AI into Software Testing

Pre-Interview exercises

Dinithi Pallawala

Contents

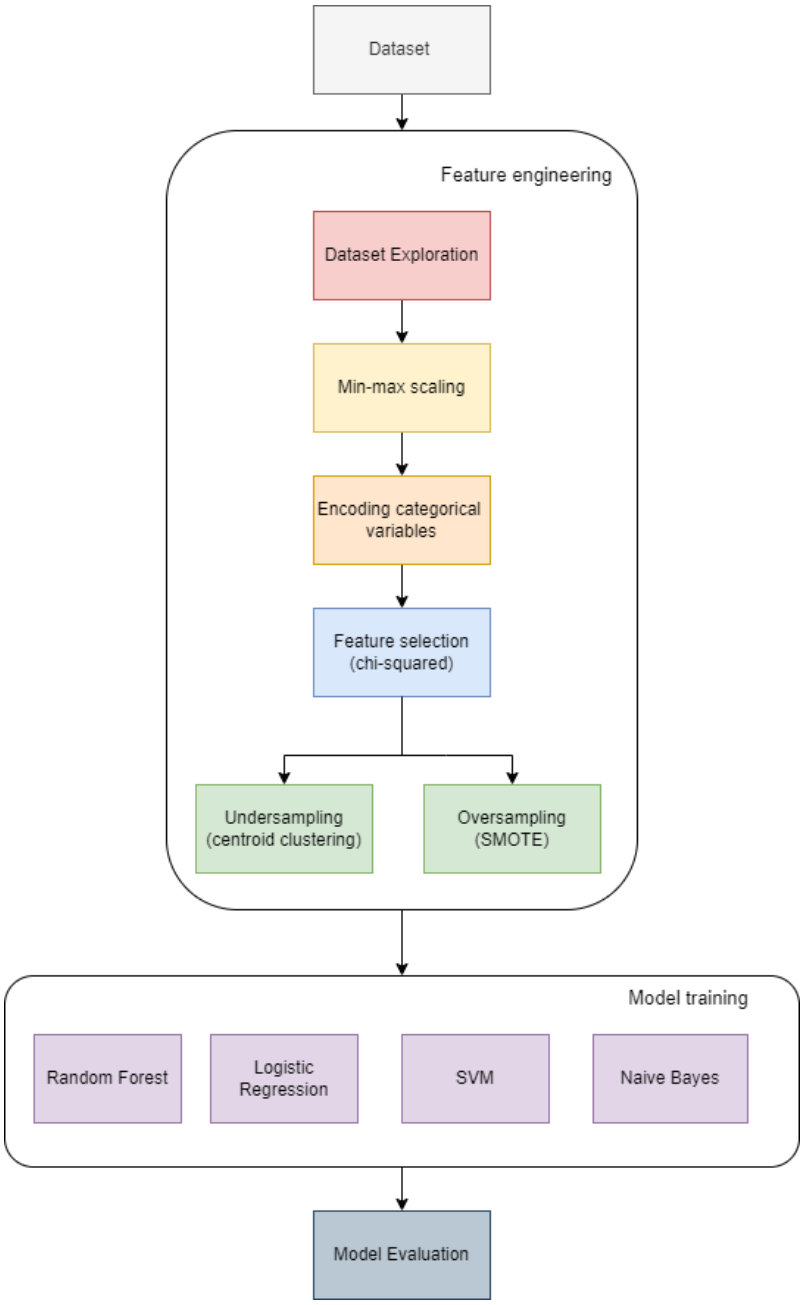
Problem 1: Defect Prediction

A typical dataset that is used for software defect prediction may consist of following attributes:

- Lines of Code Changed
- Branch count
- Time of Code Change
- Developer Experience
- Code Complexity Metrics
- Historical Defect Data

To address the task of predicting error-prone or risky areas in a software, I propose a machine learning approach which involves harnessing the power of predictive models to anticipate potential defects in the codebase as it undergoes changes and inspect how much each feature contributes to the model's decision-making process to identify the error-prone areas. The proposed steps to execute this approach are as follows:

Proposed Approach



1. Data Understanding and Exploration

This initial phase involves a meticulous exploration of the data, gaining insights into its structure and a comprehensive understanding of the dataset.

- First import the dataset into the chosen data analysis environment, be it Python with Pandas allowing to access the raw data for exploration.

- Conduct a preliminary overview of the dataset to grasp its size and dimensions like number of rows and columns. Identify all the features present in the dataset which includes potential predictors and target variables.

- Identify the data types of each feature. Thereby, distinguishing between numerical and categorical variables.

- Generate descriptive statistics for numerical features, including measures like mean, median, standard deviation, and quartiles which helps in understanding the central tendency and spread of the data.

- Use data visualization strategies to identify trends, patterns, and possible outliers. Scatter plots, box plots and histograms aid to understand the distribution of each attribute and the connections between them. In some instances, correlation analysis also can be used to find patterns in the numerical features.

- Exploring the target variable's distribution is crucial for understanding the balance between classes and potential class imbalances.

2. Data Preprocessing

After getting a clear understanding of data, we transition to the phase of data preprocessing; that is to refine the raw dataset for optimal analysis and modeling. This phase involves systematic steps to ensure the dataset is primed for robust machine learning applications.

