Bachelor Project

Gabriele Bavota, Giuseppe Crupi

Report Writing and Poster Design

Report Writing

Engineering

You build a tool or improve an existing tool.

- GitHub repositories search engine. *O. Dabić.*
- Enhancing GHS for mining code-related metrics and repository topics. *A. Cerfeda.*
- Code quality assessment in real time. *L. Frunzio.*

Research

You run a research project.

- Develop and experiment a novel approach, algorithm, ML model, network protocol, programming language.
- Technology-oriented studies (e.g., evaluate self-driving agents on simulators).
- Human-oriented studies (e.g., study the level of students' attention during lectures).
- Data science and analytics: analyze, visualize, and interpret large-scale datasets (either existing or created from scratch) to extract meaningful insights.
- Interdisciplinary research: combining informatics with other fields of study, e.g., looking at ethical and social implications of AI, algorithmic bias.

Engineering

Research

Engineering

Abstract

Research

Abstract

Context

Problem

Limitations in SOA

Contribution & Findings

Engineering

Context

Code metrics can be used to assess the internal quality of software systems, and in particular their adherence to good design principles. For example, metrics can tell us how complex is the code we wrote, or how cohesive are the responsibilities implemented in a class.

Problem

Limitations in SOA

Contribution & Findings

Context

Code metrics can be used to assess the internal quality of software systems, and in particular their adherence to good design principles. For example, metrics can tell us how complex is the code we wrote, or how cohesive are the responsibilities implemented in a class.

Problem

While providing hints about code quality, metrics are difficult to interpret. Indeed, they take a code component as input and assess a quality attribute (e.g., code readability) by providing a number as output.

Limitations in SOA

Contribution & Findings

Context

Code metrics can be used to assess the internal quality of software systems, and in particular their adherence to good design principles. For example, metrics can tell us how complex is the code we wrote, or how cohesive are the responsibilities implemented in a class.

Problem

While providing hints about code quality, metrics are difficult to interpret. Indeed, they take a code component as input and assess a quality attribute (e.g., code readability) by providing a number as output.

Limitations in SOA

Current tools do not provide any support in the interpretation of code metrics, making unclear to the developers whether a given value should be considered good or bad for the specific code at hand.

Contribution & Findings

Context

Code metrics can be used to assess the internal quality of software systems, and in particular their adherence to good design principles. For example, metrics can tell us how complex is the code we wrote, or how cohesive are the responsibilities implemented in a class.

Problem

While providing hints about code quality, metrics are difficult to interpret. Indeed, they take a code component as input and assess a quality attribute (e.g., code readability) by providing a number as output.

Limitations in SOA

Current tools do not provide any support in the interpretation of code metrics, making unclear to the developers whether a given value should be considered good or bad for the specific code at hand.

Contribution & Findings

We present RETICULA (REal TIme Code qUaLity Assessment), a plugin for the IntelliJ IDE to assist developers in perceiving code quality during software development. RETICULA compares the quality metrics for a project under development in the IDE with those of similar open source systems previously analyzed. With the visualized results, developers can gain insights about the quality of their code. A video illustrating the features of RETICULA can be found at: https://reticulaplugin.github.io/.

Research

Context

Code completion is a key feature of Integrated Development Environments (IDEs), aimed at predicting the next tokens a developer is likely to write, helping them write code faster and with less effort.

Problem

Limitations in SOA

Contribution & Findings

Context

Code completion is a key feature of Integrated Development Environments (IDEs), aimed at predicting the next tokens a developer is likely to write, helping them write code faster and with less effort.

Problem

Preliminary approaches to automate code completion relied on rule-based or symbolic techniques (e.g., suggesting APIs to invoke according to the type of the variable that will store the returned value), but were only capable of predicting the next token to write.

Limitations in SOA

Contribution & Findings

Context

Code completion is a key feature of Integrated Development Environments (IDEs), aimed at predicting the next tokens a developer is likely to write, helping them write code faster and with less effort.

Problem

Preliminary approaches to automate code completion relied on rule-based or symbolic techniques (e.g., suggesting APIs to invoke according to the type of the variable that will store the returned value), but were only capable of predicting the next token to write.

