Bug Predictor

A Machine Learning-Based Software Risk Analysis Tool

Project Report

Software Engineering Mini-Project

December 7, 2025

Contents

1 Introduction

1.1 Project Overview

The **Bug Predictor** is an automated software analysis tool designed to identify at-risk files in software projects. By analyzing version control history and code complexity metrics, the system provides developers with actionable insights about which files are most likely to contain bugs or require frequent maintenance.

1.2 Motivation

Software maintenance accounts for a significant portion of development costs. Early identification of problematic code can help teams:

- Prioritize code review efforts
- Allocate testing resources effectively
- Reduce technical debt proactively
- Improve overall code quality

1.3 Objectives

The primary objectives of this project are:

1. Develop an automated risk assessment system for source code files
2. Create an intuitive web-based dashboard for visualizing analysis results
3. Implement scalable analysis algorithms for large codebases
4. Provide actionable metrics to guide development decisions

2 System Architecture

2.1 Technology Stack

The Bug Predictor is built using modern Python technologies:

- **Backend Framework**: FastAPI - High-performance asynchronous web framework
- **Version Control Analysis**: GitPython - Python library for Git repository interaction
- **Complexity Analysis**: Radon - Code metrics and static analysis tool
- **Data Models**: Pydantic - Data validation using Python type annotations
- **Template Engine**: Jinja2 - Modern templating for Python
- **Frontend**: HTML/TailwindCSS - Responsive web interface
- **Testing**: pytest - Python testing framework

2.2 Architectural Design

The system follows a three-tier architecture:

Presentation Layer Web-based dashboard providing user interface and visualization

Business Logic Layer Analysis services including Git history analysis, complexity calculation, and risk scoring

Data Layer File system access and Git repository interaction

2.3 Component Structure

The application is organized into the following modules:

- `app/main.py` - FastAPI application entry point and route handlers
- `app/services/` - Core analysis services
  - `git_analysis.py` - Git history and churn metrics
  - `complexity.py` - Cyclomatic complexity calculation
  - `risk.py` - Risk scoring algorithm
- `app/models/` - Data models and schemas
  - `analysis_model.py` - File and project analysis models
- `app/templates/` - HTML templates for web interface
- `tests/` - Unit and integration tests

3 Core Features

3.1 Git History Analysis

The system analyzes version control history to identify files that have been frequently modified in bug-fix commits. This is based on the premise that files with high churn rates are more likely to contain bugs.

Key Metrics:

- Commit frequency per file
- Number of bug-fix related commits
- File modification patterns over time

3.2 Complexity Analysis

Using the Radon library, the system calculates cyclomatic complexity for each source file. Cyclomatic complexity measures the number of linearly independent paths through a program's source code.

Complexity Indicators:

- Function-level complexity scores
- File-level aggregated complexity
- Maintainability index

3.3 Risk Scoring Algorithm

The risk score combines both historical data (churn metrics) and static analysis (complexity) to generate a composite risk indicator for each file.

Risk Formula:

$$RiskScore = f(ChurnCount, ComplexityScore) \tag{1}$$

Files with higher risk scores should be prioritized for:

- Code review
- Refactoring efforts
- Additional testing coverage
- Documentation improvements

3.4 Web Dashboard

The interactive dashboard provides:

- Project path input form
- Tabular display of analysis results
- Sortable columns (filename, churn, complexity, risk)
- Visual indicators for high-risk files
- Summary statistics for the entire project

4 Implementation Details

4.1 Analysis Workflow

The analysis process follows these steps:

1. User submits project path via web form
2. System validates directory existence
3. Git repository is analyzed for historical churn data
4. Directory tree is traversed, excluding common ignore patterns
5. For each source file:
    - Retrieve churn count from Git analysis
    - Calculate cyclomatic complexity
    - Compute risk score
6. Results are aggregated and sorted by risk score
7. Dashboard displays ranked list of files

4.2 Path Normalization

The system handles platform-specific path separators to ensure compatibility across Windows and Unix-based systems. Git typically uses forward slashes (`/`), while Windows uses backslashes (\\), requiring careful normalization.

