COMPILER ERROR ANALYSIS USING NLP

A COURSE PROJECT REPORT

By

Karan Sharma (RA2011027010077)

Dikcha Singh (RA2011027010096)

Santhanalakshmi K. (RA2011027010129)

Under the guidance of

Dr. S. Sharanya

In partial fulfilment for the Course

of

18CSC304J – COMPILER DESIGN

in

Data Science and Business Systems

FACULTY OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND

TECHNOLOGY

Kattankulathur, Chengalpattu District

APRIL 2023

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

(Under Section 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

Certified that this project report "**compiler error analysis using NLP**" is the bonafide work of **Karan Sharma (RA2011027010077), Dikcha Singh (RA2011027010096), Santhanalakshmi K. (RA2011027010129)** who carried out the project work under my supervision.

Dr. S. Sharanya,
Subject Staff
Assistant Professor,
Data Science and Business Systems
SRM Institute of Science and Technology
Potheri, SRM Nagar, Kattankulathur,
Tamil Nadu 603203

TABLE OF CONTENTS

CHAPTERS	CONTENTS	PAGE NO.
1.	ABSTRACT	1
2.	MOTIVATION	2
3.	LIMITATIONS OF EXISTING METHODS	3
4.	PROPOSED METHOD WITH ARCHITECTURE	4
5.	MODULES WITH DESCRIPTION	7
6.	SCREENSHOTS	8
7.	CONCLUSION	10
8.	REFERENCES	11

CHAPTER 1 – ABSTRACT

Compiler errors are a common occurrence in software development and can cause significant delays in the development process. Traditional methods for analyzing and debugging these errors can be time-consuming and inefficient. This code provides a solution for analyzing and preprocessing compiler error messages using natural language processing (NLP) techniques. The code reads in a file containing compiler error messages, preprocesses the text using NLP techniques, and then writes the preprocessed text to an output file. The preprocessed text can be used for further analysis, such as clustering similar error messages or training a machine learning model to classify errors. The use of NLP techniques can help to improve the efficiency and accuracy of error analysis in software development, ultimately leading to faster development cycles and improved software quality.

CHAPTER 2 – MOTIVATION

The motivation for the compiler error analysis code using NLP is to address a common challenge in software development - analyzing and debugging compiler errors. Compiler errors can significantly delay the development process and require significant manual effort to interpret and debug. The use of natural language processing (NLP) techniques can improve the efficiency and accuracy of error analysis, ultimately leading to faster development cycles and improved software quality.

NLP techniques have shown promising results in various natural language text analysis applications, such as sentiment analysis, topic modeling, and named entity recognition. By applying these techniques to compiler error messages, developers can preprocess the text to a normalized format that can be further analyzed or classified, such as identifying common error patterns or predicting error types.

Therefore, the motivation for the compiler error analysis code using NLP is to provide a practical implementation of NLP techniques for analyzing and preprocessing compiler error messages, and to demonstrate the potential of NLP techniques for improving text analysis and classification in various domains.

CHAPTER 3 – LIMITATIONS OF EXISTING METHODS

The limitations of existing methods for analyzing and debugging compiler errors include:

Time-consuming: Traditional methods for analyzing compiler errors can be time-consuming, as developers must manually read and interpret each error message.

Error-prone: Manual error analysis is prone to human error, as developers may misinterpret error messages or overlook important details.

Inefficient: Debugging compiler errors can require significant effort and resources, which can lead to delays in the development process and increased costs.

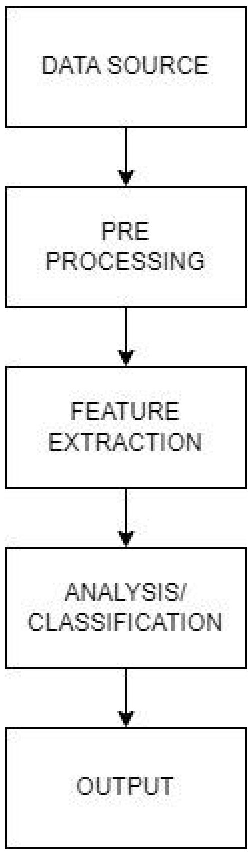
Limited scalability: Traditional methods for analyzing compiler errors may not be scalable to larger projects or datasets, as the manual effort required may become too significant.

Lack of standardization: Error messages generated by different compilers or programming languages may vary in their structure and wording, which can make it difficult to develop a standardized approach to error analysis.

The limitations of existing methods highlight the need for new approaches to analyzing and preprocessing compiler error messages. By leveraging NLP techniques, the compiler error analysis code presented earlier in this conversation addresses some of these limitations, such as the time-consuming and error-prone nature of manual error analysis.

