Research Laboratory
Inria / IRISA

263 Av. Général Leclerc, 35042 Rennes

Masters in Informatics
(2nd year)

Software Engineering

Sergiu MOCANU

Under the supervision of:
Olivier Barais, olivier.barais@irisa.fr, Enterprise
Nathalie Girard, nathalie.girard@irisa.fr, ISTIC

Code Quality Metrics as a Proxy Test

The study of the relevance of code quality metrics: Can textual-similarity metrics replace tests?

Internship dates: march 15th 2024 to september 6th 2024
Internship defense in Rennes, august 30th 2024

Abstract

Any software product can be a target of malicious attacks. Both open and closed source apps might have security vulnerabilities that can be exploited in order to gain unauthorized access. While such vulnerabilities are much easier to detect in open source software (due to the full access to code), closed source products are not immune to security attacks either. This is called a *black box attack*: when the attacker has little to no prior knowledge about the internal systems of the software. This type of attack aims to find new information about the implementation and the infrastructure of an app which could be exploited. One possible solution that can prevent black box attacks is the employment of single-use software. Implementing a system in which every use offers a different implementation of the same service would fend off against black box attacks that exploit the same vulnerability. This practice is called *code diversification*.

Lately, a new element emerged that could potentially augment this practice–the technological advancements in Large Language Models. Generative AI has shown promising results in the last few years, including its capabilities in code generation. This technology could push the single-use software strategy even further. However, due to the *unpredictable* nature of Large Language Models, it is necessary to verify and validate that the generated variant respects the desired specification. The problem lies in the fact that verification and validation techniques are costly in terms of time and energy consumption, and applying them to all the AI-generated variants is unreasonable.

In the scope of this internship, we researched the use of simpler and cheaper tools, such as textual-similarity metrics, as a prefiltering mechanism for LLM-generated software variants. Most studied metrics–with the exception of one–are purely textual-based, meaning that they measure the degree of textual similarity between a given script and the reference(s). Such metrics are often applied in academic literature when ranking the ability of Large Language Models in code generation. In order to determine their performance in evaluating the “quality” of the code, we built a framework that applies textual-metrics to a dataset of LLM-generated code and employs *machine learning* in order to estimate the performance. The experimental results obtained thus far show that, in its current state, textual-similarity metrics are not suitable for evaluating code.

Table of Contents

1. Introduction

Security vulnerabilities can be found in any software product. The advancements in programming led to the creation of big and complex apps. The bigger the size of a software, the greater the potential number of *attack vectors* that can be exploited by malicious actors in order to acquire unauthorized access. Even closed source products are not immune to security attacks. There are techniques for attacking a software without having any knowledge about its architecture, called *black box attacks*. These techniques aim to discover information about an app that could enable the attacker to breach the security protocols. Once the vulnerability is discovered, it can be exploited on multiple instances of the same app–this is known as *mass exploitation attack*. One way of tackling this type of attack is to adopt a *single-use software* paradigm: for every service request, the user is presented with a different implementation of the same service. Such variety in service implementation would randomize the attack surface, potentially removing a common vulnerability necessary for mass *exploitation attacks*. This practice is called *code diversification*.

The recent advancements in generative AI, in particular Large Language Models (LLM), show very promising results, including their capabilities for code generation. This technology might drastically augment the practice of *code diversification.* Nonetheless, the unpredictable nature of LLMs leads to unstable performance, and the AI-generated code must be tested and validated in order to ensure the correct functionality. Unfortunately, the verification and validation techniques are expensive in terms of time and energy resources; employing these techniques for every generated variant is unreasonable and inefficient.

The subject of this internship was the research of *textual-similarity metrics* and their performance as a *proxy test*. We tried to determine if textual metrics can be utilized for prefiltering AI-generated code, which would allow us to focus the testing ressources only on the most promising variants. The cheap cost of employing such metrics would make them an efficient filtering mechanism. In order to research this subject, we started building a framework that aims to measure the relevance of using textual metrics as a proxy test. This is done by selecting a set of well-known textual metrics, measuring the score on AI-generated scripts from a benchmark and training a *prediction model* on the obtained scores. The performance results of the predictive model are then analyzed in order to draw conclusions.

This report details the background and the motivation of this research subject, describes all the steps of the experimental protocol, displays the obtained results and discusses the final observations and conclusions made during the research.

LLM for Code Diversification & Code Quality Metrics

Inria | IRISA UMR

DiverSE – Diversity-Centric Software Engineering

What is **Code Diversification**?

Changing the *implementation* without changing the *functionality*

```java
public class FibonacciIterative {
    public static void fibonacci(int n) {
        int a = 0, b = 1;
        System.out.print("Fibonacci Series up to " + n + " terms:");
        for (int i = 0; i < n; i++) {
            System.out.print(a + " ");
            int temp = a;
            a = b;
            b = temp + b;
        }
    }

    public static void main(String[] args) {
        int n = 10; // Change the value of n as needed
        fibonacci(n);
    }
}
```

```java
public class FibonacciRecursive {
    public static int fibonacci(int n) {
        if (n <= 1)
            return n;
        else
            return fibonacci(n - 1) + fibonacci(n - 2);
    }

    public static void main(String[] args) {
        int n = 10; // Change the value of n as needed
        System.out.println("Fibonacci Series up to " + n + " terms:");
        for (int i = 0; i < n; i++) {
            System.out.print(fibonacci(i) + " ");
        }
    }
}
```

Why Code Diversification?

It is one of the techniques used in ***Moving Target Defense***

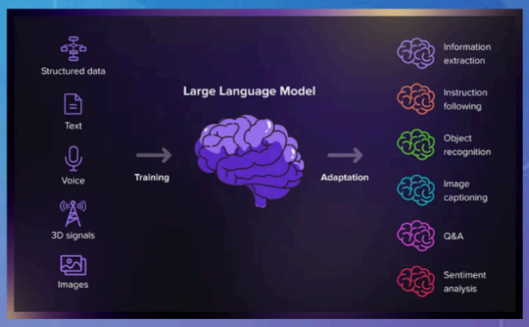
"Can't hack it if you don't know what you're looking for!"

How Code Diversification?

Deterministic tools are not very effective in code diversification thus far.

Therefore, *Introduce…*

Large Language Models



But what are the *challenges* of using LLMs?

They are **strong** yet *fragile*

They are **versatile** yet *unpredictable*

Simply put: they have no notion of **truth**

How can we *mitigate* these weaknesses?

Code Quality Metrics as Test Proxy

Instead of testing all the AI-generated code, use a code metric and a *confidence threshold* in order to filter out generated code which is obviously **faulty**.

Metrics from our research scope:
BLEU | **CodeBLEU** | **ROUGE** | **METEOR** | **ChrF**

What are their limitations?

Most studied metrics (with the exception of **CodeBLEU**) are purely *textual-based:* meaning that they only analyze the **textual correlation** between the *reference* and the *input*. Therefore, noise introduced by differences in variable names, the order of instructions, etc., will lower the score. On the other hand, syntactic/semantic incoherences will not be spotted by the metrics and will not affect the score.

Why **CodeBLEU** is any different?
This metric doesn't stop at the textual-level analysis: it also measures a weighted score, giving more weights to key-words that are native to a given programming language (e.g., *def, int, var*); an AST and data-flow match scores.

However, this metric does not parse the input code and, therefore, doesn't spot any compilation errors, which can lead to false positives.

Our current research question is:

Can we use existing metrics as a test proxy?
If not, can we create one?

Author: Sergiu MOCANU

Figure 1.1 - Poster detailing the big picture of the research subject

2. Presenting the laboratory and the team

The research was done at the biggest informatics laboratory in France, a fusion of two laboratories–*Inria and IRISA*–hosting on average 850 researchers, PhD students, engineers, and 150 interns, who are separated in 35 research teams which cover various fields of study in both informatics and electronics. My internship was done while being part of the team titled DiverSE (which stands for *"Diversity-Centric Software Engineering"*), which consists of around 45 teacher-researchers, professors, engineers and PhD students. As the name implies, the subjects tackled by the team are quite diverse (e.g., model-driven engineering, domain-specific languages, artificial intelligence), but ultimately aim to augment the field of software engineering.

The internship was supervised by the professors Olivier BARAIS and Mathieu ACHER, who established the general direction of the research, but I also got help from over members of the team. Special contribution to this research came from Théo MATRICON–a third year PhD student who came to visit our team for 2 months–who was the first one to start the discussion about the relevance of using textual-similarity metrics when evaluating code.

3. Research subject

3.1 Motivation

3.1.1 LLM for code diversification

As stated previously, this research is a subsection of a larger research subject–the use of Large Language Models (LLM) for *code diversification*. Code diversification is the practice of changing the implementation of code without changing its functionality. This is one of the strategies used in Moving Target Defense, a field of informatics that aims to avoid *massive exploitation attacks* which target the same vulnerability in multiple instances of a software product. The existing deterministic tools that accomplish the task of code diversification do not achieve a desirable level of diversity. Therefore, we want to determine if LLMs can be a robust tool for code diversification.

3.1.2 Proxy test

In the context of this research, a *proxy test* is a mechanism for prefiltering AI-generated code using the textual-similarity metrics before running the full set of tests. The prefiltering component is highly valuable due to two factors: the cost of testing code and the unpredictable nature of Large Language Models.

The verification and validation process is very expensive in terms of time and energy consumption. When testing a program, the entirety of the code must be executed multiple times.

In order to cover all the use cases, the testing scenarios are often numerous and generally surpass the scope of the tested program itself.

Furthermore, if we are to employ Large Language Models for code diversification, we have to account for the *unpredictable* nature of generative AI. LLMs are not *robust*, meaning that the generated answer is a product of *statistics* and *probabilities* that add up to an answer that seems closest to a response that the user *expects*. There is no notion of *truth* in the generated answer and sometimes LLMs tend to *hallucinate.* Therefore, one of the solutions would be generating multiple versions of code and utilizing only those that respect the desired specification. However, the testing process is very expensive. In contrast, textual-similarity metrics are faster and consume less energy, which would make them an efficient prefiltering mechanism. This would allow us to discard all the generated code with low scores and launch the full array of tests only on the promising implementations.

3.1.3 Textual-similarity metrics in academic literature

One of the main motivations behind this research is the common use of textual metrics in research papers and articles that talk about state-of-the-art Large Language Models and their capabilities in code generation. In the academic literature, the textual-metrics are used to determine the “quality” of the generated code, and the obtained scores are used to compare the code-generation capabilities of different LLMs. This strategy of establishing the “quality” of the code might be questionable due to textual-metrics’ limitations, which will be covered in the next section.

3.1.4 Textual metrics and code evaluation

The very nature of textual-similarity metrics might make them unsuitable for evaluating code. When reflecting upon the manner in which the score is measured, one can justifiably arrive at the conclusion that there is no correlation between the obtained score and the “quality” of the code. Apriori, these metrics can be both too *strict* and too *lax* when evaluating code. The noise introduced by simply changing the names of functions or variables, or changing the order of the instructions, will lower the obtained score. On the other hand, syntactic or semantic incoherencies will not affect the score. For example, not respecting the syntax of a given program (e.g., lack of closing parenthesis) or using a variable before initializing it will not be detected by textual-metrics and will not have an impact on the final score.

