WashburnPaulHW1

September 17, 2018

##
CSCI E-82
##
HW 1 Dimensionality Reduction
###
Due: Sept 17, 2018 11:59pm EST

Note that this is an individual homework to be completed without collaborations except through Piazza.

We encourage you to make progress this weekend since the second homework will likely come out in a week before this one is due.

0.0.1 Your name:

PAUL M. WASHBURN

```
In [1]: import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        import seaborn as sns
        import random
        from sklearn.decomposition import PCA
        %matplotlib inline
        from functools import wraps
        import time
        from sklearn.manifold import MDS
        from sklearn.manifold import TSNE

        def timing_function(some_function):
            '''
            Decorator function.  Outputs the time a function takes to execute.
            '''
            @wraps(some_function)
            def wrapper(*args, **kwargs):
                t1 = time.time()
                result = some_function(*args, **kwargs)
                t2 = time.time()
                time_elapsed = round((t2 - t1), 2)
                print('Runtime: ' + str(time_elapsed) + ' seconds')
                return result

            return wrapper
```

0.0.2 Problem 1 (5 points)

$$\mathbf{X} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

$$\mathbf{Y} = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 1 & 2 \end{bmatrix}$$

Compute XYT. The answer can be computed by hand and written in Markdown like the above matrices, or computed in python. Either way is acceptable.

```
In [2]: X = np.arange(1, 10).reshape(3, 3)
        Y = np.array([[1, 2, 1], [2, 1, 2]])
        np.matmul(X, Y.T)

Out[2]: array([[ 8, 10],
               [20, 25],
               [32, 40]])

In [3]: np.dot(X, Y.T)

Out[3]: array([[ 8, 10],
               [20, 25],
               [32, 40]])
```

0.0.3 Problem 2

This problem goes through a combination of python data manipulations as well as the full math projection using PCA. We have divided the problem into multiple parts.

0.0.4 Problem 2a (5 points)

Download and load in the data set from the UCI archive https://archive.ics.uci.edu/ml/machine-learning-databases/ecoli/. Print the dimensions and the first few rows to demonstrate a successful load.

```
In [4]: # read data from source
        url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/ecoli/ecoli.data'
        colnames = ['sequence_name', 'mcg', 'gvh', 'lip', 'chg', 'aac', 'alm1', 'alm2', 'targe
        df = pd.read_fwf(url, header=None)
        df.columns = colnames
        print(df.head())
        # download
        df.to_csv('data/ecoli_data.csv', index=False)
```

```
  sequence_name   mcg   gvh   lip  chg   aac  alm1  alm2 target
0      AAT_ECOLI  0.49  0.29  0.48  0.5  0.56  0.24  0.35     cp
1     ACEA_ECOLI  0.07  0.40  0.48  0.5  0.54  0.35  0.44     cp
2     ACEK_ECOLI  0.56  0.40  0.48  0.5  0.49  0.37  0.46     cp
3     ACKA_ECOLI  0.59  0.49  0.48  0.5  0.52  0.45  0.36     cp
4      ADI_ECOLI  0.23  0.32  0.48  0.5  0.55  0.25  0.35     cp
```

```
In [5]: # load
        df = pd.read_csv('data/ecoli_data.csv')

        # describe dimensions
        print('rows = %i, columns = %i' %(df.shape[0], df.shape[1]))

        # first five
        df.head()
```

```
rows = 336, columns = 9
```

```
Out[5]:   sequence_name   mcg   gvh   lip  chg   aac  alm1  alm2 target
        0      AAT_ECOLI  0.49  0.29  0.48  0.5  0.56  0.24  0.35     cp
        1     ACEA_ECOLI  0.07  0.40  0.48  0.5  0.54  0.35  0.44     cp
        2     ACEK_ECOLI  0.56  0.40  0.48  0.5  0.49  0.37  0.46     cp
        3     ACKA_ECOLI  0.59  0.49  0.48  0.5  0.52  0.45  0.36     cp
        4      ADI_ECOLI  0.23  0.32  0.48  0.5  0.55  0.25  0.35     cp
```

0.0.5 Problem 2b (10 points)

Compute and print the covariance matrix for all columns excluding the first and last. Rather than use the built-in function, compute this using python code for practice. The following equation will suffice for this.

Cov(X, Y) = (Xi - X) (Yi - Y) / N

```
In [6]: def cov_xy(m, y):
            '''
            Input vectors as 1D arrays
            '''
            X = np.concatenate((m[:,np.newaxis], y[:,np.newaxis]), axis=1)
            N = X.shape[0] - 1
            X -= X.mean(axis=0)
            C = (np.dot(X.T, X.conj()) / N).squeeze()
            return C

        def cov_matrix(df, cols):
```

```
    '''
    Returns covariance matrix with column names
    '''
    # create empty dataframe with variance on the diagonals
    # while preserving names -- leverages pd.DataFrame object
    # df.var() method
    cov_df = pd.DataFrame(np.diag(df[cols].var()), columns=cols, index=cols)

    # populate cov_df empty dataframe with covariances
    for X in cols:
        for Y in cols:
            # we already have the variances, so ignore when X == Y
            if X != Y:
                # populate empty dataframe with symmetrical value
                # that is off the diagonal
                _cov = cov_xy(df[X], df[Y])[0, 1]
                cov_df.loc[Y, X] =  _cov
            else:
                pass
    return cov_df

cols = ['mcg', 'gvh', 'lip', 'chg', 'aac', 'alm1', 'alm2']
C = cov_matrix(df, cols)
C
```

Out[6]:

```
           mcg       gvh       lip       chg       aac      alm1      alm2
mcg   0.037882  0.013115  0.002529  0.000373  0.005257  0.016670  0.006810
gvh   0.013115  0.021950  0.000574  0.000075  0.001266  0.005546 -0.003729
lip   0.002529  0.000574  0.007831  0.000753  0.000760  0.001829 -0.001067
chg   0.000373  0.000075  0.000753  0.000744 -0.000149 -0.000045 -0.000298
aac   0.005257  0.001266  0.000760 -0.000149  0.014976  0.007379  0.006475
alm1  0.016670  0.005546  0.001829 -0.000045  0.007379  0.046549  0.036566
alm2  0.006810 -0.003729 -0.001067 -0.000298  0.006475  0.036566  0.043853
```

In [7]:

```
# make sure above is correct
pd.DataFrame(np.cov(df[cols].T), columns=cols, index=cols)
```

