评分预测问题

离线实验方法

在给定用户评分数据集后，将数据集按照一定的方式分成训练集和测试集，然后根据训练集建立用户兴趣模型来预测测试集中的用户评分。对于测试集中的一对用户和物品(u,i)，用户 u 对物品 i 的真实评分是r_{ui}，而推荐算法预测的用户 u 对物品 i 的评分为$\hat{r}_{ui}$，那么一般可以用均方根误差 RMSE 度量预测的精度：

$$\mathrm{RMSE}=\frac{\sqrt{\sum_{(u,i)}(r_{ui}-\hat{r}_{ui})^2}}{|Test|}$$

如果是和时间无关的预测任务，可以以均匀分布随机划分数据集，即对每个用户，随机选择一些评分记录作为测试集，剩下的记录作为测试集。如果是和时间相关的任务，那么需要将用户的旧行为作为训练集，将用户的新行为作为测试集。

评分预测算法

平均值

全局平均值

$$\hat{r}_{ui}=\mu=\frac{\sum_{(u,i)\in Train} r_{ui}}{\sum_{(u,i)\in Train} 1}$$

用户评分平均值

$$\hat{r}_{ui}=\bar{r}_u=\frac{\sum_{i\in N(u)} r_{ui}}{\sum_{i\in N(u)} 1}$$

物品评分平均值

$$\hat{r}_{ui}=\bar{r}_i=\frac{\sum_{i\in N(u)} r_{ui}}{\sum_{i\in N(u)} 1}$$

用户分类对物品分类的平均值

假设有两个分类函数，一个是用户分类函数$\emptyset$，一个是物品分类函数φ。$\emptyset(u)$定义了用户 u 所属的类，$\varphi(i)$定义了物品 i 所属的类。那么，我们可以利用训练集中同类用户对同类物品评分的平均值预测用户对物品的评分，即：

$$\hat{r}_{ui}=\frac{\sum_{(v,j)\in Train,\emptyset(u)=\emptyset(v),\varphi(i)=\varphi(j)} r_{vi}}{\sum_{(v,j)\in Train,\emptyset(u)=\emptyset(v),\varphi(i)=\varphi(j)} 1}$$

- **用户和物品的平均分** 对于一个用户，可以计算他的评分平均分。然后将所有用户按照评分平均分从小到大排序，并将用户按照平均分平均分成 N 类。物品也可以用同样的方式分类。
- **用户活跃度和物品流行度** 对于一个用户，将他评分的物品数量定义为他的活跃度。得到用户活跃度之后，可以将用户通过活跃度从小到大排序，然后平均分为 N 类。物品的流行度定义为给物品评分的用户数目，物品也可以按照流行度均匀分成 N 类。

```
def PredictAll(records, user_cluster, item_cluster):
   total = dict()
   count = dict()
   for r in records:
      if r.test != 0:
         continue
      gu = user_cluster.GetGroup(r.user)
      gi = item_cluster.GetGroup(r.item)
      basic.AddToMat(total, gu, gi, r.vote)
      basic.AddToMat(count, gu, gi, 1)
   for r in records:
      gu = user_cluster.GetGroup(r.user)
      gi = item_cluster.GetGroup(r.item)
   average = total[gu][gi] / (1.0 * count[gu][gi] + 1.0)
   r.predict = average

class Cluster:
   def __init__(self,records):
      self.group = dict()
   def GetGroup(self, i):
      return 0
class IdCluster(Cluster):
   def __init__(self, records):
      Cluster.__init__(self, records)
   def GetGroup(self, i):
      return i
class UserActivityCluster(Cluster):
   def __init__(self, records):
      Cluster.__init__(self, records)
      activity = dict()
      for r in records:
         if r.test != 0:
            continue
         basic.AddToDict(activity, r.user, 1)
      k = 0
      for user, n in sorted(activity.items(),
                     key=itemgetter(1), reverse=False):
         c = int((k * 5) / (1.0 * len(activity)))
         self.group[user] = c
         k += 1
   def GetGroup(self, uid):
      if uid not in self.group:
         return -1
      else:
```

```
         return self.group[uid]
class ItemPopularityCluster(Cluster):
   def __init__(self, records):
      Cluster.__init__(self, records)
      popularity = dict()
      for r in records:
         if r.test != 0:
            continue
         basic.AddToDict(popularity, r.item, 1)
      k = 0
      for item, n in sorted(popularity.items(),
                     key=itemgetter(1), reverse=False):
         c = int((k * 5) / (1.0 * len(popularity)))
         self.group[item] = c
         k += 1
   def GetGroup(self, item):
      if item not in self.group:
         return -1
      else:
         return self.group[item]
class UserVoteCluster(Cluster):
   def __init__(self, records):
      Cluster.__init__(self, records)
      vote = dict()
      count = dict()
      for r in records:
         if r.test != 0:
            continue
         basic.AddToDict(vote, r.user, r.vote)
         basic.AddToDict(count, r.user, 1)
      k = 0
      for user, v in vote.items():
         ave = v / (count[user] * 1.0)
         c = int(ave * 2)
         self.group[user] = c
   def GetGroup(self, uid):
      if uid not in self.group:
         return -1
      else:
         return self.group[uid]
class ItemVoteCluster(Cluster):
   def __init__(self, records):
      Cluster.__init__(self, records)
      vote = dict()
```

```
        count = dict()
        for r in records:
            if r.test != 0:
                continue
            basic.AddToDict(vote, r.item, r.vote)
            basic.AddToDict(count, r.item, 1)
        k = 0
        for item, v in vote.items():
            ave = v / (count[item] * 1.0)
            c = int(ave * 2)
            self.group[item] = c
    def GetGroup(self, item):
        if item not in self.group:
            return -1
        else:
            return self.group[item]
```

