Predicting Test Smells based on Deep Learning

Internship Report

HAGHIGHI Camilia

Supervisors: Mr. Nguyen Thanh Binh, Mr. Huynh Ngoc Khoa

April - July 2025

1 Introduction

During this internship, the objective was to explore the use of Deep Learning techniques to automatically detect the presence of test smells in unit test code. Test smells are indicators of poor design or maintenance issues in test suites, and detecting them accurately is crucial for improving software quality.

To address this challenge, the work was structured in several key stages:

- A thorough literature review to understand the foundations of test smells, the tools developed to detect them, and the current use of machine learning in this area.
- The construction of a high-quality dataset, including manual validation of test smell annotations to ensure accuracy.
- Preprocessing and selection of relevant features to prepare the dataset for model training.
- The development and evaluation of deep learning models, particularly Artificial Neural Networks (ANN), compared to traditional machine learning baselines such as Random Forest.
- Experimental evaluations, including multi-label and mono-label settings, followed by retraining strategies to improve model performance.

This report presents the methodology followed, the experiments conducted, and the insights gained, highlighting the potential of deep learning for scalable and flexible test smell detection.

2 Proper bases

To gain a solid understanding of test smells, the first part of the internship was dedicated to reviewing the main research papers published over the last two decades on this topic.

This literature review covered:

- The origin and definition of test smells
- The tools developed to detect them
- The growing use of machine learning and large language models (LLMs) for detection

1. Understanding Test Smells

Test smells are indicators of underlying issues or poor design choices in test code. They typically represent violations of best practices and testing principles. The concept was initially introduced by Van Deursen et al. (2001) [1].

Test smells negatively impact the maintainability, readability, and reliability of test suites. They can make tests harder to understand and modify, increase execution time, and even reduce the overall quality of the production code.

Over the years, the original catalog of 11 smells has expanded to more than 60.
These smells persist in software systems for various reasons:

- They often go unnoticed by developers, including experienced ones
- They make test maintenance and debugging more difficult
- They increase the likelihood of fragile tests
- They tend to remain in the codebase once introduced

2. Tools for Test Smell Detection

To assist developers, several tools have been created to automatically detect test smells. A total of 24 tools have been identified, including:

- **JNose (2020)** : Uses static analysis using an Abstract Syntax Tree (AST) to identify 19 different test smells
- **tsDetect (2019)** : Detects 19 test smells in Java (JUnit) and is considered state-of-the-art
- **TASTE (2018)** : Uses textual analysis and Information Retrieval techniques (Java)
- **TestQ (2008)** : Offers a visual interface for detecting 12 smells in C++
- Others include: TRex, TestLint, PyNOSE, TeReDetect, DARTS, RAIDE, soCRATES...

Most of these tools are heuristic-based, relying on fixed rules and patterns. However, they often target only a small subset of smells (e.g., General Fixture, Eager Test, Assertion Roulette, Mystery Guest) and primarily focus on Java code. No single tool currently detects all smells accurately.

3. Machine Learning and LLM-Based Detection

Several studies have explored the use of machine learning to detect test smells, particularly Eager Test, Mystery Guest, Resource Optimism, Test Redundancy. Algorithms tested include Random Forest, Support Vector Machine (SVM), Naive Bayes, Logistic Regression...

Among them, Random Forest achieved the best results, outperforming traditional heuristics. However, overall performance remained limited (F1 $\leq 51\%$), suggesting a high rate of false positives or false negatives.

More recently, researchers have evaluated the performance of large language models (LLMs) such as ChatGPT, Gemini Advanced, and Mistral Large:

- 30 test smells across 7 languages (Java, Python, Ruby, C#, Smalltalk, JavaScript, TTCN-3)
- Zero-shot prompts used (no specific training)

- **ChatGPT-4** : Detected up to 26/30 smells after retries
- **Gemini Advanced** : up to 24/30 after retries
- **Mistral Large** : up to 21/30 after retries

Despite promising results, all models failed to detect certain smells.

4. Conclusion on Current Approaches

Most Machine Learning-based methods have focused on a few common smells, often with modest performance. Detection of more subtle or complex smells remains a challenge.
Deep learning is still underexplored in this field, although LLM-based studies have shown that it could be a promising direction.

This context provides the motivation for our work: investigating the use of a Deep Learning model (Artificial Neural Network) for detecting multiple test smells with improved accuracy and generalizability..

3 Internship Contributions and Implementation

To train a deep learning model capable of predicting test smells, a reliable and well-labeled dataset was essential. For this purpose, I used the publicly available dataset introduced in the paper *"On the diffusion of test smells and their relationship with test code quality of Java projects" (Luana Martins et al., 2022) [21]* . This dataset includes over 400,000 test methods collected from more than 13,000 open-source GitHub projects, annotated with the presence or absence of 20 different test smells *(refer to Appendix, Section 5)* using the tool JNose.

3.1 Dataset Construction and Manual Validation

Sampling and Manual Labeling

From this large dataset, I randomly sampled 5% of the data (approximately 20,000 methods). To ensure label quality, I performed a thorough manual verification on about 10,000 of these methods. For each selected test smell, I created a dedicated column to manually annotate its presence.
The manual labeling process followed these steps:

- Carefully reading each test method line by line.
- Identifying specific patterns and design violations based on formal definitions of test smells.
- Comparing my annotations with JNose's automatic results to flag mismatches

Test smells were identified using syntactic patterns and contextual information from the code.

However, some smells (such as General Fixture or Eager Test) required more than basic pattern matching. Their detection often depended on understanding the intent and structure of the test method. For instance, identifying an Eager Test involves recognizing when a single test checks multiple behaviors, while a General Fixture may involve unnecessary setup not used by all tests. This made the detection process more complex, as it required interpreting the logic and design of the test code.

