基于时间和影响力因子的 Github Pull Request 评审人推荐①

卢 松[1,2], 杨 达[1], 胡 军[1], 张 潇[1,2]

[1](中国科学院软件研究所 基础软件国家工程研究中心, 北京 100190)
[2](中国科学院大学, 北京 100190)

摘 要: 开源社区 github 提供了 pull request 的机制让开发者可以把自己的代码集成到 github 的开源项目中从而为项目做出贡献. Pull request 的代码评审是 github 这类分布式软件开发社区维护开源项目代码质量的非常重要的方式. 为一个新到来的 pull request 指派合适的代码评审人可以有效减少 pull request 从提交到开始审核的延迟. 目前 github 是由项目核心成员人工来完成评审人的指派, 为了减少这种人力损耗, 我们提出代码评审人的推荐系统, 该系统基于信息检索的方法, 并考虑了评审人的影响力因子以及评审的时间衰减的因素, 对新到来的 pull request, 自动推荐最相关的评审人. 我们的方法对 top 1 的准确度达到了 68%, 对 top 10 的召回率达到了 78%.

关键词: pull request; 代码评审; 信息检索; 时间因子; 影响力因子

Code Reviewer Recommendation Based on Time and Impact Factor for Pull Request in Github

LU Song[1,2], YANG Da[1], HU Jun[1], ZHANG Xiao[1,2]

[1](Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[2](University of Chinese Academy of Sciences, Beijing 100190, China)

Abstract: The pull request mechanism is widely used for integrating developers' code in github, so that developers can make contribution for open source projects. The code review of pull request is an essential method to maintain the high quality of code in github. Assigning appropriate reviewers for a newly coming pull request can effectively reduce the delay between the submission of a pull request and the actual review of it. At present, the pull request is assigned manually by core developers in the project. To reduce this cost, we propose a reviewer recommender system based on information retrieval. This method can automatically recommend highly relevant reviewers for a newly coming pull request. Our method has also taken the impact factor and time decaying factor into consideration, and has received good performance that the top 1 precision can reach 68% and top 10 recall rate can reach 78%.

Key words: pull request; code review; information retrieval; time factor; impact factor

1 概述

Github 是当前非常知名的一种分布式的版本控制系统, 拥有 140 多万开发者用户的开源社区. 开发者可以使用 watch、fork、star、pull request 等方式实现对感兴趣的项目进行社交化编程(social coding)[1].

随着 github 的项目的发展, pull request 成为了社交化编程以及代码持续集成的行之有效的方式, 据统计, 目前 github 中采用这种社交化编程的协作式项目数量占到了一半以上[2], 而且采用 pull request 进行代码集成的项目的数量将来只会越来越多. Github 的 contributor 为社区做出贡献的流程可以分为如下 5 步:

1) 首先 contributor 找到一个感兴趣的项目, 并 follow 一些该项目中的知名的开发者, watch 该项目.

2) Fork 该项目的一部分到本地, 相当于克隆到本

① 基金项目:国家自然科学基金(91218302,91318301)
收稿时间:2016-03-24;收到修改稿时间:2016-04-14 [doi:10.15888/j.cnki.csa.5455]

地; contributor 在克隆的版本上实现一个新的特性或者修复了一些 bug; 然后 contributor 使用 pull request 的方式, 把完善好的代码发送给原始的项目;

3) 项目内的所有的开发者都能够在项目的 pull request 库中评审已提交的 pull request. 他们可以讨论项目是否需要这个新特性, 提交的代码是否符合规范, 以及能否提升该 pull request 代码的质量;

4) contributor 根据评审人的建议, 会完善并更新他的 pull request, 接着评审人再次评审修改后的 pull request;

5) 项目的核心负责人基于所有评审人的意见决定是把该 pull request 合并到项目代码中还是拒绝它.