Out[7]:

```
           mcg       gvh       lip       chg       aac      alm1      alm2
mcg   0.037882  0.013115  0.002529  0.000373  0.005257  0.016670  0.006810
gvh   0.013115  0.021950  0.000574  0.000075  0.001266  0.005546 -0.003729
lip   0.002529  0.000574  0.007831  0.000753  0.000760  0.001829 -0.001067
chg   0.000373  0.000075  0.000753  0.000744 -0.000149 -0.000045 -0.000298
aac   0.005257  0.001266  0.000760 -0.000149  0.014976  0.007379  0.006475
alm1  0.016670  0.005546  0.001829 -0.000045  0.007379  0.046549  0.036566
alm2  0.006810 -0.003729 -0.001067 -0.000298  0.006475  0.036566  0.043853
```

In [8]:

```
# double-check to make sure above is correct
print(np.allclose(cov_matrix(df, cols), df[cols].cov()))
print(np.allclose(cov_matrix(df, cols), np.cov(df[cols].T)))
```

```
True
True
```

0.0.6 Problem 2c (10 points).

Compute the decomposition of the covariance matrix using singular value decomposition. Using a python function is definitely the way to go here.

```
In [9]: u, s, v = np.linalg.svd(C)
        print('u = ', u, '\n')
        print('s = ', s, '\n')
        print('v = ', v, '\n')
        print('Note same as covariance matrix:')
        pd.DataFrame(np.dot(u, np.dot(np.diag(s), v)))
```

```
u =  [[-3.41720629e-01 -7.29824958e-01  4.56914809e-01  3.52555665e-01
  -1.29441525e-01  2.62199804e-02 -8.56193962e-03]
 [-9.17492644e-02 -5.28794379e-01 -7.04480258e-01 -3.42514767e-01
  -1.39018617e-01  2.81111220e-01  2.89703065e-04]
 [-1.99698823e-02 -7.25012267e-02  9.24097367e-02 -1.46705018e-02
   8.54762221e-01  4.93627297e-01 -1.06318528e-01]
 [ 9.74209562e-04 -1.16809378e-02  8.81896510e-03  1.64849996e-02
   8.61437517e-02  6.21081678e-02  9.94100049e-01]
 [-1.47115119e-01 -4.80496808e-02  4.70463079e-01 -8.66343630e-01
  -4.86492931e-02 -3.90505852e-02  1.64278490e-02]
 [-6.89914858e-01  7.21275410e-02 -2.54072739e-01 -1.83906883e-02
   3.50569568e-01 -5.75266798e-01  9.64476070e-03]
 [-6.13827312e-01  4.18124394e-01  2.07514962e-02  8.37356323e-02
  -3.17196689e-01  5.83358230e-01 -5.01770938e-03]]

s =  [0.08970253 0.04243901 0.01463252 0.01288363 0.00853362 0.00493527
 0.00065892]

v =  [[-3.41720629e-01 -9.17492644e-02 -1.99698823e-02  9.74209562e-04
  -1.47115119e-01 -6.89914858e-01 -6.13827312e-01]
 [-7.29824958e-01 -5.28794379e-01 -7.25012267e-02 -1.16809378e-02
  -4.80496808e-02  7.21275410e-02  4.18124394e-01]
 [ 4.56914809e-01 -7.04480258e-01  9.24097367e-02  8.81896510e-03
   4.70463079e-01 -2.54072739e-01  2.07514962e-02]
 [ 3.52555665e-01 -3.42514767e-01 -1.46705018e-02  1.64849996e-02
  -8.66343630e-01 -1.83906883e-02  8.37356323e-02]
 [-1.29441525e-01 -1.39018617e-01  8.54762221e-01  8.61437517e-02
  -4.86492931e-02  3.50569568e-01 -3.17196689e-01]
 [ 2.62199804e-02  2.81111220e-01  4.93627297e-01  6.21081678e-02
  -3.90505852e-02 -5.75266798e-01  5.83358230e-01]
 [-8.56193962e-03  2.89703065e-04 -1.06318528e-01  9.94100049e-01
   1.64278490e-02  9.64476070e-03 -5.01770938e-03]]
```

```
Note same as covariance matrix:
```

```
Out[9]:          0         1         2         3         4         5         6
        0  0.037882  0.013115  0.002529  0.000373  0.005257  0.016670  0.006810
        1  0.013115  0.021950  0.000574  0.000075  0.001266  0.005546 -0.003729
        2  0.002529  0.000574  0.007831  0.000753  0.000760  0.001829 -0.001067
        3  0.000373  0.000075  0.000753  0.000744 -0.000149 -0.000045 -0.000298
        4  0.005257  0.001266  0.000760 -0.000149  0.014976  0.007379  0.006475
        5  0.016670  0.005546  0.001829 -0.000045  0.007379  0.046549  0.036566
        6  0.006810 -0.003729 -0.001067 -0.000298  0.006475  0.036566  0.043853
```

