Machine Learning for Software Analysis (MLSA)

IMT School For advanced Studies Lucca

IMT
SCUOLA ALTI STUDI LUCCA

Fabio Pinelli
fabio.pinelli@imtlucca.it

26/11/2024

Outline

IMT SCUOLA ALTI STUDI LUCCA

- Where we are
- Intro into Software Analysis
- Overview of some applications and papers
- Preparation for the project

Timeline

Day	N. of hours
24/09/2024	3
26/09/2024	2
01/10/2024	3
03/10/2024	2
08/10/2024	3
10/10/2024	2
15/10/2024	3
17/10/2024	2
22/10/2024	3
24/10/2024	2
29/10/2024	0
31/10/2024	2
05/11/2024	3
07/11/2024	2
12/11/2024	3
14/11/2024	2
19/11/2024	3
21/11/2024	0
26/11/2024	3
28/11/2024	2
03/12/2024	3
Total hours	**48.00**

IMT SCUOLA ALTI STUDI LUCCA

Software analysis

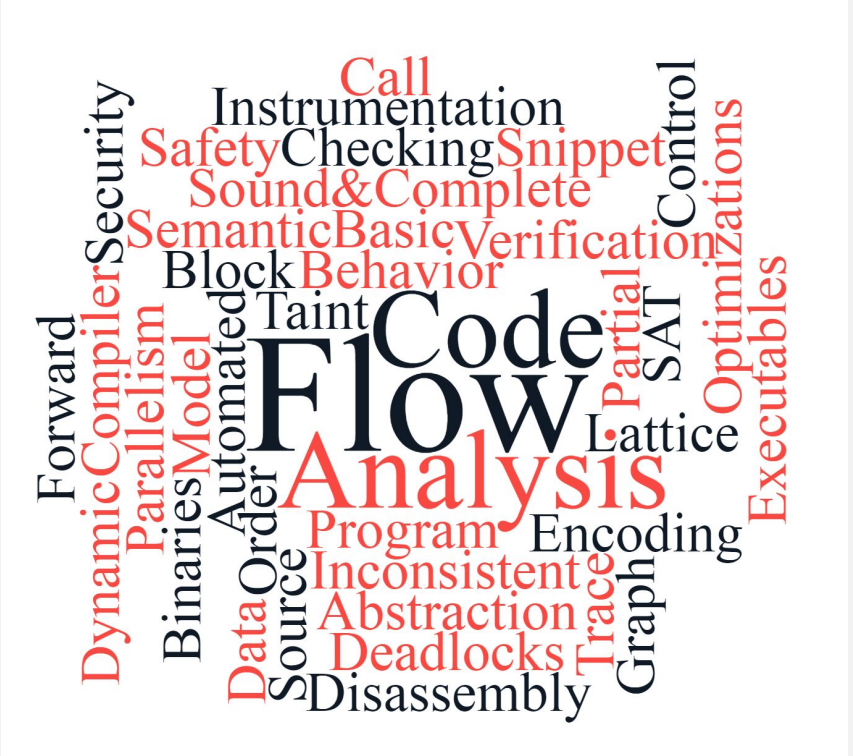
IMT SCUOLA ALTI STUDI LUCCA

Software or Program analysis refers to the process of examining and evaluating software artifacts:

- source code,
- executables, and
- documentation

to gain insights into various aspects of software quality, performance, security, and maintainability.

The primary goal of program analysis is to improve **software reliability, efficiency, and maintainability while reducing the risk of errors, vulnerabilities, and defects.**



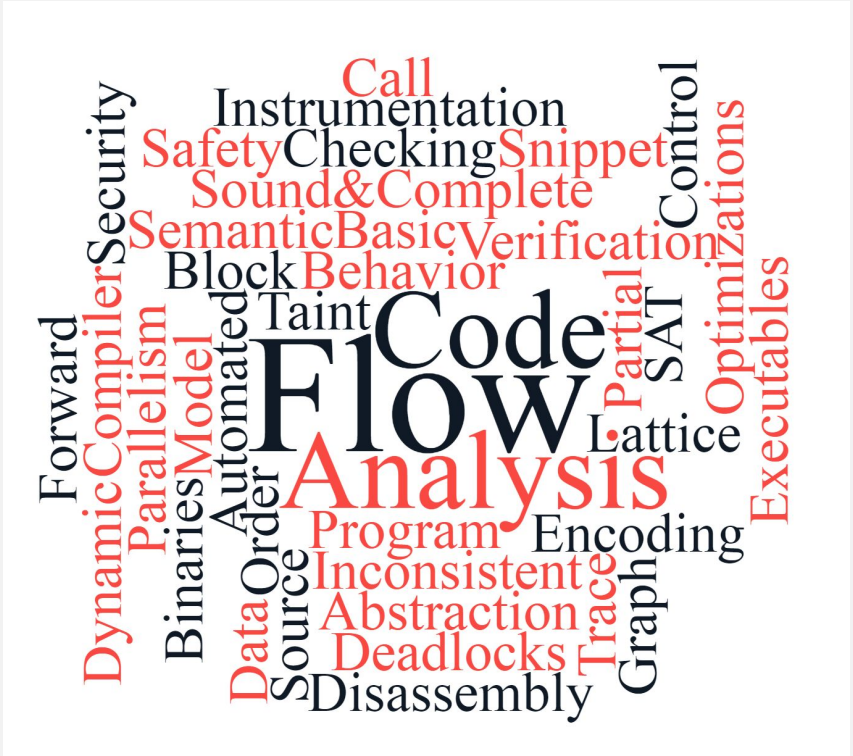
Software analysis

Software analysis encompasses a range of techniques, methods, and tools aimed at

- understanding,
- evaluating, and
- improving

software artifacts throughout the software development lifecycle.

It involves **static** and **dynamic** analysis approaches to identify software **defects**, **security vulnerabilities**, **performance bottlenecks**, and **compliance issues**.



Static analysis

IMT SCUOLA ALTI STUDI LUCCA

- Static analysis involves examining software artifacts **without executing the code**. It analyzes source code, configuration files, and documentation to identify potential defects, coding standards violations, and security vulnerabilities.

- **Techniques** include syntax checking, data flow analysis, control flow analysis, and abstract interpretation.

- **Tools**: Static analysis tools such as linters, static code analyzers, and code review platforms automate the process of identifying code issues and enforcing coding standards. Reverse engineering techniques are also used.

Abstract Syntax Tree (AST)

IMT SCUOLA ALTI STUDI LUCCA

An **Abstract Syntax Tree (AST)** is a tree representation of the syntactic structure of source code. Each node in the tree represents a construct occurring in the source code, such as an operator, variable, or control structure (e.g., loops, conditionals). ASTs are central to parsing code because they abstract away unnecessary syntactic details (e.g., parentheses or formatting) while retaining the code's logical structure.

Key Features of AST:

1. **Hierarchical Structure**:
   - The root node represents the entire program.
   - Internal nodes represent statements or expressions.
   - Leaf nodes represent identifiers, literals, or specific constructs.
2. **Semantics and Syntax**:
   - AST captures not only the syntactic structure but often includes semantic information like types or scopes.
3. **Language-Specific**:
   - ASTs are tailored to the grammar of the programming language being analyzed.
4. **Abstract**:
   - ASTs abstract away some details of the concrete syntax, focusing on logical constructs.

Abstract Syntax Tree (AST)

IMT SCUOLA ALTI STUDI LUCCA

Applications of AST:

- **Static Code Analysis**:
  - Detect bugs, vulnerabilities, and code smells.
- **Code Transformation**:
  - Used in code refactoring and optimization.
- **Code Comprehension**:
  - Useful for understanding program structure and behavior.
- **Machine Learning**:
  - Features from ASTs are used in code embeddings and ML models.
- **Compilers**:
  - ASTs are a fundamental step in compiling code, bridging parsing and code generation.

Tools for Extracting AST

IMT SCUOLA ALTI STUDI LUCCA

Different tools and libraries exist to generate and work with ASTs. These tools often depend on the target programming language. Here's a list of popular tools for some major languages:

1. Python

- `ast` **Module (Built-in)**:
  - Extracts ASTs from Python source code.

```
In [1]: import ast
   ...:
   ...: code = "x = 1 + 2"
   ...: tree = ast.parse(code)
   ...: print(ast.dump(tree, indent=2))
Module(
  body=[
    Assign(
      targets=[
        Name(id='x', ctx=Store())],
      value=BinOp(
        left=Constant(value=1),
        op=Add(),
        right=Constant(value=2)))],
  type_ignores=[])
```

Tools for Extracting AST

IMT SCUOLA ALTI STUDI LUCCA

2. Java

- **Eclipse JDT (Java Development Tools)**: Provides APIs to parse and analyze Java code.
- **JavaParser**: A library for generating and working with Java ASTs.
- **ANTLR**: A parser generator that can produce ASTs for custom grammars, including Java.

3. JavaScript

- **Esprima**: A lightweight JavaScript parser producing ASTs.
- **Acorn**: A fast JavaScript parser.
- **Tree-Sitter**: Also supports JavaScript for fast incremental parsing.

4. Multi-Language Tools

- **Tree-Sitter**: A fast and efficient parser generator supporting many languages (e.g., Python, Java, JavaScript, Rust).
- **ANTLR**: General-purpose parser generator for custom language grammars.

Control Flow Graph

IMT SCUOLA ALTI STUDI LUCCA

A **Control Flow Graph (CFG)** is a graphical representation of all possible execution paths through a program or a segment of code.

Each node in the graph represents a basic block—a sequence of instructions with a single entry and exit point. Directed edges between nodes represent the flow of control, such as branching due to conditionals or loops.

Nodes and Edges:

- **Nodes**: Represent basic blocks of code.
- **Edges**: Indicate the transfer of control between blocks (e.g., loops, conditionals, method calls).

Branching Logic:

- Shows how control flows through conditional statements (e.g., `if`, `while`).
- Represents loops, function calls, and exceptions.

Program Flow Representation:

- Helps visualize and analyze paths of execution within a program.

