NLP/AI Based Techniques for Programming Exercises Generation

Tiago Carvalho Freitas
ALGORITMI Research Centre/LASI, University of Minho, Braga, Portugal

Alvaro Costa Neto
Instituto Federal de Educação, Ciência e Tecnologia de São Paulo, Barretos, Brazil

Maria João Varanda Pereira
Research Centre in Digitalization and Intelligent Robotics,
Polythechnic Insitute of Bragança, Portugal

Pedro Rangel Henriques
ALGORITMI Research Centre/LASI, University of Minho, Braga, Portugal

Abstract

This paper focuses on the enhancement of computer programming exercises generation to the benefit of both students and teachers. By exploring Natural Language Processing (NLP) and Machine Learning (ML) methods for automatic generation of text and source code, it is possible to semi-automatically construct programming exercises, aiding teachers to reduce redundant work and more easily apply active learning methodologies. This would not only allow them to still play a leading role in the teaching-learning process, but also provide students a better and more interactive learning experience. If embedded in a widely accessible website, an exercises generator with these Artificial Intelligence (AI) methods might be used directly by students, in order to obtain randomised lists of exercises for their own study, at their own time. The emergence of new and increasingly powerful technologies, such as the ones utilised by ChatGPT, raises the discussion about their use for exercise generation. Albeit highly capable, monetary and computational costs are still obstacles for wider adoption, as well as the possibility of incorrect results. This paper describes the characteristics and behaviour of several ML models applied and trained for text and code generation and their use to generate computer programming exercises. Finally, an analysis based on correctness and coherence of the resulting exercise statements and complementary source codes generated/produced is presented, and the role that this type of technology can play in a programming exercise automatic generation system is discussed.

2012 ACM Subject Classification Social and professional topics → Computer science education; Software and its engineering → Imperative languages; Computing methodologies → Machine learning; Software and its engineering → Parsers

Keywords and phrases Natural Language Processing, Computer Programming Education, Exercises Generation, Text Generation, Code Generation

Digital Object Identifier 10.4230/OASIcs.ICPEC.2023.9

Funding This work has been supported by FCT – Fundação para a Ciência e Tecnologia within the R&D Units Project Scope: UIDB/00319/2020 and UIDB/05757/2020.

1 Introduction

With technological advancements, computer programming education has become increasingly important in recent years, and along with it, the need for effective methods of teaching programming languages has become a priority.

A large part of the students have a lot of difficulties and consequent low approval rates in introductory computer programming courses, much because of the lack of motivation and attention inside and outside the classes [21]. To overcome those difficulties, students need to

practice frequently solving problems with an increasing complexity. On account of that, one of the major challenges faced by teachers is creating a large number of unique programming exercises that address different levels of difficulty. One possible way to help solve these issues is the development of a system for automatic generation of programming exercises based on Natural Language Processing (NLP) and Artificial Intelligence (AI) techniques capable of generating text and source code. In this paper we present and discuss the preliminary results of a research project focused on the analysis of ML models for text and code generation, aiming at building a generator for programming exercises.

This paper starts with a review at how programming exercises are usually constructed (Section 2), followed by a discussion on modern approaches for text and code generation (Section 3). Then, case studies regarding different generation mechanisms are presented, with the subsequent analysis and discussion of their results (Section 4). Finally, it is presented a summary of the paper and our main contribution, as well as a proposal of a road map for the future construction of a website that automatically generates computer programming exercises (Section 5).

2 Classification and Characterisation of Programming Exercises

Programming exercises are the key object of study for this project. It is then essential to gather the most used formats and understand their common characteristics to group them by type, aiming to automatically create them. In this way it is possible to avoid creating exercises that are unfamiliar to students and teachers.

2.1 Exercise Types

Listed below are some of the most common types of exercises in programming courses, gathered and adapted from the study published in [28]. It is important to note that, although there are additional taxonomies and categories, such as the ones proposed in [4, 26, 23], the former was chosen as it better resembles the structural nature of the exercises to be generated. The purpose in gathering such a variety of types is to present different challenges to students, on several themes. While some types are frequently used in open-ended questions, each one of these types can be adapted to multiple-choice formats, where students must select the correct answer from a list of options. This adaption is not trivial though, since it is necessary to generate not only the correct option, but also a set of incorrect ones.

Seven types were collected:

1. **Code from Scratch:** Students receive a blank sheet to write down the complete solution to a problem. Something that is easy for teachers to prepare, but also presents quite a lot of difficulties to those who are being evaluated. Listing 1 contains an example of a simple exercise that asks to write a function in Python.

Listing 1 Example of *Code from Scratch* type exercises.

```
Write a program in Python that reverses a string.
Input: "cool"
Output: "looc"
```

2. **Code Completion:** To solve this type of problem the students are provided with a sample code containing some blanks to fill. There are usually two approaches: one in which an important part of the code is missing and it is necessary to fill it with the correct statements; and another whose blanks must be filled with values for which the program returns a given output. Listing 2 contains an example in which it is necessary to complete the condition in a selection statement in the body of a certain function in Python whose purpose is described.

Listing 2 Example of *Code Completion* type exercise.

```
Complete the following Python function that receives a list of numbers
and returns another one only with the even numbers.

def even(input_list):
  even_list = []
  for n in input_list:
    if _________:
      even_list.add(n)
  return even_list
```

3. **Code Improvement:** Another approach consists in providing students a complete snippet of code that solves a given problem which should be improved. These improvements may have as objective improving the performance or reducing lines of code. Listing 3 contains an exercise in which it is proposed to rewrite a function in Python to a reduced version.

Listing 3 Example of *Code Improvement* type exercises.

```
Rewrite the following code snippet in order to make it smaller,
mantaining its funcionality.

my_list = list(range(1,100))
res = 0
i = 0
while i < len(my_list):
  if my_list[i] % 3 == 0:
    if my_list[i] % 2 == 1:
      res += my_list[i]
  i += 1
return res
```

4. **Bug Finding:** As the name suggests, in this type of exercise the students must find in code purposefully added bugs without correcting them *per se*. This type of exercise is used to check if students attained the first step of learning to program, that is to be capable of interpreting the code written by others in terms of the syntax or semantics of the language used. Listing 4 contains an exercise in which it is asked to identify bugs in a function and explain in a few lines why they are bugs.

Listing 4 Example of *Bug Finding* type exercises.

```
Identify the bug(s) present in the following code sample.

def my_function(n):
  a = 1
  my_list = [1,2,3]
  if n > 3:
    return n + b
  else:
    return a + my_list[3]
```

5. **Debugging and Fixing:** Interestingly, if the two previous types of exercises are mixed together, the result is also quite common, in which students must rewrite a source code that contains bugs. It is necessary not only to detect them, but also to fix the code.
6. **Code Interpretation:** As the name suggests, in this type of exercise the student has to interpret a given code snippet. There are many possibilities for what is asked, such as: describing its behaviour, identifying its goals, reporting the evolution of a variable's value throughout its execution, etc. Listing 5 contains an exercise in which it is asked to describe what a given function does.

Listing 5 Example of *Code Interpretation* type exercises.

```
Describe the behaviour and the goal of the following function.

def my_function(str):
  if len(str) < 2:
    return ''
  return str[0:2] + str[-2:]
```

7. **Output or State Prediction:** Finally, and in a more traditional format, the students are asked to find out either the output of the program, or the value of a variable at the end or during its run-time. This ends up being very similar to what was discussed in the last item, as it is necessary to analyse and interpret the provided code to obtain the correct solution. Listing 6 contains an exercise where it is asked to indicate what value a function call returns.

Listing 6 Example of *Output or State Prediction* type exercises.

```
Indicate what the following function returns as an output.

def my_function(n):
  my_list = []
  for i in range(n):
    my_list.append(i*i)
  return my_list

my_function(3)
```

2.2 Exercise Components

It is important to understand the components that make up a programming exercise in order to effectively design and create them. There are usually three main components that need to be considered:

- **Problem Statement:** Initial component containing text that is presented to the student. It explains the context and parameters of the problem, and the type of answer they must provide. It can be lengthy if the topic needs an introduction or concise and objective if it only requires the essential instructions for solving the problem. It is important to provide enough information to solve the problem, specially when the domain is unknown to the student.
- **Code:** Most programming exercises contain a section dedicated to code snippets. This section may vary in form, containing whole programs, snippets with blanks to be filled, and correct or erroneous versions to be analysed of fixed. The purpose of this component is to support the *Problem Statement* to establish a basis for solving the problem.
- **Answer field or options:** There are two usual possibilities for the answer section, either a designated space or an area embedded in another component. The former includes a blank space where students are free to write whatever they want, or a list of available options to choose from. The latter usually consists of blank spaces embedded in the *Code* component, visually identifying where the students must fill in their answers.

While not a component in itself, the evaluation of the students' answers must be taken into account. Its complexity may vary depending on the type of the exercise, ranging from a direct verification of the correct option, to more sophisticated mechanisms, such as a set of tests performed on the code that the student has written. Although, some characteristics of the answers present a qualitative nature that prevents its automatic evaluation, such as code sophistication and legibility.

In summary, understanding the components of a programming exercise is crucial to design effective exercises. These components can vary depending on the type of exercise and its objectives.

3 Text Generation

Natural Language Generation (NLG) is a sub-field of Natural Language Processing (NLP) that involves building systems capable of producing coherent and useful text in multiple languages [24]. NLG systems are used to create chatbots, translate text, and generate

complex articles and stories [6]. This process involves taking input data, such as keywords or a set of facts, and transforming them into meaningful output text. Reiter and Dale [25] identified six tasks that are essential for NLG, still present in current models:

1. **Content Determination:** Deciding what information should be included in the text under construction;
2. **Discourse Planning or Text Structuring:** Defining in which order information will be presented and the structure of such presentation;
3. **Sentence Aggregation:** Grouping information into sentences;
4. **Lexicalization:** Deciding which words and phrases should be chosen to express the required information;
5. **Referring Expression Generation:** According to Reiter and Dale [25], it consists of "selecting the words and phrases to identify domain objects";
6. **Linguistic Realisation:** Combining all words and phrases by applying grammatical rules to produce a text which is syntactically and orthographically correct.

This type of system can be built using different architectures, being the most common the three-stage pipeline, also proposed by Reiter and Dale [24]. The pipeline includes text planning, sentence planning, and linguistic realisation. In text planning, content determination and discourse planning are combined to decide what information to include and how to structure it. In sentence planning, sentence aggregation, lexicalization, and referring expression generation are combined to select words and phrases and group them into sentences. Finally, in linguistic realisation, grammatical rules are applied to produce a syntactically and orthographically correct text.

3.1 Models

Natural Language Processing (NLP) is a sub-field of Artificial Intelligence (AI) that focuses on the interaction between computers and human language [13]. Machine Learning algorithms have been developed and trained using text data to perform various natural language tasks such as text prediction and generation [17]. This section will explore some of the most commonly used and successful NLP models, as well as other generation systems.

Recurrent Neural Networks (RNNs)

Neural Networks are a subset of Machine Learning that are inspired by the structure and function of the human brain. They are used to classify and cluster data, and consist of layers of artificial neurons that send data to each other. The output of these neurons is determined by weights and threshold values, and training data is used to improve their accuracy over time [11].

One type of neural network architecture that is well-suited for processing sequential data, such as text, is Recurrent Neural Networks (RNNs). RNNs have feedback connections that allow them to incorporate information from previous time steps or observations into their current output. This makes them effective at modelling dependencies between elements in a sequence to predict the next one [12].

- **Long Short-Term Memory (LSTM) networks:** A type of RNN that is particularly well-suited for modelling long-term dependencies in sequential data, such as natural language text. LSTM networks are able to "remember" important information from the past for extended periods of time, by using state cells, making them ideal for generating coherent and realistic text [9]. For example, in the paper [15], the authors trained a LSTM on a dataset of Shakespearean plays was able to generate text that was difficult to distinguish from human-written equivalents.

- **Seq2seq models:** Have been widely used for NLP tasks that involve the generation of an output sequence based on an input sequence. Seq2seq models consist of an encoder and a decoder (each one a model with neural network architecture), which process the input and output sequences. One of their key advantages is the ability to handle variable-length input and output sequences [8]. Seq2seq models can be implemented using various types of neural network architectures, although RNNs are the most common. For example, in the article [29], the authors use a Seq2seq model with an RNN architecture to generate translations from one language to another.

In conclusion, Recurrent Neural Networks, such as Long Short-Term Memory networks and Seq2seq models are among the most commonly used and successful NLP models at the moment. They are able to effectively process sequential data, such as natural language text, and generate coherent and realistic output.

Transformers

Transformers consist in a type of neural network architecture that was first described in an article by Vaswani et al. [30], which has been effective in NLP tasks. The capacity of transformers to model long-range dependencies between words in a sentence or text is one of its most important features. This is done by utilising self-attention mechanisms, which give the network the ability to weigh the importance of the different words or tokens in the input based on how they relate to other words or tokens. Transformers are thus well suited for tasks that require a thorough comprehension of the context and semantics of a particular text. Some of the most popular and successful transformer models are:

- **Generative Pre-training Transformer (GPT):** Transformer-based language model developed by OpenAI [19], that has been widely used for natural language processing tasks such as text generation. It uses unsupervised learning to pre-train a large neural network on a massive dataset of text, discovering "hidden patterns or data groupings without the need for human intervention" [14], and then fine-tunes the network on a smaller dataset of labeled text for a specific task. GPT-3 is the third generation of GPT models, which was introduced in 2020 [5] and. This model are significantly larger and more powerful than its predecessor models, since it is trained on 175 billion parameters (over ten times the size of GPT-2) and requires very little fine-tuning on a task-specific dataset to achieve good performance on a wide range of NLP tasks. OpenAI also launched GPT-3.5, an upgrade for this model, and a new version, GPT-4 [20], which is even more powerful (100 trillion parameters). ChatGPT is a well-known application of GPT that is specifically designed for conversational language understanding and generation, trained on conversational data [18].
  It is worth noting that, as GPT-3 is an unsupervised model, it is vulnerable to biases present in its training dataset, and may generate texts that reflect them.
- **BERT:** Transformer-based language model that was developed by Google in 2018 [7] and, in similar form to GPT, it also uses unsupervised learning to train a neural network on a large dataset of text. The main difference between GPT and BERT is that BERT is a masked language model, trained to predict missing words in a sentence rather than the next word, as GPT does. This makes the text generated by BERT less likely to diverge from the original context of the input. In one study, researchers used BERT to generate news articles that were evaluated by human annotators, who found them comparable in quality to those written by humans [34].

Source:

```
prompt = "Create a programming exercise in Python about string
          manipulation and write the answer."
exercise = openai.Completion.create(engine="text-davinci-003",
           prompt=prompt, temperature=0.7, max_tokens=500, top_p=1,
           frequency_penalty=0, presence_penalty=0)
```

Output:

```
Write a program to print out the last three characters of a string.

#Answer:
string = input("Enter a string: ")
length = len(string)
last_three_chars = string[length - 3:]
print(last_three_chars)
```

Figure 1 Example of a programming exercise generation using GPT-3.5.

4 Case Studies and Results

This section introduces some case studies to explore different implementations of up-to-date text generation models. The outcomes are then analysed in order to assess some of the most commonly used mechanisms and how they can be improved for the generation of programming exercises.

4.1 OpenAI API

OpenAI has made available APIs to GPT-3 and GPT-3.5 models. Using an access token it is possible to make requests by sending a prompt with a certain maximum number of tokens, which will be answered with the generated text. Each of these models has some variations: some are applied to simpler and faster tasks while others are used for more complex text generation. Despite its technical capacity and current popularity, there is an important tangential disadvantage: the API is not completely free. After a gracing period of three months, there are monetary usage fees for the different OpenAI models, which despite appearing minimal (GPT-3.5 is 0.002$ per 1K tokens) may become a problem in an academic context with a lot of tests. Beyond that, for actual implementation in harsher socioeconomic contexts, it might become a dividing criteria that opposes the already slow-paced democratisation of science and education, while concurrently enlarging the technological gap between world-wide communities.

In the context of generating programming exercises, this tool presents satisfactory results. By making requests with the type of the exercise, theme or more concrete characteristics, such as the mandatory use of a language structure or construct, it is possible to obtain complete exercises, including their corresponding answers. The only disadvantage that was realised about the quality and effectiveness of the results is the possible incoherence derived from the lack of context of the model, that is, it may produce correct text, but outside the scope of the problem in question. To improve this, OpenAI also provides the option of fine-tuning, in which the model is further trained with a shorter dataset to perform a specific task. By fine-tuning GPT, it would be possible to obtain faster and more adequate results, in the sense that the exercises and their parts would be in accordance with a pre-defined structure and better contextualised with the domain in question.

Figure 1 shows the generation of a programming exercise from a prompt, in which the latest GPT-3.5 model was used (*text-davinci-003*). The model was configured using parameters recommended in the OpenAI documentation. As can be seen, the result for a

Source:

```
prompt = """Exercise

Instructions:
Write a Python program to reverse a string.

Code:"""

model.generate(prompt=prompt, max_length=500, temperature=0.3, top_p=0.9)
```

Output:

```
Exercise

Instructions:
Write a Python program to reverse the binary representation of a given
integer.

Code:
def test(n):
  return int(bin(n)[::-1][:-2], 2)
```

Figure 2 Example of a programming exercise generation using a finetuned GPT-2 model.

relatively small input is quite satisfactory and there are a lot of possible generations for the same input. The potential is clearly enormous with regard to the type, structure, subject and difficulty of the exercises.

4.2 Finetuning GPT-2

Among GPT models, it is possible to work with older versions for free. There are even libraries such as *aitextgen* [33] that provide the possibility to finetune the GPT-2 model from a dataset. Using PyTorch and TensorFlow (machine learning libraries), it is possible to improve the performance of a specific task within text generation. However, this version is not as powerful as the most recent ones, since it was trained with 100 times fewer parameters, which makes the results not as adequate. Furthermore, it does not provide as much flexibility in order to perform different text generation tasks (question answering, summarization, text classification and so on) [27]. The fact that there is no remote access to use and train GPT-2 makes it free, but also requires all processes to be done locally, demanding a significantly greater use of computational resources. Figure 2 displays the output of an attempt that was made to finetune GPT-2 from a dataset consisting of a list of 300 Python basic exercises taken from the W3 Resource website [31].

As GPT-2 is not prepared to answer requests, but to complete text, the dataset was modified in order to define the structure of the exercise as *Problem Statement* and *Code*. When analysing the result obtained for the provided prompt, it can be seen that the model was not able to generate according to the predefined *Problem Statement* and, with that, generated another text with some similarities, maintaining the structure, but incoherently and out of the context.

The quality of the results is not the one that is expected and this is mainly due to the small size of the dataset. The model to be trained with this dataset took about 1 hour, which implies that, with a dataset of adequate size, it would require spending a huge amount of time.

Source:

```
keystotext_model.predict(["delete", "list", "odd numbers"])
```

Output:

```
Write a function to delete odd numbers from a list.
```

Figure 3 Example of a *Problem Statement* component generation using the trained *keys-to-text* model.

4.3 Key-to-Text

The *keytotext* library [2] was used in the attempt to generate the *Problem Statement* component of an exercise, which implements a model that receives keywords as input and generates sentences. It is based on the Text-to-Text Transfer Transformer (T5) model, a transformer-based language model developed by Google's AI team that employs sequence-to-sequence (Seq2Seq) pre-training and uses a fine-tuning approach to perform a variety of NLP tasks [22]. The T5 model is one of the largest transformer-based language models to date, with 11 billion parameters.

The *keytotext* library simplified the process of fine-tuning the original model to work with computing programming exercises generation. Using the Mostly Basic Python Problems (MBPP) dataset from Google Research [1], which consists of 1000 programming problems designed to be solvable by entry level programmers, the model was fine-tuned to perform in a contextualised and efficient way. In order to train the model to generate exercise *Problem Statement* from keywords, a field with the three most relevant keywords that represent the text of each exercise was included in the dataset. The NLTK platform [3] was used to accomplish this, since it has numerous libraries and functionalities for working with natural language, such as RAKE that can extract and sort words from a given text by importance.

By applying this method of text generation to the context of the programming exercises, it was possible to obtain satisfactory results. It should be noted that, as the model was trained with a trio of keywords for each text, it will loose its efficiency if provided with a different quantity of keywords.

As illustrated in Figure 3, the results are quite satisfactory and coherent within the context of the dataset provided. Sometimes, the generated text does not correspond to what the user expects, but it manages to make sense according to the words entered. This provides a wider variety of exercises helping in the creative process.

4.4 CodeT5

CodeT5 [32] builds on the T5 architecture, but incorporates source code-specific knowledge to endow the model with better code understanding. It was trained on the CodeSearchNet dataset, which consists of millions of lines of code written in popular programming languages such as Python, Java, and JavaScript, as well as popular frameworks and libraries such as TensorFlow, React, and Django [10]. This enables the model to generate code that is not only syntactically but also functionally correct and efficient. It was tested using one of the checkpoints of this model (*CodeT5-large-ntp-py*) which was trained on millions of Python source code files from GitHub (GCPY - Python Github Code Dataset) [16].

The source shown in Figure 4 requested from the model a function in Python with substantial complexity, including string manipulation and the use of a repetition structure (loop). The model produced code that performs what was requested in an optimised way. However, being a somewhat computationally demanding mechanism, it required significant time to load and generate the corresponding code.

Source:

```
txt = "Function that counts the frequency of each character in a string."
input_ids = tokenizer(txt, return_tensors="pt").input_ids
generated_ids = model.generate(input_ids, max_length=128)
print(tokenizer.decode(generated_ids[0], skip_special_tokens=True))
```

Output:

```
def count_characters(string):
  char_freq = {}
  for char in string:
    char_freq[char] = char_freq.get(char, 0) + 1
  return char_freq
```

Figure 4 Example of code generation using *CodeT5*.

Despite using a constant string as input to the model in Figure 4 (variable `txt`, used for exemplification purposes only), the real test was conducted by linking the output of the *keytotext* model to the input of *CodeT5*, creating a pipeline that generated not only the *Problem Statement*, but also the *Code* for that specific exercise directly.

4.5 Summary

After the review and implementation of NLP/ML methods that can be integrated into a tool for automatic generation of programming exercises, it was possible to analyse various approaches, comparing advantages and disadvantages between them.

As drawbacks, the price may be seen as a barrier for public and free implementations, while computational resources required by some implementations, such as GPT-3, also incur in initial costs to infrastructure construction and maintenance. These factors are undoubtedly important in order to choose which approach to adopt for the implementation of an exercise generator. Another aspect that should obviously be discussed is the quality of the generated exercises, in terms of lexical, syntactic and semantic correctness and overall coherence to the input parameters.

The initial approach to generate a complete exercise (problem statement, code snippet and answer options), using theme and type as inputs, required an advanced model (GPT-3). This raised three problems:

- Implementing such a system on an ordinary machine is quite complicated, as it requires resources that common computers do not have, making this process very slow or even unattainable;
- The need to use costly APIs would present a financial strain that could be unsustainable in the medium or long term;
- Creating a system dependent on an Internet connection could lead to communication failures and reliability issues.

The approach of generating the components of an exercise separately has proven to be more efficient, as it lies upon the use of simpler models. The first component was generated by the *keytotext* library in tandem with the MBPP dataset, resulting in a fully constructed and comprehensible problem statement. In turn, this problem statement was then fed to the CodeT5 model in order to generate the second component of the exercise, namely the source code snippet. Besides other models with higher complexity were tested, this approach – used to generate around 100 exercises – demonstrated equally capable results to GPT-3, with approximately 80% rate of success in generating comprehensible and coherent exercises.

5 Conclusion

The investigation and exploration reported in this paper contributed to establish a panorama on how to automatically generate computer programming exercises, using acceptable NLP and ML mechanisms to construct both the *Problem Statement* and its accompanying *Code*. A prototype for the system that implements these mechanisms has been developed and tested, in order to pave the way to the final semi-automatic exercise generator we intend to build.

Future works in this project include a compiler to a Domain Specific Language (DSL) that has been designed to facilitate the exercises generation process, templates to automatically create different versions of the *Problem Statement* and *Code*, and user interfaces to create new templates and retrieve lists of exercises. In that way teachers and students will benefit from our system, getting different and challenging problems to practice and test their programming abilities.

References

1 Jacob Austin, Augustus Odena, Maxwell Nye, et al. Program synthesis with large language models. *arXiv preprint*, 2021. arXiv:2108.07732.

2 Gagan Bhatia. keytotext. URL: https://github.com/gagan3012/keytotext.

3 Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit*. O'Reilly Media, Inc., 2009.

4 Matt Bower. A taxonomy of task types in computing. *SIGCSE Bull.*, 40(3):281–285, June 2008. doi:10.1145/1597849.1384346.

5 Tom B. Brown, Benjamin Mann, Nick Ryder, et al. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020. arXiv:2005.14165.

6 Asli Celikyilmaz, Elizabeth Clark, and Jianfeng Gao. Evaluation of text generation: A survey, 2021. arXiv:2006.14799.

7 Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019. doi:10.18653/v1/n19-1423.

8 Pranay Dugar. Attention – Seq2Seq Models. https://towardsdatascience.com/day-1-2-attention-seq2seq-models-65df3f49e263, 2019.

9 Alex Graves. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850, 2013. arXiv:1308.0850.

10 Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436, 2019. arXiv:1909.09436.

11 IBM. What are Neural Networks? https://www.ibm.com/topics/neural-networks, 2023.

12 IBM. What are Recurrent Neural Networks? https://www.ibm.com/topics/recurrent-neural-networks, 2023.

13 IBM. What is Natural Language Processing? https://www.ibm.com/topics/natural-languageprocessing, 2023.

14 IBM. What is Unsupervised Learning? https://www.ibm.com/topics/unsupervised-learning, 2023.

15 Andrej Karpathy. The unreasonable effectiveness of recurrentneural networks, 2015. URL: http://karpathy.github.io/2015/05/21/rnneffectiveness/.

16 Hung Le, Yue Wang, Akhilesh Gotmare, Silvio Savarese, and Steven Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning, July 2022. doi:10.48550/arXiv.2207.01780.

17 Archna Oberoi. What are Language Models in NLP? https://insights.daffodilsw.com/blog/what-are-language-models-in-nlp, 2020.
18 OpenAI. ChatGPT. https://openai.com/blog/chatgpt.
19 OpenAI. OpenAI. https://www.openai.com/product.
20 OpenAI. Gpt-4 technical report, 2023. arXiv:2303.08774.
21 Mário Pinto and Teresa Terroso. Learning Computer Programming: A Gamified Approach. In Alberto Simões and João Carlos Silva, editors, *Third International Computer Programming Education Conference (ICPEC 2022)*, volume 102 of *Open Access Series in Informatics (OASIcs)*, pages 11:1–11:8, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/opus/volltexte/2022/16615.
22 Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(1), January 2020.
23 Noa Ragonis. Type of questions – The case of computer science. *Olympiads in Informatics*, 6:115–132, January 2012.
24 Ehud Reiter and Robert Dale. *Building Natural Language Generation Systems*. Cambridge University Press, 2000.
25 Ehud Reiter and Robert Dale. Building applied natural language generation systems. *Natural Language Engineering*, 3, March 2002.
26 Alexander Ruf, Marc Berges, and Peter Hubwieser. Classification of programming tasks according to required skills and knowledge representation. In *Informatics in Schools. Curricula, Competences, and Competitions - 8th International Conference on Informatics in Schools: Situation, Evolution, and Perspectives, ISSEP 2015, Ljubljana, Slovenia, September 28 - October 1, 2015, Proceedings*, volume 9378, September 2015. doi:10.1007/978-3-319-25396-1_6.
27 Gianetan Sekhon. Gpt-2 vs gpt-3. https://medium.com/@gianetan/gpt-2-vs-gpt-3-e915ac43e981, 2023.
28 Alberto Simões and Ricardo Queirós. On the Nature of Programming Exercises. In Ricardo Queirós, Filipe Portela, Mário Pinto, and Alberto Simões, editors, *First International Computer Programming Education Conference (ICPEC 2020)*, volume 81 of *OpenAccess Series in Informatics (OASIcs)*, pages 24:1–24:9, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.ICPEC.2020.24.
29 Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems – Volume 2*, NIPS'14, pages 3104–3112, Cambridge, MA, USA, 2014. MIT Press.
30 Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pages 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.
31 w3resource. Python exercises, practice, solution. https://www.w3resource.com/python-exercises/, 2023.
32 Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. URL: https://aclanthology.org/2021.emnlp-main.685.
33 Max Woolf. aitextgen. https://docs.aitextgen.io/, 2021.
34 Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with bert, 2020.