CS598PS - Problem Set 2

Preliminaries

[14]
```
%load_ext autoreload
%autoreload 2
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

from numpy import *
from matplotlib.pylab import *

# Some optional beautification
rcParams['figure.figsize'] = (14,6)
rcParams['lines.linewidth'] = 1
rcParams['image.cmap'] = 'Greys'
rcParams['axes.spines.right'] = False
rcParams['axes.spines.top'] = False
rcParams['font.family'] = 'Avenir Next LT Pro'
rcParams['font.weight'] = 400
rcParams['xtick.color'] = '#222222'
rcParams['ytick.color'] = '#222222'
rcParams['grid.color'] = '#dddddd'
rcParams['grid.linestyle'] = '-'
rcParams['grid.linewidth'] = 0.5
rcParams['axes.titlesize'] = 11
rcParams['axes.titleweight'] = 600
rcParams['axes.labelsize'] = 10
rcParams['axes.labelweight'] = 400
rcParams['axes.linewidth'] = 0.5
rcParams['axes.edgecolor'] = [.25,.25,.25]
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

Define all the decompositions

[15]
```
# Perform PCA by diagonalizing the coavariance
def pca( x, k):
    # Remove data mean
    xm = x - mean( x, axis=1, keepdims=True)
```

```
    # Get covariance estimate
    C = xm.dot( xm.T) / (xm.shape[1]-1)

    # Get top k PCA covariance eigenvectors/values
    v,u = scipy.sparse.linalg.eigsh( C, k=k)

    # Get overall transform and the input's projection to k
dimensions
    w = diag( 1./sqrt(v)).dot( u.T)
    y = w.dot( xm)

    return w,y

# Perform ICA using infomax (assumes white data)
def ica( x):
    I = eye( x.shape[0])
    w = I
    for i in range( 500):
        y = w.dot( x)
        dw = (x.shape[1]*I - 2*tanh( y).dot( y.T)).dot( w)
        w = w + .0001*dw

    return w,y

# Perform KL-NMF
def nmf( x, k):
    w = rand( x.shape[0], k)
    h = rand( k, x.shape[1])
    eps = .000001 # use this to avoid potential divisions by 0
    for i in range( 100):
        h = h * w.T.dot( x) / (w.T.dot( w).dot( h)+eps)
        w = w * x.dot( h.T) / (w.dot( h).dot( h.T)+eps)

    return w,h
```

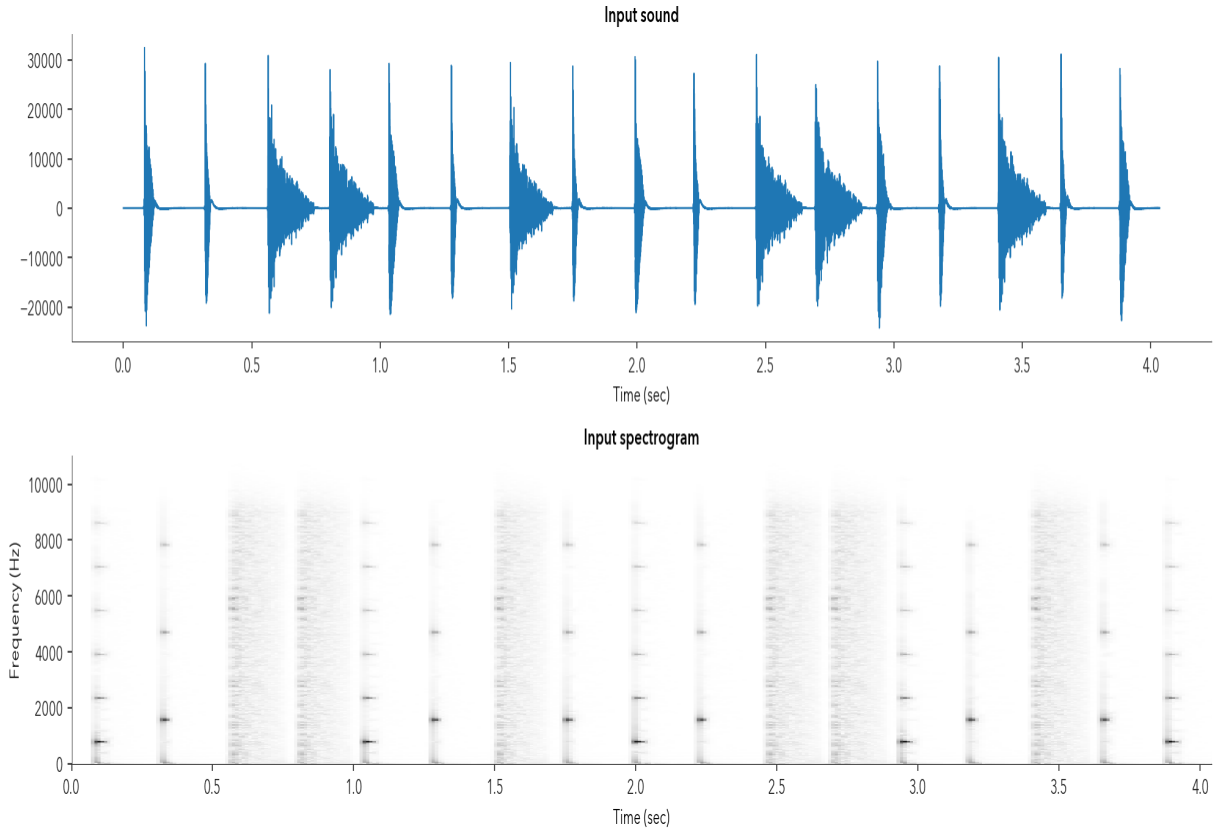
Problem 1

Load the sound and plot it

[16]
```
# Load the file and convert to floating point
import scipy.io.wavfile
fs,s = scipy.io.wavfile.read( '/Users/paris/Desktop/vl1.wav')
s = s.astype( float)
subplot( 2, 1, 1), plot( arange( 0, len( s))/fs, s)
title( 'Input sound'), xlabel( 'Time (sec)')

