Notes for Recommending System

实验方法

- 1.离线实验：用于测试大量的算法并选取合适的
- 2.在线实验：上线测试
- 3.用户调查：

数据集

- *1.MovieLens
  *2.豆瓣
  3.Netflix
  4.IMDb

评测指标

- 1.TopN推荐预测度量：准确率(precision)&召回率(recall)，然后画出准确率/召回率曲线
  - 设$R(u)$为训练集的推荐列表，$T(u)$为测试集上的列表

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

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

```
def PrecisionRecall(test, N):
        hit = 0
        n_recall = 0
        n_precision = 0
        for user, items in test.items():
                rank = Recommend(user, N)
                hit += len(rank & items)
                n_recall += len(items)
                n_precision += N
return [hit / (1.0 * n_recall), hit / (1.0 * n_precision)]
```

```
增加train集
def Recall(train, test, N):
        hit = 0
        all = 0
        for user in train.keys():
                tu = test[user]
                rank = GetRecommendation(user, N)
                for item, pui in rank:
                        if item in tu:
                                hit += 1
                all += len(tu)
        return hit / (all * 1.0)
def Precision(train, test, N):
        hit = 0
        all = 0
        for user in train.keys():
                tu = test[user]
                rank = GetRecommendation(user, N)
                for item, pui in rank:
                        if item in tu:
                                hit += 1
                all += N
        return hit / (all * 1.0)
```

- 2.覆盖率(Coverage)：推荐系统能够推荐出来的物品占总物品集合的比例，是描述一个推荐系统对物品长尾的发掘能力
  - 好的推荐系统需要有比较高的用户满意度，也可以考虑较高的覆盖率

    - 2.1 简单覆盖率：设用户集合为U，$R(u)$为训练集的推荐列表，I为总物品的集合

      $Coverage = \frac{U_{R(u)}}{I}$
    - 2.2 信息熵和基尼系数：通过研究物品在推荐列表中出现次数的分布描述推荐系统挖掘长尾的能力
      - 设p()计算物品流行程度，i和j为物品量

      $Entropy = -\sum_{i=1}^{n} P(i) logP(i)$

      $Gini = \frac{1}{n-1}\sum_{j=1}^{n}(2j-n-1)P(j)$ (通行算法)
    - 2.3 社会学马太效应：系统会增大热门物品和非热门物品的流行度差距，而推荐系统的初衷是希望消除马太效应，使得各种物品能被展示给对它们感兴趣的某一类人群
      - 用Gini系数衡量马太效应，减少推荐算法中基尼系数和实际基尼系数两者之间的差距

```
# 覆盖率
def Coverage(train, test, N):
        recommend_items = set()
        all_items = set()
        for user in train.keys():
                for item in train[user].keys():
                        all_items.add(item)
                rank = GetRecommendation(user, N)
                for item, pui in rank:
                        recommend_items.add(item)
        return len(recommend_items) / (len(all_items) * 1.0)
```

```
# 基尼系数
def GiniIndex(p):
                j = 1
                n = len(p)
                G = 0
                for item, weight in sorted(p.items(), key=itemgetter(1)):
                                G += (2 * j - n - 1) * weight
return G / float(n - 1)
```

- 3.多样性：使推荐列表能覆盖用户的多个兴趣点
  - 可以按照电影的类别对推荐结果中的电影分类，然后每种类别都选出几部电影组成最终的推荐结果
  - 设$s(i,j) \in [0,1]$为物品i和j之间的相似度

    $Diversity = 1 - \frac{\sum s(i,j)}{0.5*|R(u)|(|R(u)|-1)}$

    $Diversity = \frac{1}{|U|\sum_{U} Diversity(R(u))}$ (整体多样性)
- 4.实时性
- 5.新颖度

算法实现

- 1.用户 & Item 协同过滤算法
- 2.基于标签的推荐系统
- 3.协同过滤算法

协同过滤算法

- 得到`用户-物品`，`物品-评分`格式的数据
- 假设有用户u和v，物品i和j，采用Jaccard公式或cos相似度的计算
  - 1.基于用户(UserCF)：评测用户之间的相似性，给用户推荐和他兴趣相投的`其他用户`喜欢的物品
    - 构建**用户**相似度矩阵w
    - **相似度计算**：$W_{uv} = \frac{|N(u) \cap N(v)|}{\sqrt{|N(u)||N(v)|}}$
      - N(u)表示用户喜欢的物品的集合
    - **相似度计算改进1**：首先计算出$|N(u) \cap N(v)| \neq 0$的用户对(u,v)进行初步筛选,之后用余弦相似度计算
    - **相似度计算改进2**：增加量惩罚机制，惩罚用户u和用户v共同兴趣列表中热门物品对他们相似度的影响