Control Flow Graph

IMT SCUOLA ALTI STUDI LUCCA

Applications of CFG:

1. **Static Code Analysis**:
   - Detect unreachable code, infinite loops, and potential bugs.
2. **Compiler Optimization**:
   - Identify redundant computations, optimize loops, and inline function calls.
3. **Program Understanding**:
   - Assist in visualizing execution paths for debugging or learning.
4. **Security Analysis**:
   - Vulnerability detection (e.g., buffer overflows, injection points).
5. **Test Case Generation**:
   - Identify paths through the program to create comprehensive test cases.

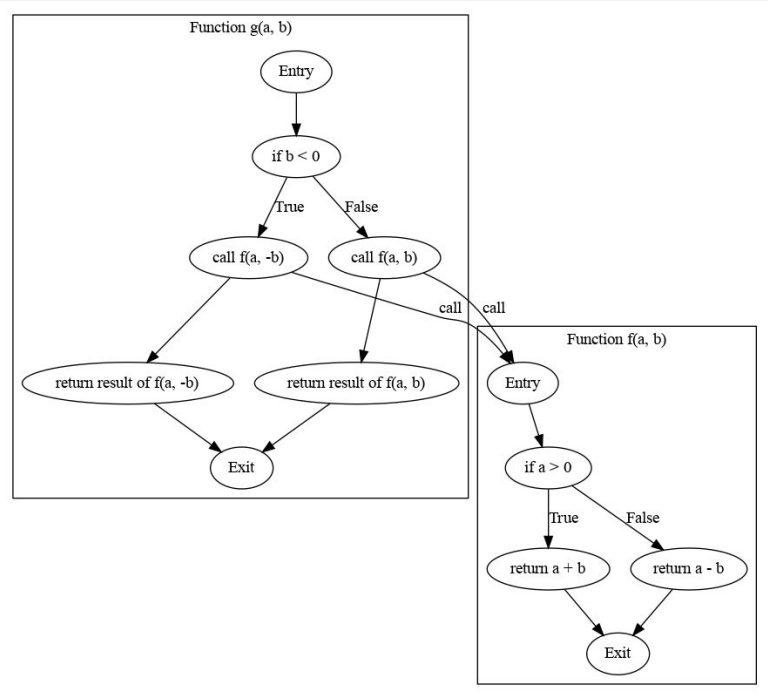
Control Flow Graph

IMT SCUOLA ALTI STUDI LUCCA

Workflow for CFG Generation

1. **Parse Code**:
   - Use a language-specific parser to convert the source code into an intermediate representation (e.g., AST or bytecode).
2. **Analyze Control Flow**:
   - Traverse the intermediate representation to identify control flow constructs (e.g., branches, loops).
3. **Construct the CFG**:
   - Represent each basic block as a node and each control transfer as an edge.
4. **Visualize or Process**:
   - Use graph libraries (e.g., `Graphviz`, `networkx`) to visualize or analyze the CFG.

Control Flow Graph




```python
def f(a,b):
    if a > 0:
        return a+b
    else:
        return a-b

def g(a,b):
    if b < 0:
        return f(a,-b)
    else:
        return f(a,b)
```

Dynamic Analysis

IMT SCUOLA ALTI STUDI LUCCA

Dynamic analysis involves analyzing software behavior during execution. It focuses on runtime properties, memory usage, performance characteristics, and error handling.

Techniques include profiling, memory analysis, code coverage analysis, and runtime monitoring.

Tools: Dynamic analysis tools such as profilers, debuggers, memory analyzers, and dynamic testing frameworks capture runtime information and diagnose performance issues, memory leaks, and runtime errors.

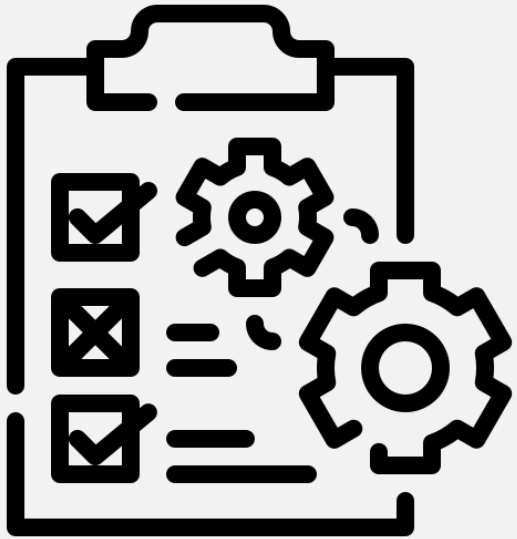
Model checking

IMT SCUOLA ALTI STUDI LUCCA

- Model checking is a **formal verification** technique used to systematically verify whether a software model **satisfies a set of desired properties or specifications**.

- It involves exhaustively exploring all possible states of a finite-state model to identify potential violations of safety and liveness properties.

- Tools: Model checking tools such as SPIN, NuSMV, and Alloy provide automated verification capabilities for concurrent and distributed systems.



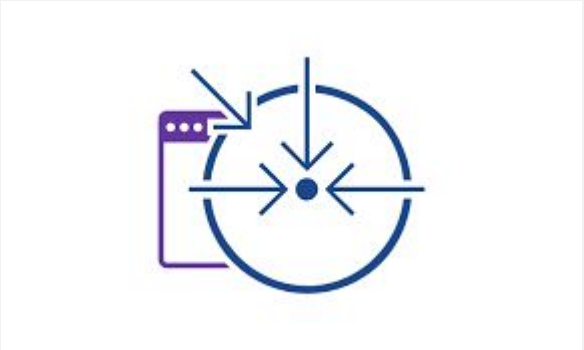
Fuzz testing

IMT SCUOLA ALTI STUDI LUCCA

- Fuzz testing, also known as fuzzing, involves providing invalid, unexpected, or random inputs to a software system to uncover vulnerabilities, crashes, and unexpected behaviors.

- It helps identify security vulnerabilities, memory corruption issues, and boundary condition errors in software applications.

- Tools: Fuzz testing frameworks such as AFL, Peach, and Radamsa automate the process of generating and executing diverse input data to stress-test software systems.

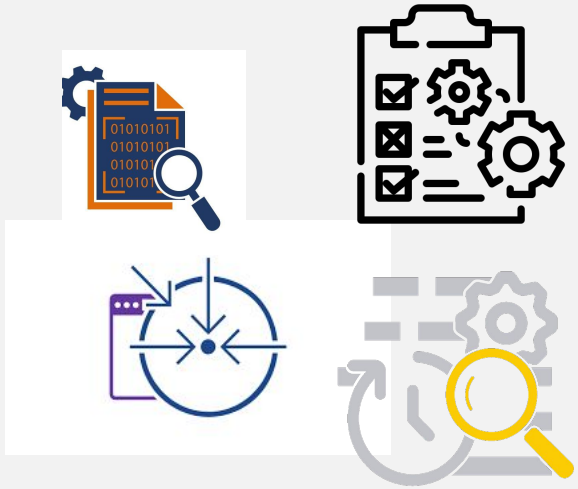


Software analysis

IMT SCUOLA ALTI STUDI LUCCA

Overall, program or software analysis plays a crucial role in ensuring software quality, reliability, and security throughout the software development lifecycle.

By employing a combination of static analysis, dynamic analysis, formal verification, and testing techniques, organizations can identify and mitigate software defects, vulnerabilities, and performance issues early in the development process.



Why machine learning for software analysis

IMT SCUOLA ALTI STUDI LUCCA

Handling Complexity

Modern software systems are highly complex, often consisting of **millions of lines of code** with intricate **dependencies** and **interactions**.

Machine learning algorithms excel at processing **large volumes** of data and identifying **complex patterns**, making them well-suited for **analyzing software systems of varying scales and complexities**.

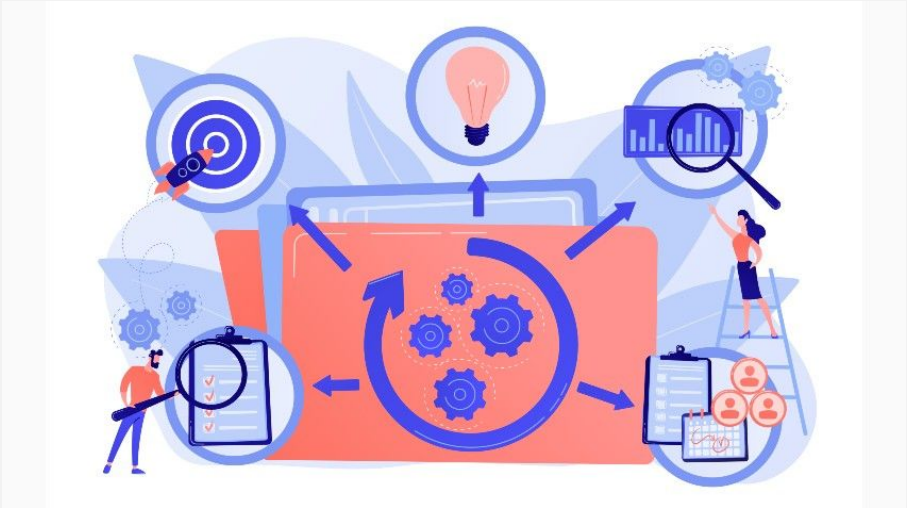
Automating Analysis Tasks

Traditional program analysis techniques often involve **manual inspection**, which can be **time-consuming** and **error-prone**, especially for large codebases.

Machine learning enables the **automation** of various program analysis tasks:

- bug detection,
- code optimization, and
- software testing,

allowing developers to focus on higher-level design and problem-solving.