Github 开发者以及项目的数量的迅速增长, 每天产生的 pull request 的数量是惊人的, 比较知名的项目每个月会产生几百甚至上千个 pull request. 这将导致 github 的更新迭代效率非常大的程度上依赖于 pull request 提交的代码能否被及时评审. 从 pull request 提交到该 pull request 真正开始被评审人评审的时延我们称之为评审时延. 实际情况中, 由于 pull request 的相关评审人很可能没注意到该 pull request 而导致评审时延过大. 据已有的调查研究表明, 15%的 contributor 抱怨他们的 pull request 得不到评审人及时的反馈[3], 也有人专门做了关于评审时延的评估分析, 他们采用了线性回归的模型来模拟 pull request 的评审时延, 得出平均的时延大小有 364 小时, 当然有的 pull request 一直得不到评审会拉高整体的平均时延时间, 于是统计了时延大小的中位数为 15 小时[4]. 我们的评审人推荐机制可以在新的 pull request 产生时, 自动匹配与该 pull request 最相关的评审人, 并给他们发送消息, 使得评审时延大大降低, 有效地提高了 github 项目的迭代效率.

现有的 pull request 代码评审人的指派有 2 种方式, 第一种是人工指派, 需要一位项目集成管理人指派给项目开发组的核心成员进行代码评审, 但使用这种指派方式的 pull request 所占的比重只有 0.89%[5]. 另外一种更加通用的指派方式是使用@标记. 比如评论中包含@张三, 则张三会收到关于这条评论的信息. 通过@某个人的方式, 不仅可以@项目的核心成员, 也可以@项目中的任何一个 contributor, 让他们得到通知并参与讨论该 pull request. 对 pull request 的讨论可以是该 pull request 的代码整体所实现功能的意义或做的贡献, 也可以是具体的某几行代码的正确与否、代码是否规范等等的评论.

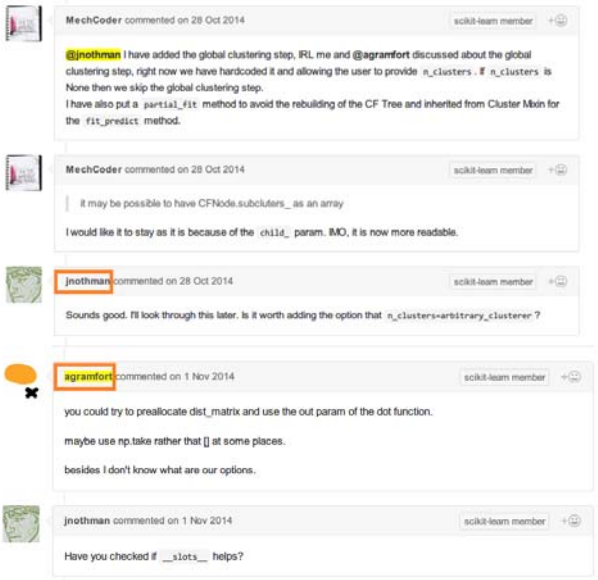

图 1 pull request 中评审人进行评审讨论

图 1 展示了在一个的 pull request 下评审人进行评审讨论的过程, 该图来自于 Github 中 scikit-learn 项目的一个 pull request, 它的标题名为 Clustering algorithm – BIRCH, 它的提交者是 MechCoder, 在提交 pull request 并进行代码修改之后, 他@jnothman 并邀请 jnothman 评审他的思路以及代码是否正确, 接着 jnothman 就给出了评论并发表了他的意见. agramfort 是另外一位评审人, 并提出了他对当前 pull request 的代码修改意见, 最后, MechCoder @jnothman @agramfort 并表达感谢他们的意见让他的思路以及代码可以不断完善, 实现了他所想要的功能, 最后他的这部分代码被 merge 到 scikit-learn 项目的主分支中. 进入该 pull request 中, 我们可以发现不仅是项目的核心成员进行了评审, 项目中的普通成员如 mblondel、coveralls 等等也加入了讨论, 只是由于图片篇幅的原因, 没在图 1 中展示出这些的评论. 所有这些评审人说的话会影响最终的 pull request 的最终决策. 作为项目的 pull request 的持续集成负责人, 他需要对项目中的 contributor 有一个比较好的了解, 才能做一些比较适当的@标记, 让负责该 pull request 这块的人能更好的对 pull request 进行评审, 提高效率. 而如果我们能自动生成和该 pull request 相关的项目组成员名单, 则可以有效减少项目持续集成负责人的工作量; 同时没