# Get the magnitude spectrogram
from scipy.signal import stft
q,t,f = stft( s, fs=fs, nperseg=1024, noverlap=768)
```

```
f = abs( f)
subplot( 2, 1, 2), pcolormesh( t, q, f**.5), title( 'Input
spectrogram')
xlabel( 'Time (sec)'), ylabel( 'Frequency (Hz)')
tight_layout()
```



In the spectrogram we can see the three instruments clearly. First we see the lower-frequency tone that has a spectrum with 6 peaks throughout the frequency spread. It is followed by the higher-frequency tone that has a spectrum with three peaks. The snare drum sound come next and has a wideband structure, meaning that it has a lot of energy throughout all the frequencies. Because of that it looks more “cloudy”. We can see these instruments repeating across the time axis as the drum loop is formed.

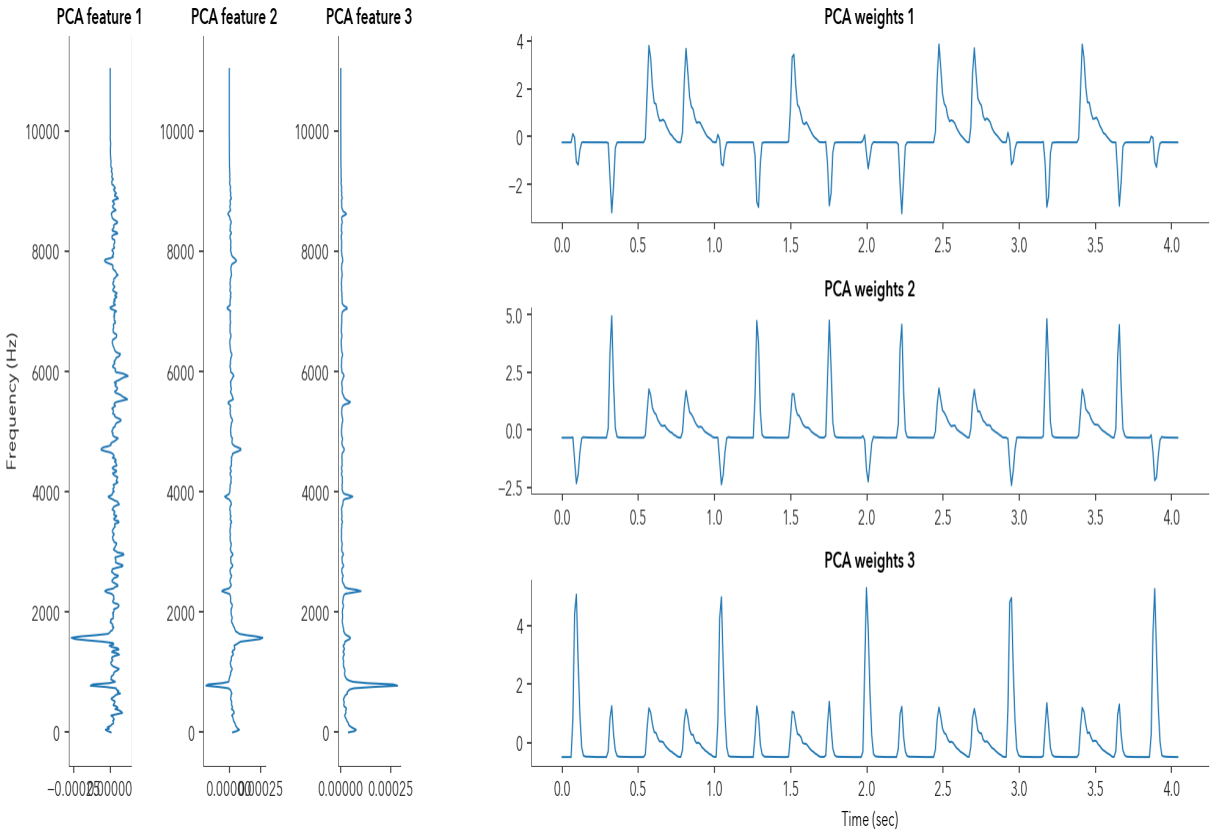
Do PCA on the audio data

[17]
```
# Do it ...
wp,xp = pca( f, 3);

