Homework 3: Two applications of SVD Decomposition

Polytechnic University of Turin

Computational linear algebra for large scale problems - 01TWYSM

Caffaro Fabio
*s*276842

1 Independent Component Analysis

Imagine being at a party. Five people are talking at the same time mixing their voices. We have five microphones and we want to separate the mixed signal, retrieving from each microphone a single source completely separated from the others. This is the so called Cocktail Party Problem (fig. 1).

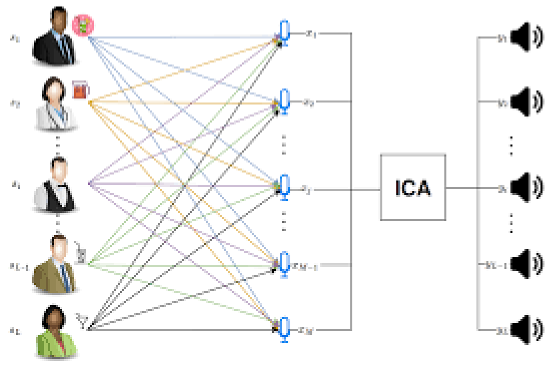


Figure 1: Representation of the Cocktail Party Problem

This task can be done with good results by humans. We have a natural ability to focus on a single source suppressing the noise around. Surprisingly it can also be performed algorithmically, with quite good results. The methodology used to perform this task is called Independent Component Analysis (ICA).

ICA is somewhat related to Principal Component Analysis (PCA).On one hand PCA tries to find 'principal components' that are uncorrelated, maximizing the variance and working with the variance-covariance matrix. On the other hand ICA as the name suggest, tries to find 'independent components' maximizing statistics of higher order, like at example Kurtosis.

ICA algorithm is based on two fundamental assumptions:

1. The source signals must not follow a Gaussian distribution
2. The source signals must be independent one from each other

Note that here we are talking of the original sources. The mixed signals obviously are dependent from each source and so one from each other and increasing the number of sources the mixed signals tend to follow a Gaussian distribution due to the Central Limit Theorem.

Considering a simple example with two different audio signals and two microphones. We can think at each signal as a vector $\mathbf{s} = (s_1, , \ldots, s_N)$ where N is the number of time steps and each component corresponds to the amplitude of the signal in the corresponding timeslot. We can represent the audio data in a matrix form

$$S = \begin{pmatrix} \mathbf{s}_1 \\ \mathbf{s}_2 \end{pmatrix}, \qquad S \in \mathbb{R}^{p,N} \tag{1}$$

Where p is the number of sources and N the number of time steps. We 'observe' the signals with our microphones. Each observation is a linear combination of all the source signals $\mathbf{x} = a\mathbf{s}_1 + b\mathbf{s}_2$ where the coefficients a and b depends on the microphone. The mixture matrix can be written as

$$X = \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix} = \begin{pmatrix} a * \mathbf{s}_1 + b * \mathbf{s}_2 \\ c * \mathbf{s}_1 + d * \mathbf{s}_2 \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} \mathbf{s}_1 \\ \mathbf{s}_2 \end{pmatrix} = AS \tag{2}$$

where the matrix $A \in \mathbb{R}^{n,p}$ is the matrix containing all the mixing coefficients. n is the numbers of observations (in the example the number of microphones) and p the number of sources. Note that n must be greater or equal to p, in order to reconstruct correctly the sources. Usually we have $n = p$ as in the example. The matrix $X \in \mathbb{R}^{n,N}$ contains the mixed signals.