基于邻域的方法

基于用户的邻域算法和基于物品的邻域算法都可以应用到评分预测中。基于用户的邻域算法认为预测一个用户对一个物品的评分，需要参考和这个用户兴趣相似的用户对该物品的评分，即:

$$\hat{r}_{ui} = \bar{r}_u + \frac{\sum_{v \in S(u,K) \cap N(i)} w_{uv}(r_{vi} - \bar{r}_v)}{\sum_{v \in S(u,K) \cap N(i)} |w_{uv}|}$$

$$w_{uv} = \frac{\sum_{i \in I}(r_{ui} - \bar{r}_u) * (r_{ui} - \bar{r}_u)}{\sqrt{\sum_{i \in I}(r_{ui} - \bar{r}_u)^2 \sum_{i \in I}(r_{vi} - \bar{r}_v)^2}}$$

```
def UserSimilarity(records):
    item_users = dict()
    ave_vote = dict()
    activity = dict()
    for r in records:
        addToMat(item_users, r.item, r.user, r.value)
        addToVec(ave_vote, r.user, r.value)
        addToVec(activity, r.user, 1)
    ave_vote = {x:y/activity[x] for x,y in ave_vote.items()}
    nu = dict()
    W = dict()
    for i,ri in item_users.items():
        for u,rui in ri.items():
            addToVec(nu,u,(rui-ave_vote[u])*(rui- ve_vote[u]))
            for v,rvi in ri.items():
                if u == v:
                    continue
```

```
            addToMat(W, u, v, (rui - ave_vote[u])*(rvi - ave_vote[v]))
    for u in W:
        W[u] = {x:y/math.sqrt(nu[x]*nu[u])
                    for x,y in W[u].items()}
    return W
def PredictAll(records, test, ave_vote, W, K):
    user_items = dict()
    for r in records:
        addToMat(user_items, r.user, r.item, r.value)
    for r in test:
        r.predict = 0
        norm = 0
        for v,wuv in sorted(W[r.user].items(),
                    key=itemgetter(1), reverse=True)[0:K]:
            if r.item in user_items[v]:
                rvi = user_items[v][r.item]
                r.predict += wuv * (rvi - ave_vote[v])
                norm += abs(wuv)
        if norm > 0:
            r.predict /= norm
        r.predict += ave_vote[r.user]
```

