LABORATORY SESSION 4

INTRODUCTION

The aim of this laboratory was to give us an hands on experience to investigate the bug fix commits using structural metrix and similar measure betweem the previous and current versions of the source code.

The structural matric was analyzed using methods of Maintainability Index(MI), Cyclomatic Complexity(CC), and Lines of Code(LOC) for each bugfix. To analyze the similarity we use semantic similarity and token similarity generated by models like codeBERT and BLEU.

We analyzed the results based on code quality improvement (or changes) and the extend of changes. Based on this changes were analyzed to classify the commits into major bug and minor bug fixes, also the conflicting assessments between both the models.

SETUP

- **Software and Tools Used:**
  - Operating System: macOS
  - Text Editor: Visual Studio Code
  - Version Control: git
  - Remote Hosting: GitHub
  - Continuous Integration: GitHub Actions
- **Repositories:**
  - `boxmot` – An open-source library that extends trackers by integrating state-of-the-art multi-object tracking (MOT) algorithms. It provides implementations of popular trackers such as DeepSORT, StrongSORT, and ByteTrack, allowing seamless combination of object detection and tracking. The library is modular, easy to integrate with detection pipelines, and designed for real-time performance, making it suitable for research as well as production-level applications.
- **Python Libraries:**

- Pydriller: A Python framework for mining software repositories. Used to analyze git repositories and extract commit information.
- Pandas: To manage dataframes and dealing with CSV files
- Counter: TO create dictionaries with default values and counting occurenes of elements.
- matplotlib.pyplot: Plotting library
- os: Provides functions to interact with Operating system.
- radon: A code analysis tool to compute software metrics like lines of codes, Maintainability index, Cyclomatic Complexity.
- re: Python's refgular expressions library for string pattern matching and text refinement.
- subprocess: Enables running external commands for interacting with system processes.
- sentences_transformers: A library for semantic similarity check using pretrained transformer model.
- torch: PyTorch deep learnings framework.
- nltk(BLEU): Natural language toolkit, for evaluating BLEU score.

- **Initial Steps:**
  - GitHub account: TejasLohia21
  - Generated CSV from Lab2
  - Virtual Environment named lab3 to manage library versions
  - Configured git username and email
  - Cloned/initialized three repository

Part A and B

0.1 METHODOLOGY AND EXECUTION

Loaded the csv "diff.csv" generated in Lab2 using pandas into a dataframe "df"

- **Compute and report baseline descriptive statistics:**
  - Using ṅunique on the Hashes stored in the df to obtain the unique number of hashes

- Using ṅunique on the Filename stored in the df to obtain the unique number of files.
- Stored all the commits and the file counts for respective commits, and then average reports the average number of modified files per commit.
- To find the fix types from LLM, we use a dictionary to store the kaywords used in commit messages and map them to six generic categories. We run through every commit message and if any word is in dictionary mapping, we return the type of that keyword.
- Used OS and counter libraries to count the top ten most frequently modified files, and extensions.

- **Codes:**

Figure 1: Loading of diff.csv file and finding total number of commits and files.

  - **Explanation of Code for total commits and files:**
    * Loaded the row which stores hashes, and used .nunique to find the total unique hashes.
    * Loaded the row which stores Filename, and used .nunique to find the total unique files modified.
  - **Results:**
    * Total number of hashes in the dataframe are **727**, while the unique number of hashes are **388**.
    * Total number of files in the dataframe are **727**, while the unique number of files which have been modified are **247**.

- **Codes:**
  - **Explanation of Code for Average file modifications per commit:**
    * For every commit, maintain a dictionary which maps it to a list of files which have been modified.

Figure 2: Average numbers of modified files per commit.

  * Iterated thru all the rows, as each row contained information about changes in distinct modifications in files.
  * Sum of all the values in dictionary divided by the length of dictionary yielded the average number of files modified per commit.

