Machine Learning Ensemble Framework for Automated Test Failure Prediction: A Student Research Project

Harrison Sean King
University of Kent Canterbury, School of Computing
Canterbury, Kent, United Kingdom
harrisonking3465@gmail.com
GitHub: github.com/zolydiac

Abstract

This paper presents a machine learning framework I developed to predict test failures in automated software testing environments. The project combines different ML approaches - Random Forest, Gradient Boosting, Neural Networks, and LSTM - to see which works best for predicting when tests might fail. Working entirely through GitHub development over several months, I created a system that processes test execution data and predicts failures with reasonable accuracy. The main challenges were getting realistic synthetic data, balancing model complexity with performance, and dealing with the typical issues that come up when you're learning ML frameworks like PyTorch while building something practical. The framework achieves decent prediction performance and provides insights into which features matter most for test stability. This work represents my exploration into applying ML to real software engineering problems as part of my research interests for graduate study.

Keywords: Software Testing, Machine Learning, Ensemble Methods, Student Research Project

1. Introduction

As a computer science student interested in both machine learning and software engineering, I wanted to tackle a practical problem that combines both areas. Test failures in continuous integration pipelines are something every developer deals with - you push code, tests fail, and you spend time figuring out if it's a real bug or just a flaky test.

The idea for this project came from my own frustration during group projects at university where we'd have automated tests that would randomly fail for no clear reason. I thought it would be interesting to see if machine learning could help predict which tests are likely to fail before they actually do.

1.1 Why This Problem Matters

From what I've read and experienced, developers spend a lot of time dealing with unreliable tests. It's especially frustrating because you can't tell immediately if a test failure means your code is broken or if the test itself has issues. I figured if I could build something that gives a heads-up about which tests are likely to fail, it could save developers time.

The challenge is that test failures aren't random - they follow patterns related to code changes, test complexity, and timing. Machine learning seemed like a good fit for finding these patterns automatically instead of trying to write rules manually.

1.2 What I Built

My project creates an ensemble of different ML models that work together to predict test failures. The main contributions are:

1. **Combination of different algorithms**: I tried Random Forest, Gradient Boosting, a neural network, and LSTM to see what works best
2. **Feature engineering for testing**: Created features that capture things like code change frequency, test complexity, and historical patterns
3. **Complete implementation**: Built everything from data generation to model evaluation, all available on GitHub
4. **Practical focus**: Designed to work with real development workflows through GitHub API integration

2. Background and Related Work

I spent time reading about existing approaches to this problem. Most of the research I found focuses on test case prioritization or detecting flaky tests after they've already failed.

Rule-based approaches use simple heuristics like "if a test failed yesterday, it's more likely to fail today." These work okay but miss complex patterns.

Machine learning approaches have been tried before, but usually with single algorithms. Luo et al. [1] did important work on analyzing flaky tests, while Rahman et al. [2] used Random Forest for GUI test prediction. However, I didn't find much work combining multiple ML approaches systematically.

Feature engineering for software testing is still being figured out. Palomba and Bacchelli [3] looked at test code quality, and Shi et al. [4] worked on test suite optimization, but there's room for more comprehensive feature sets.

My approach differs by combining multiple algorithms and focusing on prediction rather than just analysis of existing failures.

3. My Approach and Implementation

3.1 Development Process

I developed this entirely through GitHub, which you can see in the commit history at github.com/zolydiac/Advanced-ML-Enhanced-Test-Failure-Prediction-Framework. The development took place over several months as I learned PyTorch and figured out the best way to structure the code.

Early Development Challenges: Initially, I tried to collect real test data from open source projects, but this turned out to be much harder than expected. Getting access to CI/CD data, handling different test formats, and dealing with privacy issues made this approach impractical for a student project.

Pivot to Synthetic Data: After struggling with real data for a few weeks, I decided to create realistic synthetic data instead. This let me control the patterns and create a dataset large enough for meaningful ML experiments.

3.2 Architecture

The framework has four main parts:

1. **Data Generation**: Creates synthetic test execution data with realistic patterns
2. **Feature Engineering**: Transforms raw data into features ML models can use
3. **Model Training**: Four different algorithms trained individually and combined
4. **Evaluation**: Cross-validation and statistical testing to measure performance

3.3 Feature Engineering

I created features based on what I thought would matter for test failures:

Time-based features: Day of week, hour of day, time since last run **Code change features**: Number of commits, files changed, authors involved **Test complexity features**: Lines of code, number of assertions, complexity metrics **Historical features**: Past failure rates, how long since the test last failed

The biggest challenge here was making sure the synthetic features reflected realistic patterns. I spent a lot of time adjusting the data generation to match what I'd expect from real software projects.

3.4 Machine Learning Models

I implemented four different approaches:

Random Forest: Good baseline that handles mixed data types well and provides feature importance **Gradient Boosting**: Sequential learning that builds on mistakes from previous models **Neural Network**: Multi-layer network to capture non-linear patterns **LSTM**: For learning temporal patterns in test execution sequences

Implementation Reality: The LSTM was particularly tricky to get working. PyTorch's documentation is good but it took me several attempts to get the data reshaping right for sequential input. The neural network was more straightforward but required a lot of experimentation with layer sizes and dropout rates.

