1 Introduction

There are clear distinctions between the object “car” and “cat”, but how are these labels assigned? It is easy for a person to group images into different labeled categories, but far less so for a computer. If a computer did possess the ability to classify unlabeled images (or parts of images), it could be applied to a number of technologies; including true image search engines and automatic medical diagnostics. In this report, I present an implementation of a system designed to do just that; automatically label categories.

My partner, Ms. Panagiota Stratou, and myself implemented a subset of the work presented in Grauman and Darrel’s recent paper, “Unsupervised Learning of Categories from Sets of Partially Matching Image Features”. Our implementation does not perform as well as hoped, classifying a very small number of categories barely better than chance, although with parameter tuning and a few minor improvements I believe it should work significantly better. After our first set of experiments, we have suggestions for improving both the speed and accuracy of our implementation.

2 Methods

The unlabeled image categorization algorithm we implemented has two major components. The first is a method to score affinity between images or portions of images. The other locates groups in a matrix of image affinity scores. Grauman and Darrel include a third component, identifying prototypical images through the use of feature masks. Unfortunately, there was not enough time to implement this, and it does not affect the initial coarse clustering of images.

Affinity scores between images are calculated using the pyramid match kernel

\[ \text{Pyramid Match Kernel} \]

---

The PMK approximates optimal nearest neighbor matching between pairs of image feature-sets with multiresolution histograms. Features are located in images using Harris-Affine interest operators, and constructed using the PCA-SIFT variant of SIFT features. The PCA-SIFT features were restricted to ten dimensions to make the scoring problem more computationally tractable.

Even with dimensional reduction, a naive implementation still has unreasonable storage requirements. For example, if the range of feature values across all images is $1e4$, which is not unreasonable in practice, then the finest resolution histogram contains $1e4 \times 1e10$ elements. Note that we did initially try the naive implementation, and discovered that Matlab cannot handle $1e14$-element matrices. To solve this problem, we took advantage of the fact that the histograms are very sparse, so the intersection between two can be calculated directly by comparing individual feature vectors.

The matrix of affinity scores is generated by calculating the PMK of each image to every other image, including the self-affinity. The self-affinity scores compose the diagonal of this triangular matrix. The affinity matrix has dimensions $N \times N$, where $N$ is the total number of images being categorized. The grouping portion of the algorithm uses min-normalized graph partitioning.

3 Implementation

Our unlabeled image categorization implementation was split into two sections that correspond to the two components mentioned above. Ms. Stratou implemented the partitioning section, while I dealt with generating the affinity matrix.

Features were extracted automatically from all images in the Caltech-256 dataset with a precompiled binary, a Perl script and a Sun Grid Engine. The resulting feature sets were formatted with a separate Perl script and read into Matlab. The outer loop of the pyramid match kernel was implemented in a Matlab script, while the inner loop was C source code compiled with MEX for speed. The final min-normalized graph partition was performed solely within Matlab.

---

3.1 Feature Extraction

Features were extracted from 25,600 images with a Linux binary from the Robotics Research Group at the University of Oxford. I extracted PCA-SIFT feature descriptors that were detected with Harris-Affine interest points. The program is also capable of using other feature detection and description methods, but we did not have time to explore them.

The expected sequential runtime of the feature extraction program on all images was on the order of weeks, so I choose to exploit the process-level parallelism provided by a Sun Grid Engine. I used a set of Perl scripts to parse the Caltech-256 directory tree and spawn a new Grid Engine job for every category. The scripts are attached in Appendix A. In this manner, each processor on the Grid processed roughly 100 images at a time, which was far more manageable. With unrestricted access to the Grid’s twenty Xeon-class machines, I was able to generate feature data for the entire Caltech-256 dataset in under 12 hours.

3.2 Data Formatting

The files produced by the feature extraction executable contain a lot of spurious information that has to be excluded before the features can be read into Matlab. Another Perl script was written and used to remove all of the columns not relevant to feature coordinates, as well as reduce the data to ten dimensions. The script also lexicographically sorts the features by their coordinate values, a necessary step for performing sparse-PMK. The script source code is attached in Appendix B.

