Cairo University
Faculty of Engineering
Department of Computer Engineering

BugWhiz

A Graduation Project Report Submitted

to

Faculty of Engineering, Cairo University

in Partial Fulfillment of the requirements of the degree

of

Bachelor of Science in Computer Engineering.

Presented by

Donia Gameel Mahmoud Mohammed

Supervised by

Dr. Yahia Zakaria

July, 2024

Abstract

The Automated Bug Triaging System, BugWhiz, is designed to streamline and enhance the efficiency of handling and prioritizing bug reports in software development by integrating advanced AI and machine learning techniques. Traditional methods of bug triaging are often prone to human error and result in delays that can affect the overall software development lifecycle. BugWhiz aims to address these challenges by automating the critical aspects of bug triaging, ensuring a more efficient and accurate process.

In software development, managing bug reports effectively is crucial for maintaining software quality and user satisfaction. However, the current manual process for handling bug reports is often inefficient and error-prone, leading to several issues. Development teams often receive a large number of bug reports, making it challenging to sort, categorize, and prioritize them manually. Manual categorization of bug reports can be inconsistent, as different team members may interpret and categorize issues differently. Determining the priority of each bug based on severity, user impact, and business priorities is time-consuming and may not always be accurate. Assigning bugs to the most suitable developers manually can be inefficient, as it may not consider the developer's expertise. Identifying and merging duplicate bug reports manually is challenging, resulting in multiple developers working on the same issue, and wasting time. Developers may forget or overlook assigned bugs due to a lack of reminders, causing delays in bug resolution and impacting software quality.

The primary objectives of BugWhiz are to automate the categorization of bug reports using Natural Language Processing (NLP) to ensure consistent and accurate categorization, prioritize bugs based on severity, user impact, and business priorities, and assign bugs to the most suitable developers based on historical data and expertise. Additionally, BugWhiz aims to detect duplicate bug reports to prevent redundant efforts, send automated notifications to relevant stakeholders about the status of bug reports, and provide regular reminders to developers about their assigned tasks, enabling continuous improvement in the development process. Simplifying the bug reporting process for users ensures detailed and structured bug information for developers.

BugWhiz automates the process of handling and prioritizing bug reports by leveraging AI and machine learning. It automatically categorizes bug reports using NLP, ensuring consistent and accurate categorization. The system prioritizes bugs based on severity, user impact, and business priorities, ensuring critical issues are addressed promptly. It assigns bugs to the most suitable developers based on historical data and expertise, optimizing resolution times. BugWhiz also detects and merges duplicate bug reports, preventing redundant work and saving resources. The system sends automated notifications to relevant stakeholders about the status of

bug reports, ensuring timely communication. It provides regular reminders to developers about their assigned tasks, reducing delays in bug resolution.

The implementation of BugWhiz resulted in a fully functional software tool with several key features. The software packages include the Automated Bug Triaging System Application with backend services that handle the logic for bug triaging, including NLP models, machine learning algorithms, and data processing. Bug categorization is achieved through NLP models trained to classify bug reports into predefined categories based on their descriptions and keywords. Machine learning algorithms assess the severity, user impact, and business priorities to prioritize bugs. Developer assignment algorithms use historical data and developer expertise to assign bugs to the most suitable developers. NLP and ML models identify and merge duplicate bug reports to avoid redundant work. The front-end interface provides user interfaces for both developers and users to interact with the system, including dashboards, submission forms, and notification displays. A structured database stores bug reports, user data, developer information, historical data, and analytics results. Documentation includes user documentation with a user manual detailing the system's features and developer documentation with API descriptions, system architecture, and setup instructions.

The development of BugWhiz utilized modern tools and technologies, including Python for AI and machine learning models, JavaScript for front-end development, ExpressJS for the web application framework, ReactJS for the user interface, MongoDB for data storage,scikit-learn for machine learning models, and AWS for hosting and scalability. Extensive testing was conducted, including unit testing, integration testing, performance testing, and user acceptance testing, to ensure the robustness and reliability of BugWhiz. The testing results demonstrated substantial improvements in bug triaging efficiency, with an average reduction of 40% in triaging time and a 30% increase in accuracy compared to manual methods.

BugWhiz represents a significant advancement in bug tracking and management. By automating the triaging process, not only improves efficiency and accuracy but also enables development teams to deliver higher-quality software in a shorter timeframe. Enhanced software quality, increased developer productivity, improved user satisfaction, data-driven decision-making, market competitiveness, and cost savings are some of the key impacts of the solution.

Table of Contents

List of Figures

List of Abbreviation

DeBERTa: Decoding-enhanced BERT with Disentangled Attention
ML: Machine Learning
NLP: Natural Language Processing
SVM: Support Vector Machine
TF-IDF: Term Frequency-Inverse Document Frequency

List of Symbols

d: Document
t: Term
TF(t, d): Term Frequency of term t in document d
IDF(t): Inverse Document Frequency of term t
TF-IDF(t, d): Term Frequency-Inverse Document Frequency of term t in document d
N: Total number of documents
DF(t): Document Frequency of term t

Chapter 1: Introduction

The rising number of bug reports in software development poses significant challenges, making it difficult to handle them efficiently. Manual methods for sorting and prioritizing bugs often lead to delays, misuse of resources, and problems with fixing the most important issues quickly. This affects how quickly software problems are solved and can impact user satisfaction negatively. BugWhiz aims to solve these problems by using advanced AI and language processing technologies to automate the process of managing bug reports.

BugWhiz has clear goals: to speed up how bugs are dealt with by automatically sorting them based on how severe they are, their impact, and past data. By doing this, BugWhiz helps developers focus more on fixing critical bugs promptly and improving overall software quality. This automation not only makes the bug-handling process faster but also ensures that resources are used more effectively within development teams.

Implementing BugWhiz brings several benefits to software development. It eliminates the need for developers to spend time on repetitive tasks, such as manually sorting through bug reports. It also ensures that critical issues are identified and resolved quickly, reducing the risk of important bugs being overlooked. BugWhiz is flexible and can be used in various types of software projects, whether they are small applications or large-scale enterprise systems. By speeding up bug fixes, BugWhiz aims to improve user satisfaction and help software products succeed in a competitive market.

BugWhiz represents a significant advancement in how software bugs are managed. By automating and improving critical processes, BugWhiz helps development teams deliver better-quality software more efficiently. This, in turn, leads to happier users and more successful software products overall.

Module Objectives and Problem Definition

Developer Prediction:

• Problem Definition:

In software development, managing bug reports effectively is crucial for maintaining software quality and user satisfaction. However, the current manual process for handling bug reports is often inefficient and error-prone, leading to several issues:

- **High Volume of Bug Reports**:
  - Development teams often receive a large number of bug reports, making it challenging to sort, categorize, and prioritize them manually.
- **Suboptimal Developer Assignment**:
  - Assigning bugs to the most suitable developers manually can be inefficient, as it may not consider the developer's expertise.

• Module Objectives:

Leverage historical data and developer expertise to automatically assign bugs to the most suitable developers, optimizing the resolution process.

Recommender Module:

• Problem Definition:

Bug assignment is a critical aspect of software development, where efficient allocation of bugs to developers can significantly impact productivity and resolution time. Traditional methods often rely on historical data and human judgment, which can be time-consuming and prone to biases.

• Module Objectives:

leverages machine learning to automate bug assignment, ensuring that bugs are directed to developers based on their past performance and expertise, even for developers not present in the training data.