- The dataset is examined meticulously to check for any missing values and strategies are employed for imputation or removal if any missing values are identified. Also, identification and treatment of outliers is done to maintain statistical robustness.

- Standardizing or normalizing numerical features is performed to bring them to a uniform scale using min-max scaling method.

- Conversion of categorical variables into a format suitable for analysis, employing methods such as one-hot encoding or label encoding based on the nature of the data and modeling requirements.

- If the dataset includes text data like test description, transformation into a numerical format is carried out using techniques like TF-IDF or word embeddings.

- Perform feature extraction to extract new features from the existing ones to reduce dimensionality while retaining valuable information. For this task, techniques like Principal Component Analysis (PCA) or t-distributed Stochastic Neighbor Embedding (t-SNE) can be applied.

- If class imbalances are observed in the distribution of target variable, especially if predicting a binary outcome, it should be addressed. Techniques like oversampling (eg: SMOTE) and under sampling (eg: centroid clustering) methods can be applied to ensure the model is not biased towards the majority class.

- Next, dividing the dataset into training and testing sets is a pivotal step in assessing the model's generalization capacity and mitigating the risk of overfitting.

3. Feature Selection

Feature selection makes use of a subset of the statistically significant features that contribute meaningfully to the prediction of defects to train the model rather than training it with irrelevant characteristics that would impair its performance.

- Various techniques can be employed to strategically select features that contribute significantly to the identification of error-prone areas.

    - Filter methods utilize statistical measures like correlation, information gain, or chi-squared tests to assess the individual relevance of each code metric, leading to the selection of impactful features.

    - Wrapper methods involve iterative model training and evaluation, optimizing for predictive accuracy by selecting different subsets of code metrics (eg: Recursive Feature Elimination (RFE) and forward/backward stepwise selection).

    - Embedded methods seamlessly integrate feature selection into the model training process, with techniques such as LASSO or tree-based methods automatically determining feature importance.

- Common approaches encompass univariate selection, where code metrics with individual impact, such as code complexity and historical defect density, are identified. Model-based selection employs machine learning algorithms like Random Forests to rank features based on their contribution, while correlation analysis explores interrelationships between code metrics to address multicollinearity.

4. Model Selection

Model selection is a crucial stage in the software defect prediction process that comes after feature engineering. It requires careful consideration of algorithmic applicability as well as a sophisticated grasp of the underlying data characteristics.

The interpretability of the chosen algorithm is a crucial factor, especially in the context of software development where insights into defect-prone areas need to be comprehensible to domain experts and developers. While more complex models might offer superior predictive performance, the trade-off is often increased opacity.

For instance, Random Forest is valued for its ability to handle non-linear relationships, feature interactions, and noise within the dataset. Its ensemble nature contributes to the reduction of overfitting, a common concern in defect prediction tasks where the goal is to generalize patterns beyond the training data.

Support Vector Machines (SVM) and Logistic Regression are also emerging models applied in software defect prediction. It is a common practice to employ a variety of models, during the model development process, evaluate their performance, and perform a comparative analysis before choosing a model.

5. Model Training

Model training phase comes into place after carefully selecting a model. This process involves exposing the algorithm to labeled examples, allowing it to adjust its parameters iteratively through an optimization process.

- In algorithm initialization, the parameters of the algorithm are set to their initial values, defining the starting configuration before exposure to the training data.

- As the next step, hyperparameters, external settings that influence the learning process, are fine-tuned. It aims to find the optimal configuration for the algorithm, balancing model complexity and generalization. Some commonly used approaches that can be leveraged for hyperparameter tuning are grid search, random search, and Bayesian optimization.

- After that, cross-validation which helps to prevent overfitting by evaluating the model on data it hasn't seen during training should be done. K-fold cross-validation, where the dataset is divided into k subsets for training and validation is a common technique used for cross validation.

- An iterative learning process should be carried out where the algorithm goes through multiple cycles (epochs) of adjusting its parameters, evaluating performance, and refining its understanding of the underlying patterns in the data. The number of iterations may vary based on convergence criteria or predefined stopping conditions.

6. Model Evaluation

After the model is trained, it should be evaluated using different evaluation metrics to understand its performance, as well as its strengths and weaknesses. Model evaluation is important to assess the efficacy of a model during initial research phases, and it also plays a role in model monitoring.

- The most popular metrics for measuring classification performance include accuracy, precision, recall, F1 score, confusion matrix, and AUC (area under the ROC curve).

7. Visualization

A deeper understanding of the underlying patterns and possible hazards in code modifications can be obtained by visualizing data in the context of software defect prediction. Through graphical representations of key metrics such as precision, recall, and F1 score, stakeholders can readily gauge the model's ability to correctly identify defects, minimize false positives, and strike a balanced trade-off. These visualizations not only facilitate real-time assessment of model performance but also empower decision-makers to fine-tune strategies for defect mitigation based on a comprehensive understanding of the predictive landscape.

- **ROC Curve and Precision-Recall Curve** - Assess the model's ability to discriminate between positive and negative instances.

  Methodology: Plot the ROC curve, depicting the trade-off between true positive rate and false positive rate, and the precision-recall curve, illustrating the trade-off between precision and recall.

