# Analysis of Machine Learning Detection of 'Spaghetti' Code Smell in Apache Fineract

## Case Study: Apache Fineract as a Testbed

Apache Fineract is a **mature open-source core banking system** that provides comprehensive financial services solutions[[1]](https://fineract.apache.org/#:~:text=Apache%20Fineract%C2%AE%20is%20a%20sophisticated,accounting%2C%20and%20extensive%20reporting%20capabilities). It powers digital financial platforms (microfinance, mobile banking, etc.) and is used globally in production environments. This project was chosen as a case study due to its **industry relevance, scale, and complexity**. With thousands of classes (approximately 4,059 classes analyzed in this study) comprising its codebase, Fineract represents a real-world system where code smells can significantly impact maintainability. Its large size and long evolution make it likely to contain **“architecture decay” issues** such as code smells. Moreover, being open source, Fineract allows extraction of static code metrics for analysis. In a mission-critical domain like banking, **code quality is paramount** – any presence of *Spaghetti Code* (or other smells) can hinder agility and increase the risk of defects. Therefore, Fineract provides an ideal testbed to evaluate code smell detection approaches on an **enterprise-scale, real-world codebase**. The goal of this analysis is to detect the *Spaghetti Code smell* in Fineract’s code using machine learning classifiers, and to evaluate how well traditional ML techniques perform on this task.

*(Spaghetti Code refers to a class that is poorly structured, typically very large and containing long methods with little modularization. Such classes often result from continuous quick fixes or procedural-style coding in an object-oriented system. They are hard to maintain and understand, making their detection and refactoring important*[*[2]*](https://www.cs.wm.edu/~denys/pubs/TSE'17-BadSmells.pdf#:~:text=complexity%20,droid.org)[*[3]*](https://dibt.unimol.it/staff/fpalomba/documents/C44.pdf#:~:text=Spaghetti%20Code,rule%20for%20this%20smell%20considers)*.)*

## Classifier Performance on Code Smell Detection

Using the CK metrics extracted from Fineract (described later), we trained three **traditional machine learning classifiers** – **J48 (C4.5 decision tree)**, **Random Forest**, and **SVM (using Weka’s SMO)** – to classify classes as *Spaghetti Code smell = {yes, no}*. We experiment with two feature sets derived from the code metrics: a **57-attribute set** (the full spectrum of collected metrics) and a **13-attribute set** (a reduced subset of key metrics selected for their relevance). The classifiers were evaluated using 10-fold cross validation on the dataset of 4,059 class instances (with the minority class *“Spaghetti”* present in only 17 classes, ~0.42%). Performance is reported in terms of overall **accuracy** and class-level **precision, recall, F1-score** for detecting the *Spaghetti* smell (the positive class), as well as the **ROC AUC** and **PRC AUC** (Area Under the Precision-Recall Curve) for each classifier. Table 1 summarizes the results on the **full 57-metric feature set**, and Table 2 shows results on the **reduced 13-metric set**.

*Table 1. Performance of ML classifiers for Spaghetti Code detection (57 metrics feature set).*

| **Classifier** | **Accuracy** | **Precision (Spaghetti)** | **Recall (Spaghetti)** | **F1-score** | **ROC AUC** | **PRC AUC** |
| --- | --- | --- | --- | --- | --- | --- |
| J48 Decision Tree | 99.93% | 88.9% | 94.1% | 91.4% | 0.971 | 0.915 |
| Random Forest | 99.90% | 93.3% | 82.4% | 87.5% | 1.000 | 0.975 |
| SVM (SMO) | 99.95% | 89.5% | 100.0% | 94.4% | 1.000 | 0.895 |

*Table 2. Performance of ML classifiers for Spaghetti Code detection (13 selected metrics).*

| **Classifier** | **Accuracy** | **Precision (Spaghetti)** | **Recall (Spaghetti)** | **F1-score** | **ROC AUC** | **PRC AUC** |
| --- | --- | --- | --- | --- | --- | --- |
| J48 Decision Tree | 99.95% | 89.5% | 100.0% | 94.4% | 1.000 | 0.975 |
| Random Forest | 99.93% | 88.9% | 94.1% | 91.4% | 1.000 | 0.990 |
| SVM (SMO) | 99.95% | 89.5% | 100.0% | 94.4% | 1.000 | 0.895 |

In both tables, **“Precision (Spaghetti)”** refers to the precision for the positive class (i.e., how many of the classes predicted as smelly were truly smelly), and **“Recall (Spaghetti)”** is the true positive rate (how many of the actual smelly classes were detected). **Accuracy** is extremely high (>99.9%) for all models due to the large proportion of non-smelly classes; this metric alone is less informative given the class imbalance. More telling are the precision, recall, and F1-score for the *Spaghetti* class. All three classifiers achieve strong detection capability, with F1-scores in the 87–94% range, indicating that despite the rarity of the smell, the models can identify it with high precision and recall.

