2012 - Cluster Lab

Monday, June 22, 2009
4:50 PM

PURPOSE

Sometimes, an unsupervised learning technique is preferred. Perhaps you do not have access to adequate training data. Or perhaps the classifications for the training data's labels events are not completely clear. Or perhaps you just want to quickly sort real-world, unseen, data into groups based on it's feature similarity. Regardless of your situation, clustering is a great option!

CLUSTERING

Now we're going to try clustering with a familiar bunch of audio files and code. Sorry, the simple drum loop is going to mak e an appearance again. However, once we prove that it works - you can experiment with other audio collections that are posted.

1. Create a new .m file for your code.

2. Load simpleLoop.wav. (Sorry - we'll use other audio files soon! It's best to start simple - because if it don't work for this file, we have a problem.)

3. Segment this file into 100ms frames **based on the onsets**.

4. Now, feature extract the frames using ***only*** zero crossing and centroid. Store the feature values in one matrix for both the kick and the snares… remember, we don't care about the labels with clustering - we just want to create some clustered groups of data.

5. Scale the features (using the scale function) from -1 to 1. (See Lab 2 if you need a reminder.)

6. It's cluster time! We're using NETLAB's implementation of the kmeans algorithm.

   Use the kmeans algorithm to create clusters of your feature. kMeans will output 2 things of interest to you:

   (1) The center-points of clusters. You can use the coordinates of the center of the cluster to measure the distance of any point from the center. This not only provides you with a distance metric of how "good" a point fits into a given cluster, but this allows you to sort by the points which are closest to the center of a given frame! Quite useful.

   (2) Each point will be assigned a label, or cluster #. You can then use this label to produce a transcription, do creative stuff, or further train another downstream classifier.

   Now, simply put, here are some examples of how you use it:

```
% Initialize # of clusters that you want to find and their initial conditions.
numCenters = 2;          % the size of the initial centers; this is passed to k-means to determine the value of k.
numFeatures = 2;       %  replace the "2" with however many features you have extracted
centers = zeros(numCenters , numFeatures );          % inits center points to 0

% setup vector of options for kmeans trainer
options(1) = 1;
options(5) = 1;
options(14) = 50;   % num of steps to wait for convergence

% train centers from data
Post  = kmeans( your_feature_data_matrix , numCenters );

%Output:
%    %    Post contains the assigned cluster number for each point in your feature matrix.  (from 1 to k)
```

7. Write a script to list which audio slices (or audio files) were categorized as Cluster # 1. Do the same or Cluster # 2. Do the clusters make sense? Now, modify the script to play the audio slices that in each cluster - listening to the clusters will help us build intuition of what's in each cluster.

8. Repeat this clustering (steps 3-7), and listening to the contents of the clusters with **CongaGroove-mono.wav**.

9. Repeat this clustering (steps 3-7) using the CongaGroove and **3 clusters. Listen to the results. Try again with 4 clusters.** Listen to the results. **(etc, etc…)**

10. Once you complete this, try out some of the many, many other audio loops in the audio loops. (Located In audio\Miscellaneous Loops Samples and SFX)

11. Let's add MFCCs to the mix. Extract the mean of the **12 MFCCs (coefficients 1-12, do not use the "0th" coefficient) for each onset** using the code that you wrote. Add those to the feature vectors, along with zero crossing and centroid. We should now have 14 features being extracted - this is started to get "real world"! With this simple example (and limited collection of audio slices, you probably won't notice a difference - but at least it didn't break, right?) Let's try it with the some other audio to truly appreciate the power of timbral clustering.

BONUS (ONLY IF YOU HAVE EXTRA TIME…)

12. Now that we can take ANY LOOP, onset detect, feature extract, and cluster it, **let's have some fun.**
Choose any audio file from our collection and use the above techniques break it up into clusters.
Listen to those clusters.

    Some rules of thumb: since you need to pick the number of clusters ahead of time, listen to your audio files first.
    - You can break a drum kit or percussion loop into 3 - 6 clusters for it to segment well. More is OK too.
    - Musical loops: 3-6 clusters should work nicely.
    - Songs - lots of clusters for them to segment well. Try 'em out!

BONUS (ONLY IF YOU REALLY HAVE EXTRA TIME…)

13. Review your script that **PLAYs** all of the audio files that were categorized as Cluster # 1 or Cluster # 2.
Now, modify your script to play and plot the audio files which are closest to the center of your clusters.

    This hopefully provides you with which files are representative of your cluster.

    Helpful Commands for sorting or measuring distance:

```
d = dist2( featureVector1 ,featureVector2 )   % measures the Euclidean distance betw/ point 1 and point 1
```

```
DIST2   Calculates squared distance between two sets of points.
        Description
        D = DIST2(X, C) takes two matrices of vectors and calculates the
        squared Euclidean distance between them.  Both matrices must be of
        the same column dimension.  If X has M rows and N columns, and C has
        L rows and N columns, then the result has M rows and L columns.  The
        I, Jth entry is the  squared distance from the Ith row of X to the
        Jth row of C.
```

```
[y,ind] = sort(    )
```

>> help sort

```
SORT   Sort in ascending or descending order.
  For vectors, SORT(X) sorts the elements of X in ascending order.
  For matrices, SORT(X) sorts each column of X in ascending order.
  For N-D arrays, SORT(X) sorts the along the first non-singleton
  dimension of X. When X is a cell array of strings, SORT(X) sorts
  the strings in ASCII dictionary order.
```

Y = SORT(X,DIM,MODE)
has two optional parameters.
DIM selects a dimension along which to sort.
MODE selects the direction of the sort
'ascend' results in ascending order
'descend' results in descending order
The result is in Y which has the same shape and type as X.

[Y,I] = SORT(X,DIM,MODE) also returns an index matrix I.
If X is a vector, then Y = X(I).
If X is an m-by-n matrix and DIM=1, then
for j = 1:n, Y(:,j) = X(I(:,j),j); end

When X is complex, the elements are sorted by ABS(X). Complex matches are further sorted by ANGLE(X).

When more than one element has the same value, the order of the elements are preserved in the sorted result and the indexes of equal elements will be ascending in any index matrix.

Example: If X = [3 7 5
0 4 2]

then sort(X,1) is [0 4 2 and sort(X,2) is [3 5 7
3 7 5] 0 2 4];

[y,ind] = sortrows ( )

SORTROWS Sort rows in ascending order.
Y = SORTROWS(X) sorts the rows of the matrix X in ascending order as a group. X is a 2-D numeric or char matrix. For a char matrix containing strings in each row, this is the familiar dictionary sort. When X is complex, the elements are sorted by ABS(X). Complex matches are further sorted by ANGLE(X). X can be any numeric or char class. Y is the same size and class as X.

SORTROWS(X,COL) sorts the matrix based on the columns specified in the vector COL. If an element of COL is positive, the corresponding column in X will be sorted in ascending order; if an element of COL is negative, the corresponding column in X will be sorted in descending order. For example, SORTROWS(X,[2 -3]) sorts the rows of X first in ascending order for the second column, and then by descending order for the third column.

[Y,I] = SORTROWS(X) and [Y,I] = SORTROWS(X,COL) also returns an index matrix I such that Y = X(I,:).

[y,ind] = sortrows (featureData_from_a_particular_cluster, clusterNum)