Media Engineering and Technology Faculty
German University in Cairo

Measurement of Functional Size Across Programming Languages: A Machine Learning Approach

Bachelor Thesis

Author: Mark Mahrous Besada

Supervisors: Dr. Milad Ghantous

Dr. Hassan Soubra

Submission Date: 19 May, 2024

Media Engineering and Technology Faculty
German University in Cairo

Measurement of Functional Size Across Programming Languages: A Machine Learning Approach

Bachelor Thesis

Author: Mark Mahrous Besada

Supervisors: Dr. Milad Ghantous

Dr. Hassan Soubra

Submission Date: 19 May, 2024

This is to certify that:

(i) the thesis comprises only my original work toward the Bachelor Degree

(ii) due acknowlegement has been made in the text to all other material used

---

Mark Mahrous Besada
19 May, 2024

Acknowledgments

I would like to express my sincere gratitude towards my supervisors, Dr. Milad and Dr. Hassan, for their motivation and encouragement, which kept me going through the coding and research. Also, I would like to thank them for their patience and time devoted to our meetings. Not to mention that their guidance and advice were always in the form of suggestions, not orders, which let me think as a leader, not a follower, and allowed me to make decisions throughout this research. It was a great experience working under his supervision. I also want to thank my parents for always supporting and encouraging me through the hard times.

Abstract

Functional Size Measurement (FSM) is pivotal in measuring software functionality. The COSMIC method, ISO 19761, stands as a second-generation FSM approach known for its high automation potential. Manual FSM poses challenges like high costs, time inefficiency, and human errors, prompting a shift towards automation. Leveraging novel Artificial Paradigms, this study introduces a "second-generation" automation tool prototype, dubbed GEN-COSMIC employing machine learning to generically measure COSMIC functional size across popular programming languages like C, Java, and Python. Through diverse machine learning models and methodologies, the tool demonstrates promising accuracy exceeding 90%, offering a robust solution to FSM challenges.

Contents

Acknowledgments **V**

1 Introduction **1**
1.1 Functional Size Measurement of Software 1
1.2 Standardized Functional Size Methods 1
1.3 The Use of COSMIC Method in Functional Size Measurement 2
1.4 Motivation 3
1.5 Aim 3
1.6 Outline 3

2 Background and Related Work **5**
2.1 A General Overview of Measuring Software Size 5
2.2 A General Overview of COSMIC Method 6
2.2.1 COSMIC measurement process 6
2.2.2 COSMIC Sizing Basics 7
2.2.3 Mapping Cosmic Elements to Arduino 8
2.2.4 Mapping Cosmic elements to Object Oriented Languages 9
2.3 Manual and Automation functional size measurement 9
2.4 Related work on COSMIC automation 10
2.5 General Overview of NLP 12
2.6 Related Work on NLP 13
2.6.1 Utilizing NLP for Automation 13
2.6.2 Utilizing NLP for COSMIC Functional Size Measurement 13

3 Methodology **15**
3.1 Dataset 15
3.1.1 Programming Languages Included 15
3.1.2 Loading and Shuffling the Dataset 15
3.1.3 Training and testing datasets 16
3.2 Machine Learning Models 17
3.2.1 Neural Network 17
3.2.1.1 TensorFlow and Keras 17
3.2.1.2 Data processing 18
3.2.1.3 Defining the Model 19

3.2.1.4 Training the Model . . . 19
3.2.2 Decision Tree . . . 20
3.2.2.1 Data Processing and Model Creation . . . 20
3.2.2.2 Training and Testing the Model . . . 21
3.2.3 Logistic Regression . . . 22
3.2.3.1 Data Processing and Model Creation . . . 22
3.2.3.2 Training and Testing the model . . . 23
3.2.4 Random Forest Classifier . . . 23
3.2.4.1 Data Processing and Model Creation . . . 24
3.2.4.2 Training and Testing the model . . . 24
3.3 Graphical User Interface (GUI) . . . 24
3.3.1 Cleaning the Code . . . 25
3.3.2 Browsing the Code File . . . 25
3.3.3 Measuring the Functional Size . . . 26
3.3.4 Tkinter GUI . . . 28

4 Results **29**
4.1 Performance Comparison . . . 29
4.2 Functional Size Measurement of code snippets . . . 29

5 Conclusion and Future Work **35**

Appendix **36**

A Lists **37**
List of Abbreviations . . . 37
List of Figures . . . 39
List of Tables . . . 40

References **44**

Chapter 1

Introduction

1.1 Functional Size Measurement of Software

Functional Size Measurement (FSM) stands as a fundamental practice in software engineering, offering a standardized approach to quantifying the functional requirements of software systems. It plays a crucial role in various phases of the software development lifecycle, including project estimation, resource allocation, and quality assessment [33] [38]. FSM enables organizations to gain insights into the complexity and scope of software projects, thereby facilitating informed decision-making and enhancing overall software development processes.

1.2 Standardized Functional Size Methods

At the time of writing this paper, there was five ISO standards conforming to ISO/IEC for the measurement of software functional size:

- **International Function Point Users Group (IFPUG)** [19]: IFPUG Function Points is one of the most widely used functional size measurement methods. It quantifies the functionality provided to users by software in terms of business-relevant features. Function points are calculated based on the counts of different types of transactions, data movements, and external interfaces within the software application.
- **Mark II (MkII)** [18]: The Mark II Function Point Analysis method is a functional size measurement method developed in the United Kingdom. It is based on the same principles as IFPUG Function Points but offers some modifications and additional rules to address specific needs and concerns. Mark II Function Points also measure the functionality of software applications based on user interactions and data movements.

- **Netherlands Software Measurement Association (NESMA)** [22]: NESMA Function Points is a functional size measurement method developed by the Netherlands Software Measurement Association. Similar to IFPUG Function Points, NESMA Function Points measure the functionality of software by quantifying user interactions, data movements, and external interfaces. However, NESMA Function Points may have slightly different counting rules and guidelines compared to IFPUG.
- **Finland Software Measurement Association (FiSMA)** [20]: FiSMA Function Points is a functional size measurement method developed by the Finland Software Measurement Association. Like IFPUG and NESMA, FiSMA Function Points measure the functionality of software applications based on user interactions, data movements, and external interfaces. It may have its own set of rules and guidelines tailored to the specific needs of Finnish software development practices.
- **Common Software Measurement International Consortium (COSMIC)** [21]: COSMIC Function Points is a functional size measurement method developed by the Common Software Measurement International Consortium. Unlike traditional function point methods like IFPUG, COSMIC measures functional size based on the fundamental processes carried out by the software, such as data movements, data processing, and data stores. It aims to provide a more universal and standardized approach to functional size measurement across different types of software applications and domains.

Each of these functional size measurement methods has its own set of rules, guidelines, and counting practices, but they all serve the common purpose of quantifying the functionality of software systems. Different organizations and countries may prefer one method over another based on their specific needs, preferences, and industry practices.

1.3 The Use of COSMIC Method in Functional Size Measurement

While the first four methods which are IFPUG, MkII, NESMA and FiSMA are known to be "first generation" functional size methods, the COSMIC method is the only one known as a "second generation"[13] method, able to measure various kinds of software (information systems, Web apps, Mobile apps, embedded real-time, SOA components, etc.).

Among the various methodologies for FSM, the COSMIC (Common Software Measurement International Consortium) method has emerged as a widely adopted standard [4]. COSMIC methodology emphasizes a functional-centric approach, focusing on the user's perspective to define and measure functional size. It provides a comprehensive framework for quantifying the functional requirements of software systems, encompassing both data and transactional functionalities. By leveraging COSMIC, organizations

can achieve greater accuracy and consistency in functional size measurement, thus improving the reliability of project estimations and resource planning.

1.4 Motivation

The motivation behind this research stems from the increasing complexity and diversity of modern software systems, which creates significant challenges to traditional FSM methods. With the rapid advancement of technology and the spread of software applications across industries, there is a growing need for powerful and adaptable FSM tools. By addressing these challenges and leveraging the capabilities of ML, this study tries to provide a good solution that helps organizations to effectively manage and measure the functional size of any of their software systems.

1.5 Aim

The main aim of this study is to introduce an approach to functional size measurement called Gen-COSMIC. Building upon the COSMIC methodology, Gen-COSMIC aims to automate the process of FSM using advanced Machine Learning (ML) techniques. By developing a generic tool capable of measuring functional size across various programming languages and software domains, this study seeks to enhance the efficiency, accuracy, and scalability of FSM practices.

1.6 Outline

The rest of this thesis is organized as follows: Chapter 2 has some background information and introduces some related work. Chapter 3 introduces the proposed method, followed by the results in Chapter 4. Finally, Chapter 5 reaches a conclusion followed by possible future directions.

Chapter 2

Background and Related Work

2.1 A General Overview of Measuring Software Size

Software plays a ubiquitous role in modern society, with its effective management and measurement being crucial for most organizations. Also measuring software sizing is useful for a lot of purposes other than project estimation [33] [38]. Understanding the size of software is essential for managing the work involved in its creation and maintenance. Through accurate measurement, organizations can estimate, control, plan, and improve software work processes more efficiently. Measurement, Key Performance Indicators (KPIs), and other metrics are commonplace in effective organizations, with COSMIC functional sizing serving as a foundational metric for measuring software work.

The primary reason for measuring software size is to estimate the effort or cost required for its development. While size is not the sole determinant of development effort or cost, it significantly influences the workload. The estimated effort is inversely proportional to productivity, where productivity data can be derived from completed projects.

Measuring software sizes offers various valuable purposes beyond project estimation. These include:

- **Comparison:** Organizations may want to compare productivity between Agile and traditional 'waterfall' project management approaches. To ensure consistency, the same measurement method is used for all project types.
- **Controlling scope, budget, and progress:** Tracking the size of new software as requirements evolve helps project managers manage 'scope creep,' thereby controlling the project budget and monitoring progress against budgetary constraints.
- **Controlling defect density:** Upon project completion, tracking defects discovered in the first month of operation and reporting the 'defect density' in defects per unit size helps assess software quality and identify areas for improvement.

The following mind map shows a range of possible uses of functional size measurements.

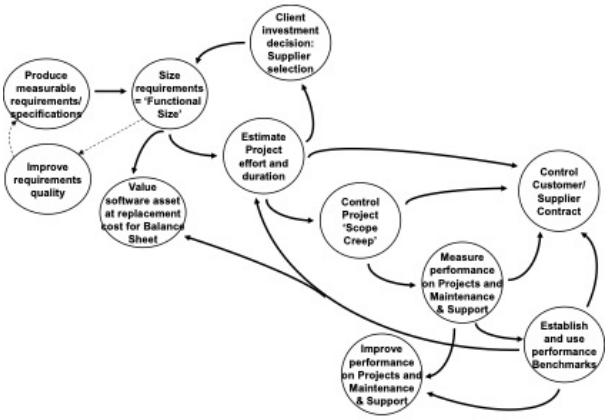

Figure 2.1: Mind map of possible uses of functional size measurements.

2.2 A General Overview of COSMIC Method

The COSMIC method establishes guidelines, regulations, and a systematic approach for quantifying the standard functional size of software. "Functional size" denotes the extent of functionality expressed in user-understandable terms, entirely unaffected by the technology employed in its development. Typically derived from requirements, COSMIC size can also be inferred from various software artifacts, such as designs or deployed systems.

2.2.1 COSMIC measurement process

COSMIC outlines a structured three-phase approach for measuring the functional size of software, which includes the measurement strategy phase, the mapping phase, and the measurement phase [11] [30]:

- **Measurement Strategy:** Here, emphasis is placed on identifying various parameters within the measurement context to ensure future interpretations of the measurement results are accurate. These parameters encompass the purpose of measurement and the scope of the software being measured. This phase produces the "Software Context Model," which involves identifying the software layers, their functional users, the documentation's granularity level, and the persistent storage associated with the software piece to be measured..
- **Mapping Phase:** This involves the alignment of Functional User Requirements (FUR) with the COSMIC "Generic Software Model." Each FUR corresponds to several functional processes (FP), each comprised of functional sub-processes responsible for data manipulation or movement. For example, a data movement can entail transferring a data group to or from a user (Entry and Exit data movements, respectively) or to or from a persistent storage (Read and Write data movements, respectively).

- **Measurement Phase:** In this phase, each data movement of a single data group is allocated a measurement unit of 1 CFP (COSMIC Function Point), as per the ISO standard. The software's functional size is determined by aggregating all identified data movements for each functional process.

The relationship of the three phases of the COSMIC method is shown below:

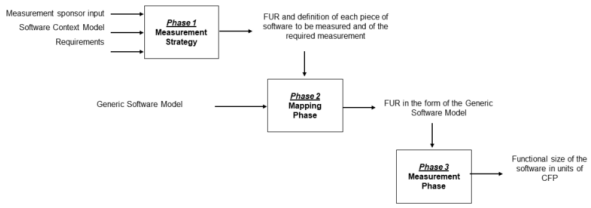

Figure 2.2: Measurement Process.

2.2.2 COSMIC Sizing Basics

- Each action involving data movement within the software corresponds to one COSMIC Function Point (CFP) [3].
- The actions include:
  - **Entry:**Data moving from a user into the software process across the boundary.
  - **Exit:**Data moving from the software process to the user across the boundary.
  - **Read:**Data moving from storage into the software process.
  - **Write:**Data moving from the software process into storage.

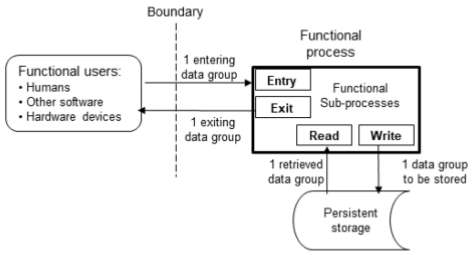

Figure 2.3: The four types of data movements.

- COSMIC follows a model called the 'COSMIC Generic Software Model', built on core software engineering principles.

- Key principles include [1]:
  - Analyzing software requirements into unique functional processes, composed of sub-processes representing data movements or manipulations.
  - Each functional process begins with an Entry data movement triggered by user input, signaling the software's response.
  - A data movement involves transferring attributes describing a single 'object of interest'.
  - For measurement purposes, data manipulation sub-processes are not individually measured.
- The size of software is determined by totaling the number of data movements (Entries, Exits, Reads, and Writes) across all functional processes, with each movement counted as one COSMIC Function Point (CFP).

2.2.3 Mapping Cosmic Elements to Arduino

Functional Process: Within the context of Arduino programming, the fundamental operational structure comprises two pivotal functions: setup() and loop(). The setup() function is executed only once, initializing the Arduino upon its initial power-up or reset. Within this function, the configuration of digital pins, delineating their roles as either inputs or outputs, is established leveraging functions such as pinMode(). Subsequently, the loop() function perpetually iterates, continuously monitoring sensor data and orchestrating the control of actuators in response to input variations. Any initiation of a functional process within this framework is recognized as an Entry data movement.

Soubra and Abran [37] presented a mapping of Cosmic elements to Arduino as follows:

- **Entry:** calling functions via input pins.
- **Exit:** calling functions via output pins.
- **Read:** calling EEPROM input functions.
- **Write:** calling EEPROM output functions.

Table 2.1 shows the mapping of COSMIC elements to Arduino, including examples.

Table 2.1: Mapping COSMIC elements to Arduino elements

COSMIC	Arduino	Examples
Functional Processes	setup() loop()	-

Entry data group movements	Functions via INPUT pin(s)	digitalRead();
		analogRead();
		Serial.readString();
		WiFi.getSocket();

Exit data group movements	Functions via OUTPUT pin(s)	digitalWrite();
		analogWrite();
		Serial.println();
		lcd.print();
		server.write();

Read data group movements	EEPROM Input functions	EEPROM.read(); EEPROM.get();
Write data group movements	EEPROM Output functions	EEPROM.write(); EEPROM.put();

2.2.4 Mapping Cosmic elements to Object Oriented Languages

Based on the rules previously mentioned, in Object Oriented Languages, the four data movements Entry, Exit, Read, and Write are defined as follows:

- **Entry:** create a function or get a user input.
- **Exit:** print or return any data.
- **Read:** read data from a file.
- **Write:** writing data to a file.

Table 2.2 shows the mapping of COSMIC elements to Object Oriented Languages, including examples.

2.3 Manual and Automation functional size measurement

Manual measurement of functional size of software faces several challenges. Firstly, it requires a significant amount of effort, particularly for those who lack experience in the

Table 2.2: Mapping COSMIC elements to Object Oriented Languages

COSMIC	Object Oriented Languages	Examples
Entry data group movements	create a function get a user input	public static void main() function add() int x = scanner.nextInt();
Exit data group movements	print data return data	print(); System.out.println(); return ;
Read data group movements	read data from a file	line = reader.readLine()
Write data group movements	writing data to a file	writer.write(data);

process. Even with expertise, tackling ambiguous or incomplete requirements further complicates the task, as it necessitates frequent consultation with project members. Secondly, inexperienced measurers are prone to errors, mainly due to the inherent ambiguities within the requirements [40]. Also Manually implementing FSM procedures can be time-consuming, posing challenges for organizations handling numerous projects within tight time frames, whether for project estimation or productivity analysis [35]. Moreover, applying FSM manually to a vast array of source code inputs demands specialized skills, particularly when dealing with multiple programming languages in which the code is written.

These issues pose significant implications for software project indicators. As a result, there has been a growing interest in automating the measurement of functional size to address these challenges [30]. Automating functional size measurement can be done through main ways [17], either automate requirements analysis where the measurement is based on grammar patterns and key words as the input is a set of textual or modeled requirements and the output is the functional size measured, or automated code analysis as the input is a source code and the output is the functional size measured.

Organizations often choose to measure from the software code to establish a baseline for the relationship between size and effort, thereby enabling the development of estimation models based on functional size. Additionally, measuring from code offers a more accurate reflection of what has been delivered, compared to relying on incomplete, ambiguous, or inadequate requirements. This approach allows for a more reliable assessment of requirements-based sizing.

2.4 Related work on COSMIC automation

Interest in automating the COSMIC method has grown within the realm of software project management. Automation offers practitioners the advantage of objectively and swiftly measuring their software's size throughout its lifecycle, starting from the initial

development stages [7] [9] [24] [25] [10] [26] [34] [12] to the later coding phases [5] [6] [31] [32] [16] [39].

Akca et al. [5] introduced a semi-automated method for measuring the functional size of source code in a "three-tier Java business application" utilizing COSMIC version 3.0.1. This approach involved employing a custom "measurement library" containing functional processes, which were activated through the application's graphical user interface (GUI). The outcomes of this measurement approach yielded an accuracy rate of 92% when compared to manual measurements.

Akca et al. [6] suggested conducting a comparative analysis of the expenses associated with semi-automatic and manual measurements across three case studies. Their findings revealed that integrating the automatic process at the onset of the coding phase could lead to a reduction in measurement costs of up to 280% when compared to manual measurement methods.

Sag et al. [31] [32] introduced a COSMIC measurement tool named 'Cosmic Solver' designed to measure either source or binary code. They put forward several rules to generate UML Sequence Diagrams from the software's execution during runtime using AspectJ technology. The functionality of 'Cosmic Solver' was showcased using a sample three-tier Java business application. The prototype extracted functional size measurements that were convergent to 96.8% when compared to those obtained through manual measurement techniques.

Gonultas et al. [16] proposed the automation of COSMIC for GUI Web Java business applications built on a three-tier architecture. Similar to the approach in [5], the automation process involved the development of a "Measurement Library," which necessitated the installation of Java application code. However, this library was specifically tailored for applications utilizing JSF, Spring, and Hibernate technologies. The automated measurement approach demonstrated a convergence rate of 94% when compared to manual measurements, alongside a significant reduction in measurement duration by approximately 97% (equivalent to 1/34 of the effort required for manual measurement). Minor deviations in accuracy were attributed to the specific technology and parsing methods utilized.

Soubra et al. [36] developed a theoretical 'universal' tool grounded in COSMIC ISO 19761 for automating the measurement of software written in various programming languages. Their research involved creating a prototype tool that integrated COSMIC and MIPS, which underwent a small-scale validation process. However, the scope of their work was constrained to a specific release of the MIPS architecture and a particular instruction set.

Ahmed et al. [14] presented a functional size measurement (FSM) procedure utilizing COSMIC ISO 19761 to evaluate software artifacts represented in ARM's base 32-bit assembly code. They introduced a prototype of an automated measurement tool capable of determining the functional size, expressed in COSMIC Function Points (CFP), for ARM programs. However, it's noteworthy that their tool did not encompass all ARM instructions.

Sahab et al. [30] created the CFP4J library to automate COSMIC functional size measurements specifically for Java web applications employing the Spring Web MVC framework. Their innovation is evident in establishing mapping guidelines from code and providing a publicly accessible software library designed to streamline the automation of COSMIC functional size evaluations for Java web applications utilizing the Spring MVC framework.

Bagriyanik et al. [8] introduced an ontology model aimed at converting requirements into COSMIC function point method concepts. They also devised a method to automatically assess the functional size of software by leveraging the created ontology. To validate their approach, they implemented a software application utilizing requirement data sourced from multiple real-world projects. Their study revealed that both manual and automated measurement outcomes aligned effectively.

Meiliana et al. [27] employed the XML structure of UML sequence diagrams to facilitate the automation of software size measurements. They utilized the COSMIC method for measuring functional size, while the structural size was determined by analyzing the control structures present within the sequence diagrams.

Zaw et al. [42] introduced an automated tool for measuring software size, incorporating a generation model founded on UML, SysML, and Petri nets. They put forward overarching mapping guidelines between the COSMIC Functional Size Measurement (FSM) and the generation model to facilitate the measurement of software size.

Abrahão et al. [2] established a measurement procedure termed OO-HCFP for Object-Oriented Hypermedia (OO-H) web applications, drawing on COSMIC principles. They contended that the outcomes derived from their proposed methodology were notably more precise compared to those obtained through alternative measurement approaches grounded in function points and design measures.

De Vito et al. [15] presented a measurement procedure aimed at deriving COSMIC functional size from UML software artifacts. They also developed a prototype tool called J-UML COSMIC. To evaluate the measurement procedure, they conducted two case studies and compared the measurements obtained by the tool with those determined by experts using the standard COSMIC method. Their findings indicated that the tool facilitated incremental accurate measurements, particularly when considering new or existing models.

2.5 General Overview of NLP

Natural Language Processing (NLP) stands at the intersection of artificial intelligence, computational linguistics, and machine learning, employing deep learning models to enable computers to comprehend human language. NLP encompasses various tasks facilitating computer understanding of human language. These tasks include:

- Speech recognition, which involves accurately transcribing spoken words into text, commonly used in applications that respond to voice commands.

- Word sense disambiguation, enabling the determination of a word's meaning by considering its context within the text, especially when it has multiple interpretations.
- Part-of-speech tagging, identifying the grammatical category of each word based on its context in the text.
- Named entity recognition, which identifies and classifies words representing significant entities.

These NLP tasks find application in various fields. For example, sentiment analysis analyzes user comments and reviews to infer their sentiment towards products, events, etc. Additionally, NLP aids in spam detection by employing text classification techniques to identify fraudulent or malicious emails based on specific words or patterns. Virtual agents and chatbots represent another application of NLP, utilizing speech recognition and natural language generation to interact with users, responding appropriately to their requests or comments.

2.6 Related Work on NLP

2.6.1 Utilizing NLP for Automation

The utilization of Natural Language Processing (NLP) has facilitated the automation of various applications. For example, [29] offered an overview of how artificial intelligence, including NLP, has contributed to the advancement of robotics, enabling the automation of specific tasks and enhancing human-robot interaction through NLP and other machine learning algorithms. Additionally, [23] leveraged NLP to extract information from electronic health records and predict the presence of lung cancer accordingly. Moreover, [28] conducted a literature review on the use of NLP for automating responses to customer queries, highlighting various domains where NLP can be beneficial for automatically addressing customer inquiries and discussing the advantages it offers to businesses.

2.6.2 Utilizing NLP for COSMIC Functional Size Measurement

In the research outlined in [41], a COSMIC Functional Size Measurement (FSM) automation tool called ScopeMaster was introduced. This tool employed Natural Language Processing (NLP) and pattern matching techniques to gauge the size of a functional requirement, along with identifying functional users (objects of interest) and data group movements based on CRUDL (Create, Read, Update, Delete, List) operations. Moreover, the tool had the capability to identify ambiguous requirements that could potentially lead to future defects.

Chapter 3

Methodology

3.1 Dataset

The dataset utilized in this study comprises manually generated and collected data from various sources of code. It consists of 1000 samples, each representing a single line of code from different programming languages, along with its corresponding classification as either CFP or not CFP. For model training, 900 samples are allocated, while the remaining 100 samples are reserved for testing purposes.

3.1.1 Programming Languages Included

- **Arduino:** This programming environment is primarily based on the C/C++ language. However, Arduino offers a simplified version of C/C++, supplemented with its own libraries and functions, catering to novices and non-experts in programming.
- **C++:** A versatile, cross-platform language suitable for developing high-performance applications.
- **Java:** A general-purpose, high-level, object-oriented language designed to minimize implementation dependencies.
- **JavaScript:** Widely regarded as the most prevalent programming language, particularly in web development.
- **Python:** A high-level, general-purpose language supporting various programming paradigms, including procedural, object-oriented, and functional programming styles.

3.1.2 Loading and Shuffling the Dataset

The dataset intended for model training is loaded using the Python Data Analysis Library "Pandas" post its importation. Each line of code's corresponding classification,

whether categorized as CFP or not, undergoes conversion into binary values. Consequently, the "Cosmic" column embodies binary representations, assigning 1 for instances labeled "CFP" and 0 for those labeled "not CFP." This conversion serves the purpose of preparing the data for binary classification, aligning with the objective of training a binary classification model to predict whether a given instance pertains to the "CFP" class or not. Subsequently, the dataset undergoes shuffling to eliminate any inherent order, such as the arrangement of programming language codes, ensuring the models are trained on generic, unordered data. The following figure illustrates the process of dataset loading and shuffling.

```
import pandas as pd

# Load the CSV data into a DataFrame
columns =["LOC","Cosmic"]
data = pd.read_csv('data.csv', names=columns)

data["Cosmic"]=(data["Cosmic"]=="CFP").astype(int)

# shuffle the data
data = data.sample(frac=1).reset_index(drop=True)
```

Figure 3.1: Loading and Shuffling the dataset.

3.1.3 Training and testing datasets

Upon loading and shuffling the dataset, it undergoes division into two distinct sets: the training dataset encompasses 90% of the data, while the testing dataset comprises the remaining 10%.

The training dataset serves as the foundation for training a machine-learning model. Through this dataset, the model assimilates and discerns patterns and correlations within the data, thereby enabling it to make accurate predictions for new inputs in subsequent phases.

Conversely, the testing dataset remains independent from the model's training process. It incorporates fresh data, unseen during the model's training phase, which facilitates the assessment of the trained model's performance. By inputting this data into the trained model, predictions are generated based on the acquired knowledge from the training dataset. The accuracy of these predictions is then gauged by comparing them against the actual outputs present in the testing dataset.

The key difference between training and testing datasets is that the training dataset is used to train the model, while the testing dataset is used to measure the performance of the model and its generalization ability to new unseen data. The following figure shows how the loaded dataset is split into 90% training dataset and 10% testing dataset.

```
sentences = []
labels = []
sum=0
# Collect sentences and labels into the lists
for item in data.LOC:
    sentences.append(item)
    sum=sum+1
for item in data.Cosmic:
    labels.append(item)

#creating training and testing data
training_size = int(sum*0.9)
training_sentences = sentences[0:training_size]
testing_sentences = sentences[training_size:]
training_labels = labels[0:training_size]
testing_labels = labels[training_size:]
```

Figure 3.2: Splitting dataset into training and testing datasets.

3.2 Machine Learning Models

A variety of models were employed in this research to conduct a comparative analysis of their performances and identify the most accurate one.

3.2.1 Neural Network

3.2.1.1 TensorFlow and Keras

TensorFlow 2 (v2.16.1 was used in this research) is an end-to-end, open-source ML platform. You can think of it as an infrastructure layer for differentiable programming. It combines four key abilities:

- Efficiently executing low-level tensor operations on CPU, GPU, or TPU.
- Computing the gradient of arbitrary differentiable expressions.
- Scaling computation to many devices, such as clusters of hundreds of GPUs.
- Exporting programs ("graphs") to external runtimes such as servers, browsers, mobile and embedded devices.

Keras is the high-level Application Programming Interface (API) of TensorFlow 2: an approachable, highly-productive interface for solving ML problems, with a focus on modern DL. It provides essential abstractions and building blocks for developing and shipping ML solutions with high iteration velocity.

Keras is a deep learning API written in Python, running on top of the ML platform TensorFlow. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result as fast as possible is key to doing good research. Keras is:

- **Simple:** Keras reduces developer cognitive load to free you to focus on the parts of the problem that matter.
- **Flexible:** Keras adopts the principle of progressive disclosure of complexity: simple workflows should be quick and easy, while arbitrarily advanced workflows should be possible via a clear path that builds upon what you've already learned.
- **Powerful:** Keras provides industry-strength performance and scalability: it is used by organizations and companies including NASA, YouTube, or Waymo.

In this research, the neural network model is developed using TensorFlow and Keras. TensorFlow provides the backend for training and executing the neural network, while Keras facilitates the quick prototyping and implementation of the model architecture.

3.2.1.2 Data processing

Data processing involves tokenizing the text data (lines of code) using the Tokenizer class provided by TensorFlow Keras. A tokenizer instance is created utilizing the imported Tokenizer class, which is then fitted on the training dataset to tokenize each word, assigning it to an integer. Consequently, every unique word in the dataset is mapped to a specific integer. The following figure illustrates the process of creating and fitting the tokenizer to the training dataset.

```
from tensorflow.keras.preprocessing.text import Tokenizer

vocab_size = 1000
oov_tok = "<OOV>"

# Tokenize the dataset
tokenizer = Tokenizer(num_words = vocab_size, oov_token=oov_tok)

# map words to numbers
tokenizer.fit_on_texts(training_sentences)
word_index = tokenizer.word_index
```

Figure 3.3: Creating Tokenizer.

After fitting the tokenizer to the training dataset, mapping each unique word to an integer, the tokenized data undergoes conversion into vectors utilizing the texts_to_sequences method offered by the Tokenizer class. Subsequently, the vectors are padded to ensure uniform length, employing the pad_sequences method provided by TensorFlow Keras. Should a vector exceed the specified maximum length, it is truncated; conversely, if it falls short, it is padded with zeros, thereby ensuring consistent length across all vectors. The following figure illustrates the process of converting the data into padded vectors.

```
from tensorflow.keras.preprocessing.sequence import pad_sequences

max_length = 20
trunc_type='post'
padding_type='post'

# Convert words to vectors
training_sequences = tokenizer.texts_to_sequences(training_sentences)
training_padded = pad_sequences(training_sequences, maxlen=max_length, padding=padding_type, truncating=trunc_type)

testing_sequences = tokenizer.texts_to_sequences(testing_sentences)
testing_padded = pad_sequences(testing_sequences, maxlen=max_length, padding=padding_type, truncating=trunc_type)
```

Figure 3.4: Converting data to padded vectors.

3.2.1.3 Defining the Model

Following the data preprocessing stage, the neural network model is defined utilizing the sequential model from TensorFlow's Keras API. The model comprises four layers, each sequentially executing specific operations:

- **Embedding Layer:** This layer learns to map each word in the vocabulary to a dense vector representation of fixed size (embedding_dim). The vocabulary's size is determined by vocab_size, and the length of input sequences is set to max_length. By capturing semantic relationships between words, this layer enables the model to comprehend the meanings of words within the input text.
- **Flatten Layer:** Responsible for converting the output of the embedding layer, which is a 2D tensor, into a 1D tensor.
- **Dense Layer:** As a hidden layer, it is fully connected, implying that each node (neuron) in this layer is connected to every node in the preceding layer. The number of nodes (neurons) in this layer is arbitrarily chosen. The ReLU activation function is employed in this hidden layer, facilitating the model's acquisition of intricate patterns.
- **Output Layer:** Serving as the final layer, it is also a dense layer with only one node (neuron), as it pertains to binary classification—determining whether a line of code is categorized as CFP or not. The sigmoid activation function utilized in this final layer constrains the output between 0 and 1, representing the probability of the input belonging to the CFP class.

Figure 3.5 depicts the neural network model defined with its four layers utilizing TensorFlow's Keras Sequential API.

3.2.1.4 Training the Model

Upon defining the model, it is configured for training using the compile method. This method requires three parameters: loss, optimizer, and metrics. In this scenario, 'binary_crossentropy' is selected as the loss function due to the binary classification nature

```
import tensorflow as tf

embedding_dim = 16

# Define the model
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim, input_length=max_length),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(24, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

Figure 3.5: Defining Neural Network model.

of the model. The 'adam' optimization algorithm is chosen as the optimizer for its efficiency and ease of implementation. The metric chosen for evaluation during training is 'accuracy'.

Subsequent to configuring the model with the compile method, training commences using the fit method. Both training and validation data are supplied to the fit method, allowing the model to undergo training on the complete training dataset over 30 epochs, while its performance is concurrently assessed using the validation data. The following figure illustrates the process of configuring the model with the compile method and training it using the fit method.

```
# Compile the model
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])

# Train the model
history = model.fit(training_padded, training_labels, epochs=30, validation_data=(testing_padded, testing_labels), verbose=2)
```

Figure 3.6: Training the model.

3.2.2 Decision Tree

The decision tree serves as a non-parametric supervised learning algorithm applicable to both classification and regression tasks. It adopts a hierarchical tree structure comprising a root node, branches, internal nodes, and leaf nodes. Within this research context, the Decision Tree algorithm is employed for classification purposes. The primary objective is binary classification, assigning either 1 or 0 to each line of code to discern whether it qualifies as a CFP or not.

3.2.2.1 Data Processing and Model Creation

The data processing approach utilized here employs a simple bag-of-words (BoW) method, specifically leveraging the CountVectorizer from scikit-learn. This tool transforms text data into a matrix of token counts, where each row represents a document—each line of code in this context—and each column signifies a unique word within the corpus. Each element in the matrix denotes the count of occurrences of the corresponding word

within the document. Consequently, this approach disregards word order and semantics, focusing solely on tallying word occurrences within each document.

Following the conversion of text data into a matrix of token counts using the CountVectorizer, it is inputted into a decision tree classifier for training. This classifier learns decision rules based on the token counts to predict whether a line of code entails data movement, thereby determining its classification as a CFP or not.

To streamline the process and ensure consistent preprocessing across training and testing data, a pipeline from the sklearn.pipeline module within the scikit-learn library is employed. This pipeline chains multiple steps sequentially, encapsulating preprocessing and model training into a single object, facilitating unified treatment as a single estimator. Pipelines offer the advantage of maintaining consistency in preprocessing steps across datasets and simplifying the model building and deployment process. Thus, a pipeline is constructed using the Pipeline class from scikit-learn, encompassing two sequential actions. The initial step involves the CountVectorizer, converting text data into a matrix of token counts, followed by the DecisionTreeClassifier, which trains a decision tree classifier on the vectorized text data. The following figure illustrates the creation of the pipeline, depicting its two sequential steps for text vectorization and model training.

```
from sklearn.pipeline import Pipeline
from sklearn.tree import DecisionTreeClassifier
from sklearn.feature_extraction.text import CountVectorizer

# Create a pipeline
decision_tree_pipeline = Pipeline([
    ('vectorizer', CountVectorizer()),
    ('classifier', DecisionTreeClassifier())
])
```

Figure 3.7: Data processing and DecisionTree Model creation.

3.2.2.2 Training and Testing the Model

Having constructed the pipeline as a unified estimator, it is now utilized to train the decision tree model on the training data employing the fit method, which accepts the training data. Consequently, the two sequential steps of the pipeline—text vectorization and training a decision tree classifier—are applied to the training data, facilitating the training of the decision tree model. Subsequently, this pipeline is poised for making precise predictions. Figure 3.8 shows the utilization of the created pipeline for training the decision tree model.

The performance of the decision tree model is assessed post-training to gauge its effectiveness. This evaluation employs data from the testing dataset. Utilizing the predict method of the pipeline, the model is tested by inputting testing data to predict the classification for each sample. Subsequently, the predictions are compared to the actual

```
# Fit the model
decision_tree_pipeline.fit(training_sentences, training_labels)
```

Figure 3.8: Training DecisionTree Model.

classifications of the testing data within the testing dataset. This comparison is facilitated by metrics such as accuracy_score, precision_score, and recall_score, imported from the metrics submodule of the Scikit-learn library. These metrics aid in evaluating various facets of the trained model's performance, including accuracy and other relevant factors. The following figure illustrates the process of evaluating the model's performance.

```
from sklearn.metrics import accuracy_score, precision_score, recall_score

# Make predictions
predictions = decision_tree_pipeline.predict(testing_sentences)

# Calculate the accuracy, precision, and recall
accuracy = accuracy_score(testing_labels, predictions)
precision = precision_score(testing_labels, predictions)
recall = recall_score(testing_labels, predictions)
```

Figure 3.9: Testing DecisionTree Model.

3.2.3 Logistic Regression

Logistic regression stands as a statistical method primarily employed for binary classification tasks. It predicts outcomes as true or false, yes or no, or 1 or 0. Despite its name, logistic regression is classified as a classification algorithm rather than a regression algorithm. This distinction is crucial, as logistic regression is adept at providing accurate predictions when confronted with binary classification tasks and linear decision boundaries. However it may not work well if the relation between the inputs and outputs is complex and non-linear. Logistic regression is popular for its simplicity, interpretability, and efficiency.

3.2.3.1 Data Processing and Model Creation

The data processing approach employed here mirrors that utilized during the training of the DecisionTree model, leveraging the bag-of-words (BoW) technique. The CountVectorizer is utilized for text vectorization. Similarly, a pipeline from the Scikit-Learn library is employed to concatenate steps of text vectorization, converting text data into a matrix of token counts. However, the divergence lies in the subsequent step: instead of routing the vectorized text data to the decision tree model, it is channeled to the logistic regression model, as depicted in the following figure.

```
from sklearn.pipeline import Pipeline
from sklearn.tree import LogisticRegression
from sklearn.feature_extraction.text import CountVectorizer

# Create a pipeline
logistic_regression_pipeline = Pipeline([
    ('vectorizer', CountVectorizer()),
    ('classifier', LogisticRegression())
])
```

Figure 3.10: Data processing and LogisticRegression Model creation.

3.2.3.2 Training and Testing the model

Once the pipeline is established, it undergoes training on a training dataset via the fit method, which accepts the training data as input. Subsequent to training, the logistic regression model is subjected to testing. Testing involves passing testing input from the testing dataset to the predict method, enabling the model to predict the classification of these inputs—whether they are classified as CFP or not. Subsequently, these predictions are compared with the actual labels in the testing dataset to evaluate the model's performance. This process mirrors the procedure undertaken with the pipeline created for training the decision tree, but this time utilizing the pipeline constructed with the logistic regression classifier, as depicted in the following figure.

```
# Fit the model
logistic_regression_pipeline.fit(training_sentences, training_labels)

from sklearn.metrics import accuracy_score, precision_score, recall_score

# Make predictions
predictions = logistic_regression_pipeline.predict(testing_sentences)

# Calculate the accuracy, precision, and recall
accuracy = accuracy_score(testing_labels, predictions)
precision = precision_score(testing_labels, predictions)
recall = recall_score(testing_labels, predictions)
```

Figure 3.11: Training and Testing LogisticRegression Model.

3.2.4 Random Forest Classifier

Random Forest is an ensemble learning method that can be used for both classification and regression purposes. A specific implementation of it that is dedicated to classification is the random forest classifier model; it's a part of the ensemble module that is available in the scikit-learn library. Random Forests are esteemed for several advantages, including their resilience to overfitting, capacity to handle high-dimensional data, and proficiency in capturing complex and non-linear relationships within datasets.

3.2.4.1 Data Processing and Model Creation

The data processing approach mirrors that employed during the training of the DecisionTree and LogisticRegression models, utilizing the bag-of-words (BoW) methodology. The CountVectorizer facilitates text vectorization, and a pipeline from the Scikit-Learn library is utilized to concatenate the steps of text vectorization, thereby converting text data into a matrix of token counts. However, the difference lies in the subsequent step: instead of routing the vectorized text data to the decision tree or logistic regression model, it is directed to the random forest classifier model, as depicted in the following figure.

```
from sklearn.pipeline import Pipeline
from sklearn.tree import RandomForestClassifier
from sklearn.feature_extraction.text import CountVectorizer

# Create a pipeline
random_forest_pipeline = Pipeline([
    ('vectorizer', CountVectorizer()),
    ('classifier', RandomForestClassifier())
])
```

Figure 3.12: Data processing and LogisticRegression Model creation.

3.2.4.2 Training and Testing the model

After setting up the pipeline, it's trained on a training dataset using the fit method, which takes in the training data. Following training, the random forest classifier model is tested by providing testing input from the testing dataset to the predict method. This allows it to predict whether these inputs are CFP or not. These predictions are then compared to the actual labels in the testing dataset to evaluate the model's performance. This process mirrors the one followed for training the decision tree and logistic regression, but this time with the pipeline created using the random forest classifier, as illustrated in Figure 3.13.

3.3 Graphical User Interface (GUI)

Following the training of models on the training dataset, the next step is to utilize these trained models to assess the functional size of any code written in any programming language present in the dataset. Since the models are trained on individual lines of code and their classification (CFP or not), evaluating the functional size of a code entails the trained models examining each line of code and predicting whether it falls under the category of CFP or not. If classified as CFP, the total cosmic function points increase by 1, indicating the presence of data movement in that line of code. Conversely, if not

```
# Fit the model
random_forest_pipeline.fit(training_sentences, training_labels)

from sklearn.metrics import accuracy_score, precision_score, recall_score

# Make predictions
predictions = random_forest_pipeline.predict(testing_sentences)

# Calculate the accuracy, precision, and recall
accuracy = accuracy_score(testing_labels, predictions)
precision = precision_score(testing_labels, predictions)
recall = recall_score(testing_labels, predictions)
```

Figure 3.13: Training and Testing RandomForestClassifier Model.

classified as CFP, the total cosmic function points remain unaffected. This iterative process continues until all lines of code are processed by the trained models, resulting in the final total cosmic function points serving as the measurement of the functional size of the code.

3.3.1 Cleaning the Code

Before predicting the classification of a line, it's essential to clean the line from any comments or extra spaces that may potentially confuse the models. The cleaned line can then be fed into the models for classification prediction. The clean method used for this purpose is implemented as depicted in the following figure.

```
# Clean each line of code
def clean_line(line):
    # remove any comments
    line = re.sub(r'//.*|#.*', '', line)
    # remove any extra spaces
    line = line.strip()
    return line
```

Figure 3.14: Clean method.

3.3.2 Browsing the Code File

A user-friendly interface is developed using Tkinter, a widely-used GUI (Graphical User Interface) toolkit for Python. This UI enables users to browse any code file, such as .py, .cpp, or .java, through a browse button that triggers the browse method. The implementation of the browse method is depicted in the following figure.

```
# Function to browse and select a file
def browse_file():
    file_path = filedialog.askopenfilename()
    if file_path:
        with open(file_path, 'r') as file:
            text = file.read()
            text_box.delete(1.0, tk.END)
            text_box.insert(tk.END, text)
```

Figure 3.15: Browse method.

3.3.3 Measuring the Functional Size

Once the code whose functional size needs measurement is obtained, users can press a "Measure" button to initiate the process. This involves several steps:

- The code is retrieved from the text box, which obtains the code from the code file.
- The code is split into lines to facilitate the prediction of each line's classification.
- Each line undergoes cleaning using the clean method.
- The cleaned line is then passed to the trained model to predict its classification.
- If the classification is identified as CFP, the total CFPs are incremented by 1; otherwise, it is skipped.
- Upon predicting the classification of all lines, the total CFPs are displayed, representing the functional size of the code.

Each model possesses its own measurement method, tailored to its unique training technique. The subsequent figures showcase the measure method for each model.

```
# Function to measure the CFPs in the text
def measure():
    text = text_box.get(1.0, tk.END)
    lines = text.split('\n')
    totalCFPs = 0
    for line in lines:
        line = clean_line(line)
        if line:
            sequences = tokenizer.texts_to_sequences([line])
            padded = pad_sequences(sequences, maxlen=max_length, padding=padding_type, truncating=trunc_type)
            prediction = model.predict(padded)
            threshold = 0.5
            if prediction > threshold:
                print(line)
                totalCFPs += 1
    totalCFPs_label.config(text='Total CFPs: ' + str(totalCFPs))
```

Figure 3.16: Measure Functional Size using Neural Network.

```
# Function to measure the CFPs
def measure():
    text = text_box.get(1.0, tk.END)
    lines = text.split('\n')
    totalCFPs = 0
    for line in lines:
        line = clean_line(line)
        if line:
            prediction = decision_tree_pipeline.predict([line])
            if prediction[0] == 1:
                totalCFPs += 1
    totalCFPs_label.config(text="Total CFPs: " + str(totalCFPs))
```

Figure 3.17: Measure Functional Size using Decision Tree.

```
# Function to measure the CFPs
def measure():
    text = text_box.get(1.0, tk.END)
    lines = text.split('\n')
    totalCFPs = 0
    for line in lines:
        line = clean_line(line)
        if line:
            prediction = logistic_regression_pipeline.predict([line])
            if prediction[0] == 1:
                totalCFPs += 1
    totalCFPs_label.config(text="Total CFPs: " + str(totalCFPs))
```

Figure 3.18: Measure Functional Size using Logistic Regression.

```
# Function to measure the CFPs
def measure():
    text = text_box.get(1.0, tk.END)
    lines = text.split('\n')
    totalCFPs = 0
    for line in lines:
        line = clean_line(line)
        if line:
            prediction = random_forest_pipeline.predict([line])
            if prediction[0] == 1:
                totalCFPs += 1
    totalCFPs_label.config(text="Total CFPs: " + str(totalCFPs))
```

Figure 3.19: Measure Functional Size using Random Forest Classifier.

3.3.4 Tkinter GUI

A simple user interface is created using Tkinter, which is a standard GUI (Graphical User Interface) toolkit for Python. The following figures show how the user interface is implemented and how it looks like.

```python
import tkinter as tk
from tkinter import filedialog

# Initialize Tkinter
root = tk.Tk()
root.title("Functional Size Measurement")
root.configure(bg="#f0f0f0")  # Set background color

# Create text box to display file content
text_box = tk.Text(root, height=20, width=70, bg="#ffffff", fg="#000000", font=("Arial", 12))
text_box.pack(pady=10)

# Button to browse for a file
browse_button = tk.Button(root, text="Browse", command=browse_file, bg="#008CBA", fg="#ffffff", font=("Arial", 12))
browse_button.pack(pady=5)

# Button to measure CFPs
measure_button = tk.Button(root, text="Measure", command=measure, bg="#4CAF50", fg="#ffffff", font=("Arial", 12))
measure_button.pack(pady=5)

# Label to display the result
totalCFPs_label = tk.Label(root, text="", bg="#f0f0f0", font=("Arial", 12, "bold"), fg="#FF0000")
totalCFPs_label.pack(pady=5)

# Run the Tkinter event loop
root.mainloop()
```

Figure 3.20: Implementation of the User Interface.

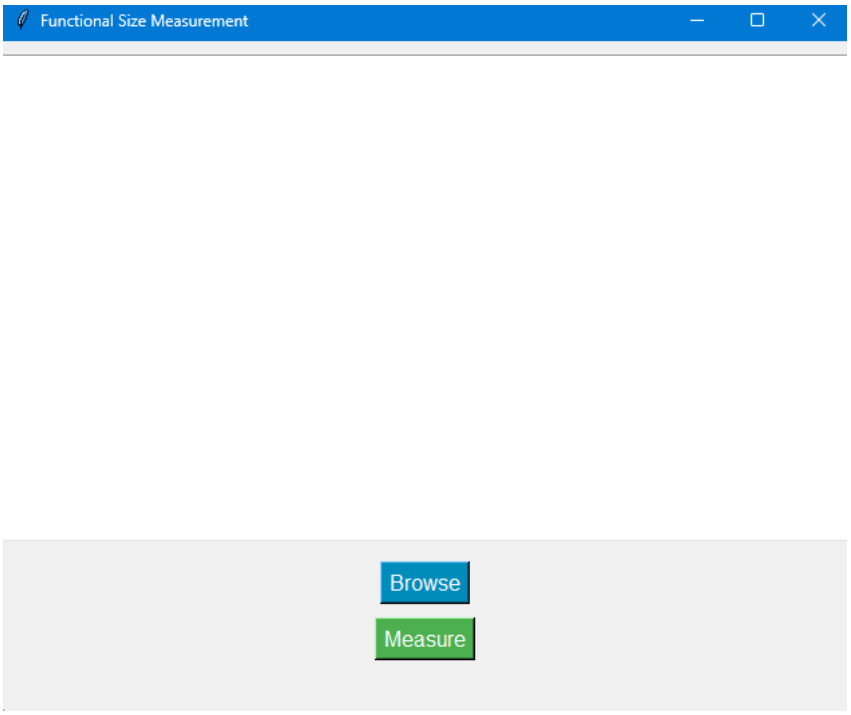

Figure 3.21: The User Interface.

Chapter 4

Results

The results section assesses the performance of the employed models to determine the most accurate ones with superior performance. Subsequently, the selected model is utilized to measure the functional size of diverse programs written in various programming languages, aiming to evaluate its accuracy in assessing the functional size of extensive codebases.

4.1 Performance Comparison

The performance evaluation of the selected models yielded promising results, with the neural network model emerging as the most accurate, achieving a remarkable accuracy rate of 95%. Table 4.1 presents a comprehensive overview of the performance metrics for all utilized models.

Table 4.1: Performance Comparison of Models

Model	**Accuracy**	**Precision**	**Recall**
Neural Network	95%	94%	98%
Decision Tree	88%	90%	84%
Logistic Regression	90%	91%	92%
Random Forest Classifier	90%	97%	82%

4.2 Functional Size Measurement of code snippets

An Arduino code underwent manual functional size measurement, yielding a result of 46 CFPs. Subsequently, this Arduino code was subjected to automatic functional size measurement using a trained neural network model. Remarkably, the neural network model provided an accurate measurement of 46 CFPs, demonstrating a 100% accuracy rate for this specific example. The following figures presents the Arduino code and the automatic functional size measurement process.

```
#include <ESP8266WIFI.h>

const char* ssid = "Your SSID"
const char* password = "Your Wifi Password"
int LM35 = A0;  //Analog channel A0 as used to measure tempreture
WifiServer server(80);

void setup(){
Serial.begin(115200);
delay(10);
// Connnectto Wifi network
Serial.println();
Serial.println();
Serial.print("Connecting to");
Serial.println(ssid);

Wifi.begin(ssid, password);
while(WiFi.status()!= WL_CONNECTED){
delay(500);
Serial.print(".");
}
Serial.println("");
Serial.println("WiFi connected");
//Start the server
server.begin();
Serial.println("Server started");
//print the IP address on serial monitor
Serial.print("Use this URL to connect");
Serial.print("https://")
Serial.println("/");
}

void loop(){
// Check if a client has connected
WiFiClient client = server.available();
if(!client){
return;
}

//Wait untill the client sends some data
Serial.println("new client");
while(!client.available()){
delay(1);
}
```

```
//Read the first line of the request
String request = client.readStringUntil("\r");
Serial.println(request);
client.flush();

//Match the request
float tempretureC;
float tempretureF;
int value = LOW;

if(request.indexOf("/Tem=ON") != -1){
float reading = analogRead(LM35);
float tempretureC = LM35/3.1;
//Serial.println("CENTI TEMP= ")
//Serial.println(tempretureC);
float tempretureF = ((tempretureC) *9.0/5.0)+32.0;
//Serial.println("FARE TEMP= ");
//Serial.print(tempretureF);

value = HIGH;
}

//Return the response
client.println("HTTP/1.1 200 OK");
client.println("Connect-Type: text/html");
client.println("<!DICTYPE HTML>");
client.println("<html>");
client.print("Celcius tempreture =");
client.print(tempretureC);
client.print("Fahrenheit tempreture =");
client.print(tempretureF);

if(value == HIGH){
client.print("Updated");
}else{
client.print("Not Updated");
}
client.print("<br><br>");
client.print("<a href= >");
client.print(<html>);
delay(1);
Serial.println("Client disconnected");
Serial.println("");
}
```

Figure 4.1: Arduino Code.

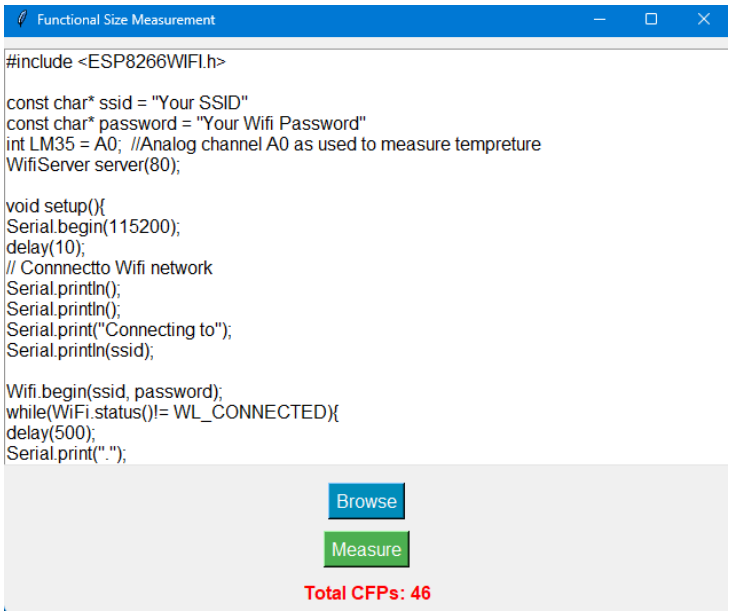

Figure 4.2: Functional size measurement of Arduino code automatically.

Another example of Java code, whose functional size was manually measured to be 26 CFPs, was subsequently fed into the trained neural network model to automatically assess its functional size. The neural network model returned a functional size estimation of 27 CFPs for this specific example, indicating a high level of accuracy at nearly 96%. The figures below depict the Java code snippet alongside the automatic measurement of its functional size.

```
import java.util.Scanner;

public class Calculator {

    // Method to add two numbers
    public static double add(double a, double b) {
        return a + b;
    }

    // Method to subtract two numbers
    public static double subtract(double a, double b) {
        return a - b;
    }

    // Method to multiply two numbers
    public static double multiply(double a, double b) {
        return a * b;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        double num1, num2;
        System.out.println("Enter first number:");
        num1 = scanner.nextDouble();
        System.out.println("Enter second number:");
        num2 = scanner.nextDouble();

        System.out.println("Choose operation:");
        System.out.println("1. Add");
        System.out.println("2. Subtract");
        System.out.println("3. Multiply");

        int choice = scanner.nextInt();

        switch (choice) {
            case 1:
                double sum = add(num1, num2);
                System.out.println("Sum: " + sum);
                break;
            case 2:
                double difference = subtract(num1, num2);
                System.out.println("Difference: " + difference);
                break;
            case 3:
                double product = multiply(num1, num2);
                System.out.println("Product: " + product);
                break;
            default:
                System.out.println("Invalid choice!");
        }

        // Example of if condition
        if (choice == 1 && sum > difference) {
            System.out.println("Sum is greater than difference.");
        } else if (choice == 2 && difference >= sum) {
            System.out.println("Difference is greater than or equal to sum.");
        }

        // Example of while loop
        int count = 0;
        while (count < 5) {
            System.out.println("Count: " + count);
            count++;
        }

        // Another example of if condition
        if (choice == 3 && product > 50) {
            System.out.println("Product is greater than 50.");
        } else if (choice == 3 && product <= 50) {
            System.out.println("Product is less than or equal to 50.");
        }

        scanner.close();
    }
}
```

Figure 4.3: Java Code.

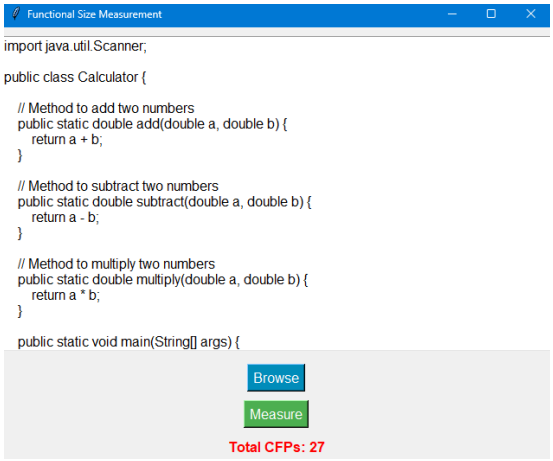

Figure 4.4: Functional size measurement of Java code automatically.

Another illustration involves a JavaScript code snippet whose functional size was initially measured manually at 17 CFPs. Subsequently, this code snippet was subjected to the trained neural network model for automatic functional size assessment. The neural network model returned a functional size estimation of 18 CFPs for this specific JavaScript example, showcasing a high accuracy rate of nearly 94%. The figures below present the JavaScript code snippet alongside the automatic measurement of its functional size.

```
// Function to add two numbers
function add(a, b) {
  return a + b;
}

// Function to multiply two numbers
function multiply(a, b) {
  return a * b;
}

// Function to subtract two numbers
function subtract(a, b) {
  return a - b;
}

// Function to get user input
function getUserInput() {
  const num1 = parseFloat(prompt("Enter the first number:"));
  const num2 = parseFloat(prompt("Enter the second number:"));
  return [num1, num2];
}

// Main function to perform operations
function main() {
  let continueCalculation = true;
```

```
  while (continueCalculation) {
    const operation = prompt("1. Add, 2. Multiply, 3. Subtract, 4. Quit");
    if (operation === "4") {
      console.log("Exiting...");
      continueCalculation = false;
    } else {
      const [num1, num2] = getUserInput();
      let result;

      if (operation === "1") {
        result = add(num1, num2);
        console.log("Result of addition:", result);
      } else if (operation === "2") {
        result = multiply(num1, num2);
        console.log("Result of multiplication:", result);
      } else if (operation === "3") {
        result = subtract(num1, num2);
        console.log("Result of subtraction:", result);
      } else {
        console.log("Invalid operation. Please choose again.");
      }
    }
  }
}
// Call the main function to start the program
main();
```

Figure 4.5: Javascript Code.

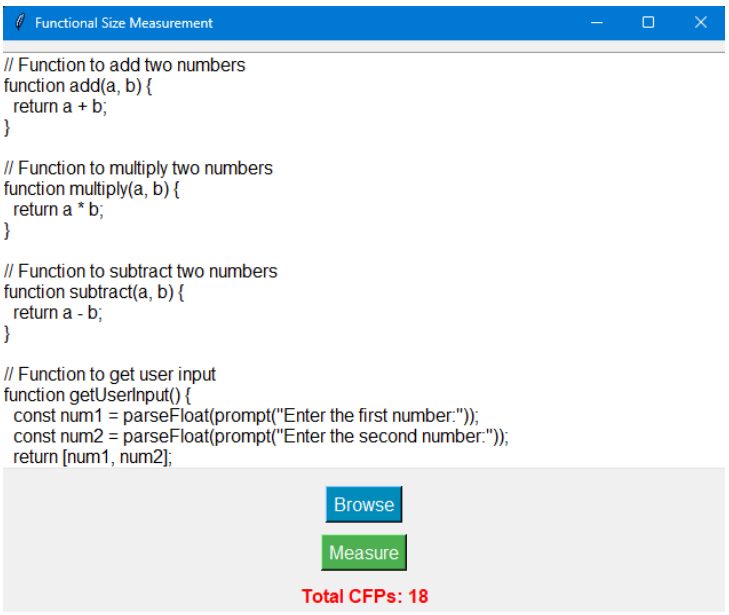

Figure 4.6: Functional size measurement of Javascript code automatically.

Another instance involves a C++ code snippet whose functional size was manually determined to be 16 CFPs. Subsequently, this code snippet underwent evaluation by the trained neural network model to automatically determine its functional size. Remarkably, the neural network model returned a functional size estimation of 16 CFPs for this specific C++ example, demonstrating an exceptional accuracy rate of 100%. The figures below illustrate the C++ code snippet alongside the automatic measurement of its functional size.

```cpp
#include <iostream>
#include <string>

using namespace std;

class MyClass {
private:
    int number;
    string name;

public:
    MyClass() {
        number = 0;
        name = "Default";
    }

    void setNumber(int num) {
        number = num;
    }

    void setName(string n) {
        name = n;
    }

    void printInfo() {
        cout << "Name: " << name << ", Number: " << number << endl;
    }
};
```

```cpp
int main() {
    MyClass obj;
    int num;
    string name;
    cout << "Enter a name: ";
    getline(cin, name);
    obj.setName(name);
    cout << "Enter a number: ";
    cin >> num;
    obj.setNumber(num);
    obj.printInfo();

    // Example of using if statement
    if (num > 0) {
        cout << "The number is positive." << endl;
    } else if (num < 0) {
        cout << "The number is negative." << endl;
    } else {
        cout << "The number is zero." << endl;
    }
    // Example of using while loop
    int i = 0;
    while (i < num) {
        cout << i << " ";
        i++;
    }
    cout << endl;
    return 0;
}
```

Figure 4.7: C++ Code.

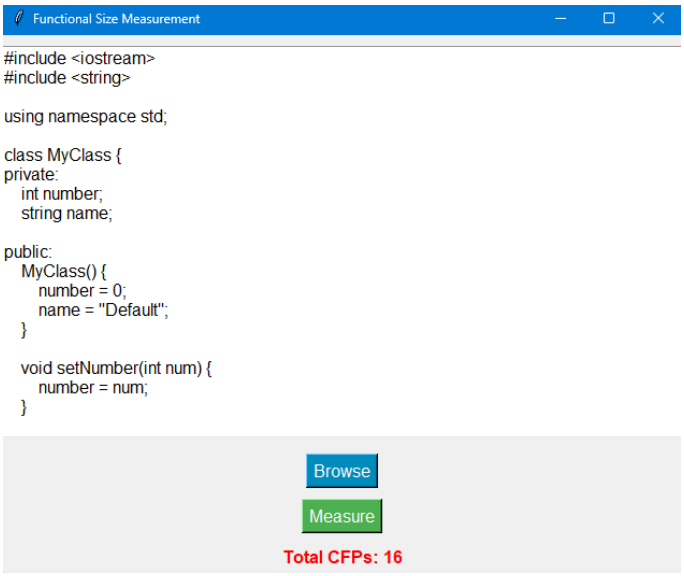

Figure 4.8: Functional size measurement of C++ code automatically.

Chapter 5

Conclusion and Future Work

This study introduces the GEN-COSMIC tool, designed to automate the measurement of functional size for code implemented in various widely-used programming languages. Leveraging diverse machine learning models including neural networks, decision trees, logistic regression, and random forest classifiers, coupled with distinct data processing techniques such as word embeddings and bag of words, the tool demonstrates promising results and performance metrics. Notably, the neural network model emerges as the most accurate, achieving an accuracy exceeding 95%, and successfully measuring the functional size of code snippets with accuracies ranging from 94% to 100%.

Future avenues of exploration include:

- Expansion of the dataset to encompass additional programming languages not currently represented, enhancing the tool's generality and applicability.
- Evaluation of alternative machine learning models, such as the Bert model, to potentially enhance performance and accuracy.
- Exploration of alternative data processing techniques, such as TF-IDF, to potentially improve performance and accuracy metrics.

These future directions aim to further refine and enhance the capabilities of the GEN-COSMIC tool, thereby advancing automated functional size measurement methodologies in software engineering practices.

Appendix

Appendix A

Lists

List of Figures

2.1 Mind map of possible uses of functional size measurements. 6
2.2 Measurement Process. 7
2.3 The four types of data movements. 7

3.1 Loading and Shuffling the dataset. 16
3.2 Splitting dataset into training and testing datasets. 17
3.3 Creating Tokenizer. 18
3.4 Converting data to padded vectors. 19
3.5 Defining Neural Network model. 20
3.6 Training the model. 20
3.7 Data processing and DecisionTree Model creation. 21
3.8 Training DecisionTree Model. 22
3.9 Testing DecisionTree Model. 22
3.10 Data processing and LogisticRegression Model creation. 23
3.11 Training and Testing LogisticRegression Model. 23
3.12 Data processing and LogisticRegression Model creation. 24
3.13 Training and Testing RandomForestClassifier Model. 25
3.14 Clean method. 25
3.15 Browse method. 26
3.16 Measure Functional Size using Neural Network. 26
3.17 Measure Functional Size using Decision Tree. 27
3.18 Measure Functional Size using Logistic Regression. 27
3.19 Measure Functional Size using Random Forest Classifier. 27
3.20 Implementation of the User Interface. 28

3.21 The User Interface. 28

4.1 Arduino Code. 30
4.2 Functional size measurement of Arduino code automatically. 30
4.3 Java Code. 31
4.4 Functional size measurement of Java code automatically. 31
4.5 Javascript Code. 32
4.6 Functional size measurement of Javascript code automatically. 32
4.7 C++ Code. 33
4.8 Functional size measurement of C++ code automatically. 33

List of Tables

2.1 Mapping COSMIC elements to Arduino elements 9
2.2 Mapping COSMIC elements to Object Oriented Languages 10

4.1 Performance Comparison of Models 29

Bibliography

[1] Cosmic official website.

[2] S. Abrahão, L. De Marco, F. Ferrucci, J. Gomez, C. Gravino, and F. Sarro. Definition and evaluation of a cosmic measurement procedure for sizing web applications in a model-driven development environment. *Information and Software Technology*, 14:144–161, 2018.

[3] Alain Abran et al. COSMIC Measurement Manual for ISO 19761, Part 1: Principles, Definitions & Rules, March 2020.

[4] Alain Abran, Charles Symons, Christof Ebert, Frank Vogelezang, and Hassan Soubra. Measurement of software size: Contributions of cosmic to estimation improvements. In *The International Training Symposium*, pages 259–267, Marriott Bristol, United Kingdom, 2016.

[5] A. Akca and A. Tarhan. Run-time measurement of cosmic functional size for java business applications: Initial results. In *International Workshop on Software Measurement and 7th International Conference on Software Process and Product Measurement - IWSM-MENSURA*, Assisi, Italy, October 2012.

[6] A. Akca and A. Tarhan. Run-time measurement of cosmic functional size for java business applications: Is it worth the cost? In *Joint Conference of the 23rd International Workshop on Software Measurement and Eighth International Conference on Software Process and Product Measurement - IWSM-MENSURA*, Ankara, Turkey, 2013.

[7] S. Azzouz and A. Abran. A proposed measurement role in the rational unified process (rup) and its implementation with iso 19761: Cosmic-ffp. In *Software Measurement European Forum - SMEF*, Rome, Italy, 2004.

[8] S. Bagriyanik and A. Karahoca. Automated cosmic function point measurement using a requirements engineering ontology. *Information and Software Technology*, 72:189–203, 2016.

[9] S. Barkallah, A. Gherbi, and A. Abran. Cosmic functional size measurement using uml models. In *Software Engineering, Business Continuity, and Education, Communications in Computer and Information Science, International Conferences ASEA,*

DRBC and EL 2011, Held as Part of the Future Generation Information Technology Conference, FGIT 2011, volume 257, 2011.

[10] V. Bévo, G. Lévesque, and A. Abran. Application de la méthode ffp à partir d'une spécification selon la notation uml: compte rendu des premiers essais d'application et questions. In *9th International Workshop Software Measurement - IWSM*, Lac Supérieur, Canada, 1999.

[11] Nadia Chamkha, Asma Sellami, and Alain Abran. Automated cosmic measurement of java swing applications throughout their development life cycle. In *IWSM/Mensura'18*, Beijing, China, September 2018.

[12] N. Condori-Fernandèz, S. Abrahão, and O. Pastor. On the estimation of software functional size from requirements specifications. *Journal of Computer Science and Technology*, 22:358–370, 2007.

[13] COSMIC. Second generation functional size measurement by design, 2020.

[14] A. Darwish and H. Soubra. Cosmic functional size of arm assembly programs. In *Proceedings of the 30th IWSM-Mensura*, Mexico City, Mexico, 2020.

[15] G. De Vito, F. Ferrucci, and C. Gravino. Design and automation of a cosmic measurement procedure based on uml models. *Software and Systems Modeling*, 19:171–198, 2020.

[16] R. Gonultas and A. Tarhan. Run-time calculation of cosmic functional size via automatic installment of measurement code into java business applications. In *41st Euromicro Conference on Software Engineering and Advanced Applications*, 2015.

[17] H. V. Heeringen. Automated function points, the game changer! In *IT Conference – ISBSG*, 2020.

[18] International Organization for Standardization (ISO). *Software Engineering - Mk II Function Point Analysis - Counting Practices Manual*. ISO, Geneva, Switzerland, 1st edition, December 2002.

[19] International Organization for Standardization (ISO). *Software and Systems Engineering - Software Measurement - IFPUG Functional Size Measurement Method 2009, 2nd Edn*. ISO, Geneva, Switzerland, 2009.

[20] International Organization for Standardization (ISO). *Information Technology - Systems and Software Engineering - FiSMA 1.1 Functional Size Measurement Method*. ISO, Geneva, Switzerland, 1st edition, 2010.

[21] International Organization for Standardization (ISO). *Software Engineering - COSMIC: A Functional Size Measurement Method*. ISO, Geneva, Switzerland, 2nd edition, 2011. Reviewed and confirmed in 2019.

[22] International Organization for Standardization (ISO). *Software Engineering - NESMA Functional Size Measurement Method - Definitions and Counting Guidelines for the Application of Function Point Analysis*. ISO, Geneva, Switzerland, 2nd edition, 2018.

[23] K. Jabir and A. Thirumurthi Raja. Prediction of lung cancer from electronic health records using cnn supported nlp. In F.J.J. Joseph, V.E. Balas, S.S. Rajest, and R. Regin, editors, *Computational Intelligence for Clinical Diagnosis*, EAI/Springer Innovations in Communication and Computing. Springer, 2023.

[24] M. S. Jenner. Automation of counting of functional size using cosmic-ffp in uml. In *12th International Workshop Software Measurement- IWSM*, Magdeburg, Germany, 2002.

[25] Z. Li, M. Nonaka, A. Kakurai, and M. Azuma. Measuring functional size of interactive software: a support system based on xforms-format user interface specifications. In *Third International Conference (QSIC'03)*, Dallas, Texas, 2003.

[26] B. Marín, O. Pastor, and G. Giachetti. Automating the measurement of functional size of conceptual models in an mda environment. In Andreas Jedlitschka and Outi Salo, editors, *Functional Size of Conceptual Models in an MDA Environment. the international conference on Product-Focused Software Process Improvement (PROFES '08)*, Berlin, Heidelberg, 2008. Springer-Verlag.

[27] Meiliana, S. Karim, S. Liawatimena, A. Trisetyarso, B. S. Abbas, and W. Suparta. Automating functional and structural software size measurement based on xml structure of uml sequence diagram. In *Proceedings of the IEEE International Conference on Cybernetics and Computational Intelligence (CyberneticsCom)*, pages 24–28, Phuket, Thailand, 2017. IEEE.

[28] P.A. Olujimi and A. Ade-Ibijola. Nlp techniques for automating responses to customer queries: a systematic review. *Discov Artif Intell*, 3(20), 2023.

[29] Abu Rayhan. Artificial intelligence in robotics: From automation to autonomous systems, 2023.

[30] Abdelaziz Sahab and Sylvie Trudel. Cosmic functional size automation of java web applications using the spring mvc framework. *Unknown*, 2020. Copyright ©2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY).

[31] M. A. Sağ and A. Tarhan. Measuring cosmic software size from functional execution traces of java business applications. In *Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement - IWSM-MENSURA*, pages 272–281, October 2014.

[32] M. A. Sağ and A. Tarhan. COSMIC Solver: A Tool for Functional Sizing of Java Business Applications. *Balkan Journal of Electrical and Computer Engineering*, 6:1–8, 2018.

[33] H. Sneed. Purpose of software measurement. *Software Measurement News*, 26(1), March 2021.

[34] H. Soubra, A. Abran, S. Stern, and A. Ramdan-Cherif. Design of a functional size measurement procedure for real-time embedded software requirements expressed using the simulink model. In *Joint Conference of the 21st Int'l Workshop on Software Measurement and 6th Int'l Conference on Software Process and Product Measurement - IWSM-MENSURA*, Nara, Japan, 2011.

[35] H. Soubra, Y. Abufrikha, and A. Abran. Towards universal cosmic size measurement automation. In *Proceedings of the 30th IWSM-Mensura*, Mexico City, Mexico, 2020.

[36] H. Soubra, Y. Abufrikha, and A. Abran. Towards universal cosmic size measurement automation. In *Proceedings of the 30th IWSM-Mensura*, Mexico City, Mexico, 2020.

[37] Hassan Soubra and Alain Abran. Functional size measurement for the internet of things (iot) an example using cosmic and the arduino open-source platform. In *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*, 2017.

[38] C. Symons. Why cosmic functional measurement? *Measurement News*, 25(2), September 2020.

[39] A. Tarhan, B. Özkan, and G. C. İçöz. A proposal on requirements for cosmic fsm automation from source code. In *Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement - IWSM-MENSURA*, pages 195–200, Berlin, October 2016.

[40] S. Trudel and A. Abran. Functional size measurement quality challenges for inexperienced measurers. In *International Workshop on Software Measurement (IWSM-Mensura) 2009*, pages 157–169, Heidelberg, 2009. Springer.

[41] Erdir Ungan, Colin Hammond, and Alain Abran. Automated cosmic measurement and requirement quality improvement through scopemaster®tool. In *IWSM-Mensura*, 2018.

[42] T. Zaw, S. Z. Hlaing, M. M. Lwin, and K. Ochimizu. An automated software size measurement tool based on generation model using cosmic function size measurement. In *Proceedings of the International Conference on Advanced Information Technologies (ICAIT)*, pages 268–273, Yangon, Myanmar, 2019. IEEE.