Classifying Documents Using Nonnegative Matrix Factorization

Due: March 17, 2016

Brian de Silva[1]

Abstract
This report is a component of the final project for AMath 582, which shares the same title. Here we give an overview of the techniques employed to attempt to cluster/classify text documents using nonnegative matrix factorization (NMF). In particular, a few key modifications to the "naive" approach improved clustering performance significantly, but many other methods were tested before these were discovered. It is the purpose of this writeup to discuss the many failed and few successful techniques used throughout the course of this endeavor.

[1] Department of Applied Mathematics, University of Washington, Seattle

Contents

1. Introduction

For this project we manually collected a set of text documents from the Project Gutenberg website (`https://www.gutenberg.org`), given below.

- Historical texts
  - *History of the Seventh Ohio Volunteer Cavalry* by R. C. Rankin
  - *History of Company E of the Sixth Minnesota Regiment of Volunteer Infantry* by Alfred J. Hill
  - *Abraham Lincoln and the Union A Chronicle of the Embattled North* by Nathaniel W. Stephenson
- Scientific texts
  - *On The Origin of Species* by Charles Darwin
  - *The Life-Story of Insects* by Geo. H. Carpenter
  - *The Chemistry of Food and Nutrition* by A. W. Duncan

Every text was broken up into five equi-length text files with some books being much longer than others. The goal of the project was to take these text files as input and to attempt to cluster and/or classify them based on their content. In order to perform clustering/classification we must interpret the documents in a mathematical setting. To this end we use the **bag-of-words** approach to model the documents as word-count vectors with nonnegative entries. Each entry corresponds to the number of times a word from some dictionary appears in the document. We could attempt to cluster the documents in terms of their word-count vectors, but this turns out to be ineffective. Rather we will generate a more ideal representation for the documents using **Nonnegative Matrix Factorization** (NMF) [1]. Given a set of documents, or a *corpus*, we can construct word-count vectors for each then concatenate them into a *histogram matrix*, H, which has columns that are word-count vectors for the documents. We then compute the Nonnegative Matrix Factorization of the Histogram matrix and use its output for our analysis.

The **Nonnegative Matrix Factorization** of a matrix H is an approximate decomposition of H into two (often low-rank) matrices with nonnegative entries such that $H \approx UV^T$. The columns of U give the "topics" present in the documents of the corpus and the columns of V^T give the coordinates of each document in this topic basis. We call the column of V^T corresponding to a document that document's "topic vector". It is this representation of the documents we use to cluster/classify them. Since the NMF gives a low-rank approximation we might hope that it extracts some underlying structure from the histogram matrix, and by extension, the corpus. Once it has been computed we apply k-means clustering on the topic vectors, though more sophisticated clustering routines may be used.

1.1 Algorithm Overview

Here we present a high-level summary of the steps involved in our algorithm:

1. Construct the dictionary and histogram matrix H (using tf-idf)
2. Use NMF to approximately decompose H into topics and topic vectors
3. Apply Gram reweighting to topic vectors
4. Cluster the documents using their topic representations with the k-means algorithm

The remainder of this writeup follows the steps in the algorithm, discussing both ideas which failed to pan out and those which improved performance. Section 2 corresponds to construction of the dictionary and histogram matrix, Section 3 to computing the NMF of the histogram matrix, Section 4 to the Gram matrix reweighting (which has yet to be discussed), and Section 5 to clustering the documents. Finally we give some concluding remarks in Section 6.

2. The Histogram Matrix

There were two major advancements that occurred in this phase of the algorithm. The first has to do with the forming of the dictionary of words. There are a large number of words that appear extremely frequently in the English language. Words such as “and”, “the”, and “a” are present in most sentences across all the documents in the corpus. Therefore they are often the words with the highest word counts in each of the documents which leads to topics which are dominated by them. While it can be interesting to infer information about various authors' writing styles from the combinations of these words they use, these types of topics do not tell us much about the true themes present in the corpus, themes such as chemistry, Abraham Lincoln, and insects. In order to deal with this issue we simply removed the troublesome “stopwords” after constructing the dictionary, but before finding word counts. We found a list of stopwords from the following website `http://www.ranks.nl/stopwords` (the “Long Stopword List”). This turned out to be an effective means of dealing with the problem.

