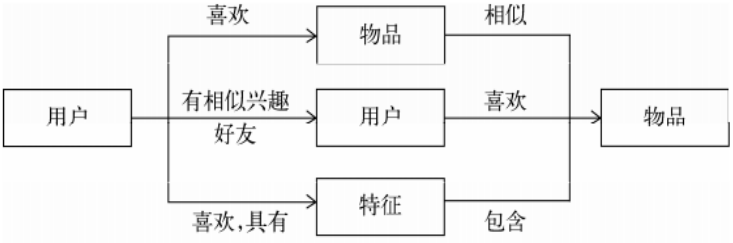
利用用户标签数据

目前流行的推荐系统基本上通过 3 种方式联系用户兴趣和物品。第一种方式是利用用户喜欢过的物品，给用户推荐与他喜欢过的物品相似的物品，这就是基于物品的算法。第二种方式是利用和用户兴趣相似的其他用户，给用户推荐那些和他们兴趣爱好相似的其他用户喜欢的物品，这是基于用户的算法。除了这两种方法，第三种重要的方式是通过一些特征（feature）联系用户和物品，给用户推荐那些具有用户喜欢的特征的物品。



标签是一种无层次化结构的、用来描述信息的关键词，它可以用来描述物品的语义。UGC（User Generated Content，用户生成的内容）的标签应用是一种表示用户兴趣和物品语义的重要方式。当一个用户对一个物品打上一个标签，这个标签一方面描述了用户的兴趣，另一方面则表示了物品的语义，从而将用户和物品联系了起来。

UGC 标签系统的代表应用

Delicious

CiteULike

Last.fm

豆瓣

Hulu

标签系统中的推荐问题

打标签作为一种重要的用户行为，蕴含了很多用户兴趣信息，因此深入研究和利用用户打标签的行为可以很好地指导我们改进个性化推荐系统的推荐质量。同时，标签的表示形式非常简单，便于很多算法处理。

用户为什么进行标注

在设计基于标签的个性化推荐系统之前，我们需要深入了解用户的标注行为（即打标签的行为），知道用户为什么要标注，用户怎么标注，只有深入了解用户的行为，我们才能基于这个行为设计出令他们满意的个性化推荐系统。

用户如何打标签

标签流行度的分布，一个标签被一个用户使用在一个物品上，它的流行度就加一。标签的

流行度分布也呈现非常典型的长尾分布，它的双对数曲线几乎是一条直线。

```
def TagPopularity(records):
    tagfreq = dict()
    for user,item,tag in records:
        if tag not in tagfreq:
            tagfreq[tag] = 1
        else:
            tagfreq[tag] += 1
    return tagfreq
```

用户打什么样的标签

基于标签的推荐系统

用户用标签来描述对物品的看法，因此标签是联系用户和物品的纽带，也是反应用户兴趣的重要数据源，如何利用用户的标签数据提高个性化推荐结果的质量是推荐系统研究的重要课题，用户的每一次打标签行为都用一个三元组（用户、物品、标签）=(u,i,b)表示。

实验设置

对于用户 u，令 R(u)为给用户 u 的长度为 N 的推荐列表，里面包含我们认为用户会打标签的物品。令 T(u)是测试集中用户 u 实际上打过标签的物品集合。然后，我们利用准确率（precision）和召回率（recall）评测个性化推荐算法的精度。

$$\text{Precision} = \frac{|R(u) \cap T(u)|}{|R(u)|}$$

$$\text{Recall} = \frac{|R(u) \cap T(u)|}{|T(u)|}$$

```
def CosineSim(item_tags, i, j):
    ret = 0
    for b,wib in item_tags[i].items():
        if b in item_tags[j]:
            ret += wib * item_tags[j][b]
        ni = 0
        nj = 0
        for b, w in item_tags[i].items():
            ni += w * w
        for b, w in item_tags[j].items():
            nj += w * w
        if ret == 0:
            return 0
    return ret / math.sqrt(ni * nj)
```

在得到物品之间的相似度度量后，我们通过如下公式计算一个推荐列表的多样性。

$$\text{Diversity} = 1 - \frac{\sum_{i \in R(u)} \sum_{j \in R(u), j \neq i} \text{Sim}(\text{item\_tags}[i], \text{item\_tags}[j])}{\binom{|R(u)|}{2}}$$

```
def Diversity(item_tags, recommend_items):
   ret = 0
   n = 0
   for i in recommend_items.keys():
      for j in recommend_items.keys():
         if i == j:
            continue
         ret += CosineSim(item_tags, i, j)
         n += 1
   return ret / (n * 1.0)
```

对于物品 i，定义它的流行度 item_pop(i)为给这个物品打过标签的用户数。而对推荐系统，我们定义它的平均热门度如下：

$$\text{AveragePopularity} = \frac{\sum_u \sum_{i \in R(u)} log(1 + item\_pop(i))}{\sum_u \sum_{i \in R(u)} 1}$$