However, these are just speculations. The main goal of the internship was creating a rigorous approach of determining the performance of textual metrics as proxy tests, as well as building a framework that would allow the evaluation of different metrics on various benchmarks.

3.2 Background

3.2.1 Textual-based metrics

Textual-based metrics are tools that allow measuring the textual similarity between a given text and a "*perfect*" reference. Originally created to evaluate machine-generated natural text (e.g., machine translation, automatic summarization), these metrics take as input AI-generated text and a human-made, "perfect" reference and outputs a score that determines how similar the two versions are. The reference can be from one or multiple examples, since the ambiguity of natural languages introduces various ways of formulating the same idea and, therefore, there is no *one* right answer. The following is a simple example of applying one of the studied metrics on natural text.

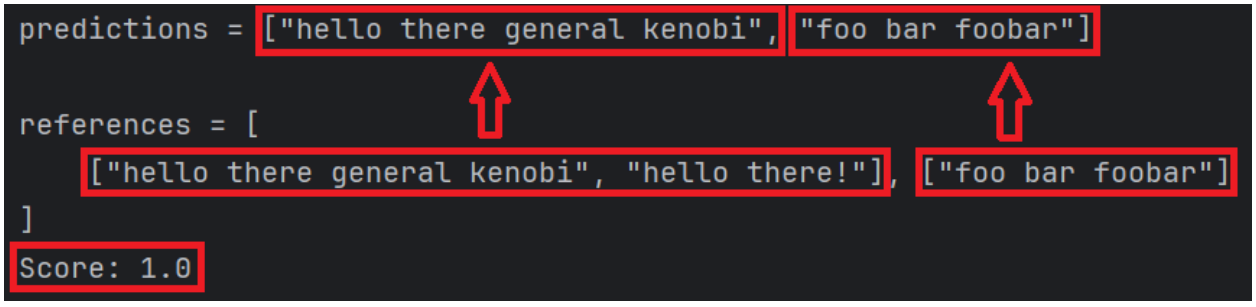


Figure 3.1 - Applying a metric to identical sentences

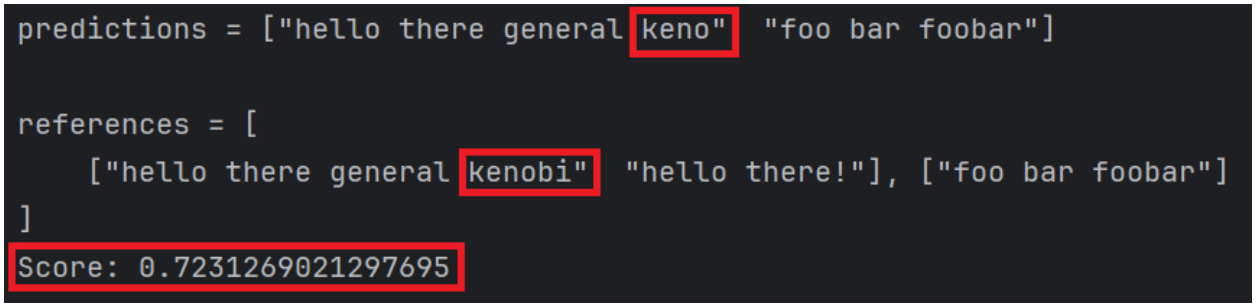


Figure 3.2 - Applying a metric to a sentence with a slight change

In Figure 3.1 we apply one of the studied metrics to a perfect match of sentences and we get a perfect score of **1.0**. Notice that the first reference has two versions; as mentioned previously, due to the ambiguity of natural languages, there are multiple ways of expressing the same idea, and the developers of textual metrics have implemented the option to compare with multiple 'perfect' references. However, the current research scope is focusing on a singular reference when comparing code. We can notice in Figure 3.2 a slight difference in the word "kenobi", which decreases the score to **0.7,** signaling a slight difference between the prediction and the reference.

3.2.2 Studied metrics

The following is the list of the metrics studied during the internship:

- BLEU - Bilingual Evaluation Understudy [1]
- CodeBLEU [2]
- ROUGE - Recall-Oriented Understudy for Gisting Evaluation [3]
- METEOR - Metric for Evaluation of Translation with Explicit ORdering [4]
- ChrF [5]

3.2.3 Differences in the studied metrics

The studied metrics focus on different aspects of the textual similarity when outputting the score. The difference lies mainly in their focus on *precision* or *recall*. To put it in simpler terms:

- Precision = among the extracted features, how many are relevant
- Recall = among the relevant features, how many were extracted

All the studied metrics–with the exception of one–are purely textual. CodeBLEU, on the other hand, was designed to compare code. While also measuring an n-gram match score (textual similarity), it also measures a *weighted* n-gram score with more weight given to words exclusive to programming languages (e.g., int, static, private), an abstract syntax tree (cf. Fig. 3.3) score and a dataflow (cf. Fig. 3.4) score.

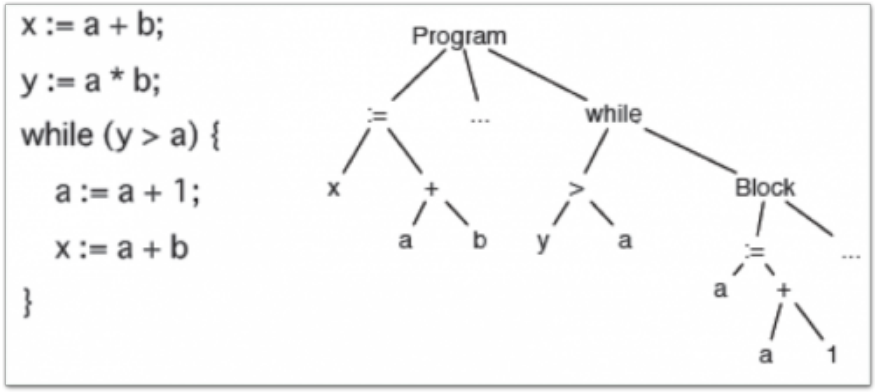


Figure 3.3 - AST representation of a simple script

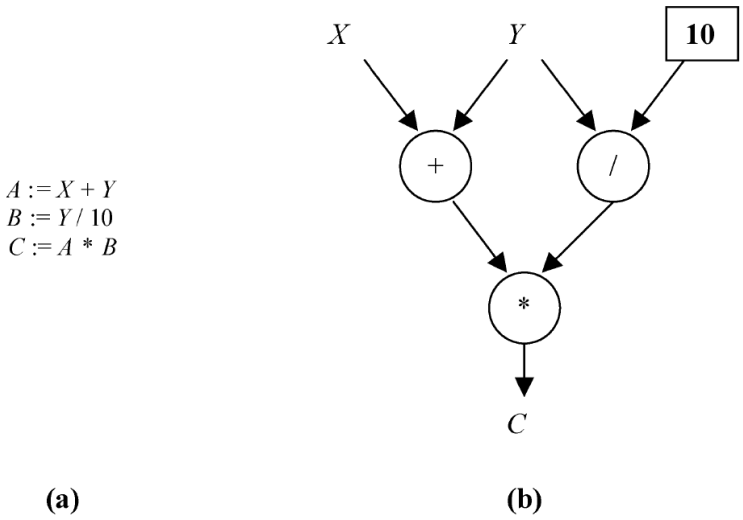


Figure 3.4 - Dataflow representation of 3 binary operations

3.2.4 Example of CodeBLEU application

```
Prediction:
def add ( a , b ) :
 return a + b

Reference:
def sum ( first , second ) :
 return second + first

Results:
codebleu: 0.5537842627792564
ngram_match_score: 0.10419044640136393
weighted_ngram_match_score: 0.11094660471566163
syntax_match_score: 1.0
dataflow_match_score: 1.0
```

Fig 3.5 - Applying CodeBLEU on two identically implemented scripts

As we can see in the Fig. 3.5, the two scripts are identical in both implementation and functionality: both output the result of the addition of two numbers given as input. The only difference is the choice of nomenclature of the function and the parameters. The similarity and the difference between the two scripts are reflected in the obtained score: the *n-gram* and *weighted n-gram* scores are low due to the difference in nomenclature, while *AST* and *dataflow* match has a perfect score of **1.0**. The “codebleu” score is the average value of all the other scores, which is the option by default. Alternatively, we can specify the parameters of the metric

in order to give different weights to the various scores, which could potentially give more coherent results for the overall “codebleu” score. However, this research focused on the default option where all the measured scores have an equal impact on the final score.

4. Experimental protocol

4.1 Benchmark

In order to research this subject, we required a database of code–preferably AI-generated–as well as the tests that cover all the use-case scenarios of the generated scripts. We chose the EvalPlus benchmark [6], which consists of 164 programming tasks implemented in Python. These tasks are fairly easy to implement, which makes this benchmark a good starting point for the research. This benchmark offers a big dataset of LLM-generated scripts that accomplish the aforementioned tasks, as well as a set of human-made, “canonical” implementations, with the addition of tests that cover all the use cases for each task. For each existing combination of LLM model/number of parameters and task, there are 200 generations of implementation. Due to the random nature of Large Language Models, they don’t output the same answer for a given prompt–at least in theory–which is why there are 200 generations for the same combination of model and task (which adds up to a total of 2,906,453 scripts).

```
from typing import List


def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """ Check if in given list of numbers, are any two numbers closer to each other than
    given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
    for i in range(len(numbers)):
        for j in range(i+1, len(numbers)):
            if abs(numbers[i]-numbers[j]) < threshold:
                return True
    return False
```

Figure 4.1 - Example of LLM-generated implementation of an EvalPlus task

```
from typing import List

def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """ Check if in given list of numbers, are any two numbers closer to each other than
    given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
    assert threshold > 0, "invalid inputs" # $_CONTRACT_$
    assert all([isinstance(v, (int, float)) for v in numbers]), "invalid inputs" # $_CONTRACT_$

    sorted_numbers = sorted(numbers)
    for i in range(len(sorted_numbers) - 1):
        if sorted_numbers[i + 1] - sorted_numbers[i] < threshold:
            return True
    return False


METADATA = {
    'author': 'jt',
    'dataset': 'test'
}

def check(candidate):
    assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.3) == True
    assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.05) == False
    assert candidate([1.0, 2.0, 5.9, 4.0, 5.0], 0.95) == True
    assert candidate([1.0, 2.0, 5.9, 4.0, 5.0], 0.8) == False
    assert candidate([1.0, 2.0, 3.0, 4.0, 5.0, 2.0], 0.1) == True
    assert candidate([1.1, 2.2, 3.1, 4.1, 5.1], 1.0) == True
    assert candidate([1.1, 2.2, 3.1, 4.1, 5.1], 0.5) == False

check(has_close_elements)
```

Human-mande implementation of an EvalPlus benchmark task

Set of tests covering all the use cases of the script

Figure 4.2 - Example of human-made implementation of an EvalPlus task with in-built tests