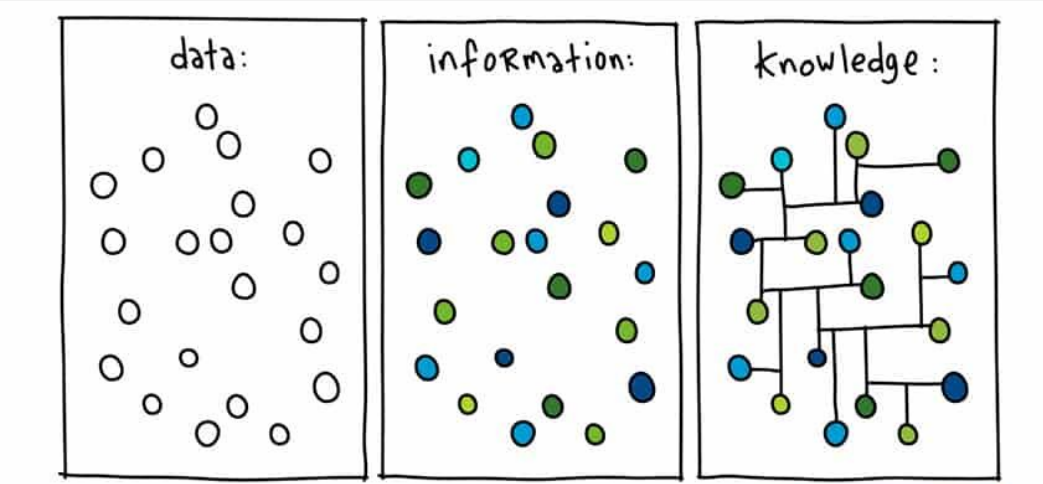
Why machine learning for software analysis

IMT SCUOLA ALTI STUDI LUCCA

Learning from Data

Machine learning algorithms **learn from data**, which is abundant in software development projects.

By training on historical **code repositories**, **bug reports**, **version control histories**, and **user feedback**, machine learning models can capture valuable insights and patterns that **may not be apparent** through **manual analysis** alone.



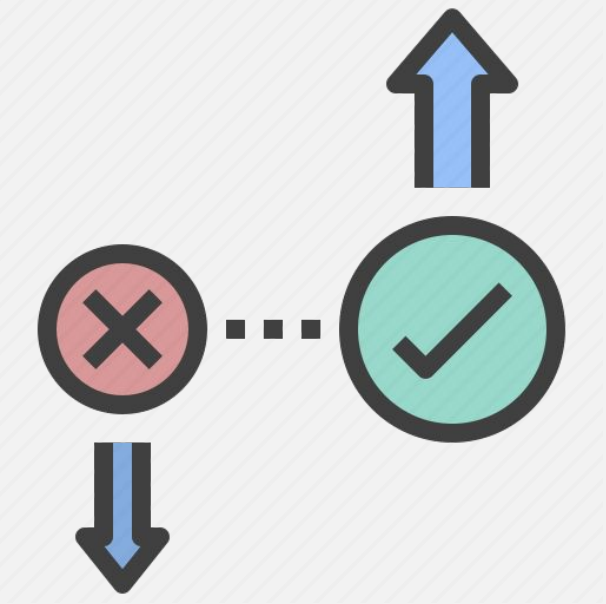
Why machine learning for software analysis

IMT SCUOLA ALTI STUDI LUCCA

Improving Accuracy and Efficiency

Machine learning techniques can enhance the accuracy and efficiency of program analysis by identifying **subtle patterns** and **anomalies** in software behavior that may **be difficult for humans to detect**.

Additionally, machine learning models can **prioritize analysis efforts** by focusing on the most critical areas of the codebase, thereby improving overall development productivity.

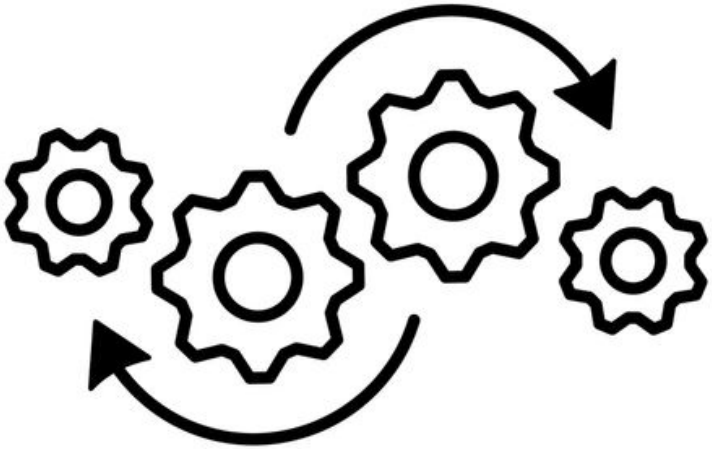


Why machine learning for software analysis

IMT SCUOLA ALTI STUDI LUCCA

Adapting to Change

Software systems evolve over time in response to changing requirements, technologies, and user needs. Machine learning models can adapt and evolve alongside software systems, continuously learning from new data and feedback to improve their performance and relevance in dynamic environments.



Why machine learning for software analysis

IMT SCUOLA ALTI STUDI LUCCA

Predictive Capabilities

Machine learning enables predictive analytics in program analysis, allowing developers to anticipate and mitigate potential issues before they arise. For example, machine learning models can forecast software defects, identify performance bottlenecks, and predict code maintainability based on historical data and patterns.

IMT SCUOLA ALTI STUDI LUCCA

Supporting Decision-Making

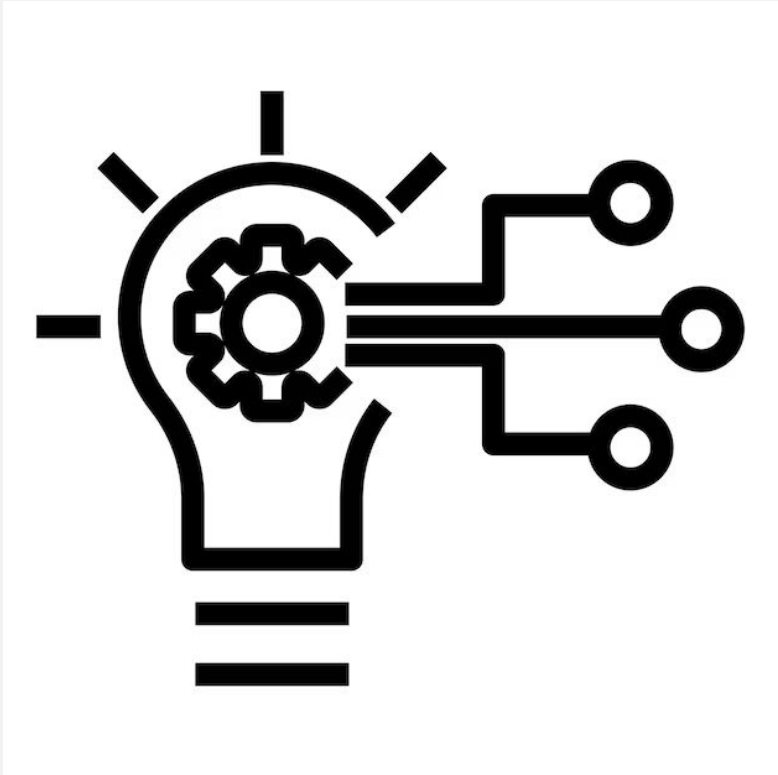
Machine learning provides valuable insights and recommendations to support decision-making processes in software development. By analyzing software metrics, code patterns, and user behavior, machine learning models can inform decisions related to resource allocation, feature prioritization, and software architecture design.

Why machine learning for software analysis

IMT SCUOLA ALTI STUDI LUCCA

Enabling Innovation

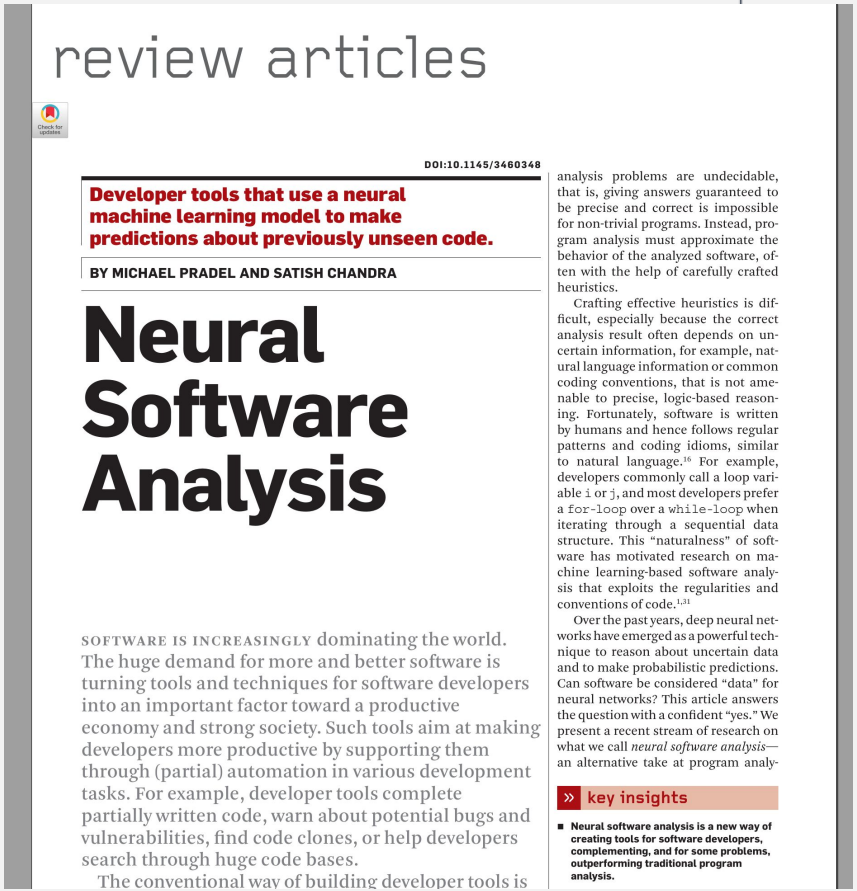
The application of machine learning in program analysis opens up new avenues for innovation in software development practices. From automated code generation to intelligent debugging tools, machine learning enables developers to explore novel approaches and solutions to complex software engineering challenges.



Neural Software Analysis

IMT SCUOLA ALTI STUDI LUCCA

Neural Software Analysis
M. PRADEL and S. CHANDRA
Communication of the ACM, January 2022



review articles

DOI:10.1145/3460348

Developer tools that use a neural machine learning model to make predictions about previously unseen code.

BY MICHAEL PRADEL AND SATISH CHANDRA

Neural Software Analysis

SOFTWARE IS INCREASINGLY dominating the world. The huge demand for more and better software is turning tools and techniques for software developers into an important factor toward a productive economy and strong society. Such tools aim at making developers more productive by supporting them through (partial) automation in various development tasks. For example, developer tools complete partially written code, warn about potential bugs and vulnerabilities, find code clones, or help developers search through huge code bases.

