paper

June 25, 2020

1 code

[170]:
```
import numpy as np
import matplotlib.pylab as py
from matplotlib.ticker import MaxNLocator
import pandas as pa
from dimredu.denseSolvers import eRPCA as RPCA
```

2 Introduction

2.1 Background

Requiring metrics for the evaluation of healthcare systems and initiatives is necessary in order to improve healthcare safety, quality and value. However, collecting and managing healthcare quality measures is expensive and time consuming. Additionally, proposed sets of health metrics are often larger than necessary. Proposed metrics often do not account for redundancy of information between metrics, cost of gathering metrics, or reliability of various metrics. [Ref AcademyHealthAbstract] Our research was performed initially with synthetic data where ground truth is known and error can be precisely calculated. These methods were then applied to health metrics obtained from public use files containing actual health metrics collected over a period of one year.The methods proposed herein can be used to solve this government problem by reducing burden and improving quality.

2.2 Summary of methods

We try the following

- PCA
- This fails because it cannot do predictions with outliers
- Combinatorial approachs (Gurobi)
- Too slow for real problems
- RPCA
- This works great!

2.2.1 Traditional PCA

Traditional Principal Components Analysis (PCA) works well for dimensionality reduction, however, it is often seen as difficult to interpret. The reason for this is that PCA produces a set of results which are linear combination of features. This is useful for understanding the rank or underlying dimension, but does not help to select specific measures from a set of attributes. Additionally, and an important note for this research, is that PCA is extremely sensitive to ouliers. One outlier in the data can completely change the results.

Note, that PCA can be phrased as an optimization problem, namely

$$\min_{L} \|L - M\| \|\text{s.t. } \rho(K) \leq k$$

This optimization perspective will inspire us in the sequel.

2.2.2 Robust PCA

The Convex Optimization approach is based on Robust Principal Component Analysis (RPCA). The RPCA approach will both remove anomalies and provide a low rank approximation of the original data. This is accomplished with a combination of a nuclear norm and a one norm whcih is regularized by a tuning parameter, λ to induce sparsity. The Robust PCA formulation is as follows.

$$\min_{L,S} ||L||_* + \lambda||S||_1 \text{s.t. } L + S = M$$

Approximation methods are used as implemented in the dimredu python package and available as open source here.

https://bitbucket.org/rcpaffenroth/dimredu/src/master/

2.2.3 Combinatorial Approach

Ok, this is important. We are not actually doing any Sparse PCA!? This indicates we are just doing Robust PCA. That does agree with my memory, now that I think about it. This makes things easier, but I am not sure what the combinatorial approach is anymore.

**Are we going to compare against Gurobi? I mean, that would be nice, and perhaps even easier than the case where we do Sparse PCA.

3 Discussion of Robust PCA versus Sparse PCA

There are two ways of thinking about the problem of interest. Namely we wish to reconstruct a data matrix from just a *few columns*. Note, it is not of interest to reconstuct the data matrix X from a few *linear combinations* of the columns since the columns are expensive to measure, and therefore linear combinations of such columns, even if there are just a few of them, will be similarily expensive to measure.

So, just to give us a bit of nomenclature we call * PCA the act of reconstructing a data matrix from a few linear combinations of columns * Sparse PCA the act of reconstructing a data matrix from a columns

Suprisingly, due to a theorem by Servi et al. PCA and Sparse PCA actually give the same answer if X is exactly low rank!

Let's do an example to fix the ideas

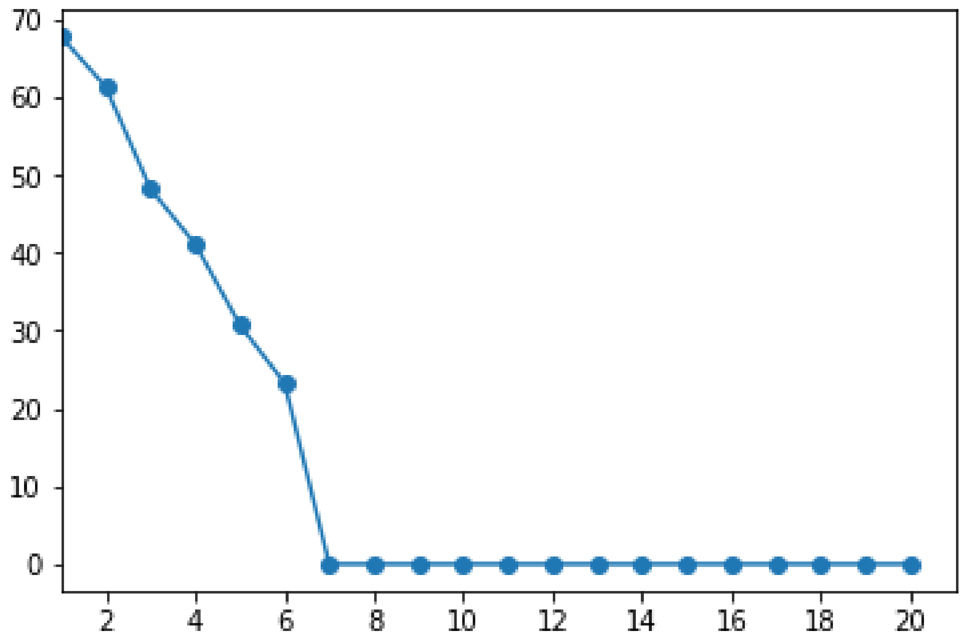
[171]:
```
# Let's do an example to make things precise
def generateData(N, n, K, numAnoms=0, sizeAnoms=0, noiseSize=0, seed=1234):
    """
    N is the number of samples/row
    n is the number of features/columns
    K is number of independent features/rank of X
    numAnoms is the number of anomalies
    sizeAnoms is the size of each anomaly
    """
    np.random.seed(seed)

    U = np.random.normal(size=[N, K])
    V = np.random.normal(size=[K, n])
    L = U @ V

    S = np.zeros(L.shape)
    for k in range(numAnoms):
        i = np.random.randint(N)
        j = np.random.randint(n)
        S[i, j] += sizeAnoms

    N = np.random.normal(size=L.shape)*noiseSize

    X = L + S + N
    return X, L, S, N
```

[172]:
```
# Here is some data
X,_,_,_ = generateData(N=200, n=20, K=6)
X_Train = X[:100,:]
X_Test = X[100:,:]
```

[173]:
```
def EPlot(E, ax):
    # Make ticks at integers
    ax.xaxis.set_major_locator(MaxNLocator(integer=True))
    ax.set_xlim(1,len(E)+1)
    ax.plot(range(1,len(E)+1), E, 'o-')
```

Note that the SVD gets the rank exactly right, to machine precision.

```
[174]: # that is exactly low rank
U,E,VT = np.linalg.svd(X_Train)
EPlot(E, py.gca())
```



3.1 The PCA approach to reconstruction.

By the SVD we know that $X = U\Sigma V^T$. If X is low rank and $n < N$ we have that

$$\begin{aligned} X &= \begin{bmatrix} U_{1:K,1:K} & U_{1:K,K+1:n} \\ U_{K+1:N,1:K} & U_{K+1:N,K+1:n} \end{bmatrix} \begin{bmatrix} \Sigma_{1:K,1:K} & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} V_{1:K,1:K} & V_{1:K,K+1:n} \\ V_{K+1:n,1:K} & V_{K+1:n,K+1:n} \end{bmatrix} && (1) \\ &= \begin{bmatrix} U_{1:K,1:K} & 0 \\ U_{K+1:N,1:K} & 0 \end{bmatrix} \begin{bmatrix} \Sigma_{1:K,1:K} & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} V_{1:K,1:K} & V_{1:K,K+1:n} \\ 0 & 0 \end{bmatrix} && (2) \\ &= \begin{bmatrix} U_{1:K,1:K} \\ U_{K+1:N,1:K} \end{bmatrix} \begin{bmatrix} \Sigma_{1:K,1:K} \end{bmatrix} \begin{bmatrix} V_{1:K,1:K} & V_{1:K,K+1:n} \end{bmatrix} && (3) \\ &= \hat{U}\hat{\Sigma}\hat{V}^T && (4) \end{aligned}$$

Our reduced representation is therefore

$X\hat{V}$

and our reconstruction is

$X\hat{V}\hat{V}^T$

[175]:
```python
# This is the code for doing the compression using PCA
class PCACompress(object):
    def __init__(self, X_Train, dim=6):
        # Compute the SVD of the input matrix
        U,E,VT = np.linalg.svd(X_Train)
        # Just make the notation a little clearer
        V = VT.T
        # This is the matrix that transforms X to the compressed version
        self.VHat = V[:,:dim]

    def compress(self, X):
        return X @ self.VHat

    def decompress(self, X):
        return X @ self.VHat.T
```

[176]:
```python
# The parameters of our calcuation
N = 200
n = 20
K = 6
```

[177]:
```python
# Here is some data
X,_,_,_ = generateData(N=N, n=n, K=K)
X_Train = X[:100,:]
X_Test = X[100:,:]
```

[178]:
```python
f = PCACompress(X_Train)
```

3.1.1 Training data

We first show results on training data.

[179]:
```python
X_Compressed = f.compress(X_Train)
```

[180]:
```python
# The shape is correct! We only have 6 columns now
X_Compressed.shape
```

[180]: (100, 6)

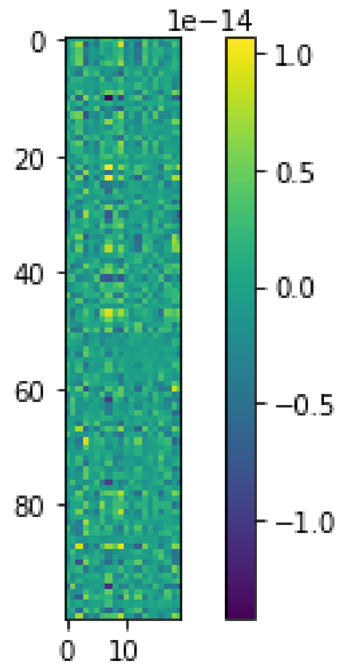
We reconstruct the training data perfectly with error at machine precision

[181]:
```python
# But we can exactly reconstruct X
X_Decompressed = f.decompress(X_Compressed)

# and gets the training X exactly
np.linalg.norm(X_Train - X_Decompressed)
```

[181]: 1.1367308024034381e-13

[182]:
```
py.imshow(X_Train - X_Decompressed)
py.colorbar()
```

[182]: <matplotlib.colorbar.Colorbar at 0x7fd7dcc39d10>



3.1.2 Testing data

Sinnce the input data is exactly low rank, we get machine precision errors on the testing data too!

[183]:
```
X_Compressed = f.compress(X_Test)
```

[184]:
```
# The shape is correct! We only have 6 columns now
X_Compressed.shape
```

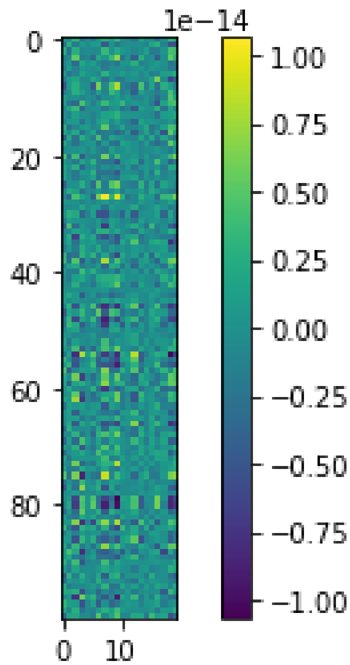
[184]: (100, 6)

[185]:
```
# But we can exactly reconstruct X
X_Decompressed = f.decompress(X_Compressed)

# and gets the training X exactly
np.linalg.norm(X_Test - X_Decompressed)
```

[185]: 1.1412344321263945e-13

```
[186]: py.imshow(X_Test - X_Decompressed)
       py.colorbar()
```

[186]: <matplotlib.colorbar.Colorbar at 0x7fd7f2818810>



3.2 The Sparse PCA approach to reconstruction

Now, we attempt a different problem. While PCA compresses the data using linear combinations, here we merely select columns! Now, the other columns do not even need to be a measured.

S is a selection matrix with $S \in \{0,1\}^{n \times K}$ and every column of S has precisely a single entry of 1 and every row of S has at most a single entry of 1.

For example

$$S = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Selects the 1-st, 3-rd and 4-th column of an X that originally had 6 columns.

So, in general we write

$$X_K = XS$$

for selecting k columns from X.

If were to estimate X from X_K we would need two properties.

First, our estimate $\hat{X}$ of X should have the property that it agrees with X on S, in other words there is some Z such that

$$\hat{X} = XSS^T + Z(I - SS^T) = X_K S^T + Z(I - SS^T)$$

Second, our estimate $\hat{X}$ must lay on the subspace spanned by singular vectors of X. Equivalently, we need that

$$\hat{X} = \hat{X}VV^T.$$

Plugging the first into the second we get

$$X_K S^T + Z(I - SS^T) = (X_K S^T + Z(I - SS^T))VV^T$$

expanding we get

$$X_K S^T + Z(I - SS^T) = X_K S^T VV^T + Z(I - SS^T)VV^T$$

Collecting terms

$$X_K(S^T - S^T VV^T) = Z((I - SS^T)VV^T - (I - SS^T))$$

which we can solve for Z and **only depends on** X_K.

[187]:
```
# This is the code for doing the compression using PCA
class SPCACompress(object):
    def __init__(self, X_Train, Selection, dim=6):
        # The user defined columns that you want to keep
        self.S = Selection
        # Compute the SVD of the input matrix
        U,E,VT = np.linalg.svd(X_Train)
        # Just make the notation a little clearer
        V = VT.T
        # This is the matrix that transforms X to the compressed version
        self.VHat = V[:,:dim]

    def compress(self, X):
        return X @ self.S
```

```
    def decompress(self, X):
        # We will later need an identity of the correct size
        I = np.eye(self.VHat.shape[0])
        A = (I - self.S @ self.S.T) @ self.VHat @ self.VHat.T - (I - self.S @␣
↪self.S.T)
        B = X @ (self.S.T - self.S.T @ self.VHat @ self.VHat.T)
        Z = B @ np.linalg.pinv(A)
        return X@self.S.T + Z@(I-self.S@self.S.T)
```

[188]:
```
# The parameters of our calcuation
N = 200
n = 20
K = 6
```

[189]:
```
# Here is some data
X,_,_,_ = generateData(N=N, n=n, K=K)
X_Train = X[:100,:]
X_Test = X[100:,:]
```

[190]:
```
# Select the columns that we like.  This is user defined!  You can pick␣
↪whatever columns you like.
Selection = np.zeros([20,6])
for i in range(6):
    Selection[i,i] = 1
```

[191]:
```
f = SPCACompress(X_Train, Selection)
```

3.2.1 Training data

Since the data is low-rank, and in general position, the training error is 0.

[192]:
```
X_Compressed = f.compress(X_Train)
```

[193]:
```
# The shape is correct! We only have 6 columns now
X_Compressed.shape
```

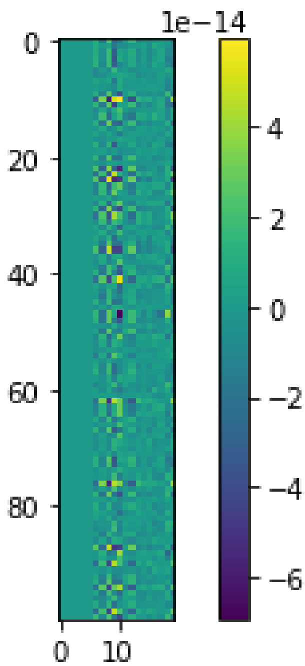
[193]:
```
(100, 6)
```

[194]:
```
# But we can exactly reconstruct X
X_Decompressed = f.decompress(X_Compressed)

# and gets the training X exactly
np.linalg.norm(X_Train - X_Decompressed)
```

[194]:
```
4.967012683616506e-13
```

[195]:
```
py.imshow(X_Train - X_Decompressed)
py.colorbar()
```

[195]: <matplotlib.colorbar.Colorbar at 0x7fd7dd091b50>



3.2.2 Testing data

As is the testing error!

[196]:
```
X_Compressed = f.compress(X_Test)
```

[197]:
```
# The shape is correct! We only have 6 columns now
X_Compressed.shape
```

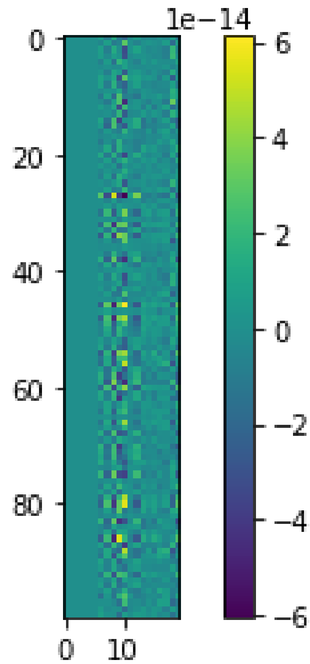
[197]: (100, 6)

[198]:
```
# But we can exactly reconstruct X
X_Decompressed = f.decompress(X_Compressed)

# and gets the training X exactly
np.linalg.norm(X_Test - X_Decompressed)
```

[198]: 4.850297235574327e-13

[199]:
```
py.imshow(X_Test - X_Decompressed)
py.colorbar()
```

[199]: <matplotlib.colorbar.Colorbar at 0x7fd7dd618e10>



4 Adding noise

We now make the problem harder by adding noise.

4.1 To PCA

[200]:
```
# The parameters of our calcuation
N = 200
n = 20
K = 6
```

[201]:
```
# Here is some data
X,_,_,_ = generateData(N=N, n=n, K=K, noiseSize=1)
X_Train = X[:100,:]
X_Test = X[100:,:]
```

[202]:
```
U,E,VT = np.linalg.svd(X_Train)

V = VT.T
VHat = V[:,:6]
```

[203]:
```
f = PCACompress(X_Train)
```

4.1.1 Training data

We now do not get an error of 0, but an error that depends on the size of the noise.

[204]:
```
X_Compressed = f.compress(X_Train)
```

[205]:
```
# The shape is correct! We only have 6 columns now
X_Compressed.shape
```

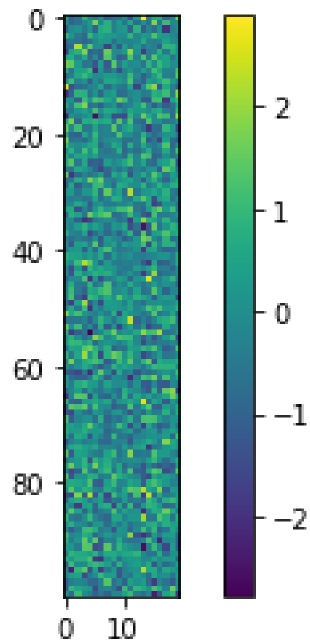
[205]:
```
(100, 6)
```

[206]:
```
# But we can exactly reconstruct X
X_Decompressed = f.decompress(X_Compressed)

# and gets the training X exactly
np.linalg.norm(X_Train - X_Decompressed)
```

[206]:
```
36.5250570541932
```

[207]:
```
py.imshow(X_Train - X_Decompressed)
py.colorbar()
```

[207]:
```
<matplotlib.colorbar.Colorbar at 0x7fd7dce9af90>
```



4.1.2 Testing data

We are not overfitting this data, so the size of the testing error is about the same.