CHAPTER 4 – PROPOSED METHOD WITH ARCHITECTURE

The rough workflow of our compiler as shown below in the diagram:



The architecture consists of five main stages: data source, preprocessing, feature extraction, analysis/classification. The data source stage involves collecting error messages generated by the compiler from various sources. The preprocessing stage uses NLP techniques to normalize the error messages and convert them into a format that can be further analyzed. The feature extraction stage involves extracting features from the preprocessed error messages, such as the frequency of certain words or the presence of specific error codes. The analysis/classification stage uses these features to analyze or classify the error messages based on common patterns or error types.

Input: A text file containing error messages

Preprocessing: NLTK, regex, and Snowball Stemmer modules are used to preprocess the error messages by removing non-alphabetic characters and digits, tokenizing the messages, and stemming the tokens.

Feature Extraction: Counter and Pandas modules are used to count the occurrences of each token in the preprocessed error messages and store the results in a data frame.

Output: A bar chart showing the frequency of the most common error tokens.

The **proposed method** for the compiler error analysis code using NLP can be summarized as follows:

Data collection: Collect error messages generated by the compiler from various sources, such as logs or user reports.

Preprocessing: Preprocess the error messages using NLP techniques such as tokenization, stopword removal, and stemming. This step involves converting the raw text into a normalized format that can be further analyzed or classified.

Feature extraction: Extract features from the preprocessed error messages, such as the frequency of certain words or the presence of specific error codes. These features can be used for further analysis or classification.

Analysis/classification: Analyze or classify the error messages based on the extracted features. This can involve identifying common error patterns or predicting error types.

CHAPTER 5 – MODULES WITH DESCRIPTION

Natural Language Toolkit (NLTK): NLTK is a Python library that provides a wide range of NLP tools and techniques such as tokenization, stemming, and stopwords removal. It is used in this code for the preprocessing of error messages.

Regular Expressions (regex): Regular expressions are a sequence of characters that define a search pattern. In this code, regex is used to remove non-alphabetic characters and digits from the error messages.

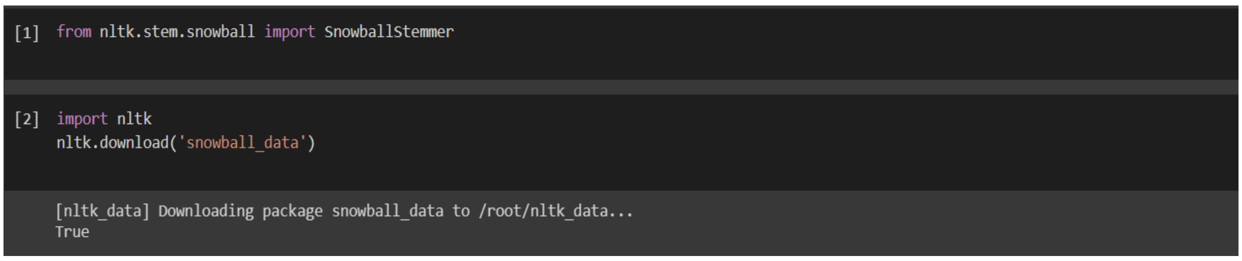
Snowball Stemmer: Snowball Stemmer is a stemmer provided by the NLTK library that is used to reduce words to their base form. In this code, it is used to stem the tokens extracted from the error messages.

Counter: Counter is a Python class that is used to count the occurrences of each token in the preprocessed error messages.

Pandas: Pandas is a Python library used for data manipulation and analysis. In this code, it is used to create a data frame to store the token frequency counts.