As was mentioned briefly in Section 1, many of the documents were much longer than others. The longer documents inherently contain more words, giving their word-count vectors larger entries. After applying NMF to the histogram matrix we found that the topics were heavily biased toward these long documents. In order to remedy this we reweighted the word-count vectors using a technique called term frequency-inverse document frequency (tf-idf). Essentially the tf-idf of a word in a given document increases the more times the word appears in the document, but decreases the more times the word appears in the entire corpus. This has two effects:

1. Words which appear often throughout the corpus are given diminished weight (this could be another way to deal with stopwords)
2. Words which appear many times in a longer document but not many times in shorter documents are given normalized weight

The second of these two effects helps alleviate the problems that come with longer text documents.

It should be noted that we tried exclusively using tf-idf to deal with both stopwords and long documents, but this was not found to be as effective as explicitly removing the stopwords and then using tf-idf.

3. Nonnegative Matrix Factorization

To compute the NMF of the histogram matrix H we used the MATLAB command `nnmf`. It took a great deal of tweaking the parameters of this function to obtain consistently satisfactory results. Perhaps the deepest problem encountered was that one must specify the number of topics that should be present in the factorization. If is already familiar with the documents in the corpus then this is an easy task, but if not then where should one start? This problem was essentially dealt with using a reweighting of the topic vectors, see Section 4.

The remaining obstacles involved getting the `nnmf` command to generate the same factorization across multiple runs of the algorithm. Computing the NMF of a matrix typically involves using an iterative procedure to approximate U and V. Iterative procedures, however, require initial guesses to begin. Without specifying a initial guesses for U and V `nnmf` generated random ones itself. This lead to an unacceptable amount of inconsistency between different executions of the code, using the same parameters. Sometimes the algorithm would produce great results, and sometimes it would fail entirely–all due to differences in the factorization produced by `nnmf`. We tried two approaches to move past this debacle. First we experimented with passing in initial guesses for U and V by just taking submatrices from the histogram matrix. This solved the problem of inconsistency between runs, but worsened the clustering performance of the algorithm. Depending on the corpus being considered we still obtained suboptimal results.

The `nnmf` command can use either an alternating least-sqaures algorithm or a multiplicative algorithm to compute U and V. The default setting is for it to use the alternating least-squares technique, since it produces more consistent output. The multiplicative method is much more sensitive to the initial guess. However, one can use the multiplicative method to generate many instances of U and V, each time using different randomly generated initial guesses, and choose those which are “best” in some sense ($\min_{U,V} \|H - UV^T\|_2$). We found this method to produce the best results if one runs enough trials (around 20-30 in our experience).

4. Gram Matrix Reweighting

One of the biggest obstacles we faced in trying to use NMF to cluster documents was that in order to compute the NMF of the histogram matrix one needs to specify the number of topics to be used (this is equivalent to specifying the rank of the approximation). Without some knowledge about the underlying dataset it is not always possible to say how many topics will do a good job of summarizing the content of the documents. Furthermore, despite having used the tf-idf, we often found that upon computing the NMF of a corpus a few documents with very distinctive diction were getting their own topics. If we used too few topics then not all the documents were well represented and if we used too many, then a few distinctive documents got their own topics, leaving others unrepresented still. Often these individualized topics were very similar to one another. The aforementioned problems frequently led to poor clustering performance.

In an attempt to solve the many-similar-topic problem, we decided to modify the topic vectors so that they reflected the likenesses of topics to one another. Our first effort was to construct an adjacency matrix which used various measures of the relationships between the topics to construct its entries. We then left-multiplied the matrix of topic vectors, V by this matrix and performed clustering. The entries of this matrix were of the form $\exp(-sim(i,j)^2/\sigma)$, where $sim(i,j)$ is some similarity measure between topics i and j and σ is a parameter. We experimented with similarity measures like the ℓ^2 inner product and the cosine similarity. This granted us limited success, but not as much as we had hoped.

