ESE 545 Project 4

Part 1 – Introduction

This program was written in Python 2.7.
The libraries and modules we used are:

numpy: used for matrix calculation.
time: used for calculate the time consumed.
matplotlib.pyplot: used for plotting curves and figures.

The program consists four python files: *readData.py*, *greedyAlgo.py*, *lazyAlgo.py*, and *project4_plot.py*, corresponding to different question part of this project, respectively.

The only thing you may need to modify is line 33 in *project4_plot.py,* where is the path of our dataset file.

Part 2 – Implementation

Question 1.1:
Read the data from *ratings.dat* and represent the data into matrix.
The code for this part is shown below, as written in *readData.py*:

```
"""
ESE 545 Project 4

@author: Yiidng Zhang, Xiangyang Cui
"""

import numpy as np

ROW = 3952 # number of distinct movies
COL = 6040 # number of distinct users

def readData(filePath):
    matrix = np.zeros((ROW, COL))
    with open(filePath, 'r') as file:
        for line in file:
            features = line.split('::')
            matrix[int(features[1])-1, int(features[0])-1] = int(features[2])
    return matrix
```

The core function is *readData*(filePath), which reads the raw data, cleans it, and represents the data in to a matrix. This function will be called in *greedyAlgo.py*, *lazyAlgo.py*.

We extracted the UserIDs and MovieIDs and Ratings information according to the format explained in *README.txt*. And take MovieIDs as row numbers of the matrix and UserIDs as column numbers of the matrix. Ratings as the values of elements of the matrix.

Question 1.2:

$$A \subseteq B \subseteq V$$

$$F(A) = \frac{1}{n}\sum_{i=0}^{n} \max_{j\in A} r_{i,j}$$

$$F(B) = \frac{1}{n}\sum_{i=0}^{n} \max_{j\in B} r_{i,j}$$

1) Monotonicity:

$$\forall \{i = k \mid k = 1, 2, 3, \cdots, n\}, \text{say } a_k = \max_{j\in A} r_{i,j}, b_k = \max_{j\in B} r_{i,j}$$

$$\because A \subseteq B, a_k \le b_k$$

$$\frac{1}{n}\sum_{i=0}^{n}(a_i - b_i) \le 0$$

$$\Rightarrow \frac{1}{n}\sum_{i=0}^{n} a_i - \frac{1}{n}\sum_{i=0}^{n} b_i \le 0$$

$$\Rightarrow F(A) \le F(B)$$

Which is monotonicity.

2) Sub-modularity:

Assume $s \in R^n, s \in V$ while $s \notin B$ and $s = [s_1, s_2, \cdots, s_n]$.

$\forall \{i = k \mid k = 1, 2, 3, \cdots, n\}$,

$$P_k = \max_{j\in(A\cup s)} r_{k,j} = \max\{a_k, s_k\}$$

$$Q_k = \max_{j\in(B\cup s)} r_{k,j} = \max\{b_k, s_k\}$$

Since we know $a_k \le b_k$,

$$\begin{cases} s_k \le a_k \le b_k \Rightarrow P_k = a_k \le Q_k = b_k \Rightarrow P_k - a_k = Q_k - b_k \\ a_k \le s_k \le b_k \Rightarrow a_k \le P_k = s_k \le Q_k = b_k \Rightarrow P_k - a_k \ge Q_k - b_k \\ a_k \le b_k \le s_k \Rightarrow P_k = s_k = Q_k \Rightarrow P_k - a_k \ge Q_k - b_k \end{cases}$$

$$\Rightarrow P_k - a_k \ge Q_k - b_k$$

$$\Rightarrow \max_{j\in(A\cup s)} r_{k,j} - \max_{j\in A} r_{i,j} \ge \max_{j\in(B\cup s)} r_{k,j} - \max_{j\in B} r_{i,j}$$

$$F(A\cup s) - F(A) \ge F(B\cup s) - F(B)$$

which is sub-modularity.

Question 1.3:

As we learned in class, the Greedy Algorithm is:

```
Start with $A_0 = \{\}$
For i = 1 to k
    $e^* = arg \max_{e\in V} F(A \cup \{e\})$
    $A_i = A_{i-1} \cup \{e^*\}$
```

We implement this algorithm in *greedyAlgo.py*, as below:

```
import time
from readData import *

def greedyAlgo(data, k = 50):
    recommendation = [] # The movies selected
    benefits = [] # F(A), or objective value
    time_stamp = []
    corr_max = np.zeros((1,COL)) # initial A0, empty

    start = time.time()
    for t in range(1, k + 1):
        new_matrix = np.maximum(data, corr_max)
        avg_rating = new_matrix.sum(axis=1) / COL # calculate f(A) for each
row
        target_idx = np.argmax(avg_rating,axis=0) # find e*
        corr_max = new_matrix[target_idx,:] # equivlent to F(A) Union e
        recommendation.append(target_idx)

        if t % 10 == 0:
            benefits.append(np.sum(corr_max) / COL)
            time_stamp.append(time.time() - start)

    return (recommendation, benefits, time_stamp)
```

As we can see, with data read in Question 1.1, we can calculate k movies to recommend. In order to find the best one to recommend in t+1 round, we used *corr_max* to record highest ratings each user gave for t movies acquired in previous t rounds which equivalent to F(A) U {e}. So in t+1 round, we need to find a movie which can maximize every element of the *corr_max*. This was implemented by *numpy.argmax()* method.

