

The Islamic University of Gaza
Deanship of Research and Graduate Studies
Faculty of Information Technology
Master of Information Technology



الجامعة الإسلامية بغزة
عمادة البحث العلمي والدراسات العليا
كلية تكنولوجيا المعلومات
ماجستير تكنولوجيا المعلومات

Automatic Python Code Generation from Free Natural Language Text using Transformer-Based Machine Translation

التوليد الآلي للرموز البرمجية بلغة بايثون من النصوص المكتوبة
باللغة الطبيعية باستخدام الترجمة الآلية القائمة على المحولات

By
Mohammed Ahmed El-Agha

Supervised by
Dr. Motaz Saad
Assistant Professor of Computer Science

**A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of information technology at the Islamic University of Gaza**

December/2022

إقرار

أنا الموقع أدناه مقدم الرسالة التي تحمل العنوان:

Automatic Python Code Generation from Free Natural Language Text using Transformer-Based Machine Translation

التوليد الآلي للرموز البرمجية بلغة بايثون من النصوص المكتوبة
باللغة الطبيعية باستخدام الترجمة الآلية القائمة على المحولات

أقر بأن ما اشتملت عليه هذه الرسالة إنما هو نتاج جهدي الخاص، باستثناء ما تمت الإشارة إليه حيثما ورد، وأن هذه الرسالة ككل أو أي جزء منها لم يقدم من قبل الآخرين لنيل درجة أو لقب علمي أو بحثي لدى أي مؤسسة تعليمية أو بحثية أخرى.

Declaration

I understand the nature of plagiarism, and I am aware of the University's policy on this. The work provided in this thesis, unless otherwise referenced, is the researcher's work and has not been submitted by others elsewhere for any other degree or qualification.

Student's name:	محمد أحمد الأغا	اسم الطالب:
Signature:	Mohammed El-Agha	التوقيع:
Date:	20/1/2021	التاريخ:

نتيجة الحكم على أطروحة الماجستير

Abstract

An Automatic Code Generation (ACG) is a tool that generates a program code from high abstraction level input created by humans. Automatic Code Generation provides features of a highly effective and performed software development process with low time and cost to deliver a high-quality software product to a live production environment.

One of the most intelligent ACG approaches is Natural Language to Code Generation (NL-to-Code), which generates program code from natural language in unstructured form, using machine learning techniques. There are several proposed solutions and research in this field. However, most of them have limitations of restricted input NL text in length, syntactic correctness, or semantic correctness.

In this research, we investigate the existing studies on NL-to-Code generation, to address limitations and restrictions in proposed models. Also, by deductive research, we consider the problem of NL-to-Code generation as a Machine Translation problem, and we collect and build “PyCorpus” NL-code pairs dataset, and propose an approach for NL-to-Python-Code generation by a Transformer-Based Model with an Attention mechanism.

Experiments, that are conducted on “PyCorpus” dataset to generate the Transformer-Based Seq2Seq model, achieve METEOR score of 0.27, and BLEU score of 0.23 which considered a good result in Machine Translation field compared to the state-of-the-art NL-to-NL tasks, and taking into account the difficulty of the task tackled in this research of NL-to-Code. In addition, 43.6% of output codes, in the validation, are correctly compiled.

Keywords: *Natural language processing, Code generation, Automatic code generation, Machine translation, Transformer.*

ملخص الدراسة

التوليد الآلي للرموز البرمجية يعتبر تقنية حاسوبية تقوم بتوليد رمز برمجي من مُدخل يكتبه ويفهمه الإنسان. التوليد الآلي للرموز البرمجية يزيد من كفاءة العمل البرمجي ويحسن عملية تطوير البرمجيات من خلال تقليل تكلفة ووقت إنتاج برمجيات ذات جودة عالية، وتشغيلها على بيئة العمل الحقيقية.

أحد أنواع التوليد الآلي الذكي للرموز البرمجية هو (التوليد الآلي للرموز البرمجية من النصوص الطبيعية)، والذي يعني توليد رمز برمجي من نص طبيعي غير مقيد، باستخدام تقنيات تعلم الآلة. توجد عدة حلول وأبحاث في هذا المجال؛ ولكن يعاني معظمها من قيود ومحددات على النص الطبيعي المدخل، وطوله، وشكله، وصحته من الناحية النحوية، وصحته من الناحية البلاغية (صحة المعنى).

في هذا البحث، نقوم بعمل دراسة على الأبحاث الموجودة في هذا المجال؛ لتحديد المشكلات والمحددات في الحلول المقترحة فيها. نحن نعتبر المشكلة مشكلة (ترجمة آلية). ولهذا، نقوم بتجميع وبناء قاعدة بيانات خاصة بنا، والتي تحتوي على أزواج من النص الطبيعي والرموز البرمجية، وأسميناها "PyCorpus"، ونقترح منهج حال قائمة على المحولات.

قمنا بتطبيق الحل، وحصلنا على النتائج التالية: **METEOR score** ↓ 0.27، و **BLEU** ↓ 0.23، **score** الذي يعتبر نتيجة جيدة -في حقل الترجمة الآلية- مقارنة بالأعمال المشابهة المتعلقة بمهام إنتاج نصوص اللغة الطبيعية من مدخلات نصوص اللغة الطبيعية، مع الأخذ بعين الاعتبار صعوبة المهمة -التي يعالجها هذا البحث- والتي تقوم بإنتاج رموز برمجية من مدخلات نصوص اللغة الطبيعية. إضافةً إلى ذلك، 43.6% من الرموز البرمجية الناتجة كانت صحيحة.

الكلمات المفتاحية: معالجة النصوص الطبيعية، توليد الرموز البرمجية، التوليد الآلي للرموز البرمجية، الترجمة الآلية، المحولات.

Dedication

To everyone proud of me, and everyone who supports me in my master's study.

To someone who is my best, my whole and my supporter.

Thank you for all the love you gave me.

Acknowledgment

I thank Allah first and foremost.

To Dr. Motaz saad, thank you for being a great supervisor. I appreciate all the help you have given me with this research. Your guidance and instructions led me to complete this work effectively and professionally.

Table of Contents

Declaration	II
Abstract	IV
ملخص الدراسة.....	V
Dedication.....	VII
Acknowledgment.....	VIII
Table of Contents	IX
List of Tables.....	XI
List of Figures.....	XII
List of Abbreviations.....	XIII
Chapter 1 Introduction	2
1.1 Research Problem	4
1.2 Research Objectives	4
1.2.1 Main Objective.....	5
1.2.2 Specific objectives.....	5
1.3 Importance of the Work	5
1.4 Scope and Limitations of the Work	6
1.5 Research Approach	6
1.6 Overview of Thesis	6
Chapter 2 Background.....	9
2.1 NLP (Natural Language Processing)	9
2.1.1 NLP	9
2.1.2 Word Embeddings.....	9
2.1.3 Code Embeddings	10
2.1.4 Sequence Processing	10
2.1.5 Seq2Seq Model / Encode-Decoder Model	11
2.2 Code Generation	11
2.2.1 Definition of Code Generation.....	11
2.2.2 Types of Automatic Code Generation.....	12
2.2.3 Approaches of Automatic Code Generation	13
Chapter 3 Literature Review	18

3.1 Datasets	18
3.1.1 Existing NL-Code Datasets	18
3.1.2 Datasets Used in Existing NL-to-Code Research Works	20
3.2 Automatic Code Generation Approaches.....	21
3.2.1 Natural Language to Code	22
3.2.2 Template to Code	25
3.2.3 Pseudocode to Code	26
3.2.4 Code to Code.....	26
3.2.5 Model to Code.....	27
Chapter 4 Methodology	29
4.1 Dataset.....	29
4.2 Data Preparation	37
4.3 Training Model.....	38
4.3.1 Previous Attempts	38
4.3.2 The Structure of Training Model	39
4.3.3 Hyperparameters and Fine-tuning.....	42
4.4 Evaluation Metrics.....	45
Chapter 5 Experiments and Results	48
5.1 Experiments Tools	48
5.2 PyCorpus Dataset	49
5.3 Results and Discussion	61
5.4 Threats to Validity	64
Chapter 6 Conclusion and Future Works.....	67
6.1 Conclusion.....	67
6.2 Future Works.....	67
The Reference List	68

List of Tables

Table 3.1: Some of Common Existing NL-Code Datasets	19
Table 3.2: Used Datasets in State-of-the-Art NL-to-Code Research Works	20
Table 3.3: Overview of Related Works.....	21
Table 3.4: Overview of NL-to-Code Related Works.....	24
Table 4.1: Description of Source and Method of Collecting PyCorpus	34
Table 4.2: Description of Hyperparameters of our Model	42
Table 4.3: Values of Hyperparameters in the Pre-Trained Model.....	43
Table 4.4: Value of Hyperparameters after our Fine-Tuning.....	44
Table 5.1: The Statistics of the Dataset in the Model	49
Table 5.2: Some Examples of Various Structures of NL Inputs in PyCorpus	50
Table 5.3: Some Examples of Various Metaphors of NL Inputs in PyCorpus	51
Table 5.4: Results of Evaluation of our Experiments	64

List of Figures

Figure 1.1: Research Methodology (High-Level View)	6
Figure 2.1: Example of Vector Representation in Word Embeddings (Intro to Word Embeddings and Vectors for Text Analysis., 2018)	10
Figure 2.2: Abstract View of Sequence Processing for Text Generation (15, Enthusiast, Chaoran,; Enthusiast, 2019).....	11
Figure 2.3: Example of Converting NL to Code (Mihalcea et al., 2006).....	14
Figure 2.4: Example of Converting Template to Code (TextFixer, Text to HTML).....	14
Figure 2.5: Example of Converting Pseudocode to Code (Kulal et al., 2019)	15
Figure 2.6: Screenshot from CodePilot which Recommend JS Code according to existing Code (CodePilot.ai).....	15
Figure 2.7: Example of UML to Java code (Modeling and Code Generation - Yatta Solutions, 2022)	16
Figure 4.1: The Methodology Steps.....	29
Figure 4.2: Steps of Building PyCorpus.....	30
Figure 4.3: An Example of Code Exercise from a Webpage (Michele Pratusевич, 2014)....	31
Figure 4.4: An Example of Code Solution from GitHub Repository Source (Kaushik, 2022)32	
Figure 4.5: An Example of Python Code before and after Data Augmentation.....	38
Figure 4.6: Structure of Encoder in our Model (Shah, 2021).....	40
Figure 4.7: Structure of Decoder in our Model (Shah, 2021)	41
Figure 4.8: Mathematical Formula of BLUE Score (Doshi, 2021)	45
Figure 4.9: Mathematical Computation of % of Compiled Python Output Codes.....	46
Figure 5.1: A Sample from Generated Code from Our Model.....	62
Figure 5.2: A Sample from Generated Code from Our Model.....	62
Figure 5.3: A Sample from Generated Code from Our Model.....	63

List of Abbreviations

ACG	Automatic Code Generation
ER	Entity Relationship
SQL	Structural Query Language
NL	Natural Language
MDA	Model-Driven Architecture
SDL	Software Development Lifecycle
NLP	Natural Language Processing
BLEU	BiLingual Evaluation Understudy
METEOR	Metric for Evaluation of Translation with Explicit ORdering
ROUGE	Recall-Oriented Understudy for Gisting Evaluation
CS	Computer Science
AI	Artificial Intelligence
Seq2Seq	Sequence to Sequence
MVC	Model-View-Controller
HTML	HyperText Markup Language
CSS	Cascading Style Sheet
PHP	PHP: Hypertext Preprocessor
IDE	integrated development environment
UML	Unified Modeling Language
SCE	Supervised Code Embeddings
AST	Abstract Syntax Tree
Bi-LSTM	Bidirectional Long Short-Term Memory
MTT	Model-to-Text Transformation
UI	User Interface
IO	Input/Output
URL	Uniform Resource Locators
API	Application Programming Interface
REST	Representational State Transfer
sqrt	Square Root
PF	Position-Wise Feed-Forward

NIST	National Institute of Standards and Technology
NumPy	Numerical Python
NLTK	Natural Language Toolkit

Chapter 1

Introduction

Chapter 1 Introduction

Code Generation, in general terms, is the process of generating structured text from an input. The term of “Code Generation” holds different meanings according to the context of the technical field. Code Generation may refer to binary code from source code in the compilation of the program and may refer to traditional extraction of code for design model in software engineering (Sebastian et al., 2020; Paolone et al., 2020), and may refer to automatically generating program code from some inputs. Sometimes, the term “code” refers to structured text, like `<pre>`; not program code.

Automatic Code Generation (ACG), or Automatic Programming, is a computer mechanism that generates computer code from a high abstraction level input created by a human (Automatic programming, 2020). In other term, ACG is “the study of generative programming in the sense that the source code is generated automatically” (Liao et al., 2010).

With increasing competitive pressure, and the importance of less time to market delivery (Li et al., 2020), it is hard to develop software with the standard software development process, and produce the final output as a high-quality product. So, there is a need for a method of software development, which in development efforts and development time is reduced to a minimum as possible, and the product is effective. Therefore, one of the best solutions is to support automation-level in software development by using efficient methods and automation technologies (Li et al., 2020). One of the best methods of supporting automation in software development is supporting automation of software development itself, which is concentrated in coding. ACG reduce the efforts and time of software development, increase the capabilities of technical automated tools and retrieve more effective program (Li et al., 2020; Hu et al., 2019).

As we search in the field of ACG, in both research and technical resources, there are several technical mechanisms of applying process which generate some output code from some input. Some researchers and tools produce or collect code from pre-defined code snippets which previously written and stored in the ACG software, like CrestApps Laravel Code Generator ((Mike), Laravel Code Generator) tool. Another type, which matches human text or visual input to code or piece of code, like converting ER (Entity Relationship) diagram to SQL (Structural Query Language) code, like ERDPlus tool (Info@erdplus.com, ERDPlus). Some other research presents intelligent models and tools which use Machine Learning methods to generate a code from human input.

Also, with a variety of types of ACG, there are different approaches for generating code from an input. The approach of ACG depends on the type of input to the ACG process. Based on this, there are five approaches for generating code from input, which is NL-to-Code (Hu et al., 2019; Xu et al., 2020; Ling et al., 2016; Yin et al., 2018; Li et al., 2020; Zeng et al., 2020; Wang et al., 2019; Elgohary et al., 2020), Template-to-Code (Arnoldus et al., 2012; Liao et al., 2010; Gavilanes et al., 2009), Pseudocode-to-Code (Zhong et al., 2020), Code-to-Code (Maclay et al., 2000) and Model-to-Code (Sebastian et al., 2020; Paolone et al., 2020; Thate et al., 2004; Korenblat et al., 2019). More details about the types and approaches of ACG are presented in Section 2.

In responding to the need for highly qualified ACG solutions, many studies have been done on this. To make a smooth overlap between ACG and the software development process, some researchers, (Sebastian et al., 2020; Paolone et al., 2020) proposed Model-Driven Architecture (MDA) is composed of software engineering methodology and software architecture, to support automation level, and add the value of automatically generating code from models which are the output of modeling phase. However, for Model-to-Code methods to be applicable, the modeling/design phase, in SDL (Software Development Lifecycle), must be completely followed by the development team, to produce all required models. That means, to be able to use Model-to-Code methods, the developer has to do the hard and time-consuming work required to fulfill design requirements in the software modeling in the SDL. Other researchers have used textually structured, the input of code generation, instead of the graphical model, such as pseudocode (Zhong et al., 2020), mathematical expression (Gavilanes et al., 2009), and structured text (Arnoldus et al., 2012). The issue with using restricted structure text is time-consuming and complex, especially with large program processes or complex logic. In a different direction, (Maclay et al., 2000) proposed an approach to generating code by adapting existing code. Of course, the quality of the output code is fully affected by the quality of the input code.

The most usable ACG model is easy to use and easy to use by a non-expert. In response to it, many researchers have gone to use Natural Language Processing (NLP) techniques in code generation (Hu et al., 2019). Generating code through natural languages (NL) is an important future direction of programming (Hu et al., 2019). It can lower the threshold of programming and facilitate the programming process, and it makes programmers more productive (Hu et al., 2019).

The nature of how the NL-to-Code model works depends on the nature of input NL texts and output codes in a used dataset in training. The main limitation of NL-to-Code works (Hu et al.,

2019; Xu et al., 2020; Yin et al., 2018; Zeng et al., 2020; Wang et al., 2019; Li et al., 2020; Elgohary et al., 2020) is used of NL-code pairs dataset consists of complete, formal, syntactic and semantic correct NL input, and standard-style comment-free code output. This leads to making restrictions on input text to correctly generate a code output in standard style. An example of complete, correct input is “write a function gives the domain name from string email parameter”.

Generally, the existing highly restricted, researchers, regardless of type and approach, try to generate correct-syntactic code, without description or code comments, from correct-syntactic, and right-semantic input. This may cause limitations on the structure of the input, even in free-text NL-to-Code Generation. In response to these limitations, we propose Natural Language Processing model for open-domain general-purpose single-file program code, in python, from not-restricted, informal, sequence or single term, free-text NL input, such as “bubble sort”, to produce practical, real-world, and developer-style code with needed classes, functions, calls, and even calls of external libraries.