[208]:
```
X_Compressed = f.compress(X_Test)
```

[209]:
```
# The shape is correct! We only have 6 columns now
X_Compressed.shape
```

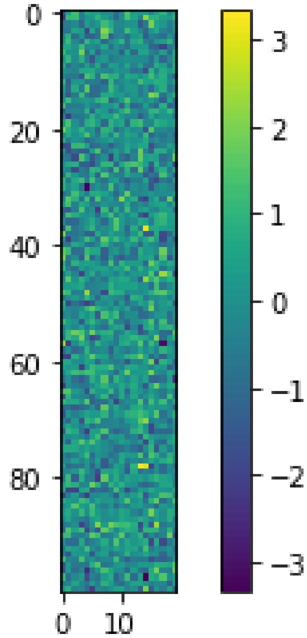
[209]: (100, 6)

[210]:
```
# But we can exactly reconstruct X
X_Decompressed = f.decompress(X_Compressed)

# and gets the training X exactly
np.linalg.norm(X_Test - X_Decompressed)
```

[210]: 38.42467039880988

[211]:
```
py.imshow(X_Test - X_Decompressed)
py.colorbar()
```

[211]: <matplotlib.colorbar.Colorbar at 0x7fd7dd082a90>



4.2 To Sparse PCA

In the presence of noise, the performance on Sparse PCA is similar, but with larger errors.

[212]:
```
# The parameters of our calcuation
N = 200
n = 20
K = 6
```

[213]:
```
# Here is some data
X,_,_,_ = generateData(N=N, n=n, K=K, noiseSize=1)
X_Train = X[:100,:]
X_Test = X[100:,:]
```

[214]:
```
# Select the columns that we like.  This is user defined!  You can pick␣
↪whatever columns you like.
S = np.zeros([20,6])
for i in range(6):
    S[i,i] = 1
```

[215]:
```
f = SPCACompress(X_Train, S)
```

4.2.1 Training data

[216]:
```
X_Compressed = f.compress(X_Train)
```

[217]:
```
# The shape is correct! We only have 6 columns now
X_Compressed.shape
```

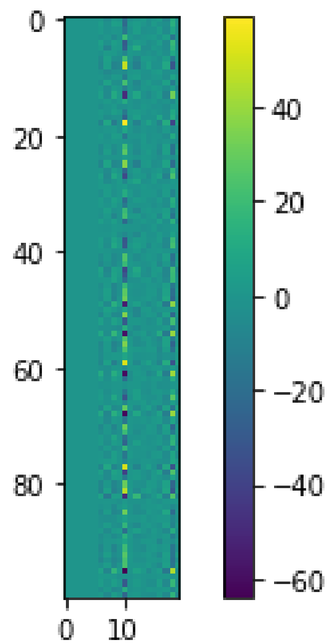
[217]:
```
(100, 6)
```

[218]:
```
# But we can exactly reconstruct X
X_Decompressed = f.decompress(X_Compressed)

# and gets the training X exactly
np.linalg.norm(X_Train - X_Decompressed)
```

[218]:
```
352.01338062315403
```

[219]:
```
py.imshow(X_Train - X_Decompressed)
py.colorbar()
```

[219]:
```
<matplotlib.colorbar.Colorbar at 0x7fd7dd34e450>
```



4.2.2 Testing data

[220]:
```
X_Compressed = f.compress(X_Test)
```

[221]:
```
# The shape is correct! We only have 6 columns now
X_Compressed.shape
```

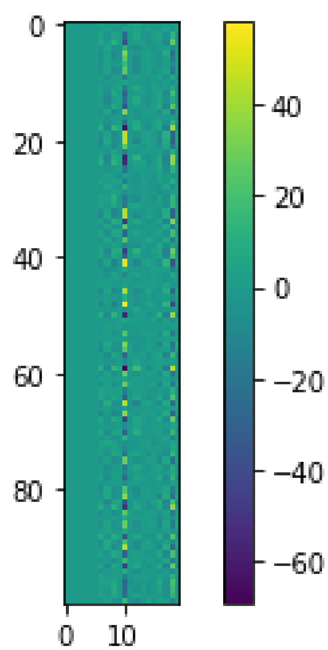
[221]:
```
(100, 6)
```

[222]:
```
# But we can exactly reconstruct X
X_Decompressed = f.decompress(X_Compressed)

# and gets the training X exactly
np.linalg.norm(X_Test - X_Decompressed)
```

[222]:
```
378.3778251843612
```

[223]:
```
py.imshow(X_Test - X_Decompressed)
py.colorbar()
```

[223]:
```
<matplotlib.colorbar.Colorbar at 0x7fd7f2e3ea50>
```



5 Adding anomalies

Now things start to get interesting. We start adding anomalies!

5.1 To PCA

[224]:
```
# The parameters of our calcuation
N = 200
n = 20
K = 6
```

[225]:
```
# Here is some data
X, L, S, _ = generateData(N=N, n=n, K=K, numAnoms=10, sizeAnoms=10)
X_Train = X[:100,:]
X_Test = X[100:,:]
```

[226]:
```
U,E,VT = np.linalg.svd(X_Train)

V = VT.T
VHat = V[:,:6]
```

[227]:
```
f = PCACompress(X_Train)
```

5.1.1 Training data

[228]:
```
X_Compressed = f.compress(X_Train)
```

[229]:
```
# The shape is correct! We only have 6 columns now
X_Compressed.shape
```

[229]:
```
(100, 6)
```

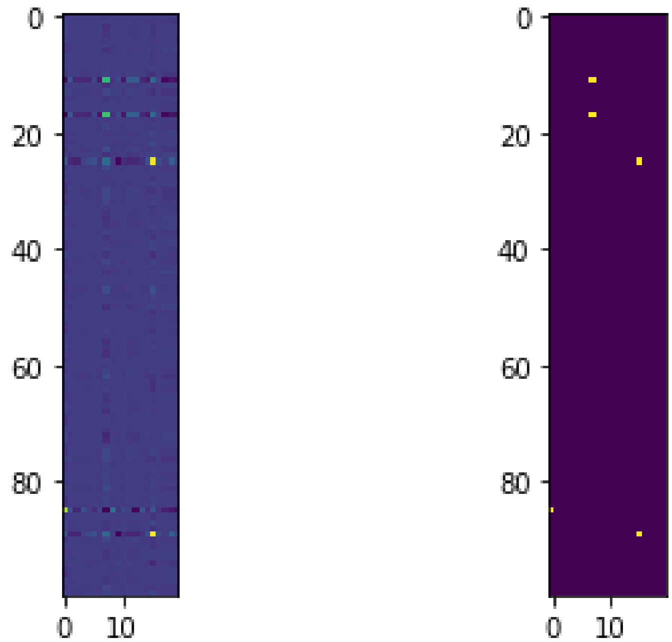
[230]:
```
# But we can exactly reconstruct X
X_Decompressed = f.decompress(X_Compressed)

# and gets the training X exactly
np.linalg.norm(X_Train - X_Decompressed)
```

[230]:
```
18.581581217138087
```

Just as we thought, the anomalies mess up a whole row!

[231]:
```
_, ax = py.subplots(ncols=2)
ax[0].imshow(X_Train - X_Decompressed)
ax[1].imshow(S[:100, :])
```

[231]:
```
<matplotlib.image.AxesImage at 0x7fd7dcfb49d0>
```



5.1.2 Testing data

[232]:
```
X_Compressed = f.compress(X_Test)
```

[233]:
```
# The shape is correct! We only have 6 columns now
X_Compressed.shape
```

[233]:
```
(100, 6)
```

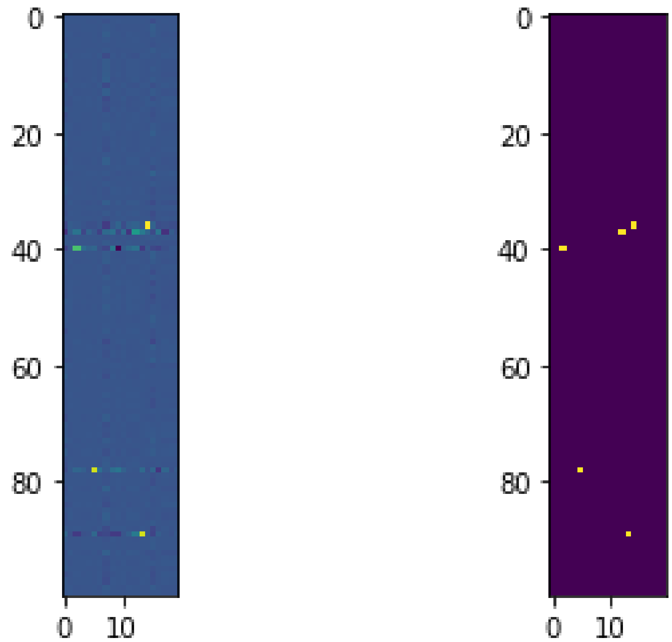
[234]:
```
# But we can exactly reconstruct X
X_Decompressed = f.decompress(X_Compressed)

# and gets the training X exactly
np.linalg.norm(X_Test - X_Decompressed)
```

[234]:
```
19.59234442730249
```

With the same thing happening on the testing data.

[235]:
```
_, ax = py.subplots(ncols=2)
ax[0].imshow(X_Test - X_Decompressed)
ax[1].imshow(S[100:, :])
```

[235]: <matplotlib.image.AxesImage at 0x7fd7e4068a90>



5.2 To Sparse PCA

[236]:
```
# The parameters of our calcuation
N = 200
n = 20
K = 6
```

[237]:
```
# Here is some data
X, L, S, _ = generateData(N=N, n=n, K=K, numAnoms=10, sizeAnoms=10)
X_Train = X[:100,:]
X_Test = X[100:,:]
```

[238]:
```
print(S)
```

```
[[0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]
 …
 [0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]]
```

[239]:
```
# Select the columns that we like.  This is user defined!  You can pick␣
↪whatever columns you like.
Selection = np.zeros([20,6])
for i in range(6):
    Selection[i,i] = 1
```

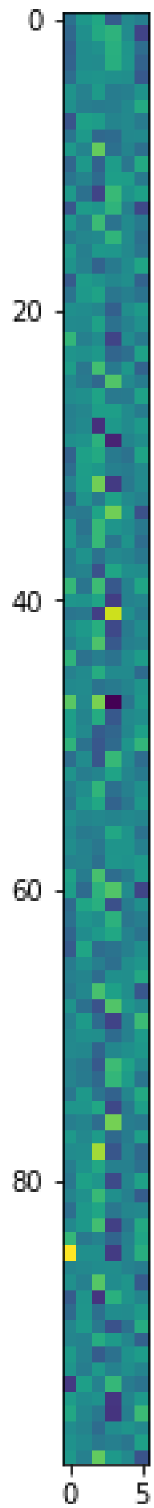
[240]:
```
f = SPCACompress(X_Train, Selection)
```

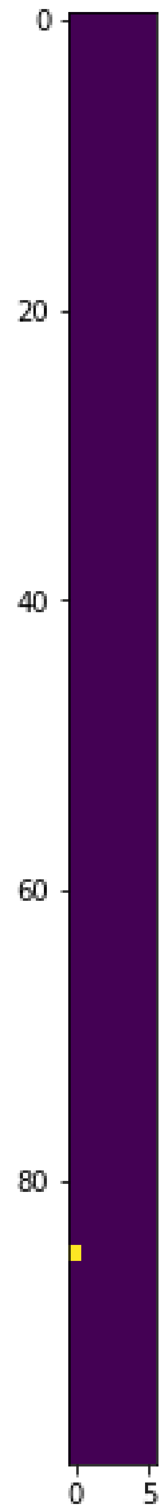
5.2.1 Training data

Checking for anomalies in the input data. Having one of these in the selected columns will mess up many things on the testing data.

[241]:
```
_, ax = py.subplots(ncols=2,figsize=(10,10))
ax[0].imshow(X[:100, :] @ Selection)
ax[1].imshow(S[:100, :] @ Selection)
```

[241]:
```
<matplotlib.image.AxesImage at 0x7fd7f27ea590>
```





[242]:
```
X_Compressed = f.compress(X_Train)
```

[243]:
```python
# The shape is correct! We only have 6 columns now
X_Compressed.shape
```

[243]: (100, 6)

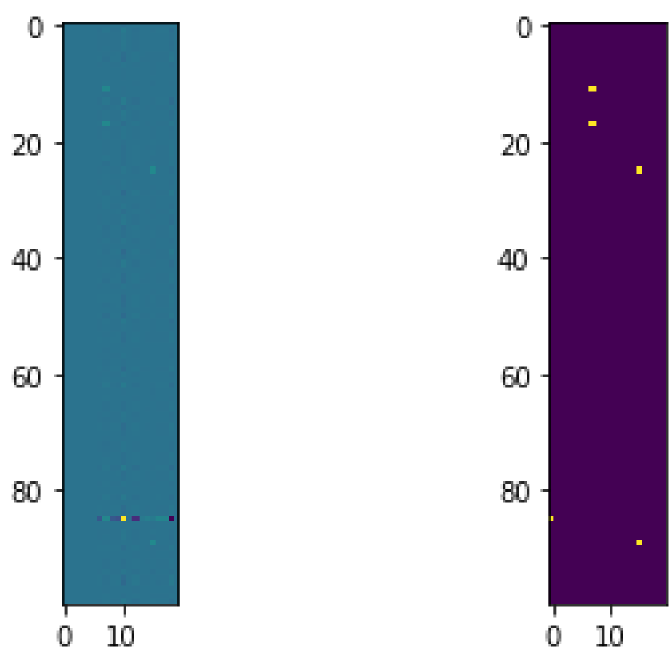
[244]:
```python
# But we can exactly reconstruct X
X_Decompressed = f.decompress(X_Compressed)

# and gets the training X exactly
np.linalg.norm(X_Train - X_Decompressed)
```

[244]: 89.68052824574707

Note, only one row is messed up (a lot) in the training data. Why? Only one anomaly happens to appear in the training data.

[245]:
```python
_, ax = py.subplots(ncols=2)
ax[0].imshow(X_Train - X_Decompressed)
ax[1].imshow(S[:100, :])
```

[245]: <matplotlib.image.AxesImage at 0x7fd7f0158090>

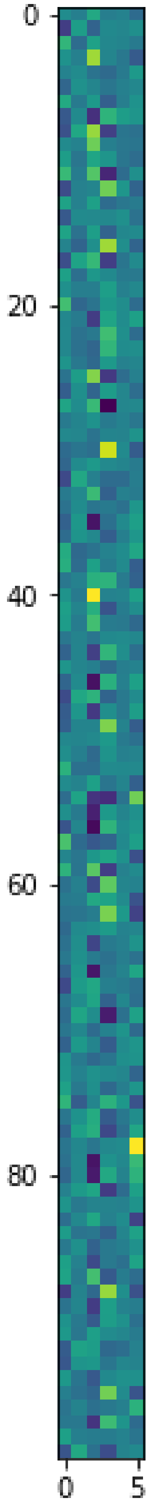


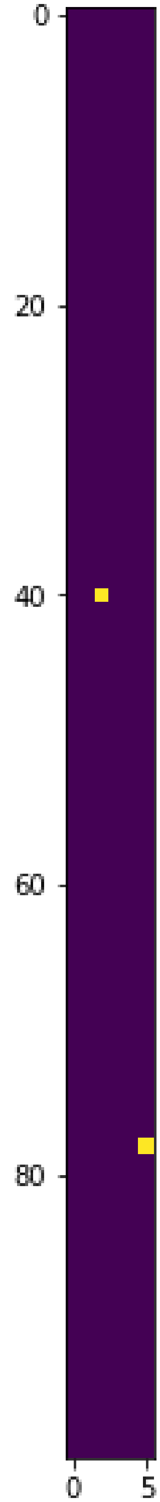
5.2.2 Testing data

Checking for anomalies in the input data

[246]:
```
_, ax = py.subplots(ncols=2,figsize=(10,10))
ax[0].imshow(X[100:, :] @ Selection)
ax[1].imshow(S[100:, :] @ Selection)
```

[246]: <matplotlib.image.AxesImage at 0x7fd7f02512d0>





[247]:
```
X_Compressed = f.compress(X_Test)
```

[248]:
```
# The shape is correct! We only have 6 columns now
X_Compressed.shape
```

[248]:
```
(100, 6)
```

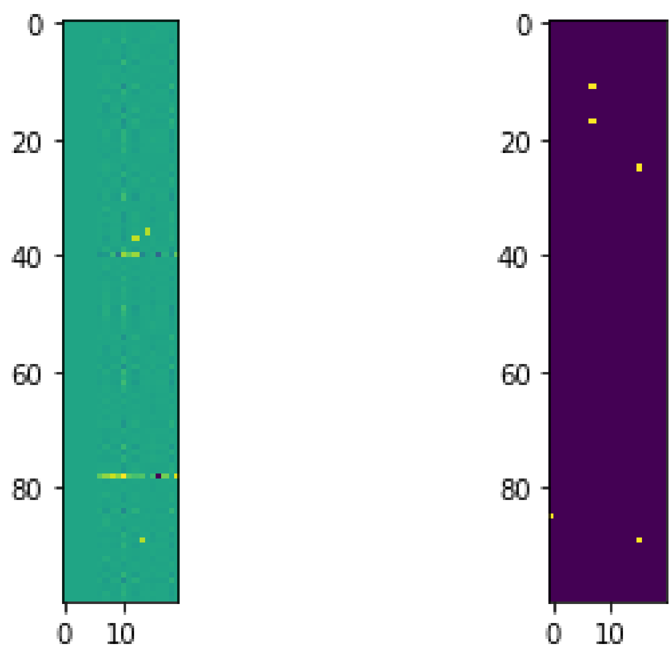
[249]:
```
# But we can exactly reconstruct X
X_Decompressed = f.decompress(X_Compressed)

# and gets the training X exactly
np.linalg.norm(X_Test - X_Decompressed)
```

[249]:
```
46.84091857456668
```

Errors appear all over the place because of the anomaly in the training data.

[250]:
```
_, ax = py.subplots(ncols=2)
ax[0].imshow(X_Test - X_Decompressed)
ax[1].imshow(S[:100, :])
```

[250]:
```
<matplotlib.image.AxesImage at 0x7fd7f25529d0>
```



6 RPCA parameters for this data size

Now we start using RPCA. Here we need to find appropriate parameters for this data size 200×20.

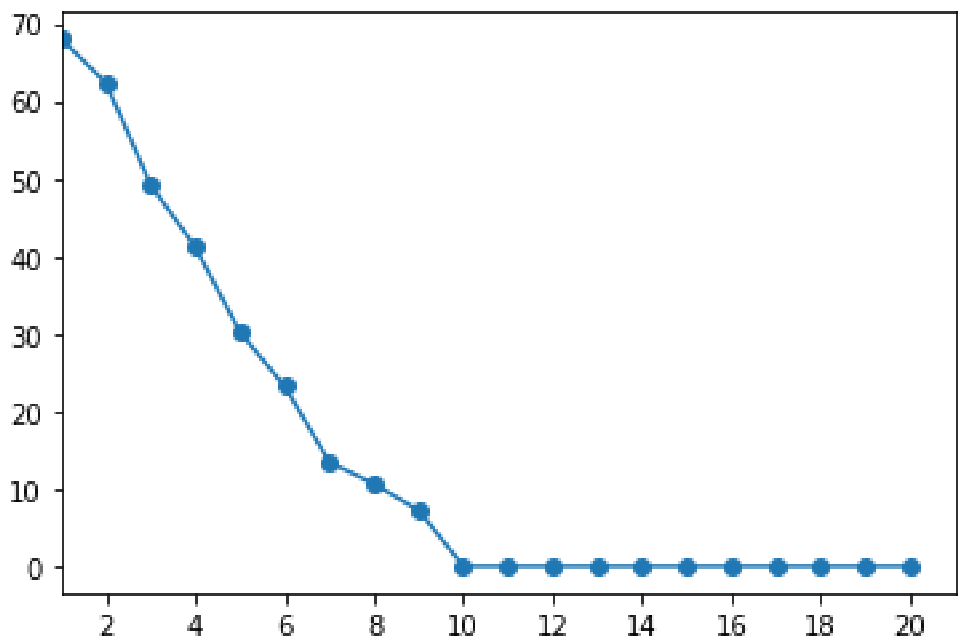
[251]:
```
# The parameters of our calcuation
N = 200
n = 20
K = 6
```

[252]:
```
# Here is some data
np.random.seed(1234)
X,L,S,_ = generateData(N=N, n=n, K=K, numAnoms=10, sizeAnoms=10)
X_Train = X[:100,:]
X_Test = X[100:,:]
```

[253]:
```
def getRank(E):
    for i in range(len(E)):
        if E[i] < 1e-3:
            return i
```

With anomalies the matrix is not low rank anymore

[254]:
```
# that is exactly low rank
U,E,VT = np.linalg.svd(X_Train)
# Make ticks at integers
EPlot(E, py.gca())
```



But we can run it through RPCA!