Starting from this matrix, the objective is to find S. However, we don't know neither S nor A. The aim is to find a matrix $W \in \mathbb{R}^{n,p}$. In the example this matrix has the form:

$$W = \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix} \tag{3}$$

where $\alpha, \beta, \gamma, \delta$ are the *unmixing coefficients* such that $Y = W^T X \approx S$

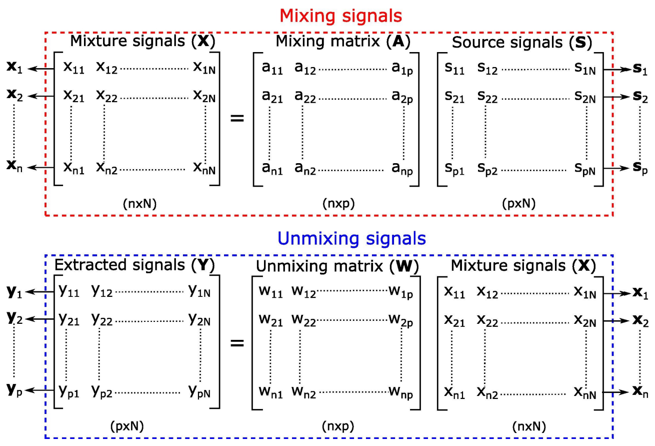


Figure 2: Rapresentation of the problem [2]

1.1 FastICA

There a lot of implementations of ICA technique. In this homework I have tried to implement a simple version of the FastICA algorithm[1].

This algorithm is composed by two main step: a preprocessing step, also called *prewithening* and the component extraction step.

In the prewithening step, firstly we center the matrix X:

$$X \quad \leftarrow \quad = X - \mu = \begin{pmatrix} \mathbf{x}_1 - \mu_1 \\ \mathbf{x}_2 - \mu_2 \\ \cdots \\ \mathbf{x}_n - \mu_n \end{pmatrix} \tag{4}$$

After the centering operation, each row of the matrix X has expected value of 0 The next step is performing the whitening, this means that we want to decorrelate each signal from all the others. A common method for whitening is by performing an eigenvalue decomposition on the covariance matrix of the centered matrix X.

$$XX^T = EDE^T \tag{5}$$

where E is the matrix containing the eigenvectors of the covariance matrix, while D is the diagonal matrix containing all the eigenvalues.

Note that these two first steps are performing exactly a PCA decomposition as we have seen during the course. The only difference here is in the centering step. In fact, for each component we are computing

$$x_{ij} \quad \leftarrow \quad = x_{ij} - \frac{1}{M}\sum_k x_{ik} \tag{6}$$

This means that we are working by row, since each row is a separated signal. Moreover, the whitening step can be performed more efficiently using a reduced SVD decomposition.

The last step is scaling the matrix. Here we want to scale each decorrelated signal to be with variance equal to 1.

$$X \quad \leftarrow \quad D^{-1/2}E^TX \tag{7}$$

Note that this step make the matrix rotationally symmetric, like a sphere. For this reason this procedure is sometimes also called *sphering*.

The second part of the algorithm it's not strictly related to the arguments of the course, so I skip the details. The main idea here is that we want to find directions for the weight (unmixing) vectors w that maximizes a measure of non-Gaussianity of the vector $w^T X$. For example the Kurtosis statistic. We want also that these components are mutually independent. To implement this part of the code, I have take advantage of the pseudocode from FastICA page of Wikipedia [1]

1.2 Matlab implementation and results

I have implemented the FastICA algorithm using matlab, and then I've tested it with a toy example. For testing the algorithm I have used three audio files provided with Matlab installation to construct the source matrix S:

```matlab
files = {'gong.mat' 'laughter.mat' 'train.mat' };

S = zeros(n, N);
for i = 1:n
    test     = load(files{i});
    y        = test.y(1:N,1);
    S(i,:)   = y;
end
```

Then I have generated a random matrix with the mixing coefficients and computed the mixed signals

```matlab
rng(42)
A = rand(n,n)*10;
X = A*S
```

For the prewithening step I have performed an svd decomposition of the mixed matrix X. In particular I have used the 'econ' paramater in order to have a reduced svd decomposition. Then I have retrieved the eigenvectors and eigenvalues of the covariance matrix and computed the scaled matrix $Z = D^{-1/2}E'X$:

```matlab
Xmean = mean(X,2);                          % Computing mean by row
for i = 1:n
    X(i,:) = X(i,:) - Xmean(i);             % Centering X
end

[U,Sig,V] = svd(X, 'econ')
D = Sig^2
Z = D^(-1/2) * E'*X                        % Prewhitened matrix
```