Challenges Faced

This manual process required focus, consistency, and attention to detail. Some of the main challenges included:

- Smells were sometimes subtle or dependent on the logic of the full class context.
- Ambiguity in definitions required interpretative judgment to maintain labeling consistency.

Outcome

The final manually verified dataset contains around 10,000 test methods. A script was computed to calculate the discrepencie rate between my manual labels and the JNose output. The observed mismatch rate was approximately 2.81 %, which suggests that JNose provides generally reliable detection, but manual verification is still valuable for improving dataset precision.

This high-quality dataset became the foundation for training and evaluating my deep learning model, ensuring that the labels were accurate and trustworthy — a critical requirement for effective supervised learning

3.2 Preparing the dataset

The first step of the project involved constructing a custom dataset tailored for test smell detection in unit test files. After preprocessing and filtering, the dataset reached a final size of 10,000 rows, with each row representing a single test case. It included several types of features:

- **Informational columns:** class names, method names, file paths...
- **Feature columns:** program metrics like LOC (Lines of Code), RWC, and others. *(refer to Appendix, Section 5)*
- **Test smells from JNose:** labels generated using a tool-based detection.
- **Manually validated test smells:** corrected or validated by human analysis.

Before any model training, I implemented a data cleaning step, which included removing irrelevant columns, selecting key features, and applying normalization.

To ensure effective training and minimize bias, I computed the percentage distribution of each test smell in the dataset and selected the two most balanced pairs:

- **Assertion Roulette:** 33.11%
- **Eager Test:** 26.55%
- **Magic Number Test:** 24.74%
- **Unknown Test:** 20.42%

Choosing test smells that are equally represented helps to avoid class imbalance, which can strongly bias the model during training. When one class dominates the dataset, the model tends to learn to always predict that class, resulting in high accuracy but poor generalization and low performance on the minority class.
I also implemented class weights to further address any residual imbalance.

3.3 Training deep learning model

The second phase involved building and evaluating an Artificial Neural Network (ANN) for test smell detection. I explored multiple architectural configurations to determine the best-performing model. The following components were systematically varied:

- Number of layers: 3 vs 4
- Neurons per layer: 256, 128, 64, 32..
- Dropout: with and without
- Scheduler: learning rate adaptation
- Batch size: best results with 16
- Epochs: 50 chosen as optimal for convergence

Model performance was evaluated using standard metrics (precision, recall, F1-score) across two test smell pairs: AssertionRoulette + EagerTest. These were chosen because they are the most frequent test smells in the dataset and offer a balanced training distribution. This made them ideal candidates to evaluate the impact of architecture, dropout, and learning rate scheduling.
Once the best configuration was identified, it was reused for all four individual models to ensure consistency and comparability.

Table 1: Summary of ANN configuration tests (F1-score macro averaged across test smells)

Architecture	**Dropout**	**Scheduler**	**F1-score**	**Excecution Time**
$128 \rightarrow 64 \rightarrow 32$	NO	NO	0.8101	283.313
$128 \rightarrow 64 \rightarrow 32$	NO	YES	0.8136	163.827
$128 \rightarrow 64 \rightarrow 32$	YES	NO	0.7979	136.717
$128 \rightarrow 64 \rightarrow 32$	YES	YES	0.7979	186.587
$256 \rightarrow 128 \rightarrow 64$	NO	NO	0.8212	132.625
$256 \rightarrow 128 \rightarrow 64$	NO	YES	0.8212	141.114
$256 \rightarrow 128 \rightarrow 64$	YES	NO	0.8126	223.291
$256 \rightarrow 128 \rightarrow 64$	YES	YES	0.8123	251.924
$256 \rightarrow 128 \rightarrow 64 \rightarrow 32$	NO	NO	0.8225	137.763
$256 \rightarrow 128 \rightarrow 64 \rightarrow 32$	NO	YES	0.8225	141.617
$256 \rightarrow 128 \rightarrow 64 \rightarrow 32$	YES	NO	0.8116	146.95
$256 \rightarrow 128 \rightarrow 64 \rightarrow 32$	YES	YES	0.8116	138.496

- Dropout and scheduler showed no significant improvement and were not retained.
- A deeper architecture (4 layers) slightly improved average F1-scores but at the cost of complexity.
- A classic 3-layer offered overall good performance.

To verify the consistency of performance, I ran both the 3-layer and 4-layer models on 5 different random sample.

Table 2: Comparison between 3-layer and 4-layer ANN models across different random seeds

Sample	**F1 Macro (3 layers)**	**F1 Macro (4 layers)**
0	0.8122	0.8078
1	0.8176	0.8327
2	0.8338	0.8144
3	0.8052	0.8203
4	0.8359	0.8322
Mean	0.8209	0.8215
Std. Deviation	0.0120	0.0098

After running 5 randomized experiments for both architectures (ANN and DeepANN), the results showed a very small difference in average F1 macro score (0.8215 vs 0.8209).

While the DeepANN model is slightly more stable (lower standard deviation), the classic ANN with 3 layers remains a better choice in our context due to its simpler structure, faster training, and similar performance.

Given that the experiments were conducted on a machine with limited RAM and processing power, opting for a lighter model was more practical.
Additionally, deeper networks require learning more parameters, which increases computational cost and the risk of overfitting, especially when working with relatively small datasets, as in this

study. Overfitting describes when a model performs very well on the training data at hand, but its performance degrades on unseen data.

Therefore, the 3-layer ANN architecture was chosen for further analysis.

The implementation of the multi-label Artificial Neural Network (ANN) is provided in Appendix Section 5

NOTE

> In the next part Random Forest is used as a traditional baseline model. To ensure consistency, it is evaluated under the same multi-label configuration as the ANN.
>
> To ensure reproducibility of all experiments, a fixed random seed (`42`) was used throughout the implementation. This applies to Python's random module, NumPy, and PyTorch, making results such as training splits and model evaluations consistent across runs.