– **Results:**

  * Average number of files modified per commit are **1.8737**.

- **Codes:**

```python
import pandas as pd
from collections import defaultdict;

fix_types = {
    "add": ["add", "create"],
    "update": ["update", "upgrade", "refactor", "restructure", "clean", "optimize", "improve", "enhance", "performance"],
    "remove": ["remove", "delete", "drop"],
    "fix": ["fix", "bug", "error", "correct"],
    "docs": ["docs", "documentation", "readme"],
    "test": ["test", "tests", "config", "build"]
}
mappings = {s2: m for m, s in fix_types.items() for s2 in s}
for m, s in fix_types.items():
    for s2 in s:
        mappings[s2] = m


def classify(commit_message):
    words = commit_message.lower().split()
    for word in words:
        if word in mappings:
            return [mappings[word]]
    return []

def counttype(df, column):
    count_lst = defaultdict(int)
    for msg in df[column].dropna():
        typematch = classify(msg)
        for t in typematch:
            count_lst[t] += 1
    return dict(count_lst)

counts = counttype(df, "LLM Inference")
print("Fix type distribution:", counts)
```

```
Fix type distribution: {'update': 497, 'add': 191, 'fix': 37, 'remove': 1}
```

Figure 3: Distribution of number of fix types.

– **Explanation of Code for Fix types distribution:**

  * As the commit messages with same fix type may have different keywords, initialized a dictionary fix_types which contains mapping from six keywords to list of words which contain frequently used keywords for that fix type.

* Created another dictionary, mappings which contains map from each of the word in list to the fix type they belong to.
* classify function iterates through all the words in each commits, and returns the type using the dictionary mapping.
* counttpe function iterates through each of the commit and calls for the function classify and increments the count of the type returned by function classify.

– **Results:**

* Distribution for the fix type is:
* update – 497
* add – 191
* fix – 37
* remove – 1

- **Codes:**

```python
import os
import matplotlib.pyplot as plt
from collections import Counter

def extret(filename):
    return os.path.splitext(filename)[1] if "." in filename else "no_ext"

def freqmax(df, column="filename", top_n=10):
    files = df[column].dropna().tolist()
    file_counts = Counter(files)
    ext_counts = Counter(extret(f) for f in files)

    return file_counts.most_common(top_n), ext_counts.most_common(top_n)


file_counts, ext_counts = freqmax(df, column="Filename")

print("📁 Top modified filenames:")
for f, c in file_counts:
    print(f"{f} → {c}")

print("\n📝 Top modified extensions:")
for e, c in ext_counts:
    print(f"{e} → {c}")
```

Figure 4: Most frequently modified filenames/extensions.

– **Explanation of Code for Most frequently modified filenames/extensions:**

* df is passed to the function freqmax which extracts the column Filename to extract extensions using another function extret.

    * freqmax function applies the instantiates the counter object over files to find the top_n number of files, and extret function returns the extension for all the files.
    * Another counter object is instantiated to create a count for all the extensions
- **Most modified extensions:**
    * Distribution for the fix type is:
    * .py – **699**
    * .yml – **14**
    * .yaml – **6**
    * .toml – **2**
    * .txt – **1**
    * .gz – **1**
    * .lock – **1**
    * .md – **1**
- **Frequently modified files:**
    * Distribution for the fix type is:
    * tracking/val.py – **40**
    * track.py – **32**
    * boxmotutils___init___ṗy – **24**
    * val.py – **14**
    * boxmottrackersbotsortbot_sortṗy – **14**
    * boxmotappearancereid_exportṗy – **13**
    * trackingevolve.py – **13**
    * ġithubworkflowsciẏml – **11**
    * boxmotappearancereid_multibackendṗy – **11**
    * boxmottrackersbyteTrack_trackerṗy – **11**