After this, I have performed the compuation of the components of the weight matrix W following the pseudocode from Wikipedia page:

```
W = zeros(p,n);

for p = 1:p

wp = rand(n,1)
wp = wp / sqrt(wp'*wp);

    for i = 1:iterations
        G       = tanh(wp'*Z);
        Gder    = 1-tanh(wp'*Z).^2;
        wp      = 1/M*Z*G' - 1/M*Gder*ones(M,1)*wp;
        dumsum  = zeros(n,1);

        for j = 1:p-1
            dumsum = dumsum + wp'*W(:,j)*W(:,j);
        end
        wp      = wp - dumsum;
        wp      = wp / sqrt(wp'*wp);
    end

    W(:,p) = wp;
end
```

Finally I have restored an approximation of the original signals $Y = \tilde{S} = W^T Z$. These are the results

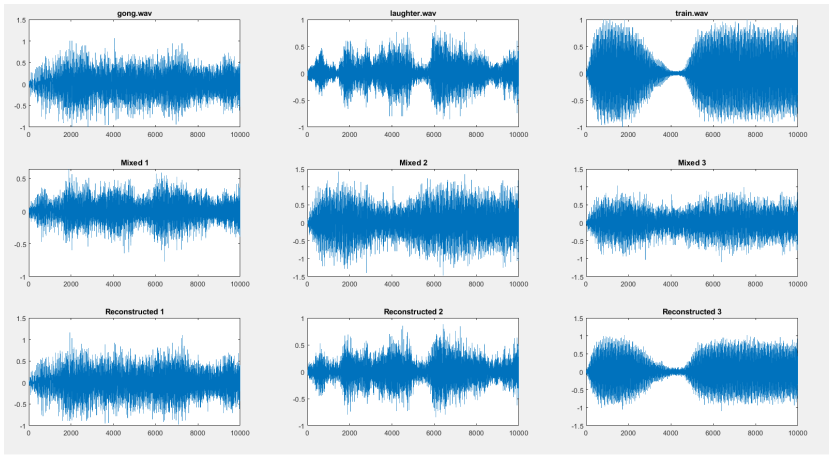


Figure 3: Results of the experiment. The sounds used are 'gong', 'laughter' and 'train'

In the first row I have plotted the three original signals. In the second row there are the three observations containing the mixed signals of the matrix X. Finally on the third row, the reconstructed signals. As the picture show, all the three signals are recovered quite well after being mixed. I have also implemented the code to save the three versions of each signal (original, mixed and recovered) as audio files in '.wav' format, in order to hear the results.

2 Latent Semantic Analysis

The second application of the SVD decomposition that I wanted to show, is Latent Semantic Analysis (LSA). This technique is one of the most important in the Natural Language Processing techniques. Given a set of documents and the set of terms that appear in these documents, the idea is to find *concepts* able to explain the relation between the documents and the terms.

LSA is an unsupervised techniques, this means that it does not use labels, but is based only on the data itself. It is based on two main pillars: a distribution hypothesis, that states that terms with similar meaning appear frequently together and the mathematical tool of svd decomposition.

