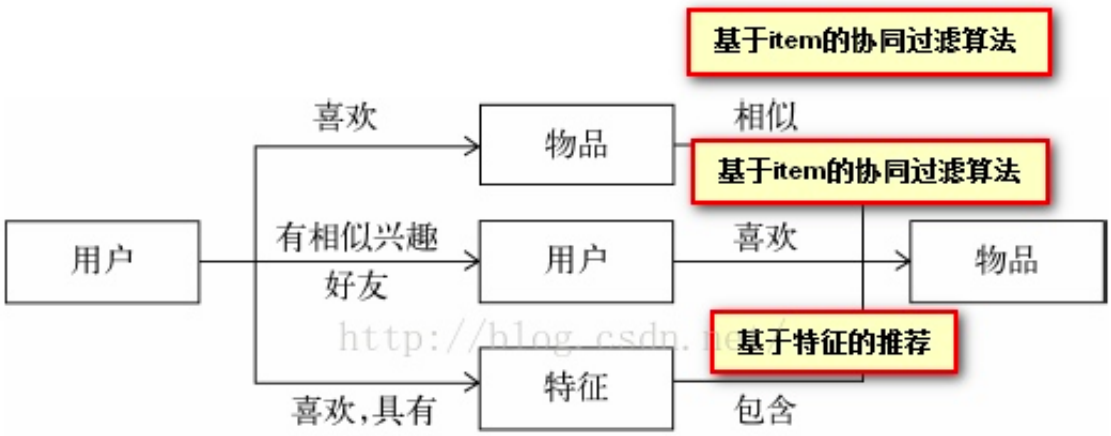
《推荐系统》基于标签的用户推荐系统

源代码查看地址：GitHub查看

一. 联系用户兴趣和物品的方式

推荐系统的目的是联系用户的兴趣和物品，这种联系方式需要依赖不同的媒介。目前流行的推荐系统基本上是通过三种方式联系用户兴趣和物品。



本图根据“Tagsplanations : Explaining Recommendations using Tags”一文中的插图重新绘制，本图的著作权归原著作权人所有

图4-1　推荐系统联系用户和物品的几种途径

1. 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，即基于item的系统过滤推荐算法（算法分析可参考：点击阅读）
2. 利用用户和兴趣用户兴趣相似的其他用户，给用户推荐哪些和他们兴趣爱好相似的其他用户喜欢的物品，即基于User的协同过滤推荐算法（算法分析可参考：点击阅读）
3. 通过一些特征联系用户和物品，给用户推荐那些具有用户喜欢的特征的物品，这里的特征有不同的表现形式，比如可以表现为物品的属性集合，也可以表现为隐语义向量，而下面我们要讨论的是一种重要的特征表现形式——标签

二. 标签系统的典型代表

Pass掉那些国外网站，比如说豆瓣图书，

网易云音乐

标签系统确实能够帮助用户发现他们喜欢和感兴趣的物品

三. 用户如何打标签

在互联网中每个人的行为都是随机的，但其实这些表面的行为隐藏着很多规律，那么我们对用户打的标签进行统计呢，便引入了标签流行度，我们定义的一个标签被一个用户使用在一个物品上，他的流行度就加**1**，可以如下代码实现：

```
#统计标签流行度
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
```