4.3 Performance Considerations

To maintain performance on large codebases:

- Common directories like `.git`, `__pycache__`, `venv`, `node_modules` are excluded
- Only files with non-zero metrics are included in results
- Asynchronous processing via FastAPI

5 Installation and Usage

5.1 System Requirements

- Python 3.8 or higher
- Git installed and accessible via command line
- Modern web browser

5.2 Installation Steps

```
# Clone the repository
git clone <repository-url>
cd BugPredictor

# Install dependencies
pip install -r requirements.txt
```

5.3 Running the Application

```
# Start the server
uvicorn app.main:app --reload

# Access the dashboard
# Open browser to http://localhost:8000
```

5.4 Using the Dashboard

1. Navigate to `http://localhost:8000`
2. Enter the absolute path to a Git repository
3. Click "Analyze" to start the analysis
4. Review the risk-sorted file list
5. Focus on files with highest risk scores for review

6 Testing Strategy

6.1 Unit Testing

Individual components are tested in isolation:

- Service layer functionality
- Risk calculation algorithms
- Data model validation

6.2 Integration Testing

End-to-end testing validates:

- API endpoint responses
- Database interactions
- Complete analysis workflow

6.3 Test Framework

The project uses `pytest` for testing, providing:

- Fixtures for test data setup
- Parametrized tests for multiple scenarios
- Coverage reporting

7 Documentation

The project includes comprehensive documentation:

- `README.md` - Quick start guide and basic usage
- `docs/design/architecture.md` - System architecture diagrams
- `docs/design/class_diagram.md` - UML class diagrams
- `docs/scrum/` - Agile development artifacts (product/sprint backlogs)
- `docs/testing/test_plan.md` - Comprehensive testing strategy

8 Future Enhancements

8.1 Planned Features

- Machine learning-based prediction models
- Historical trend visualization
- Integration with CI/CD pipelines
- Multi-repository comparative analysis
- Customizable risk weighting factors
- Export reports in multiple formats (PDF, CSV, JSON)

8.2 Scalability Improvements

- Caching mechanism for large repositories
- Incremental analysis for updated files only
- Parallel processing for multi-core systems
- Database backend for historical data persistence

9 Conclusion

The Bug Predictor project successfully demonstrates the application of software metrics and version control analysis to identify high-risk code. By combining historical churn data with static complexity analysis, the tool provides valuable insights that can guide code review priorities and refactoring efforts.

The modular architecture and use of modern Python frameworks ensure that the system is maintainable and extensible. The web-based interface makes the tool accessible to all team members, promoting data-driven development practices.

9.1 Key Achievements

- Functional prototype with core analysis capabilities
- Clean, modular codebase following best practices
- Comprehensive documentation and testing
- User-friendly web interface
- Foundation for future machine learning enhancements

9.2 Learning Outcomes

This project provided valuable experience in:

- Modern Python web development with FastAPI
- Software metrics and static code analysis
- Git repository manipulation and analysis

- Full-stack application development
- Software engineering best practices

A Dependencies

The complete list of Python dependencies from `requirements.txt`:

```
fastapi
uvicorn
gitpython
radon
pydantic
jinja2
pytest
python-multipart
```

B API Endpoints

Endpoint	Method	Description
`/`	GET	Render main dashboard page
`/analyze`	POST	Analyze project and return results

Table 1: Available API endpoints

C Project Structure

```
BugPredictor/
          app/
                __init__.py
                main.py
                models/
                      analysis_model.py
                services/
                      git_analysis.py
                      complexity.py
                      risk.py
                templates/
                    index.html
          docs/
                design/
                scrum/
                testing/
          tests/
          requirements.txt
          README.md
```