As for the training dataset, we collect and build “PyCorpus” dataset of NL-code pairs from various formal and individual informal resources, like tutorial websites, coding challenges websites, GitHub repositories, technical communities, and like others.

1.1 Research Problem

Most existing research works provide effective models for NL-to-Code Generation with high focus and attention for correctly understanding semantics and parsing of syntactic of complete input NL text, to produce high-accurate and correct program code, in different ways.

The main limitation of existing models is the failure in handling short, single-term, informal, or inputs that are in an incorrect syntactic or semantic manner. This limitation is because the existing datasets are collected from formal sources, which are a set of standard-style code or complete long input sentences that are syntactically and semantically correct. Hence, there is a need for a model that can generate single-file developer-style code from unrestricted grammar, unrestricted length, unrestricted formality input text.

1.2 Research Objectives

In this section, we present the main and specific objectives of the research work.

1.2.1 Main Objective

The main objective of this research is to generate open-domain general-purpose single-file Python code, from unrestricted grammar, unrestricted length, unrestricted formality, free-text NL input; to produce practical, real-world, developer-styled, and developer-friendly code with needed classes, functions, calls, and even calls of external libraries. We aim to collect and build NL-code dataset with features serve the goal model, and try to enhance NL-to-Code generation by adding the capability of handling short and single-term NL input.

1.2.2 Specific objectives

1. Create “PyCorpus” dataset of NL-Code pairs from informal tutorials and open-source repositories which contains the coding solution of programming problems and challenges.
2. Implement a Supervised Code Embeddings layer to create a correct representation of code, in the “PyCorpus”.
3. Implement an appropriate NLP Transformer-Based model for Seq2Seq, to understand NL input, then generate the correct Python code.
4. Evaluation created a model using execution-based evaluation by BLEU, METEOR, automatic Python compile.

1.3 Importance of the Work

With a highly competitive software market and high pressure, by software solutions provider enterprises, in developers, there is a real need for creative solutions to rapidly up the software development process with no effect on the quality of the process and product.

The most used solution, in all business fields, for speeding up the process; is automation. So, Automatic Code Generation is a good solution to reduce the cost, time, and efforts of developing software, and rapid delivery to the production environment.

Now, Automatic Code Generation is an available solution for non-experts in coding. Although some research works to support the use of ACG tools by non-experts by an easy and usable method which is Natural Language, the provided solutions have some restrictions and limitations in real use.

Our model allows using unrestricted natural language in ACG, for more usability and effectiveness of the process.

1.4 Scope and Limitations of the Work

- We will not compare evaluation results with state-of-the-art, because we focus on resolving limitations of handling short and single-term NL text input, which is not supported in the state-of-the-art.
- We focus, in this thesis, on the efficiency of the model in terms of accuracy, not on-time performance.

1.5 Research Approach

To achieve our main objective to develop a model that can generate Python code from free natural text, we follow a research methodology, as we can see in Figure 1.1.

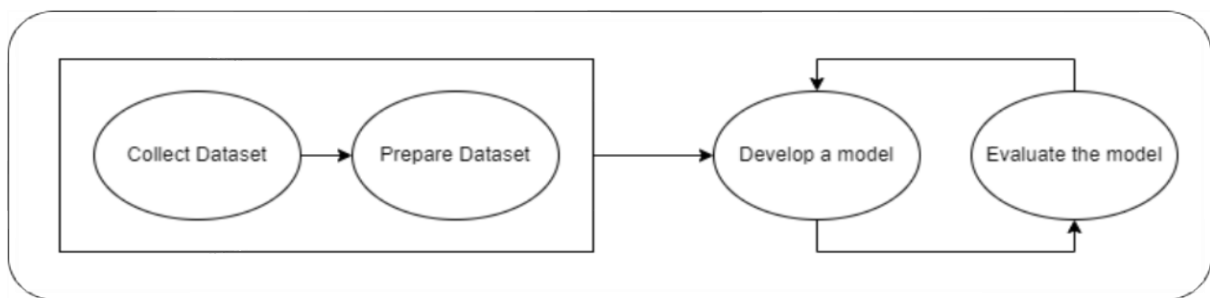


Figure 1.1: Research Methodology (High-Level View)

Phase 1: Dataset Collection: The first phase in our methodology is to search and collect a dataset with NL-code pairs. Our approach to data collection is quantitative.

Phase 2: Data Preparation: The second phase is to prepare and build the structure of NL-code pairs which be able to be processed by the model in Python.

Phase 3: Development: In this phase, we use Neural Machine Translation modeling to train the model.

Phase 4: Evaluation: We evaluate our models using BLEU, METEOR and code compile.

1.6 Overview of Thesis

The thesis consists of seven chapters. these chapters are organized as follows:

Chapter 1: Introduction: This chapter describes an overview of Automatic Code Generation and introduces research problems, research objectives, the importance of this work, scope, and limitation and briefly presents our methodology.

Chapter 3: Literature Review: This chapter reviews other related works related to the thesis, including works of NL-to-Code, Model-to-Code, Code-to-Code, Pseudocode-to-Code and Template-to-Code.

Chapter 4: Methodology: This chapter describes the methodological steps we follow, approach of collecting and structing our PyCorpus dataset, and the architecture of our proposed methods with a full explanation of each step we used.

Chapter 5: Experiments and Results: In this chapter, we discuss and present in detail the PyCorpus dataset, the experiments we conduct, and their results. We also present a comparison between these results and the way we evaluated these experiments.

Chapter 6: Conclusion and Future work: This chapter presents the conclusions of the research and presents future works.

Chapter 2

Background

Chapter 2 Background

Before we describe the proposed system, we briefly present the essential concepts of background knowledge on Automatic Code Generation and core-related areas of Natural Language Processing.

2.1 NLP (Natural Language Processing)

As well Natural Language is one of the most important aspects of people's lives, it is also one of the research fields in computer science. In recent years, a lot of researchers' efforts are forwarded to the science of NLP (Natural Language Processing).

2.1.1 NLP

NLP (Natural Language Processing) is a combination of subfields of linguistics, Computer Science (CS), and Artificial Intelligence (AI) concerned with enabling machines to understand, process, and analyze natural language, to produce machine-readable, human-readable natural language output (Natural language processing, 2020).

There is a lot of Computer Science (CS), and Artificial Intelligence (AI) fields and applications that use NLP, such as Automatic Text Summarization, Speech Recognition, Machine Translation, Text Generation, Syntactic Analysis, etc.

2.1.2 Word Embeddings

All NLP tasks, which need to use the semantic or context of the word, need some way or method to identify the meaning of the word. This method is a vector representation of a word (Jurafsky et al., 2012), indicating the meaning of the word in the original text (Jurafsky et al., 2012). The representation of the word in the text to represent semantic meaning in context is called Word Embeddings (Jurafsky et al., 2012). Figure 2.1 (Intro to Word Embeddings and Vectors for Text Analysis., 2018) presents a simple example of vector representation of some vocabularies.

Word Embeddings is one of the main pre-processing tasks in the NLP process and a mandatory layer in the NLP process that uses the meaning or context of the word.

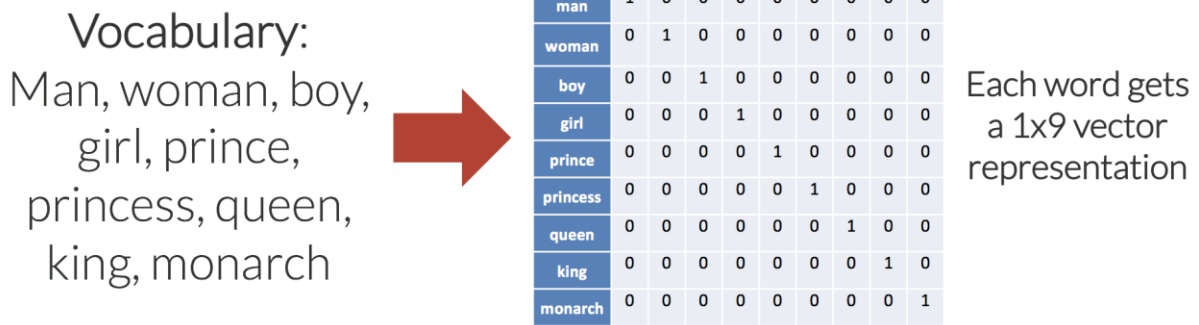


Figure 2.1: Example of Vector Representation in Word Embeddings (Intro to Word Embeddings and Vectors for Text Analysis., 2018)

2.1.3 Code Embeddings

Code Embedding is like Word Embedding, but it is customized/special for program code. Code Embedding allows us to model correspondence between code snippets and labels naturally and effectively (Alon et al. 2019). Code Embeddings are continuous vectors for representing snippets of code. Code Embeddings are distributed vector representations for code.

2.1.4 Sequence Processing

Sequence Processing is one of the NLP tasks, concerned with processing a continuous input stream of indefinite-length text (Jurafsky et al., 2012) of a sequence of words, to generate new text of a single term, like Text Classification, or sequence of terms, like Text Generation. Figure 2.2 (15, Enthusiast, Chaoran,; Enthusiast, 2019) shows an abstract view of Sequence Processing for Text Generation.

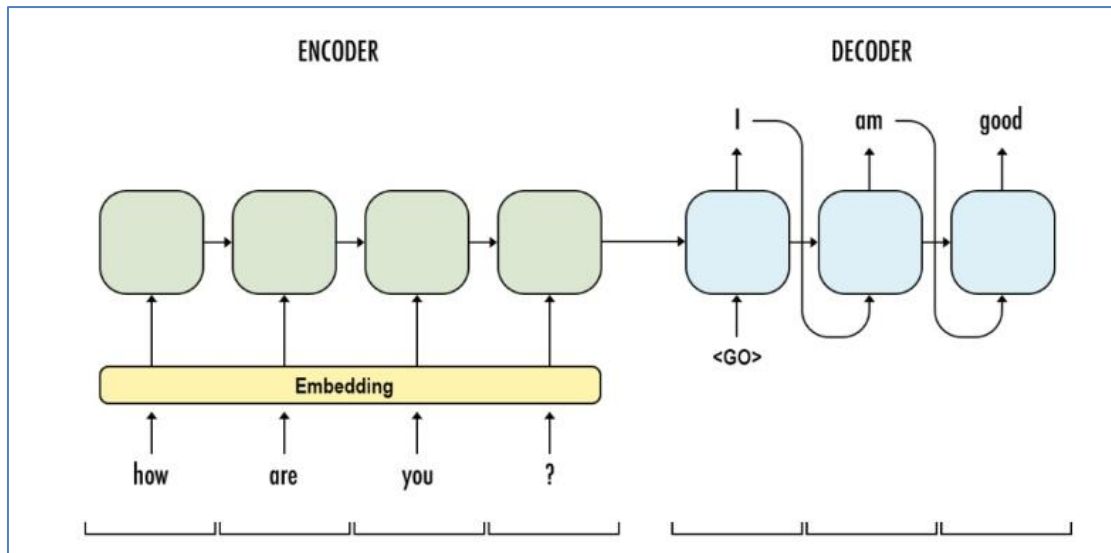


Figure 2.2: Abstract View of Sequence Processing for Text Generation (15, Enthusiast, Chaoran,; Enthusiast, 2019)

2.1.5 Seq2Seq Model / Encode-Decoder Model

Seq2Seq is a type of Sequence Processing, concerned with generating NL output as a sequence that may have the same length of input text; or not.

Seq2Seq consists of two components which are: the Encoder network and the Decoder network. Encoder network tasks input sequence and creates the appropriate text representation (Jurafsky et al., 2012). This representation is passed to the Decoder network which generates a task-specific output sequence (Jurafsky et al., 2012).

Text Generation, Machine Translation, and Automatic Code Generation are applications of Seq2Seq.

2.2 Code Generation

According to (Merriam-Webster) dictionary, the term of “code” refers to the “systematic statement of a body”. “Generation” term is commonly known as producing output from different-type or different-shape input. So, the term of “Code Generation”, in general, is the process of producing structured text output from some input.

2.2.1 Definition of Code Generation

As we find, Code Generation, in the technical field, has different definitions according to the context of the technical field. For example:

1. In the compiler, Code Generation is the process by which the compiler converts program code, written in a high-level or mid-level programming language, to machine-language code (Code generation (compiler), 2020).
2. In the compilation of some platform-independent programming languages, the program code must be compiled into byte code than machine code (Mohammad Waseem, 2020).
3. In traditional software engineering, the implementation phase is a software development activity, which is manually performed by developers, to create application code extracted from design models produced from the previous modeling phase. This can be expressed as Code Generation (Sebastian et al., 2020; Paolone et al., 2020).
4. Automatic Code Generation (ACG), called Automatic Programming, is the process of generating program code from high-level, abstract human-created, human-readable input. For example, automatically generate SQL code from ER (Entity Relationship) Diagram using some tool.

In this work, we focus on ACG.

2.2.2 Types of Automatic Code Generation

We present types of Automatic Code Generation according to four factors: domain, purpose, nature of output code, and method.

According to the target business domain factor, two types of ACG are (Xu et al., 2020; Wang et al., 2019):

1. Close-Domain ACG: refers to the ACG that is restricted in a specific business field. For example, generating SQL queries for only air flights data.
2. Open-Domain ACG: refers to that the ACG is generic and not dedicated to a specific business field.

According to the purpose factor, two types of ACG are (Xu et al., 2020):

1. Special-Purpose ACG: refers to that the output code is valid or applicable for a specific goal. For example, generating Python code for only Neural Networks, or automatically generating code for unit test cases (Zhong et al., 2022).
2. General-Purpose ACG: refers to that there is no pre-defined goal that restricts or limits of process or result of ACG.

According to the factor of the nature of output code, two types of ACG are (Hu et al., 2019):

1. Single-Line Structured Code: refers to code or snippet code written correctly in a single line, and has a clear structure, like SQL.

2. Multi-Line Semantic Code: refers to snippet code as multiple lines representing a sequence of instructions, which must be syntactically and semantically correct.

According to timing, there are two types what are:

1. Real-Time ACG, which enables the user to get output code at the same time as writes the input.
2. Offline ACG, like the request-response process.

According to the method, we present three types of ACG with names which we called.

1. Fixed Pre-defined Code Snippets Generation, which refers to generating code from pre-coded snippets, or collecting pieces of code from pre-defined and stored code snippets. For example, generate models' and controllers' classes, in Model-View-Controller (MVC) framework, by entering the name of the required model by the developer. Another example is the generation of HTML, CSS table, or form layouts from input names and types of fields. One of the example tools is CrestApps Laravel Code Generator ((Mike), Laravel Code Generator) which helps developers generate common resource files and classes in PHP Laravel.
2. Matching Snippets (Hu et al., 2019), which refers to the direct mapping between some text or visual input to code, such as: converting ER (Entity Relationship) Diagram to SQL code, like ERDPlus tool (Info@erdplus.com, ERDPlus) which is a database modeling tool for creating diagrams and schemas, and exporting SQL code from diagram models.
3. Intelligent Input-Semantic-Considered Code Generation, which refers to smart code generation from the model (graph-based input like sketches or standard software design model) or text, with consideration of the semantic side of input, and semantic and syntactic side of output code. Machine Learning and Natural Language Processing are used in this type (Hu et al., 2019). One of the commonly used tools, of this type, is CodePilot (CodePilot.ai) which is a development IDE that uses AI techniques to provide developers with live code recommendations, examples, and highlight lines of codes that may cause a bug.

In our research, we focus on the Intelligent Input-Semantic-Considered Code Generation type, as it is still open for research.

2.2.3 Approaches of Automatic Code Generation

ACG Approach refers to how the system generates output program code from some input. ACG approach is like ACG type based on the input type, but we name it as “approach” because the

type of input determines how the system will be built and structured, and how the process will be programmed and executed. There are five approaches of ACG which are:

1. Natural Language NL-to-Code (Hu et al., 2019; Xu et al., 2020; Ling et al., 2016; Yin et al., 2018; Li et al., 2020; Zeng et al., 2020; Wang et al., 2019; Elgohary et al., 2020): refers to generate code from input written in human-readable natural-language. The input may be free text or restricted by some conditions. Figure 2.3 shows an example of converting natural text to Perl code.

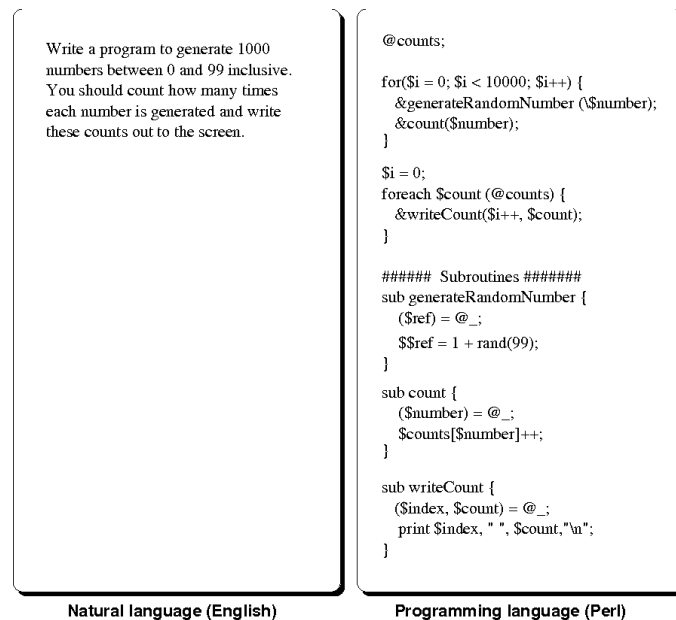


Figure 2.3: Example of Converting NL to Code (Mihalcea et al., 2006)

2. Template-to-Code (Arnoldus et al., 2012; Liao et al., 2010; Gavilanes et al., 2009): refers to generating code from structured-text or graph-based input. The input has structure specifications. One of the example tools is TextFixer (Text to HTML) which is a simple tool that converts structured text to HTML structured code. Figure 2.4 shows an example of converting content to HTML code.

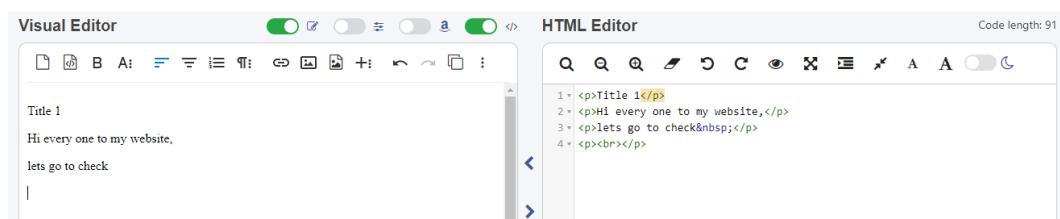


Figure 2.4: Example of Converting Template to Code (TextFixer, Text to HTML)

3. Pseudocode-to-Code (Zhong et al., 2020): refers to converting pseudocode to code, like PseudoEditor (Pseudocode to java converter: Pseudoeditor) tool which converts

pseudocode to Java code. Figure 2.5 shows an example of converting pseudocode to code.

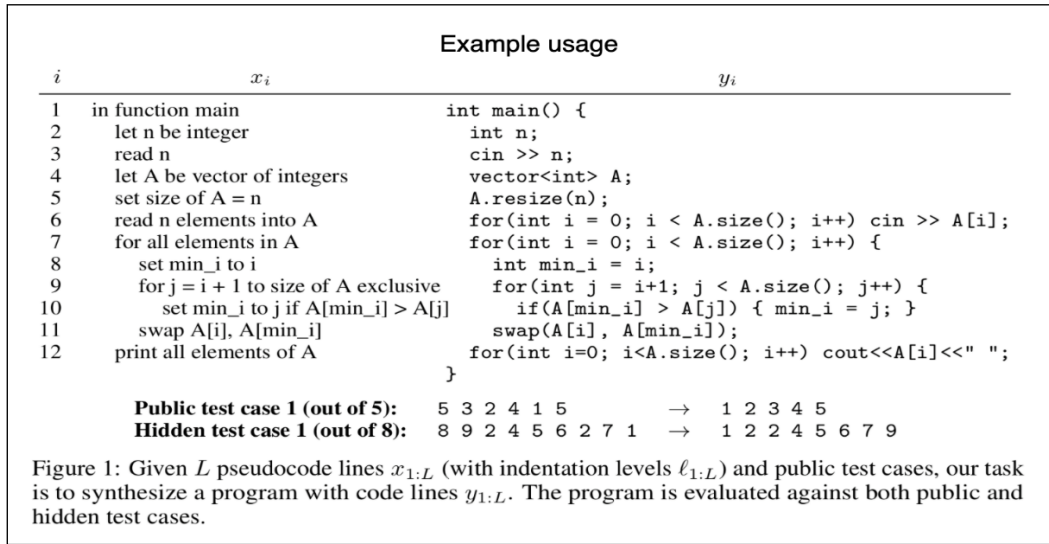


Figure 2.5: Example of Converting Pseudocode to Code (Kulal et al., 2019)

4. Code-to-Code (Maclay et al., 2000): refers to adapting existing code to extract new code with different functionality, like CodePilot (CodePilot.ai) which provides code recommendations based on written developer's code. Figure 2.6 presents a screenshot from CodePilot which recommend a code according to the existing code.

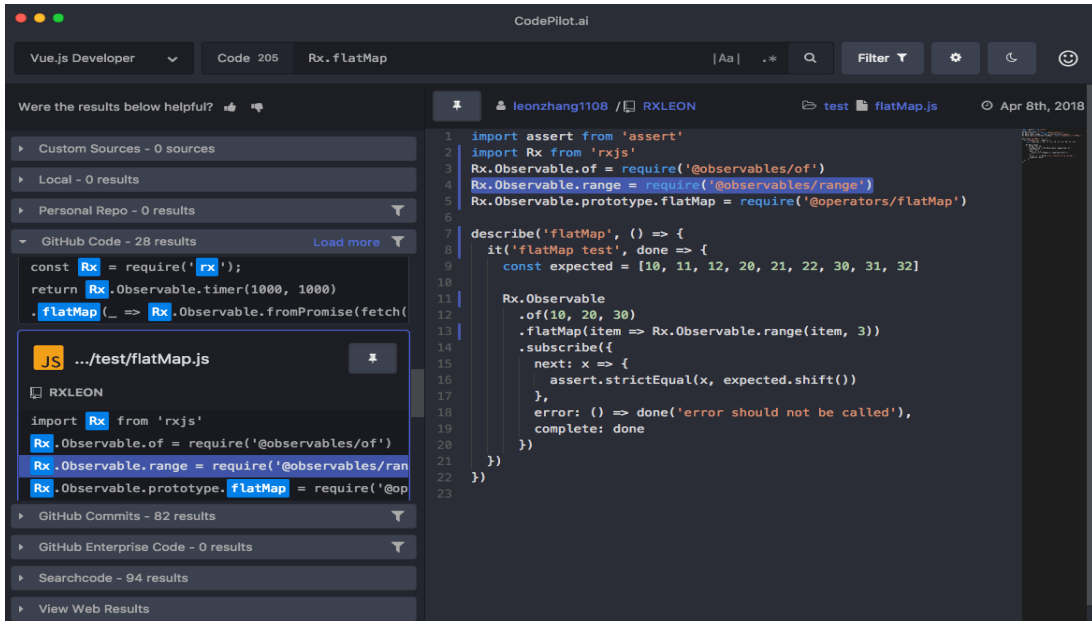


Figure 2.6: Screenshot from CodePilot which Recommend JS Code according to existing Code (CodePilot.ai)

5. Model-to-Code (Sebastian et al., 2020; Paolone et al., 2020; Thate et al., 2004; Korenblat et al., 2019): refers to extracting code from software design model or GUI

(Graphical User Interface) design. The model, used in the approach, can be a software design, like a UML diagram, or a graphical design built by user-friendly items, like drag-and-drop website builders or sketch-up tools. Now, some intelligent tools build code from human-drawn UI's, like (Microsoft AI, Transform sketches into HTML using AI, 2021) from Microsoft. Figure 2.7 shows an example of converting a UML diagram to Java code.

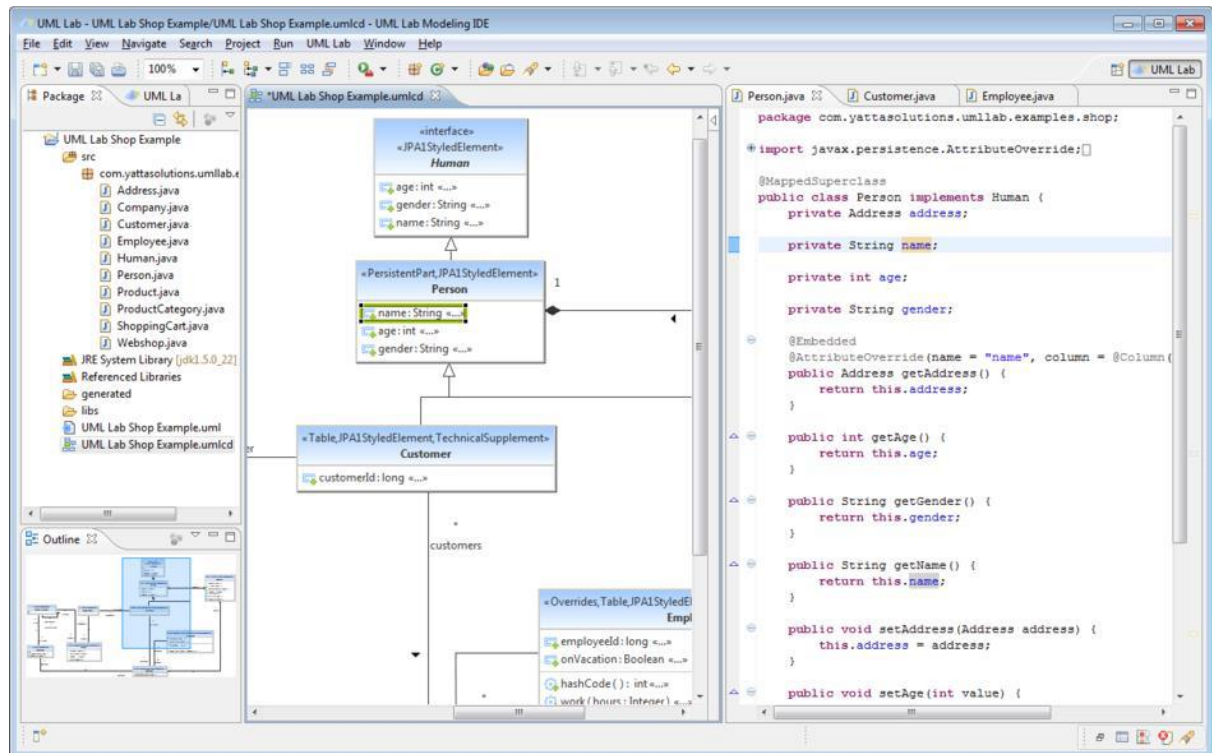


Figure 2.7: Example of UML to Java code (Modeling and Code Generation - Yatta Solutions, 2022)

Chapter 3

Literature Review

Chapter 3 Literature Review

In responding to pressure made by companies on developers for delivering software fast with insuring high-quality products, some researchers have proposed scientific and technical solutions for code generation; to support automation level in the software development process (Li et al., 2020), in purpose to reduce development effort and time, and import software development efficiency (Wang et al., 2019). However, the studies on ACG are few (Li et al., 2020). In this section, we present and discuss the selected related works which provided solutions for our presented problem, and common used datasets that used and can be used in ACG research works.

3.1 Datasets

3.1.1 Existing NL-Code Datasets

There are many common open-source datasets of mapping natural language to code. In Table 3.1, we present some of the common datasets used in our state-of-the-art.

The nature of NL-Code mapping, the nature of the output code, and the purpose of the dataset are differing among the existing datasets.

As for the nature of NL-Code in the dataset, CONCODE (Iyer et al., 2018) struct the data as a map of Java method/class official documentation text to code. Spider (Yu et al., 2018) builds the input NL as a question format, and the output SQL may be assigned to many questions. Cosql (Yu et al., 2019) is built as a dialogue conversation of input as a question and response as SQL query.

As for the nature of code data, Spider (Yu et al., 2018) and Cosql (Yu et al., 2019) contain code of only structural SQL language. Awesome Java (Akullpp, 2022) provides a varied curated list of source code of a lot of Java frameworks, libraries, and systems. CONCODE (Iyer et al., 2018) contains Java code as only classes.

Some dataset collection works (Iyer et al., 2018; Akullpp, 2022) aim to provide open-source datasets. Other datasets (Yu et al., 2019; Yu et al., 2018) are designed to help researches to build models for generating new SQL queries and database schemas. BigCloneBench (Svajlenko et al., 2015) is built and designed in purpose to provide a benchmark for evaluating code generation models.

The common open-source datasets are rich in data rows. However, in most of it, the NL part is a structured text, formal-styled (Akullpp, 2022; Iyer et al., 2018), correct-grammared sentence (Yu et al., 2018; Yu et al., 2019), or a complete-meaning sentence.

Table 3.1: Some of Common Existing NL-Code Datasets

Reference	Dataset Name	Language	Source	Size (#K of samples)
(Iyer et al., 2018)	CONCODE	Java	Online code repositories	~100
(Svajlenko et al., 2015)	BigCloneBench	Java	Open-source Java systems	~901
(Akullpp, 2022)	Awesome Java	Java	Source code of some Java frameworks and systems	~264
(Yu et al., 2019)	Cosql	SQL	Manually	~40
(Yu et al., 2018)	Spider	SQL	Manually	~11

CONCODE (Iyer et al., 2018) contains over 100K of NL-code pairs. BigCloneBench (Svajlenko et al., 2015) contains around 901K Java files, and Awesome Java (Akullpp, 2022) contains around 264K Java files. Cosql (Yu et al., 2019) contains over 40K pairs of NL and SQL queries. Spider (Yu et al., 2018) contains 10181 NL questions mapped to 5693 unique complex SQL queries.

3.1.2 Datasets Used in Existing NL-to-Code Research Works

Each ACG research work has its purpose. A dataset, which is used in research work, and dataset collection methodology are determined by the purpose of the research work. Most of NL-to-Code generation research works aim to propose a solution to solve common drawbacks in code generation or enhance the performance of code generation (Hu et al., 2019; Ling et al., 2016; Zeng et al., 2020). (Yin et al., 2018) aims to build a high-quality NL-code dataset to support more quality in code generation modeling. (Liu et al., 2019) aims to rebuild an existing text-to-SQL dataset to provide a representation of databases and columns relations within the dataset, to enable the code generation model to generate generalized new SQL query and database schema. (Li et al., 2019) proposes a metric model specialized in the evaluation of code generation models. Table 3.2 lists our state-of-the-art NL-to-Code research works and used datasets in its.

Most of the open-source datasets are built from clear and clean NL inputs that be complete sentences, correct-grammared sentences, structured text, or formal-styled text. The NL inputs collected from programming language official documentation, or some resources provide a complete description of code solution, like StackOverflow, or are created manually.

Most of the research works, especially which propose a model of an enhanced model of code generation, use open-source datasets. That causes a limitation in that the existing models work effectively with only clean NL input as complete and correct sentences or structured or formal-styled text.

Table 3.2: Used Datasets in State-of-the-Art NL-to-Code Research Works

Paper	Goal	Source Dataset	of	Used Dataset
(Hu et al., 2019)	Propose a new method that solves drawbacks in automatic code generation	Open-source datasets		Awesome Java, CONCODE, BigCloneBench, and JDK source codes

(Ling et al., 2016)	Generate code of function with arbitrary of inputs	Open-source implementation	
(Yin et al., 2018)	Build a high-quality dataset	Collected from StackOverflow	
(Zeng et al., 2020)	Proposes an application solution to enhance the matching accuracy of output SQL	Open-source datasets	CoSQL, Multi-WOZ
(Yu et al., 2018)	Re-build existing dataset to address schema relationships	Open-source dataset	Spider

3.2 Automatic Code Generation Approaches

Generally, the related works differ in research academia position, type of research, the goal of research, and used approach. Table 3.3 presents the selected research with information about the type of research, publishing year, number of citations, and used ACG approach.

Table 3.3: Overview of Related Works

Research/Work	ACG Approach	Citations	Year
(Arnoldus et al., 2012)	Template-to-Code	18	2012
(Elgohary et al., 2020)	NL-to-Code	1	2020
(Gavilanes et al., 2009)	Template-to-Code	3	2009
(Hu et al., 2019)	NL-to-Code	0	2019
(Korenblat et al., 2019)	Model-to-Code	0	2019
(Li et al., 2020)	NL-to-Code	0	2020
(Liao et al., 2010)	Template-to-Code	7	2010
(Ling et al., 2016)	NL-to-Code	205	2016
(Paolone et al., 2020)	Model-to-Code	0	2020
(Sebastian et al., 2020)	Model-to-Code	1	2020

(Thate et al., 2004)	Model-to-Code	30	2004
(Wang et al., 2019)	NL-to-Code	29	2019
(Xu et al., 2020)	NL-to-Code	2	2020
(Yin et al., 2018)	NL-to-Code	50	2019
(Zeng et al., 2020)	NL-to-Code	0	2019
(Zhong et al., 2020)	Pseudocode-to-Code	2	2020

Generally, the selected related works, regardless of the used approach, tried to generate correct-syntactic code, without description or code comments, from highly restricted, correct-syntactic, and right-semantic input. This may cause limitations on the structure of the input, even in free-text NL-to-Code Generation.

3.2.1 Natural Language to Code

In recent years, there has been an evolution in the use of NLP techniques as assistance tools in code programming, in sub-fields of Code Generation (Wang et al., 2019), Code Refactoring, Testing, and Quality Assurance. In the field of ACG, there is a persistent need to use these techniques which are beneficial for reducing development efforts, generating effective code, increasing the quality of product, and constructing smarter Integrated Development Environments (IDEs) (Wang et al., 2019; Hu et al., 2019). Therefore, many researchers have paid attention to ACG, and have proposed various NL-to-Code generation solutions with different approaches to using Machine Learning and NLP (Hu et al., 2019).

As known, the general trait of business applications takes the structure of two high-level components: (1) software system, and (2) storage. The most commonly used storage engine is SQL Database Management Systems (Choudhury, 2020). So, Some researchers have studied ACG, from NL, of program code (Hu et al., 2019; Xu et al., 2020; Ling et al., 2016; Yin et al., 2018; Li et al., 2020), and others have studies ACG, from NL, for SQL (Zeng et al., 2020; Wang et al., 2019; Elgohary et al., 2020). The nature of generating program code is different from generating SQL, because of the difference between its' definitions. The program code is a set of interactive instructions built by logical sequence, but the SQL statement is a single-line well-structured statement representing an operation on the database (Define SQL and state the differences between SQL and other conventional programming Languages, 2017). In this subsection, we discuss both.

The solutions were presented vary in their acceptability of text is unrestricted by structure specification. To achieve a usable method of using the NL-to-Code technique, a lot of research, as we found, handles free-text-NL input (Hu et al., 2019; Xu et al., 2020; Yin et al., 2018; Zeng

et al., 2020; Wang et al., 2019; Li et al., 2020; Elgohary et al., 2020), but their others restrict NL input by structured specification, which called template. Using a different, medial approach, (Ling et al., 2016) presents the NL-to-Code approach of ACG from text input mixed of free-text and structured-specification text.

Highlighting the goal of the proposed solution, in mentioned works in this subsection, provides more clear vision and reasoning of the proposed approach of the solution. (Hu et al., 2019; Ling et al., 2016; Wang et al., 2019) aimed to produce a novel model of NL-to-Code ACG. In another direction, (Xu et al., 2020; Yin et al., 2018) proposed methods for mining datasets, to achieve high quality in mined NL-Code pairs. Other researchers focused on quality factors, like (Li et al., 2020) who focused on proposing the metric model based on the quality and efficiency of automatic code generation; and (Zeng et al., 2020, Elgohary et al., 2020) which focused on that output code meets the requirement of the semantic side of NL input, by allowing the user to provide feedback.

Several used techniques in the presented approaches. (Hu et al., 2019) used Supervised Code Embeddings (SCE) to resolve the drawback that the length of output code is always longer than the length of input NL, and Abstract Syntax Tree (AST) to resolve the challenge of code snippet with the same functionality could be implemented in various ways. Some researchers have used Semantic Parser as the base model of their' approaches (Xu et al., 2020; Zeng et al., 2020; Wang et al., 2019; Elgohary et al., 2020). Also, Neural Networks (Ling et al., 2016) and Long-Short Term Memory (LSTM) learning (Zeng et al., 2020; Wang et al., 2019) were widely used in NL-to-Code models. (Yin et al., 2018) take the different direction of his approach which is based on the probabilistic model. Generally, the most commonly used NLP method is Seq2Seq Encoder-Decoder Model.

The state-of-the-art of NL-to-Code models provided approaches for ACG and mining datasets. In general, the used datasets in presented NL-to-Code approaches are pre-collected or open-source datasets (Hu et al., 2019; Zeng et al., 2020; Wang et al., 2019), or new datasets collected from formal resources (Xu et al., 2020), like Documentation, or QA communities (Xu et al., 2020; Yin et al., 2018), like StackOverflow. The nature of NL input, in these datasets, is a complete sentence or question, in semantically or syntactically correct form; that causes limitations on handling short or single-term NL input, e.g. "Bubble Sort". Another issue, we think that there should be more focus and effort in text preprocessing, like stemming, on NL input, for more accurate generalization.

Our research belongs to the NL-to-Code approach, and we focus on the discussion of NL-to-Code works. Table 3.4 presents a subset of related works in the field of NL-to-Code Generation

with brief information about output code, the goal of the research, and the used model. The most of the existing NL-to-code research works use open-source datasets, and aim to product or enhance an ACG modeling. That indicates its do not take into their considerations the limitation of ACG that will be occurred because of the restrictions of completeness and correctness of NL input.

Table 3.4: Overview of NL-to-Code Related Works

Research/Work	Dataset	Method	Main Goal	Output Code	Used Model	Evaluation metric/s
(Hu et al., 2019)	Several open-source datasets	Encoder-Decoder model with pretrained code embedding	Produce a Model	Java	Supervised Code Embedding (SCE) model to promote the alignment between natural language and code, an Abstract Syntax Tree (AST), new distributed representation of code snippets for functionality could be implemented in various ways	BLEU, Precision, Recall and F-score
(Xu et al., 2020)	Dataset mined NL-code pairs from the online programming QA forum Stack Overflow and programming language API documentation	Semantic Parsing	Data Collection	Python	Semantic Parsing of text for enhancing model by enhancing the quality of dataset pairs	BLEU
(Ling et al., 2016)	Two new datasets derived from the collectible trading card games Magic the Gathering and Hearthstone	Latent Predictor Networks	Produce a Model	Python and Java	Novel Neural Network architecture generates output sequence (code) from mixed input of NL and structured specification	Evaluation (on samples or real cases) on the task of producing code for Trading Card Games like Magic and Hearthstone.
(Yin et al., 2018)	Dataset collected from Stack Overview depending on a method to ensure high-quality of mined NL-code pairs	Probabilistic Neural Networks	Data Collection	Python and Java	A probabilistic model for mining a high-quality aligned dataset of NL-Code from StackOverflow	precision

(Zeng et al., 2020)	Open source dataset (Spider dataset)	Encoder-Decoder with Neural Semantic Parser	Correction of Input	SQL	Encoder-Decoder with Neural Semantic Parser Bi-LSTM; for resolving ambiguity in NL input, in NL-to-SQL parser, by corrector question	Accuracy and F1-score
(Wang et al., 2019)	Open source dataset (Spider dataset)	BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding	Produce a Model	SQL	Encoder-Decoder with Bi-LSTM embeddings; to solve the challenge of generalization to unseen database schema when parsing NL-to-SQL query	Accuracy
(Elgohary et al., 2020)	Open source dataset (SPLASH dataset)	Semantic Parser	Correction of Input	SQL	the model allows to user to reply to the first SQL result with free natural language feedback. The feedback is taken by the system to correct SQL interpretation, and generate a new more accurate SQL statement	Correction accuracy

As we see in the Table 3.4, most of state-of-the-art works use open-source datasets or collected from single source, that causes the limitations of restricted nature and grammar correction and perfection rules of the input text, and standard-style output code. The state-of-the-art works, that use BLEU score as an evaluation metric, achieve near results to our work, but they use a high datasets with the mentioned restrictions.

3.2.2 Template to Code

The semantic and syntactic expressions in a text is corresponding to used language (Arnoldus et al., 2012). Template, in the field of ACG, refers to the structured specification of input of the ACG process. According to selected works (Liao et al., 2010; Gavilanes et al., 2009; Arnoldus et al., 2012) in this model, the template may be NL text, expression text, like mathematical expression, structured data, or graph-based.

Some Template-to-Code approaches were composed in software engineering or software architecture. (Liao et al., 2010) proposed software development is composed of Prototyping and multi-layered software architecture, to generate data tables according to the data model, to support DM component in Expert System.

(Gavilanes et al., 2009) proposed ACG tool for CGR “Automatic Code Generation for Recurrence”, Which is used in only programming problems resolved by recurrence relations. So, in (Gavilanes et al., 2009), the input is structured expression.

(Arnoldus et al., 2012) is a book that explained code generation from the template, focusing on structured text inputs and graph-based inputs.

Although the use of the Template-to-Code model is useful in some cases, like learning first-year IT students (Gavilanes et al., 2009), it has limitations in general-purpose cases and coding assistance tools, because of its complexity or needed long-time in building templates. Moreover, the template, in some tools, may be dedicated and built for only the tool, so it can not be used or be useful in another software development task. Therefore, the template may be a wasted effort.

3.2.3 Pseudocode to Code

The use of pseudocode, which is an abstract textual description of the algorithm, in ACG, helps to bridge the gap between non-expert users and basic automation or full-fledged software development (Zhong et al., 2020).

(Zhong et al., 2020) proposed a lightweight method of code generation (pseudocode-to-code) based on semantic scaffolds.

Although the use of pseudocode is useful for the non-expert (Zhong et al., 2020), it may cause a high level of complexity and difficulty to understand. Generally, pseudocode is not a general-purpose language, so it may complexity or ambiguous in reading or writing it. Pseudocode lacks standards and follows very few rules (Chaubey, 2012)s. Due to this, some programmers reject the code because they fail to see the logic behind it (Chaubey, 2012), and this may cause the wrong action in Pseudocode-to-Code models or tools.

3.2.4 Code to Code

As we found, few researchers studied generating code from existing code or program context. (MacLay et al., 2000) is a code generation for embedded systems, which adapts existing code to new code, instead of building the new code from scratch.

This model requires a large number of code projects and repositories of well-tested and high-quality code. The quality of output code is determined and affected by the quality of output code. Also, this method lacks standardization, because the output code is generated from people-oriented input code.

3.2.5 Model to Code

Model-to-Text Transformation (MTT) is the process of generating text from the model (Difference between code generation and model-to-text transformation, 2019). “Model” in Model-to-Code refers to one of two things: (1) software design model (), such as UML, or (2) graphical design, like Drag-and-Drop UI design in some tools.

The use of modeling in software development is common, so some researchers have taken advantage of this to present Model-to-Code solutions integrated into software engineering as a Model-Drive Architecture (MDA) approach for the transformation of design model to code (Sebastian et al., 2020; Paolone et al., 2020).

In some platforms, there are more important models than UML. e.g. in the embedded system, a special type of graphical model. (Thate et al., 2004) is a model-based rapid-prototyping approach for generating MatLab, for embedded systems, from the graphical model and executed on rapid-prototyping hardware. Another example, is that the mobile development depends on User Interface (UI) design, and it is important to provide a usable approach, like (Korenblat et al., 2019) which provided a drag-and-drop tool for the manual creation of a graphical model of UI's, and then automatically generated to UI code in Android.

In both formal models and custom models, there is an issue with time. In general, modeling is time-consuming, which conflicts with the main goal of ACG. This approach assumes the development goes through the modeling phase; that is unrealistic.

Chapter 4

Methodology

Chapter 4 Methodology

This chapter describes the proposed methodology we follow for building NL-to-code ACG model. We describe the dataset, data pre-processing, NLP model structure and training, and the evaluation metrics we used to evaluate our contribution. We summarize the overall approach in Figure 4.1, and details about these steps are explained in the following subsections.

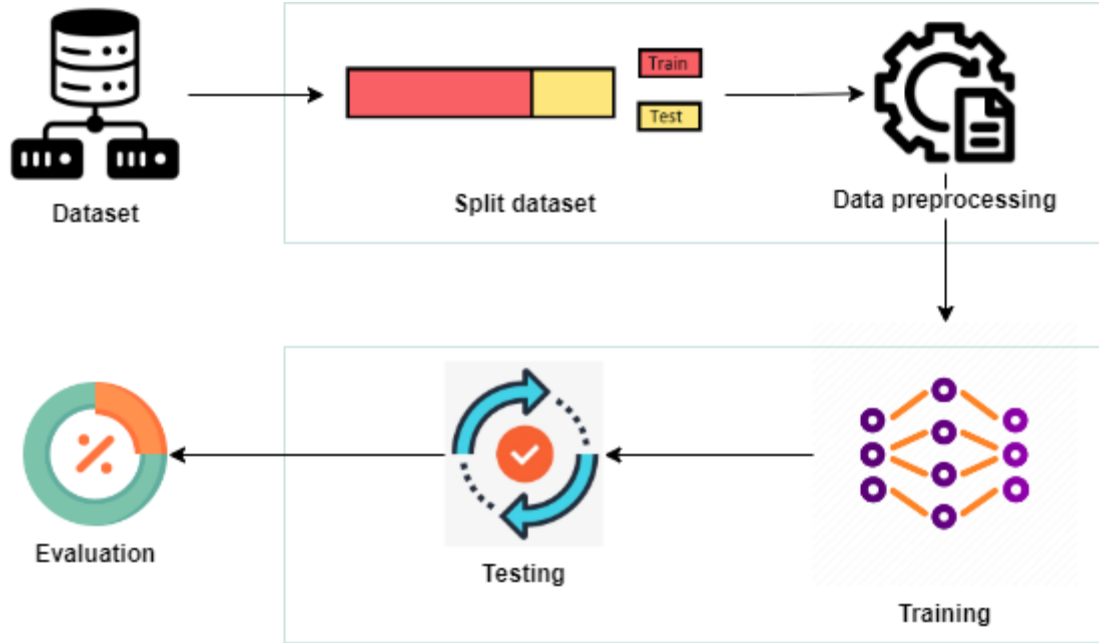


Figure 4.1: The Methodology Steps

4.1 Dataset

We use our collected dataset that we named “PyCorpus”. Figure 4.2 presents the steps of collecting PyCorpus.

To build PyCorpus, we search for sources provide Python codes or code snippets with its natural language description. To collect different structures and metaphors of NL inputs, and various structures and coding styles of code, we determine various sources. We categorize the used sources, which we use to collect PyCorpus from, as follows:

- Websites
 - Websites for Python code examples
 - Websites for Python programming exercises
- GitHub repositories
 - GitHub repositories for Python codes of common programming problems, or implementation of common algorithms

- GitHub repositories for solutions of Python programming challenges
- GitHub repositories for Python programming interview questions and its answers

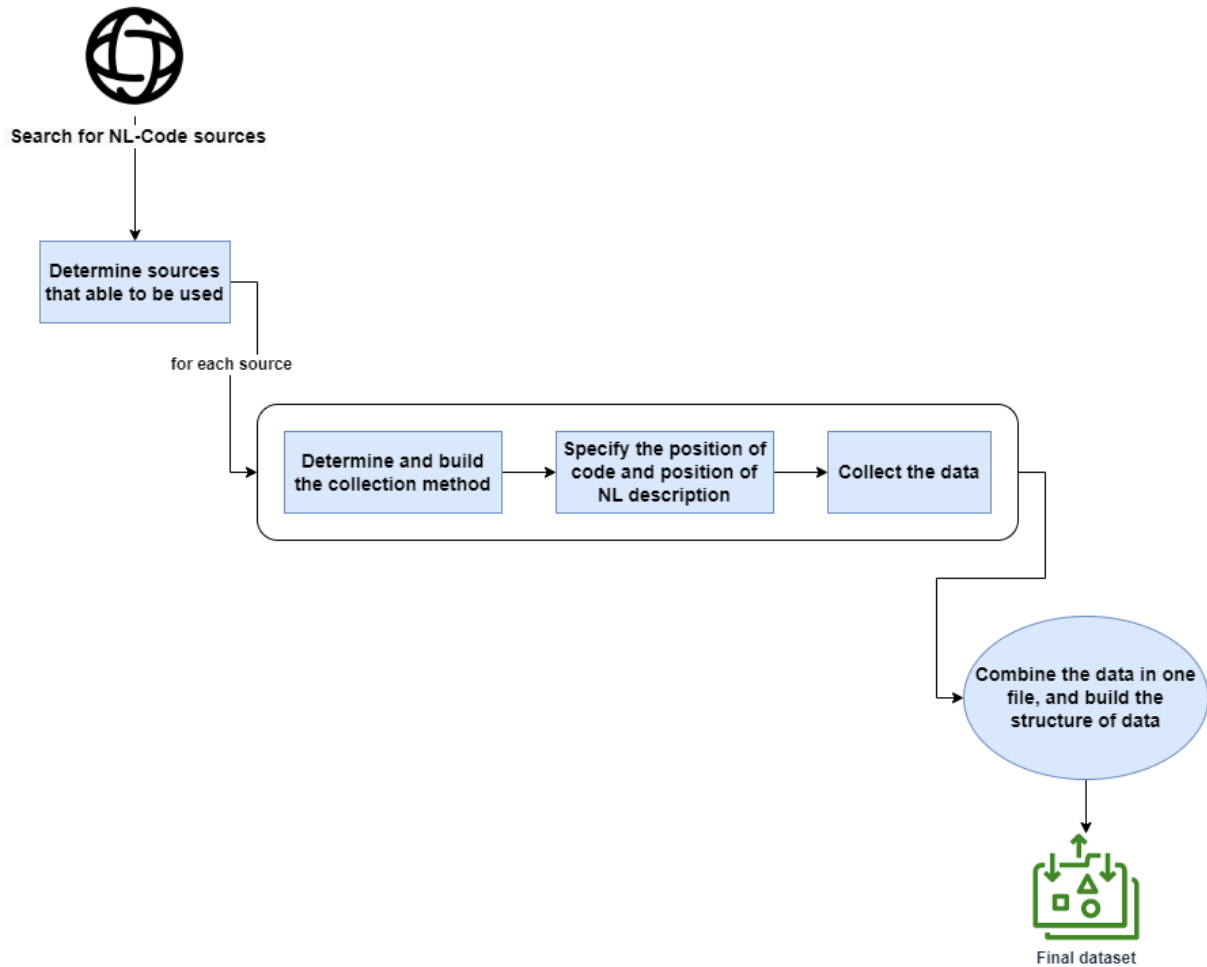


Figure 4.2: Steps of Building PyCorpus

Figure 4.3 shows an example of webpage of code exercise, and Figure 4.4 shows an example from code solution from a GitHub repository.

Pick Word Solutions 🐍🐍

Exercise 30

This exercise is Part 1 of 3 of the Hangman exercise series. The other exercises are: [Part 2](#) and [Part 3](#).

In this exercise, the task is to write a function that picks a random word from a list of words from the [SOWPODS dictionary](#). Download this file and save it in the same directory as your Python code. This file is [Peter Norvig's](#) compilation of the dictionary of words used in professional Scrabble tournaments. Each line in the file contains a single word.

Hint: use the Python random library for picking a random word.

Sample solution

To solve this exercise, you need to do three things:

1. Read all the lists of words
2. Generate a random number
3. Take that word

Here is a sample solution:

Here is an alternate approach, with some comments:

```
# import the random library
import random

# read all the list of words
words = []
with open('sowpods.txt', 'r') as f:
    line = f.readline().strip()
    words.append(line)
    while line:
        line = f.readline().strip()
        words.append(line)

# generate a random number
random_index = random.randint(0, len(words))

# take the word
print("Random word: ", words[random_index])
```

There are many solutions to every problem. Enjoy hacking around to find another solution!

Figure 4.3: An Example of Code Exercise from a Webpage (Michele Pratusевич, 2014)

```

1 """
2 Problem Link: https://practice.geeksforgeeks.org/problems/0-1-knapsack-problem0945/1
3
4 You are given weights and values of N items, put these items in a knapsack of capacity W to get the maximum total value in the
5 knapsack. Note that we have only one quantity of each item.
6 In other words, given two integer arrays val[0..N-1] and wt[0..N-1] which represent values and weights associated with N
7 items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[]
8 such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the
9 complete item, or don't pick it (0-1 property).
10
11 Example 1:
12 Input:
13 N = 3
14 W = 4
15 values[] = {1,2,3}
16 weight[] = {4,5,1}
17 Output: 3
18
19 Example 2:
20 Input:
21 N = 3
22 W = 3
23 values[] = {1,2,3}
24 weight[] = {4,5,6}
25 Output: 0
26
27 Your Task:
28 Complete the function knapSack() which takes maximum capacity W, weight array wt[], value array val[] and number of items
29 n as a parameter and returns the maximum possible value you can get.
30 Expected Time Complexity: O(N*W).
31 Expected Auxiliary Space: O(N*W)
32 Constraints:
33 1 ≤ N ≤ 1000
34 1 ≤ W ≤ 1000
35 1 ≤ wt[i] ≤ 1000
36 1 ≤ v[i] ≤ 1000
37 """
38 # Time Complexity -> O(N*W)
39 class Solution:
40
41     #Function to return max value that can be put in knapsack of capacity W.
42     def knapSack(self,W, wt, val, n):
43         dp = [[0] * (W+1) for _ in range(n+1)]
44
45         for i in range(1, n+1):
46             for j in range(1, W+1):
47                 if wt[i-1] <= j:
48                     dp[i][j] = max(val[i-1] + dp[i-1][j-wt[i-1]], dp[i-1][j])
49                 else:
50                     dp[i][j] = dp[i-1][j]
51
52         return dp[-1][-1]
53

```

Figure 4.4: An Example of Code Solution from GitHub Repository Source (Kaushik, 2022)

To collect a list of codes with its NL description from its source, we use a specific collection method for each source because of that each source has own UI's and way of categorizing and viewing of its data. In general, we use three types of collection methods, that are:

- Web scraping: for the websites has a unified structure of HTML design. We use urllib (Urllib — URL Handling Modules — Python 3.7.2 Documentation, 2001) Python library to get HTML content of page, and BeautifulSoup (Beautiful Soup Documentation — BeautifulSoup 4.9.0 Documentation, 2015) Python library to parse HTML content. In some GitHub repositories, we use web scraping.

- API: for the most of GitHub repositories. In this method, we use GitHub REST API (GitHub REST API, 2022).
- Manual: for the sources that we cannot collect data from it by Web scraping because of complex structure of HTML design or UI design, and GitHub repositories have non-unified mechanism of ordering of code files.

In collecting the data, we faced some challenges. We describe the main challenges in data collection of PyCorpus as follows:

- Each source has own structure of pages or files. For the sources that we use Web Scraping to collect data from it, we design a different scraper and collector for each one.
- In the most of website sources, we use Web Scraping method. The challenge of building web scraper, collects codes with its NL description, is the structure of pages. A website categorizes Python code examples or exercises into categories, like array, string, files, etc. That means that the web scraper has to parse links and subpages, or subpages of subpages, and then correctly get the required data from it.
- Some websites list many Python codes in a single page, but others show each code example/exercise in a separated page.
- In the most of GitHub repositories, we use GitHub REST API (GitHub REST API, 2022). However, each repository designs and presents its content in a different structure from other repositories. Some repositories group its content as folders and subfolders, and some of it does not use folder. So, we design a different mechanism of using and handling data from the GitHub REST API (GitHub REST API, 2022).
- Some GitHub repositories list many Python codes in a single file, but others show each code example/exercise in a separated file.
- How to extract the Python code from each snippet. A web scraper or API client has to find the part of Python code from the retrieved content of page or file, for both page/file contains one snippet or many snippets.
- How to extract the NL description of code. In the most of websites, the NL description be viewed before the Python code snippet, so the web scraper defines the HTML part of NL description and the HTML part of the Python code. As for the GitHub repositories, some of it present the NL description in a comment, as one or multiple lines, within the Python code, but others present the NL description in the name of

Python file, especially when the NL description be a name of algorithm or problem as a single term. So, the identifying of NL description of code is a hard challengeable task. Finally, we collect PyCorpus of 5013 NL-Code pairs. Table 4.2 presents the statistical details of PyCorpus.

Table 4.1 presents the used sources of PyCorpus, and type of source, type of collection method and position of NL description of code for each source. The sources, in Table 4.1, sorted by date order of use it, from oldest to newest.

Table 4.1: Description of Source and Method of Collecting PyCorpus

#	Reference	Source Name	Type	Description	Structure	Collection Method	NL position
1	(GeeksforGeeks, 2022)	GeeksforGeeks	Website	A website provides programming tutorials.	The code examples are divided into topics. Each example is in a separated page.	Web Scraper	In the same page of code, before code snippet.
2	(Kaushik, 2022)	Competitive_Programming	GitHub Repository	A GitHub repository provides solutions for questions from various common coding websites.	Each solution in a separated Python file. The files are grouped into folders according to source coding website.	GitHub REST API	As a multi-line comment within Python code file.
3	(Yuan, 2015)	CC150_Python	GitHub Repository	A GitHub repository provides solutions for common Python interview questions.	Each solution in a separated Python file. The files are grouped into two folders.	Web Scraping	As a multi-line comment within Python code file.
4	(Build Software Better, Together, 2019)	Leet_Code	GitHub Repository	A GitHub repository provides solutions for questions from LeetCode (LeetCode - the World's Leading Online Programming Learning Platform, 2022) online programming website.	Each solution in a separated Python file.	Web Scraping	Name of the Python file.

5	(Janke, 2019)	LeetCode-Solutions	GitHub Repository	A GitHub repository provides solutions for questions from LeetCode (LeetCode - the World's Leading Online Programming Learning Platform, 2022) online programming website.	Each solution in a separated file. Each file in a separated folder.	GitHub REST API	Name of the folder that contains the solution Python file.
6	(Kumar, 2022)	python-interview-questions	GitHub Repository	A GitHub repository provides solutions for common Python interview questions.	Each solution in a separated file.	GitHub REST API	Name of the Python file.
7	(Hu, 2018)	Python-programming-exercises	GitHub Repository	A GitHub repository provides solutions for common Python challenge programming exercises.	All codes, with its NL description, in a one file.	Manually	As a multi-line comment within Python code file.
8	(Michele Pratusевич, 2014)	Practice Python	Website	A website for practicing Python by beginner-level Python exercise.	Each exercise in a separated file.	Manually	In the same page of code, before code snippet.
9	(Raj, 2022)	PythonEdabitSolution	GitHub Repository	A GitHub repository provides solutions for some coding challenges from Edabit (Edabit, 2021) website.	Each solution in a separated file. The files are grouped into folders according to hard level.	GitHub REST API	Name of the Python file.
10	(Rashid, 2019)	Edabit-Solutions	GitHub Repository	A GitHub repository provides solutions for some coding challenges from Edabit (Edabit, 2021) website.	Each solution in a file, and each file in a folder.	GitHub REST API	Name of the folder that contains the solution Python file.
11	(Yu, 2017)	coding-problems	GitHub Repository	A GitHub repository provides collection of Python implementations of various coding challenges, puzzles, and interview questions.	Each solution in a file.	GitHub REST API	As a multi-line comment within Python code file.

12	(Galon, 2021)	Challenges	GitHub Repository	A GitHub repository provides solutions for random coding challenges on the internet.	Each solution in a file. The files are grouped into folders according to name of source website of code challenges.	GitHub REST API	As a multi-line comment within Python code file.
13	(Čemisovs, 2020)	sololearn-challenges	GitHub Repository	A GitHub repository provides solutions for problems from SoloLearn Code Coach (Code Coach, n.d.).	Each solution in a file, and each file in a folder. This folders are grouped into folders according to hard level.	GitHub REST API	As a multi-line comment within Python code file.
14	(Trajkovski, 2019)	coding-problems	GitHub Repository	A GitHub repository provides solutions for various coding problems.	Each solution in a file. The files are grouped into folders according to topic (arrays, strings, DP, etc.)	GitHub REST API	As a multi-line comment within Python code file.
15	(Pantyukhin, 2017)	TheAlgorithms/Python	GitHub Repository	A GitHub repository provides implementations of common algorithms, problems, machine learning models.	Each code in a file. The files are grouped into folders according to topic (file, graph, greedy, hash, etc.)	GitHub REST API	Name of the Python file.
16	(Flash, 2019)	Bitesofpy	GitHub Repository	A GitHub repository provides solutions for code exercises.	Each solution in a python file, and the NL description in a HTML file. Each code file and its description file putted in own folder.	GitHub REST API	In a file separated from the code file.
17	(Biswas, 2019)	coding-interview-gym	GitHub Repository	A GitHub repository contains around 700 problems in total, with solutions using Python.	Each solution in a python file.	GitHub REST API	Name of the Python file.
18	(PYnative, 2018)	PYnative	Website	A website for learning Python with tutorials, exercise and quizzes.	The exercises are grouped by the topic. Each topic contains a list of exercises.	Manually	Codes and NL descriptions in the same page.

19	(Web Development Tutorials From, 2011)	w3resouce	Website	A website for learning programming with tutorials and exercise.	The exercises are grouped by the topic. Each topic page contains list of hyperlinks to exercises pages. Each exercise in a page.	Web Scraper	In the same page of code, before code snippet.
20	(Corsmeier, 2019)	Natural-Language-Processing-with-Python-Analyzing-Text-with-the-Natural-Language-Toolkit	GitHub Repository	A GitHub repository presents solutions to select exercises from “Natural Language Processing with Python – Analyzing Text with the Natural Language Toolkit” (Bird et al., 2009) book.	The solution notebooks are grouped into folders by chapter number. Each solution notebooks contains all solutions with NL descriptions.	Manually	Many codes and its NL descriptions in the same notebook.

4.2 Data Preparation

In the most of NLP research and application works, data pre-processing, including text cleaning, is an essential step to create better text representation leads to less complexity model and better results. However, in our research, we make only the basic steps of data preparation without deep pre-processing and text cleaning. We suppose that the NL-code pairs dataset, with punctuations, markup tags, digit repetition, code comments, and other similar pieces, supports our research goal of building a model is able to deal with wrong or incomplete NL input, and generate developer-style commented code.

For this principle, we build a preparation steps as follows:

1. Read the dataset file, then **split** NL-code pairs, then split each pair into NL input and code output.
2. **Tokenize NL inputs** using spaCy tokenize (Tokenizer · spaCy API Documentation, 2016).
3. **Tokenize code outputs** using tokenize library (Tokenize - Tokenizer for Python Source, 2022). tokenize library is a tokenizer for Python code. We use tokenize library because the regular tokenizers are meant to tokenize English sentences and are not optimized towards Python's syntax (Shah, 2021).
4. Use of **data augmentations** to **mask the common names of variables** (like var_1, var_2), to ensure that the model does not fixate on variable naming way, and actually tries to understand logic and syntax of Python code. In this step, we avoid keywords,

control structures and object properties in the masking. Figure 4.5 shows an example of code before and after augmenting Python code.

5. **Shuffle** the dataset records.
6. **Build the training set and validation set.** Table 4.2 presents the records distribution between training and validation.

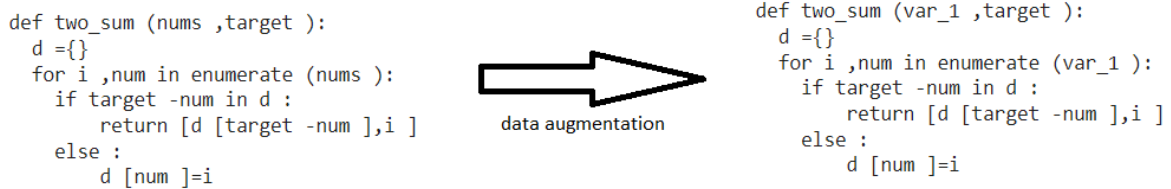


Figure 4.5: An Example of Python Code before and after Data Augmentation

4.3 Training Model

In our research, we use (Shah, 2021) that is open-source pre-trained model for English-to-Python converting. The pre-trained model is transformer-based with self-attention layer. We fine-tune and customize the model for PyCorpus and goal task.

In this section, we present the model structure and details of fine-tuning that we apply.

4.3.1 Previous Attempts

Before we select (Shah, 2021) pre-trained model for our training model, we make some tries to build a Machine Translation model from scratch. We briefly describe two of these tries.

The first one is neural machine translation with attention using TensorFlow. In this model, we use basic three components in the encoder-decoder, that are: attention layer, GRU layer and embedding layer. We set embedding dimension to 256 and epochs to 1000, and Adam optimizer is used in fitting. The main issues with this model are (1) too long training time that may be weeks, and (2) need for huge amount of data to achieve a good learning curve.

The second one is building a machine translation model with Addon Networks using TensorFlow. Actually, we find that the Addon Networks MT model is appropriate for short sentences in input and output. It is too slow in training and testing for long sentences.

So, when the approach of building a MT model from scratch, we decide to use an appropriate pre-trained model.

4.3.2 The Structure of Training Model

The pre-trained model (Shah, 2021), that we use, is a Transformer-Based with Self-Attention. Generally, a pre-trained model is a saved network/structure of a model that was previously trained on a dataset (Transfer Learning and Fine-tuning | TensorFlow Core, 2022).

The Attention mechanism is proposed to overcome the limitation of the Encoder-Decoder model that has a problem with long sequence, because the encoding of long sequence makes difficult for neural networks to cope with it (Agrawal, 2021; Self -attention in NLP - GeeksforGeeks, 2020). The Attention mechanism is a part of neural architecture helps the Encoder-Decoder model to focus on relevant features of the text data (Galassi et al., 2021) by allows the output to focus on the features of input (Self -attention in NLP - GeeksforGeeks, 2020). The self-attention allows inputs to interact with each other, and enables the model to deal with all of its (Self -attention in NLP - GeeksforGeeks, 2020).

The Transformer is a machine learning model adopts the Attention mechanism for sequence processing by process the entire input all at once; not word by word (Transformer (machine learning model) - Wikipedia, 2022).

Our transformer consists of three components that are (Shah, 2021):

- An Encoder that encodes the input sequence into representation vectors.
- An Attention that enables the Transformer model to focus on the right aspects of sequence, and support contextualization of input data in Encoder and Decoder.
- A Decoder that decodes the representation vector to target output sequence.

The transformer, with its components, is implemented for our Seq2Seq problem, followed by Adam optimizer (Team, 2017). In the following sub-sections, we describe the detailed structure of each component.

▪ Encoder

The Encoder, which supported by the Attention mechanism, takes a batch of sequences and sequences masks as input (Shah, 2021). The input represented as vector of 1 (in locations of valid values in the sequence) and 0 (where the sequence has values) (Shah, 2021). The input representation ensures that the Attention mechanism will pay attention for relevant features, not for values (Shah, 2021).

Figure 4.6 (Shah, 2021) presents an abstract view of Encoder structure. The Encoder consists of five components, that are (Shah, 2021):

1. **Token Embedding** for converting the source sequence into tokens.

2. **Position Embedding** for tagging each token, with its positional index, to sequential information.
3. **Encoder Layer** for implementing Transformer's Encoder component.
4. **Dropout** layer.
5. **Sqrt** (square root) **Scaling** layer.

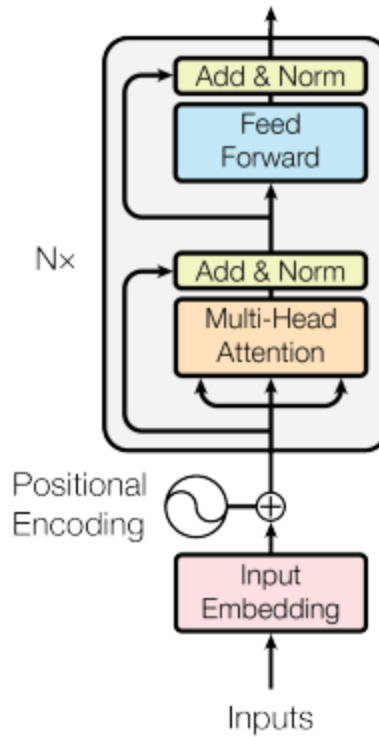


Figure 4.6: Structure of Encoder in our Model (Shah, 2021)

▪ Encoder Layer

The Encoder Layer is the basic building block of the Transformer's Encoder.

The masked source tensor is entered into multi-head self-attention operation to help the model focus on relevant features. Then, the source tensor is normalized to prevent vanishing and exploding gradients during training. Then, the ready tensor is sent to Position-wise Feed-Forward Layer with Relu activation function (Shah, 2021).

▪ Attention

Attention accepts the input as a Query, and represents it as Query Vector. Then, it extracts the indices, from Query, and represents it as Key vector, and match it up to the right attention weights as Value vector (Shah, 2021).

Attention, in our model, is implemented by multi-head attention layer that is used to process an entire long sequence or batch of sequences in parallel. This increases the learning capacity of the model and therefore leads to better results (Shah, 2021).

▪ Decoder

The structure of Decoder is similar to the structure of Encoder, but the Decoder depends on an input combined from two sources: the target sequence and the state representation of vector for the Encoder (Shah, 2021). Figure 4.7 (Shah, 2021). presents the structure of the Decoder in our model.

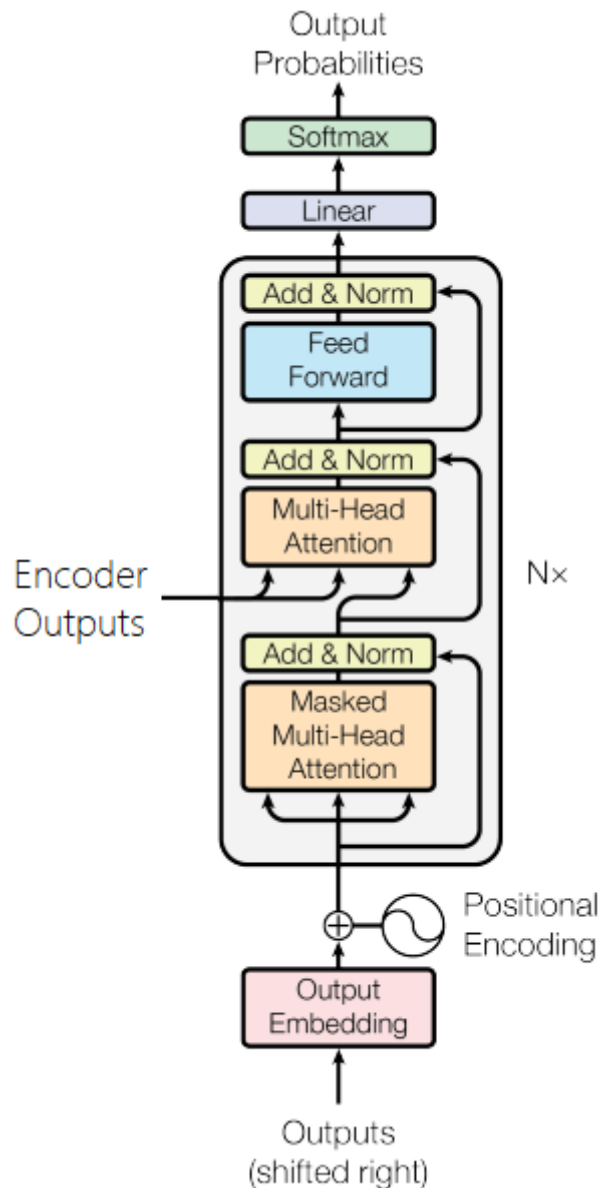


Figure 4.7: Structure of Decoder in our Model (Shah, 2021)

▪ Decoder Layer

The Decoder Layer forms the building block of our Transformer's decoder. Each Decoder Layer involves two attention operations: Self-attention on target embedding, and Multi-head attention operation that uses the target as query vector and the encoder outputs act as the key and value vectors (Shah, 2021).

The presence of an extra Multi-head attention operation differentiates the Decoder Layer from an Encoder Layer (Shah, 2021).

The attention outputs from self-attention are normalized and combined with the trg embedding using a residual connection (Shah, 2021). This is then sent into the multi-head attention operation along with the encoder outputs (Shah, 2021). The attention layer outputs are then combined with the target input again and normalized before sending it into the position-wise feedforward layer to generate the final outputs of the Decoder Layer (Shah, 2021).

The purpose of all normalization operations is to prevent vanishing and exploding gradients during training (Shah, 2021).

4.3.3 Hyperparameters and Fine-tuning

As we mentioned before, we use a pre-trained Transformer-Based Model (Shah, 2021). In the section 4.3.1, we describe the structure of Seq2Seq model that implements Transformer with multiple types of Attention mechanism.

In general, a pre-trained model is a saved network/structure of a model that was previously trained on a dataset (Transfer Learning and Fine-tuning | TensorFlow Core, 2022). In our research work, we apply transfer learning to use only the structure of the model and customize the hyperparameters of the model and training for our research task.

Table 4.2 briefly describes the used hyperparameters, and Table 4.3 lists the values of hyperparameters in the pre-trained model, and finally Table 4.4 describes our fine-tuning.

Table 4.2: Description of Hyperparameters of our Model

Name of Hyperparameter	Description of the Hyperparameter
Input Dimension	Input vocab size \times Embedding length.
Output Dimension	Output vocab size \times Embedding length.
Hidden Size	The number of Attention nodes.
Encoder Layers	The number of Encoder Layers in the Encoder.
Decoder Layers	The number of Decoder Layers in the Decoder.

Encoder Heads	The number of heads in Attention layer in the Encoder.
Decoder Heads	The number of heads in Attention layer in the Decoder.
Encoder PF Dimension	The dimension of Position-Wise Feed-Forward network in the Encoder.
Decoder PF Dimension	The dimension of Position-Wise Feed-Forward network in the Decoder.
Encoder Dropout	The value of dropout parameter of Dropout layer in the Encoder.
Decoder Dropout	The value of dropout parameter of Dropout layer in the Decoder.
Learning Rate	The step size at each iteration in training. It passed as parameter for the optimizer.
Batch Size	The number of training records taken in each iteration.
Epochs	The number of training iterations.

Table 4.3: Values of Hyperparameters in the Pre-Trained Model

Hyperparameter	Value of the Hyperparameter in the Pre-Trained Model
Hidden Size	256
Encoder Layers	3
Decoder Layers	3
Encoder Heads	16
Decoder Heads	16
Encoder PF Dimension	512
Decoder PF Dimension	512
Encoder Dropout	0.1

Decoder Dropout	0.1
Learning Rate	0.0005
Batch Size	16
Epochs	50

Table 4.4 describes the fine-tuning of hyperparameters of the pre-trained model that we use. In the fine-tuning, we use “try and measure” method to find the optimal values.

Table 4.4: Value of Hyperparameters after our Fine-Tuning

Hyperparameter	Value of it after Fine-Tuning	Reason of tune it	Description of change it
Hidden Size	16	Reduce the Hidden Size to reduce the complexity of model for faster training time.	The initial value is 256. We make 4 tries to find the value = 16. In each try, the value is divided by 2.
Encoder Layers	4	Increase Encoder Layers for better processing for better results.	The initial value is 3. When we try to increase it by 1, the loss value is decreased but the training time is highly increased. So, so we finally select 4 as the number of layers.
Decoder Layers	4	Increase Decoder Layers for better processing for better results.	The initial value is 3. When we try to increase it by 1, the loss value is decreased but the training time is highly increased. So, so we finally select 4 as the number of layers.
Batch Size	1	Reduce Batch Size to avoid Out of Memory.	The initial value is 16. The script, with PyCorpus, cannot be run because of OOM error. We make many tries by dividing it by 2, until we reach the minimum value that is 1.

Epochs	100	Increase Epochs for more learning to better results.	The initial value is 50. We try 100, 150, 200, 500 epochs. Actually, the 100 epochs considered as saturation point.
--------	-----	--	---

4.4 Evaluation Metrics

There are a lot of evaluation metrics for Machine Learning and NLP models. For Machine Translation, there are a specific set dedicated for it. It is common to use BLEU, METEOR, NIST and others (Han et al., 2016). For Code Generation, other evaluation metrics used because the special nature of the task.

In our research work, we use Machine Translation for Automatic Code Generation. So, we evaluate our work using some evaluation metrics of Machine Translation and other for Code Generation. We use three evaluation metrics that are: BLEU, METEOR and Python compilation.

BLEU (Bilingual Evaluation Understudy) measures the quality of output text by correspondence between a machine's output and that of a human (which in the validation set) word (BLEU - Wikipedia, 2021). Figure 4.8 (Doshi, 2021) shows the mathematical formula of BLUE score, which use N-Gram (1 to 4) outputs.

$$\begin{aligned}
 \log Bleu &= \min(1 - \frac{r}{c}, 0) + \sum_{n=1}^4 \frac{\log p_n}{4} \\
 &= \min(1 - \frac{r}{c}, 0) + \frac{\log p_1 + \log p_2 + \log p_3 + \log p_4}{4}
 \end{aligned}$$

Figure 4.8: Mathematical Formula of BLUE Score (Doshi, 2021)

METEOR (Metric for Evaluation of Translation with Explicit ORdering) measures the correlation between machine-translated text and human text by the harmonic mean of unigram precision and recall, with recall weighted higher than precision (METEOR - Wikipedia, 2021). Figure 4.9 shows a simple representation of mathematical formula of METEOR that depends on precision and recall.

$$M = F_{\text{mean}} (\text{of precision and recall}) \times (1 - \text{Penalty})$$

Figure 4.10: Mathematical Formula of METEOR Score (METEOR - Wikipedia, 2021)

For the third evaluation metric, we compute the percentage of correctly compiled Python output codes from all outputted codes. Figure 4.8 shows a simple representation of mathematical computation of % of compiled Python output codes.

$$\% \text{ of correctly compiled codes} = \frac{\text{correctly compiled codes in the validation set}}{\text{number of codes in the validation set}} \times 100$$

Figure 4.9: Mathematical Computation of % of Compiled Python Output Codes

In the comparison between our fine-tuned model and other or baseline model, we focus on the BLEU score because it is a commonly used metric in Machine Translation. The BLEU metric ranges from 0 to 1 (Brownlee, 2017). Few translations will attain a score of 1 unless they are identical to a reference translation (Brownlee, 2017). A score of 0.6 or 0.7 is considered the best you can achieve (Doshi, 2021). Generally, the good range of BLEU score in human language to human language translation is 0.25 to 0.34 (Brownlee, 2017). In the ACG, the good range of BLEU score is 0.18 to 0.21 (Hu et al., 2019).

Chapter 5

Experiments and Results

Chapter 5 Experiments and Results

This chapter describes our experiments we conducted to evaluate our method proposed for Automatic Code Generation, and the final results of our research work including the dataset and ACG model results. First, we briefly describe the technical specification of environment and libraries. Second, we describe the features of PyCorpus dataset. Third, we describe and discuss the results of these experiments. Finally, we asses our method compared to the pre-trained model we used.

5.1 Experiments Tools

To implement our approach, we use Python programming language on Google Colab Jupyter notebook environment that supports the most of Python libraries and run on the cloud. We use the Pro+ premium plan of Google Colab.

We use many Python libraries, such as:

1. **PyTorch** (PyTorch, 2019): is an open-source Python library to develop and train ML and NLP models.
2. **torchtext** (Torchtext — Torchtext 0.13.0 Documentation, 2022): is a part of PyTorch (PyTorch, 2019) library, which contains utilities and implementations for NLP. We use it our Seq2Seq modelling
3. **spaCy** (Tokenizer · spaCy API Documentation, 2016): is a Python library for text tokenization. We use it to tokenize NL inputs before training.
4. **tokenize** (Tokenize - Tokenizer for Python Source, 2022): is a Python library for tokenization of Python code text. We use it to tokenize code outputs in PyCorpus before training.
5. **pandas** (Pandas - Python Data Analysis Library, 2018): is a one of most common Python libraries provides data structures and data handling methods like formatting, grouping, filtering, sorting and others.
6. **NumPy** (NumPy, 2019): is a one of most common Python libraries that used to deal with numbers and sequences of numbers.
7. **nlTK** (NLTK :: Natural Language Toolkit, 2019): is a common Python library provides easy-to-use interfaces for text processing. We use it to apply evaluation by BLEU and METEOR.
8. **ast** (Ast — Abstract Syntax Trees — Python 3.11.0 Documentation, 2022): is a Python library to process and check Python abstract syntax tree.

5.2 PyCorpus Dataset

As we can see in Section 3.1, there are some restrictions on NL input to be formal description, structured text, or a correct and complete sentence. This cause a limitation of output code generation in cases of fully free, short or grammarly wrong NL input.

For more generalization, we collected and built PyCorpus that allows ACG models deal to with fully free, short, or grammarly wrong NL input. We name it as “PyCorpus”. We describe the features of PyCorpus as follows:

- The dataset is designed as NL-code pairs.
- The dataset contains general-purpose data. It is not restricted or specialized in some technical field, such as IO, database, machine learning, or whatever.
- The dataset is open-domain. The data is not limited to some business domains.
- The dataset contains various structures of code, such as scripts, classes, and functions.
- The structure of NL input, in the dataset, is fully free. It can be a sentence, single term, or paragraph. Table 5.2 presents some examples of various structures of NL input in the dataset.
- The metaphor of NL input, in the dataset, is fully free. It can be started with a verb, started with a name, question, or term. Table 5.3 presents some examples of various metaphors of NL input in the dataset.
- Some of the code, in the dataset, contain one or more comments. This enables the ACG model to generate a code with comments.

The dataset contains various structures and metaphors of NL input, such as short sentence, long sentence, multi-line paragraph, single term, and others. Also, it contains various coding styles of code output, such as standardized style and developer-style.

The dataset is composed of 5013 NL-code pairs. Table 5.1 presents the statistics of the dataset in the training model.

Table 5.1: The Statistics of the Dataset in the Model

Number of records	5013
Maximum length of NL (words count)	437
Maximum length of code (words count)	2112

Vocabulary size of NL	14968
Vocabulary size of code	73423
Average number of lines of code	26
Average number of words in code	98
Average number of words in NL	38
Average length of code (# of characters)	771
Average length of NL (# of characters)	226
Number of records in training set	4260 (85% of dataset)
Number of records in validation set	753 (15% of dataset)

Table 5.2: Some Examples of Various Structures of NL Inputs in PyCorpus

Structure of NL	Example
Started with a verb	Write a Python function that takes a sequence of numbers and determines whether all the numbers are different from each other.
Single complete sentence	Sometimes, while working with a python list, we can have a problem in which we need to find positions of all the integers other than 0. This can have application in day-day programming or competitive programming. Let's discuss a shorthand by which we can perform this particular task.
Single complete short sentence	Given a 2D list, write a Python program to find the most common element in each column of the 2D list.

Very short sentence	a Python program to read an entire text file.
Single term	primelib
	rsa vectorization
Question	What are ordered words?

Table 5.3: Some Examples of Various Metaphors of NL Inputs in PyCorpus

Metaphor of NL	NL	Code
Story paragraph	Vishal loves running. He often visits his favourite Nehru Park and runs for very long distances. On one such visit he found that \nthe number of girls in the park was unusually high. Now he wants to use this as an opportunity to impress a large number of girls with his awesome speed.	<pre> for _ in range(int(input())): n,k = map(int,input().split()) arr = list(map(int,input().split())) maxx = 0 for i in range(n-k+1): if maxx < sum(arr[i:k+i]): maxx = sum(arr[i:k+i]) print(maxx) </pre>

	<p>The track on which he runs is an N kilometres long straight path. There are a_i girls standing within the ith kilometre of this path. \nA girl will be impressed only if Vishal is running at his maximum speed when he passes by her. But he can run at his best speed only \nfor a single continuous stretch of K kilometres.</p> <p>Now Vishal wants to know what is the maximum number of girls that he can impress.</p> <p>Input: First line of the input contains the number of testcases T.</p> <p>For each test case,First line contains two space-separated</p>	
--	--	--

	<p>integers N and K, the length of the track and the maximum distance he can run at his best speed.</p> <p>Second line contains N space-separated integers, the number of girls within each kilometre of the track.</p> <p>Output: For each test case print one line containing an integer, denoting the maximum number of girls Vishal can impress.</p>	
Long code description with a sample	<p>Given a non-negative number represented as an array of digits, add 1 to the number (increment the number represented by the digits).</p>	<pre>t = int(input()) while t > 0: input() arr = input().split() result = list(str(int("".join(arr))+1)) while len(arr) > len(result): result.insert(0,'0') print(*result) t -= 1</pre>

	<p>The digits are stored such that the most significant digit is at the head of the list.</p> <p>Example: If the array has [4, 5, 6] the resultant array should be [4, 5, 7] as $456 + 1 = 457$.</p> <p>Input: The first line of input contains an integer T denoting the number of test cases. The description of T test cases follows.</p> <p>The first line of each test case contains a single integer N denoting the size of array.</p> <p>The second line contains N space-separated integers A_1, A_2, \dots, A_N denoting the elements of the array.</p>	
--	--	--

	<p>Output: Print the space separated resultant array in a separate line for each case.</p> <p>Input:</p> <p>2</p> <p>4</p> <p>5 6 7 8</p> <p>3</p> <p>9 9 9</p> <p>Output:</p> <p>5 6 7 9</p> <p>1 0 0 0</p>	
<p>Short real-world description with technical code description with a sample</p>	<p>In a party everyone is in couple except one. People who are in couple have same numbers. Find out the person who is not in couple.</p> <p>Input: The first line contains an integer 'T' denoting the total number of test cases. In each test cases, the first line contains an integer 'N' denoting the</p>	<pre>for _ in range(int(input())): n = int(input()) arr = list(map(int,input().split())) res = 0 for i in arr: res ^= i print(res)</pre>

	<p>size of array.</p> <p>The second line contains N space-separated integers A1, A2, ..., AN denoting the elements of the array. \n(N is always odd)</p> <p>Output: In each seperate line print number of the person not in couple.</p> <p>Constraints:</p> <p>$1 \leq T \leq 30$</p> <p>$1 \leq N \leq 500$</p> <p>$1 \leq A[i] \leq 500$</p> <p>$N \% 2 == 1$</p> <p>Example:</p> <p>Input:</p> <p>1</p> <p>5</p> <p>1 2 3 2 1</p> <p>Output: 3</p>	
--	---	--

<p>Direct and clear technical description of the code required, in one sentence</p>	<p>Find all duplicates in an array where all elements are positive(0<) and the biggest element in the array could be equal to the length of array. Note: solve it in one iteration.</p>	<pre> ===== Each value has its own position/index in the array, mark the value on that position as negative when the element is found for the first time. Time Complexity: O(N) Space Complexity: O(D) , array (in this case set) to save all duplicates In the second solution 2 hashsets are used, one to check if already exists element like current and the other has same functionality as the hashset in the first solution. * This solution is for all kind of numbers (not as the previous solution, only for positive numbers or smaller elements than the length of array). Time Complexity: O(N) Space Complexity: O(D) ... ##### # Solution 1 # ##### def find_duplicates(arr): n = len(arr) duplicates = set() for i in range(n): idx = abs(arr[i]) - 1 val = arr[idx] if val > 0: # mark element as found for the first time arr[idx] = -val else: # this element is a duplicate duplicates.add(idx + 1) return duplicates ##### # Solution 2 # ##### def find_duplicates_2(arr): n = len(arr) duplicates = set() elements = set() for i in range(n): if arr[i] in duplicates: # this element is already found as duplicate continue if arr[i] in elements: # a duplicate is found duplicates.add(arr[i]) elements.remove(arr[i]) else: # a new number elements.add(arr[i]) return duplicates ##### # Testing # ##### # Test 1 # Correct result => [1] print(find_duplicates([1, 1, 1, 1])) print(find_duplicates_2([1, 1, 1, 1])) # Test 2 # Correct result => [4, 2] print(find_duplicates([4, 2, 4, 2, 1, 4])) print(find_duplicates_2([4, 2, 4, 2, 1, 4])) </pre>
---	--	--

<p>Direct and clear technical description of a programming problem.</p>	<p>This problem can be solved using a dictionary (hash map), but in order to use a dictionary you'll need to find a way to calculate the keys for all strings. This is a same solution but 2 different hash functions.</p> <p>Sort the letters from the strings, and use the sorted letters as key.</p> <p>Time Complexity: $O(N * K \log K)$, N = number of strings, K = number of characters (chars in the string with most chars)</p> <p>Space Complexity: $O(N)$</p> <p>Use a letter counter (some kind of counting sort).</p>	<pre># Function to sort an array using # insertion sort def insertionSort(A, size): i, key, j = 0, 0, 0 for i in range(size): key = A[i] j = i-1 # Move elements of A[0..i-1], that are # greater than key, to one position # ahead of their current position. # This loop will run at most k times while j >= 0 and A[j] > key: A[j + 1] = A[j] j = j - 1 A[j + 1] = key</pre>
---	--	---

	<p>Time Complexity: $O(N * K)$, $O(N * K * 26) = O(N * K)$, if all of the strings have several chars (less than ~8) the first hash function is better.</p> <p>Space Complexity: $O(N)$</p>	
--	---	--

<p>Single term of name of algorithm required to be coded</p>	<p>rsa factorization</p>	<p>An RSA prime factor algorithm.</p> <p>The program can efficiently factor RSA prime number given the private key d and public key e. Source: on page 3 of https://crypto.stanford.edu/~dabo/papers/RSA-survey.pdf More readable source: https://www.di-mgt.com.au/rsa_factorize_n.html large number can take minutes to factor, therefore are not included in doctest. """</p> <pre> import math import random from typing import List def rsafactor(d: int, e: int, N: int) -> List[int]: """ This function returns the factors of N, where p*q=N Return: [p, q] We call N the RSA modulus, e the encryption exponent, and d the decryption exponent. The pair (N, e) is the public key. As its name suggests, it is public and is used to encrypt messages. The pair (N, d) is the secret key or private key and is known only to the recipient of encrypted messages. >>> rsafactor(3, 16971, 25777) [149, 173] >>> rsafactor(7331, 11, 27233) [113, 241] >>> rsafactor(4021, 13, 17711) [89, 199] """ k = d * e - 1 p = 0 q = 0 while p == 0: g = random.randint(2, N - 1) t = k while True: if t % 2 == 0: t = t // 2 x = (g ** t) % N y = math.gcd(x - 1, N) if x > 1 and y > 1: p = y q = N // y break # find the correct factors else: break # t is not divisible by 2, break and choose another g return sorted([p, q]) if __name__ == "__main__": import doctest doctest.testmod() </pre>
--	--------------------------	--

Description of problem by very short sentence	792 Number of Matching Subsequences	<pre> from collections import defaultdict class Solution(object): def numMatchingSubseq(self, S, words): waiting = defaultdict(list) for w in words: waiting[w[0]].append(iter(w[1:])) for c in S: for it in waiting.pop(c, ()): waiting[next(it, None)].append(it) return len(waiting[None]) sol = Solution() S = "abcde" words = ["a", "bb", "acd", "ace"] out = sol.numMatchingSubseq(S, words) print('Res: ', out) </pre>
Direct and clear description of required, as a one sentence	Write a Python program to find and print all li tags of a given web page.	<pre> import requests from bs4 import BeautifulSoup url = 'https://www.w3resource.com/' reqs = requests.get(url) soup = BeautifulSoup(reqs.text, 'lxml') print("\nFind and print all li tags:\n") for tag in soup.find_all("li"): print("{0}: {1}".format(tag.name, tag.text)) </pre>

5.3 Results and Discussion

As described in the previous section, the PyCorpus dataset consists of 5013 NL-code pairs. Each pair consists of output Python code mapped to its NL description input. Before training, the dataset is shuffled. 753 of pairs (15% of the dataset) passed to validation process. Figures 5.1, 5.2 and 5.3 show samples of generated Python codes from ours model. As we can see the Figures 5.1, 5.2 and 5.3, our model is able to generate highly acceptable Python code snippets that highly match the requirements mentioned in the free NL input of any text structure, grammar or syntax.

```

▶ print(predictions_df_data_cleaned.iloc[500]['solution'])
# write a python code to get last element occurrence

# Python3 code to demonstrate
# to get last element occurrence
# using join() + rfind()

# initializing list
test_list = ['G', 'e', 'e', 'k', 's', 'f', 'o', 'r',
             'g', 'e', 'e', 'k', 's']

# using join() + rfind()
# to get last element occurrence
res = ''.join(test_list).rindex('e')

# printing result
print ("The index of last element occurrence: " + str(res))

```

Figure 5.1: A Sample from Generated Code from Our Model

```

[33] print(predictions_df_data.iloc[540]['question'])

Write a Python program to search a date from a given string using arrow module.

[35] print(predictions_df_data.iloc[540]['solution'])

import arrow
print("\nSearch a date from a string:")
d1 = arrow.get('David was born in 11 June 2003', 'DD MMMM YYYY')
print(d1)

```

Figure 5.2: A Sample from Generated Code from Our Model

```

▶ print(predictions_df_data.iloc[600]['question'])

Boggle Board

[ ] print(predictions_df_data.iloc[600]['solution'])

def boggleBoard(board, words):
    trie = Trie()
    for word in words:
        trie.add(word)
    finalWords = {}
    visited = [[False for letter in row] for row in board]
    for i in range(len(board)):
        for j in range(len(board[i])):
            explore(i, j, board, trie.root, visited, finalWords)
    return list(finalWords.keys())

def explore(i, j, board, trie, visited, finalWords):
    if visited[i][j] == True:
        return
    letter = board[i][j]
    if letter not in trie:
        return

    visited[i][j] = True
    trieNode = trie[letter]
    if "*" in trieNode:
        finalWords[trieNode["*"]] = True
    neighbours = getNeighbours(i, j, board)
    for neighbour in neighbours:
        explore(neighbour[0], neighbour[1], board, trieNode, visited, finalWords)
    visited[i][j] = False

def getNeighbours(i, j, board):
    neighbours = []
    possibleDirections = [(-1, 0), (-1, 1), (0, 1), (1, 1), (1, 0), (1, -1), (0, -1), (-1, -1)]
    for direction in possibleDirections:
        di, dj = direction
        newI, newJ = i + di, j + dj
        if 0 <= newI < len(board) and 0 <= newJ < len(board[0]):
            neighbours.append([newI, newJ])
    return neighbours

class Trie:
    def __init__(self):
        self.root = {}
        self.endSymbol = "*"

    def add(self, word):
        current = self.root
        for letter in word:
            if letter not in current:
                current[letter] = {}
            current = current[letter]
        current[self.endSymbol] = word

```

Figure 5.3: A Sample from Generated Code from Our Model

We evaluate our fine-tuned model with three metrics that are: BLEU, METEOR and percentage of correctly compiled output codes from the total output codes from the validation. For BLEU

and METEOR, we use nltk (NLTK :: Natural Language Toolkit, 2019) library. For automatic compilation of codes, we use ast (Ast - Abstract Syntax Trees - Python 3.11.0 Documentation, 2022) library.

We think that we can achieve better evaluation results with a large dataset. The mission of collecting NL-code data, characterized by the features of PyCorpus, is hard. However, if we can collect a huge dataset like PyCorpus, we can achieve better results.

To ensure that our research proofs it's the main objective, we compare the evaluation results of our experiments with the pre-trained model before our fine-tuning. We train and validate the pre-trained model on PyCorpus, and compare our method with it using BLEU score. The pre-trained model, before fine-tuning, achieve BLEU score is equals to 0.1 on PyCorpus.

We also try to use other Seq2Seq models. Table 5.4 presents the final results of evaluation of our experiments compared to the pre-trained model and some previous failed tries.

In the from scratch models, we evaluate it using only BLEU score. We don't use the other metrics because the BLEU score is too low, so no need to other.

As a final result, we achieve BLEU score is 0.23 that is in the range of highly good in Machine Translation to non-human language. Generally, the good range of BLEU score in human language to human language translation is 0.25 to 0.34 (Brownlee, 2017). In the ACG, the good range of BLEU score is 0.18 to 0.21 (Hu et al., 2019).

Table 5.4: Results of Evaluation of our Experiments

Model	BLEU	METEOR	% of complied
Pre-Trained (baseline)	0.1	0.12	33.8%
Fine-Tuned	0.23	0.27	43.6%
Tensorflow Attention (from scratch)	0.01		
Tensorflow Addon Network (from scratch)	0.01		

5.4 Threats to Validity

We ensure on that the fine-tuning that we conducted on the used pre-trained model is experimental on PyCorpus dataset. The strategy of the fine-tuning is changed with different or larger dataset.

Also, we focus, in this thesis, on the concept of ability of generating program code from free-text, without taking the text conditions and specific requirements written in the input text. For example, if the input text is “write a function computes the sum of two integers”, then our model is assumed to generate a Python code for computing the sum of two integers, but the output code may be not a function.

Chapter 6

Conclusion and Future Works

Chapter 6 Conclusion and Future Works

6.1 Conclusion

In the recent years, software development companies pay attention to Automatic Code Generation, because of its effects on fast delivery of software product and high quality of software code. There are many research works and technical tools for NL-to-code generation, but the most of its deal with structured or formal-style NL text in a complete sentence and correct grammar manner. Also, the existing works product standardized-style code.

In this research, we propose an approach for NL-to-code generation able to product developer-style commented Python code from short or wrong-structured NL text. We collect a dataset consists of NL-pairs with high variety of NL structures, NL lengths, NL metaphors, coding styles and code technical structure. We implement the proposed approach as a Machine Translation problem by Transformer-Based Model with Attention mechanism that focus on the important features of the text.

Finally, we achieve BLEU score 0.23, METEOR score 0.27 and 43.6% of output code, in the validation, are correctly compiled. For the field of Machine Translation, the 0.23 BLEU score is considered a good result.

6.2 Future Works

In our future work on NL-to-code generation, we need to increase the dataset with more NL-pairs have similar features and variety of NL and code texts. Also, we will try to apply deep fine-tuning or build a new Seq2Seq model can achieve better results.

The Reference List

- 15, C., Enthusiast, C., Chaoran, & Enthusiast, D. (2019, January 15). Build a machine translator using Keras (part-1) seq2seq with lstm. Retrieved December 28, 2020, from <https://6chaoran.wordpress.com/2019/01/15/build-a-machine-translator-using-keras-part-1-seq2seq-with-lstm/>
- Aaron Li-Feng Han, & Derek F. Wong. (2016). Machine Translation Evaluation: A Survey. ArXiv: Computation and Language.
- ACM Digital Library. (2019). Retrieved December 22, 2020, from <https://dl.acm.org/>
- Agrawal, N. (2021, December 13). Understanding Attention Mechanism: Natural Language Processing. Medium. <https://medium.com/analytics-vidhya/https-medium-com-understanding-attention-mechanism-natural-language-processing-9744ab6aed6a>
- Akullp, Awesome Java, (2022), GitHub repository, <https://github.com/akullpp/awesome-java>
- Alon, U., Zilberstein, M., Levy, O., & Yahav, E. (2019). code2vec: Learning distributed representations of code. Proceedings of the ACM on Programming Languages, 3(POPL), 1-210.
- Anant Kaushik, Competitive_Programming, (2020), GitHub repository, https://github.com/anantkaushik/Competitive_Programming
- Andy Yu, coding-problems, (2017), GitHub repository, <https://github.com/andyyu/coding-problems>
- Arnoldus, J., Van den Brand, M., Serebrenik, A., & Brunekreef, J. J. (2012). Code generation with templates (Vol. 1). Springer Science & Business Media.
- ast — Abstract Syntax Trees — Python 3.11.0 documentation. (2022). Retrieved November 5, 2022, from <https://docs.python.org/3/library/ast.html>
- Beautiful Soup Documentation — Beautiful Soup 4.9.0 documentation. (2015). Retrieved October 31, 2022, from <https://www.crummy.com/software/BeautifulSoup/bs4/doc/>

- Bird, S., Klein, E., & Loper, E. (2009). *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit* (1st ed.). O'Reilly Media.
- Biswas, P. (2019). GitHub - partho-maple/coding-interview-gym: leetcode.com , algoexpert.io solutions in python and swift. GitHub. <https://github.com/partho-maple/coding-interview-gym>
- BLEU. (2021, December 02). Retrieved March 05, 2022, from <https://en.wikipedia.org/wiki/BLEU>
- Brownlee, J. (2017). A Gentle Introduction to Calculating the BLEU Score for Text in Python. Machine Learning Mastery. <https://machinelearningmastery.com/calculate-bleu-score-for-text-python/>
- Build software better, together. (n.d.). GitHub. Retrieved November 1, 2022, from https://github.com/akapoor13/Leet_Code
- Čemisovs, J. (2020). GitHub - rabestro/sololearn-challenges: Descriptions of SoloLearn Coding Challenges (all 72, including pro) and solutions in C, Java, Python and Ruby. GitHub. <https://github.com/rabestro/sololearn-challenges>
- Chaubey, M. (2012, July 18). Pseudocode: Advantages, Limitations, And Keywords Used To Write Pseudocode. India Study Channel. <https://www.indiastudychannel.com/resources/153644-Pseudocode-Advantages-Limitations-And-Keyw-ords-Used-To-Write-Pseudocode.aspx>
- Choudhury, A. (2020, November 21). 10 Most Used Databases By Developers In 2020. <https://analyticsindiamag.com/10-most-used-databases-by-developers-in-2020/>
- Code coach. (n.d.). SoloLearn. Retrieved November 4, 2022, from <https://www.sololearn.com/Discuss/2113498/code-coach>
- Code generation (compiler). (2020, November 06). Retrieved December 27, 2020, from [https://en.wikipedia.org/wiki/Code_generation_\(compiler\)](https://en.wikipedia.org/wiki/Code_generation_(compiler))
- Code. (n.d.). Retrieved December 27, 2020, from <https://www.merriam-webster.com/dictionary/code>
- CodePilot.ai. (2020). Retrieved March 24, 2022, from <https://codepilot.netlify.app/>
- Corsmeier, M. (2019, December). GitHub - Sturzgefah- r/Natural-Language-Processing-with-Python-Analyzing-Text-with-the-Natural-Language-Toolkit: My solutions to selected

exercises to “Natural Language Processing with Python – Analyzing Text with the Natural Language Toolkit” by Steven Bird, Ewan Klein, and Edward Loper. GitHub. <https://github.com/Sturzgefahr/Natural-Language-Processing-with-Python-Analyzing-Text-with-the-Natural-Language-Toolkit>

DanglingElseDanglingElse 22344 silver badges1212 bronze badges, & Juan Aguilar GuisadoJuan Aguilar Guisado 1. (1963, December 1). Difference between code generation and model-to-text transformation. Stack Overflow. <https://stackoverflow.com/questions/26379261/difference-between-code-generation-and-model-to-text-transformation>

Define SQL and state the differences between SQL and other conventional programming Languages.: Practice: GeeksforGeeks. GeeksforGeeks Courses. (2017, December 12). <https://practice.geeksforgeeks.org/problems/define-sql-and-state-the-differences-between-sql-and-other-conventional-programming-languages>

Doshi, K. (2021, May). Foundations of NLP Explained — Bleu Score and WER Metrics. Towards Data Science. <https://towardsdatascience.com/foundations-of-nlp-explained-bleu-score-and-wer-metrics-1a5ba06d812b>

Edabit. (2021). Retrieved November 3, 2022, from <https://edabit.com/>

Elghohary, A., Hosseini, S., & Awadallah, A. H. (2020). Speak to your Parser: Interactive Text-to-SQL with Natural Language Feedback. arXiv preprint arXiv:2005.025310.


EMNLP 2020. (2022). Retrieved December 22, 2020, from <https://2020.emnlp.org/>

FarzanRashid, Edabit-Solutions, (2019), GitHub repository,
<https://github.com/FarzanRashid/Edabit-Solutions>

Find and share research. (2019). Retrieved December 22, 2020, from <https://www.researchgate.net/>

Flash, bitesofpy, (2020), GitHub repository, <https://github.com/etoFlash/bitesofpy>

Flash. (2019). GitHub - etoFlash/bitesofpy: My solutions for exercises (PyBites) from <https://codechalleng.es/bites/>. GitHub. <https://github.com/etoFlash/bitesofpy>

- Galon, N. (2021, July). GitHub - nirgn975/challenges: My solutions for random coding challenges . GitHub. <https://github.com/nirgn975/Challenges>
- Galassi, A., Lippi, M., & Torroni, P. (2021). Attention in Natural Language Processing. *IEEE Transactions on Neural Networks and Learning Systems*, 32(10), 4291–4308. <https://doi.org/10.1109/tnnls.2020.3019893>
- Gavilanes, A., Martín, P. J., & Torres, R. (2009, May). A tool for automatic code generation from schemas. In *International Conference on Computational Science* (pp. 63-73). Springer, Berlin, Heidelberg.
- GeeksforGeeks. (2020, September 5). Self -attention in NLP. <https://www.geeksforgeeks.org/self-attention-in-nlp-2/>
- GeeksforGeeks. (2022). GeeksforGeeks | A computer science portal for geeks. Retrieved November 1, 2022, from <https://www.geeksforgeeks.org/>
- GitHub REST API. (2022). GitHub Docs. Retrieved October 31, 2022, from <https://docs.github.com/en/rest>
- Google Colab. (2019). Retrieved January 9, 2021, from <https://colab.research.google.com/notebooks/intro.ipynb>
- Google Colaboratory. (2019). Retrieved November 5, 2022, from <https://colab.research.google.com/>
- Google Drive. (2019). Retrieved January 9, 2021, from https://www.google.com/intl/ar_tn/drive/
- Google Scholar. (n.d.). Retrieved December 22, 2020, from <https://scholar.google.com/>
- Hu, H., Chen, Q., & Liu, Z. (2019, December). Code Generation from Supervised Code Embeddings. In *International Conference on Neural Information Processing* (pp. 388-396). Springer, Cham.
- Info@erdplus.com. (2021). ERDPlus. Retrieved March 24, 2022, from <https://erdplus.com/>
- Iyer, S., Konstas, I., Cheung, A., & Zettlemoyer, L. (2018). Mapping language to code in programmatic context. *arXiv preprint arXiv:1808.09588*.

- Janke, J. (2019, October). GitHub - novajon/LeetCode-Solutions: Just some random LeetCode questions I worked on. GitHub. <https://github.com/novajon/LeetCode-Solutions>
- Jeffrey Hu, Python-programming-exercises, (2020), GitHub repository, <https://github.com/zhiwehu/Python-programming-exercises>
- Jegors Čemisovs, sololearn-challenges, (2020), GitHub repository, <https://github.com/rabestro/sololearn-challenges>
- Jurafsky, D. (2000). *Speech & language processing*. Pearson Education India.
- Kaushik, A. (2022, February). GitHub - anantkaushik/Competitive_Programming: Questions solved from Various Coding websites viz. HackerRank, HackerEarth, CodeChef and other websites. This repository also contains Questions from various offline and onsite competitions. GitHub. https://github.com/anantkaushik/Competitive_Programming
- Korenblat, K., & Ravve, E. V. (2019, September). Automatic Code Generator for Screen Based Systems. In European Conference on Advances in Databases and Information Systems (pp. 253-265). Springer, Cham.
- Kulal, S., Pasupat, P., Chandra, K., Lee, M., Padon, O., Aiken, A., & Liang, P. S. (2019). Spoc: Search-based pseudocode to code. Advances in Neural Information Processing Systems, 32.
- Kumar, P. (2022). GitHub - learning-zone/python-interview-questions: Python Interview Questions (v3.x). GitHub. <https://github.com/learning-zone/python-interview-questions>
- LeetCode - The World's Leading Online Programming Learning Platform. (2022). LeetCode. Retrieved November 2, 2022, from <https://leetcode.com/>
- Li, Z., Jiang, Y., Zhang, X. J., & Xu, H. Y. (2020). The Metric for Automatic Code Generation. Procedia Computer Science, 166, 279-286.
- Liao, H., Jiang, J., & Zhang, Y. (2010). A Study of Automatic Code Generation. 2010 International Conference on Computational and Information Sciences, 689–691. <https://doi.org/10.1109/ICCIS.2010.171>
- Ling, W., Grefenstette, E., Hermann, K. M., Kočiský, T., Senior, A., Wang, F., & Blunsom, P. (2016). Latent predictor networks for code generation. arXiv preprint arXiv:1603.06744.

Maclay, D. (2000). Click and code. *IEE Review*, 46(3), 25-28.

Matthew Corsmeier, Natural-Language-Processing-with-Python-Analyzing-Text-with-the-Natural-Language-Toolkit, (2019), GitHub repository, <https://github.com/SturzgefahR/Natural-Language-Processing-with-Python-Analyzing-Text-with-the-Natural-Language-Toolkit>

METEOR. (2021, June 09). Retrieved March 05, 2022, from <https://en.wikipedia.org/wiki/METEOR>

Meto Trajkovski, coding-problems, (2020), GitHub repository, <https://github.com/MTrajK/coding-problems>

Michele Pratusевич, michele.pratusевич@gmail.com. (2014). Practice Python. Retrieved November 3, 2022, from <https://www.practicepython.org>

Mihalcea, R., Liu, H., & Lieberman, H. (2006). NLP (Natural Language Processing) for NLP (Natural Language Programming). *CICLing*.

(Mike), M. A. (n.d.). Laravel Code Generator. CrestApps. Retrieved March 24, 2022, from <https://crestapps.com/>

Natural language processing. (2020, December 21). Retrieved December 28, 2020, from https://en.wikipedia.org/wiki/Natural_language_processing

Nir Galon, challenges, (2020), GitHub repository, <https://github.com/nirgn975/Challenges>

NLTK :: Natural Language Toolkit. (2019). Retrieved November 5, 2022, from <https://www.nltk.org/>

NumPy. (n.d.). Retrieved November 5, 2022, from <https://numpy.org/>

pandas - Python Data Analysis Library. (2018). Retrieved November 5, 2022, from <https://pandas.pydata.org/>

Paolone, G., Marinelli, M., Paesani, R., & Di Felice, P. (2020). Automatic Code Generation of MVC Web Applications. *Computers*, 9(3), 56. <https://doi.org/10.3390/computers9030056>

Partho Biswas, coding-interview-gym, (2020), GitHub repository, <https://github.com/partho-maple/coding-interview-gym>

- Pradeep Kumar, python-interview-questions, (2020), GitHub repository, <https://github.com/learning-zone/python-interview-questions>
- PYnative. (2018). Retrieved December 31, 2020, from <https://pynative.com>
- PyTorch. (2019). Retrieved November 5, 2022, from <https://pytorch.org/>
- Python. (n.d.). Retrieved January 9, 2021, from <https://www.python.org/>
- PythonProjects, PythonEdabitSolution, (2020), GitHub repository, <https://github.com/PythonAlgorithms/PythonEdabitSolution>
- Pseudocode to java converter: Pseudoeditor. Pseudocode Editor Online. (n.d.). Retrieved March 24, 2022, from <https://pseudoeditor.com/converters/pseudocode-to-java>
- Raj, D. (2022, May). GitHub - codePerfectPlus/PythonEdabitSolution: Edbait Problem solution using python programming language. GitHub. <https://github.com/codePerfectPlus/PythonEdabitSolution>
- Rashid, F. (2019, October). GitHub - FarzanRashid/Edabit-Solutions: Solutions of edabit challenges. GitHub. <https://github.com/FarzanRashid/Edabit-Solutions>
- ROUGE (metric). (2022, February 12). Retrieved March 05, 2022, from [https://en.wikipedia.org/wiki/ROUGE_\(metric\)](https://en.wikipedia.org/wiki/ROUGE_(metric))
- Sebastian, G., Tesoriero, R., & Gallud, J. A. (2020). Automatic Code Generation for Language-Learning Applications. IEEE Latin America Transactions, 18(08), 1433–1440. <https://doi.org/10.1109/TLA.2020.9111679>
- Shah, D. (2021, March 22). Building a Python Code Generator. Towards Data Science. <https://towardsdatascience.com/building-a-python-code-generator-4b476eec5804>
- SilversRL-Yuan, CC150_Python, (2015), GitHub repository, https://github.com/SilversRL-Yuan/CC150_Python
- Svajlenko, J., & Roy, C. K. (2015, September). Evaluating clone detection tools with bigclonebench. In 2015 IEEE international conference on software maintenance and evolution (ICSME) (pp. 131-140). IEEE.

Team, K. (2017). Keras documentation: Adam. Retrieved November 5, 2022, from <https://keras.io/api/optimizers/adam/>

Text to HTML. TextFixer. (2007). Retrieved March 24, 2022, from <https://www.textfixer.com/html/convert-text-html.php>

Thate, J. M., Kendrick, L. E., & Nadarajah, S. (2004). Caterpillar automatic code generation (No. 2004-01-0894). SAE Technical Paper.

The Algorithms, Python, (2020), GitHub repository, <https://github.com/TheAlgorithms/Python>

tokenize — Tokenizer for Python source — Python 3.11.0 documentation. (2022). Retrieved November 5, 2022, from <https://docs.python.org/3/library/tokenize.html>

Tokenizer · spaCy API Documentation. (2016). Tokenizer. Retrieved November 5, 2022, from <https://spacy.io/api/tokenizer/>

torchtext — Torchtext 0.13.0 documentation. (2022). Retrieved November 5, 2022, from <https://pytorch.org/text/stable/index.html>

Transfer learning and fine-tuning | TensorFlow Core. (2022). TensorFlow. Retrieved November 5, 2022, from https://www.tensorflow.org/tutorials/images/transfer_learning

Transform sketches into HTML using AI. Sketch2Code. (2021). Retrieved March 5, 2022, from <https://sketch2code.azurewebsites.net/>

Trajkovski, M. (2019). GitHub - MTrajK/coding-problems: Solutions for various coding/algorithmic problems and many useful resources for learning algorithms and data structures. GitHub. <https://github.com/MTrajK/coding-problems>

Ubuntu 20.04.1 LTS (Focal Fossa). (2018). Retrieved January 9, 2021, from <https://releases.ubuntu.com/20.04/>

Uml-lab.com. 2022. Modeling and Code Generation - Yatta Solutions. [online] Available at: <https://www.uml-lab.com/en/uml-lab/tutorials/modellierung-und-codegenerierung/> [Accessed 17 April 2022].

Unknown, Leet_Code, (Unknown), GitHub repository (Deleted), https://github.com/akapoor13/Leet_Code

- urllib — URL handling modules — Python 3.11.0 documentation. (2001). Retrieved October 31, 2022, from <https://docs.python.org/3/library/urllib.html>
- w3resource. (2011). Retrieved December 31, 2020, from <https://www.w3resource.com/>
- Wang, B., Shin, R., Liu, X., Polozov, O., & Richardson, M. (2019). Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. arXiv preprint arXiv:1911.04942.
- web development tutorials from. (2011). W3resource. Retrieved November 4, 2022, from <https://www.w3resource.com>
- What is Java Bytecode and how does it work? Edureka. (2020, April 24). Retrieved December 27, 2020, from <https://www.edureka.co/blog/java-bytecode/>
- Welcome to. (2022, November 1). Python.org. <https://www.python.org/>
- Wikimedia Foundation. (2020, June 29). Automatic programming. Wikipedia. https://en.wikipedia.org/wiki/Automatic_programming
- Wikipedia contributors. (2022, November 2). Transformer (machine learning model). Wikipedia. [https://en.wikipedia.org/wiki/Transformer_\(machine_learning_model\)](https://en.wikipedia.org/wiki/Transformer_(machine_learning_model))
- Xu, F. F., Jiang, Z., Yin, P., Vasilescu, B., & Neubig, G. (2020). Incorporating external knowledge through pre-training for natural language to code generation. arXiv preprint arXiv:2004.09015.
- Yin, P., Deng, B., Chen, E., Vasilescu, B., & Neubig, G. (2018, May). Learning to mine aligned code and natural language pairs from stack overflow. In 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR) (pp. 476-486). IEEE.
- Yu, A. (2017). GitHub - andyyu/coding-problems: A collection of Python implementations of various coding challenges, puzzles, and interview questions. GitHub. <https://github.com/andyyu/coding-problems>
- Yu, T., Zhang, R., Er, H. Y., Li, S., Xue, E., Pang, B., ... & Radev, D. (2019). Cosql: A conversational text-to-sql challenge towards cross-domain natural language interfaces to databases. arXiv preprint arXiv:1909.05378.

- Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., ... & Radev, D. (2018). Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. arXiv preprint arXiv:1809.08887.
- Yuan, S. (2015, February 28). GitHub - SilversRL-Yuan/CC150_Python: Cracking the Coding Interview Questions and Answers in Python. GitHub. https://github.com/SilversRL-Yuan/CC150_Python
- Zeng, J., Lin, X. V., Xiong, C., Socher, R., Lyu, M. R., King, I., & Hoi, S. C. (2020). Photon: A Robust Cross-Domain Text-to-SQL System. arXiv preprint arXiv:2007.15280.
- Zhong, M., Liu, G., Li, H., Kuang, J., Zeng, J., & Wang, M. (2022). CodeGen-Test: An Automatic Code Generation Model Integrating Program Test Information. arXiv preprint arXiv:2202.07612.
- Zhong, R., Stern, M., & Klein, D. (2020). Semantic Scaffolds for Pseudocode-to-Code Generation. arXiv preprint arXiv:2005.05927.