有被@的项目成员可能和该 pull request 相关性也很高, 但没有收到消息会导致他对该 pull request 的评审延迟, 我们的评审人推荐可以使用@推荐项目成员的方式通知和该 pull request 感兴趣的项目成员, 从而减少评审时延, 有效地提高 github 的社交化编程效率.

总体来说, 本文的贡献如下:

1) 使用了信息检索(information retrieval)的方法在评审人相关性评估中, 我们加入了影响力因子这个重要的考量因素. 本文首次提出在 github 项目中基于 follower 的数量来对影响力因子的量化计算方法, 我们认为 follower 数量越多的人, 他的评审的权重应该比 follower 数量少的成员权重高.

2) 我们还引入了时间维度衰减的策略, 基于信息检索方法获得与新到来 pull request 最相关的历史 pull request, 这些历史 pull request 的评审人构成了候选评审人集合. 在计算候选评审人与新到来的 pull request 相关性得分时, 引入对历史 pull request 评审的时间衰减因子. 在历史 pull request 获得相关性相同的情况下, 距离新到来 pull request 时间越长, 该历史 pull request 下的评审人所获得的相关性得分越低.

2 相关工作

我们的工作借鉴了开源社区中代码 bug 的分类报告和代码评审的组织方式. 开源社区的大型项目比如 Bugzilla 是一个开源的缺陷跟踪系统, 它可以管理软件开发中缺陷的提交(new), 修复(resolve), 关闭(close)等整个生命周期. 每天都会有几十个新 bug report 提交给 bug 追踪系统, 而分配 bug report 给适合的 bug 修复者是一件很耗精力的事情. 同样的, 代码评审系统每天也会有很多提交请求, 需要进行组织并指派相关人员进行评审.

前人做了很多研究来进行 bug report 分类或代码评审, 有机器学习的方法(如: SVM), 也有信息检索(Information Retrieval)的方法. Cubranic 等[6] 提出文本分类的方法, 目的是将 bug 对应的文本分到已经定义好的 bug report 分类标签的集合中去, 利用 bug report 的标题、描述和关键词信息来进行 bug 指派. Canfora 和 Cerulo[7]使用历史的 bug report 的文本描述作为 document 来标识开发者, 而新的新的 bug report 的描述作为 query, 如此可以构建一个信息检索系统. Kagdi 和 Linares-Vasquez[8] 提取出源代码的评论和描述信息, 并通过隐性语义索引(LSI)的方法把这些数据索引起来. 对于一个新的 bug report, 这些索引可以用于鉴别谁在之前的一段时间有修复相似的 bug. Tamrawi 等[9]对人进行建模, 认为拥有相同兴趣的人对各自的 bug report 互相评论的次数会越多. Y Yu 等[10]将这种对人的建模方法进一步深入探讨并引入到 pull request 的代码评审人推荐的情境中, 他们构建了评审人的社交网络, 网络的点就是开发者, 网络的边的权重就是评论人的相关性, 并且认为提交 pull request 的人和评审 pull request 的人对该 pull request 所属领域具有共同的兴趣.

本文所做的工作是基于信息检索的方法对 pull request 进行建模, pull request 的标题和描述信息总结了它的代码所做的贡献以及修改的主题. 所以我们把训练集中的 pull request 作为历史的 document, 而新的 pull request 的标题和描述信息作为 query, 构建信息检索系统. 并且在评审人推荐的过程中加入了评审人影响力因子的考量和时间维度衰减的考量.

