Dimensionality Reduction

Loyola Marymount University
CMSI-533: Data Science and Machine Learning
Alex Wong

alexw@cs.ucla.edu

The Curse of Dimensionality

Let’s consider a dataset with 3 features

$$\mathbf{x} = (x_1, x_2, x_3)$$

and each feature can be one of 100 discrete values

$$x \in \{1, 2, 3, \ldots, 100\}$$

So each observation or example will live in the each space of

$$\mathbf{x} \in \{1, 2, 3, \ldots, 100\}^3$$

This is a space of 1 million discrete cells shaped in a cube

The Curse of Dimensionality

Now let's consider another dataset with 14 features

$$\mathbf{x} = (x_1, x_2, x_3, .., x_{14})$$

and each feature still can be one of 100 discrete values

$$x \in \{1, 2, 3, \ldots, 100\}$$

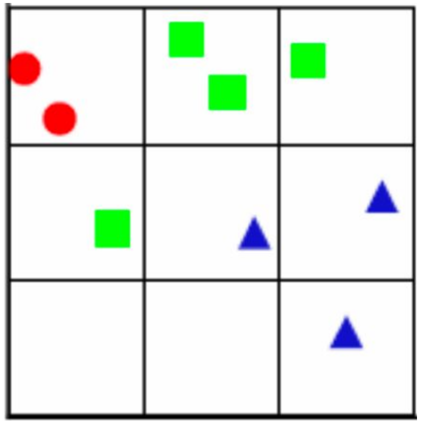
The size of our feature space now grows to 10,000,000,000,000,000,000,000,000,000

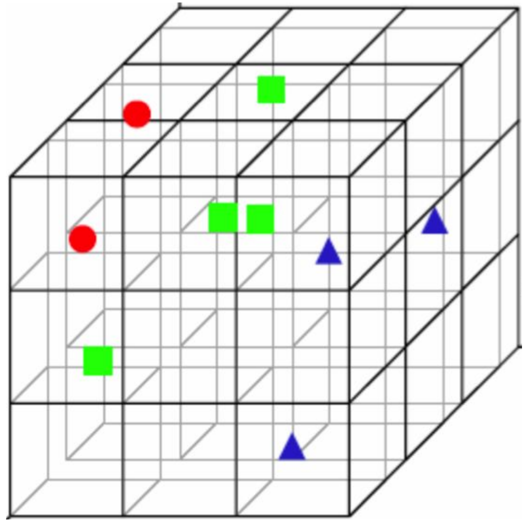
Rarely will we have this many observations
So there won't be any observations in most of the feature space

The Curse of Dimensionality

The volume of the space increases so fast that the available data become sparse

In the case of classification, because observations are sparse, each observation will seem dissimilar despite belonging to the same class



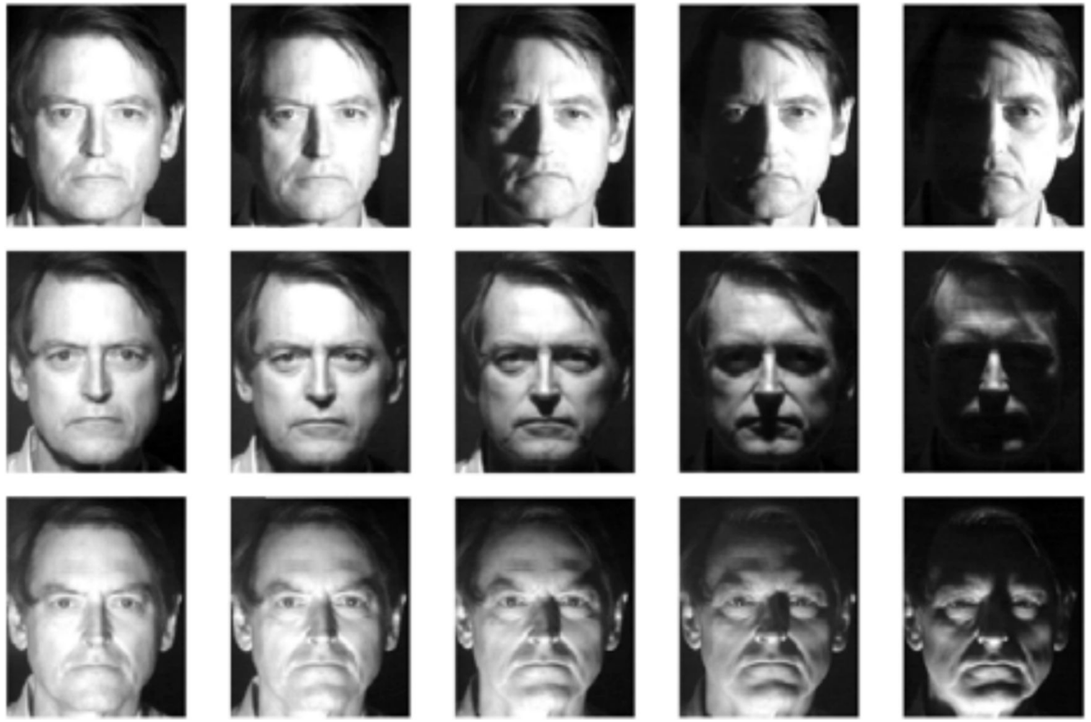


What Dimensions Are Necessary?

Here we have a dataset of a face taken under different illumination settings

Each image is 128 x 128 pixels, effectively, 16384 dimensions

Suppose we want to model the illumination changes, how many dimensions do we really need?