Plots

- **Fix_type distribution**

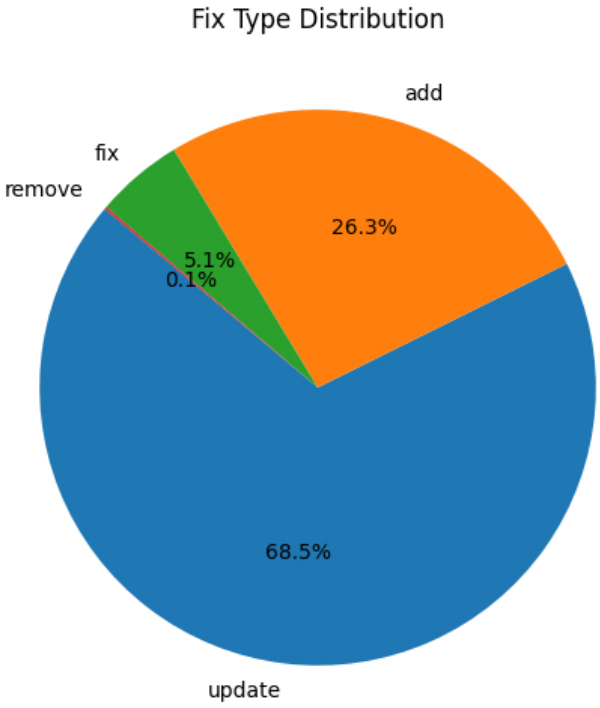

Figure 5: Fix_type distribution.

- **frequently modified files**

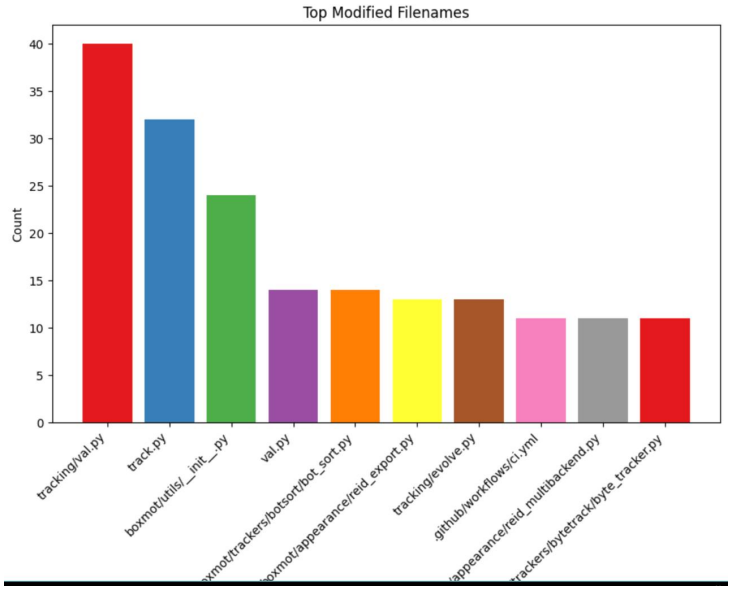

Figure 6: Top modified filenames.

- **frequently modified extensions**

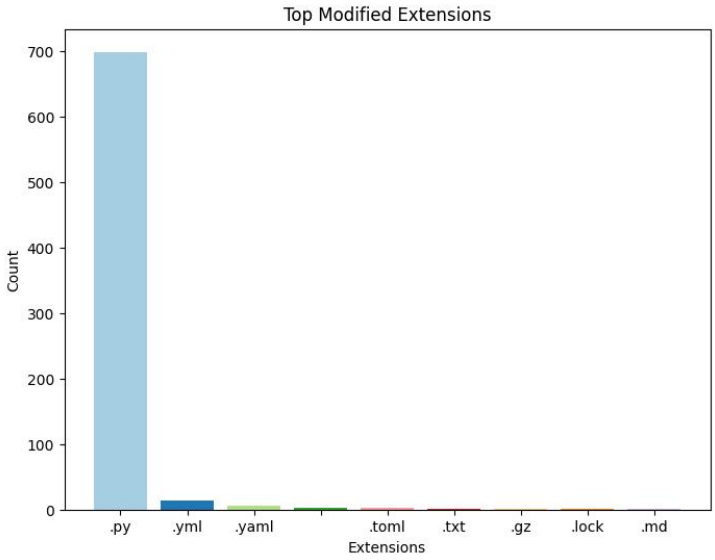

Figure 7: Top modified filenames.

