

## Article

# SevPredict: Exploring the Potential of Large Language Models in Software Maintenance

Muhammad Ali Arshad<sup>1</sup>, Adnan Riaz<sup>2,\*</sup> , Rubia Fatima<sup>3</sup>  and Affan Yasin<sup>4,\*</sup> 

<sup>1</sup> Department of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China; aliarshadciitswl@gmail.com

<sup>2</sup> Department of Computer Science and Engineering, University of Bologna, 40126 Bologna, Italy

<sup>3</sup> Faculty of Computing and Emerging Technologies, Emerson University, Multan 60000, Pakistan; rubiafatima91@hotmail.com

<sup>4</sup> School of AI and Advanced Computing, Xi'an Jiaotong-Liverpool University, Suzhou 215123, China

\* Correspondence: adnan.riaz3@unibo.it (A.R.); affan.yasin@outlook.com (A.Y.)

**Abstract:** The prioritization of bug reports based on severity is a crucial aspect of bug triaging, enabling a focus on more critical issues. Traditional methods for assessing bug severity range from manual inspection to the application of machine and deep learning techniques. However, manual evaluation tends to be resource-intensive and inefficient, while conventional learning models often lack contextual understanding. This study explores the effectiveness of large language models (LLMs) in predicting bug report severity. We propose a novel approach called SevPredict using GPT-2, an advanced LLM, and compare it against state-of-the-art models. The comparative analysis between the proposed approach and state-of-the-art approaches suggests that the proposed approach outperforms the state-of-the-art approaches in terms of performance evaluation metrics. SevPredict shows improvements over the best-performing state-of-the-art approach (BERT-SBR) with 1.72% higher accuracy, 2.18% higher precision, and 4.94% higher MCC. The improvements are even more substantial when compared to the approach by Ramay et al., with SevPredict demonstrating 10.66% higher accuracy, 10.39% higher precision, 3.29% higher recall, 7.19% higher F1-score, and a remarkable 41.27% higher MCC. These findings not only demonstrate the superiority of our GPT-2-based approach in predicting the severity of bug reports but also highlight its potential to significantly advance automated bug triaging and software maintenance. This research introduces a severity prediction tool named SevPredict.

**Keywords:** mining software repository; severity prediction; large language models



**Citation:** Arshad, M.A.; Riaz, A.; Fatima, R.; Yasin, A. SevPredict: Exploring the Potential of Large Language Models in Software Maintenance. *AI* **2024**, *5*, 2739–2760. <https://doi.org/10.3390/ai5040132>

Academic Editor: Isidoros Perikos

Received: 29 September 2024

Revised: 13 November 2024

Accepted: 19 November 2024

Published: 5 December 2024



**Copyright:** © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction

In today's digital age, software applications have gained immense popularity globally, with millions of users interacting with them daily. These users often voice their feedback and needs through various channels, including app stores, bug repositories, and developer forums. This is a philosophy demonstrated in Mozilla principle, where it states that “The heart of Mozilla is people”. Users often run into issues like crashes, hangs, or security holes and may also ask for improvements like glossier user interfaces or new functionality. Bug fixing is a crucial element in software development. Companies are using tools like Bugzilla from Mozilla (<https://bugzilla.mozilla.org/>, accessed on 14 January 2024), GitHub (<https://www.atlassian.com/software/jira/bug-tracking>, accessed on 14 January 2024), and Jira (<https://github.com/features>, accessed on 14 January 2024) to keep an eye on user feedback. This feedback is critical to improving software applications [1]. However, a significant challenge arises from the overwhelming amount of bug reports and the limited capacity of developers to address them [2]. Bugzilla processed around 300 bug reports/day

on average in 2013; however, Mozilla reported needing to triage 135 files a day [3,4]. The breadth of these reports can become overwhelming for developers trying to process them.

The reliance on software applications has been on the rise over the past 10 years, resulting in larger numbers of bug reports as well. Handling these reports manually is laborious and monotonous, making it one of the major contributors to the high maintenance cost, which is estimated to be around 60% of the software development life cycle (SDLC) and is the expensive phase of the SDLC [5–7]. According to the Consortium for Information and Software Quality (CISQ) (<https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2022-report/>, accessed on 14 January 2024), poor software cost the U.S economy \$2.08 trillion in 2020 and 2.41 trillion in 2022. In such a context, automatic severity prediction comes up as a key support, making it possible for developers to determine the severity of bug reports that saves many resources, including time, money, and labor during bug triaging.

- We introduce SevPredict, a novel severity prediction framework leveraging GPT-2's transformer architecture for automated bug report classification. Our approach implements a fine-tuned language model that processes unstructured bug report text to extract semantic features and contextual patterns relevant to severity assessment.
- Through comprehensive evaluation on bug repositior, SevPredict demonstrates statistically significant improvements over state-of-the-art baselines: a 1.72% increase in accuracy and 4.94% in Matthews Correlation Coefficient compared to BERT-SBR, and more substantial gains of 10.66% in accuracy and 41.27% in MCC when compared to traditional machine learning approaches.
- We provide SevPredict as an open-source tool that integrates seamlessly with existing bug tracking systems. Researchers and practitioners can access our pre-trained models and user-friendly APIs through our public repository at <https://huggingface.co/spaces/AliArshad/SeverityPrediction>, accessed on 14 January 2024.

Together, these contributions significantly improve the efficiency and accuracy of bug severity prediction processes in software development, promising substantial enhancements in software quality and maintenance. The demonstrated performance of SevPredict over existing approaches underscores its potential to revolutionize automated bug triaging and streamline software maintenance workflows.

The rest of the paper is organized as follows: Section 2 reviews the theoretical background of severity prediction and examines current state-of-the-art approaches in this field. Section 3 details our proposed methodology, describing the dataset and evaluation metrics employed. Section 4 presents our experimental findings and provides a comprehensive analysis of the results. Finally, Section 5 summarizes our conclusions and outlines directions for future research.

## 2. Background

Automatically predicting severity has become a significant area of focus in software development. Severity represents the degree of impact a bug has on the functionality of a software system and is crucial for prioritizing bug resolution and planning bug fixing activities. This section describes various methods for predicting the severity of bug reports.

Menzies et al. (2008) proposed SEVERIS, an automated approach to severity assessment using text mining. However, the effectiveness of their approach was limited by the size of the training dataset, which highlights the importance of larger datasets to obtain reliable results [8]. Lamkanfi et al. (2010, 2011) focused on text information in bug reports and utilized mining algorithms such as Naive Bayes, SVM, and Multinomial Naive Bayes. Their results indicated that the prediction accuracy varied across different software components [9,10]. Valdivia et al. (2014) proposed methods for predicting blocking bugs using models such as random forest classifiers and integrating structured and free text data [11]. Sharma et al. (2015) proposed a dictionary of key terms for severity prediction. Their approach solely focused on the summary of the bug report, potentially overlooking key information in the detailed description [12]. Zhang et al. (2016) and

Sabor et al. (2016, 2019) made significant contributions by employing topic modeling and combining stack traces with categorical features to enhance the accuracy of severity predictions [13–15]. Yang et al. (2018) introduced an emotion-based method, modifying the Naive Bayes algorithm to incorporate emotional aspects for improved prediction accuracy [16].

In recent years, several researchers have explored deep learning models and multiple attributes for severity prediction. These methods have shown improvements in accuracy, precision, and recall, addressing some limitations of earlier approaches. Ramay et al. (2019) [17] used Senti4SD to calculate the sentiments score of the bug reports. Sharma et al. (2019) developed models based on multiple attributes, demonstrating that combinations of attributes like bug age and summary weight could yield better results [18]. Ali et al. (2024) [19] used BERT for the severity prediction of bug reports for the maintenance of mobile applications and achieved better performance over other machine learning and deep-learning models.

While previous studies have made significant contributions to bug severity prediction, they face several limitations. Earlier works often relied on smaller datasets, limiting their generalizability. Many approaches depended on manual feature engineering or simple text analysis techniques, potentially missing complex patterns in bug reports. Some methods focused solely on specific aspects such as summary text or emotional content, overlooking other potentially crucial information. Additionally, traditional machine learning models used in many studies may struggle to capture the nuanced contextual relationships present in bug report text. In contrast, our approach leverages the capabilities of pretrained large language models—specifically, GPT-2 for classification and BERT for sentiment analysis. This methodology offers several advantages: it can potentially capture more complex patterns and contextual information from bug reports, reduce the need for manual feature engineering, and benefit from transfer learning to improve generalization across different types of bug reports and software projects. By combining advanced text classification with nuanced sentiment analysis, both based on pretrained models, our method aims to provide a more comprehensive and accurate prediction of bug severity, addressing many of the limitations found in previous works.

### 3. Methodology

The proposed method involves several key steps. Initially, we extracted and pre-processed the dataset. Subsequently, we utilized BERT to calculate the sentiments of the bug reports. Following that, the GPT-2 tokenizer was employed to convert the text into embeddings, which were then fed into our classifier. Finally, the GPT-2 model underwent fine-tuning using the feature vectors generated by the GPT-2 tokenizer. The details of each step are as follows:

#### 3.1. Dataset

In our experimental analysis, we utilized the datasets as detailed by [20]. This dataset encompasses eight distinct projects: Eclipse Platform, Eclipse JDT, Eclipse CDT, Eclipse PDE, Mozilla Core, Mozilla Firefox, Mozilla Thunderbird, and Mozilla Bugzilla. Each dataset comprises bug reports, categorized into nine resolution statuses—fixed, remind, incomplete, invalid, duplicate, not\_eclipse, worksforme, later, and wontfix—alongside severity labels such as blocker, critical, major, minor, normal, trivial, and enhancement.