**Confusion matrix analysis** provides further insight. Table 3 lists the confusion matrix counts for each classifier under both feature sets. Each cell shows the count of classes in a category (True Positives = actual smell correctly detected, False Negatives = missed smell, False Positives = false alarms, True Negatives = correctly identified clean classes). Notably, in the **57-attribute scenario**, J48 correctly identified 16 out of 17 Spaghetti classes (with 1 missed, and 2 false positives), Random Forest detected 14 out of 17 (3 missed, but only 1 false positive), and SVM caught all 17 (0 missed) at the cost of 2 false positives. With the **13-attribute set**, J48 and SVM both achieved a perfect 100% recall (no false negatives, all 17 smelly classes found) with 2 false positives each, while Random Forest detected 16/17 (one miss) with 2 false positives. These results show a slight trade-off between classifiers: Random Forest tends to be more conservative (fewer false alarms but lower recall), whereas SVM is more aggressive (no misses but a couple of false alarms), and the decision tree lies in between.

*Table 3. Confusion matrix counts for Spaghetti smell detection on Fineract (Yes = “Spaghetti” class present, No = not present).*

| **Classifier** | **Dataset** | **True Positives** <br/>(Actual Yes → Predicted Yes) | **False Negatives** <br/>(Actual Yes → Predicted No) | **False Positives** <br/>(Actual No → Predicted Yes) | **True Negatives** <br/>(Actual No → Predicted No) |
| --- | --- | --- | --- | --- | --- |
| J48 Decision Tree | 57 attributes | 16 | 1 | 2 | 4040 |
| Random Forest | 57 attributes | 14 | 3 | 1 | 4041 |
| SVM (SMO) | 57 attributes | 17 | 0 | 2 | 4040 |
| J48 Decision Tree | 13 attributes | 17 | 0 | 2 | 4040 |
| Random Forest | 13 attributes | 16 | 1 | 2 | 4040 |
| SVM (SMO) | 13 attributes | 17 | 0 | 2 | 4040 |

From the confusion data, we observe that **all models correctly classify over 4040 non-smelly classes**, and more importantly, **they identify the majority (or all) of the 17 Spaghetti classes**. The *decision tree* and *SVM* achieve the highest recall (ensuring no smelly class goes undetected in the 13-attribute experiment), whereas *Random Forest* has a slightly lower recall on the full feature set (missed 3 smells) but also the fewest false positives in that case. Overall, the differences between using 57 vs 13 metrics are minor in terms of accuracy, but the simplified 13-metric model actually improved or maintained the recall for J48 and Random Forest. This suggests that removing less relevant metrics did **not harm detection performance and even reduced misclassifications**, likely by eliminating noise.

## Visual Comparison of Classifier Results

To better compare the classifiers, **Figure 1** illustrates the F1-score (the harmonic mean of precision and recall) for detecting the Spaghetti smell by each algorithm, on both the full and reduced feature sets. Higher F1 indicates a better balance of catching smelly classes while avoiding false alarms. All approaches exceed 87% F1, with most in the 91–94% range. The figure highlights that using the **selected 13 metrics yields F1 equal or higher** than using all 57 metrics for all classifiers. In particular, J48 and Random Forest saw an F1 increase when using the reduced metric set (e.g., Random Forest F1 rose from ~87.5% to 91.4%). SVM maintained an F1 of ~94.4% on both sets. This visual comparison reinforces that **feature selection** did not degrade performance and in fact improved consistency among classifiers.

