Blind Source Separation

May 5, 2018

0.1 EECS491 - A5

0.1.1 Blind Source Separation

Tristan Maidment (tdm47)

Goal The goal for this assignment is to implement the BSS algorithm by hand, and apply it on a mixed distribution of two independent waveforms.

Implementation The first step is to import the wave files used in the demo.

```
In [1]: from scipy.io import wavfile

        srate, dataBach = wavfile.read('data/bach.wav')
        _, dataSpeech = wavfile.read('data/speech.wav')
```

```
In [2]: import numpy as np


        np.random.seed(519)

        def audionorm(data):
            # ensure data is ndarray with float numbers
            data = np.asarray(data).astype('float')
            # calculate lower and upper bound
            lbound, ubound = np.min(data), np.max(data)
            if lbound == ubound:
                offset = lbound
                scalar = 1
                data = np.zeros(size=data.shape)
            else:
                offset = (lbound + ubound) / 2
                scalar = 1 / (ubound - lbound)
                data = (data - offset) * scalar
            # return normalized data
            return data
```

Using these two data files and the normalization function, we can greate the ground truth waveforms.

```
In [3]: gtruthS = audionorm((dataBach, dataSpeech))
```

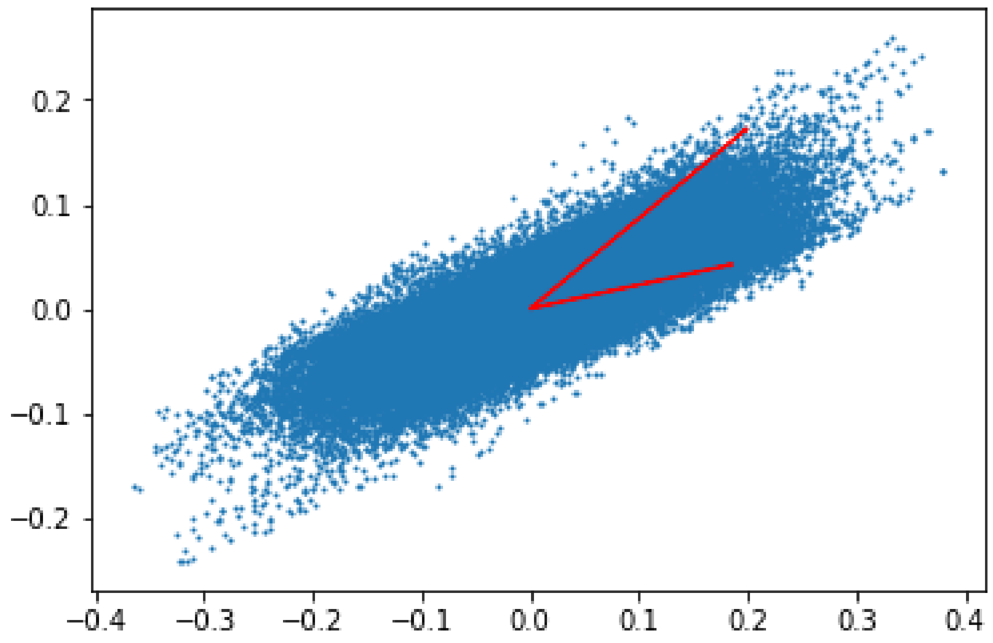
The next step is to mix the independent distributions into a random multivariate distribution. This can be done by specifying a mixing matrix that linearly modifies the shape of each distribution.

```
In [4]: def simpleMixer(S):
            nchannel = S.shape[0]
            # generate a random matrix
            A = np.random.uniform(size = (nchannel,nchannel))
            # generate mixed audio data
            X = A.dot(S)

            return X, A
```

```
In [5]: %matplotlib inline

        import matplotlib.pyplot as plt

        def drawDataWithMixingMatrix(data, mat):
            # plot data points
            plt.scatter(data[0], data[1], s=1)
            # calculate axis length
            lenAxis = np.sqrt(np.sum(np.square(mat), axis=0))
            # calculate scale for illustration
            scale = np.min(np.max(np.abs(data), axis=1) / lenAxis.T)
            # draw axis as arrow
            plt.arrow(0, 0, scale * mat[0,0], scale * mat[1,0], shape='full', color='r')
            plt.arrow(0, 0, scale * mat[0,1], scale * mat[1,1], shape='full', color='r')
```

Using the defined functions, we can define a mixed distribution which simulates input from two different microphones. In addition, we can plot the true mixing matrix values on top to visualize the true disributions.

```
In [6]: X, gtruthA = simpleMixer(gtruthS)
        drawDataWithMixingMatrix(X, gtruthA)
```



```
In [7]: wavfile.write('data/mixedTrackA.wav', srate, X[0])
        wavfile.write('data/mixedTrackB.wav', srate, X[1])
```

The **BSS** function works by defining a prior distribution, and fitting the A matrix so that this probablility of this data given this distribution is maximal. For that reason, the a gradient descent is performed on matrix A, to find the mixing matrix that most optimally fits the data to the distribution.