- **Confusion Matrix Heatmap -** Offer a visual representation of the model's prediction outcomes.

  Methodology: Construct a heatmap where color intensity corresponds to the frequency of true positives, true negatives, false positives, and false negatives.

- **Residual Analysis -** Examine the residuals (the differences between predicted and actual values).

  Methodology: Plot residuals to identify patterns or heteroscedasticity, aiding in understanding where the model might struggle.

- **Learning Curve -** Evaluate the model's performance across different training dataset sizes.

  Methodology: Plot the learning curve to observe how model performance evolves as more data is used for training.

Identifying areas that are more likely to be error-prone

Feature importance indicates how much each feature contributes to the model prediction resulting in the discovery of areas that are likely to be risky in the code. Feature importance can be represented using a numeric value that we call the score, where the higher the score value has, the more important it is.

Visualizing feature importance through plots like bar charts or heatmaps accentuates which features wield more significance in predicting defects. This aids in pinpointing error-prone areas within the codebase.

Assumptions:

- ✓ Historical defect data is a reliable indicator of future defects.
- ✓ The dataset is sufficiently large for training a machine learning model.

Problem 2: Smart Test Selector

• Imagine a scenario where you have a continuous integration pipeline, and you want to implement a smart test selection mechanism based on code changes

• Develop a strategy for selecting an appropriate set of tests to run, considering efficiency and coverage.

Provide details on the criteria used for test selection, the technology stack or tools involved, and any challenges you anticipate.

Strategy for AI-based Approach

I've developed a robust solution for smart test selection based on code changes. With this model, we can analyze a particular code change to find all potentially impacted tests that transitively depend on modified files, and then estimate the probability of that test detecting a regression introduced by the change. Based on those estimates, the system selects the tests that are most likely to fail for a particular change.

- Test Selection Criteria

1. Code Change and Impact

    - Analyze code changes to identify all potentially impacted tests.
    - The number of changes made to modified files identifies active areas of development that are more prone to regressions.
    - The number of files and lines modified in the change measure its size; large changes are more likely to introduce breakages.

    - Reason

      By understanding the changes made to the codebase, the model can identify tests that depend on the modified files. This ensures that tests covering areas affected by the code changes are considered.

2. History of recent test runs

    - Analyze the historical performance of recent test runs.

    - Reason

      Consider the execution history of tests to understand their past behavior, execution times, and reliability. Historical performance data provides insights into the stability and efficiency of tests.

3. Number of contributors

    - Analyze the number of contributors recently touching modified files.

- Reason

  The number of contributors recently touching modified files is chosen as a criterion to gauge the ownership and familiarity with specific changes. A higher number of contributors may indicate lower ownership and potentially less familiarity with the modified code.

4. Evolution of the code base

   - Analyze the historical relationships between specific portions of code and associated tests.

- Reason

  The analysis of the historical relationships between specific portions of code and associated tests provides valuable insights into how frequently code has been modified in conjunction with specific tests over time.

5. Topology of build dependencies

   - Analyze the topology of build dependencies to understand the relationships between modified code and tests.

- Reason

  Build Dependency Analysis is crucial for understanding the structural relationships between the modified code and various tests in the system. It reveals how changes in the codebase might impact other components and functionalities.

➢ Technology Stack and Tools

1. Machine learning framework

Model: Gradient-Boosted Decision Tree Model

- Gradient-Boosted Decision Trees are known for their inherent explainability. The model produces a series of decision trees, and the decision-making process for each prediction can be easily understood and interpreted. In the context of smart test selection, explainability is crucial. Knowing how the model arrives at its predictions helps build trust in the decisions made regarding test selection for a given code change.

- Gradient-Boosted Decision Trees are relatively easy to train compared to more complex models. Training involves sequentially adding trees, and the model tends to perform well even with default hyperparameters. Given that the model needs to adapt quickly to new code changes, ease of training is valuable. The model can efficiently learn from historical data and be updated to accommodate variations in code changes over time.

- Decision trees, including gradient-boosted ones, are often part of standard machine learning frameworks and libraries. They are well-integrated into existing machine learning infrastructures.

- Relevance to the Model: Seamless integration into the infrastructure is essential for incorporating the model into the continuous integration pipeline.

2. CI Platform Integration

Tool: Jenkins

Jenkins a well-rounded and reliable choice for implementing a smart test selection Mechanism in a continuous integration pipeline focused on code changes, efficiency, and coverage.

Justification

- Jenkins boasts a vast and active plugin ecosystem, providing pre-built integrations for a wide range of tools and functionalities.
- Jenkins has a large and active community of developers, administrators, and users.
- Jenkins allows for flexible and customizable workflows through the use of pipelines.
- Jenkins provides a cost-effective solution.
- Jenkins is designed to automate parts of the software development process, making it well-suited for scenarios where smart test selection needs to be seamlessly integrated and automated within the CI pipeline.
- Jenkins has robust API support, facilitating streamlined communication and interaction with external tools and processes.

➢ Challenges in Implementing a Smart Test Selection Mechanism

1. Model Generalization

Challenge: Ensuring that the model generalizes well to diverse code changes and scenarios is crucial. Overfitting to the training data may result in poor performance on new, unseen code changes.

