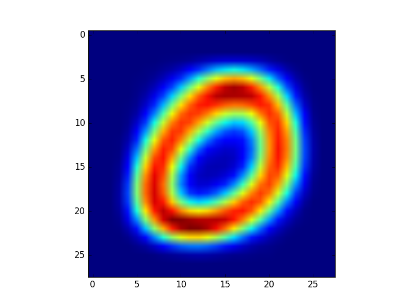
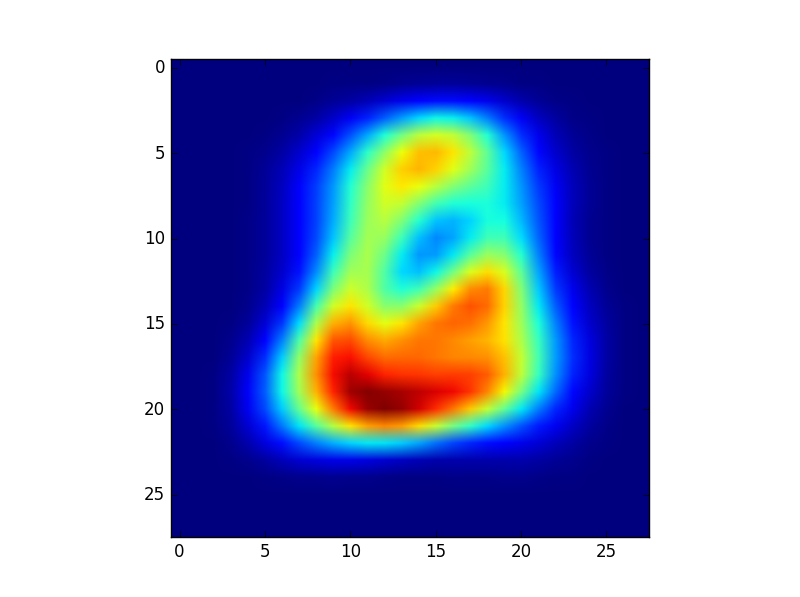
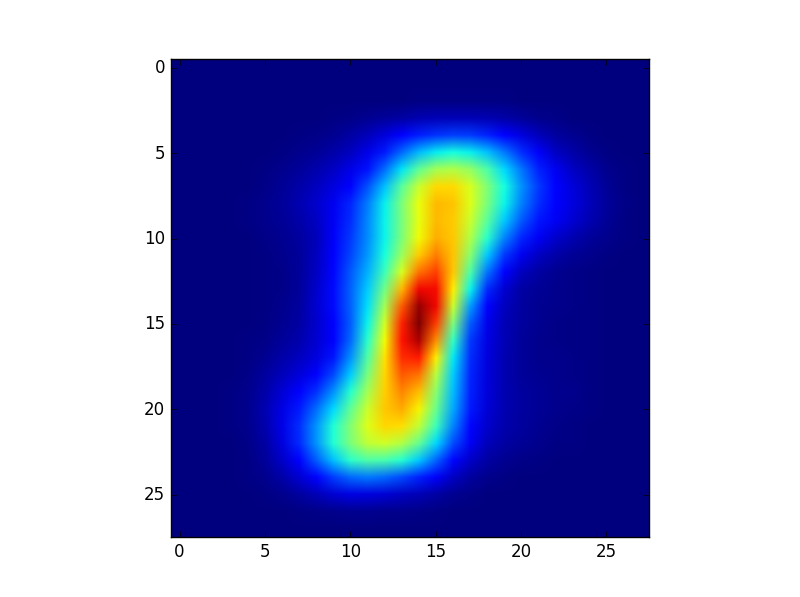
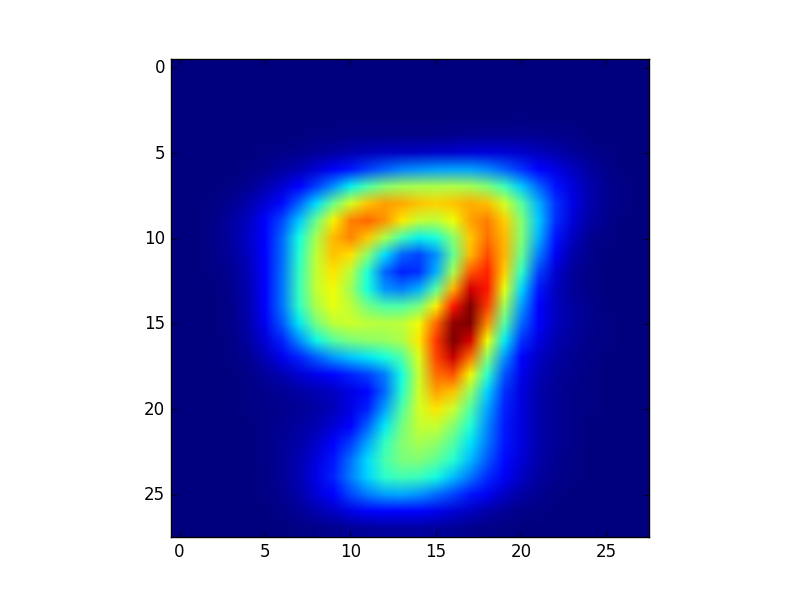
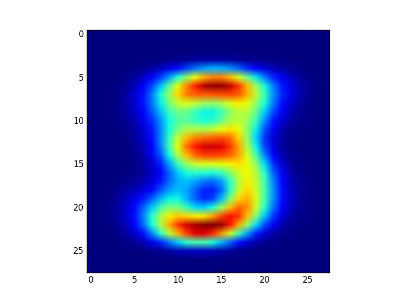
Jesse Li

