利用文本相似度比较，定位和bug report相关联的源文件

高也(M201973167)，徐志信(M201973166)。

1. 背景介绍。

软件的质量对于一个软件项目的成功非常重要。虽然我们已经有了很多提高软件质量的方法(比如软件测试、软件审查、静态检查)，现实中的软件系统还是会有很多缺陷(bugs)。对于一个大型的、不断迭代演变的软件系统，软件工程团队会收到大量的错误报告(bug report)。举例来说，Eclipse在2009年收到大约4414个bug report。

软件工程团队一旦收到并确认一个错误报告，他们需要定位和该错误相关联的源文件。然而，当源文件数量特别多时，由人工去完成这项定位工作非常耗时。这进一步导致了软件维护成本高昂。

在这篇报告中，我们做出了一个“bug定位器”(BugLocator)，它可以从大量源文件中找到和bug report相关联的源文件。BugLocator的原理是一个改良版的空间向量模型(revised Vector Space Model, namely rVSM)，它基于bug report的文本对所有的源文件按相似度大小进行排序。和传统的空间向量模型相比，rVSM还考虑了文本的长度，取得了更好的效果。除此之外，我们还考虑到，文本相似的bug report，它们相关联的源文件很可能也是相同的。我们在Eclipse的98个bug report上做了实验，实验结果表明BugLocator是可以有效工作的。

我们相信BugLocator可以帮助软件工程团队更好地对bug report进行定位，从而降低软件的维护成本和提升客户的使用体验。

1. 算法原理。

近年来，一些研究人员使用信息抽取技术来自动搜索和bug report相关联的源文件。它们将原始的错误报告视为一个查询(query)，然后根据bug report和源文件的文本相似度对所有的源文件进行排序。开发者再按顺序查看源文件，直到将bug修复。这种方法就是“基于信息抽取的bug定位法”。这种方法不需要程序的执行信息，它们仅仅根据原始的bug report来定位和bug相关联的源文件。

接下来，我们将使用一个简单例子来说明BugLocator的原理。图1所示的Eclipse 3.1的一份真实的bug report，它的ID是80720。当开发者收到这份报告之后，他们需要从数以万计的Eclipse源文件中找到相关联的源文件，以便修复这个bug。我们可以发现，这份bug report中的很多词汇类似于pin, console, view, display。因此，我们可以推断出这个bug和控制台视图(console view)有关。在Eclipse 3.1的源文件中，有一个源文件是ConsoleView.java。它的代码中出现了很多类似的词汇。如图1所示，bug report 80720和源文件ConsoleView.java在文本上可以很好地匹配。

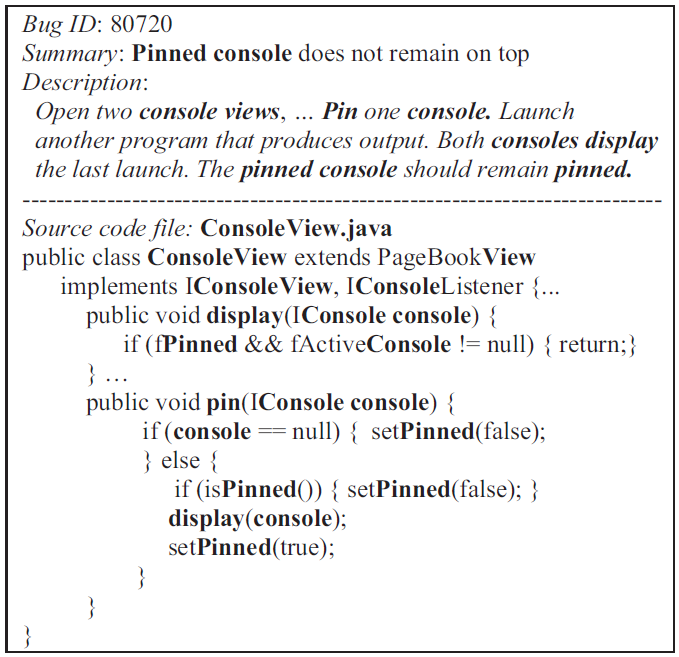


图1：一份bug report和与它相关联的源文件。

我们可以把bug report和源文件都视为文本文件，然后计算他们之间的文本相似度(rVSMScore)。在计算完全部源文件和该bug report的文本相似度后，我们根据相似度对全部源文件由大到小进行排序。软件开发者按顺序依次排查源文件，直到找到和bug report相关联的文件为止。