3.3.1 Multi-label Algorithm

Eager Test, Assertion Roulette

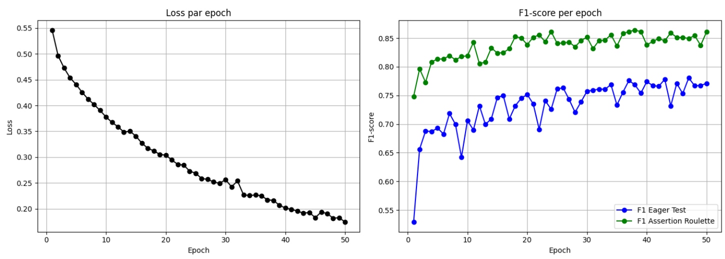


Figure 1: Training Loss and F1-Score Progression by Epoch (Eager Test - Assertion Roulette)

In the graph above, we can see that the loss curve shows a smooth and continuous decrease throughout training, indicating good convergence of the model. This suggests that the model is learning effectively from the data over the 50 epochs.

The F1-score improves rapidly during the first few epochs and stabilizes afterward. Assertion Roulette reaches higher F1-scores (0.85) than Eager Test (0.77), suggesting it is easier to detect in this dataset.

The ANN model trains correctly but performs better on Assertion Roulette than on Eager Test, most likely due to a higher pattern complexity.

Table 3: Classification Report for Eager Test

Class	Precision	Recall	F1-score	Support
No	0.75	0.69	0.72	489
Yes	0.75	0.80	0.77	558
Accuracy			0.75	1047

F1-score macro : 0.7452
F1-score micro : 0.7479

Table 4: Classification Report for Assertion Roulette

Class	Precision	Recall	F1-score	Support
No	0.76	0.83	0.80	406
Yes	0.89	0.84	0.86	641
Accuracy			0.83	1047

F1-score macro : 0.8286
F1-score micro : 0.8348

For comparison purposes, we evaluated the performance of a Random Forest model on the same dataset.

Table 5: Classification Report for Eager Test (Random Forest)

Class	Precision	Recall	F1-score	Support
No	0.79	0.77	0.78	489
Yes	0.80	0.82	0.81	558
Accuracy			0.80	1047

F1-score macro : 0.7963
F1-score micro : 0.7975

Table 6: Classification Report for Assertion Roulette (Random Forest)

Class	Precision	Recall	F1-score	Support
No	0.86	0.72	0.78	406
Yes	0.84	0.93	0.88	641
Accuracy			0.85	1047

F1-score macro : 0.8321
F1-score micro : 0.8462

The results appear consistent, as both the ANN and Random Forest models achieve similar performance levels within the same range of F1-scores.

Unknown Test, Magic Number Test

Since the Eager Test test smell is most likely the most difficult to detect. We will try the model on two other test smells that have more logical pattern that should be easier for the model to detect.

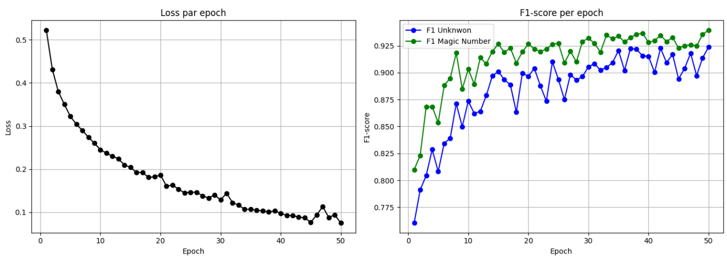


Figure 2: Training Loss and F1-Score Progression by Epoch (Unknown Test - Magic Number Test

In the graph above, we can see that the training loss steadily decreases and stabilizes at a low value, showing successful training of the model for these two smells.

Both labels show high and stable F1-scores across epochs. Magic Number performs slightly better (0.93) than Unknown Test (0.91), which confirms the model's strong ability to detect both smells.

Table 7: Classification Report for Unknown Test

Class	Precision	Recall	F1-score	Support
No	0.92	0.97	0.94	513
Yes	0.95	0.90	0.92	393
Accuracy			0.94	906

F1-score macro : 0.9343
F1-score micro : 0.9360

Table 8: Classification Report for Magic Number

Class	Precision	Recall	F1-score	Support
No	0.95	0.88	0.92	399
Yes	0.91	0.97	0.94	507
Accuracy			0.93	906

F1-score macro : 0.9288
F1-score micro : 0.9305

We also compare these results to those obtained using the Random Forest model: :

Table 9: Classification Report for Unknown Test (Random Forest)

Class	Precision	Recall	F1-score	Support
No	0.91	0.96	0.94	513
Yes	0.95	0.88	0.91	393
Accuracy			0.93	906

F1-score macro : 0.9252
F1-score micro : 0.9272

Table 10: Classification Report for Magic Number (Random Forest)

Class	Precision	Recall	F1-score	Support
No	0.97	0.89	0.93	399
Yes	0.92	0.98	0.95	507
Accuracy			0.94	906

F1-score macro : 0.9377
F1-score micro : 0.9393

The results appear consistent, as both the ANN and Random Forest models achieve similar performance levels within the same range of F1-scores.

Confusion Matrix

To keep the main body concise, the confusion matrices for the multi-label neural network model and the Random Forest baseline are included in Appendix, Section 5

To complement the numerical evaluation, we analyze the confusion matrices for both the multi-label neural network and the Random Forest models on the same test set. These matrices help to visualize each model's ability to distinguish between positive and negative cases for each test smell.