0.0.7 Problem 2d (10 points)

Compute the projection of the raw data onto the appropriate two eigenvectors. Consider which columns should be projected and the normalizations.

```
In [10]: # subtract mean
         cols = ['mcg', 'gvh', 'lip', 'chg', 'aac', 'alm1', 'alm2']
         df_0 = df[cols] - df[cols].mean(axis=0)

         # get covariance matrix
         C = cov_matrix(df_0, cols)

         assert np.linalg.det(C) >= 0

         eigenvalues, eigenvectors = np.linalg.eigh(C)
         print('Eigenvalues \n%s\n' %eigenvalues)
         print('Eigenvectors \n%s\n' %eigenvectors)

         for vec in eigenvectors:
             assert np.allclose(1, np.linalg.norm(vec))
         #print(eigenvectors[:,1])

         # sort eigenvalues in desc order
         idx = np.argsort (-eigenvalues)
         eigenvalues = eigenvalues [ idx ]
         eigenvectors = eigenvectors [: , idx ]

         print('eigenvalues = \n', eigenvalues, '\n')
         print('sorted idx = ', idx, '\n')
         print('sorted eigenvectors = ', eigenvectors, '\n')

         print('eigenvectors[:,0] = ', eigenvectors[:,0])
         print('eigenvectors[:,1] = ', eigenvectors[:,1])
```

```
Eigenvalues
[0.00065892 0.00493527 0.00853362 0.01288363 0.01463252 0.04243901
```

```
 0.08970253]

Eigenvectors
[[-8.56193962e-03  2.62199804e-02 -1.29441525e-01  3.52555665e-01
   4.56914809e-01 -7.29824958e-01 -3.41720629e-01]
 [ 2.89703065e-04  2.81111220e-01 -1.39018617e-01 -3.42514767e-01
  -7.04480258e-01 -5.28794379e-01 -9.17492644e-02]
 [-1.06318528e-01  4.93627297e-01  8.54762221e-01 -1.46705018e-02
   9.24097367e-02 -7.25012267e-02 -1.99698823e-02]
 [ 9.94100049e-01  6.21081678e-02  8.61437517e-02  1.64849996e-02
   8.81896510e-03 -1.16809378e-02  9.74209562e-04]
 [ 1.64278490e-02 -3.90505852e-02 -4.86492931e-02 -8.66343630e-01
   4.70463079e-01 -4.80496808e-02 -1.47115119e-01]
 [ 9.64476070e-03 -5.75266798e-01  3.50569568e-01 -1.83906883e-02
  -2.54072739e-01  7.21275410e-02 -6.89914858e-01]
 [-5.01770938e-03  5.83358230e-01 -3.17196689e-01  8.37356323e-02
   2.07514962e-02  4.18124394e-01 -6.13827312e-01]]

eigenvalues =
 [0.08970253 0.04243901 0.01463252 0.01288363 0.00853362 0.00493527
 0.00065892]

sorted idx =  [6 5 4 3 2 1 0]

sorted eigenvectors =  [[-3.41720629e-01 -7.29824958e-01  4.56914809e-01  3.52555665e-01
  -1.29441525e-01  2.62199804e-02 -8.56193962e-03]
 [-9.17492644e-02 -5.28794379e-01 -7.04480258e-01 -3.42514767e-01
  -1.39018617e-01  2.81111220e-01  2.89703065e-04]
 [-1.99698823e-02 -7.25012267e-02  9.24097367e-02 -1.46705018e-02
   8.54762221e-01  4.93627297e-01 -1.06318528e-01]
 [ 9.74209562e-04 -1.16809378e-02  8.81896510e-03  1.64849996e-02
   8.61437517e-02  6.21081678e-02  9.94100049e-01]
 [-1.47115119e-01 -4.80496808e-02  4.70463079e-01 -8.66343630e-01
  -4.86492931e-02 -3.90505852e-02  1.64278490e-02]
 [-6.89914858e-01  7.21275410e-02 -2.54072739e-01 -1.83906883e-02
   3.50569568e-01 -5.75266798e-01  9.64476070e-03]
 [-6.13827312e-01  4.18124394e-01  2.07514962e-02  8.37356323e-02
  -3.17196689e-01  5.83358230e-01 -5.01770938e-03]]

eigenvectors[:,0] =  [-0.34172063 -0.09174926 -0.01996988  0.00097421 -0.14711512 -0.68991486
 -0.61382731]
eigenvectors[:,1] =  [-0.72982496 -0.52879438 -0.07250123 -0.01168094 -0.04804968  0.07212754
  0.41812439]
```

```
In [11]: # get principal component projections
         df_0[cols].dot(eigenvectors)[[0, 1]].T
```

```
Out[11]:          0         1         2         3         4         5         6    \
0   0.285601  0.290838  0.104676  0.087943  0.366268  0.080230  0.385554
1   0.035274  0.330159 -0.015248 -0.122218  0.210366 -0.083273  0.187390

         7         8         9      ...        326       327       328  \
0   0.331901  0.064330  0.371515    ...     0.092863  0.198870  0.087381
1   0.235198  0.283871  0.002960    ...    -0.310786 -0.153296 -0.322197

         329       330       331       332       333       334       335
0  -0.028480 -0.016919 -0.084233  0.139026  0.111904  0.106204 -0.108764
1  -0.262023  0.028185 -0.274803 -0.274116 -0.187103 -0.178851 -0.281129

[2 rows x 336 columns]
```

```
In [12]: # check to make sure PCA results return same-ish as above
         pd.DataFrame(PCA(n_components=2).fit_transform(df[cols])).T
```

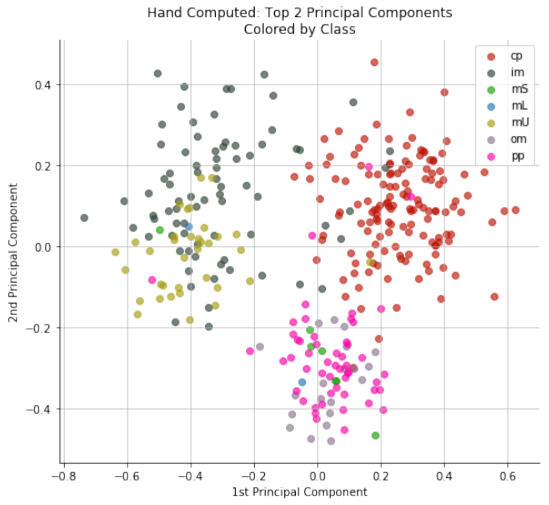
```
Out[12]:        0         1         2         3         4         5         6  \
         0 -0.285601 -0.290838 -0.104676 -0.087943 -0.366268 -0.080230 -0.385554
         1 -0.035274 -0.330159  0.015248  0.122218 -0.210366  0.083273 -0.187390

                7         8         9      ...        326       327       328  \
         0 -0.331901 -0.064330 -0.371515    ...    -0.092863 -0.198870 -0.087381
         1 -0.235198 -0.283871 -0.002960    ...     0.310786  0.153296  0.322197

                329       330       331       332       333       334       335
         0  0.028480  0.016919  0.084233 -0.139026 -0.111904 -0.106204  0.108764
         1  0.262023 -0.028185  0.274803  0.274116  0.187103  0.178851  0.281129

         [2 rows x 336 columns]
```

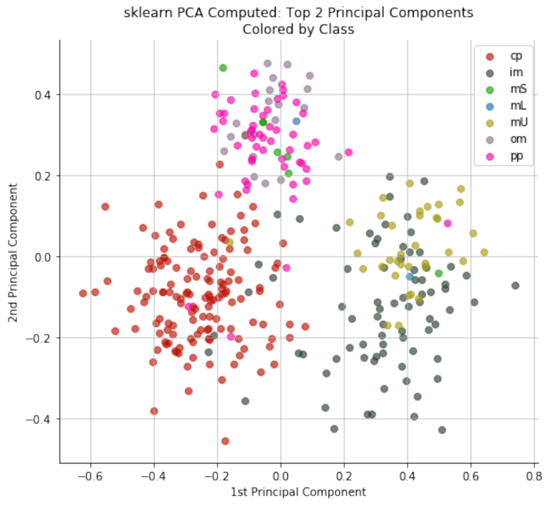
0.0.8 Problem 2e (10 points)

Plot the projected points such that the 8 different classes can be visually identified. Be sure to label the classes and axes. Commont on the quality of the separation of the different classes using PCA.

```
In [13]: def random_hex(seed):
             np.random.seed(seed)
             r = lambda: random.randint(0, 255)
             return '#%02X%02X%02X' % (r(),r(),r())

         # map targets to colors randomly
         unq_tgts = df.target.unique()
         colors = [random_hex(i) for i,cls in zip(range(len(unq_tgts)), unq_tgts)]

         # do projection by hand
         df_pca = df_0[cols].dot(eigenvectors)[[0, 1]]
         df_pca['target'] = df['target']

         # plot the first two PCs
```

```
fig, axes = plt.subplots(1, 2, figsize=(17, 7))

ax = axes[0]
i = 0
for target, _df in df_pca.groupby('target'):
    color = colors[i]
    label = unq_tgts[i]
    ax.scatter(_df[0], _df[1], color=color, label=label, alpha=.65)
    i += 1
ax.grid(alpha=.7)
sns.despine()
ax.set_xlabel('1st Principal Component')
ax.set_ylabel('2nd Principal Component')
ax.legend(loc='best')
ax.set_title('Hand Computed: Top 2 Principal Components\nColored by Class')

# perform PCA using sklearn
pca = PCA(n_components=2, random_state=77)
pca.fit(df[cols])
df_pca = pd.DataFrame(pca.transform(df[cols]))

# add target for plotting
df_pca['target'] = df['target']

# plot the first two PCs
ax = axes[1]
i = 0
for target, _df in df_pca.groupby('target'):
    color = colors[i]
    label = unq_tgts[i]
    ax.scatter(_df[0], _df[1], color=color, label=label, alpha=.65)
    i += 1
ax.grid(alpha=.7)
sns.despine()
ax.set_xlabel('1st Principal Component')
ax.set_ylabel('2nd Principal Component')
ax.legend(loc='best')
ax.set_title('sklearn PCA Computed: Top 2 Principal Components\nColored by Class')
plt.show()
```





The separation of the classes using PCA is noticeable but not stark. Also note how the explicitly computed PCA is a mirror image of the PCA computed with `sklearn.decomposition.PCA`.

0.0.9 Problem 2f (10 points)

The PCA that you have just completed takes each data point and projects it using a weighted sum of features. One could also do the opposite to map the features as a weighted sum of the data entries. How could this be done? What is a potential issue? Describe these in a few sentences (do not code it).

To recover the "original" data after doing principal component analysis we can simply reverse the operations. Since the original projection Y is a linear combination of the mean adjusted original data A and the feature vector F, we can reverse $Y = FA$ using $A = F^T Y$. Finally we would account for the mean by adding it back in. One potential issue of reversing PCA to recover the original data would be non-linearity in the dataset. Since PCA is a linear transformation, any non-linearities in the data will be lost when recovering.

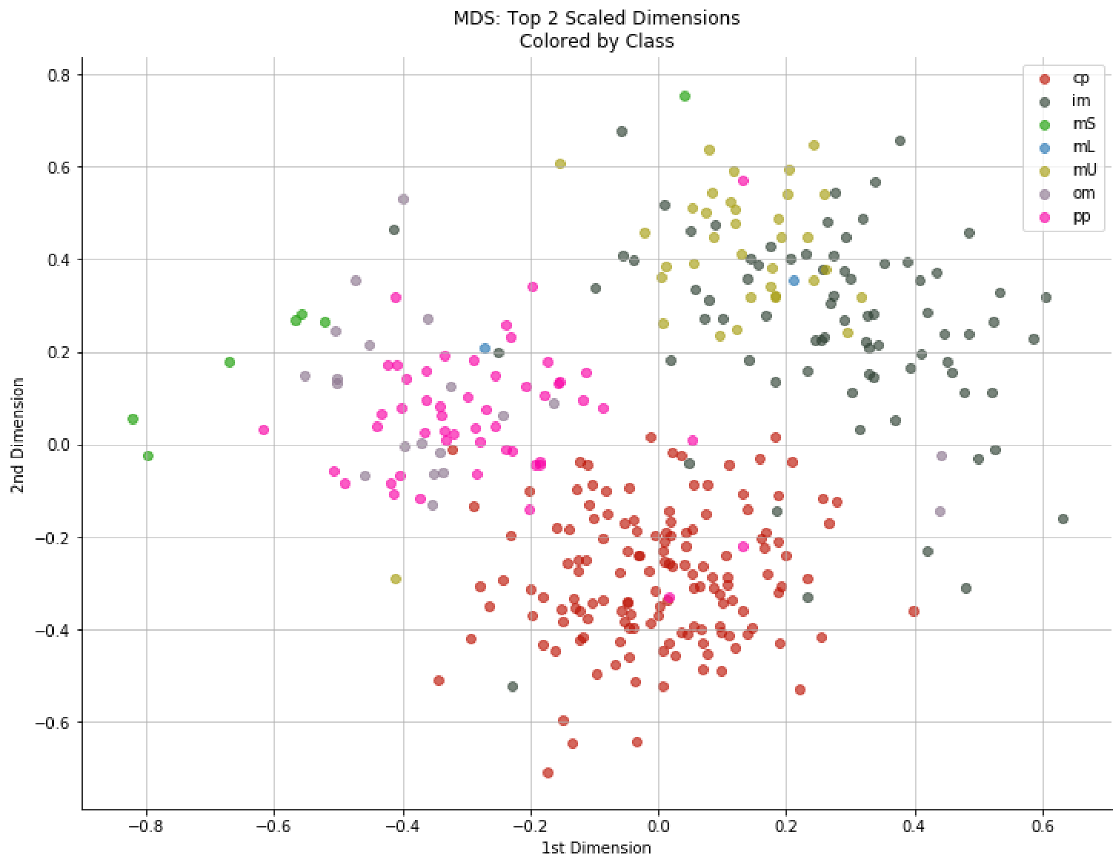
If our goal is to do a projection of features into a lower space weighted by data points, as was clarified on the forums, then we could rearrange the equation above to solve for F such that $Y = A/F^T$.

0.0.10 Problem 3 MDS (10 points)

For the same data set, repeat 2e using sklearn's Multidimensional scaling algorithm.

```
In [16]: # perform PCA using sklearn
         mds = MDS(n_components=2, random_state=77)
         df_mds = pd.DataFrame(mds.fit_transform(df[cols]))

         # add target for plotting
         df_mds['target'] = df['target']

         # plot the first two PCs
```

```
fig, ax = plt.subplots(figsize=(12, 9))
i = 0
for target, _df in df_mds.groupby('target'):
    color = colors[i]
    label = unq_tgts[i]
    ax.scatter(_df[0], _df[1], color=color, label=label, alpha=.65)
    i += 1
ax.grid(alpha=.7)
sns.despine()
ax.set_xlabel('1st Dimension')
ax.set_ylabel('2nd Dimension')
ax.legend(loc='best')
ax.set_title('MDS: Top 2 Scaled Dimensions\nColored by Class')
plt.show()
```

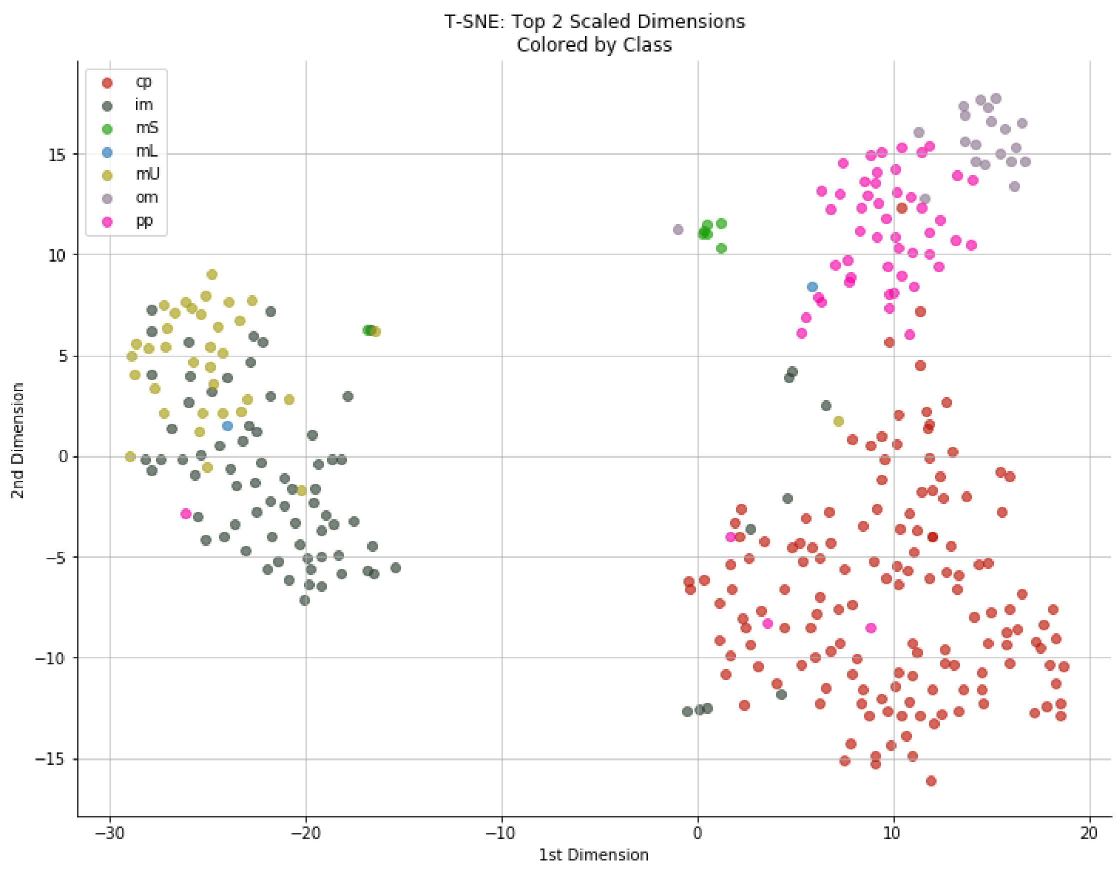


The separation of the classes using `MDS` appears similar to `PCA`: groups are similarly overlapping and are similarly far apart when comparing the two approaches.

0.0.11 Problem 4a t-SNE (5 points)

Repeat 2e using a t-SNE plot with the default settings.

```
In [17]: @timing_function
         def plot_tsne(df, cols, perplexity=30, ax=None, title=None, method='barnes_hut', rand
             '''
             DataFrame must have target column.  Value for perplexity
             defaults to that for sklearn's API. Parameter `cols` must
             be numeric names of DataFrame `df`.
             '''
             # perform TSNE using sklearn
             tsne = TSNE(n_components=2,
                         random_state=random_state,
                         perplexity=perplexity,
                         method=method)
             df_tsne = pd.DataFrame(tsne.fit_transform(df[cols]))

             # add target for plotting
             df_tsne['target'] = df['target']

             # plot the first two vectors
             if ax is None:
                 fig, ax = plt.subplots(figsize=(12, 9))
             i = 0
             for target, _df in df_tsne.groupby('target'):
                 color = colors[i]
                 label = unq_tgts[i]
                 ax.scatter(_df[0], _df[1], color=color, label=label, alpha=.65)
                 i += 1
             ax.grid(alpha=.7)
             sns.despine()
             ax.set_xlabel('1st Dimension')
             ax.set_ylabel('2nd Dimension')
             ax.legend(loc='best')
             if title is None:
                 ax.set_title('T-SNE: Top 2 Scaled Dimensions\nColored by Class')
             else:
                 ax.set_title(title)
             if ax is None:
                 plt.show()

         plot_tsne(df, cols, perplexity=30)

Runtime: 5.85 seconds
```



T-SNE, even with default settings, clearly does a much better job than both `PCA` and `MDS` of separating classes while still preserving the intra-group proximity between points.

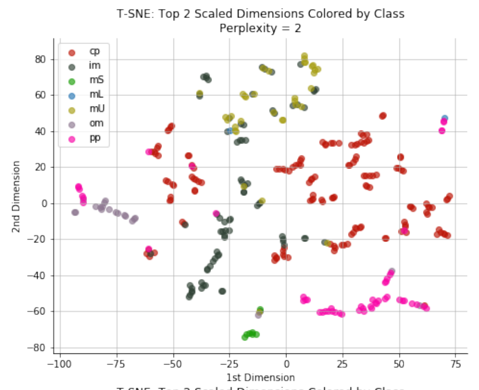
0.0.12 Problem 4b t-SNE perplexity (5 points)

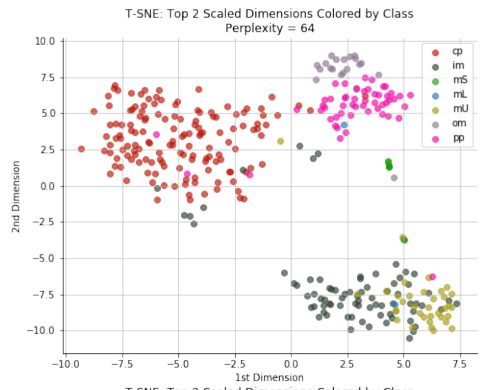
Try out a few t-SNE plots by varying the perplexity. State the best perplexity for separating the 8 different classes and describe your rationale in a sentence or two. Report the average calculation time for the t-SNE projection over a number of iterations.

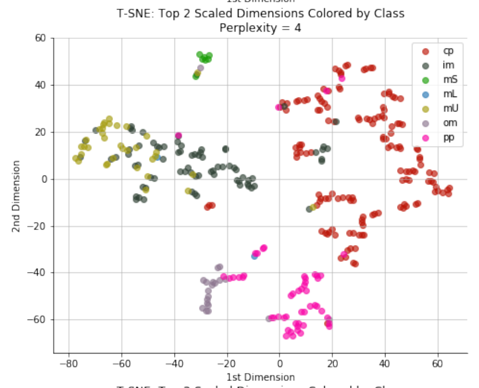
```
In [18]: # use %timemit
         perplexities = [2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
         nrows = len(perplexities)//2
         height = 7*nrows
         fig, axes = plt.subplots(nrows, 2, figsize=(17, height))
         for p, perplexity in enumerate(perplexities):
             title = 'T-SNE: Top 2 Scaled Dimensions Colored by Class\nPerplexity = {}'.format
             if p < nrows:
                 ax = axes[p][0]
             else:
                 ax = axes[p-nrows][1]
             print('Running t-SNE @perplexity = %i ' %perplexity)
```

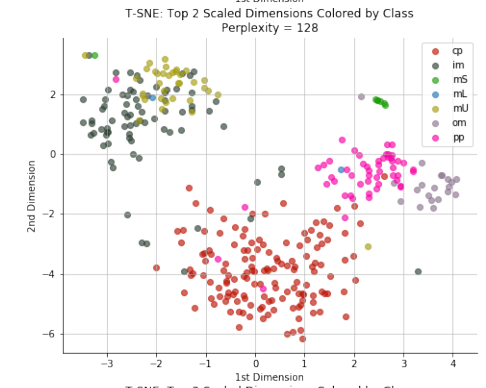
```
            plot_tsne(df, cols, perplexity=perplexity, ax=ax, title=title)
            print('')
        plt.show()
```

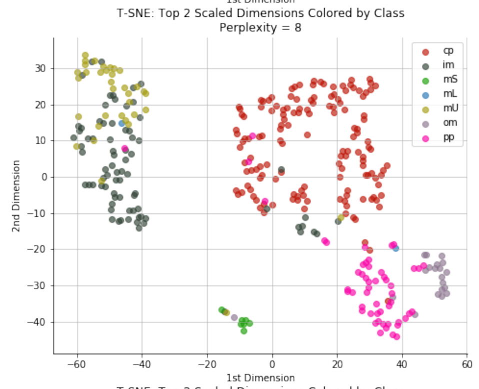
```
Running t-SNE @perplexity = 2
Runtime: 2.37 seconds

Running t-SNE @perplexity = 4
Runtime: 2.5 seconds

Running t-SNE @perplexity = 8
Runtime: 3.12 seconds

Running t-SNE @perplexity = 16
Runtime: 4.13 seconds

Running t-SNE @perplexity = 32
Runtime: 5.64 seconds

Running t-SNE @perplexity = 64
Runtime: 7.71 seconds

Running t-SNE @perplexity = 128
Runtime: 9.55 seconds

Running t-SNE @perplexity = 256
Runtime: 10.71 seconds

Running t-SNE @perplexity = 512
Runtime: 6.1 seconds

Running t-SNE @perplexity = 1024
Runtime: 6.0 seconds
```

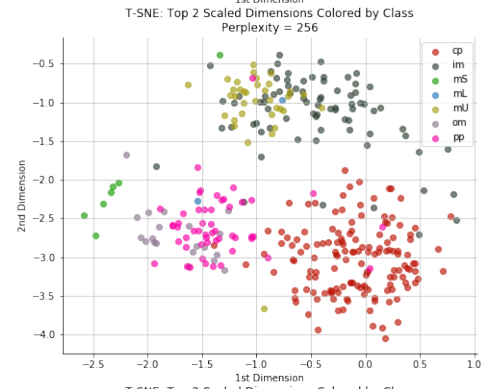

T-SNE: Top 2 Scaled Dimensions Colored by Class
Perplexity = 2
cp
im
mS
mL
mU
om
pp
1st Dimension
2nd Dimension

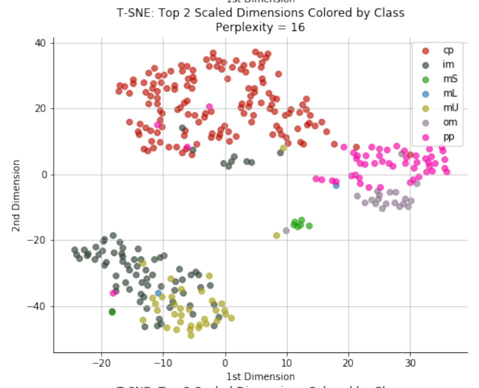

T-SNE: Top 2 Scaled Dimensions Colored by Class
Perplexity = 64
cp
im
mS
mL
mU
om
pp
1st Dimension
2nd Dimension

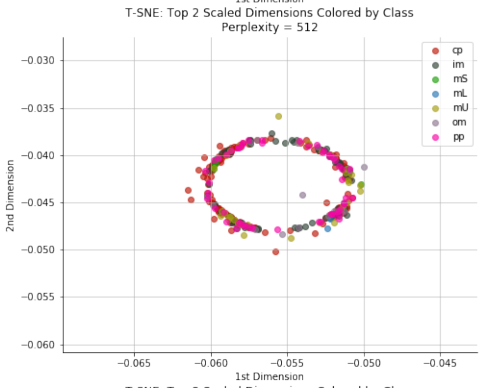

T-SNE: Top 2 Scaled Dimensions Colored by Class
Perplexity = 4
cp
im
mS
mL
mU
om
pp
1st Dimension
2nd Dimension

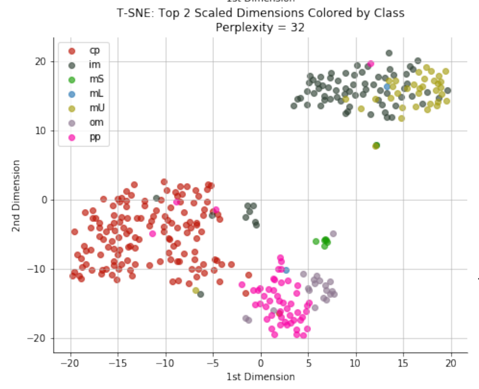

T-SNE: Top 2 Scaled Dimensions Colored by Class
Perplexity = 128
cp
im
mS
mL
mU
om
pp
1st Dimension
2nd Dimension

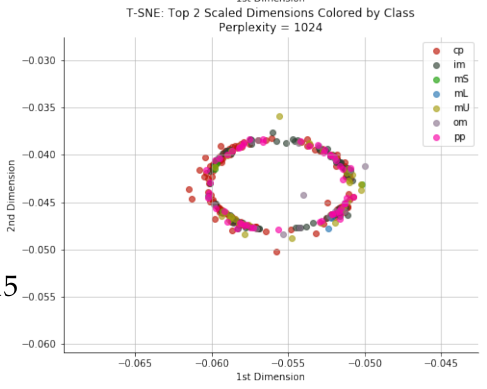

T-SNE: Top 2 Scaled Dimensions Colored by Class
Perplexity = 8
cp
im
mS
mL
mU
om
pp
1st Dimension
2nd Dimension


T-SNE: Top 2 Scaled Dimensions Colored by Class
Perplexity = 256
cp
im
mS
mL
mU
om
pp
1st Dimension
2nd Dimension


T-SNE: Top 2 Scaled Dimensions Colored by Class
Perplexity = 16
cp
im
mS
mL
mU
om
pp
1st Dimension
2nd Dimension


T-SNE: Top 2 Scaled Dimensions Colored by Class
Perplexity = 512
cp
im
mS
mL
mU
om
pp
1st Dimension
2nd Dimension


T-SNE: Top 2 Scaled Dimensions Colored by Class
Perplexity = 32
cp
im
mS
mL
mU
om
pp
1st Dimension
2nd Dimension


T-SNE: Top 2 Scaled Dimensions Colored by Class
Perplexity = 1024
cp
im
mS
mL
mU
om
pp
1st Dimension
2nd Dimension

```
In [23]: times = {2: 2.37, 4: 2.5, 8: 3.12, 16: 4.13, 32: 5.64, 64: 7.71, 128: 9.55, 256: 10.7
         avg_time = pd.Series(times).mean()
         print('Average runtime = %.4f seconds' %avg_time)
```

```
Average runtime = 5.7589 seconds
```

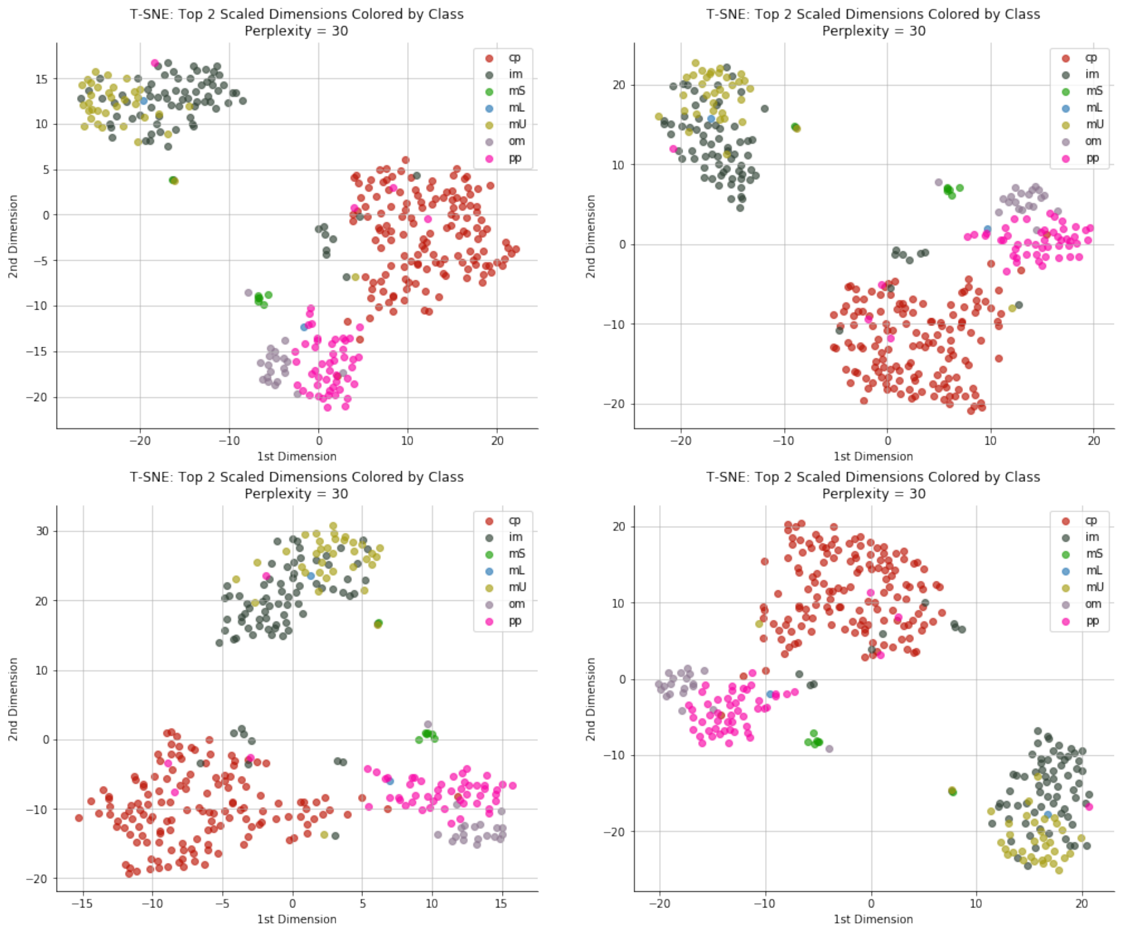
While the approach is imperfect, a `perplexity` value of 32 appears to have the strongest separation between groups while still preserving local proximities within groups. This is visually apparent in the plot on the lower left where `perplexity=32`: we see that the red cluster is more dense than at lower `perplexity` values, yet the distance from other dissimilar groups is preserved. Beyond `perplexity=64` we observe that the clusters first begin to grow more chaotic before finally conveging into some elliptical-like shape.

0.0.13 Problem 4c t-SNE randomization (10 points)

The S of t-SNE means stochastic or random, usually as a function of time. Explore whether you can reproduce the result in 4b through a second projection and plot.

```
In [19]: perplexities = [30 for i in range(4)] # use default 4 separate times
         nrows = len(perplexities)//2
         height = 7*nrows
         fig, axes = plt.subplots(nrows, 2, figsize=(17, height))
         for p, perplexity in enumerate(perplexities):
             title = 'T-SNE: Top 2 Scaled Dimensions Colored by Class\nPerplexity = {}'.format
             if p < nrows:
                 ax = axes[p][0]
             else:
                 ax = axes[p-nrows][1]
             print('Running t-SNE @perplexity = %i ' %perplexity)
             np.random.seed(p)
             plot_tsne(df, cols, perplexity=perplexity, ax=ax, title=title, random_state=p)
             print('')
         plt.show()
```

```
Running t-SNE @perplexity = 30
Runtime: 6.76 seconds

Running t-SNE @perplexity = 30
Runtime: 6.62 seconds

Running t-SNE @perplexity = 30
Runtime: 6.27 seconds

Running t-SNE @perplexity = 30
Runtime: 6.69 seconds
```



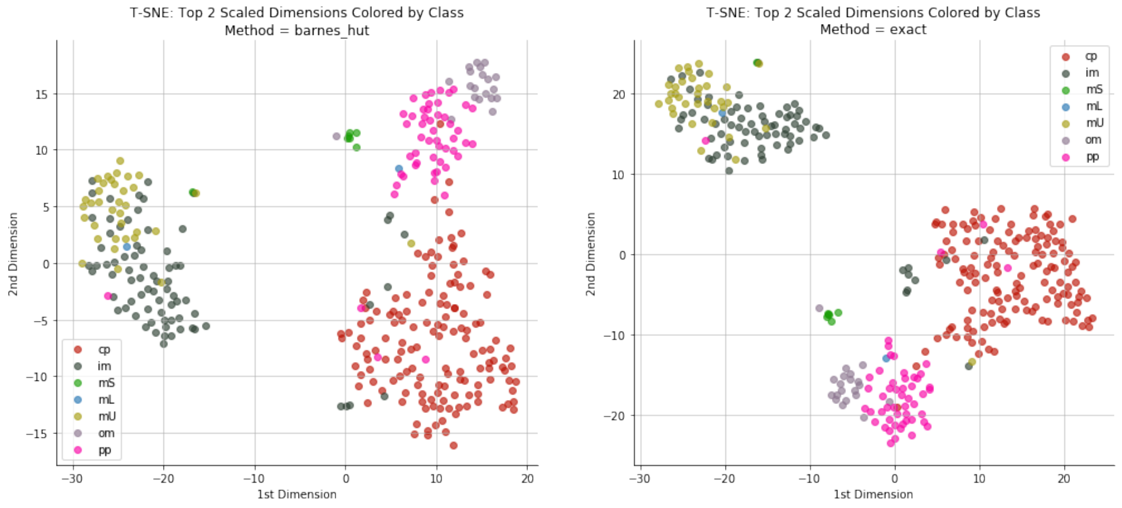
We see above that varying the `random_seed` four separate times yields four unique plots. While the plots are similar (i.e. they distinguish between groups in a similar way, by preserving distances between groups adn within) they are clearly different when different seeds are set. This is due to the fact that stochastic methods are used to perform t-SNE, and the initial randomization leads to different results.

0.0.14 Problem 4d t-SNE Barnes-Hut (5 points)

The default t-SNE method of 4b uses the Barnes-Hut approximation. Keeping the other parameters the same as 4b, plot the t-SNE result using the exact method. Which method do you prefer? Compare the average calculation time for the exact method over a number of iterations.

```
In [24]: perplexities = [30 for i in range(2)] # use default 2 separate times
         methods = ['barnes_hut', 'exact']
         fig, axes = plt.subplots(1, 2, figsize=(17, 7))
         for p, perplexity in enumerate(perplexities):
             title = 'T-SNE: Top 2 Scaled Dimensions Colored by Class\nMethod = {}'.format(meth
             ax = axes[p]
             print('Running t-SNE @method = %s ' %methods[p])
             plot_tsne(df, cols, perplexity=perplexity, ax=ax, title=title, method=methods[p])
             print('')
         plt.show()
```

```
Running t-SNE @method = barnes_hut
Runtime: 5.69 seconds

Running t-SNE @method = exact
Runtime: 4.03 seconds
```



```
In [27]: def run_tsne(method):
             # perform TSNE using sklearn
             # vary method to compare times
             tsne = TSNE(n_components=2,
                         method=method)
             df_tsne = pd.DataFrame(tsne.fit_transform(df[cols]))
             return None

         %timeit run_tsne('barnes_hut')
         %timeit run_tsne('exact')
```

```
5.71 s ś 161 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)
3.93 s ś 65.2 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)
```

The plots using the two methods appear to highlight the same clusters as one another, yet the average runtime for the `method='exact'` approach (3.93 seconds)9 goes faster on average (and with a tighter distribution) than `method='barnes_hut'` (5.71 seconds). The exact approach appears to be the better way to go, at least w.r.t. computation time.

0.0.15 How many hours did this homework take?

This will not affect your grade. We will be monitoring time spent on homework to be sure that we are not over-burdening students.

First pass, roughly 6 hours. Making sure all is good for submission, roughly 9 hours.

0.0.16 Last step (5 points)

Save this notebook as LastnameFirstnameHW1.ipynb such as MuskElonHW1.ipynb. Create a pdf of this notebook named similarly. Submit both the python notebook and the pdf version to the Canvas dropbox. We require both versions.

```
In [28]: print('OK.')
```

```
OK.
```