Dimensionality Reduction

Goal: find the set of necessary feature dimensions for describing the data

Criterion: minimize the information loss

Dimensionality Reduction

Goal: find the set of necessary feature dimensions for describing the data

Criterion: minimize the information loss

What we can do: map the original features into a different space via some transformation

$$f : \mathbb{R}^D \mapsto \mathbb{R}^d \quad \text{where} \quad d \ll D$$

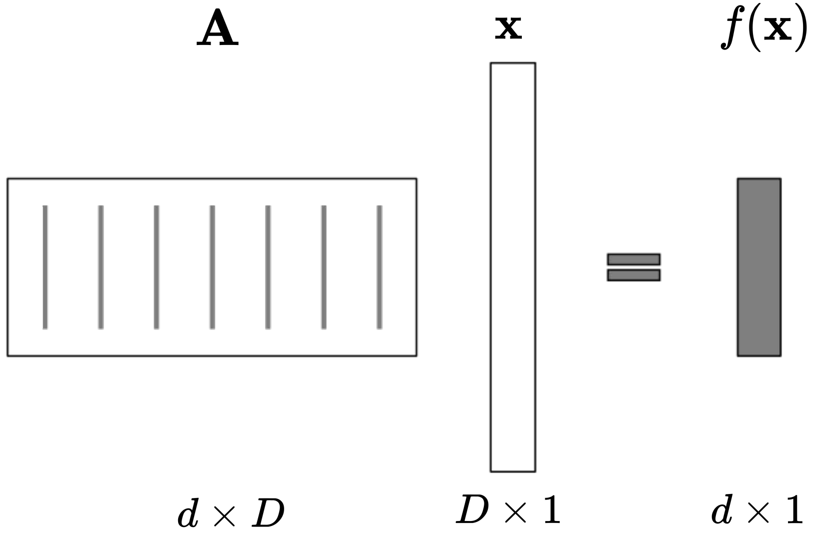
namely $f(\mathbf{x}) = \mathbf{A}\mathbf{x}$

Recall how this looks very similar to our functions used for regression and classification

Dimensionality Reduction

$$f(\mathbf{x}) = \mathbf{A}\mathbf{x}$$

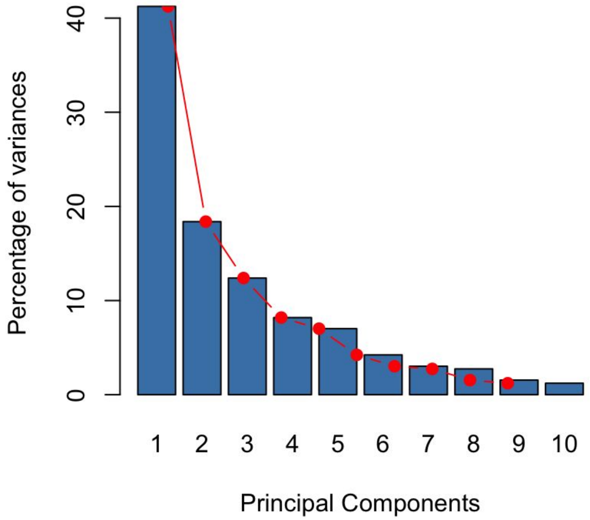
We will apply a linear transformation to the data (which projects the data to a new space to obtain a new set of features whose dimensions are lower than the original

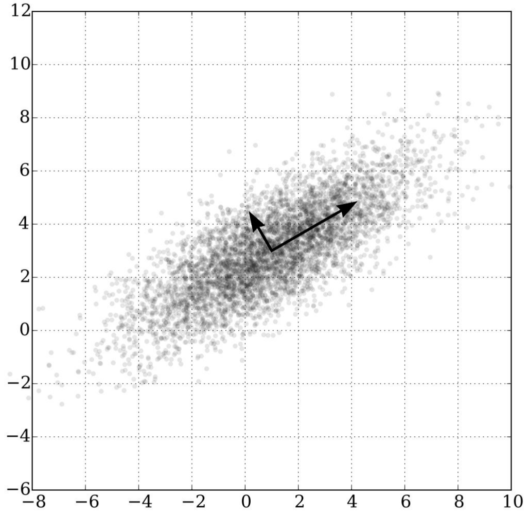


Principal Components

Principal Components (PCs) are orthogonal vectors
We order them based on the fraction of the total information (variation) in the corresponding directions

In other words, the first principal component is the direction with the most variance and the second principal component is the direction with the second most variance





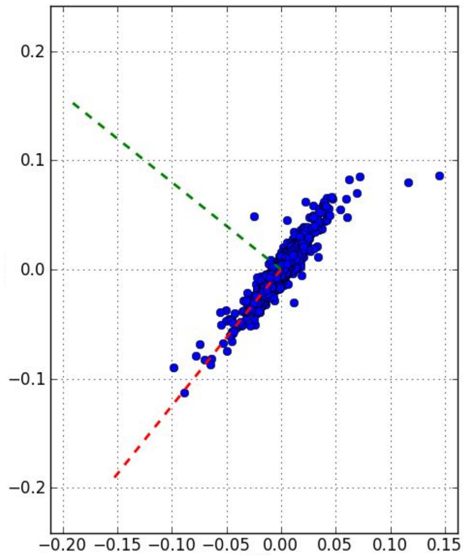
Principal Components Analysis

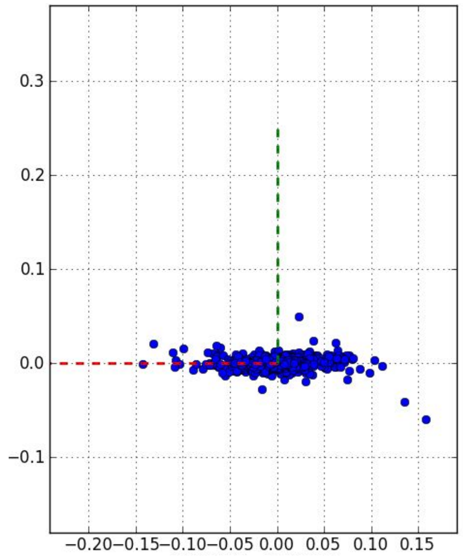
Principal Components Analysis (PCA) is a dimensionality reduction technique that reduces the dimensionality of the data while preserving the variation present in the dataset as much as possible

In this first case, we have 2 features
and PCA finds the two principal components

We select the top 2 components (all of them)
and project our data onto the new axes

