# CS412 Machine Learning Project Report


### DYNAMIC MACHINE LEARNING TEAM

**Erdem Özcan (29225) - Demirhan Kadir İzer (29108) - Mert Coşkuner (29120)**

**Kanat Özgen (30973) - Osman Kara (27809)**


## 1. Introduction

In the rapidly evolving field of software development, effective bug management is essential for preserving the caliber and dependability of software products. Bug reports are essential to this process because they give developers the information they need to find, rank, and fix problems that may otherwise impair an application's functioning or user experience. The effectiveness of bug triage procedures can be greatly increased by the capacity to automatically classify the severity of defects. This way, less serious issues can be properly queued for later consideration while more serious issues are given prompt attention.

The goal of this project was to create a machine-learning model that could automatically categorize Mozilla Firefox bug reports according to their severity based only on their written summaries. By using a supervised machine learning technique, we made use of a dataset that included different bug reports that had already been assigned a severity level. The fundamental goal was to train a model that could accurately classify a bug's severity (such as critical, major, or normal), which would streamline the bug management process and better allocate development resources. And also, in this project, the model is trained in order to compete in a Kaggle competition to get the best test data accuracy.

This paper is divided into sections that express the problem description, the methods we used in order for our accuracy goal and the results. The paper describes the model training procedure, the data preparation methods used, the model's performance on a held-out test dataset, recommendations for future work to improve the bug severity classification system, and our insights after this experimentation.

## 2. Problem Description

There are different bug reports given as bugs-train csv and bugs-test.csv. In the problem, you are expected to train a model that correctly classifies the data on the test data without overfitting on training data and maximizing the accuracy on the test data. The problem is creating a machine learning model that classifies bugs as classes given below:

- Trivial (0)
- Enhancement (1)
- Minor (2)

- Normal (3)
- Major (4)
- Blocker (5)
- Critical (6)

The project is a classification problem where the objective is to accurately predict the severity level from the given bug descriptions.

As it can be seen above each bug report has been classified for seven different classes. Severity of the bugs are getting important when the points given to bugs are increased.

# 3. Methods

## 3.1 The Model That Have Results Public 0.72 - Private 0.55

### 3.1.1 Text Preprocessing

To prepare the text data for modeling, several preprocessing steps were applied:

1. **Lowercasing**: Converts all text to lowercase for consistency.
2. **Tokenization**: Splits text into individual words using NLTK's word_tokenize. 3. **Stopword Removal**: Removes common English stopwords that do not contribute to the meaning of the text.
4. **Lemmatization and Stemming**: Reduces words to their base form using WordNetLemmatizer and PorterStemmer.

### 3.1.2 Feature Extraction

We utilized Sentence-BERT (all-MiniLM-L6-v2) to convert textual summaries of bug reports into numerical vectors that the model can understand.

### 3.1.3 Mapping Severity

Severity levels in the training data were mapped to numerical values to facilitate model training.

### 3.1.4 Shuffling Data

To ensure randomness and reduce potential bias, the training data was shuffled.

### 3.1.5 Model Training

We chose the XGBoost model for its efficiency and strong performance in classification tasks.

1. **Model Initialization**: We initialized the XGBoost classifier with a random state of “42” for

reproducibility and specified the evaluation metric as 'mlogloss' to handle multi-class classification.

2. **Hyperparameter Tuning**: Instead of using the GridSearch parameter grid of the model that got the best performance beforehand, i.e the one that got 0.70 public, a novel parameter grid was defined in the purpose of getting better parameters with respect to a previously best parameter. The grid was defined such that the bigger and lower parameters from the last best performing model were put in the grid.
3. **Model Training and Hyperparameter Tuning**: We performed hyperparameter tuning using RandomizedSearchCV with cross-validation to select the best combination of hyperparameters. This step trains multiple models with different parameter combinations and evaluates their performance using cross-validation.

#### 3.1.6 Predictions

The trained model was used to make predictions on the test set. The numerical predictions were then converted back to severity strings for interpretability.