# Show me the eigenvectors
subplot2grid( (1,9), (0,0)), plot( wp[0], q), title( 'PCA feature
1'), ylabel( 'Frequency (Hz)')
subplot2grid( (1,9), (0,1)), plot( wp[1], q), title( 'PCA feature
2')
```

```
subplot2grid( (1,9), (0,2)), plot( wp[2], q), title( 'PCA feature
3')

# Show me the projection
subplot2grid( (3,3), (0,1), colspan=2), plot( t, xp[0]), title( 'PCA
weights 1')
subplot2grid( (3,3), (1,1), colspan=2), plot( t, xp[1]), title( 'PCA
weights 2')
subplot2grid( (3,3), (2,1), colspan=2), plot( t, xp[2]), title( 'PCA
weights 3'), xlabel('Time (sec)')
tight_layout()
```



In the PCA features we see some elements of the spectra of the three instruments. The first feature seems to be a sign-inverted representation of the low tone (the energies are negative as opposed to positive as we would prefer to perceive them). The second feature seems to have be a mix of a sign-inverted low tone and the high tone. The third feature is the spectrum of the snare drum, but we also see the high tone and to a lesser degree the low tone both being mixed in and sign-inverted. In the PCA weights we see some of the structure of the drum beat, but things are a little mixed together.

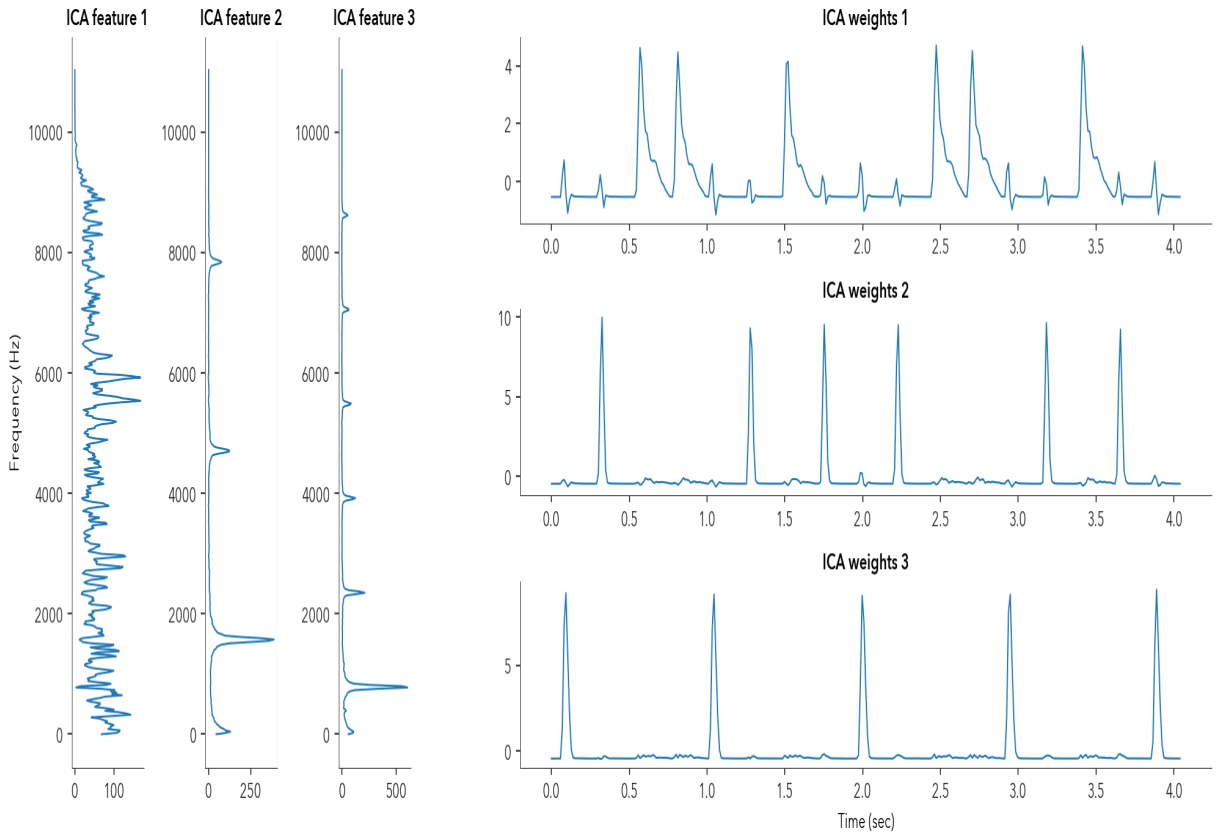
Get the ICA features

[18]
```
# Do ICA on the already whitened data
wi,xi = ica( xp)
```

