

1. Summary of your algorithm

- GradientDescent라는 class를 만들어서 rating matrix를 matrix factorization을 하여 점수를 예측함.
- latent feature의 경우, 그 크기를 변화시키면서 RMSE 값의 변화를 관찰함
- GradientDescent에서는 단순히 user vector와 item vector 뿐만 아니라 각각의 user와 item에 대한 bias, 그리고 전체 matrix의 bias가 있다고 가정하여 진행을 함. 전체 matrix bias는 사용자가 응답한 점수의 평균을 활용하였다.
- user vector와 item vector 그리고 각각의 bias들은 user i와 item j만을 고려하여 업데이트를 진행하였고, 따라서 반복 횟수를 x라고 하면 총 $x * \# \text{ user } * \# \text{ item}$ 반복하였다. 따라서 각각의 user와 item에 접근할 때마다 예측하는 matrix를 변화시키면 그 속도가 느려져 각각의 점에서 필요한 정보만을 얻기 위해 predict라는 메소드를 활용하였다.
- 학습이 원활하게 되기 위해서 regularization term을 넣었다.
- 학습은 RMS Error가 0.001이 될 때까지 진행이 되었다.

2. Detailed description of your codes

- 유사한 사용자 추출 : collaborate filtering에서는 처음, rating을 예측하고자 하는 사용자와 유사한 사람을 추려서 rating을 계산하게 된다. 따라서 cosine similarity을 활용해서 threshold보다 높으면 같은 취향을 가진 사람이라고 판단하여, 그 사람들만을 가지고 MF를 진행하였었다. 하지만 그렇게 진행한 결과는 RMSE가 1.12정도로 나오게 되었다. cosine similarity를 높게 설정하거나 낮게 설정해도 그 결과는 크게 달라지지 않았고 오히려 유사한 사람들을 추리는 과정이 들어가면 rating을 예측하는데 시간이 더 걸려서 적절하지 않다고 생각하였다. 따라서 유사한 사용자를 추출하는 과정을 하지 않고 전체의 데이터를 가지고 학습을 진행했을 경우 이보다 높은 RMSE를 보여 유사한 사용자 추출 단계를 건너뛰었다.
- hyperparameter 설정 : 여기서 hyperparameter는 종료 조건, learning rate, regularization term coefficient, latent factor가 존재한다. 이 때 learning rate의 경우 그 크기가 너무 크면 최소값에 찾아가는 것이 아닌 발산을 해버려서, 0.9 이상으로는 설정을 할 수가 없었다. 또한 그 값이 너무 작으면 학습이 되는데 시간이 오래 걸렸다. 따라서 실험적으로 나머지 hyperparameter를 고정시킨 상태에서 learning rate이 0.01 정도면 충분히 작아서 발산을 하지 않으면서도 충분히 커서 빠른 시간 안에 학습을 할 수 있었다. regularization term coefficient의 경우도 learning rate와 마찬가지로 그 크기가 크면 발산을 하고, 그 크기가 작으면 regularization term의 역할을 제대로 하지 못한다. 따라서 다른 hyperparameter를 고정시킨 뒤 그 값을 찾았을 때, 그 값의 변화에 따라서 0.01 - 0.5까지는 큰 성능변화를 나타내지 않아서 가장 작은 값인 0.01을 활용하였다. 그 다음 종료 조건은 그 크기에 따라서 성능에 많은 영향을 미쳤다. 0.1인 경우에는 그 성능이 RMSE가 약 2 정도가 나왔고, 그 크기를 줄일수록 1.2에 근접해지다가 그 크기가 너무 작아지면,

다시 RMSE가 커지고 학습 시간 또한 많이 걸리게 되었다. 이러한 이유는 처음에 종료조건이 너무 컸을 경우는 데이터의 특징을 충분히 학습하지 못한 경우이며 그 조건을 더 줄이면서 학습이 원활하게 이루어지다가 너무 조건이 커지는 경우에는 주어진 데이터에 overfitting이 되며 오히려 안 좋은 영향을 미치는 것이라고 생각했다. latent factor의 경우, 그 크기가 너무 크면 충분히 학습이 되지 않아서 RMSE가 낮게 측정이 되었다. latent factor의 크기를 30으로 했을 때, RMSE는 약 1.7을 보였으며, 그 크기를 줄이면서 1.2에 가까워지다가 그 크기를 1로 하였을 때, 약 1.05 정도의 크기를 보였다. 우선 latent factor가 많은 경우, 학습할 데이터의 양이 충분하지 않아서 제대로 학습이 되지 않은 것이라고 생각하여 가장 적절한 값을 찾았다.

