CS412- COURSE PROJECT

Project Members

Bülent Emin Üstün - 27822
Tolga Mert Çalışkan - 29335
Ahmet Eren Çağatay - 29041
Arya Düzenli - 28281
Şevki Aybars Türel - 28238

Date: 05.06.2024

Introduction

In this project, the aim was to develop a machine-learning model to predict the severity of bugs based on their textual descriptions. Accurate prediction of bug severity is crucial for efficient software maintenance, as it enables developers to prioritize the most critical issues, thereby reducing downtime and improving overall software quality.

The problem involves categorizing bug reports into predefined severity levels using natural language processing (NLP) techniques combined with machine learning algorithms. To tackle this problem, several models and methodologies, including logistic regression, XGBoost, random forest, and pre-trained models such as BERT were utilized. Each of these models has its strengths and weaknesses, and the main goal was to identify the most effective approach for the dataset.

During the experimentation phase, two models were identified that outperformed the others in terms of macro precision scores on Kaggle. These models are the stack ensemble model and a neural network-based approach developed using PyTorch. The stack ensemble model combines multiple base models to improve prediction accuracy, leveraging the strengths of each individual model. The neural network model, on the other hand, utilizes deep learning techniques to capture complex patterns in the data.

The stack ensemble model was prioritized due to its superior performance. This approach involved comprehensive data preprocessing, feature extraction using TF-IDF, and training a stacking classifier with Random Forest, XGBoost, and Multi-layer Perceptron (MLP) as base models, with Logistic Regression serving as the meta-learner. This ensemble method allows us to combine the predictive power of multiple models, leading to improved accuracy and robustness.

The neural network model also demonstrated excellent performance. This model employed a neural network architecture with several fully connected layers, implemented using PyTorch. The neural network approach highlights the potential of deep learning in handling NLP tasks, providing another viable solution for bug severity prediction.

Through extensive experimentation and cross-validation, we evaluated these models to ensure their effectiveness and reliability. The results indicated that both models could significantly aid in prioritizing bug fixes, thereby enhancing software maintenance processes.

Problem Description

This project aims to create a machine-learning model that can automatically determine the severity of bugs by analyzing their textual summaries. This involves training the model on a dataset of bug reports, each tagged with a severity level such as enhancement, trivial, normal, minor, major, critical, and blocker. The challenge for the model is to detect patterns and key linguistic indicators that reflect the bug's severity. A notable issue we encountered in the dataset was class imbalance, where some severity categories were significantly underrepresented compared to others. This imbalance can skew the model's learning process, leading to poor performance in minority classes. To address this, and to ensure the textual data is suitable for model training, we implemented several preprocessing steps. These included normalizing the text by removing special characters, converting all text to lowercase, and employing techniques like tokenization and lemmatization. These preprocessing steps help in reducing noise and enhancing the model's ability to learn meaningful patterns from the text.

Methods

The dataset must first be cleaned to complete the project. For this reason, the bug descriptions under the summary column were first arranged so that all words were in lowercase. In this process, unnecessary characters, unreadable characters, and punctuation marks were removed from the dataset to standardize all words. Then, the stopwords with the highest frequency in the dataset (a, an, the, etc.) were removed from the dataset to create much more successful models. Several data cleaning methods were applied in the project and every method has different accuracy for the different models (see. Pictures 1 & 2). To determine which words are used meaninglessly and frequently, the frequencies of the thirty most used words for different severities were determined and visualized (see. Picture 3).

Before the usage of the detailed data cleaning methods, many meaningless letter clusters such as n't , 's , j, etc. were commonly observed. Detailed cleaning data is used to remove these words and give some weight to punctuation. For instance, ! was returned as "negative". However, when the detailed cleaning was applied the models that consisted of neural networks, gave worse results than the data applied with simple data cleaning. Therefore, most of the time simple data-cleaning code is applied.