```
# Get the inverse of the overall transform (PCA & ICA) to get the
generating components
a = linalg.pinv( wi.dot( wp))

# Show me the features
subplot2grid( (1,9), (0,0)), plot( a[:,0], q), title( 'ICA feature
1'), ylabel( 'Frequency (Hz)')
subplot2grid( (1,9), (0,1)), plot( a[:,1], q), title( 'ICA feature
2')
subplot2grid( (1,9), (0,2)), plot( a[:,2], q), title( 'ICA feature
3')

# Show me the projection
subplot2grid( (3,3), (0,1), colspan=2), plot( t, xi[0]), title( 'ICA
weights 1')
subplot2grid( (3,3), (1,1), colspan=2), plot( t, xi[1]), title( 'ICA
weights 2')
subplot2grid( (3,3), (2,1), colspan=2), plot( t, xi[2]), title( 'ICA
weights 3'), xlabel('Time (sec)')
tight_layout()
```



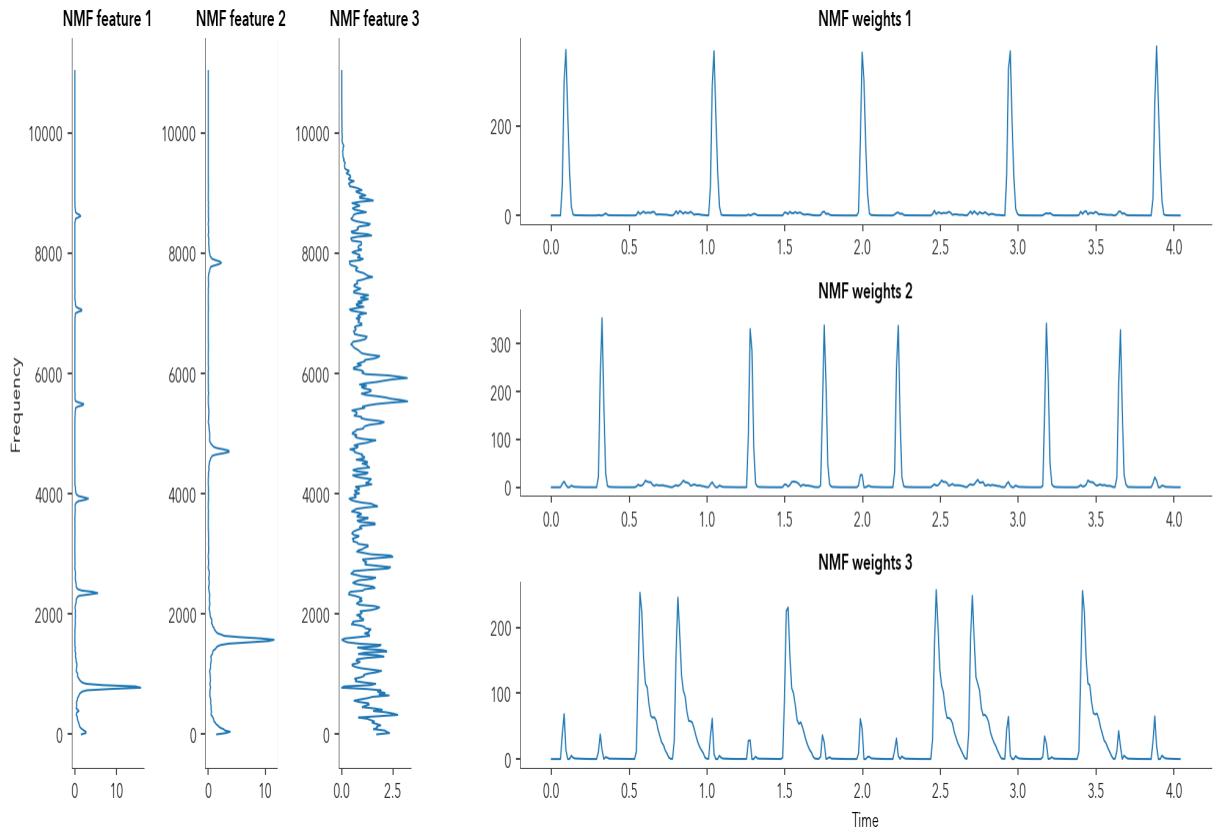
In the ICA features we see the three instrument spectra being represented perfectly, aside from the fact that the low tone is sign-inverted. The corresponding weights provide a very good explanation of the input. The first weight tells us how the low tone gets activated in time. The weights are sign inverted since the feature itself is as well. The product of these two vectors will reconstruct only the part of the sound that corresponds to that instrument. Likewise the second and third features describe how the other two instruments are activated in time. This happens because the three instruments are statistically independent, and ICA strives to obtain a statistically independent feature set it gives us this satisfying description.

NMF features

[19]
```
# Do NMF on the original input data
w,h = nmf( f, 3)