3.5 Ensemble Strategy

Rather than picking one "best" model, I combine predictions from all four. Each model is trained independently, then their predictions are averaged with weights based on their individual performance.

This approach worked better than any single model, though the improvement wasn't as dramatic as I initially hoped.

4. Experimental Results

4.1 Dataset and Evaluation Setup

I generated 2,500 test execution samples with realistic failure patterns. The overall failure rate is about 13%, which matches what I've seen in literature about real test suites.

For evaluation, I used 5-fold cross-validation to get reliable performance estimates. I focused on AUC (Area Under the ROC Curve) as the main metric since it's good for ranking which tests are most likely to fail.

4.2 Performance Results

Here's what I found:

Model	AUC Score	Standard Deviation
Random Forest	0.728	±0.034
Gradient Boosting	0.756	±0.029
Neural Network	0.742	±0.031
LSTM	0.719	±0.038
Ensemble	**0.779**	**±0.025**

The ensemble approach works best, but honestly the improvement over Gradient Boosting alone is smaller than I expected. Still, it's consistent across different random seeds.

Statistical Testing: I used Wilcoxon signed-rank tests to check if the differences between models are statistically significant. The ensemble significantly outperforms individual models ($p < 0.05$), which is encouraging.

4.3 What Features Matter Most

According to the Random Forest feature importance:

1. **Previous failure rate** (30 days) - Past behavior strongly predicts future behavior
2. **Code change frequency** - Tests fail more when code changes frequently
3. **Test complexity** - More complex tests are more likely to fail
4. **Time patterns** - Some days of the week have higher failure rates

5. **Maintenance indicators** - Tests that haven't been updated in a while are riskier

This matches my intuition about what makes tests unreliable.

4.3 Implementation Challenges I Faced

Data Pipeline Issues: Getting the feature engineering pipeline to work consistently was harder than expected. I had several bugs where features were calculated differently during training vs. prediction, which took a while to debug.

Model Training Stability: The neural networks were sensitive to initialization and learning rates. I ended up having to experiment with different architectures and hyperparameters more than I initially planned.

Performance vs. Complexity Trade-offs: The LSTM model was the most complex but didn't perform as well as simpler approaches. This taught me that more sophisticated doesn't always mean better for this problem.

Evaluation Methodology: Making sure my cross-validation was done correctly took several iterations. I initially had data leakage between folds which inflated my performance numbers.

5. Discussion and Lessons Learned

5.1 What Worked Well

The ensemble approach does provide consistent improvements over single models. The feature engineering captured patterns that make intuitive sense - tests fail more when code changes frequently, complex tests are less reliable, and past failures predict future ones.

The framework is reasonably fast and could potentially be used in a real CI/CD pipeline, though I haven't tested it at scale.

5.2 What Didn't Work As Expected

LSTM Performance: I expected the LSTM to perform much better since test failures should have temporal dependencies. However, it was the weakest individual model. This might be because my synthetic data doesn't capture the right kind of temporal patterns, or because the sequences aren't long enough for LSTM to shine.

Feature Engineering Impact: While my features are reasonable, I suspect they're not capturing all the important patterns. Real test failures probably depend on more subtle code change characteristics that are hard to simulate.

5.3 Limitations and Future Work

Synthetic Data: The biggest limitation is using synthetic rather than real data. While I tried to make it realistic, I'm sure it misses important patterns from actual software projects.

Limited Test Types: I focused mainly on unit and integration tests. End-to-end tests, performance tests, and other types might have different patterns.

Scalability: I haven't tested how well this works with very large test suites or high-frequency CI/CD environments.

For future work, I'd love to:

- Get access to real test execution data from industry or open source projects
- Expand to different types of testing (mobile, API, performance)
- Add more sophisticated temporal modeling
- Build a real-time integration with a CI/CD system

6. Conclusion

This project taught me a lot about both machine learning and software testing. The ensemble approach works reasonably well for predicting test failures, achieving AUC scores around 0.78 which is decent for this type of problem.

The most valuable part was going through the complete ML pipeline - from problem formulation to data generation, feature engineering, model selection, and evaluation. I learned that getting the data right is often harder and more important than choosing the perfect algorithm.

While the results are promising, I know there's still a lot of work needed to make this truly useful in practice. The experience has definitely reinforced my interest in applying ML to software engineering problems, which is why I'm excited about pursuing graduate research in this area.

The complete implementation is available at github.com/zolydiac/Advanced-ML-Enhanced-Test-Failure-Prediction-Framework for anyone interested in building on this work.

References

[1] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in Proceedings of the 2014 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014, pp. 643-653.

[2] M. M. Rahman, L. Karampinos, A. Schaad, and L. Williams, "Predicting GUI test failure using machine learning," in 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2020, pp. 678-688.

[3] F. Palomba and A. Bacchelli, "Does test code quality affect the reliability of test outcomes?," in Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018, pp. 722-732.

[4] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov, "Balancing trade-offs in test-suite reduction," in Proceedings of the 2014 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014, pp. 246-256.