Programming to Learn Language Models

Luca Chiodini[0000−0002−2712−9248] and
Matthias Hauswirth[0000−0001−5527−5931]

Software Institute, Università della Svizzera italiana, Lugano, Switzerland
{luca.chiodini,matthias.hauswirth}@usi.ch

Abstract. *Programming to learn* is a pedagogical approach in which students implement a program to learn a certain topic. Programming can thus become a powerful medium for understanding topics in many disciplines, including informatics itself.
This short conceptual paper argues that *programming to learn* can be used to design a sequence of programming activities that use only simple programming language constructs and are suitable for secondary school students to learn about fundamental ideas of language models. Widespread student use of products based on language models calls for activities that demystify their behavior. Without having to deal with the complexity of modern model architectures based on neural networks, students can still experience ideas such as generative text models, training and inference, model parameters, and prompts, intertwined with concrete discussions on issues such as biases, ethical and copyright concerns, and the perception of intelligence.
In this paper, we show that the created activities realize the *programming to learn* approach through their design. We analyze each activity by breaking it down into a fine-grained sequence of programming tasks, and then map each task to specific learning objectives about language models.

Keywords: Programming · Language Models · Secondary Education.

1 Introduction

Programming requires writing an unambiguous, executable description of how something works, which promotes deep understanding. Accordingly, university-level informatics courses commonly assign students programming projects, such as implementing a toy compiler to understand how compilers work. This pedagogical approach is known as *programming to learn* [6].

The approach is commonly used in advanced courses for informatics students who have already mastered programming. However, now that programming is increasingly becoming a core component of secondary education, it should also be feasible to adopt the approach at a level suitable for secondary schools.

This short conceptual paper argues that a sequence of programming activities can be designed to follow this pedagogical approach, applying it to one of the most popular topics currently in artificial intelligence: language models. After reviewing related work on the pedagogical approach and how language models

are taught in secondary education (Section 2), we describe the design principles of our activities (Section 3) and show that they follow the *programming to learn* approach by mapping each programming task to learning objectives (Section 4).

2 Related Work

2.1 Programming to Learn

In their influential introductory programming textbook, *Structure and Interpretation of Computer Programs*, Abelson and Sussman describe the importance of programming languages: "A computer language is not just a way of getting a computer to perform operations but rather it is a novel formal medium for expressing ideas about methodology" [1]. Programming can thus become a medium to *learn* other topics. Sussman cites classical mechanics as an example [14], but the same principle applies equally to simpler topics, such as learning by programming educational robots. Papert's turtle geometry [13], for example, is widely used in schools and is posited to enhance the understanding of certain aspects of geometry. Requiring students to *program* the drawing of a triangle helps them understand internal angles and their sum.

This pedagogical approach is referred to as *programming to learn* [10,11]. When students know a programming language, they can write programs to learn certain topics in various subjects (e.g., music, science, math) [17]. Informatics itself can also become the target. Students may, for example, learn about version control systems by programming a simplified version of Git [6], or learn about logic and theory of computation by programming automata [4].

Applying the *programming to learn* approach to informatics itself can bring double benefits: at the same time, students are both strengthening their programming skills and learning about a new topic in informatics.

2.2 Learning Objectives about Language Models

The field of artificial intelligence has been evolving rapidly, challenging educators to find a good set of learning objectives in a quickly changing landscape. The rise of language models, which went from being one of the many topics in artificial intelligence to dominating the discourse, has made attempts to define learning objectives even more challenging.

In 2019, Touretzky et al. [16] proposed "five big ideas" in artificial intelligence that every school student should experience. First, computers perceive the world using sensors. Second, agents maintain models of the world and use them for reasoning. Third, computers can learn from data. Fourth, making agents interact comfortably with humans is a substantial challenge. Fifth, artificial intelligence applications can impact society in both positive and negative ways.

The second and the third ideas are also at the core of language models, which learn from a textual corpus and build an internal representation that is then used to "reason" and generate text. The importance of the fifth idea

is more evident than ever, given the remarkable capabilities of large language models. And finally, the first and the fourth ideas become relevant when textual language models grow into multi-modal models (e.g., with vision capabilities) or get integrated with external tools for human-friendly input and output.

These five big ideas have since been expanded at `ai4k12.org` to detail specific "learning objectives" (what students should be able to do) and "enduring understanding" (what students should know) for various school grades.

Some authors use the umbrella term "AI literacy" [9] to refer to similar ideas. Specifically for language models, competencies include understanding the steps of a machine learning pipeline, the ability to learn from data and maintain representations, and the role of humans in developing artificial intelligence.