The conventional way of building developer tools is analysis problems are undecidable, that is, giving answers guaranteed to be precise and correct is impossible for non-trivial programs. Instead, program analysis must approximate the behavior of the analyzed software, often with the help of carefully crafted heuristics.

Crafting effective heuristics is difficult, especially because the correct analysis result often depends on uncertain information, for example, natural language information or common coding conventions, that is not amenable to precise, logic-based reasoning. Fortunately, software is written by humans and hence follows regular patterns and coding idioms, similar to natural language.[16] For example, developers commonly call a loop variable `i` or `j`, and most developers prefer a `for-loop` over a `while-loop` when iterating through a sequential data structure. This "naturalness" of software has motivated research on machine learning-based software analysis that exploits the regularities and conventions of code.[1,31]

Over the past years, deep neural networks have emerged as a powerful technique to reason about uncertain data and to make probabilistic predictions. Can software be considered "data" for neural networks? This article answers the question with a confident "yes." We present a recent stream of research on what we call *neural software analysis*—an alternative take at program analy-

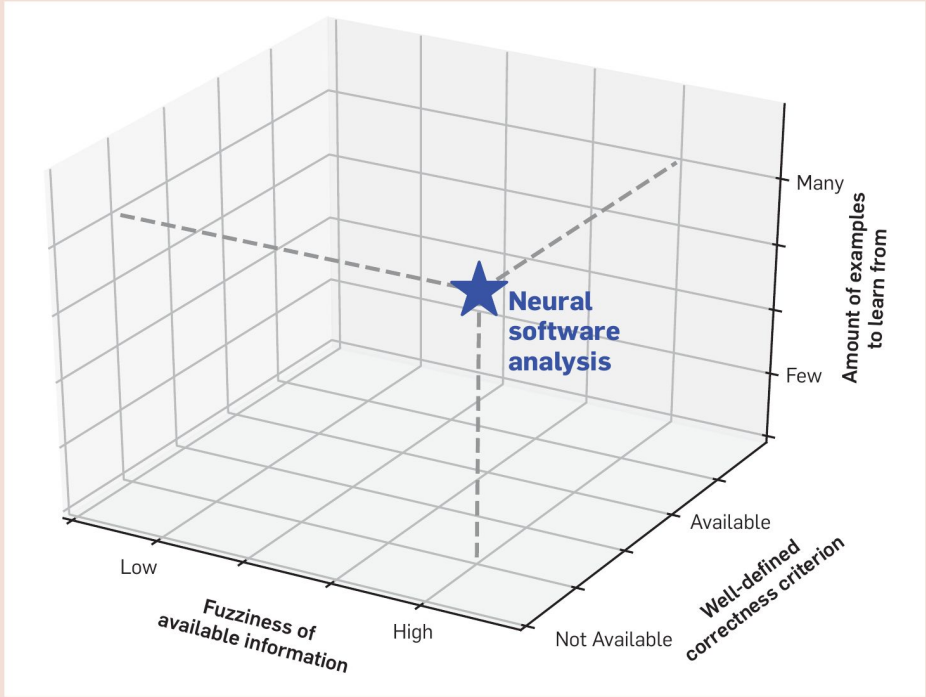
» key insights

- **Neural software analysis is a new way of creating tools for software developers, complementing, and for some problems, outperforming traditional program analysis.**

When ML is useful

IMT SCUOLA ALTI STUDI LUCCA

Figure 1. Three dimensions to determine whether to use neural software analysis.



Dimension 1: **From** “If the code has property A, then B holds”
to “If the code is like pattern A, then B is likely to hold.”

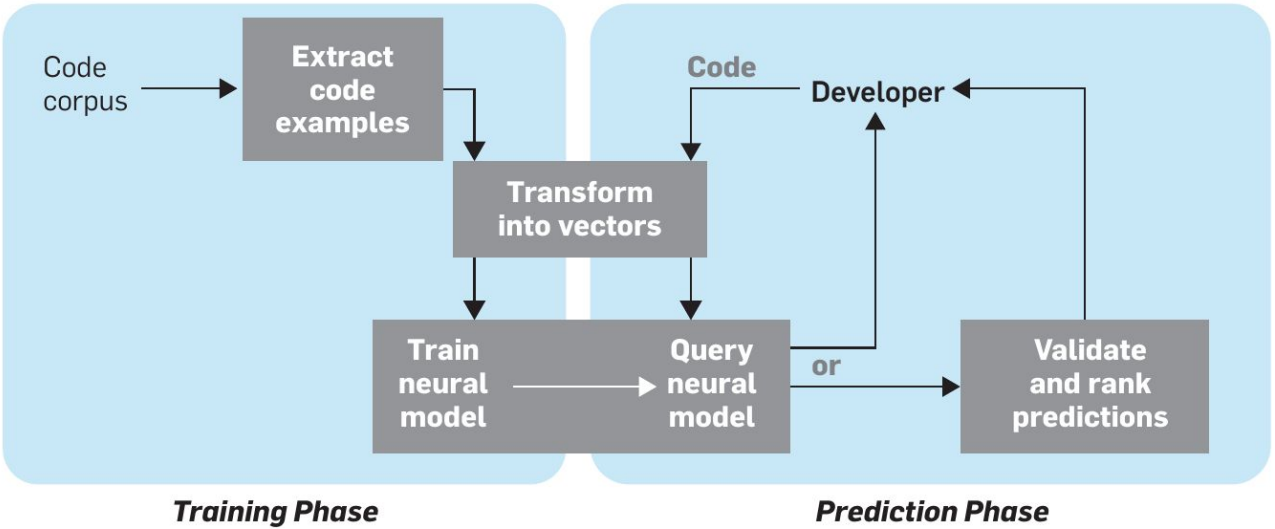
Dimension 2: Neural software analysis often outperforms traditional analysis for problems that lack a well-defined correctness criterion

Dimension 3: Neural models are data- hungry, and hence, neural software analysis works best if there are plenty of examples to learn from

Pipeline

IMT SCUOLA ALTI STUDI LUCCA

Figure 2. Typical components of a neural software analysis.



Pipeline description

IMT SCUOLA ALTI STUDI LUCCA

Code Example Extraction:

- A code corpus is used to extract examples relevant to the analysis problem.

Code Transformation:

- Extracted code examples are transformed into vectors using:
  - Intermediate representations (e.g., token sequences or abstract syntax trees).
  - Novel code representations designed for learning-based analysis.

Model Training:

- Examples serve as training data for a neural model.
- This training phase occurs once, before deployment.

Prediction Phase:

- Developers query the trained neural model with unseen code examples.
- The model generates predictions for the queries.

Optional Validation and Ranking:

- Predictions can be validated and ranked before being presented.
- Validation and ranking may use traditional program analysis to combine neural and logic-based reasoning.

Extracting code example

IMT SCUOLA ALTI STUDI LUCCA

- **Lightweight Static Analysis**:
  - Most neural software analyses rely on *lightweight static analysis* tools.
  - These tools include tokenizers (split code into tokens) and parsers (create abstract syntax trees or ASTs).
  - Benefits of lightweight analysis:
    - **Scalability**: Efficiently handles large code corpora for effective training.
    - **Portability**: Easily adapts analysis tools across different programming languages.
- **Obtaining Labeled Examples**:
  - Neural software analysis primarily uses supervised learning, requiring labeled code examples.
  - Labeled examples include:
    - **Type Prediction**: Learns from existing type annotations.
    - **Code Edits Prediction**: Learns from version control system edit histories.
  - **Impact on Research Focus**:
    - Availability of large, annotated datasets influences the tasks studied in the neural software analysis community.

Representing software as Vectors

IMT SCUOLA ALTI STUDI LUCCA

Representing Software as Vectors:

1. **Key Design Decision**:
    - Convert code examples into vector representations.
    - Two aspects:
        - Representing basic building blocks (e.g., individual code tokens).
        - Composing these representations into larger code snippets (e.g., statements or functions).

Representing software as vectors

IMT SCUOLA ALTI STUDI LUCCA

Representing Code Tokens:

1. **Techniques**:
   - **Abstract Non-Standard Tokens**:
     - Replace variable names with generic labels (e.g., `var1`, `var2`).
     - Reduces vocabulary size but discards useful information.
   - **Token Embedding**:
     - Map each token to a fixed-size vector.
     - Aim to make semantically similar tokens (e.g., `len` and `size`) close in vector space.
   - **Embedding Methods**:
     - Pre-train token embeddings before training the neural model.
     - Jointly train token embedding and neural model.
2. **Challenges**:
   - Vocabulary growth due to unique identifiers (e.g., variable/function names) increases linearly with project size.
   - Solution:
     - Fix vocabulary to the top 10,000 most common tokens; represent others with an "unknown" vector.
     - Split tokens into **subwords** (e.g., `writeFile` → `write`, `file`), represent subwords individually, and combine them.

Representing software as vectors

IMT SCUOLA ALTI STUDI LUCCA

Representing Snippets of Code:

1. **Sequence-Based Techniques**:
   - Map tokens into sequences of vectors.
   - Example: Code snippet `x=true;` → vectors for `x`, `=`, `true`, `;`.
   - Utilize Abstract Syntax Trees (ASTs) to extract paths and map nodes into vectors (e.g., Code2vec).
2. **Graph-Based Techniques**:
   - Represent code snippets as graphs of vectors, typically derived from ASTs.
   - Include additional edges for relationships like data flow, control flow, etc.
   - Advantages:
     - Capture rich structural and semantic information.
   - Disadvantages:
     - Less portable across programming languages.
     - Computationally expensive compared to sequence-based models.

