RPI Case Presentation

ECO 4934
Topics in Econometrics
Summer C, 2024

Group B: Jonathon Lewis
Python Irvin
Alexis Leclerc
Polina Baikova

July 31, 2024

Problem Description

In continuous software engineering, each major release introduces significant new or modified functionality. Developers strive to resolve as many bugs as possible. Analyzing bug reports leverages extensive data from defect tracking systems to gain valuable insights into the bug triaging process.

In this case we will use The Eclipse and Mozilla Defect Tracking Dataset. We will focus on analyzing the Eclipse subset. The task is to explore the dataset to develop insights into the factors that affect the likelihood of a bug being fixed. The goal is then to develop a prediction model for which bugs would be fixed.

Problem Description

Our main objective is to develop prediction model using machine learning techniques including logistic regression, logistic regression with LASSO regularization, random forest, bagging, and boosting.
Given the dataset's complexity, the following steps will be implemented to complete the task:

- Exploratory Data Analysis: Understanding the dataset and identifying key factors influencing bug resolution.
- Feature Engineering: Creating relevant variables that could impact the likelihood of bug resolution.
- Model Development: Building and evaluating models, tuning the parameters.
- Model Predictions: Applying the developed models to a hold-out dataset to predict bug fixes.

Data

The Eclipse dataset provided for this analysis includes twelve CSV files: the reports file, that includes the two report attributes are unchangeable: the reporter and the opening time of the reported bug, and files with different attributes, each with a list of updates and changes that have been performed during the bug's lifetime.

We used Python to clean, filter and sort the original CSV files. Using MariaDB MySQL Database, we joined them into one table by primary keys 'ID', 'Who', 'When'. The resulting dataset contains 143,553 observations.

Data

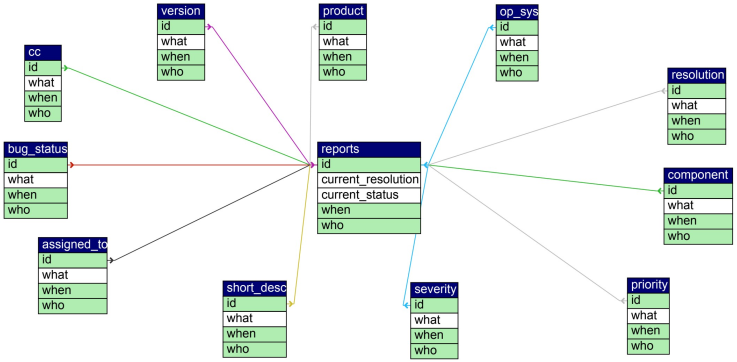

Figure: ER Diagram

Data

Our next step was to separate the non-update features (severity, priority, component, `op_sys`, who, when, `current_status`, `current_resolution`, version, product, `report_id`) into a dedicated database. Each category within these features was transformed into dummy variables. Subsequently, we split and indexed the resulting table into Train, Test, and Validation datasets.

We then created another database to calculate and generate additional variables, which included attributes such as reports, resolution, `bug_status`, `assigned_to`, and `cc_email`. These additional variables were merged with the Train, Test, and Validation datasets. Finally, the comprehensive dataset was downloaded as a .dat file for model implementation.

Data

During development we implemented MySQL code in R using RMySQL and DBI libraries, that connected to the database, pulling the data into the R environment while excluding one dummy for each category to prevent multicollinearity and removed features containing less than 3 successes. (resembling a data pipeline). This was done to facilitate testing our models with different combinations of features. In the final stages we switched to .dat files to reduce computational complexity and simplify the creation of the makefile. All of the databases had a backup to prevent any unexpected event.

Logistic Regression

Logistic regression is a statistical method used when the dependent variable is binary. In our case, the dependent variable is whether a bug will be fixed (1) or not (0).
The logistic regression model can be expressed as:

$$logit\,(p) = \ln\left(\frac{p}{(1-p)}\right) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_n X_n$$

where:

- p is the probability that the bug will be fixed
- β_0 is the intercept term
- β_1, β_2, ... β_n are coefficients for the predictor variables X_1, X_2, ... X_n

Logistic Regression with LASSO regularization

When combined with Lasso regularization, logistic regression becomes a powerful tool for feature selection and preventing overfitting. Lasso (Least Absolute Shrinkage and Selection Operator) regularization is a technique that adds a penalty equal to the absolute value of the magnitude of coefficients to the loss function. The Lasso regularization term is added to the logistic regression cost function to control the complexity of the model.
The regularized logistic regression model can be expressed as:

$$logit\,(p) = \ln\left(\frac{p}{(1-p)}\right) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_n X_n + \lambda \sum |\beta_i|$$

where:

- λ is the regularization parameter that controls the amount of shrinkage applied to the coefficients
- $\sum |\beta_i|$ is the sum of the absolute values of the coefficients

Logistic Regression with LASSO regularization

To predict the likelihood of bug resolution, we constructed a LASSO model using the 'glmnet' package. This method allowed us to automatically select the most relevant features by determining the optimal lambda value through cross-validation, effectively reducing model complexity and preventing overfitting. By penalizing less significant predictors, we achieved a more robust and interpretable model, enhancing the accuracy of our predictions for bug resolution likelihood.

