CS 189:289A Introduction to Spring 2017 Machine Learning — Homework 7: Unsupervised Due Fri., April 28

Instructions

- We prefer that you typeset your answers using LaTeX or other word processing software. Neatly handwritten and scanned solutions will also be accepted.
- Please make sure to start **each question on a new page**, as grading (with Gradescope) is much easier that way!
- This assignment contains some optional questions. Feel free to work on them. We encourage you to try them out to expand your understanding. They will not be graded and will not count towards your grade.
- Deliverables. Submit a **PDF of your writeup** to the Homework 7 assignment on Gradescope. Include your code in your writeup in the appropriate sections. Submit your **code zip** and a README to the Homework 7 Code assignment on Gradescope. Finally, submit **your predictions** for the test sets to Kaggle. Be sure to include your Kaggle display name and score in your writeup.
- Due **Friday, April 28, 2017 at 11:59 PM.**

Collaborators

- I got some help on this homework from my fellow classmates Zisu Dong, Rong Huang, Tong Li, Yika Luo and Joseph Yan.

1 k-means clustering

Given a dataset $x_1, ..., x_n \in \mathbb{R}^d$ and an integer $1 \leq k \leq n$, recall the following k-means objective function

$$\min_{\pi_1,...,\pi_k} \sum_{i=1}^{k} \sum_{j \in \pi_i} \|x_j - \mu_i\|_2^2 , \quad \mu_i = \frac{1}{|\pi_i|} \sum_{j \in \pi_i} x_j . \tag{1}$$

Above, $\{\pi_i\}_{i=1}^k$ is a partition of $\{1, 2, ..., n\}$. The objective (1) is NP-hard[1] to find a global minimizer of. Nevertheless the commonly used heuristic which we discussed in lecture, known as Lloyd's algorithm, typically works well in practice.

(a) Implement Lloyd's algorithm for solving the k-means objective (1). Do not use any off the shelf implementations, such as those found in `scikit-learn`.

(i) Run your algorithm on MNIST with $k = 5, 10, 20$ cluster centers. Use the image data at `mnist_data/images.mat`, which contains 60,000 unlabeled images, and each image contains 28×28 pixels. Each pixel represents one coordinate in the k-means algorithm.

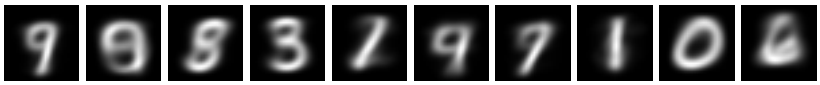
(ii) Visualize the centers you get, viewing each coordinate as a pixel (i.e., each center is represented by an image of 28×28 pixels). What are the differences between results with different numbers of cluster centers?

(iii) **(Optional)** Implement the `kmeans++` initialization scheme[2] for your k-means implementation. Note that this initialization scheme is widely used in practice.

Solution:

(ii) At k = 5: We can see that the clusters are centered at

At k = 10: We can see that the clusters are centered at



At k = 20: We can see that the cluster are centered at

[1] To be more precise, it is both NP-hard in d when $k = 2$ and k when $d = 2$. See the references on the wikipedia page for k-means for more details.

[2] See `http://ilpubs.stanford.edu:8090/778/1/2006-13.pdf`.

2 Low-Rank Approximation

Low-rank approximation tries to find an approximation to a given matrix, where the approximation matrix has a lower rank compared to the original matrix. This is useful for mathematical modeling and data compression. Mathematically, given a matrix D, we try to find $\hat{D}$ in the following optimization problem,

$$\underset{\hat{D}}{\operatorname{argmin}} \|D - \hat{D}\|_F \quad \text{subject to} \quad \operatorname{rank}(\hat{D}) \leq k \tag{2}$$

where $\|A\|_F = \sqrt{\sum_i \sum_j a_{ij}^2}$ represents the Frobenius norm, i.e., sum of squares of all entries in the matrix followed by a square root.

This problem can be solved using Singular Value Decomposition (SVD). Specifically, let $D = U\Sigma V^\top$, where $\Sigma = \operatorname{diag}(\sigma_1, ..., \sigma_n)$. Then a rank-$k$ approximation of D can be written as $\hat{D} = U\hat{\Sigma}V^\top$, where $\hat{\Sigma} = \operatorname{diag}(\sigma_1, ..., \sigma_k, 0, ..., 0)$. In this problem, we aim to perform this approximation method on gray-scale images, which can be thought of as a 2D matrix.

