Variational Inference - Assignment 2
Printout

Ravi Regulagedda

README

To run the code, just run the code.py file. You will be asked for a choice of dataset 1. 500-500-30 and 2. 100-k. After making this choice, part 1 and part 2 is printed to stdout and the plots are created as png files in the directory.

Code

```
# homework.py
from scipy.sparse import lil_matrix
from matrix import *
import scipy.sparse
import time
import matplotlib.pyplot as plt

print("Please enter choice of dataset: \n")
print("1. 500-500-3\n")
print("2. 100-k\n")
choice = int(input("Please enter your choice: 1 (or) 2 [To Debug, type 0] - "))

name = "40-40-2"
data = "hw2data/data-40-20-2.txt"
rank = [1]
rank1 = 1

if choice == 1:
    name = "500-500-3"
    data = "hw2data/data-500-500-3.txt"
    rank = [1, 2, 3, 5, 10, 20]
    rank1 = 3

if choice == 2:
    data = "hw2data/ml-100k.txt"
    rank = [1, 2, 3, 5, 20, 100]
    rank1 = 5

# reading the dataset and storing in a sparse matrix

with open(data, encoding="utf-8") as d:
    test = []
    instance = d.readline()
    # splitting the instance at space
    users, items, scores = map(int, instance.split())
    # storing users and items into a list of lists (sparse) matrix
    m = scipy.sparse.lil_matrix((users, items))
    # doing a half split
    split = int(scores / 2)
    for _ in range(split):
        inst = d.readline().split()
        x = int(inst[0])
        y = int(inst[1])
        # m will be our training data
        m[x, y] = float(inst[2])
    for _ in range(scores - split):
        inst = d.readline().split()
        x = int(inst[0])
```

```python
        y = int(inst[1])
        test.append((x, y, float(inst[2])))

# making sure m is a sparse matrix
m = m.tocsr()

# baseline
a, b = [], []
c = scipy.sparse.find(m)[2].mean()
for i in range(0, m.shape[0]):
    if len(scipy.sparse.find(m[i])[0]) == 0:
        a.append(c)
    else:
        a.append(m[i].mean())
for j in range(0, m.shape[1]):
    if len(scipy.sparse.find(m.T[j])[0]) == 0:
        b.append(c)
    else:
        b.append(m.T[j].mean())


# defning an RMSE function (since we have to use it many times)
def rmse(true, pred):
    # convert them to np.array just in case
    true = np.array(true)
    pred = np.array(pred)
    difference_2 = (true - pred) ** 2
    error = np.sqrt(difference_2.mean())
    return error


# calculating the baseline score and the baseline rmse
true_scores, base_preds = [], []
for instance in test:
    # unpacking
    user, item, score = instance
    true_scores.append(score)
    base_preds.append((a[user] + b[item]) / 2)
    base_rmse = rmse(true_scores, base_preds)

print("Baseline rmse is: ", base_rmse)

# part 1 (given dataset/rank has already been chosen)
part1_rmse, times = [], []
start = time.time()
time_taken = 0
# since we used a generator, we can step through each u, v and see how the rmse
# goes down per iteration (till convergence or end of 100 iterations)
for u, v in matrix_factorization(m, rank1):
    part1_preds = []
    end = time.time()
    time_taken += end - start
    times.append(time_taken)
    for instance in test:
        user, item, score = instance
        pred = u[user].dot(v[item])  # making the predictions
        part1_preds.append(pred)
    part1_rmse.append(rmse(true_scores, part1_preds))  # rmse per iteration
    start = time.time()  # resetting time for next iteration

print("For Part 1: \n")
print("Total Iterations: ", len(part1_preds), "\n")
if len(part1_preds) < 100:
    print("We had early stopping due to difference between iterations < 0.01\n")
print("RMSE: ", part1_rmse)
print("Time taken: ", times)
print("\n")

# part 2
# part 2 (given dataset has already been chosen)
part2_rmse = []
for r in rank:
    # we do this to get the last item from our generator
```

```python
    output = list(matrix_factorization(m, r))
    final = output[-1]