Part C:

0.2 METHODOLOGY AND EXECUTION

Loaded the csv "diffprocessed.csv" generated in Lab2 using pandas into a dataframe "df"

- **Structural Metrics with radon:**
  - Three types of results are obtained in Structural metric.
  - Maintainability Index: Measure how well the code is structured in terms of easiness to understanding the code, modifications and extension of the code. It compresses static code properties on a scale of 0100 to obtain a score. This method uses parameters; Size of Code, Control flow Complexity and token level volume.
  - Cyclomatic Complexity: Measures the Complexity of control flow of the code. It counts the number of independent execution path through a piece of code. The higher the Complexity, implies more the number of paths, making it difficult to test and debug.
  - Lines of Code (LOC): It counts the number of lines present in the source code.
  - Change in MI: MI tells whether the bug-fix commit made the code easier to interpretable.
  - Change in Lines of Code: Indicates the Change in the lines of code, and this is the simplest parameter used for classification, as changes in few lines of code indicate minor changes.
  - Cyclomatic Complexity: Increase in the CC indicates, that the number of paths have increased making the code complex to maintain.

- **Codes:**

```python
import numpy as np
from radon.metrics import mi_visit, analyze
from radon.complexity import cc_visit

csv_path = "diff.csv"
df = pd.read_csv(csv_path)

def radonmet(source_code):
    if not isinstance(source_code, str) or not source_code.strip():
        return np.nan, np.nan, np.nan
    cleaned_code = source_code.encode('ascii', 'ignore').decode('utf-8')
    try:
        loc = analyze(cleaned_code).sloc
        mi = mi_visit(cleaned_code, multi=True)
        cc_results = cc_visit(cleaned_code)
        total_cc = sum(block.complexity for block in cc_results)
        return mi, total_cc, loc
    except Exception:
        return np.nan, np.nan, np.nan

metrics_before = []
metrics_after = []

for index, row in df.iterrows():
    code_before = str(row["Source Code (prev)"])
    code_after = str(row["Source Code (current)"])
    metrics_before.append(radonmet(code_before))
    metrics_after.append(radonmet(code_after))
```

Figure 8: Loading of diff.csv file and finding total number of commits and files.

```python
df[['MI_Before', 'CC_Before', 'LOC_Before']] = metrics_before
df[['MI_After', 'CC_After', 'LOC_After']] = metrics_after


df['MI_Change'] = df['MI_After'] - df['MI_Before']
df['CC_Change'] = df['CC_After'] - df['CC_Before']
df['LOC_Change'] = df['LOC_After'] - df['LOC_Before']



display_cols = [
    'Hash', 'Filename', 'MI_Before', 'MI_After', 'MI_Change',
    'CC_Before', 'CC_After', 'CC_Change',
    'LOC_Before', 'LOC_After', 'LOC_Change'
]
if 'Filename' not in df.columns:
    display_cols.remove('Filename')

print(df[display_cols].to_string())
metric_cols = ['MI_Before', 'CC_Before', 'LOC_Before', 'MI_After', 'CC_After','LOC_After', 'MI_Change', 'CC_Change', 'LOC_Change']
cleaned_df = df.dropna(subset=metric_cols)
output_filename = 'diffprocessed.csv'
cleaned_df.to_csv(output_filename, index=False)
```

Figure 9: Loading of diff.csv file and finding total number of commits and files.