Limitations in SOA

Modern code completion approaches are often powered by deep learning (DL) models. However, the swift evolution of pro- gramming languages poses a critical challenge to the performance of DL-based code completion models: *Can these models generalize across different language versions?*

Contribution & Findings

Context	Code completion is a key feature of Integrated Development Environments (IDEs), aimed at predicting the next tokens a developer is likely to write, helping them write code faster and with less effort.
Problem	Preliminary approaches to automate code completion relied on rule-based or symbolic techniques (e.g., suggesting APIs to invoke according to the type of the variable that will store the returned value), but were only capable of predicting the next token to write.
Limitations in SOA	Modern code completion approaches are often powered by deep learning (DL) models. However, the swift evolution of programming languages poses a critical challenge to the performance of DL-based code completion models: *Can these models generalize across different language versions?*
Contribution & Findings	This project delves into such a question. In particular, we assess the capabilities of a state-of-the-art model to generalize across nine different Java versions, ranging from Java 2 to Java 17, while being exclusively trained on Java 8 code. The results of our study reveal a noticeable disparity among language versions, with the worst performance being obtained in Java 2 and 17—the most far apart versions compared to Java 8. We investigate possible causes for the performance degradation and show that the adoption of a limited version-specific fine-tuning can partially alleviate the problem.

Engineering

Abstract

1. Introduction

Research

Abstract

1. Introduction

Introduction is an "expanded abstract"

(same structure)

Introduction is an "expanded abstract"

Tips:

- Avoid repetitions from the abstract
- Acknowledge seminal work in the area (very similar tool or research)
- In the last paragraph list all your contributions
- Add a subsection titled "Report structure" where you briefly discuss the content of each following section (e.g., In section 2, we review previous studies in the context of ....).

Engineering

Abstract

1. Introduction

2. State of the art

Research

Abstract

1. Introduction

2. State of the art

State of the art

Not enough to describe what others did

Summarize the main findings

Carefully explain the differences with your work

As compared to Bavota et al. [1], in our work ...
While we share the same goal of the study by Bavota et al. [1], we ...

Not enough to describe what others did

Summarize the main findings

Carefully explain the differences with your work

As compared to Bavota et al. [1], in our work ...
While we share the same goal of the study by Bavota et al. [1], we ...

It could feature “background information”

E.g., for a project dealing with bias in DL models, you may want to have a background section explaining the different types of biases, and then describe the literature that deal with each of them.

Engineering

Abstract

1. Introduction

2. State of the art

3. Functional and non-functional requirements

Research

Abstract

1. Introduction

2. State of the art

3. Study design

Engineering

Functional and non-functional requirements

Requirements

Functional: What your software is supposed to do in terms of implemented features.

Example: A web application managing students' enrolment to exams

Requirements

Functional: What your software is supposed to do in terms of implemented features.

Example: A web application managing students' enrolment to exams

User Authentication & Authorization:

Users must be able to register and log in.

Different roles should have different permissions (e.g., teachers can create exams, students can only take them).

Password recovery and multi-factor authentication should be available.

Exam Creation & Management:

Teachers should be able to create, edit, and delete exams.

Teachers should be able to schedule exams.

Teachers should be able to see the list of students enrolled for their exams.

...

Requirements

Functional: What your software is supposed to do in terms of implemented features.

Example: A web application managing students' enrolment to exams

User Authentication & Authorization:

Users must be able to register and log in.

Different roles should have different permissions (e.g., teachers can create exams, students can only take them).

Password recovery and multi-factor authentication should be available.

Exam Creation & Management:

Teachers should be able to create, edit, and delete exams.

Teachers should be able to schedule exams.

Teachers should be able to see the list of students enrolled for their exams.

...

Note: Only a few showed as example. Each of them should be explained in a few paragraphs.

Requirements

Non-functional: <u>How</u> your software is supposed to operate. Measurable.

Requirements

Non-functional: <u>How</u> your software is supposed to operate. Measurable.

Example: A web application managing students' enrolment to exams

Performance & Scalability:

The exam schedule must load within 2 seconds in 95% of cases.

The system must support at least 100 concurrent users without performance degradation.

Security:

Passwords must be hashed using bcrypt with a minimum work factor of 12.

Usability:

All features must be reachable with a maximum of 5 user interactions.

Requirements

Non-functional: <u>How</u> your software is supposed to operate. Measurable.