3 基于信息检索的pull request 评审人推荐

我们旨在推荐和新到来的 pull request 最相关的评审人, 用于减少评审时延并有效提高 github 的社交化编程的开发效率. 已有的 bug 分类研究[11-13]所采用的信息检索的方法适用于我们的 pull request 推荐的场景. Pull request 的标题和描述信息作为标记 pull request 的 document; 新到来的 pull request 的标题和描述信息作为 query. 以此对 pull request 建模, 找出和目标 pull request 最相关的 top k 个 pull request, 这 top k 个 pull request 下的评审人构成推荐的候选评审人集合, 并基于评论次数对每位候选人进行相关性打分, 返回得分最高的 top 10 个候选人作为最终的评审推荐人.

3.1 pull request 的向量空间模型

在一个项目的场景下, 每个 pull request 采用它的标题和描述信息来标识为该 pull request 的 document. 该 pull request 下发表评论的开发者就是该 pull request 的评审人. 首先将 pull request 的停用词和特殊符号去掉, 对剩下的词做词干还原. 我们采用向量空间模型将 pull request 通过标题和描述信息映射成高维空间下的一个向量, 向量空间模型下的每一个维度代表一个词, 而该 pull request 中的单词出现的次数越多, 该维度获得的权重越大. 我们利用 tf-idf 来计算每个词的权

重, 公式如下所示:

$$tfidf(t,p_r,P_R)=log(\frac{n_t}{N_{p_r}}+1)\times log\frac{N_{P_R}}{|p_r\in P_R:t\in p_r|}$$

其中:

t: 1 个单词 (term)

p_r: 1 个 pull request

P_R: 给定项目中的所有 pull request 的仓库

n_t: 在 pull request p_r 中单词 t 所出现的次数

N_{p_r} : p_r 中的所有词的个数

N_{p_R} : 给定项目中所有 pull request 的个数

上述计算 tfidf 的公式表达的含义是:

$$\text{tfidf}(t,\ pr,\ PR)=\log\left(\frac{\text{t在当前pr出现的次数}}{\text{当前pr的term个数}}+1\right)\times\left(\frac{\text{1个project的所有pr数}}{\text{出现了t的pr数}}\right)$$

3.2 pull request 相似度

我们利用向量空间模型对 pull request 建模, 用 tfidf 对每一个 pull request 进行向量空间的表示, 通过计算余弦相似度可以获得任何两个 pull request 的相似度得分, 由此, 当新到来一个 pull request 的时候, 可以找到历史上和该 pull request 语义上最相似的 top k 个历史 pull request. 余弦相似度的计算方法如下公式:

$$\text{similarity}(pr_{new},\ pr_{old})=\frac{v_{new}\bullet v_{old}}{|v_{new}||v_{old}|}$$

其中:

pr_{new}: 新到来的 pull request

pr_{old}: 历史 pull request

v_{new}: 新到来 pull request 的向量表示

v_{old} : 历史 pull request 的向量表示

对新到来的 pull request, 获得与其相似度最高的 top k 个 pull request 之后, 我们将这 k 个 pull request 的评审人作为新到来的 pull request 的评审人候选集合. 并且乘以评论的次数求得每位候选评审人与新到来的 pull request 的相关性得分, 最终返回得分最高的 top 10 个评审人作为评审人推荐的结果.

4 基于时间和影响力因子的评审人推荐

第三部分描述的是用于 pull request 评审人推荐的传统信息检索方法. 这部分则描述了本文提出的基于时间和影响力因子的 pull request 评审人推荐对传统信息检索方法的改进.

4.1 基于时间和影响力因子的候选评审人的得分

本文基于评论的次数、评审时间维度、评审人的影响力因子这三个方面求得每位候选评审人与新到来的 pull request 的相关性得分.