Next we tried left-multiplication by a slightly different matrix. For our purposes the Gram matrix (or Grammian) is given by

$$G = \frac{1}{\|U\|^2} U^* U.$$

Each of its entries is simply the inner product between two topics (the columns of U). This matrix modified the topic vectors in the desired manner. Left-multiplication by G modifies the topic vectors so that the weight of their entries are distributed amongst topics with similar compositions. Supposed a topic vector has only one large component, i.e. its content is described by just one topic, say topic 1. Then multiplying it by G will have the effect of increasing its components corresponding to topics related to topic 1. This means that we do not necessarily need to know the optimal number of topics in advance. If we select too large a number of topics then the Gram matrix will automatically create groups of topic vectors with similar structures by "spreading" the influence of similar topics among documents which are thematically alike. These groups are easy to cluster.

5. Clustering

Initially we ran into the same trouble with k-means clustering that we did using NMF, namely we were not getting consistent results between runs with identical parameters. The underlying cause was the same in both instances–the algorithms are iterative and their output relies on the initial guesses provided. Rather than make the same mistake as before and try supplying intial guesses chosen in some consistent manner, we ran k-means multiple times with different initial guesses and used the clustering with the lowest residual (sum of squared distances to the centroids) as our final grouping. Given enough runs, we were able to get much more consistent output.

The MATLAB command `kmeans` allows one to specify how you would like it to compute distances. After applying all of the previously mentioned modifications to our algorithm we were still not getting performance which was as good as we had hoped. k-means was using the ℓ^2 (Euclidean) norm to measure the differences between pairs of topic vectors. We found the ℓ^1 norm to improve correct classification rates somewhat. The cosine similarity, however, gave dramatically better results. Its idea is simple: given two vectors, it returns the cosine of the angle between them. If the two topic vectors are collinear then the angle between them is 0, so their cosine similarity is 1 and if they are orthogonal then their cosine similarity is 0. Since topic vectors often contain nonzero components in only a few topics, vectors corresponding to documents on different subject matter are often orthogonal. Thus the cosine similarity does a stellar job of distinguishing between documents with different thematic content.

6. Summary and Conclusions

In attempting to classify data, it is often not enough to represent it in a more ideal basis. Often one must also employ a series of other techniques or choose parameters in a particular way in order to obtain decent results. This project was no exception. To get meaningful output we had to use some preprocessing methods to clean up our data before applying any data analysis techniques. To get said data science algorithms to consistently accomplish the tasks set before them we had to both modify the data further and tweak the parameters involved with the algorithms themselves. In the end we were able to accomplish our goal of clustering/classifying the text documents from our artificial corpus into their original books.

References

[1] Farial Shahnaz, Michael W Berry, V Paul Pauca, and Robert J Plemmons. Document clustering using nonnegative matrix factorization. *Information Processing & Management*, 42(2):373–386, 2006.

Appendix A: MATLAB Code

See the following pages published in MATLAB for the implementation of the algorithm presented in Section 1.1 along with supporting functions. The function `lexcmp` was downloaded from `http://www.mathworks.com/matlabcentral/fileexchange/23035-lexcmp`.

Table of Contents

Main script

```
% Queries the user for a directory to search then constructs a
 histogram
% and dictionary. Then applies nonnegative matrix factorization and
% attempts to cluster the documents into their original books.
% Compares different methods side-by-side (nmf, gram
% matrix, weighted adjacency matrix)

close all
```

