BPR 算法设计文档

小组成员：龚润宇、梅楚鹤、王胜广、柳俊志、尹国健

1. BPR 算法介绍

1.1 背景

最早的第一类排序算法类别是点对方法(Pointwise Approach)，这类算法将排序问题被转化为分类、回归之类的问题，并使用现有分类、回归等方法进行实现。第二类排序算法是成对方法(Pairwise Approach)，在序列方法中，排序被转化为对序列分类或对序列回归。所谓的 pair 就是成对的排序，比如(a,b)一组表明 a 比 b 排的靠前。我们要讲到的 BPR 就属于这一类。第三类排序算法是列表方法(Listwise Approach)，它采用更加直接的方法对排序问题进行了处理。它在学习和预测过程中都将排序列表作为一个样本。排序的组结构被保持。

1.2 BPR 建模思路

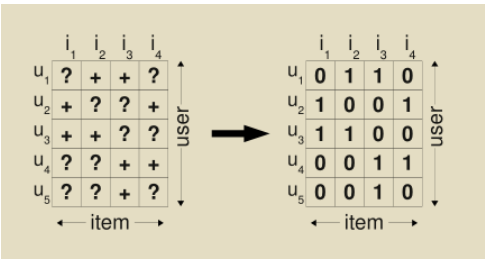
贝叶斯个性化排序(Bayesian Personalized Ranking, 以下简称 BPR)，为用户进行个性化推荐需要实现一种排序的机制，使得用户对物品的偏好可以量化比较。用户 u 的排序规则可以用$>_u$表示，那么$i >_u j$表示用户 u 对物品 i 的偏好大于其对物品 j 的偏好.这样的排序规则应该满足如下的性质：

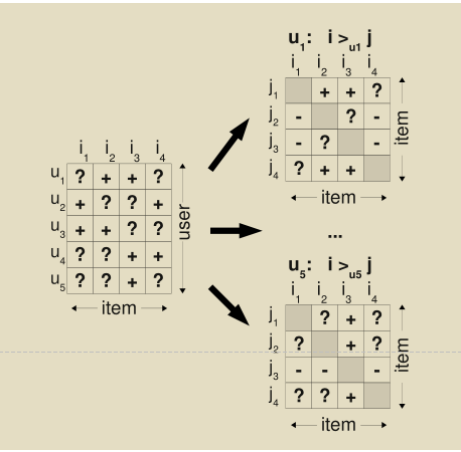
$$\forall i, j \in I : i \neq j \Rightarrow i >_u j \vee j >_u i \qquad (totality)$$

$$\forall i, j \in I : i >_u j \wedge j >_u i \Rightarrow i = j \qquad (antisymmetry)$$

$$\forall i, j, k \in I : i >_u j \wedge j >_u k \Rightarrow i >_u k \qquad (transitivity)$$

传统的推荐方法将用户—物品的稀疏矩阵 U-I 中的所有缺失值设为 0，即统一认为是用户不感兴趣的.而事实上这些缺失值代表的是用户不感兴趣的物品和感兴趣而未发生关联的物品的集合.因此用简单的 0 代替全部缺失值后，再经过机器学习拟合出来的模型理论上不能推荐任何东西，只能通过正则化的方法来避免过拟合.





因此，将原有的 U-I 的矩阵拆分成每个用户的物品 I-J 的偏好矩阵，避免全部替换为 0 造成的混淆，再经过算法计算得到其余物品之间未知的偏好关系，从而实现物品间的排序.

1.2.1 基本假设

（a）用户间的行为相互独立。
（b）对于某一用户而言，该用户对某一物品对(i, j)的排序，与该用户对其他物品对的排序之间是相互独立的。
（c）定义用户 u 对 i 比 j 的偏好值：

$$\hat{x}_{uij} \coloneqq \hat{x}_{ui} - \hat{x}_{uj} \tag{1}$$

（d）定义用户 u 对物品 i 相对于 j 的偏好概率：

$$p(i >_u j|\Theta) \coloneqq \sigma\left(\hat{x}_{uij}(\Theta)\right) \tag{2}$$

$$\sigma(x) \coloneqq \frac{1}{1+e^{-x}} \tag{3}$$

（e） $P(\theta)$的概率分布服从均值为 0，协方差矩阵为$\lambda_\theta I$的正态分布.