The dataset's composition in terms of bug reports for each project is as follows: Eclipse Platform (24,775), Eclipse JDT (10,814), Eclipse CDT (5640), Eclipse PDE (5655), Mozilla Core (74,292), Mozilla Firefox (69,879), Mozilla Thunderbird (19,237), and Mozilla Bugzilla (4616), culminating in a total of 214,888 entries. This total includes all bug reports across the spectrum of resolution statuses and severity labels. The detail of the dataset is given in Table 1.

**Table 1.** Bug severity distribution across projects.

Project	Blocker	Critical	Major	Normal	Minor	Trivial
Bugzilla	275 (5.96%)	176 (3.81%)	506 (10.96%)	2478 (53.68%)	766 (16.59%)	415 (8.99%)
CDT	78 (1.38%)	166 (2.94%)	490 (8.69%)	4547 (80.62%)	275 (4.88%)	84 (1.49%)
Core	451 (0.61%)	10,542 (14.19%)	4243 (5.71%)	56,125 (75.55%)	2072 (2.79%)	859 (1.16%)
Firefox	233 (0.33%)	6603 (9.45%)	9486 (13.57%)	47,635 (68.17%)	4145 (5.93%)	1777 (2.54%)
JDT	94 (0.87%)	274 (2.53%)	1000 (9.25%)	8306 (76.81%)	781 (7.22%)	359 (3.32%)
PDE	47 (0.83%)	117 (2.07%)	476 (8.42%)	4693 (82.99%)	208 (3.68%)	114 (2.02%)
Platform	415 (1.68%)	989 (3.99%)	2718 (10.97%)	18,891 (76.25%)	1088 (4.39%)	674 (2.72%)
Thunderbird	65 (0.34%)	1894 (9.85%)	2982 (15.50%)	12,429 (64.61%)	1415 (7.36%)	452 (2.35%)

Binary Classification Statistics		
Project	Severe Bugs	Percentage
Bugzilla	957/4616	20.7%
CDT	734/5640	13.0%
Core	15,236/74,292	20.5%
Firefox	16,322/69,879	23.4%
JDT	1368/10,814	12.7%
PDE	640/5655	11.3%
Platform	4122/24,775	16.6%
Thunderbird	4941/19,237	25.7%

Notes: Severe bugs include Blocker, Critical, and Major severity levels. Non-Severe bugs include Normal, Minor, and Trivial severity levels. Numbers in parentheses show the percentage within each project, and percentages may not sum to 100% due to rounding.

For our study, we refined the dataset to include only those bug reports labeled with a resolution status of ‘fixed’. The details of all the bug reports with resolution status = fixed are given in Table 2. We performed the severity prediction of the bug report task using a binary categorization approach. A fine-grained approach utilizes a detailed set of distinct labels for classification or analysis. In contrast, a coarse-grained approach involves broader categorization by consolidating labels into fewer, more general classes. Based on our literature review [21–23], we found that a fine-grained approach decreases the performance of the classifier. Therefore, we adopted the coarse-grained approach, which has been used in recent studies [21–24]. Specifically, we transformed the multilabel categories (Blocker, Critical, Major, Normal, Minor, and Trivial) into a binary classification: Blocker, Critical, and Major were grouped as ‘severe’, while Normal, Minor, and Trivial were grouped as ‘non-severe’.

This filtration resulted in a subset of 88,682 bug reports. This refined dataset, pivotal to our experimental framework, has been made publicly available on the Hugging-Face (<https://huggingface.co/>, Verified: 10 September 2024) platform under the repository name ‘AliArshad/Bugzilla\_Eclipse\_Bug\_Reports\_Dataset’ ([https://huggingface.co/datasets/AliArshad/Bugzilla\\_Eclipse\\_Bug\\_Reports\\_Dataset](https://huggingface.co/datasets/AliArshad/Bugzilla_Eclipse_Bug_Reports_Dataset), accessed on 14 January 2024). The details of this dataset are presented in Tables 1 and 2. Excluding normal bug reports results in a dataset of 16,512 bug reports, of which 11,307 are severe and 5205 are non-severe.

**Table 2.** Bug severity distribution of fixed bugs across projects.

Project	Blocker	Critical	Major	Normal	Minor	Trivial	Total
Bugzilla	265 (10.90%)	100 (4.11%)	253 (10.40%)	1033 (42.48%)	492 (20.23%)	289 (11.88%)	2432
CDT	43 (1.01%)	89 (2.10%)	303 (7.15%)	3539 (83.51%)	194 (4.58%)	70 (1.65%)	4238
Core	334 (0.71%)	5834 (12.42%)	2043 (4.35%)	36,960 (78.70%)	1157 (2.46%)	633 (1.35%)	46,961
Firefox	122 (1.04%)	275 (2.34%)	670 (5.70%)	9787 (83.21%)	474 (4.03%)	434 (3.69%)	11,762
JDT	38 (0.64%)	127 (2.15%)	547 (9.25%)	4508 (76.23%)	400 (6.76%)	294 (4.97%)	5914
PDE	24 (0.60%)	81 (2.04%)	303 (7.64%)	3312 (83.47%)	155 (3.91%)	93 (2.34%)	3968
Platform	142 (1.02%)	418 (3.01%)	1412 (10.17%)	10,793 (77.75%)	580 (4.18%)	537 (3.87%)	13,882
Thunderbird	34 (0.92%)	145 (3.94%)	287 (7.80%)	2842 (77.27%)	222 (6.04%)	148 (4.02%)	3678

Binary Classification Statistics		
Project	Severe Bugs	Percentage
Bugzilla	618/2432	25.4%
CDT	435/4238	10.3%
Core	8211/46,961	17.5%
Firefox	1067/11,762	9.1%
JDT	712/5914	12.0%
PDE	408/3968	10.3%
Platform	1972/13,882	14.2%
Thunderbird	466/3678	12.7%

Notes: Severe bugs include Blocker, Critical, and Major severity levels. Numbers in parentheses show the percentage within each project, and percentages may not sum to 100% due to rounding.

### 3.2. Sentiment Calculation

The sentiments are usually used to describe whether a given statement is positive, negative, or neutral. There are many use cases of sentiment calculation, such as in movie reviews and application reviews. Researchers in the past, including [19,22,25], have shown that adding sentiments to a classifier can help increase the performance of the classifier in the domain of severity/priority prediction. The hypothesis behind it is that bug reports with negative sentiments show urgency, and those are mostly severe, while bug reports with positive sentiments are non-severe and require no urgency. To calculate the sentiments of the bug reports, there are several tools available such as SentiCR [26], DEVA [27], SentiWordNet [28], and Senti4SD [29]. Many researchers have recently used BERT [30] for the sentiment calculation such as [31,32]. Researchers in the field of Severity Prediction have only used SentiWordNet and Senti4SD, and they missed experimenting with BERT. The sentiment calculation tools used by researchers for Severity Prediction are mentioned in Table 3. We experimented using SentiWordNet, Senti4SD, and BERT. BERT outperformed all the existing tools available for severity prediction. So, we chose BERT for sentiment calculation for the proposed approach. One reason it outperforms others might be due to contextual and semantic embeddings that other tools miss.

**Table 3.** Researches and corresponding sentiment calculation tools.

Researches	Sentiment Calculation Tool
Ali et al. (2024) [19]	SentiWordNet
M. A. Arshad et al. (2021) [32]	Senti4SD
Qasim Umar et al. (2019) [22]	Senti4SD
Ramay et al. (2019) [17]	Senti4SD
SevPredict (Proposed)	BERT

To calculate the sentiments for the bug reports using BERT, we followed the following steps: First, we imported essential libraries to facilitate the execution of sentiment analysis on textual data. These include the pandas library for efficient data handling and the pipeline

module from the Hugging Face Transformers library for easy utilization of pretrained BERT models. We used the default pipeline, `sentiment_pipeline = pipeline("sentiment-analysis", which uses the "nlptown/bert-base-multilingual-uncased-sentiment" model for sentiment calculation. This model is trained on multilingual data and can perform sentiment analysis on text in various languages. The output from the sentiment prediction model is Positive or Negative, with its score as shown in Table 4.`

**Table 4.** Bug reports sentiment sample.

Bug Report	Sentiment	Confidence
Typo in error message	NEGATIVE	0.9992406368
Cookies are incorrectly detained when logging out	NEGATIVE	0.9980615973
An arrayref is always "true"	POSITIVE	0.9996670484
New charts feature crashes	NEGATIVE	0.9993322492

We ignored the sentiments score, used the sentiment values, and converted them to binary—1 for positive and 0 for negative. Then, we incorporated these values with bug summary embeddings to fine-tune our classifier.

The dataset with sentiment scores calculated using SentiWordNet, Senti4SD, and BERT is available on HuggingFace with the name "Bug\_Reports\_with\_Sentiments" ([https://huggingface.co/datasets/AliArshad/Bug\\_Reports\\_with\\_Sentiments](https://huggingface.co/datasets/AliArshad/Bug_Reports_with_Sentiments), accessed on 14 January 2024).