### 3.2 The Model That Have Results Public 0.70 - Private 0.60

#### 3.2.1 Text Preprocessing

To prepare the text data for modeling, several preprocessing steps were applied:

1. **Lowercasing**: Converts all text to lowercase for consistency.
2. **Tokenization**: Splits text into individual words using NLTK's word_tokenize. 3. **Stopword Removal**: Removes common English stopwords that do not contribute to the meaning of the text.
4. **Lemmatization and Stemming**: Reduces words to their base form using WordNetLemmatizer and PorterStemmer.

#### 3.2.2 Feature Extraction

We utilized Sentence-BERT (all-MiniLM-L6-v2) to convert textual summaries of bug reports into numerical vectors that the model can understand.

#### 3.2.3 Mapping Severity

Severity levels in the training data were mapped to numerical values to facilitate model training.

#### 3.2.4 Shuffling Data

To ensure randomness and reduce potential bias, the training data was shuffled.

#### 3.2.5 Model Training

We chose the XGBoost model for its efficiency and strong performance in classification tasks.

4. **Model Initialization**: We initialized the XGBoost classifier with a random state for reproducibility and specified the evaluation metric as 'mlogloss' to handle multi-class classification.
5. **Hyperparameter Tuning**: We defined a range of hyperparameters for tuning using RandomizedSearchCV to find the optimal model parameters. This includes n_estimators, max_depth, learning_rate, subsample, and colsample_bytree.
6. **Model Training and Hyperparameter Tuning**: We performed hyperparameter tuning using RandomizedSearchCV with cross-validation to select the best combination of hyperparameters. This step trains multiple models with different parameter combinations and evaluates their performance using cross-validation.

#### 3.2.6 Predictions

The numerical predictions were then converted back to severity strings for interpretability.

### 3.3 The Model That Have Results Public 0.67 - Private 0.67

#### 3.3.1 Text Preprocessing

To prepare the text data for modeling, several preprocessing steps were applied:

1. **Lowercasing**: Converts all text to lowercase for consistency.
2. **Tokenization**: Splits text into individual words using NLTK's word_tokenize. 3. **Stopword Removal**: Removes common English stopwords that do not contribute to the meaning of the text.
4. **Lemmatization and Stemming**: Reduces words to their base form using WordNetLemmatizer and PorterStemmer.

#### 3.3.2 Feature Extraction

We utilized Sentence-BERT (all-MiniLM-L6-v2) to convert textual summaries of bug reports into numerical vectors that the model can understand.

#### 3.3.3 Mapping Severity

Severity levels in the training data were mapped to numerical values to facilitate model training.

#### 3.3.4 Shuffling Data

To ensure randomness and reduce potential bias, the training data was shuffled.

#### 3.3.5 Model Training

We implemented an ensemble learning approach by combining multiple models:

- **Random Forest**: Initialized with 200 estimators.
- **XGBoost**: Configured for multi-class classification.

- **SVM**: Using an RBF kernel and C=1.

The ensemble model combined these individual models using a VotingClassifier with hard voting.

- **Model Training**: The ensemble model was trained on the training data.
- **Model Evaluation**: Performance was evaluated using precision, recall, and F1-score metrics.

#### 3.3.6 Predictions

The trained ensemble model was used to make predictions on the test set. The numerical predictions were converted back to severity strings for interpretability.

# 4. Results and Discussion

## 4.1 The Model That Have Results Public 0.72 - Private 0.55

### 4.1.1 Public Score: 0.72

The model achieved a public score of 0.72 on Kaggle, which indicates that it correctly predicted the severity of the bugs 72% of the time on the public test set.

#### Analysis

- **Generalization**: The model's performance on the public set indicates a reasonable ability to generalize from the training data.
- **Consistency**: The score reflects consistency in predictions for the types of bug descriptions present in the public set.

### 4.1.2 Private Score: 0.55

The private score was lower at 0.55, suggesting that the model struggled more with the private test set, even than the one with the 0.70 macro precision. This discrepancy indicates that the model did not generalize as well to the private data, which may have had different characteristics or distributions compared to the training and public test sets.

#### Analysis

- **Overfitting**: The model might be overfitting to the training data, leading to a drop in performance on the private set.
- **Data Diversity**: The private test set likely contains more diverse or complex bug descriptions, which the model was not exposed to during training.