The NLTK library was used to tokenize words and remove the stopwords. It helped to create tokens that only include meaningful words with their single forms. After that, the words in the summary are returned into vectors to train models more easily by using TfidfVectorizer. It will return a statistical formula according to the relevance of the word in all words. It will give a higher value to the most relevant word that is used commonly in all words. Besides that, the severity feature of the dataset turned into the integers that represent the severity. It is applied for the sake of simplicity. The LabelEncoder class of the sklearn library was used (see picture 4).

Another problem with the dataset is that it is very imbalanced. Therefore, different methods were applied to make the dataset balanced. One of them is SMOTE and another one is related to undersampling and oversampling. However, both of them did not affect the results of the models. When these methods were applied to the dataset, most of the models started to catch more cases with the minority class. However, they are worse in the catching majority class.

In this project, two main approaches were employed to predict bug severity from textual descriptions: a stack ensemble model and a neural network-based model. Each method involves specific data preprocessing steps, feature extraction techniques, and model training processes. Below, a detailed description of the methods used can be seen:

The stack ensemble model included three base models:

- Random Forest: An ensemble learning method based on decision trees, providing robustness and handling non-linear relationships.
- XGBoost: An optimized gradient boosting algorithm, known for its high performance and efficiency.
- Multi-layer Perceptron (MLP): A neural network model with one hidden layer of 100 neurons, capable of capturing complex patterns in the data.

The outputs of the base models were combined using a stacking classifier (see picture 5), with Logistic Regression as the final estimator. This approach leverages the strengths of each base model to improve overall prediction accuracy. To evaluate the model's performance and ensure its generalizability with predefined hyperparameters, 5-fold cross-validation was used (see picture 6). This method splits the dataset into five parts, training the model on four parts and validating it on the remaining part, and repeating this process five times.

The neural network model was built using PyTorch and included the following layers:

- Input Layer: Connected to the TF-IDF features.
- Hidden Layers: Three fully connected layers with ReLU activation and dropout for regularization.
- Output Layer: Producing class probabilities using LogSoftmax activation.

The model was trained using negative log-likelihood loss (NLLLoss) and the Adam optimizer. The training was conducted over 10 epochs with a batch size of 64, ensuring sufficient learning while preventing overfitting.

Some stacking models were applied to obtain better results. In this stacking algorithm LGBM and Random Forest classifier. These two models were chosen because they gave higher accuracy separately. Therefore, if both models join together, it is expected to get higher accuracy. For this reason, a stacking algorithm is used. It will make the last decision by using logistic regression.

Another model is trained by using a transformer-based machine learning model. For this purpose, DistilBERT was also tried because it is faster than BERT. The tokenization process was done by the DistilBERT that is already trained for language models. After that, batch size and epoch values were tried to get a higher accuracy.

Results and Discussion

In our project, we evaluated several machine learning models, including BERT, Logistic Regression, and SVM, to predict bug severity. Our experimentation focused on comparing the performances of a stack ensemble model and a neural network. On the public leaderboard of the Kaggle competition, the stack ensemble model demonstrated superior performance, leading the pack due to its robust generalization from training data to unseen data. Conversely, the neural network performed better on the private leaderboard, indicating a strong ability to handle the complexities of the private test set. (see Picture 7&8 for more information) In conclusion, we decided to proceed with the stacked ensemble as our final model for submission based on its consistent performance on the public leaderboard (see Picture 9), which offered a broader and more immediate validation of its effectiveness.

Appendix

Work done by group members:

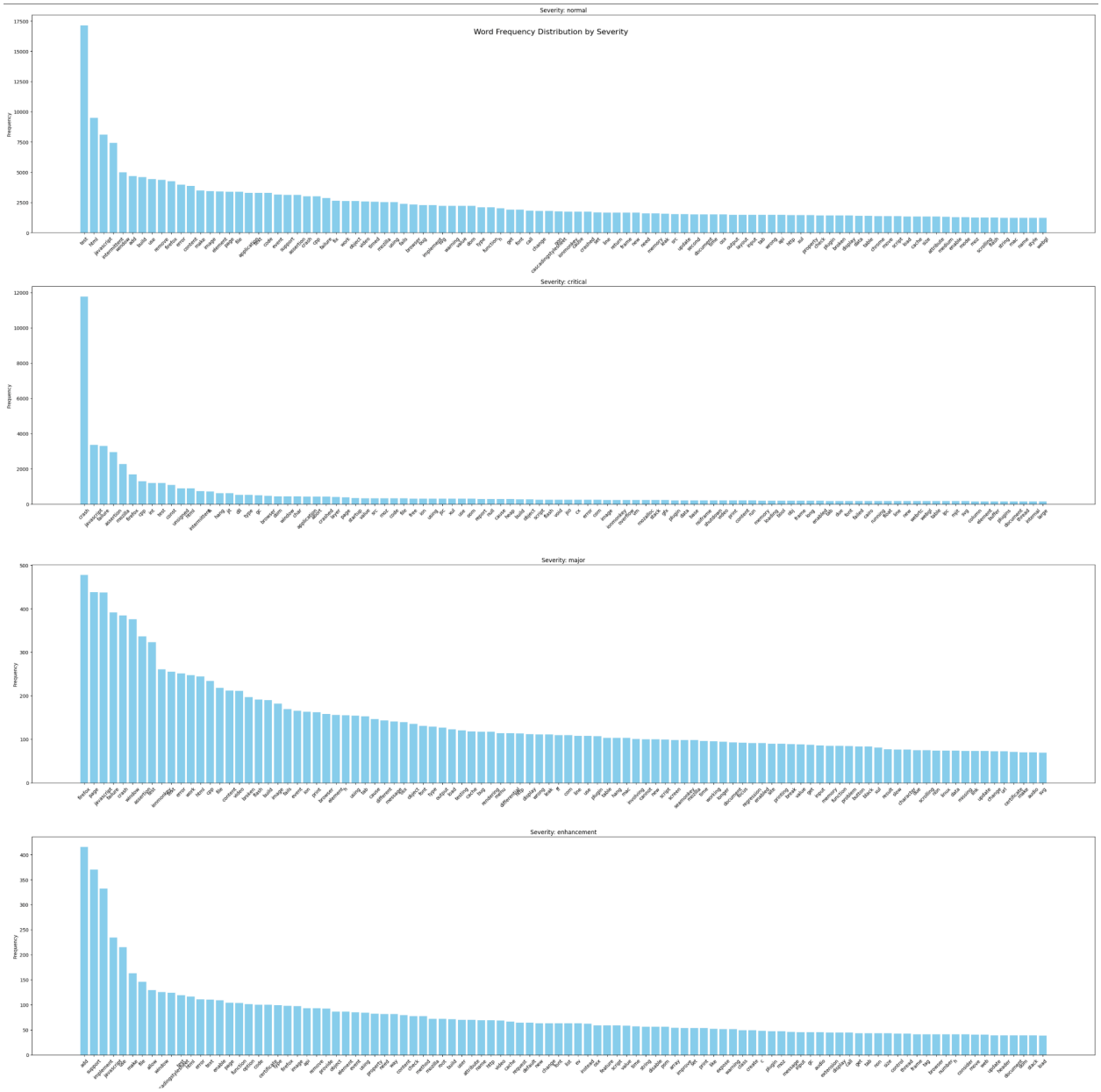
1. Emin Üstün: Implementing the random forest, logistic regression, and neural network model.
2. Şevki Aybars Türel: Implementing and updating the data preprocessing & visualization, implementing the BERT, SVM, and stacking algorithm which stacks LGBM and Random Forest.
3. Tolga Mert Çalışkan: Implementing several models such as XGBoost, Random Forest, stack ensemble with XGBoost as base model and logistic regression as final estimator, stack ensemble with XGBoost, Random Forest and MLP as base models and logistic regression as final estimator.
4. Ahmet Eren Çağatay: Focused on experimenting with pre-trained models, specifically BERT and DistillBERT. Due to high training times, I tested these models with various batch sizes and epoch numbers. Despite these efforts, the results were not satisfactory when considering average macro precision. The performance of other base models outperformed the pre-trained models, which could be attributed to the trade-off between performance and training time.
5. Arya Düzenli: Contributed by implementing the SVM and BERT models.