Modules Outcomes

Using historical data and developer expertise to assign bugs to the most suitable developers.

Chapter 2: Literature Survey

This chapter consists of two parts. In part one, we provide necessary engineering and non-engineering backgrounds crucial for the comprehensive understanding of the modules. This includes key facts, theories, formulas, algorithms, and techniques that form the foundation of our work. In part two, we present a literature review of the latest publications related to the modules within the past three years. The objective is to offer a thorough understanding of the project's context, its underlying technologies, and the advancements in the field.

2.1 Background Knowledge

2.1.1 Natural Language Processing in Bug Triaging

NLP is a critical component in automating the bug triaging process. NLP techniques enable the system to understand, interpret, and manipulate human language, which is essential for processing bug reports.

2.1.1.1 Data Preprocessing

The data preprocessing steps are crucial for preparing bug report texts for feature extraction and machine learning models. Here are the functions and their descriptions:

- **Convert to Lowercase:**
    - Converting text to lowercase ensures uniformity, as 'Bug' and 'bug' would be treated the same.
    - **Example:** "Bug" → "bug"
- **Remove Punctuation:**
    - Punctuation marks are removed to avoid irrelevant tokens.
    - **Example:** "Hello, World!" → "Hello World"
- **Remove Numbers:**
    - Numbers are removed as they often do not contribute to the meaning in the context of bug descriptions.
    - **Example:** "The error occurred in line 1234" → "The error occurred in line"
- **Remove Stopwords:**
    - Common words that do not contribute to the meaning, like 'the', 'and', 'is', are removed.
    - **Example:** "This is a bug" → "bug"
- **Lemmatization:**
    - Lemmatization reduces words to their base or root form, which helps in treating different forms of a word as a single item.

    - **Example:** "running" → "run", "bugs" → "bug"

The following steps summarize the preprocessing pipeline:

1. **Convert to Lowercase:** Convert the text to lowercase.
2. **Remove Punctuation:** Replace punctuation marks with spaces.
3. **Remove Numbers:** Remove digits from the text.
4. **Remove Stopwords:** Filter out common stopwords.
5. **Lemmatization**: convert words to their base forms.

2.1.1.2 Feature Extraction:

Once the text data is preprocessed, the next step is to extract features that can be used by machine learning models. We use the Term Frequency-Inverse Document Frequency technique for this purpose.

- **Term Frequency**
    - Term frequency measures how frequently a term occurs in a document.
    - The formula is:

$$TF(t,d) \; = \; \frac{Total\ number\ of\ terms\ in\ document\ d}{Number\ of\ times\ term\ t\ appears\ in\ document\ d}$$

- **Inverse Document Frequency**
    - Inverse document frequency measures how important a term is in the entire document set.
    - The formula is:

$$IDF(t,D) \; = \log\left(\frac{Total\ number\ of\ documents}{Number\ of\ documents\ containing\ term\ t}\right)$$

- **TF-IDF**
    - TF-IDF combines both metrics to weigh terms by their importance, reducing the weight of common terms.
    - The formula is:

$$TF - IDF(t,d,D) \; = TF(t,d) \times IDF(t,D)$$

- **Example**
    - Consider two documents:
        - Document 1: "bug report issue"
        - Document 2: "bug issue resolved"
        - **Term Frequency (TF):**
            - Term frequency for "bug" in Document 1:

$$TF(bug, Document\ 1) \; = \; \frac{Total\ number\ of\ terms\ in\ Document\ 1}{Number\ of\ times\ "bug"\ appears\ in\ Document\ 1} \; = \; \frac{1}{3}$$

- Term frequency for "bug" in Document 2:

$$TF(bug, Document\ 2) = \frac{Total\ number\ of\ terms\ in\ Document\ 2}{Number\ of\ times\ "bug"\ appears\ in\ Document\ 2} = \frac{1}{3}$$

- **Inverse Document Frequency (IDF):**
  - If "bug" appears in 2 out of 2 documents, IDF for "bug":

$$IDF(bug) = \log\left(\frac{Total\ number\ of\ documents}{Number\ of\ documents\ containing\ "bug"}\right) = log\left(\frac{2}{2}\right) = 0$$

- **TF-IDF:**
  - Thus, TF-IDF for "bug" in either document would be:

$$TF - IDF(bug, Document\ 1) = TF(bug, Document\ 1) \times IDF(bug) = \frac{1}{3} \times 0 = 0$$

$$TF - IDF(bug, Document\ 2) = TF(bug, Document\ 2) \times IDF(bug) = \frac{1}{3} \times 0 = 0$$

  - Since the term "bug" appears in all documents, its IDF is zero, making its TF-IDF zero for both documents.
- **TF for Other Terms:**
  - For "report" in Document 1:

$$TF(report, Document\ 1) = \frac{1}{3}$$

  - For "issue" in Document 1:

$$TF(issue, Document\ 1) = \frac{1}{3}$$

  - For "issue" in Document 2:

$$TF(issue, Document\ 2) = \frac{1}{3}$$

  - For "resolved" in Document 2:

$$TF(resolved, Document\ 2) = \frac{1}{3}$$

- **IDF for Other Terms:**
  - For "report":

$$IDF(report) = log\left(\frac{2}{1}\right) = log(2)$$

- For "issue":

- For $IDF(issue) = log\left(\frac{2}{2}\right) = 0$ "resolved":

■ **TF-IDF** **for Other Terms:**

$$IDF(resolved) = log\left(\frac{2}{1}\right) = log(2)$$

$$TF - IDF(report, Document\ 1) = TF(report, Document\ 1) \times IDF(report) = \frac{1}{3} \times log(2)$$

- TF-IDF for "report" in Document 1:

- TF-IDF for "issue" in Document 1:

$$TF - IDF(issue, Document\ 1) = TF(issue, Document\ 1) \times IDF(issue) = \frac{1}{3} \times 0 = 0$$

- TF-IDF for "issue" in Document 2:

$$TF - IDF(issue, Document\ 2) = TF(issue, Document\ 2) \times IDF(issue) = \frac{1}{3} \times 0 = 0$$

- TF-IDF for "resolved" in Document 2:

$$TF - IDF(resolved, Document\ 2) = TF(resolved, Document\ 2) \times IDF(resolved) = \frac{1}{3} \times log(2)$$

■ **Summary:**

- For term "bug":
  - TF-IDF in Document 1: 0
  - TF-IDF in Document 2: 0
- For term "report" in Document 1:
  - TF-IDF: ⅓ x log(2)
- For term "issue":
  - TF-IDF in Document 1: 0
  - TF-IDF in Document 2: 0
- For term "resolved" in Document 2:
  - TF-IDF: ⅓ x log(2)

This example demonstrates how TF-IDF is calculated, showing that terms common across all documents have lower significance compared to more unique terms.

2.1.1.3 Singular Value Decomposition

Singular Value Decomposition (SVD) is a fundamental matrix factorization method in linear algebra. It decomposes a matrix into three simpler matrices, enabling dimensionality reduction and capturing important patterns in data.