Parameters

----------------------- PARAMETERS-------------------------- %

```
% Parameter (number of topics to be used in nonnegative matrix
% factorization)
num_topics = 10;

% Parameter (constant used to compute adjacency matrix for weight
 matrix)
sigm = .3;

% Parameter (number of search results to display)
num_display = 10;

% Parameter (indicates whether or not to use cosine similarity to
 constuct
% adjacency matrix for reweighting of V
use_cos_adj = false;

% Parameter (indicates whether or not to use tf-idf)
use_tfidf = true;

% Parameter (indicates which corpus to load)
% Choices are: {all, bio, biochem, chem, hist}
corpus = 'all';

% Parameter (indicates how many clusters are to be used by k-means)
clusters = 6;
```

```
% Parameter (indicates which similarity measure to use for clustering)
% Options:
%             sqeuclidean - Squared Euclidean distance
%             cityblock - L1 distance
%             cosine - 1 minus cosine of the angle betweent the points
 (treated as vectors)
%             correlation - 1 minus the correlation between points
dist_type = 'cosine';

% Parameter (kmeans will run num_trials times and return the
 clustering
% which gave the best residual)
num_trials = 10;

% Parameter (nnmf will run num_facts times and return the clustering
% which gave the best residual)
num_facts = 30;

% --------------------------------------------------------------%
```

Construct dictionary & histogram

```
if exist('corpus','var')
    load([corpus '_hist.mat'])
    doc_count = length(filenames);
end

if ~exist('histmat','var')
    dictionary = [];
    % queries the user for the folder containing the txt files
    folder = uigetdir;
    dirListing = dir(folder);
    doc_count = 0;

    % constructs the dictionary
    for i = 1:length(dirListing)
        filename = fullfile(folder,dirListing(i).name);

        if ~isempty(regexp(filename,'\.(txt)', 'once'))
            A = parser(filename);

            dictionary = char(dictionary,A);
            dictionary = unique(dictionary,'rows');
            doc_count = doc_count + 1;

        end

    end


    % Remove stop words
    load('stop_words.mat');
    dictionary = setdiff(dictionary, stop_words,'rows');
    celldict = cellstr(dictionary);
    clear A stop_words;

    % Initialize histogram matrix
    histmat = zeros(length(dictionary), length(dirListing));


    % Construct histogram matrix (histmat)
    filenames = cell(1,length(dirListing));
    for i = 1:length(dirListing)
        filename = fullfile(folder,dirListing(i).name);
        filenames(i) = cellstr(dirListing(i).name);

        if ~isempty(regexp(filename,'\.(txt)', 'once'))
            A = sort(parser_cell(filename));


            index = 1;
            % cycle through each word in the dictionary
            for j = 1:length(dictionary)
                count = 0;
                for k = index:length(A);
                    if cellfun(@strcmp,celldict(j),A(k))
                        count = count + 1;
                        index = k+1;
                    end

                    if lexcmp(celldict(j),A(k))==-1
                        break;
                    end

                end
                histmat(j,i) = count;
            end
        end
    end

    clear i j k index A count filename dirListing;

    % Remove hidden documents containing no words from histogram
    filenames( :, ~any(histmat, 1) ) = [];
    histmat( :, ~any(histmat, 1) ) = [];
end
```

Apply NMF and reweigthings

Apply nonnegative matrix factorization (and possibly tf-idf) to the histogram matrix

```
if use_tfidf
    modified_histmat = tf_idf(histmat);
    [U V] =
 nnmf(modified_histmat,num_topics,'algorithm','mult','replicates',num_facts);
else
```

```
    [U V] =
 nnmf(histmat,num_topics,'algorithm','mult','replicates',num_facts);
end


% Gram matrix modification
gram = U.' * U / (norm(U)^2);
V_gram = gram * V;
clear gram;


% Adjacency matrix modification
if use_cos_adj
    A = squareform(pdist((U ./ norm(U))','cosine'));
else
    A = squareform(pdist((U ./ norm(U))'));
end
W = exp( - (A .^2) ./ sigm);
V_adj = W * V;
clear W A;
```