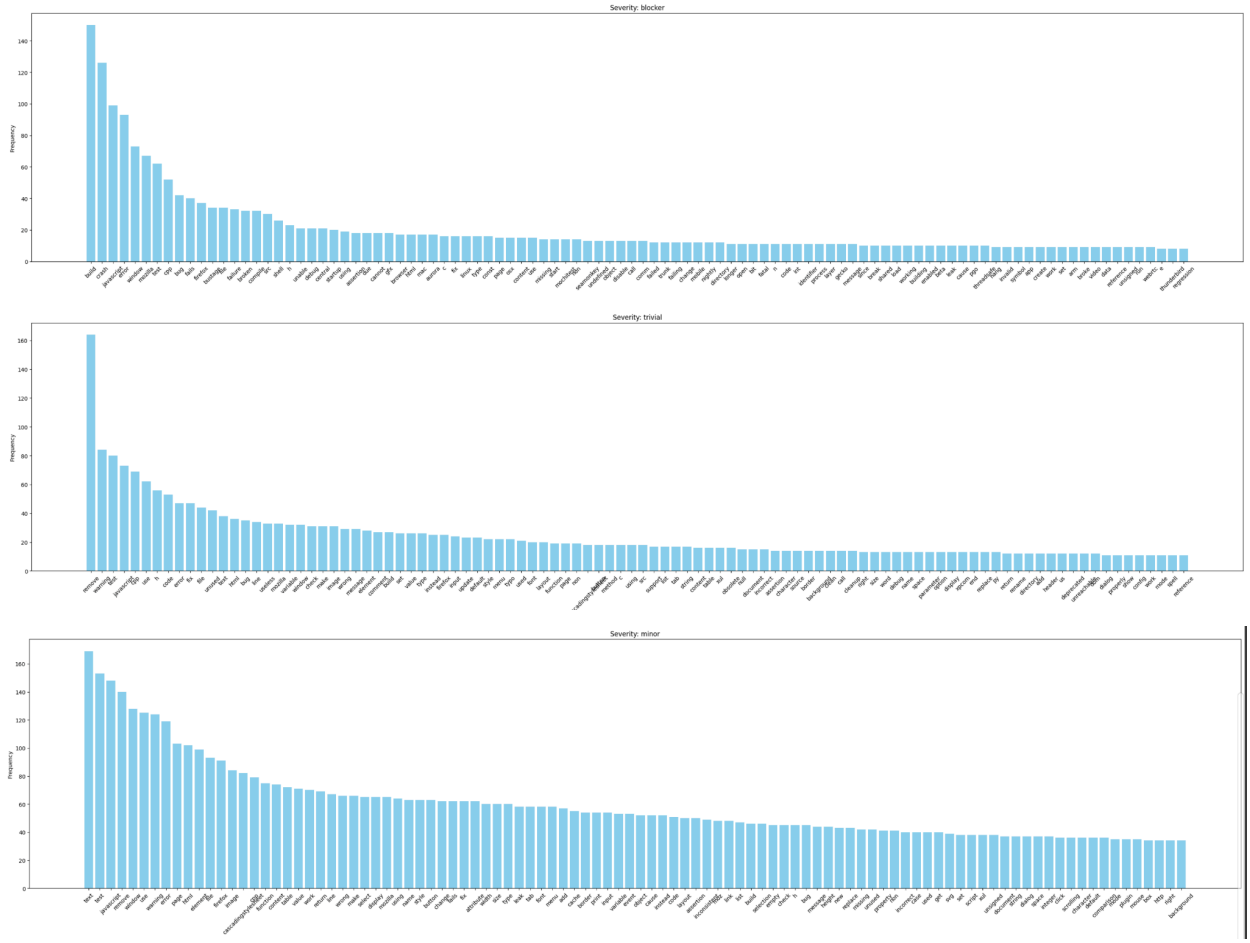
```
def preprocess(text):
    text = text.lower()
    text = re.sub(r'[^\w\s]', '', text)
    tokens = word_tokenize(text)
    filtered_tokens = [word for word in tokens if word.lower() not in english_stop_words
    return ' '.join(filtered_tokens)
```