– **Explanation of the code:**

  * Loaded the dataframe processed in the previous question to extract the source code before commit and the source code after commit.
  * Function `radonmet` which filters the code and asserts the code to be a python instance. Function calls for inbuilt functions from the radon library to extract Lines of Code (LOC), Maintainability Index (MI) and Cyclomatic Complexity (CC).
  * Code iterates through all the rows in the loaded dataframe, and uses the defined function `radonmet` to obtain Maintainability index, Cyclomatic Complexity and Lines of Code respectively.
  * We call this function on both the source code before and source code after the commit and store them in lists.
  * In the main dataframe, six columns are added for Maintainability Index, Cyclomatic Complexity and Lines of Code before and after respectively.
  * Three more columns are appended in the dataframe which store the difference between MI, CC and LOC.
  * Because of the presence of some characters which the csv files are not able to store properly, radon returns errors which are handled as an exception in the `radonmet` function by returning a NaN value. These rows are filtered out to avoid processing errors further.

– **Results:**

  * CSV file was updated with addition of 9 columns
    · Average of Maintainability Index before: 55.32
    · Average of Maintainability Index after: 55.06
    · Average of Cyclomatic Complexity before: 40.46
    · Average of Cyclomatic Complexity after: 40.57
    · Average of LOC before: 178.80
    · Average of LOC after: 181.20
    · Average MI change: -0.26
    · Average CC change: +0.12
    · Average LOC change: +2.39

Part D:

0.3 METHODOLOGY AND EXECUTION

Loaded the csv "diffprocessed.csv" generated in previous section.

- **Change Magnitude Metrics:**
  - Bilingual Evaluation Understud (BLEU) – Its a metric designed for machine translation evaluation. BLUE provides an evaluation score to find textual similarity between two sequences.
  - **CodeBERT** – A pre-trained model for source code and natural language, used to compute semantic similarity between two code snippets.
  - How its an Evaluation Method: – Similarity score between source code before and after the commit and tokens of codes give information about the extend of change in the files modified in the commit.
  - classification – Similarity score between codes and token could be set as thresholds for classification of the commit into major and minor change category.
- **Codes:**
  - **Explanation of the code:**
    - The CodeBERT model from HuggingFace's `transformers` library is used to compute semantic similarity between the `Source Code (prev)` and `Source Code (current)`.
    - Each source code snippet is tokenized and passed through CodeBERT to generate embeddings.
    - Cosine similarity is then calculated between the embeddings of the two code snippets, and this value is stored in a new column `CodeBERT_Similarity`.
    - In cases where the code before or after the commit is missing (such as additions or deletions), a similarity score of 0.0 is assigned.
    - Similarly, the BLEU model is applied as a complementary metric. The `BLEU_Similarity` column is appended to the dataframe to capture token-level similarity.
    - The code iterates over each row, splits the source code into tokens, and applies the `sentence_bleu` function from NLTK with smoothing to compute a BLEU score.

* If either the previous code or the current code is missing, a BLEU score of 0.0 is assigned to handle such cases gracefully.
* Thus, the two models provide different perspectives: CodeBERT for semantic similarity (meaning of code), and BLEU for token similarity (surface-level code overlap).

```python
import pandas as pd
import nltk
from nltk.translate.bleu_score import sentence_bleu, SmoothingFunction
nltk.download("punkt")

csv_path = "diffprocessed.csv"
df = pd.read_csv(csv_path)



df["BLEU_Similarity"] = 0.0

smooth_fn = SmoothingFunction().method1

for i, row in df.iterrows():
    prev_code = str(row["Source Code (prev)"])
    curr_code = str(row["Source Code (current)"])

    prev_tokens = prev_code.split()
    curr_tokens = curr_code.split()

    if prev_tokens and curr_tokens:
        bleu = sentence_bleu([prev_tokens], curr_tokens, smoothing_function=smooth_fn)
    else:
        bleu = 0.0

    df.at[i, "BLEU_Similarity"] = bleu

    if i % 25 == 0:
        print(f"{i}/{len(df)} processed...")

out_csv = "diffprocessed.csv"
df.to_csv(out_csv, index=False)
print("done ->", out_csv)
```