1.2.2 建模

由前一章节了解到 BPR 的主要目标是求取用户 u 对 i 比 j 的偏好值$\hat{x}_{uij}$，根据论文给出的定义该问题转化为求取$\hat{x}_{ui}$，这可以使用矩阵分解的方法，对于用户集 U 和物品集 I 的对应的 U×I 的预测排序矩阵 X，我们期望得到两个分解后的用户矩阵 W:|U|×k 和物品矩阵 H:|I|×k，满足

$$\hat{X} \coloneqq WH^t \tag{4}$$

其中 W 中的每一行表示一个用户 u，H 中的每一行表示一个物品 i，上述公式可以表示为：

$$\hat{x}_{ui} = \langle w_u, h_i \rangle = \sum_{f=1}^{k} w_{uf} \cdot h_{if} \tag{5}$$

$\Theta = (\mathrm{W}, \mathrm{H})$就是模型的参数.

将任意用户 u 对应的物品进行标记，如果用户 u 在同时有物品 i 和 j 的时候点击了 i，那么我们就得到了一个三元组<u,i,j>，它表示对用户 u 来说，i 的排序要比 j 靠前。如果对于用户 u 来说我们有 m 组这样的反馈，那么我们就可以得到 m 组用户 u 对应的训练样本。

1.2.3 优化

由先前的假设，我们可以推出：

$$P(\theta| >_u) \propto P(>_u |\theta)P(\theta) \tag{6}$$

对于$P(>_u |\theta)$：

$$\prod_{u \in U} P(>_u |\theta) = \prod_{(u,i,j) \in (U \times I \times J)} P(i >_u j|\theta)^{\delta((u,i,j) \in D)} \left(1 - P(i >_u j|\theta)\right)^{\delta((u,i,j) \notin D)} \tag{7}$$

其中,

$$\delta(b) = \begin{cases} 1 & if\ b\ is\ true \\ 0 & else \end{cases} \tag{8}$$

化简得到,

$$P(i >_u j|\theta) = \left(\hat{x}_{uij}(\theta)\right) \tag{9}$$

将式(1)代入式(9)得到，

$$\prod_{u\in U} P(>_u|\theta) = \prod_{(u,i,j)\epsilon D} \sigma(\hat{x}_{ui} - \hat{x}_{uj}) \quad (10)$$

对于$P(\theta)$，根据假设 v 有：

$$\ln P(\theta) = \lambda\|\theta\|^2 \quad (11)$$

将式(10)和(11)代入式(6)，有

$$\ln P(\theta|>_u) \propto \ln P(>_u|\theta)P(\theta) = \ln \prod_{(u,i,j)\epsilon D} \sigma(\hat{x}_{ui} - \hat{x}_{uj}) + \ln P(\theta)$$

$$= \sum_{(u,i,j)\epsilon D} \ln \sigma(\hat{x}_{ui} - \hat{x}_{uj}) + \lambda\|\theta\|^2 \quad (12)$$

这个式子可以用梯度上升法来优化求解模型参数。对 θ 求导，我们有

$$\frac{\partial \ln P(\theta|>_u)}{\partial} \propto \sum_{(u,i,j)\epsilon D} \frac{1}{1+e^{\hat{x}_{ui}-\hat{x}_{uj}}} \frac{\partial(\hat{x}_{ui} - \hat{x}_{uj})}{\partial} + \lambda\theta \quad (13)$$

由于

$$\hat{x}_{ui} - \hat{x}_{uj} = \sum_{f=1}^{k} w_{uf} \cdot h_{if} - \sum_{f=1}^{k} w_{uf} \cdot h_{jf} \quad (14)$$