# Show me the bases
subplot2grid( (1,9), (0,0)), plot( w[:,0], q), title( 'NMF feature
1'), ylabel( 'Frequency')
subplot2grid( (1,9), (0,1)), plot( w[:,1], q), title( 'NMF feature
2')
subplot2grid( (1,9), (0,2)), plot( w[:,2], q), title( 'NMF feature
3')

# Show me the activations
subplot2grid( (3,3), (0,1), colspan=2), plot( t, h[0]), title( 'NMF
weights 1')
subplot2grid( (3,3), (1,1), colspan=2), plot( t, h[1]), title( 'NMF
weights 2')
subplot2grid( (3,3), (2,1), colspan=2), plot( t, h[2]), title( 'NMF
weights 3'), xlabel( 'Time')
tight_layout()
```



The NMF features are very similar to the ICA features, only this time because of the non-negativity constraint we don't see the sign-inversion we had with ICA. This results in a very

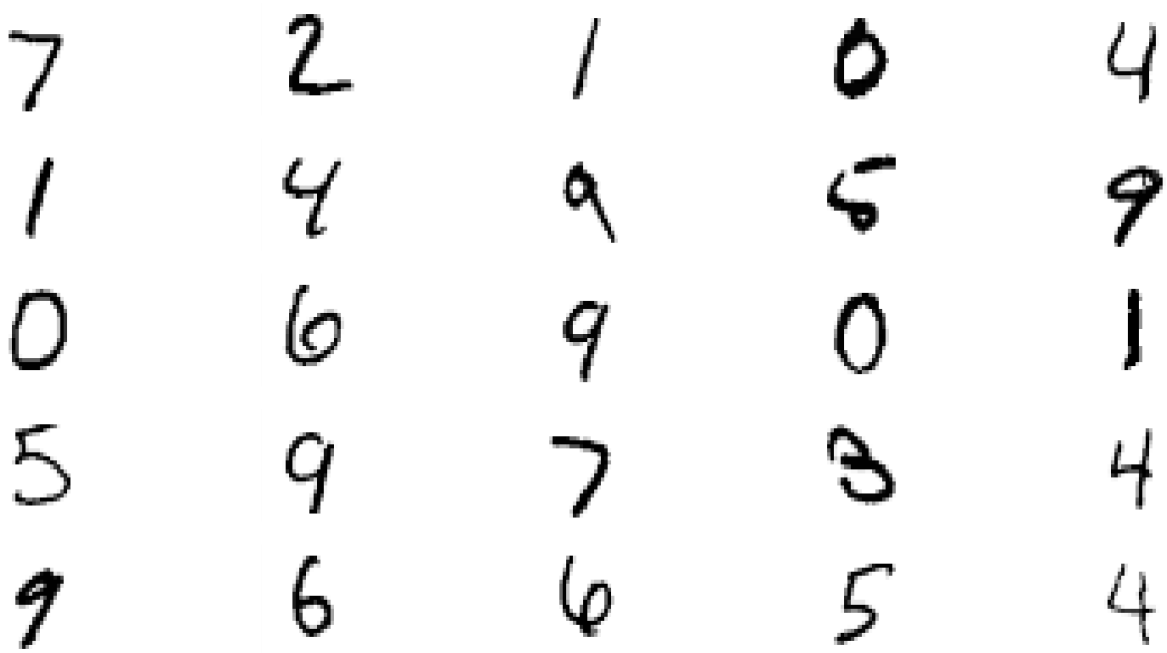
natural decomposition of the data where we can clearly see both the spectral and the temporal structure of the three instruments.

Problem 2

[20]
```
# Load the data
import scipy.io
d = scipy.io.loadmat( '/Users/paris/Desktop/digits.mat')
l = d['l']
d = d['d']

