P252 Project Descriptions

Michaelmas 2019

December 20, 2019

P252 is structured around solving a machine learning task on source code. Everyone on the course will be allocated a different task. This document specifies the tasks which are available to choose between.

Please read through these descriptions and rank the tasks in terms of your interest. We will try to assign tasks by 'maximising happiness' across all students on the course. We will tell you your assigned task in the first lecture of the course and provide a list of related papers for you to read before developing your project proposal. We are withholding the list of papers associated with each task at this time because we do not expect you to undertake a significant amount of work before starting the course. The format and expectations for the project proposal will be discussed in the first lecture.

Each task is open-ended with a range of potential approaches of varying difficulty and so we encourage you to rank your preferences based on your personal enthusiasm for the task rather than (for example) any concerns about maximising marks on the module.

1 Method2Comment

Predict method documentation of code in natural language. Models can be trained to predict the JavaDoc comments of methods within the course's corpus, ignoring JavaDoc tags (e.g. `@param`, `@throws`). The goal of these models is to generate descriptive natural language comments, or detect if a comment is up-to-date or stale.

Possible Approaches

The code can be encoded as a sequence using LSTM/GRU, CNNs or Transformers; as a tree or graph; or using a sequential abstract of trees.

The decoder generating the method documentation can be implemented as an RNN (with attention), CNN, or Transformer.

2 Param2Comment

Given a method parameter and the method implementation, the goal is to predict the JavaDoc `@param` natural language comment. Models can be trained to predict the `@param` JavaDoc

comments that exist in the course's corpus. This is a particular case of Method2Comment with the potential to focus on (for example) dataflow to understand the use of a single parameter. Such a model can be to generate descriptive natural language @param comments.

Possible Approaches

The code can be encoded as a sequence using LSTM/GRU, CNNs or Transformers; as a tree or graph; or using a sequential abstract of trees .

The decoder generating the method documentation can be implemented as an RNN (with attention), CNN, or Transformer.

3 Code Deobfuscation

Given a Java method where all the variable names have been obfuscated (for instance as a result of compilation), recover meaningful and informative names for all the variables. Models can be trained by automatically renaming all variables in the course's corpus and asking the model to learn to predict the ground-truth names. Such a model should be able to help a developer understand obfuscated code.

Possible approaches

- Sequence-based Models (Contextual models, masked Transformer Language Models).
- Graph-based models:
  - Graph Neural Networks
  - CRFs
  - Neural CRFs and Graph Neural Networks http://nlp.seas.harvard.edu/pytorch-struct/

Variable names could be predicted either as a sequence of subtokens or as a sequence of characters.

4 Variable Naming

Propose a descriptive name for a single variable. Models can be trained to predict the names of the variables in the course's corpus. Such a model can be a part of an auto-completion system or a part of a linter that flags non-descriptive or misleading variable names.

Possible Approaches

- Sequence-based models
  - Token-based autoencoder + RNN
  - Transformers
- Graph-based models

  - Graph Neural Networks
  - Neural path-based model
  - structured prediction - CRFs

Variable names could be predicted either as a sequence of subtokens or at a character-level.

5 Method Naming

Propose a descriptive name for a method given its implementation. Models can be trained to predict the names of the methods in the course's corpus. This might be used for auto-completion in an IDE or for detecting anomalous names in a public API.

Possible Approaches

- Sequence-based models (GRUs/LSTMs, CNN, Transformers), Relational Transformers
- Path-based models
- Graph Neural Networks

6 Semantic Code Search

Given a natural language query rank a set of methods according to decreasing relevance. By using the JavaDoc comments of the course's corpus as queries, a system can be trained to detect the (ir)relevance of a given pair of code and a query.

Possible approaches

- Token-level models similar to those in CodeSearchNet
- Graph-based models

7 Code Completion

Given an incomplete code snippet (i.e. a hole), predict the chunk of code that should be placed in the hole (next token, line, etc). Training happens by partially "hidding" parts of existing code and asking the model to predict them back. Such models can find application in code editors.

Possible approaches

- Token-level language models (this is the course's first practical and is *not* available as a choice)
- Insertion transformer-like architectures
- Relational transformers
- Tree-generative models (based on AST)

8 Deep Argument Swapping Detection

Predict whether a given function call passes parameters in the correct order. For example, invoking `s.Substring(start, offset)` vs `s.Substring( offset, start)`. To train this model, incorrect method calls would have to be added to the course's corpus, by copying existing function calls and introducing type-correct swaps. Such a model can act as an analyzer in the IDE or in a build system.

Possible Approaches

- A neural repair model using formal parameter names
- Sequence Models

9 Variable Misuse

Given code locate incorrect variable usages that could not be detected by the type checker. For example, instead of `i` the developer should have used `j`. A model can be trained to predict variable misuses that are synthetically introduced to the corpus.

Note: this task requires some program analysis for introducing valid synthetic misuses.

Possible Approaches

- Graph Neural Networks
- Transformers

10 Text2Code

Given some text describing a method the goal is to generate the code of the described method. Training can happen using the JavaDoc comments in the course corpus. Given that this task is hard, Text2Code models will be evaluated based on the perplexity.

Possible Approaches

- Sequence-based models (LSTMs, Transformers etc)
- syntactic neural model (AST + grammar model)
- retrieve & edit
- probabilistic context-free grammar

11 Anomalous Code Detection

Train a model, which given a code snippet, finds various code anomalies, such as bugs or coding style issues. The task would involve either introducing synthetic bugs in the some snippets of code (e.g. switching relational operators) and asking the model to locate them. Such a machine learning model could find use within a code editor.

Possible Approaches

- Graph Neural Networks
- Transformers

12 Predicting Types

Although Java is statically typed, resolving types often needs to know the full build environment. This may not be possible in many settings, where the environment cannot be resolved (StackOverflow, when browsing code in GitHub). The goal of this task is to predict the type of a variable given its uses. Training can happen by using the existing compiler information. Such a method could have an application as a browser plugin for GitHub or StackOverflow.

Possible approaches

- Sequence-based models, such as Transformers or biRNNs.
- NL2Type
- Graph Neural Networks

13 Syntactic Program Repair

Given a method containing a simple bug (e.g. off-by-one error, wrong method call, wrong java keyword) in an unknown location, return the method with the issue fixed. One way to approach this would be to deliberately introduce syntax errors in to a compiling program and train the model to predict how to repair them.

Possible approaches

- Sequence-based models
- Neural Machine Translation
- Graph-2-diff models