3.3 The Pyramid Match Kernel

Since the naive implementation of the pyramid match kernel could not be computed in reasonable time or space, I implemented a version that exploits the histogram sparsity. Histograms of the feature data are never explicitly calculated, though intersections between multiple histograms are.

Suppose there are two images, \(I_1\) and \(I_2\), along with their associated multiresolution histogram sets \(\{H^{1,1}_1, H^{0}_1, H^{1}_1, \ldots H^{n}_1\}\) and \(\{H^{2,1}_2, H^{0}_2, H^{1}_2, \ldots H^{n}_2\}\). Instead of comparing histograms from the multiresolution sets directly, we instead compare the feature sets which the histograms are derived from. For instance, in order to compare the \(H^{0}_1\) and \(H^{0}_2\) histograms (\(H^{1,1}_1\) and \(H^{2,1}_2\) share no points by construction), their feature sets are first lexicographically sorted and then compared through pointer walking. The pointer walking algorithm is illustrated in pseudocode below.

```plaintext
PointerWalking(list1 list2)
    shared <- 0
    ptr1 <- 0
    ptr2 <- 0
```
WHILE (ptr1 != endOfList1) & (ptr2 != endOfList2)
    val1 <- list1[ptr1]
    val2 <- list2[ptr2]
    IF (val1 > val2)
        ptr2 <- ptr2 + 1
    FI
    ELSEIF (val1 < val2)
        ptr1 <- ptr1 + 1
    FI
    ELSEIF (val1 = val2) // and we don’t run off the end of the list
        count1 <- 0
        count2 <- 0
        WHILE (val1 = val2)
            count1 <- count1 + 1
            count2 <- count2 + 1
            ptr1 <- ptr1 + 1
            ptr2 <- ptr2 + 1
            val1 <- list1[ptr1]
            val2 <- list2[ptr2]
        ENDWHILE
        shared <- shared + count1 // count1 and count2 should be equal
    FIESLE
    ENDWHILE
    RETURN shared

Custom comparison functions are also necessary for the feature vectors, in order to determine if two vectors are “less than”, “greater than” or “equal”. At the end of the pointer walking step, the intersection between two histograms is returned, where the intersection for two bins being compared is equivalent to \( \min(\text{binCount}_1, \text{binCount}_2) \).

I wrote the intersection-calculating function in C, since it is the inner loop of the PMK. Even with this optimized version of the function, Matlab’s profiling functions showed that “intersection” calls dominated the clock time of the algorithm. This code is attached in Appendix D.

The outer loop of the PMK, which deals with data normalization; histogram resolution changes; and iteration over the data set, is scripted entirely in Matlab. The feature data is normalized by applying a rounding function, which guarantees with high probability that every feature is in its own bin at the finest resolution level. The outer loop Matlab source is attached in Appendix C.

The fastest way to calculate intersections for resolutions below the finest is to modify the feature vector directly. This is done by dividing a feature vector \( \vec{v} \) by the resolution \( r \) and rounding towards zero. In other words, \( \lceil \frac{v}{r} \rceil \) for feature vector members below zero and \( \lfloor \frac{v}{r} \rfloor \) for members above zero. Rounding towards zero guarantees that at the coarsest resolution every feature vector ends up in the same bin. A better way to accomplish this would shift every point by a constant
factor to guarantee that it is above zero at the start of the algorithm, then apply
the formula \( \left\lfloor \frac{v}{r} \right\rfloor \) consistently across all members for resolution changes.

This method directly calculates the intersection of two histograms while only
indirectly representing the histograms. It is only necessary, in the worst case,
to compare the sum of the size of the two sets of feature vectors. Though it
is not explicitly mentioned, this is the same method that Grauman and Darrel
use in their 2006 CVPR paper.