    u, v = final
    part2_preds = []
    for instance in test:
        user, item, score = instance
        pred = u[user].dot(v[item])
        part2_preds.append(pred)
    part2_rmse.append(rmse(true_scores, part2_preds))

print("For Part 2: ")
print("For the given ranks: ", rank)
print("The RMSE changes as follows: ", part2_rmse)

plt.plot(times, part1_rmse)
plt.title(name + " Part 1")
plt.xlabel("Time taken ")
plt.ylabel("RMSE")
plt.savefig("Part 1 " + name + ".png")

plt.clf()

plt.plot(rank, part2_rmse)
plt.xticks(rank)
plt.title(name + " Part 2")
plt.xlabel("Rank")
plt.ylabel("Chnage in RMSE")
plt.savefig("Part 2 " + name + ".png")

print("Please check your current directory for generated plots")
```

```python
# matrix.py
import numpy as np
from scipy.sparse import csr_matrix
import scipy.sparse


# defining the matrix decomposition
# this is a generator i.e, it 'yields' a value after 100 iterations
# that makes it easy for us to store and plot the performance metrics
# i used black to format my code. if it looks strange blame that :)
def matrix_factorization(m, rank):
    """Performs matrix factorization according to Lim & Teh, 2007. Takes in m: a sparse matrix
     and rank"""
    # initalize the variables
    # shape of the matrix
    # this assumes we have a 2D matrix (which we do)

    i_m = m.shape[0]
    j_m = m.shape[1]
    d = rank

    # variables and constants
    sigma_2 = np.ones(d)
    rho_2 = np.ones(d) / d
    tau_2 = 1
    u = []
    v = []
    # notation from paper
    S, phi, psi, t = [], [], [], []

    # intializing our variables with ones and random values according to the question
    for i in range(i_m):
        phi.append(np.eye(d))
        u.append(np.random.normal(0, 1, d))
    for j in range(j_m):
        psi.append(np.eye(d))
        S.append(np.diag(1 / rho_2))
        t.append(np.zeros(d))
        v.append(np.random.normal(0, 1, d))

    # make sure everything is an numpy array so there are no runtime/indexing errors
    phi = np.array(phi)
```

```
psi = np.array(psi)
S = np.array(S)
t = np.array(t)
u = np.array(u)
v = np.array(v)

norm_u = 0
norm_v = 0

N = []
for i in range(i_m):
    # from the paper N[i] is 1 if we observe m_ij. Since we observe a 0 or 1, we just
append it
    N.append(scipy.sparse.find(m[i])[1])
# all the places in the sparse matrix where there are elements
ob = scipy.sparse.find(m)

# we perform our EM now
# iterating 100 times
for _ in range(100):
    # E step
    # Q(U)
    for i in range(0, i_m):
        # this stores the matrix product (outer product in numpy)
        outer = np.zeros((d, d))
        N_i = N[i]
        for j in N_i:
            outer += np.outer(v[j], v[j])
        phi[i] = np.linalg.inv(
            np.diag(1 / sigma_2) + (psi[N_i].sum(0) + outer) / tau_2
        )
        mtplr = ((m[i, N_i] * (v[N_i])) / tau_2).sum(0)
        u[i] = phi[i].dot(mtplr)
        S[N_i] += (phi[i] + np.outer(u[i], u[i])) / tau_2
        t[N_i] += np.outer(csr_matrix.toarray(m[i, N_i]), (u[i])) / tau_2

    # Q(v)
    psi = np.linalg.inv(S)
    for j in range(j_m):
        v[j] = psi[j].dot(t[j])

    # M step
    for l in range(d):
        sigma_2[l] = ((phi[:, l, l] + u[:, l] ** 2).sum()) / (i_m - 1)

    K = len(ob[1])
    Tr = 0  # trace
    for i, j in np.array([ob[0], ob[1]]).T:
        A = phi[i] + np.outer(u[i], u[i])
        B = psi[j] + np.outer(v[j], v[j])
        Tr += np.trace(A.dot(B))

    tau_2 = (
        (
            (ob[2] ** 2) - (2 * ob[2] * np.einsum("ij,ij->i", u[ob[0]], v[ob[1]]))
        ).sum()
        + Tr
    ) / (K - 1)

    new_norm_u = np.linalg.norm(u)
    new_norm_v = np.linalg.norm(v)

    if abs(new_norm_u - norm_u) < 0.01 or abs(new_norm_v - norm_v) < 0.01:
        # early stopping
        break
    else:
        norm_u, norm_v = new_norm_u, new_norm_v
    yield np.array(u), np.array(v)
```