The variables selected by LASSO were then subjected to logistic regression with LASSO regularization, specifically trained to predict the `current_resolution_FIXED` status. Predictions were generated for the validation set, and the model's performance was evaluated using ROC analysis.

Logistic Regression with LASSO regularization

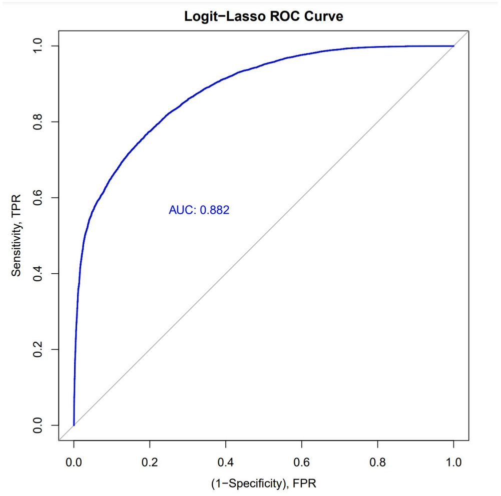

Figure: Logit-LASSO ROC curve

Variables

Our dataset comprised 143,553 observations with 185 features. Initially, we ran our Logit model with 166 features, excluding 19 features due to multicollinearity and insufficient success counts. After applying the Lasso-Logit model, an additional 7 features were excluded by the LASSO regression. Consequently, all subsequent models utilized 159 features, including 150 Bernoulli random variables and 6 continuous random variables.

Variables

The Continuous Random Variables:

1. Our first variable measures the Average Success Rate of the aggregated Bug Assignees for each unique bug report id.
2. Our second variable measures the Average Success Rate of each Bug Reporter for all bug reports.
3. Our third variable measures the Reputation of each Bug Reporter, calculated by multiplying the The Success Rate of each Bug Reporter by the Log of Total Bug Reports for the corresponding Bug Reporter.
4. Our fourth variable measures the proportion of edits for each unique existing bug reporter for each bug report id multiplied by the bug reporters reputation.
5. Our fifth variable measures the length of time the bug report was open.

Decision Tree

A decision tree is a supervised learning algorithm used for classification and regression tasks. It splits the dataset into subsets based on the most significant feature at each node, forming a tree-like structure of decisions. Internal nodes represent tests on attributes, branches represent outcomes, and leaf nodes represent class labels or continuous values.

We constructed a decision tree model to predict the likelihood of a bug being fixed using the rpart package. The model was trained on the training dataset without pruning (cp = 0). Predictions were made in parallel for the validation set using the foreach package to improve computational efficiency. The predicted probabilities for the positive class were then evaluated using ROC analysis to assess the model's performance.

Decision Tree

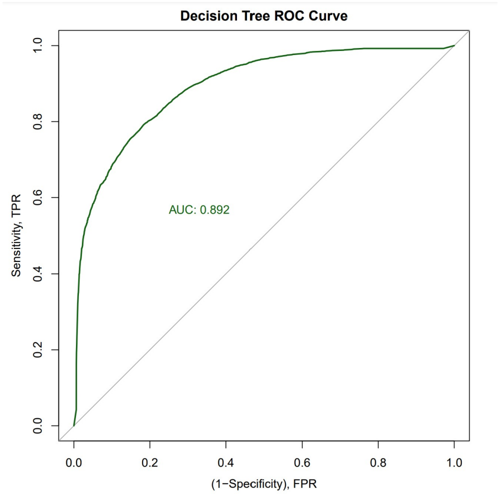

Figure: Decision Tree ROC curve

Random Forest

Random Forest is an ensemble learning method that improves predictive performance by combining multiple decision trees. It is particularly effective in predicting whether a software bug will be fixed, based on various characteristics of the bug report. Each tree in the forest is trained on a bootstrap sample of the data with random feature selection at each node. This method reduces variance, mitigates overfitting, and provides feature importance insights, improving prediction reliability.

We developed a custom random forest model to predict bug resolution likelihood. The model was built using 1000 decision trees, each trained on bootstrap samples of the training data with a random selection of 10 features per tree. Predictions for the validation set were aggregated through majority voting from all the individual trees.