### 3.4 Graph Partitioning

We did not normalize the affinity matrix because it is not mentioned in the origi-
ナル paper. However, normalizing the matrix such that each diagonal member of
the matrix is set to unity is one of the things I would like to try to improve per-
formance of the algorithm. Two different techniques were used to partition the
matrix. For the 2-category experiment, a simple 2nd-from-top eigenvalue par-
titioning was used, which could be extended to more categories using recursive
bi-partitioning. K-means was used for partitioning the 8-category experiment.
As I have mentioned, neither gave satisfactory results. The partitioning Matlab
source code is attached in Appendix E.

### 4 Results

We ran the algorithm on two datasets, the trivial dataset with two categories
and a more advanced dataset with eight categories. An attempt was made
to run a 16 category dataset, but a single machine was unable to handle the
memory requirements. All of the datasets were drawn from the Caltech-256 in
numerical order. Thus the 2-category dataset was ak47s and american flags,
and the 8-category dataset was ak47s, american flags, backpacks, baseball bats,
baseball gloves, basketball hoops, bats and bathtubs. All of the images in every
category were used as input in the experiments.

The categorization experiment results are presented in Table 1 and Table
2. Each column represents the actual category boundaries, while each row
represents how our implementation split up the categories. If the categorization
algorithm were perfect, you would expect these matrices to be diagonal. Since
they are obviously not, our implementation needs revision.

Of the two experiments, the 2-category one shows better results than the 8-
category one, in that there’s at least one category which excludes other images.
However, it seems that the partitioning algorithm is being too strict in how it
clusters images. For the 8-category experiment there is an obvious pattern to
the data, but I admit I do not know how to interpret it in order to know what
needs to be fixed.

---

8Kristen Grauman, e-mail message to author, May 19, 2006.
<table>
<thead>
<tr>
<th>Cat 1</th>
<th>Cat 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>98</td>
<td>0</td>
</tr>
<tr>
<td>82</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 1: Categorization Results for 2 Categories

<table>
<thead>
<tr>
<th>Cat 1</th>
<th>Cat 2</th>
<th>Cat 3</th>
<th>Cat 4</th>
<th>Cat 5</th>
<th>Cat 6</th>
<th>Cat 7</th>
<th>Cat 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>30</td>
<td>39</td>
<td>25</td>
<td>40</td>
<td>30</td>
<td>40</td>
<td>61</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>36</td>
<td>28</td>
<td>64</td>
<td>48</td>
<td>70</td>
<td>43</td>
<td>34</td>
<td>104</td>
</tr>
<tr>
<td>8</td>
<td>28</td>
<td>12</td>
<td>8</td>
<td>7</td>
<td>11</td>
<td>25</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>11</td>
<td>36</td>
<td>43</td>
<td>31</td>
<td>5</td>
<td>5</td>
<td>55</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2: Categorization Results for 8 Categories

4.1 Further Improvement

There are some obvious areas where this implementation could be improved, and foremost are questions of performance and correctness. The performance of our implementation is dismal when compared to the paper it is based on. While this is disappointing, I believe that our implementation’s performance could be dramatically improved with some small improvements.

I think the normalization (or lack thereof) of the affinity matrix is suspicious. I am curious to run further experiments with a properly normalized affinity matrix to see if there is a commensurate improvement in the results.

I also think that the feature vectors should have their values shifted above zero, so that a consistent vector transform can be applied to all values. I suspect that something odd occurs to feature vectors as they are rounded in different directions which could negatively affect the algorithm’s performance.

Finally, to extend this work, it is possible to increase the number of categories in an experiment using the Sun Grid Engine. Computing the value of any given element in the affinity matrix is independent of the other values and therefore easily parallelizable. The affinity matrix could be segmented along category boundaries, each submatrix calculated separately, and then reassembled. In this manner, it would be possible to tackle a larger dataset than was possible on a single machine in this project.