(a) Using the image `low-rank_data/face.jpg`, perform a rank-5, rank-20, and rank-100 approximation on the image. Show the obtained images in your report.

(b) Plot the Mean Squared Error (MSE) of using different rank approximations compared to the original image, ranging from rank 1 to rank 100.

(c) Now perform the same rank-5, rank-20, and rank-100 approximation on `low - rank_data/sky.jpg`. Show the obtained images in your report.

(d) Find the lowest rank approximation that you begin to have a hard time differentiating the original and the approximated images. Compare your results for the face and the sky image. What are the possible reasons for the difference?

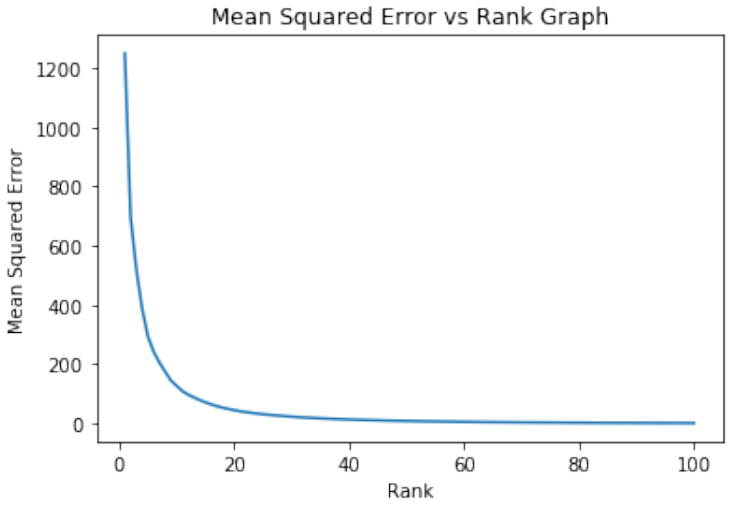
Solution:

(a) At k = 5:

At k = 20:

At k = 100:

(b) The Mean Squared Error (MSE) of using different rank approximations compared to the original image, ranging from rank 1 to rank 100 for face is



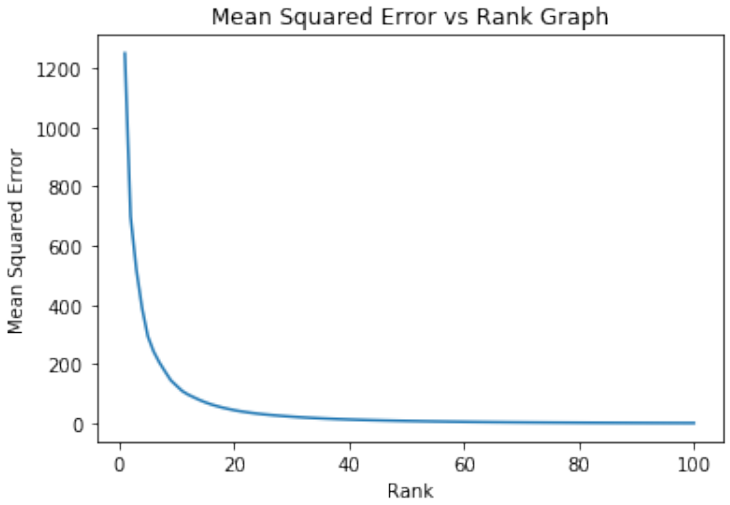
(c) At k = 5:

At k = 20:

At k = 100:

The Mean Squared Error (MSE) of using different rank approximations compared to the original

image, ranging from rank 1 to rank 100 for sky is



(d) I tried different ranks for sky and face.
For face, after rank is greater than **10**, all the graphs looks almost the same as the original graph to me.
For sky, after rank is greater than **50**, all the graphs looks almost the same as the original graph to me. We can verify this by looking at the MSE plot as well.
The difference might be caused by the sharpness (frequency) of the image. The SVD here keeps the largest k elements. Sky is relatively simple, so we can use a few eigenvalue to capture most of the image, while the face is more complex, and required more eigenvalues to capture.