# Function to show a digit
def digshow( x):
    imshow( reshape( x, (28, 28), 'F'))
    xticks( []), yticks( []), axis( 'off')

# Show me a few
for i in range( 25):
    subplot( 5, 5, i+1), digshow( d[:,i])
```



Do PCA

[21]
```
# As promised
wp,xp = pca( d, 36)

# Show me
```

```
for i in range( 36):
    subplot( 6, 6, i+1), digshow( wp[i])
```

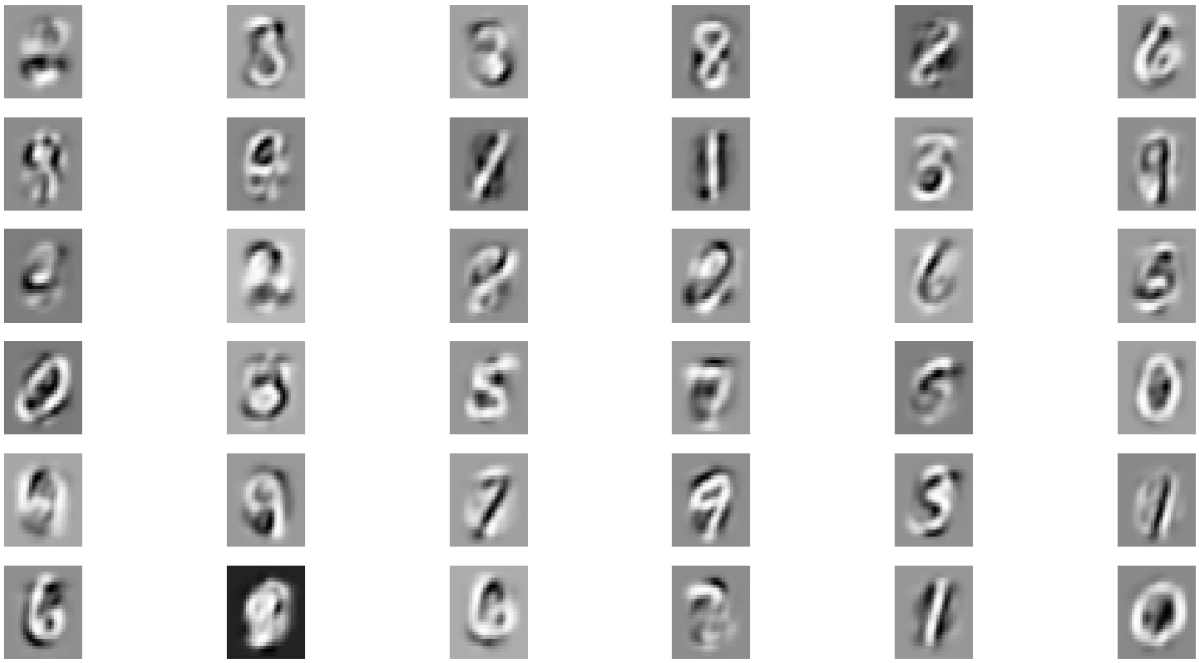
The PCA features of the digits look similar to the eigenfaces example. We see some ghostly images that capture a lot of the frequency information of the input. The first component on the top left is the average of all the inputs, and as we move to the less important components we see higher frequency elements appearing, however they do so only where the digits tend to be (i.e., not around the edges of the image).

Do ICA

[22]
```
# Do ICA on the already whitened digits
wi,xi = ica( xp)