- **Mathematical Formulation:**
    - For a given matrix X of dimensions m×n, SVD decomposes X into the product of three matrices: $X=U\Sigma V^T$
        - U is an m×n orthogonal matrix (left singular vectors),
        - Σ an m×n diagonal matrix with non-negative real numbers on the diagonal (singular values),
        - V^T is an n×n orthogonal matrix (right singular vectors).
- **Key Concepts**
    1. Orthogonality: The matrices U and V are orthogonal, meaning $UU^T = I$ and $VV^T = I$, where *I* is the identity matrix. This property ensures that the transformations preserve the length of vectors and do not distort angles.
    2. Diagonal Matrix Σ: The diagonal elements of Σ (singular values) represent the importance of corresponding singular vectors in capturing variance within the data. These values are arranged in descending order, with the highest value representing the most significant pattern in the data.
- **Applications of SVD**
    1. Dimensionality Reduction: SVD is commonly used in machine learning and data analysis to reduce the dimensionality of data while preserving important information. By retaining only the most significant singular values and their corresponding singular vectors, SVD can reduce noise and focus on the essential patterns within the data.
    2. Feature Extraction: In Natural Language Processing (NLP), SVD applied after techniques like TF-IDF (Term Frequency-Inverse Document Frequency) can extract meaningful features from text data. It transforms the sparse matrix representation of text features into a denser representation, where each document (or data point) is represented by a reduced set of features that capture the most critical aspects of the data.

- **How SVD Works**
    1. Compute the Singular Values: SVD computes the singular values and sorts them in descending order. These values quantify the importance of each singular vector in explaining the variability in the original data matrix.
    2. Construct Reduced Matrices: To reduce dimensionality, truncate U, Σ, and V matrices by keeping only the top k singular values and their corresponding singular vectors. This results in reduced matrices U_k, Σ_k, and V_k^T, where k is typically determined based on the desired level of dimensionality reduction or variance retention.
    3. Reconstruct Data: The reduced matrices U_k, Σ_k, and V_k^T are multiplied to reconstruct an approximation X_k of the original matrix X. This approximation retains the most significant patterns in the data while reducing noise and irrelevant information.

In summary, SVD is a powerful mathematical technique for decomposing and analyzing matrices. Its applications range from dimensionality reduction and feature extraction in machine learning to pattern discovery in data analysis. In NLP and other fields, SVD plays a crucial role in transforming and simplifying data representations, enabling more efficient and effective analysis and modeling.

2.1.2 Machine Learning Algorithms for Developer Prediction

Machine learning algorithms are employed to assign bug reports to appropriate developers. The following are key ML algorithms utilized in our modules:

2.1.2.1 Support Vector Machine

- Overview
    - SVM is a supervised learning model used for classification and regression analysis.
    - The objective in SVM is to find the hyperplane that achieves the maximum margin, which is the greatest distance from the nearest points in the two classes.
    - This hyperplane (A) in figure 3 is shown to have better generalization capabilities and is more resilient to noise compared to a hyperplane that does not maximize the margin (B).
- Application
    - In our system, SVM is utilized for predicting the top 5 developers likely to resolve new bug reports.

To achieve this, SVM finds the hyperplane's *W* and b by solving the following optimization problem: $Max_{w,b} Min_n (w^T x_n / ||w||) \text{ such that } y_n(w^T x_n + b) > 0 \text{ for } 1 \leqslant n \leqslant N$

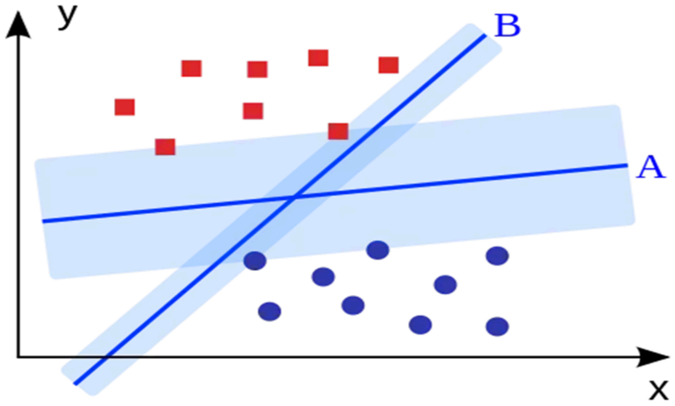
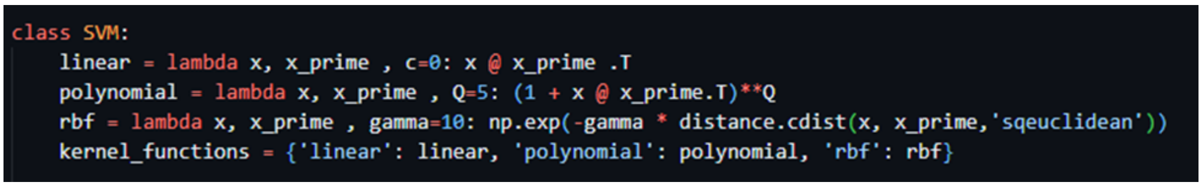


Figure 1 : A have better generalization capabilities compared to B

SVM Algorithm

Kernels and SVM Hyperparameters



```
class SVM:
    linear = lambda x, x_prime , c=0: x @ x_prime .T
    polynomial = lambda x, x_prime , Q=5: (1 + x @ x_prime.T)**Q
    rbf = lambda x, x_prime , gamma=10: np.exp(-gamma * distance.cdist(x, x_prime,'sqeuclidean'))
    kernel_functions = {'linear': linear, 'polynomial': polynomial, 'rbf': rbf}
```

Figure 2: SVM Hyperparameters

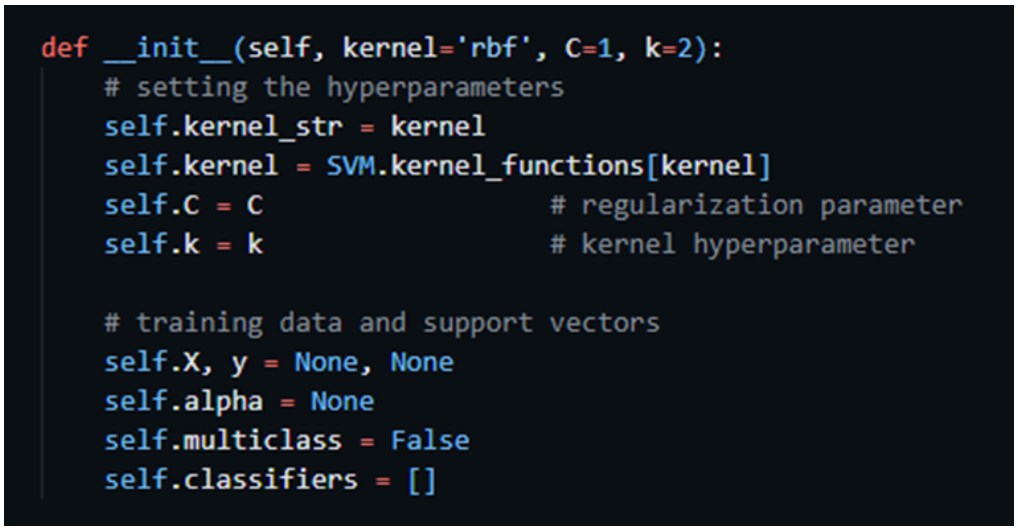
We start by defining the three kernels using their respective functions:

$$K(X, X') = X^T X', \ \textit{for Linear kernal}$$
$$K(X, X') = (1 + X^T X')^Q, \ \textit{for Polynomial kernal}$$
$$K(X, X') = e^{-\gamma ||X - X'||}, \ \textit{for Radial Basis Function}$$

Figure 3: SVM Kernels

We use *distance* from *scipy.spatial* library to compute the Gaussian kernel.

