A Neural Model for Generating Natural Language Summaries of Program Subroutines

——论文复现报告

程龙

摘要

对生成代码的自然语言描述的研究被称为源代码摘要，摘要最常见的目标之一是程序中的子程序，可以帮助程序员理解子程序在程序中的作用。传统的代码摘要方法，依赖于启发式和人工创建的范本；近年来，以神经机器翻译为基础的数据驱动方法迅速发展，但是几乎所有的技术都依赖程序有完善的内部说明；本文的提出的神经模型，将代码的单词与从 AST 得到的代码结构结合，该模型将代码中的文本视为单独的输入，这样模型能够独立于代码文本学习代码结构。即使没有提供内部说明，模型也能够提供连续的摘要。

关键词：代码摘要；神经机器翻译；抽象语法树

1 引言

对生成代码的自然语言描述的研究被称为源代码摘要，其主要工作集中在子程序摘要的生成上，过去几年，基于内容选择和句子模板的研究取得了重大进展。然而，正如进来的研究表明，这些技术在很大程度上已经不如基于大数据输入的 AI 方法了。基于 Al 的代码摘要的绝大多数工作的灵感来源于自然语言处理研究社区的神经机器翻译（NMT）。这种 NMT 系统将一种自然语言转换为另一种，它通常被认为是序列到序列（seq2seq）学习。在软件工程研究中，机器翻译可以被理解为源代码摘要：子例程中的单词和序列作为输入，而所需的自然语言摘要作为目标序列。NMT 在代码摘要中的应用在各种应用中都显示出了巨大的优势。然而传统的工具如 JavaDoc[1]和 Doxygen 等仍然需要程序员花大量时间在写文本和例子上；之前的技术大都是依靠程序中有以关键字、注释等形式的较为完善的说明，这是因为在传统的 NMT 中，输入和输出必然是有一定的关联。但是在软件中，程序代码中的单词不一定与代码行为相关联。子程序的行为取决于定义控制流、数据流的关键字和序列的结构。这一不同是提升智能化软件工程效率的一大障碍[2]。

简单来讲，近年来，以神经机器翻译为基础的数据驱动方法迅速发展，但是几乎所有的技术都依赖程序有完善的内部说明；本文提出一种神经模型，将代码的单词与从 AST 得到的代码结构结合，该模型将代码中的文本视为单独的输入，这样模型能够独立于代码文本学习代码结构。即使在没有提供内部说明的情况下，模型也有较好的效果。

2 相关工作

本部分包含了模型的基本背景知识，以及代码摘要方面前人的相关工作。

2.1 源码摘要工作

源码摘要相关工作可被大体分为 AI/数据驱动和启发式/模板驱动。

2.1.1 数据驱动

关于数据驱动技术，最近的相关工作是 Hu[3]等人的文章，提出了使用 AST 来注释源码中的单词，然后使用注释过的表示形式作为 seq2seq 神经模型的输入。模型本身是一个现有的编码器-解码器结构，主要的改进是 AST 注释的表示形式——基于结构的遍历（SBT）。SBT 实质上是一种 AST 的扁平化技术，确保代码中的单词能够和 AST 节点类型联系起来。例如：代码 requestl.remove(id) 变为：

$(MethodInvocation$

$(SimpleName\_request)SimpleName\_request$

$(SimpleName\_remove)SimpleName\_remove$

$(SimpleName\_id)SimpleName\_id$

$)MethodInvocation$

这使单词能与代码上下文结构结合，在上例，是一个方法调用结点。SBT 表示也是本文提出模型的一个重要基线。本实验的 BLEU 值与其没有直接的可比性，因为 Hu 的实验依据函数划分数据集（同一个项目的函数可能会分别出现在训练集和测试集当中），但是本论文的数据集是根据项目划分（一个项目中的方法只会在三种数据集中的一个出现）。同时，该项目还做了其他的预处理，如自动生成代码的去除（避免上述情况）。其他 AI/数据摘要自动生成的相关工作包括：Hu 等人[4]做的基于 API 调用生成摘要以及 Iyer 等人[5]的 CODE-NN。与之前方法相似的，生成代码的特定表示形式并将其作为输入送入现有的的 seq2seq 模型。