隐语义模型与矩阵分解模型

传统的 SVD 分解

如果补全后矩阵的特征值和补全之前矩阵的特征值相差不大，就算是扰动比较小。

给定 m 个用户和 n 个物品，和用户对物品的评分矩阵 $R_{m\times n}$。首先需要对评分矩阵中的缺失值进行简单地补全，比如用全局平均值，或者用户/物品平均值补全，得到补全后的矩阵R'。接着，可以用 SVD 分解将R'分解成如下形式：

$$R' = U^T S V$$

为了对R'进行降维，可以取最大的 f 个奇异值组成对角矩阵S_f，并且找到这 f 个奇异值中每个值在 U、V 矩阵中对应的行和列，得到U_f、V_f，从而可以得到一个降维后的评分矩阵：

$$R_f{}' = U_f{}^T S_f V_f$$

- 需要用一个简单的方法补全稀疏评分矩阵。一般来说，推荐系统中的评分矩阵是非常稀疏的，一般都有 95%以上的元素是缺失的。而一旦补全，评分矩阵就会变成一个稠密矩阵，从而使评分矩阵的存储需要非常大的空间，这种空间的需求在实际系统中是不可能接受的。
- 依赖的 SVD 分解方法的计算复杂度很高，特别是在稠密的大规模矩阵上更是非常慢。

Simon Funk 的 SVD 分解

从矩阵分解的角度说，如果我们将评分矩阵 R 分解为两个低维矩阵相乘：

$$\hat{R} = P^T Q$$

对于用户 u 对物品 i 的评分的预测值$\hat{R}(\mathrm{u},\mathrm{i}) = \hat{r}_{ui}$，可以通过如下公式计算：

$$\hat{r}_{ui} = \sum_f p_{uf} q_{if}$$

可以直接通过训练集中的观察值利用最小化 RMSE 学习 P、Q 矩阵。既然我们用 RMSE 作为评测指标，那么如果能找到合适的 P、Q 来最小化训练集的预测误差，那么应该也能最小化测试集的预测误差。

$$C(p,q) = \sum_{(u,i)\in Train} (r_{ui} - \hat{r}_{ui})^2 = \sum_{(u,i)\in Train} (r_{ui} - \sum_{f=1}^{F} p_{uf} q_{if})^2$$

直接优化上面的损失函数可能会导致学习的过拟合，因此还需要加入防止过拟合项：

$$C(p,q) = \sum_{(u,i)\in Train} (r_{ui} - \sum_{f=1}^{F} p_{uf} q_{if})^2 + \lambda(\|p_u\|^2 + \|q_i\|^2)$$

$$\frac{\partial C}{\partial p_{uf}} = -2q_{ik} + 2\lambda p_{uk}$$

$$\frac{\partial C}{\partial q_{if}} = -2p_{uk} + 2\lambda q_{ik}$$

```
def LearningLFM(train, F, n, alpha, lambda):
   [p,q] = InitLFM(train, F)
   for step in range(0, n):
      for u,i,rui in train.items():
         pui = Predict(u, i, p, q)
         eui = rui - pui
         for f in range(0,F):
            p[u][k] += alpha * (q[i][k] * eui - lambda * p[u][k])
            q[i][k] += alpha * (p[u][k] * eui - lambda * q[i][k])
      alpha *= 0.9
   return list(p, q)
def InitLFM(train, F):
   p = dict()
   q = dict()
   for u, i, rui in train.items():
      if u not in p:
         p[u] = [random.random()/math.sqrt(F) for x in range(0,F)]
      if i not in q:
         q[i] = [random.random()/math.sqrt(F) for x in range(0,F)]
   return list(p, q)
def Predict(u, i, p, q):
   return sum(p[u][f] * q[i][f] for f in range(0,len(p[u]))
```