通过这种方法，开发者可以快速找到和bug report相关联的源文件。因此我们的目标是，让和bug report相关联的源文件在排序中尽可能靠前。

1. 算法步骤。

3.1 建立bug语料库和code语料库。

对每一个bug report和源文件进行词法分析，将它们转化为由词汇组成的词向量。每个词向量包含该文档中的所有词汇，且两两各不相同。这个过程我们需要注意以下几点。(1) 源文件中一些词汇，比如编程语言中常见的关键词(int, char, double)，我们需要将它们去除掉。同时，一些常见的冠词(a, the)，我们同样把它们删除掉。(2) 我们注意到，源文件的变量名中有很多由两个英文单词组成连接而成。比如，变量名TypeDeclaration由单词type和declaration连接而成，变量名isCommitable由单词is和Commitable连接而成。对于这种变量名，我们将它们分解成一个个独立的词汇。(3) 此外，很多词汇有相同的词根。比如，delegating，delegate，delegation的词根都是delegat。我们使用Porter Stemming算法将所有单词还原为它的词根。

比如，bug语料库中的bug report 100040将会转变为如下词向量：

(averag, time, mirror, icon, includ, ilc, menu, perspect, test, numer, swt, ui, os, ms, number, window, increas, show, toolbar, run, perform, eclips, imagelist, make, elaps, process, singl, due, debug, build)

比如, code语料库的源文件org.eclipse.swt.accessibility.ACC.java将会转变为如下词向量：

(client, access, scrollbar, radiobutton, listitem, acc, delet, deprec, multipl, tablecel, state, press, tabitem, insert, tablerowhead, focus, contain, combobox, dialog, tooltip, pushbutton, select, label, expand, separ, sizeabl, constant, hottrack, offscreen, treeitem, childid, multiselect, progressbar, menu, text, slider, readonli, tablecolumnhead, tree, menubar, checkbutton, check, defin, list, role, area, link, tablecolumn, sinc, xd, xc, xb, xa, us, invis, window, toolbar, object, tabfold, tabl, menuitem, busi, collaps)

3.2 建立词典。

在完成步骤1后，我们遍历bug语料库和code语料库，为其中的所有单词加上序号(index)，从而建立词典。形成的词典如表1所示(仅列出前10行)。

|  |  |
| --- | --- |
| fffffff | 0 |
| stepit | 1 |
| wide | 2 |
| cbn | 3 |
| fusesgetlasterror | 4 |
| ostoop | 5 |
| wider | 6 |
| setvers | 7 |
| getlineindex | 8 |
| trackpo | 9 |
| ... | ... |

表1：词典示意图。

3.3 为语料库中的每个单词计算权重。

我们使用tf-idf公式计算每个单词的权重。每一个单词的权重w由这个单词的词频(term frequency, namely tf)和逆向文件频率(inverse document frequency, namely idf)共同决定。在传统的空间向量模型(Vector Space Model, namely VSM)中，词频(tf)和逆向文件频率(idf)的定义如下：

 (1)

其中ftd代表一个单词t在文本d中出现的频率，#term代表文本d包含的单词的总量，#docs代表语料库中文本的总量，nt代表含有单词t的文本的数量。这些年来，tf(t, d)公式有了很多变种，我们发现用以下变种可以取得更好的结果：

 (2)

因此，我们计算每个单词权重的公式为：

 (3)

 (4)

 (5)

由此，我们可以得到bug语料库中每个bug report的向量分量权重和code语料库中源文件的向量分量权重。

3.4 计算每个源文件的。

我们把所有的源文件视为语料库，把每一个bug report视为一个查询。我们计算bug report和所有源文件的余弦相似度(cosine similarity)，并按相似度由大到小进行排序。在传统的空间向量模型(VSM)中，bug report和源文件的余弦相似度的定义如下：

CosineSimilarity(bug, code) = (Vbug • Vcode) / (|Vbug| \* |Vcode|) (6)