下面的是一个标签流行度分布图（横坐标是标签流行度K，纵坐标是流行度K对应的标签数目），也是符合典型的长尾分布，他的双对数曲线几乎是一条直线

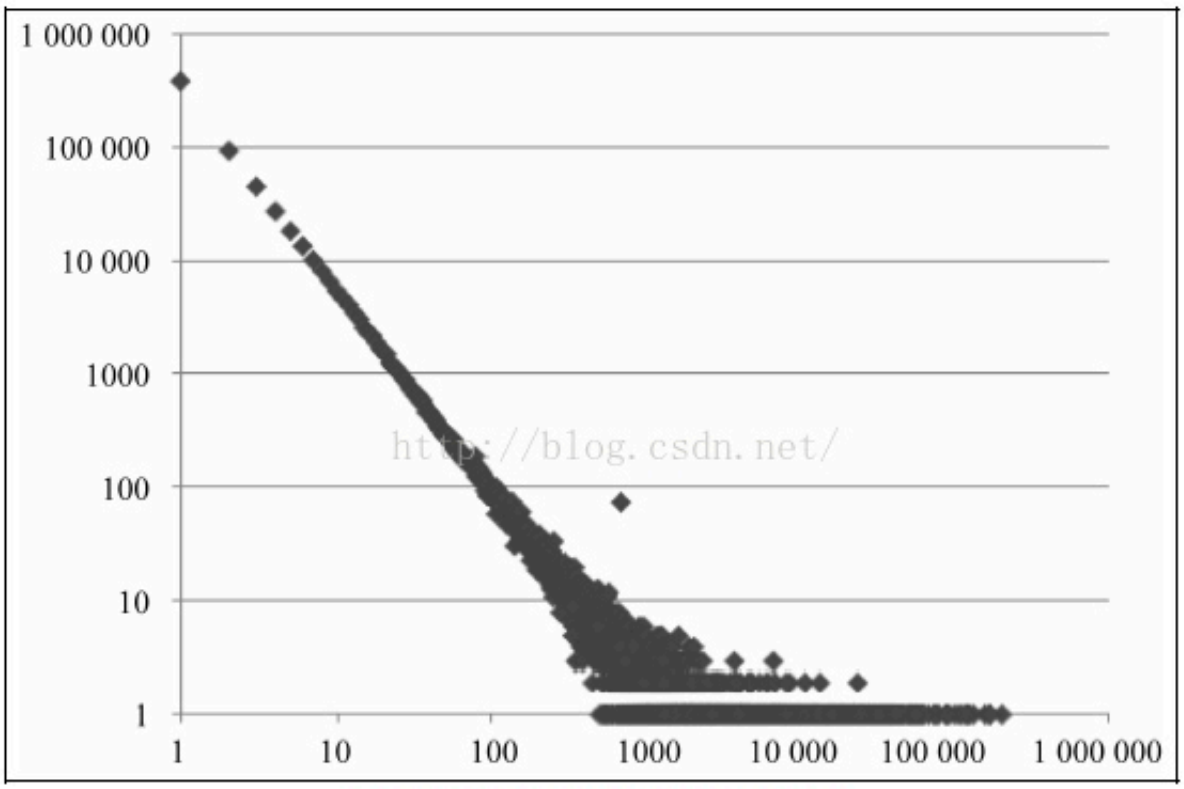


图4-8　标签流行度的长尾分布

在用户看到一个物品时，我们希望他打的标签是能够准确的描述物品内容属性的关键词，但用户往往不是按照我们的想法操作，而是可能给物品打上各种各样奇奇怪怪的标签，此时便需要我们人工编辑一些特定的标签供用户选择，Scott A. Golder 总结了Delicious上的标签，将它们分为如下几类。

- 表明物品是什么

  比如是一只鸟，就会有“鸟”这个词的标签；是豆瓣的首页，就有一个标签叫“豆瓣”；是乔布斯的首页，就会有个标签叫“乔布斯”。

- 表明物品的种类

  比如在Delicious的书签中，表示一个网页类别的标签包括 article（文章）、blog（博客）、book（图书）等。

- 表明谁拥有物品

  比如很多博客的标签中会包括博客的作者等信息。

- 表达用户的观点

  比如用户认为网页很有趣，就会打上标签funny（有趣），认为很无聊，就会打上标签boring（无聊）。

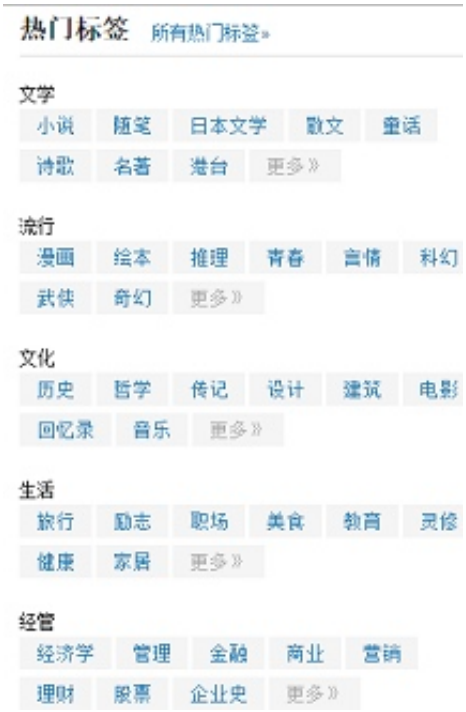
- 用户相关的标签

  比如 my favorite（我最喜欢的）、my comment（我的评论）等。

- 用户的任务

  比如 to read（即将阅读）、job search（找工作）

比如在豆瓣上，标签便被分为如下几个类别



四. 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。

豆瓣很好地利用了标签数据，它将标签系统融入到了整个产品线中。

首先，在每本书的页面上，豆瓣都提供了一个叫做“豆瓣成员常用标签”的应用，它给出了这本书上用户最常打的标签。

同时，在用户给书做评价时，豆瓣也会让用户给图书打标签。

最后，在最终的个性化推荐结果里，豆瓣利用标签将用户的推荐结果做了聚类，显示了对不同标签下用户的推荐结果，从而增加了推荐的多样性和可解释性。

一个用户标签行为的数据集一般由一个三元组的集合表示，其中记录(u, i, b) 表示用户u给物品i打上了标签b。当然，用户的真实标签行为数据远远比三元组表示的要复杂，比如用户打标签的时间、用户的属性数据、物品的属性数据等。但是为了集中讨论标签数据，只考虑上面定义的三元组形式的数据，即用户的每一次打标签行为都用一个三元组（用户、物品、标签）表示。