6.1 Basic setup

Gets the answer close to right but does not converge very well

Note, vecE being 0 is a bad thing! I need to look at my code carefully to know what to do, but epsilon1 gets used to get vecE if it is zero!

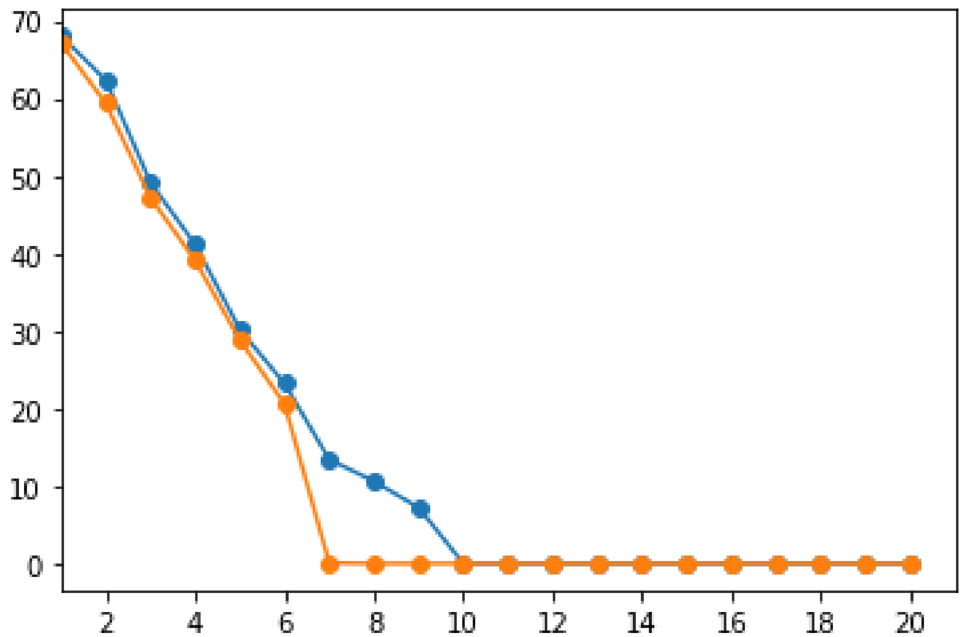
With there parameters, the rank is correct, but the convergence is slow.

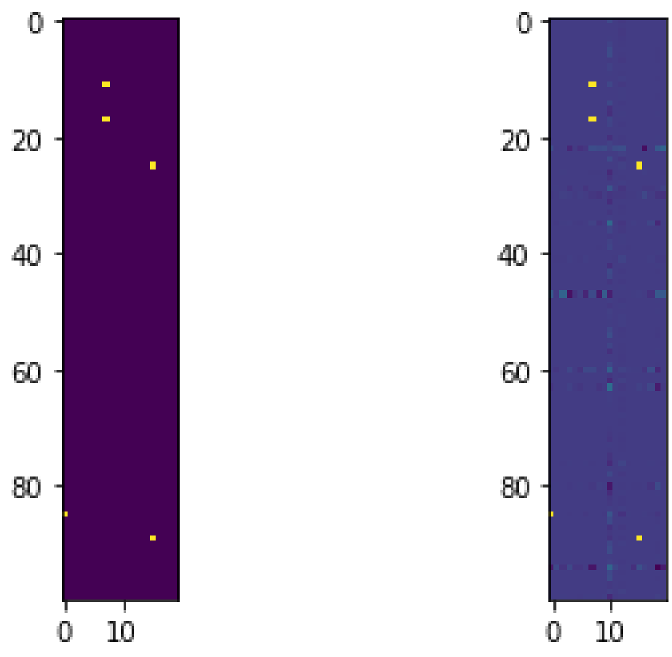
[255]:
```python
lam1 = 1. / np.sqrt(np.max([100, 20]))
U_RPCA,E_RPCA,VT_RPCA,S_RPCA = RPCA(X_Train, X_Train*0.0001, maxIteration=200,
    lam=lam1)
EPlot(E, py.gca())
EPlot(E_RPCA, py.gca())
print('rank',getRank(E_RPCA))
_,ax = py.subplots(ncols = 2)
ax[0].imshow(S[:100])
ax[1].imshow(S_RPCA.todense()[:100])
py.imshow
```

```
criterion1 is the constraint
criterion2 is the solution
iteration criterion1 epsilon1 criterion2 epsilon2 rho      mu
       10   2.59e-02 1.00e-05   2.37e-04 1.00e-04 3.08e+00 4.52e-02
       20   6.57e-03 1.00e-05   3.77e-04 1.00e-04 3.08e+00 1.39e-01
       30   2.18e-03 1.00e-05   3.10e-04 1.00e-04 3.08e+00 4.28e-01
       40   6.75e-04 1.00e-05   2.49e-04 1.00e-04 3.08e+00 1.32e+00
       50   2.57e-04 1.00e-05   1.00e-04 1.00e-04 3.08e+00 1.32e+00
       60   8.81e-05 1.00e-05   6.57e-05 1.00e-04 3.08e+00 4.05e+00
       70   3.18e-05 1.00e-05   8.25e-05 1.00e-04 3.08e+00 1.24e+01
       80   2.33e-05 1.00e-05   8.39e-05 1.00e-04 3.08e+00 1.24e+01
       90   2.14e-05 1.00e-05   9.11e-05 1.00e-04 3.08e+00 1.24e+01
      100   1.93e-05 1.00e-05   8.90e-05 1.00e-04 3.08e+00 1.24e+01
      110   2.00e-05 1.00e-05   8.61e-05 1.00e-04 3.08e+00 1.24e+01
      120   1.83e-05 1.00e-05   9.65e-05 1.00e-04 3.08e+00 1.24e+01
      130   1.86e-05 1.00e-05   1.03e-04 1.00e-04 3.08e+00 1.24e+01
      140   1.71e-05 1.00e-05   9.21e-05 1.00e-04 3.08e+00 1.24e+01
      150   1.82e-05 1.00e-05   1.08e-04 1.00e-04 3.08e+00 1.24e+01
      160   1.70e-05 1.00e-05   1.00e-04 1.00e-04 3.08e+00 1.24e+01
      170   1.73e-05 1.00e-05   1.04e-04 1.00e-04 3.08e+00 1.24e+01
      180   1.64e-05 1.00e-05   9.87e-05 1.00e-04 3.08e+00 1.24e+01
      190   1.90e-05 1.00e-05   1.10e-04 1.00e-04 3.08e+00 1.24e+01
      200   2.09e-05 1.00e-05   1.23e-04 1.00e-04 3.08e+00 1.24e+01
rank 7
```

[255]:
```
<function matplotlib.pyplot.imshow(X, cmap=None, norm=None, aspect=None,
interpolation=None, alpha=None, vmin=None, vmax=None, origin=None, extent=None,
shape=<deprecated parameter>, filternorm=1, filterrad=4.0, imlim=<deprecated
```

```
parameter>, resample=None, url=None, *, data=None, **kwargs)>
```





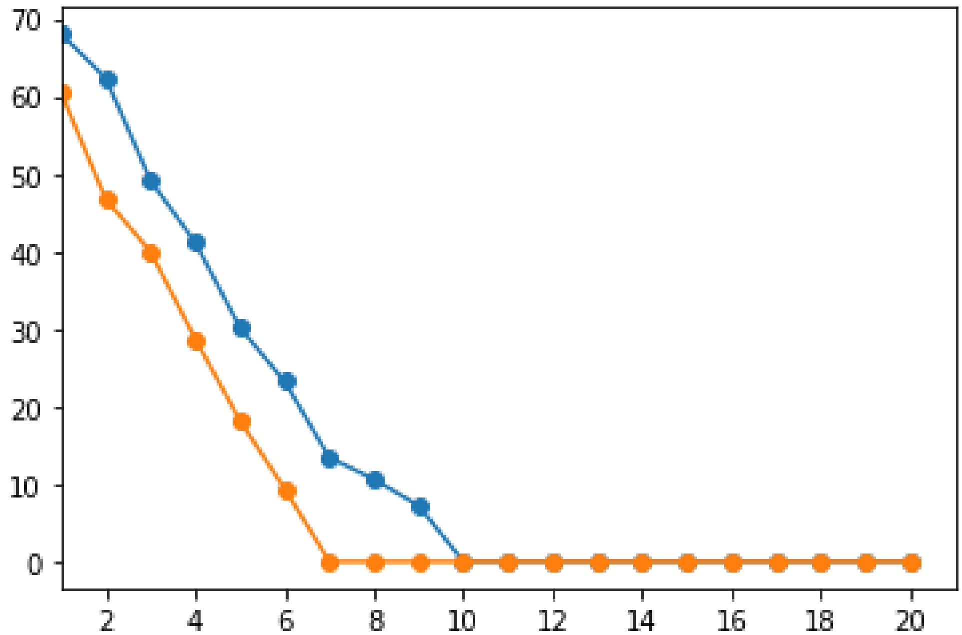
6.2 Custom λ

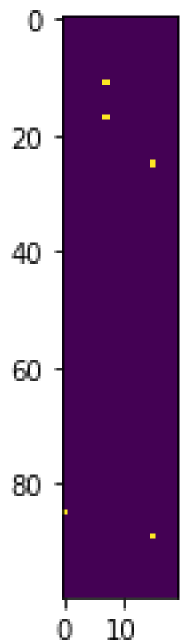
Good convergerence and correct answer

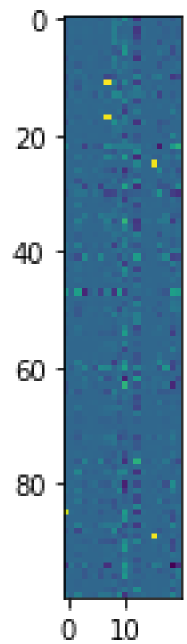
[256]:
```
lam1 = 1. / np.sqrt(np.max([100, 20]))
U_RPCA,E_RPCA,VT_RPCA,S_RPCA = RPCA(X_Train, X_Train*0.0001, maxIteration=200,␣
↪lam=0.8*lam1)
EPlot(E, py.gca())
EPlot(E_RPCA, py.gca())
print('rank',getRank(E_RPCA))
_,ax = py.subplots(ncols = 2)
ax[0].imshow(S[:100])
ax[1].imshow(S_RPCA.todense()[:100])
py.imshow
```

```
criterion1 is the constraint
criterion2 is the solution
iteration criterion1 epsilon1 criterion2 epsilon2 rho      mu
       10   3.91e-02 1.00e-05   2.63e-04 1.00e-04 3.08e+00 1.47e-02
       20   1.14e-02 1.00e-05   1.42e-04 1.00e-04 3.08e+00 4.52e-02
       30   3.75e-03 1.00e-05   2.03e-04 1.00e-04 3.08e+00 1.39e-01
       40   1.14e-03 1.00e-05   2.24e-04 1.00e-04 3.08e+00 4.28e-01
       50   3.11e-04 1.00e-05   1.48e-04 1.00e-04 3.08e+00 1.32e+00
       60   1.04e-04 1.00e-05   8.46e-05 1.00e-04 3.08e+00 4.05e+00
       70   5.77e-05 1.00e-05   7.47e-05 1.00e-04 3.08e+00 1.24e+01
       80   3.27e-05 1.00e-05   3.67e-05 1.00e-04 3.08e+00 1.24e+01
       90   2.03e-05 1.00e-05   8.24e-05 1.00e-04 3.08e+00 3.83e+01
      100   1.42e-05 1.00e-05   8.11e-05 1.00e-04 3.08e+00 3.83e+01
      110   1.32e-05 1.00e-05   7.20e-05 1.00e-04 3.08e+00 3.83e+01
      114   9.53e-06 1.00e-05   6.05e-05 1.00e-04 3.08e+00 3.83e+01
rank 6
```

[256]:
```
<function matplotlib.pyplot.imshow(X, cmap=None, norm=None, aspect=None,
interpolation=None, alpha=None, vmin=None, vmax=None, origin=None, extent=None,
shape=<deprecated parameter>, filternorm=1, filterrad=4.0, imlim=<deprecated
parameter>, resample=None, url=None, *, data=None, **kwargs)>
```







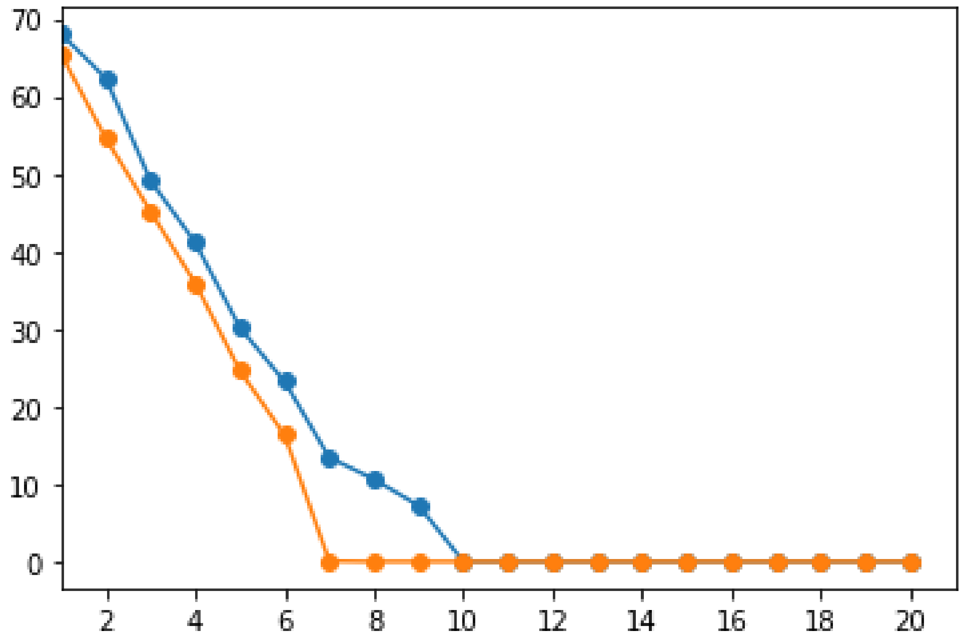
Good convergence, but not quite right answer

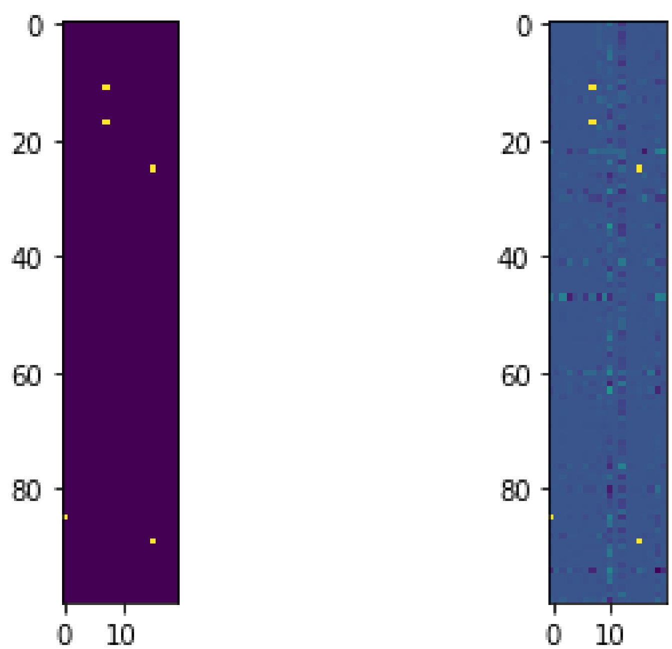
[257]:
```
lam1 = 1. / np.sqrt(np.max([100, 20]))
```

```
U_RPCA,E_RPCA,VT_RPCA,S_RPCA = RPCA(X_Train, X_Train*0.0001, maxIteration=200,␣
 ↪lam=0.9*lam1)
EPlot(E, py.gca())
EPlot(E_RPCA, py.gca())
print('rank',getRank(E_RPCA))
_,ax = py.subplots(ncols = 2)
ax[0].imshow(S[:100])
ax[1].imshow(S_RPCA.todense()[:100])
py.imshow
```

```
criterion1 is the constraint
criterion2 is the solution
iteration criterion1 epsilon1 criterion2 epsilon2 rho      mu
       10   3.31e-02 1.00e-05   1.28e-04 1.00e-04 3.08e+00 1.47e-02
       20   9.50e-03 1.00e-05   1.12e-04 1.00e-04 3.08e+00 4.52e-02
       30   3.12e-03 1.00e-05   1.58e-04 1.00e-04 3.08e+00 1.39e-01
       40   1.14e-03 1.00e-05   1.58e-04 1.00e-04 3.08e+00 4.28e-01
       50   4.28e-04 1.00e-05   1.88e-04 1.00e-04 3.08e+00 1.32e+00
       60   2.03e-04 1.00e-05   1.61e-04 1.00e-04 3.08e+00 4.05e+00
       70   9.39e-05 1.00e-05   5.44e-05 1.00e-04 3.08e+00 4.05e+00
       80   3.31e-05 1.00e-05   6.94e-05 1.00e-04 3.08e+00 1.24e+01
       90   2.55e-05 1.00e-05   3.10e-05 1.00e-04 3.08e+00 1.24e+01
      100   1.66e-05 1.00e-05   3.46e-05 1.00e-04 3.08e+00 1.24e+01
      110   1.91e-05 1.00e-05   9.06e-05 1.00e-04 3.08e+00 3.83e+01
      120   1.74e-05 1.00e-05   1.08e-04 1.00e-04 3.08e+00 3.83e+01
      123   9.59e-06 1.00e-05   7.80e-05 1.00e-04 3.08e+00 3.83e+01
rank 7
```

[257]:
```
<function matplotlib.pyplot.imshow(X, cmap=None, norm=None, aspect=None,
interpolation=None, alpha=None, vmin=None, vmax=None, origin=None, extent=None,
shape=<deprecated parameter>, filternorm=1, filterrad=4.0, imlim=<deprecated
parameter>, resample=None, url=None, *, data=None, **kwargs)>
```





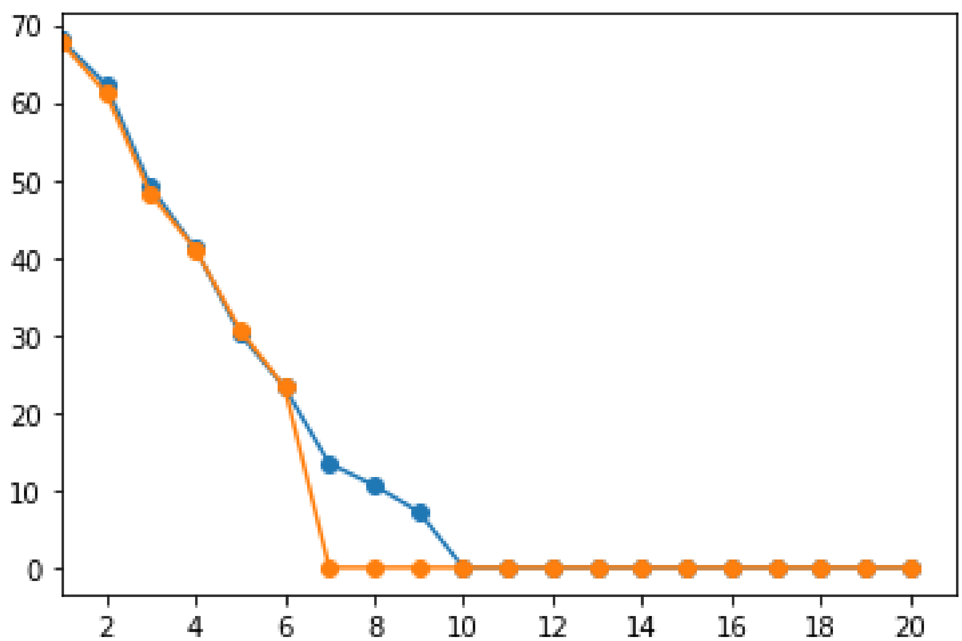
Good convergence and good answer

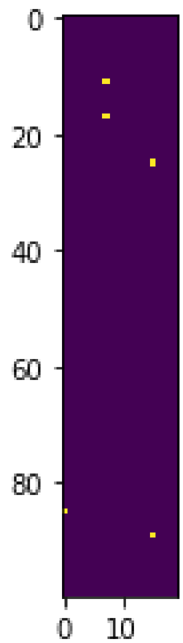
```
[258]: lam1 = 1. / np.sqrt(np.max([100, 20]))
```

