Analyzing Software using Deep Learning

– Project Description, Summer Semester 2020 –

Prof. Dr. Michael Pradel, Jibesh Patra

May 22, 2020

1 Goal

The goal of this project is to design, implement, and evaluate a neural network-based program analysis that detects software bugs. Given a source code file, the analysis will report a ranked list of warnings about specific lines in the file that are likely to contain a bug. At the core of the approach is a learned classification model that determines whether a given code example is correct or incorrect. To train this model, the approach relies in a given corpus of code, which is assumed to be mostly correct, and a set of program transformations, which create likely incorrect code examples. Figure 1 gives an overview of the approach. The overall design is similar to DeepBugs [1]. Each of the boxes in the figure corresponds to one component to be implemented in the project.

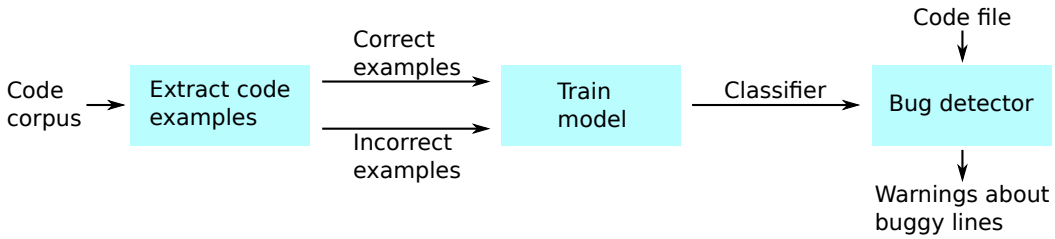

Figure 1: Overview of the approach.

The project will focus on JavaScript as the target language, i.e., the resulting tool will search for bugs in JavaScript code. The analysis itself will be implemented in Python.

2 Bug Patterns

The approach will focus on a common class of bugs, namely incorrect conditionals. Specifically, the project will focus on the following five bug patterns, each illustrated by an example:

1. *Incomplete conditional expression.* For this project, only existing conditions that are missing a subexpressions are in scope, but not `if` statements that are entirely missing.

```
// Correct
if (isBox && elem.nodeType === 1) {
  // ...
}
```

```
// Incorrect
if (isBox) {
  // ...
}
```

2. *Incorrectly ordered boolean expression.* All logical expressions that are not commutative are in scope.

```
// Correct
if (data[index] && data[index].stop) {
  // ...
}
```

```
// Incorrect
if (data[index].stop && data[index]) {
  // ...
}
```

3. *Wrong identifier.* Any use of an correct identifier within a conditional expression is in scope.

```
// Correct
if (isArrayLike(obj)) {
  length = obj.length;
  // ...
}
```

```
// Incorrect
if (isArrayLike(doc)) {
  length = obj.length;
  // ...
}
```

4. *Negated condition.* Any expression or subexpressions that should (or should not be) negated with `!` is in scope, but not pairs of operators that logically negate each other, such as `>` and `<=`.

```
// Correct
if (!blueBlink) {
  // ...
}
```

```
// Incorrect
if (blueBlink) {
  // ...
}
```

5. *Wrong operator.* Any operator, both binary and unary, that is used within a conditional expression is in scope.

```
// Correct
if (idx < l.length) {
  // ...
}
```

```
// Incorrect
if (idx > l.length) {
  // ...
}
```

For all of these bugs patterns, the project focuses on conditionals in `if` statements (including the conditionals used in the `else if` part of an `if` statement). Other conditionals, e.g., in the header of `for` or `while` loops, and any other statements that contain boolean expressions are out of scope.

3 Components of the Analysis

The analysis consists of three main components, as illustrated in Figure 1. Here are some hints for implementing these components.

3.1 Extracting Code Examples

Given a corpus of JavaScript code, the first component extracts code examples that will serve as training data for the neural model. Training the classification model in a supervised manner requires examples of correct code and examples of incorrect code. The approach should assume that all code in the given code corpus is correct (which in reality is true for most but not all of the code) and extract correct code examples from conditionals in the given code. To create incorrect code examples, the approach should transform the conditionals in the given code corpus based on the five bug patterns. For example, given a conditional `x && x.prop`, these transformations could yield incorrect conditions `x.prop` (pattern 1), `x.prop && x` (pattern 2), `y && x.prop` (pattern 3), `!x && x.prop` (pattern 4), and `x || x.prop` (pattern 5).