Notice that it looks like our data has been
rotated and scaled



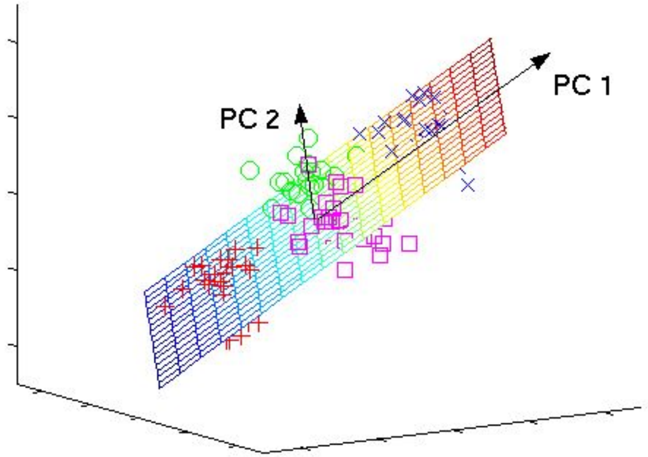


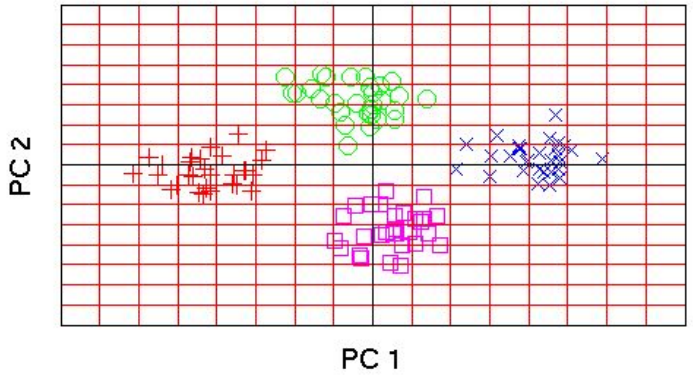
Also known as the the Karhunen–Loève (KL) transform

Principal Components Analysis

Principal Components Analysis (PCA) is a dimensionality reduction technique that reduces the dimensionality of the data while preserving the variation present in the dataset as much as possible

Let's project (change of coordinates) from 3 dimensional space down to 2 dimensional space
Notice that the data now lies on a 2-d plane, where the axes are the principal components

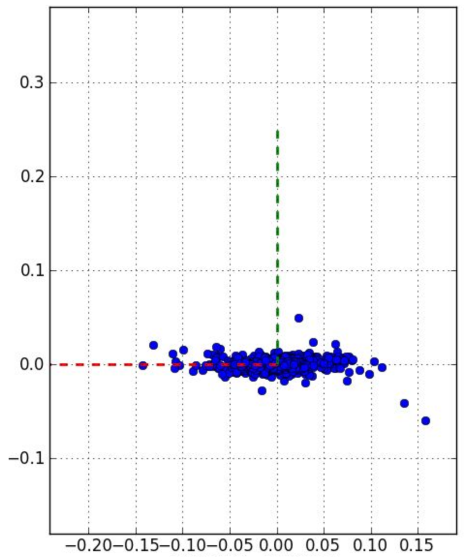




Principal Components Analysis

The principal components (directions or axes) we want to find are the eigenvectors of the data

Notice that any of the points in the figure can be found by multiplying each of the vectors (green and red) by a scalar



Specifically, given an eigenvector, if we apply any transformation to it, then the resulting will always be the eigenvector multiplied by a scalar

$$\mathbf{A}v = \lambda v$$

Here is an example that illustrates this:

$$\begin{bmatrix} 2 & 3 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 3 \\ 2 \end{bmatrix} = \begin{bmatrix} 12 \\ 8 \end{bmatrix} = 4 \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

The scalar is known as an eigenvalue

Principal Components Analysis

Principal Component Analysis is an unsupervised learning algorithm

Given a dataset $\mathbf{X} = (\mathbf{x}^0, \mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^n)$ where $\mathbf{x} \in \mathbb{R}^D$

1) Subtract the mean $\mathbf{B} = \mathbf{X} - \mu$ where $\mu = \frac{1}{N} \sum_{n}^{N} \mathbf{x}^n$
2) Compute the covariance matrix $\mathbf{C} = \frac{1}{N-1} \mathbf{B}^\top \mathbf{B}$
3) Compute the eigenvectors $\mathbf{V}^{-1}\mathbf{C}\mathbf{V} = \mathbf{\Sigma}$ where $\mathbf{V}$ is the eigenvectors and $\mathbf{\Sigma}$ the eigenvalues
4) Sort $\mathbf{V}$ based on $\mathbf{\Sigma}$ from largest to smallest, and choose the top k and form $\mathbf{W}$
5) Project to the new feature space (a subspace) $\mathbf{Z} = \mathbf{B}\mathbf{W}$ (latent vector)

Principal Components Analysis

$$\mathbf{Z} = \mathbf{B}\mathbf{W}$$

Now that we have projected our data into a new feature space (subspace), how do we get the data back?

Principal Components Analysis

$$\mathbf{Z} = \mathbf{B}\mathbf{W}$$

Now that we have projected our data into a new feature space (subspace), how do we get the data back?

$$\hat{\mathbf{X}} = \mathbf{Z}\mathbf{W}^\top + \mu$$

We can back project (undo the transformation) and add back the mean to recover the data

How do we know that our data is correctly recovered?

Principal Components Analysis

$$\mathbf{Z} = \mathbf{B}\mathbf{W}$$

Now that we have projected our data into a new feature space (subspace), how do we get the data back?

$$\hat{\mathbf{X}} = \mathbf{Z}\mathbf{W}^\top + \mu$$

We can back project (undo the transformation) and add back the mean to recover the data

How do we know that our data is correctly recovered?