Results

Display top words in each topic

```
display('Top ten words in each topic:');
display(celldict(top_words(U,10)));


% Display documents in terms of topics grayscale image
figure();

subplot(1,3,1);
imshow((1 - V'),'InitialMagnification','fit');
axis on;
colormap(gray);
title('Topic Vectors');
xlabel('Topics');
ylabel('Documents');

subplot(1,3,2);
imshow((1 - V_adj'),'InitialMagnification','fit');
axis on;
colormap(gray);
title('Adjacency Matrix Topic Vectors');
xlabel('Topics');
ylabel('Documents');

subplot(1,3,3);
imshow((1 - V_gram'),'InitialMagnification','fit');
axis on;
colormap(gray);
title('Gram Matrix Topic Vectors');
```

```
xlabel('Topics');
ylabel('Documents');


% Cluster/classify the documents using k-means

% Topic vectors
idx =
 kmeans(V.',clusters,'Distance',dist_type,'Replicates',num_trials);

% Adjacency matrix reweighting
idx_adj =
 kmeans(V_adj.',clusters,'Distance',dist_type,'Replicates',num_trials);

% Gram reweighting
idx_gram =
 kmeans(V_gram.',clusters,'Distance',dist_type,'Replicates',num_trials);


% Bar plots of clusterings
figure()
subplot(1,3,1)
bar(idx)
xlabel('Document')
ylabel('Cluster')
title('Topic Vectors')

subplot(1,3,2)
bar(idx_adj)
xlabel('Document')
ylabel('Cluster')
title('Adjacency Reweighting')

subplot(1,3,3)
bar(idx_gram)
xlabel('Document')
ylabel('Cluster')
title('Gram Reweighting')


% Print performance of clusterings
ground_truth = ones(size(idx));
for k=2:(doc_count/5)
    ground_truth((5*(k-1)+1):5*k) = k;
end


[px, ~] = purity(ground_truth, idx);
fprintf('Percentage of correct classifications (topic vectors): ');
fprintf([num2str(px) '\n\n'])

[px_adj, ~] = purity(ground_truth, idx_adj);
fprintf('Percentage of correct classifications (adjacency reweighted
 topic vectors): ');
```

```
fprintf([num2str(px_adj) '\n\n'])

[px_gram, ~] = purity(ground_truth, idx_gram);
fprintf('Percentage of correct classifications (gram reweighted topic
 vectors): ');
fprintf([num2str(px_gram) '\n\n'])


clear distances indx results i ind more_words need_valid_file num_topics prompt1 p
```

```
Top ten words in each topic:
  Columns 1 through 4

    'capt'          'fig'              'lincoln'         'rejoined'
    'col'           'cuticle'          'government'      'company'
    'rankin'        'insects'          'war'             'camp'
    'regiment'      'wings'            'chase'           'regiment'
    'cumberland'    'insect'           'congress'        'fort'
    'ky'            'winged'           '1862'            'enlisted'
    'river'         'dragonfly'        '1861'            'sick'
    'gen'           'lifestory'        'lincolns'        'sergeant'
    'command'       'wingrudiments'    'vallandigham'    'miles'
    'ohio'          'wingless'         'secretary'       'henricks'

  Columns 5 through 8

    'larva'           'kansas'         'brigade'      'food'
    'larval'          'political'      'battery'      'acid'
    'fig'             'party'          '7th'          'foods'
    'larvae'          'douglas'        'amunition'    'proteid'
    'cuticle'         'democrats'      'gen'          'uric'
    'imaginal'        'whigs'          'ga'           'meat'
    'maggot'          'slavery'        'enemy'        'quantity'
    'caterpillars'    'sectional'      'selma'        'flesh'
    'pupa'            'republicans'    'columbus'     'starch'
    'discs'           'politics'       'iowa'         'extract'

  Columns 9 through 10

    'col'           'lincoln'
    'garrards'      'douglas'
    'knoxville'     'buchanan'
    'moved'         'seward'
    'morristown'    'republicans'
    'brigade'       'political'
    'clinch'        'slavery'
    'gen'           'republican'
    'river'         'southern'
    'enemy'         'union'

Percentage of correct classifications (topic vectors): 0.93333

Percentage of correct classifications (adjacency reweighted topic
 vectors): 0.93333
```

```
function c = lexcmp(s1,s2)
% LEXCMP  C-style array/string comparison (by lexical order)