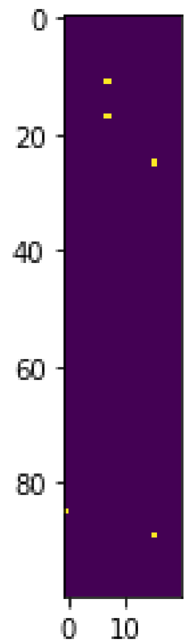
```
U_RPCA,E_RPCA,VT_RPCA,S_RPCA = RPCA(X_Train, X_Train*0.0001, maxIteration=200,␣
 ↪lam=1.5*lam1)
EPlot(E, py.gca())
EPlot(E_RPCA, py.gca())
print('rank',getRank(E_RPCA))
_,ax = py.subplots(ncols = 2)
ax[0].imshow(S[:100])
ax[1].imshow(S_RPCA.todense()[:100])
py.imshow
```

```
criterion1 is the constraint
criterion2 is the solution
iteration criterion1 epsilon1 criterion2 epsilon2 rho      mu
       10   2.94e-03 1.00e-05   1.96e-04 1.00e-04 3.08e+00 1.39e-01
       20   7.57e-04 1.00e-05   3.25e-04 1.00e-04 3.08e+00 4.28e-01
       30   4.13e-05 1.00e-05   2.08e-04 1.00e-04 3.08e+00 1.24e+01
       40   9.81e-06 1.00e-05   5.40e-05 1.00e-04 3.08e+00 1.24e+01
       40   9.81e-06 1.00e-05   5.40e-05 1.00e-04 3.08e+00 1.24e+01
rank 6
```

[258]:
```
<function matplotlib.pyplot.imshow(X, cmap=None, norm=None, aspect=None,
interpolation=None, alpha=None, vmin=None, vmax=None, origin=None, extent=None,
shape=<deprecated parameter>, filternorm=1, filterrad=4.0, imlim=<deprecated
parameter>, resample=None, url=None, *, data=None, **kwargs)>
```







7 RPCA detection rates of anomalies in training data

[259]:
```
def runRPCAAnomalyTest(N, n, K, numAnoms, sizeAnoms):
    # Here is some data
    np.random.seed(1234)
    X,L,S,_ = generateData(N=N, n=n, K=K, numAnoms=numAnoms,␣
↪sizeAnoms=sizeAnoms)
    X_Train = X[:100,:]
    #X_Test = X[100:,:]

    print('size anomalies: %f min of L: %f max of L: %f'%(sizeAnoms, np.min(np.
↪abs(L)), np.max(np.abs(L))))

    # Solve using RPCA
    lam1 = 1. / np.sqrt(np.max([100, 20]))
    U_RPCA,E_RPCA,VT_RPCA,S_RPCA = RPCA(X_Train, X_Train*0.0001,␣
↪maxIteration=200, lam=1.5*lam1)
    S_RPCA = S_RPCA.todense()

    # This is a 0,1 mask on where the anomalies are
    Omega = S != 0

    # A threshold for detecting an anomaly
```

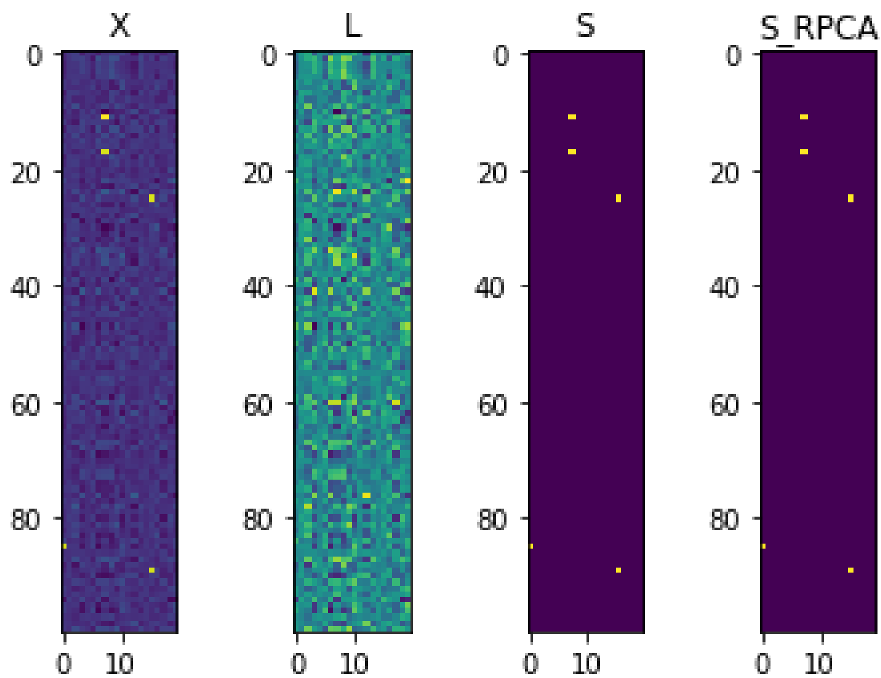
```
    alpha = 1e-2

    _, ax = py.subplots(ncols=4)
    ax[0].imshow(X[:100, :])
    ax[0].set_title('X')
    ax[1].imshow(L[:100, :])
    ax[1].set_title('L')
    ax[2].imshow(S[:100, :])
    ax[2].set_title('S')
    ax[3].imshow(S_RPCA)
    ax[3].set_title('S_RPCA')

    # Compute confusion matrix
    TP = 0
    TN = 0
    FP = 0
    FN = 0
    for i in range(S[:100, :].shape[0]):
        for j in range(S.shape[1]):
            # A true negative is we think there is no anomaly where there is␣
↪not one
            if S[:100, :][i, j] == 0 and np.abs(S_RPCA[i, j]) < alpha:
                TN += 1
            # A true positive is we think there is an anomaly where there is one
            if S[:100, :][i, j] != 0 and np.abs(S_RPCA[i, j]) > alpha:
                TP += 1
            # A false negative is we think there is no anomaly where there is␣
↪one
            if S[:100, :][i, j] != 0 and np.abs(S_RPCA[i, j]) < alpha:
                FN += 1
            # A false positive is we think there is an anomaly where there is␣
↪not one
            if S[:100, :][i, j] == 0 and np.abs(S_RPCA[i, j]) > alpha:
                FP += 1

    print('confusion matrix')
    print('----------------')
    print('%6d %6d'%(TP, FP))
    print('%6d %6d'%(FN, TN))
```

With a few large anomalies, the problem is very easy.

[260]:
```
N = 200
n = 20
K = 6
numAnoms = 10
sizeAnoms = 50
```

```
runRPCAAnomalyTest(N, n, K, numAnoms, sizeAnoms)
```

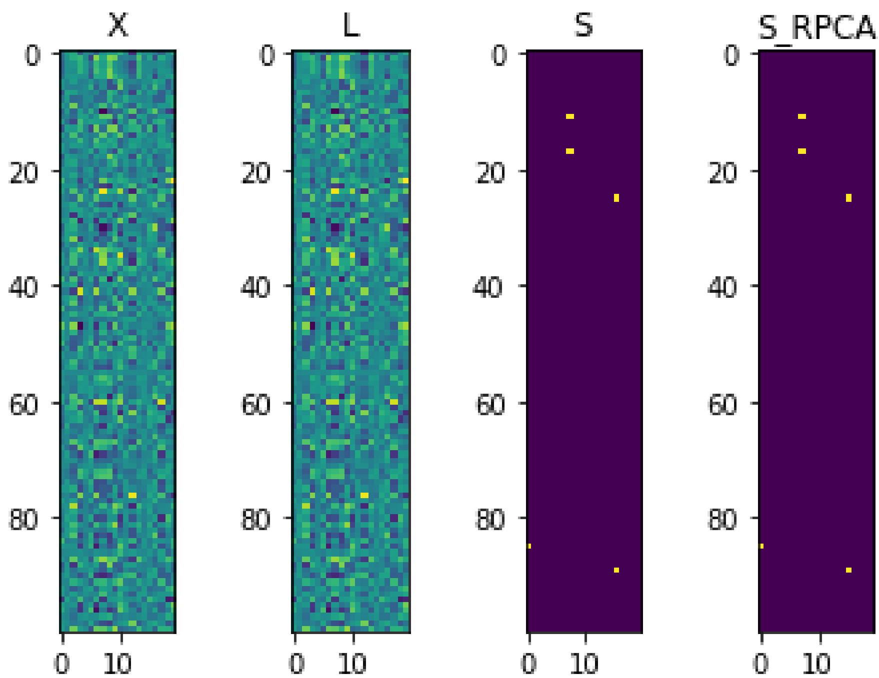
```
size anomalies: 50.000000 min of L: 0.000770 max of L: 10.798646

criterion1 is the constraint
criterion2 is the solution
iteration criterion1 epsilon1 criterion2 epsilon2 rho      mu
       10   3.69e-03 1.00e-05   2.09e-04 1.00e-04 3.08e+00 1.07e-01
       20   2.66e-04 1.00e-05   1.26e-04 1.00e-04 3.08e+00 1.01e+00
       30   2.63e-05 1.00e-05   1.16e-04 1.00e-04 3.08e+00 9.59e+00
       40   1.22e-05 1.00e-05   1.13e-04 1.00e-04 3.08e+00 2.95e+01
       41   8.16e-06 1.00e-05   7.65e-05 1.00e-04 3.08e+00 2.95e+01
confusion matrix
----------------
     5      0
     0   1995
```



Even when making the anomalies smaller the algorithm still works fine.

[261]:
```
N = 200
n = 20
K = 6
numAnoms = 10
sizeAnoms = 1
runRPCAAnomalyTest(N, n, K, numAnoms, sizeAnoms)
```

```
size anomalies: 1.000000 min of L: 0.000770 max of L: 10.798646

criterion1 is the constraint
criterion2 is the solution
iteration criterion1 epsilon1 criterion2 epsilon2 rho      mu
       10   2.03e-03 1.00e-05   1.27e-04 1.00e-04 3.08e+00 1.40e-01
       20   1.72e-04 1.00e-05   1.78e-04 1.00e-04 3.08e+00 1.32e+00
       30   1.67e-05 1.00e-05   1.18e-04 1.00e-04 3.08e+00 1.25e+01
       36   9.90e-06 1.00e-05   3.21e-05 1.00e-04 3.08e+00 1.25e+01
confusion matrix
----------------
     5      0
     0   1995
```

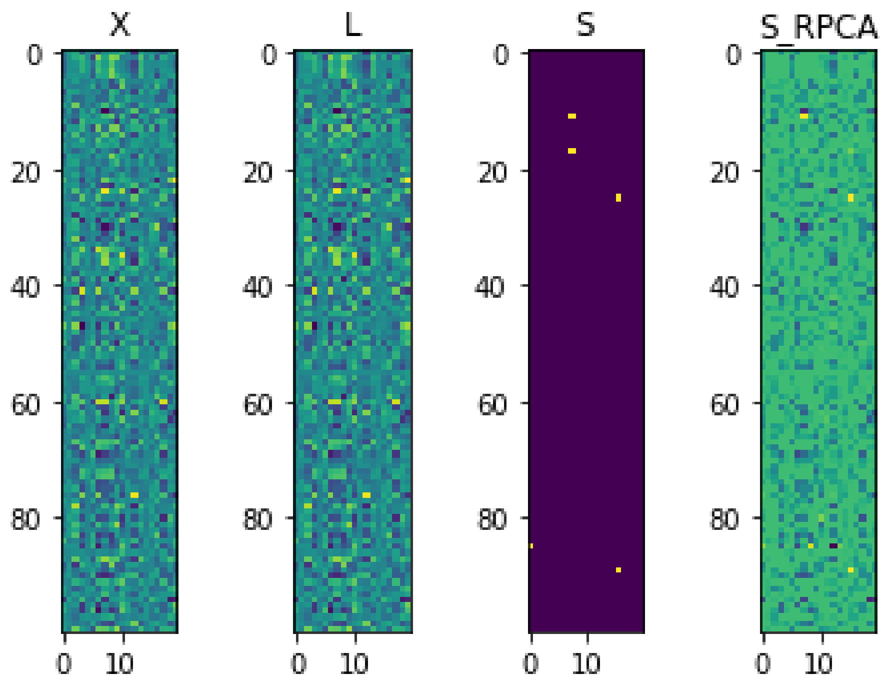


There is a limit though, if the anomalies are too small then RPCA misses them.

[262]:
```
N = 200
n = 20
K = 6
numAnoms = 10
sizeAnoms = 0.001
runRPCAAnomalyTest(N, n, K, numAnoms, sizeAnoms)
```

```
size anomalies: 0.001000 min of L: 0.000770 max of L: 10.798646
```

```
criterion1 is the constraint
criterion2 is the solution
iteration criterion1 epsilon1 criterion2 epsilon2 rho      mu
       10   1.87e-03 1.00e-05   2.15e-04 1.00e-04 3.08e+00 4.30e-01
       20   1.20e-04 1.00e-05   8.12e-05 1.00e-04 3.08e+00 1.32e+00
       30   1.95e-05 1.00e-05   3.85e-05 1.00e-04 3.08e+00 1.25e+01
       40   1.40e-05 1.00e-05   4.68e-05 1.00e-04 3.08e+00 1.25e+01
       46   9.24e-06 1.00e-05   4.75e-05 1.00e-04 3.08e+00 3.85e+01
confusion matrix
----------------
     0      0
     5   1995
```

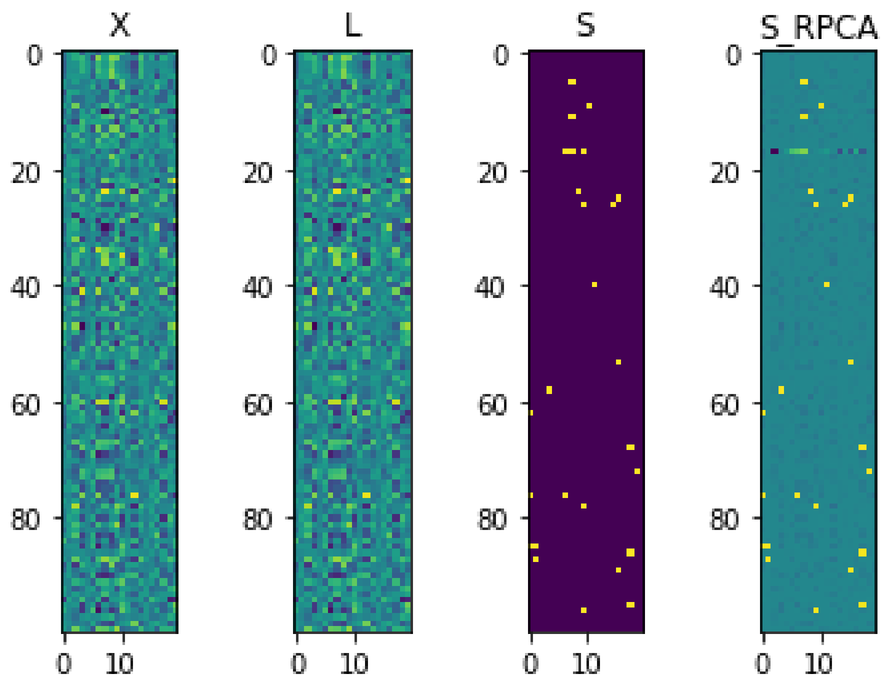


Given more anomalies it still finds them.

[263]:
```
N = 200
n = 20
K = 6
numAnoms = 50
sizeAnoms = 0.01
runRPCAAnomalyTest(N, n, K, numAnoms, sizeAnoms)
```

```
size anomalies: 0.010000 min of L: 0.000770 max of L: 10.798646

criterion1 is the constraint
criterion2 is the solution
```

```
iteration criterion1 epsilon1 criterion2 epsilon2 rho      mu
       10   1.90e-03 1.00e-05   2.15e-04 1.00e-04 3.08e+00 4.30e-01
       20   3.67e-04 1.00e-05   8.01e-05 1.00e-04 3.08e+00 1.32e+00
       30   9.23e-05 1.00e-05   1.12e-04 1.00e-04 3.08e+00 4.07e+00
       40   3.28e-05 1.00e-05   1.76e-04 1.00e-04 3.08e+00 1.25e+01
       50   3.80e-05 1.00e-05   2.31e-04 1.00e-04 3.08e+00 3.85e+01
       58   8.39e-06 1.00e-05   9.32e-05 1.00e-04 3.08e+00 3.85e+01
confusion matrix
----------------
     8      0
    18   1974
```

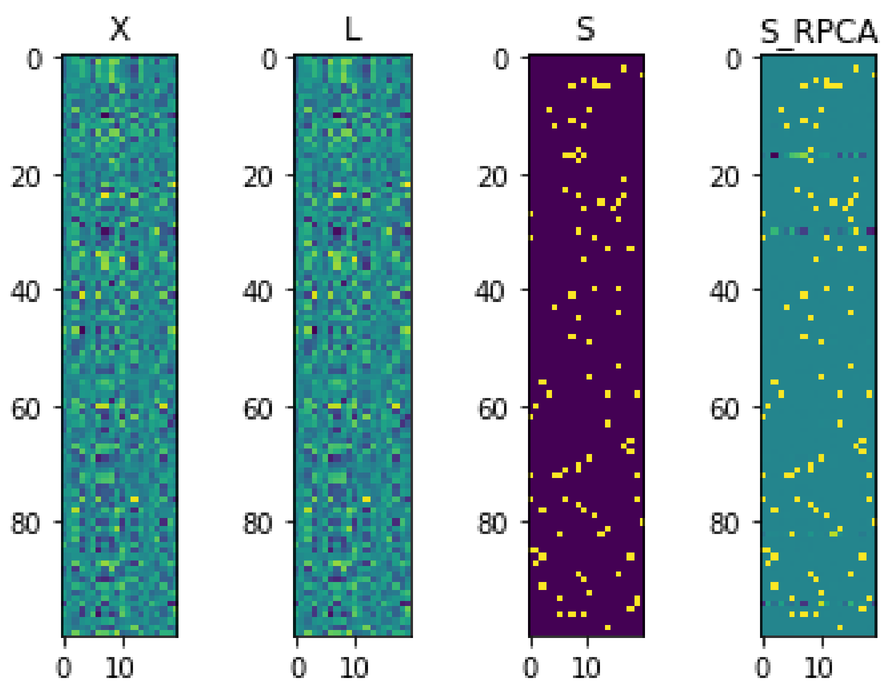


It works suprisingly well, even with many small anomalies!