3 Joke Recommender System

You will build a personalized joke recommender system. There are $m = 100$ jokes and $n = 24,983$ users. As historical data, every user read a subset of jokes and rated them. The goal is to recommend more jokes, such that the recommended jokes match the individual user's sense of humour.

3.1 Data Format

The historical rating is represented by a matrix $R \in \mathbb{R}^{n \times m}$. The entry R_{ij} represents the user i's rating on joke j. The rating is a real number in $[-10, 10]$: a higher value represents that the user is more satisfied with the joke. If the joke is not rated, then the corresponding entry value is `NaN`.

The directory `joke_data/jokes/` contains the text of all 100 jokes. Read them before you start! In addition, you are provided with three files at `joke_data/`:

- `joke_train.mat` is given as the training data. It contains the matrix R specified above.
- `validation.txt` contains user-joke pairs that doesn't appear in the training set. Each line takes the form "`i, j, s`", where `i` is the user index, `j` is the joke index, `s` indicates whether the user likes the joke. More specifically, `s` = 1 if and only if the user gives a positive rating to the joke.
- `query.txt` contains user-joke pairs that are neither in the training set nor in the validation set. Each line takes the form "`id, i, j`". You are asked to predict if user `i` likes joke `j`. The integer `id` is a unique id for the user-joke pair. Use it to submit the prediction to Kaggle (see Section 3.3).

3.2 Latent Factor Model

Latent factor model is the state-of-the-art method for personalized recommendation. It learns a vector representation $u_i \in \mathbb{R}^d$ for each user and a vector representation $v_j \in \mathbb{R}^d$ for each joke, such that the inner product $\langle u_i, v_j \rangle$ approximates the rating R_{ij}. You will build a simple latent factor model using two different methods in the following.

(a) The first method is using singular value decomposition (SVD) to learn a lower dimensional vector representation for users and jokes.

(i) Start by replacing all missing values in the data matrix by zero. Then apply SVD on the resulting matrix, and compute the corresponding representations for users and jokes. Recall this means to project the data vectors to lower dimensional subspaces of their corresponding spaces, spanned by singular vectors. Refer to the lecture materials on SVD, PCA and dimensionality reduction.

(ii) Evaluate the learnt vector representations by mean squared error:

$$\text{MSE} = \sum_{(i,j) \in S} (\langle u_i, v_j \rangle - R_{ij})^2 \quad \text{where } S := \{(i, j) : R_{ij} \neq \texttt{NaN}\}. \tag{3}$$

Try $d = 2, 5, 10, 20$. How does the MSE vary as a function of d? Use the inner product $\langle u_i, v_j \rangle$ to predict if user i likes joke j. Report prediction accuracies on the validation set.

(b) For sparse data, replacing all missing values by zero is not a completely satisfying solution. A missing value means that the user has not read the joke, but doesn't mean that the rating should be zero. A more reasonable choice is to minimize the MSE only on rated joke.

(i) Let's define a loss function:

$$L\Big(\{u_i\},\{v_j\}\Big) := \sum_{(i,j)\in S} (\langle u_i, v_j\rangle - R_{ij})^2 + \lambda \sum_{i=1}^{n} \|u_i\|_2^2 + \lambda \sum_{j=1}^{m} \|v_j\|_2^2,$$

where set S has the same definition as in equation (3) and $\lambda > 0$ is the regularization coefficient. Implement an algorithm to learn vector representations by minimizing the loss function $L(\{u_i\},\{v_j\})$.

Hint: you may want to employ an alternating minimization scheme. First, randomly initialize $\{u_i\}$ and $\{v_j\}$. Then minimize the loss function with respective to $\{u_i\}$ by treating $\{v_j\}$ as constant vectors, and minimize the loss function with respect to $\{v_j\}$ by treating $\{u_i\}$ as constant vectors. Iterate these two steps until both $\{u_i\}$ and $\{v_j\}$ converge. Note that when one of $\{u_i\}$ or $\{v_j\}$ is given, minimizing the loss function with respect to the other part has closed-form solutions.

(ii) Compare the resulting MSE and the prediction error with (a). Note that you need to tune the hyper-parameter λ to optimize the performance.

Solution:

(a) When we calculate the MSE using different d, we can get:

When d is 2, MSE is 20.3919949864, accuracy is 0.7051490514905149.

When d is 5, MSE is 18.0607906844, accuracy is 0.7154471544715447.

