Task 1 (visit prediction)

(Kaggle) user name: yfzhou13 display name: Vidal

Method

We decide to apply One Class Recommendation model to this prediction.

The reason is that in this task, what need to predict is only visit or non-visit. It is quite like matrix factorization problem where all the elements are zero or one instead of other numerical feedback. So we define positive feedback and negative feedback as follows:

- Positive feedback: $P_u = \{i\}$ is defined as the set of items i towards which user u shows explicit positive feedback.
- Negative feedback: $N_u = \{j\}$ is a set of items j towards which user does not shown positive feedback until now.

In this particular dataset, each record of user visiting business is seen as positive feedback. Then We assume that users prefer positive feedback business to those negative ones, which is captured by such an objective function:

$$\max_{u,v} \sum_{u,i,j} \ln \sigma(u_u v_i - u_u v_j)$$

Here $u_u v_i$ and $u_u v_j$ stands for user u's preferences towards business i and j, built by a latent factor model.

When conducting optimization, the objective actually changes by adding the regularization component.

$$\max_{u,v} \sum_{u,i,j} \ln \sigma(u_u v_i - u_u v_j) - \sum_u \left||u_u|\right|_2^2 - \sum_i \left||v_i|\right|_2^2$$

Since the number of entire possible pairs of all positive feedback and negative feedback for every user are extremely large, we optimize it based on stochastic gradient ascent. Specifically, for one user, some of its positive and negative feedback pairs are chosen, which contribute to gradients updating in one iteration.

Implementation

Particularly, here are some equations for gradient ascent updating.

$$z = \sigma(u_u v_i - u_u v_j)$$

$$\frac{\partial L}{\partial u_u} = \sum_{(i,j)} (1 - z)(v_i - v_j) - \lambda u_u$$

$$\frac{\partial L}{\partial v_i} = \sum_{(u)} (1 - z)(u_u) - \lambda v_i$$

$$\frac{\partial L}{\partial v_j} = \sum_{(u)} (1 - z)(-u_u) - \lambda v_i$$

To monitor what's going on during the training, we output the value of original objective function, regularized objective function, and also performance on validation set after several iterations (Figure 1).

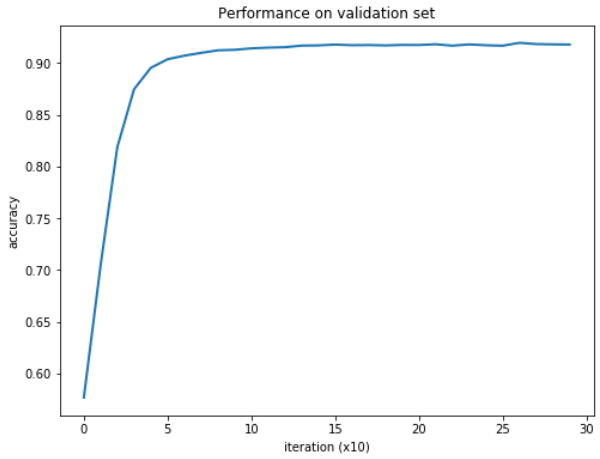

Figure 1

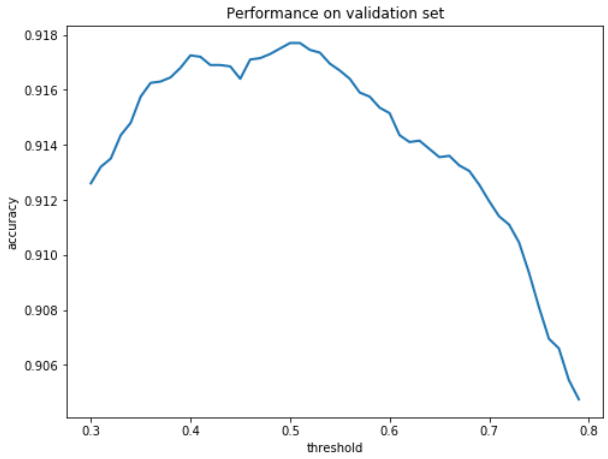

Figure 2

Observation: Training is more efficient if users are covered as many as possible even at the cost of reducing sampling pairs of one user.

It is not difficult to guess that the user-business matrix is quite sparse, in which there are only a few positive label for almost all users. This means if we choose too many positive/negative pairs for one user, it is highly possible that lots of positive records will be repeated more than once. So during gradient updating, it seems more necessary to satisfy 'breath' requirement instead of 'depth'.