# Get the inverse of the overall transform (PCA & ICA) to get the
generating components
a = linalg.pinv( wi.dot( wp))

# Show me
for i in range( 36):
    subplot( 6, 6, i+1), digshow( a[:,i])
```



Just as in the previous problem the ICA features look more like the actual independent elements in the input data (in this case characters). But you can see that once again they can be sign-inverted.

Do NMF

[23]
```
# Do NMF on the original input data
w,h = nmf( d, 36)

# Show me
for i in range( 36):
    subplot( 6, 6, i+1), digshow( w[:,i])
```

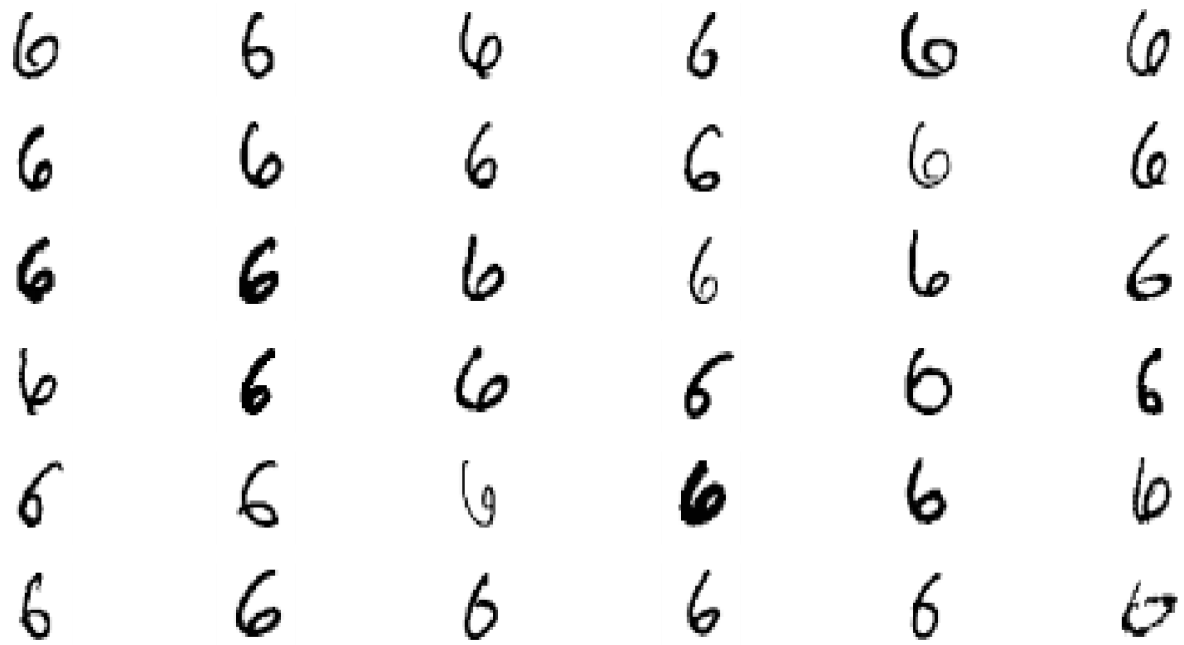
The NMF features are now much different. We see that they focus on parts of the characters, such as individual strokes or segments. Because NMF cannot use cross-cancellation like PCA and ICA, it was to resort to such a representation where the features have minimal overlap with each other.

Problem 3

[24]

```
# Get the sixes
d6 = d[:,find( l==6)]