Example: A web application managing students' enrolment to exams

Performance & Scalability:

The exam schedule must load within 2 seconds in 95% of cases.

The system must support at least 100 concurrent users without performance degradation.

Security:

Passwords must be hashed using bcrypt with a minimum work factor of 12.

Usability:

All features must be reachable with a maximum of 5 user interactions.

Note: Only a few showed as example. The value set for each of them should be justified.

Research

Study Design

Study Design

Describe the goal, context, and methodology of your study. The following may apply or not in specific cases:

Research questions: The questions you are trying to answer with the study.

Objects: Describe the data you analyzed to answer the research questions.

Subjects: Describe the sample of participants used in your study (if any).

Data collection method: Detail how data is gathered (e.g., surveys, interviews, lab tests).

Variables: What and how you measure (dependent) and manipulate (independent).

Data analysis: How you plan to analyze the collected data (plots, statistics, tests).

Replication package: Provide a link with the code/data needed to replicate your results.

Engineering

Abstract

1. Introduction
2. State of the art
3. Functional and non-functional requirements
4. System Design

Research

Abstract

1. Introduction
2. State of the art
3. Study design
4. Results discussion

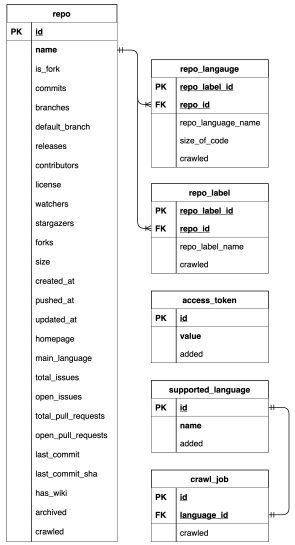
Engineering

System design

System design

Describe main components of your software. The following may apply or not in specific cases:

Database schema: What are the main entities you store and their relationships.

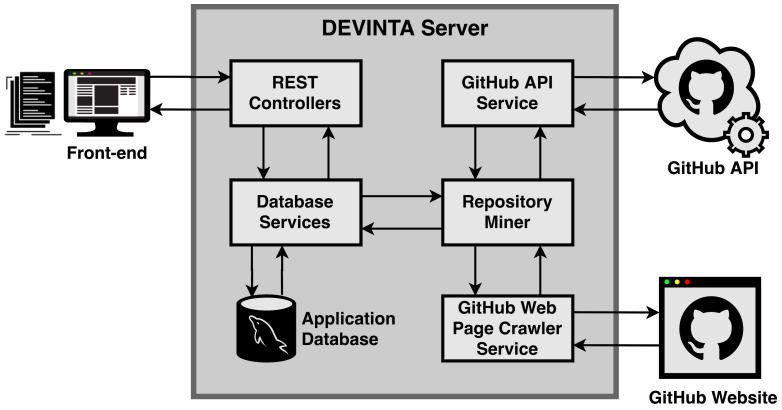


System design

Describe main components of your software. The following may apply or not in specific cases:

Database schema: What are the main entities you store and their relationships.

Architecture: From an abstract point of view, what are the main components and how do they interact? Consider using a component diagram here.



System design

Describe main components of your software. The following may apply or not in specific cases:

Database schema: What are the main entities you store and their relationships.

Architecture: From an abstract point of view, what are the main components and how do they interact? Consider using a component diagram here.

Components description: E.g., Front-end, back-end, APIs, algorithms, AI-based components (are those pre-trained models? Did you train them? If yes, using which data?), third-party libraries/APIs, *etc.* The point is not to show the code, but to describe their functioning, interesting implementation choices you made, technologies used, etc.

Research

Results discussion

Results discussion

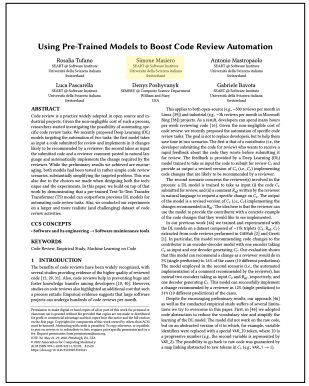
Present and discuss data answering your research questions.

Figures/tables: Results discussion is largely about commenting figures and tables. Don't go cheap on figures and tables.

Discuss by RQ: Presents the results by research question (e.g., one subsection per RQ). Explicitly answer each RQ at the end of the corresponding subsection.