可以求得：

$$\frac{\partial(\hat{x}_{ui} - \hat{x}_{uj})}{\partial} = \begin{cases} (h_{if} - h_{jf}) & if\ \theta = w_{uf} \\ w_{uf} & if\ \theta = h_{if} \\ -w_{uf} & if\ \theta = h_{jf} \end{cases}$$

2 编程实现

数据集：MovieLens 100K 数据集。包含 943 个用户对 1682 部电影的打分。由于 BPR 是排序算法，因此数据集里的打分会被我们忽略，主要是假设用户看过的电影会比用户没看的电影的排序评分高。

2.1 算法流程、伪码：

输入：训练集 D 三元组，梯度步长 α，正则化参数 λ，分解矩阵维度 k.

输出：模型参数，矩阵 W，H.

1). 随机初始化矩阵 W，H

2). 迭代更新模型参数：

$$w_{uf} = w_{uf} + \alpha\left(\sum_{(u,i,j)\epsilon D} \frac{1}{1+e^{\hat{x}_{ui}-\hat{x}_{uj}}}(h_{if} - h_{jf}) + \lambda w_{uf}\right)$$

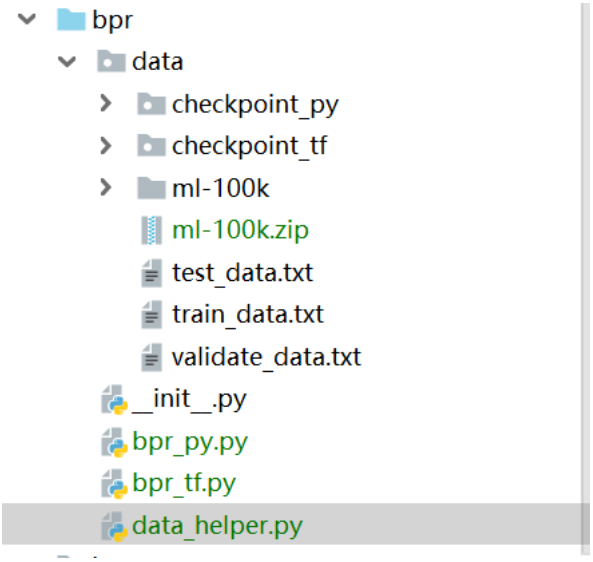
$$h_{if} = h_{if} + \alpha\left(\sum_{(u,i,j)\epsilon D} \frac{1}{1+e^{\hat{x}_{ui}-\hat{x}_{uj}}} w_{uf} + \lambda h_{if}\right)$$

$$h_{jf} = h_{jf} + \alpha\left(\sum_{(u,i,j)\epsilon D} \frac{1}{1+e^{\hat{x}_{ui}-\hat{x}_{uj}}}(-w_{uf}) + \lambda h_{jf}\right)$$

3). 如果 W，H 收敛，则算法结束，输出 W，H，否则回到步骤 2.

2.2 代码详解

代码分为 tensorflow 版和纯 python numpy 版两个部分，其中数据载入代码两个版本公用。



2.2.1 数据载入（data_helper.py）

1）数据集切分

```
def split_data(self, samples, validation_scale, test_scale):
    """切分数据集"""
    test_num = int(len(samples) * test_scale) if isinstance(
        test_scale, float) else test_scale  # 测试数据所占比重 或者测试数据数
    validate_num = int(len(samples) * validation_scale) if isinstance(
        validation_scale, float) else validation_scale  # 测试数据所占比重 或
    random.shuffle(samples)
    train_data = samples[:len(samples) - validate_num - test_num]
    validate_data = samples[len(samples) - validate_num - test_num:len(sample
    test_data = samples[len(samples) - validate_num:]
    for datas, data_type in [(train_data, "train"), (validate_data, "validat
        with open(os.path.join(data_dir, "{}_data.txt".format(data_type)), "
            _lines = [",".join(one_sample) for one_sample in datas]
```