Outputs

```
❯ python homework.py
Please enter choice of dataset:

1. 500-500-3

2. 100-k

Please enter your choice: 1 (or) 2 [To Debug, type 0] - 1
Baseline rmse is:  1.8019448325127836
For Part 1:

Total Iterations:  25000

RMSE:  [1.781923883557573, 1.5937212241838992, 1.2321690119758026, 0.722144708569602, 0.3283120448423085, 0.23598144269743926, 0.21916363355676868, 0.2160546121083858
]
Time taken:  [0.48094797134399414, 0.8806238174438477, 1.3032078742980957, 1.7059125900268555, 2.105543613433838, 2.519871711730957, 2.923938751220703, 3.326377868652
3438]


For Part 2:
For the given ranks:  [1, 2, 3, 5, 10, 20]
The RMSE changes as follows:  [1.5068963472083776, 1.1600153878586528, 0.21661693422373104, 0.22146709229941794, 0.24548545758228488, 0.2738502473472803]
Please check your current directory for generated plots
```

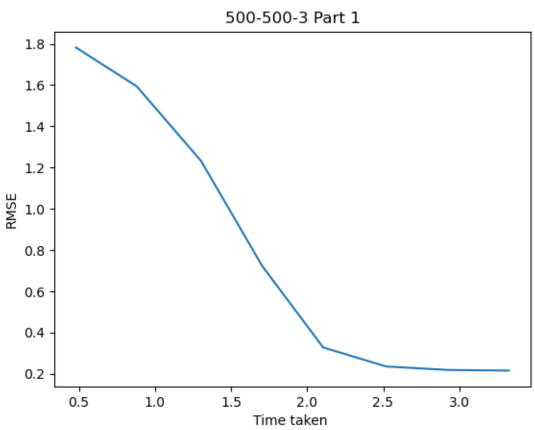
Figure 1: Output after running matrix factorization on 500-500 dataset

```
❯ python homework.py
Please enter choice of dataset:

1. 500-500-3

2. 100-k

Please enter your choice: 1 (or) 2 [To Debug, type 0] - 2
Baseline rmse is:  3.4397503673365066
For Part 1:

Total Iterations:  50000

RMSE:  [3.4112381981697038, 1.7459647590787626, 1.259336614373994, 1.0717570365475515, 1.0094145671801862, 0.9869535940433072, 0.9773663235991397, 0.9729829997699876,
 0.971110051528889, 0.9706030114922143, 0.970895231935521, 0.9716668351359257, 0.9727191894855078, 0.973922483582565, 0.9751909066372002, 0.9764687906820774, 0.977721
5207516838, 0.9789289840670536, 0.9800807532549812, 0.9811726218729699, 0.9822042020987551, 0.9831773211356803, 0.9840949888228743, 0.9849607561742121, 0.985778331668
2548, 0.9865513619941978, 0.9872833146365183, 0.9879774217741703, 0.9886366600621885, 0.9892637507258457, 0.9898611706150541, 0.9904311686869036, 0.9909757846923342,
0.9914968682259249, 0.9919960971235808, 0.9924749946841002, 0.9929349454815756, 0.9933772097052951, 0.9938029360595462, 0.9942131733075552, 0.9946088805674178, 0.9949
909364767786, 0.9953601473413681, 0.9957172543769532, 0.9960629401452848, 0.9963978342757025, 0.9967225185542518, 0.9970375314535498, 0.9973433721682833, 0.9976405042
139241, 0.9979293586393044, 0.9982103368982594, 0.9984838134197467, 0.9987501379115318, 0.9990096374283076]
Time taken:  [1.014092206954956, 1.9421393871307373, 2.8472652435302734, 3.7813050746917725, 4.687999963760376, 5.588575124740601, 6.562986135482788, 7.52633404731750
5, 8.519048929214478, 9.499677896499634, 10.39844298362732, 11.366978168487549, 12.267967224121094, 13.193376302719116, 14.053032159805298, 14.950924396514893, 16.002
52628326416, 16.864455461502075, 17.84455132484436, 18.75015616416931, 19.7137610912323, 20.627712965011597, 21.552350997924805, 22.5420880317688, 23.435638189315796,
 24.443799257278442, 25.36793327331543, 26.301979303359985, 27.26687240600586, 28.163235425949097, 29.044275283813477, 29.99522614479065, 30.884100437164307, 31.81834
840774536, 32.71967530250549, 33.687013149261475, 34.61783504486084, 35.56900691986084, 36.43928599357605, 37.2837769985199, 38.5317702293396, 39.508172273635864, 40.
4709312915802, 41.342119216918945, 42.29035234451294, 43.22833251953125, 44.223445415496826, 45.111032247543335, 46.00450420379639, 46.965280294418335, 47.98959326744
0796, 48.999062299728394, 50.0300931930542, 51.08395743370056, 52.17591118812561]


For Part 2:
For the given ranks:  [1, 2, 3, 5, 20, 100]
The RMSE changes as follows:  [0.9709810673837846, 0.9801935787643553, 0.9847149864732342, 1.0167913768950378, 1.1632308654873804, 1.1061426231943978]
Please check your current directory for generated plots
```