### 3.3. SevPredict

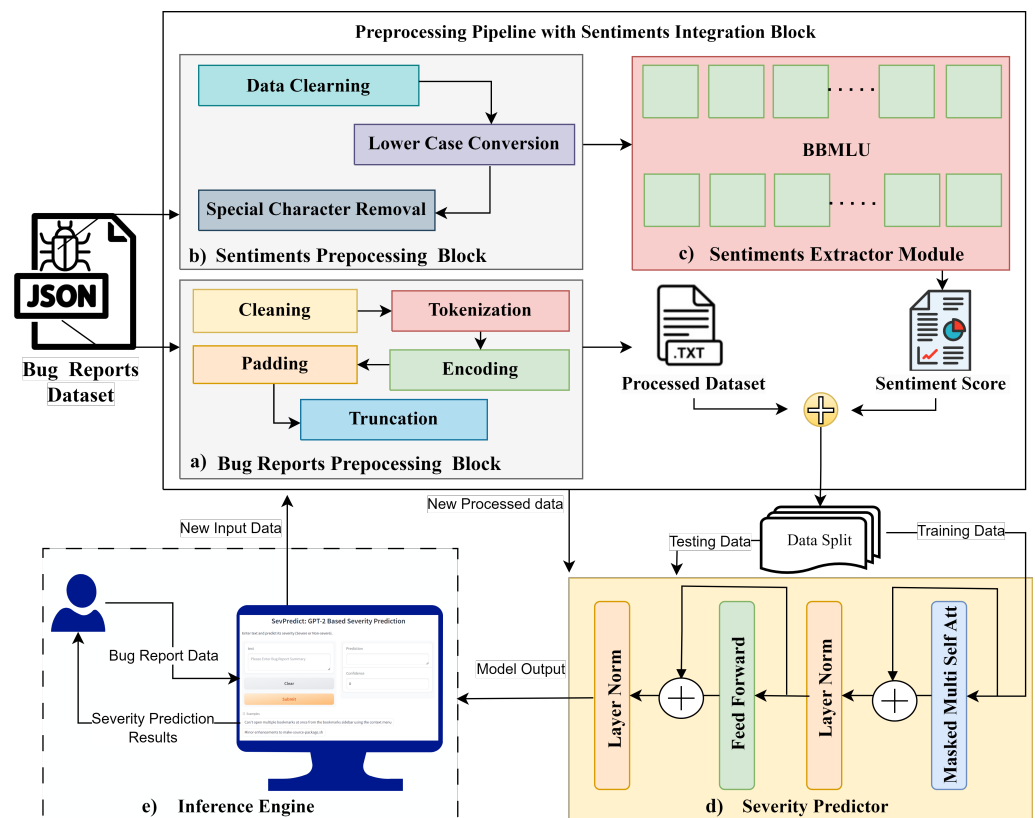
The SevPredict methodology presents a novel approach to bug severity prediction through a comprehensive pipeline that integrates sentiment analysis with traditional bug report processing. The architecture comprises five primary components: bug report preprocessing, sentiment preprocessing, sentiment extraction, severity prediction, and inference deployment. The pipeline initiates with the ingestion of JSON-formatted bug reports, which undergo dual-stream processing. In the primary stream, the Bug Reports Preprocessing Block implements a sequential transformation including cleaning, tokenization, encoding, padding, and truncation operations to standardize the textual data. Concurrently, the Sentiments Preprocessing Block processes the emotional context through systematic data cleaning, lower case conversion, and special character removal, ensuring optimal text normalization for sentiment analysis. A distinctive feature of our methodology is the Sentiments Extractor Module, which employs the bert-base-multilingual-uncased-sentiment (BBMLU) pipeline. This sophisticated pipeline processes the preprocessed textual data, effectively capturing the nuanced sentiment patterns within bug reports. The extracted sentiment scores are then integrated with the processed dataset through a fusion mechanism, creating a rich, multi-dimensional feature representation. The Severity Predictor component implements a cascaded architecture comprising four essential stages: Masked Multihead Self Attention, Layer Normalization, Feed Forward, and Layer Normalization. This component leverages both training and testing datasets through a structured data split mechanism, ensuring robust model validation. The predictor's architecture is specifically designed to handle the complex interplay between textual features and sentiment indicators, enabling more accurate severity classifications. The methodology culminates in an Inference Engine that operationalizes the trained model through a user-centric interface. This component facilitates real-time severity predictions by processing new bug report data through the established pipeline, presenting results in an interpretable format. The entire framework demonstrates a holistic approach to bug severity prediction, incorporating both syntactic and semantic aspects of bug reports while maintaining practical deployability. This integrated approach significantly advances the state-of-the-art in automated bug severity prediction by considering not only the technical content of bug reports but also their underlying sentiment context, potentially leading to more accurate and nuanced severity

assessments in software development environments. The graphical representation of the methodology is presented in Figure 1.

### Overview of GPT-2

GPT, known for its efficacy in natural language processing (NLP), plays a pivotal role in our methodology. GPT-2, leveraging transfer learning, is pretrained on a vast corpus of text data, which equips it with a nuanced understanding of language patterns and structures.

- **Computational Resources:** The study was conducted using Google Colab, leveraging its cloud-based environment. The computational hardware included a Tesla T4 GPU, which provided the necessary computational power for training and evaluating the GPT-2 as well as other models described in this research paper.
- **GPT-2 Model Configuration and Training:** GPT-2 was the primary model for predicting the severity of bugs in software projects. The methodology incorporated the following key steps:



**Figure 1.** Comprehensive architecture of the SevPredict methodology for bug severity prediction. The pipeline consists of the following: (a) Bug Reports Preprocessing Block implementing cleaning, tokenization, encoding, padding, and truncation; (b) Sentiments Preprocessing Block handling text normalization; (c) Sentiments Extractor Module utilizing nlptown/bert-base-multilingual-uncased-sentiment for sentiment analysis; (d) Severity Predictor incorporating multi-layer prediction components; (e) Inference Engine for real-time severity assessment. The architecture demonstrates the integration of both textual and sentiment features through a systematic data flow, enabling comprehensive bug severity prediction.

### 3.4. Dataset Preparation and Preprocessing

This section details our comprehensive approach to preparing and processing bug reports for machine learning analysis. To ensure reproducibility and clarity, we demonstrate our methodology using bug report #330,186 “Crash when changing the status of a bug

which has dependencies” as a representative example. Our pipeline encompasses several critical stages—text cleaning, tokenization, encoding, and dataset creation—each designed to optimize the data for our neural network model.

### 3.4.1. Text Cleaning and Preprocessing

The initial stage of our pipeline focuses on data cleaning while preserving essential semantic information. Bug reports often contain various special characters, formatting elements, and inconsistencies that could decrease model performance. To address this, we developed a systematic cleaning process using the GPT-2 tokenizer from the Hugging Face’s Transformers library (Listing 1). This choice was motivated by GPT-2’s robust tokenization capabilities and its proven effectiveness in handling technical text.

Our preprocessing strategy specifically targets special characters while maintaining alphanumeric content and spaces, ensuring that critical technical information remains intact. We introduced special padding tokens (‘[PAD]’) to standardize input lengths, which is crucial for batch processing in neural networks. The implementation is straightforward yet effective (Listing 2):

**Listing 1.** Text cleaning implementation.

---

```
# Preserve alphanumeric characters and spaces only
return re.sub(r'[^A-Za-z0-9 ]+', '', text)
```

---

**Listing 2.** Tokenizer initialization.

---

```
tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
tokenizer.add_special_tokens({'pad_token': '[PAD]'})
```

---

### 3.4.2. Tokenization and Encoding Process

The tokenization and encoding phase transforms the cleaned text into a format suitable for neural network processing. This multi-step process is crucial for maintaining semantic relationships while converting text into numerical representations that our model can process efficiently.

#### 1. Tokenization

The first step employs the GPT-2 tokenizer, which breaks down text into meaningful subword units. This approach offers a balance between character-level and word-level tokenization, allowing for effective handling of technical terminology and rare words common in bug reports:

#### 2. Parameter Definition

We carefully selected parameters to balance computational efficiency with model performance (Listing 3):

**Listing 3.** Parameter configuration.

---

```
max_len = 100 # Maximum sequence length
pad_token_id = tokenizer.pad_token_id
```

---

The maximum sequence length of 100 was chosen based on empirical analysis of our bug report corpus, providing sufficient context while maintaining computational efficiency.