2.3 Language Models in Secondary Education

Existing approaches to teach language models in secondary education do not typically ask students to build their own language model from scratch. Instead, learners are asked to engage with an existing model. Students can then, for example, choose their own training corpus or change a hyperparameter of the model to then observe how text is generated statistically, one token at a time.

Online playgrounds such as SoekiaGPT [7] and Markov Chain Demo [15] can be used for this purpose. Having students interact with models that allow them to take a peek under the hood is certainly a valuable didactic approach that actively involves them. Unfortunately, although learners can tweak certain aspects of the model, they are still not required to develop a complete understanding of *how* a model works inside out, because the model has already been programmed by the authors of the activity.

Bootstrap's workbooks on artificial intelligence [2] include a section on language models, in which students are asked to complete pen-and-paper activities to understand the probabilities and the statistical predictions. This approach engages students with creating a tiny model from scratch, but the manual work cannot scale beyond extremely small examples, which can still be perceived as qualitatively very different from the output of existing large language models.

3 Designing the Programming Activities

To adhere to the *programming to learn* approach, we established two design principles that all our programming activities should follow:

- *Programming should be done "from scratch".* Even though language models could also be programmed by using one of the many powerful machine learning libraries and only writing calls to the functions offered by the library, doing so would defeat the purpose of the programming to learn approach, whose goal is to help students understand the inner workings of a certain topic. Therefore, we eschewed any third-party library that offered prepackaged solutions to one or more parts of building a language model.

– *Programming should be done using simple language constructs.* Professional programming languages such as Python include both an extensive standard library and many sophisticated constructs. For instance, Python's standard library includes a specialized `Counter` class to count object occurrences, and offers operator overloading to define the semantics of operators on user-defined types. Although experienced programmers find it convenient to use such features, they can obscure the underlying logic and also deprive beginners of a learning opportunity (e.g., of programming a "counter" using a simple dictionary). Throughout the activities, we consistently favored simple programs—with logic made explicit—over short programs.

Karpathy's notebooks [8] explain how to build a neural network from scratch, following our first design principle. His programming activities, however, are primarily aimed at field experts, with the intent of aligning with professional machine learning libraries (e.g., PyTorch). As such, the activities make use of relatively sophisticated techniques, such as operator overloading. Moreover, they assume a certain degree of mathematical proficiency (mainly, derivatives) that is usually only reached towards the end of secondary education.

Norvig [12] uses a mathematically simpler method that avoids neural networks and instead uses n-grams. Remarkably coherent output is produced by using characters (as opposed to using entire words) as the elementary unit. Norvig builds on top of the work of Goldberg, who observed how character-level language models can be unreasonably effective, despite their simplicity [5].

Norvig's programs [12] also follow our first design principle, with the added benefit of keeping the mathematical prerequisites low, but they strive for compact Python code, fully exploiting the library and the features of the language.

Following the steps of Norvig and Goldberg, we designed programming activities that build character-level language models. We created six activities using Jupyter notebooks that enable students to experience fundamental ideas about generative language models. We implemented the activities in Python given its current widespread use in secondary education, although they do not depend on a specific programming language.

The programming activities assume that students are familiar with basic programming concepts, including defining and using functions, conditionals and repetition (e.g., using loops). They also require lists and dictionaries as two essential data structures. By design, they do not require working with object-oriented language features (e.g., students never need to call methods nor instantiate classes). These requirements are typically met after a programming course in the first years of upper-secondary education, although the programming content covered varies significantly by country.

The complete version of the activities, which includes all explanations, exact instructions, and starter code, is permanently available in the online appendix at `10.5281/zenodo.20039104`. An interactive version, which students can use to program directly in their browser, is currently available as the "Build Your Own Language Model" curriculum on the PyTamaro Web platform [3, Chapter 7] at `pytamaro.si.usi.ch`.

4 Mapping Programming Tasks to Learning Objectives

To show that our programming activities indeed adhere to the *programming to learn* approach, we break down each activity into fine-grained programming tasks. Each task corresponds to a single code cell within that activity's Jupyter notebook. We then map each task to a set of detailed learning objectives. This fulfills the promise of *programming* (i.e., completing the task) *to learn* (in this case, fundamental ideas on language models).

The sequence of programming tasks within an activity is denoted by numbers, with each task briefly summarized (the full version is linked in Section 3). Below a task, the set of detailed learning objectives directly connected to that task is denoted with letters.