1. 试验设置

本节将数据集随机分成10份。

这里分割的键值是用户和物品，不包括标签。也就是说，用户对物品的多个标签记录要么都被分进训练集，要么都被分进测试集，不会一部分在训练集，另一部分在测试集中。然后，我们挑选1份作为测试集，剩下的9份作为训练集，通过学习训练集中的用户标签数据预测测试集上用户会给什么物品打标签。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。

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

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

将上面的实验进行10次，每次选择不同的测试集，然后将每次实验的准确率和召回率的平均值作为最终的评测结果。为了全面评测个性化推荐的性能，我们同时评测了推荐结果的覆盖率（coverage）、多样性（diversity）和新颖度。覆盖率的计算公式如下：

$$\text{Coverage} = \frac{|\bigcup_{u \in U} R(u)|}{|I|}$$

接下来我们用物品标签向量的余弦相似度度量物品之间的相似度。对于每个物品i，item_tags[i]存储了物品i的标签向量，其中item_tags[i][b]是对物品i打标签b的次数，那么物品i和j的余弦相似度可以通过如下程序计算。

```
#计算余弦相似度
def CosineSim(item_tags,i,j):
    ret = 0
    for b,wib in item_tags[i].items():      #求物品i,j的标签交集数目
        if b in item_tags[j]:
            ret += wib * item_tags[j][b]
    ni = 0
    nj = 0
```

```
    for b, w in item_tags[i].items():       #统计 i 的标签数目
        ni += w * w
    for b, w in item_tags[j].items():       #统计 j 的标签数目
        nj += w * w
    if ret == 0:
        return 0
    return ret/math.sqrt(ni * nj)           #返回余弦值
```

在得到物品之间的相似度度量后，我们可以用如下公式计算一个推荐列表的多样性：

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

Python实现为：

```
#计算推荐列表多样性
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)
```

推荐系统的多样性为所有用户推荐列表多样性的平均值。

至于推荐结果的新颖性，我们简单地用推荐结果的平均热门程度（AveragePopularity）度量。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：

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

2. 一个简单的算法

拿到了用户标签行为数据，相信大家都可以想到一个最简单的个性化推荐算法。这个算法的描述如下所示。

- 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。
  对于上面的算法，用户u对物品i的兴趣公式如下：

$$p(u,i) = \sum_b n_{u,b} n_{b,i}$$

这里，B(u)是用户u打过的标签集合，B(i)是物品i被打过的标签集合，$n_{u,b}$是用户u打过标签b的次数，$n_{b,i}$是物品i被打过标签b的次数。本章用SimpleTagBased标记这个算法。

在Python中，我们遵循如下约定：

- 用 records 存储标签数据的三元组，其中records[i] = [user, item, tag];
- 用 user_tags 存储$n_{u,b}$，其中user_tags[u][b] = $n_{u,b}$;
- 用 tag_items存储$n_{b,i}$，其中tag_items[b][i] = $n_{b,i}$。

如下程序可以从records中统计出user_tags和tag_items：

```
# 从records中统计出user_tags和tag_items
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)
```

统计出user_tags和tag_items之后，我们可以通过如下程序对用户进行个性化推荐：

```
# 对用户进行个性化推荐
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
```

五. 算法的改进

再次回顾四中提出的简单算法

$$p(u,i)=\sum_b n_{u,b}n_{b,i}$$

该算法存在许多缺点，比如说对于热门商品的处理，数据稀疏性的处理等，这也是在推荐系统中经常会遇见的问题

1. TF-IDF

前面这个公式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。这里我们可以借鉴TF–IDF的思想，对这一公式进行改进：

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

这里，$n_b^{(u)}$记录了标签b被多少个不同的用户使用过。这个算法记为TagBasedTFIDF。

同理，我们也可以借鉴TF–IDF的思想对热门物品进行惩罚，从而得到如下公式：

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

其中， $n_i^{(u)}$记录了物品i被多少个不同的用户打过标签。这个算法记为TagBasedTFIDF++。

2. 数据稀疏性

在前边的算法中，用户兴趣和物品的联系是通过B(u) B(i)交集得到的，但是对于新用户，这个交集

的结果将会非常小，为了提高推荐结果的可靠性，这里我们要对标签进行扩展，，比如若用户曾经用过“推荐系统”这个标签，我们可以将这个标签的相似标签也加入到用户标签集合中，比如“个性化”、“协同过滤”等标签。