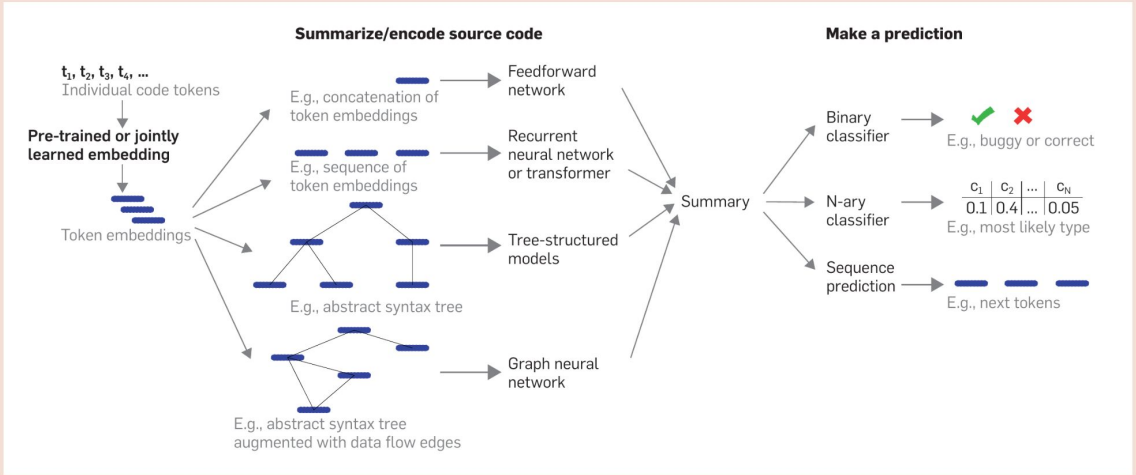
Neural models of software

IMT SCUOLA ALTI STUDI LUCCA

Neural models consist of two parts:

- **Summarization/Encoding**: Compactly represents the source code.
- **Prediction**: Uses the summary to make predictions about the code.

Figure 3. Neural components popular for analyzing software. In principle, any of the summarization components on the left can be freely combined with any of the prediction components on the right.



Summarizing

IMT SCUOLA ALTI STUDI LUCCA

Methods:

1. **Concatenation**: Combine vectors representing the code and reduce them to a shorter vector using a feedforward network.
2. **Recurrent Neural Networks (RNNs)**: Traverse sequences (e.g., token vectors) to summarize them.
3. **Transformers**: Use learned attention to selectively focus on elements of the sequence.
4. **Tree-Structured Models**: Summarize tree representations, such as Abstract Syntax Trees (ASTs).
5. **Graph Neural Networks (GNNs)**:
    - Operate on graph representations of code.
    - Repeatedly update nodes based on neighboring node representations, propagating information across code elements.

Making predictions

IMT SCUOLA ALTI STUDI LUCCA

1. **Classification Models**:
   - **Binary Classifier**: Predicts whether code is correct or buggy.
   - **N-ary Classifier**: Predicts the class of code from a set of N classes (e.g., variable types).
   - Outputs a **probability distribution** over classes using the softmax function.
2. **Sequence Predictions**:
   - Tasks include predicting code edits or *generating natural language descriptions of code*.
   - **Encoder-Decoder Models**:
     - Combine a sequence encoder with a decoder for sequence predictions.
     - Decoder outputs probabilities for each token in the sequence.
3. **Next-Token Prediction**:
   - A classic task for language models.
   - Uses a **decoder-only model**, which encodes the sequence state in the decoder.

Training & querying

IMT SCUOLA ALTI STUDI LUCCA

1. **Training**:
   - Optimize model parameters (weights and biases) using examples of input-output pairs.
   - Use a loss function (e.g., cross entropy) to compare predictions with expected outputs.
   - Minimize loss using **stochastic gradient descent**.
2. **Querying**:
   - Predict outputs for unseen code by generalizing from training examples.

Ranking

IMT SCUOLA ALTI STUDI LUCCA

Methods:

1. Use numeric vectors to identify predictions with the highest confidence.
2. For classification models:
    - Interpret softmax outputs as probability distributions.
    - Rank classes by predicted probability.
3. For encoder-decoder models:
    - Use **beam search** to find the k most likely predictions for complex outputs (e.g., token sequences)

Three examples

IMT SCUOLA ALTI STUDI LUCCA

Table 2. Three neural software analyses and how they map onto the conceptual framework in Figure 2.

Component of conceptual framework	Neural software analyses		
	Bug detection (DeepBugs)	**Type prediction (TypeWriter)**	**Code completion**
Code corpus	JavaScript (68M lines of open-source code)	Python (2.7M lines of open-source code and a larger commercial corpus)	Python (16M lines of open source code)
Extraction of code examples	Code snippets as-is and with artificially introduced bugs	Functions with their parameter and return types	Code token sequences, offset by one for next token prediction
Transformation into vectors	Concatenation of token embeddings and context information	Token embeddings for code, word em- beddings for comments	End-to-end learned token embeddings for code
Neural model	Simple feedforward model	Hierarchical model built from several recurrent neural networks	Encoder (bi-directional LSTM) and decoder (LSTM)
Validation and ranking	Rank warnings by predicted probability that code is buggy	Search and validate correct types with type checker	Rank output tokens by probability that it is the next token

Three examples

IMT SCUOLA ALTI STUDI LUCCA

Figure 4. Python implementation of a tic-tac-toe game.

```
1  class Board:
2     TypeWriter infers the function signature
      (Board,  int,  int,  str)  ->  Bool
4    def mark_point(self, x, y, player_name):
5      """
6        Marks the given point on the board
7        as chosen by the given player.
8        Returns whether the move gives the player
9        three marked fields in a row.
10     """
11     self.field[x][y] = player_name
12     has_three_in_a_row = False
13     ... # compute whether the player has won
14     return has_three_in_a_row
15
16   def show_winner(self, player_name):
17     ...
18
19  while not game_done:
20    active_player = ...
21    x = ...
22    y = ...
23     DeepBugs warns about a bug here:
24     has_won = board.mark_point(y, x, active_player)
25     if has_won:
26       # notify player
27       game_done = True
28       board.??? Neural model suggests completions here
```

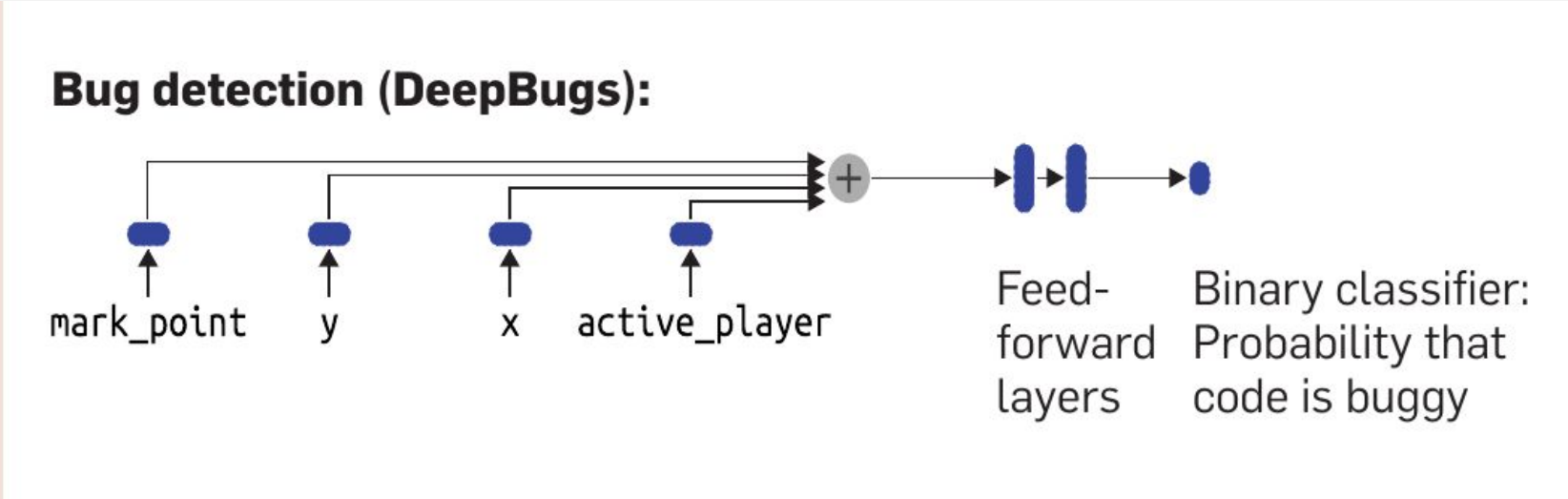
Type inference

Bug detection

Code completion

Learning for finding bugs

IMT SCUOLA ALTI STUDI LUCCA



Learning for finding bugs

IMT SCUOLA ALTI STUDI LUCCA

DeepBugs: A Learning-Based Bug Detection Framework

Overview:

1. Tackles bug detection as a **classification problem** (predicts whether code is correct or buggy).
2. Exploits implicit information in **natural language identifiers**, often ignored by traditional program analyses.
3. Learning-based approach is suitable due to:
    - Fuzziness of identifier information.
    - Difficulty in determining correctness without human input.

Ranking

IMT SCUOLA ALTI STUDI LUCCA

Extracting Code Examples:

1. Focuses on specific statements and bug patterns (e.g., **swapped argument bugs**).
2. Correct examples:
   - Assumes most code is correct.
   - Extracts function calls with at least two arguments.
3. Incorrect examples:
   - Artificially injects bugs (e.g., swapping arguments in function calls).
   - Generic framework supports additional bug patterns.

Ranking

IMT SCUOLA ALTI STUDI LUCCA

Transformation into Vectors:

1. Represents code examples as concatenated vectors including:
   - **Natural language identifiers** (e.g., function and argument names).
   - **Contextual information** (e.g., AST ancestor nodes, operators).
2. Uses **pre-trained Word2Vec embeddings** to generalize across similar identifiers (e.g., `x` and `y`).

Neural Model:

3. **Simple Feedforward Neural Network**:
   - Concatenates input embeddings and predicts a probability (`p`) that the code is buggy.
   - Trained to predict:
     - `p = 0.0` for correct code.
     - `p = 1.0` for artificially injected bugs.
4. Makes predictions on unseen code based on features extracted during training.
5. Reports warnings if the predicted probability exceeds a threshold.