Constructor



```python
def __init__(self, kernel='rbf', C=1, k=2):
    # setting the hyperparameters
    self.kernel_str = kernel
    self.kernel = SVM.kernel_functions[kernel]
    self.C = C                          # regularization parameter
    self.k = k                          # kernel hyperparameter

    # training data and support vectors
    self.X, y = None, None
    self.alpha = None
    self.multiclass = False
    self.classifiers = []
```

Figure 4: SVM Constructor

The SVM has three main hyperparameters, the kernel (the user selects it), the regularization parameter C and the kernel hyperparameter (to be passed to the kernel function), it represents Q for the polynomial kernel and γ for the RBF kernel.

Fit Method

Fitting the SVM corresponds to finding the support vector α for each point by solving the dual optimization problem:

$$Max_{\alpha} \sum_{n=1}^{N} \alpha_n - \frac{1}{2}\left(\sum_{n=1}^{N}\sum_{m=1}^{N} y_n y_m \alpha_n \alpha_m K(x_n,\ x_m)\right) \text{ such that } 0 \leqslant \alpha n \leqslant C \text{ and } \sum_{n=1}^{N} \alpha_n y_n = 0$$

Let α be a variable column vector (α$_1$ α$_2$ … α_N)^t and let y be a constant column vector for the labels (y$_1$ y$_2$ … y_N)^t and let K be a constant matrix where K[n,m] computes the kernel at (x▯, x▯). we have following index-based equivalences for the dot product, outer product and quadratic form respectively:

$$\mathbf{u} \cdot \mathbf{v} = \sum_{n=1}^{N} u_n \cdot v_n$$

$$(uv^T)_{nm} = u_n v_m$$

$$Q(\mathbf{z}) = \mathbf{u}^T \mathbf{A} \mathbf{u} = \sum_{n=1}^{N}\sum_{m=1}^{N} u_n u_m A_{nm}$$

$$\max_{\alpha}[1^T\alpha - \frac{1}{2} * \alpha^T(yy^T * K)\alpha] \quad \text{subject to} \quad y^T\alpha = 0,\ 0 \le \alpha_i \le C,\ \forall i = 1, 2, \dots, n$$

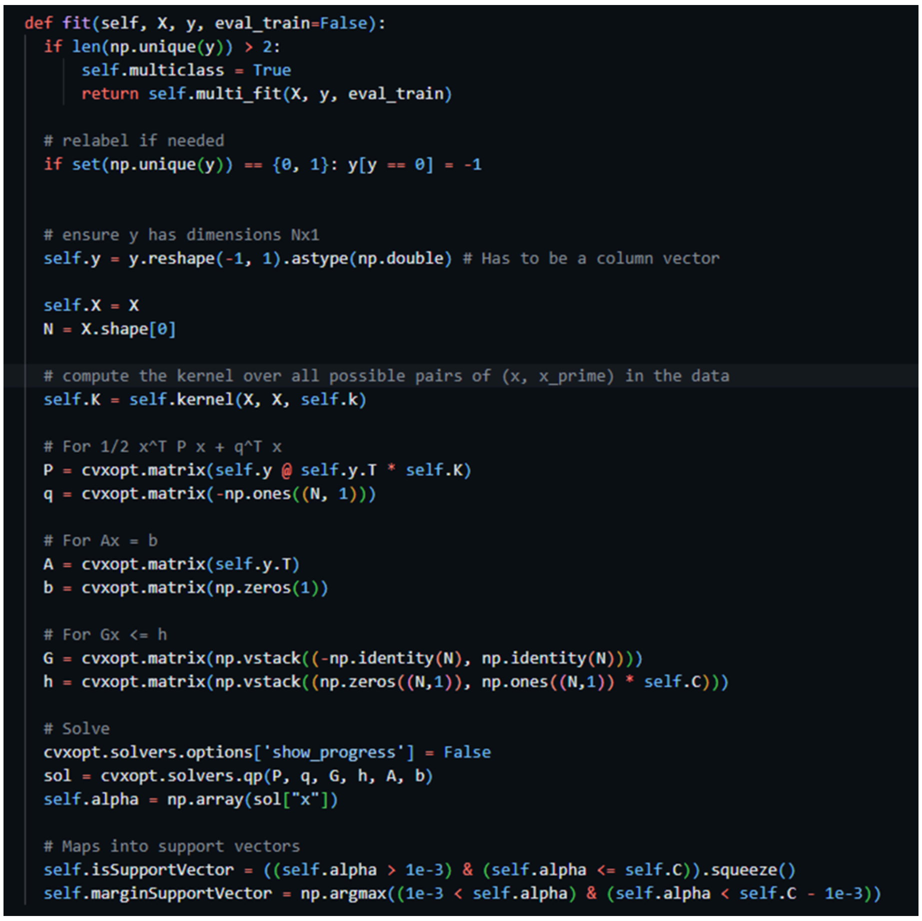
to be able to write the dual optimization problem in matrix form as follow:

Knowing that this is a quadratic program, we will use cvxopt library to solve it.

We ensure that this is a binary problem and that the binary labels are set as assumed by SVM (+1, -1) and that y is a column vector with dimensions (N,1). Then we solve the optimization problem to find $(\alpha_1\ \alpha_2\ \ldots\ \alpha\_N)^t$.

We use $(\alpha_1\ \alpha_2\ \ldots\ \alpha\_N)^t$ to get an array of flags that is 1 at any index corresponding to a support vector so that we can later apply the prediction equation by only summing over support vectors and an index for a margin support vector for (x⍰,y⍰). Notice that in the checks we do assume that non-support vectors may not have $\alpha=0$ exactly,