4.2 Functionality test

The first step of the experimental protocol is the testing of the LLM-generated scripts against the set of available tests in order to establish which generated scripts were “correct” (i.e., which scripts successfully pass the tests). This was done through iterating over the generated scripts, concatenating them with the appropriate set of tests and executing it in an isolated subprocess. The results of the test, including the type and the message error, were saved locally in a JSON file, all the while being categorized in a coherent system of directories and file names. The categorization of the test results in a coherent system of folders, as well as the structure of the JSON file, was vital for the future work and analysis of the obtained data.

```
"5.py": {
  "successful": false,
  "error_type": "NameError",
  "error_message": "name 'log10' is not defined"
},
"6.py": {
  "successful": true
},
"7.py": {
  "successful": false,
  "error_type": "NameError",
  "error_message": "name 'math' is not defined"
```

Figure 4.3 - Snippet of a JSON file containing the test results of AI-generated scripts

4.3 Metric measurement

Once the functionality tests were complete, the next step was the measurement of the metric scores for each studied textual-similarity metric. As a first step of the research, we chose as a baseline the canonical, human-made implementation of the benchmark's tasks. Before measuring the score, all comments and whitespaces were removed in order to avoid any unwanted noise. The results were saved locally in a CSV file, along with the LLM model, the name of the script and the *pass/fail* label of the functionality test. Once again, the structure of the CSV file was meant to facilitate the subsequent analysis of the obtained data.

model&temp	script	pass	score
chatgpt_temp_0.0	0.py	TRUE	0.1667678
chatgpt_temp_0.8	0.py	TRUE	0.2091105
chatgpt_temp_0.8	1.py	TRUE	0.2467534
chatgpt_temp_0.8	2.py	FALSE	0.1885202
chatgpt_temp_0.8	3.py	FALSE	0.2772066

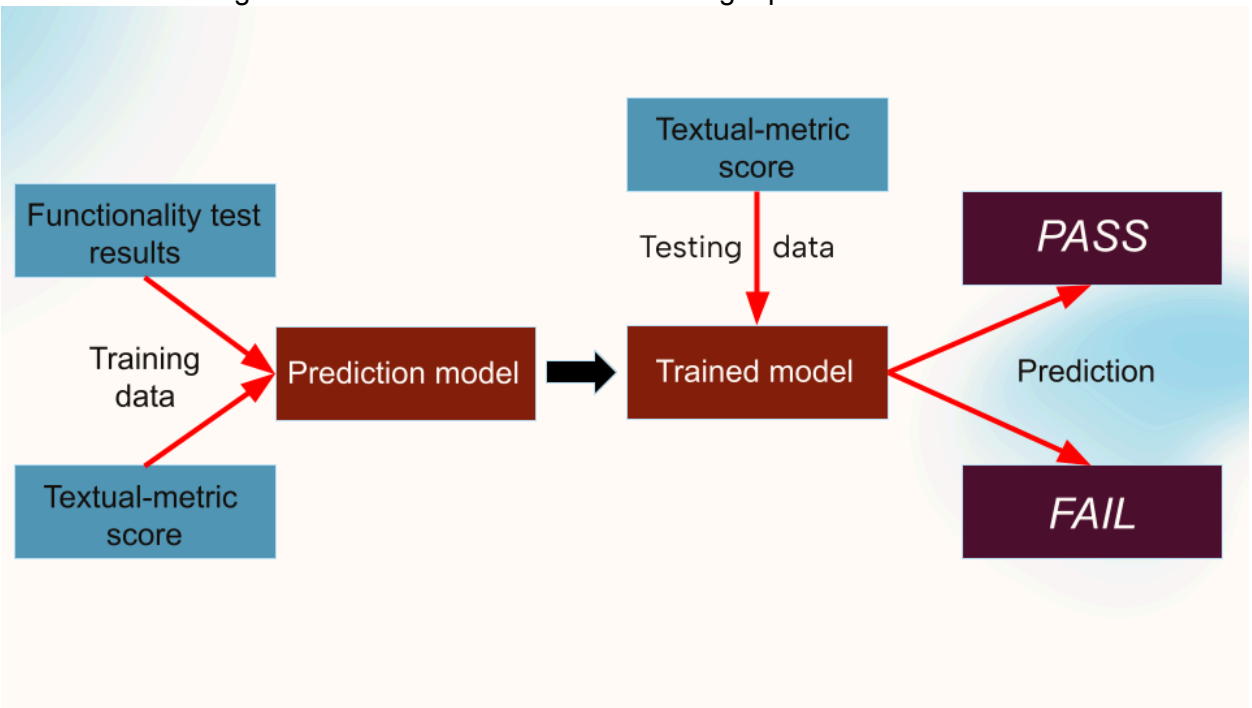
Figure 4.4 - Example of a CSV file containing the metric results

Although the name of the benchmark task and the metric do not appear explicitly in the CSV file, those can be inferred from the file's location within the project. Hence, the effort of creating a coherent folder system for stocking the experimental results.

4.4 Textual-metric performance measurement

After some deliberation, we have decided to use machine learning in order to determine the performance of textual metrics as test proxy. This was done through training a predictive model on the test results and metric scores, then testing the performance of the obtained classifier on unseen metric scores.

Figure 4.5 - Workflow scheme of training a predictive model



4.4.1 Logistic Regression

Logistic regression [7] is a supervised machine learning algorithm that accomplishes binary or multi-class classification based on a set of *features*. It is somewhat similar to linear regression (cf. Fig. 4.6), but instead of outputting a continuous value, it predicts the membership of an object to a certain class, based on a set of *features* (i.e., data that characterizes the predicted object). Generally speaking, logistic regression is used for binary classification, but multi-class classification is also available.

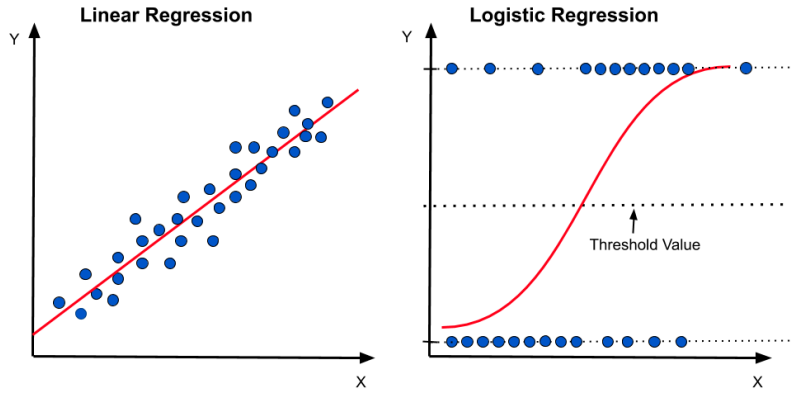


Figure 4.6 - Linear and logistic regression comparison

4.4.2 Model training

The way that logistic regression models are trained is pretty straight-forward. The model starts off with a default sigmoid function as seen in Fig. 4.6. Then, it iterates over the features–in our case, the metric score of a script–and the information it is supposed to predict, called *target* (which, in our case, is the *pass/fail* label of a script). While iterating over the features and target values, it applies modifications to the sigmoid function–through the use of formulas from mathematics and statistics–to better adapt itself to the training data. Once the training is complete, the model is ready to accomplish the classification based solely on the features–*if* the training data is of a good quality. To better visualize the classification process, let us take an example from our research context and apply it to the Fig. 4.6. Suppose that the 'x' axis is the metric score value, and the 'y' axis is the *pass/fail* probability. Given a certain 'x' score, the value of 'y' will be equal to the position of the sigmoid function at the value 'x': this gives us the probability that the script passed or failed the tests. Then, based on the model's *threshold value* (generally known as the *decision boundary*), the model predicts whether or not the script will pass the tests. If the obtained probability is above the decision boundary, then the model will classify the script as True (or *pass*). *Fail* otherwise.

4.4.3 Model testing

In order to test the performance of the trained classification model, we have to use the same type of data that is used during training. Traditionally, in machine learning, the data is split into ¾ for training and ¼ for testing. During training, both the *features* and the *target* data is available to the model in order to enable it to "learn" a correlation between the two; during testing, however, only the *feature* values are visible and the model has to predict the *target* value for each classified object. In machine learning, the split of data into training/testing is a necessary factor. Training data cannot be used for testing, otherwise it will lead to *model*

overfitting (i.e., overfitting the model's parameters which leads to *model bias*). In other words, the predictive model becomes a *switch case* statement.

In the interest of avoiding a favorable split of training/testing data, which would give results that are too optimistic, the process of training and testing the model is done 100 times and the obtained data is aggregated with an average value. Which brings us to the next section.

4.4.4 Model performance metrics


```
"pass": {
  "precision": 0.806201378609395,
  "recall": 0.9647326487283424,
  "f1-score": 0.8783712768380314,
  "support": 565679.0
},
```


The logistic regression module from scikit-learn gives very descriptive data about the performance of a predictive model (cf. Fig. 4.7), but before analyzing the obtained results, we first have to delve deeper into their meaning.

Figure 4.7 - Snippet of model performance results

4.4.4.1 TP/FP/TN/FN

The first and easiest aspect of machine-classification performance evaluation is the concept of True/False and Positive/Negative. As the name implies, a binary classification is the act of associating an object to one of two classes: positive/negative, 0/1, *pass*/*fail*. On the other hand, 'True' or 'False' are meant to describe the correctness of a prediction. A 'True Positive' (TP) means that the classified object is part of the class 'positive' (or *pass* in our case) and the model correctly predicted it to be 'positive'. Alternatively, a 'False Negative' (FP) means that the classified object belongs to the class 'positive' but was mistakenly predicted to be part of the class 'negative' (*fail* in our context).

4.4.4.2 Precision, recall, f1-score, accuracy

Now that we are familiar with the concept of True/False and Positive/Negative, we can tackle the performance metrics found in the Fig. 4.7.

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

Precision is the ratio between TP and TP+FP. In other words, how many 'positive' predictions are accurate.

Figure 4.8 - Formula for Precision

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

Recall is the ratio between TP and TP+FP (i.e., among all the 'positive' instances, how many were predicted to be positive).

Figure 4.9 - Formula for Recall

$$\text{F1-score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

F1-score is the harmonic mean of Precision and Recall. Put differently, it is the average of the two ratios.

Figure 4.10 - Formula for F1-score

$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{Total Samples}}$$

Accuracy is the ratio between all the correct predictions and the total number of samples.