A decently large step size is viable for simple distributions.

```
In [8]: def bss(data, stop=1e-8, max_iter=1000, q=1, nu=0.5):
            sources, length = data.shape
            I = np.identity(sources)
            likelihood = 1

            A = np.identity(sources)

            for i in range(max_iter):
                if stop > likelihood:
                    break
                s = np.linalg.inv(A) @ data
                A_grad = -A @ (d_logP(s, q) @ s.T + I)
                A += np.multiply(nu, A_grad)
                likelihood = np.mean(Pxa(s, A, q))
            return A/A.max(), np.linalg.inv(A) @ data

        def P(s, q):
```

```
        return np.exp(-np.power(np.abs(s),q))

    def d_logP(s, q):
        return -np.divide(q * np.power(np.abs(s), q), s)

    def Pxa(s, A, q):
        return P(s, q)/np.linalg.det(A)
```

Given synthentic laplacian data, the mixing matrix converges fairly quickly. In addition, it very closely represents the original mixing matrix.

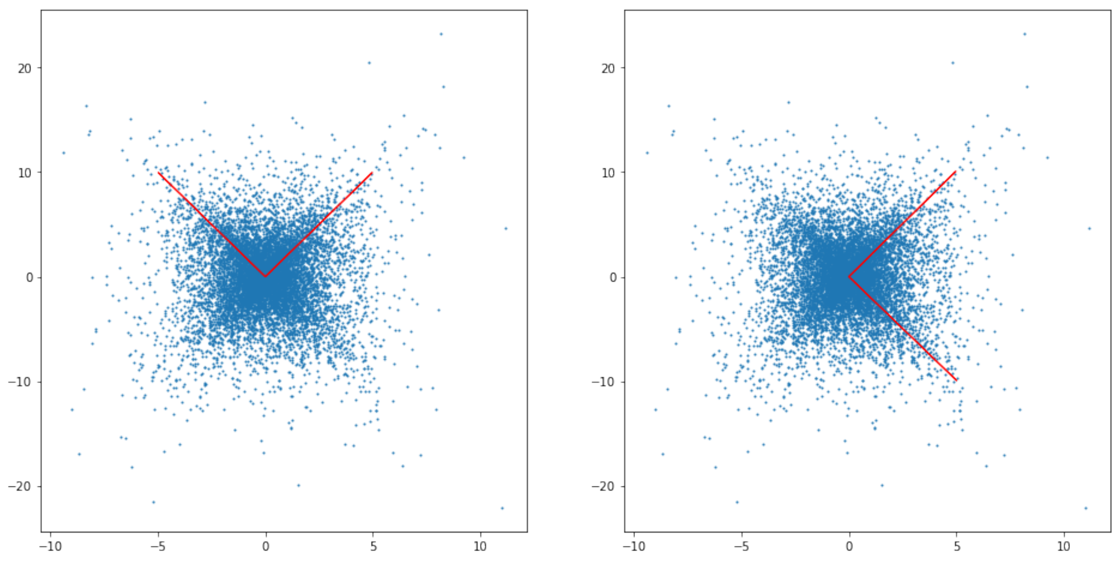
```
In [9]: def syntheticDataGenerate(A, nsamples):
            S = np.random.laplace(size = (2,nsamples))
            # generate mixed audio data
            X = A @ S

            return X
```

```
In [10]: # quantity of data points
         nsamples = 10000
         # specific mixing matrix (for illustration purpose)
         verifyA = np.asarray([[-1, 1],[2, 2]])
         # generate synthetic data
         synthData = syntheticDataGenerate(verifyA, nsamples)
         # do optimization with bss function
         estimateA, recoverData = bss(synthData, stop=3e-11)
```

```
In [11]: def compareMixingMatrix(data, matA, matB):
             plt.figure(figsize=(16, 8))
             # plot first mixing matrix
             plt.subplot(1,2,1)
             drawDataWithMixingMatrix(data, matA)
             # plot first mixing matrix
             plt.subplot(1,2,2)
             drawDataWithMixingMatrix(data, matB)
```