Mitigation: Implement robust cross-validation techniques during model training, consider augmenting the training dataset with diverse examples, and continuously monitor the model's performance on new code changes.

2. Difficulty in Feature Engineering

Challenge: Identifying and selecting relevant features from code changes and test results is a non-trivial task. The effectiveness of the model heavily depends on the choice of features.

Mitigation: Iteratively refine feature sets based on model performance and explore automated feature selection techniques.

3. Code Abstraction Accuracy

Challenge: The abstraction of new code changes for comparison with historical data may not accurately capture the subtleties of the changes, leading to inaccurate predictions.

Mitigation: Invest in refining the code abstraction process, incorporate feedback loops for continuous improvement, and consider incorporating natural language processing (NLP) techniques for more nuanced abstraction.

4. Maintaining Model Accuracy

Challenge: Ensuring that the model maintains high accuracy over time, especially as the codebase evolves, requires strategies for continuous training and adaptation.

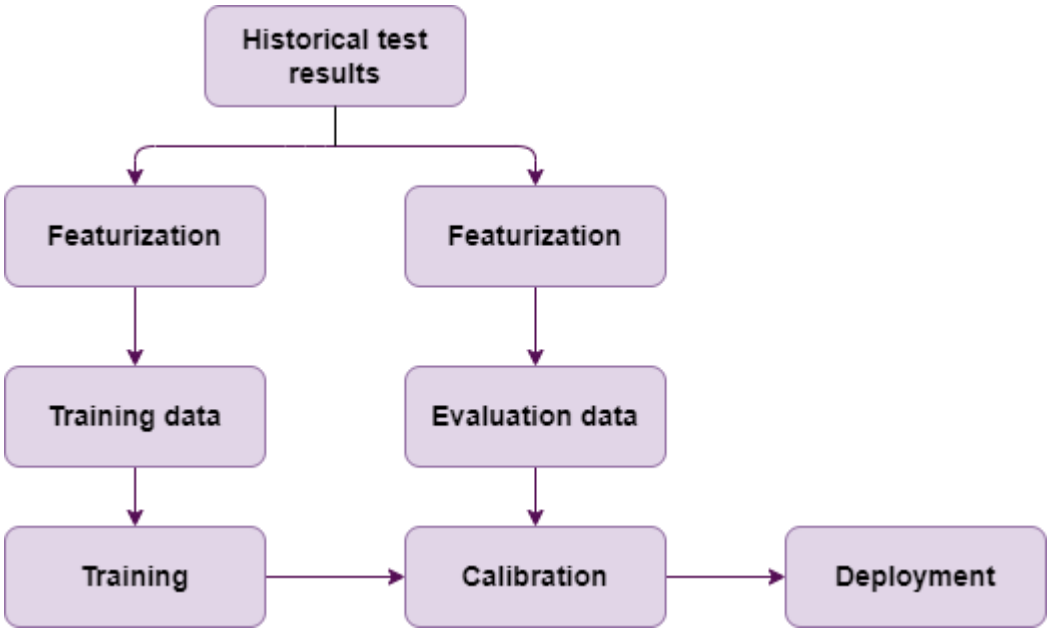
Mitigation: Implement a robust model monitoring system, retrain the model periodically with fresh data, and consider online learning approaches to adapt the model to changing conditions.

5. Integration with CI Pipeline

Challenge: Integrating the predictive test selection model seamlessly into the continuous integration pipeline may face technical and operational challenges.

Mitigation: Work closely with CI/CD platform administrators, leverage existing Integrations and APIs, and conduct thorough testing to ensure the reliability and efficiency of the integrated solution.

- Predictive Test Selection Workflow



- Evaluation of model

  - Accuracy - Measure the overall accuracy of the model in predicting whether a test will pass or fail.
  - Precision and Recall - Precision measures the accuracy of positive predictions, while recall assesses the ability of the model to capture all relevant instances. These metrics are particularly important in the context of identifying failing tests.
  - F1 Score - The F1 score is the harmonic mean of precision and recall, providing a balanced measure of a model's performance.
  - Confusion Matrix - Examine the confusion matrix to understand the distribution of true positives, true negatives, false positives, and false negatives.
  - Validation Set Testing - Assess the model on a separate validation set consisting of recent code changes to ensure its generalization to new data.

Sample Project (Software Detect Prediction)

I carried out a simple project on software defect prediction using a publicly available dataset.

Dataset

This is a dataset which is used in software defect prediction. This dataset has 6052 learning instances which contain 3026 defective instances and 3026 non-defective instances, and 21 static metrics are calculated for the total 6052 instances in dataset which is the data used for the baseline approaches. Table 1 contains the features of the dataset.