Table 11: Comparison of Confusion Matrix Results (Multi-label vs Random Forest)

Test Smell	**Model**	**TP**	**FP**	**FN**	**TN**
EagerTest	Multi-label	445	151	113	388
	Random Forest	458	112	100	377
AssertionRoulette	Multi-label	536	68	105	338
	Random Forest	595	115	46	291
UnknownTest	Multi-label	352	17	41	496
	Random Forest	347	20	46	493
MagicNumber	Multi-label	490	46	17	353
	Random Forest	497	45	10	354

Legend: TP: True Positive, FP: False Positive, FN: False Negative, TN: True Negative

Overall, both the multi-label ANN and the Random Forest classifier perform similarly across the four test smells. While minor differences can be observed, variations remain limited.
The confusion matrices reinforce the numerical evaluation shown earlier, indicating that both models are capable of distinguishing between positive and negative samples with good accuracy. The results confirm that multi-label learning is a reliable and scalable alternative to traditional binary classifiers, especially when addressing several test smells jointly.

3.3.2 Mono-label Algorithm

In this part of the study, I experimented with mono-label classification to investigate whether training separate models for each test smell could lead to improved performance. Instead of using a multi-label approach, I trained one model per label (e.g., one for EagerTest, one for AssertionRoulette), with the aim of simplifying the learning task and potentially enhancing prediction accuracy. Each test smell is trained on the same data sample from the multi-model algorithm to keep consistency between the results.

To further optimize the models, I selected the top 15 most relevant features for each label using feature importance scores computed from a Random Forest classifier *(refer to Appendix, Section 5).* By ranking the features based on their contribution to prediction performance, I was able to reduce dimensionality and focus the training process on the most informative attributes. This approach helps reduce overfitting and speeds up training while potentially improving generalization.

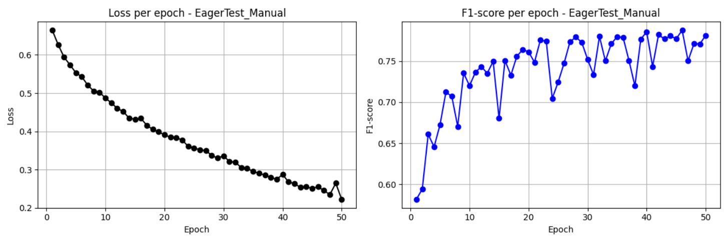


Figure 3: Training Loss and F1-Score Progression by Epoch (Eager Test)

Table 12: Classification Report for Eager Test

	Precision	**Recall**	**F1-score**	**Support**
Non	0.75	0.74	0.75	489
Oui	0.78	0.78	0.78	558
Accuracy			0.77	1047

F1-score macro : 0.7648
F1-score micro : 0.7660

Interpretation:
Loss per epoch: The curve shows a smooth decline, reflecting proper training convergence.
F1-score per epoch: The score reaches approximately 0.78, which is similar to the multi-label result. However, the multi-label model achieves this performance without isolating the training, making it a more scalable and efficient solution when dealing with several smells simultaneously.

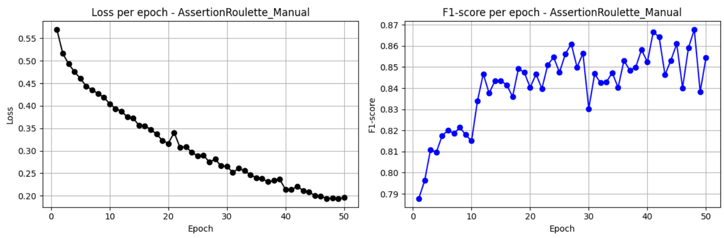


Figure 4: Training Loss and F1-Score Progression by Epoch (Assertion Roulette)

Table 13: Classification Report for Assertion Roulette

	Precision	**Recall**	**F1-score**	**Support**
Non	0.78	0.74	0.76	406
Oui	0.84	0.87	0.85	641
Accuracy			0.82	1047

F1-score macro : 0.8067
F1-score micro : 0.8185

Interpretation:
Loss per epoch: The loss steadily decreases, indicating effective training for this single label.
F1-score per epoch: The model reaches an F1-score of around 0.86, which is quite good. However, the multi-label model also achieves this performance while learning multiple classes at once, showing its capacity to generalize efficiently across labels.

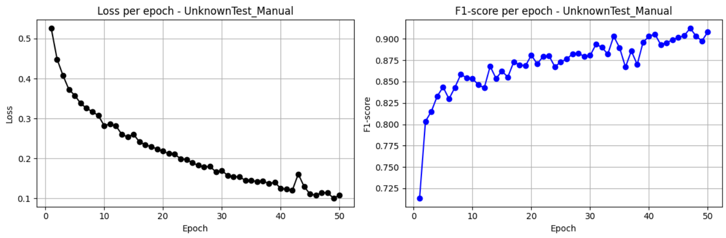


Figure 5: Training Loss and F1-Score Progression by Epoch (Unknown Test)

Table 14: Classification Report for Unknown Test

	Precision	**Recall**	**F1-score**	**Support**
Non	0.93	0.93	0.93	513
Oui	0.91	0.91	0.91	393
Accuracy			0.92	906

F1-score macro : 0.9191
F1-score micro : 0.9205

Interpretation:
Loss per epoch: The loss shows consistent improvement across epochs.
F1-score per epoch: It stabilizes around 0.91, again matching the performance of the multi-label version, despite the latter being trained on multiple labels concurrently.