#### 3. Text Encoding

The encoding process converts tokenized text into numerical tensors suitable for model input (Listing 4):



**Listing 4.** Text encoding process.

---

```
train_encodings = tokenizer(train_texts,
                             truncation=True,
                             padding=True,
                             max_length=max_len,
                             return_tensors='pt')
```

---

To illustrate the complete transformation process, we present the step-by-step changes to our example bug report:

1. After Tokenization: This stage shows how the text is broken into subword units, with 'Ġ' indicating word boundaries:

```
['Crash', 'Ġwhen', 'Ġchanging', 'Ġthe', 'Ġstatus', 'Ġof', 'Ġa',
 'Ġbug', 'Ġwhich', 'Ġhas', 'Ġdependencies']
```

2. After Encoding: The tokenized text is converted into numerical indices corresponding to the model's vocabulary:

```
[47598, 618, 5609, 262, 3722, 286, 257, 5434, 543, 468, 20086]
```

---

3. After Padding and Truncation: Sequences are standardized to the specified maximum length:

```
[47598, 618, 5609, 262, 3722, 286, 257, 5434, 543, 468, 20086,
 None, None, ..., None]
```

---

4. Attention Masks: Binary masks are created to distinguish actual tokens from padding:

```
[1, 1, 1, 1, ..., 0, 0, 0]
```

---

### 3.4.3. Dataset Creation

To efficiently manage the processed data during training and evaluation, we implemented a custom PyTorch dataset class. This implementation facilitates batch processing and ensures proper handling of both features and labels (Listing 5):

**Listing 5.** Custom dataset implementation.

---

```
class CustomDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: val[idx] for key, val in self.encodings.items()}
        item['labels'] = torch.tensor(self.labels[idx])
        return item

    def __len__(self):
        return len(self.labels)
```

---

### 3.4.4. Model Configuration and Training Setup

For our classification task, we utilized a pretrained GPT-2 model adapted for sequence classification. The model was initialized with support for binary classification (Listing 6):

**Listing 6.** Model initialization.

---

```
1 GPT2ForSequenceClassification.from_pretrained('gpt2', num_labels=2)
```

---

The fine-tuning process was carefully configured to optimize model performance while preventing overfitting. Table 5 presents the comprehensive configuration parameters used during model fine-tuning:

**Table 5.** GPT-2 fine-tuning configuration.

Parameter	Value
Computational Environment	Google Colab with Tesla T4 GPU
Maximum Sequence Length	100
Batch Size	32
Number of Epochs	5
Model	GPT2ForSequenceClassification
Number of Labels	2
Optimizer	AdamW
Learning Rate Scheduler	Linear with Warmup
Warmup Steps	500
Weight Decay	0.01

The optimization strategy employed the AdamW optimizer, selected for its effectiveness in training transformer-based models. We implemented a linear warmup schedule with weight decay for regularization, utilizing the Hugging Face’s Trainer API. This configuration was designed to balance training stability with computational efficiency while minimizing the risk of overfitting. The warmup period of 500 steps allows the model to gradually adapt to the task-specific features of bug report classification.

### 3.5. Experimental Setup

To evaluate model generalization across naturally grouped data, we employed the Leave-One-Group-Out (LOGO) cross-validation strategy, a specialized variant of  $k$ -fold cross-validation [33]. This approach is particularly valuable for our study as our dataset contains bug reports from different software projects and time periods [34]. In our implementation, LOGO cross-validation systematically partitions the dataset by keeping each project’s data together as a distinct group. The process begins with preparing the training data ( $app_{train}$ ), which include all bug reports except those from a designated test group ( $group_i$ ). We then train multiple models, including large language models (GPT-2, XLNet, Electra, GPT Neo 1.7b, BERT, and ERNIE) and traditional machine learning approaches (CNN, Multinomial Naive Bayes, Random Forest, and Logistic Regression) using this training set. Each model’s performance is subsequently evaluated on the held-out test group ( $group_i$ ). This process is repeated iteratively, ensuring each group serves as the test set exactly once while the remaining groups form the training set.

To calculate the overall performance, we compute the LOGO Cross-Validation (LOGO CV) metric as follows:

$$\text{LOGO CV} = \frac{1}{N} \sum_{i=1}^N \text{Model Performance on } group_i \quad (1)$$

where  $N$  represents the total number of groups, and Model Performance on  $group_i$  indicates the performance metric for the model trained on all groups except  $i$  and tested on group  $i$ . This comprehensive evaluation approach ensures a thorough assessment of our models’ ability to predict bug severity across diverse software projects, providing insights into their real-world generalization capabilities.

### 3.6. Performance Metrics

To comprehensively evaluate the effectiveness of the GPT-2 model in bug prediction, we employ a suite of established classification performance metrics [35]. Each metric

offers unique insights into different aspects of model performance, particularly crucial for real-world applications where distinguishing between bug and non-bug reports presents inherent challenges.

The foundation of our evaluation begins with accuracy, which measures the overall proportion of correct predictions in bug identification tasks [36]. Accuracy is computed as the ratio of correctly predicted observations (true positives and true negatives) to the total number of bug reports:

$$\text{Accuracy} = \frac{\text{True Positives (TP)} + \text{True Negatives (TN)}}{\text{Total Bug Reports}} \quad (2)$$

While accuracy provides a general overview, we incorporate precision and recall metrics for a more nuanced evaluation [37]. Precision quantifies the accuracy of bug predictions, indicating the proportion of correctly identified bugs among all predicted bugs:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{False Positives (FP)}} \quad (3)$$

Recall complements precision by measuring the model's ability to identify all actual bugs in the dataset:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{False Negatives (FN)}} \quad (4)$$

To balance the trade-off between precision and recall, we utilize the F1-Score [38], which represents their harmonic mean:

$$\text{F1-Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad (5)$$

Finally, we incorporate the Matthews Correlation Coefficient (MCC) [39], which is particularly valuable for our bug prediction task due to the potential imbalance between bug and non-bug reports in software projects:

$$\text{MCC} = \frac{\text{TP} \times \text{TN} - \text{FP} \times \text{FN}}{\sqrt{(\text{TP} + \text{FP})(\text{TP} + \text{FN})(\text{TN} + \text{FP})(\text{TN} + \text{FN})}} \quad (6)$$

This comprehensive set of metrics enables us to evaluate our model's performance from multiple perspectives, ensuring a thorough assessment of GPT-2's capability in distinguishing between bug and non-bug reports across different software projects:

- True Positives (TP): Correctly identified bug reports;
- True Negatives (TN): Correctly identified non-bug reports;
- False Positives (FP): Non-bug reports incorrectly classified as bugs;
- False Negatives (FN): Bug reports incorrectly classified as non-bugs.

#### 4. Experiments Results and Evaluation

This section presents a comprehensive evaluation of our proposed approach for bug severity prediction. We begin by outlining six research questions that guide our investigation, followed by detailed analyses of experimental results and their implications. Our evaluation framework examines multiple aspects of the model's performance, from basic effectiveness to comparative analysis against state-of-the-art approaches.

##### 4.1. Research Questions

To systematically evaluate our approach, we formulated the following research questions:

- RQ1. What is the performance of the GPT-2 base model in predicting bug severity?
- RQ2. What advantages does GPT-2's transformer architecture provide for bug severity prediction compared to traditional machine learning models?

- RQ3. How does our SevPredict model compare to state-of-the-art approaches [17,19] approaches across key metrics?
- RQ4. How do different validation strategies impact model performance?
- RQ5. What impact does sentiment analysis have on bug severity prediction accuracy?
- RQ6. How do different data balancing strategies affect model performance?

#### 4.2. Base Model Performance (RQ1)

Our initial evaluation focused on assessing GPT-2's performance using LOGO cross-validation across various software projects. Table 6 presents the results of this analysis.

**Table 6.** Comprehensive analysis of GPT-2 performance across different software projects.

Project	Confusion Matrix and Distribution			Performance Metrics	
Bugzilla	<b>Actual Condition</b>			Accuracy	68.2%
	<b>Predicted</b>	Bug	Non-Bug	Precision	71.6%
	Bug	330 (31.8%)	150 (14.4%)	Recall	67.8%
	Non-Bug	180 (17.3%)	379 (36.5%)	F1-score	69.7%
CDT	<b>Actual Condition</b>			MCC	0.365
	<b>Predicted</b>	Bug	Non-Bug	Accuracy	73.2%
	Bug	170 (26.2%)	91 (14.0%)	Precision	77.0%
	Non-Bug	83 (12.8%)	304 (46.9%)	Recall	78.6%
Core	<b>Actual Condition</b>			F1-score	77.8%
	<b>Predicted</b>	Bug	Non-Bug	MCC	0.439
	Bug	1199 (13.7%)	525 (6.0%)	Accuracy	<b>87.2%</b>
	Non-Bug	592 (6.8%)	6416 (73.5%)	Precision	<b>92.4%</b>
Firefox	<b>Actual Condition</b>			Recall	<b>91.6%</b>
	<b>Predicted</b>	Bug	Non-Bug	F1-score	<b>92.0%</b>
	Bug	626 (37.3%)	214 (12.7%)	MCC	<b>0.602</b>
	Non-Bug	167 (9.9%)	672 (40.0%)	Accuracy	77.3%
JDT	<b>Actual Condition</b>			Precision	75.9%
	<b>Predicted</b>	Bug	Non-Bug	Recall	80.1%
	Bug	466 (40.5%)	134 (11.6%)	F1-score	77.9%
	Non-Bug	152 (13.2%)	399 (34.7%)	MCC	0.547
Platform	<b>Actual Condition</b>			Accuracy	75.2%
	<b>Predicted</b>	Bug	Non-Bug	Precision	74.9%
	Bug	717 (27.9%)	266 (10.4%)	Recall	72.4%
	Non-Bug	334 (13.0%)	1248 (48.7%)	F1-score	73.6%
Thunderbird	<b>Actual Condition</b>			MCC	0.502
	<b>Predicted</b>	Bug	Non-Bug	Accuracy	76.6%
	Bug	221 (31.7%)	96 (13.8%)	Precision	82.4%
	Non-Bug	74 (10.6%)	307 (44.0%)	Recall	78.9%
				F1-score	80.6%
				MCC	0.512

**Average Performance:** Accuracy: 76.2%, Precision: 78.6%, Recall: 78.6%, F1-score: 78.6%, MCC: 0.496

#### Notes:

##### 1. Confusion Matrix Interpretation:

True Positive (Bug/Bug): Correctly identified bugs;

False Positive (Bug/Non-Bug): Incorrectly classified as bugs;

False Negative (Non-Bug/Bug): Missed bugs;

True Negative (Non-Bug/Non-Bug): Correctly identified non-bugs.