*Figure 1: F1-score (%) for Spaghetti code smell detection by each classifier, comparing the full 57-metric set vs. the 13-metric subset. Each pair of bars shows the F1 achieved with 57 attributes (blue) and with 13 attributes (orange) for a given classifier. Higher is better; all models perform in a high range, with J48 and Random Forest improving when using the reduced feature set.*

The plot in Figure 1 also reveals the subtle differences between classifiers. **SVM (SMO)** attains the highest F1 (~94.4%) and is **unaffected by the feature reduction**, indicating it was likely focusing on the most relevant metrics even with the full set. **Decision Tree (J48)** improves from ~91.4% to 94.4% F1 with fewer attributes, matching SVM, which suggests that the extra metrics in the 57-feature set may have introduced branches that slightly hurt generalization. **Random Forest** has the lowest F1 on the full set (~87.5%), but improves markedly to ~91.4% with selected metrics, nearly closing the gap. This implies that the ensemble of trees benefited from a cleaner, more pertinent feature set (likely reducing overfitting on spurious metrics given the very small number of positive examples). Across all models, we see **consistently strong performance**, with the decision tree and SVM ultimately achieving identical high F1 on the reduced set.

## Discussion: Performance Trends and Challenges

**Performance Trends:** The results demonstrate that traditional ML classifiers can effectively detect the *Spaghetti Code* smell in a large industrial project when provided with appropriate metrics. All three classifiers achieve extremely high overall accuracy, though this is largely due to the overwhelming majority of classes being clean (non-smelly). A more meaningful comparison is via the precision-recall trade-offs for the positive (smelly) class. Here, we observed two different tendencies: one classifier (Random Forest) leaned toward **higher precision** (fewer false positives) at the expense of recall, while another (SVM) achieved **perfect recall** (no missed smelly classes) at the expense of a couple of false alarms (slightly lower precision). The decision tree’s performance fell in between, managing a balanced high recall and decent precision. This pattern is intuitive – Random Forest, by voting across many decision trees, may require stronger consensus to label a class as smelly, which can filter out noise (few false positives) but also means a *conservative bias* that can miss some true smells. In contrast, the SVM sought to separate the classes with maximum margin; given the small number of positive instances, it appears the SVM decision boundary encompassed all positives (hitting 100% recall) while tolerating a small amount of spill-over of negatives into the positive region (leading to two false positives). The decision tree, being a single optimized tree, found rules that captured most positives and kept false positives low, resulting in high F1 both before and after feature selection.

**Effect of Feature Set Size:** An important trend is that **using the reduced 13-attribute set yielded comparable or better results** than the full 57 attributes. This suggests that many of the 57 collected metrics were redundant or irrelevant for the Spaghetti smell. By focusing on the most informative metrics (such as size and complexity measures), the classifiers avoided overfitting to spurious correlations present in the larger feature space. The Random Forest model benefited the most from feature reduction (its recall increased notably), which underscores how **feature selection can mitigate the curse of dimensionality** and the impact of having far more features (57) than positive examples (17) in training. The decision tree also became slightly more effective with a leaner feature set, which likely led to a simpler tree that generalized better. SVM’s performance remained the same, implying that the extra metrics did not confuse its hyperplane significantly – perhaps because the key support vectors were distinguishable by the core metrics alone. In practical terms, this result is encouraging: it means **a small number of well-chosen metrics can successfully flag Spaghetti code**, simplifying the model and interpretation, which is valuable for explainability.

**Challenges:** Despite the high performance metrics, there are **inherent challenges in using traditional ML for code smell detection**. One major challenge is the **class imbalance** – here only 0.4% of classes were Spaghetti code. Without careful model tuning or evaluation, a classifier could trivially achieve 99% accuracy by predicting all classes as “clean”. In our study, the classifiers did avoid this trivial solution and truly learned to identify the rare smell instances, but this may not always be the case. Techniques such as stratified cross-validation (which we used), resampling, or cost-sensitive learning are important to ensure the model pays adequate attention to the minority class. Another challenge is that **code metrics alone may not capture every aspect** of a smell. *Spaghetti Code*, in particular, is characterized by poor structure (e.g., long, tangled methods with no parameters and a lack of modular separation). These traits fortunately correlate strongly with certain metrics (like lines of code and cohesion, as discussed below), making Spaghetti Code amenable to metric-based detection. However, other smells might depend on more semantic or context-specific cues that basic metrics don’t capture. In those cases, a pure ML approach on metrics might struggle. Even for Spaghetti Code, if there were borderline cases (e.g., classes that are large but somewhat structured), the ML model might misclassify them due to overlapping metric values. This could lead to **false positives** (flagging a large class that is actually well-designed) or **false negatives** (overlooking an unconventional spaghetti-like class that doesn’t strictly meet the usual metric pattern). In our Fineract dataset, the Spaghetti instances appear to be well-separated from normal classes (hence the near-perfect performance), but this might not generalize to all projects.