Let’s measure the mean squared loss:

$$\frac{1}{N}\sum_{n}^{N} ||\mathbf{x}^n - \hat{\mathbf{x}}^n||_2^2$$

Principal Components Analysis

We can interpret PCA as:

Maximum Variance Subspace

PCA finds vectors such that projections on to the vectors capture maximum variance in the data

Minimum Reconstruction Error

PCA finds vectors such that projection on to the vectors yields minimum MSE reconstruction

As a least square solution:

PC1 is a minimum distance fit to a vector in the original feature space
PC2 is a minimum distance fit to a vector in the plane perpendicular to PC1

Dimensionality Reduction using PCA

In high-dimensional problems, data sometimes lies near a linear subspace

We want to maintain the variance in our data so we choose the principal components
with largest eigenvalue

When eigenvalues are small, this means the information along the component is basically noise

Since we select the top-k principal components, we might lose some information
but if eigenvectors whose eigenvalues are small enough, then we won't lose too much information

Applying PCA

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn import datasets as skdata

# We will use the iris dataset
iris_dataset = skdata.load_iris()
X = iris_dataset.data # (150, 4)
y = iris_dataset.target
```

Applying PCA

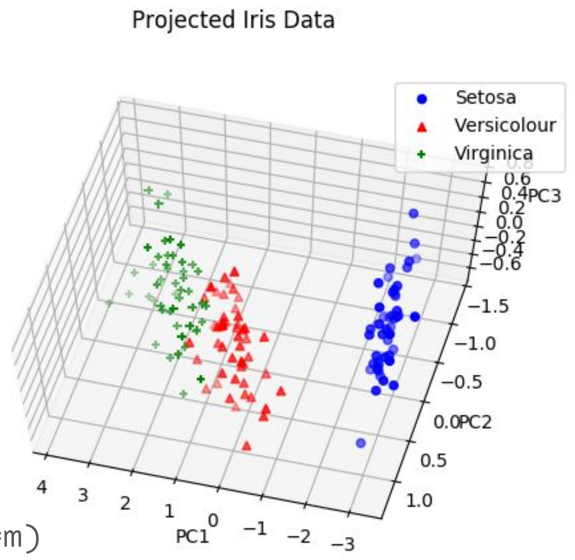
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn import datasets as skdata

# We will use the iris dataset
iris_dataset = skdata.load_iris()
X = iris_dataset.data # (150, 4)
y = iris_dataset.target

# Compute the mean
mu = np.mean(X, axis=0)

# Center the data
B = X-mu
```

Applying PCA

```
# Compute the mean
mu = np.mean(X, axis=0)

# Center the data
B = X-mu # (150, 4)

# Compute the covariance matrix
C = np.matmul(B.T, B)/(B.shape[0]) # (4, 150) x (150, 4) => (4, 4)

# Eigen decomposition
S, V = np.linalg.eig(C)

# Select the top 3 dimensions
order = np.argsort(S)[::-1]
W = V[:, order][:, 0:3] # Transformation for projecting X to subspace

# Project our data
Z = np.matmul(B, W) # (150, 3)
```

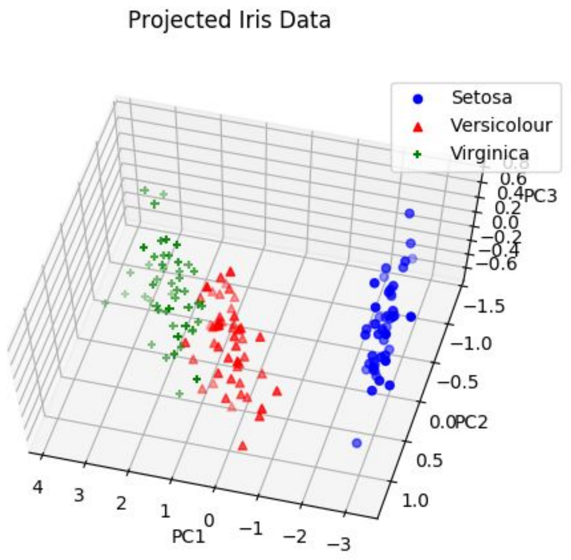
Applying PCA

```
# Let's visualize our new feature space
data_split = \
  (Z[np.where(y == 0)[0], :], Z[np.where(y == 1)[0], :], Z[np.where(y == 2)[0], :])
colors = ('blue', 'red', 'green')
labels = ('Setosa', 'Versicolour', 'Virginica')
markers = ('o', '^', '+')

fig = plt.figure()
fig.suptitle('Projected Iris Data')
ax = fig.add_subplot(1, 1, 1, projection='3d')
ax.set_xlabel('PC1')
ax.set_ylabel('PC2')
ax.set_zlabel('PC3')

for z, c, l, m in zip(data_split, colors, labels, markers):
  ax.scatter(z[:, 0], z[:, 1], z[:, 2], c=c, label=l, marker=m)
  ax.legend(loc='upper right')
```



Applying PCA

```
# Recover our data
X_hat = np.matmul(Z, W.T)+mu
mse = np.mean((X-X_hat)**2) # 0.005919048088406607

# Seems like we recovered our data pretty well
# Let's instead choose only two dimensions
W_2 = V[:, 0:2] # Transformation for projecting X to subspace

# Project our data
Z_2 = np.matmul(B, W_2)

X_hat_2 = = np.matmul(Z_2, W_2.T)+mu
mse = np.mean((X-X_hat_2)**2) # 0.02534107393239825

# As we reduce more dimensions, we lose more information
```



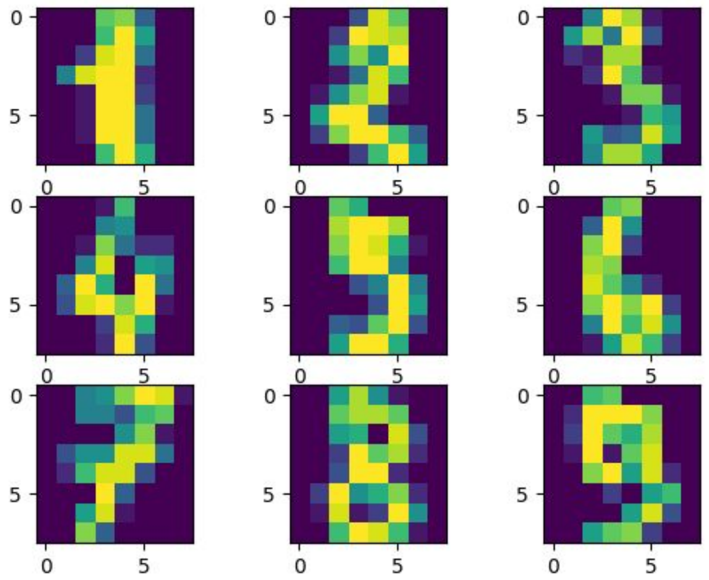
Applying PCA to Image Data

Images have very large dimensionality, a 64 by 64 image has a total of 4096 dimensions!