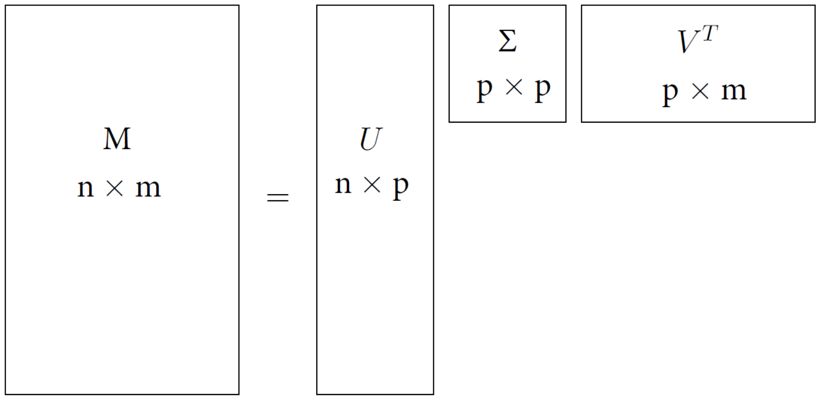
The whole procedure can be summarized in two main steps. In the first step we want to represent each document as a vector in a multidimensional space. Starting from a dataset containing n documents, we can organize it in a *document-term* matrix $M \in \mathbb{R}^{n,m}$. This matrix contains as rows the documents present in the dataset and as columns the m distinct terms present in the whole collection of documents. As one can imagine, for a fair number of documents this matrix is usually very sparse.

There are a lot of different types of document-term matrices that differ on the method that we use to weight the words in a document. The most simple one is the word-count matrix. In this matrix the element $m_{i,j}$ of the matrix is the number of occurrences of the word j in the document i.

With a matrix in a document-term form we can say, that initially we are considering each word representing a separated concept. Obviously such a consideration is not a smart idea. Moreover analyzing this very big matrix is quite expensive. The aim of the second step is to apply a dimensionality reduction considering $p << m$ new concepts that are a combination of the words present in the dataset and creating new representation of the documents (and also of the words) in terms of these concepts.

Note that in typical applications, the dimensionality reduction is enormous. Consider for example a dataset containing 200 articles. These articles may contain something like 10000 different words. Maybe applying some preprocessing techniques, as for example removing the stopwords and the very infrequent words, we can reduce them to only 4000-5000 words. Now that's quite an improvement, but the matrix is still very large. If, for example we assume that these articles treats 5 different topics, like 'Economics', 'Sport', 'Technology', 'Politics' and 'Gossip'. We can produce a LSA considering only 5 concepts and the representation of each article with respect to this concepts. The dimensions of the problem reduces from the original 200×2000 to only 200×5.

What we do in practice in this second step is to use a truncated svd decomposition, where we consider only the first p components. The components that we find with the decomposition are the concepts that we were looking for.



We can define the matrix $D = MM^T \in \mathbb{R}^{n \times n}$ as the *document-document* matrix, whose elements $d_{i,j}$ are the number of words in common between the document i and the document j. Similarly, the matrix $T = M^T M \in \mathbb{R}^{m \times m}$ is the *term-term* matrix. whose elements $t_{i,j}$ are the number of documents in which the term i and the term j occur together. The matrix $\Sigma \in \mathbb{R}^{p \times p}$ contains the first p singular values of the matrix M. These are the 'weights' that each concept has in the dataset. The matrix $U \in \mathbb{R}^{n \times p}$ is the *document-concept* matrix. It is the matrix of the eigenvectors of D. Finally the matrix $V \in \mathbb{R}^{m \times p}$ is the *term-concept* matrix. It contains the eigenvectors of T.

With the help of these three matrices we can now express the documents and the terms with regard to the new concepts. In particular the new document representation can be retrieved as $U * \Sigma$. This matrix has p columns representing the concepts and n rows representing the documents. This matrix tells us how much each document is strong in each concept (or topic). The new term representation instead, can be computed as $V * \Sigma$. This matrix has dimensions $m \times p$. This matrix tells us how much each word is contributing to each concept.

Intuitively, we are introducing an hidden layout in the dataset. We don't compare directly documents and terms, but we compare documents with the concepts and concepts with the terms. These concept in fact are not explicit but are hidden in the data, hence are called *latent*.