Creating a large and diverse set of incorrect code examples for training is crucial. To obtain balanced training data, i.e., the same number of correct and incorrect examples, we recommend pairing each correct code examples with one incorrect example.

To simplify implementing this component, we provide you with a pre-processed dataset that contains three representations of JavaScript source code files: (i) the raw source code, (ii) the source code tokenized into a sequence of tokens, and (iii) the source code parsed into an abstract syntax tree (AST). The dataset consists of JSON files that contain these three representations; each JSON file corresponds to one JavaScript file. We anticipate that representations (ii) and (iii) are sufficient for most projects, but also provide (i) to allow working on any other kind of representation. To better understand the AST format, please play with the parser demo of Esprima[1] and follow the ESTree documentation[2]. Each AST node and each token are annotated with their "range", which refers to their character position in the source code file, and with the line number in the source code file.

[1] `https://esprima.org/demo/parse.html`

[2] `https://github.com/estree/estree/blob/master/es5.md`

3.2 Training the Model

Given the dataset of correct and incorrect code examples, the second component trains a classification model that learns to distinguish between the two kinds of examples. To this end, the code must be transformed into a vector representation and fed into a suitable model. You can freely choose the representation and the model. For example, this component could feed code tokens into a simple feedforward model or an RNN model, feed code tokens into a hierarchical neural model, or feed an AST into a graph-based model. For representing identifier names, e.g., `isBox` and literals, e.g., `true`, we recommend mapping each into a vector based on pre-trained word embedding. We provide such an embedding as part of the project; *run_bug_finding.py* shows an example of using the pre-trained embedding.

Each input given to the model should contain code from one conditional. In addition to the conditional itself, you can include additional context information, such as the code before and/or after the conditional. The classifier predicts for the given conditional whether it is correct or incorrect. During training, the expected output is "correct" for the code examples that were extracted as-is from the code corpus, and "incorrect" for the code examples that were transformed based on the bug patterns.

This component must be implemented based on the PyTorch library.

3.3 Detecting Bugs

Given the trained classification model, the third and final component predicts which lines in a given JavaScript file are likely to contain an incorrect conditional. The input given to the bug detector is the same kind of JSON file as in the pre-processed dataset. The bug detection component must extract all conditionals from the given file, represent them in the same way as in the second component, and query the trained classifier for whether the conditional is correct or incorrect. For any conditional predicted to be incorrect, the bug detector should report a warning at the corresponding source code line (in case a conditional spans across multiple lines, use the first of these lines). The data provided along with this project contains a not very clever example implementation of the bug detector, which illustrates the interface the bug detector must implement.

4 Dataset and Hardware

The project comes with a dataset of 10,000 JSON files, provided as the "public dataset", which originate from 10,000 JavaScript files gathered from various open-source projects. We recommend using different subsets of this dataset as follows:

- While *developing your model*, use a small subset of the dataset, e.g., only five JSON files, to test and debug your implementation. For developing your model, it is usually the most convenient to implement and execute the project on your local laptop.
- To *train and evaluate your model for yourself*, split the dataset into a training set, e.g., 80% of all files, and a validation set, e.g., the remaining 20% of the files. Generate positive and negative examples for both sets, then train the model with the training set, and measure how well the trained model works on the validation set. The easiest measure of success is accuracy, i.e., for how many code examples the model correctly predicts whether the code is correct or buggy. If your validation set is balanced, i.e., it contains the same number of correct and buggy examples, then a completely untrained model will trivially achieve an accuracy of 50%. Your goal is to reach an accuracy that is significantly higher than 50%.
- To *evaluate your model in the competition* (see below for details), train the model on the full dataset of 10,000 JSON files. By using the full dataset, you will obtain the best possible results. We will use a different set of files to evaluate the uploaded models.

If training and evaluating on your local laptop takes too long, please consider using a cloud computing service. Each student has free access to some computing resources via the bwCloud[3].

[3] `https://www.tik.uni-stuttgart.de/dienste-a-z/bwCloud/`

Alternatively, you can use external cloud computing services, some of which provide a limited amount of free credits to students.[4]

5 Competition and Leaderboard

To evaluate and compare the different solutions, we use an online competition with a leaderboard hosted at CodaLab:

`https://competitions.codalab.org/competitions/24920?secret_key=46beefc0-3e4e-4416-bc47-32f6c6ad5c27`

Each student must participate in the competition to gain credit for the course project. The effectiveness of the uploaded solutions, as measured and shown on the leaderboard, is the most important part of the grade for the project. Each student can submit as often as they like (subject to some limits, e.g., on the number of submissions per day). Only the best solution uploaded by each student influences the grade. It is strongly recommended to upload solutions throughout the semester to understand the scoring used for the competition and to get feedback on where each student stands compared to the others.