Feature	Description
CBO	Coupling between objects. Counts the number of dependencies a class has.
WMC	Weight Method Class or McCabe's complexity. It counts the number of branch instructions in a class.
DIT	Depth Inheritance Tree. It counts the number of "fathers" a class has. All classes have DIT at least 1 (everyone inherits java.lang.Object).
rfc	Response for a Class. Counts the number of unique method invocations in a class.
lcom	Lack of Cohesion of Methods. Calculates LCOM metric.
totalMethods	Counts the number of methods.
totalFields	Counts the number of fields.
NOSI	Number of static invocations. Counts the number of invocations to

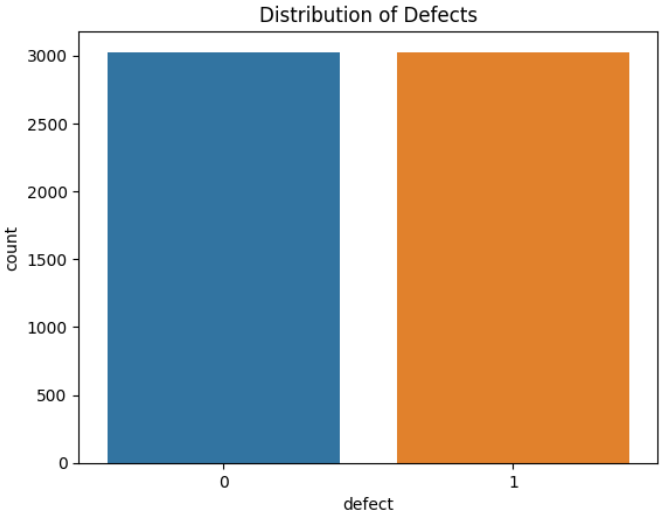
	static methods.
LOC	Lines of code. It counts the lines of count, ignoring empty lines.
returnQty	Quantity of returns. The number of return instructions.
loopQty	Quantity of loops. The number of loops (i.e., for, while, do while, enhanced for).
comparisonsQty	Quantity of comparisons. The number of comparisons (i.e., == and !=).
tryCatchQty	Quantity of try/catches. The number of try/catches.
parenthesizedExpsQty	Quantity of parenthesized expressions. The number of expressions inside parenthesis.
stringLiteralsQty	String literals. The number of string literals (e.g., "John Doe").
numbersQty	Quantity of Number. The number of numbers (i.e., int, long, double, float) literals.
assignmentsQty	Quantity of Variables. Number of declared variables.
mathOperationsQty	Quantity of Math Operations: The number of math operations (times, divide, remainder, plus, minus, left shit, right shift).
variablesQty	Quantity of Variables. Number of declared variables.
maxNestedBlocks	Max nested blocks. The highest number of blocks nested together.
uniqueWordsQty	Number of unique words. Number of unique words in the source code.

Exploratory Data Analysis

Let us first understand the data.

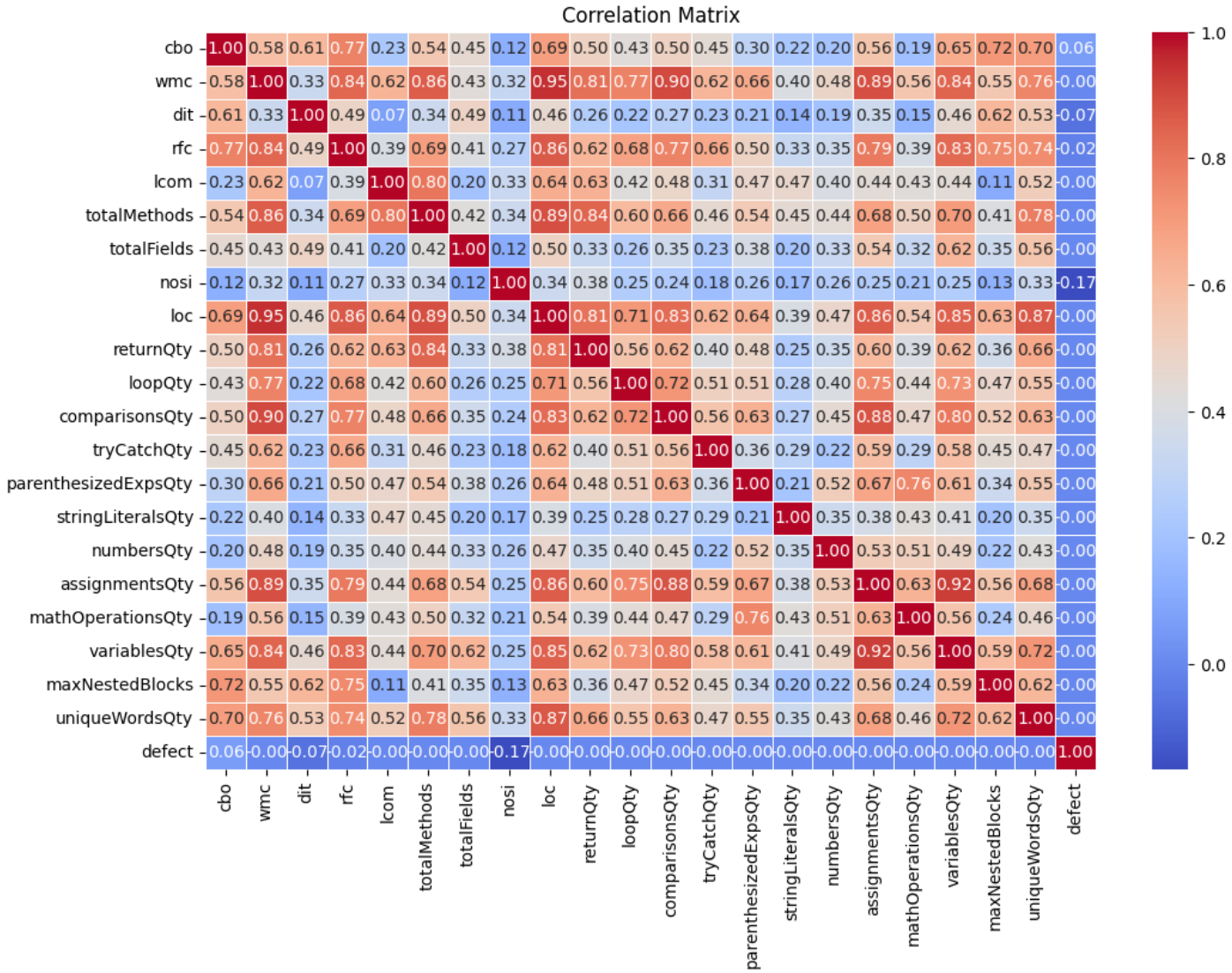
- The original dataset contains object and int64 data types.
- Shape of the data is (6052, 23)
- No missing values found in the dataset.