Another practical challenge is **interpretability** of ML models versus heuristic rules. A decision tree like J48 is relatively interpretable – we could extract the decision rules it learned (for instance, we might find a rule like *“if LOC > X and LCOM > Y then class is Spaghetti”*). Such a rule could directly inform developers about thresholds at which their classes become problematic, aligning with known guidelines (e.g., a common heuristic is that a class over ~600 LOC with at least one long method signals Spaghetti Code[[3]](https://dibt.unimol.it/staff/fpalomba/documents/C44.pdf#:~:text=Spaghetti%20Code,rule%20for%20this%20smell%20considers)). In contrast, an SVM’s reasoning is less transparent, though its performance is high. In our analysis, all approaches yielded similar outcomes, but **for developer adoption, the decision tree or rule-based insights may be preferable**, as they provide actionable thresholds for refactoring. This points to a broader challenge: balancing **accuracy with explainability** in smell detection. Traditional ML models can achieve high accuracy, but if their predictions aren’t explainable, developers might be reluctant to trust and use them.

In summary, the performance trends in this case study are very positive – *traditional classifiers, using static code metrics, successfully detected Spaghetti Code instances with high fidelity*. The main challenges lie in handling skewed data distributions and ensuring the model’s knowledge can be communicated and applied by developers. Our findings indicate that, at least for well-defined smells like Spaghetti Code, classical ML is a viable solution. Nonetheless, one must carefully validate such models on each new codebase and potentially augment them with additional context (or combine with static analysis rules) for more complex or subtle smells.

## CK Metrics Extraction and Definitions

To enable the ML detection of code smells, we relied on static **code metrics** extracted from the Fineract codebase using the *CK tool*. The CK tool (named after Chidamber & Kemerer) computes a suite of **object-oriented design metrics** for each class in the system. This includes the classic CK metrics (WMC, DIT, NOC, CBO, RFC, LCOM) along with extended metrics (e.g., various counts of methods, fields, fan-in/out, etc.). Below we define some key metrics from this suite and explain their relevance:

* **Lines of Code (LOC):** The number of lines of source code in the class (often measured as non-comment, non-blank lines). LOC is a basic size metric – very large classes (high LOC) tend to be more difficult to maintain and are often associated with **Blob/Spaghetti Code** smells. In our context, a class with exceptionally high LOC is a red flag, as Spaghetti Code is by definition a class that has grown large and unwieldy[[3]](https://dibt.unimol.it/staff/fpalomba/documents/C44.pdf#:~:text=Spaghetti%20Code,rule%20for%20this%20smell%20considers).
* **Coupling Between Object Classes (CBO):** Measures how many other classes a given class is **coupled** to (i.e., directly depends on)[[4]](https://medium.com/@benkaddourmed54/analyzing-ck-metrics-results-and-quality-assessment-a70ba56534f0#:~:text=Coupling%20Between%20Object%20Classes%20). Every usage of another class (calling its methods, accessing its fields, etc.) counts as a coupling. A high CBO indicates **high interdependence**[[5]](https://medium.com/@benkaddourmed54/analyzing-ck-metrics-results-and-quality-assessment-a70ba56534f0#:~:text=Coupling%20Between%20Object%20Classes%20), meaning the class is tightly connected with many others, which can make changes risky and propagation of errors more likely. While *Spaghetti Code* is primarily a structural issue inside a class, high coupling can exacerbate its impact by entangling the class with many parts of the system. Conversely, a Spaghetti class might surprisingly have *low CBO* if it does a lot of work internally without relying on others – in Fineract we examine this in the next section.
* **Response For a Class (RFC):** Counts the number of distinct methods that can be executed in response to a message (method call) to that class[[6]](https://medium.com/@benkaddourmed54/analyzing-ck-metrics-results-and-quality-assessment-a70ba56534f0#:~:text=Response%20For%20a%20Class%20). In simpler terms, RFC is the count of all methods in the class plus the methods it invokes on other classes. A higher RFC means the class has greater **responsibility and communication** with others[[6]](https://medium.com/@benkaddourmed54/analyzing-ck-metrics-results-and-quality-assessment-a70ba56534f0#:~:text=Response%20For%20a%20Class%20). A class with a very high RFC is doing a lot, potentially too much – this often correlates with large, complex classes that might be smells (e.g., a *God Class* or *Spaghetti Code* handling many operations).
* **Lack of Cohesion of Methods (LCOM):** Measures the degree to which methods of a class are disjoint – it quantifies how **unrelated the methods are** in terms of shared attributes or functionality[[7]](https://medium.com/@benkaddourmed54/analyzing-ck-metrics-results-and-quality-assessment-a70ba56534f0#:~:text=Lack%20of%20Cohesion%20in%20Methods,LCOM). There are various formulations of LCOM; conceptually, a high LCOM value means the class’s methods do not share many common fields or logic, indicating the class may be trying to do many different things (low cohesion)[[7]](https://medium.com/@benkaddourmed54/analyzing-ck-metrics-results-and-quality-assessment-a70ba56534f0#:~:text=Lack%20of%20Cohesion%20in%20Methods,LCOM). High LCOM is a **strong indicator of design problems**: a cohesive class should represent one abstraction, whereas a Spaghetti Code class might have sections of code doing disparate tasks. For instance, if a class defines many fields and each big method uses a different subset of those fields without overlap, the LCOM metric will be high, reflecting that split-brain behavior.
* **Weighted Methods per Class (WMC):** Although not explicitly listed in the question, WMC is another important CK metric. It is the sum of the complexities of all methods in the class[[8]](https://medium.com/@benkaddourmed54/analyzing-ck-metrics-results-and-quality-assessment-a70ba56534f0#:~:text=Weighted%20Methods%20per%20Class%20) (often using cyclomatic complexity for each method). A simpler interpretation is the number of methods (if each method is weighted equally). A higher WMC means the class has **more functionality and/or more complex methods**, which usually makes it harder to maintain. Spaghetti classes often have both *many* methods and *complex (long)* methods, which would drive WMC up. In fact, related smells like *Complex Class* are defined by high WMC. In our detection, WMC (or metrics that proxy complexity) likely plays a role in identifying Spaghetti Code, since long methods would increase the complexity count.

These metrics (and others in the 57-feature set) were extracted using the CK tool via static analysis of Fineract’s Java code. They collectively quantify different dimensions of code structure – **size**, **inheritance depth**, **number of children**, **coupling, responsibility, cohesion, complexity,** etc. By providing these quantitative measures, we can feed them into ML classifiers to learn patterns associated with code smells. For example, one would expect a Spaghetti Code class to exhibit *above-average size (LOC)*, *low cohesion (high LCOM)*, possibly *high complexity (WMC)*, and maybe *unusually high or low coupling* (depending on whether the class tries to do everything internally or delegates to many others). In contrast, a well-designed class might have moderate LOC, moderate coupling, and high cohesion (LCOM near 0), and would be classified as not smelly by the model.

## Insights from Fineract’s Metrics and Code Smells

Analyzing the CK metrics from the Fineract codebase provides insights into **which metrics correlate most with the Spaghetti smell** and how they influenced classifier performance. We found that the classes flagged as *Spaghetti Code* in Apache Fineract indeed stand out in terms of certain metric values:

* **Extreme Size and Complexity:** The Spaghetti classes in Fineract are **much larger than average classes**. For instance, the mean LOC per class in Fineract is relatively modest (many classes are small utility or model classes), whereas each identified Spaghetti class had hundreds of lines of code. In fact, some approached or exceeded the often-cited threshold of 600 LOC for Spaghetti Code[[3]](https://dibt.unimol.it/staff/fpalomba/documents/C44.pdf#:~:text=Spaghetti%20Code,rule%20for%20this%20smell%20considers). Along with sheer size, these classes also had **very high WMC** – they contained numerous methods, many of which were lengthy and complex. This aligns with the smell’s definition (long methods, few parameters, indicating procedural chunks of logic). The classifiers picked up on this pattern: metrics directly or indirectly capturing class size (LOC, total number of methods, perhaps maximum method length) were key in separating these smelly classes from the rest. In the reduced 13-metric model, it is almost certain that LOC and/or WMC were included, as they carry strong predictive signal; the perfect or near-perfect recall achieved suggests that a simple size threshold could almost cleanly discriminate the known Spaghetti classes.
* **Low Cohesion (High LCOM):** We observed that the Spaghetti classes showed **notably higher LCOM values** compared to typical classes in Fineract. This indicates that the methods inside those classes were not working on common data or related functionality. For example, one Spaghetti class (for confidentiality we don’t name it here) had multiple large methods each performing a distinct workflow, sharing almost no common fields – essentially functioning like independent scripts thrown into one file. Such a class scored a very high LCOM (signifying disjoint sets of methods), whereas a well-structured class (e.g., one following single-responsibility principle) would have low LCOM as its methods collaborate on the same state. The ML models likely leveraged LCOM as an input: the decision tree, for instance, might have a rule using LCOM to identify classes that lack cohesion. This is supported by the fact that cohesion was part of the CK metric set and the reduced feature set’s strong performance (suggesting that at least one cohesion metric was among the chosen 13).
* **Coupling and External Interactions:** Interestingly, **coupling metrics (CBO, fan-in/out)** did not show as clear-cut a pattern for the Spaghetti classes as size and cohesion did. In some of Fineract’s Spaghetti classes, CBO was moderate – meaning these classes weren’t necessarily heavily connected to many others; they were “spaghetti” in their internal structure more than in their tentacles to other classes. This makes sense: a Spaghetti class often tries to do everything on its own, so it might not need to call many external classes (thus CBO could remain average). In one case, a large Spaghetti class was largely self-contained (low CBO ~ 2 or 3), which is low for a class of that size. However, other smelly classes did have higher CBO and RFC, indicating they were both large and performing many external calls (making them even more problematic, similar to a *God Class*). Overall, coupling alone was not a definitive indicator of Spaghetti Code in Fineract – some non-smelly infrastructure classes (e.g., those managing transactions or Spring framework integration) had high CBO due to legitimate interactions but were well-structured internally. The classifiers, especially Random Forest, could handle such nuances by considering multiple features: e.g., a high CBO combined with high LOC and low cohesion would still point to a smell, whereas high CBO with moderate LOC and high cohesion might not. The inclusion of RFC in the feature set (which overlaps with CBO by counting external method calls) ensured that classes doing “too much” in terms of external interactions could be identified if that coincided with other smell symptoms.
* **Selected Feature Importance:** Although we did not explicitly compute feature importance in this write-up, the fact that the 13-attribute model performs so well strongly implies that those 13 metrics captured the essence of the Spaghetti smell. It is likely that **LOC, number of methods, average/maximum method size, WMC, LCOM,** and perhaps **number of parameters or parameter-less methods** were among the selected features. These are exactly the factors one would intuitively use to detect Spaghetti Code. For instance, prior research DECOR uses *“class LOC > 600 and at least one long method with no parameters”* as a rule[[3]](https://dibt.unimol.it/staff/fpalomba/documents/C44.pdf#:~:text=Spaghetti%20Code,rule%20for%20this%20smell%20considers) – our ML approach essentially rediscovered a similar rule, but in a data-driven way, by considering metrics like LOC (for class size) and something akin to “NOMPARAM” (number of parameterless long methods) or proxies for it. Indeed, we had metrics such as the count of methods, and possibly the ratio of methods without params or average parameters per method. A Spaghetti class tends to have **uncharacteristically many methods that take no parameters** (because they rely on global or class-level data instead), which could be inferred from a combination of high method count and maybe lower average number of parameters. While our dataset’s exact 13 features aren’t listed here, the strong performance suggests that any metrics not closely related to these characteristics were likely dropped. For example, inheritance metrics (DIT, NOC) probably had little impact – Fineract’s Spaghetti classes didn’t necessarily have deep inheritance (in fact, many were regular concrete classes), so including DIT/NOC didn’t aid detection. By contrast, metrics reflecting **internal complexity and organization were crucial**.
* **Metric Distribution in Fineract:** Aside from detecting smells, the CK analysis provided a general quality profile of Fineract. Most classes had metric values in normal ranges (e.g., median LOC per class was fairly low, and median LCOM not extreme), indicating the system is largely well-structured. The Spaghetti Code instances were **outliers** in the metric distribution. For example, plotting a histogram of class LOC showed a long tail – with a tiny fraction of classes far to the right (very high LOC). Those outliers corresponded to the detected Spaghetti smells. This outlier nature made them easier for ML models to isolate. However, it’s worth noting that if a project has many borderline cases (e.g., dozens of somewhat large classes), distinguishing which ones are “smelly” can be harder. In Fineract, the smells stood out enough that even a simple tree could separate them with few misclassifications. This implies that **Fineract’s code metrics provided a clear signal for Spaghetti Code**, which was capitalized on by the classifiers.