I also think it would be interesting to explore more incremental categorization schemes. This work requires that all of the images which are being categorized for all time be present at the start of the algorithm. While it works, it is probably not the way our brains perform categorization. That being said, I consider this a novel and interesting application of a graph partitioning algorithm.
A Feature Extraction Source
#!/usr/bin/perl
#
# Process images in the image database directory
# Output them to the data directory
# Also converts images to ppm files. I'll have to rewrite this if we
# ever decide to run it a second time.

$root_image_dir = "/home/sidd/projects/Dom/Vision/256_ObjectCategories";
$root_output_dir = "/cs/research/ic/projects/Dom/Vision/data";
$extract_features_dir = "/cs/research/ic/projects/Dom/Vision/code"; # Path to extract features executable

# For the argument directory, get the list of files there, process them,
# and filter the output.
$dir = SARGV[0]; # Directory images are located in
$output_dir = "$root_output_dir/$dir" . ".out";
$image_dir = "$root_image_dir" . "/" . "$dir";

# Create an output directory that mirrors this subdirectory
print "Making $output_dir output directory\n";
mkdir "$output_dir" or die "Can't make directory $output_dir\n";

# Enter the directory and process image files there
print "Entering $image_dir\n";
chdir $image_dir;
@ls = `ls`;

# Filter out non-jpeg files from @ls
@images = grep('/.jpg', @ls);
print "@images";
chomp(@images); # remove all newlines
# Extract the root names of the images
@roots = @images;
chop(@roots);chop(@roots);chop(@roots); # remove the last three characters
print(@roots);

# 1) Convert jpeg-compressed images into ppm files
# 2) Extract features from every image and store the results in the
# output directory.
foreach $root (@roots) {
    $input_jpg = "$root" . "jpg";
    $output_ppm = "$root" . "ppm";
    $output_pca = "$root" . "pca";

    # Convert jpg to ppm
    `jpegtopnm $input_jpg > $output_ppm`;
    # Extract features and write out pca
    `exract_features_dir/extract_features.ln -haraff -pca -i $output_ppm -o1 $output_dir/$output_pca -thres 1000`;
}
B Data Formatting Source
#!/usr/bin/perl
#
# Trim the first n and last m columns from the data

print "Warning: Be careful that the correct columns are being deleted.\n";

$dir = $ARGV[0]; # Directory data is in
$output_dir = "$dir" . "/trimmed-data";
rm -rf $output_dir;
mkdir "$output_dir" or die "Can't make directory $output_dir\n";

# Find all the files in the data directory
chdir $dir;
@ls = `ls`;
chomp(@ls);
@ls = grep(/.pca/, @ls);

# For each file in the directory, trim the data
foreach $file (@ls) {
    # Open input file and convert to array
    open (INPUT, $file) or die "Can't open file $file.\n";
    @input = <INPUT>;
    close INPUT;

    $size = @array;
    # Drop the first two lines of input
    shift(@input);
    shift@input;

    # Open the output file
    $output_file = ">$output_dir" . "/$file";
    open (OUTPUT, $output_file) or die "Can't open file $output_file for writing.\n";

    foreach $feature (@input) {
        # Split each feature into columns
        @columns = split / /, $feature;
        print OUTPUT "@columns[5...14]\n";
    }

    close OUTPUT;