      $W_{uv} = \frac{|N(u) \cap N(v)| \frac{1}{log1+N(i)}}{\sqrt{|N(u)||N(v)|}}$

- **推荐公式：** $P(u,i)=\sum_{v\in S(u,k)\cap N(i)} W_{uv}R_{vi}$
  - S(u,k)表示和用户u兴趣最接近的K个用户，N(i)表示对物品i有过行为的用户集合，W_{uv}表示用户u和用户v的兴趣相似度，R_{vi}表示用户v对物品i的评分)
  - 选取不同k进行评测，得出最佳推荐数量的物品

```
# 相似度计算
def UserSimilarity(train):
        W = dict()
        for u in train.keys():
                for v in train.keys():
                        if u == v:
                                continue
                        W[u][v] = len(train[u] & train[v])
                        W[u][v] /= math.sqrt(len(train[u]) * len(train[v]) * 1.0)
        return W
```

(然而非常耗时)

```
# 相似度计算改进
def UserSimilarity(train):
        # build inverse table for item_users(倒排表)
        item_users = dict()
        for u, items in train.items():
                for i in items.keys():
                        if i not in item_users:
                                item_users[i] = set()
                        item_users[i].add(u)

        #calculate co-rated items between users
        C = dict()
        N = dict()
        for i, users in item_users.items():
                for u in users:
                        N[u] += 1
                        for v in users:
                                if u == v:
                                        continue
                                C[u][v] += 1
                                # C[u][v] += 1/math.log(1+len(users))

        #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

# 推荐
def Recommend(user, 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, rvi in train[v].items:
                if i in interacted_items:
                        #we should filter items user interacted before
                        continue
                rank[i] += wuv * rvi
        return rank
```

- 2.基于物品(ItemCF)：评测物品之间的相似性，给用户推荐和他之前喜欢物品`相似的物品`
  - 构建**物品**相似度矩阵w
  - **相似度计算1：** $W_{ij}=\frac{|N(i)\cap N(j)|}{|N(i)|}$
    - N(i)表示对物品i有过行为的用户集合，可以理解为喜欢物品i的用户中有多少比例的用户也喜欢物品j
  - **相似度计算2**$W_{ij}=\frac{|N(i)\cap N(j)|}{\sqrt{|N(i)||N(j)|}}$

    - 如果物品j很热门，很多人都喜欢，那么W_{ij}就会很大,需要避免一直推荐出太热门的物品
- **相似度计算改进**：软性惩罚机制 $W_{ij}=\frac{|N(i)\cap N(j)|\frac{1}{log1+|N(u)|}}{\sqrt{|N(i)||N(j)|}}$

  $W_{ij}=\frac{|N(i)\cap N(j)|}{|N(i)|^{1-a}|N(j)|^{a}} a\in[0.5,1]$
- 推荐公式：$P(i,j)=\sum_{v\in S(j,k)\cap N(u)} W_{ij}R_{ui}$
    - S(j,K)是和物品j最相似的K个物品的集合，N(u)表示用户喜欢的物品的集合，W_{ji}是物品j对i的相似度，R_{ui}是用户u对物品i的兴趣
- **归一化**：用于提升准确度，覆盖率和多样性 $W_{ij'}=\frac{W_ij}{max_jW_ij}$

```
# 相似度计算
def ItemSimilarity(train):
        #calculate co-rated users between items
        C = dict()
        N = dict()
        for u, items in train.items():
                for i in users:
                        N[i] += 1
                        for j in users:
                                if i == j:
                                        continue
                                C[i][j] += 1
                                # C[i][j] += 1 / math.log(1+len(items)*1.0)

        #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

# 推荐公式
def Recommendation(train, user_id, W, K):
        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 in ru:
                        continue
                rank[j] += pi * wj
        return rank
```

- UserCF vs ItemCF

	UserCF	ItemCF
性能	适用于用户较少的场合，如果用户很多，计算用户相似度矩阵代价很大	适用于物品数明显小于用户数的场合，如果物品很多（网页），计算物品相似度矩阵代价很大
领域	时效性较强，用户个性化兴趣不太明显的领域	长尾物品丰富，用户个性化需求强烈的领域
实时性	用户有新行为，不一定造成推荐结果的立即变化	用户有新行为，一定会导致推荐结果的实时变化
冷启动	在新用户对很少的物品产生行为后，不能立即对他进行个性化推荐，因为用户相似度表是每隔一段时间离线计算的	新用户只要对一个物品产生行为，就可以给他推荐和该物品相关的其他物品
	新物品上线后一段时间，一旦有用户对物品产生行为，就可以将新物品推荐给和对它产生行为的用户兴趣相似的其他用户	但没有办法在不离线更新物品相似度表的情况下将新物品推荐给用户
推荐理由	很难提供令用户信服的推荐解释	利用用户的历史行为给用户做推荐解释，可以令用户比较信服

基于标签的推荐

隐含语义分析(LFM)

- other: LSA,LDA,LCM,LTM,matrix factorization

系统冷启动问题

推荐系统架构

- (http://www.mymediaproject.org/default.aspx)