% usage: lexcmp(s1,s2)
%
% If S1 and S2 are strings, LEXCMP(S1,S2) returns:
%    -1,  S1 < S2
%     0,  S1 = S2
%     1,  S1 > S2
% where S1 and S2 are compared by the first element that differs.
%
% S1 and S2 can be numeric or logical arrays with possibly different
% lengths and possibly containing inf/NaN values (treating pairs of
 NaN
% values as equal and all other values less than NaN, as per the
 convention
% of the built-in version of SORT).
%
% S1 and S2 can theoretically be arrays of any objects for which SIGN,
% MINUS, ISINF and ISNAN are defined, e.g. fractions (see below).
%
% If S1 and S2 are cell arrays with the same number of elements, they
 are
% compared element-by-element.  If one is a cell array and the other
 is
% not, then each element of one is compared to the other.
%
% Example:
%   lexcmp('abc','abc') %  0
%   lexcmp('abc','abd') % -1
%   lexcmp('a','ab')    % -1
%   lexcmp(nan,nan)     %  0
%   lexcmp(1:2,1:3)     % -1
%   lexcmp(fr(1,3),fr(1,2)) % -1 (using Fractions Toolbox)
%
% Implementation note:
%
% The core function could be implemented more concisely if vectorized
% rather than looped, but tests indicated that loops seem to be faster
 on
% average (in the original version with no inf/nan checking).  The
 biggest
% performance gains occur when the strings are long and differ near
 the
% beginning, as the loop exits early and avoids unnecessary
 comparisons;
% similarly when the arrays are non-numeric objects with significant
% overhead per comparison.  To get any further performance gains, the
 best
% option would probably be to translate the core function to MEX.
%
% See also STRCMP (built-in), STRCMPC (FEX #3987), Fractions Toolbox
 (FEX #24878).
```

```
% Ben Petschel 19/2/2009
%
% Version history:
%   19/2/2009 - First release.  The core function is based on
 S.Helsen's
%     STRCMPC (http://www.mathworks.com/matlabcentral/
fileexchange/3987).
%     The find part has been replaced with a loop which is slightly
 faster
%     in the current version of MATLAB.
%   14/9/2009 - Updated lexcmp_core to allow infinite or NaN values
%     and added some examples to the help text.



% decide whether to compare directly or element-by-element
if iscell(s1)
  if iscell(s2)
    if numel(s1)==numel(s2),
      % compare element by element
      c=zeros(size(s1));
      for i=1:numel(s1),
        c(i)=lexcmp_core(s1{i},s2{i});
      end;
    else
      error('lexcmp:cellsize','cell array inputs must have same number
 of elements');
    end;
  elseif isvector(s2)
    % compare each element of s1 with s2
    c=zeros(size(s1));
    for i=1:numel(s1),
      c(i)=lexcmp_core(s1{i},s2);
    end;
  else
    error('lexcmp:inclass','input s2 must be a cell or an array');
  end;
elseif isvector(s1)
  if iscell(s2)
    % compare s1 with each element of s2
    c=zeros(size(s2));
    for i=1:numel(s2),
      c(i)=lexcmp_core(s1,s2{i});
    end;
  elseif isvector(s2)
    % compare s1 and s2 directly
    c=lexcmp_core(s1,s2);
  else
    error('lexcmp:inclass','input s2 must be a cell or an array');
  end;
else
  error('lexcmp:inclass','input s1 must be a cell or an array');
end
```

```
end % main function lexcmp(...)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%
function c=lexcmp_core(s1,s2)
% helper function for lexcmp

L=min(numel(s1),numel(s2));
c=0;
i=1;
% find the first place where the strings differ
while (c==0) && (i<=L)
  c=sign(s1(i)-s2(i));
  if isnan(c)
    % have (inf,inf) or (-inf,-inf) or (x,nan) or (nan,x)
    if isinf(s1(i)) && isinf(s2(i))
      % case (inf,inf) or (-inf,-inf)
      c=0;
    else
      % use convention x<nan unless x is nan
      % (nan,nan) -> 0,  (x,nan) -> -1,  (nan,x) -> 1
      c=isnan(s1(i))-isnan(s2(i));
    end;
  end;
  i=i+1;
end;
if c==0
  % strings agree so far -> compare on lengths
  c=sign(numel(s1)-numel(s2));
end;

end % helper function lexcmp_core
```

```
Not enough input arguments.

Error in lexcmp (line 57)
if iscell(s1)
```