进行标签扩展的方法有很多，比如说话题模型（参考博客），这里遵循简单原则介绍一种基于邻域的方法。

标签扩展的本质是找到与他相似的标签，也就是计算标签之间的相似度。最简单的相似度可以是同义词。如果有一个同义词词典，就可以根据这个词典进行标签扩展。如果没有这个词典，我们可以从数据中统计出标签的相似度。

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：

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

3. 标签清理

不是所有标签都能反应用户的兴趣。比如，在一个视频网站中，用户可能对一个视频打了一个表示情绪的标签，比如“不好笑”，但我们不能因此认为用户对“不好笑”有兴趣，并且给用户推荐其他具有“不好笑”这个标签的视频。相反，如果用户对视频打过“成龙”这个标签，我们可以据此认为用户对成龙的电影感兴趣，从而给用户推荐成龙其他的电影。同时，标签系统里经常出现词形不同、词义相同的标签，比如recommender system和recommendation engine就是两个同义词。

标签清理的另一个重要意义在于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。

一般来说有如下标签清理方法：

- 去除词频很高的停止词；
- 去除因词根不同造成的同义词，比如 recommender system和recommendation system；
- 去除因分隔符造成的同义词，比如 collaborative_filtering和collaborative-filtering。

为了控制标签的质量，很多网站也采用了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。

推荐程序（更新查看地址：GitHub // 数据集下载）：

```
1 #!/usr/bin/env python
```

```
#-*-coding:utf-8-*-
import random
# 统计各类数量
def addValueToMat(theMat,key,value,incr):
    if key not in theMat: # 如果key没出先在theMat中
        theMat[key]=dict();
        theMat[key][value]=incr;
    else:
        if value not in theMat[key]:
            theMat[key][value]=incr;
        else:
            theMat[key][value]+=incr;# 若有值，则递增

user_tags = dict();
tag_items = dict();
user_items = dict();
user_items_test = dict();# 测试集数据字典

# 初始化，进行各种统计
def InitStat():
    data_file = open('delicious.dat')
    line = data_file.readline();
    while line:
        if random.random()>0.1:# 将90%的数据作为训练集，剩下10%的数据作为测试集
            terms = line.split("\t");# 训练集的数据结构是[user, item, tag]形式
            user=terms[0];
            item=terms[1];
            tag=terms[2];
            addValueToMat(user_tags,user,tag,1)
            addValueToMat(tag_items,tag,item,1)
            addValueToMat(user_items,user,item,1)
            line = data_file.readline();
        else:
            addValueToMat(user_items_test,user,item,1)
    data_file.close();

# 推荐算法
def Recommend(usr):
    recommend_list = dict();
    tagged_item = user_items[usr];# 得到该用户所有推荐过的物品
    for tag_,wut in user_tags[usr].items():# 用户打过的标签及次数
        for item_,wit in tag_items[tag_].items():# 物品被打过的标签及被打过的次数

            if item_ not in tagged_item:# 已经推荐过的不再推荐
                if item_ not in recommend_list:
                    recommend_list[item_]=wut*wit;# 根据公式
                else:
                    recommend_list[item_]+=wut*wit;
```

```
    return sorted(recommend_list.iteritems(), key=lambda a:a[1],reverse=True
)

InitStat()
recommend_list = Recommend("48411")
# print recommend_list
for recommend in recommend_list[:10]:  # 兴趣度最高的十个itemid
    print recommend
```

运行结果：

('912', 610)

('3763', 394)

('52503', 238)

('39051', 154)

('45647', 147)

('21832', 144)

('1963', 143)

('1237', 140)

('33815', 140)

('5136', 138)

六. 标签推荐

当用户浏览某个物品时，标签系统非常希望用户能够给这个物品打上高质量的标签，这样才能促进标签系统的良性循环。因此，很多标签系统都设计了标签推荐模块给用户推荐标签。

1. 为什么给用户推荐标签
    - 方便用户输入标签 让用户从键盘输入标签无疑会增加用户打标签的难度，这样很多用户不愿意给物品打标签，因此我们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 同一个语义不同的用户可能用不同的词语来表示。这些同义词会使标签的词表变得很庞大，而且会使计算相似度不太准确。而使用推荐标签时，我们可以对词表

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。

2. 标签推荐的四种简单算法

- a. 给用户u推荐整个系统里最热门的标签（这里将这个算法称为PopularTags）令tags[b]为标签b的热门程度，那么这个算法的实现如下：

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

- b. 给用户u推荐物品i上最热门的标签（这里将这个算法称为ItemPopularTags）。令item_tags[i][b]为物品i被打上标签b的次数

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

- c. 是给用户u推荐他自己经常使用的标签（这里将这个算法称为UserPopularTags）。令user_tags[u][b]为用户u使用标签b的次数

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

- d. 前面两种的融合（这里记为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]
```

注意在上面的实现中，我们在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。