Assignment 1

Jun Wang

Date:2022-11-9

Task 1 - Predicting the Read

Description of the problem

In this task, we need to predict given a (user,book) pair from 'pairs Read.csv' whether the user would read the book (0 or 1). Accuracy will be measured in terms of the categorization accuracy (fraction of correct predictions). The test set has been constructed such that exactly 50% of the pairs correspond to read books and the other 50% do not.

Dataset

train_Interactions.csv.gz

200,000 ratings to be used for training. This data should be used for the 'read prediction' (both classes) and 'rating prediction' **(CSE258 only)** tasks. It is not necessary to use *all*
ratings for training, for example if doing so proves too computationally intensive.
userID The ID of the user. This is a hashed user identifier from Goodreads.
bookID The ID of the book. This is a hashed book identifier from Goodreads.
rating The star rating of the user's review.

- **Preprocess for the dataset**

Frist, read the dataset and split them into train set and valid set:

```
# input file using gzip
path = "train_Interactions.csv.gz"
f = gzip.open(path,"rt",encoding="utf8")
reader = csv.reader(f,delimiter = ",")
```

```
# reading the file to build dataset
dataset = []
first = True
for line in reader:
    if first:
        header = line
        first = False
    else:
        d = dict(zip(header,line))
        # convert strings to integers for some fields
        d["rating"] = int(d["rating"])
        dataset.append(d)
```

```
# split the training data
data_train = dataset[:190000]
data_valid = dataset[190000:]
```

There is a problem here: since all the dataset is positive, we need to build some negative dataset for the pairs.

```
# negative validation data
negative_valid_dict = {}
for c in data_valid:
    bid = random.choice(unique_books)
    uid = c["userID"]
    while bid in UsersReadBooks[c["userID"]]:
        bid = random.choice(unique_books)
    if negative_valid_dict.get(uid):
        negative_valid_dict[uid].append(bid)
    else:
        negative_valid_dict[uid]= [bid]
```

Then, prepare the neccessary data structure:

```
UsersPerBook = defaultdict(set)
BooksPerUser = defaultdict(set)

for i in data_train:
    TrainUserID.append(i["userID"])
    TrainBookID.append(i["bookID"])
    UsersPerBook[i["bookID"]].add(i["userID"])
    BooksPerUser[i["userID"]].add(i["bookID"])
```

```
bookCount = defaultdict(int)
userCount = defaultdict(int)
totalRead = 0

for c in dataset:
    user,book = c["userID"],c["bookID"]
    bookCount[book] += 1
    userCount[user] += 1
    totalRead += 1
```

Methods

In homework3, we notice that 75% can be obtained based on Book popularity. So we should make the best use of it. Also we have the Jaccard similarity, which ranges from 0 to 1. Here is the idea, we can create a new variable c = Jaccard similarity * book counts. Based on the new variable, we can make a list and sort them. If the pair (u,b)'s c is in the top x percent of the ranking, we predict y to be "1". Otherwise, we predict y to be 0.

```
# Jaccard Predictor
def Jaccard(s1, s2):
    numerator = len(s1.intersection(s2))
    denominator = len(s1.union(s2))
    if(denominator == 0):
        return 0
    return numerator/denominator
```

Based on the similarity, we have lots of things to improve. That is what similarity we should choose, and whether the mean or max or min of the similarity list we should choose.

For the first questions, since it is the ranking for items, we should calculate the similarity based on items. In the following function, we can get a 79% accuaracy.

```
def mostSimilarFast(user, book):
    similarities = []
    books = BooksPerUser[user]

    for b in books:
        if b == book:
            continue
        users = UsersPerBook[b]
        sim = Jaccard(users, UsersPerBook[book])
        similarities.append(sim)

    if len(similarities) > 0 :
         mean = sum(similarities)/len(similarities)
    else:
         mean = 0

    return mean
```

But, in some case, the item is not seen. We need to use the infomation from the users, so we can interchange user and item and set a factor alpha for them.

```
def mostSimilarFast_improve2(user, book, alpha = 0.5):
    similarities1 = []
    similarities2 = []

    books = BooksPerUser[user]
    users = UsersPerBook[book]

    for b in books:
        if b == book: continue

        users = UsersPerBook[b]
        sim = Jaccard(users, UsersPerBook[book])
        similarities1.append(sim)

    if len(similarities1) > 0 :
           mean1 = sum(similarities1)/len(similarities1)
    else:
           mean1 = 0

    for u in users:
        if u == user: continue
        books = BooksPerUser[u]
        sim = Jaccard(books, BooksPerUser[user])
        similarities2.append(sim)

    if len(similarities2) > 0 :
           mean2 = sum(similarities2)/len(similarities2)
    else:
           mean2 = 0

    #there is a factor alpha that decides which part is more important in the similarity rating
    mean = alpha * mean1 + (1-alpha) * mean2
```

mean = alpha*mean1 + (1–alpha)*mean2

Fintuning the alphas: Get a 81% accuracy and best alpha is 0.97 which makes sense that the unseen item situation is rare.

```
# fintuning
# In the first try, the best occurs when alphas in 0.9 - 1
alphas = [i/100 for i in range(90,101)]
```

```
max_acc = max(accs)
best_alpha = alpha_acc[max_acc]
print("My best accurate %f" % max_acc)
print("The best alpha is %f " % best_alpha)
```

```
My best accurate 0.815750
The best alpha is 0.970000
```