一个最简单的算法

- 统计每个用户最常用的标签。
- 对于每个标签，统计被打过这个标签次数最多的物品。
- 对于一个用户，首先找到他常用的标签，然后找到具有这些标签的最热门物品推荐给这个用户。

用户 u 对物品 i 的兴趣度：$p(u,i) = \sum_b n_{u,b} n_{b,i}$

```
def InitStat(records):
   user_tags = dict()
   tag_items = dict()
   user_items = dict()
   for user, item, tag in records.items():
      addValueToMat(user_tags, user, tag, 1)
      addValueToMat(tag_items, tag, item, 1)
      addValueToMat(user_items, user, item, 1)
def Recommend(user):
   recommend_items = dict()
   tagged_items = user_items[user]
   for tag, wut in user_tags[user].items():
      for item, wti in tag_items[tag].items():
         #if items have been tagged, do not recommend them
         if item in tagged_items:
            continue
         if item not in recommend_items:
            recommend_items[item] = wut * wti
         else:
            recommend_items[item] += wut * wti
   return recommend_items
```

算法的改进

TF-IDF

倾向于给热门标签对应的热门物品很大的权重，因此会造成推荐热门的物品给用户，从而降低推荐结果的新颖性。另外，这个公式利用用户的标签向量对用户兴趣建模，其中每个标签都是用户使用过的标签，而标签的权重是用户使用该标签的次数。这种建模方法的缺点是给热门标签过大的权重，从而不能反应用户个性化的兴趣。

$$p(u,i)=\sum_b \frac{n_{u,b}}{\log(1+n_b^{(u)})} n_{b,i}$$

借鉴 TF-IDF 的思想对热门物品进行惩罚可以获得：

$$p(u,i)=\sum_b \frac{n_{u,b}}{\log(1+n_b^{(u)})} \frac{n_{b,i}}{\log(1+n_i^{(u)})}$$

适当惩罚热门标签和热门物品，在增进推荐结果个性化的同时并不会降低推荐结果的离线精度。

数据稀疏性

对于新用户或者新物品，用户兴趣和物品的联系$(B(u)\cap B(i))$中的标签数量会很少。为了提高推荐的准确率，我们可能要对标签集合做扩展，可以将这个标签的相似标签也加入到用户标签集合中。标签扩展法，其中常用的有话题模型（topic model），不过这里遵循简单的原则介绍一种基于邻域的方法。

如果认为同一个物品上的不同标签具有某种相似度， 那么当两个标签同时出现在很多物品的标签集合中时，我们就可以认为这两个标签具有较大的相似度。对于标签 b，令 N(b)为有标签 b 的物品的集合，n_{b,i}为给物品 i 打上标签 b 的用户数，我们可以通过如下余弦相似度公式计算标签 b 和标签 b’的相似度：

$$\mathrm{sim}(b,b')=\frac{\sum_{i\in N(b)\cap N(b')} n_{b,i}n_{b',i}}{\sqrt{\sum_{i\in N(b)} {n_{b,i}}^2 \sum_{i\in N(b')} {n_{b',i}}^2}}$$

进行标签扩展确实能够提高基于标签的物品推荐的准确率和召回率，但可能会稍微降低推荐结果的覆盖率和新颖度。

标签清理

将标签作为推荐解释。如果我们要把标签呈现给用户，将其作为给用户推荐某一个物品的解释，对标签的质量要求就很高。首先这些标签不能包含没有意义的停止词或者表示情绪的词，其次这些推荐解释里不能包含很多意义相同的词语。

- 去除词频很高的停止词；
- 去除因词根不同造成的同义词；
- 去除因分隔符造成的同义词。

为了控制标签的质量，采用让用户进行反馈的思想，即让用户告诉系统某个标签是否合适。

基于图的推荐算法

用户顶点、物品顶点和标签顶点：如果我们得到一个表示用户 u 给物品 i 打了标签 b 的用户标签行为(u,i,b)，那么最自然的想法就是在图中增加 3 条边，首先需要在用户 u 对应的顶点 v(u)和物品 i 对应的顶点 v(i)之间增加一条边（如果这两个顶点已经有边相连，那么就应该将边的权重加 1），同理，在 v(u)和 v(b)之间需要增加一条边，v(i)和 v(b)之间也需要边相

连接。

利用 PersonalRank 算法计算所有物品节点相对于当前用户节点在图上的相关性，然后按照相关性从大到小的排序，给用户推荐排名最高的 N 个物品。

用图模型解释前面的简单算法

给定用户标签行为记录(u,i,b)，SimpleTagGraph 会增加两条有向边，一条由用户节点 v(u)指向标签节点 v(b)，另一条由标签节点 v(b)指向物品节点 v(i)。从这个定义可以看到，SimpleTagGraph 相对于前面提到用户—物品—标签图少了用户节点和物品节点之间的边。

基于标签的推荐解释

基于标签的推荐其最大好处是可以利用标签做推荐解释，让用户自己根据兴趣选择相关的标签，得到推荐结果，从而极大地提高了推荐结果的多样性，使得推荐结果更容易满足用户多样的兴趣。