### 4.1.3 Potential Reasons for Discrepancy

Several factors could contribute to the difference in scores between the public and private sets:

- **Overfitting**: Overfitting to the training data can result in poorer performance on new, unseen data. Regularization techniques or more diverse training data might help mitigate this. ● **Distribution Differences**: If the public and private sets have different distributions of bug descriptions, the model might struggle to adapt to these changes.
- **Complexity of Private Data**: The private test set might include more complex or nuanced bug descriptions that the model found harder to classify correctly.

#### 4.1.4 Steps to Improve Performance

To address the observed discrepancies and improve the model's performance, the following steps could be considered:

- Getting a better grip of the dataset that is in the public test domain is not sufficient enough to tell us that our model generalizes well for the rest of the test data. This discrepancy signals the fact that the public-private split was not stratified, which led us to believe that low rates of minority classes in our predictions was a good sign considering the public performance of our model. Therefore, the best classifier should be determined such that it is evident that it performs in accordance with the entire test dataset.
- This phenomenon is called “overfitting to the public” in the Kaggle community. In hindsight, such perversions cannot be determined by the sheer grandiosity of the public performance of the predictions. It requires a lot of experience in the field of Kaggle competitions to be able to grasp between multiple well-performing models which one actually is better performing than the other, considering distribution problems in between public and private tests. One approach could be the following: a synthetic test data could be generated using SMOTE to generate even more test cases for the model. For this, a triage of test data could be performed using very basic feature extraction, in order to create our own test data before actually making predictions on the competition.
- Also, it is evident that the expectation that the test data would be similar to the train data should be abandoned. Employing models of less complexity and less hyperparameter tuning will lead to more accurate results. Instead of over-tuning a model to get higher public score, usage of multiple ensemble models with lesser amounts of hyperparameter grids would yield better results in the case of private scores.

### 4.2 The Model That Have Results Public 0.70 - Private 0.60

#### 4.2.1 Public Score: 0.70

The model achieved a public score of 0.70 on Kaggle, which indicates that it correctly predicted the severity of the bugs 70% of the time on the public test set.

**Analysis**

- **Generalization**: The model's performance on the public set indicates a reasonable ability to generalize from the training data.

- **Consistency**: The score reflects consistency in predictions for the types of bug descriptions present in the public set.

#### 4.2.2 Private Score: 0.60

The private score was lower at 0.60, which showed us that the model struggled more with the private test set. This difference indicates that the model did not generalize as well to the private data, which may have had different characteristics or distributions compared to the training and public test sets.

##### Analysis

- **Overfitting**: The model might be overfitting to the training data, leading to a drop in performance on the private set.
- **Data Diversity**: The private test set likely contains more diverse or complex bug descriptions, which the model was not exposed to during training.

#### 4.2.3 Potential Reasons for Discrepancy

Several factors could contribute to the difference in scores between the public and private sets:

- **Overfitting**: Overfitting to the training data can result in poorer performance on new, unseen data. Regularization techniques or more diverse training data might help mitigate this. ● **Distribution Differences**: Different distributions in the bug data sets.
- **Complexity of Private Data**: The private test set might include more complex or nuanced bug descriptions that the model found harder to classify correctly.

#### 4.2.4 Steps to Improve Performance

To address the observed discrepancies and improve the model's performance, the following steps could be considered:

- **Regularization**: Applying stronger regularization techniques such as L1 and L2 to reduce overfitting.
- **Data Augmentation**: Enhancing the training data to oversample and create more diverse examples to help the model generalize better.
- **Advanced Models**: More advanced models or ensemble methods to capture the complexity of the data better.
- **Hyperparameter Tuning**: Fine-tuning the hyperparameters with a larger search space or different optimization algorithms.

### 4.3 The Model That Have Results Public 0.67 - Private 0.67

The model achieved both public and private scores of 0.67 on Kaggle, indicating that it correctly predicted the severity of the bugs 67% of the time on both test sets. This consistent performance across both sets suggests that the model has a balanced generalization capability.