Distribution of the target variable



- Target variable is equally distributed.

Correlation of features



Data Preprocessing

- Standardizing or normalizing numerical features is performed to bring them to a uniform scale using min-max scaling method.
- There was no categorical data to be encoded and no text data to be transformed into numerical format.
- There was no class imbalance observed.

Feature Selection

In order to select the most appropriate features, a chi-squared test was performed between each non-negative feature and the target class and then select the features above significance level of 0.05.

Model Training

The 4 models Random Forest Classifier, Naïve Bayes Classifier, Support Vector Machines and Logistic Regression were trained, and their performances were compared to select the best model. After thorough examination Random Forest Classifier was chosen as the most appropriate model for this software defect prediction scenario.

Sample code snippet (Random Forest Classifier):

```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

# Separate features (X) and (y)
X = df.drop(columns=['defect'])
y = df['defect']

# Split the dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize the model
rf_classifier = RandomForestClassifier(max_depth=5, min_samples_leaf=4, min_samples_split=10, n_estimators=150)

# Train the model
rf_classifier.fit(X_train, y_train)

# Make predictions on the test set
y_pred = rf_classifier.predict(X_test)

# Evaluate the model performance
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
classification_rep = classification_report(y_test, y_pred)

# Print the results
print(f'Accuracy: {accuracy:.2f}')
print('\nConfusion Matrix:')
print(conf_matrix)
print('\nClassification Report:')
print(classification_rep)
```

Feature Importance

Performed feature importance to identify which features contribute to the defects in the code.

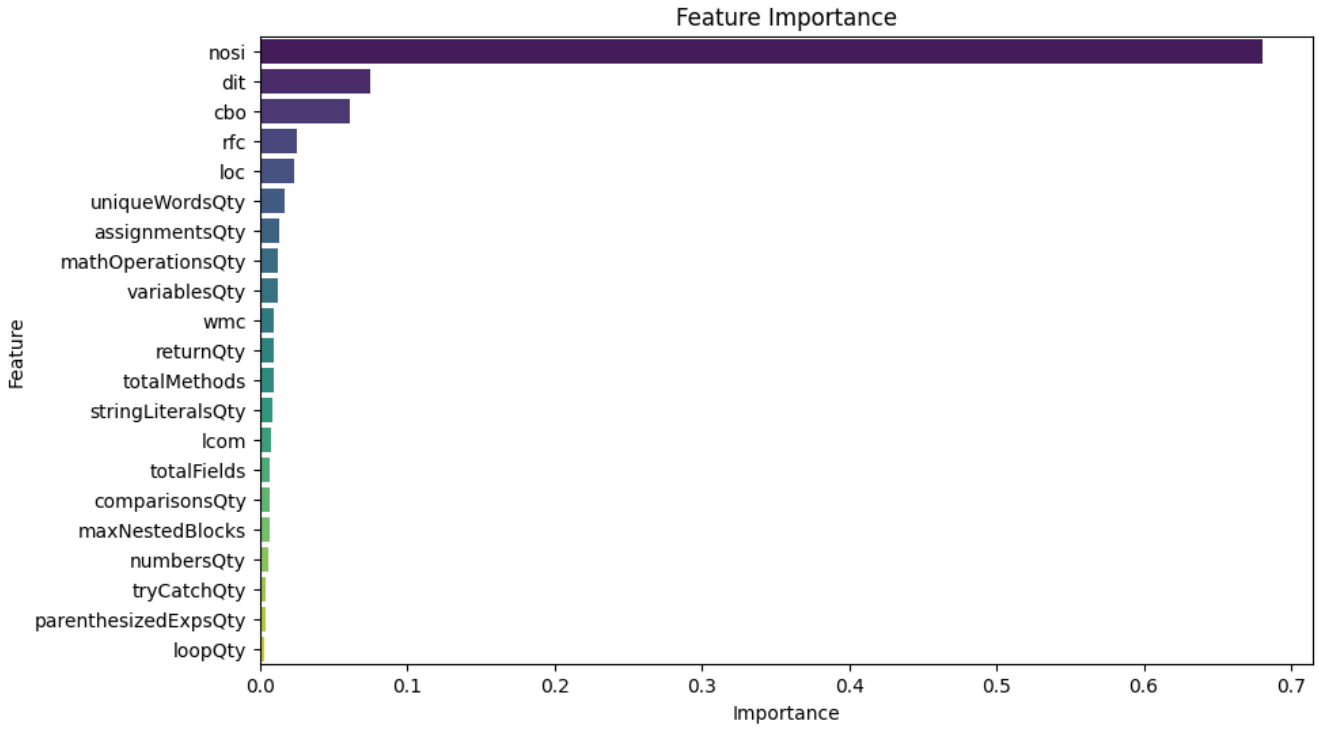
```
# Get feature importances
feature_importances = rf_classifier.feature_importances_

# Create a DataFrame
feature_importance_df = pd.DataFrame({'Feature': X.columns, 'Importance': feature_importances})

# Sort features by importance
feature_importance_df = feature_importance_df.sort_values(by='Importance', ascending=False)

# Plot feature importance
plt.figure(figsize=(10, 6))
sns.barplot(x='Importance', y='Feature', data=feature_importance_df, palette='viridis')
plt.title('Feature Importance')
plt.show()
```