When d is 10, MSE is 15.6635581099, accuracy is 0.7165311653116531.

When d is 20, MSE is 12.4995106349, accuracy is 0.6859078590785908.

As we can see, as d increase, MSE decreases.

(b) (i)

$$L\Big(\{u_i\},\{v_j\}\Big) := \sum_{(i,j)\in S} (\langle u_i, v_j\rangle - R_{ij})^2 + \lambda \sum_{i=1}^{n} \|u_i\|_2^2 + \lambda \sum_{j=1}^{m} \|v_j\|_2^2$$

$$L\Big(\{u_i\},\{v_j\}\Big) := \sum_{(i,j)\in S} (u_i^T v_j - R_{ij})^2 + \lambda \sum_{i=1}^{n} (u_i^T u_i) + \lambda \sum_{j=1}^{m} (v_j^T v_j)$$

$$\nabla_{u_i}(L) = \sum_{(i,j)\in S} 2(u_i^T v_j - R_{ij})v_j + 2\lambda u_i$$

Note that here $u_i^T v_j$ is a scalar, and a scalar's transpose is itself. So $u_i^T v_j = (u_i^T v_j)^T = v_j^T u_i$. Also, for a scalar a, $a \cdot \vec{v} = \vec{v} \cdot a$. We can now get

$$\nabla_{u_i}(L) = \sum_{(i,j)\in S} 2v_j v_j^T u_i - 2v_j R_{ij} + 2\lambda u_i$$

$$\nabla_{u_i}(L) = \Big(\sum_{(i,j)\in S} 2v_j v_j^T u_i + 2\lambda u_i\Big) - \Big(\sum_{(i,j)\in S} 2v_j R_{ij}\Big)$$

$$\nabla_{u_i}(L) = (\sum_{j=1}^{m} 2v_j v_j^T u_i + 2\lambda u_i) - (\sum_{(i,j)\in S} 2v_j R_{ij}) = 0$$

$$(\sum_{j=1}^{m} 2v_j v_j^T + 2\lambda I)u_i - (\sum_{(i,j)\in S} 2R_{ij} v_j) = 0$$

$$u_i = (\sum_{j=1}^{m} v_j v_j^T + \lambda I)^{-1}(\sum_{(i,j)\in S} R_{ij} v_j)$$

$$\nabla_{v_j}(L) = \sum_{(i,j)\in S} 2(u_i^T v_j - R_{ij})u_i + 2\lambda v_j$$

$$\nabla_{u_i}(L) = \sum_{(i,j)\in S} 2u_i u_i^T v_j - 2u_i R_{ij} + 2\lambda v_j = 0$$

$$v_j = (\sum_{i=1}^{n} u_i u_i^T + \lambda I)^{-1}(\sum_{(i,j)\in S} R_{ij} u_i)$$

When the loss function is minimal

$$u_i = (\sum_{j=1}^{m} v_j v_j^T + \lambda I)^{-1}(\sum_{(i,j)\in S} R_{ij} v_j),\ v_j = (\sum_{i=1}^{n} u_i u_i^T + \lambda I)^{-1}(\sum_{(i,j)\in S} R_{ij} u_i)$$

(ii) Instead of replacing all missing values by 0, with our trained model, the new MSE and validation accuracy with $\lambda = 300$ and 50 iterations (in Kaggle submission, I used $\lambda = 350$) is

When d is 2, MSE is 17.2457818185, accuracy is 0.7067750677506776.

When d is 5, MSE is 14.9480340459, accuracy is 0.7195121951219512.

When d is 10, MSE is 13.1721849309, accuracy is 0.7341463414634146.

When d is 20, MSE is 5.87582841243, accuracy is 0.6720867208672087.

3.3 Recommending Jokes

(a) Using the methods you have implemented to predict the users' preference on unread jokes. For each line "`id, i, j`" in the query file, output a line "`id, s`" to a file named `kaggle_submission.txt`. Here, `s = 1` means that user `i` will give a positive rating to joke `j`, while `s = 0` means that the user will give a non-positive rating. The first line of `kaggle_submission.txt` should be the field titles: "`Id, Category`". Submit `kaggle_submission.txt` to Kaggle.
Solution:

My Kaggle Score is 0.73681

My Kaggle display name is Clark Fan

When deciding which λ to use, I tried the following lambda and their validation accuracy. At lambda is 0.001, our validation accuracy is 0.6655826558265583