##### 2. Percentages show the proportion of each cell in the project's total dataset.

##### 3. Best performance metrics across projects are highlighted in **bold**.

The experimental results demonstrate varying performance across different projects, with substantial differences in both accuracy and the distribution of prediction outcomes. The Core project exhibited exceptional performance, achieving the highest scores across all metrics with an accuracy of 0.872 and an F1-score of 0.920. This superior performance is evidenced by its confusion matrix values, showing 1199 true positives and 6416 true negatives out of 8732 total cases, indicating strong predictive power for both bug and non-bug reports. The relatively low false positive (525) and false negative (592) rates, despite having the largest dataset, further underscore the model's reliability for this project.

In contrast, the Bugzilla project showed relatively lower performance metrics (accuracy: 0.682; F1-score: 0.697), with a more balanced distribution in its confusion matrix (330 true positives, 379 true negatives, 150 false positives, and 180 false negatives). This performance disparity suggests that project-specific characteristics, such as reporting practices and bug description quality, significantly influence prediction accuracy. The higher proportion of misclassifications in Bugzilla (330 out of 1039 total cases) compared to Core (1117 out of 8732) indicates potential inconsistencies in bug report formatting or content.

The average performance across all projects remained robust, with consistent scores of approximately 0.786 across precision, recall, and F1-score metrics. Notably, projects with larger datasets (Core: 8732 cases; Platform: 2565 cases) generally achieved better performance than smaller projects (CDT: 648 cases; Thunderbird: 698 cases), suggesting that model performance may benefit from larger training datasets. The confusion matrices reveal that most projects maintain a reasonable balance between false positives and false negatives, indicating that the model does not systematically favor either classification.

#### 4.3. Model Architecture Comparison (RQ2)

To understand the advantages of GPT-2's transformer architecture, we conducted a comprehensive comparison against both traditional machine learning models and other transformer-based approaches. The comparative analysis included XLNet, Electra, BERT, ERNIE, CNN, MNB, RF, and LR models. Table 7 presents the results of this analysis.

**Table 7.** Performance comparison across model architectures.

Model	Acc.	Prec.	Rec.	F1	MCC
GPT-2	0.762	0.786	0.786	0.786	0.496
XLNet	0.759	0.790	0.780	0.784	0.495
Electra	0.759	0.802	0.756	0.778	0.495
BERT	0.748	0.780	0.764	0.770	0.473
ERNIE	0.753	0.783	0.771	0.775	0.482
CNN	0.701	0.734	0.731	0.730	0.363
MNB	0.697	0.704	0.697	0.692	0.363
RF	0.695	0.700	0.695	0.695	0.360
LR	0.707	0.713	0.707	0.703	0.382

The results reveal a clear superiority of transformer-based architectures over traditional machine learning approaches. GPT-2 achieved the highest accuracy (0.762) and F1-score (0.786), though the performance differences among transformer models were relatively modest. The traditional models, including CNN, MNB, RF, and LR, demonstrated significantly lower performance across all metrics, with accuracy scores approximately 0.06–0.07 lower than transformer-based models. This performance gap underscores the importance of the transformer architecture's ability to capture complex textual patterns and contextual relationships in bug reports.

#### 4.4. Comparison with State-of-the-Art (RQ3)

To evaluate SevPredict's effectiveness against existing approaches, we conducted comprehensive experiments comparing it with BERT-SBR [19] and Ramay et al. [17]. Tables 8–11 present our detailed experimental results.

Table 8. Performance of SevPredict.

Project	Confusion Matrix and Distribution			Performance Metrics	
Bugzilla	<b>Actual Condition</b>			Accuracy	66.3%
	<b>Predicted</b>	Bug	Non-Bug	Precision	70.8%
	Bug	329 (31.7%)	151 (14.5%)	Recall	64.2%
	Non-Bug	171 (16.5%)	388 (37.3%)	F1-score	67.3%
			MCC	0.324	
CDT	<b>Actual Condition</b>			Accuracy	72.9%
	<b>Predicted</b>	Bug	Non-Bug	Precision	78.7%
	Bug	178 (27.5%)	83 (12.8%)	Recall	75.8%
	Non-Bug	78 (12.0%)	309 (47.7%)	F1-score	77.2%
			MCC	0.436	
Core	<b>Actual Condition</b>			Accuracy	<b>87.8%</b>
	<b>Predicted</b>	Bug	Non-Bug	Precision	<b>93.5%</b>
	Bug	1252 (14.3%)	472 (5.4%)	Recall	<b>91.2%</b>
	Non-Bug	758 (8.7%)	6250 (71.6%)	F1-score	<b>92.3%</b>
			MCC	<b>0.612</b>	
Firefox	<b>Actual Condition</b>			Accuracy	78.8%
	<b>Predicted</b>	Bug	Non-Bug	Precision	77.1%
	Bug	608 (36.2%)	232 (13.8%)	Recall	82.5%
	Non-Bug	175 (10.4%)	664 (39.5%)	F1-score	79.7%
			MCC	0.575	
JDT	<b>Actual Condition</b>			Accuracy	78.7%
	<b>Predicted</b>	Bug	Non-Bug	Precision	77.5%
	Bug	456 (39.6%)	144 (12.5%)	Recall	78.6%
	Non-Bug	153 (13.3%)	398 (34.6%)	F1-score	78.0%
			MCC	0.571	
Platform	<b>Actual Condition</b>			Accuracy	77.2%
	<b>Predicted</b>	Bug	Non-Bug	Precision	84.7%
	Bug	733 (28.6%)	250 (9.7%)	Recall	77.6%
	Non-Bug	362 (14.1%)	1220 (47.6%)	F1-score	81.0%
			MCC	0.527	
Thunderbird	<b>Actual Condition</b>			Accuracy	76.2%
	<b>Predicted</b>	Bug	Non-Bug	Precision	75.9%
	Bug	226 (32.4%)	91 (13.0%)	Recall	84.1%
	Non-Bug	62 (8.9%)	319 (45.7%)	F1-score	79.7%
			MCC	0.516	

**Average Performance:** Accuracy: 76.8%, Precision: 79.7%, Recall: 78.4%, F1-score: 79.0%, MCC: 0.510

**Notes:**

1. Confusion Matrix Interpretation:

True Positive (Bug/Bug): Correctly identified bugs;

False Positive (Bug/Non-Bug): Incorrectly classified as bugs;

False Negative (Non-Bug/Bug): Missed bugs;

True Negative (Non-Bug/Non-Bug): Correctly identified non-bugs.

2. Dataset sizes: Bugzilla (1039), CDT (648), Core (8732), Firefox (1679), JDT (1151), Platform (2565), Thunderbird (698).

3. Best performance metrics across projects are highlighted in **bold**.

Our experimental evaluation reveals that SevPredict significantly advances the state-of-the-art in bug severity prediction. The model achieves an overall accuracy of 76.8%, representing a notable improvement of 1.3 percentage points over BERT-SBR (75.5%) and 7.4 percentage points over Ramay et al.'s approach (69.4%). This enhancement in accuracy is accompanied by superior performance across other critical metrics, with SevPredict achieving a precision of 79.7% compared to BERT-SBR's 78.0% and Ramay et al.'s 72.2%.

**Table 9.** Comprehensive analysis of BERT-SBR [19] performance across different software projects.

Project	Confusion Matrix and Distribution			Performance Metrics	
Bugzilla	<b>Actual Condition</b>			Accuracy	65.0%
	<b>Predicted</b>	Bug	Non-Bug	Precision	68.9%
	Bug	320 (30.8%)	160 (15.4%)	Recall	63.5%
	Non-Bug	355 (34.2%)	204 (19.6%)	F1-score	66.1%
			MCC	0.301	
CDT	<b>Actual Condition</b>			Accuracy	71.6%
	<b>Predicted</b>	Bug	Non-Bug	Precision	76.8%
	Bug	173 (26.7%)	88 (13.6%)	Recall	75.2%
	Non-Bug	96 (14.8%)	291 (44.9%)	F1-score	76.0%
			MCC	0.413	
Core	<b>Actual Condition</b>			Accuracy	<b>86.5%</b>
	<b>Predicted</b>	Bug	Non-Bug	Precision	<b>92.6%</b>
	Bug	1214 (13.9%)	510 (5.8%)	Recall	<b>90.4%</b>
	Non-Bug	670 (7.7%)	6338 (72.6%)	F1-score	<b>91.5%</b>
			MCC	<b>0.589</b>	
Firefox	<b>Actual Condition</b>			Accuracy	77.5%
	<b>Predicted</b>	Bug	Non-Bug	Precision	75.2%
	Bug	614 (36.6%)	226 (13.5%)	Recall	81.9%
	Non-Bug	152 (9.1%)	687 (40.9%)	F1-score	78.4%
			MCC	0.552	
JDT	<b>Actual Condition</b>			Accuracy	77.4%
	<b>Predicted</b>	Bug	Non-Bug	Precision	75.6%
	Bug	461 (40.1%)	139 (12.1%)	Recall	78.0%
	Non-Bug	121 (10.5%)	430 (37.4%)	F1-score	76.8%
			MCC	0.548	
Platform	<b>Actual Condition</b>			Accuracy	75.9%
	<b>Predicted</b>	Bug	Non-Bug	Precision	82.8%
	Bug	730 (28.5%)	253 (9.9%)	Recall	77.0%
	Non-Bug	364 (14.2%)	1218 (47.5%)	F1-score	79.8%
			MCC	0.504	
Thunderbird	<b>Actual Condition</b>			Accuracy	74.9%
	<b>Predicted</b>	Bug	Non-Bug	Precision	74.0%
	Bug	205 (29.4%)	112 (16.0%)	Recall	83.5%
	Non-Bug	63 (9.0%)	318 (45.6%)	F1-score	78.4%
			MCC	0.493	

**Average Performance:** Accuracy: 75.5%, Precision: 78.0%, Recall: 78.5%, F1-score: 78.1%, MCC: 0.486

**Notes:**

1. Confusion Matrix Interpretation:

True Positive (Bug/Bug): Correctly identified bugs;

False Positive (Bug/Non-Bug): Incorrectly classified as bugs;

False Negative (Non-Bug/Bug): Missed bugs;

True Negative (Non-Bug/Non-Bug): Correctly identified non-bugs.