✓ 0.7s

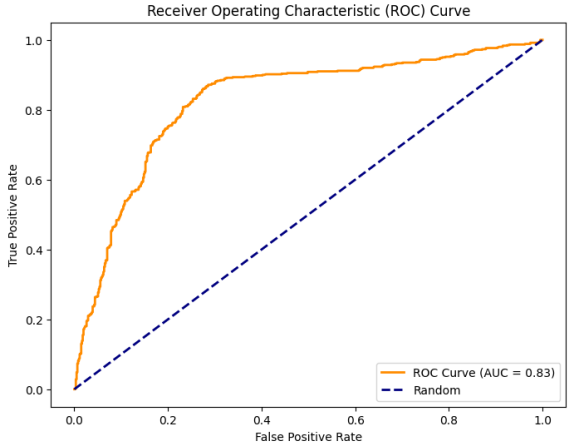


Model Evaluation

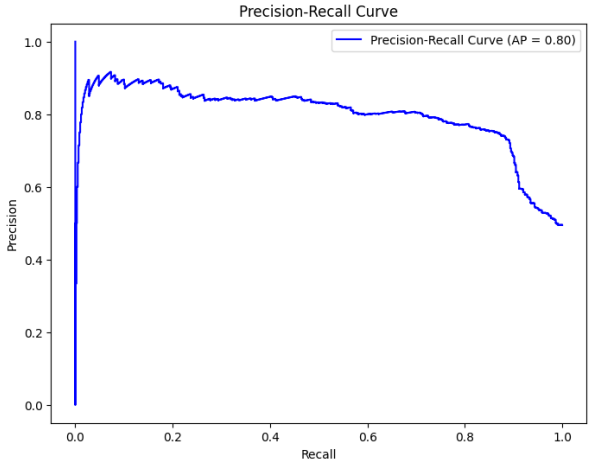
Model	**Accuracy**
Random Forest Classifier	**0.79**
Logistic Regression	**0.73**
Support Vector Machines	**0.70**
Naïve Bayes Classifier	**0.66**