At lambda is 0.01, our validation accuracy is 0.67289972899729

At lambda is 0.1, our validation accuracy is 0.6802168021680217

At lambda is 1, our validation accuracy is 0.6804878048780488

At lambda is 10, our validation accuracy is 0.6796747967479675

At lambda is 100, our validation accuracy is 0.7084010840108401

At lambda is 200, our validation accuracy is 0.7292682926829268

At lambda is 250, our validation accuracy is 0.729810298102981

At lambda is 300, our validation accuracy is 0.7314363143631436

At lambda is 350, our validation accuracy is 0.7346883468834688

At lambda is 400, our validation accuracy is 0.724390243902439

At lambda is 500, our validation accuracy is 0.7102981029810298

At lambda is 1000, our validation accuracy is 0.6685636856368564

Then, I tried several different d values.

When lambda is 350, at d = 5, our validation accuracy is 0.7203252032520325

When lambda is 350, at d = 10, our validation accuracy is 0.7311653116531165

When lambda is 350, at d = 15, our validation accuracy is 0.734959349593496

When lambda is 350, at d = 20, our validation accuracy is 0.7346883468834688

When lambda is 350, at d = 25, our validation accuracy is 0.7333333333333333

When lambda is 350, at d = 30, our validation accuracy is 0.7330623306233063

When lambda is 350, at d = 35, our validation accuracy is 0.7317073170731707.

Then, I train more iteration.

When lambda is 350, at d = 20, our validation accuracy with 20 iteration is 0.7365853658536585

When lambda is 350, at d = 15, our validation accuracy with 20 iteration is 0.7344173441734417

When lambda is 350, at d = 20, our validation accuracy with 30 iteration is 0.7357723577235772

When lambda is 350, at d = 15, our validation accuracy with 30 iteration is 0.7336043360433604

When lambda is 350, at d = 20, our validation accuracy with 40 iteration is 0.7357723577235772

When lambda is 350, at d = 15, our validation accuracy with 40 iteration is 0.7314363143631436

(b) (Optional) Build a joke recommender system for your friends: randomly display 5 jokes to receive their ratings, then recommend 5 best jokes and 5 worst jokes. Report your friends' ratings on the jokes that the system recommends.

In [ ]:
```
import math
import numpy as np
from scipy.io import loadmat
import csv
import random
import pandas as pd
from scipy.ndimage.interpolation import rotate
from sklearn.preprocessing import normalize, scale
import matplotlib
import matplotlib.cm as cm
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt
from collections import Counter
from sklearn.utils import shuffle
from joblib import Parallel, delayed
import multiprocessing
from scipy.misc import imsave
from skimage.io import imread
import pickle
```

In [ ]:
```
def write_CSV(prediction, fileName):
    header = "Id,Category\n"
    output = ""
    for i in range(len(prediction)):
        output += str(i+1) + "," + str(prediction[i]) + '\n'
    answers = header + output
    with open(fileName, "w") as f:
        f.write(answers)

def save_obj(obj, name):
    f = open(name, 'wb')
    pickle.dump(obj, f)
    f.close()

def load_obj(name):
    f = open(name, 'rb')
    obj = pickle.load(f)
    f.close()
    return obj
```

In [ ]:
```
## Q1 Data
mnist = loadmat("/Users/clark/Desktop/189Data/hw7_data/mnist_data/image
s.mat")
# print(mnist.keys())
mnist_data = mnist["images"]
# print(mnist_data.shape)
mnist_data = mnist_data.reshape(28*28, 60000).T
# print(mnist_data.shape)
np.random.shuffle(mnist_data)
```

k means

In [ ]:
```
def k_means(data, k = 10, max_iter = 500):
    length = len(data)
    partition = []
    part_length = []
    for i in range(k):
        partition.append([])
        part_length.append(0)
    #Forgy method: (step 2)
    indices = np.random.randint(low = 0, high=length, size=k)
    mu = []
    count = 0
    for i in indices:
        mu.append(data[i])
    while (True):
#         print(count)
        count += 1
        if (count >= max_iter):
            print("Reach max iterations")
            return mu

        temp_part = []
        #step 2:
        for i in range(length):
            point = data[i]
            centroid_index = np.argmin(np.linalg.norm(point - mu,
axis=1))
            partition[centroid_index].append(point)
        #step 1:
        for i in range(k):
            mu[i] = np.mean(partition[i], axis=0)
            temp_part.append(len(partition[i]))
        save_obj(mu, "mu" + str(k)+".pckl")
        if (sum(abs(np.array(temp_part) - np.array(part_length))) >= k):
            part_length = temp_part
        else:
            print("done")
            return mu
```