    $outputted_file = ">$output_dir" . "/$file";
    $tmp = "$output_dir" . "/tmp";
    # Sort lexicographically by column
    sort -b -n -k 1 -k 2 -k 3 -k 4 -k 5 -k 6 -k 7 -k 8 -k 9 -k 10 $outputted_file > $tmp; # or die "Error: $!\n";
    cat $tmp > $outputted_file;
}
rm -rf $tmp
C Intersection C Source
```c
#include "mex.h"
/*@include <math.h>*/

#define HIST1_IN prhs[0]
#define HIST2_IN prhs[1]
#define INTERSECTION_OUT plhs[0]
#define GT 0
#define LT 1
#define EQ 2
#define BUFFER 5000

int fCompare(double[], double[]);
void intersection(mxArray*[], double[]);
void mexFunction(int, mxArray*[], int, const mxArray*[]);

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    INTERSECTION_OUT = mxCreateDoubleScalar(1.0);
    intersection(prhs, mxGetPr(INTERSECTION_OUT));
}

int fCompare(double f1[], double f2[])
{
    int i = 0;
    for (i=0; i<10; i++) {
        if (f1[i] < f2[i])
            return LT;
        if (f1[i] > f2[i])
            return GT;
    }
    return EQ;
}

void intersection(mxArray *prhs[], double out[])
{
    int col = 0;
    int row = 0;
    double hist1[BUFFER][10];
    double hist2[BUFFER][10];
    double *holdVal;
    int count = 0;
    int hist1cnt = 0;
    int hist2cnt = 0;
    int relation = 0;
    int hist1points = 0;
    int hist2points = 0;
    int i = 0;
    int j = 0;

    for (col=0; col<mxGetN(HIST1_IN); col++)
        for (row=0; row<mxGetM(HIST1_IN); row++)
            hist1[row][col] = mxGetPr(HIST1_IN)[row+col*mxGetM(HIST1_IN)];
    for (col=0; col<mxGetN(HIST2_IN); col++)
        for (row=0; row<mxGetM(HIST2_IN); row++)
            hist2[row][col] = mxGetPr(HIST2_IN)[row+col*mxGetM(HIST2_IN)];

    hist1cnt = mxGetM(HIST1_IN);
    hist2cnt = mxGetM(HIST2_IN);
    while(i<hist1cnt && j<hist2cnt) /*modified here...*/
    {
        relation = fCompare(hist1[i], hist2[j]);
        if (relation == LT)
            i++;
        else if (relation == GT)
            j++;
        else if (relation == EQ) {
```
holdVal = hist1[i];
while (relation == EQ && i < hist1cnt) {
  hist1points++;
  i++;
  relation = fCompare(hist1[i], holdVal);
}

holdVal = hist2[j]; /* does this need to change? */
relation = EQ; /* reset state of the relation */
while (relation == EQ && j < hist2cnt) {
  hist2points++;
  j++;
  relation = fCompare(hist2[j], holdVal);
}

if (hist1points < hist2points)
  count = count + hist1points;
else if (hist1points >= hist2points)
  count = count + hist2points;
else
  mexErrMsgTxt("# of points in hist 1 is neither <, >, or = to the # of points in hist 2");

hist1points = 0;
hist2points = 0;
}
else
  mexErrMsgTxt("Unknown relationship between feature 1 and feature 2
");
out[0] = count;
return;
D PMK Matlab Source
```matlab
clear;
% Dimensionality of the feature vector
d = 10;

%images_root = '/cs/research/ic/projects/Dom/Vision/data/';

% Root directories for linux
exe_dir = '/cs/research/ic/projects/Dom/Vision/code/';
image_root = '/cs/research/ic/projects/Dom/Vision/data-tmp/';

% Root directories for windows
exe_dir = 'Y:\ic\projects\Dom\Vision\code\';
image_root = 'Y:\ic\projects\Dom\Vision\data-tmp\';

% Record the list of category directories
directories = dir(image_root);
size(directories);
n = ans(1);
directories = directories(3:n);
n = n - 2;

% Visit each directory. Record all files in the visited directory.
% Load those files into matlab variables. Record the variable names in a
% record.
categories = cell(1,n);

% Used for determining the feature value ranges
% Initializing to intmin and intmax guarantee that maxing and mining over
% images works correctly.
max_vect = cast(ones(1,d), 'int32') .* intmin;
min_vect = cast(ones(1,d), 'int32') .* intmax;
max_vect = ones(1,d) * realmin;
min_vect = ones(1,d) * realmax;

% Iterate through all of the directories.
imageCount = 0;
for i = 1:n,
    file_dir = strcat(image_root, directories(i).name, '/trimmed-data/');
    files = dir(fullfile(file_dir, '*.pca'));
    size(files);
    m = ans(1);

    % Create a nested cell to hold all the image feature vectors
categories{i} = cell(1,m);
    cd(file_dir);
    % Iterate through each file in the current directory
    for j = 1:m,
        % Load file contents into a matlab variable
        imageCount = imageCount + 1;
        categories{i}{1,j} = sortrows(round(load(fullfile(file_dir, files(j).name))));
        % Possibly change this to categories{i}{j,1}
        % Then place the multiresolution histogram in categories{i}{j,2}
        % This histogram rep will be a cell consisting of all the histogram
        % concatenated together?
        image_max = max(categories{i}{1,j});
        image_min = min(categories{i}{1,j});
        max_vect = max(image_max, max_vect);
        min_vect = min(image_min, min_vect);
    end
end

cd(exe_dir);
%
% All of the data is now in matlab data structures.
% TODO could I just save and load categories once at the start? That'd be
```
% *really* nice, and hopefully save some time.