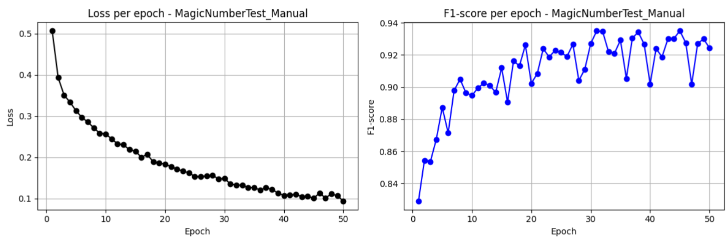


Figure 6: Training Loss and F1-Score Progression by Epoch (Magic Number Test)

Table 15: Classification Report for Magic Number Test

	Precision	**Recall**	**F1-score**	**Support**
Non	0.92	0.88	0.90	399
Oui	0.91	0.94	0.92	507
Accuracy			0.91	906

F1-score macro : 0.9123
F1-score micro : 0.9139

Interpretation:
Loss per epoch: The model quickly converges, with low final loss.
F1-score per epoch: The F1-score climbs to around 0.93, which is excellent. Still, this result

is very close to what the multi-label model already provides, meaning there's limited gain from training this label separately.

Summary:

While mono-label training produces strong results, the multi-label model demonstrates remarkable efficiency by delivering comparable, and in some cases equivalent, F1-scores across all test smells in a single training run. This not only simplifies the workflow but also confirms that the model can effectively handle inter-label interactions and shared features, making it more robust and versatile for real-world applications involving multiple test smells.

3.3.3 Test on a virgin dataset

Evaluation and Re-training on a Virgin Dataset

The first evaluation was conducted on a virgin dataset (10% random sample (997 rows)). This step provides an unbiased measurement of the model's generalization capability before any targeted fine-tuning. The table below summarizes the results from both the initial evaluation (Attempt 1) and after retraining on the same data (Attempt 2):

Table 16: Summary of Evaluation and Retraining on Virgin Dataset (All Four Test Smells)

Metric	EagerTest	AssertionRoulette	UnknownTest	MagicNumber
Attempt 1 — First Evaluation				
Actual positives	256	313	200	244
Predicted positives	449	454	543	433
Correct predictions	777	801	636	802
Errors	221	197	361	195
Lines to review (any error)	363 / 997 (36.4%)		541 / 997 (54.3%)	
Attempt 2 — After Retraining				
Predicted positives	259	283	157	240
True Positives (TP)	241	280	151	215
False Positives (FP)	18	3	6	25
False Negatives (FN)	15	33	49	29
True Negatives (TN)	724	682	791	728
Lines to review (any error)	66 / 997 (6.6%)		109 / 997 (10.9%)	

The evaluation reveals a clear improvement in the model's performance after retraining. In Attempt 1, the model struggled particularly with Unknown Test and Magic Number, leading to 54.3% of the lines requiring review, compared to 36.4% for Eager Test and Assertion Roulette. This indicates that the model had more difficulty generalizing to certain test smells initially.
After retraining (Attempt 2), the number of misclassified rows dropped significantly across all smells. For Eager Test and Assertion Roulette, review-worthy cases dropped to 6.6 %, and for

Unknown Test and Magic Number, they decreased to 10.9%. This shows that even a single round of targeted retraining on the same dataset substantially improved the model's predictive precision.
These results validate the effectiveness of iterative correction strategies and demonstrate the model's flexibility in refining its understanding of diverse test smells.

Re-training on misclassified data

In parallel, I conducted an iterative correction process over three attempts. After each iteration, the model was retrained exclusively on the lines it previously misclassified, simulating a "human" correction process.

Test Smells: Eager Test & Assertion Roulette

Table 17: Error Reduction Across Attempts : Eager Test & Assertion Roulette

Attempt	Eager Test	Assertion Roulette	Total Error Rows
Initial Prediction (Attempt 1)	221	197	363 (36%)
After Retraining (Attempt 2)	12	23	29 (2.9%)
Final Correction (Attempt 3)	0	1	1 (0.1%)

The model initially misclassified 36% of the dataset. After two retraining iterations, it correctly classified all but one row, showing strong capacity to improve when exposed to its own mistakes.

Test Smells: Unknown Test & Magic Number Test

Table 18: Error Reduction Across Attempts : Unknown Test & Magic Number Test

Attempt	Unknown Test	Magic Number Test	Total Error Rows
Initial Prediction (Attempt 1)	361	195	541 (54%)
After Retraining (Attempt 2)	5	6	11 (1.1%)
Final Correction (Attempt 3)	0	0	0 (0%)

The model faced more initial difficulty on these test smells on the first try. However, after two targeted retrainings, it was able to perfectly correct all remaining misclassifications.

Summary:

The combination of full evaluation on a virgin dataset and targeted retraining on misclassified data reveals the model's strengths in both generalization and adaptability. The virgin dataset experiment highlights initial weaknesses, especially for Unknown Test and Magic Number, but also shows that even a single round of retraining can reduce error rates by over 80%. Meanwhile, the iterative correction process demonstrates how quickly the model converges toward near-perfect classification when trained specifically on its past errors, reaching 0–1 misclassification after just two retraining cycles.

These findings suggest that while the model benefits from broad training for generalization, it also performs exceptionally well in an interactive or assisted annotation loop, making it a viable candidate for semi-automated labeling tools in test smell detection.

3.3.4 Summary of Experimental Results

Table 19: Comparison of macro and micro F1-scores of ANN and Random Forest models by test smell

Test Smell	Modl	Type	F1 macro	F1 micro
Eager Test	ANN	Multi-label	0.7452	0.7479
	ANN	Mono-label	0.7648	0.7660
	Random Forest	Multi-label	0.7963	0.7975
Assertion Roulette	ANN	Multi-label	0.8286	0.8348
	ANN	Mono-label	0.8067	0.8185
	Random Forest	Multi-label	0.8321	0.8462
Unknown Test	ANN	Multi-label	0.9343	0.9360
	ANN	Mono-label	0.9191	0.9205
	Random Forest	Multi-label	0.9252	0.9272
Magic Number	ANN	Multi-label	0.9288	0.9305
	ANN	Mono-label	0.9123	0.9139
	Random Forest	Multi-label	0.9377	0.9393

This study explored the application of an Artificial Neural Network (ANN) for the detection of multiple test smells in unit tests, using both multi-label and mono-label approaches.