23822462

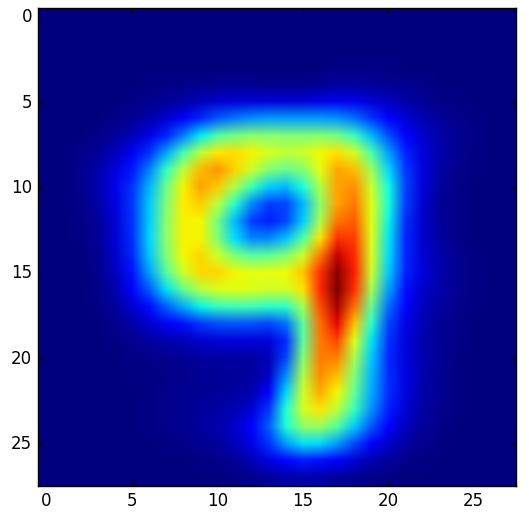
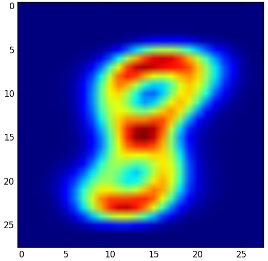
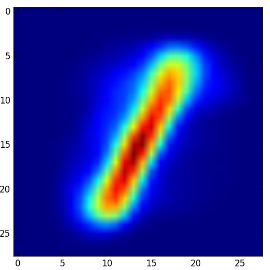
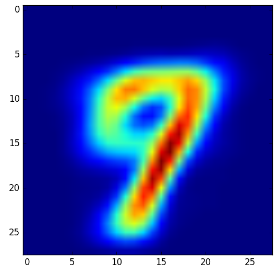
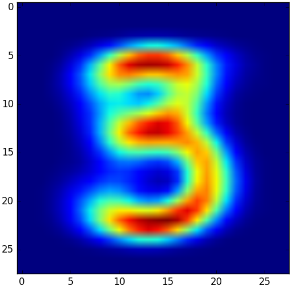
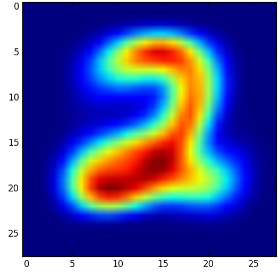
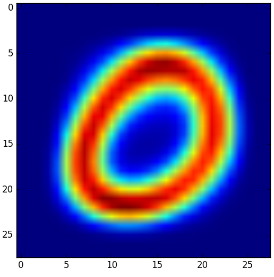
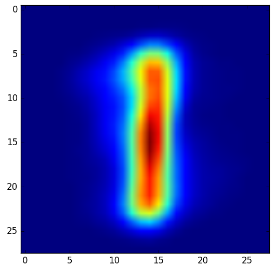
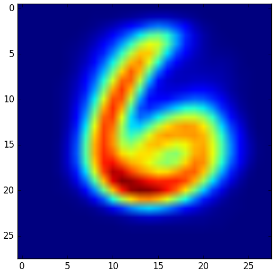
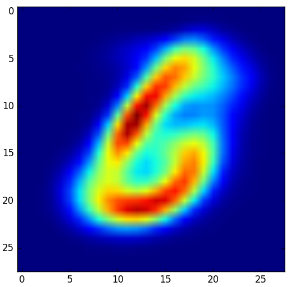
HW7 Writeup