4.1.1 评论次数

评论次数即候选评审人在相关的历史 pull request 中评论了多少次. 评论次数越多, 候选评审人相关性的得分应该越高. 对评论次数我们进行 log 平滑, 公式如下:

$$N_{j,i}=\log(1+n_{j,i})$$

其中:

$n_{j,i}$: 评审人 j 在 pri 下的评论次数

$Nj_{,i}$: 评审人 j 在 pri 下基于评论次数获得的权重

4.1.2 评审时间维度

评审时间因素是我们重点考虑的维度之一, 比如与新到来的 pull request 最相关的 top k 个历史 pull request 中, 1 年前的历史 pull request 和 1 周内的历史 pull request 所占的权重应该是不一样的, 我们认为评审人在一段时间内的兴趣是集中的, 所以最近的 pull request 显然应该赋予更高的权重[11], 我们采用了时间归一化的公式来进行时间维度的权重分配.

$$t_{pr_i}=\frac{timestamp(pr_i)-baseline}{deadline-baseline}\in(0,1]$$

其中:

pr_i : 第 i 个相关的历史 pull request

$timestamp(pr_i)$: pr_i 的提交时间

$baseline$: 训练集中提交最早的 pull request 的提交前一天

$deadline$:训练集中提交最晚的 pull request 的提交当天

但是上述时间归一化会导致最早的历史 pull request 的评审人在该 pull request 即使相关度非常高的情况下, 也会因为上述时间归一化而大大衰减这部分的得分. 所以我们采用线性模型进行时间维度的参数训练, 以获得最终的时间权重计算公式:

$$T_{pr_i}=\alpha*t_{pr_i}+\beta$$

其中 t_{pr_i} 是上述时间归一化的结果. 经过模型的训练, 我们得出 α =0.6, β =0.4 的时候, 返回推荐结果的准确率比直接对时间进行归一化要好得多.

4.1.3 评审人的影响力因子

评审人在 github 项目中的影响力因子同样也是非

常重要的维度, github 中有些很优秀的人, 他们在某些项目领域经验丰富, 为开源社区的项目贡献了很多高质量、优秀的代码, 也是项目的核心成员, 他们在 pull request 的评审中给出的意见也非常有效而准确, 这些杰出的人会吸引很多人 follow 他们. 这样基于 follow 和被 follow 的关系, 我们可以构建 github 中开发者的关系网, 它是一张以开发者为节点的图, 从而利用著名的 pagerank 算法我们就可以获得 github 中任何一位开发者的影响力因子的得分. 但是本文的评审人推荐应用情境是在一个具体项目中的评审人(开发者)的影响力. 因此计算整个 github 中所开发者在整个 github 下的影响力是非常费时而没有意义的. 而如果具体到 1 个项目内使用 pagerank 计算影响力, 由于开发者的所有 follower 可能在不同的项目中都有分布, 所以会造成项目内的 follow 关系图很难确定. 于是本文提出基于 follower 数量来对影响力因子的量化计算方法, 这种方法实际上是对 pagerank 方法的简化, 我们认为 follower 数量越多的人, 他的评审的权重应该比 follower 数量少的成员权重要高. 具体的计算公式如下:

$$F_i = 1 + \frac{N_{(followers\ of\ i)\in P}}{N_{MAX\_followers}} \in [1,2]$$

其中:

F_i: 评审人 i 的影响力;

P: 项目 P, 评审人推荐的作用范围就是某个特定的项目 P;

$N_{(followers\ of\ i)\in P}$: 评审人 i 在项目 P 中的 follower 的个数;

$N_{MAX\_followers}$: 项目 P 中, follower 数的最大值.

该公式的右侧相当于对 follwer 的数目进行了归一化, 如果一个开发者在项目 P 中 follower 数为 0, 则他在项目 P 中的影响力是最低的 1; 开发者在 P 中的 follower 数越多, 则他在 P 中的影响力越大, 影响力最大可以为 2.