Figure 10: Applications of BLUE Model over source code before and after.

```python
import pandas as pd
import torch
from transformers import RobertaTokenizer, RobertaModel
from sklearn.metrics.pairwise import cosine_similarity

csv_path = "diffprocessed.csv"
df = pd.read_csv(csv_path)

tokenizer = RobertaTokenizer.from_pretrained("microsoft/codebert-base")
model = RobertaModel.from_pretrained("microsoft/codebert-base")
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
model.eval()

df["CodeBERT_Similarity"] = 0.0

def get_codebert_embedding(text):
    inputs = tokenizer(text, return_tensors="pt", truncation=True, padding=True, max_length=512)
    inputs = {k: v.to(device) for k, v in inputs.items()}
    with torch.no_grad():
        outputs = model(**inputs)
        embedding = outputs.last_hidden_state[:, 0, :].cpu().numpy()
    return embedding

for i, row in df.iterrows():
    prev_code = str(row["Source Code (prev)"])
    curr_code = str(row["Source Code (current)"])
    if prev_code.strip() and curr_code.strip():
        prev_emb = get_codebert_embedding(prev_code)
        curr_emb = get_codebert_embedding(curr_code)
        similarity = cosine_similarity(prev_emb, curr_emb)[0][0]
    else:
        similarity = 0.0
    df.at[i, "CodeBERT_Similarity"] = similarity
    if i % 25 == 0:
        print(f"{i}/{len(df)} processed...")

out_csv = "diffprocessed.csv"
df.to_csv(out_csv, index=False)
```

Figure 11: Applications of CodeBERT Model over source code before and after.

– **Results:**

  * Average BLEU_similarity score is 92.76%.

  * Average CodeBERT_similarity between source code tokens 99.95%.

Part E:

0.4 METHODOLOGY AND EXECUTION

Loaded the csv `diffprocessed.csv` generated in previous section.

- **Classification and Agreement:**
  - Classification of commits is important for maintainance and risk assessments.
  - **Classification:** Commits can be classified as a major commit, which implies significant changes, while a minor commit, which implies small changes.
  - Commits which are classified as major commit need more testing for maintainance purposes, thus making this important.
  - Method: Once the similarity values between the previous code and the current code is obtained, certain thresholds need to be set for classification tasks.
  - To find an approximate THRESHOLD value, code plots the similarity vs frequency histogram and chooses the optimal value of 0.97 for BLEU score and CodeBERT model.
  - The `Classes_Agree` column is introduced to check whether the classifications from BLEU similarity and CodeBERT similarity are consistent.
  - If both methods give the same label, it is marked as "YES", otherwise it is marked as "NO", thus helping to analyze the alignment between semantic and token-level classifications.

- **Codes:**

```python
import matplotlib.pyplot as plt
temp = pd.read_csv("diffprocessed.csv").copy()
col = temp["BLEU_Similarity"]
plt.figure(figsize=(8,5))
plt.hist(col, bins=30, edgecolor="black", alpha=0.7, density=True, label="Frequency")
col.plot(kind="kde", color="red", label="Density Curve")
plt.xlabel("BLEU Similarity Score")
plt.ylabel("Frequency (Density)")
plt.title("Distribution of BLEU Similarity Scores")
plt.legend()
plt.grid(alpha=0.3)
plt.show()
```