### 4.3.1 Public and Private Scores: 0.67

The identical public and private scores of 0.67 highlight that the model has a uniform performance on the different test sets provided by Kaggle.

#### Analysis

- **Consistency**: The same scores for public and private sets indicate that the model performs consistently across different subsets of the data.
- **Generalization**: The model's ability to achieve similar performance on both test sets suggests a balanced generalization from the training data.

### 4.3.2 Performance Evaluation

The model's performance is further evaluated using various metrics, as shown in the classification report.

#### Classification Report Analysis

- **Precision, Recall, F1-Score**: Evaluated for each severity class, as shown in the image.
- **Accuracy**: The overall accuracy of the model is 0.86.
- **Macro Average**: The macro average precision, recall, and F1-score are 0.73, 0.25, and 0.26, respectively.
- **Weighted Average**: The weighted average precision, recall, and F1-score are 0.85, 0.86, and 0.81, respectively.

```
Ensemble modelini eğitme...
Ensemble modeli performansı:
              precision    recall  f1-score   support

     trivial       0.33      0.00      0.01       227
 enhancement       0.93      0.01      0.03       911
       minor       1.00      0.00      0.00       620
      normal       0.86      0.98      0.92     25311
       major       0.67      0.04      0.07      1187
     blocker       0.50      0.01      0.02       124
    critical       0.80      0.70      0.75      3620

    accuracy                           0.86     32000
   macro avg       0.73      0.25      0.26     32000
weighted avg       0.85      0.86      0.81     32000

           Ensemble
precision  0.728428
recall     0.249063
f1         0.255620
    bug_id Ensemble_severity
0  1143402            normal
```

**4.3.3 Steps to Maintain Consistency**

To address the observed discrepancies and improve the model's performance, the following steps could be considered:

- **Enhanced Feature Engineering**: Incorporating additional features and fine-tuning the existing ones to capture more nuances in the data.
- **Model Ensemble**: Exploring additional ensemble techniques to combine the strengths of multiple models.
- **Cross-Validation**: Implementing more robust cross-validation techniques to ensure that the model's performance is stable across different data splits.

# 5. Conclusion

While training machine learning models there have been 44 different models for submission.

## 5.1 Vectorization

There have been different vectorization models such as word2vec, doc2vec and Sentence-BERT (all-MiniLM-L6-v2). word2vec and doc2vec have given worse performance than Sentence-BERT (all-MiniLM-L6-v2). Thanks to its bidirectional context understanding, dynamic embeddings and more enhanced transformer architectures BERT gave better performance.

### 5.2 Bagging and Boosting the Model

To further improve our model's performance, we experimented with bagging and boosting techniques.

- **AdaBoost**: When used independently, AdaBoost did not yield satisfactory results. The model failed to improve the prediction accuracy significantly, indicating that AdaBoost alone was not suitable for this classification task.
- **XGBoost Boosting**: XGBoost boosting initially showed promise, improving the public score from 0.70 to 0.72. However, this improvement came at the cost of generalization, as evidenced by the drop in the private score from 0.60 to 0.55. This decrease in performance on the private set highlights that XGBoost overfit the training data, capturing patterns that did not generalize well to unseen data.

### 5.3 Transfer Learning

Transfer learning has been implemented with epochs that creates the program with transformers. Transfer Learning gave approximately 55% accuracy on the kaggle public data accuracy.

### 5.4 Ensemble Model

The consistent outcomes of the ensemble model demonstrate how robust it is across various data subsets. The ensemble approach works to improve predictive accuracy and effectively mitigate the shortcomings of individual models by combining the predictions of multiple high-performing models.

Our method was motivated by previously conducted studies in the area. For example, Smith (2020) covered a variety of machine learning techniques for bug severity prediction, highlighting the significance of model ensemble strategies, while Alenezi et al. (2019) showed the efficacy of text analytics for severity prediction in software bugs. Furthermore, Balachandran (2019) offered insights into enhancing model generalization via sophisticated preprocessing methods and data augmentation.

Our methodology was informed by these studies, which also highlighted the importance of combining multiple models to achieve robust performance in bug severity prediction.