[264]:
```
N = 200
n = 20
K = 6
numAnoms = 200
sizeAnoms = 0.1
runRPCAAnomalyTest(N, n, K, numAnoms, sizeAnoms)
```

```
size anomalies: 0.100000 min of L: 0.000770 max of L: 10.798646

criterion1 is the constraint
criterion2 is the solution
```

```
iteration criterion1 epsilon1 criterion2 epsilon2 rho      mu
       10   6.63e-03 1.00e-05   2.78e-04 1.00e-04 3.08e+00 4.30e-01
       20   1.29e-03 1.00e-05   2.83e-04 1.00e-04 3.08e+00 4.30e-01
       30   5.29e-04 1.00e-05   2.38e-04 1.00e-04 3.08e+00 1.32e+00
       40   1.47e-04 1.00e-05   1.60e-04 1.00e-04 3.08e+00 4.07e+00
       50   4.81e-05 1.00e-05   9.96e-05 1.00e-04 3.08e+00 1.25e+01
       60   2.43e-05 1.00e-05   1.13e-04 1.00e-04 3.08e+00 1.25e+01
       70   2.64e-05 1.00e-05   1.16e-04 1.00e-04 3.08e+00 1.25e+01
       80   2.23e-05 1.00e-05   7.64e-05 1.00e-04 3.08e+00 1.25e+01
       90   1.26e-05 1.00e-05   5.40e-05 1.00e-04 3.08e+00 1.25e+01
      100   1.54e-05 1.00e-05   9.26e-05 1.00e-04 3.08e+00 1.25e+01
      110   1.00e-05 1.00e-05   5.23e-05 1.00e-04 3.08e+00 1.25e+01
      120   1.19e-05 1.00e-05   6.06e-05 1.00e-04 3.08e+00 1.25e+01
      130   1.25e-05 1.00e-05   6.02e-05 1.00e-04 3.08e+00 1.25e+01
      140   1.40e-05 1.00e-05   6.78e-05 1.00e-04 3.08e+00 1.25e+01
      145   9.44e-06 1.00e-05   5.84e-05 1.00e-04 3.08e+00 1.25e+01
confusion matrix
----------------
    88      34
     3    1875
```

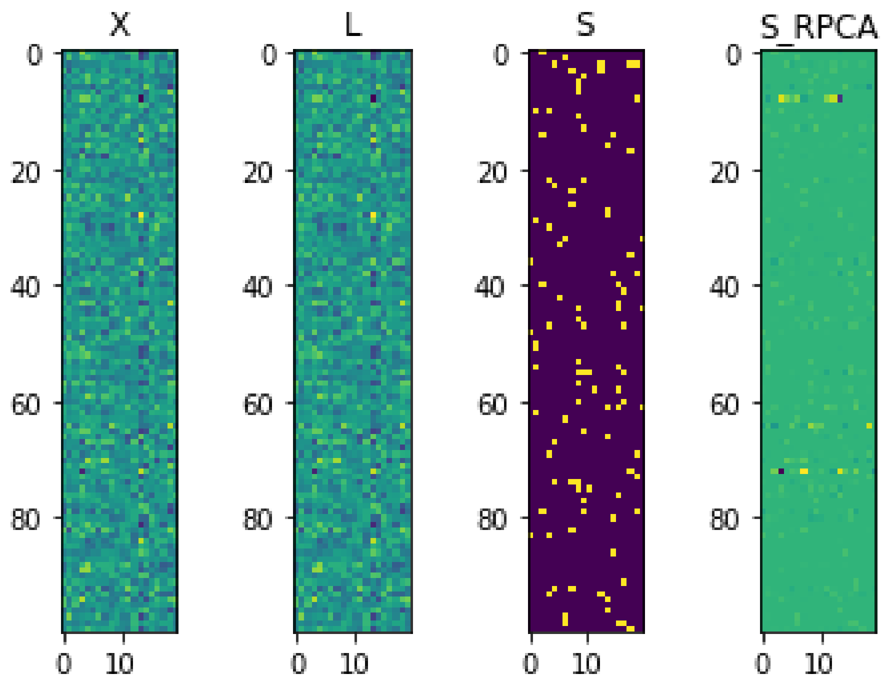


But if you make the rank too large it fails.

[265]:
```
N = 200
n = 20
```

```
K = 10
numAnoms = 200
sizeAnoms = 0.1
runRPCAAnomalyTest(N, n, K, numAnoms, sizeAnoms)
```

```
size anomalies: 0.100000 min of L: 0.000486 max of L: 15.765333

criterion1 is the constraint
criterion2 is the solution
iteration criterion1 epsilon1 criterion2 epsilon2 rho      mu
       10   1.64e-02 1.00e-05   2.19e-04 1.00e-04 3.08e+00 3.94e-02
       20   4.83e-03 1.00e-05   1.14e-04 1.00e-04 3.08e+00 1.21e-01
       30   2.16e-03 1.00e-05   3.23e-04 1.00e-04 3.08e+00 3.73e-01
       40   1.02e-03 1.00e-05   1.10e-04 1.00e-04 3.08e+00 3.73e-01
       50   4.23e-04 1.00e-05   1.31e-04 1.00e-04 3.08e+00 1.15e+00
       60   1.38e-04 1.00e-05   1.37e-04 1.00e-04 3.08e+00 3.53e+00
       70   5.04e-05 1.00e-05   1.61e-04 1.00e-04 3.08e+00 1.08e+01
       80   2.60e-05 1.00e-05   1.21e-04 1.00e-04 3.08e+00 1.08e+01
       90   2.51e-05 1.00e-05   1.36e-04 1.00e-04 3.08e+00 1.08e+01
      100   2.69e-05 1.00e-05   1.68e-04 1.00e-04 3.08e+00 1.08e+01
      110   2.38e-05 1.00e-05   1.45e-04 1.00e-04 3.08e+00 1.08e+01
      120   2.48e-05 1.00e-05   1.55e-04 1.00e-04 3.08e+00 1.08e+01
      130   2.56e-05 1.00e-05   1.49e-04 1.00e-04 3.08e+00 1.08e+01
      140   2.25e-05 1.00e-05   1.44e-04 1.00e-04 3.08e+00 1.08e+01
      150   2.47e-05 1.00e-05   1.47e-04 1.00e-04 3.08e+00 1.08e+01
      160   2.44e-05 1.00e-05   1.48e-04 1.00e-04 3.08e+00 1.08e+01
      170   2.43e-05 1.00e-05   1.37e-04 1.00e-04 3.08e+00 1.08e+01
      180   2.32e-05 1.00e-05   1.40e-04 1.00e-04 3.08e+00 1.08e+01
      190   2.42e-05 1.00e-05   1.43e-04 1.00e-04 3.08e+00 1.08e+01
      200   2.26e-05 1.00e-05   1.32e-04 1.00e-04 3.08e+00 1.08e+01
confusion matrix
----------------
    77    213
    26   1684
```



8 RPCA training and testing

Now we implement the Servi metrics.

- We do a confusion matrix for how many true anomalies we detect in the testing data.
- We do RMSE on the rest of the entries.

[266]:
```
def runFullTrainingAndTesting(N, n, K, noiseSize, numAnoms, sizeAnoms):
    # Here is some data
    np.random.seed(1235)
    X,L,S,_ = generateData(N=N, n=n, K=K, noiseSize=noiseSize,␣
↪numAnoms=numAnoms, sizeAnoms=sizeAnoms)
    X_Train = X[:100,:]
    X_Test = X[100:,:]
    L_Test = L[100:,:]
    S_Test = S[100:,:]

    print('size anomalies: %f min of L: %f max of L: %f'%(sizeAnoms, np.min(np.
↪abs(L)), np.max(np.abs(L))))

    print('X, L, and S used for training')
    _, ax = py.subplots(ncols=3)
    ax[0].imshow(X[:100,:])
```

```
    ax[0].set_title('X')
    ax[1].imshow(L[:100,:])
    ax[1].set_title('L')
    ax[2].imshow(S[:100,:])
    ax[2].set_title('S')
    py.show()

    # Solve using RPCA
    lam1 = 1. / np.sqrt(np.max([100, 20]))
    U_RPCA,E_RPCA,VT_RPCA,S_RPCA = RPCA(X_Train, X_Train*0.0001,␣
↪maxIteration=200, lam=1.5*lam1)
    S_RPCA = S_RPCA.todense()
    L_RPCA = U_RPCA @ np.diag(E_RPCA) @ VT_RPCA

    print('L recovery error',np.linalg.norm(L[:100,:]-L_RPCA, 'fro'))

    print('L and S returned by RPCA, and X-L')
    _, ax = py.subplots(ncols=3)
    ax[0].imshow(L_RPCA)
    ax[0].set_title('L_RPCA')
    ax[1].imshow(S_RPCA)
    ax[1].set_title('S_RPCA')
    ax[2].imshow(X_Train-L_RPCA)
    ax[2].set_title('X_Train-L_RPCA')
    py.show()

    # Select the columns that we keep for predicting the rest.  Without RPCA we␣
↪are stuck with just picking arbitrary columns.
    Selection = np.zeros([20,6])
    for i in range(6):
        Selection[i, i] = 1

    # These are the columns in X_Train-L_RPCA that are "clean"
    Selection_clean = np.zeros([20,6])
    S_Train = X_Train-L_RPCA
    j = 0
    for i in range(20):
        S_Train_column_size = np.linalg.norm(S_Train[:, i])
        print(i, S_Train_column_size)
        if S_Train_column_size < 0.1:
            Selection_clean[i, j] = 1
            j += 1
            if j >= 6:
                break

    # Create a projector using PCA and sparse PCA on the bad data in X
    f_PCA_bad = PCACompress(X_Train, dim=K)
```

```
f_SPCA_bad = SPCACompress(X_Train, Selection, dim=K)

# Create a projector using PCA and sparse PCA on the good data in L
f_PCA_good = PCACompress(L_RPCA, dim=K)
f_SPCA_good = SPCACompress(L_RPCA, Selection_clean, dim=K)

def evaluate(L, S, L_test, S_test, title, Selection=None):
    print('--------%s--------'%(title,))
    print("RMSE on L",np.linalg.norm(L-L_test,'fro'))
    # This is a 0,1 mask on where the anomalies are
    Omega = (S != 0)

    # A threshold for detecting an anomaly
    alpha = 1e-2

    Selection_columns = np.zeros(S_test.shape)
    S_mask = np.ones(S.shape)
    if not Selection is None:
        Selected_columns = set()
        # If we have selected some columns, then we just lighten them
        # for visualization
        for i in range(Selection.shape[0]):
            for j in range(Selection.shape[1]):
                if Selection[i, j] == 1:
                    Selected_columns.add(i)
                    Selection_columns[:, i] += 2.0
        # Also get rid of those entries in S_true since they shouldn't count
        for i in range(S.shape[0]):
            for j in range(S.shape[1]):
                # print(i, j, Selected_columns, j in Selected_columns,␣
↪S[i,j])
                if j in Selected_columns and np.abs(S_test[i, j]) > 0:
                    S_mask[i, :] *= np.nan

    _, ax = py.subplots(ncols=2, figsize=(10, 5))
    ax[0].imshow(S_test+Selection_columns, aspect='auto')
    ax[0].set_title('S true')
    ax[1].imshow(S, aspect='auto')
    ax[1].set_title('S computed')
    py.show()

    if not Selection is None:
        _, ax = py.subplots(ncols=2, figsize=(10, 5))
        ax[0].imshow(S_test+Selection_columns, aspect='auto')
        ax[0].set_title('S true')
        ax[1].imshow(np.multiply(S, S_mask), aspect='auto')
        ax[1].set_title('S computed')
```

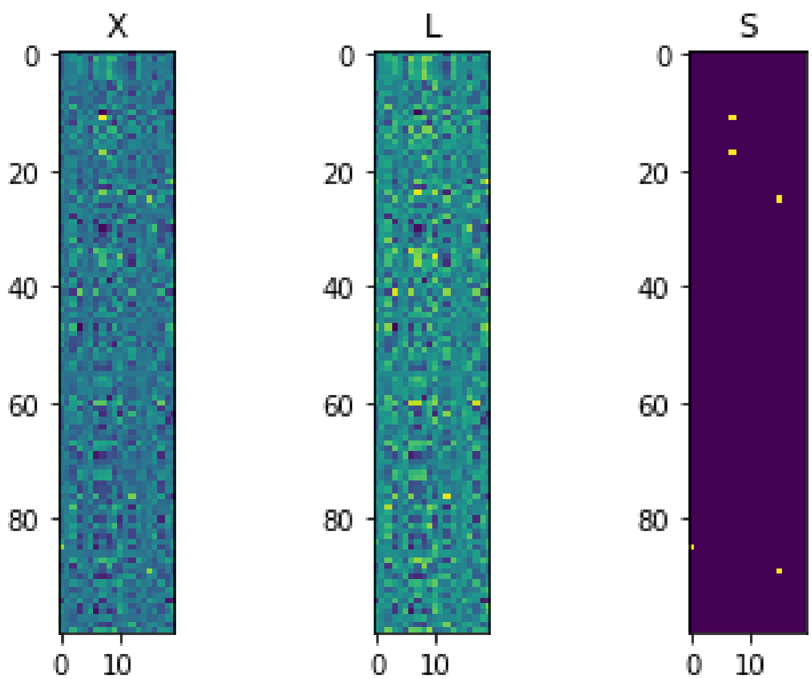
```
            py.show()

        # Compute confusion matrix
        TP = 0
        TN = 0
        FP = 0
        FN = 0
        for i in range(S[:100, :].shape[0]):
            for j in range(S.shape[1]):
                # A true negative is we think there is no anomaly where there␣
↪is not one
                if S_test[i, j] == 0 and np.abs(S[i, j]*S_mask[i, j]) < alpha:
                    TN += 1
                # A true positive is we think there is an anomaly where there␣
↪is one
                if S_test[i, j] != 0 and np.abs(S[i, j]*S_mask[i, j]) > alpha:
                    TP += 1
                # A false negative is we think there is no anomaly where there␣
↪is one
                if S_test[i, j] != 0 and np.abs(S[i, j]*S_mask[i, j]) < alpha:
                    FN += 1
                # A false positive is we think there is an anomaly where there␣
↪is not one
                if S_test[i, j] == 0 and np.abs(S[i, j]*S_mask[i, j]) > alpha:
                    FP += 1

        print('confusion matrix on S')
        print('----------------')
        print('%6d %6d'%(TP, FP))
        print('%6d %6d'%(FN, TN))

    L_tmp = f_PCA_bad.decompress(f_PCA_bad.compress(X_Test))
    evaluate(L_tmp, X_Test-L_tmp, L_Test, S_Test, "PCA prediction without RPCA␣
↪first")

    L_tmp = f_SPCA_bad.decompress(f_SPCA_bad.compress(X_Test))
    evaluate(L_tmp, X_Test-L_tmp, L_Test, S_Test, "SPCA prediction without RPCA␣
↪first", Selection)

    L_tmp = f_PCA_good.decompress(f_PCA_good.compress(X_Test))
    evaluate(L_tmp, X_Test-L_tmp, L_Test, S_Test, "PCA prediction with RPCA␣
↪first")

    L_tmp = f_SPCA_good.decompress(f_SPCA_good.compress(X_Test))
    evaluate(L_tmp, X_Test-L_tmp, L_Test, S_Test, "SPCA prediction with RPCA␣
↪first", Selection_clean)
```

```
py.show()
```

Big anomalies

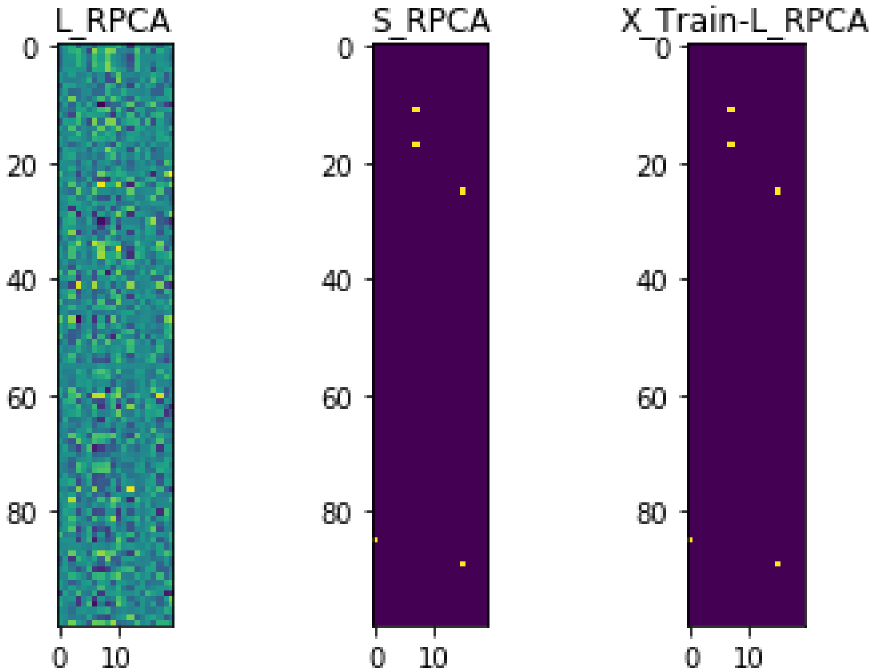
[267]:
```
N = 200
n = 20
K = 6
noiseSize = 0.0
numAnoms = 10
sizeAnoms = 10
runFullTrainingAndTesting(N, n, K, noiseSize, numAnoms, sizeAnoms)
```

```
size anomalies: 10.000000 min of L: 0.000770 max of L: 10.798646
X, L, and S used for training
```

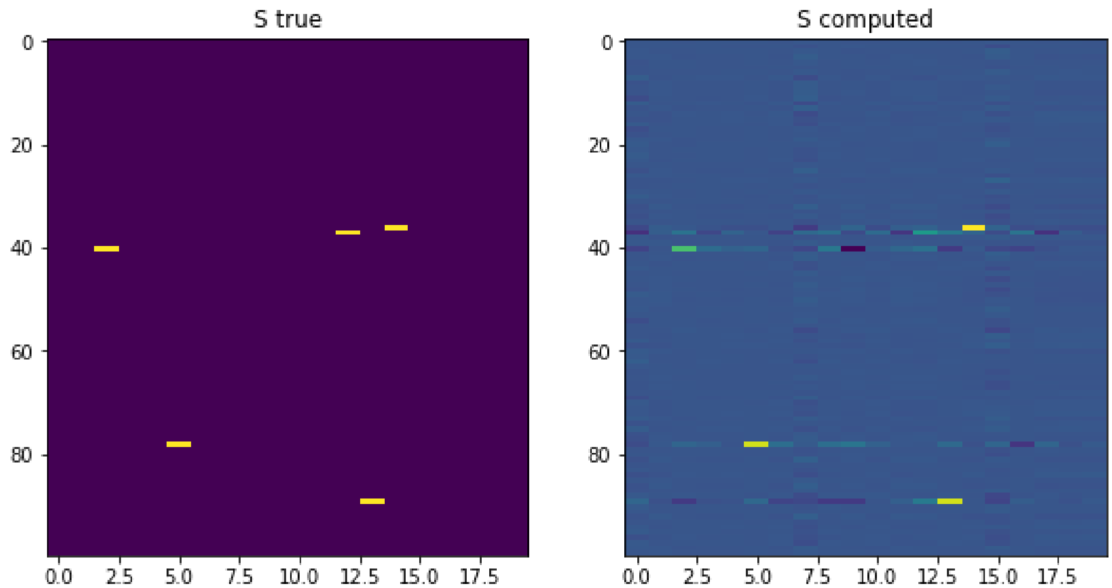



```
criterion1 is the constraint
criterion2 is the solution
iteration criterion1 epsilon1 criterion2 epsilon2 rho      mu
       10   2.94e-03 1.00e-05   1.96e-04 1.00e-04 3.08e+00 1.39e-01
       20   7.57e-04 1.00e-05   3.25e-04 1.00e-04 3.08e+00 4.28e-01
       30   4.13e-05 1.00e-05   2.08e-04 1.00e-04 3.08e+00 1.24e+01
       40   9.81e-06 1.00e-05   5.40e-05 1.00e-04 3.08e+00 1.24e+01
       40   9.81e-06 1.00e-05   5.40e-05 1.00e-04 3.08e+00 1.24e+01
```