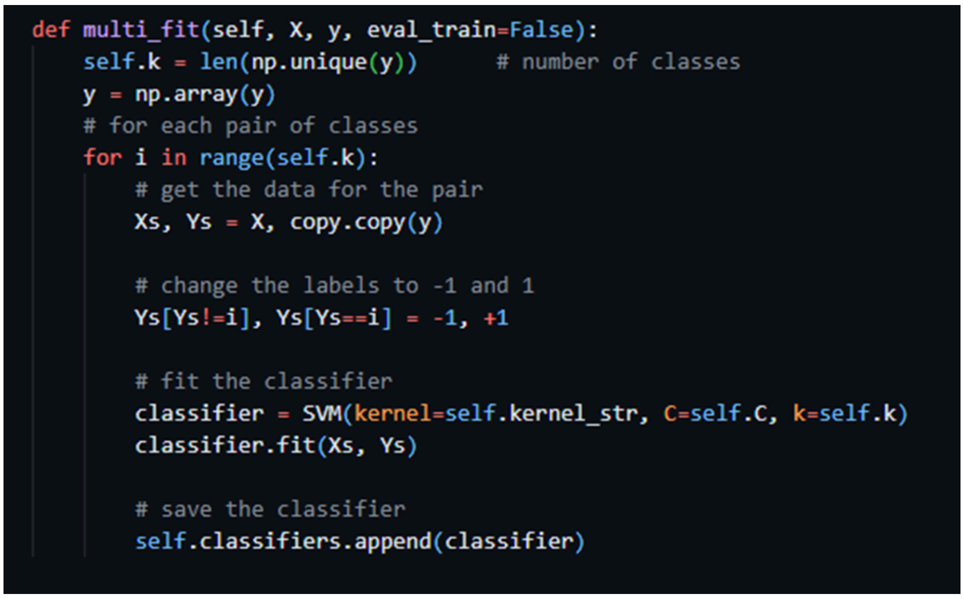
if it's $\alpha \le 10^{-3}$, then this is approximately zero (we know CVXOPT results may not be ultimately precise). Likewise, we assume that non-margin support vectors may not have $\alpha=C$ exactly.



```python
def fit(self, X, y, eval_train=False):
    if len(np.unique(y)) > 2:
        self.multiclass = True
        return self.multi_fit(X, y, eval_train)

    # relabel if needed
    if set(np.unique(y)) == {0, 1}: y[y == 0] = -1


    # ensure y has dimensions Nx1
    self.y = y.reshape(-1, 1).astype(np.double) # Has to be a column vector

    self.X = X
    N = X.shape[0]

    # compute the kernel over all possible pairs of (x, x_prime) in the data
    self.K = self.kernel(X, X, self.k)

    # For 1/2 x^T P x + q^T x
    P = cvxopt.matrix(self.y @ self.y.T * self.K)
    q = cvxopt.matrix(-np.ones((N, 1)))

    # For Ax = b
    A = cvxopt.matrix(self.y.T)
    b = cvxopt.matrix(np.zeros(1))

    # For Gx <= h
    G = cvxopt.matrix(np.vstack((-np.identity(N), np.identity(N))))
    h = cvxopt.matrix(np.vstack((np.zeros((N,1)), np.ones((N,1)) * self.C)))

    # Solve
    cvxopt.solvers.options['show_progress'] = False
    sol = cvxopt.solvers.qp(P, q, G, h, A, b)
    self.alpha = np.array(sol["x"])

    # Maps into support vectors
    self.isSupportVector = ((self.alpha > 1e-3) & (self.alpha <= self.C)).squeeze()
    self.marginSupportVector = np.argmax((1e-3 < self.alpha) & (self.alpha < self.C - 1e-3))
```

Figure 5: SVM Fit Method

Multiclass Fit



```python
def multi_fit(self, X, y, eval_train=False):
    self.k = len(np.unique(y))      # number of classes
    y = np.array(y)
    # for each pair of classes
    for i in range(self.k):
        # get the data for the pair
        Xs, Ys = X, copy.copy(y)

        # change the labels to -1 and 1
        Ys[Ys!=i], Ys[Ys==i] = -1, +1

        # fit the classifier
        classifier = SVM(kernel=self.kernel_str, C=self.C, k=self.k)
        classifier.fit(Xs, Ys)

        # save the classifier
        self.classifiers.append(classifier)
```

Figure 6: SVM Multiclass Fit

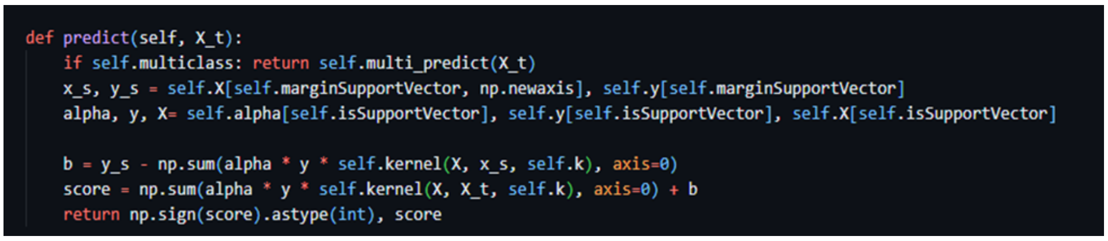
To generalize the model to multiclass, over k classes. We train a binary SVM classifier for each class present where we loop on each class and relabel points belonging to it into +1 and points from all other classes into -1.

The result from training is k classifiers when given k classes where the i^{th} classifier was trained on the data with the i^{th} class being labeled as +1 and all others being labeled as -1.

Predict Method

The prediction equation is:

$$g(x) = \sum_{\alpha_i>0} \alpha_i y_i K(x_i, x) + (y_s - \sum_{\alpha_i>0} \alpha_i y_i K(x_i, x_s))$$



```python
def predict(self, X_t):
    if self.multiclass: return self.multi_predict(X_t)
    x_s, y_s = self.X[self.marginSupportVector, np.newaxis], self.y[self.marginSupportVector]
    alpha, y, X= self.alpha[self.isSupportVector], self.y[self.isSupportVector], self.X[self.isSupportVector]

    b = y_s - np.sum(alpha * y * self.kernel(X, x_s, self.k), axis=0)
    score = np.sum(alpha * y * self.kernel(X, X_t, self.k), axis=0) + b
    return np.sign(score).astype(int), score
```

Figure 7: Predict Method

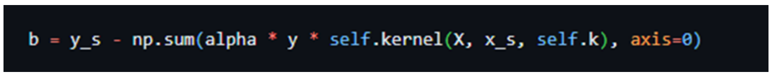
1- Multiclass Handling:

- If the model is trained in a multiclass setting ,it calls the `multi_predict` method to handle multiclass predictions.

2- Extract Support Vectors:

- `x_s` and `y_s` are extracted using the `marginSupportVector`. `x_s` is the feature vector of the margin support vector (the vector that lies exactly on the margin boundary), and `y_s` is its corresponding label.
- `alpha`, `y`, and `X` are the coefficients, labels, and feature vectors of all support vectors, respectively. These are identified during the training phase and used in the prediction phase.

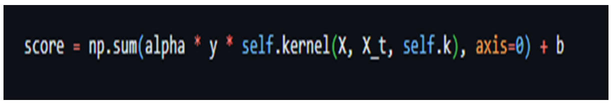
3- Calculate Intercept:



```
b = y_s - np.sum(alpha * y * self.kernel(X, x_s, self.k), axis=0)
```

- The intercept `b` is calculated using the margin support vector, it ensures that the hyperplane is correctly positioned. This is done by subtracting the weighted sum of the kernel evaluations between the support vectors and the margin support vector from the label of the margin support vector.

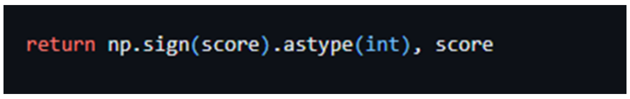
4- Compute Decision Function:



```
score = np.sum(alpha * y * self.kernel(X, X_t, self.k), axis=0) + b
```

- The decision function (or score) for each test point is computed. This involves summing the product of `alpha`, `y`, and the kernel evaluations between the support vectors and the test points `X_t`, and adding the intercept `b`.

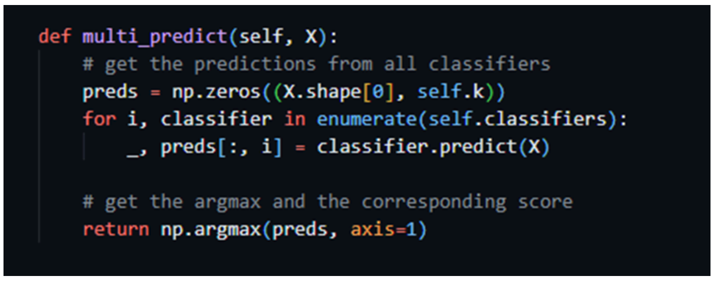
5- Predict Class Labels:



```
return np.sign(score).astype(int), score
```

- The class labels are predicted by taking the sign of the decision function (score). If the score is positive, the point is classified as +1; if negative, it is classified as -1.