4.2 候选人的得分计算

结合评论的次数、评审时间维度、评审人的影响力因子这三个方面, 最终的候选评审人与新到来的 pull request 的相关性得分计算公式如下:

$$Score_j = \left(\sum_{i=1}^{k} similarity(pr_{new}, pr_i) \times N_{j,i} \times T_{pr_i}\right) \times F_j$$

其中:

$Score_j$: 候选评审人 j 的最终得分

k: 与新到来的 pull request 最相关的 k 个历史 pull request

pr_{new}: 新到来的 pull request

pr_i : 第 i 个相关的历史 pull request

$N_{j,i}$: 评审人 j 在 pr_i 下评论的次数的权重

T_{pr_i}: 第 i 个相关的 pull request 评论的时间维度所获得的权重

F_j: 候选评审人 j 的影响力因子

5 实验

5.1 数据集的选取

我们选取了 github 中比较热门的 10 个项目用于实验, 他们是 scikit-learn、scala、rails、swift、ipython、jquery、akka、node、xbmc 和 homebrew. 这些项目都拥有大于 1500 个 pull request, 数据集足够充分用于进行实验. 同时我们做了如下过滤:

1) 首先对每个项目抓取最近的 1500 条 pull request 的数据, 对项目中的 pull request 标题和描述信息,去除停用词并进行词干还原之后, 将那些标题和描述部分的单词总和小于 10 个词的 pull request 去掉, 因为小于 10 个词会导致描述该 pull request 的信息量过少.

2) 停用词并进行词干还原之后, 将那些标题和描述部分的单词总和小于 10 个词的 pull request 去掉, 因为小于 10 个词会导致描述该 pull request 的信息量过少.

3) 然后, 我们去掉评审人少于 2 个的 pull request, 因为至少 2 个评审人进行评审才能使评审可信[14,15].

4) 对这 1500 个 pull request 过滤之后, 用最新的 1000 个 pull request 作为最终的数据集. 并划分训练集和测试集, 前 900 个作为训练集, 后 100 个作为测试集.

候选评审人过滤: 训练集中的候选评审人如果只对 1 个 pull request 进行过评审, 则将这个候选评审人排除掉. 我们认为候选评审人更有可能是进行过多次评审的开发者, 而不是偶然对 1 个 pull request 进行过评审的开发者. 因为 github 的开发者以及评审人都希望为自己所在的项目做出贡献, 所以他们会自发的利用自己在项目领域内的经验, 不断的多次对项目做出

贡献, 所以我们不难发现大型项目中, 大多数评审人都会对多个 pull request 进行多次的评审, 而核心成员担任评审人角色时, 他自主发起评审数目甚至会达到数十甚至上百个. 评审人经过上述的过滤之后, 通过要余弦相似度的计算, 可以得到与新到来的 pull request 最相关的历史 pull request, 在这些历史 pull request 下进行评审的开发者于是构成了候选评审人的集合.

5.2 评估方式

本文采用准确率(precision)和召回率(recall)对评审人推荐结果进行评估, 对返回的 10 个推荐结果, 分别计算 top 1 到 top 10 的准确率和召回率. 计算方法如下公式:

$$Precision = \frac{|Rec\_Reviewers \cap Actual\_Reviewers|}{|Rec\_Reviewers|}$$

$$Recall = \frac{|Rec\_Reviewers \cap Actual\_Reviewers|}{|Actual\_Reviewers|}$$

5.3 实验效果对比

这部分我们对比了三组实验的结果, 分别是 baseline、传统的信息检索方法(IR-base)的效果和本文提出的优化后的考虑了影响力和时间因子的信息检索方法(IR-optimal)的效果.

5.3.1 实验的 baseline

Github 中的大部分 pull request 是由各项目组的核心成员评审的. 因为他们对项目更加了解, 而且具备丰富的编程经验以及项目相关经验. 所以他们经常能给出重要、准确而且有效的评审意见, 并帮助 pull request 提交者改善他的代码. 为了评估本文提出的实验方法的效果, 我们采用了对比的 baseline, 在训练集中评审了 pull request 数最多的 top k 个开发者成为活跃评审人集合, 每个新到来的 pull request 都被分配给活跃评审人集合. 这些最活跃的开发者所获得的推荐效果就构成了我们的 baseline.

5.3.2 推荐效果评估

三组实验的准确率、召回率曲线如图 2 所示, 我们描绘了 10 个项目中 top 1 到 top 10 的平均准确率以及平均召回率曲线.