Random Forest Classifier

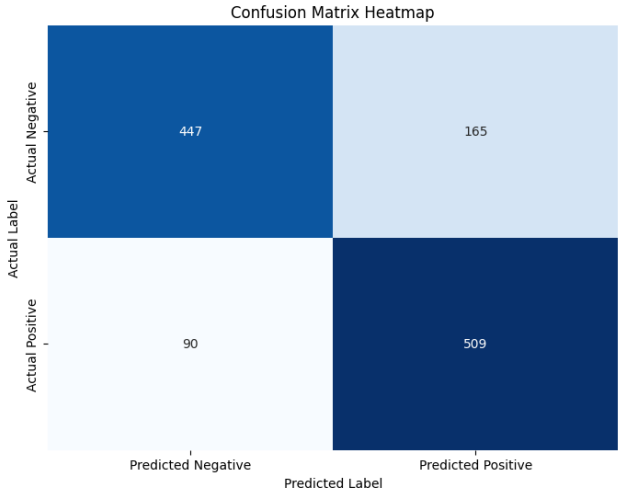
ROC Curve



Precision- Recall Curve

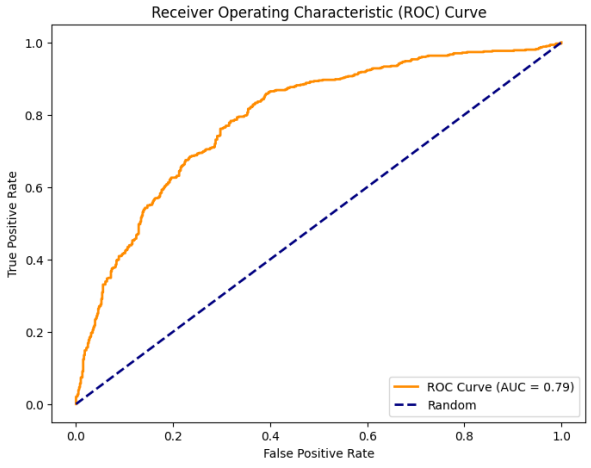


Confusion Matrix Heatmap

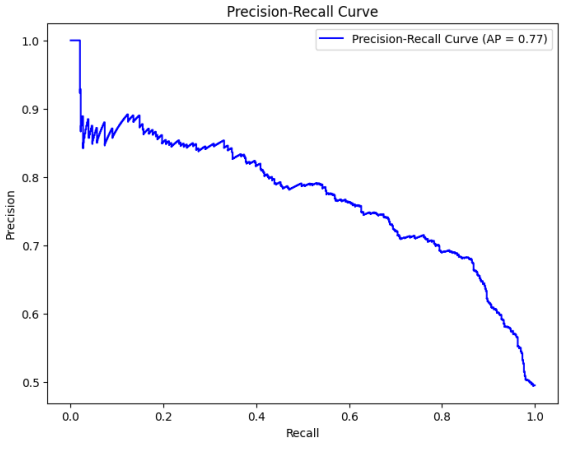


Logistic Regression

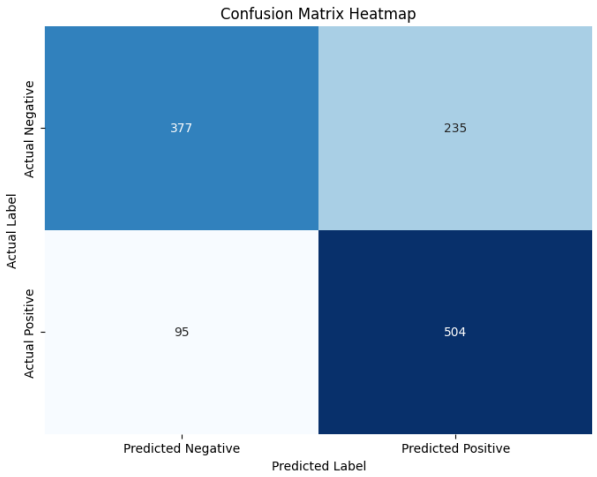
ROC Curve



Precision- Recall Curve

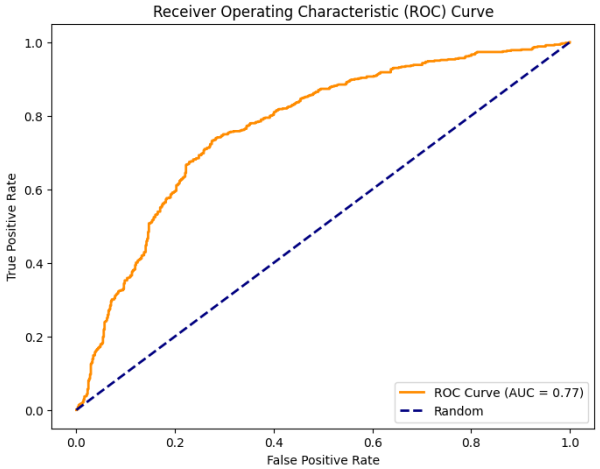


Confusion Matrix Heatmap

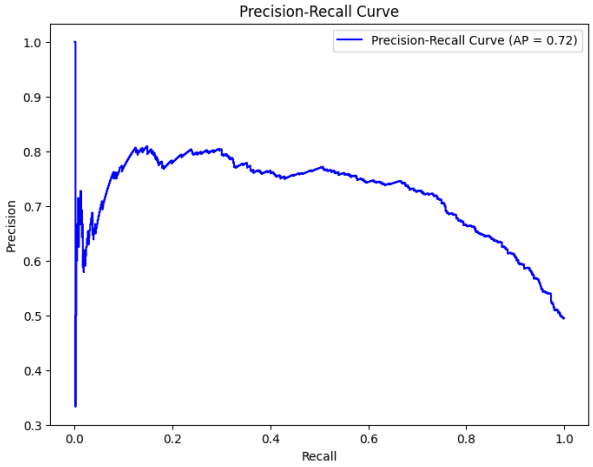


Support Vector Machines

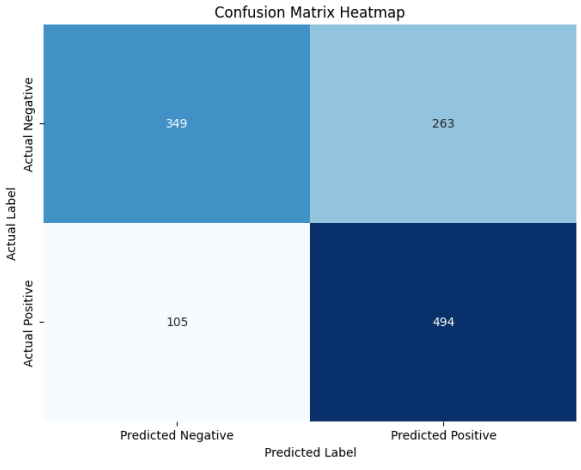
ROC Curve



Precision- Recall Curve



Confusion Matrix Heatmap



References

[1]N. S. Bommi and A. Negi, 'A Standard Baseline for Software Defect Prediction: Using Machine Learning and Explainable AI', *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*, pp. 1798–1803, 2023.

[2]M. Machalica, A. Samylkin, M. Porth, and S. Chandra, 'Predictive Test Selection', *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 91–100, 2018.