Figure 4.11 - Formula for Accuracy

4.4.4.3 Support, macro and weighted average

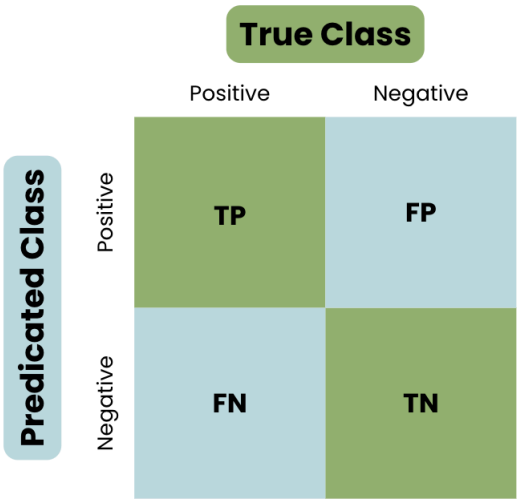
The set of *precision*, *recall* and *f1-score* results are given for each predicted class, as well as “support": this is simply the number of instances of the predicted classes. For example, in the Fig. 4.7, the *test* split of the dataset contained 565,679 instances of scripts that *passed* the tests. The “support” value is important due to the following prediction-performance metrics: macro and weighted average.


```
"macro avg": {
  "precision": 0.5730935788309718,
  "recall": 0.5051343569147471,
  "f1-score": 0.4847122031249831,
  "support": 370440.0
}
"weighted avg": {
  "precision": 0.8193737802527777,
  "recall": 0.885387647122341,
  "f1-score": 0.8380347043676977,
  "support": 370440.0
}
```


Figure 4.12 - Example of macro and weighted average

Macro is the average value of the prediction-performance results that does not take into consideration the *pass/fail* label distribution. On the other hand, weighted average takes this factor into account, which can have a big impact in the case of an unbalanced dataset (cf. Fig. 4.12).

4.4.4.4 Confusion matrix



Another way of visualizing the performance of a classification model is the *confusion matrix.* It is a 2 by 2 matrix that displays the number of TP, FP, TN and FN predicted by the model. This is a more visual and intuitive way of representing the performance of binary classification AI.

Figure 4.13 - Example of a confusion matrix

5. Experimental results

5.1 Disclaimer

Before looking at the experimental results, it is worth noting 2 aspects of the chosen dataset that might have an impact on the data obtained from the experiments. The first aspect is the fact that there are many generated scripts that are exact copies of one another, even down to the names of variables (a total of 1,424,695 duplicate scripts). This creates a lot of repetition in the training and testing data, which could have an impact on the model's training outcome. The second aspect worth noting is that the majority of the generated code fails the tests (more than 77%), a factor that also could have an impact on the classifier training and performance.

5.2 Logistic regression results

These are the results of the logistic regression done on the metric score coupled with the *pass/fail* label. The training and testing process was repeated 100 times per studied metric for the purposes of reproducibility and credibility (i.e., to avoid any favorable split of *train/test* datasets). The average as well as the variance values were measured for each set of 100-iteration results, but in order to avoid overwhelming the reader with too much data, we will focus only on the average value (in depth analysis of the following results is not necessary).

Metric Type	Category	Precision	Recall	F1-Score	Support	Accuracy
average	pass	0.599120057458506	0.184878508290734	0.282561308107619	16095041	
average	fail	0.806197299769631	0.964801147878969	0.878397153096227	56566359	
average	accuracy					0.79204239114578
average	macro avg	0.702658678614069	0.574839828084851	0.580479230601923	72661400	
average	weighted avg	0.760328569166486	0.79204239114578	0.746415116685634	72661400	
variance	pass	0.00193326231435223	0.000722584648121569	0.000942334770036712	16095041	
variance	fail	0.00034492940410778	0.000255168280425967	0.000231133888909772	56566359	
variance	accuracy					0.000359134010630131
variance	macro avg	0.00100077671505128	0.000373814850200365	0.000541486097178179	72661400	
variance	weighted avg	0.000563243791532892	0.000359134010630131	0.000457460821539343	72661400	

Figure 5.1 - BLEU logistic regression results

Metric Type	Category	Precision	Recall	F1-Score	Support	Accuracy
average	pass	0.587500985368584	0.173667301124733	0.268085875052282	16093926	
average	fail	0.804150747408613	0.96530772202679	0.877390297910001	56567474	
average	accuracy					0.789965538786
average	macro avg	0.695825866388599	0.569487511575761	0.572738086481142	72661400	
average	weighted avg	0.756165000113754	0.789965538786756	0.742434227812138	72661400	
variance	pass	0.00156758412401308	0.000649773161380703	0.000827429053807463	16093926	
variance	fail	0.000342368781200849	0.000210867567454652	0.000211039683188525	56567474	
variance	accuracy					0.000324877307621
variance	macro avg	0.000779917634627227	0.000305046272335191	0.000453674848609869	72661400	
variance	weighted avg	0.00043407108818064	0.000324877307621383	0.000435787946438467	72661400	

Figure 5.2 - CodeBLEU logistic regression results

Metric Type	Category	Precision	Recall	F1-Score	Support	Accuracy
average	pass	0.659105583712514	0.312252287213715	0.423750732664999	16097059	
average	fail	0.829773852776844	0.954040325282565	0.887578610761139	56564341	
average	accuracy					0.811861607
average	macro avg	0.74443971824468	0.63314630624814	0.655664671713069	72661400	
average	weighted avg	0.791965078775844	0.811861607400903	0.784824449118501	72661400	
variance	pass	0.00115756702907647	0.000762821371533572	0.000840863951791285	16097059	
variance	fail	0.000376207021690645	0.000230159132543865	0.000257117779854557	56564341	
variance	accuracy					0.000386271115
variance	macro avg	0.000641759616974833	0.000397209277559772	0.000489628091804012	72661400	
variance	weighted avg	0.000454219825338401	0.000386271115062469	0.000461701650894947	72661400	

Figure 5.3 - ROUGE logistic regression results

Due to the fact that the logistic regression results for METEOR and ChrF metrics are similar to the above, we can focus on the first three metrics. A few observations can be made thus far:

- All the metrics have relatively similar results
- The results for the *fail* label seem promising at a first glance
- The results for the *pass* label are underwhelming
- The *weighted* average results are better compared to *macro*
- The overall ‘accuracy’ value of 79%-81% seems promising

Despite the fact that some results seem encouraging (like the ‘accuracy’ and the *fail* class detection rate) there is a serious issue regarding the use of textual-metrics as *test proxy*. Although the above results are vital for drawing conclusions, the big amount of numerical data makes it hard to detect the underlying issue just by observing the data, which is why the *confusion matrix* representation is useful in this situation.

5.2.1 Confusion matrix representation

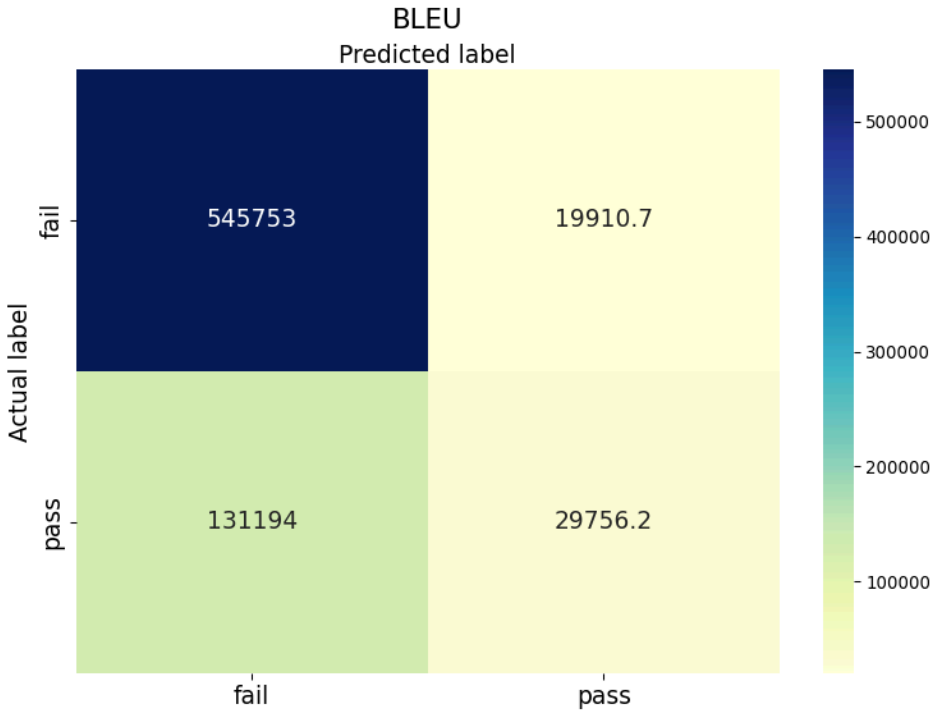
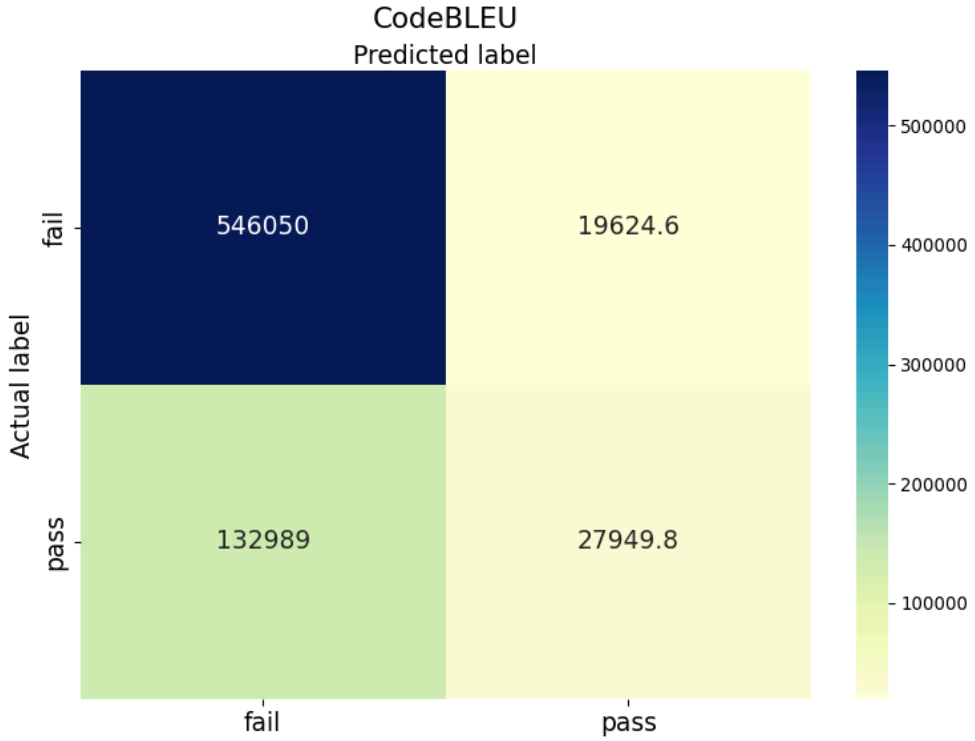