在代码摘要之外的相关工作有：Jiang 等人[6]和 Loyola 等人[7]生成代码变化的描述；Allamanis 等人[8]从子程序体预测子程序的名字；Dda 等人从源代码生成伪代码；Yin 等人[9]、Movshiovitz 等人[10]、Allamanis 等人[11]关注于基于公共数据集生成代码片段的注释。Gu 等人[12]已经证明使用神经网络模型用于源代码搜索的可能性，这也是由公共数据集促进的另一项越来越受欢迎的任务。值得注意的是，Bahdanau 等人[13]提出的带有注意力机制的编码器-解码器 seq2seq 模型是以上大多相关工作的核心。

2.1.2 启发式/模板驱动

Haiduc 等人[14]是最先做生成源码的文本摘要的，并且是第一个提出“代码摘要”术语的，这些早期的方法通过计算矩阵（TF/TDF）中出现次数最多的几个关键词提取出摘要。有相关自然语言生成的研究表明[15]，由于数据驱动技术灵活和简单（不用费力去构建模板），如今已经很大程度取代了基于模板的技术。

2.2 神经网络机器翻译

大多数神经机器翻译的主力是编码-解码器结构；在该结构中，至少有两个循环神经网络（RNNs）。第一个，叫编码器，将任意长度的序列转换为指定长度的单个向量表示；另一个，叫解码器，将编码器送来的向量转换成另外一个任意长度的序列（输入输出序列不是同一种语言）。该结构一次预测一个单词，解码器通常不会一次预测一整句话。网络接收三种数据：1）整个输入序列；2）已经输出的部分；3）正确的下一个单词。

在推理期间，训练模型输入一个序列，用来预测输出句子中的第一个单词；然后输入序列与第一个预测的单词一同送入模型，解码器输出句子第二个单词的预测……，直到编码器预测句子结束的标

签。

上述方法的问题是，编码器每次都要负担着生成一个适合每次输出预测的向量形式，事实上，在一次特定的预测中，输入句子中的某些单词是比其他的要重要的多的，这就是“注意力机制的编码-解码网络”的启发。实际上，并非上述的一个单独的输入序列的向量表示，而是一个注意力装置被放在编码器和解码器之间，注意力装置在每次的预测中，接收编码器的输入，在句子中有几个单词，就生成几个对应位置的向量。注意力装置每次选择一个向量使用，这样，解码器就能根据输入的不同位置进行预测。本文模型便是建立在该方法上的。

3 本文方法

3.1 模型概述

本文提出的模型实质上是一个带有注意力机制的编码器-解码器结构，带有两个解码器：一个处理 code/text 数据，一个处理 AST 数据。组合不同数据源的先例主要来自图像识别（例如合并卷积带有标签列表的图像输出），本着尽可能保持简单的原则，本文为编码器使用了相同大小的嵌入层和循环层，将每个编码器的注意机制输出连接在一起，如下图所示：

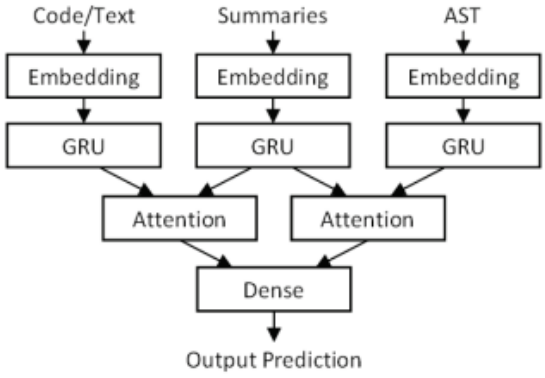

图 1: 模型示意图

3.2 语料库准备

从 Sourcerer 下载项目代码（超过 5100 万个 java 方法）[16]，使用工具包提取 java 方法存入 SQL 数据库。然后筛选 JavaDoc 注释过的（有/**），文章只是用被十位 JavaDocs 的注释，因为假设注释的第一句会是方法的行为摘要。通过找第一节或者第一行提取出其中的第一句话，下一步使用 langdetect 库去除不是英语的注释，在这之后只剩 400 万左右的 java 方法了。

另外一个重要的问题是自动生成的代码，因为这些代码非常相似，如果特征相似的代码被放进了训练集和测试集，模型的学习便会很容易，会同时使在像 BLEU 上评估时分数虚高，而真实表现不行。解决方法也很简单，移除文件中带有“generated by”字样的方法。在这之后，只剩大约 200 万个 java 方法。