Picture 1: Simple Data Cleaning

```
def clean_text(text):
    text = re.sub(r'js_', ' javascript ', text)
    text = re.sub(r'Js_', ' javascript ', text)
    text = re.sub(r'JS_', ' javascript ', text)
    text = re.sub(r'@ js', ' javascript ', text)
    text = re.sub(r'@ Js', ' javascript ', text)
    text = re.sub(r'@ JS', ' javascript ', text)
    text = re.sub(r'@ jS', ' javascript ', text)
    text = re.sub(r'\bOS X\b', 'osx', text)
    text = re.sub(r'\bOSX\b', 'osx', text)
    text = re.sub(r'\bOS\b', 'osx', text)
    text = re.sub(r'\bos\b', 'osx', text)
    text = re.sub(r'\bJS\b', ' javascript ', text)
    text = re.sub(r'\bJs\b', ' javascript ', text)
    text = re.sub(r'\bjs\b', ' javascript ', text)
    text = re.sub(r'css', 'cascadingstylesheet', text)
    text = re.sub(r'.css', ' cascadingstylesheet', text)
    text = re.sub(r'CSS', 'cascadingstylesheet', text)
    text = re.sub(r'(\d+)\s*x\s*(\d+)', r' multiplication ', text)
    text = re.sub(r"n't", ' negative', text)
    text = re.sub(r'\d+', ' ', text)
    text = re.sub(r"'s", ' possession', text)
    text = re.sub(r"'", ' ', text)
    text = re.sub(r"!", 'cautious ', text)
    text = re.sub(r"@ ns", 'mozillalibrary ', text)
    text = re.sub(r'/\S*', 'pathname', text)
    text = text.encode('ascii', 'ignore').decode('ascii')
    text = re.sub(r"[^a-zA-Z0-9'\s]", " ", text)
    text = " ".join(text.split())
    return text
```

Picture 2: Detailed Data Cleaning

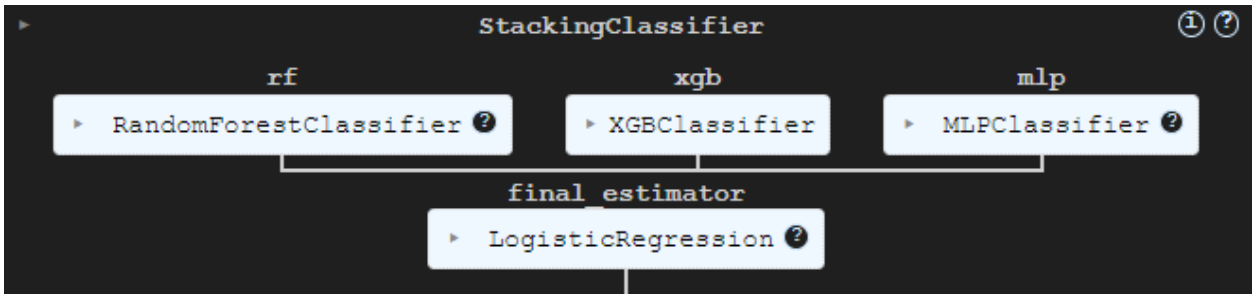

Word Frequency Distribution by Severity
Severity: normal
Severity: critical
Severity: major
Severity: enhancement
Frequency



Picture 3: Frequency Distribution

```
{'blocker': 0,
 'critical': 1,
 'enhancement': 2,
 'major': 3,
 'minor': 4,
 'normal': 5,
 'trivial': 6}
```