- test 데이터의 user와 item을 모두 포함하기 위해서 train과 test data set을 읽어 드린 뒤, 그 중 가장 큰 user id 혹은 item id를 가진 것을 기준으로 matrix를 생성하였다. 이는 u1-u4까지는 user id와 item id가 train이 전부 크거나 같지만, u5의 경우 test data의 id가 더 큰 경우가 있어 이 때는 test id를 기준으로 rating matrix를 만들었다. test data의 rating은 들어가지 않았고 단순히 그 크기만을 고려하여 test data가 학습에 영향을 미치지 않는았다.

```
def globalBiasSet(self):
```

```
    # global bias를 평균으로 설정하였다. 학습을 시킬 때, 1 - 3까지 수를 넣어서 진행을 해보았지만 적절한 방식이 아니라고 생각하여서 평균으로 진행 하였다.
    self.global_bias = np.mean(self.data[self.mask])
```

```
def predict(self,x,y):
```

```
    # user x가 item y에 대한 rating을 구하는 식이다.
    prediction = self.global_bias + self.user_bias[x] + self.item_bias[y] +
self.user_vectors[x,:].dot(np.transpose(self.item_vectors[y,:]))
    return prediction
```

```
def iteration(self):
```

```
    for i in range(self.users):
        for j in range(self.items):
            # 입력된 데이터만을 고려하여 학습을 진행
            if self.data[i,j] > 0:
                # 필요한 예측값을 계산
                prediction = self.predict(i,j)
                # error 계산
                error = self.data[i,j] - prediction
                # user bias, item bias를 error를 기반으로 update
                self.user_bias[i] += self.learning_rate * (error - self.regular *
self.user_bias[i])
                self.item_bias[j] += self.learning_rate * (error - self.regular *
```

```

self.item_bias[j])

        # user vector, item vector를 error 기반으로 update
        du = (error * self.item_vectors[j,:]) - (self.regular *
self.user_vectors[i,:])
        di = (error * self.user_vectors[i,:]) - (self.regular *
self.item_vectors[j,:])

        self.user_vectors[i,:] += self.learning_rate * du
        self.item_vectors[j,:] += self.learning_rate * di
        # 모든 item과 user에 대해서 학습을 진행했으면 prediction update
        self.prediction = self.global_bias +
self.user_vectors.dot(np.transpose(self.item_vectors)) + self.user_bias[:,np.newaxis] +
self.item_bias[np.newaxis,:]
        # 입력한 matrix와 예측한 matrix의 Error 측정
        cost = 0
        count = 0
        for i in range(self.users):
            for j in range(self.items):
                if self.data[i,j] > 0:
                    count += 1
                    cost += np.power(self.data[i,j] - self.prediction[i,j],2)
        cost = np.sqrt(cost)
        cost /= len(self.mask[0])

        cost = np.sqrt(cost/count)

        self.cost = cost

def train(self):
    # global bias 설정
    self.globalBiasSet()

    while True:
        # cost가 기준보다 높은 경우 학습을 진행
        self.iteration()
        # cost가 기준보다 낮은 경우 학습을 멈추고 결과 저장
        if self.cost < self.crit:
            self.model = self.prediction
            break
- RMSE 함수는 예측한 matrix의 결과를 반올림하여 정수로 변환하고 오차를 계산해주는 함수이다.

```

```

def RMSE(post,label):
    output = 0
    # test의 개수 확인
    (row,column) = label.shape
    # output file의 내용 저장 배열 preallocation
    array = np.zeros(row*3).reshape(row,3)
    for i in range(row):
        # 결과 저장, rating의 경우 반올림하여 저장
        predict = post[label[i,0]-1,label[i,1]-1]
        array[i,0] = label[i,0]
        array[i,1] = label[i,1]
        array[i,2] = np.around(predict)
        if array[i,2] > 5:
            array[i,2] = 5
        elif array[i,2] < 1:
            array[i,2] = 1
        answer = label[i,2]
        output += np.power((predict-answer),2)
    return np.sqrt(output/row), array

```

3. Instructions for compiling your source codes at TA's computer
python recommender.py ux.base ux.test 입력

```

C:\Users\이찬호\Documents\GitHub\DataScienceProject\LongTermProject>python recommender.py u1.base u1.test
running time : 6.515318155288696
Test Result : 1.2289753299320718

```

4. Any other specification of your implementation and testing

- preuseMatrix의 활용 시도 : 연구 결과 중에서 preusematrix를 활용해서 사용자의 사전 인지 정보를 예측하여 이를 가지고 성능을 향상 시킨 방식이 있다고 언급을 하셔서 구현하기 위해서 진행을 해봄. zero-injection 방식의 경우 존재하지 않은 점수인 0점을 넣는 것이기 때문에 1점이나 2점을 preusematrix의 예측된 값에 따라서 넣는 것을 진행해 봄. 하지만 preusematrix를 활용한 경우 그 결과의 RMSE가 1.7정도로 높게 나오고, 사용하지 않았을 경우 1.2 정도로 낮게 나와서 사용하지 않음. 이 방법이 실패한 이유는 단순히 threshold를 잡아서 1과 2를 할당하였는데 이는 중요한 데이터 또한 복구를 해주지만, 오히려 데이터가 적절하지 않은 것이 들어가는 경우가 많아 오히려 학습을 방해하여 이러한 결과가 나오는 것이라고 생각함.
- WRMF의 활용 시도 : WRMF의 방식은 Gradient Descent 방식과 달리 한 값이 존재한다고 가정을 하여 고정을 시키고 그 값을 가지고 다른 값을 찾는 방식이다. 즉 user vector를 안다고 가정하고 item vector를 찾고 item vector를 안다고 가정하고 user vector를 찾는 방식이다. 이 방식은 여러 번 반복을 하게 된다면 알맞은 답을 낼 수 있다. 하지만 개인적으로 구현한 코드는 항상 0과 1 사이의 값만을 가지게 되어서 rating matrix에 활용을

할 수 없었고, 앞서 말한 것처럼 preuseMatrix를 예측하는데 사용하여 좋은 효과를 보여주었지만, preuseMatrix를 활용하는데 문제가 있어서 WRMF는 구현하였지만 사용하지 못하였다.