利用上下文信息

如何将时间信息和地点信息建模到推荐算法中，从而让推荐系统能够准确预测用户在某个特定时刻及特定地点的兴趣。仍然研究 TopN 推荐，即如何给用户生成一个长度为 N 的推荐列表，而该列表包含了用户在某一时刻或者某个地方最可能喜欢的物品。

时间上下文信息

首先介绍各种不同的时间效应，然后研究如何将这些时间效应建模到推荐系统的模型中，最后通过实际数据集对比不同模型的效果。

时间效应简介

时间是一种重要的上下文信息，对用户兴趣有着深入而广泛的影响。

- **用户兴趣是变化的**　我们这里提到的用户兴趣变化是因为用户自身原因发生的变化。如果我们要准确预测用户现在的兴趣，就应该关注用户最近的行为，因为用户最近的行为最能体现他现在的兴趣。当然，考虑用户最近的兴趣只能针对渐变的用户兴趣，而对突变的用户兴趣很难起作用。
- **物品也是有生命周期的**　当我们决定在某个时刻给某个用户推荐某个物品时，需要考虑该物品在该时刻是否已经过时了，不同系统的物品具有不同的生命周期。
- **季节效应**　季节效应主要反映了时间本身对用户兴趣的影响。

时间效应举例

系统时间特性的分析

在给定时间信息后，推荐系统从一个静态系统变成了一个时变的系统，而用户行为数据也变成了时间序列。研究一个时变系统，需要首先研究这个系统的时间特性。通过研究时变的用户行为数据集来研究不同类型网站的时间特性。包含时间信息的用户行为数据集由一系列三元组构成，其中每个三元组(u,i,t)代表了用户 u 在时刻 t 对物品 i 产生过行为。

- **数据集每天独立用户数的增长情况**
- **系统的物品变化情况**
- **用户访问情况**

数据集的选择

物品的生存周期和系统的时效性

- **物品平均在线天数**　如果一个物品在某天被至少一个用户产生过行为，就定义该物品在这一天在线。因此，我们可以通过物品的平均在线天数度量一类物品的生存周期。
- **相隔 T 天系统物品流行度向量的平均相似度**　取系统中相邻 T 天的两天，分别计算这两天的物品流行度，从而得到两个流行度向量。然后，计算这两个向量的余弦相似度，如果相似度大，说明系统的物品在相隔 T 天的时间内没有发生大的变化，从而说明系统的时效性不强，物品的平均在线时间较长。相反，如果相似度很小，说明系统中的物品在相隔 T 天的时间内发生了很大变化，从而说明系统的时效性很强，物品的平均在线时间很短。

推荐系统的实时性

用户兴趣是不断变化的，其变化体现在用户不断增加的新行为中。一个实时的推荐系统需要能够实时响应用户新的行为，让推荐列表不断变化，从而满足用户不断变化的兴趣。

- 实时推荐系统不能每天都给所有用户离线计算推荐结果，然后在线展示昨天计算出来的结果。所以，要求在每个用户访问推荐系统时，都根据用户这个时间点前的行为实时计算推荐列表。
- 推荐算法需要平衡考虑用户的近期行为和长期行为，即要让推荐列表反应出用户近期行为所体现的兴趣变化，又不能让推荐列表完全受用户近期行为的影响，要保证推荐列表对用户兴趣预测的延续性。

推荐算法的时间多样性

推荐系统每天推荐结果的变化程度被定义为推荐系统的时间多样性，时间多样性高的推荐系统中用户会经常看到不同的推荐结果。

- 需要保证推荐系统能够在用户有了新的行为后及时调整推荐结果，使推荐结果满足用户最近的兴趣；
- 需要保证推荐系统在用户没有新的行为时也能够经常变化一下结果，具有一定的时间多样性。

针对第一种，第一是从推荐系统的实时性角度分析。有些推荐系统会每天离线生成针对所有用户的推荐结果，然后在线直接将这些结果展示给用户。这种类型的系统显然无法做到在用户有了新行为后及时调整推荐结果。第二，即使是实时推荐系统，由于使用的算法不同，也具有不同的时间多样性。

针对第二种，在生成推荐结果时加入一定的随机性；记录用户每天看到的推荐结果，然后在每天给用户进行推荐时，对他前几天看到过很多次的推荐结果进行适当地降权；每天给用户使用不同的推荐算法。