This is because there are 4096 pixels in the image

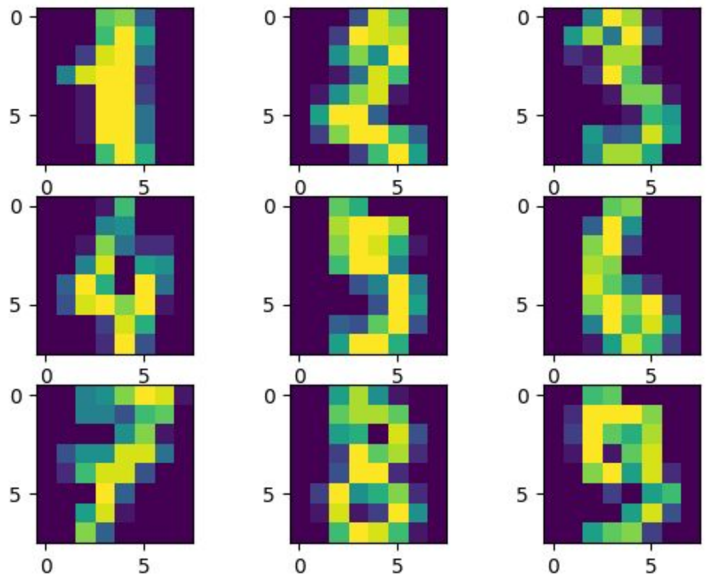
Do we really need all 4096 dimensions to represent the image?

Let’s consider the handwritten digits dataset



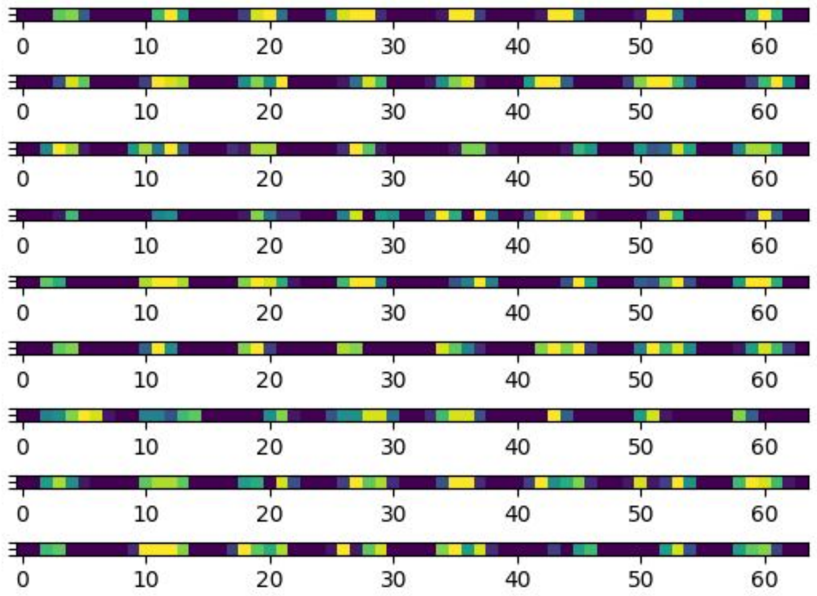
Applying PCA to Handwritten Digits

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn import datasets as skdata

# We will use the digits dataset
digits_dataset = skdata.load_digits()
X = digits_dataset.data # (1797, 64)
y = digits_dataset.target

# Visualize our data
X = np.reshape(X, (-1, 8, 8)) # (1797, 8, 8)
fig = plt.figure()
for i in range(1, 10):
  ax = fig.add_subplot(3, 3, i)
  ax.imshow(X[i, ...])

plt.show()
```



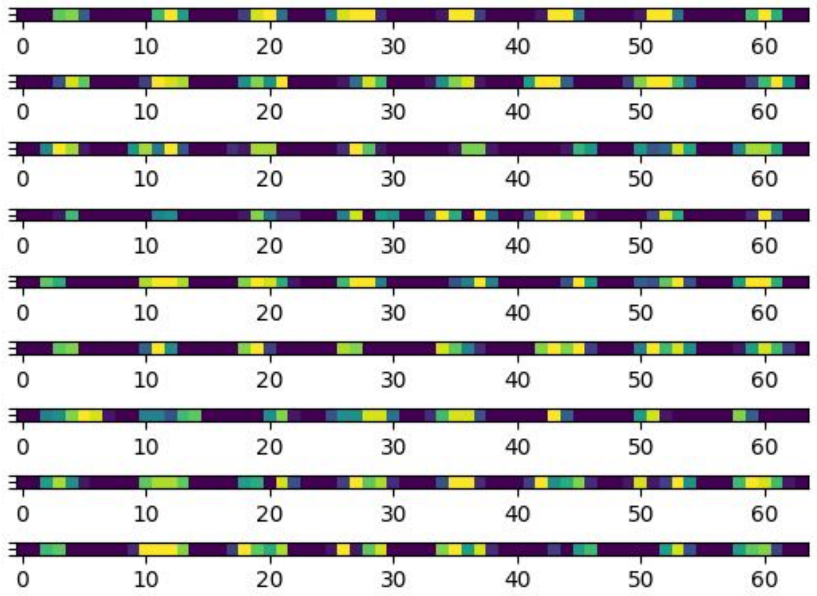
Applying PCA to Handwritten Digits