2）生成训练数据

```
def generate_samples(self, batch_size=1, epoch_num=1, shuffle=True, data_type="train"):
    """
    :param type: train test 训练或测试
    :param test_scale: count 或 比例
    :return:
    """
    data = {"train": self.train_data, "validate": self.validate_data, "test": self.test_data
    for epoch_num in range(epoch_num):  # 每个周期数据分多少批数据
        if shuffle:
            shuffle_indices = np.random.permutation(np.arange(len(data)))
            shuffled_data = np.array(data)[shuffle_indices]
        else:
            shuffled_data = data
        batch_data = []
```

2.2.2 模型构建

1）python 版

```
def train_step(self, batch_samples, update_parm=True):
    """一次训练迭代过程"""
    u = batch_samples[:, 0] - 1  # 编号-1成索引
    i = batch_samples[:, 1] - 1
    j = batch_samples[:, 2] - 1
    r_ui = np.dot(self.U[u], self.V[i].T) + self.biasV[i]
    r_uj = np.dot(self.U[u], self.V[j].T) + self.biasV[j]
    r_uij = r_ui - r_uj
    loss_func = -1.0 / (1 + np.exp(r_uij))
    loss_func = np.mean(loss_func)
    # update U and V
    if update_parm:
        self.U[u] += -self.lr * (loss_func * (self.V[i] - self.V[j]) + self.reg * self.U[u])
        self.V[i] += -self.lr * (loss_func * self.U[u] + self.reg * self.V[i])
        self.V[j] += -self.lr * (loss_func * (-self.U[u]) + self.reg * self.V[j])
        # update biasV
        self.biasV[i] += -self.lr * (loss_func + self.reg * self.biasV[i])
        self.biasV[j] += -self.lr * (-loss_func + self.reg * self.biasV[j])
    return loss_func
```

2）tensorflow 版

```
def bpr_mf(user_count, item_count, hidden_dim):
    """模型定义"""
    u = tf.placeholder(tf.int32, [None], name="input_u")
    i = tf.placeholder(tf.int32, [None], name="input_i")
    j = tf.placeholder(tf.int32, [None], name="input_j")
    # with tf.device("/cpu:0"):
    user_emb_w = tf.get_variable("user_emb_w", [user_count + 1, hidden_dim],
                                 initializer=tf.random_normal_initializer(0, 0.1))
    item_emb_w = tf.get_variable("item_emb_w", [item_count + 1, hidden_dim],
                                 initializer=tf.random_normal_initializer(0, 0.1))
    u_emb = tf.nn.embedding_lookup(user_emb_w, u)
    i_emb = tf.nn.embedding_lookup(item_emb_w, i)
    j_emb = tf.nn.embedding_lookup(item_emb_w, j)
    # MF predict: u_i > u_j
    x = tf.reduce_sum(tf.multiply(u_emb, (i_emb - j_emb)), 1, keep_dims=True)
    # AUC for one user:
    # reasonable iff all (u,i,j) pairs are from the same user
    # average AUC = mean( auc for each user in test set)
    mf_auc = tf.reduce_mean(tf.to_float(x > 0))
```

2.2.3 模型保存

1）Python numpy 版

```
print("step:{}, loss: {}, val_loss:{}".format(step, train_loss, val_loss))
model_path = os.path.join(checkpoint_dir, "{}_{:.4f}.npy".format(step, val_loss))
np.save(model_path, [self.U, self.V])
```

2）tensorflow 版

```
step += 1
if step % 1000 == 0:
    print("step:{}, loss: {}".format(step, loss))
    path = saver.save(session, checkpoint_prefix, global_step=step)
    print("Saved model checkpoint to {}\n".format(path))
```

2.2.4 模型加载

1）Python 版

```
def load_model(self, saved_model_path=""):
    """模型加载"""
    if not saved_model_path:
        saved_model_path = self.get_min_loss_model_path()
    U, V = np.load(saved_model_path)
    return U, V
```