由图 2 可以看出随着推荐的人数增多, 准确率呈下降趋势, 召回率呈现上升趋势. 图中显示传统信息检索方法以及本文考虑了影响力和时间因子的信息检索方法的推荐效果明显好于 baseline.

相对于传统信息检索方法而言, 本文提出的基于影响力因子和时间维度衰减的信息检索方法在 top 1 到 top 4 上效果提升比较明显. 我们列出了 top 1 到 top 5 的效果对比表格. 如表 1 所示. 我们的方法在 top 1 上准确率达到 68%, 比传统信息检索方法提升了 8%, 召回率达到 18%.

此外, 我们通过计算本文提出的优化后的信息检索方法的 F 值, 发现在 top 4 的时候 F 值最大, 达到了 0.514. 而且曲线表明推荐 top 5 到 top 10 的准确率下降非常明显, 这和我们的预期也相符, 实际上大多数 pull request 的评审人一般是少于 5 个的, 假设一个 pull request 只有 3 位评审人, 而我们推荐系统推荐 top 10 位评审的时的准确率可想而知是会大大降低的. 前 5 个准确率能得到保证是因为基于信息检索的推荐方法效果比较好, 但正因为实际评审人少于 5 个, 所以 top 5 到 top 10 的候选评审人相关性得分会比较低, 造成这部分的推荐不能做到 top1 到 top 4 那么准确, 从而造成 p@k 的准确率下降明显. 所有方法在 top 1 到 top 4 上召回率上升速度比较快. 当我们推荐 top 10 个评审人时, 三种方法的召回率都达到了 78%, 说明三种方法当推荐到 10 位候选评审人时, 都能保证比较好的召回.

此外, 通过深入研究我们发现, pull request 推荐结果会倾向于推荐那些历史上进行过多次评审的活跃的评审人, 这些人在开源项目中具有很多 follower, 这与实际的 github 社区的评审情况相符合, 也从侧面印证了 baseline 的有效性. 但本文提出的方法还考虑了 pull request 的之间的相关性, 评审人的影响力因子, 以及评审的时间维度衰减, 使得最终的推荐准确率相对于传统信息检索方法以及 baseline 都有比较好的提升.

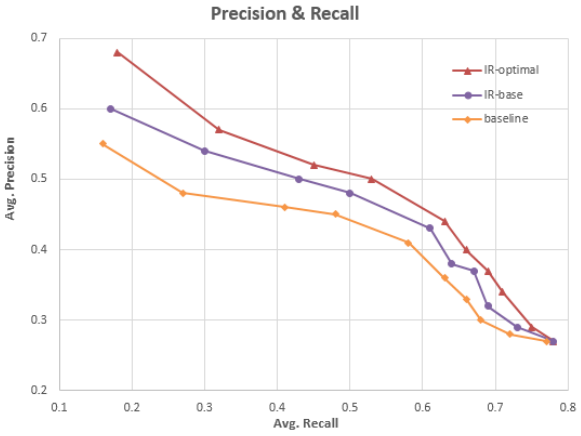

图 2 评审人推荐的准确率、召回率曲线

表 1 三组实验 top 1 到 top5 的效果对比

Number of reviewer	baseline		IR-base		IR-optimal	
	准确率	召回率	准确率	召回率	准确率	召回率
Top 1	0.55	0.16	0.6	0.17	**0.68**	0.18
Top 2	0.48	0.27	0.54	0.3	**0.57**	0.32
Top 3	0.46	0.41	0.5	0.43	**0.52**	0.45
Top 4	0.45	0.48	0.48	0.5	**0.5**	0.53
Top 5	0.41	0.58	0.43	0.61	**0.43**	0.63

6 结论和未来工作

在本文中, 首先我们将应用于 bug 分类的信息检索方法, 拓展应用于 pull request 的评审人推荐中. 对新到来的 pull request, 基于标题和描述信息, 获得和该 pull request 最相关 top k 个的历史 pull request. 这 top k 个历史 pull request 下的评审人构成了我们的候选评审人集合. 然后, 我们基于评审人的评论次数、github 评审人的影响力因子以及基于 pull request 评审时间维度的衰减计算每位候选评审人相对于新到来的 pull request 的相关性得分, 最终返回得分 top 10 的评审人作为评审人推荐结果.