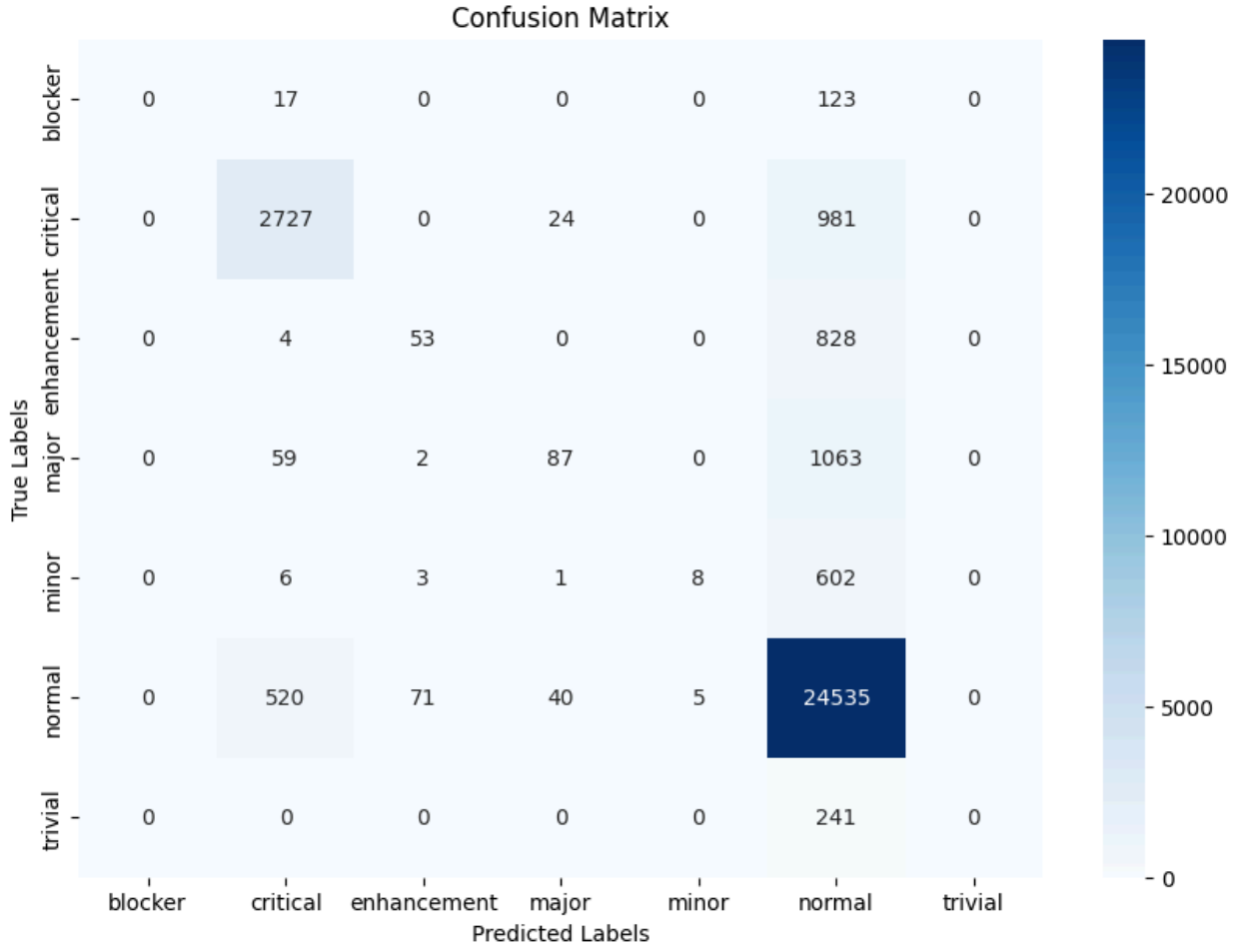
Picture 4: Label Encoding



Picture 5: Demonstration of stack ensemble (MLP, Random Forest, XGBoost)

```
Cross-validation scores (Macro Precision): [0.64874241 0.82958391 0.65864183 0.6740618  0.68667703]
Average cross-validation score (Macro Precision): 0.6995413957429178
```

Picture 6: Cross Validation scores of stack ensemble (MLP, Random Forest, XGBoost)



Picture 7: Confusion Matrix

Picture 8: Private and Public macro precision scores of NN respectively.

Picture 9: Private and Public macro precision scores of stack ensemble respectively.