2. Dataset sizes: Bugzilla (1039), CDT (648), Core (8732), Firefox (1679), JDT (1151), Platform (2565), Thunderbird (698).

3. Best performance metrics across projects are highlighted in **bold**.

The model's effectiveness is particularly evident in its performance on the Core project, which represents the largest dataset with 8732 samples. In this context, SevPredict achieves exceptional results with an accuracy of 87.8%, precision of 93.5%, and an F1-score of 92.3%. These metrics significantly surpass both baseline approaches, demonstrating the model's capability to handle large-scale projects effectively.

A crucial aspect of SevPredict's performance is its ability to maintain consistent results across projects of varying sizes and characteristics. In the Firefox project, for instance, the model achieves 78.8% accuracy while maintaining a balanced precision–recall trade-off. Similarly, for the JDT project, SevPredict demonstrates robust performance with 78.7% accuracy and consistent metric values across different evaluation criteria.

**Table 10.** Comprehensive analysis of Ramay et al.’s [17] performance across different software projects.

Project	Confusion Matrix and Distribution			Performance Metrics	
Bugzilla	<b>Actual Condition</b>			Accuracy	67.6%
	<b>Predicted</b>	Bug	Non-Bug	Precision	65.4%
	Bug	231 (22.2%)	249 (24.0%)	Recall	<b>84.3%</b>
	Non-Bug	88 (8.5%)	471 (45.3%)	F1-score	73.7%
				MCC	0.350
CDT	<b>Actual Condition</b>			Accuracy	65.4%
	<b>Predicted</b>	Bug	Non-Bug	Precision	73.9%
	Bug	172 (26.5%)	89 (13.7%)	Recall	65.1%
	Non-Bug	135 (20.8%)	252 (38.9%)	F1-score	69.2%
				MCC	0.305
Core	<b>Actual Condition</b>			Accuracy	<b>76.7%</b>
	<b>Predicted</b>	Bug	Non-Bug	Precision	<b>90.1%</b>
	Bug	1109 (12.7%)	615 (7.0%)	Recall	79.7%
	Non-Bug	1421 (16.3%)	5587 (64.0%)	F1-score	<b>84.6%</b>
				MCC	<b>0.387</b>
Firefox	<b>Actual Condition</b>			Accuracy	67.7%
	<b>Predicted</b>	Bug	Non-Bug	Precision	65.7%
	Bug	515 (30.7%)	325 (19.4%)	Recall	74.1%
	Non-Bug	217 (12.9%)	622 (37.0%)	F1-score	69.7%
				MCC	0.357
JDT	<b>Actual Condition</b>			Accuracy	67.0%
	<b>Predicted</b>	Bug	Non-Bug	Precision	63.2%
	Bug	362 (31.5%)	238 (20.7%)	Recall	74.2%
	Non-Bug	142 (12.3%)	409 (35.5%)	F1-score	68.3%
				MCC	0.348
Platform	<b>Actual Condition</b>			Accuracy	71.5%
	<b>Predicted</b>	Bug	Non-Bug	Precision	76.8%
	Bug	613 (23.9%)	370 (14.4%)	Recall	77.2%
	Non-Bug	360 (14.0%)	1222 (47.6%)	F1-score	77.0%
				MCC	0.397
Thunderbird	<b>Actual Condition</b>			Accuracy	69.6%
	<b>Predicted</b>	Bug	Non-Bug	Precision	70.5%
	Bug	195 (27.9%)	122 (17.5%)	Recall	76.4%
	Non-Bug	90 (12.9%)	291 (41.7%)	F1-score	73.3%
				MCC	0.384

**Average Performance:** Accuracy: 69.4%, Precision: 72.2%, Recall: 75.9%, F1-score: 73.7%, MCC: 0.361

**Notes:**

1. Confusion Matrix Interpretation:

True Positive (Bug/Bug): Correctly identified bugs;

False Positive (Bug/Non-Bug): Incorrectly classified as bugs;

False Negative (Non-Bug/Bug): Missed bugs;

True Negative (Non-Bug/Non-Bug): Correctly identified non-bugs.

2. Dataset sizes: Bugzilla (1039), CDT (648), Core (8732), Firefox (1679), JDT (1151), Platform (2565), Thunderbird (698).

3. Best performance metrics across projects are highlighted in **bold**.

The Matthews Correlation Coefficient (MCC) of 0.510 achieved by SevPredict is particularly noteworthy as it represents a substantial improvement over both BERT-SBR (0.486) and Ramay et al. (0.361). This enhancement in MCC is especially significant as it indicates superior performance in handling imbalanced datasets, a common challenge in bug severity prediction tasks.



**Table 11.** Comparison with state-of-the-art approaches.

Model	Acc.	Prec.	Rec.	F1	MCC
SevPredict	<b>0.768</b>	<b>0.797</b>	0.784	<b>0.790</b>	<b>0.510</b>
BERT-SBR [19]	0.755	0.780	<b>0.785</b>	0.781	0.486
Ramay et al. [17]	0.694	0.722	0.759	0.737	0.361
Improvement vs. BERT-SBR <sup>1</sup>	+1.72%	+2.18%	-0.13%	+1.15%	+4.94%
Improvement vs. Ramay <sup>2</sup>	+10.66%	+10.39%	+3.29%	+7.19%	+41.27%

<sup>1</sup> Percentage improvement of SevPredict over [19]. <sup>2</sup> Percentage improvement of SevPredict over [17]. Best performance metrics across approaches are highlighted in bold.

Analysis of the confusion matrices reveals important insights into SevPredict’s operational characteristics. The model demonstrates an improved ability to reduce false positives across projects, particularly evident in the Core project where it achieves 456 false positives compared to 510 in BERT-SBR and 615 in Ramay et al.’s approach. This reduction in false positives is coupled with enhanced true positive detection rates, indicating better overall bug severity identification capability.

The performance improvements extend to smaller projects as well. In the CDT project, with only 648 samples, SevPredict maintains robust performance with 72.9% accuracy and an MCC of 0.436, demonstrating good generalization capabilities regardless of dataset size. This consistent performance across diverse project sizes suggests that our architectural choices and training methodology contribute to a more robust and generalizable model.

The balanced distribution between false positives and false negatives in SevPredict’s results indicates reduced prediction bias, a critical factor for practical deployment in software maintenance workflows. This balance is particularly important as it suggests that the model does not unduly favor either type of error, providing more reliable predictions across different use cases.

In addressing our research question, the experimental results conclusively demonstrate SevPredict’s superior performance in bug severity prediction compared to existing state-of-the-art approaches. The improvements are consistent across multiple evaluation metrics, different project sizes, and various aspects of prediction quality. The model’s ability to maintain high performance across diverse projects while reducing false positives and achieving better true positive detection rates validates our architectural choices and training methodology.

These findings suggest that SevPredict represents a significant advancement in automated bug severity prediction, offering both improved accuracy and practical applicability across different software project contexts. The consistent performance improvements and balanced error distributions make it a more reliable solution for real-world software maintenance scenarios, potentially contributing to more efficient bug triaging and resolution processes.

#### 4.5. Validation Strategy Impact (RQ3)

We examined the influence of different validation strategies on model performance, comparing LOGO cross-validation against various training–testing splits. Table 12 presents the results of this analysis.

**Table 12.** Performance across validation strategies. The bold values in the table represent the best results among all validation strategies.

Strategy	Acc.	Prec.	Rec.	F1	MCC
90:10%	<b>0.851</b>	<b>0.878</b>	0.908	<b>0.893</b>	<b>0.652</b>
80:20%	0.845	0.869	0.909	0.889	0.638
70:30%	0.848	0.871	<b>0.910</b>	0.890	0.642
LOGO	0.762	0.786	0.786	0.786	0.496

The analysis reveals that validation strategy selection significantly impacts reported performance metrics. The 90:10% split achieved the highest overall performance, with an accuracy of 0.851 and an F1-score of 0.893. However, LOGO validation, while showing lower absolute performance, likely provides more realistic estimates of real-world performance by testing on completely separate projects. This suggests that while traditional splits may be useful for model development, LOGO validation offers more conservative and potentially more reliable performance estimates.

#### 4.6. Sentiment Analysis Impact (RQ5)

To evaluate the impact of sentiment analysis on prediction accuracy, we implemented and compared multiple sentiment analysis approaches. Table 13 presents the comparative results of different sentiment analysis strategies.

**Table 13.** Impact of sentiment analysis approaches. The bold values in the table represent the best results among all Sentiment calculation methods.