被移除的这 200 万个方法大概是从 10 万个不同示例中复制过来的，但是自动生成的代码注释同样重要，因此我们有将这 10 万个不同的示例的副本又添加进数据集，并且确保只在训练集中，最终生下了一个有 210 万方法的数据集。本文的其他预处理步骤遵循了许多软件工程论文，把代码和注释分开了在驼峰大小写和下划线上，删除了非字母字符，并设置为小写，没有执行词干分析。本次复现使用的是原文献作者提供的现成的数据集。将数据集划分为训练集、交叉验证集、测试集，根据项目

划分，预期比例为 9：0.5：0.5，一个项目中的方法不会被分开。由于项目中方法数量的不同，最终方法比例为：9.1：0.48：0.42。

为了获得 AST 数据，首先使用 srcml 提取出每个方法的 XML 表达，然后构建工具来将 XML 表达转换成扁平化的 SBT 表示，生成 hu 等人描述的 SBT 格式的输出，最终创建了自己的 SBT 表示，完整保留代码结构，但是又用特殊的 <OTHER> 序列替换了所有的单词（除了官方的 JavaAPI 类的名字），我们称这种处理后的表示为 SBT-AO，我们使用这种修改来模拟只有 AST 能被提取的情况。

从这个 java 方法的语料库中，我们创建了两个数据集：

- 标准数据集，包括每个 java 方法的三个元素：预处理的 java 源码、预处理的注释、Java 代码的 SBT-AO 表示。
- 挑战数据集，包括 java 方法的连个要素：预处理的注释、Java 方法的 SBT-AO 表示。

3.3 评估方法

通过性能指标 BLEU 的得分反映模型的好坏，也与 NMT 中的标准实践保持一致。BLEU 是预测摘要和参考摘要之间文本相似性的度量。除了 BLEU1 到 BLEU4 之外，还有综合 BLEU 分数（BLEUn 是 n 长度子序列与整个摘要句子的相似性的度量）。在实现中使用了 nltk.translate.bleu_score 。

3.3.1 实验问题

1. 我们的方法和其他方法在标准条件下的表现有什么不同？假定有内部文档；
2. 我们的方法与其他方法在挑战条件下有什么不同？假定只有 AST。

3.3.2 基线标准

为回答问题一，三个比较 (本文献提出的方法被称为：ast-attendgru)：

1. 一般的注意力编码解码模型（attendgru）；
2. Hu 等人提出的 SBT 方法；
3. Iyer 等人的 codenn

4 复现细节

4.1 核心代码

首先是三个输入层，分别对应代码/原文序列、注释序列、扁平化后的 AST 序列。规定输入序列的长度：代码和 AST 限制 100 词，注释限制 13 词，短的补零长的截去。

```
txt_input = Input(shape = (self.txtlen,))
com_input = Input(shape = (self.comlen,))
ast_input = Input(shape = (self.astlen))
```

该模型从一个相当通用的结构开始，包含了为每种输入类型设置的嵌入层。嵌入层输出的形状是（batch_size, txtvocabsize, embdims）；如：（200，100，100）表示在每个批处理的 200 个示例中，每个

有 100 个单词，每个单词用一个长度为 100 的嵌入层向量表示。（作者发现两个独立的嵌入层比统一的嵌入空间性能更好）；

ee = Embedding(output_dim=self.embdims, input_dim=self.txtvocabsize)(txt_input)

se = Embedding(output_dim=self.embdims, input_dim=self.astvocabsize)(ast_input)

接下来一层是带有 256 个 rnndims 单元的 GRU 层来编码 AST，同时使用 CuDNNGRU 来提升训练速度。通过 return_state flag 标签我们能够知道 AST 编码器最终的隐藏声明；使用 return_sequences flag 标签，是因为我们想要每个单元格的状态而不只是最后的状态，以此来为注意力机制服务。

ast_enc = CuDNNGRU(self.rnndims, return_state=True, return_sequences=False)

astout, sa = ast_enc(se)

代码/文本 GRU 的 GRU 层与 AST 的 GRU 层工作方式类似，只是代码/文本的 GRU 层同时以 AST 的 GRU 层的隐藏输出作为其中一个输入。其效果与我们简单地将输入连接起来类似，但同时该方法又能 1) 我们保持独立的嵌入层空间，2) 允许注意力以不同的方式集中在每个输入上，而不是跨输入类型，3) 我们确保一个输入不会被另一个输入类型的过长序列截断，4) 为进一步处理“保持大门敞开”，例如通过卷积层对一种输入类型有利而对另一种无效。

txt_enc = CuDNNGRU(self.rnndims, return_state=True, return_sequences=True)

txtout, st = enc(ee, initial_state=sa)