```
# Reshape the image into a vector
X = np.reshape(X, (-1, 64))
fig = plt.figure()
for i in range(1, 10):
  ax = fig.add_subplot(9, 1, i)
  ax.imshow(np.expand_dims(X[i, ...], axis=0))

plt.show()
```



Applying PCA to Handwritten Digits

```
# Reshape the image into a vector
X = np.reshape(X, (-1, 64))
fig = plt.figure()
for i in range(1, 10):
  ax = fig.add_subplot(9, 1, i)
  ax.imshow(np.expand_dims(X[i, ...], axis=0))

plt.show()

# Before we start applying PCA, let's split
# the dataset so we can see how good our
# projection (latent vector) is

# Split the data 90-10 training/testing
split_idx = int(0.90*X.shape[0])
X_train, y_train = X[:split_idx, :], X[:split_idx]
X_test, y_test = X[split_idx:, :], X[split_idx:]
```



Applying PCA to Handwritten Digits

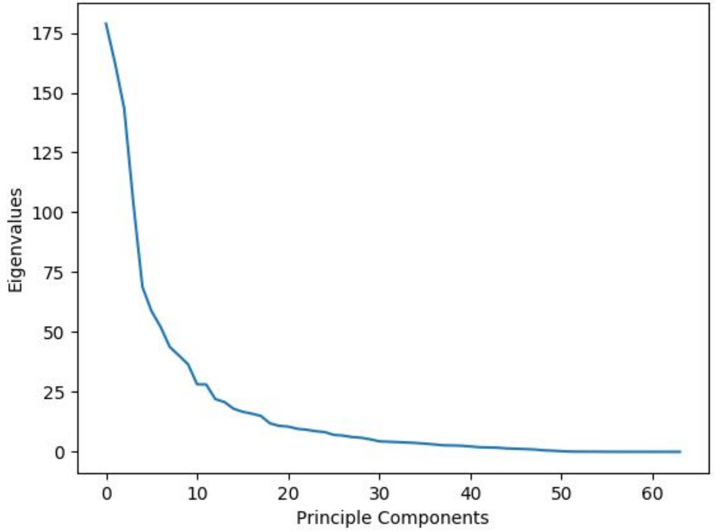
```
# Compute the mean and center the data
mu_train = np.mean(X_train, axis=0)
B_train = X_train-mu_train

# Compute the covariance matrix
C = np.matmul(B_train.T, B_train)/(B_train.shape[0]) # (64, 1617) x (1617, 64) => (64, 64)

# Eigen decomposition
S, V = np.linalg.eig(C)

# Select the top 3 dimensions
order = np.argsort(S)[::-1]
V = V[:, order]
W = V[:, 0:3] # Transformation for projecting X to subspace

# Project our data
Z_train = np.matmul(B_train, W) # (1617, 3)
```

Applying PCA to Handwritten Digits

```
# Recover our data
X_train_hat = np.matmul(Z_train, W.T)+mu_train
mse = np.mean((X_train-X_train_hat)**2) # 11.22242909000862

# Looks like we need more than just 3 dimensions!
# But how many do we need?
```

Applying PCA to Handwritten Digits

```
# Recover our data
X_train_hat = np.matmul(Z_train, W.T)+mu_train
mse = np.mean((X_train-X_train_hat)**2) # 11.22242909000862

# Looks like we need more than just 3 dimensions!
# But how many do we need?

# To figure this out,
# let's visualize our singular values
# from largest to smallest and choose from there

# Looks like we need around 45 eigenvectors
```



Applying PCA to Handwritten Digits

```
# Select the top 45 dimensions
W = V[:, 0:45] # Transformation for projecting X to subspace

# Project our data
Z_train = np.matmul(B_train, W) # (1617, 45)

X_train_hat = np.matmul(Z_train, W.T)+mu_train
mse = np.mean((X_train-X_train_hat)**2) # 0.07962481763287754

# Looks like we did much better this time
```

Applying PCA to Handwritten Digits

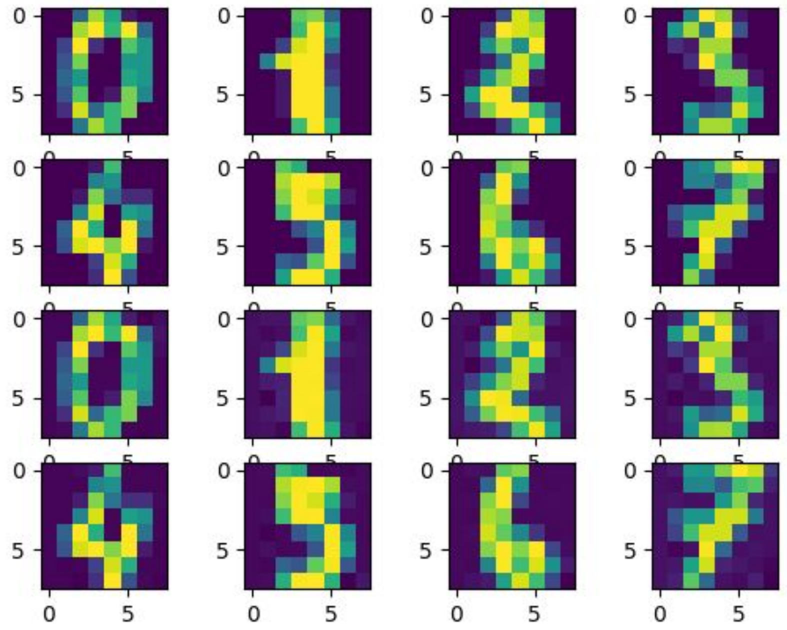
```
# What if we were to take 10 more eigenvectors?
W = V[:, 0:55] # Transformation for projecting X to subspace

# Project our data
Z_train = np.matmul(B_train, W) # (1617, 55)

X_train_hat = np.matmul(Z_train, W.T)+mu_train
mse = np.mean((X_train-X_train_hat)**2) # 0.00048743672799306617

# We improve by ~0.06
# Should we increase the number or dimensions?
```

Applying PCA to Handwritten Digits

```
# What if we were to take 10 more eigenvectors?
W = V[:, 0:55] # Transformation for projecting X to subspace