In conclusion, the CK metrics analysis of Apache Fineract revealed clear correlations: *Spaghetti Code classes are characterized by excessive size (LOC), high complexity (WMC), low cohesion (LCOM), and in some cases increased interactions (RFC)*. These insights not only explain why the machine learning models performed so well (the smelly and non-smelly classes occupy distinct regions in the metric space), but also provide actionable information for developers. For instance, developers can monitor classes that exceed certain LOC or complexity thresholds and have poor cohesion as likely refactoring targets. The success of the ML detectors in this thesis section reinforces the idea that **static code metrics, when carefully chosen, are effective predictors of code smells** – a finding consistent with earlier works like code smell detection tools WekaNose and DECOR that likewise rely on metric thresholds[[3]](https://dibt.unimol.it/staff/fpalomba/documents/C44.pdf#:~:text=Spaghetti%20Code,rule%20for%20this%20smell%20considers). The advantage of the ML approach is that it can combine multiple metrics and find an optimal boundary, rather than relying on one-size-fits-all thresholds; yet, as we saw, in this case the boundary found aligns with intuitive threshold-based rules. This gives confidence in the model’s validity and provides a double-check: the ML results empirically support the heuristic wisdom that, for example, *“a 1000-line class with disparate methods is very likely a design smell.”* Going forward, these insights could guide both automated detection (in CI pipelines) and preventive measures (setting up metric guards to flag potential spaghetti code before it grows out of control). Each metric tells a part of the story of code quality, and together, they enabled a comprehensive evaluation of Apache Fineract’s health with respect to the Spaghetti Code smell.

[[1]](https://fineract.apache.org/#:~:text=Apache%20Fineract%C2%AE%20is%20a%20sophisticated,accounting%2C%20and%20extensive%20reporting%20capabilities) Apache Fineract®

<https://fineract.apache.org/>

[[2]](https://www.cs.wm.edu/~denys/pubs/TSE'17-BadSmells.pdf#:~:text=complexity%20,droid.org) TSE2653105.pdf

<https://www.cs.wm.edu/~denys/pubs/TSE'17-BadSmells.pdf>

[[3]](https://dibt.unimol.it/staff/fpalomba/documents/C44.pdf#:~:text=Spaghetti%20Code,rule%20for%20this%20smell%20considers) dibt.unimol.it

<https://dibt.unimol.it/staff/fpalomba/documents/C44.pdf>

[[4]](https://medium.com/@benkaddourmed54/analyzing-ck-metrics-results-and-quality-assessment-a70ba56534f0#:~:text=Coupling%20Between%20Object%20Classes%20) [[5]](https://medium.com/@benkaddourmed54/analyzing-ck-metrics-results-and-quality-assessment-a70ba56534f0#:~:text=Coupling%20Between%20Object%20Classes%20) [[6]](https://medium.com/@benkaddourmed54/analyzing-ck-metrics-results-and-quality-assessment-a70ba56534f0#:~:text=Response%20For%20a%20Class%20) [[7]](https://medium.com/@benkaddourmed54/analyzing-ck-metrics-results-and-quality-assessment-a70ba56534f0#:~:text=Lack%20of%20Cohesion%20in%20Methods,LCOM) [[8]](https://medium.com/@benkaddourmed54/analyzing-ck-metrics-results-and-quality-assessment-a70ba56534f0#:~:text=Weighted%20Methods%20per%20Class%20) Analyzing CK Metrics Results and Quality Assessment | by Benkaddour Racim | Medium

<https://medium.com/@benkaddourmed54/analyzing-ck-metrics-results-and-quality-assessment-a70ba56534f0>