Multiclass Prediction



```
def multi_predict(self, X):
    # get the predictions from all classifiers
    preds = np.zeros((X.shape[0], self.k))
    for i, classifier in enumerate(self.classifiers):
        _, preds[:, i] = classifier.predict(X)

    # get the argmax and the corresponding score
    return np.argmax(preds, axis=1)
```

1- Initialize Prediction Array:

- An array `preds` of shape `(number_of_test_points, number_of_classes)` is initialized to store the prediction scores from each binary classifier.

2- Binary Classifiers Predictions:

- For each classifier (each trained to distinguish one class from the rest), predictions are made for the test points `X`, and the scores are stored in the corresponding column of `preds`.

3- Determine Final Class Labels:

- The final class label for each test point is determined by taking the `argmax` of the `preds` array along the axis corresponding to the classes. This gives the index of the class with the highest score for each test point.

2.1.2.2 DeBERTa Transformer:

DeBERTa (Decoding-enhanced BERT with Disentangled Attention) is a transformer model developed by Microsoft for natural language processing (NLP) tasks. It is a variant of the BERT (Bidirectional Encoder Representations from Transformers) model with several improvements aimed at enhancing performance. Here’s an overview of what DeBERTa is and how it works:

What is DeBERTa?

DeBERTa is an advanced transformer model designed to improve upon the original BERT architecture by addressing some of its limitations. It introduces several key innovations:

- **Disentangled Attention Mechanism**: Unlike BERT, which uses a single attention matrix for both content and position embeddings, DeBERTa uses disentangled attention. This means it separates the attention mechanisms for content and position, which allows the model to better capture the relationship between different words in a sentence.
- **Enhanced Mask Decoder**: DeBERTa employs an enhanced mask decoder to improve the model’s ability to predict masked tokens during training. This enhances the model's pretraining process and leads to better performance on downstream tasks.

How Does DeBERTa Work?

DeBERTa works by leveraging the transformer architecture, which includes an encoder made up of multiple layers of self-attention mechanisms and feed-forward neural networks. Here’s a breakdown of its working:

1. **Tokenization and Embeddings**:
    - Input text is tokenized into subwords or tokens.
    - Each token is converted into two types of embeddings: content embeddings (which represent the token itself) and position embeddings (which represent the token's position in the sequence).

2. **Disentangled Attention**:
    - The disentangled attention mechanism separates the processing of content and position embeddings. This means the model creates two separate attention matrices: one for content and one for position.
    - During attention computation, these two matrices are combined to form a more nuanced understanding of the token relationships.
3. **Transformer Layers**:
    - The token representations pass through multiple transformer layers. Each layer consists of multi-head self-attention and feed-forward neural networks.
    - The self-attention mechanism helps the model focus on relevant parts of the input sequence by weighting the importance of different tokens.
4. **Output Representations**:
    - After passing through the transformer layers, the final token representations can be used for various NLP tasks such as classification, named entity recognition, or machine translation.
5. **Pretraining and Fine-tuning**:
    - DeBERTa is pretrained on a large corpus of text using a masked language modeling objective, where certain tokens in the input are masked, and the model learns to predict them.
    - The pretrained model is then fine-tuned on specific downstream tasks with task-specific labeled data to adapt the general language understanding to the particular task.

Advantages of DeBERTa:

- **Improved Context Understanding**: The disentangled attention mechanism helps DeBERTa better capture the relationships between tokens, improving context understanding.
- **Enhanced Performance**: DeBERTa has shown superior performance on various NLP benchmarks compared to traditional BERT and other transformer models.
- **Flexibility**: It can be applied to a wide range of NLP tasks, making it a versatile tool in the field.

2.1.2.3 Adam optimizer:

The Adam optimizer is an advanced optimization algorithm designed to train deep learning models more efficiently. It combines the best features of two other popular optimization algorithms: AdaGrad and RMSProp. Here's a detailed explanation of what the Adam optimizer is and how it works:

What is Adam Optimizer?

Adam (short for Adaptive Moment Estimation) is an optimization algorithm used in training machine learning and deep learning models. It is particularly well-suited for handling sparse gradients on noisy problems. Adam is designed to maintain an adaptive learning rate for each parameter by using estimates of the first and second moments (mean and uncentered variance) of the gradients.

How Does Adam Optimizer Work?

Adam works by computing adaptive learning rates for each parameter. Here’s a step-by-step breakdown of the algorithm:

Initialization:

- Initialize the parameters θ\thetaθ (weights) of the model.
- Initialize the first moment vector m=0m = 0m=0 and the second moment vector v=0v = 0v=0.
- Set the timestep t=0t = 0t=0.
- Set hyperparameters:
  - α (learning rate, typically set to 0.001).
  - β1 (decay rate for the first moment estimates, typically set to 0.9).
  - β2 (decay rate for the second moment estimates, typically set to 0.999).
  - ϵ (small constant to prevent division by zero, typically set to 10−810^{-8}10−8).

Parameter Update Loop (for each training step):

- Increment the timestep t.
- Compute the gradient g_t of the objective function with respect to the parameters θ at timestep t.
  - Update the biased first moment estimate:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$

  - Update the biased second moment estimate:

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$$

  - Compute bias-corrected first moment estimate:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

  - Compute bias-corrected second moment estimate:

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

  - Update the parameters:

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

-

Repeat:

- Repeat the update loop until convergence or for a predefined number of iterations.

Advantages of Adam Optimizer

1. **Adaptive Learning Rate**: The learning rate adapts based on the mean and variance of the gradients, which can lead to faster convergence.
2. **Bias Correction**: Adam includes bias correction to address the issue of the moment estimates being biased towards zero, especially in the initial steps.
3. **Efficient**: Adam is computationally efficient and requires little memory, making it suitable for large datasets and high-dimensional parameter spaces.
4. **Robust to Noisy Data**: It works well with sparse gradients and noisy data, making it versatile for various types of machine learning problems.

2.2 Recent Literature Review

2.2.1 Advances in NLP for Bug Triaging

Recent studies have explored the application of machine learning techniques to improve the accuracy and efficiency of bug triaging systems. Support Vector Machines and Term Frequency-Inverse Document Frequency have been particularly effective in enhancing bug report classification.

2.2.2 Integration of AI in Software Development Tools

The integration of AI-driven tools in software development has been a growing trend. Research has emphasized the benefits of automation in reducing manual effort and enhancing overall productivity. AI tools help in various stages of software development, from code review to bug triaging, thereby improving efficiency and accuracy.

2.3 Implemented Approach

Based on our literature review and comparative study, we have chosen an approach that leverages state-of-the-art NLP and machine learning models to automate the developer prediction of the bug reports.

2.3.1 Chosen Approach

- **NLP Models:**

    - We use TF-IDF for feature extraction
- **Machine Learning Algorithms:**
    - We use SVM for developer prediction.

2.3.2 Justification

The selected approach provides a balance between accuracy, efficiency, and scalability. By using advanced models like SVM and leveraging the power of machine learning, our system is designed to handle large volumes of bug reports with high precision and minimal manual intervention. This ensures timely resolution of critical issues, improves software quality, and enhances user satisfaction.

2.3.3 Conclusion

In conclusion, this chapter has provided a detailed review of the foundational knowledge and recent advancements related to the Automated Bug Triaging System. The insights gained from this literature survey have informed the design and implementation of our modules, ensuring it is built on a solid foundation of proven methodologies and cutting-edge research.