推荐系统需要首先保证推荐的精度，在此基础上适当地考虑时间多样性。

时间上下文推荐算法

最近最热门

在没有时间信息的数据集中，我们可以给用户推荐历史上最热门的物品。那么在获得用户行为的时间信息后，最简单的非个性化推荐算法就是给用户推荐最近最热门的物品了。给定时间 T，物品 i 最近的流行度$n_i(T)$可以定义为：

$$n_i(T) = \sum_{(u,i,t)\in Train, t<T} \frac{1}{1+\alpha(T-t)}$$

```
def RecentPopularity(records, alpha, T):
    ret = dict()
    for user,item,tm in records:
        if tm >= T:
            continue
        addToDict(ret, item, 1 / (1.0 + alpha * (T - tm)))
    return ret
```

时间上下文相关的 ItemCF 算法

基于物品（item-based）的个性化推荐算法是商用推荐系统中应用最广泛的，首先利用用户行为离线计算物品之间的相似度；其次根据用户的历史行为和物品相似度矩阵，给用户做在线个性化推荐。

- **物品相似度** 用户在相隔很短的时间内喜欢的物品具有更高相似度。
- **在线推荐** 用户近期行为相比用户很久之前的行为，更能体现用户现在的兴趣。因此在预

测用户现在的兴趣时，应该加重用户近期行为的权重，优先给用户推荐那些和他近期喜欢的物品相似的物品。

$$\text{sim(i,j)} = \frac{\sum_{u \in N(i) \cap N(j)} 1}{\sqrt{|N(i)||N(j)|}}, p(u,i) = \sum_{j \in N(u)} sim(i,j)$$

在得到时间信息（用户对物品产生行为的时间）后， 我们可以通过如下公式改进相似度计算：

$$\text{sim(i,j)} = \frac{\sum_{u \in N(i) \cap N(j)} f(|t_{ui} - t_{uj}|)}{\sqrt{|N(i)||N(j)|}}$$

在分子中引入了和时间有关的衰减项$f(|t_{ui} - t_{uj}|)$，其中t_{ui}是用户 u 对物品 i 产生行为的时间。f 函数的含义是，用户对物品 i 和物品 j 产生行为的时间越远，则$f(|t_{ui} - t_{uj}|)$越小。

```
def ItemSimilarity(train, alpha):
   #calculate co-rated users between items
   C = dict()
   N = dict()
   for u, items in train.items():
      for i,tui in items.items():
         N[i] += 1
         for j,tuj in items.items():
            if i == j:
               continue
            C[i][j] += 1 / (1 + alpha * abs(tui - tuj))
   #calculate finial similarity matrix W
   W = dict()
   for i,related_items in C.items():
      for j, cij in related_items.items():
         W[u][v] = cij / math.sqrt(N[i] * N[j])
   return W
```

除了考虑时间信息对相关表的影响，我们也应该考虑时间信息对预测公式的影响。一般来说，用户现在的行为应该和用户最近的行为关系更大。

$$p(u,i) = \sum_{j \in N(u) \cap S(i,K)} sim(i,j) \frac{1}{1 + \beta |t_0 - t_{uj}|}$$

```
def Recommendation(train, user_id, W, K, t0):
   rank = dict()
   ru = train[user_id]
   for i,pi in ru.items():
      for j, wj in sorted(W[i].items(), key=itemgetter(1),
                          reverse=True)[0:K]:
         if j,tuj in ru.items():
            continue
```

```
            rank[j] += pi * wj / (1 + alpha * (t0 - tuj))
    return rank
```

时间上下文相关的 UserCF 算法

- **用户兴趣相似度** 两个用户兴趣相似是因为他们喜欢相同的物品，或者对相同的物品产生过行为。但是，如果两个用户同时喜欢相同的物品，那么这两个用户应该有更大的兴趣相似度。
- **相似兴趣用户的最近行为** 在找到和当前用户 u 兴趣相似的一组用户后，这组用户最近的兴趣显然相比这组用户很久之前的兴趣更加接近用户 u 今天的兴趣。也就是说，我们应该给用户推荐和他兴趣相似的用户最近喜欢的物品。