加入偏置项后的 LFM

一个评分系统有些固有属性和用户物品无关，而用户也有些属性和物品无关，物品也有些属性和用户无关。

$$r_{ui} = \mu + b_u + b_i + p_u^T * q_i$$

- $\boldsymbol{\mu}$ 训练集中所有记录的评分的全局平均数。而全局平均数可以表示网站本身对用户评分的影响。

- b_u 用户偏置（user bias）项。这一项表示了用户的评分习惯中和物品没有关系的那种因素。
- b_i 物品偏置（item bias）项。这一项表示了物品接受的评分中和用户没有什么关系的因素。

```
def LearningBiasLFM(train, F, n, alpha, lambda, mu):
   [bu, bi, p,q] = InitLFM(train, F)
   for step in range(0, n):
      for u,i,rui in train.items():
         pui = Predict(u, i, p, q, bu, bi, mu)
         eui = rui - pui
         bu[u] += alpha * (eui - lambda * bu[u])
         bi[i] += alpha * (eui - lambda * bi[i])
         for f in range(0,F):
            p[u][k] += alpha * (q[i][k] * eui - lambda * p[u][k])
            q[i][k] += alpha * (p[u][k] * eui - lambda * q[i][k])
      alpha *= 0.9
   return list(bu, bi, p, q)
def InitBiasLFM(train, F):
   p = dict()
   q = dict()
   bu = dict()
   bi = dict()
   for u, i, rui in train.items():
      bu[u] = 0
      bi[i] = 0
      if u not in p:
         p[u] = [random.random()/math.sqrt(F) for x in range(0,F)]
      if i not in q:
         q[i] = [random.random()/math.sqrt(F) for x in range(0,F)]
   return list(p, q)
def Predict(u, i, p, q, bu, bi, mu):
   ret = mu + bu[u] + bi[i]
   ret += sum(p[u][f] * q[i][f] for f in range(0,len(p[u]))
   return ret
```

考虑邻域影响的 LFM

SVD++：将用户历史评分的物品加入到了 LFM 模型中。首先讨论一下如何将基于邻域的方法也像 LFM 那样设计成一个可以学习的模型。

$$\hat{r}_{ui} = \frac{1}{\sqrt{|N(u)|}} \sum_{j \in N(u)} w_{ij}$$

$$\mathrm{C(w)} = \sum_{(u,i) \in Train} (r_{ui} - \sum_{j \in N(u)} w_{ij}\, r_{uj})^2 + \lambda w_{ij}^2$$

对 w 矩阵也进行分解，将参数个数降低到 2*n*F 个:

$$\hat{r}_{ui} = \frac{1}{\sqrt{|N(u)|}} \sum_{j \in N(u)} x_i^T y_j = \frac{1}{\sqrt{|N(u)|}} x_i^T \sum_{j \in N(u)} y_j$$

$$\hat{r}_{ui} = \mu + b_u + b_i + p_u^T * q_i + \frac{1}{\sqrt{|N(u)|}} x_i^T \sum_{j \in N(u)} y_j$$

为了不增加太多参数造成过拟合，可以令 x=q，从而得到最终的 SVD++模型：

$$\hat{r}_{ui} = \mu + b_u + b_i + q_i^T * (p_u + \frac{1}{\sqrt{|N(u)|}} x_i^T \sum_{j \in N(u)} y_j)$$

```
def LearningBiasLFM(train_ui, F, n, alpha, lambda, mu):
   [bu, bi, p, q, y] = InitLFM(train, F)
   z = dict()
   for step in range(0, n):
      for u,items in train_ui.items():
         z[u] = p[u]
         ru = 1 / math.sqrt(1.0 * len(items))
         for i,rui in items items():
            for f in range(0,F):
            z[u][f] += y[i][f] * ru
         sum = [0 for i in range(0,F)]
         for i,rui in items items():
            pui = Predict()
            eui = rui - pui
            bu[u] += alpha * (eui - lambda * bu[u])
            bi[i] += alpha * (eui - lambda * bi[i])
            for f in range(0,F):
               sum[k] += q[i][k] * eui * ru
               p[u][k] += alpha * (q[i][k] * eui - lambda * p[u][k])
               q[i][k] += alpha * ((z[u][k] + p[u][k]) * eui
                              - lambda * q[i][k])
         for i,rui in items items():
            for f in range(0,F):
               y[i][f] += alpha * (sum[f] - lambda * y[i][f])
      alpha *= 0.9
   return list(bu, bi, p, q)
```