Question 1.4:

As we learned in class, the Lazy Greedy Algorithm is very similar to Greedy Algorithm, the only difference how to choose marginal benefit in each iteration:

First Iteration as usual (compute all the marginals in the first round).
Keep an ordered list of marginal benefits Δ_i from previous iteration.
Re-evaluate Δ_i only for the top element.
If the re-evaluated Δ_i stays on top, use it, otherwise re-sort.

We implement this algorithm in *lazyAlgo.py*, as below:

```
import time
from readData import *

def lazyAlgo(data, k = 50):
    recommendation = [] # The movies selected
    benefits = [] # F(A), or objective value
    time_stamp = []

    prev_max = np.zeros((1,COL)) # initial A0, empty
    prev_benefit = 0 # F(A0) = 0
    # Iteration 1
    start = time.time()
    avg_rating = np.sum(data, axis=1) / COL # calculate F(A) for each row
    indices = np.argsort(avg_rating,axis=0)[::-1] # sort the marginal benefits
    recommendation.append(indices[0]) # pick the row with largest marginal
benefit
    corr_benefit = avg_rating[indices[0]] # F(A1) = F(e*)
    corr_max = data[indices[0]] # A1 = e*
    # Subsequent iterations
    i = 1
    for t in range(2, k + 1):
        corr_delta = marginalBenefit(data, indices[i], corr_max, corr_benefit)
# like Del(e2|A1)
        prev_delta = marginalBenefit(data, indices[i+1], prev_max,
prev_benefit) # like Del(e3|A0)
        if corr_delta >= prev_delta:
            recommendation.append(indices[i])
            prev_max = corr_max
            corr_max = np.maximum(data[indices[i]], corr_max) # A Union e*
            prev_benefit = corr_benefit
            corr_benefit = corr_delta + corr_benefit
            i = i + 1
        else:
            data = np.maximum(data, corr_max)
            avg_rating = np.sum(data, axis=1) / COL
            indices = np.argsort(avg_rating,axis=0)[::-1] # re_sort according
to marginal benefits
            recommendation.append(indices[0])
            corr_benefit = avg_rating[indices[0]]
            corr_max = data[indices[0]]
            i = 1
        if t % 10 == 0:
            benefits.append(corr_benefit)
            time_stamp.append(time.time() - start)
    print("")
    return (recommendation, benefits, time_stamp)

def marginalBenefit(data, idx, corr_max, benefit):
    return np.maximum(data[idx], corr_max).sum() / COL - benefit
```

As we can see, this time, we used *indices*, which is an ordered list, to record the indices of marginal benefits from previous iteration. And each time, we only re-evaluate the marginal benefit only for the top element. If the re-evaluated marginal benefit stays on top, use it, otherwise re-sort.

