Paper 1: IntelliEstimator: Estimating Maintenance Cost and Prediction of Software Quality, Reliability, and Maintenance

- **Features Trained On: A comprehensive set of static code metrics and novel, engineered features. These include:**
    - **Static Code Metrics: Cyclomatic Complexity (CYCLO), Lines of Code (LOC), Coupling Between Objects (CBO), Depth of Inheritance Tree (DIT), Weighted Methods per Class (WMC), Response For a Class (RFC), Halstead Metrics.**
    - **Code Smell Metrics: Quantified metrics for Long Methods, Large Classes, Duplicate Code, Broken Modularization, etc.**
    - **Novel Composite Metrics: Maintenance Index (MI), Quality Index (QI), Security Index (SI), WMCDIT Ratio, Cyclomatic Density, Implementation Smell Density (ISD), Abstraction Density (AD).**
- **Data Sources: 10,000 code samples from 25 Java projects sourced from public GitHub repositories. Metrics were extracted by parsing source code into Abstract Syntax Trees (ASTs) and labeled using historical commit and issue tracker data.**
- **Models Used: A Stacking RFCXGB Classifier, which is an ensemble model combining:**
    - **Base Models (Level 0): Random Forest (RFC) and XGBoost (XGB).**
    - **Meta-Model (Level 1): Logistic Regression.**
- **Model Targets: The primary target is the classification of Maintainability (High/Low). This enables derived predictions for:**
    - **Software Quality Level (High/Medium/Low)**
    - **Software Security Level (High/Low)**
    - **Maintenance Cost (in currency) and Effort (in person-months)**
    - **Project Reliability (High/Medium/Low)**

Paper 2: Navigating the Software Quality Maze: Detecting Scattered and Tangled Crosscutting Quality Concerns

- **Features Trained On: A multi-modal feature set extracted from bug report text:**
  - **Semantic Features: Subject-Verb-Object triplets from parsed dependency trees, augmented with BERT and analyzed with entropy measures.**
  - **Lexical Features: N-grams, Part-of-Speech (PoS) tag sequences, clause-level tags, lexical fractions, and syntactic subtree features.**
  - **Shallow Features: Text length, Bag-of-Words (BoW), and other surface-level metrics.**
- **Data Sources: 5,400 manually labeled bug reports from open-source projects, collected from Bugzilla (Apache HTTP Server, Eclipse IDE) and Jira (Derby, Drools, Groovy, Maven). Reports were labeled based on a quality taxonomy (ISO 25010, FURPS).**
- **Models Used: Multiple classic machine learning classifiers were trained and compared, including:**
  - **Support Vector Machine (SVM)**
  - **Logistic Regression (LR)**
  - **Random Forest (RF)**
  - **Multinomial Naive Bayes (MNB)**
  - **k-Nearest Neighbors (k-NN)**
  - **Decision Tree (J48)**
  - **A One-vs-All (OvA) strategy was used for multi-label classification, with SMOTE for handling data imbalance.**

- **Model Targets: Multi-label classification of bug reports into one or more quality categories:**
    - **Quality Types: Reliability, Maintainability, Usability, Portability, Security, Performance.**
    - **Other Types: Functional (FR) and Non-Code (NC).**

Paper 3: What Really Changes When Developers Intend to Improve Their Source Code: A Commit-Level Study

- **Features Trained On:** The model was trained exclusively on the **textual content of commit messages**.
- **Data Sources:** 125,482 commits from 54 Apache Java projects. A ground-truth set of 2,533 commits was first manually classified by researchers, which was then used to train the model to label the entire dataset.
- **Models Used:** A deep learning model for Natural Language Processing (NLP):
    - **seBERT**: A BERT model pre-trained exclusively on software engineering data and fine-tuned on the manually classified commit messages.
- **Model Targets:** A three-class classification of **developer intent** based on commit messages:
    - **Perfective:** Commits intended for internal quality improvements (code structure, readability).
    - **Corrective:** Commits for bug fixes and external quality improvements.

Other: All other commits (e.g., feature additions, repository maintenance).

Conclusion

The reviewed papers provide a strong methodological foundation for our specific goal: **predicting code quality scores (0.0-1.0) using machine learning models based on static code analysis of source code**. Paper 1 (IntelliEstimator) offers the most directly applicable approach, demonstrating how static code metrics like complexity, coupling, and size can be effectively used as features for ML models. While their current implementation uses classification (High/Low maintainability), their feature engineering and model architecture could be adapted for regression to predict continuous quality scores between 0.0 and 1.0. The success of their ensemble model (Stacking RFCXGB) validates that sophisticated ML techniques can capture the complex relationships between static code properties and quality attributes.

The composite metrics introduced in Paper 1, particularly the Maintenance Index (MI) and Quality Index (QI), provide a valuable blueprint for developing our own normalized scoring system. We can build upon this approach to create a unified quality score that aggregates multiple quality dimensions into a single, interpretable value. Papers 2 and 3, while focused on different aspects of quality assessment, reinforce the importance of comprehensive feature engineering and robust model training.

This research validates our proposed direction of building a tool that extracts static code metrics from source repositories and uses trained ML models to predict continuous quality scores. By adapting these methodologies for regression tasks and developing appropriate normalization techniques, we can create a system that provides granular, quantitative quality assessments in the 0.0-1.0 range, offering developers more nuanced insights into code quality than simple binary classifications.