As we can see, the estimated mixing matrix very closely approximates the data. Occasionally, it appears to be facing the wrong direction. A simple explanation for this is that there is not enough equations to solve for all variables in the equation. For this reason, the sign +/- is estimated, which respresents a "flipped" distribution. This is represented by a vector facing the wrong direction.

```
In [12]: compareMixingMatrix(synthData, verifyA, estimateA)
```

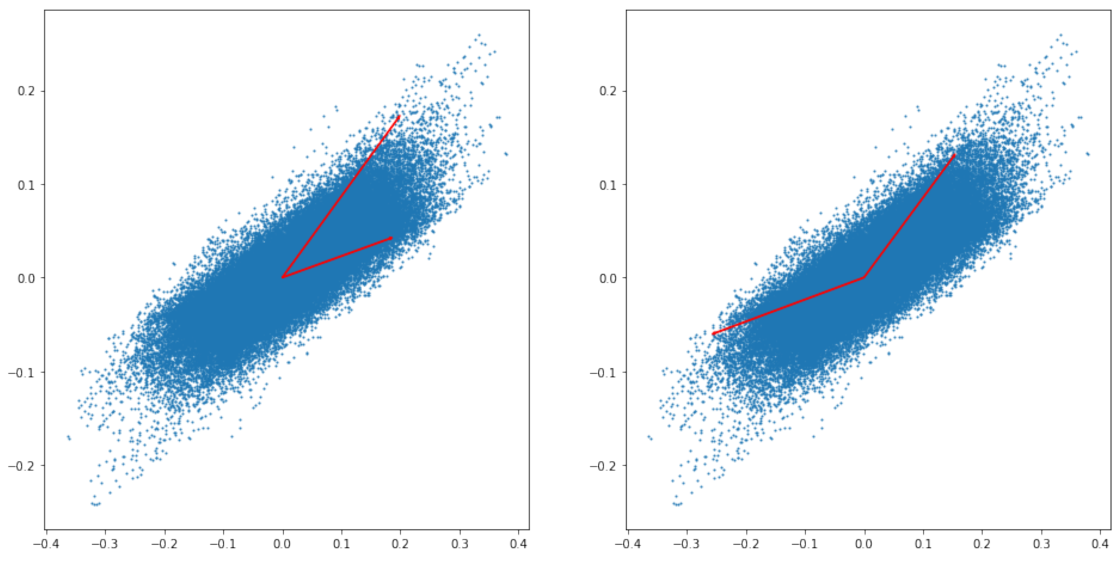


```
In [13]: A, S = bss(X)
```

```
In [14]: ### normalized sound tracks
         S = audionorm(S)
         # write recovered sound track into WAV files
         wavfile.write('data/separatedTrackA.wav', 22050, S[0])
         wavfile.write('data/separatedTrackB.wav', 22050, S[1])
```

I have not set a random seed for this notebook, to allow the functions to be run by other users. As mentioned previously, the vectors (which represent the positive direction of the distributions) are ocassionally pointing the wrong direction, indicating a flipped distribution.

```
In [15]: compareMixingMatrix(X, gtruthA, A)
```