Parameters

There are also some parameters we can control, like reduction dimension and regularization parameter. In addition to regularization parameter. For example, in evaluating, how to choose the threshold which determines visit or non-visit.

Reduction dimension K and regularization λ

Technically, as K becomes larger, the performance should be better in training, validation and test set. Practically, we find that, for example when K is chosen from 100, 200, 400, 800, the training and validation performance do not show huge difference. However, on test dataset, larger K reveals stronger ability of generalization obviously. Generalization here points to a model trying to reduce overfitting on training and validation dataset, and keep the prediction ability as accurate as possible. Finally we maintain $K = 800$.

Following the pipeline of choosing optimal λ, finial $\lambda = 0.6$

Threshold

(a) Threshold as a number.

It's hard to estimate a threshold only with prior knowledge. Following the similar idea of choosing parameter lambda, we try a bunch of numbers and find the best.

(b) Threshold as a function. (Figure 2)

Different users have distinct preference profiles, thus their threshold might be different naturally. Here we model the threshold as a function, to user activity.

Intuition behind it is that, more active a user is, more business it will prefer. So lower threshold should be modified on highly active users. For example, such function is defined:

$$threshold(u) = upbound - \left(count(u)^{\alpha} \frac{1}{scale}\right)$$

$count$ is the number of business a user visit before (in training set), representing users' activity. And it is a constant number for one user which can be computed easily.

$upbound$, $scale$, α is hyper parameter we need to tune.

The result shows that threshold function below improves the performance (accuracy) by about 0.3% compared to 91.8% under optimal fixed threshold.

$$threshold(u) = 0.56 - (\frac{count(u)^{0.9}}{171})$$

Results

We divide data into training set and validation set, with quantity of 99% and 1%.

On public leaderboard (seen portion of test data)

Accuracy	K=100	K=200	K=400	K=800
Fixed threshold	0.91665	0.91705	0.91820	0.91825
Dynamic threshold	0.91790	0.91855	0.92050	0.92170

On private leaderboard (unseen portion of test data)

Finally, the best accuracy is obtained with 0.92095, under dynamic threshold case of $K = 800$, $\lambda = 0.6$. What's more, there is a little difference between seen and unseen data when K gets larger, while performance dramatically drops if K is small. I think the reason may be that smaller K leads to a not bad accuracy on training and validation set but it over fits the data.

Task 2 (rating prediction)

(Kaggle) user name: yfzhou13 display name: Vidal

Method

We apply latent-factor model to this rating prediction task. It is an idea of dimensional reduction.

$$R = UV^T$$

Based on the simple model in homework 3, which only includes α and β, the entire latent-factor model is describe as follow.

$$f(u,i) = \alpha + \beta_u + \beta_i + \gamma_u\gamma_i$$

In this model, α is a constant. β_u and β_i represents bias on different users and items respectively. $\gamma_u\gamma_i$ is a multiplication of 1 by K vector and K by 1 vector, showing preference of this user towards this item.

Thus optimization objective is square error plus regularization.

$$\min_{\alpha,\beta,\gamma} \sum_{u,i} \big(f(u,i) - R_{u,i}\big)^2 + \lambda[\sum_u \beta_u^2 + \sum_i \beta_i^2 + \sum_u \big||\gamma_u|\big|_2^2 + \sum_i \big||\gamma_i|\big|_2^2]$$

Implementation

Basically, we train these unknown vectors or matrices on training dataset, then tune parameter λ on validation set. Because the optimization objective is not convex, we can just solve it approximately. Here are particular steps for training.

- 1. Initialize α, β, γ randomly.
- 2. Find the solution for α, β, using derivatives updating until convergence.
- 3. Keep α, β unchanged, compute gradient of γ_u and γ_i.
- 4. Update γ_u and γ_i with the gradient computed.
- 5. Repeat step 3 and step 4.

In step 2, let derivative equal to 0.

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

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

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

In step 3, compute gradients as follows.

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

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

Results

We divide data into training set and validation set, with quantity of 99% and 1%.

On public leaderboard (seen portion of test data)

Only with α and β, we obtain accuracy of 0.74583 on test data under optimal $\lambda = 4.1$. By adding γ, our results become more accurate.

RMSE	Lambda=5	Lambda=10
K=1	0.74572	0.74546
K=2	0.74566	0.74534
K=5	0.74564	0.74548

On private leaderboard (unseen portion of test data)

Best performance is obtained at $\lambda = 10.0$, with RMSE of 0.75746. This is a considerable drop compared to another part of test data. The reason might be iteration times is not enough to train a model which avoid serious overfitting.