2.1 Matlab implementation and results

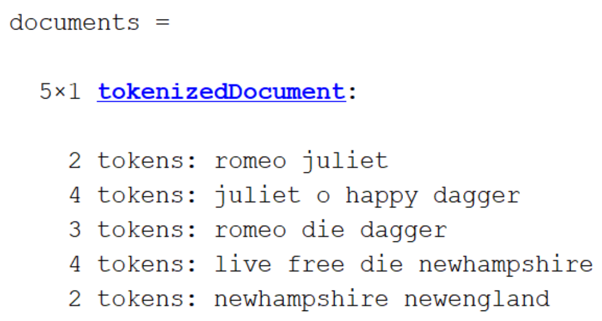
Also in this case, I have used Matlab to implement the algorithm. To show the functioning of the algorithm I have used a toy example from [3]. The dataset used in this example is a text document containing 5 rows. Each row represent a very simple document:

- *d1: Romeo and Juliet*
- *d2: Juliet: O happy dagger!*
- *d3: Romeo died by dagger*
- *d4: "Live free or die", that's the New-Hampshire motto*
- *d5: Did you know, New-Hampshire is in New-England*

In the first part of the code, after loading the document and splitting it in rows, I have tokenized each row to obtain a list of separated terms. Then, using some functions already present in the 'Text Anlytics Toolbox' of Matlab, I have applied a very simple preprocessing step in order to clean the data:

```
documents = lower(tokenizedDocument(textData))
documents = removeStopWords(documents)
documents = normalizeWords(documents, 'Style', 'lemma')
documents = erasePunctuation(documents)

documents = removeShortWords(documents, 2)
documents = removeLongWords(documents, 10)
```

Two important steps here are the removal of the stopwords (like articles, and conjunctions) and the punctuation that add only noise to the data, and the lemmatization step. In this latter step we are reducing each term to it's radix. Terms like *die*, *dying*, *died* for example, express all the same concept, so we don't want to make a distinction among them. Thus, we reduce them all to their original form. Note that a second but also important reason to do this step, is that we don't want to end up with a too much sparse matrix, otherwise it become more and more difficult retrieve connections between the documentsand the terms.

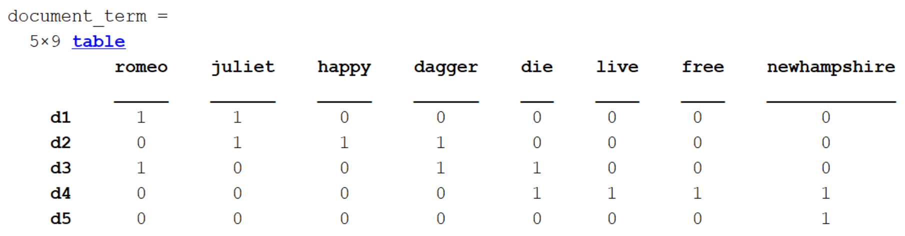


```
documents =

  5×1 tokenizedDocument:

    2 tokens: romeo juliet
    4 tokens: juliet o happy dagger
    3 tokens: romeo die dagger
    4 tokens: live free die newhampshire
    2 tokens: newhampshire newengland
```

Figure 4: Documents after preprocessing step

After this I have constructed the document-term matrix M using the count-word method, and applied the svd decomposition truncating the results to $N\_COMPONENTS = 2$:

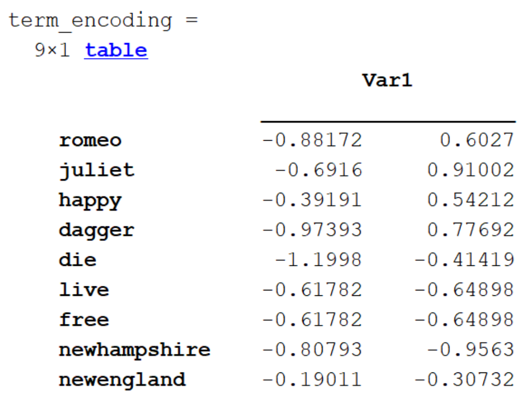
```
M = encode(bag, documents)
[U, S, V] = svd(full(M))

Uk = U(:, 1:N_COMPONENTS)
Sk = S(1:N_COMPONENTS, 1:N_COMPONENTS)
Vk = V(:, 1:N_COMPONENTS)
```



```
document_term =
  5×9 table
          romeo    juliet    happy    dagger    die    live    free    newhampshire
          _____    ______    _____    ______    ___    ____    ____    ____________

    d1      1        1         0        0        0      0       0           0
    d2      0        1         1        1        0      0       0           0
    d3      1        0         0        1        1      0       0           0
    d4      0        0         0        0        1      1       1           1
    d5      0        0         0        0        0      0       0           1
```