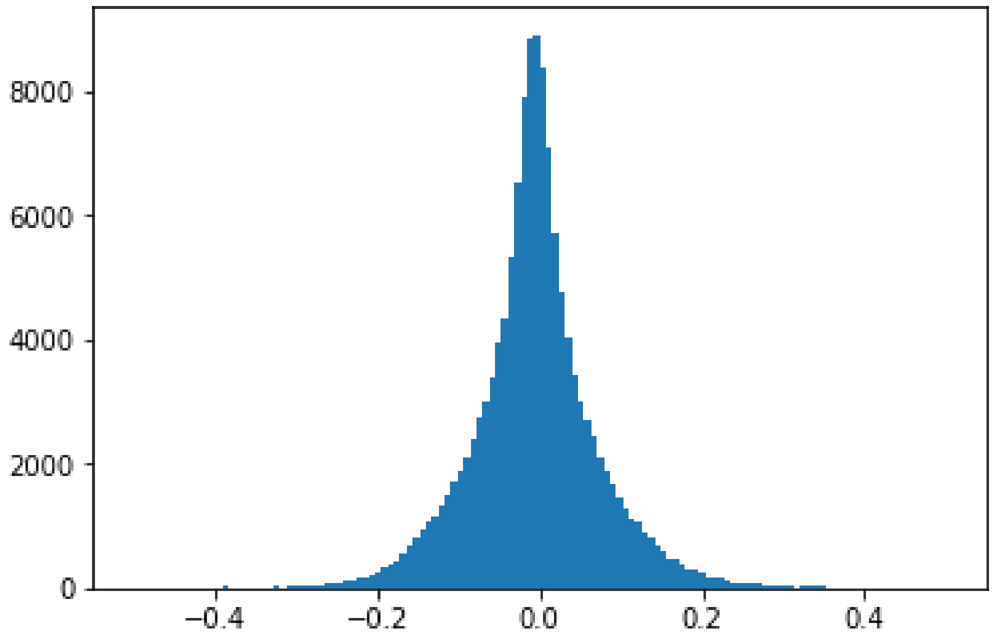


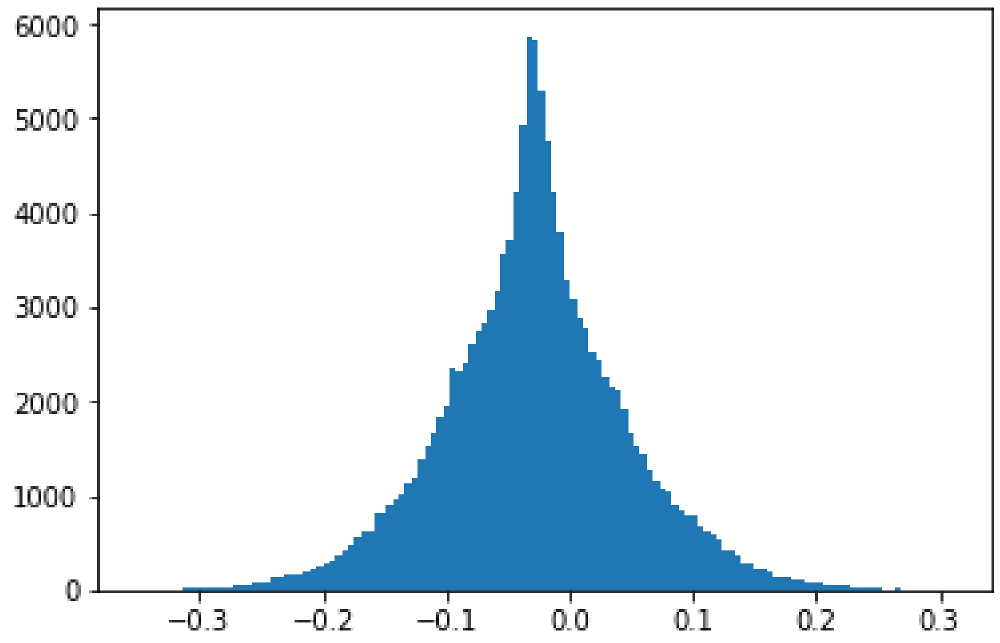
```
In [16]: print('\nMixing Matrix (Our Estimation)\n\n', A)
         print('\nMixing Matrix (Groud Truth)\n\n', gtruthA)
```

```
Mixing Matrix (Our Estimation)

 [[ 1.         -1.66824441]
 [ 0.85183554 -0.39164574]]

Mixing Matrix (Groud Truth)

 [[0.61847544 0.66618213]
 [0.14127466 0.57728887]]
```

We can output the distribution of the estimated signals. The distributions are very laplacian, indicating that the maximization function worked as expected.

```
In [17]: plt.hist(S[0], 128)
         plt.show()
         plt.hist(S[1], 128)
         plt.show()
```





Full Unsupervised BSS is technically considered an semi supervised algorithm due to the necessity to specify a prior distribution. This is a limitation for real world applications of the algorithm since the distribution is generally unknown.

To fix this problem, we will calculate the log-likelihood of distributions of various q values. In addition, we will be descending a set amount of steps for each q. The values of q with the highest log-likelihood after the set amount of steps is the q that results in the smallest mixing matrix gradients. This indicates that the model is converging faster. As such, we will fit the data with the value of q that provides the smallest gradients, i.e. converges fastest, with the assumption that the distribution is best fit by this q.