本文主要讨论了 github 评审人的影响力因子量化度量方法, 以及基于 pull request 评审时间维度的衰减. 并将这两者结合到信息检索方法中, 得出优化后的信息检索方法用于评审人推荐. 并对 baseline、传统信息检索方法、优化后的信息检索方法进行效果评估, 实验表明, 优化后的信息检索方法推荐的准确率提升比较明显, 尤其是 top 1 到 top 4 的推荐效果明显优于其他两种方法. 未来, 我们会对评审人的影响力因子量化进行更深入的探索, 分析开发者之间的有效评论, 基于语义评估评审人的评论对该 pull request 所做的贡献. 并构建开发者之间的评论关系网, 由此获得影响力因子评估的更精确的量化方法, 使得本文提出的信息检索方法的准确率和召回率效果进一步提升.

参考文献

1 Begel A, Bosch J, Storey MA. Social networking meets software development: Perspectives from GitHub, MSDN, stack exchange, and TopCoder. Software IEEE, 2013, 30(1): 52–66.

2 Gousios G, Zaidman A, Storey MA, et al. Workpractices and challenges in pull-based development: The integrator's perspective. 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE). IEEE Computer Society. 2015. 358–368.

3 Gousios G, Bacchelli A. Work practices and challenges in pull-based development: The contributor's perspective. IEEE Software, 2015, 32(1).

4 Yu Y, Wang H, Filkov V, et al. Wait for it: Determinants of pull request evaluation latency on GitHub. 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR). IEEE. 2015. 367–371.

5 Vasilescu B, Yu Y, Wang H, et al. Quality and productivity outcomes relating to continuous integration in GitHub. Proc. of the 2015 10th Joint Meeting on Foundations of Software Engineering. ACM. 2015. 805–816.

6 Cubranic D, Murphy GC. Automatic bug triage using text categorization. Seke: Sixteenth International Conference on Software Engineering & Knowledge Engineering. 2004. 92–97.

7 Canfora G, Cerulo L. Supporting change request assignment in open source development. Proc. of the 2006 ACM Symposium on Applied Computing. ACM. 2006. 1767–1772.

8 Linares-Vásquez M, Hossen K, Dang H, et al. Triaging incoming change requests: Bug or commit history, or code authorship? 2012 28th IEEE International Conference on Software Maintenance (ICSM). IEEE. 2012. 451–460.

9 Tamrawi A, Nguyen TT, Al-Kofahi JM, et al. Fuzzy set and cache-based approach for bug triaging. Proc. of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. ACM. 2011. 365–375.

10 Yu Y, Wang H, Yin G, et al. Who should review this pull-request: Reviewer recommendation to expedite crowd collaboration. 2014 21st Asia-Pacific Software Engineering Conference (APSEC). IEEE. 2014, 1. 335–342.

11 Anvik J, Hiew L, Murphy GC. Who should fix this bug? Proc. of the 28th International Conference on Software Engineering. ACM. 2006. 361–370.

12 Bhattacharya P, Neamtiu I, Shelton CR. Automated, highly-accurate, bug assignment using machine learning and tossing graphs. Journal of Systems and Software, 2012, 85(10): 2275–2292.

13 Jeong G, Kim S, Zimmermann T. Improving bug triage with bug tossing graphs. Proc. of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. ACM. 2009. 111–120.

14 Sauer C, Jeffery DR, Land L, et al. The effectiveness of software development technical reviews: A behaviorally motivated program of research. IEEE Trans. on Software Engineering, 2000, 26(1): 1–14.

15 Rigby PC, Bird C. Convergent contemporary software peer review practices. Proc. of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM. 2013. 202−212.