Be objective: Discuss what can be observed from the data, not what you expected.

Qualitative analysis: It is difficult to read a results section only reporting numbers. Show qualitative examples.



Using Pre-Trained Models to Boost Code Review Automation

Rosalia Tufano
SEART @ Software Institute
Università della Svizzera Italiana
Switzerland

Simone Masiero
SEART @ Software Institute
Università della Svizzera Italiana
Switzerland

Antonio Mastropaolo
SEART @ Software Institute
Università della Svizzera Italiana
Switzerland

Luca Pascarella
SEART @ Software Institute
Università della Svizzera Italiana
Switzerland

Denys Poshyvanyk
SEMERU @ Computer Science Department
William and Mary
USA

Gabriele Bavota
SEART @ Software Institute
Università della Svizzera Italiana
Switzerland

ABSTRACT

Code review is a practice widely adopted in open source and industrial projects. Given the non-negligible cost of such a process, many research studies have been devoted to the automation of specific code review tasks. Recently proposed Deep Learning (DL) models targeting the automation of two tasks: the first model takes as input a code submitted for review and implements in it changes likely to be recommended by a reviewer; the second takes as input a code submitted for review and a natural language description of changes and automatically implements the change required by the reviewer. While the preliminary results we achieved are encouraging, both models had been tested in rather simple code review scenarios, substantially simplifying the targeted problem. This threatens the validity of the research since the results may not hold in larger and the experiments. In this paper, we build on top of that work by demonstrating that a pre-trained Text-To-Text Transfer Transformer (T5) model can outperform previous DL models for automating code review tasks. Such a model is then pre-trained on a larger and more realistic (and challenging) dataset of code review activities.

CCS CONCEPTS

• Software and its engineering → Software maintenance tools

KEYWORDS

Code Review, Empirical Study, Machine Learning on Code

1. INTRODUCTION

Modern code review practices have been widely recognized, with several studies providing evidence of the higher quality of reviewed code [15, 20, 52]. Also, code reviews help in preventing bugs and foster knowledge transfer among developers [18, 40]. Moreover, studies on code reviews also highlighted an additional cost that such practice adds to the development process. For instance, Microsoft developers are undergo hundreds of code reviews per month

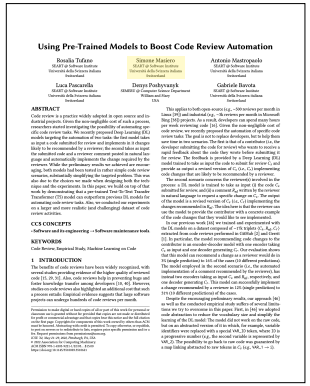
This applies to both open-source (e.g., ~500 reviews per month in Linux [20]) and industrial (e.g., ~6k reviews per month in Microsoft Bing [18]) projects. As a result, developers can spend many hours per week just reviewing code [18]. To ease the manual effort in code review, we recently proposed the automation of specific code review tasks. The goal is not to replace developers, but to help them save time in two scenarios. The first is that of a contributor (i.e., the developer individuating the code for review) who wants to receive a quick feedback before sending the code to the actual reviewer(s) for review. The feedback is provided by a Deep Learning (DL) model trained to take as input the code to submit for review C_0 and automatically implement in it changes likely to be recommended by code changes that are likely to be recommended by a reviewer.

The second scenario concerns the reviewer(s) involved in the process: a DL model is trained to take as input (1) the code C_0 submitted for review, and (2) a command R_{NL} written by the reviewer describing a change required to the code in C_0. The output is the new of the model is a revised version of C_0, i.e., C_1 implementing the changes recommended in R_{NL}. The idea here is that the reviewer can use the model to provide the contributor with a concrete example of the code changes that they would like to see implemented.

The first study on the DL models for code review automation [39] the DL models on a dataset composed of ~17k triplets $\langle C_0, R_{NL}, C_1 \rangle$ extracted from code reviews performed in GERRIT [2] and Gerrit Code Review [3] projects. The model proposed in the first of the two contributions is an encoder-decoder model with one encoder taking C_0 as input and one decoder generating C_1. Our evaluation shows that this model can recommend a change as a reviewer would do in 35 (single prediction) to 105 of the cases (10 different predictions). Instead, the model in the second study, aimed at the automatic implementation of a comment recommended by the reviewer(s), has instead two encoders taking as input C_0 and R_{NL}, respectively, and one decoder generating C_1. This model can successfully implement a change recommended by a reviewer in 126 (single prediction) to 370 (10 different predictions) cases.