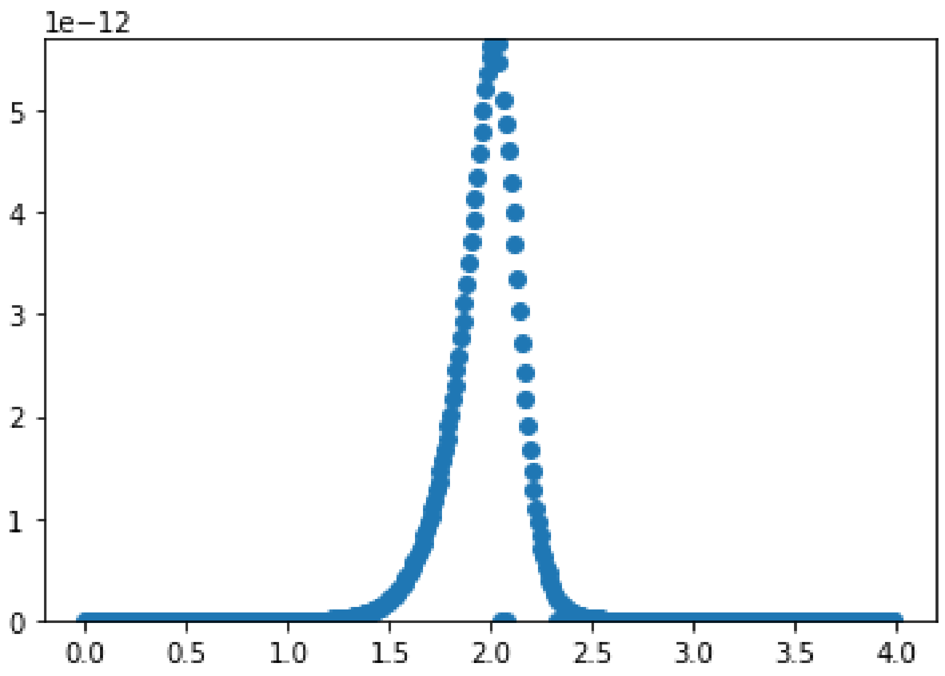
This method is fully unsupervised. However, it is not perfect for situations where certain distributions fit the mixed data better than the true prior distribution. This will be explored when testing this method on the synthetic data.

```
In [18]: def bss(data, max_iter=20, q=1, nu=0.5, max_grad=10, q_max = 4):
             sources, length = data.shape
             I = np.identity(sources)
             likelihood = np.zeros((q_max * 100))
             B = np.zeros((q_max * 100, sources, sources))
             q_range = np.arange(0, q_max, 0.01)
             for q in q_range:
                 j = int(q * 100)
                 A = np.identity(sources)
                 for i in range(max_iter):
                     s = np.linalg.inv(A) @ data
                     A_grad = -A @ (d_logP(s, q + 0.01) @ s.T + I)
```

```
                A += np.multiply(nu, A_grad)
            likelihood[j] = np.mean(Pxa(s, A, q + 0.01))

            B[j,:,:] = A

        fig = plt.figure()
        ax = plt.gca()
        ax.scatter(q_range, likelihood)
        plt.ylim(likelihood.min(), likelihood.max())
        #ax.set_yscale('log')
        plt.show()
        print("q:", likelihood.argmax()/100)
        A = B[likelihood.argmax()]
        return A/A.max(), np.linalg.inv(A) @ data

    def P(s, q):
        return np.exp(-np.power(np.abs(s),q))

    def d_logP(s, q):
        return -np.divide(q * np.power(np.abs(s), q), s)

    def Pxa(s, A, q):
        #print("s", s, "A", A, "q", q)
        return P(s, q)/np.linalg.det(A)

    #def d_logP_q(s, q):
    #    return -np.power(np.abs(s), q) * np.log(np.abs(s))

    def d_logP_q(s, q):
        A = np.log2(q/s.shape[1] * np.power(np.abs(np.subtract(s, np.mean(s))), q))
        B = q**2
        return np.divide(A, B)
```

As we can see, the optimal value for q, without any prior information, is `1.92`. This isn't the best guess, as it assumes a gaussian distribution. However, a multivariate laplace distribution is gaussian along the orthogonal direction, so this result is understandable.