$$w_{uv} = \frac{\sum_{i \in N(u) \cap N(v)} \frac{1}{1 + \alpha |t_{ui} - t_{vi}|}}{\sqrt{|N(i)||N(j)|}}$$

```
def UserSimilarity(train):
    # build inverse table for item_users
    item_users = dict()
    for u, items in train.items():
        for i,tui in items.items():
            if i not in item_users:
                item_users[i] = dict()
            item_users[i][u] = tui
    #calculate co-rated items between users
    C = dict()
    N = dict()
    for i, users in item_users.items():
        for u,tui in users.items():
            N[u] += 1
            for v,tvi in users.items():
                if u == v:
                    continue
                C[u][v] += 1 / (1 + alpha * abs(tui - tvi))
    #calculate finial similarity matrix W
    W = dict()
    for u, related_users in C.items():
        for v, cuv in related_users.items():
            W[u][v] = cuv / math.sqrt(N[u] * N[v])
    return W
```

在得到用户相似度后，UserCF 通过如下公式预测用户对物品的兴趣：

$$p(u,i) = \sum_{v \in S(u,K)} w_{uv} r_{vi}$$

如果考虑和用户 u 兴趣相似用户的最近兴趣，

$$p(u,i) = \sum_{v \in S(u,K)} w_{uv} r_{vi} \frac{1}{\alpha(t_0 - t_{vi})}$$

```
def Recommend(user, T, train, W):
    rank = dict()
    interacted_items = train[user]
    for v, wuv in sorted(W[u].items, key=itemgetter(1),
              reverse=True)[0:K]:
        for i, tvi in train[v].items:
            if i in interacted_items:
                #we should filter items user interacted before
                Continue
            rank[i] += wuv / (1 + alpha * (T - tvi))
    return rank
```

时间段图模型

时间段图模型$G(U,S_U,I,S_I,E,w,\sigma)$是一个二分图。U 是用户节点集合，$S_U$是用户时间段节点集合。一个用户时间段节点$v_{ut} \in S_U$会和用户 u 在时刻 t 喜欢的物品通过边相连。I 是物品节点集合，S_I是物品时间段节点集合。一个物品时间段节点$v_{it} \in S_I$会和所有在时刻 t 喜欢物品 i 的用户通过边相连。E 是边集合，它包含了 3 种边：(1)如果用户 u 对物品 i 有行为，那么存在边 $e(v_u,v_i) \in E$；(2)如果用户 u 在 t 时刻对物品 i 有行为，那么就存在两条边$e(v_{ut},v_i),e(v_u,v_{it}) \in E$。w(e)定义了边的权重，$\sigma(e)$定义了顶点的权重。

路径融合算法的方法，通过该算法来度量图上两个顶点的相关性：

- 两个顶点之间有很多路径相连；
- 两个顶点之间的路径比较短；
- 两个顶点之间的路径不经过出度比较大的顶点。

首先提取出两个顶点之间长度小于一个阈值的所有路径，然后根据每条路径经过的顶点给每条路径赋予一定的权重，最后将两个顶点之间所有路径的权重之和作为两个顶点的相关度。

假设$\mathrm{P}=\{v_1,v_2,\cdots,v_n\}$是连接顶点$v_1$和$v_n$的一条路径，这条路径的权重$\Gamma(\mathrm{P})$取决于这条路径经过的所有顶点和边：

$$\Gamma(\mathrm{P})=\sigma(v_n)\prod_{i=1}^{n-1}\frac{\sigma(v_i)*w(v_i,v_{i+1})}{|out(v_i)|^{\rho}}$$

在定义了一条路径的权重后，就可以定义顶点之间的相关度。对于顶点 v 和v'，令$p(v,v',K)$为这两个顶点间距离小于 K 的所有路径，那么这两个顶点之间的相关度可以定义为：

$$d(v,v')=\sum_{p\in P(v,v',K)}\Gamma(\mathrm{P})$$

所有边的权重都定义为 1，而顶点的权重$\sigma(v)$定义如下：

$$\sigma(v)=\begin{cases}1-\alpha(v\in U)\\ \alpha(v\in S_U)\\ 1-\beta(v\in I)\\ \beta(v\in S_I)\end{cases}$$

```
def PathFusion(user, time,G,alpha)
    Q = []
    V = set()
```

```
depth = dict()
rank = dict()
depth['u:' + user] = 0
depth['ut:' + user + '_' + time] = 0
rank ['u:' + user] = alpha
rank ['ut:' + user + '_' + time] = 1 - alpha
Q.append('u:' + user)
Q.append('ut:' + user + '_' + time)
while len(Q) > 0:
   v = Q.pop()
   if v in V:
      continue
   if depth[v] > 3:
      continue
   for v2,w in G[v].items():
      if v2 not in V:
         depth[v2] = depth[v] + 1
         Q.append(v2)
      rank[v2] = rank[v] * w
return rank
```