K-means clustering

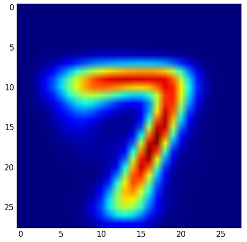
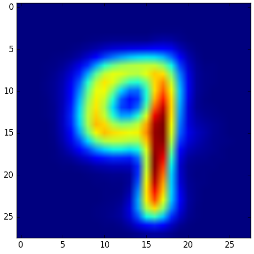
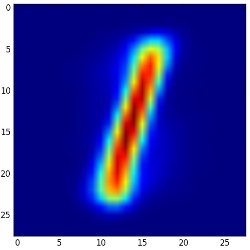
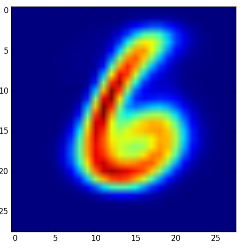
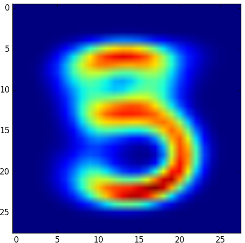
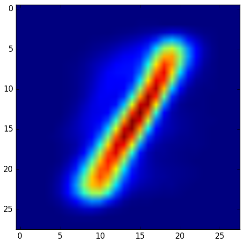
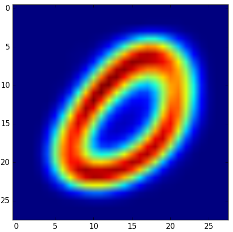
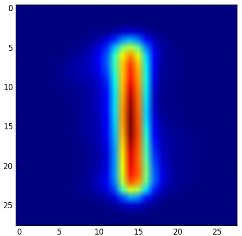
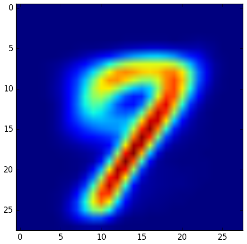
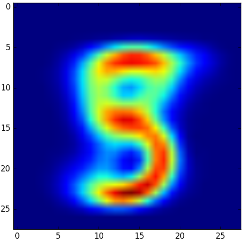
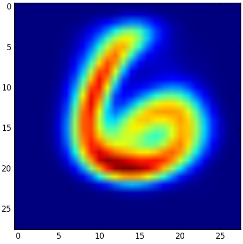
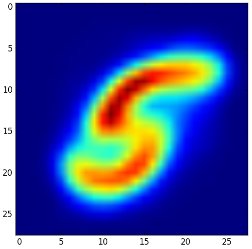
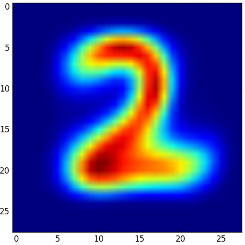
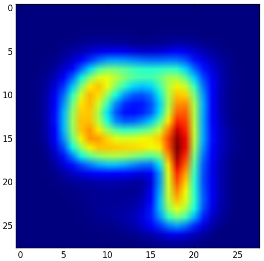
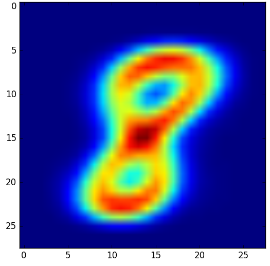
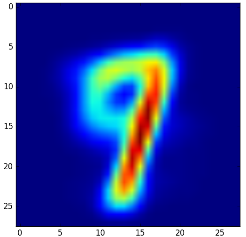
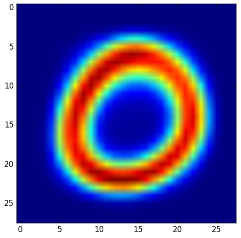
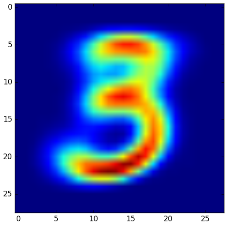
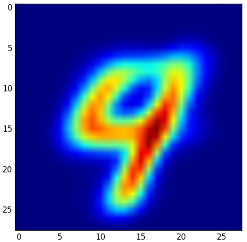
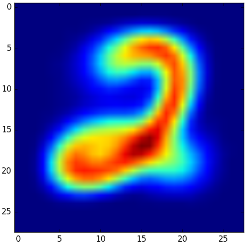
Visualization of clusters for k = 5:



Visualization of clusters for k = 10:



Visualization of clusters for k = 10:



The k-means loss varies between different runs with random initialization.