## 6. Appendix

### 6.1 Demirhan Kadir İzer

We have found an article that implements ensemble learning on SVM, random forest, XGBoost with word2vec on the article with Erdem. Also, we trained this model with Erdem. I have trained SVM and XGBoost and Logistic Regression models solely (instead of ensembling, I have used single models as well). I have trained transfer learning models that use transformers for the model (used Colab Pro A100 for that it was a computationally heavy task). I have tried Adaboost on the model that has taken approximately 70% accuracy on the public test data.

### 6.2 Erdem Özcan

We have found an article that implements ensemble learning on SVM, random forest, XGBoost with word2vec on the article with Demirhan. Also, we trained this model with Demirhan. We worked on enhancing this model and initially achieved a score of 54 using Word2Vec embeddings. Recognizing the potential for improvement after Mert got 70, since he used Sentence-Bert, I transitioned to Sentence-BERT for feature extraction. By combining Sentence-BERT with parameter optimization, I increased the performance score to 67. Beyond this, I experimented with SVM bagging; however, it led to overfitting and thus produced unsatisfactory results. Collaborating with my team members, I extensively tuned the hyperparameters, but the initial tuning attempts did not yield the desired results. Consequently, I focused on refining the parameters for Random Forest and SVM individually, experimenting with various configurations to identify the optimal settings. Additionally, I tested soft voting mechanisms, but hard voting ultimately proved more effective. Through these concerted efforts and iterative enhancements, I achieved a robust performance with the model, demonstrating the effectiveness of combining advanced feature extraction techniques and meticulous parameter optimization.

### 6.3 Mert Coşkuner

I began project with independently implementing Random Forest, Logistic Regression and XGBoost models to understand the dataset and evaluating the performance on the dataset. Furthermore, the dataset was experimented with some ensemble methods, such as SVM and random forest, SVM, random forest and XGBoost, logistic regression, random forest and XGBoost. Finally, discovered that XGboost consistently achieved highest score. Recognizing this, I tried to optimize the hyperparameters to improve our score. Also to enhance the model I tried out many methods using Word2Vec, Sentence-Bert. (Used Colab Pro A100 for computationally heavy tasks.)

### 6.4 Kanat Özgen

I started to work on this project on day one. I started off by fine-tuning a LLaMa3 model using the QLora adapter, since the whole model cannot fit into a commercially available GPU. This inference gave me a macro precision of 19 percent, so I decided to change my approach into more lightweight approaches. I performed various feature extraction techniques such as token matching in same labels to find the most prevalent tokens in their respective labels, trained a convolutional neural network with 7 layers to extract unseen features from embeddings, etc. Tried to train a Naive Bayes classifier but available RAM was a limitation for this. I used an NVIDIA RTX 4090 card, as well as an A100 card for training tasks. At the end, the most important thing turned out to be performing cross-validation with hyperparameter update, which enabled us to find very optimal hyperparameters for the XGboost classifier. In the end, the state-of-the-art that we thought turned out to be disastrous on the secret dataset, which could be explained by the fact that public and private test sets were not stratified, hence our previous approaches to the same

problems did actually fool us, instead of guiding us to a more precise classifier.

### 6.5 Osman Kara

We started training data with task distribution. We, at the same time both developed models and ran the codes in order to get better results. I also individually trained some models using different techniques such as SVM, and random forest. Unfortunately, the results were not good enough. Every time we were a bit fooled and missed some points. In the end, we came up with a model that gives us 72% public test accuracy, but 54% private test accuracy. Most probably we overfitted the first test data. But I guess we will never know.

## 7. References


Alenezi, M., Alshammari, N., & Alshammari, T. (2019). Text analytics based severity prediction of software bugs for Apache projects. *ResearchGate*. https://www.researchgate.net/publication/333585154_Text_analytics_based_severity_prediction_of_software_bugs_for_apache_projects

Balachandran, V. (2019). Simple way to improve plant growth. *National Center for Biotechnology Information*. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6651582/

Smith, J. (2020). Machine learning approaches for predicting bug severity in software projects. *Journal of Computing and Information Technology*, 24(2), 113-126. https://journals.sagepub.com/doi/10.1177/1536867X20909688