The results show that the ANN model performs competitively, especially on well-defined and balanced test smells such as Unknown Test and Magic Number, where it achieves high F1-scores in both macro and micro averages.

Although traditional models like Random Forest achieve slightly higher F1-scores and remain strong baselines, the ANN model demonstrates comparable, and in some cases nearly equivalent, performance, particularly in the multi-label setup. This highlights the potential of neural networks to handle complex, concurrent predictions while maintaining robustness.

However, the Artificial Neural Network offers several advantages that make it a compelling alternative:

Flexibility: The ANN supports multi-label classification naturally, allowing it to learn multiple test smells simultaneously. The multi-label ANN configuration stands out for its ability to generalize across multiple test smells simultaneously, enabling efficient detection of co-occurring smells without a significant loss in accuracy. This offers a clear scalability advantage over training separate models for each label.

Retraining Capability: The model showed strong adaptability through iterative retraining on previously misclassified data, significantly improving its predictions on a new, unseen dataset. Mono-label training further demonstrated that the model can be fine-tuned to optimize performance on specific smells if needed.

Consistent Learning Curve: The loss curves and per-epoch F1-scores indicate good generalization and learning stability across different labels.

In conclusion, while Random Forest remains a solid reference point, the ANN model provides a promising solution for automated test smell detection and offers significant practical benefits in terms of adaptability, scalability, and generalization, especially in multi-label and iterative learning contexts.

4 Conclusion

This internship explored the use of Deep Learning, specifically Artificial Neural Networks (ANN), for the automated detection of test smells in unit test code. Beginning with a detailed literature review and dataset preparation, the work focused on constructing a high-quality, manually validated dataset, which served as the foundation for training and evaluating the model.

Several ANN architectures were tested, and a 3-layer configuration emerged as the best compromise between performance, training time, and simplicity. The model was assessed using both multi-label and mono-label classification approaches, with results showing strong performance across multiple test smells. Although Random Forest performed slightly better on some cases, the ANN achieved similar results, particularly in the multi-label setup, where it demonstrated strong generalization and scalability.

The model also showed excellent adaptability through iterative retraining: after being exposed to its own misclassifications, it was able to correct almost all errors in just two rounds. This highlights the ANN's adaptability to improve over time and its suitability for real-world assisted labeling systems.

In conclusion, this work highlights the relevance of Deep Learning for test smell detection, providing a robust, adaptable, and scalable solution that complements or even surpasses traditional methods. It lays a solid foundation for future research involving larger datasets and more diverse smells or more advanced neural architectures. Most importantly, these promising results could serve as the basis for developing a practical tool that automatically detects the presence of test smells in source code, ultimately helping developers improve the quality and maintainability of their test suites.

References

[1] A. van Deursen et al., *Refactoring test code*, 2001.

[2] B. Van Rompaey et al., *Characterizing the relative significance of a test smell*, 2006.

[3] B. Van Rompaey et al., *On the detection of test smells: A metrics-based approach for general fixture and eager test*, 2007.

[4] G. Bavota et al., *An empirical analysis of the distribution of unit test smells and their impact on software maintenance*, 2012.

[5] M. Greiler et al., *Automated Detection of Test Fixture Strategies and Smells*, 2013.

[6] F. Palomba et al., *On the diffusion of test smells in automatically generated test code: An empirical study*, 2016.

[7] F. Palomba et al., *Automatic test smell detection using information retrieval techniques*, 2018.

[8] D. Spadini et al., *On the relation of test smells to software code quality*, 2018.

[9] V. H. S. Durelli et al., *Machine Learning Applied to Software Testing: A Systematic Mapping Study*, 2019.

[10] M. I. Azeem et al., *Machine learning techniques for code smell detection: a systematic literature review and meta-analysis*, 2019.

[11] A. Ahmad et al., *An Evaluation of Machine Learning Methods for Predicting Flaky Tests*, 2020.

[12] T. Virgínio et al., *An empirical study of automatically-generated tests from the perspective of test smells*, 2020.

[13] P. Štochl and T. Pitner, *Artificial Intelligence in Software Test Automation: Overview*, 2020.

[14] D. Spadini et al., *Investigating severity thresholds for test smells*, 2020.

[15] A. Peruma et al., *TsDetect: An Open Source Test Smells Detection Tool*, 2020.

[16] W. Aljedaani et al., *Test smell detection tools: A systematic mapping study*, 2021.

[17] V. Pontillo et al., *Machine Learning-Based Test Smell Detection*, 2024.

[18] L. Martins et al., *On the diffusion of test smells and their relationship with test code quality of Java projects*, 2022.

[19] L. Martins et al., *Smart prediction for refactorings in the software test code*, 2022.

[20] S. Panichella et al., *Test smells 20 years later: detectability, validity, and impact on maintainability*, 2022.

[21] T. Virgínio et al., *On the test smells detection: an empirical study on the JNose Test accuracy*, 2021.

[22] A. Fontes and G. Gay, *The integration of machine learning into automated test generation: A systematic mapping study*, 2023.

[23] V. Pontillo et al., *Machine Learning-Based Test Smell Detection*, 2024.

[24] S. Mezzaro et al., *An Empirical Study on How Large Language Models Impact Software Testing Learning*, 2024.

[25] M. Aranda III et al., *Evaluating Large Language Models in Detecting Test Smells*, 2024.