Appendix:

1. Visit prediction: one-class

```
def OneClass(lam, K, learnRate, max_iter):
    gammaU = np.random.rand(Lu, K)/1 - 0.5
    gammaB = np.random.rand(K, Lb)/1 - 0.5
    accRec = []
    for it in range(max_iter):
        objective = 0
        regularization = 0
        gu = np.zeros((Lu, K))
        gb = np.zeros((K, Lb))

        for (user, busi) in trainPair:
            u = userDict[user]
            i = busiDict[busi]
            j = busiDict[findNegBusi(user)]
            z = sigmoid(np.dot(gammaU[u,:],gammaB[:,i]) - \
                        np.dot(gammaU[u,:],gammaB[:,j]))
            gu[u,:] += (1-z)*(gammaB[:,i]-gammaB[:,j])
            gb[:,i] += (1-z)*(gammaU[u,:])
            gb[:,j] += (1-z)*(-gammaU[u,:])

            objective += log(z)

        gu -= lam*gammaU
        gb -= lam*gammaB
        gammaU += learnRate*gu
        gammaB += learnRate*gb

        regularization = objective - lam*np.sum(np.square(gammaU)) - \
                                           lam*np.sum(np.square(gammaB))
        if (it+1)%2==0:
            print 'iteration: ' + str(it+1) + '\t' + str(regularization) \
                                      + '\t' + str(objective)
        if (it+1)%10==0:
            accCur = max([Accuracy(gammaU,gammaB,t) for t in \
                          [0.3,0.4,0.5,0.6,0.7]])
            accRec.append(accCur)
            print accCur

    return gammaU, gammaB, accRec
```

2. Rating prediction: latent-factor model

```python
# latent factor model (alpha, beta)
def LFM1(lam, max_iter):
    alpha = 0.1
    betaU = {i:0.0 for i in uniUser}
    betaB = {i:0.0 for i in uniBusi}
    for it in range(max_iter):
        alpha = 0
        loss = 0
        for (user, busi, rating) in trainPair:
            alpha += (rating - betaU[user] - betaB[busi])/len(trainPair)
        for (user, busi, rating) in trainPair:
            diff = alpha + betaU[user] + betaB[busi] - rating
            loss += diff**2
        squareError = loss
        for (user, busi_list) in userBusi_train.items():
            betaU[user] = 0
            for (busi,rating) in busi_list:
                betaU[user] += (rating - alpha - betaB[busi])/(lam + \
                     len(busi_list))
            loss += lam*(betaU[user]**2)
        for (busi, user_list) in busiUser_train.items():
            betaB[busi] = 0
            for (user,rating) in user_list:
                betaB[busi] += (rating - alpha - betaU[user])/(lam + \
                     len(user_list))
            loss += lam*(betaB[busi]**2)
        if (it+1)%5==0:
            print 'iteration: ' + str(it+1)
            print loss, squareError
            meanSquaredError(validPair,[alpha,betaU,betaB])
    return alpha, betaU, betaB
```

```python
# latent factor model (alpha, beta, gamma)
def LFM3(lam, K, learnRate, max_iter):
    alpha, betaU, betaB = LFM1(lam = 4.0, max_iter = 30)
    errList = []
    gammaU = {i:np.random.rand(K)/10-0.05 for i in uniUser}
    gammaB = {i:np.random.rand(K)/10-0.05 for i in uniBusi}
    for it in range(max_iter):
        loss = 0

        gra_gammaU = {user:lam*gammaU[user] for user in uniUser}
        gra_gammaB = {busi:lam*gammaB[busi] for busi in uniBusi}

        for (user, busi, rating) in trainPair:
            diff = alpha + betaU[user] + betaB[busi] + \
                                 np.inner(gammaU[user],gammaB[busi]) - rating
            gra_gammaU[user] += gammaB[busi]*diff
            gra_gammaB[busi] += gammaU[user]*diff
            loss += diff**2
        squareError = loss
        for (user, busi_list) in userBusi_train.items():
            gammaU[user] -= learnRate*gra_gammaU[user]
            loss += lam*(betaU[user]**2+sum(gammaU[user]**2))
        for (busi, user_list) in busiUser_train.items():
            gammaB[busi] -= learnRate*gra_gammaB[busi]
            loss += lam*(betaB[busi]**2+sum(gammaB[busi]**2))
        if (it+1)%10==0:
            print 'iteration: ' + str(it+1)
            print loss, squareError
            err = meanSquaredError(validPair,[alpha,betaU,betaB,gammaU,gammaB])
            errList.append(err)
    return alpha,betaU,betaB,gammaU,gammaB,errList
```