加入时间信息

一种是将时间信息应用到基于邻域的模型中，另一种是将时间信息应用到矩阵分解模型中。

基于邻域的模型融合时间信息

$$\hat{r}_{uit} = \frac{\sum_{j \in N(u) \cap S(i,K)} f(w_{ij}, \Delta t) r_{uj}}{\sum_{j \in N(u) \cap S(i,K)} f(w_{ij}, \Delta t)}$$

$$\sigma(x) = \frac{1}{1+\exp(-x)}$$

$$f(w_{ij}, \Delta t) = \sigma(\delta * w_{ij} * \exp\left(\frac{-|\Delta t|}{\beta}\right) + \gamma)$$

随着Δt 增加，$f(w_{ij}, \Delta t)$会越来越小，也就是说用户很久之前的行为对预测用户当前评分的影响越来越小。

基于矩阵分解的模型融合时间信息

可以将用户—物品—时间三维矩阵如下分解：

$$\hat{r}_{uit} = \mu + b_u + b_i + b_t + p_u^T * q_i + x_u^T * y_t + s_u^T * z_t + \sum_f g_{uf} h_{if} l_{tf}$$

模型融合

模型级联融合

假设已经有一个预测器$\hat{r}^{(k)}$，对于每个用户—物品对(u, i)都给出预测值，那么可以在这个预测器的基础上设计下一个预测器$\hat{r}^{(k+1)}$来最小化损失函数：

$$C = \sum_{(u,i)\in Train} (r_{ui} - \hat{r}_{ui}^{(k)} - \hat{r}_{ui}^{(k+1)})^2$$

和 Adaboost 算法类似，该方法每次产生一个新模型，按照一定的参数加到旧模型上去，从而使训练集误差最小化。不同的是，这里每次生成新模型时并不对样本集采样，针对那些预测错的样本，而是每次都还是利用全样本集进行预测，但每次使用的模型都有区别。

```
def Predict(train, test, alpha):
   total = dict()
   count = dict()
   for record in train:
      gu = GetUserGroup(record.user)
      gi = GetItemGroup(record.item)
      AddToMat(total, gu, gi, record.vote - record.predict)
      AddToMat(count, gu, gi, 1)
   for record in test:
      gu = GetUserGroup(record.user)
      gi = GetUserGroup(record.item)
      average = total[gu][gi] / (1.0 * count[gu][gi] + alpha)
   record.predict += average
```

模型加权融合

假设我们有 K 个不同的预测器$\{\hat{r}^{(1)}, \hat{r}^{(2)}, \cdots, \hat{r}^{(K)}\}$，最简单的融合算法就是线性融合，即最终的预测器$\hat{r}$是这 K 个预测器的线性加权：

$$\hat{r} = \sum_{k=1}^{K} \alpha^k \hat{r}^{(k)}$$

评分预测问题的解决需要在训练集上训练 K 个不同的预测器，然后在测试集上作出预测。但是，如果我们继续在训练集上融合 K 个预测器，得到线性加权系数，就会造成过拟合的问题。

- 假设数据集已经被分为了训练集 A 和测试集 B，那么首先需要将训练集 A 按照相同的分割方法分为 A1 和 A2，其中 A2 的生成方法和 B 的生成方法一致，且大小相似。
- 在 A1 上训练 K 个不同的预测器，在 A2 上作出预测。因为我们知道 A2 上的真实评分值，所以可以在 A2 上利用最小二乘法计算出线性融合系数α^k。
- 在 A 上训练 K 个不同的预测器，在 B 上作出预测，并且将这 K 个预测器在 B 上的预测结果按照已经得到的线性融合系数加权融合，以得到最终的预测结果。