For k = 5:

Run 1: ended on iteration 50, loss = 176820225507.0

Run 2: ended on iteration 49, loss = 176820225507.0

Run 3: ended on iteration 50, loss = 176820236534.0

For k = 10:

Run 1: ended on iteration 210, loss = 162585035492.0

Run 2: ended on iteration 139, loss = 162585618069.0

Run 3: ended on iteration 70, loss = 162587380290.0

For k = 20:

Run 1: ended on iteration 112, loss = 148699882230.0

Run 2: ended on iteration 99, loss =149111968809.0

Run 3: ended on iteration 77, loss =149107176083.0

Code for K-means:

import numpy as np

import scipy.io

import matplotlib.pyplot as plt

import matplotlib

import math

train = scipy.io.loadmat('data/mnist\_data/images.mat')

train\_images= train["images"]

train\_images = np.float64(train\_images.reshape(-1, train\_images.shape[-1])).T

np.random.shuffle(train\_images)

def montage\_images(images):

num\_images=min(1000,np.size(images,2))

numrows=math.floor(math.sqrt(num\_images))

numcols=math.ceil(num\_images/numrows)

img=np.zeros((numrows\*28,numcols\*28));

for k in range(num\_images):

r = k % numrows

c = k // numrows

img[r\*28:(r+1)\*28,c\*28:(c+1)\*28]=images[:,:,k];

return img

def kmeans(k):

mu = [np.random.rand(train\_images.shape[1])]\*k

classes = np.array([[] for \_ in range(k)])

iterate = True

iteration = 0

prev\_loss = float('inf')

while iterate:

iteration += 1

loss = 0

for i, lst in enumerate(classes):

for j in lst:

loss += np.sum((j - mu[i])\*\*2)

if prev\_loss == loss:

iterate = False

prev\_loss = loss

new\_classes = [[] for \_ in range(k)]

for image in train\_images:

best\_i, best\_mu, best = 0, mu[0], float('inf')

for i in range(k):

cost = np.sum((mu[i] - image)\*\*2)

if cost < best:

best\_i, best\_mu, best = i, mu[i], cost

new\_classes[best\_i].append(image)

if not np.array\_equal(classes,np.array(new\_classes)):

classes = np.array(new\_classes)

for i in range(k):

if classes[i]:

mu[i] = np.sum(np.array(classes[i]),axis=0)/len(classes[i])

print(iteration)

print(loss)

return mu

k\_list = [5,10,20]

for i,k in enumerate(k\_list):

mu = kmeans(k)

for mean in mu:

img = montage\_images(mean.reshape(28,28,1))

plt.imshow(img)

plt.show()

Warm-up

Basic recommendation validation accuracy = 0.6203

k-NN accuracies:

k = 10 🡪 0.649051490515

k = 100 🡪 0.689430894309

k = 1000 🡪 0.694037940379

The accuracies for k-NN are higher than the basic recommendation system.

Latent Factor Model

SVD for d = 2:

MSE = 18441623.0179

Accuracy = 0.705149051491

SVD for d = 5:

MSE = 16333384.4202

Accuracy = 0.715447154472

SVD for d = 10:

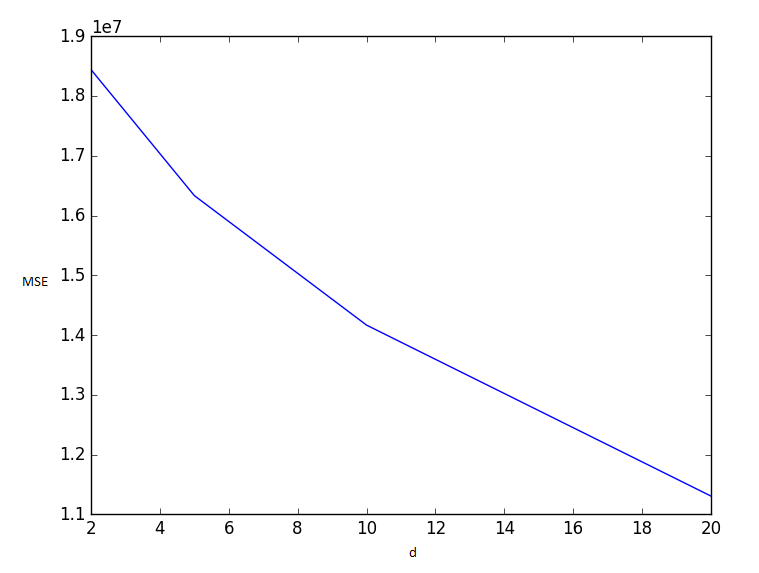
MSE = 14165432.758

Accuracy = 0.716531165312

SVD for d = 20:

MSE = 11304007.4397

Accuracy = 0.685907859079



MSE decreases as d increases.