Ranking

IMT SCUOLA ALTI STUDI LUCCA

Validation and Ranking:

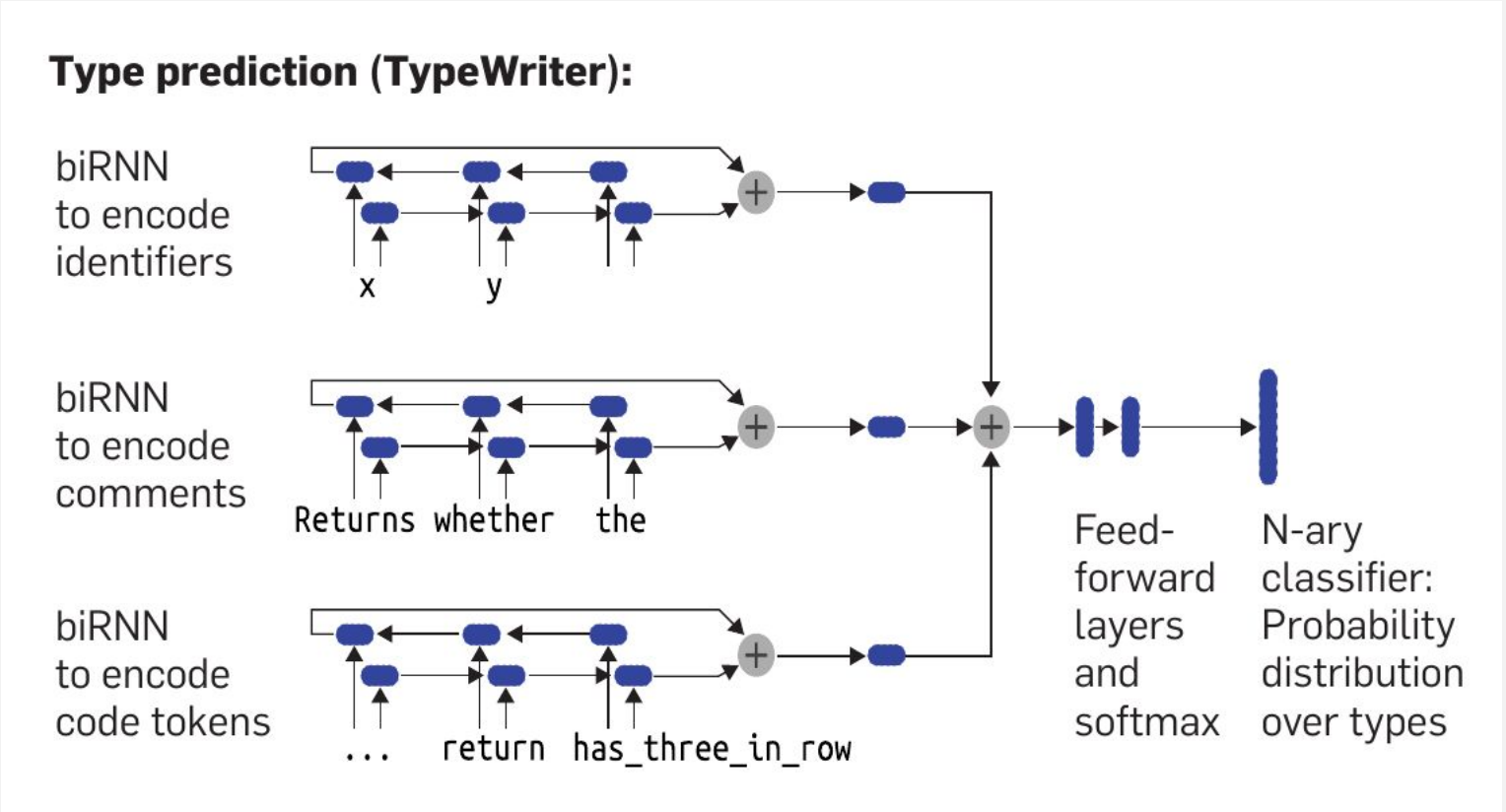
- Does not validate potential bugs further before reporting warnings.
- **Ranking**:
  - Sorts warnings by the predicted probability (p).
  - Developers inspect bugs starting from the most likely.

Applications:

- DeepBugs-inspired tools are available as plugins for **JetBrains IDEs**.
- Supports JavaScript and Python, with thousands of downloads by developers.

Learning to predict types

IMT SCUOLA ALTI STUDI LUCCA



Learning to predict types

IMT SCUOLA ALTI STUDI LUCCA

TypeWriter: A Neural Software Analysis for Type Prediction

Overview:

- Predicts **type annotations** for dynamically typed languages like Python or JavaScript.
- Adds type annotations to help ensure correctness, facilitate maintenance, and improve IDE support.
- Fits neural software analysis:
  - **Dimension 1**: Source code hints (e.g., variable names, comments, usage patterns) are fuzzy.
  - **Dimension 2**: No universal criterion for correct type predictions.
  - **Dimension 3**: Large datasets with existing type annotations provide ample training data.

Extracting Code Examples

IMT SCUOLA ALTI STUDI LUCCA

Extracting Code Examples

- Gathers two types of information from Python code:
  - **Natural Language Information**: Function argument names, associated comments.
  - **Programming Language Information**: Usage patterns (e.g., `return` statements).
- Example:
  - From the `mark_point` function:
    - Extracts the return statement, variable name, and comment to infer the Boolean return type.
- Uses existing type annotations as ground truth for training.

Transformation into Vectors:

- **Code Tokens and Identifiers**:
  - Represented using pre-trained embeddings specific to Python code.
- **Comments**:
  - Mapped as word sequences using a pre-trained word embedding.
  - Trained on comments from the code corpus for relevant vocabulary.

Ranking

IMT SCUOLA ALTI STUDI LUCCA

Neural Model:

- **Architecture**:
  - Three recurrent neural networks (RNNs):
    - One for identifiers.
    - One for code tokens.
    - One for natural language words (e.g., comments).
  - Outputs of RNNs are concatenated into a vector.
  - Vector passed to a feedforward classifier to predict a probability distribution over the 1,000 most common types.
- **Output**:
  - Ranked list of possible types, with the highest probability type at the top.

Ranking

IMT SCUOLA ALTI STUDI LUCCA

Validation and Ranking:

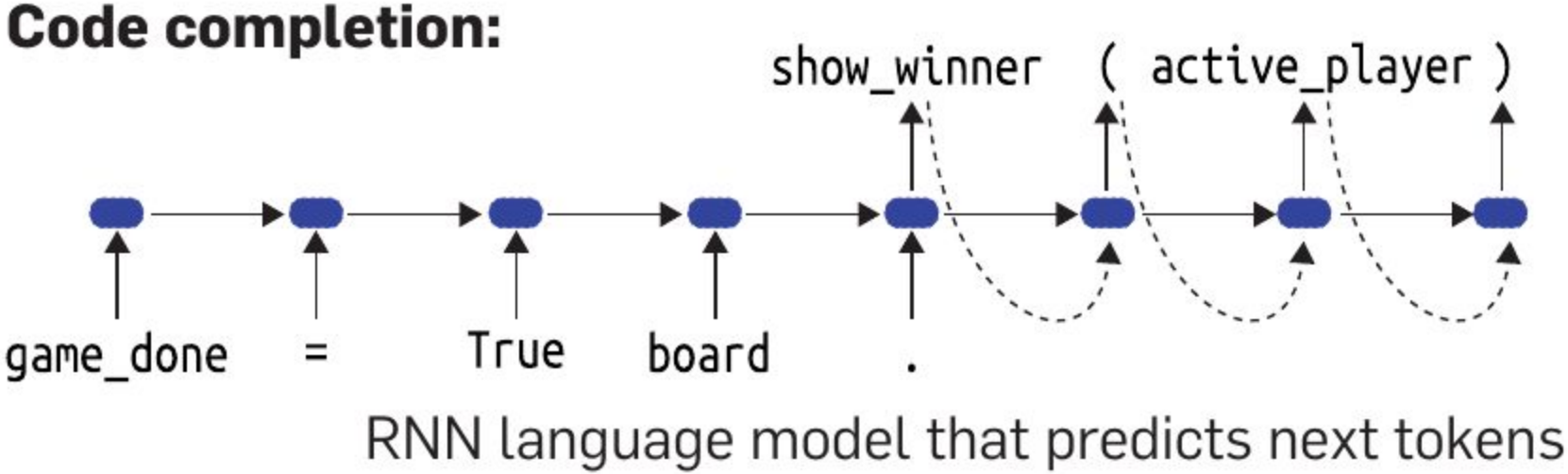
- **Validation**:
  - Uses a **gradual type checker** to ensure that suggested types do not introduce type errors.
  - Type checker validates predicted types, ensuring consistency with existing annotations.
- **Ranking**:
  - Treats type assignment as a combinatorial search problem to minimize type errors while adding as many annotations as possible.
  - Combines neural analysis predictions with traditional program analysis for validation.

Impact:

- Developed by Facebook to add type annotations to Python codebases.
- Has successfully added thousands of type annotations to software used by billions of users.

Learning to complete partial code

IMT SCUOLA ALTI STUDI LUCCA



Learning to complete partial code

IMT SCUOLA ALTI STUDI LUCCA

Neural Code Completion

Overview:

- Predicts the next token at a cursor position as a developer types.
- Produces a **probability distribution** over potential tokens; top 5–10 most likely tokens are shown.
- Well-suited to neural software analysis:
  - **Dimension 1**: Fuzzy information; no deterministic rules for token prediction.
  - **Dimension 2**: No well-defined correctness criterion (other than compilability).
  - **Dimension 3**: Abundant training data from existing codebases.

Ranking

IMT SCUOLA ALTI STUDI LUCCA

Extracting Code Examples:

- Extracts preceding tokens (context) and the immediate next token from a code corpus.
- Example:
  - Context: `<game_done, =, True, board, .>`
  - Expected prediction: `show_winner`.

Transformation into Vectors:

- Tokens mapped to indices in a fixed-size vocabulary.
- Out-of-vocabulary tokens represented by a special "unknown" index.
- Sequences padded to a uniform length using a padding token.

Ranking

IMT SCUOLA ALTI STUDI LUCCA

Neural Model:

1. **RNN-Based Model**:
    - Uses **recurrent neural networks (RNNs)** to condition predictions on preceding tokens.
    - At each step:
        - Input: Hidden vector summarizing prior context + embedding of the current token.
        - Output: Probability distribution over the vocabulary for the next token.
    - Loss function: Negative log-likelihood of the expected token at each step.
    - Embeddings learned during training in an end-to-end manner.
2. **Transformer-Based Model**:
    - Uses the **transformer architecture** (e.g., GPT-2) for token prediction.
    - Addresses RNN limitations in remembering long-term context.