To submit to the competition, you must upload a trained model and all code required to use this model for bug detection. The competition is based on a set of JavaScript files (represented as JSON files, see above) that contain bugs caused by incorrect conditionals. We have manually created these bugs and hence know where they are. For each uploaded solution, we compute three measures of success:

- Precision, which is $\frac{\text{Nb. of reported lines that are actually buggy}}{\text{Nb. of lines reported as buggy}}$. For example, if the bug detector flags five lines as buggy, out of which two are indeed buggy, then the precision is 40%.
- Recall, which is $\frac{\text{Nb. of buggy lines that are reported as buggy}}{\text{Nb. of buggy lines}}$. For example, if the files we use for evaluation contain 20 bugs and your bug detector finds five of them, then the recall is 25%.
- F1-score, which is the harmonic mean of precision and recall, i.e., $\frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$

The F1-score is shown in the leaderboard. In other words, the overall goal of the bug detector is to find as many of the bugs as possible, without producing an overwhelming number of warnings. For example, a solution that reports a bug at each line will achieve 100% recall, but have a poor precision, whereas a solution that detects only some kinds of bugs may achieve a high precision, but will have poor recall.

To run your bug detector for the competition, we use Python 3.8. In addition, the following third-party packages (and their respective dependencies) are available: tqdm, torch, nltk, numpy, scipy, fasttext, sklearn, pandas, matplotlib, seaborn, datetime, sympy, gensim. If you want to use additional third-party packages, please contact Jibesh Patra, so we can make available these packages for the competition.

6 Getting Started

To get started, please follow these steps:

1. Register at CodaLab. Please use your university email address (which will not be publicly visible) and any username you like (which will be publicly visible).
2. Link your CodaLab user name to your student account by sending a message via Ilias to Jibesh Patra. The message must contain your CodaLab user name.
3. Download and unpack the starting kit and the public dataset from CodaLab.
4. Install Python 3 and the dependencies required by the starting kit, e.g., using `pip install torch` and `pip install fasttext`.

[4] E.g., see `https://edu.google.com/programs/students/`.

5. Run the trivial baseline bug detector provided as part of the project:
   `python3 evaluation.py dataset_dir` where `dataset_dir` is the directory of public dataset. If you want to speed up this step, create a copy of the `dataset_dir` where you keep only a few of the JSON files.

6. Zip the folder that includes the starting kit and upload this baseline bug detector to the competition. It will add your user name to the leaderboard (with an F1-score of 0%).

7. Modify the bug detector, e.g., so that it predicts a bug at each line, at each line with an `if`, or at a random subset of lines with an `if`. Upload the modified bug detector to the competition, which may improve your F1-score.

8. Now you are ready to begin with the actual project by implementing the three components described earlier in this document.

7 Deadline and Deliverables

The deadline for submitting the project is July 17, 2020 (end of day, Stuttgart time). This deadline is firm and will not be extended. Submitting the project means to upload a complete and runnable implementation to the competition. An implementation that does not show up on the leaderboard, e.g., because it fails due to some error, cannot be graded. In addition to the code necessary to run your bug detector, please also include any code used for generating training data and for training the model, so we can consider this code for grading. In addition to the submission, there will be short oral presentations of the projects in the week of July 20–24, 2020.

The overall grade is computed as follows:

Criterion	Weight
Originality of your approach	20%
Effectiveness of your implementation	50%
Code quality and documentation	10%
Oral presentation	20%

We strongly recommend to regularly consult with Jibesh Patra (jibesh.patra@iste.uni-stuttgart.de) about the progress of your project, to resolve any questions, to ensure progress into the right direction, and to help you submit a successful project in time.

Finally, a note on plagiarism: The project is individual. As for any individual, graded assignment, project, or exam, collaborating with other students is not allowed. In particular, this means you are not allowed to show other students your implementation or to explain its design in detail. Likewise, your implementation may not contain any code copied from other sources, except for well-marked third-party dependencies. Cases of plagiarism will be punished by failing the course project and by reporting the case to the study administration. We do, however, encourage everybody to use the Ilias forum for discussions on technical problems, e.g., with the PyTorch library, or on general questions about the project.

References

[1] Michael Pradel and Koushik Sen. DeepBugs: A learning approach to name-based bug detection. *PACMPL*, 2(OOPSLA):147:1–147:25, 2018.