Despite the encouraging preliminary results, our approach [39] as well as the conducted empirical study suffers of several limitations. First, we adopted a "vocabulary" of tokens of 1k and 2k tokens (code abstraction) to reduce the vocabulary size and accelerate the training of the DL model. The model did not work on the raw code, but on an abstracted version of it in which, for example, variable identifiers were replaced with a special VAR_ID token, where ID is a number uniquely identifying the variable (e.g., the variable foo is VAR_2). The possibility to go back to raw code was guaranteed by a map linking abstracted to raw tokens in C_1 (e.g., VAR_1 → i).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE'19, May 25–31, 2019, Montreal, QC, Canada
© 2019 Copyright held by the owner/author(s) Publication rights licensed to ACM.
ACM ISBN 978-1-2345-678-9/19/05 ... $15.00
https://doi.org/10.1145/3324884.3416591



Using Pre-Trained Models to Boost Code Review Automation

Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, Gabriele Bavota

ABSTRACT

Code review is a practice widely adopted in open source and industrial projects. Given the non-negligible cost of such a process, researchers started investigating the possibility of automating specific code review tasks. We recently proposed Deep Learning (DL) models trained and tested on large datasets of code review tasks. The goal is not to replace developers, but to help them save time in two scenarios. The first is that of a contributor (i.e. the developer submitting the code for review) who wants to receive a first feedback while waiting for a peer to be available for review. The feedback is provided by a Deep Learning (DL) model trained to take as input the code to submit for review C_1 and provide as output a revised version of C_1 (i.e., C_2) implementing code changes that are likely to be recommended by a reviewer.

This applies to both open source (e.g., ~500 reviews per month in Linux [39]) and industrial (e.g., ~36 reviews per month in Microsoft Bing [38]) projects. As a result, developers can spend many hours waiting for a peer review. The second scenario is that of a peer reviewer who wants to exploit the automation of specific code review tasks. The goal is not to replace developers, but to help them save time in two scenarios. The first is that of a contributor (i.e. the developer submitting the code for review) who wants to receive a first feedback while waiting for a peer to be available for review. The feedback is provided by a Deep Learning (DL) model trained to take as input the code to submit for review C_1 and provide as output a revised version of C_1 (i.e., C_2) implementing code changes that are likely to be recommended by a reviewer.

In this paper, we build on top of that work by demonstrating that a pre-trained Text-To-Text Transfer Transformer (T5) model can outperform previous DL models for automating code review tasks. Also, we conducted our experiments on a T5-based and more realistic (and challenging) dataset of code review activities.

CCS CONCEPTS

• Software and its engineering → Software maintenance tools.

KEYWORDS

Code Review, Empirical Study, Machine Learning on Code

1 INTRODUCTION

The benefits of code reviews have been widely recognized, with several studies providing evidence of the higher quality of reviewed code [6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37] and various other benefits such as knowledge transfer among developers [34, 36]. However, studies on code reviews also highlighted an additional cost that such a process entails. Empirical evidence suggests that large software projects can undergo hundreds of code reviews per month.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE '21, May 22–30, 2021, Madrid, Spain, Spain, Pittsburgh, PA, USA, ...
© 2021 Association for Computing Machinery ...
ACM ISBN 978-1-4503-9190-4/21/05 ...
https://doi.org/10.1145/3510419.3510621

This applies to both open source (e.g., ~500 reviews per month in Linux [39]) and industrial (e.g., ~36 reviews per month in Microsoft Bing [38]) projects. As a result, developers can spend many hours waiting for a peer review. The second scenario is that of a peer reviewer who wants to exploit the automation of specific code review tasks. The goal is not to replace developers, but to help them save time in two scenarios. The first is that of a contributor (i.e. the developer submitting the code for review) who wants to receive a first feedback while waiting for a peer to be available for review. The feedback is provided by a Deep Learning (DL) model trained to take as input the code to submit for review C_1 and provide as output a revised version of C_1 (i.e., C_2) implementing code changes that are likely to be recommended by a reviewer.