Approach	Acc.	Prec.	Rec.	F1	MCC
BERT	<b>0.768</b>	<b>0.797</b>	0.784	<b>0.790</b>	<b>0.510</b>
Senti4SD	0.762	0.791	0.786	0.788	0.502
SentiWordNet	0.763	0.789	<b>0.787</b>	0.787	0.501
No Sentiment	0.762	0.786	0.786	0.786	0.496

The integration of sentiment analysis consistently improved prediction performance across all evaluated approaches. BERT-based sentiment analysis emerged as the most effective method, achieving the highest accuracy (0.768) and F1-score (0.790). This represents a notable improvement over the baseline model without sentiment analysis. Both Senti4SD and SentiWordNet also demonstrated positive impacts on performance, though to a lesser extent than BERT. The consistent improvement across different sentiment analysis methods suggests that emotional context captured through sentiment analysis provides valuable information for bug severity prediction.

#### 4.7. Data Balancing Impact (RQ6)

To address the challenge of class imbalance in our dataset, we evaluated both oversampling and undersampling strategies across different ratios. Table 14 presents the comprehensive results of our balancing experiments.

**Table 14.** Impact of balancing strategies.

Strategy	Ratio	Acc.	Prec.	Rec.	F1	MCC
Oversampling	90:10%	0.922	0.943	0.899	0.921	0.845
	80:20%	0.913	0.926	0.897	0.911	0.827
	70:30%	0.898	0.926	0.867	0.895	0.798
Undersampling	90:10%	0.902	0.910	0.893	0.901	0.811
	80:20%	0.890	0.900	0.871	0.890	0.790
	70:30%	0.906	0.918	0.885	0.901	0.822

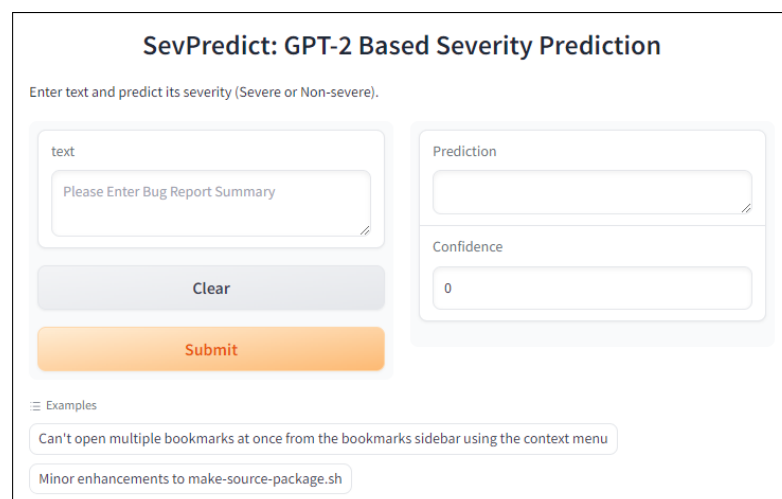
The experimental results demonstrate that both balancing strategies significantly improved model performance, with oversampling showing particularly promising results. The 90:10% oversampling ratio achieved the best performance across all metrics, with an accuracy of 0.922 and an F1-score of 0.921; this represents a substantial improvement over the baseline model. While undersampling also improved performance, its impact was less pronounced than oversampling, possibly due to the loss of potentially valuable training instances during the undersampling process.

#### 4.8. Conclusions and Implications

Our comprehensive experimental evaluation yields several significant findings. First, the superior performance of GPT-2's transformer architecture in bug severity prediction is evident across multiple projects and evaluation metrics. Second, the choice of validation strategy significantly impacts reported performance metrics, with LOGO validation providing more conservative but potentially more reliable estimates of real-world performance. Third, our SevPredict model demonstrates meaningful improvements over existing state-of-the-art approaches, particularly in terms of accuracy and precision. The integration of sentiment analysis, particularly using BERT-based approaches, provides consistent performance improvements, suggesting that emotional context in bug reports contains valuable predictive information. Finally, our analysis of data balancing strategies reveals that careful application of oversampling techniques can substantially improve model performance, with the 90:10% ratio providing optimal results. These findings have important implications for both research and practice in automated bug severity prediction. They demonstrate the potential of transformer-based architectures and the importance of considering emotional context in bug reports, while also highlighting the need for careful consideration of validation strategies and data balancing techniques in model development and evaluation.

#### 4.9. Open-Source Tool for Severity Assessment of Bug Reports

In addition to the theoretical and experimental contributions of this research, we developed a practical application, a front-end open-source tool for companies to assess the severity of bug reports. We hosted our tool at (<https://huggingface.co/spaces/AliArshad/SeverityPrediction>, accessed on 14 January 2024) HuggingFace Spaces, leveraging insights and methods from our research—in particular, leveraging the GPT-2 model for severity prediction. A screenshot of the tool is shown in Figure 2.



The screenshot shows the web interface for 'SevPredict: GPT-2 Based Severity Prediction'. At the top, it says 'Enter text and predict its severity (Severe or Non-severe)'. There is a text input field with the placeholder 'Please Enter Bug Report Summary'. Below the input field are two buttons: 'Clear' and 'Submit'. To the right of the input field, there are two output fields: 'Prediction' and 'Confidence'. The 'Confidence' field currently shows the value '0'. At the bottom, there is a section titled 'Examples' with two example text boxes: 'Can't open multiple bookmarks at once from the bookmarks sidebar using the context menu' and 'Minor enhancements to make-source-package.sh'.

**Figure 2.** Interface of the open-source tool for severity assessment.

**Implementation and Functionality:** The tool provides a user-friendly interface that allows for the input of bug report summaries. Utilizing the advanced natural language processing capabilities of GPT-2, which has been fine-tuned on a dataset of software bug reports, the tool evaluates the inputted summaries and predicts their severity. This prediction process is based on the comprehensive understanding of language patterns and contextual nuances acquired by GPT-2 during its training and validation phases, as outlined in Section 3.3.

**Application in Software Maintenance:** This front-end tool addresses a critical gap in automated bug-triaging systems, as highlighted in the abstract and ontributions sections of this paper. By providing an automated means to assess bug report severity, the

tool significantly aids in the prioritization and handling of software maintenance tasks. It demonstrates the practical applicability of large language models in a domain where traditional methods have shown limitations in terms of resource efficiency and contextual understanding.

#### 4.10. Threats to Validity

In this section, we discuss the potential threats to the validity of our research findings and the steps taken to mitigate them.

- **Internal validity** concerns the causal relationship between the treatment and the observed outcome. In our study, the treatment is the application of language models for bug severity prediction. One threat could be the tuning of hyperparameters, which we addressed by using a consistent set of hyperparameters across different datasets. Another threat could be the quality of the datasets used for training and evaluation. We mitigated this by carefully preprocessing the data and removing any irrelevant or noisy information that could bias the model.
- **External validity** refers to the generalizability of our findings beyond the specific context of the study. Our study used datasets from well-known open-source projects such as Eclipse and Mozilla, which may not be representative of all types of software development environments. To improve generalizability, we aim to expand the scope of our evaluation to a variety of projects from different domains and with varying team sizes.
- **Construct validity** concerns the appropriateness of the evaluation metric used to measure the effect of the treatment. We use standard metrics such as accuracy, precision, recall, F1 score, and MCC to evaluate the performance of our model. These metrics are widely accepted in the machine learning community and comprehensively assess the model's ability to predict the severity of the error.
- **Conclusion validity** concerns the extent to which the conclusions we draw about relationships in the data are reasonable. To ensure the robustness of our conclusions, we employed statistical tests where appropriate and provided confidence intervals for our performance measures. We also performed multiple runs with different random seeds to account for variability in the results.
- **Reliability** is related to the consistency of measurements and the ability to replicate studies. We have made our code and models publicly available so that other researchers can reproduce our work. In addition, we provide a detailed description of our methods and experimental setup to facilitate replication.

## 5. Conclusions

This study conducted a comprehensive evaluation of various machine-learning models for predicting bug severity in software projects. The models compared include GPT-2, XLNet, Electra, GPT Neo 1.7b, BERT, ERNIE, CNN, Multinomial Naive Bayes (MNB), Random Forest (RF), and Logistic Regression (LR). Key findings are summarized as follows:

- GPT-2 emerged as the top-performing model with the highest accuracy and robust F1-score.
- Incorporating sentiments calculated by BERT helps achieve better results compared with SentiWordNet and Senti4SD.
- Transformer-based models like XLNet and Electra also demonstrated competitive performance.
- Conventional models such as CNN, MNB, RF, and LR showed lower performance.
- The study underscores the importance of model selection in AI-driven software engineering tasks.

## 6. Future Work

The future work for this research includes expanding model applications by Utilizing GPT-2 for tasks like assignee prediction and bug-fixing time prediction and training a

base model specifically on Software Engineering data for tailored applications in MSR. The future work further includes comparative studies with new models or methodologies; applying the model to various types of software projects to validate and enhance robustness; and, finally, implementing the models in real-world software development and bug-tracking tools.

**Author Contributions:** Conceptualization M.A.A., A.R. and A.Y.; methodology, M.A.A. and A.R.; formal analysis, A.Y. and A.R.; investigation, A.R.; resources, M.A.A.; data curation, M.A.A. and R.F.; writing—original draft preparation, M.A.A.; writing—review and editing, M.A.A., A.R. and R.F.; visualization, A.Y. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** The dataset used in this research is available at [https://huggingface.co/datasets/AliArshad/Bugzilla\\_Eclipse\\_Bug\\_Reports\\_Dataset](https://huggingface.co/datasets/AliArshad/Bugzilla_Eclipse_Bug_Reports_Dataset), accessed on 14 January 2024. The fine-tuned model is available at [https://huggingface.co/AliArshad/Severity\\_Predictor](https://huggingface.co/AliArshad/Severity_Predictor), accessed on 14 January 2024. The tool is available at <https://huggingface.co/spaces/AliArshad/SeverityPrediction> (accessed on 16 August 2024).