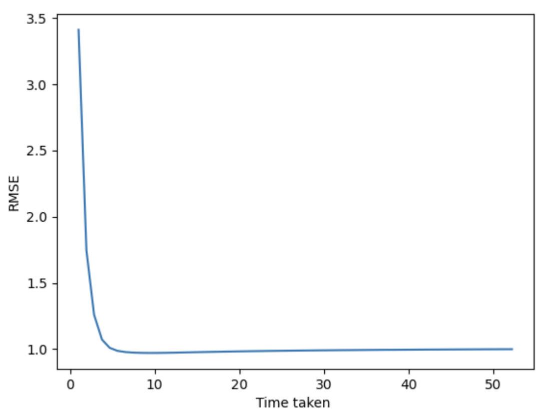
Figure 2: Output after running matrix factorization on 100-k dataset

Results

From the graphs below, we can see that the RMSE for both the datasets goes down per iteration, showing that they converge to a minimum.

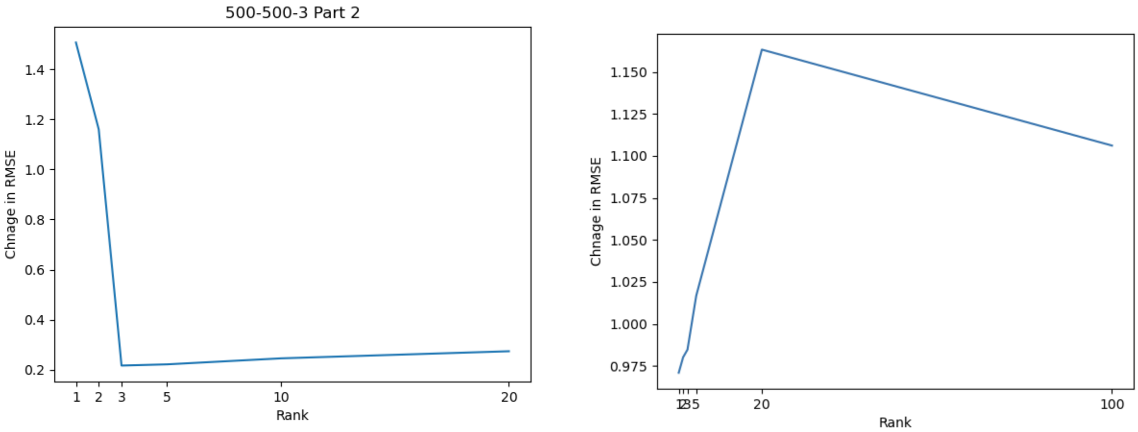


(a) Time v/s RMSE for 500-500 dataset



(b) Time v/s RMSE for 100k dataset

For the $500 - 500 - 3$ dataset, the RMSE falls sharply at rank 2 and 3 from rank 1. Following that the RMSE plateaus and rises only slightly. Meanwhile, in the 100k dataset, lowest RMSE is at rank 1 and rises sharply till rank 20 following which it falls as we move towards higher rank but not as low.



(a) Rank v/s RMSE for 500-500 dataset

(b) Rank v/s RMSE for 100k dataset

For both the datasets, the performance is hugely better than the baseline.