In [ ]:
```
def draw_centroid(centroids):
    length = len(centroids)
    for i in range(length):
        temp = np.array(centroids[i]).reshape(28, 28)
        imsave(str(i) + '.png', temp)

def visualization_kmeans(centroids):
    length = len(centroids)
    for i in range(length):
        temp = np.array(centroids[i]).reshape(28, 28)
        plt.imshow(temp)
        plt.show()
```

In [ ]:
```
centroids_k5 = k_means(mnist_data, k = 5, max_iter = 100)
centroids_k10 = k_means(mnist_data, k = 10, max_iter = 100)
centroids_k20 = k_means(mnist_data, k = 20, max_iter = 100)
```

In [ ]:
```
visualization_kmeans(centroids_k5)
draw_centroid(centroids_k5)
```

In [ ]:
```
visualization_kmeans(centroids_k10)
draw_centroid(centroids_k10)
```

In [ ]:
```
visualization_kmeans(centroids_k20)
draw_centroid(centroids_k20)
```

Low-Rank Approximation

In [ ]:
```
face = imread("/Users/clark/Desktop/189Data/hw7_data/low-rank_data/face.jpg")
sky = imread("/Users/clark/Desktop/189Data/hw7_data/low-rank_data/sky.jpg")
```

In [ ]:
```
def rank_approx(image, rank):
    u,s,v = np.linalg.svd(image)
    #Note: Here, it factors the matrix as u * np.diag(s) * v, so we do not use v.T
    S = np.zeros((u.shape[1], v.shape[0]))
    for i in range(rank):
        S[i][i] = s[i]
    return np.dot(np.dot(u, S), v)
def visualization_rank(image):
    image = image.reshape(face.shape)
    plt.imshow(image, cmap='gray')
    plt.show()
```

In [ ]:
```
def mse(orig, low_rank):
    result = np.mean((orig-low_rank) ** 2)
    return result
def plot_mse(image):
    errors = []
    for i in range(1, 101):
        low_rank = rank_approx(image, i)
        error = mse(image, low_rank)
        errors.append(error)
    plt.plot(range(1, 101), errors)
    plt.xlabel("Rank")
    plt.ylabel("Mean Squared Error")
    plt.title("Mean Squared Error vs Rank Graph")
    plt.show()
```

In [ ]:
```
rank5 = rank_approx(face, 5)
imsave("face_rank5" + '.png', rank5)
rank20 = rank_approx(face, 20)
imsave("face_rank20" + '.png', rank20)
rank100 = rank_approx(face, 100)
imsave("face_rank100" + '.png', rank100)
```

In [ ]:
```
visualization_rank(rank5)
visualization_rank(rank20)
visualization_rank(rank100)
```

In [ ]:
```
plot_mse(face)
```

In [ ]:
```
rank5 = rank_approx(sky, 5)
imsave("sky_rank5" + '.png', rank5)
rank20 = rank_approx(sky, 20)
imsave("sky_rank20" + '.png', rank20)
rank100 = rank_approx(sky, 100)
imsave("sky_rank100" + '.png', rank100)
```

In [ ]:
```
visualization_rank(rank5)
visualization_rank(rank20)
visualization_rank(rank100)
```

In [ ]:
```
plot_mse(sky)
```

In [ ]:
```
for i in range(50):
    temp = rank_approx(face, i)
    print(i)
    visualization_rank(temp)
```

In [ ]:
```
for i in range(50):
    temp = rank_approx(sky, i)
    print(i)
    visualization_rank(temp)
```

Joke Recommender System

In [ ]:
```
# m: jokes
num_jokes = 100
# n: users
num_users = 24983
```

Data Preprocessing

In [ ]:
```
def read_txt(file):
    lst = []
    f = open(file, "r")
    lines = f.readlines()
    for line in lines:
        line = line.split(",")
        lst.append([int(line[0]), int(line[1]), int(line[2])])
    return np.array(lst)
```