Validation and Ranking:

- Displays a ranked list of top predictions (e.g., top 5 tokens).
- **Heuristics**:
    - Project-specific API usage tweaks the ranked list.
- For predicting multiple tokens (e.g., full lines of code), uses **beam search**.

Ranking

IMT SCUOLA ALTI STUDI LUCCA

Advances and Challenges:

1. **Out-of-Vocabulary Tokens**:
    - Addressed by:
        - Splitting identifiers into subwords.
        - Copying tokens from the context using attention mechanisms.
2. **Memory Limitations**:
    - Overcome by transformer models, which better handle long-term context.
3. **Multi-Token Predictions**:
    - Solved using beam search to predict entire token sequences.

Industry Applications:

- Used in:
    - **TabNine**: AI-powered autocompletion.
    - **Facebook**: Internal tools.
    - Popular IDEs:
        - JetBrains IntelliJ.
        - Microsoft Visual Studio IntelliCode.

To Read on code completion

IMT SCUOLA ALTI STUDI LUCCA

18. Karampatsis, R., Babii, H., Robbes, R., Sutton, C., and Janes, A.
Big code = big vocabulary: Open-vocabulary models for source code.
In Proceedings of 42nd Intern. Conf. on Softw. Eng. https://doi.org/10.1145/3377811.3380342

19. Kim, S., Zhao, J., Tian, Y., and Chandra, S.
Code prediction by feeding trees to transformers.
In Proceedings of IEEE/ACM Intern. Conf. on Softw. Eng.

Li, J., Wang, Y., Lyu, M., and King, I.
Code completion with neural attention and pointer networks.
In Proceedings of the 27th Intern. Joint Conf. on Artificial Intelligence. AAAI Press,
4159–25.

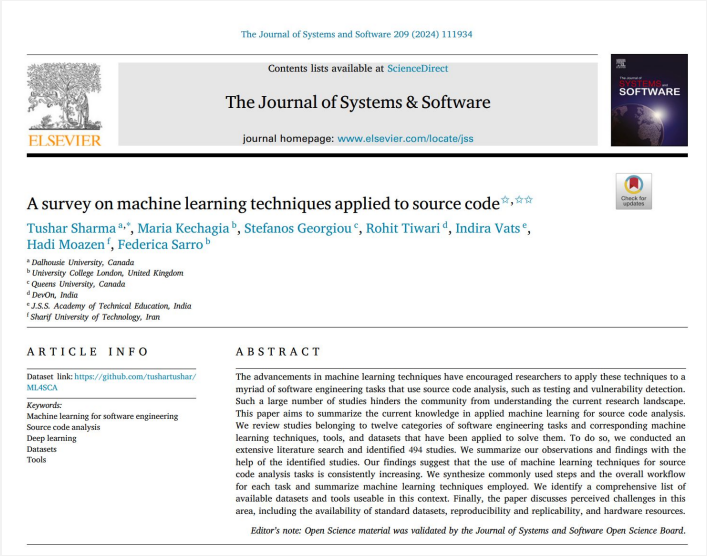
34. Svyatkovskiy, A., Deng, S., Fu, S., and Sundaresan,N.
IntelliCode compose: Code generation using transformer.
In Proceedings of the 28th ACM Joint European Softw. Eng. Conf. and Symp. Foundations of Softw. Eng. https://doi.org/10.1145/3368089.3417058

Where actually is machine learning applied for Software Analysis

A survey on machine learning techniques applied to source code
Tushar Sharma, Maria Kechagia, Stefanos Georgiou, Rohit Tiwari, Indira Vats, Hadi Moazen, Federica Sarro
Journal of Systems and Software, 2024

Reports that:

- The use of ML techniques is constantly increasing for source code analysis.
- A wide range Software Engineering (SE) tasks involving source code analysis use ML.
- The study identifies challenges in the field and potential mitigations.
- They identify commonly used datasets and tools used in the field.



The Journal of Systems and Software 209 (2024) 111934

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A survey on machine learning techniques applied to source code ☆, ☆☆

Tushar Sharma [a,*], Maria Kechagia [b], Stefanos Georgiou [c], Rohit Tiwari [d], Indira Vats [e], Hadi Moazen [f], Federica Sarro [b]

[a] Dalhousie University, Canada
[b] University College London, United Kingdom
[c] Queens University, Canada
[d] DevOn, India
[e] J.S.S. Academy of Technical Education, India
[f] Sharif University of Technology, Iran

ARTICLE INFO

Dataset link: https://github.com/tushartushar/ML4SCA

Keywords:
Machine learning for software engineering
Source code analysis
Deep learning
Datasets
Tools

ABSTRACT

The advancements in machine learning techniques have encouraged researchers to apply these techniques to a myriad of software engineering tasks that use source code analysis, such as testing and vulnerability detection. Such a large number of studies hinders the community from understanding the current research landscape. This paper aims to summarize the current knowledge in applied machine learning for source code analysis. We review studies belonging to twelve categories of software engineering tasks and corresponding machine learning techniques, tools, and datasets that have been applied to solve them. To do so, we conducted an extensive literature search and identified 494 studies. We summarize our observations and findings with the help of the identified studies. Our findings suggest that the use of machine learning techniques for source code analysis tasks is consistently increasing. We synthesize commonly used steps and the overall workflow for each task and summarize machine learning techniques employed. We identify a comprehensive list of available datasets and tools useable in this context. Finally, the paper discusses perceived challenges in this area, including the availability of standard datasets, reproducibility and replicability, and hardware resources.

Editor's note: Open Science material was validated by the Journal of Systems and Software Open Science Board.

Search terms and corresponding relevant studies found in the second round of phase 1.

Category	Search terms	#Studies
Vulnerability analysis	Feature learning in source code	9
	Vulnerability prediction in source code using machine learning	70
	Deep learning-based vulnerability detection	8
	Malicious code detection with machine learning	45
Testing	Word embedding in software testing	2
	Automated Software Testing with machine learning	12
	Optimal machine learning based random test generation	1
Refactoring	Source code refactoring prediction with machine learning	39
	Automatic clone recommendation with machine learning	14
	Machine learning based refactoring detection tools	16
	Search-based refactoring with machine learning	6
Quality assessment	Web service anti-pattern detection with machine learning	25
	Code smell prediction models	34
	Machine learning-based approach for code smells detection	17
	Software design flaw prediction	37
	Linguistic smell detection with machine learning	2
	Software defect prediction with machine learning	66
	Machine learning based software fault prediction	35
Program synthesis	Automated program repair methods with machine learning	45
	Program generation with machine learning	2
	Object-oriented program repair with machine learning	15
	Predicting patch correctness with machine learning	3
	Multihunk program repair with machine learning	9
Program comprehension	Autogenerated code with machine learning	6
	Commits analysis with machine learning	34
	Supplementary bug fixes with machine learning	9
Code summarization	Automatic source code summarization with machine learning	43
	Automatic commit message generation with machine learning	19
	Comments generation with machine learning	11
Code review	Security flaws detection in source code with machine learning	20
	Intelligent source code security review with machine learning	2
Code representation	Design pattern detection with machine learning	10
	Human-machine-comprehensible software representation	1
	Feature learning in source code	6
Code completion	Missing software architectural tactics prediction with machine learning	1
	Software system quality analysis with machine learning	6
	Package-level tactic recommendation generation in source code	3
	Identifier prediction in source code	13
	Token prediction in source code	29

IMT SCUOLA ALTI STUDI LUCCA

MLSA Pipeline

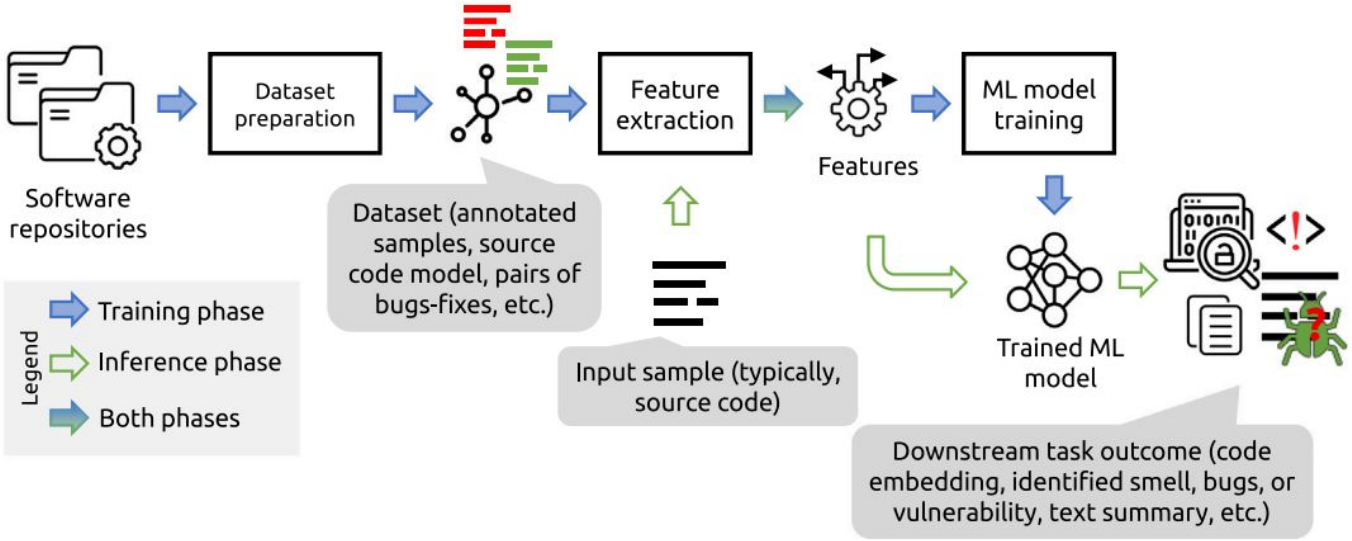


Fig. 7. Overview of the software engineering task implementation pipeline using ML.