Figure 5.4 - BLEU confusion matrix

Figure 5.5 - CodeBLEU confusion matrix



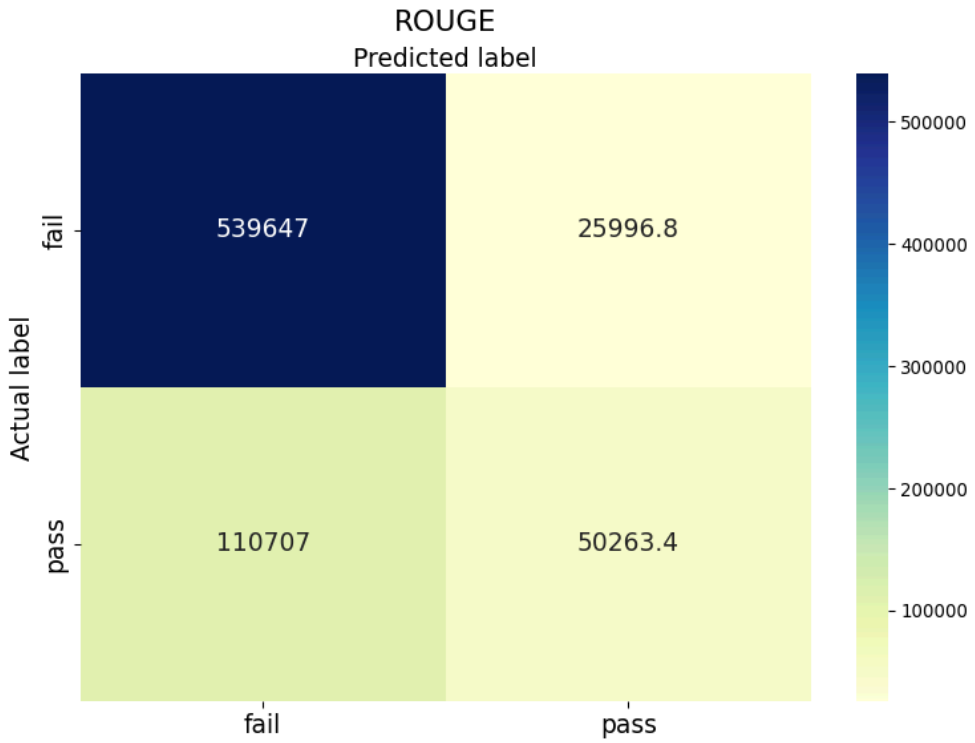


Figure 5.6 - ROUGE confusion matrix

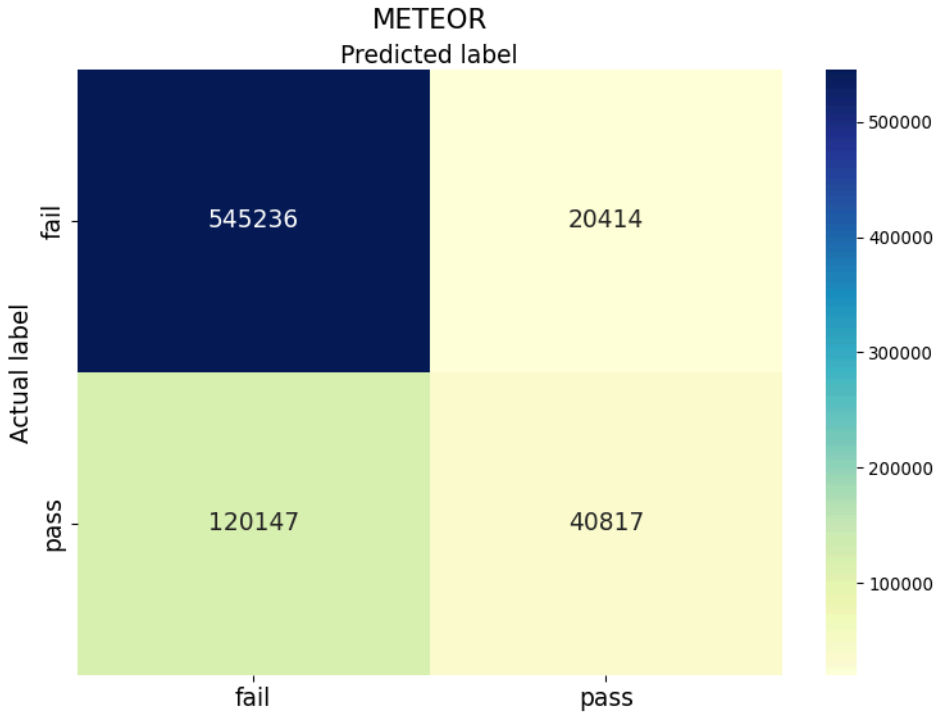


Figure 5.7 - METEOR confusion matrix

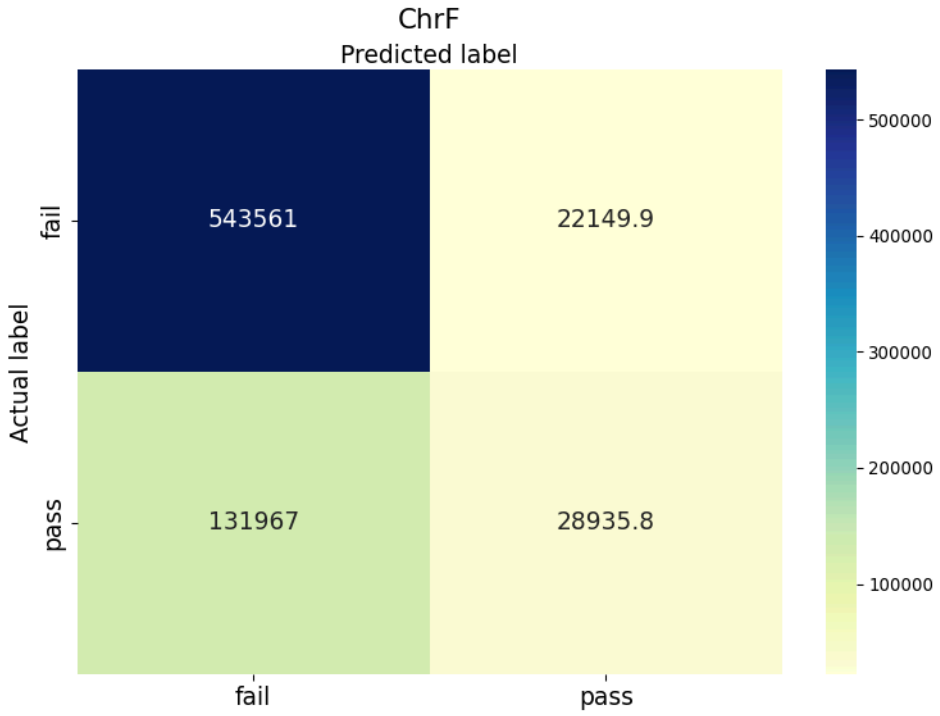
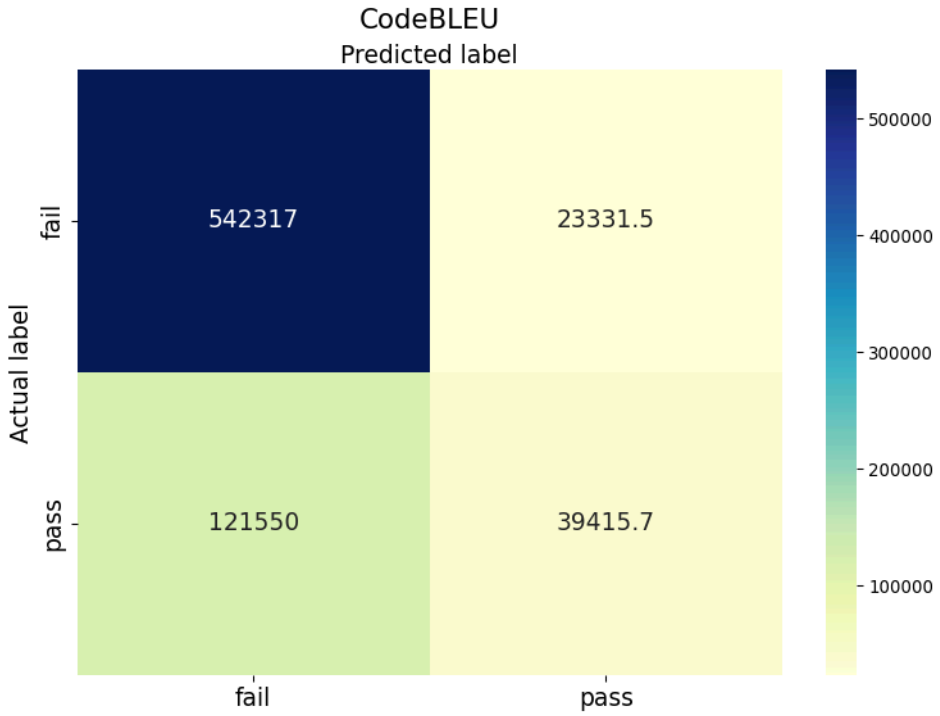


Figure 5.8 - ChrF confusion matrix

Additionally, the classification *train/test* procedure was done on CodeBLEU with the use of the more granular scores (n-gram, weighted n-gram, AST and dataflow) in order to establish if the classifier can make better predictions if given more information about a script. The first classifier was given all the scores, including the overall metric score (which is simply the average value of all the other ones), and the second classifier was given only the finer scores.

Figure 5.9 - CodeBLEU confusion matrix with all the scores



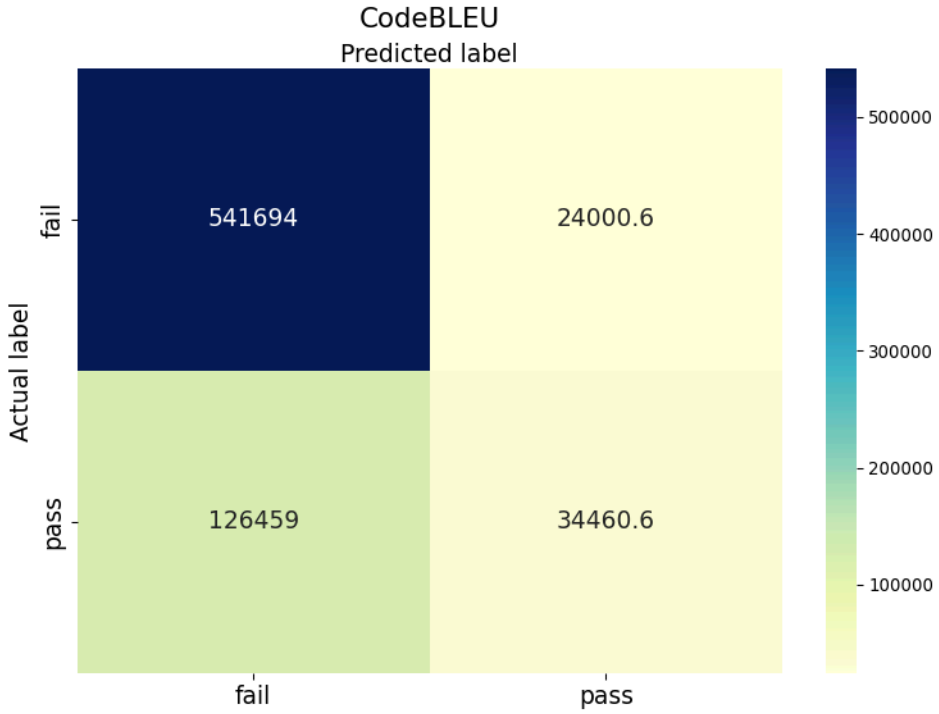


Figure 5.10 - CodeBLEU confusion matrix with granular scores only

5.2.2 Results analysis and observations

Visualizing the classifiers' performance results in the form of a confusion matrix paints a clearer picture and allows us to explain the somewhat good results for the *fail* class prediction rate, in spite of the overall bad performance. Just like the numerical representation of the results, the confusion matrix of all the textual-metric classifiers look very similar. The first thing worth noting is that for all the classifiers, the majority of the predictions were for the class *fail*, averaging at around 543k TN predictions and 130k FN. This explains why the prediction rate for the class *fail* has such promising results in the performance metrics. Let us take as an example the performance results of the BLEU (cf. Fig. 5.4) metric: if we take the information from the confusion matrix and apply the formula for *precision*–TN / (TN + FN)–we would get the same value as the one displayed in the numerical representations of the classification results (0.8061). This data is coherent with the dataset selected for this research, which contains 77% of scripts that fail the tests: this affects the behavior of the trained classifier and increases the probability of correctly classifying a script as *fail*. Therefore, the classifier's capability of predicting TN in this context is completely justifiable. However, if we look at the ratio between FN and TP, we can conclude that the classifiers predict on average 4.5 times more FN than TP, meaning that the great majority of scripts that belong to the class *pass* are actually labeled as *fail*.

Another important thing to note is that among all the predictions, the classifiers predicted the least FP (meaning the scripts that belong to the class *fail* but were classified as *pass*). However, there are on average twice as many TP than there are FP, which is not a good result since ⅓ of *pass* predictions are false.

The reason for such underwhelming results might be the unbalanced dataset with a majority of faulty scripts and almost half of them being duplicates, which can also cause model overfitting during the training process. Therefore, the next step of the research is restarting the machine learning procedure on a version of the dataset that has no duplicates.

5.3 Logistic regression with distinct scripts

After removing all the duplicate scripts from the dataset, its size decreased from 2,906,453 scripts down to 1,481,758. This also had an impact on the distribution of *pass/fail* classes–with 87% of test fail rate–which makes the dataset even more unbalanced. However, the lack of duplicate data will avoid any overfitting of the prediction models and might have a positive impact on the performance of the classifiers.

5.3.1 Confusion matrix results

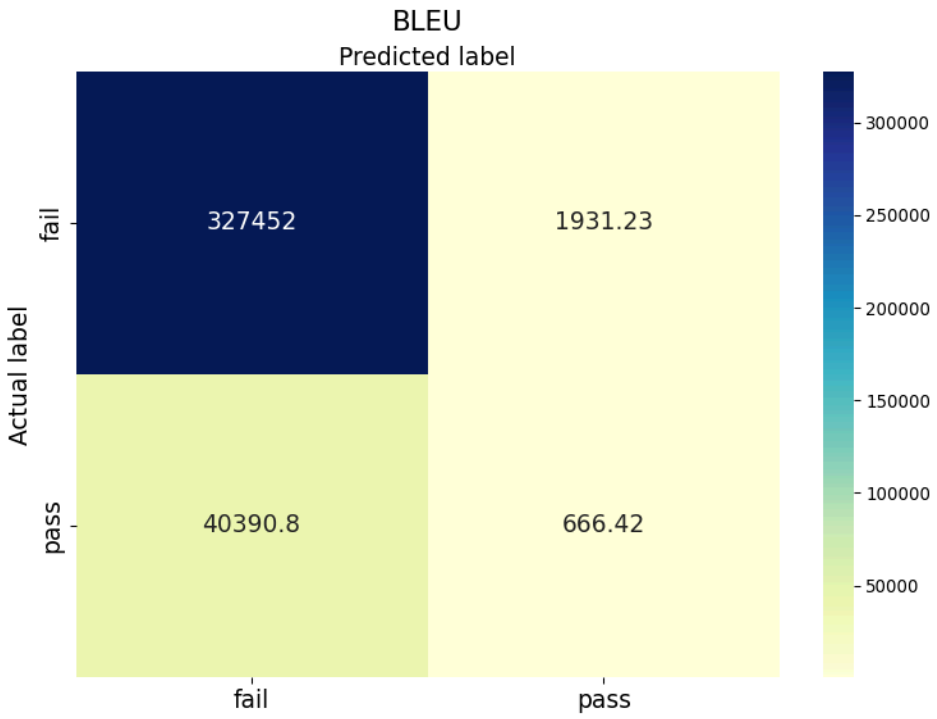


Figure 5.11 - BLEU confusion matrix with distinct scripts

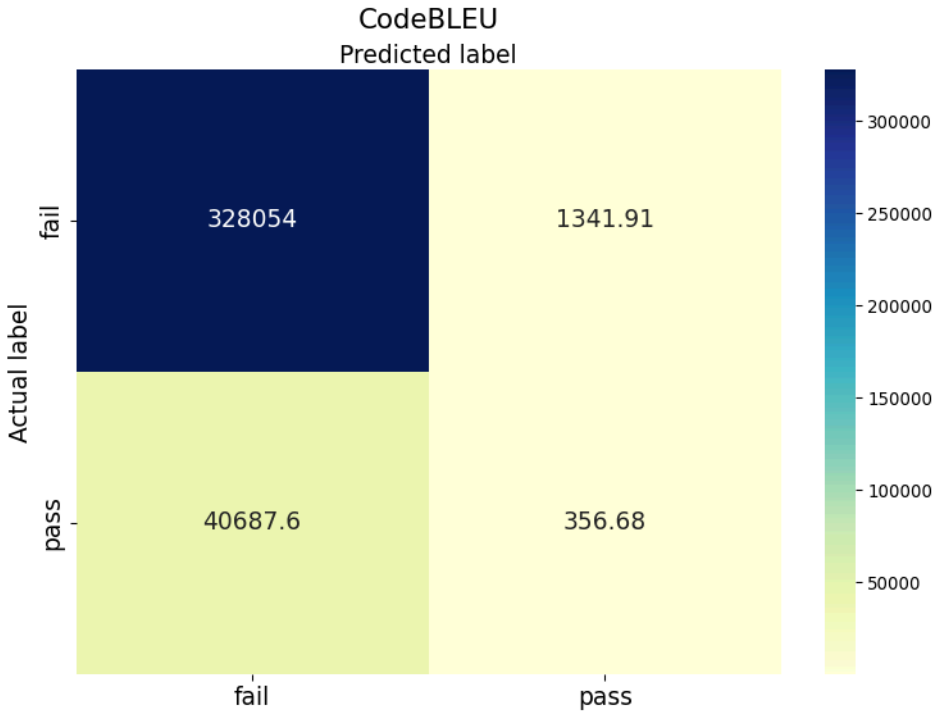


Figure 5.12 - CodeBLEU confusion matrix with distinct scripts

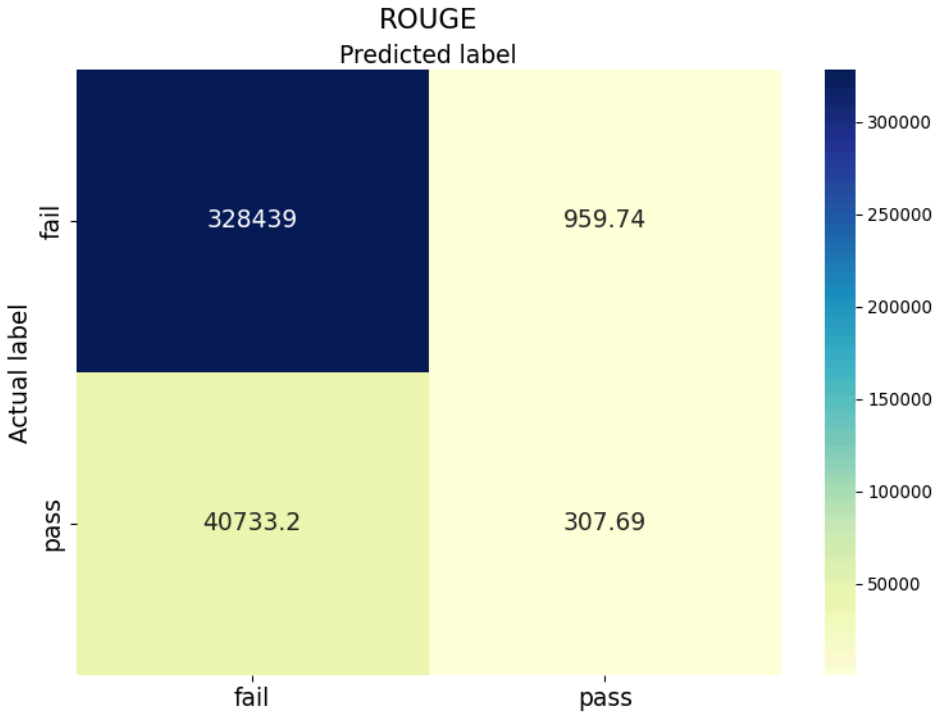


Figure 5.13 - ROUGE confusion matrix with distinct scripts

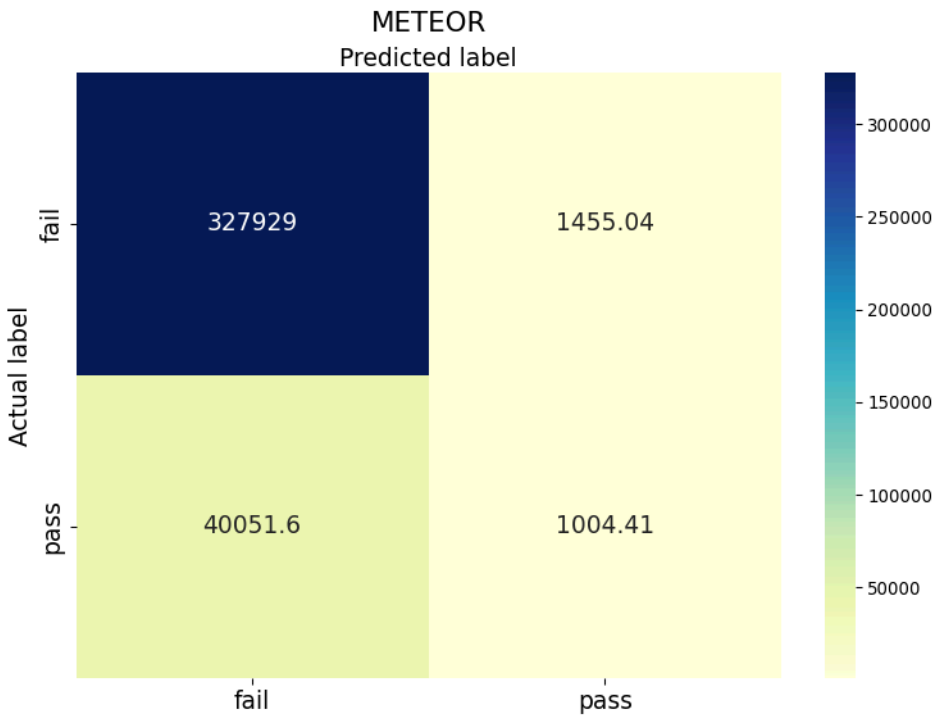


Figure 5.14 - METEOR confusion matrix with distinct scripts

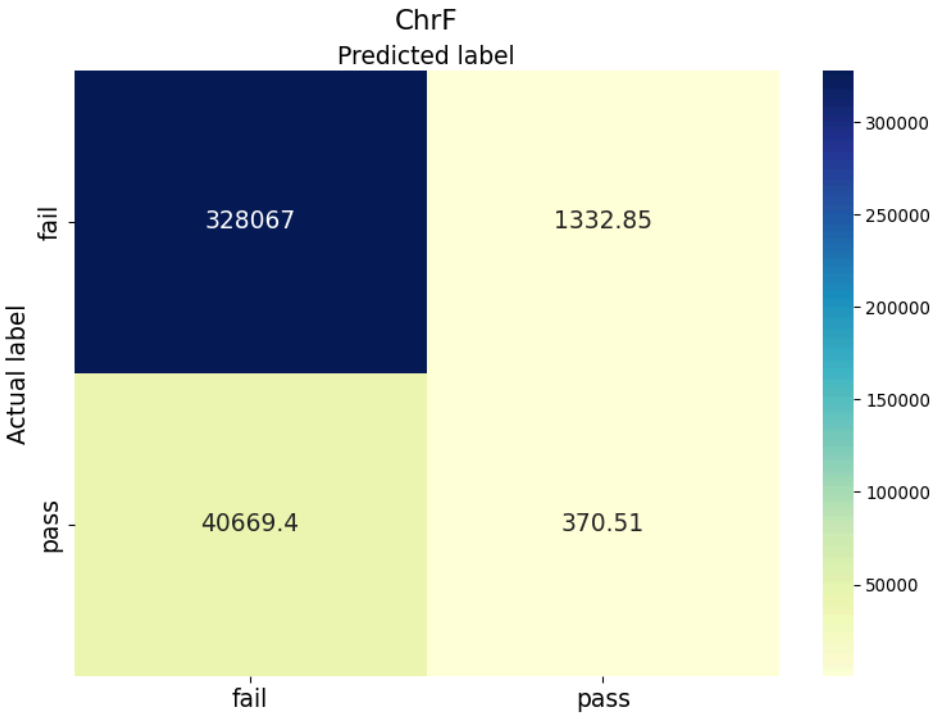


Figure 5.15 - ChrF confusion matrix with distinct scripts

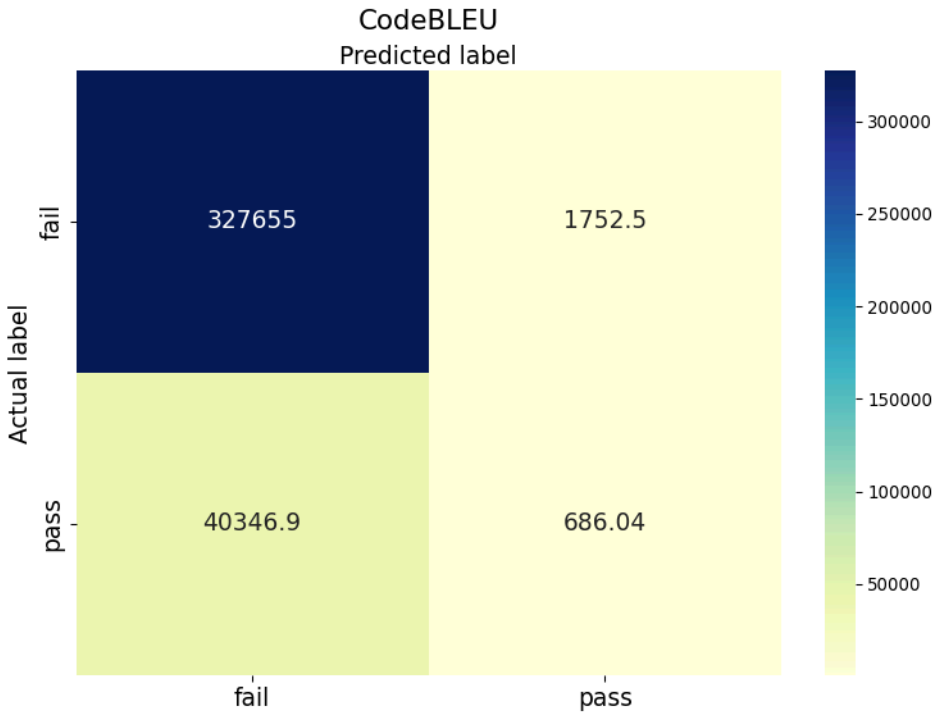


Figure 5.16 - CodeBLEU confusion matrix with distinct scripts and all the metric

5.3.2 Results analysis and observations

After repeating the logistic regression on the filtered dataset, we can notice a change in the behavior of the classifiers. Once again, the classifiers of all the metrics have a similar behavior. However, the results show that the number of TN (correct classification of faulty scripts) and the number of FN (functional scripts that were misclassified as being faulty) have stabilized for all the classifiers. There are on average 32.8k TN and 40.5k FN, with very little variation. On the other hand, the ratio between TP and FP is fluctuating between the different metrics and does not show any obvious pattern, except the fact that there are more FP than TP predictions, which means that the majority *positive* predictions are incorrect and point to faulty scripts.

Also, the average number of FN predictions is noticeably lower after removing all the duplicate scripts. With an average of 40.5k FN predictions, there are 4 times fewer functional scripts that are mislabeled as *fail*. Although the improvement is considerable, this performance is far from acceptable.

Even the performance of the CodeBLEU classifier is similar to the textual metrics and shows no substantial advantage in code evaluation, despite it being the only metric that was designed for code analysis. The poor performance of the studied metrics might be caused by both the quality of the chosen dataset as well as the very nature of the studied metrics, which will be discussed in the following section.

5.4 Discussion on benchmark and textual-metrics

The choice of the benchmark used for this research has a great impact on the experimental results. The difference in the distribution of the *pass/fail* classes offered few examples of scripts that pass the tests, which leads to fewer *positive* training data for the prediction models. Additionally, the quality and the nature of the scripts present in the dataset might have an impact on the scores obtained with textual metrics. Due to the origin of scripts–AI generation–the reasons for failed tests are varied. A good portion of errors come from mistakes that are similar to those made by human developers: assertion errors (bad functionality), runtime errors (mistakes in code that lead to faulty executions), infinite loops, out of bounds access, missing imports, etc. However, there are also errors that are exclusive to Large Language Models. Occasionally, LLMs tend to *hallucinate*–a showcase of their unpredictable nature–and lose control over the generated answer, which can lead to rather peculiar responses. The following is an overview of some mistakes that are exclusive to Large Language Models.

5.4.1 Examples of LLM-exclusive errors

As mentioned previously, generative AI can sometimes lose control over the generated answer and give responses that are not factually correct or do not make sense. This is due to the way the answer is generated: after reading the prompt, the AI makes a list of words that have the highest probability of appearing next, then a word is chosen somewhat at random and placed in the answer. After each generated word, the AI reads the input prompt and the answer generated thus far and starts the whole process from scratch. No fact-checking or "control" mechanism is used to steer the generation of the response–everything is based purely on the statistical heuristics learned during the training phase. This way of operating can sometimes make the AI *hallucinate* an answer.

Examples of such phenomena can lead to implementations that are ended prematurely. In other cases, an instruction is being copied and pasted an absurd amount of times inside a *for* loop (which leads to a stack overflow). Sometimes, the AI would not respect the syntax of the programming language and leave the uncommented name of the programming language at the top of the script (cf. Fig. 5.17).

```
python
def separate_paren_groups(paren_string: str) -> List[str]:
    groups = []
    stack = []
    for char in paren_string:
        if char == '(':
            stack.append(char)
        elif char == ')':
            stack.pop()
            if not stack:
                group = ''.join(stack)
                groups.append(group)
        else:
            # Ignore any spaces in the input string.
            continue
    return groups
```

Figure 5.17 - Script containing the uncommented name of the programming language

Other times, the LLM would simply refuse to implement a

function, instead returning the given signature and raising a “NotImplementedError” (cf. Fig. 5.18).

```
def file_name_check(file_name):

    # Implementation goes here
    raise NotImplementedError('This is just an example test, the actual implementation doesn't even exist yet.")
```

Figure 5.18 - Example of unimplemented function

```
def modp(n: int, p: int):
    """Return 2^n modulo p (be aware of numerics).
    >>> modp(3, 5)
    3
    >>> modp(1101, 101)
    2
    >>> modp(0, 101)
    1
    >>> modp(3, 11)
    8
    >>> modp(100, 101)
    1
    """
    if n == 0:
        return 1
    elif n == 1:
        return 2
    elif n < 0:
        return -1
    elif n == p:
        return n
    elif n > p:
        return n % p
    else:
        raise Exception("WTF?")
```

In some rare cases, the LLM hallucination can lead to implementation choices that might *raise* some ethical questions. (cf. Fig 5.19).

Figure 5.19 - Example of a made-up exception

Overall, the variety of sources of errors from this dataset–coupled with their quantity–poses a difficulty when trying to establish the “quality” of the code. The chosen benchmark proves itself as a challenging experimental “playground”, which is both favorable and disadvantageous (or rather, demanding) for the study of textual metrics and their relevance as a *proxy test*. In order to understand it better, we have to reflect upon the way textual-metrics work and try to mentally apply it to the examples found in the dataset, focusing on scenarios that could lead to incoherent scores regarding the *pass/fail* label.

5.4.2 Textual-metrics as test proxy

As stated previously, this study aimed to establish if textual-similarity metrics can replace functionality tests or, alternatively, if they can be employed as test proxy, filtering out all the implementations that have little probability of being a viable solution–which would allow the developer to focus the testing ressources only on promising implementations. If one was to use generative AI for *code diversification*–or any other programming task–such a filtering mechanism would be extremely valuable in terms of saving time, computing and energy resources.

That said, the results of the experiments done thus far show that textual metrics have obvious flaws in the context of code *qualification*. The unbalanced–and perhaps chaotic–nature of the dataset most probably had an impact on the metrics' performance, but the underwhelming results might also be due to the way these metrics operate. As a reminder, most of the studied metrics were created for machine-generated natural text analysis, which is a slightly different context than that of code. The strict syntax of programming languages might create unwanted noise when comparing only the textual similarity of different scripts.

5.4.2.1 Textual-metric limitations

As mentioned previously, the textual metrics might be both too *strict and lax* when analyzing code. If the naming scheme is different or if the order of instructions is changed, the metrics are too *strict* and lower the score due to textual differences. Moreover, programming languages can be ambiguous, meaning that they can sometimes offer multiple ways of implementing the same functionality. However, the semantic meaning of scripts is not taken into consideration, and the various ways of implementing the same task is not reflected in the textual side of the code, which creates unwanted noise. On the other hand, these metrics are too *lax* in situations where the corpus of the code is identical to the reference–with the exception of a minute difference (e.g., missing import, missing closing parenthesis). In such cases the textual similarity between the reference and the generated code is high and leads to *false positives.*

5.4.2.2 CodeBLEU

Even CodeBLEU–which is the only studied metric that was designed for programming languages–struggles to give coherent scores regarding the *pass/fail* label. One possible reason for this underperformance might be due to an issue that was raised on its GitHub repository, which points to incoherent *dataflow* score measurement. The issue was raised months ago and was not addressed as of yet. However, another important thing to note about CodeBLEU is that, in the research work done thus far, only the default parameters were used and no experiments were done with varying score weights. By default, all the granular scores that the metric outputs (n-gram, weighted n-gram, AST and dataflow) have equal impact on the overall score; meaning that, for example, the differences in the textual similarity and the AST have equal impact on the final score. Experimental results show that analyzing code based on the textual similarity might

not be the best strategy, and more attention should be focused on the syntactic and semantic meaning of a script. Experimenting with CodeBLEU's parameters might yield better results in code evaluation, which is part of the project's *future work* that will be discussed shortly.

6. Accomplished work

In this section, I will present the work that was done during this internship, as well as touch on the things I learned and acquired while doing the research: expertise, habits and skills which will prove most useful in the future work that I plan to do.

6.1. Programming-related work

6.1.1 Mastering Python

During this internship, all the code was written in the Python programming language. The main software that was produced is all the code that was applied for AI-script testing, metric measurement, machine learning and experimental result analysis, which will serve as a foundation for a framework [8][9] that will allow developers to test their own "code-quality" metrics on the desired benchmark. From the very beginning, a considerable effort was focused on producing code that is *robust*, easy to understand, maintain and refactor. Best practices and coding paradigms were applied while implementing the programs, practices which are meant to facilitate and ameliorate the coding process (e.g., single responsibility principle, KISS, modularity). It is worth noting that the extent of my experience with Python was limited to the two-and-a-half internship in the same laboratory. While Python is very similar to other object-oriented programming languages (in terms of both syntax and semantics) many functionalities and syntactic options were yet unknown. Therefore, a good amount of valuable programming experience was obtained while building this framework.

6.1.2 Bug-fixing and result validation

A substantial amount of time and energy was spent addressing the bug-fixes, bugs that were introduced due to a lack of experience, as well as the huge amount of data. Having to work with almost 3 million scripts posed a great challenge to bug detection, which motivated me to implement execution-monitoring mechanisms that would display in a succinct and relevant way the real-time data regarding the experimental results. This, coupled with the implemented functions that iterate over the obtained data and analyze their coherence, allowed me to monitor the experimental process, detect any bugs and validate the acquired data–ultimately, enabling me to implement a robust experimental pipeline.

6.1.3 Experimental resumption mechanism

Another important aspect that demanded substantial effort was the implementation of experimental *resumption* mechanisms. The experience gained from the previous internship showed me that it is vital to have a mechanism that allows the user to resume an experiment without starting from scratch. When working with big amounts of data or launching computationally-taxing programs, the time necessary to complete an experiment can span over multiple hours or even days. When starting the implementation of an experimental pipeline, bugs are almost guaranteed to appear in the first versions of the framework–either execution errors or incorrect functionality that leads to incoherent results. Restarting the experimental process every time a problem occurs is very tedious and takes too much valuable time. This proves the necessity of employing an experimental *resumption* mechanism.

Furthermore, two crucial aspects of such a system are its performance and the way it is implemented. The mechanism must have a good performance in order to justify its use in the first place: if it takes too much time to find the continuation point of the experiment, the use of a resumption system is irrelevant. In a best case scenario, the mechanism will have a constant complexity O(1), which, luckily, I managed to achieve for the *functionality test* and the *metric measurement* of the scripts–the two components of the experimental pipeline that take (by far) the most amount of time. While the performance is an important factor, the manner in which the mechanism is implemented is arguably equally important. If the implementation of the *resumption* protocol is: too big in size or complexity; hard to read, understand and/or modify and maintain–the pay-off of having such a system is negligible.

6.1.4 Data structuring

An additional factor that contributed to the efficient work rhythm–the ease of building the framework and analyzing the acquired results–was the implementation of a coherent data-structuring system (e.g., the folder system, the file nomenclature, the structure of the JSON and CSV files). This allowed me to organize the work more efficiently and easily analyze the experimental data. A good directory structure, explicit file nomenclature and organized format of the acquired data facilitated both the coding process as well as empirical observations of the results. When coding, the coherent structure allowed me to easily iterate over the dataset in an orderly and relevant manner and accomplish the desired tasks (functionality tests, metric measurement, etc.), all the while facilitating the process of navigating the folders with the OS' file explorer GUI and observing the results in a more direct and visual manner. Additionally, the good folder structuring allowed for a better performance of the framework: clustering the relevant data and iterating over it in a well-planned approach makes good use of the CPU cache and the RAM, and avoids any overhead of unnecessary disk read/write operations.

6.2 Academic-related work

Taking into consideration that this was a *research* internship, a substantial amount of effort was focused on aspects that are not directly related to programming–or, rather, to *producing* code. As the name implies, *research* is the act of studying and experimenting with aspects of a scientific field that have yet to be covered. This includes experimenting on "uncharted grounds", studying academic literature, as well as *presenting* the obtained knowledge/results to other researchers.

6.2.1. Research

During this internship, a considerable amount of time was invested in reading and understanding research papers [10] that touch on the subject of Large Language models, especially in the earlier stages. Prior to this, I had some experience studying and working with LLMs during the previous internship. The research subject was the use of OpenAI's API for ChatGPT and program synthesis. The main goal was to determine if ChatGPT is capable of producing *robust* code based solely on a set of input/output examples. For a better understanding of the technology, I read a scientific article [11] that touches on multiple aspects of Large Language Models (e.g., artificial networks, word distribution, predictive models). While it was a good foundation, a deeper dive was necessary for a clearer image of the picture.

The current internship started with a first attempt of researching the state-of-the-art Large Language Models specialized in programming: analyzing their specifications, the number of parameters, their performance with different programming benchmarks, etc. Once this was done, I started studying their architecture. I was eager to understand what makes them so versatile and powerful in certain situations. Special attention was given to the study of their *attention mechanism*, which allows them to link different words from the input and enables them to better "understand" what they are asked. This is the core component of LLMs that enables them to read and generate: natural text, code, mathematical equations, etc. While this study was not necessary for the internship, it offered a better grasp of the subject, as well as served as a good mental-training exercise that prepared me for future research.

This was followed by the study of textual-metrics. It was necessary to understand and be able to explain how they work and what impacts the final score. Furthermore, it was important to establish the textual differences between natural text and code, and how these metrics would react to this dissimilarity. This allowed me to analyze and explain the experimental results, and, ultimately, draw conclusions. But not without the help of machine learning.

Although I have studied Artificial Intelligence at the university, and have seen some of the concepts that were discussed in this report, this was my first experience actually training and testing a predictive model. It took a substantial amount of time and mental resources in order to analyze the different performance scores that the predictive model outputs. It was important to link these results with the empirical observations about the utilized dataset in order to answer some of the research questions posed during this study.

6.2.2 Expression skills

As mentioned previously, research includes not only the study of a certain science, but also the need to be able present the acquired knowledge. This requires a certain set of expression skills: preparing a talk, creating comprehensive and informative slides, giving a presentation in front of a public, writing academic literature, etc. All these skills were also employed throughout the internship.

Soon after starting the research, I gave a presentation of myself, my background and my research subject in front of the team, as well as answered a few of the subsequent questions. Although it was a short talk (only 20 minutes), it was a good first experience in oratory. Additionally, I have participated in a reading group focused on LLM, where I presented a research paper [12] on the types of bugs introduced by generative-AI. Furthermore, I have created a poster detailing the global picture of the research subject for the team seminar, as well as presented it in front of a few members of the team.

7. Future work

Although the internship comes to a conclusion, my supervisors and I will continue the research of this subject. The work that was accomplished in the last 6 months will serve as a solid foundation for the future work we plan to do. We have numerous ideas for the future direction of this research, which will be listed in this section.

7.1 CodeBLEU parameterization

As mentioned previously, the CodeBLEU metric was applied with the default parameters. The experimental results point to the fact that the textual side of the code is not suitable for determining its “quality”, and more weight should be focused on the syntactic and semantic side of the code. We think that machine learning can be employed in order to determine the best CodeBLEU parameters, which might yield the most coherent and relevant scores.

7.2 Combine metrics

It is possible that combining the results of two or more metrics might yield better results compared to employing them separately. While the performance of classifiers was somewhat similar among the studied metrics, it does not mean that they outputted the same score for a given script. It is possible that different metrics capture different valuable features, and combining them might yield better overall performance. Once again, machine learning will help us determine if the combined metrics offer better results than applying them separately.

7.3 Decision boundary

Although the decision boundary is generally set to 0.5 when applying logistic regression, some academic literature points to the fact that using different decision boundaries may prove fruitful in particular situations. One such case is when the dataset is unbalanced, which coincides with our context.

7.4 Add other programming languages

The scripts that were studied during the internship were limited to Python, mainly because all the programming was done in the same language and it is relatively easy to execute external Python scripts in an isolated subprocess for the purposes of script *validation and verification*. However, the possibility to analyze scripts from other programming languages would be a welcome addition to the framework. Executing scripts written in a different language might prove challenging but worth looking into.

7.5 Improve the experimental framework

One important goal of this research was the creation of a framework that would allow users to test and compare the performance of different existing metrics–or even test newly created ones–on any desired benchmark. The best case scenario would be implementing a benchmark that demands no code refactoring in order to adapt it to different needs: just input the desired metric and benchmark and acquire all the performance results. A substantial amount of the effort will go into ensuring that the framework can be easily installed and utilized in any environment.

8. Conclusion

This report details the research of textual-based metrics and their performance as a test proxy. The results obtained thus far show underwhelming performance, but have not pointed to a definite answer as of yet. The underlying nature of textual metrics might make them unsuitable for code analysis, but combining the metrics might yield better results. Further research with different benchmarks and programming languages will enable us to give a clearer answer regarding their relevance as a proxy test.

Bibliography

[1] BLEU GitHub repository. https://github.com/huggingface/evaluate/tree/main/metrics/bleu
[2] CodeBLEU GitHub repository. https://github.com/k4black/codebleu
[3] ROUGE GitHub repository. https://github.com/huggingface/evaluate/tree/main/metrics/rouge
[4] METEOR GitHub repository.
https://github.com/huggingface/evaluate/tree/main/metrics/meteor
[5] ChrF GitHub repository. https://github.com/huggingface/evaluate/tree/main/metrics/chrf
[6] EvalPlus GitHub repository. https://github.com/evalplus/evalplus/releases/tag/v0.1.0
[7] Guide for the scikit-learn's Logistic Regression.
https://www.datacamp.com/tutorial/understanding-logistic-regression-python
[8] GitLab repository of the created framework.
https://gitlab.inria.fr/diverse/llm-program-synthesis
[9] GitLab repository of the dataset utilized during research.
https://gitlab.inria.fr/diverse-public/llm-program-diversification-ai-code
[10] GoogleDrive link to the repository of the researched papers.
https://drive.google.com/drive/folders/1i4tdgFo1ZXdQmI-BPohhGMNVePZe4pnw?usp=sharing
[11] Stephen Wolfram's explanation of the functionality of ChatGPT.
https://writings.stephenwolfram.com/2023/02/what-is-chatgpt-doing-and-why-does-it-work/
[12] Research paper about coding bugs made by Large Language Models.
https://arxiv.org/abs/2308.03109#