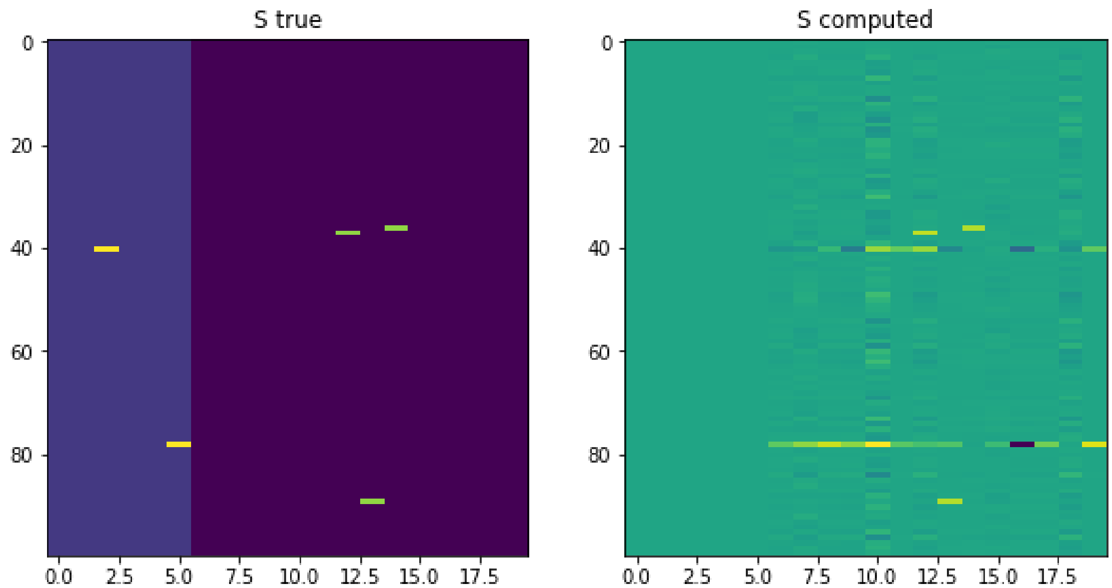
```
L recovery error 0.011772628801154175
L and S returned by RPCA, and X-L
```

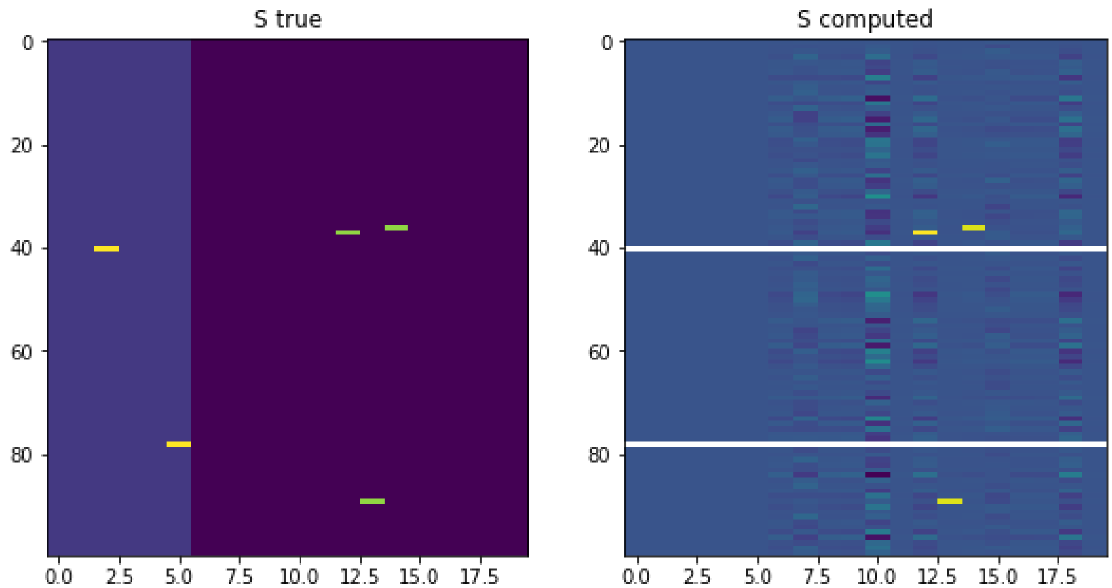



```
0 10.000038479129945
1 0.0010570973050482257
2 0.0030372896845528717
3 0.0033298379341060156
4 0.0008196195725734844
5 0.002108716825902913
6 0.003616002104479496
--------PCA prediction without RPCA first--------
RMSE on L 12.515222893367133
```

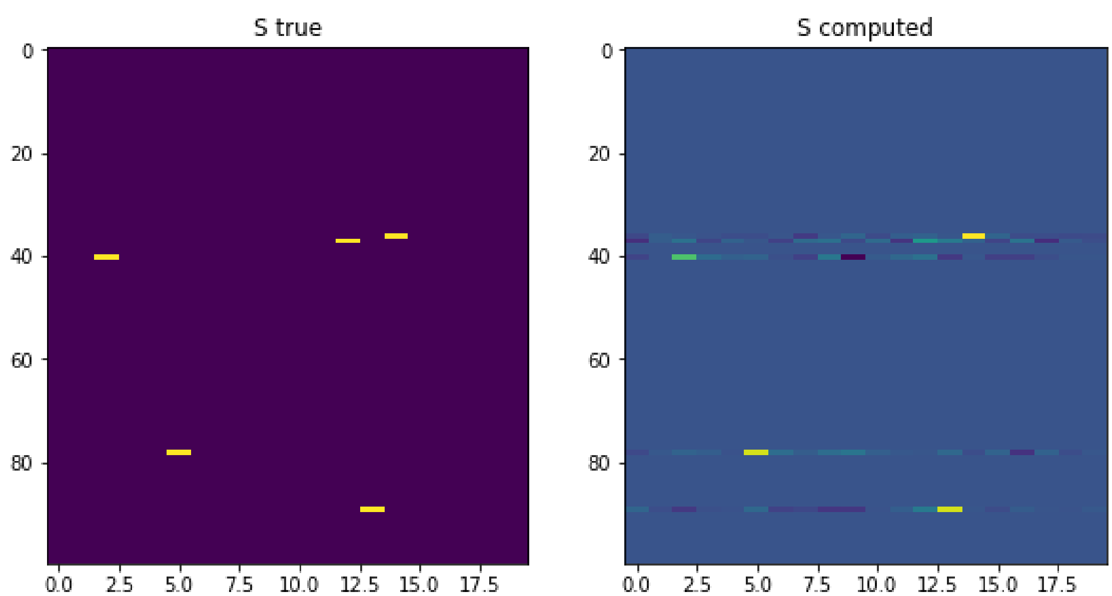



```
confusion matrix on S
----------------
     5   1727
     0    268
--------SPCA prediction without RPCA first--------
RMSE on L 45.621830605903476
```



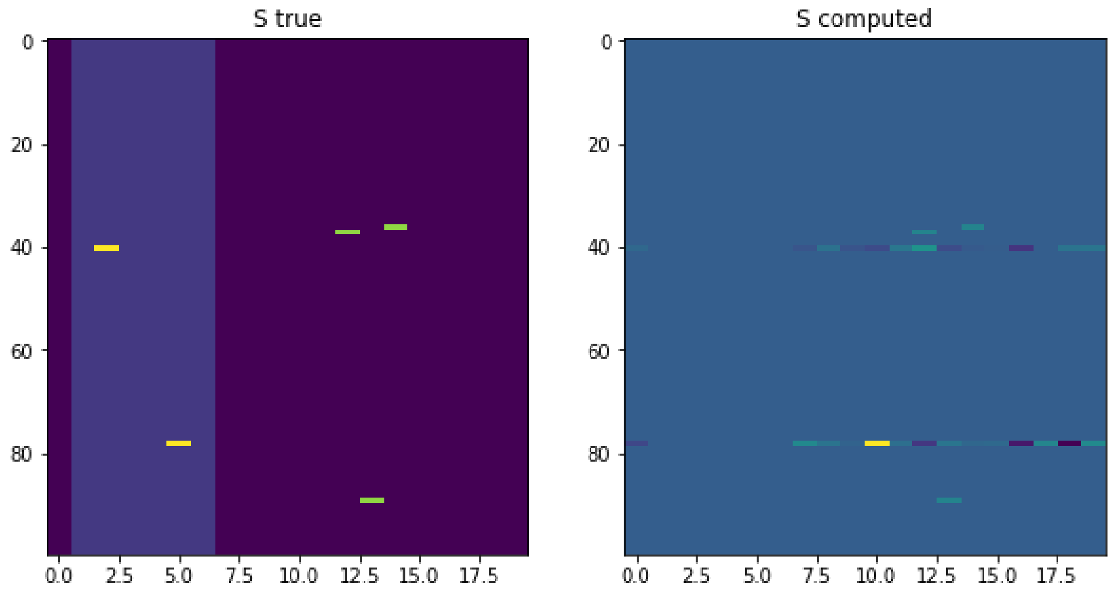


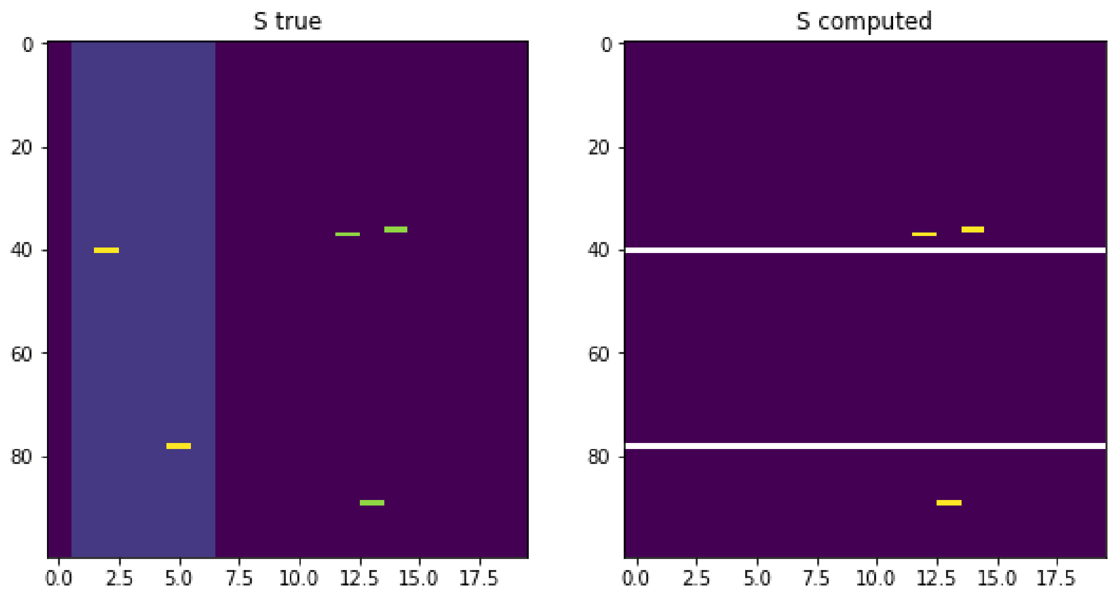

```
confusion matrix on S
----------------
     3   1284
     0    673
--------PCA prediction with RPCA first--------
RMSE on L 11.713570162337959
```




```
confusion matrix on S
```

```
----------------
     5     95
     0   1900
--------SPCA prediction with RPCA first--------
RMSE on L 63.66309876151903
```





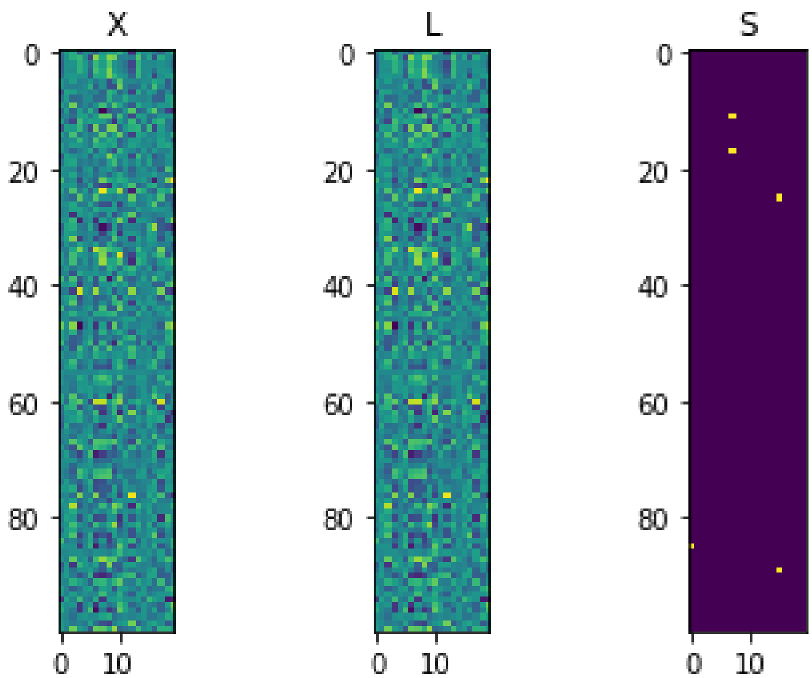

```
confusion matrix on S
----------------
```

```
3      0
0   1957
```

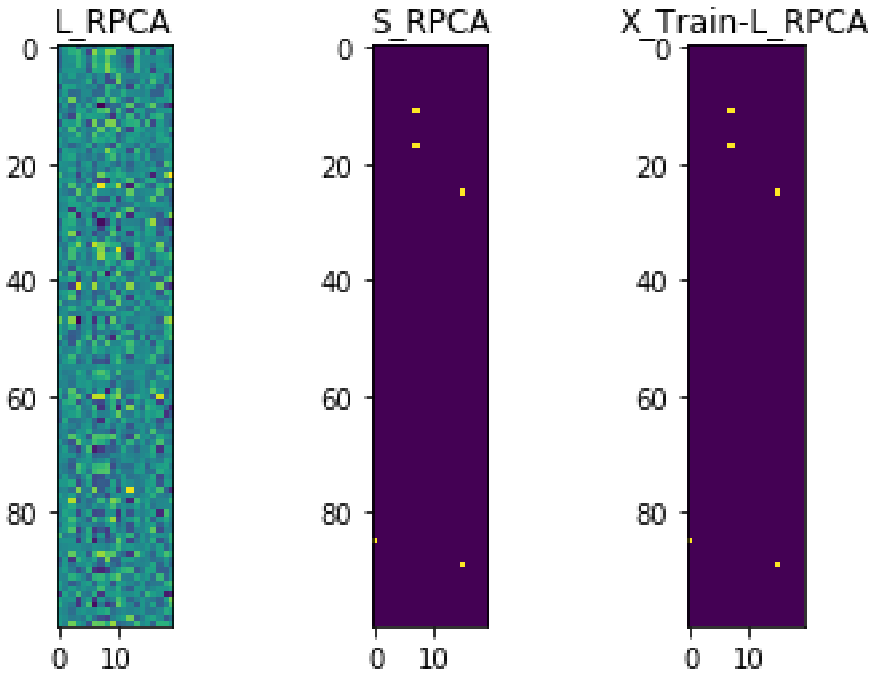
Small anomalies

[268]:
```
N = 200
n = 20
K = 6
noiseSize = 0.0
numAnoms = 10
sizeAnoms = 1
runFullTrainingAndTesting(N, n, K, noiseSize, numAnoms, sizeAnoms)
```

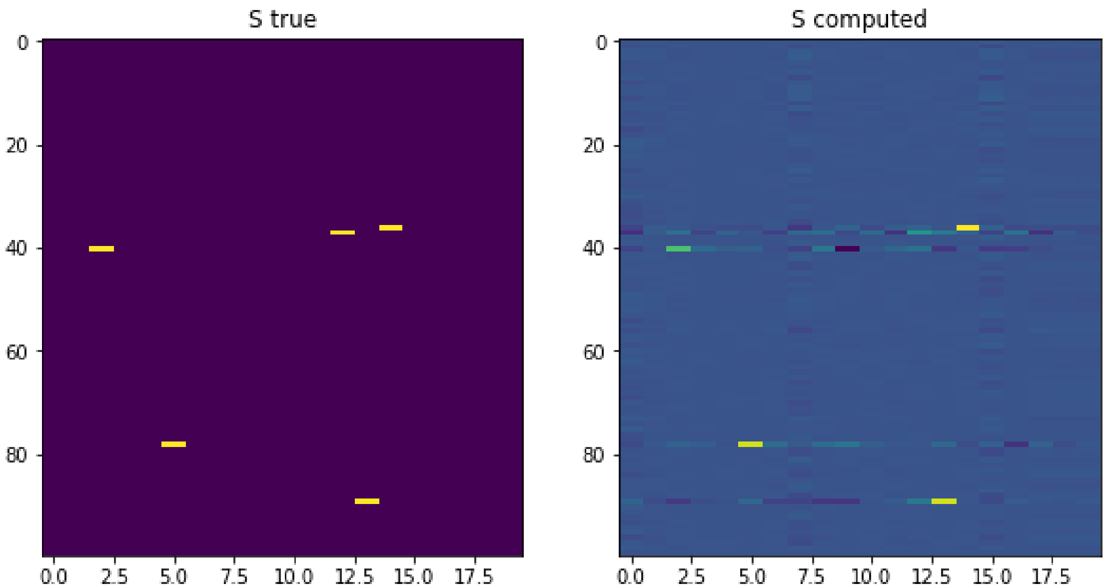
```
size anomalies: 1.000000 min of L: 0.000770 max of L: 10.798646
X, L, and S used for training
```



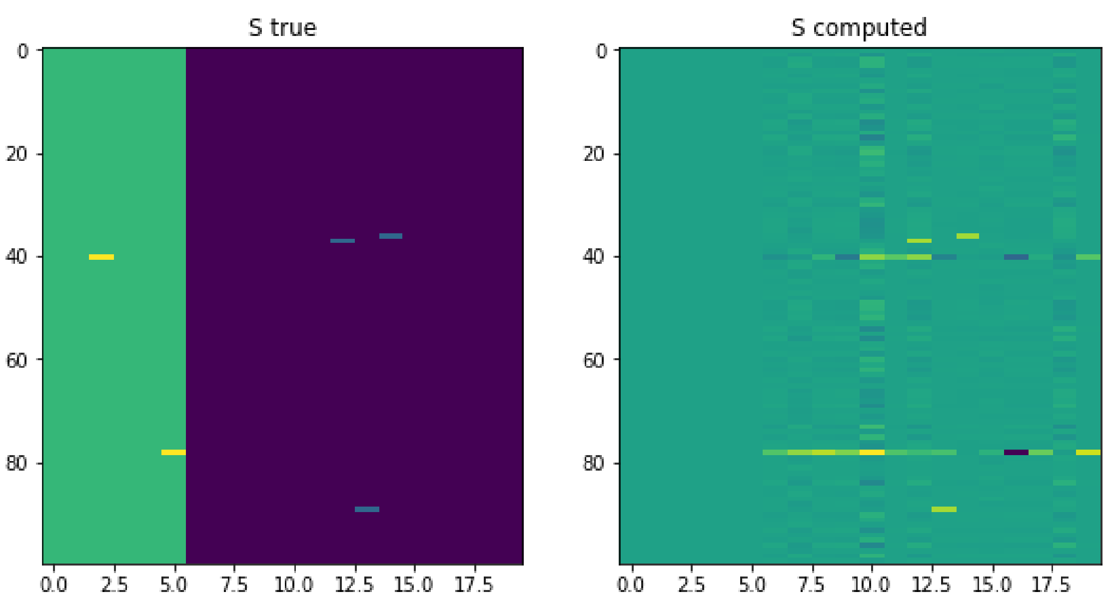

```
criterion1 is the constraint
criterion2 is the solution
iteration criterion1 epsilon1 criterion2 epsilon2 rho      mu
       10   2.03e-03 1.00e-05   1.27e-04 1.00e-04 3.08e+00 1.40e-01
       20   1.72e-04 1.00e-05   1.78e-04 1.00e-04 3.08e+00 1.32e+00
       30   1.67e-05 1.00e-05   1.18e-04 1.00e-04 3.08e+00 1.25e+01
       36   9.90e-06 1.00e-05   3.21e-05 1.00e-04 3.08e+00 1.25e+01
L recovery error 0.011845704144466637
L and S returned by RPCA, and X-L
```