# Project our data
Z_train = np.matmul(B_train, W) # (1617, 55)

X_train_hat = np.matmul(Z_train, W.T)+mu_train
mse = np.mean((X_train-X_train_hat)**2) # 0.0004874367279930617

# We improve by ~0.06
# Should we increase the number or dimensions?


# Before we decide to take on more dimensions
# let’s look at the range of values of the images
print(np.min(X), np.max(X)) # The range of intensities in the image between [0, 16]
```

Applying PCA to Handwritten Digits

```
# Let's visualize what our reconstructed digits look like
W = V[:, 0:45]
# Project our data
Z_train = np.matmul(B_train, W) # (1617, 55)
X_train_hat = np.matmul(Z_train, W.T)+mu_train

# Visualize our data
X_train = np.reshape(X_train, (-1, 8, 8))
X_train_hat = np.reshape(X_train_hat, (-1, 8, 8))
fig = plt.figure()
for i in range(0, 16):
  ax = fig.add_subplot(4, 4, i+1)
  if i < 8:
    ax.imshow(X_train[i, ...])
  else:
    ax.imshow(X_train_hat[i-8, ...])

plt.show()
```



Generative Model

Recall that $\mathbf{Z} = \mathbf{BW}$ is the latent vector (variable)
Therefore our transformation is a set of weights multiplied by the variables

This means that depending on the variables, we can actually obtain a different $\hat{\mathbf{X}} = \mathbf{Z}\mathbf{W}^\top + \mu$

Let’s try to generate novel images by sampling from the distribution of $\mathbf{Z}$

We can do this by assuming that $\mathbf{Z}$ follows a Gaussian distribution $\mathbf{Z} \sim \mathcal{N}(0, \sigma^2)$
We know that $\mathbf{Z}$ has zero mean, what about its variance?

Generative Model

Recall that $\mathbf{Z} = \mathbf{BW}$ is the latent vector (variable)
Therefore our transformation is a set of weights multiplied by the variables

This means that depending on the variables, we can actually obtain a different $\hat{\mathbf{X}} = \mathbf{Z}\mathbf{W}^\top + \mu$

Let's try to generate novel images by sampling from the distribution of $\mathbf{Z}$

We can do this by assuming that $\mathbf{Z}$ follows a Gaussian distribution $\mathbf{Z} \sim \mathcal{N}(0, \sigma^2)$
We know that $\mathbf{Z}$ has zero mean, what about its variance?

Recall that the singular values $\mathbf{\Sigma}$ is our variance

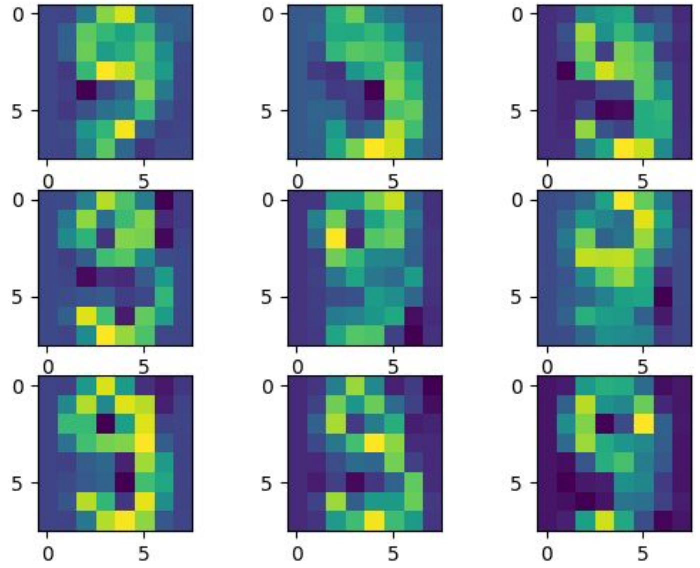
Let's assume that now have a latent vector sampled from $\mathbf{Z} \sim \mathcal{N}(0, \sigma^2)$
Our novel examples can be generated again by $\hat{\mathbf{X}} = \mathbf{Z}\mathbf{W}^\top + \mu$

Generating Novel Imagery

```
# Let’s try to create novel 9s
idx_9s = np.where(y == 9)[0]
X_9s = X[idx_9s, :]

mu_9s = np.mean(X_9s, axis=0)
B_9s = X_9s-mu_9s

# Covariance matrix
C_9s = np.matmul(B_9s.T, B_9s)/(B_9s.shape[0]) # (64, 64)

# Eigen decomposition
S_9s, V_9s = np.linalg.eig(C_9s)

order_9s = np.argsort(S_9s)[::-1]
W_9s = V_9s[:, order_9s]
```

Generating Novel Imagery

```
# Let's sample from Z to get 9 samples
Z_9s = np.random.normal(0, np.sqrt(S_9s), (9, S_9s.shape[0]))
# Our novel examples
X_9s_hat = np.matmul(Z_9s, W_9s.T)+mu_9s

# Visualize our data
X_9s_hat = np.reshape(X_9s_hat, (-1, 8, 8))
fig = plt.figure()
for i in range(0, 9):
  ax = fig.add_subplot(3, 3, i+1)
  ax.imshow(X_9s_hat[i, ...])