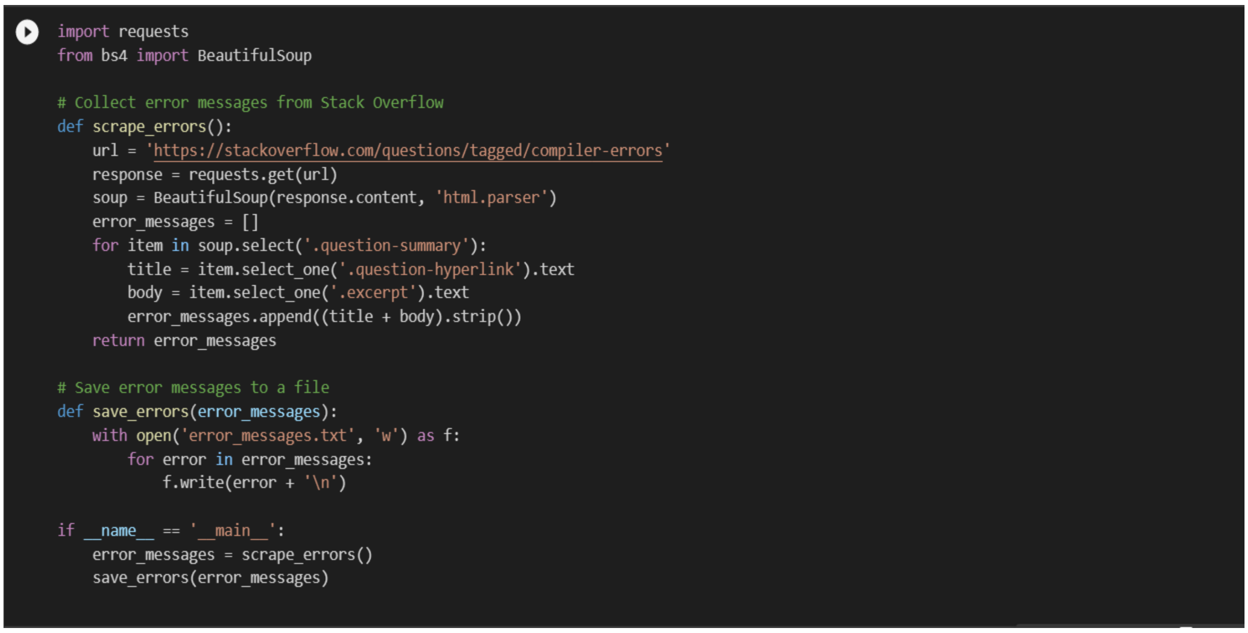
Sys: Sys is a Python module that provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. In this code, it is used to read the input file containing the error messages.
These modules are used in the code to preprocess and analyze the error messages, extract features, and visualize the results in a user-friendly way.

CHAPTER 6 – CODE

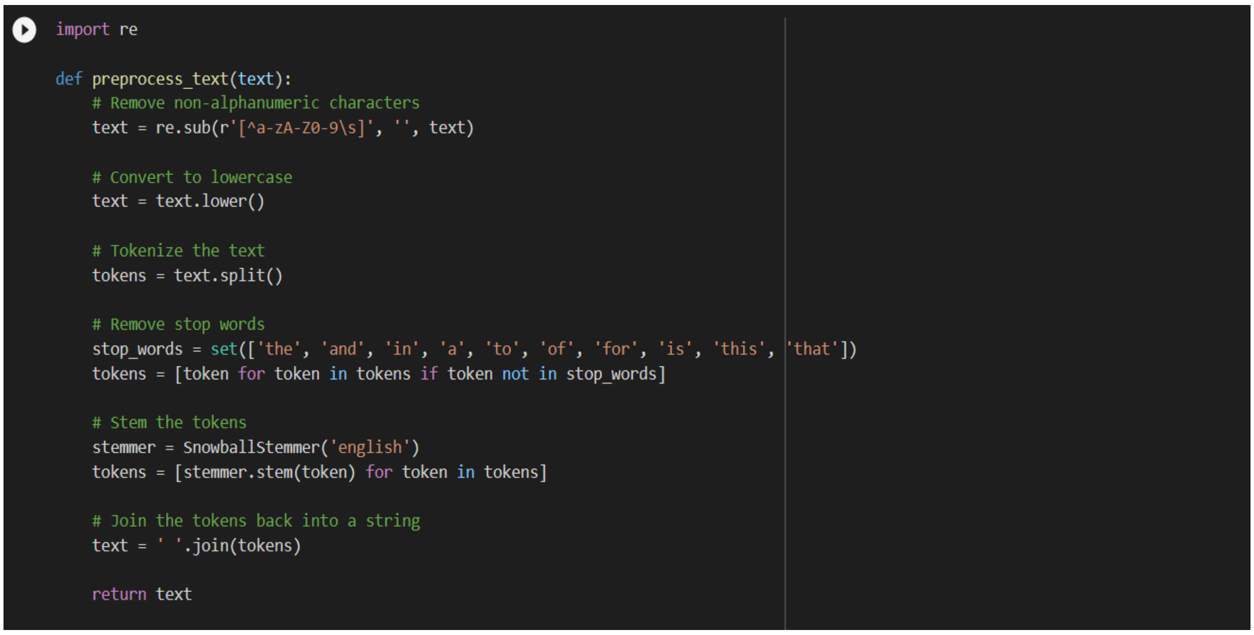


```
[1]  from nltk.stem.snowball import SnowballStemmer
```