ML Analysis tasks

IMT SCUOLA ALTI STUDI LUCCA

1. **Code representation**
2. Testing
   a. Test data and test cases generation
3. Program synthesis
   a. Program repair
   b. Code generation
   c. Program Translation
4. Quality Assessment
   a. Code smell detection
   b. Code clone detection
   c. Defect prediction
   d. Quality assessment/prediction
5. **Code completion**
6. Program comprehension
   a. Code summarization
   b. Program classification
   c. Change analysis
   d. Entity identification/recommendation
7. Code review
8. Code search
9. Refactoring
10. Vulnerability analysis

Code auto-completion

IMT SCUOLA ALTI STUDI LUCCA

Dataset Preparation:

1. **Custom Datasets**:
   - **Gopalakrishnan et al. (2017a)**: Analyzed 116,000 open-source systems for correlations between topics and developer tactics.
   - **Han et al. (2009, 2011)**: Sampled 4919 source code lines from open-source projects.
   - **Raychev et al. (2016)**: Used GitHub codebases for JavaScript and Python predictions.
   - **Svyatkovskiy et al. (2019)**: Evaluated their **Pythia** approach using 2700 Python repositories.
2. **Existing Datasets**:
   - **Rahman et al. (2020)**: Data from Aizu Online Judge (AOJ).
   - **Liu et al. (2020c, 2020d)**: Evaluated on three real-world datasets.
   - **Schuster et al. (2021)**: Public GitHub archive from 2020.

Code auto-completion

IMT SCUOLA ALTI STUDI LUCCA

Feature Extraction:

1. **Source Code Information**:
    - Relationships between topics and developer tactics (e.g., **Gopalakrishnan et al., 2017a**).
    - Hierarchical structural information and long-term dependencies (e.g., **Liu et al., 2020c,d**).
    - Locally repeated terms for out-of-vocabulary (OoV) handling (e.g., **Li et al., 2018**).
    - AST-based predictions (e.g., **Raychev et al., 2016**).
    - **Pythia**: Large-scale DL model trained on AST-based code contexts (**Svyatkovskiy et al., 2019**).

Code representation

IMT SCUOLA ALTI STUDI LUCCA

ML Model Training:

1. **Recurrent Neural Networks (RNNs)**:
   - **LSTMs**:
     - Used for next-token prediction (e.g., **Terada and Watanobe, 2019**; **Rahman et al., 2020**).
     - Enhanced with attention mechanisms (**Wang et al., 2019**).
   - **GRUs**:
     - Capture contextual, syntactical, and structural dependencies (e.g., **Hussain et al., 2020**).
   - **Optimized RNN Approaches**:
     - Token repetition and memory optimization (**Yang et al., 2019a**).
2. **Probabilistic Models**:
   - Hidden Markov Models (**Han et al., 2009, 2011**).
   - Bayesian Networks (**Proksch et al., 2015**).
   - Decision Trees (**Raychev et al., 2016**).
   - Markov Chains for ranked API recommendations (**Svyatkovskiy et al., 2019**).
3. **Other Techniques**:
   - Multi-task learning (**Liu et al., 2020c,d**).
   - Code representation-based methods for logging decisions (**Lee et al., 2021**).
   - Tree-to-sequence (Tree2Seq) models for structural analysis and comment generation (**Chen and Wan, 2019**).

Timeline

Day	N. of hours
24/09/2024	3
26/09/2024	2
01/10/2024	3
03/10/2024	2
08/10/2024	3
10/10/2024	2
15/10/2024	3
17/10/2024	2
22/10/2024	3
24/10/2024	2
29/10/2024	0
31/10/2024	2
05/11/2024	3
07/11/2024	2
12/11/2024	3
14/11/2024	2
19/11/2024	3
21/11/2024	0
26/11/2024	3
28/11/2024	2
03/12/2024	3
Total hours	**48.00**

IMT SCUOLA ALTI STUDI LUCCA

Project

IMT SCUOLA ALTI STUDI LUCCA

The project you need to provide a fully working example of code auto-completion.
You can rely on existing methods trying to re-implement their solution, or provide your own solution. Not necessarily it should be the latest implementation available in the literature but it should be strongly supported by your choices.

Therefore, in the next two lessons we/you will focus on 4 information that are needed for the project:

- Datasets, which are the datasets used in existing papers, are they available? Can you get more data?
- Tokenizers, which kind of tokenizers? Some are available?
- Embeddings, which kind of embeddings? Is the code available? Do we need to implement them?
- Architectures, which are the architectures? Can we propose one? Which loss function?

Given the surveys explored today, since you have to work on the project, you can start learning and studying.

Code representation

IMT SCUOLA ALTI STUDI LUCCA

Code representation

IMT SCUOLA ALTI STUDI LUCCA

IMT

SCUOLA
ALTI STUDI
LUCCA

IMT School for Advanced Studies Lucca

IMT
SCUOLA
ALTI STUDI
LUCCA

Legal headquarters
Piazza San Ponziano, 6
55100 Lucca (Italy)

Campus
Piazza San Francesco, 19
55100 Lucca (Italy)

Via Brunero Paoli, 31
55100 Lucca (Italy)

Telephone
+39 0583 4326561

Email
info@imtlucca.it

imtlucca.it

MLSA Pipeline

Dataset preparation:

- identifying the source of required data, typically source code repositories.
- The activity involves selecting and downloading the required repositories,
- collecting supplementary data (such as GitHub issues)
- The outcome of this activity is a dataset.
- Depending upon the context, the dataset may contain information such as annotated code samples, source code model (e.g., **ast**), and pairs of buggy code and fixed code

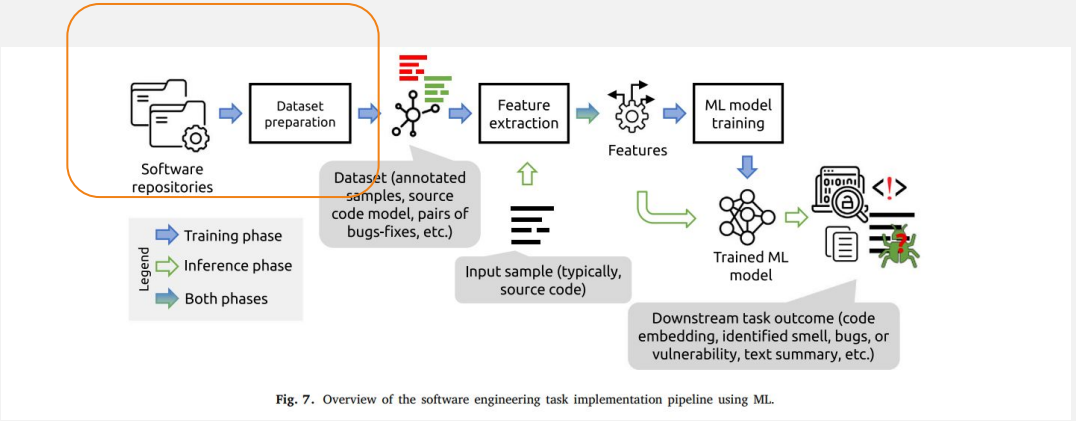


Fig. 7. Overview of the software engineering task implementation pipeline using ML.

MLSA Pipeline

Feature extraction
Performance of a ml model depends significantly on the provided kind and quality of features.

- source code metrics, source code tokens, their properties, and representation, changes in the code (code diff),
- vector representation of code and text,
- dependency graph, and vector representation of **ast**, **cfg**, or **ast diff**.
- Obviously, selection of the specific features depends on the downstream task.

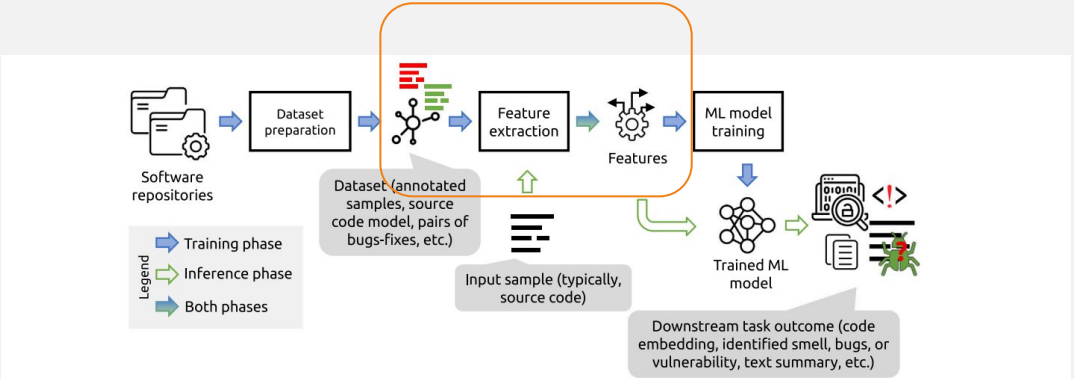


Fig. 7. Overview of the software engineering task implementation pipeline using ML.

MLSA Pipeline

IMT SCUOLA ALTI STUDI LUCCA

Model choice/training
Selecting a ml model for a given task depends on many factors:

- The type features, if they are fixed like the code metrics we can use standard Machine learning models
- With embeddings we use Deep Learning models like recurrent neural networks, sequence to sequence approaches, etc.
- Other methods can rely on the graph nature of the code applying Graph Neural Networks

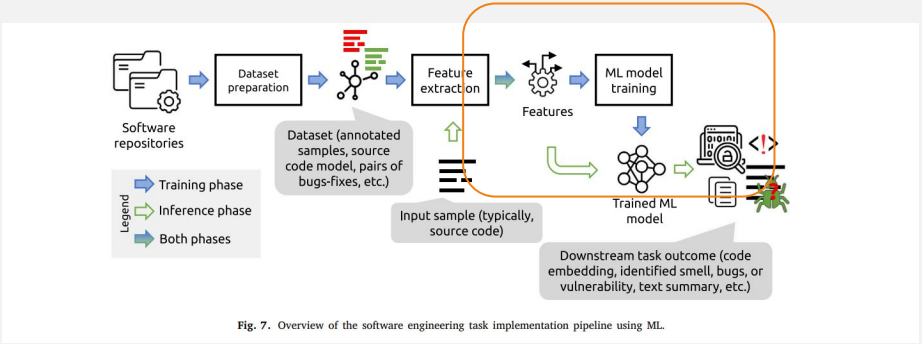


Fig. 7. Overview of the software engineering task implementation pipeline using ML.