5 Appendix

Test Smell	Description
Assertion Roulette	An assertion that lacks a descriptive message, making it difficult to understand what failed when the test breaks.
Constructor Initialization	A test class that uses a constructor for setup instead of a dedicated setUp() method.
Conditional Test Logic	A test method containing conditional structures (if, switch) or loops (for, while).
Duplicate Assert	Repetitive assertions with identical parameters.
Handling Exception	A test method that contains try/catch statements
Empty Test	A test method that does not contain any executable statements
Eager Test	A test method that invokes multiple methods from different classes under test
General Fixture	The setUp() method initializes objects that are not used by every test.
Ignored Test	A test method that contains an annotation to ignore the method (@Ignore).
Lazy Test	Multiple test methods call the same class under test methods.
Mystery Guest	A test method using external resources like files or databases.
Magic Number Test	An assertion that uses unexplained numeric literals directly instead of named constants.
Print Statement	A method that invokes print methods
Redundant Assertion	An assertion where the expected and actual values are identical
Resource Optimism	A test method that does not verify the existence of a file before using it
Sensitive Equality	A test that compares string representations of values
Sleepy Test	A statement invokes a sleep thread
Unknown Test	A test method that lacks any assertions
Verbose Test	A test method that has more than 30 lines

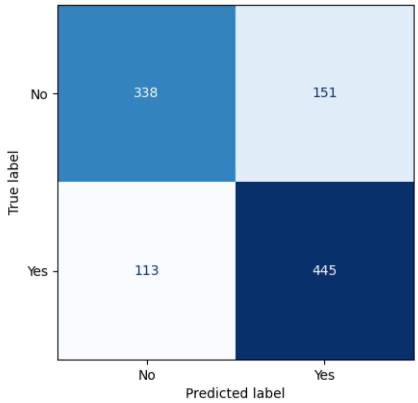
Table 20: Descriptions of test smells used in the database

Acronym	Full Name	Description
constructor	Constructor Usage	Indicates whether the method uses a constructor.
line	Line Number	Line number of the method in the file.
cbo	Coupling Between Objects	Number of different classes used by the test class.
wmc	Weighted Methods per Class	Total number of methods in the class, weighted by their complexity.
rfc	Response For a Class	Total number of methods that can be invoked in response to a message.
loc	Lines of Code	Number of lines of code in the method.
returnsQty	Return Statements Quantity	Number of `return` statements.
variablesQty	Variables Quantity	Total number of local variables declared.
parametersQty	Parameters Quantity	Number of parameters in the method.
methodsInvokedQty	Methods Invoked Quantity	Total number of invoked methods.
methodsInvokedLocalQty	Local Methods Invoked Quantity	Number of locally defined methods invoked.
methodsInvokedIndirectLocalQty	Indirect Local Methods Invoked Quantity	Number of indirectly invoked local methods.
loopQty	Loop Statements Quantity	Number of loop constructs (for, while, etc.).
comparisonsQty	Comparison Operators Quantity	Number of comparison operations (==, `!=`, etc.).
tryCatchQty	Try-Catch Blocks Quantity	Number of try-catch blocks.
parenthesizedExpsQty	Parenthesized Expressions Quantity	Number of expressions enclosed in parentheses.
stringLiteralsQty	String Literals Quantity	Number of string literal values (`"..."`).
numbersQty	Numbers Quantity	Number of numeric values used.
assignmentsQty	Assignments Quantity	Total number of assignment operations (=).
mathOperationsQty	Mathematical Operations Quantity	Total number of math operations (`+`, `-`, `*`, `/`).
maxNestedBlocksQty	Max Nested Blocks Quantity	Maximum depth of nested blocks (if, for, etc.).
anonymousClassesQty	Anonymous Classes Quantity	Number of anonymous classes used.
innerClassesQty	Inner Classes Quantity	Number of inner classes defined.
lambdasQty	Lambda Expressions Quantity	Number of lambda expressions used.
uniqueWordsQty	Unique Words Quantity	Number of unique words in the method body.
modifiers	Modifiers Used	Number of modifiers applied (public, static, etc.).
logStatementsQty	Logging Statements Quantity	Number of logging or print statements used.

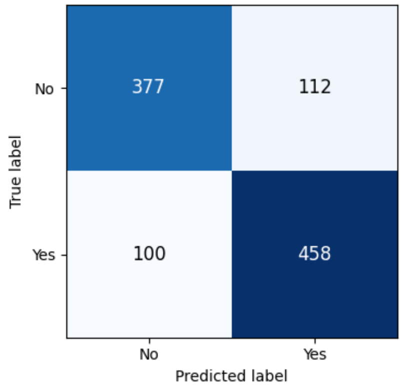
Table 21: Description of the features used in the database

```
# --- Model definition ---
class ANN(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(ANN, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, hidden_size // 2)
        self.relu2 = nn.ReLU()
        self.fc3 = nn.Linear(hidden_size // 2, num_classes)

    def forward(self, x):
        x = self.relu1(self.fc1(x))
        x = self.relu2(self.fc2(x))
        return self.fc3(x)

# --- Initialization ---
input_size = X_train.shape[1]
hidden_size = 256
num_classes = 2
model = ANN(input_size, hidden_size, num_classes)

# Handling class imbalance
train_class_counts = y_train.sum(axis=0)
total_train = len(y_train)
pos_weights = torch.tensor([
    (total_train - train_class_counts[i]) / train_class_counts[i]
    for i in range(num_classes)
], dtype=torch.float32)

# Loss, optimizer
criterion = nn.BCEWithLogitsLoss(pos_weight=pos_weights)
optimizer = optim.Adam(model.parameters(), lr=0.001)

# --- Training ---
num_epochs = 50
for epoch in range(num_epochs):
    model.train()
    for inputs, targets in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()

# --- Evaluation ---
model.eval()
with torch.no_grad():
    outputs = model(X_test_tensor)
    preds = torch.round(torch.sigmoid(outputs)).int().cpu().numpy()
    y_true = y_test.astype(int)
```