Figure 5: Document-term matrix M

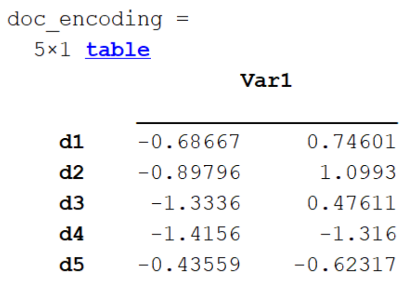
Then I have obtained the new representation of the terms:



```
term_encoding =
  9×1 table
                              Var1
                    ______________________

    romeo           -0.88172        0.6027
    juliet           -0.6916       0.91002
    happy           -0.39191       0.54212
    dagger          -0.97393       0.77692
    die              -1.1998      -0.41419
    live            -0.61782      -0.64898
    free            -0.61782      -0.64898
    newhampshire    -0.80793       -0.9563
    newengland      -0.19011      -0.30732
```

Figure 6: Representation of the terms w.r.t. the concept

and the new representation of the documents



```
doc_encoding =
  5×1 table
                     Var1
          ______________________

    d1    -0.68667       0.74601
    d2    -0.89796        1.0993
    d3     -1.3336       0.47611
    d4     -1.4156        -1.316
    d5    -0.43559      -0.62317
```

Figure 7: Representation of the documents w.r.t. the concepts

We can see how all the documents are negatively strong in the first component, while the second component make a distinction between the first three documents and the last two. To see better the results I have used a K-Means algorithm using the cosine similarity measure between the documents $\frac{d_i * d_j}{|d_i||d_j|}$ to cluster them (fig. 8). Searching for $K = 2$, we ends up with: $C1 : d_1, d_2, d_3$ and $C2 : d_4, d_5$. Looking at the documents we can have intuitively a sense of that.

The first three documents are related from the joint presence of the words "Romeo", "Juliet" and "dagger", while the fourth and the fifth documents by the term "New-Hampshire". Looking at the plot we can also see how document 3 and document 4 are not so distant, due to the fact that they share the term "die". Also document 1 and document 5 are in the same portion of the graph, hence are somehow related. The relation here is a bit-trickier to spot, since they don't have any words in common. However, considering the chain "Romeo" - "die" - "New-Hampshire" (d1, d3, d4, d5) we can explain it. Now, this is a quite forced link that shows up in this example only because we are treating a small problem, so only few relations are present in the dataset. However, LSA is so powerful exactly because of this capacity to find hidden links between terms in the documents.

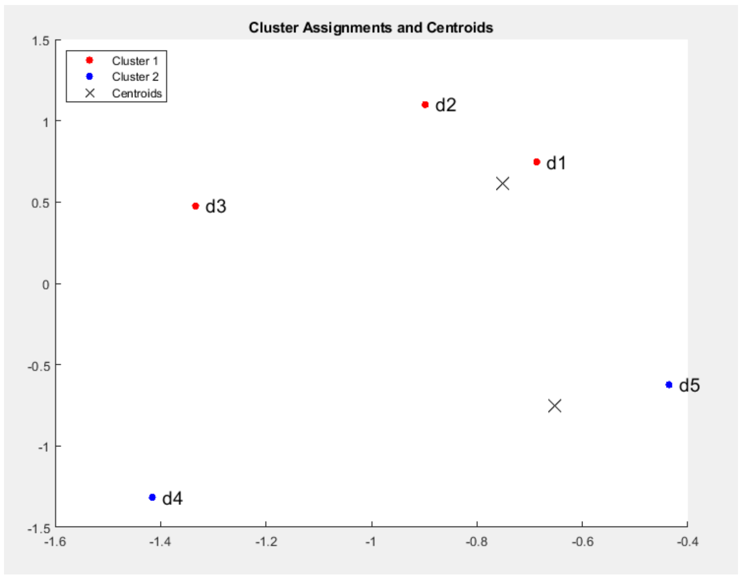


Figure 8: Document cluster obtained using K-Mean algorithm with K=2 and cosine similarity