In this paper, we build on top of that work by demonstrating that a pre-trained Text-To-Text Transfer Transformer (T5) model can outperform previous DL models for automating code review tasks. Also, we conducted our experiments on a T5-based and more realistic (and challenging) dataset of code review activities.

CCS CONCEPTS

• Software and its engineering → Software maintenance tools.

KEYWORDS

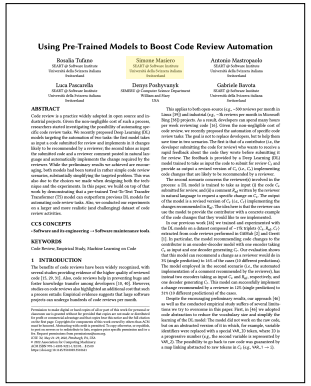
Code Review, Empirical Study, Machine Learning on Code

1 INTRODUCTION

The benefits of code reviews have been widely recognized, with several studies providing evidence of the higher quality of reviewed code [6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37] and various other benefits such as knowledge transfer among developers [34, 36]. However, studies on code reviews also highlighted an additional cost that such a process entails. Empirical evidence suggests that large software projects can undergo hundreds of code reviews per month.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE '21, May 22–30, 2021, Madrid, Spain, Spain, Pittsburgh, PA, USA, ...
© 2021 Association for Computing Machinery ...
ACM ISBN 978-1-4503-9190-4/21/05 ...
https://doi.org/10.1145/3510419.3510621

"The DL model can generate review comments equivalent to those of human reviewer in 15% of test set instances."



Using Pre-Trained Models to Boost Code Review Automation

Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, Gabriele Bavota

ABSTRACT

Code review is a practice widely adopted in open source and industrial projects. Given the non-negligible cost of such a process, researchers started investigating the possibility of automating specific code review tasks. We recently proposed Deep Learning (DL) models trained and tested on large datasets of code review tasks. The goal is not to replace developers, but to help them save time in two scenarios. The first is that of a contributor (i.e. the developer submitting the code for review) who wants to receive a first feedback while waiting for a peer to be available for review. The feedback is provided by a Deep Learning (DL) model trained to take as input the code to submit for review C_1 and provide as output a revised version of C_1 (i.e., C_2) implementing code changes that are likely to be recommended by a reviewer.

This applies to both open source (e.g., ~500 reviews per month in Linux [39]) and industrial (e.g., ~36 reviews per month in Microsoft Bing [38]) projects. As a result, developers can spend many hours waiting for a peer review. The second scenario is that of a peer reviewer who wants to exploit the automation of specific code review tasks. The goal is not to replace developers, but to help them save time in two scenarios. The first is that of a contributor (i.e. the developer submitting the code for review) who wants to receive a first feedback while waiting for a peer to be available for review. The feedback is provided by a Deep Learning (DL) model trained to take as input the code to submit for review C_1 and provide as output a revised version of C_1 (i.e., C_2) implementing code changes that are likely to be recommended by a reviewer.

In this paper, we build on top of that work by demonstrating that a pre-trained Text-To-Text Transfer Transformer (T5) model can outperform previous DL models for automating code review tasks. Also, we conducted our experiments on a T5-based and more realistic (and challenging) dataset of code review activities.

CCS CONCEPTS

• Software and its engineering → Software maintenance tools.

KEYWORDS

Code Review, Empirical Study, Machine Learning on Code

1 INTRODUCTION

The benefits of code reviews have been widely recognized, with several studies providing evidence of the higher quality of reviewed code [6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37] and various other benefits such as knowledge transfer among developers [34, 36]. However, studies on code reviews also highlighted an additional cost that such a process entails. Empirical evidence suggests that large software projects can undergo hundreds of code reviews per month.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE '21, May 22–30, 2021, Madrid, Spain, Spain, Pittsburgh, PA, USA, ...
© 2021 Association for Computing Machinery ...
ACM ISBN 978-1-4503-9190-4/21/05 ...
https://doi.org/10.1145/3510419.3510621

"The DL model can generate review comments equivalent to those of human reviewer in 15% of test set instances."

code-to-comment

"Please make this one a variable as well"

```
public void handleSetDeviceLifecycleStatusByChannelResponse([...]) { [...] ResponseMessage.newResponseMessageBuilder().[...]}
```

"Extract the building of the ResponseMessage to it's own variable (in eclipse, select the text, right-click > refactor > extract local variable / select code + shift+alt+L). This will make the code a bit more readable, especially when you'll be passing in other things besides the ResponseMessage."