其中Vbug代表bug语料库中每个bug report的向量分量权重组成的向量，Vcode代表code语料库中每个源文件的向量分量权重组成的向量。其中Vbug • Vcode代表两个向量的内积，|Vbug|和 |Vcode|代表两个向量的大小。

考虑到Vbug和Vcode的维数不一定相同，我们采用一下方法重新定义它们的“内积”。找到Vbug和Vcode中相同的单词，将它们的权重相乘，最后在加起来算总和。即：

Vbug • Vcode = Σbug∩code weight(bug\_word) \* weight(code\_word) (7)

经典VSM在文档长度短时表现很好，但是当文档长度变大时表现不好。这是因为当文档变长时，余弦相似度的值会变小。根据前人的研究，更长的文档有更大的可能性出bug，因此更长的源文件在全体源文件的排序中应该更为靠前。我们定义一个文本长度函数g，并把它作为乘法因子放入rVSMScore的定义式中:

 (8)

rVSMScore(bug, code) = g(CodeLength) \* CosineSimilarity(bug, code) (9)

等式(8)是一个逻辑函数，它可以使更长的文档在计算中获得更高的权重。我们使用等式(8)来计算每个文档的g(#term), 其中#term是指每个文档中词汇的数量。

在等式(8)中，我们将#term的值正则化后作为指数函数的输入值。其中正则化函数N(x)的定义为，设X是一组数的集合，xmax和xmin是这组数中的最大值和最小值，则对于集合中的任意x，它的正则化定义为：

 (10)

综合上述所有等式，当给定一个bug report, 我们可以计算出它和所有源文件的相似度，我们将这个相似度记为。

3.5 计算每个源文件的SimiScore值。

对于一个bug report, 我们同样可以参考之前已经修复的bug, 来猜测它可能和哪些源文件相关联。这种做法基于一个假设，那就是文本更相似的bug, 它们关联文件相同的概率更大。

如图2所示，我们构造一个3层图来说明这种方法。最顶层（第一层）只有一个结点，表示未知关联源文件的bug report，记为B。第二层的结点表示之前修复的bug，也就是说第二层结点表示的bug report相关联的文件是我们已知的。 第一层结点(未知关联源文件的bug report)和第二层结点(已知关联源文件的bug report)的连线代表bug report之间的文本相似度。第三层结点代表程序所有的源文件。第二层结点和第三层结点的连线代表每一个已知关联源文件的的bug report所相关联的源文件。

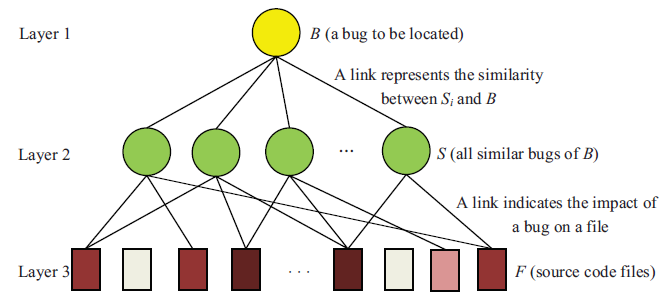
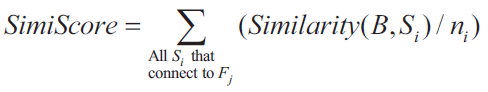


图2：bug-源文件关联图。

第二层每一个结点的权重是“未知关联源文件的bug report”和每一个“已知关联源文件的bug report”的文本相似度。我们同样用余弦相似度来计算，如等式(11)所示。第三层每一个结点的权重是“未知关联源文件的bug report”和每一个源文件的相似度。它的计算方法如等式12所示。

CosineSimilarity(bug, bug) = BugVector \* BugVector / (|BugVector| \* |BugVector|) (11)

 (12)

其中代表某个源文件，代表和相关联的bug report, 代表和相关联的bug report的数量。

我们利用等式(12)为每一个源文件计算出它和bug report B的文本相似度，我们将这个相似度记为。

3.6 合并和。

通过bug report和源文件的相似度计算，我们得到了。通过“未知关联源文件的bug report”和“已知关联源文件的bug report”的文本相似度计算，我们得到了。我们首先利用等式(10)将它们正则化。

我们接下来将正则化后的和结合起来：

 (13)

其中是一个介于0和1之间的参数。是和正则化后的加权和。