Another application of LSA that exploit exactly this capacity, is ranking documents based on a query. Imagine to have a database of documents. After constructing the LSA vectorial space, we can query it to find the most relevant documents based on the query. Imagine for example that we want to sort our documents based on the query *q: die dagger*. First we need to tokenize the query, and preprocess it as we have done with the documents. In this case the result is simply the set $q : \{'die','dagger'\}$.

Then, we must transform the query to a vector in our new space. A simple way to do this is to consider the new representation of the words (fig. 6), and do a weighted sum of the words in the query. We have now a point representing the query in our new space.

$$q = \frac{'die' +' dagger'}{2} \tag{8}$$

where $'die' = (-1.1998, -0.41419)$ and $'dagger' = (-0.97393, -0.77682)$.

Finally, we need a similarity measure to compare the query with all the documents, and find the most similar ones. For this simple example I have used the euclidean distance, and considered the reciprocal to have a measure of similarity. I have also normalized the scores. Sorting the documents by relevance, the results are:

```
textQuery =
    "die dagger"
0.77    ->    (1) d3: Romeo died by dagger
0.43    ->    (2) d1: Romeo and Juliet
0.32    ->    (3) d2: Juliet: O happy dagger!
0.29    ->    (4) d5: Did you know, New-Hampshire is in New-England
0.19    ->    (5) d4: 'Live free or die', that's the New-Hampshire motto
```

Figure 9: Documents sorted based on the query q : *die dagger*

Obviously the more pertinent result is document 3, which contains both the words 'die' and 'dagger'. More interesting is the fact, that at the second place shows up document 1 which does not contain either of the two words. The system in fact, is 'smart' enough to learn that there is strict correlation between "Romeo" and "Juliet" and "die dagger". All of these words appear in similar context and so they are treated similarly.

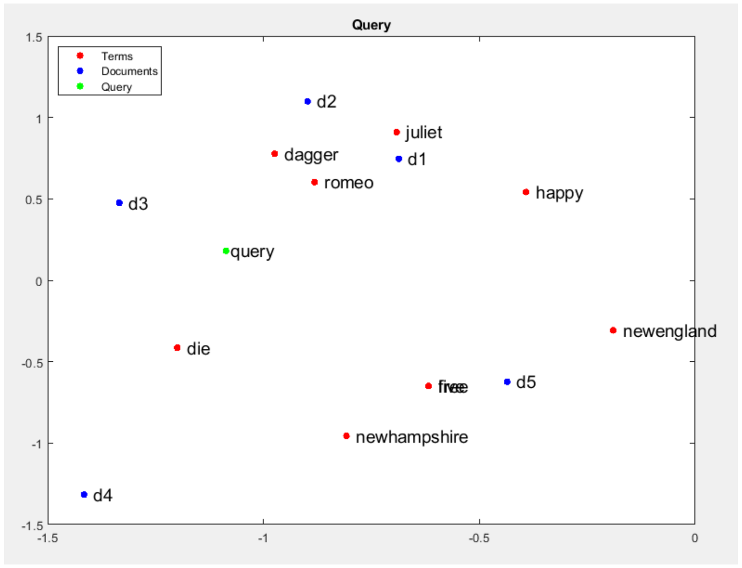


Figure 10: Plot of terms, documents and query in our 'concept-subspace'

References

[1] https://en.wikipedia.org/wiki/FastICA

[2] Independent component analysis: An introduction
https://www.sciencedirect.com/science/article/pii/S2210832718301819

[3] Latent Semantic Analysis (Tutorial) - Alex Thomo
https://www.engr.uvic.ca/ seng474/svd.pdf