%% N.B. I'm suspicious of the technique I used to get the maximal and
%% minimal values.

% Calculate the maximal range of feature vector values
max_val = max(max_vect);
min_val = min(min_vect);
D = max_val - min_val; % = maximal range

% Calculate the number of histograms
L = ceil(log2(D));

% Calculate the bin-size vector
tmp = 1:L;
B = 2.^tmp;

% 1. For each image1 in images
% 2.   For each level1 in levels
% 3.     Calculate and store histogram points
% N.B. This takes a surprisingly short amount of time.
size(categories);
n = ans(2);
for i = 1:n,
    size(categories{i});
m = ans(2);
    m %print m
    for j = 1:m,
        for k = 2:L
            categories{i}{k,j} = sortrows(fix(categories{i}{1,j}./B(k)));
        end
    end
end

% (do this triangularly?) Is the pyramid kernel symmetric? PMK(I1,I2)
% = PMK(I2,I1)?
% 1. For each image1 in images
% 2.   For each image2 in images
% 3.       For each histogram1 and histogram2 in image histograms
% 4.           Calculate intersection vector (size = # histogram levels)
size(categories);
n1 = ans(2);
x=0; y=0;
affinity = zeros(imageCount, imageCount);
weights = (1./B(1:L-1))';
for i = 1:n1, % Start first image selection
    size(categories{i});
n2 = ans(2);
    for j = 1:n2, % End first image selection
        x = x + 1;
        for k = 1:n1, % Start second image selection
            size(categories{k});
n3 = ans(2);
            for l = 1:n3, % End second image selection
                y = y + 1;
                intersect = zeros(1, L);
                for m = 2:L,
                    %Useful work goes here
                    intersect(m) = isect(categories{i}{m,j}, categories{k}{m,l});
                end
                for m = 1:L-1,
                    newpts(m) = intersect(m + 1) - intersect(m);
                end
            end
        end
    end
end
% if (min(newpts) < 0)
%   newpts
% end
intersect

affinity(x,y) = newpts * weights;
end
end
y = 0;

end

cd(exe_dir);
E Partitioning Matlab Source
% we are given a weighted graph G=(V,E)
% E=[1 2 3 ... k] node indeces
% V=kxk symmetric link weights

% E=[1 2 3 4 5 6]
% V=[0 1 0 0 3 0;
%   1 0 1 1 1 1;
%   0 1 0 1 0 8;
%   0 1 1 0 0 1;
%   3 1 0 0 0 2;
%   0 1 8 1 2 0]

load affinity2;
V=affinity;
E=[1:size(V,1)];

% form W and D
% W weights
n=size(E,2);
[D,W]=makeDW(E,V);

[Ncut,x1ind,x2ind]=n_cut(D,W,0)

% check stability critirion
% probably check Ncut if it is min

%---------------------
E1=[1:size(x1ind,1)]
E2=[1:size(x2ind,1)]

[W1,W2]=partitionW(x1ind,x2ind,W);

cont=1;
nmax=max(n,10); % max # of groups
nmax=10;
count=1;
keepindeces=[ {x1ind} {x2ind}]