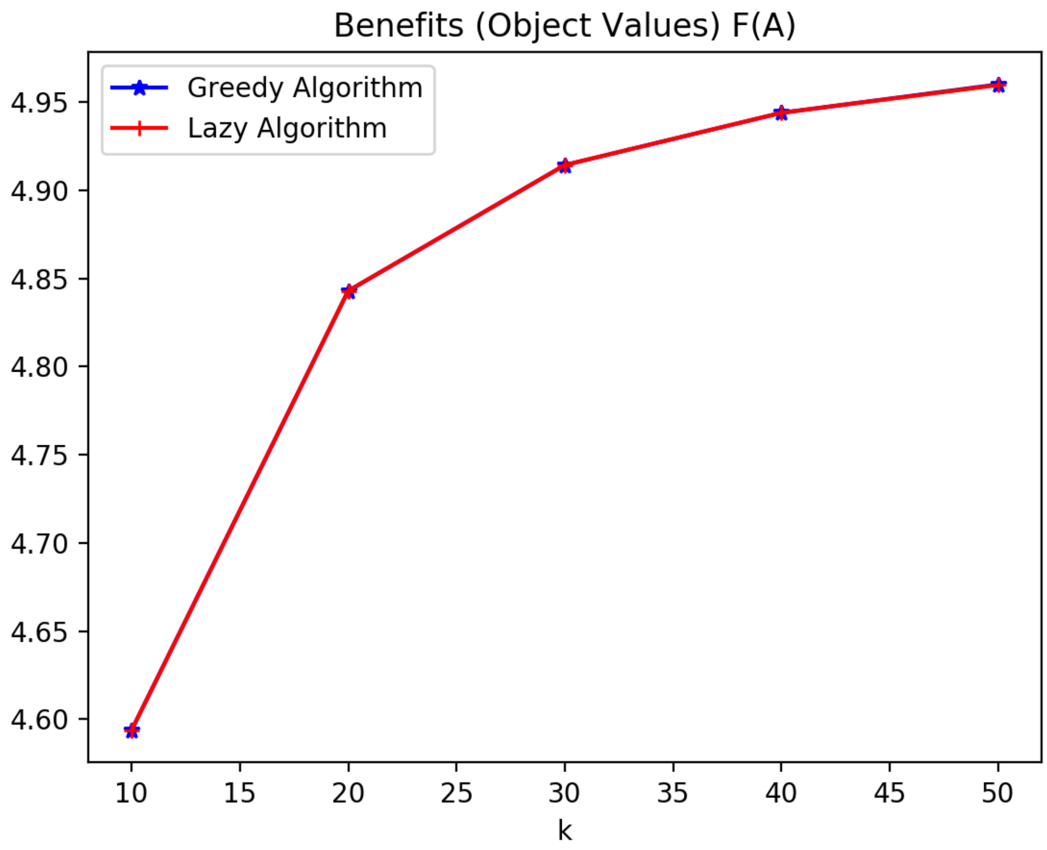
The main method to call methods above is:

```
import matplotlib.pyplot as plt
from greedyAlgo import *
from lazyAlgo import *

def plotBenefits(G_bnft, L_bnft, k_range):
    plt.plot(k_range, G_bnft, 'b*-', label="Greedy Algorithm")
    plt.plot(k_range, L_bnft, 'r+-', label="Lazy Algorithm")
    plt.title("Benefits (Object Values) F(A)")
    plt.xlabel("k")
    plt.legend()
    plt.show()

def plotTimeStamps(G_time, L_time, k_range):
    plt.plot(k_range, G_time, 'b*-', label="Greedy Algorithm")
    plt.plot(k_range, L_time, 'r+-', label="Lazy Algorithm")
    plt.title("Running Time")
    plt.xlabel("k")
    plt.ylabel("Time /sec")
    plt.legend()
    plt.show()

def main(k = 50):
    print("Reading data...")
    data = readData("ml-1m/ratings.dat")
    print("Complete!\n")

    print("Greedy Algorithm...")
    (G_rcmd,G_bnft,G_time) = greedyAlgo(data, k)
    print("movie IDs to be recommended through Greedy Algorithm:")
    print(G_rcmd)

    print("\nLazy Algorithm...")
    (L_rcmd,L_bnft,L_time) = lazyAlgo(data, k)
    print("movie IDs to be recommended through Lazy Algorithm:")
    print(L_rcmd)

    k_range = np.array([10, 20, 30, 40, 50])
    plotBenefits(G_bnft, L_bnft, k_range)
    plotTimeStamps(G_time, L_time, k_range)

if __name__ == "__main__":
    main()
```

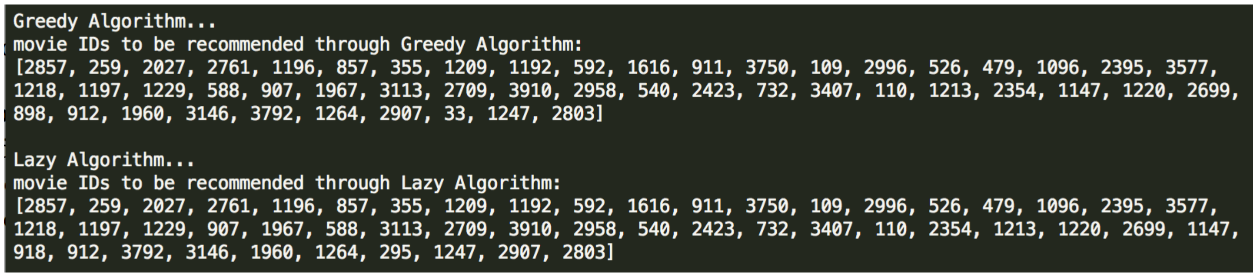
As we can see, the main method first called *readData()* to read the *ratings.dat*. Then, it ran Greedy Algorithm and Lazy Greedy Algorithm one by one. Each algorithm, it would get recommend movies, object values and time stamps for k is 10, 20, 30, 40, and 50. Last, it draws two plots, first is Object Values vs k, while the second is Running Times vs k.