In [ ]:
```
train = loadmat("/Users/clark/Desktop/189Data/hw7_data/joke_data/joke_train.mat")["train"]
#Replace all missing values by zero
train_filled_with_zero = np.nan_to_num(train)
u,s,v = np.linalg.svd(train_filled_with_zero)
validation = read_txt("/Users/clark/Desktop/189Data/hw7_data/joke_data/validation.txt")
test = read_txt("/Users/clark/Desktop/189Data/hw7_data/joke_data/query.txt")
```

In [ ]:
```
def low_dim_approx(u,s,v, rank):
    #Computation of U,S,V is too slow. Thus, we will only compute it onc
e.
    #Note: Here, it factors the matrix as u * np.diag(s) * v, so we do n
ot use v.T
    S = np.zeros((u.shape[1], v.shape[0]))
    for i in range(rank):
        S[i][i] = s[i]
    return np.dot(np.dot(u, S), v)

def usv_process(u,s,v,d):
    new_u = np.array([row[:d] for row in u])
    new_v = np.array([row[:d] for row in v.T])
    new_s = np.identity(d)
    for i in range(d):
        new_s[i][i] = s[i]
    return new_u, new_s, new_v

def mse(orig, pred):
    uv = []
    r = []
    for i in range(orig.shape[0]):
        for j in range(orig.shape[1]):
            if not np.isnan(orig[i][j]):
                r.append(orig[i][j])
                uv.append(pred[i][j])
    uv = np.array(uv)
    r = np.array(r)
    #Don't know why, but the equation on pdf seems like it doesn't take
 the mean.
    #Instead, it takes the sum. I will use the mean here since by defini
tion MSE should be the mean
    result = np.mean((uv-r) ** 2)
    return result

#Recall that <u_i, v_j> approximates R_{ij}
#s = 1 iff rating > 0
def validate(validation, prediction):
    correct = 0.0
    #User index and joke index start at 1
    user_index = [val[0] for val in validation]
    joke_index = [val[1] for val in validation]
    user_like = [val[2] for val in validation]
    length = len(validation)
    for index in range(length):
        i = user_index[index] - 1
        j = joke_index[index] - 1
        s = user_like[index]
        #User like joke:
        if (s == 1 and prediction[i][j] > 0):
            correct += 1
        elif (s == 0 and prediction[i][j] <= 0):
            correct += 1
    return correct / length
```

In [ ]:
```
def train_and_pred(u, v, data, d = 20, max_iter=1, lamb=350):
    u_shape = u.shape
    v_shape = v.shape
    new_u = u
    new_v = v
    for count in range(max_iter):
        for i in range(u_shape[0]):
            left = lamb * np.identity(d)
            right = np.zeros((d, 1))
            for j in range(v_shape[0]):
                if not np.isnan(data[i][j]):
                    vj = new_v[j].reshape(d, 1)
                    left += np.dot(vj, vj.T)
                    right += data[i][j] * vj
            #Ax = b -> x = A^{-1} b -> numpy.linalg.solve
#             print(np.linalg.inv(left).shape)
#             print(right.shape)
#             print(np.dot(np.linalg.inv(left), right).shape)
#             print(new_u[i].shape)
            new_u[i] = np.dot(np.linalg.inv(left), right).reshape(d,)
        for j in range(v_shape[0]):
            left = lamb * np.identity(d)
            right = np.zeros((d, 1))
            for i in range(u_shape[0]):
                if not np.isnan(data[i][j]):
                    ui = new_u[i].reshape(d, 1)
                    left += np.dot(ui, ui.T)
                    right += data[i][j] * ui
            new_v[j] = np.dot(np.linalg.inv(left), right).reshape(d,)
        print(count)
#         if count % 10 == 0:
#             save_obj(new_u, str(d) + "_u_iter_" + str(count) + ".p")
#             save_obj(new_v, str(d) + "_v_iter_" + str(count) + ".p")
    return np.dot(new_u, new_v.T)
```

In [ ]:
```
def predict(approx, test):
    #Each line takes the form “id, i, j”. You are asked to predict if us
er i likes joke j
    user_id = [t[0] for t in test]
    user_index = [t[1] for t in test]
    joke_index = [t[2] for t in test]
    length = len(user_id)
    s = np.zeros(length, dtype=int)
    for index in range(length):
        i = user_index[index] - 1
        j = joke_index[index] - 1
        #User like joke:
        if (approx[i][j] > 0):
            s[index] = 1
        else:
            s[index] = 0
    return s
```