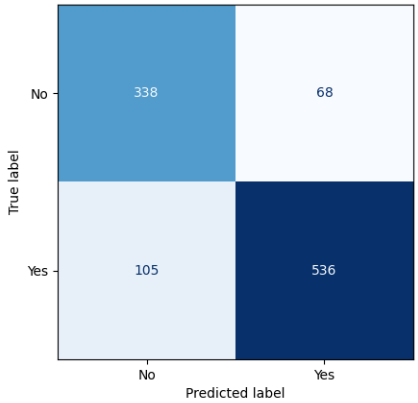
Listing 1: Code: ANN model definition, training and evaluation



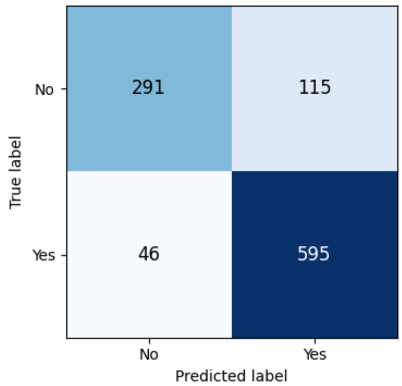
(a) Eager Test (Multi-label)



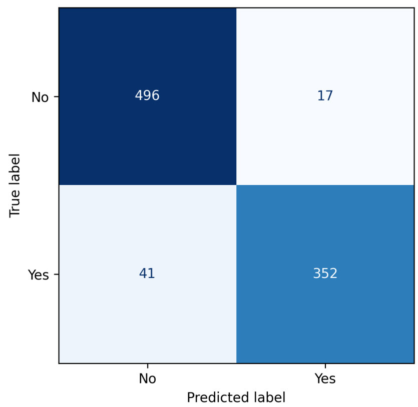
(b) Eager Test (Random Forest)



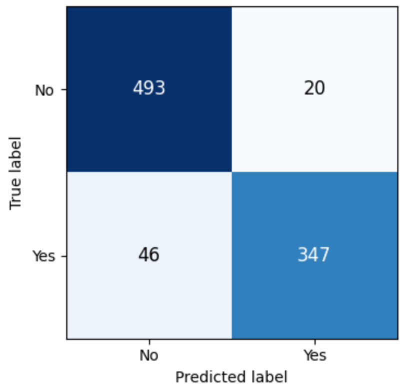
(c) Assertion Roulette (Multi-label)



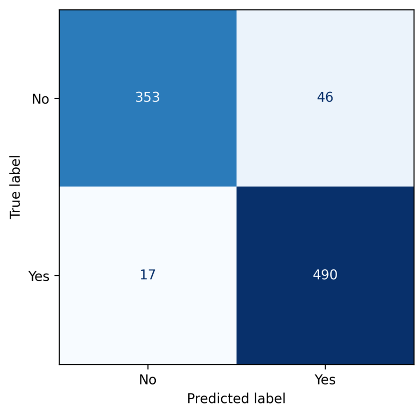
(d) Assertion Roulette (Random Forest)



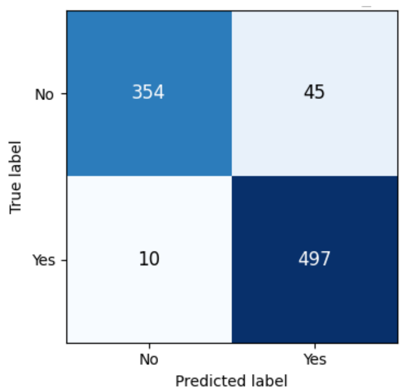
(e) Unknown Test (Multi-label)



(f) Unknown Test (Random Forest)



(g) Magic Number Test (Multi-label)



(h) Magic Number Test (Random Forest)

Figure 7: Comparison of confusion matrices between Multi-label and Random Forest models

Eager Test	**Assertion Roulette**	**Magic Number**	**Unknown Test**
loc: 0.1763	uniqueWordsQty: 0.1811	cbo: 0.3198	uniqueWordsQty: 0.1196
line: 0.1216	line: 0.125	rfc: 0.0739	line: 0.1157
rfc: 0.1	numbersQty: 0.1126	uniqueWordsQty: 0.0733	loc: 0.0802
uniqueWordsQty: 0.0991	loc: 0.0873	methodsInvokedQty: 0.0698	methodsInvokedQty: 0.077
methodsInvokedQty: 0.0892	rfc: 0.0736	line: 0.0695	stringLiteralsQty: 0.0741
cbo: 0.0725	variablesQty: 0.0619	loc: 0.0644	rfc: 0.0667
variablesQty: 0.0706	cbo: 0.0597	assignmentsQty: 0.0508	numbersQty: 0.0664
assignmentsQty: 0.0652	methodsInvokedQty: 0.0574	variablesQty: 0.0502	wmc: 0.0565
stringLiteralsQty: 0.0629	stringLiteralsQty: 0.0568	stringLiteralsQty: 0.0487	cbo: 0.0515
numbersQty: 0.0558	assignmentsQty: 0.05	numbersQty: 0.0308	loopQty: 0.0466
methodsInvokedLocalQty: 0.0151	modifiers: 0.0276	lambdasQty: 0.0281	mathOperationsQty: 0.0416
methodsInvokedIndirect-	wmc: 0.0173	anonymousClassesQty: 0.0232	variablesQty: 0.0386
-LocalQty: 0.0104	loopQty: 0.0167	methodsInvokedIndirect-	assignmentsQty: 0.0365
wmc: 0.0096	tryCatchQty: 0.0162	-LocalQty: 0.0229	comparisonsQty: 0.0321
tryCatchQty: 0.0095	mathOperationsQty: 0.0161	methodsInvokedLocalQty: 0.0214	maxNestedBlocksQty: 0.0266
maxNestedBlocksQty: 0.0086		parenthesizedExpsQty: 0.0093	

Table 22: Top 15 feature importances for each test smell