Part 3 – Results & Analysis

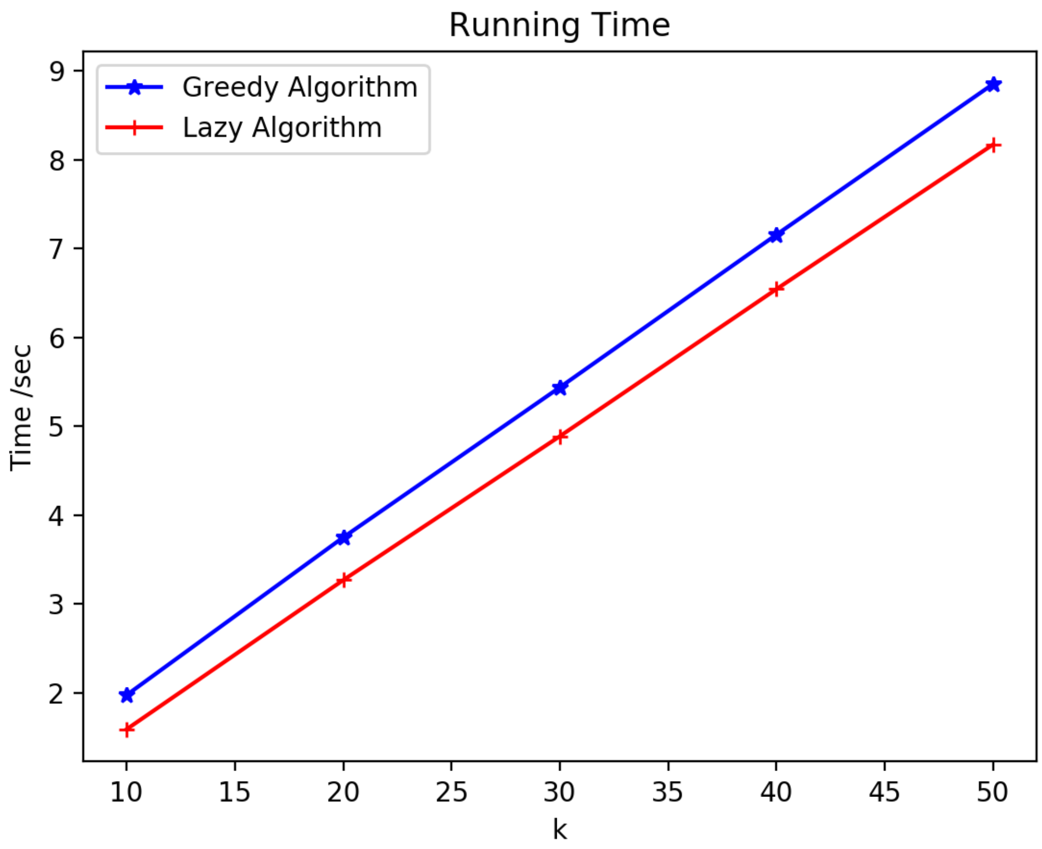
The following figure shows the objective values versus k for k = 10, 20, 30, 40, 50 using Greedy Algorithm and Lazy Greedy Algorithm:



As we can see, with increasing of k, the object value logarithmically increases. For both algorithm, the results are almost the same.


```
Greedy Algorithm...
movie IDs to be recommended through Greedy Algorithm:
[2857, 259, 2027, 2761, 1196, 857, 355, 1209, 1192, 592, 1616, 911, 3750, 109, 2996, 526, 479, 1096, 2395, 3577,
1218, 1197, 1229, 588, 907, 1967, 3113, 2709, 3910, 2958, 540, 2423, 732, 3407, 110, 1213, 2354, 1147, 1220, 2699,
898, 912, 1960, 3146, 3792, 1264, 2907, 33, 1247, 2803]

Lazy Algorithm...
movie IDs to be recommended through Lazy Algorithm:
[2857, 259, 2027, 2761, 1196, 857, 355, 1209, 1192, 592, 1616, 911, 3750, 109, 2996, 526, 479, 1096, 2395, 3577,
1218, 1197, 1229, 907, 1967, 588, 3113, 2709, 3910, 2958, 540, 2423, 732, 3407, 110, 2354, 1213, 1220, 2699, 1147,
918, 912, 3792, 3146, 1960, 1264, 295, 1247, 2907, 2803]
```

But if we check the indices of movies recommended, we can see there are two movies are different: 918 and 295. And the order of several of others are different.



For Greedy Algorithm, this took us 9.12 seconds to finish the algorithm. Since we used a lot of time to improve our code efficiency by numpy matrix features and numpy methods, it doesn't take too much time. With unimproved code which we programed first, it costed near 1800 seconds to finish Greedy Algorithm.

For the Lazy Greedy Algorithm, it used near 8.1 seconds which was shorter than Greedy Algorithm.