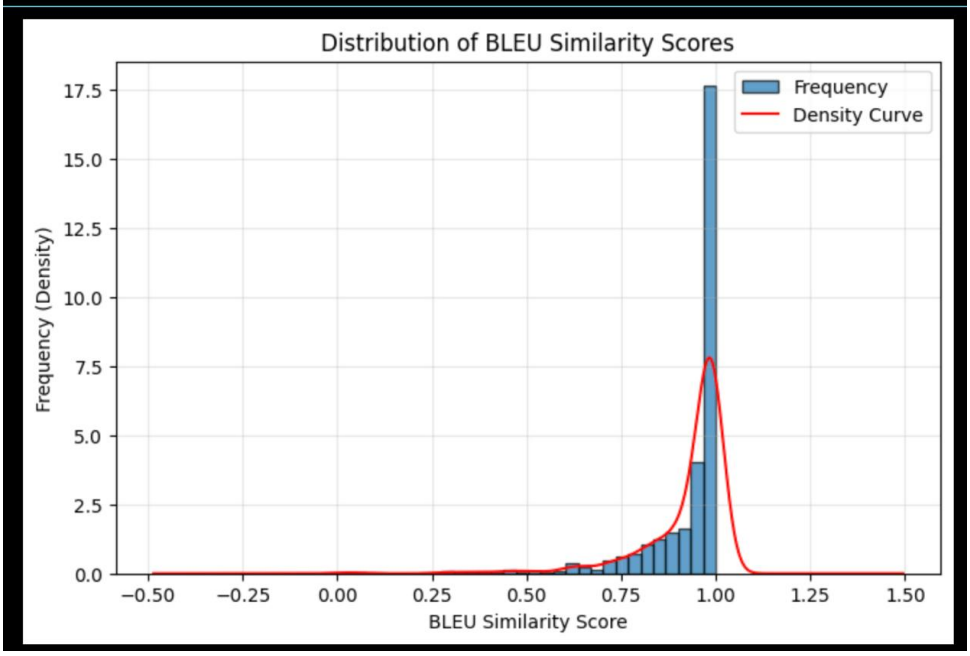

Figure 12: Applications of CodeBERT Model over source code before and after.

```python
import pandas as pd

df = pd.read_csv("diffprocessed.csv").copy()
cutoff = 0.925
df["Semantic_class"] = df["BLEU_Similarity"].apply(lambda x: "High" if x >= cutoff else "Low")
df["Token_class"] = df["CodeBERT_Similarity"].apply(lambda x: "High" if x >= cutoff else "Low")
df.to_csv("diffprocessed.csv", index=False)
```

Figure 13: Applications of CodeBERT Model over source code before and after.

```
import pandas as pd


df = pd.read_csv("diffprocessed.csv")
df["Classes_Agree"] = df.apply(
    lambda row: "YES" if row["Semantic_class"] == row["Token_class"] else "NO",
    axis=1
)
df.to_csv("diffprocessed.csv", index=False)

print(df[["Semantic_class", "Token_class", "Classes_Agree"]].head())
print(f"Number of commits which are major according to CodeBERT Model {(df["Semantic_class"] == "High").count()}")
print(f"Number of commits which are major according to Bleu Tokens {(df["Token_class"] == "High").count()}")
print(f"Number of commits where methods agree {(df["Classes_Agree"] == "YES").count()}")
```

✓ 0.7s

```
  Semantic_class Token_class Classes_Agree
0           High        High           YES
1           High        High           YES
2           High        High           YES
3           High        High           YES
4           High        High           YES
Number of commits which are major according to CodeBERT Model 660
Number of commits which are major according to Bleu Tokens 660
Number of commits where methods agree 660
```

Figure 14: Applications of CodeBERT Model over source code before and after.

- **Explanation of the code:**
  - Plotted the Distribution histogram for the BLEU scored and set threshold of 0$\dot{9}$7.
  - Based on the threshold value, Semantic and Token_class columns are generated.
  - These columns are binary based on classification at THRESHOLD values.
  - If classification in both the values are equal, it is classified as Agree.
- **Results:**
  - Number of commits which are major according to CodeBERT Model 369.
  - Number of commits which are major according to Bleu Tokens 660
  - Number of commits where methods agree 369.