtitle("KMeans Centers for P") +
  xlab("X-coord") + ylab("Y-coord")
```

Week 9

I will continue to work on this, but implementing these algorithms seems to be above my head for the time I can allocate to the class. Here I will outline the algorithm and show some psuedo code I hope to fill in.

Pre-Hadoop Phase

1. Take a small sample of the data and cluster it in main memory.
2. Select a small set of points from each cluster to be representative points. These points should be chosen to be as far from one another as possible, using the method described in Section 7.3.2.

After all of this, we have a list of clusters, each with a set of representative points and a cluster center, or centroid.

3. Move each of the representative points a fixed fraction of the distance between its location and the centroid of its cluster. Perhaps 20% is a good fraction to choose.

After all of this, we have a list of clusters, each with a set of representative points (that are now a little closer to the centroid of the cluster) and a cluster center, or centroid.

Hadoop Phase

The next phase of CURE is to merge two clusters if they have a pair of representative points, one from each cluster, that are sufficiently close. That means, we will have multiple map-reduce phases until we detect that the cluster assignments have not changed since last time. The map phase will generate key value pairs of the form $(1, (cluster_i dx1, cluster_i dx2, dist))$. The reduce task gets all the keys, since they are all one, and will merge clusters if any *dist* is less than some set threshold. The hadoop phase stops when there are no new cluster asisgnments.

Post Hadoop

This is not specific to CURE, but once we have the cluster asisgnments, we can go back to the whole data set and assign a a cluster to each point.

```
cure.mr = function(P, cluster.dist.threshold, max.iter) {
    # Filter P, the data set, to only take a small portion of the data

    # From this small selection, run heirarchical clustering to get an initial clustering

    # For each cluster
      # Find all points in cluster
      # Computer centroid
      # Move all points 20% closer in each direction / dimension


    cure.map = function(., P) {
      # For each cluster
        # For every other cluster
          # Generate (1, (cluster1, cluster2, dist(cluster1, cluster2))) key value pairs
```

```
    }


    cure.reduce = {
      # For each key value pairs
        # If dist < cluster.dist.threshold
          # Merge cluster1 and cluster2
          # Which means delete cluster1 and cluster2 from assignments, use all points from those
          # clusters to re-calc the new centroid

      # Output new cluster mappings
    }


    # Main control of algo

    # Place to store new clusters
    C_new = NULL

    # Looping just to keep a maximum number of loops
    for(i in 1:max.iter ) {
      # Map-reduce call will use C as a global, copying it, updating it
      # and returning it via DFS
      C_new = values(
              from.dfs(
                mapreduce(P,
                          map = cure.map,
                          reduce = cure.reduce)
              )
            )

      if (C == C_new) {
        break; # we are done when there is no change
      }
      else {
        C = C_new # Update cluster assignments
      }
    }

  # Return clusters
  C
}
```