对于每一批输入，textout 的张量通常都是一个 rnndims 长度的向量表示，shape 为（batch_size, rnndims），但是，因为我们规定 return_sequences 为 true，则 encout 的 shape 为 (batch_size, datvocabsize, rnndims)，这是一个针对于句中每个单词的向量，所以我们会看到该向量会随着序列中的单词变化而变化。return_state 标志表示我们得到了来自最后一个单元的 runndims 向量——st，用作解码器的初始化。其中解码器为：跟在一个循环层后的特定的嵌入层；RNN 最终的 code/text 状态被送入其中。

de = Embedding(output_dim=self.embdims, input_dim=self.comvocabsize)(com_input)

dec = CuDNNGRU(self.rnndims, return_sequences=True)

decout = dec(de, initial_state=st)

下一步是 code/text 注意力机制，取解码器和 code/text 编码器对输出作点积；如：decout：形状为（batch_size, 13, 256），txtout：形状为（batch_size, 100, 256）；对第二个轴作点积，得到张量形状为（batch_size, 13, 100）

txt_attn = dot([decout, txtout], axes=[2, 2])

txt_attn = Activation(’softmax’)(txt_attn)

结果便是，解码器中的序列的 13 个位置如今被长度为 100 的向量表示，这些向量中的每个元素反映了解码器和编码器序列中元素的相似程度。也就是说，张量中的的元素反映了解码器输出与编码器输出是怎么相似的。13 个输入位置中每个位置的 100 长度矢量反映了给定的输入位置与输出中的位置相似（应该“注意”）的程度，然后使用 softmax 给这 13 个向量（放大影响），使注意力机制会注意更相似的输入向量。

使用注意力向量为 code/text 输入创建上下文向量。使用注意力向量扫描编码器向量，以此能够将注意力放在输入的特定位置来产生对应的输出。结果是一个 context 矩阵，对每一个输出语句中的元

素，都有一个向量与之对应。

txt_context=dot([txt_attn, txtout],axes=[2, 1])

我们仍需要结合 code/text、AST、和解码器序列信息，因为我们每次只送入一个单词。在此我们有的是两个上下文矩阵（batch_size, 13, 256）和一个解码器输出矩阵（(batch_size, 13, 256），结合形成一个张量（batch_size, 13, 768）。

context = concatenate([txt_context, ast_context, decout])

接下来 TimeDistributed 层为上下文矩阵中的每个向量提供一个密集层。结果是解码器序列中每个元素的一个 rnndims 长度矢量（解码器输出序列 13 个位置每个 256 的向量）。

out = TimeDistributed(Dense(self.rnndims, activation=”relu”))(context)

然而，我们只需要输出一个单词，即序列的下一个单词，最终我们只需要一个单独的长度为 comsvocabsize 的输出向量。因此，扁平化（13, 256）的矩阵，成为（3328）的向量，然后使用 dense 输出层，长度为 comsvocabisze，应用 softmax。

out = Flatten()(out)

out = Dense(self.comvocabsize, activation=”softmax”)(out)

结果就是一个输入为 code/text、AST、注释序列，输出为注释序列的下一个单词的模型。

model = Model(inputs=[txt_input, com_input, ast_input], outputs=out)