Chapter 3: Modules Design and Architecture

This chapter represents the main body of our Automated Developer prediction and recommender modules. It provides a detailed description of the modules design and architecture, answering the questions "what has been done?" and "how it has been done?". We will highlight and discuss the steps taken to realize the project, clarifying our scientific approaches and methodologies.

3.1. Overview and Assumptions

In designing our Automated Developer prediction and recommender modules, we aimed to automate the process of handling bug reports. Our system leverages machine learning and NLP techniques to achieve high accuracy and efficiency. The key assumptions we employed are:

- **Data Quality**:
    - We assumed the availability of high-quality bug report data, with sufficient historical data for training machine learning models.
- **Consistent Terminology**:
    - We assumed that bug reports follow a consistent terminology, which is crucial for accurate text processing and feature extraction.

- **Scalability Requirements**:
    - The system is designed to handle a large volume of bug reports efficiently, assuming that the underlying infrastructure supports scalable solutions.

3.2. Developer prediction

Predicting the developer responsible for resolving a bug is a crucial step in the bug triaging process. This task involves using machine learning algorithms to analyze historical bug reports and predict the most suitable developers for new reports. By accurately identifying the top 5 developers likely to resolve an issue, this process helps in organizing and managing bug reports more efficiently, ensuring that bugs are quickly assigned to the right individuals for resolution.

3.2.1. Functional Description

The developer prediction module uses machine learning algorithms to predict the top 5 developers likely to resolve new bug reports. This model leverages historical data to learn patterns associated with specific developers, aiding in the efficient assignment of bug reports. By predicting the most suitable developers, the model helps streamline the bug triage process, ensuring faster and more accurate resolution of issues.

3.2.2. Modular Decomposition

The classification module consists of the following submodules:

- **Dataset exploring and cleaning:**
    - **Function:**
        - Performs various preprocessing steps on multiple datasets to prepare them for analysis, including cleaning the data to ensure quality and consistency.
    - **Steps:**
        - Data Acquisition:
            - Kaggle Dataset: Collected a dataset from Kaggle.
            - Eclipse and Mozilla Datasets: Gathered datasets from different products and components in Bugzilla.
        - Initial Exploration: Printed the shape (number of rows and columns) and columns of each dataset to understand their structure.
        - Data Cleaning:

            - **Remove Duplicates:** Identified and removed duplicate rows to ensure each entry is unique.
            - **Filter Resolutions:** Removed rows where the 'Resolution' column is marked as 'DUPLICATE' or 'INVALID' to focus on valid bug reports.
            - **Drop Unnecessary Columns:** Removed columns that are not needed for the analysis to simplify the dataset.
            - **Handle Null Values:** Removed rows with null values to ensure the dataset is complete.
            - **Summary Text Cleaning:** Cleaned the text in the 'Summary' column by removing special characters, newlines, and hyperlinks to standardize the text data.
- **Preprocessing:**
    - **Function:**
        - Combines and preprocesses multiple datasets to create a unified and cleaned dataset, ready for analysis. This includes merging datasets, text preprocessing, and filtering based on specific criteria.
    - **Steps:**
        - Dataset Merging: Combined the Kaggle, Eclipse, and Mozilla datasets into a single dataset.
        - Data Cleaning:
            - **Remove Duplicates:** Identified and removed duplicate rows from the merged dataset to ensure each entry is unique.
            - **Filter Short Summaries:** Removed rows where the 'Summary' column contains fewer than 10 words to ensure sufficient detail in the text data.
        - Text Preprocessing:
            - **Tokenization:** Applied tokenization to the 'Summary' column, breaking down the text into individual words (tokens).
            - **Remove Stop Words:** Removed common stop words (e.g., 'the', 'and', 'is') from the tokens to focus on the most meaningful words.
            - **Stemming:** Applied stemming to reduce words to their base or root form (e.g., 'running' becomes 'run').
            - **Rejoin Tokens:** Joined the processed tokens back into a single string for each entry in the 'Summary' column.
        - Data Filtering:
            - **Owner Occurrence:** Filtered the dataset to include only those entries where each owner has at least 5 occurrences, ensuring enough data for each owner.
- **Feature Vector Generation:**

    - **Function:**
        - Uses TF-IDF to convert bug report text into numerical feature vectors.
    - **Steps:**
        - Generates a vector where each dimension represents the importance of a term in the document and across the entire corpus.
- **Model Training:**
    - **Function:**
        - Trains a Support Vector Machine (SVM) model with a linear kernel to predict the developer responsible for solving a bug based on historical bug reports.
    - **Steps:**
        - **Model Selection:** Chose an SVM model with a linear kernel for its simplicity and effectiveness in high-dimensional spaces.
        - **Hyperparameter Setting:**Set the regularization parameter CCC to 10. The parameter CCC controls the trade-off between achieving a low training error and a low testing error, which helps in avoiding overfitting.
        - **Training Data:** Utilized historical bug reports as the training data. These reports included features extracted during preprocessing and the target variable, which is the developer (assignee) who resolved the bug.
        - **Training Process:** Fit the SVM model on the training dataset, where the input features were the processed bug report texts and the target labels were the developers.
- **Prediction:**
    - **Function:**
        - Uses the trained SVM model to predict the top 5 developers likely to resolve new bug reports.

3.2.3. Design Constraints

The clustering module must handle high-dimensional feature vectors efficiently and produce clusters in a timely manner. The choice of kkk (number of clusters) is crucial and can significantly impact the clustering quality. The algorithm should be scalable to accommodate a growing number of bug reports.

3.2.4. Other Description of Module 5

SVM was chosen for its effectiveness in high-dimensional spaces and its robustness in handling imbalanced data. SVMs are particularly well-suited for text classification tasks, as they can effectively manage the sparse nature of text data and identify complex decision boundaries.

Hyperparameter Tuning:

Various hyperparameters were explored to optimize the SVM model, with the best model chosen based on validation accuracy:

- **C Values:** Different values of CCC (0.1, 1, 10, 20, 30, 50, 100) were tested. The regularization parameter CCC controls the trade-off between achieving a low training error and a low testing error. Through experimentation, it was determined that C = 10 provided the best validation accuracy, balancing both underfitting and overfitting.
- **Kernels:** Both linear and RBF (Radial Basis Function) kernels were evaluated. The linear kernel was found to be more suitable for this specific task, as it performed better with the high-dimensional, sparse nature of the text data.
- **Class Weights:** The impact of class weights (balanced, none) was investigated to address class imbalance. Although balanced class weights can adjust for imbalances, the results indicated that the default setting (none) provided better validation accuracy.

Impact of SVD (Singular Value Decomposition):

SVD was applied after TF-IDF and before SVM to reduce the dimensionality of the feature space. However, the results showed that the performance was better without applying SVD. The dimensionality reduction introduced by SVD potentially led to the loss of important information, thereby affecting the classifier's ability to accurately predict the developer.

In summary, The developer prediction module leverages SVM with a linear kernel and C=10 to predict the top 5 developers likely to resolve new bug reports. Despite evaluating alternative models and various hyperparameters, SVM was preferred due to its strong theoretical foundation and suitability for high-dimensional data, making it ideal for this text classification task. This module is essential for streamlining the bug triaging process, ensuring efficient and accurate assignment of bug reports to the appropriate developers for resolution.