% while ((count<nmax)&&(cont==1))
% count=count+1;
% clear x11 x12 x21 x22;
% [D1,W1]=makeDW(E1,W1);
% [Ncut1,x11,x12]=n_cut(D1,W1,0)
% [D2,W2]=makeDW(E2,W2);
% [Ncut2,x21,x22]=n_cut(D2,W2,0)
% keepindeces=[keepindeces; {x11 x12}]
% keepindeces=[keepindeces; {x21 x22}]
% %clear
% end
% we are given a weighted graph G=(V,E)
% E=[1 2 3 ... k] node indeces
% V=kxk symmetric link weights

% E=[1 2 3 4 5 6]
% V=[0 1 0 0 3 0; 1 0 1 1 1 1; 0 1 0 1 0 8; 0 1 1 0 0 1; 3 1 0 0 0 2; 0 1 8 1 2 0]

load affinity8;
V=affinity;
E=[1:size(V,1)];

% form W and D
% W weights
n=size(E,2);
W=V;
D=zeros(n);
for i=1:n
    temp=0;
    for j=1:n;
        temp=temp+W(i,j);
    end
    D(i,i)=temp;
end

% # of categories we want
k=8;
[indices,centers]=K_cut(D,W,k)
function \([W_1, W_2] = \text{partitionW}(x1ind, x2ind, W)\)

\[ \begin{align*}
ii &= 0; \\
\text{for } i &= x1ind'; \\
    jj &= 0; \\
    ii &= ii + 1; \\
\text{for } j &= x1ind(1:ii)'; \\
    jj &= jj + 1; \\
    W1(ii, jj) &= W(i, j); \\
    W1(jj, ii) &= W1(ii, jj); \\
\end{align*} \]

\[ \begin{align*}
\text{end}
\end{align*} \]

\[ \begin{align*}
ii &= 0; \\
\text{for } i &= x2ind'; \\
    jj &= 0; \\
    ii &= ii + 1; \\
\text{for } j &= x2ind(1:ii)'; \\
    jj &= jj + 1; \\
    W2(ii, jj) &= W(i, j); \\
    W2(jj, ii) &= W2(ii, jj); \\
\end{align*} \]

\[ \begin{align*}
\text{end}
\end{align*} \]
function [D,W]=makeDW(E,V)

n=size(E,2);
W=V;
D=zeros(n);
for i=1:n
    temp=0;
    for j=1:n;
        temp=temp+W(i,j);
    end
    D(i,i)=temp;
end
function \([Ncut,A,B]=\text{n\_cut}(D,W,\text{Thr});\)  
%\[P,Ncut,A,B\]
%P indeces of node categorization 1->A -1->B
%Ncut(A,B)
%A,B partition indeces

\(k=3;\)
[\(T,\text{lambda}\)]=eigs(D-\(W,D,k\),'sm');

%take the second eigenvector
x=\(T(:,2);\)

%choose threshold
\(\text{Thr}=0;\)

\(n=\text{size}(x,1);\)
\(P=\text{zeros}(n,1);\)
\(A=\text{find}(x\geq \text{Thr});\)
\(B=\text{find}(x<\text{Thr});\)
\(P(A)=1;\)
\(P(B)=-1;\)

%------------------------
cutAB=0;
assocA=0;
assocB=0;

for \(i=A\)
    for \(j=B\)
        cutAB=cutAB+W(i,j);
        if \((i=A(1))\) assocB=assocB+sum(W(j,:));
    end
end
assocA=assocA+sum(W(i,:));
end

%cutAB=cutBA cause W is symmetric

\(Ncut=\text{cutAB}*(1/\text{assocA} + 1/\text{assocB});\)
function [IND,C]=K_cut(D,W,k);
%C centers of clusters
%IND indeces
%k the number of cluster we want
if k==size(D,1)
    k=k-1;
end
[T,lambda]=eigs(D-W,D,k+1,'sm')
[IND,C]=kmeans(T(:,2:end),k);