Published with MATLAB® R2015b

```
function result = parser_cell(filename)
%     Converts .txt file stored at filename to an array of alpha-
numeric characters
%     delimited by spaces.
%     filename shoule be a string specifying path of .txt file to be
%     parsed.
%     returns a cell rather than a char array

    A = textread(filename, '%s', 'delimiter', ' ');

%     converts all words to lower case
    A = lower(A);

%     function that locates non alpha-numeric characters
    f = @(str) str(isstrprop(str,'alphanum'));

%     Remove punctutation
    result = cellfun(f,A,'uniformOutput',false);

end
```

```
Not enough input arguments.

Error in parser_cell (line 8)
    A = textread(filename, '%s', 'delimiter', ' ');
```

Published with MATLAB® R2015b

```
function [ out, SECout ] = purity(GT, I)
% purity
%
%   GT is Ground Truth
%   I is the assignment in question.
%

[a,~] = size(GT);
[e,~] = size(I);
if(a ~= e)
    disp('Unequal input sizes');
    return;
end

T = unique(I);
S = unique(GT);

[g,~] = size(T);
SECout = zeros(g,2);
[a,~] = size(S); % REASSIGNMENT OF a
%T = T';
running = 0;

for i = 1:g
    count = zeros(a,1);
    Mask = (I == T(i,1));
    indices = find(Mask);
    [c,~] = size(indices);
    for j = 1:c
        Mask2 = (S == GT(indices(j,1),1)); %Sums for each gang j the
 number of cluster i members actually from that gang.
        count = count + double(Mask2);
    end

    [num, outI] = max(count);
    running = running + num(1,1);

    %Secondary Output
    SECout(i,1:2) = [(num(1,1)/c) outI(1,1)];

end

out = running/e;

end
```

```
Not enough input arguments.

Error in purity (line 8)
[a,~] = size(GT);
```

Published with MATLAB® R2015b

```
% tf_idf: takes in a matrix histmat, which has word histograms of
 documents
% as its columns, i.e. the i,jth entry of histmat corresponds to the
 number of
% times the ith word in your dictionary appears in the jth document of
 your
% corpus.
% Returns a modified histogram matrix with the term frequency-inverse
 document
% frequency of each word in each document replacing the raw frequency.
function modified_histmat = tf_idf(histmat)

%     Initialize modified_histmat with augmented term frequencies
    max_freq = max(histmat);
    modified_histmat = bsxfun(@rdivide,histmat,max_freq);

%     Alternative version of term frequency
    % modified_histmat = bsxfun(@rdivide,histmat .* (.5),max_freq) +
 (.5);

%     Compute inverse document frequency (idf)
    num_occurrences = sum(logical(histmat),2);
    idf = log(size(histmat,2) ./ (num_occurrences));

%     Compute tf-idf
    modified_histmat = bsxfun(@times,modified_histmat,idf);
end
```

```
Not enough input arguments.

Error in tf_idf (line 10)
    max_freq = max(histmat);
```

Published with MATLAB® R2015b

```
function result = top_words(U, n)
%     returns a matrix, result, with the top n words of each topic in
%     descending order
%     Each column corresponds to a topic
%     U is the basis matrix returned by NMF

[~, IX] = sort(U,'descend');

result = IX(1:n,:);

end
```

```
Not enough input arguments.

Error in top_words (line 7)
[~, IX] = sort(U,'descend');
```

Published with MATLAB® R2015b