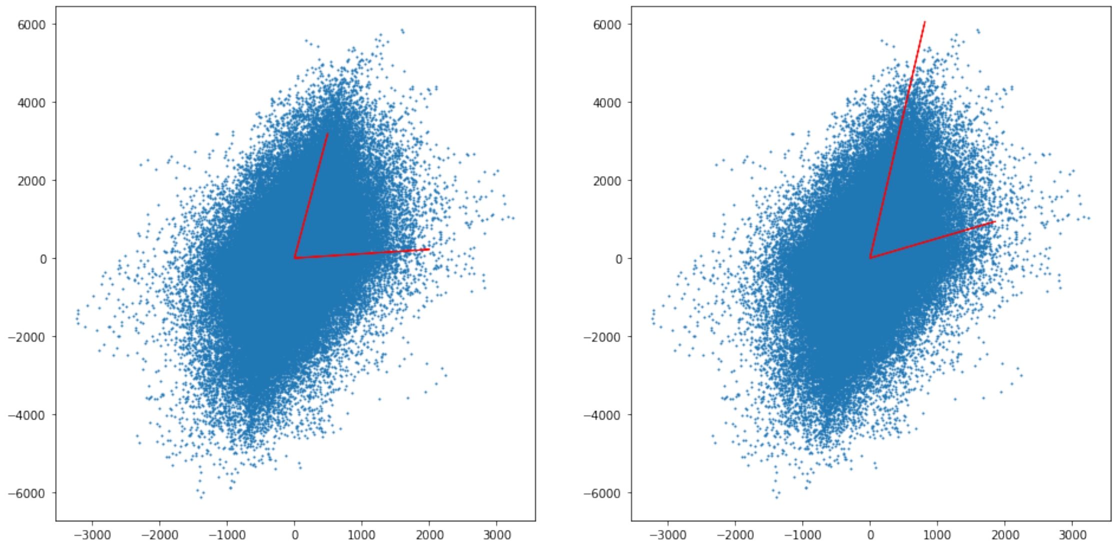
```
In [19]: gtruthS = np.asarray((dataBach, dataSpeech))
         X, gtruthA = simpleMixer(gtruthS)
         A, S = bss(X)

         S = audionorm(S)
         # write recovered sound track into WAV files
         wavfile.write('data/separatedTrackA_2.wav', 22050, S[0])
         wavfile.write('data/separatedTrackB_2.wav', 22050, S[1])
```



```
q: 2.02
```

The resulting mixing matrix is fairly accurate, and is more than enough to audibly seperate the two sources.

```
In [20]: compareMixingMatrix(X, gtruthA, A)
```



Images Since we have a method for estimating q, we can now apply BSS on unknown distributions with some accuracy. Since the distribution of the pixel intensities is not known in two mixed images, this methodology should provide a mechanism seperating the mixture into two images.

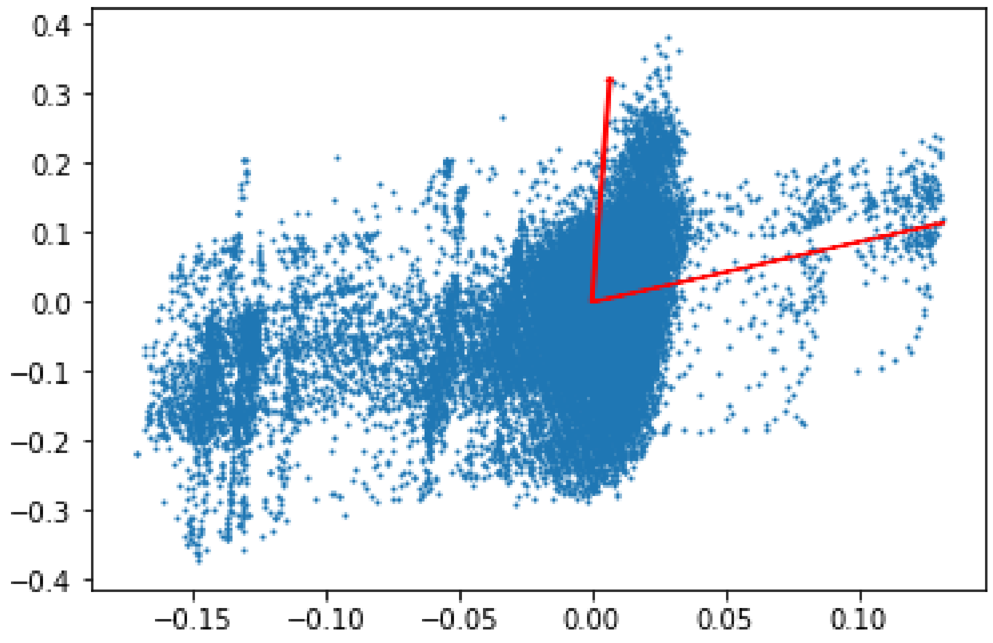
```
In [21]: import cv2
         import matplotlib.pyplot as plt

         img0 = cv2.imread('data/frame3.png', 0)
         img1 = cv2.imread('data/frame7.png', 0)

         img0_vec = np.ravel(img0/img0.max())
         img1_vec = np.ravel(img1/img1.max())

         img0_vec -= np.mean(img0_vec)
         img1_vec -= np.mean(img1_vec)
```

```
In [22]: gtruthI = np.asarray([img0_vec, img1_vec])
```

```
In [23]: I, ItruthA  = simpleMixer(gtruthI)
```