3.3. Developers Recommender

Bug assignment is a critical aspect of software development, where efficient allocation of bugs to developers can significantly impact productivity and resolution time. Traditional methods often rely on historical data and human judgment, which can be time-consuming and prone to biases. In response, we developed a recommender module that leverages machine learning to automate bug assignment, ensuring that bugs are directed to developers based on their past performance and expertise, even for developers not present in the training data.

3.3.1. Functional Description

The recommender module operates by receiving developers associated with the detected bug category and their historical bug-solving records. It utilizes a classifier trained on historical bug data to predict the top 5 likely bug categories for each developer's previous bug solutions and for the input bug. The module then selects the top 5 developers based on the overlap between the predicted bug categories for the input bug and those for each developer. This approach ensures that bugs are assigned to developers who have demonstrated proficiency in resolving similar issues, optimizing bug resolution efficiency.

3.3.2. Modular Decomposition

The Developer Recommender module consists of the following submodules:

- **Bug Category Detection:**
    - **Function:** Identifies the category of the input bug.
- **Developer Profiling:**
    - **Function:** Collects and profiles developers associated with the detected bug category.
    - **Implementation:** Gathers historical data on bugs resolved by each developer within the category, forming a basis for their bug-solving expertise.
- **Classifier Application:**
    - **Function:** Applies the trained classifier to predict bug categories for each developer's historical bug-solving records and for the input bug.
    - **Implementation:** Utilizes the classifier's predictions to rank bug categories and determine the top 5 relevant categories for both developers and the input bug.
- **Developer Selection:**
    - **Function:** Selects the top 5 developers based on the intersection of the predicted bug categories for the input bug and each developer's historical bug-solving records.
    - **Implementation:** Computes the overlap between the predicted categories, ensuring that developers with the most relevant expertise are prioritized for bug assignment.

3.3.3. Design Constraints

The recommender module must meet several design constraints to ensure its effectiveness and reliability:

- **Scalability:** Capable of handling large volumes of bug data and developers across multiple categories to support scalable bug assignment.
- **Real-Time Performance:** Provides bug assignment recommendations in real-time to maintain workflow efficiency and responsiveness.
- **Accuracy and Fairness:** Ensures that bug assignments are based on objective criteria (predicted bug categories) to mitigate biases and ensure fair distribution of workload.
- **Interpretability:** Offers transparency in bug assignment decisions by providing clear rationale based on predicted bug categories and developer expertise.

3.3.4. Other Description of Module 6

The Recommender System enhances the bug triaging process by ensuring that bugs are assigned to the most capable developers. This leads to quicker and more effective bug resolution, improving the overall quality and reliability of the software.

The system's effectiveness is largely dependent on the quality of the developer profiles and the accuracy of the classifier. To this end, the module continuously updates developer profiles with new data and feedback, ensuring that recommendations remain relevant and accurate.

In summary, the Recommender System module plays a crucial role in optimizing the bug resolution process by matching bugs with the best-suited developers. By leveraging historical data and machine learning techniques, it ensures that bug reports are handled efficiently and effectively, ultimately enhancing the software development lifecycle.

Conclusion

The detailed examination of the two modules within our Automated Bug Triaging System highlights their individual functionalities. Here is a concise conclusion of each module discussed in Chapter 3:

Developer Prediction Model

The developer prediction model leverages Support Vector Machine (SVM) to predict the top 5 developers most likely to resolve new bug reports based on historical data. This module significantly improves the efficiency of bug triaging by automating the assignment process and ensuring that bugs are directed to developers with relevant expertise. By accurately predicting the most suitable developers, the model reduces the time required for manual bug assignment and enhances the overall productivity of the development team.

Recommender Model

The recommender model builds on the developer prediction model by extending its capabilities to developers not present in the training data. It receives all developers within the detected category and their historical bug-solving records, then applies the classifier to predict the top bug categories for each developer. By matching these categories with the top categories of the input bug, the module selects the top 5 developers with the most relevant expertise. This approach ensures that bugs are assigned efficiently and fairly, leveraging historical data to optimize bug resolution processes and improve the effectiveness of bug management.

Chapter 4: Modules Testing and Verification

- **Developer Prediction Module**
    - **Test Description:**
        - The SVM-based developer assignment module is tested for its accuracy in assigning bug reports to the appropriate developer.
    - **Test Cases:**
        - Input: Preprocessed bug reports labeled with assigned developer.
        - Expected Output: Correctly predicted developers for the test bug reports.
    - **Results:**
        - The SVM model achieves an accuracy of 74.1% in predicting the correct developer with the highest probability, and 96.5% in including the correct developer within the top-5 predictions, demonstrating reliable developer assignment.
- **Developers Recommender Module**
    - **Test Description:**
        - The Developers Recommender Module is evaluated for its accuracy in predicting the most suitable top-5 developers for a given bug report.
    - **Test Cases:**
        - Input:
            - All developers in the team as identified by the categorization module.
            - The bugs previously solved by these developers.
            - The input bug report.
        - Expected Output: The five most suitable developers to address the bug, based on the following criteria:

- Developers who have previously solved bugs similar to the input bug report.
- Developers who have not yet solved any bugs, ensuring they have the opportunity to address new issues.

Chapter 5: Conclusions and Future Work

This chapter provides a comprehensive summary of the Automated Developer prediction and recommender modules, highlighting their limitations. It also discusses the challenges faced and the skills gained during the project. Finally, directions for future work are outlined to guide further research and development in this area.

5.1. Faced Challenges

Throughout the development of the Automated Developer prediction and recommender modules, several challenges were encountered:

- Data Quality and Availability:
  - Ensuring high-quality and comprehensive datasets was crucial. Inconsistent or incomplete data could lead to inaccurate predictions. This was mitigated by implementing robust data preprocessing steps to clean and normalize the data before training the models.
- Model Performance and Accuracy:
  - Achieving high accuracy in developer assignment models was challenging due to the diversity and complexity of bug reports. Extensive hyperparameter tuning was used to optimize model performance.

5.2. Gained Experience

Working on the Automated Developer prediction and recommender modules provided valuable experience and skills, including:

- **Advanced NLP Techniques:**
  - Gained in-depth knowledge of natural language processing techniques, including TF-IDF.
- **Machine Learning Proficiency:**

    - Developed expertise in training and optimizing machine learning models, such as SVM and DeB, for specific tasks like developer prediction.
- **Data Preprocessing:**
    - Learned the importance of data preprocessing and the various techniques used to clean and prepare data for analysis.

Limitations:

- **Dependency on Data Quality:**
    - The accuracy of the system is heavily dependent on the quality of the input data. Poor data quality can lead to inaccurate predictions.
- **Integration Challenges:**
    - Integrating with diverse development environments may require additional customization and effort.

5.3. Future Work

The modules opens up several avenues for future research and development:

- **Enhanced Models:**
    - Exploring more advanced models and techniques, such as ensemble methods or neural network architectures, to further improve accuracy and efficiency.
- **Real-time Processing:**
    - Developing capabilities for real-time processing and triaging of bug reports to provide instant feedback and resolution recommendations.
- **Scalability Improvements:**
    - Further optimizing the system for scalability to handle even larger datasets and more complex bug reports efficiently.

In conclusion, the Automated Developer prediction and recommender modules provide a robust and efficient solution for managing bug reports, leveraging advanced NLP and machine learning techniques. The experiences gained and the challenges overcome during this project lay a strong foundation for future enhancements and developments in automated bug triaging.

References

A Comparative Study of Transformer-based Neural Text Representation Techniques on Bug Triaging