Results discussion

Present and discuss data answering your research questions.

Figures/tables: Results discussion is largely about commenting figures and tables. Don't go cheap on figures and tables.

Discuss by RQ: Presents the results by research question (e.g., one subsection per RQ). Explicitly answer each RQ at the end of the corresponding subsection.

Be objective: Discuss what can be observed from the data, not what you expected.

Qualitative analysis: It is difficult to read a results section only reporting numbers. Show qualitative examples.

Discuss implications: What are the practical implications of your results? For whom are those relevant (e.g., other researchers, people using AI, software developers, *etc.*)

Engineering

Abstract

1. Introduction

2. State of the art

3. Functional and non-functional requirements

4. Design

5. Usage scenarios

Research

Abstract

1. Introduction

2. State of the art

3. Study design

4. Results discussion

5. Threats to validity

Engineering

Usage scenarios

Usage scenarios

Showcase the software by showing what the user would see using its killer features

Use this section to showcase the UI

It is basically one running example showing how to use the app

Research

Threats to validity

Maximizing validity

Internal validity

Mostly confounding factors which can affect your measurements

Can be addressed by good experimental design

External validity

Issues if results are not generalizable but only apply to narrow population/conditions you tested

Requires use of intuition, judgment

Internal validity threats

Group threat

Control & experimental groups are too different

Time threat (w/ people)

Maturation (learning effect, tiring effect)

Passage of time and historical events introduce behavioral changes

Time threat (w/ measurement)

Machines get better/worse (e.g., battery level)

Experimenter gets better at using instruments

Internal validity threats

Differential mortality

People drop out from control/experimental group. Maybe only very motivated people are left.

Experimenter effect

People behave differently when measured

Evaluation apprehension: Anxiety of being tested

Social desirability: People want to look good

Demand characteristics: People want to please, what to make experiment “work”

Internal validity threats

Experimental Settings

Lack of hyperparameters tuning in a study about DL models

Usage of noisy data in the training of a DL model, or in a performed data analysis

Experimenter effect

People behave differently when measured

Evaluation apprehension: Anxiety of being tested

Social desirability: People want to look good

Demand characteristics: People want to please, what to make experiment “work”

External validity threats

Using special participant groups

Most popular group in SE paper: CS undergraduate students as representative of developers :)

Volunteers often have higher respect for science

External validity threats

Using special participant groups

Most popular group in SE paper: CS undergraduate students as representative of developers :)

Volunteers often have higher respect for science

Using too few participants/instances

Hard to generalize from only a few instances

This could related to participants in a human-oriented experiment or to data points in general (e.g., study 2 DL models to learn about their bias, but those are only 2 of all existing models)

Engineering

Abstract

1. Introduction
2. State of the art
3. Functional and non-functional requirements
4. Design
5. Usage scenarios
6. Conclusions

Research

Abstract

1. Introduction
2. State of the art
3. Study design
4. Results discussion
5. Threats to validity
6. Conclusions

Conclusions

Tips:

- Avoid repetitions from the abstract and introduction
- Summarize what was planned and the motivation behind it
- Summarize what has been achieved
- Add a “Future Work” subsection where to discuss e.g., improvements you foresee for your tool or additional experiments/analyses not performed due to the lack of time/resources

Name Faculty or Institute

Name Surname
Function
Email

Name Surname
Function
Email

Name Surname
Function
Email

Name Faculty or Institute

Name Surname
Function
Email

Name Surname
Function
Email

Name Surname
Function
Email

Context

Problem & Limitations of SOTA

Solution

Results

Engineering

Name Faculty or Institute

Name Surname
Function
Email

Name Surname
Function
Email

Name Surname
Function
Email

Context

Problem & Limitations of SOTA

System design

Algorithms

Implementation choices

GUI

Usage Scenario

Plan for a live demo!

Name Faculty or Institute

Name Surname
Function
Email

Name Surname
Function
Email

Name Surname
Function
Email

Research

Context

Problem & Limitations of SOTA

Study design

Proposed technique

Results

Implications