Random Forest

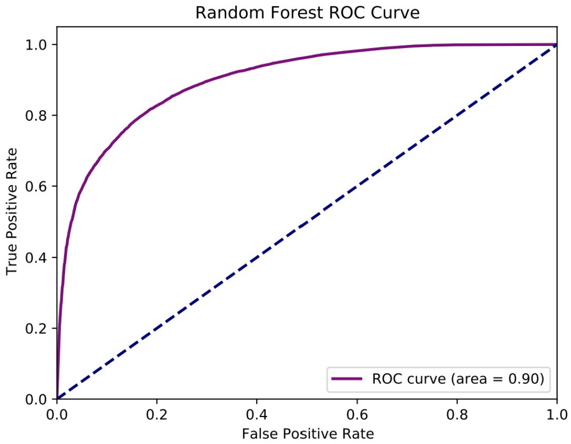

Figure: Decision Tree ROC curve

Bagging

Bagging, or Bootstrap Aggregating, is an ensemble learning technique that enhances the accuracy and stability of machine learning models by reducing variance. It involves creating multiple decision trees from different bootstrap samples of the training data and combining their predictions. This method improves model robustness to outliers and noise. By leveraging multiple models, bagging produces more accurate and reliable predictions.

A bagging model was constructed using the treebag method from the 'caret' package. The model training involved 5-fold cross-validation, allowing for parallel processing to enhance performance. After training, predictions for the validation set were generated, focusing on the probability of the positive class.

Bagging

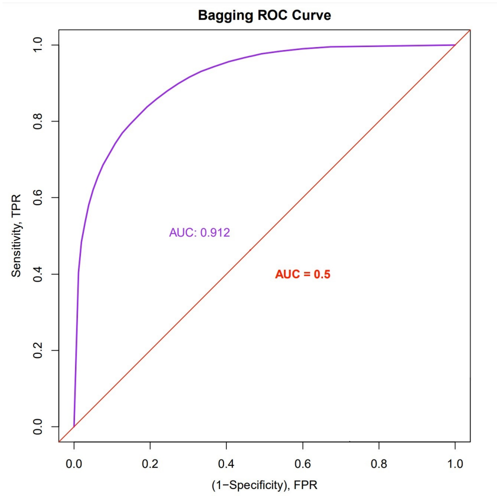

Figure: Bagging ROC curve

Boosting

Boosting is an ensemble learning technique that improves the accuracy and robustness of predictive models by combining several base models sequentially, each correcting the errors of its predecessor. In the context of software bug reports, boosting can predict whether a bug will be fixed by focusing on misclassified instances and iteratively training new models.

We implemented a boosting model using the gbm package. Boosting involves training multiple weak learners sequentially, each trying to correct the errors of its predecessor. We set specific parameters for the boosting process, including the number of trees (n.trees = 10000), interaction depth (interaction.depth = 3), shrinkage rate (shrinkage = 0.01), and subsampling rate (bag.fraction = 0.5).

Boosting

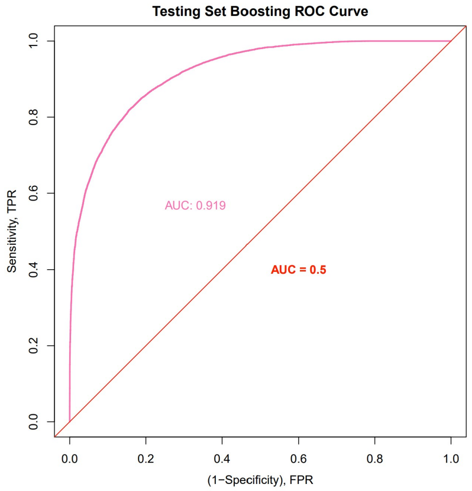

Figure: Boosting ROC curve

Results

The ROC curve analysis compares the performance of four predictive models: Logit-Lasso, Decision Tree, Bagging, and Boosting, in predicting bug resolution. Boosting outperformed all models with the highest AUC of 0.915, indicating its superior predictive power by iteratively refining predictions. Bagging followed with an AUC of 0.898, enhancing model stability through averaging multiple decision trees. The Decision Tree model achieved an AUC of 0.893, performing well but slightly less robust than the ensemble methods. Logit-Lasso had the lowest AUC of 0.881, providing solid but less effective predictions due to its linear nature.

Results

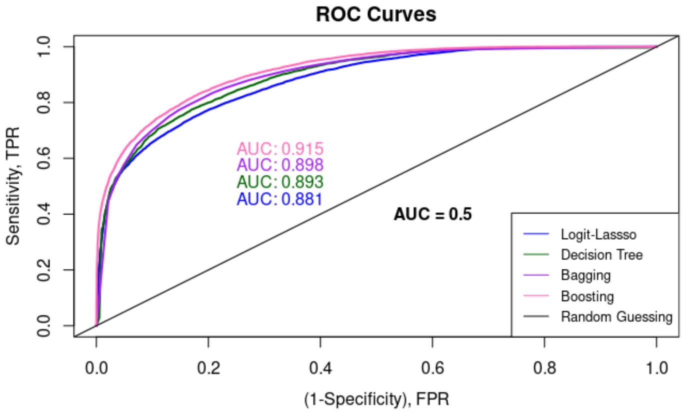

Figure: ROC curves for Logit-LASSO, Decision Tree, Bagging, and Boosting