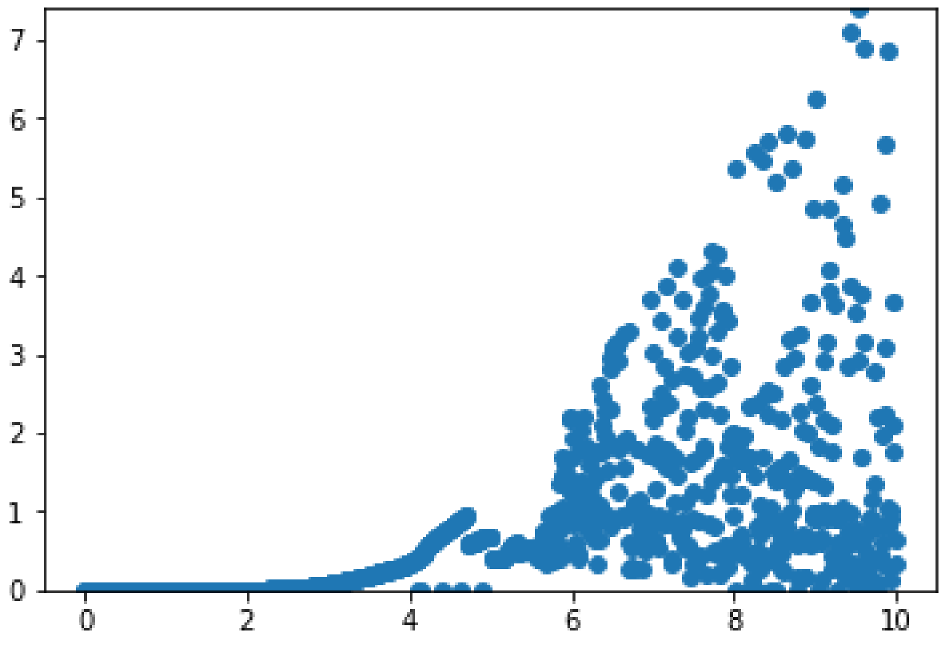
As with before, we can plot the distribution and it's corresponding true mixing matrix.

```
In [24]: drawDataWithMixingMatrix(I, ItruthA)
```



Next, we will estimate the optimal distribution and find it's corresponding q.

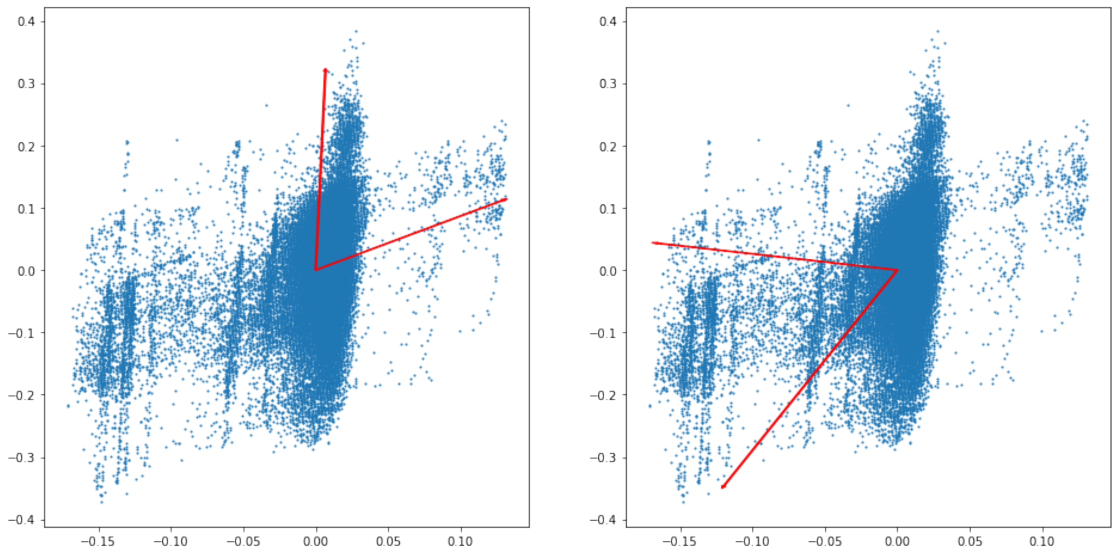
```
In [25]: IestimateA, recoverI = bss(I, q_max = 10)
```



```
q: 9.52
```