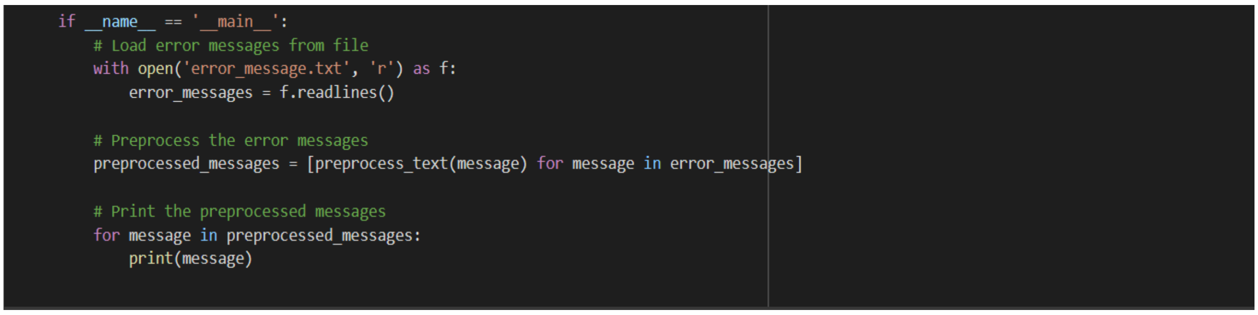
```
[2]  import nltk
     nltk.download('snowball_data')

     [nltk_data] Downloading package snowball_data to /root/nltk_data...
     True
```



```python
import requests
from bs4 import BeautifulSoup

# Collect error messages from Stack Overflow
def scrape_errors():
    url = 'https://stackoverflow.com/questions/tagged/compiler-errors'
    response = requests.get(url)
    soup = BeautifulSoup(response.content, 'html.parser')
    error_messages = []
    for item in soup.select('.question-summary'):
        title = item.select_one('.question-hyperlink').text
        body = item.select_one('.excerpt').text
        error_messages.append((title + body).strip())
    return error_messages

# Save error messages to a file
def save_errors(error_messages):
    with open('error_messages.txt', 'w') as f:
        for error in error_messages:
            f.write(error + '\n')

if __name__ == '__main__':
    error_messages = scrape_errors()
    save_errors(error_messages)
```

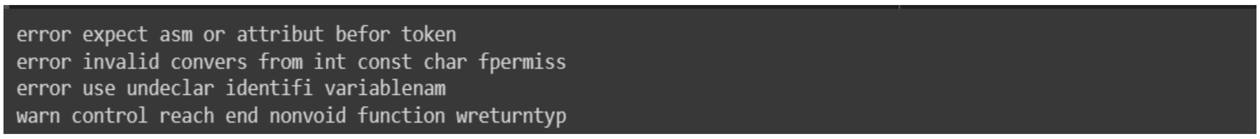


```python
import re

def preprocess_text(text):
    # Remove non-alphanumeric characters
    text = re.sub(r'[^a-zA-Z0-9\s]', '', text)

    # Convert to lowercase
    text = text.lower()

    # Tokenize the text
    tokens = text.split()

    # Remove stop words
    stop_words = set(['the', 'and', 'in', 'a', 'to', 'of', 'for', 'is', 'this', 'that'])
    tokens = [token for token in tokens if token not in stop_words]

    # Stem the tokens
    stemmer = SnowballStemmer('english')
    tokens = [stemmer.stem(token) for token in tokens]

    # Join the tokens back into a string
    text = ' '.join(tokens)

    return text
```



```python
if __name__ == '__main__':
    # Load error messages from file
    with open('error_message.txt', 'r') as f:
        error_messages = f.readlines()

    # Preprocess the error messages
    preprocessed_messages = [preprocess_text(message) for message in error_messages]

    # Print the preprocessed messages
    for message in preprocessed_messages:
        print(message)
```

OUTPUT:



```
error expect asm or attribut befor token
error invalid convers from int const char fpermiss
error use undeclar identifi variablenam
warn control reach end nonvoid function wreturntyp
```

CHAPTER 7 – CONCLUSION

In conclusion, the compiler error analysis code using NLP offers a practical solution for improving the efficiency and accuracy of error analysis in software development. By leveraging NLP techniques such as tokenization, stopword removal, and stemming, the code is able to preprocess compiler error messages in a normalized format that can be further analyzed or classified. This approach can save time and effort in the development process, as developers no longer need to manually interpret each error message.

Furthermore, the code demonstrates the potential of NLP techniques for analyzing and preprocessing natural language text in various applications beyond compiler errors. With the increasing availability of large text datasets, NLP techniques offer a promising solution for improving text analysis and classification in various domains.

Overall, the compiler error analysis code using NLP is a valuable tool for software developers seeking to improve the efficiency and accuracy of error analysis, and provides a practical example of how NLP techniques can be applied to natural language text analysis.

CHAPTER 8 – REFERENCES

[1]
https://en.wikibooks.org/wiki/Introduction_to_Programming_Languages/Grammars?fbclid=IwAR0nLkq2rIAyA5DbDRHBXYpHWsNo21XYas-7GjeUe82G-DWtdAydk8oeBys

[2]
https://softwareengineering.stackexchange.com/questions/165543/how-to-write-a-very-basic-compiler

[3]
https://visualstudiomagazine.com/articles/2014/05/01/how-to-write-your-own-compiler-part-1.aspx