In [ ]:
```
for d in [2, 5, 10, 20]:
    pred = low_dim_approx(u,s,v, d)
    mean_square_error = mse(train, pred)
    accuaracy = validate(validation, pred)
    print("When d is " + str(d) + ", MSE is " + str(mean_square_error) +
 ", accuracy is " + str(accuaracy) + ".")
```

Train lambda

In [ ]:
```
for l in np.arange(-3,4):
    lam = 10 ** l
    ddd= 20
    new_u, new_s, new_v = usv_process(u,s,v,ddd)
    estimate2 = train_and_pred(new_u, new_v, train, ddd, max_iter=5, lam
b=lam)
    save_obj(estimate2, "lambda_" + str(lam) + "_5times_"+str(ddd)+".p")
    print("At lambda is " + str(lam) + ", our validation accuracy is " +
 str(validate(validation, np.array(estimate2))))
```

In [ ]:
```
for iter_time in np.arange(10, 50, 10):
    lam = 350
    ddd= 20
    new_u, new_s, new_v = usv_process(u,s,v,ddd)
    estimate2 = train_and_pred(new_u, new_v, train, ddd, max_iter=iter_t
ime, lamb=lam)
    save_obj(estimate2, "lambda_" + str(350) + "_" + str(iter_time) + "t
imes_"+str(20)+".p")
    print("When lambda is 350, at d = " + str(ddd) + ", our validation a
ccuracy with " + str(iter_time) + " iteration is " + str(validate(valida
tion, np.array(estimate2))))

    dddddd= 15
    new_u, new_s, new_v = usv_process(u,s,v,dddddd)
    estimate3 = train_and_pred(new_u, new_v, train, dddddd, max_iter=ite
r_time, lamb=lam)
    save_obj(estimate3, "lambda_" + str(350) + "_" + str(iter_time) + "t
imes_"+str(15)+".p")
    print("When lambda is 350, at d = " + str(dddddd) + ", our validatio
n accuracy with " + str(iter_time) + " iteration is " +  str(validate(va
lidation, np.array(estimate3))))
```

In [ ]:
```
pred2 = load_obj("./lambda_300_50times_2.p")
mean_square_error2 = mse(train, pred2)
accuaracy2 = validate(validation, pred2)
pred5 = load_obj("./lambda_300_50times_5.p")
mean_square_error5 = mse(train, pred5)
accuaracy5 = validate(validation, pred5)
pred10 = load_obj("./lambda_300_50times_10.p")
mean_square_error10 = mse(train, pred10)
accuaracy10 = validate(validation, pred10)
pred20 = load_obj("./lambda_300_50times_20.p")
mean_square_error20 = mse(train, pred20)
accuaracy20 = validate(validation, pred20)
print("When d is " + str(2) + ", MSE is " + str(mean_square_error2) + ",
 accuracy is " + str(accuaracy2) + ".")
print("When d is " + str(5) + ", MSE is " + str(mean_square_error5) + ",
 accuracy is " + str(accuaracy5) + ".")
print("When d is " + str(10) + ", MSE is " + str(mean_square_error10) +
", accuracy is " + str(accuaracy10) + ".")
print("When d is " + str(20) + ", MSE is " + str(mean_square_error20) +
", accuracy is " + str(accuaracy20) + ".")
```

Train Classifier with best lambda so far ¶

In [ ]:
```
d = 20
new_u, new_s, new_v = usv_process(u,s,v,d)
estimate = train_and_pred(new_u, new_v, train, d, max_iter=50, lamb=500)
```

In [ ]:
```
for ddd in [2, 5, 10, 20]:
    new_u, new_s, new_v = usv_process(u,s,v,ddd)
    estimate2 = train_and_pred(new_u, new_v, train, ddd, max_iter=50, la
mb=300)
    save_obj(estimate2, "lambda_350_50times_"+str(ddd)+".p")
```

Test

In [ ]:
```
model = np.array(train_and_pred(new_u, new_v, train, d, max_iter=50, lam
b=350))
print(validate(validation, np.array(model)))
```

In [ ]:
```
naive = predict(model, test)
```

In [ ]:
```
write_CSV(naive, "jokes.txt")
```