（以上代码参考文献内容实现）

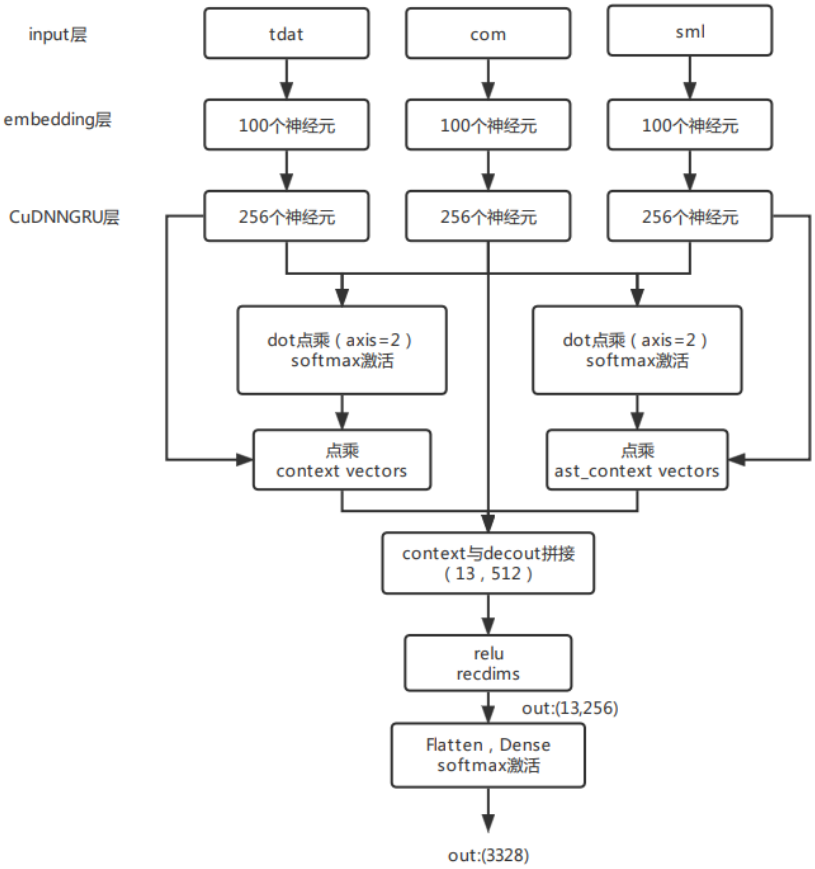

图 2: 模型实现示意图

4.2 实验环境搭建

本次复现使用浪潮人工智能平台，环境配置为 Ubuntu 18.04，Python 3.6, Keras 2.2.4, TensorFlow 1.15. 以及其他一些必要的 Python 库。

4.3 代码运行说明

获取文献提供数据集后，解压，创建如下三个文件夹：/scratch/funcom/data/outdir/models/ ，/scratch/-funcom/data/outdir/histories/，/scratch/funcom/data/outdir/predictions/。模型训练：在命令行执行：time python3 train.py –model-type= ‘ast-attendgru’ –gpu=0 –epochs=5; 每个时期的历史信息都存储在 pkl 文件中，例如 /scratch/funcom/data/outdir/histories/attendgru_hist_1551297717.pkl。文件末尾的整数是训练开始的 Epoch 时间，用于指示历史、配置、模型和预测数据。预测：在控制台输入命令：time python3 predict.py /scratch/funcom/data/outdir/models/attendgru_E05_1551297717.h5 –gpu=0; 输出的预测将会写入文件如：/scratch/funcom/data/outdir/predictions/predict-attendgru_E05_1551297717.txt; 计算得分：在控制台输入命令：time python3 bleu.py /scratch/funcom/data/outdir/predictions/predict-attendgru_E05_1551297717.txt, 这将会输出预测文件的 BLEU 得分。

4.4 创新点

近年来在源码摘要领域，以神经机器翻译为基础的数据驱动方法迅速发展，但是几乎所有的技术都依赖程序有完善的内部说明；本文的提出的神经模型，将代码的单词与从 AST 得到的代码结构结合，该模型将代码中的文本视为单独的输入，这样模型能够独立于代码文本学习代码结构。即使没有提供内部说明，模型也能够提供连续的摘要。

5 实验结果分析

5.1 实验结果

对于 BLEU 得分，ast-attendgru 和 attendgru 性能大致相等：19.6 BLEU 与 19.4 BLEU。SBT 较低，约为 14 BLEU，而 codenn 约为 10，如下图所示：

model	B	B1	B2	B3	B4	dataset
ast-attendgru	19.6	39.3	22.2	14.9	11.4	starndard
attendgru	19.4	39	22	14.8	11.3	
sbt	14	31.8	16	10.1	7.5	
codenn	9.95	21.2	9.7	7.6	6.3	