The result is somewhat close to the original mixing matrix, especially given the lack of any information about the distribution. The image distributions are inheritly multi-modal, which is poorly represented by a generalized gaussian. Even so, the algorithm is able to provide a fairly good estimate.

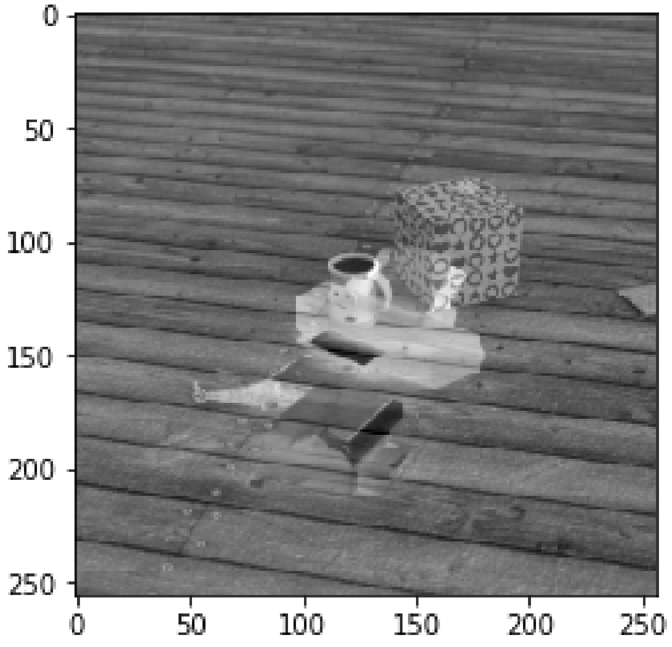
```
In [26]: compareMixingMatrix(I, ItruthA, IestimateA)
```



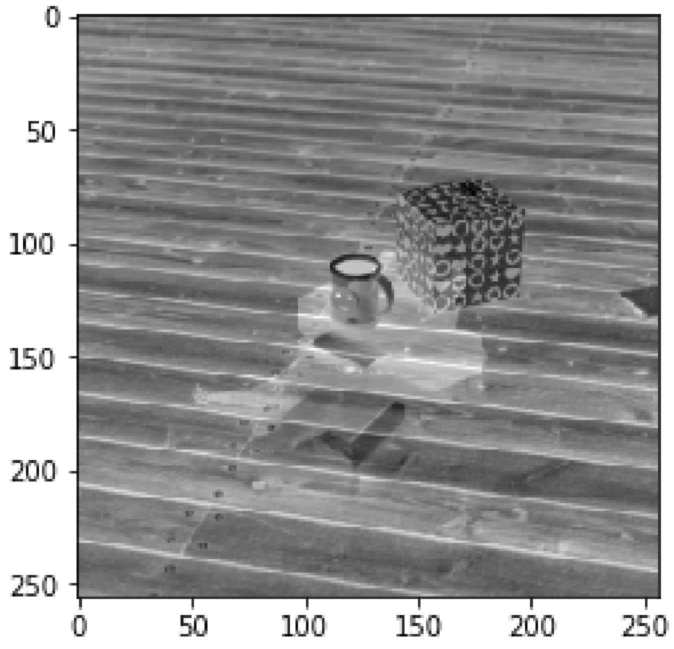
```
In [27]: def img_reshape(img):
             return np.reshape(img, (256, 256))
```

Note that the reconstructed images may be inverted for the same reason that the vectors are flipped. However, the stucture of the image and the differences between the intensities are kept.

```
In [28]: plt.imshow(img_reshape(recoverI[0]), cmap='gray')
         plt.show()
```



```
In [29]: plt.imshow(img_reshape(recoverI[1]), cmap='gray')
         plt.show()
```



0.1.2 Conclusion

This project provided a lot of insight on Blind Source Separation, the application of ICA is very broad and can be extrapolated to a variety of modelling problems with varying success. ICA appears to be limited to the availability of a prior, which can be found via a learning algorithm. A basic implementation of such a learning algorithm was implemented here.