plt.show()
```

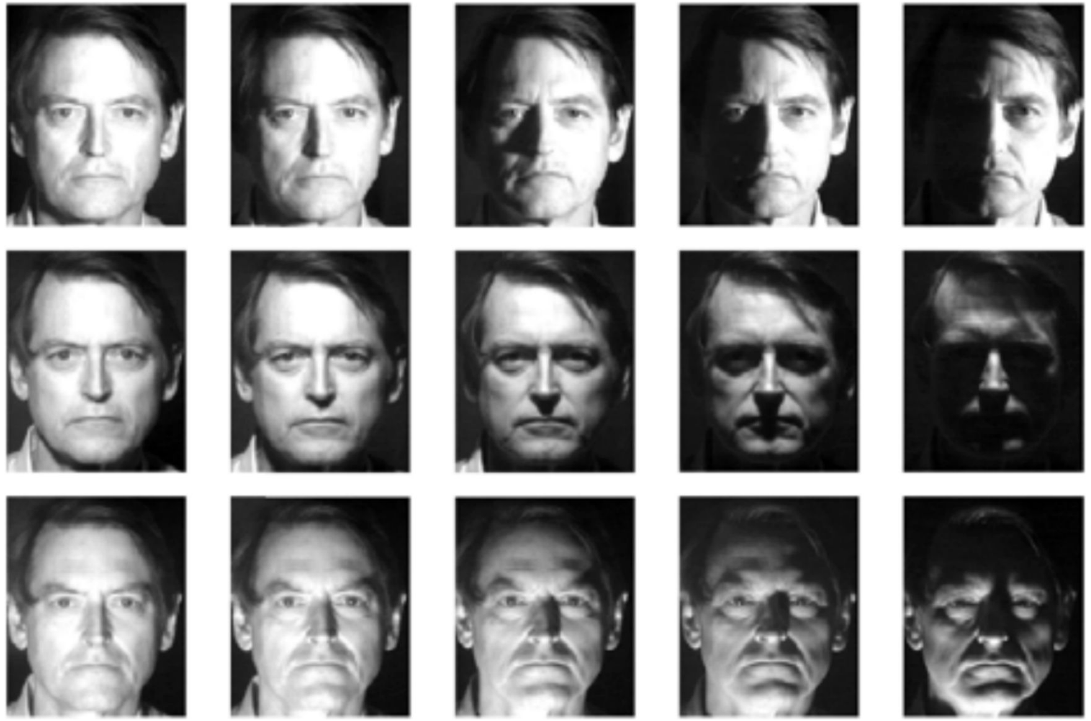


Eigenfaces

Recall the images to the right, these faces are very high dimensional

They are the photo of the same person under different lighting conditions

Is it possible to reduce the dimensions of faces?

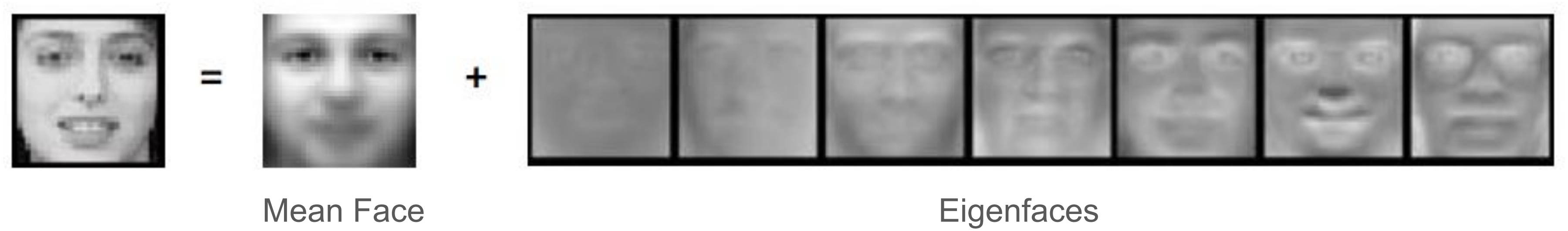


Eigenfaces

Eigenface: construct a low-dimensional linear subspace that contains most of the face images possible (possibly with small errors)

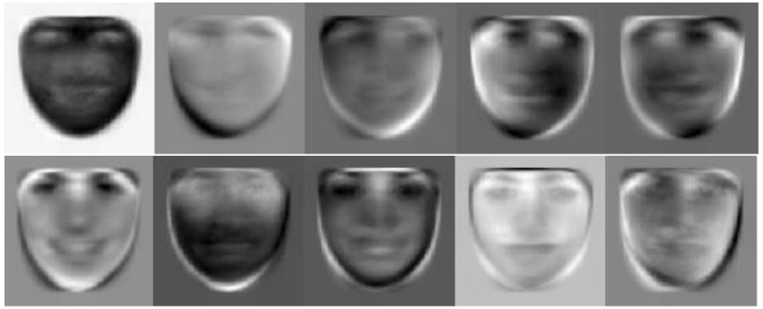
We can apply PCA to images of faces to reduce their dimensionality

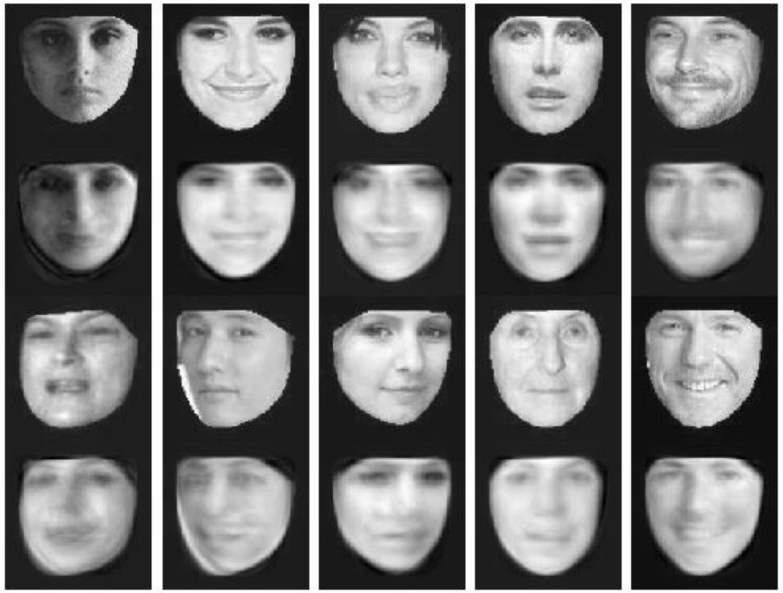
After we have selected the top k eigenvectors, we can reshape them back into the an image
The result of which are eigenfaces



Reconstructing Faces using Eigenfaces

Just like what we have done with the digits and iris datasets
we can reconstruct the faces by projecting and back projecting to the original image space





Each of the faces were reconstructed using eigenfaces

The first and third rows are the original images and the second and fourth rows are the reconstructions using 50 eigenfaces

Synthesizing Faces using Eigenfaces

We can also synthesize new faces by sampling from the distribution of the latent vector