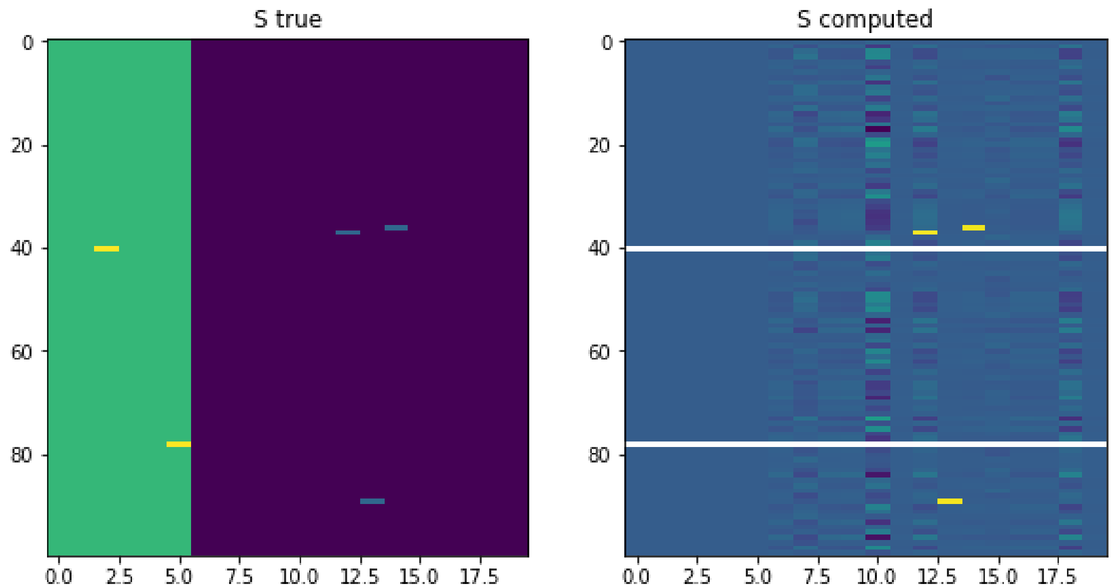



```
0 1.0000076724416558
1 0.0010598088801157765
2 0.002997452696991299
3 0.0033081333140789933
4 0.0008174875745263246
5 0.002108837416274617
6 0.003619940408408073
--------PCA prediction without RPCA first--------
RMSE on L 1.252476735322752
```

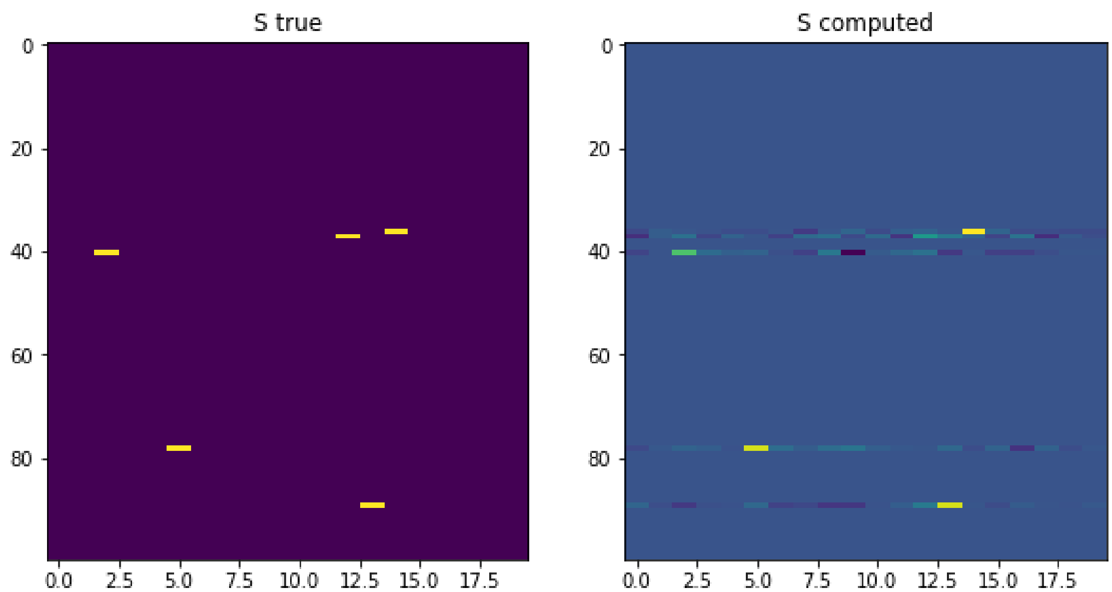



```
confusion matrix on S
----------------
     5    422
     0   1573
--------SPCA prediction without RPCA first--------
RMSE on L 4.718985919036294
```



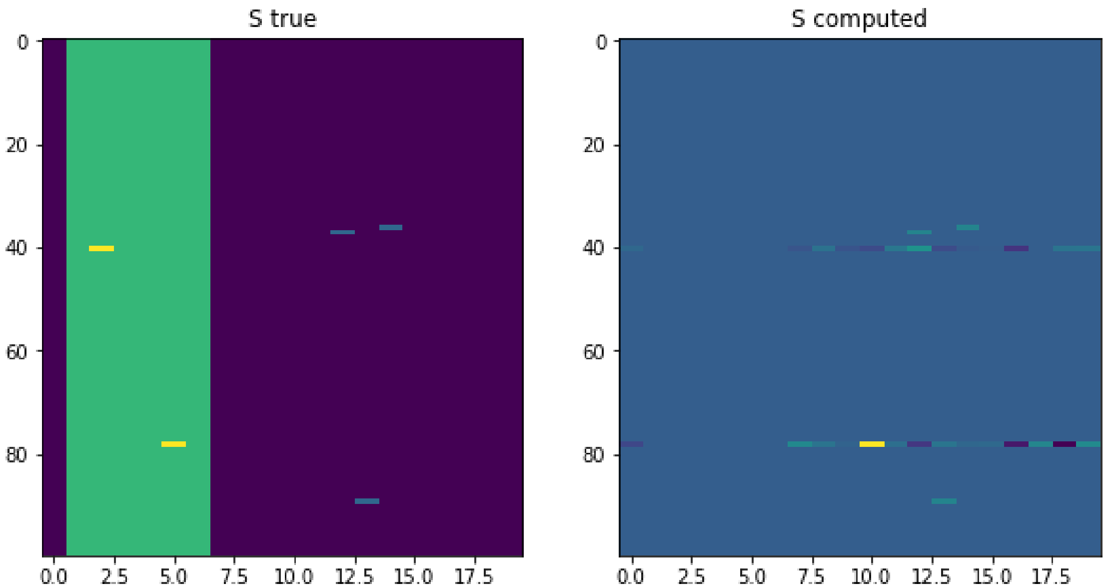


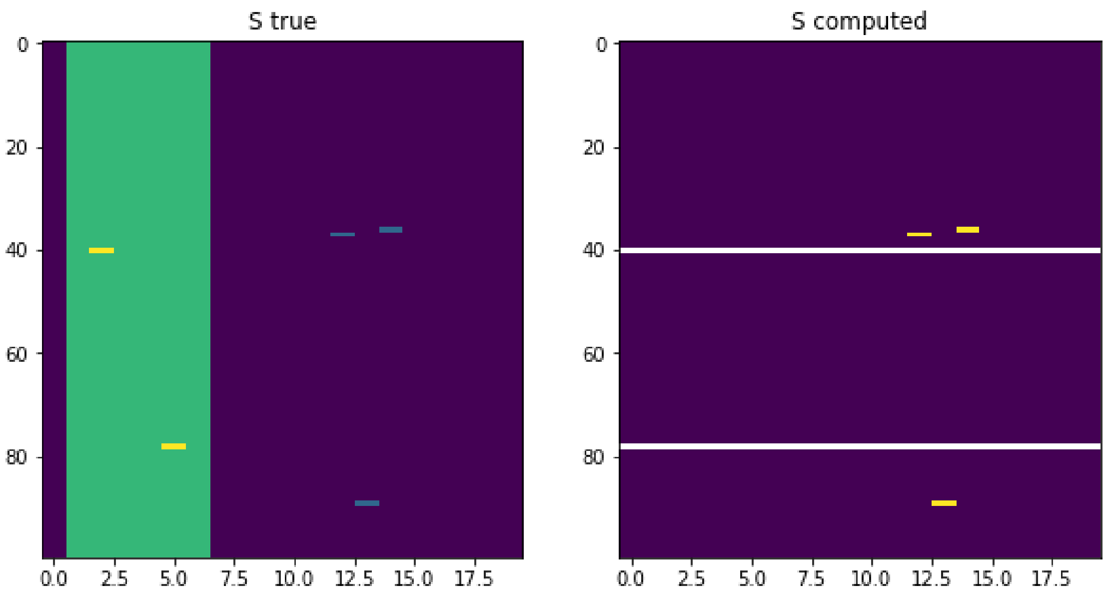

```
confusion matrix on S
----------------
     3    835
     0   1122
--------PCA prediction with RPCA first--------
RMSE on L 1.1713544346946287
```




```
confusion matrix on S
```

```
----------------
     5     83
     0   1912
--------SPCA prediction with RPCA first--------
RMSE on L 6.3663036148077445
```






```
confusion matrix on S
----------------
```

```
3       0
0    1957
```

9 Real data

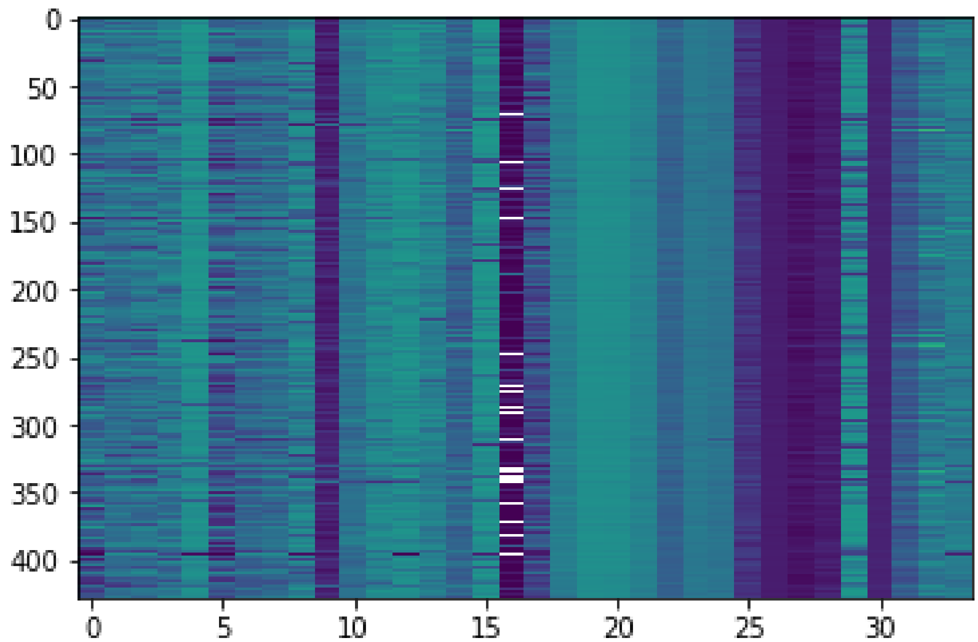
[269]:
```
XReal = pa.read_csv('ACO_Y2016_30FeaturesClean.csv')
XReal
```

[269]:
```
     ACO13  ACO14  ACO15  ACO16  ACO17  ACO18  ACO19  ACO20  ACO21  ACO27  \
0    87.54  92.19  85.49  91.91  98.30  84.76  87.94  89.74  98.56  10.65
1    69.59  77.68  62.17  93.75  98.03  62.39  59.77  49.84  96.24  11.29
2    54.25  78.51  73.39  83.47  95.97  56.28  66.94  68.15  84.21  12.50
3    79.11  77.94  91.23  74.64  97.84  62.10  75.40  75.18  70.59   9.27
4    67.17  49.57  62.46  77.01  83.08  56.02  50.58  65.04  67.77  25.27
..     …      …      …      …      …      …      …      …      …      …
423  29.45  65.92  73.52  70.00  94.43  40.84  53.32  65.72  72.73  14.99
424  37.89  63.58  74.30  45.80  82.73  21.42  57.90  75.04  78.01  33.19
425  45.45  63.95  70.13  73.65  92.63  52.38  56.13  65.17  75.72  23.92
426  64.51  70.03  84.85  70.95  92.23  19.35  78.45  80.08  85.60  14.23
427  45.42  74.10  60.16  84.20  89.43  37.01  70.05  77.34  91.18  16.20

     …   ACO7  ACO34   ACO8   ACO9  ACO10   ACO11  ACO35  ACO36  ACO37  \
0    …  72.16  32.51  14.93  12.83  15.22   85.71  17.71  52.36  69.19
1    …  66.28  17.39  14.78   3.14  15.17   90.16  18.20  47.91  59.25
2    …  70.85  27.09  15.27  10.22  12.60   95.41  18.93  52.07  73.44
3    …  74.00  26.52  15.41   9.96  19.06   91.35  17.71  56.17  85.56
4    …  72.55  23.35  15.12   6.87  13.11   82.88  17.88  50.78  81.03
..   …    …      …      …      …      …       …      …      …      …
423  …  67.86  27.47  13.85   8.34  13.60  100.00  16.63  50.18  70.28
424  …  69.82  38.15  14.17  12.07  13.79   86.15  17.65  54.65  71.41
425  …  74.16  23.99  15.31  12.30  20.77   99.12  17.76  59.36  89.67
426  …  74.49  20.97  15.23   8.33  14.78   63.64  19.66  48.45  61.51
427  …  73.69  27.98  13.88   6.67  19.05   88.89  17.21  46.55  59.93

     ACO42
0    91.52
1    73.67
2    86.86
3    86.32
4    70.39
..     …
423  77.25
424  74.87
425  83.53
426  74.83
427  73.97
```

```
[428 rows x 34 columns]
```

[270]:
```python
py.imshow(XReal, aspect='auto')
```

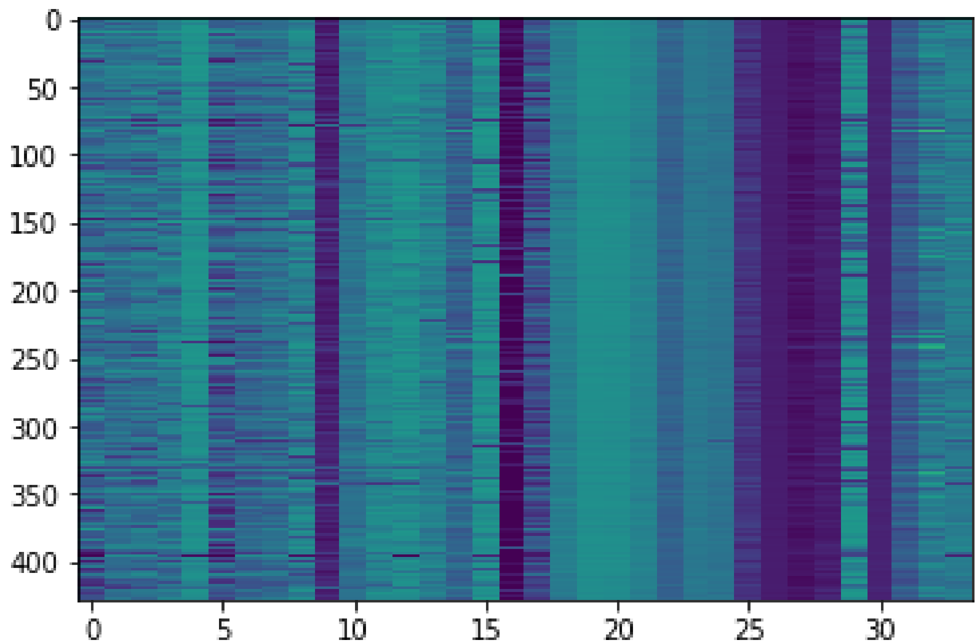
[270]: <matplotlib.image.AxesImage at 0x7fd7dd5cf0d0>



[271]:
```python
XReal = XReal.fillna(0)
```

[272]:
```python
py.imshow(XReal, aspect='auto')
```

[272]: <matplotlib.image.AxesImage at 0x7fd7e426ac90>



Z-transform the data

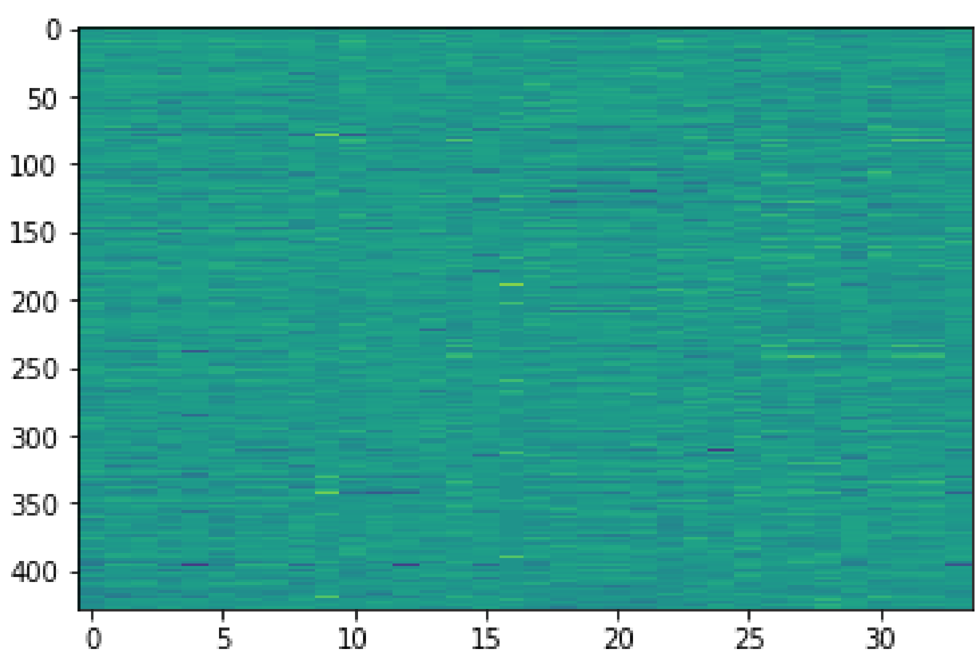
[273]:
```
XReal = (XReal - XReal.mean())/XReal.std()
XReal
```

[273]:
```
        AC013     AC014     AC015     AC016     AC017     AC018     AC019  \
0    1.203700  1.911988  1.055211  1.321523  0.716734  1.467833  1.996477
1    0.284382  0.749615 -0.455870  1.460827  0.690313  0.413154 -0.132261
2   -0.501263  0.816105  0.271160  0.682545  0.488724  0.125086  0.409558
3    0.771953  0.770443  1.427150  0.014040  0.671719  0.399482  1.048860
4    0.160441 -1.502233 -0.437078  0.193469 -0.772674  0.112828 -0.826727
..        …         …         …         …         …         …         …
423 -1.771406 -0.192461  0.279584 -0.337247  0.338022 -0.602864 -0.619672
424 -1.339148 -0.379914  0.330126 -2.169390 -0.806924 -1.518459 -0.273572
425 -0.951959 -0.350274  0.059920 -0.060912  0.161876 -0.058787 -0.407327
426  0.024208  0.136785  1.013741 -0.265324  0.122733 -1.616053  1.279341
427 -0.953495  0.462827 -0.586113  0.737812 -0.151272 -0.783437  0.644574

        AC020     AC021     AC027  …      AC07     AC034      AC08  \
0    1.828692  1.371528 -0.829181  …  0.120137  1.064969  0.352413
1   -1.467687  1.225395 -0.759292  … -2.034521 -2.160547  0.133201
2    0.045012  0.467644 -0.627159  … -0.359897 -0.091267  0.849295
3    0.625803 -0.390259 -0.979878  …  0.794384 -0.212864  1.053893
4   -0.211924 -0.567887  0.767339  …  0.263048 -0.889113  0.630082
..        …         …         … …       …         …         …
423 -0.155745 -0.255464 -0.355248  … -1.455548 -0.010203 -1.225917
```

```
424  0.614237  0.077115  1.632212  …  -0.737329  2.268138 -0.758264
425 -0.201184 -0.067128  0.619918  …   0.853014 -0.752583  0.907752
426  1.030622  0.555198 -0.438241  …   0.973938 -1.396833  0.790838
427  0.804254  0.906673 -0.223115  …   0.680788  0.098594 -1.182075

         AC09     AC010     AC011     AC035     AC036     AC037     AC042
0    1.281329  0.235479  0.145040 -0.353630 -0.070085 -0.444670  1.479421
1   -2.198249  0.218913  0.382213  0.024967 -0.512420 -1.198857 -0.433825
2    0.344105 -0.632557  0.662025  0.588997 -0.098911 -0.122207  0.979940
3    0.250742  1.507715  0.445637 -0.353630  0.308634  0.797385  0.922061
4   -0.858845 -0.463588 -0.005792 -0.222280 -0.227139  0.453676 -0.785390
..        …         …         …         …         …         …         …
423 -0.330983 -0.301246  0.906660 -1.188086 -0.286779 -0.361968 -0.050104
424  1.008421 -0.238297  0.168491 -0.399988  0.157544 -0.276230 -0.305203
425  1.091012  2.074258  0.859758 -0.314997  0.625723  1.109226  0.623016
426 -0.334574  0.089702 -1.031234  1.153028 -0.458743 -1.027382 -0.309491
427 -0.930663  1.504402  0.314526 -0.739952 -0.647605 -1.147262 -0.401669

[428 rows x 34 columns]
```