# Show me some
for i in range( 36):
    subplot( 6, 6, i+1), digshow( d6[:,i])
```

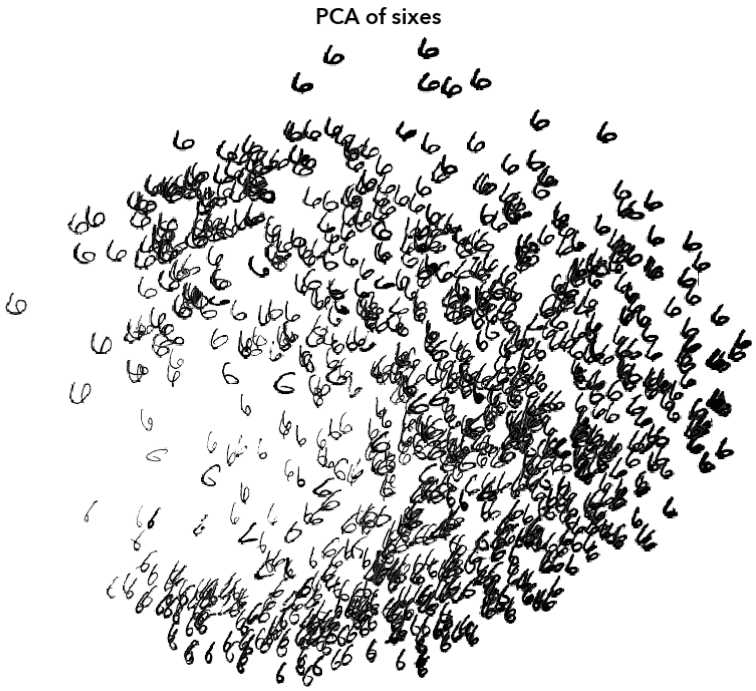


PCA of sixes

[25]
```
# Get 2d embedding using PCA
wp,xp = pca( d6, 2)

# Make a gray colormap where white is transparent
ga = matplotlib.colors.LinearSegmentedColormap.from_list( 'Grays_a',
[(1,1,1,0), (0,0,0,1)], N=128)

# Show me the sixes in their space
for i in range( d6.shape[1]):
    imshow( reshape( d6[:,i], (28,28), 'F'), zorder=0, cmap=ga,
            extent=[xp[0,i], xp[0,i]+.2, xp[1,i], xp[1,i]+.2])
axis( [min( xp[0]), max( xp[0])+.2, min( xp[1]), max( xp[1])+.2])
axis( 'off')
title( 'PCA of sixes');
```



If we plot the projection of our data through the principal components we get a crescent ordering in which one tip contains thin sixes with the ascender pointing towards the right, and we gradually see that trend change as we move to the other tip where the sixes are wider and the ascender points upwards.

Laplacian Eigenmaps of sixes

[26]

```
# Get distance matrix
w = array( [sqrt( sum( (d6-d6[:,i][:,None])**2, axis=0)) for i in
range( d6.shape[1])])

# # Keep only 10 closest neighbors
j = argsort( w, axis=0);
for i in range( w.shape[0]):
    w[i,j[10:,i]] = 0
w = exp( -w);

# Get Laplacian and normalize it
d = sum( w, axis=0)
l = w - diag( d)
l = diag( d**(-.5)).dot( l).dot( diag( d**(-.5)))

# Get the decomposition and keep smallest non-zero eigenvalues
v,u = scipy.sparse.linalg.eigs( l, k=3, which='SR')
xp = real( u[:,:2]).T.dot( diag( d**.5))

# Show me the sixes in their space
```

```
figure( figsize=(12,8))
sz = .25
for i in range( 0, d6.shape[1], 8): # Show only 1 out of every 5 for
clarity
    imshow( reshape( d6[:,i], (28,28), 'F'), zorder=0, cmap=ga,
             extent=[xp[0,i], xp[0,i]+sz, xp[1,i], xp[1,i]+sz])
axis( [min( xp[0]), max( xp[0])+sz, min( xp[1]), max( xp[1])+sz])
title( 'Laplacian Eigenmaps of sixes');
axis( 'off');
```

With the manifold analysis we ignore neighbors that are further away and instead obtain a cleaner topology. Now we can clearly see the ordering of the sixes, almost on a curve as opposed to a denser shape as with PCA. This space represents the orientation of the ascender just as before, but much more crisply.

Some notes

One lesson that you've probably learned is to use your code and not code from others. A lot of you had trouble getting existing PCA/ICA routines to work, which was a lot more trouble

than getting the three lines of code needed for it. This is because some functions expect the input to be samples × dimensions, and not dimensions × samples as we do here.

Remember that in PCA the synthesis and analysis features are the same, but that's not the case with ICA or NMF. Some of you who used the FastICA plotted the analysis features and not the synthesis ones.

NMF includes a lot of divisions in the iterations. There is always the risk that you will divide by zero. In order to avoid that you can add a small value to the denominator, which would turn a potential zero to a non-zero without appreciably influencing the non-zero cases. This is a good trick to use whenever you run into division-by-zero trouble.