- RelSort 对推荐物品做解释时使用的是用户以前使用过且物品上有的标签，给出了用户对标签的兴趣和标签与物品的相关度，但标签按照和物品的相关度排序
- PrefSort 对推荐物品做解释时使用的是用户以前使用过且物品上有的标签，给出了用户对标签的兴趣和标签与物品的相关度，但标签按照用户的兴趣程度排序
- RelOnly 对推荐物品做解释时使用的是用户以前使用过且物品上有的标签，给出了标签与物品的相关度，且标签按照和物品的相关度排序
- PrefOnly 对推荐物品做解释时使用的是用户以前使用过且物品上有的标签，给出了用户对标签的兴趣程度，且标签按照用户的兴趣程度排序。

  客观事实类的标签优于主观感受类标签。
- 用户对标签的兴趣对帮助用户理解为什么给他推荐某个物品更有帮助；
- 用户对标签的兴趣和物品标签相关度对于帮助用户判定自己是否喜欢被推荐物品具有同样的作用；
- 物品标签相关度对于帮助用户判定被推荐物品是否符合他当前的兴趣更有帮助；
- 客观事实类标签相比主观感受类标签对用户更有作用。

给用户推荐标签

为什么要给用户推荐标签

- **方便用户输入标签**需要一个辅助工具来减小用户打标签的难度，从而提高用户打标签的参与度。
- **提高标签质量** 使用推荐标签时，可以对词表进行选择，首先保证词表不出现太多的同义词，同时保证出现的词都是一些比较热门的、有代表性的词。

如何给用户推荐标签

- 给用户推荐整个系统里最热门的标签（PopularTags）

```
def RecommendPopularTags(user,item, tags, N):
    return sorted(tags.items(), key=itemgetter(1), reverse=True)[0:N]
```

- 给用户推荐物品最热门的标签（ItemPopularTags）

```
def RecommendItemPopularTags(user,item, item_tags, N):
    return sorted(item_tags[item].items(),
                  key=itemgetter(1), reverse=True)[0:N]
```

- 给用户推荐他自己经常使用的标签（UserPopularTags）

```
def RecommendUserPopularTags(user,item, user_tags, N):
   return sorted(user_tags[user].items(),
              key=itemgetter(1), reverse=True)[0:N]
```

- 前面两种的融合（HybridPopularTags），该方法通过一个系数将上面的推荐结果线性加权，然后生成最终的推荐结果

```
def RecommendHybridPopularTags(user,item, user_tags,
                   item_tags, alpha, N):
   max_user_tag_weight = max(user_tags[user].values())
   for tag, weight in user_tags[user].items():
      ret[tag] = (1 - alpha) * weight / max_user_tag_weight
   max_item_tag_weight = max(item_tags[item].values())
   for tag, weight in item_tags[item].items():
      if tag not in ret:
         ret[tag] = alpha * weight / max_item_tag_weight
      else:
         ret[tag] += alpha * weight / max_item_tag_weight
   return sorted(ret[user].items(),
              key=itemgetter(1), reverse=True)[0:N]
```

将两个列表线性相加时都将两个列表按最大值做了归一化，这样的好处是便于控制两个列表对最终结果的影响，而不至于因为物品非常热门而淹没用户对推荐结果的影响，或者因为用户非常活跃而淹没物品对推荐结果的影响。

实验设置

```
def SplitData(records, train, test):
   for user,item, tag in records:
      if random.randint(1,10) == 1:
         test.append([user,item,tag])
      else:
         train.append([user,item,tag])
   return [train, test]
```

对于测试集中的每一个用户物品对(u,i)，我们都会推荐 N 个标签给用户 u 作参考。令 R(u,i)为我们给用户 u 推荐的应该在物品 i 上打的标签集合，令 T(u,i)为用户 u 实际给物品 i 打的标签的集合，我们可以利用准确率和召回率评测标签推荐的精度：

$$Precision = \frac{\sum_{(u,i)\in Test} |R(u,i) \cap T(u,i)|}{\sum_{(u,i)\in Test} |R(u,i)|}$$

$$Recall = \frac{\sum_{(u,i)\in Test} |R(u,i) \cap T(u,i)|}{\sum_{(u,i)\in Test} |T(u,i)|}$$

前面提到的基于统计用户常用标签和物品常用标签的算法有一个缺点，就是对新用户或者不热门的物品很难有推荐结果。

- 从物品的内容数据中抽取关键词作为标签；
- 针对有结果，但结果不太多的情况，可以做一些关键词扩展实现标签扩展的关键就是计算

标签之间的相似度。

基于图的标签推荐算法

重新定义顶点的启动概率

$$r_{v(k)} = \begin{cases} \alpha(v(k) = v(u)) \\ 1 - \alpha(v(k) = v(i)) \\ 0 \end{cases}$$

只有用户 u 和物品 i 对应的顶点有非 0 的启动概率，而其他顶点的启动概率都为 0。