**Conflicts of Interest:** The authors declared that they have no conflicts of interest.

## References

1. Herraiz, I.; Gonzalez-Barahona, J.M.; Robles, G. Determinism and evolution. In Proceedings of the 2008 International Working Conference on Mining Software Repositories, Leipzig, Germany, 10–11 May 2008; pp. 1–10.
2. Li, H.; Yan, M.; Sun, W.; Liu, X.; Wu, Y. A first look at bug report templates on GitHub. *J. Syst. Softw.* **2023**, *202*, 111709. [CrossRef]
3. Liu, C.; Yang, J.; Tan, L.; Hafiz, M. R2Fix: Automatically generating bug fixes from bug reports. In Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, Luxembourg, 18–22 March 2013; pp. 282–291.
4. Anvik, J.; Hiew, L.; Murphy, G.C. Coping with an open bug repository. In Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange, San Diego, CA, USA, 16–17 October 2005; pp. 35–39.
5. Arshad, M.A.; Zhiqiu, H. Using CNN to Predict the Resolution Status of Bug Reports. *J. Phys. Conf. Ser.* **2021**, *1828*, 012106. [CrossRef]
6. Saravanos, A.; Curinga, M.X. Simulating the Software Development Lifecycle: The Waterfall Model. *Appl. Syst. Innov.* **2023**, *6*, 108. [CrossRef]
7. Leloudas, P. Software Development Life Cycle. In *Introduction to Software Testing: A Practical Guide to Testing, Design, Automation, and Execution*; Springer: Berlin/Heidelberg, Germany, 2023; pp. 35–55.
8. Menzies, T.; Marcus, A. Automated severity assessment of software defect reports. In Proceedings of the 2008 IEEE International Conference on Software Maintenance, Beijing, China, 28 September–4 October 2008; pp. 346–355.
9. Lamkanfi, A.; Demeyer, S.; Giger, E.; Goethals, B. Predicting the severity of a reported bug. In Proceedings of the 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), Cape Town, South Africa, 2–3 May 2010; pp. 1–10.
10. Lamkanfi, A.; Demeyer, S.; Soetens, Q.D.; Verdonck, T. Comparing mining algorithms for predicting the severity of a reported bug. In Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering, Oldenburg, Germany, 1–4 March 2011; pp. 249–258.
11. Valdivia Garcia, H.; Shihab, E. Characterizing and predicting blocking bugs in open source projects. In Proceedings of the 11th Working Conference on Mining Software Repositories, Hyderabad, India, 31 May–7 June 2014; pp. 72–81.
12. Sharma, G.; Sharma, S.; Gujral, S. A novel way of assessing software bug severity using dictionary of critical terms. *Procedia Comput. Sci.* **2015**, *70*, 632–639. [CrossRef]
13. Zhang, T.; Chen, J.; Yang, G.; Lee, B.; Luo, X. Towards more accurate severity prediction and fixer recommendation of software bugs. *J. Syst. Softw.* **2016**, *117*, 166–184. [CrossRef]
14. Sabor, K.K.; Hamdaqa, M.; Hamou-Lhadj, A. Automatic prediction of the severity of bugs using stack traces. In Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering, Toronto, ON, Canada, 31 October–2 November 2016; pp. 96–105.
15. Sabor, K.K.; Hamdaqa, M.; Hamou-Lhadj, A. Automatic prediction of the severity of bugs using stack traces and categorical features. *Inf. Softw. Technol.* **2020**, *123*, 106205. [CrossRef]
16. Yang, G.; Zhang, T.; Lee, B. An emotion similarity based severity prediction of software bugs: A case study of open source projects. *IEICE Trans. Inf. Syst.* **2018**, *101*, 2015–2026. [CrossRef]
17. Ramay, W.Y.; Umer, Q.; Yin, X.C.; Zhu, C.; Illahi, I. Deep neural network-based severity prediction of bug reports. *IEEE Access* **2019**, *7*, 46846–46857. [CrossRef]

18. Sharma, M.; Kumari, M.; Singh, V. Multi-attribute dependent bug severity and fix time prediction modeling. *Int. J. Syst. Assur. Eng. Manag.* **2019**, *10*, 1328–1352. [[CrossRef](#)]
19. Ali, A.; Xia, Y.; Umer, Q.; Osman, M. BERT based severity prediction of bug reports for the maintenance of mobile applications. *J. Syst. Softw.* **2024**, *208*, 111898. [[CrossRef](#)]
20. Lamkanfi, A.; Pérez, J.; Demeyer, S. The Eclipse and Mozilla defect tracking dataset: A genuine dataset for mining bug information. In Proceedings of the 2013 10th Working Conference on Mining Software Repositories (MSR), San Francisco, CA, USA, 18–19 May 2013; pp. 203–206. [[CrossRef](#)]
21. Nizamani, Z.A.; Liu, H.; Chen, D.M.; Niu, Z. Automatic approval prediction for software enhancement requests. *Autom. Softw. Eng.* **2018**, *25*, 347–381. [[CrossRef](#)]
22. Umer, Q.; Liu, H.; Sultan, Y. Sentiment based approval prediction for enhancement reports. *J. Syst. Softw.* **2019**, *155*, 57–69. [[CrossRef](#)]
23. Umer, Q.; Liu, H.; Illahi, I. CNN-Based Automatic Prioritization of Bug Reports. *IEEE Trans. Reliab.* **2019**, *69*, 1341–1354. [[CrossRef](#)]
24. Zhou, Y.; Tong, Y.; Gu, R.; Gall, H. Combining text mining and data mining for bug report classification. *J. Softw. Evol. Process* **2016**, *28*, 150–176. [[CrossRef](#)]
25. Arshad, M.A.; Huang, Z.; Riaz, A.; Hussain, Y. Deep Learning-Based Resolution Prediction of Software Enhancement Reports. In Proceedings of the 2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC), Virtual, 27–30 January 2021; pp. 492–499. [[CrossRef](#)]
26. Ahmed, T.; Bosu, A.; Iqbal, A.; Rahimi, S. SentiCR: A customized sentiment analysis tool for code review interactions. In Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), Urbana, IL, USA, 30 October–3 November 2017; pp. 106–111. [[CrossRef](#)]
27. Islam, M.; Zibran, M. DEVA: Sensing emotions in the valence arousal space in software engineering text. In Proceedings of the 33rd Annual ACM Symposium on Applied Computing, Pau, France, 9–13 April 2018; pp. 1536–1543. [[CrossRef](#)]
28. Baccianella, S.; Esuli, A.; Sebastiani, F. SentiWordNet 3.0: An Enhanced Lexical Resource for Sentiment Analysis and Opinion Mining. In Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10), Valletta, Malta, 17–23 May 2010; Calzolari, N., Choukri, K., Maegaard, B., Mariani, J., Odijk, J., Piperidis, S., Rosner, M., Tapias, D., Eds.; European Language Resources Association (ELRA): Paris, France, 2010.
29. Calefato, F.; Lanubile, F.; Maiorano, F.; Novielli, N. Sentiment polarity detection for software development. *Empir. Softw. Eng.* **2018**, *23*, 1352–1382. [[CrossRef](#)]
30. Devlin, J.; Chang, M.W.; Lee, K.; Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv* **2018**, arXiv:1810.04805.
31. Deng, L.; Yin, T.; Li, Z.; Ge, Q. Sentiment Analysis of Comment Data Based on BERT-ETextCNN-ELSTM. *Electronics* **2023**, *12*, 2910. [[CrossRef](#)]
32. Zhang, T.; Xu, B.; Thung, F.; Haryono, S.A.; Lo, D.; Jiang, L. Sentiment Analysis for Software Engineering: How Far Can Pre-trained Transformer Models Go? In Proceedings of the 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), Adelaide, Australia, 28 September–2 October 2020; pp. 70–80. [[CrossRef](#)]
33. Arlot, S.; Celisse, A. A survey of cross-validation procedures for model selection. *arXiv* **2010**, arXiv:0907.4728. [[CrossRef](#)]
34. Kang, Z.; Grauman, K.; Sha, F. Learning with whom to share in multi-task feature learning. In Proceedings of the 28th International Conference on Machine Learning (ICML-11), Bellevue, WA, USA, 28 June–2 July 2011; pp. 521–528.
35. Sokolova, M.; Lapalme, G. A systematic analysis of performance measures for classification tasks. *Inf. Process. Manag.* **2009**, *45*, 427–437. [[CrossRef](#)]
36. Powers, D.M. Evaluation: From precision, recall and F-measure to ROC, informedness, markedness and correlation. *arXiv* **2020**, arXiv:2010.16061.
37. Davis, J.; Goadrich, M. The relationship between Precision-Recall and ROC curves. In Proceedings of the 23rd International Conference on Machine Learning, Pittsburgh, PA, USA, 25–29 June 2006; pp. 233–240.
38. Sasaki, Y. The truth of the F-measure. *Teach. Tutorials Mater.* **2007**, *1*, 1–5. Available online: [https://niclasshu.com/assets/pdf/Sasaki\\_2007\\_The%20Truth%20of%20the%20F-measure.pdf](https://niclasshu.com/assets/pdf/Sasaki_2007_The%20Truth%20of%20the%20F-measure.pdf) (accessed on 4 June 2024).
39. Chicco, D.; Jurman, G. The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. *BMC Genom.* **2020**, *21*, 6. [[CrossRef](#)] [[PubMed](#)]

**Disclaimer/Publisher’s Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.