Also fintuning the thresholds:

```
threshold_acc = defaultdict()
for threshold in thresholds:
    y_pred = []
    for x in X_valid:
        u = x[0]
        b = x[1]
        i = (u,b)
        book_sim_list = user_book_sim[u]
        p = 1
        for n in range(int(len(book_sim_list)*threshold)):
            if i in book_sim_list[n]:
                p = 0
        y_pred.append(p)
    acc = accuracy_score(y_pred, y_valid)
    accs.append(acc)
    threshold_acc[acc] = threshold
```

```
max_acc = max(accs)
best_threshold = threshold_acc[max_acc]
print("My best accurate %f" % max_acc)
print("The best threshold is %f " % best_threshold)
```

```
My best accurate 0.815850
The best threshold is 0.624000
```

Task 2 - Predicting the Rate

Description of the problem

Predict people's star ratings as accurately as possible, for those (user,item) pairs in 'pairs Rating.txt'. Accuracy will be measured in terms of the *mean-squared error* (MSE).

- **Preprocess for the dataset**

Frist, read the dataset and split them into train set and valid set like the task 1.

Prepare the neccessary data stucture: beta_user, beta_book

```
# calculate initial value of alpha,beta_user and beta_book
alpha = globalAverage
userRatings = defaultdict(list)
bookRatings = defaultdict(list)

for l in data_train:
    user, book = l["userID"], l["bookID"]
    userRatings[user].append(l["rating"])
    bookRatings[book].append(l["rating"])

# beta_user,beta_book
userBias = defaultdict(float)
for u in userRatings:
    userBias[u] = globalAverage - (sum(userRatings[u]) / len(userRatings[u]))

bookBias = defaultdict(float)
for b in bookRatings:
    bookBias[b] = globalAverage - (sum(bookRatings[b]) / len(bookRatings[b]))
```

- **Methods: latent factor models**

Traditional one:

Rating prediction

Iterative procedure – repeat the following updates until convergence:

$$\alpha = \frac{\sum_{u,i \in \text{train}} (R_{u,i} - (\beta_u + \beta_i))}{N_{\text{train}}}$$

$$\beta_u = \frac{\sum_{i \in I_u} R_{u,i} - (\alpha + \beta_i)}{\lambda + |I_u|}$$

$$\beta_i = \frac{\sum_{u \in U_i} R_{u,i} - (\alpha + \beta_u)}{\lambda + |U_i|}$$

(exercise: write down derivatives and convince yourself of these update equations!)

Here I use different lambdas for user-bias and book-bias, and finetune the factors based on these two lambda:

```
#Find the best lambda
```

```
lamb1 = [i for i in range(1,20)]
lamb2 = [i for i in range(1,20)]
mse = []
lambs_mse = defaultdict(list)
#when optimal the lamb1,epsilon should be bigger
for r1 in lamb1:
  for r2 in lamb2:
    globalAverage_new, userBias_new, bookBias_new = differenciate(
        globalAverage, userBias, bookBias, r1,r2, epsilon=1e-05
    )
    m = MSE(data_valid)
    lambs_mse[m]=[r1,r2]
    mse.append(m)
```

```
print(lambs_mse)
```

```
# Optimal λ
best_mse = min(mse)
best_lamda = lambs_mse[best_mse]
print("Optimal λ:", best_lamda, "; MSE:", best_mse)
```

updata alpha:

Here alpha = globalAverage:

```
# update alpha
globalAverage = 0
for i in dataset:
    user, book = i["userID"], i["bookID"]
    globalAverage += i["rating"] - userBias_last[user] - bookBias_last[book]

globalAverage = globalAverage / len(dataset)
```

updata beta:

```
# update beta_user
num_book = defaultdict(int)
for u in userBias:
    userBias[u] = 0
for i in dataset:
    user, book = i["userID"], i["bookID"]
    num_book[user] += 1
    userBias[user] += i["rating"] - globalAverage - bookBias_last[book]
for u in userBias:
    userBias[u] = userBias[u] / (lamb1 + num_book[u])

# update beta_book
num_user = defaultdict(int)
for b in bookBias:
    bookBias[b] = 0
for i in dataset:
    user, book = i["userID"], i["bookID"]
    num_user[book] += 1
    bookBias[book] += i["rating"] - globalAverage - userBias[user]
for b in bookBias:
    bookBias[b] = bookBias[b] / (lamb2 + num_user[b])
```

End of iteration: set a bound

```
cost = MSE
for u in userBias:
    cost += lamb1 * (userBias[u]**2)
for i in bookBias:
    cost += lamb2 * (bookBias[i]**2)


if (
    abs(MSE - MSE_last) < epsilon
    and abs(cost - cost_last) < epsilon
):
    end = True
else:
    globalAverage_last = globalAverage
    userBias_last = userBias
    bookBias_last = bookBias
    MSE_last = MSE
    cost_last = cost
```

Fine-tune the bound

```
#finetuning the epsilon
```

```
best_lamda1 = 2
r1 = best_lamda[0]
r2 = best_lamda[1]
lamb2 = [i for i in range(1,20)]
mse = []
epsilon_mse = defaultdict()
epsilons = [10**(-i) for i in range(3,6)]
for epsilon in epsilons:
    globalAverage_new, userBias_new, bookBias_new = differenciate(
        globalAverage, userBias, bookBias, r1,r2, epsilon
    )
    m = MSE(data_valid)
    epsilon_mse[m]=r1
    mse.append(m)
```

Finally, use all the dataset to predict the rate.

I have also tried to use some SVD models to do it, but the results are not better than the latent factor one.