离线实验

在得到由（用户、物品、时间）三元组组成的数据集后，对每一个用户，将物品按照该用户对物品的行为时间从早到晚排序，然后将用户最后一个产生行为的物品作为测试集，并将这之前的用户对物品的行为记录作为训练集。推荐算法将根据训练集学习用户兴趣模型，给每个用户推荐 N 个物品，并且利用准确率和召回率评测推荐算法的精度。

$$\text{Recall@N} = \frac{\sum_u |R(u,N) \cap T(u)|}{\sum_u |T(u)|}$$

$$\text{Precision@N} = \frac{\sum_u |R(u,N) \cap T(u)|}{\sum_u |R(u,N)|}$$

地点上下文信息

LARS（Location Aware Recommender System,位置感知推荐系统）的和用户地点相关的推荐系统。该系统首先将物品分成两类，一类是有空间属性的，另一类是无空间属性的物品。同时，它将用户也分成两类，一类是有空间属性的，另一类用户并没有相关的空间属性信息。

- （用户，用户位置，物品，评分），每一条记录代表了某一个地点的用户对物品的评分；
- （用户，物品，物品位置，评分），每一条记录代表了用户对某个地方的物品的评分；
- （用户，用户位置，物品，物品位置，评分），每一条记录代表了某个位置的用户对某个位置的物品的评分。

用户兴趣和地点相关的两种特征：

- **兴趣本地化** 不同地方的用户兴趣存在着很大的差别
- **活动本地化** 一个用户往往在附近的地区活动

对于第一种数据集，LARS 的基本思想是将数据集根据用户的位置划分成很多子集。因为

位置信息是一个树状结构。因此，数据集也会划分成一个树状结构。然后，给定每一个用户的位置，我们可以将他分配到某一个叶子节点中，而该叶子节点包含了所有和他同一个位置的用户的行为数据集。这样做的缺点是，每个叶子节点上的用户数量可能很少，因此他们的行为数据可能过于稀疏，从而无法训练出一个好的推荐算法。为此，我们可以从根节点出发，在到叶子节点的过程中，利用每个中间节点上的数据训练出一个推荐模型，然后给用户生成推荐列表。而最终的推荐结果是这一系列推荐列表的加权。

对于第二种数据集，每条用户行为表示为四元组（用户、物品、物品位置、评分），表示了用户对某个位置的物品给了某种评分。对于这种数据集，LARS 会首先忽略物品的位置信息，利用 ItemCF 算法计算用户 u 对物品 i 的兴趣 P(u,i)，但最终物品 i 在用户 u 的推荐列表中的权重定义为：

$$\mathrm{RecScore}(u,i) = P(u,i) - \mathrm{TravelPenalty}(u,i)$$

TravelPenalty(u,i)表示了物品 i 的位置对用户 u 的代价。计算 TravelPenalty(u,i)的基本思想是对于物品 i 与用户 u 之前评分的所有物品的位置计算距离的平均值（或者最小值）。关于如何度量地图上两点的距离，最简单的是基于欧式距离。当然，欧式距离有明显的缺点，因为人们是不可能沿着地图上的直线距离从一点走到另一点的。比较好的度量方式是利用交通网络数据，将人们实际需要走的最短距离作为距离度量。

为了避免计算用户对所有物品的 TravelPenalty，LARS 在计算用户 u 对物品 i 的兴趣度 RecScore(u,i)时，首先对用户每一个曾经评过分的物品（一般是餐馆、商店、景点），找到和他距离小于一个阈值 d 的所有其他物品，然后将这些物品的集合作为候选集，然后再利用上面的公式计算最终的 RecScore。

对于第三种数据集，相对于第二种数据集增加了用户当前位置这一信息。而在给定了这一信息后，我们应该保证推荐的物品应该距离用户当前位置比较近，在此基础上再通过用户的历史行为给用户推荐离他近且他会感兴趣的物品。