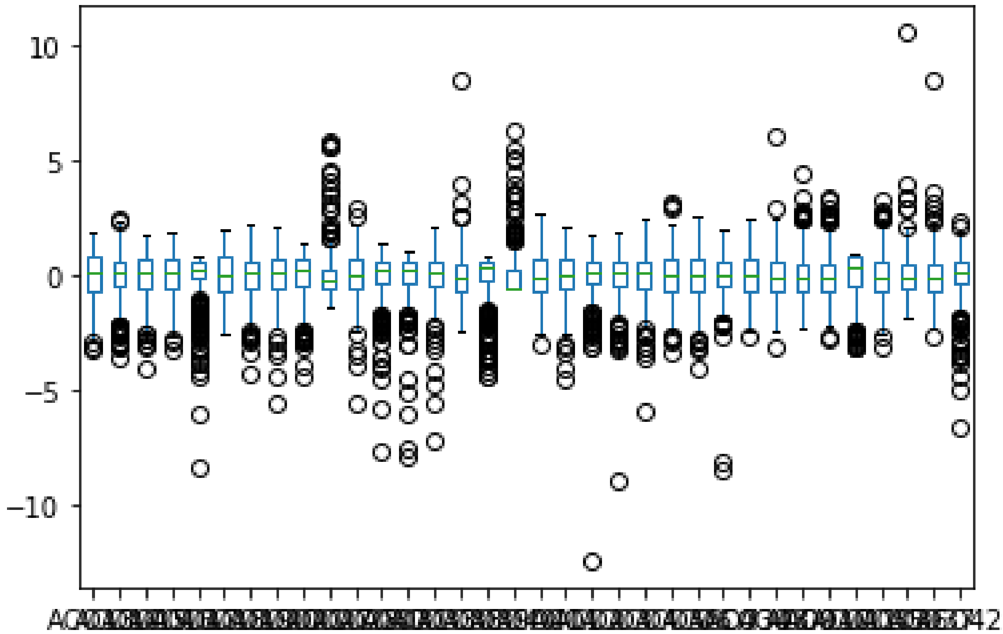
[274]:
```python
py.imshow(XReal, aspect='auto')
```

[274]: <matplotlib.image.AxesImage at 0x7fd7dc929550>



[275]:
```python
XReal.plot.box()
```

[275]: <matplotlib.axes._subplots.AxesSubplot at 0x7fd7f2925890>

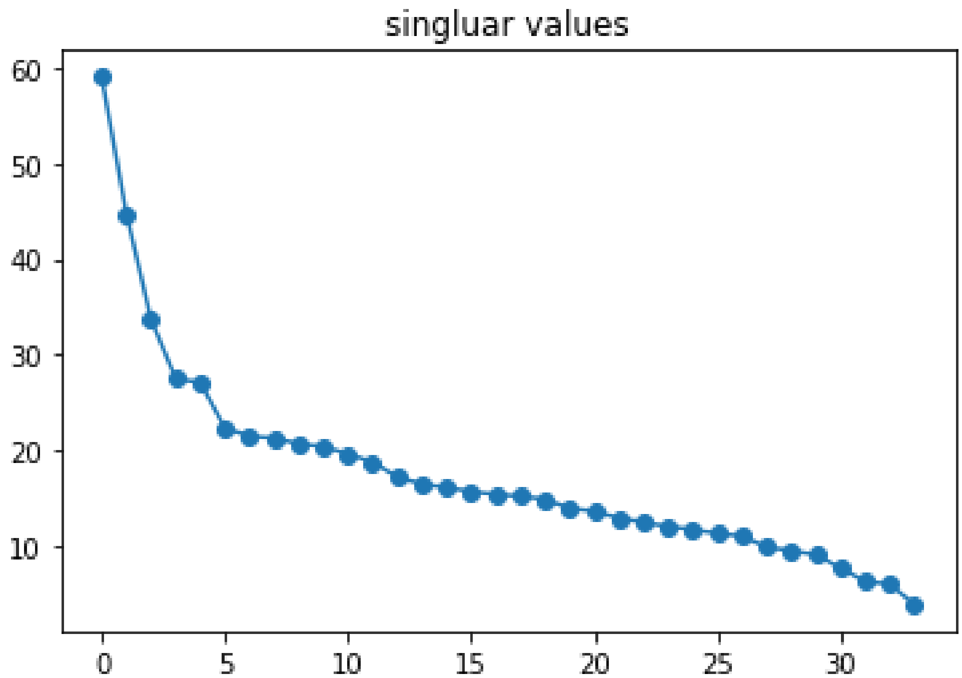


[276]:
```
XReal = np.array(XReal)
```

[277]:
```
U,E,VT = np.linalg.svd(XReal)
```

[278]:
```
py.plot(E, 'o-')
py.title('singluar values')
```

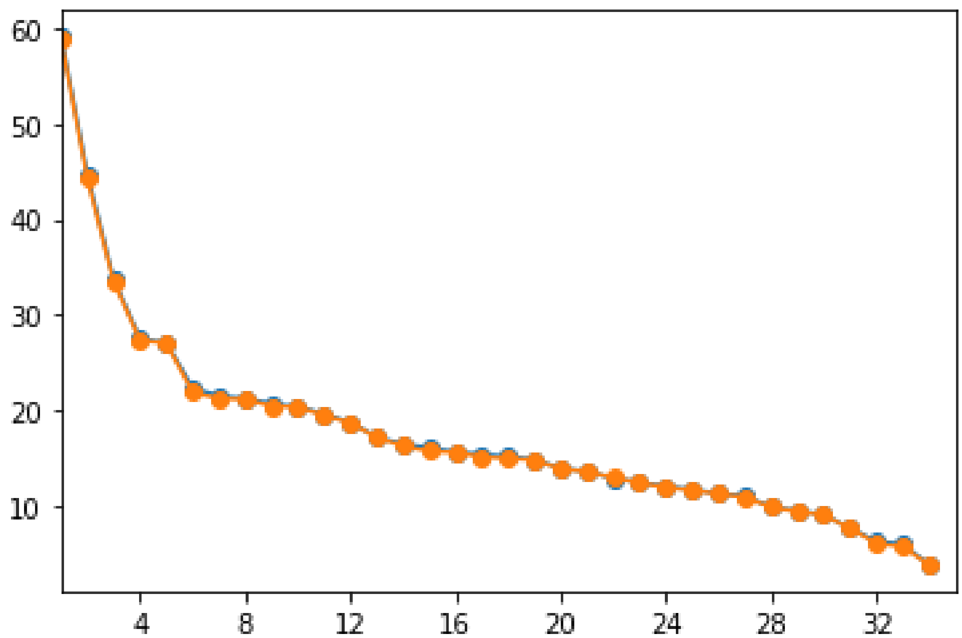
[278]: Text(0.5, 1.0, 'singluar values')

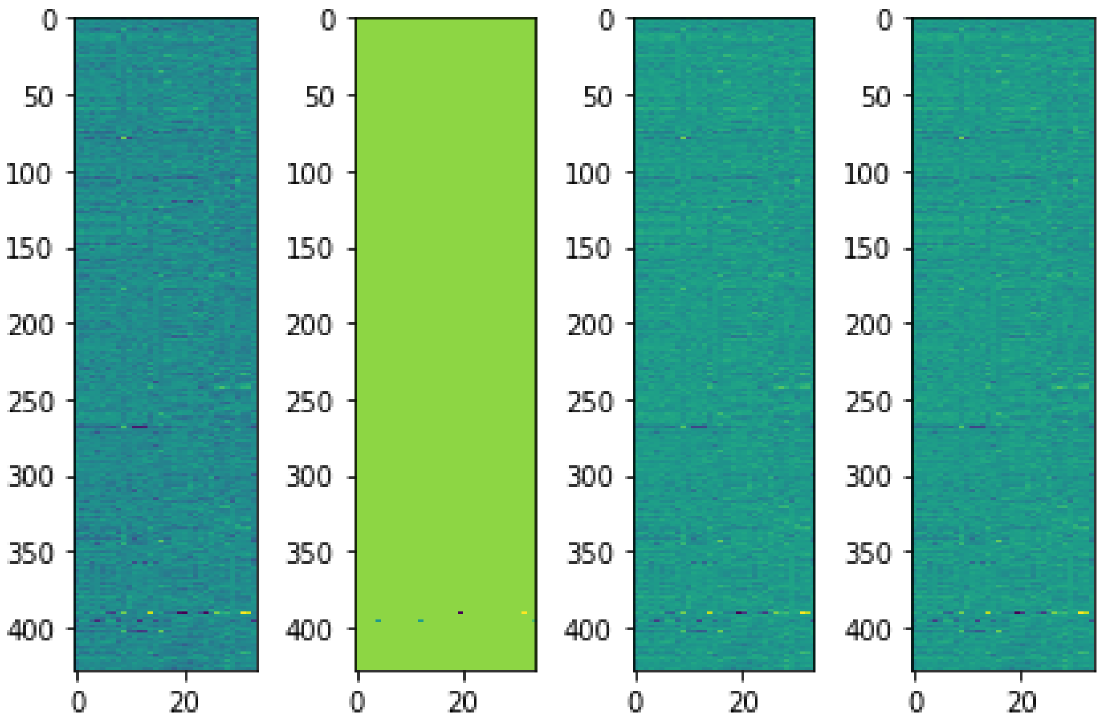



```
[281]: lam1 = 1. / np.sqrt(np.max(XReal.shape))
U, E, VT = np.linalg.svd(XReal)
U_RPCA,E_RPCA,VT_RPCA,S_RPCA = RPCA(XReal, XReal*0.00001, maxIteration=200,␣
 ↪lam=6*lam1)
S_RPCA = S_RPCA.todense()
L_RPCA = XReal-S_RPCA

EPlot(E, py.gca())
EPlot(E_RPCA, py.gca())
print('rank',getRank(E_RPCA))
_,ax = py.subplots(ncols = 4)
ax[0].imshow(L_RPCA, aspect='auto')
ax[1].imshow(S_RPCA, aspect='auto')
ax[2].imshow(L_RPCA+S_RPCA, aspect='auto')
ax[3].imshow(XReal, aspect='auto')
py.tight_layout()
```

```
criterion1 is the constraint
criterion2 is the solution
iteration criterion1 epsilon1 criterion2 epsilon2 rho      mu
       10   6.20e-02 1.00e-05   4.39e-04 1.00e-04 3.08e+00 1.74e-02
Smallest singular value may be too big, consider
increasing maxRank.  This will make the solver slower,
but improve convergence
```

```
Smallest singular value may be too big, consider
increasing maxRank.  This will make the solver slower,
but improve convergence
Smallest singular value may be too big, consider
increasing maxRank.  This will make the solver slower,
but improve convergence
Smallest singular value may be too big, consider
increasing maxRank.  This will make the solver slower,
but improve convergence
Smallest singular value may be too big, consider
increasing maxRank.  This will make the solver slower,
but improve convergence
       18   9.98e-06 1.00e-05   5.42e-05 1.00e-04 3.08e+00 1.56e+00
rank None
```



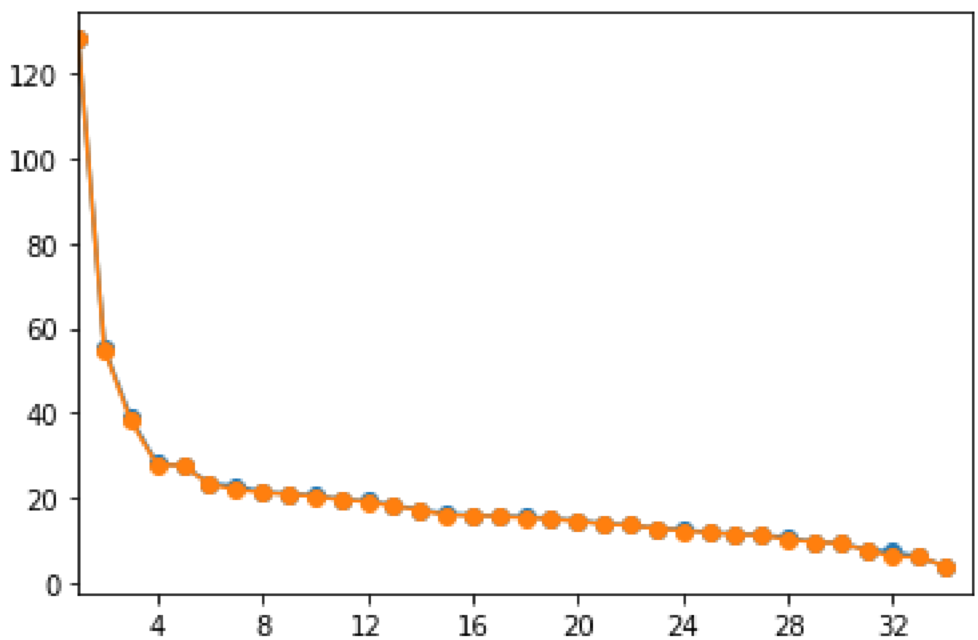


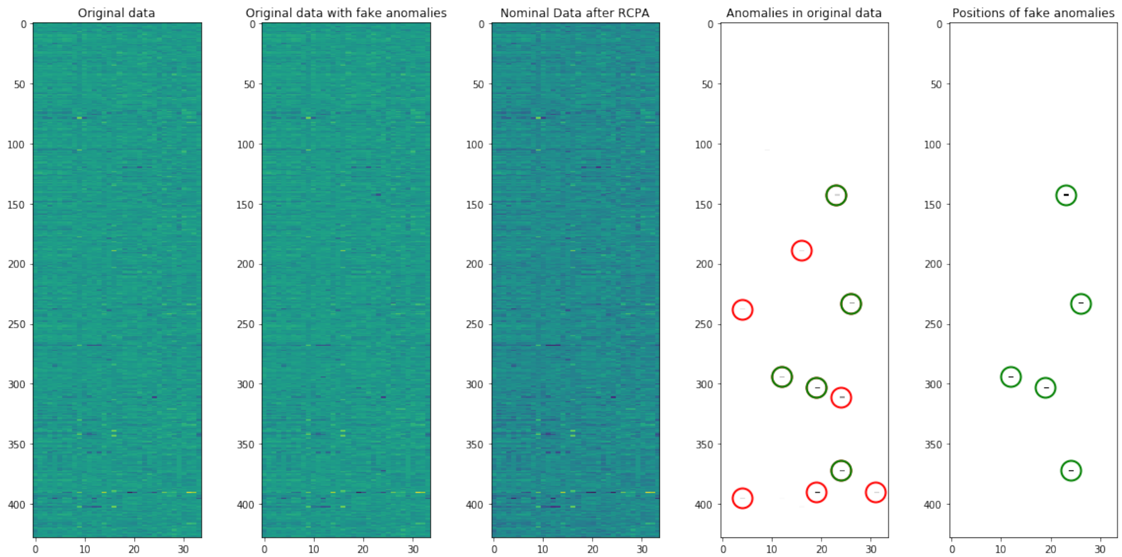
[282]:
```
np.random.seed(1234)
lam1 = 1. / np.sqrt(np.max(XReal.shape))
Fake_S = np.ones(XReal.shape)
Fake_anomalies = []
for k in range(5):
    i = np.random.randint(0, XReal.shape[0])
    j = np.random.randint(0, XReal.shape[1])
    print(i,j)
    Fake_anomalies += [(i,j)]
    Fake_S[i, j] = -5

XTest = Fake_S + XReal

U, E, VT = np.linalg.svd(XTest)
U_RPCA,E_RPCA,VT_RPCA,S_RPCA = RPCA(XTest, XTest*0.00001, maxIteration=200,␣
 ↪lam=6*lam1)
S_RPCA = S_RPCA.todense()
L_RPCA = XReal-S_RPCA

EPlot(E, py.gca())
EPlot(E_RPCA, py.gca())
print('rank',getRank(E_RPCA))
cmap = 'binary'
```

```
_,ax = py.subplots(ncols = 5,figsize=(16,8))
ax[0].imshow(XReal, aspect='auto')
ax[0].set_title('Original data')

ax[1].imshow(XTest, aspect='auto')
ax[1].set_title('Original data with fake anomalies')

ax[2].imshow(L_RPCA, aspect='auto')
ax[2].set_title('Nominal Data after RCPA')

ax[3].imshow(np.abs(S_RPCA), aspect='auto', cmap=cmap)
ax[3].set_title('Anomalies in original data')

ax[4].imshow(np.abs(Fake_S), aspect='auto', cmap=cmap, interpolation='none')
ax[4].set_title('Positions of fake anomalies')

for i in range(S_RPCA.shape[0]):
    for j in range(S_RPCA.shape[1]):
        if np.abs(S_RPCA[i, j]) > 0.3:
            ax[3].plot(j, i, 'o',
                       ms=20, mec='r', mfc='none', mew=2)

for anomalie in Fake_anomalies:
    ax[3].plot(anomalie[1], anomalie[0], 'o',
               ms=20, mec='g', mfc='none', mew=2)
    ax[4].plot(anomalie[1], anomalie[0], 'o',
               ms=20, mec='g', mfc='none', mew=2)


#     ax[3].annotate("fake",
#                    xy=(anomalie[1], anomalie[0]), xycoords='data',
#                    xytext=(anomalie[1]-5, anomalie[0]-5), textcoords='data',
#                    arrowprops=dict(arrowstyle="->",
#                                    connectionstyle="arc3"),)
#     ax[4].annotate("fake",
#                    xy=(anomalie[1], anomalie[0]), xycoords='data',
#                    xytext=(anomalie[1]-5, anomalie[0]-5), textcoords='data',
#                    arrowprops=dict(arrowstyle="->",
#                                    connectionstyle="arc3"),)
py.tight_layout()
```

```
303 19
294 12
372 24
143 23
233 26
```

```
criterion1 is the constraint
criterion2 is the solution
iteration criterion1 epsilon1 criterion2 epsilon2 rho      mu
       10   2.25e-02 1.00e-05   1.81e-04 1.00e-04 3.08e+00 7.52e-02
Smallest singular value may be too big, consider
increasing maxRank.  This will make the solver slower,
but improve convergence
Smallest singular value may be too big, consider
increasing maxRank.  This will make the solver slower,
but improve convergence
Smallest singular value may be too big, consider
increasing maxRank.  This will make the solver slower,
but improve convergence
Smallest singular value may be too big, consider
increasing maxRank.  This will make the solver slower,
but improve convergence
Smallest singular value may be too big, consider
increasing maxRank.  This will make the solver slower,
but improve convergence
       15   3.77e-06 1.00e-05   4.56e-05 1.00e-04 3.08e+00 2.19e+00
rank None
```




Original data
Original data with fake anomalies
Nominal Data after RCPA
Anomalies in original data
Positions of fake anomalies

[ ]: