

# Principal Component Analysis (PCA)

We will implement the PCA algorithm. We will first implement PCA, then apply it (once again) to the MNIST digit dataset.

## Learning objective

1. Write code that implements PCA.
2. Write code that implements PCA for high-dimensional datasets

Let's first import the packages we need for this week.

```
In [2]: # PACKAGE: DO NOT EDIT THIS CELL
import numpy as np
import timeit
```

```
In [3]: # PACKAGE: DO NOT EDIT THIS CELL
import matplotlib as mpl
mpl.use('Agg')
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
from ipywidgets import interact
```

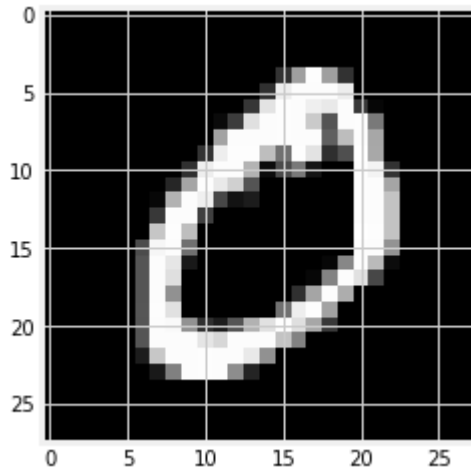
```
In [4]: from load_data import load_mnist

MNIST = load_mnist()
images, labels = MNIST['data'], MNIST['target']
```

```
In [5]: %matplotlib inline
```

Now, let's plot a digit from the dataset:

```
In [6]: plt.figure(figsize=(4,4))
plt.imshow(images[0].reshape(28,28), cmap='gray');
```



Before we implement PCA, we will need to do some data preprocessing. In this assessment, some of them will be implemented by you, others we will take care of. However, when you are working on real world problems, you will need to do all these steps by yourself!

The preprocessing steps we will do are

1. Convert unsigned integer 8 (uint8) encoding of pixels to a floating point number between 0-1.
2. Subtract from each image the mean  $\mu$ .
3. Scale each dimension of each image by  $\frac{1}{\sigma}$  where  $\sigma$  is the standard deviation.

The steps above ensure that our images will have zero mean and one variance. These preprocessing steps are also known as Data Normalization or Feature Scaling ([https://en.wikipedia.org/wiki/Feature\\_scaling](https://en.wikipedia.org/wiki/Feature_scaling)).

## 1. PCA

Now we will implement PCA. Before we do that, let's pause for a moment and think about the steps for performing PCA. Assume that we are performing PCA on some dataset  $\mathbf{X}$  for  $M$  principal components. We then need to perform the following steps, which we break into parts:

1. Data normalization (normalize).
2. Find eigenvalues and corresponding eigenvectors for the covariance matrix  $S$ . Sort by the largest eigenvalues and the corresponding eigenvectors (eig).

After these steps, we can then compute the projection and reconstruction of the data onto the space spanned by the top  $n$  eigenvectors.

In [7]: *# GRADED FUNCTION: DO NOT EDIT THIS LINE*

```
def normalize(X):
    """Normalize the given dataset X
    Args:
        X: ndarray, dataset

    Returns:
        (Xbar, mean, std): tuple of ndarray, Xbar is the normalized dataset
        with mean 0 and standard deviation 1; mean and std are the
        mean and standard deviation respectively.

    Note:
        You will encounter dimensions where the standard deviation is
        zero, for those when you do normalization the normalized data
        will be NaN. Handle this by setting using `std = 1` for those
        dimensions when doing normalization.
    """
    mu = np.mean(X,axis=0) # <-- EDIT THIS, compute the mean of X
    std = np.std(X, axis=0)
    std_filled = std.copy()
    std_filled[std==0] = 1.
    Xbar = (X - mu)/std_filled # <-- EDIT THIS, compute the normalized
data Xbar
    return Xbar, mu, std_filled

def eig(S):
    """Compute the eigenvalues and corresponding eigenvectors
    for the covariance matrix S.
    Args:
        S: ndarray, covariance matrix

    Returns:
        (eigvals, eigvecs): ndarray, the eigenvalues and eigenvectors

    Note:
        the eigenvals and eigenvecs should be sorted in descending
        order of the eigen values
    """
    eigen_values, eigen_vectors = np.linalg.eig(S)
    idx = eigen_values.argsort()[::-1]
    eigen_values = eigen_values[idx]
    eigen_vectors = eigen_vectors[:,idx]
    return (eigen_values, eigen_vectors) # <-- EDIT THIS to return the eigenvalues and corresponding eigenvectors

def projection_matrix(B):
    """Compute the projection matrix onto the space spanned by `B`
    Args:
        B: ndarray of dimension (D, M), the basis for the subspace

    Returns:
        P: the projection matrix
    """
    #formula on book only works if they're orthogonal, if they aren't, the more general one used before is needed
```

```

    return (B@(np.linalg.inv(B.T@B))@B.T) # <-- EDIT THIS to compute the projection matrix

def PCA(X, num_components):
    """
    Args:
        X: ndarray of size (N, D), where D is the dimension of the data,
           and N is the number of datapoints
        num_components: the number of principal components to use.
    Returns:
        X_reconstruct: ndarray of the reconstruction
                       of X from the first `num_components` principal components.
    """
    X, mean, std = normalize(X)
    S = np.cov(X, rowvar=False, bias=True)
    eig_vals, eig_vecs = eig(S)
    P = projection_matrix(eig_vecs[:, :num_components])
    X = (P @ X.T).T # your solution should take advantage of the functions
                    # you have implemented above.
    return X # <-- EDIT THIS to return the reconstruction of X

```

```

In [8]: ## Some preprocessing of the data
NUM_DATAPOINTS = 1000
X = (images.reshape(-1, 28 * 28)[:NUM_DATAPOINTS]) / 255.
Xbar, mu, std = normalize(X)

```

```

In [10]: for num_component in range(1, 20):
          from sklearn.decomposition import PCA as SKPCA
          # We can compute a standard solution given by scikit-learn's implementation of PCA
          pca = SKPCA(n_components=num_component, svd_solver='full')
          sklearn_reconst = pca.inverse_transform(pca.fit_transform(Xbar))
          reconst = PCA(Xbar, num_component)
          np.testing.assert_almost_equal(reconst, sklearn_reconst)
          print(np.square(reconst - sklearn_reconst).sum())

```

```

(8.35369033929e-24+0j)
(9.21103380181e-24+0j)
(1.22493094923e-23+0j)
(1.14593249289e-23+0j)
(1.33847906792e-23+0j)
(1.42708325423e-23+0j)
(2.09569958253e-23+0j)
(2.55478426206e-23+0j)
(3.02773708717e-23+0j)
(2.8646702889e-22+0j)
(1.32364609416e-22+0j)
(9.24583956264e-23+0j)
(1.61042349997e-20+0j)
(7.36449101244e-23+0j)
(3.66168498364e-23+0j)
(4.25613852911e-23+0j)
(3.66128801246e-23+0j)
(4.67823296081e-23+0j)
(6.24191871076e-23+0j)

```

The greater number of principal components we use, the smaller will our reconstruction error be. Now, let's answer the following question:

How many principal components do we need in order to reach a Mean Squared Error (MSE) of less than 100 for our dataset?

We have provided a function in the next cell which computes the mean squared error (MSE), which will be useful for answering the question above.

```
In [15]: def mse(predict, actual):
         """Helper function for computing the mean squared error (MSE)"""
         return np.square(predict - actual).sum(axis=1).mean()
```

```
In [16]: loss = []
         reconstructions = []
         # iterate over different number of principal components, and compute the MSE
         for num_component in range(1, 100):
             reconst = PCA(Xbar, num_component)
             error = mse(reconst, Xbar)
             reconstructions.append(reconst)
             # print('n = {:d}, reconstruction_error = {:.f}'.format(num_component, error))
             loss.append((num_component, error))

         reconstructions = np.asarray(reconstructions)
         reconstructions = reconstructions * std + mu # "unnormalize" the reconstructed image
         loss = np.asarray(loss)
```

```
In [17]: import pandas as pd
         # create a table showing the number of principal components and MSE
         pd.DataFrame(loss).head()
```

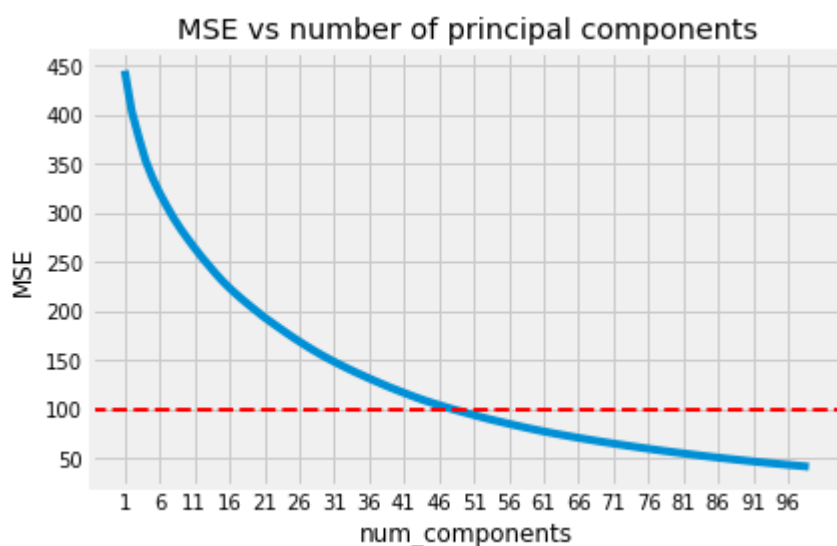
Out[17]:

	0	1
0	(1+0j)	(445.176488428+0j)
1	(2+0j)	(403.941979232+0j)
2	(3+0j)	(377.592430651+0j)
3	(4+0j)	(353.300124337+0j)
4	(5+0j)	(335.44289452+0j)

We can also put these numbers into perspective by plotting them.

```
In [18]: fig, ax = plt.subplots()
ax.plot(loss[:,0], loss[:,1]);
ax.axhline(100, linestyle='--', color='r', linewidth=2)
ax.xaxis.set_ticks(np.arange(1, 100, 5));
ax.set(xlabel='num_components', ylabel='MSE', title='MSE vs number of principal components');
```

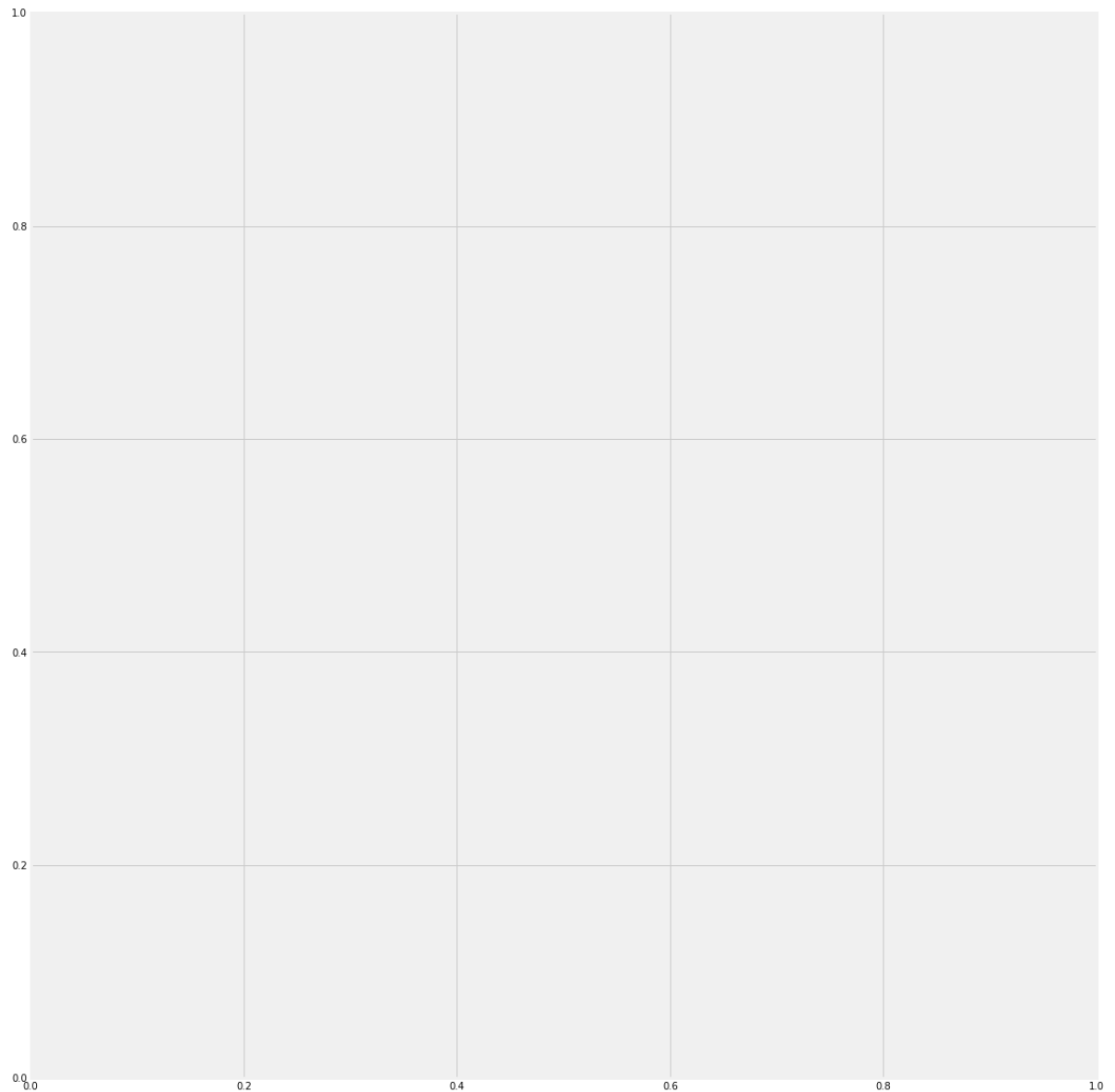
/opt/conda/lib/python3.6/site-packages/numpy/core/numeric.py:531: ComplexWarning: Casting complex values to real discards the imaginary part  
 return array(a, dtype, copy=False, order=order)



But *numbers don't tell us everything!* Just what does it mean *qualitatively* for the loss to decrease from around 450.0 to less than 100.0?

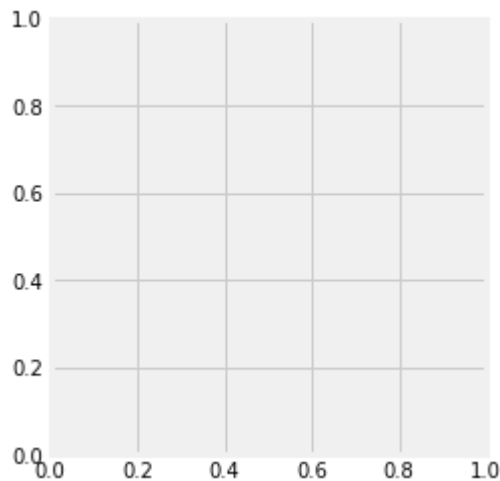
Let's find out! In the next cell, we draw the the leftmost image is the original digit. Then we show the reconstruction of the image on the right, in descending number of principal components used.

```
In [23]: @interact(image_idx=(0, 1000))
def show_num_components_reconst(image_idx):
    fig, ax = plt.subplots(figsize=(20., 20.))
    actual = X[image_idx]
    # concatenate the actual and reconstructed images as large image before plotting it
    x = np.concatenate([actual[np.newaxis, :], reconstructions[:, image_idx]])
    ax.imshow(np.hstack(x.reshape(-1, 28, 28)[np.arange(10)]), cmap='gray');
    ax.axvline(28, color='orange', linewidth=2)
```



We can also browse through the reconstructions for other digits. Once again, `interact` becomes handy for visualizing the reconstruction.

```
In [20]: @interact(i=(0, 10))
def show_pca_digits(i=1):
    """Show the i th digit and its reconstruction"""
    plt.figure(figsize=(4,4))
    actual_sample = X[i].reshape(28,28)
    reconst_sample = (reconst[i, :] * std + mu).reshape(28, 28)
    plt.imshow(np.hstack([actual_sample, reconst_sample]), cmap='gray')
    plt.show()
```



## 2. PCA for high-dimensional datasets

Sometimes, the dimensionality of our dataset may be larger than the number of samples we have. Then it might be inefficient to perform PCA with your implementation above. Instead, as mentioned in the lectures, you can implement PCA in a more efficient manner, which we call "PCA for high dimensional data" (PCA\_high\_dim).

Below are the steps for performing PCA for high dimensional dataset

1. Compute the matrix  $XX^T$  (a  $N$  by  $N$  matrix with  $N \ll D$ )
2. Compute eigenvalues  $\lambda$ s and eigenvectors  $V$  for  $XX^T$
3. Compute the eigenvectors for the original covariance matrix as  $X^T V$ . Choose the eigenvectors associated with the  $M$  largest eigenvalues to be the basis of the principal subspace  $U$ .
4. Compute the orthogonal projection of the data onto the subspace spanned by columns of  $U$ . Functions you wrote for earlier assignments will be useful.



```
In [9]: # GRADED FUNCTION: DO NOT EDIT THIS LINE
        ### PCA for high dimensional datasets

        def PCA_high_dim(X, n_components):
            """Compute PCA for small sample size but high-dimensional features.
            Args:
                X: ndarray of size (N, D), where D is the dimension of the sample,
                   and N is the number of samples
                num_components: the number of principal components to use.
            Returns:
                X_reconstruct: (N, D) ndarray. the reconstruction
                               of X from the first `num_components` principal components.
            """
            X, mean, std = normalize(X)
            S = np.cov(X, rowvar=False, bias=True)
            eig_vals, eig_vecs = eig(S)
            P = projection_matrix(eig_vecs[:, :n_components])
            X = (P @ X.T).T
            return X # <-- EDIT THIS to return the reconstruction of X
```

Given the same dataset, PCA\_high\_dim and PCA should give the same output. Assuming we have implemented PCA, correctly, we can then use PCA to test the correctness of PCA\_high\_dim. Given the same dataset, PCA and PCA\_high\_dim should give identical results.

We can use this **invariant** to test our implementation of PCA\_high\_dim, assuming that we have correctly implemented PCA.

```
In [10]: np.testing.assert_almost_equal(PCA(Xbar, 2), PCA_high_dim(Xbar, 2))
```

Now let's compare the running time between PCA and PCA\_high\_dim.

**Tips** for running benchmarks or computationally expensive code:

When you have some computation that takes up a non-negligible amount of time. Try separating the code that produces output from the code that analyzes the result (e.g. plot the results, compute statistics of the results). In this way, you don't have to recompute when you want to produce more analysis.

The next cell includes a function that records the time taken for executing a function *f* by repeating it for repeat number of times. You do not need to modify the function but you can use it to compare the running time for functions which you are interested in knowing the running time.

```
In [12]: def time(f, repeat=10):
    times = []
    for _ in range(repeat):
        start = timeit.default_timer()
        f()
        stop = timeit.default_timer()
        times.append(stop-start)
    return np.mean(times), np.std(times)
```

We first benchmark the time taken to compute  $X^T X$  and  $XX^T$ . Jupyter's magic command %time is quite handy.

The next cell finds the running time for computing  $X^T X$  and  $XX^T$  for different dimensions of  $X$ .

```
In [18]: times_mm0 = []
    times_mm1 = []

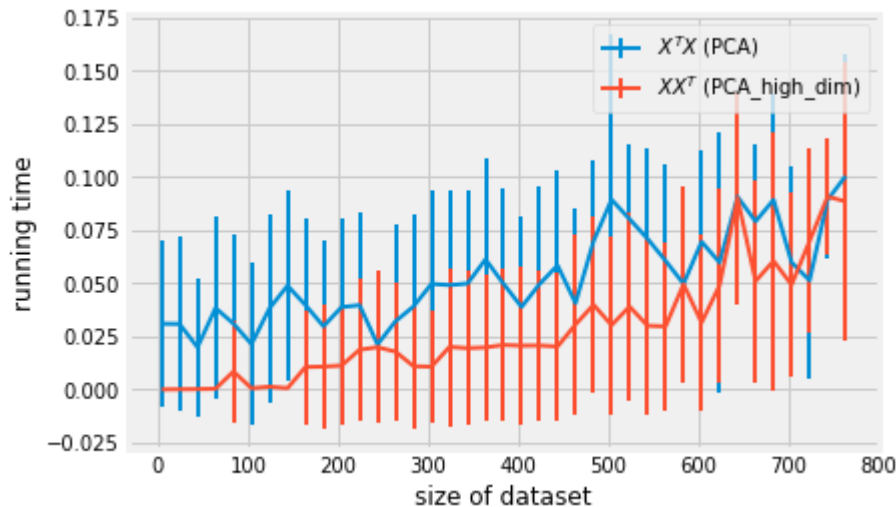
    # iterate over datasets of different size
    for datasetsize in np.arange(4, 784, step=20):
        XX = Xbar[:datasetsize] # select the first `datasetsize` samples in the dataset
        # record the running time for computing  $X.T @ X$ 
        mu, sigma = time(lambda : XX.T @ XX)
        times_mm0.append((datasetsize, mu, sigma))

        # record the running time for computing  $X @ X.T$ 
        mu, sigma = time(lambda : XX @ XX.T)
        times_mm1.append((datasetsize, mu, sigma))

    times_mm0 = np.asarray(times_mm0)
    times_mm1 = np.asarray(times_mm1)
```

Having recorded the running time for computing  $X @ X.T$  and  $X @ X.T$ , we can plot them.

```
In [19]: fig, ax = plt.subplots()
ax.set(xlabel='size of dataset', ylabel='running time')
bar = ax.errorbar(times_mm0[:, 0], times_mm0[:, 1], times_mm0[:, 2], label="$X^T X$ (PCA)", linewidth=2)
ax.errorbar(times_mm1[:, 0], times_mm1[:, 1], times_mm1[:, 2], label="$X X^T$ (PCA_high_dim)", linewidth=2)
ax.legend();
```



Alternatively, use the time magic command for benchmarking functions.

```
In [20]: %time Xbar.T @ Xbar
%time Xbar @ Xbar.T
pass # Put this here so that our output does not show result of computing `Xbar @ Xbar.T`

CPU times: user 64.8 ms, sys: 176 ms, total: 241 ms
Wall time: 133 ms
CPU times: user 49.8 ms, sys: 97.6 ms, total: 147 ms
Wall time: 7.45 ms
```

Next we benchmark PCA, PCA\_high\_dim.

```

In [13]: times0 = []
         times1 = []

         # iterate over datasets of different size
         for datasetsize in np.arange(4, 784, step=100):
             XX = Xbar[:datasetsize]
             npc = 2
             mu, sigma = time(lambda : PCA(XX, npc), repeat=10)
             times0.append((datasetsize, mu, sigma))

             mu, sigma = time(lambda : PCA_high_dim(XX, npc), repeat=10)
             times1.append((datasetsize, mu, sigma))

         times0 = np.asarray(times0)
         times1 = np.asarray(times1)

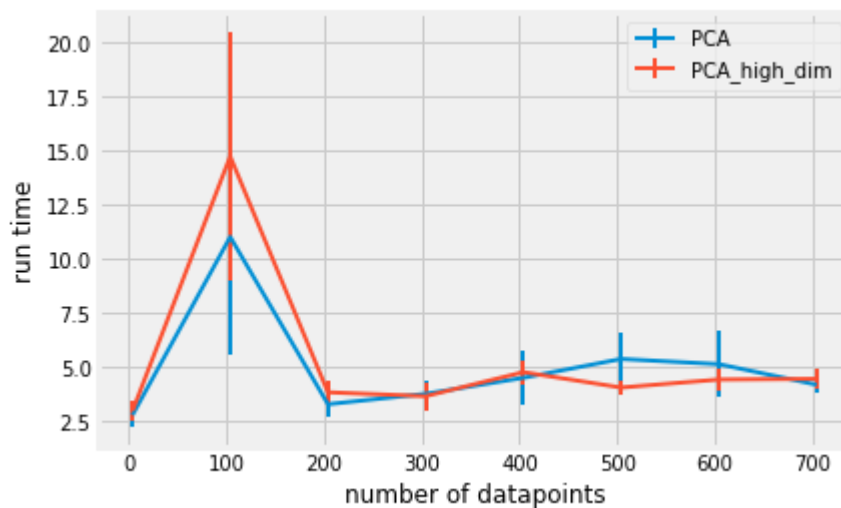
```

Let's plot the running time. Spend some time and think about what this plot means. We mentioned in lectures that PCA\_high\_dim are advantageous when we have dataset size  $N < \text{data dimension } M$ . Although our plot does not for the two running time does not intersect exactly at  $N = M$ , it does show the trend.

```

In [14]: fig, ax = plt.subplots()
         ax.set(xlabel='number of datapoints', ylabel='run time')
         ax.errorbar(times0[:, 0], times0[:, 1], times0[:, 2], label="PCA", linewidth=2)
         ax.errorbar(times1[:, 0], times1[:, 1], times1[:, 2], label="PCA_high_dim", linewidth=2)
         ax.legend();

```



Again, with the magic command time.

```
In [23]: %time PCA(Xbar, 2)
          %time PCA_high_dim(Xbar, 2)
          pass
```

CPU times: user 10.6 s, sys: 28.5 s, total: 39.1 s

Wall time: 19.5 s

CPU times: user 11 s, sys: 29.9 s, total: 40.9 s

Wall time: 20.4 s