2）tensorflow 版

```
def load_model():
    """模型加载"""
    graph = tf.Graph()
    with graph.as_default(), tf.Session() as sess:
        checkpoint_file = tf.train.latest_checkpoint(checkpoint_dir)
        saver = tf.train.import_meta_graph("{}.meta".format(checkpoint_file))
        saver.restore(sess, checkpoint_file)
        user_emb_w, item_emb_w = sess.run(
            [graph.get_tensor_by_name('user_emb_w:0'), graph.get_tensor_by_name("item_emb_w:0")])
    return user_emb_w, item_emb_w
```

2.2.5 AUC

1）Python 版

```
def samples_auc(batch_samples):
    u = batch_samples[:, 0] - 1  # index = user_id-1
    i = batch_samples[:, 1] - 1
    j = batch_samples[:, 2] - 1
    r_ui = np.dot(self.U[u], self.V[i].T) + self.biasV[i]  # predict
    r_uj = np.dot(self.U[u], self.V[j].T) + self.biasV[j]  # predict
    r_uij = r_ui - r_uj
    return r_uij

auc = []
loss = []
for batch_samples in self.data_helper.generate_samples(batch_size=100
    r_uij = samples_auc(batch_samples)
    _loss = self.train_step(batch_samples, update_parm=False)
    auc.append(r_uij)
    loss.append(_loss)
auc = np.array(auc)
test_auc = np.sum(auc > 0) / np.prod(auc.shape)
```

2）tensorflow 版

```
# MF predict: u_i > u_j
x = tf.reduce_sum(tf.multiply(u_emb, (i_emb - j_emb)), 1, keep_dims=True)
# AUC for one user:
# reasonable iff all (u,i,j) pairs are from the same user
# average AUC = mean( auc for each user in test set)
mf_auc = tf.reduce_mean(tf.to_float(x > 0))
```

2.2.6 推荐预测

1）Python 版

```
def predict(self, user_id):
    """预测推荐"""
    U, V = self.load_model()
    _predict = np.dot(U[user_id], V.T)
    _predict = largest_indices(_predict, 3)
    return _predict
```

2） tensorflow 版

```
def predict(user_id):
    """预测推荐"""
    U, V = load_model()
    _predict = np.dot(U[user_id], V.T)
    _predict = largest_indices(_predict, 3)
    return _predict
```

2.3 运行截图

1）Python 版

```
step:10000, loss: 0.4486, val_loss:0.3941
step:20000, loss: 0.2482, val_loss:0.3349
step:30000, loss: 0.2534, val_loss:0.2963
step:40000, loss: 0.1776, val_loss:0.2714
step:50000, loss: 0.3708, val_loss:0.2541
step:60000, loss: 0.3260, val_loss:0.2426
step:70000, loss: 0.2459, val_loss:0.2312
step:80000, loss: 0.1417, val_loss:0.2229
step:90000, loss: 0.0777, val_loss:0.2136
step:100000, loss: 0.1390, val_loss:0.2093

test_loss:0.2114, test_auc:0.8482


user_id: 10, predict:[ 50 181 100]
```

2）tensorflow 版

```
step:8000, loss: 0.6911089420318604
Saved model checkpoint to /mnt/e/prj/recommender-system/recommendation/b
-8000

step:9000, loss: 0.7103589177131653
Saved model checkpoint to /mnt/e/prj/recommender-system/recommendation/b
-9000

step:10000, loss: 0.7058853507041931
Saved model checkpoint to /mnt/e/prj/recommender-system/recommendation/b
-10000

test loss:0.5019999742507935, test auc:0.7518948912620544

user_id:1, predict: [1474 1193  143]
```

3 总结

训练中发现得 bach_size 设定对训练的效果有很大影响。后续应用中还可以优化 BPR 的损失函数。比如我们可以对$\hat{x}_{uij} \coloneqq \hat{x}_{ui} - \hat{x}_{uj}$这个式子做改进，加上一个基于评分时间的衰减系数，这样我们的排序推荐还可以考虑时间等其他因素。