图 3: 各模型 BLEU 得分

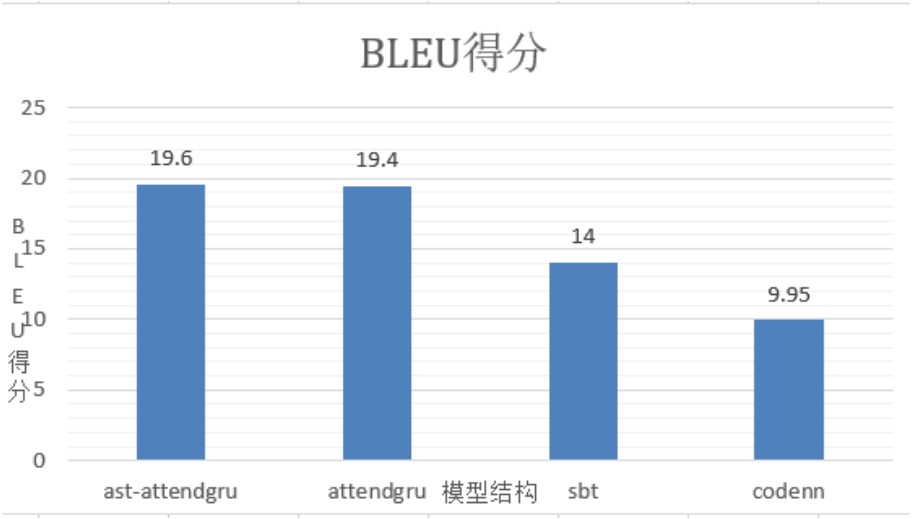

图 4: 各模型 BLEU 得分

5.2 分析

在标准实验中，ast_attendgru 与传统的 attendgru 表现不相上下，在文献中表示为正交的结果（这也是对模型进行进一步改进的启发），但均好于 sbt 方法和 codenn 方法, 在挑战实验中，本文献提出的 ast_attendgru 具有创新性，同时有 9.5 的 BLUE 得分。对于 SBT，其得分与原文献中较高的得分有些出入（原文献中分数较高），分析原因，可能是模型结构略有差异、数据集不同、原文献未按照项目划分数据集等原因。

6 总结与展望

文章提出了一个用于生成子程序自然语言描述的神经模型。实现了提出的模型，并在 Java 方法的数据集上对其进行了评估，就 BLEU 得分而言，优于 SE 文献中的基线，并略微领先于 NLP 很有竞争性的现成方法。鉴于传统 attendgru 和本文提出的 ast_attendgru 表现出正交的（各有长处）的特点，文章又提出构建两者的联合解码器模型，在每次预测时取结果较好的，模型最终也有较好的表现。同时，在没有良好内部说明文档的情况下，其他方法表现不好，而本文提出的模型仍可以生成连贯的摘要预测。

参考文献

[1] KRAMER D. Api documentation from source code comments: a case study of javadoc[C]//Proceedings of the 17th annual international conference on Computer documentation. 1999: 147-153.

[2] HELLENDOORN V J, DEVANBU P. Are deep neural networks the best choice for modeling source code?[C]//Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. 2017: 763-773.

[3] HU X, LI G, XIA X, et al. Deep code comment generation[C]//2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC). 2018: 200-20010.

[4] HU X, LI G, XIA X, et al. Summarizing source code with transferred api knowledge[J]., 2018.

[5] IYER S, KONSTAS I, CHEUNG A, et al. Summarizing source code using a neural attention model[C]

//Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). 2016: 2073-2083.

[6] JIANG S, ARMALY A, MCMILLAN C. Automatically generating commit messages from diffs using neural machine translation[C]//2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). 2017: 135-146.

[7] LOYOLA P, MARRESE-TAYLOR E, MATSUO Y. A neural architecture for generating natural language descriptions from source code changes[J]. arXiv preprint arXiv:1704.04856, 2017.

[8] ALLAMANIS M, PENG H, SUTTON C. A convolutional attention network for extreme summarization of source code[C]//International conference on machine learning. 2016: 2091-2100.

[9] YIN P, DENG B, CHEN E, et al. Learning to mine aligned code and natural language pairs from stack overflow[C]//2018 IEEE/ACM 15th international conference on mining software repositories (MSR). 2018: 476-486.

[10] MOVSHOVITZ-ATTIAS D, COHEN W. Natural language models for predicting programming comments[C]//Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers). 2013: 35-40.

[11] ALLAMANIS M, TARLOW D, GORDON A, et al. Bimodal modelling of source code and natural language[C]//International conference on machine learning. 2015: 2123-2132.

[12] GU X, ZHANG H, KIM S. Deep code search[C]//2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). 2018: 933-944.

[13] BAHDANAU D, CHO K, BENGIO Y. Neural machine translation by jointly learning to align and translate[J]. arXiv preprint arXiv:1409.0473, 2014.

[14] HAIDUC S, APONTE J, MORENO L, et al. On the use of automated text summarization techniques for summarizing source code[C]//2010 17th Working Conference on Reverse Engineering. 2010: 35-44.

[15] SUTSKEVER I, MARTENS J, HINTON G E. Generating text with recurrent neural networks[C]// ICML. 2011.

[16] LOPES C. UCI source code data sets[J]. http://www. ics. uci. edu/-lopes/datasets/, 2010.