4.1 Programming Activity 1: Trivial Language Model

1. Define a string `alphabet` consisting of lowercase letters and a space.
   (a) See characters as one of the possible elementary units to generate text.
2. Implement a function `generate_text` that produces a fixed-length text, picking each character randomly.
   (a) See a piece of text as the concatenation of characters.
   (b) Understand what a generative language model is.
   (c) Reflect on the probability with which a specific character is generated.
   (d) Understand the need to gather a textual corpus.

4.2 Programming Activity 2: Single Character

1. Inspect a string that contains the text of a Wikipedia page, to see how long it is and print a slice of it.
   (a) Connect the ideas of an alphabet and a text.
2. Implement a function `count` to count the occurrences of the alphabet characters in a given text. Use the function on the provided text.
   (a) Recognize simple mathematical features of a text, such as the number of occurrences of a character.
3. Implement a function `probability` to turn counts into probabilities. Use it with the counts computed earlier.
   (a) Connect empirical probabilities to probabilities usable for generation.
4. Compute the probabilities of characters with a different Wikipedia page using the `probability` and `count` functions.
   (a) Understand that the same functions can be used with different corpora.
5. Use the `heatmap_1d` function to visualize the two probability distributions.
   (a) Experience how different texts lead to different probabilities.
   (b) Recognize what may cause a bias that leads to higher probabilities for certain characters.
6. Combine the two texts and explore the probabilities of the resulting text.
   (a) See how the simple combination of texts leads to a bigger corpus.

   (b) Experience how a diverse corpus mitigates the biases arising from using only a small specific source.
   (c) Understand that there is no set of "right probabilities".
7. Implement a `generate_text` function that produces a fixed-length text, picking each character according to its probability. Use the function to generate text using the probabilities computed on the two texts combined.
   (a) Understand the meaning of a parameter of a language model.
   (b) Experience how characters generated with different probabilities lead to a realistic length of the words.
   (c) Critically reflect on the legitimacy of using a text with a certain license.

4.3 Programming Activity 3: Pairs of Characters

1. Inspect a slice of a string containing fragments of Shakespeare plays.
   (a) Experience the use of a realistic, public-knowledge corpus.
2. Implement a `train` function that, given a text, computes the probability of each pair of characters.
   (a) See training as a clear, separate phase in building a model.
   (b) Understand that the training of a language model uses a text to determine values for the model parameters.
3. Use the `heatmap_2d` function to visualize the values of the parameters.
   (a) Understand the increase of parameters when considering character pairs.
   (b) Realize that many pairs of characters never occur.
4. Implement a `generate_text` function that produces a fixed-length text that starts with a random character and continues by generating the next character according to the probabilities of the pairs of characters. Use the function to generate text using the parameters computed in the training phase.
   (a) Understand that the inference phase requires the parameters computed during the training phase.
   (b) Experience how words become more realistic when the probability depends on the previous character.
   (c) Understand that the generation is non-deterministic: multiple calls to the generation function produce different results.

4.4 Programming Activity 4: N-Grams

1. Implement a `train` function that computes the occurrences of sequences of N characters (after a prefix of $N-1$ characters, how many times each next character occurs). Train a model on the Shakespeare text with $N = 3$.
   (a) Reason about the explosion of the number of theoretical parameters for sequences of length N.
   (b) Understand that a huge corpus would be needed to estimate all the probabilities for long sequences.
   (c) Expand the notion of an alphabet to include non-alphabetic characters.

2. Implement a `generate_text` function that produces a fixed-length text that starts with a randomly sampled prefix of $N-1$ characters and continues by generating the next character with the probability determined each time by the last $N-1$ characters. Use the function to generate text using the parameters computed in the training phase with $N=3$.
   (a) See how the generated text becomes realistic when considering triplets of characters and generating all characters, such as punctuation.
   (b) Understand that the generation mechanism is essentially the same as in the very first activity, the difference lies in the number of characters considered to estimate the probabilities.
3. Train a model with increasing N ($N=5$, $N=10$).
   (a) Use the same function to train bigger models with more parameters.
4. Generate text using the parameters computed for a larger N.
   (a) Observe how the larger the N, the more realistic the text looks.
   (b) Understand why a larger N causes the model to "copy" more and more from the text used for training.
5. Train a model on a text in a different natural language (Divina Commedia).
   (a) Understand that a language model can be trained on any text, without changing the training function at all.
6. Use the newly generated model to produce text in Italian, Dante style.
   (a) Understand that the output is determined by the parameters' values.
   (b) Reason about whether the model "knows" Italian and English or not.

4.5 Programming Activity 5: Prompt

1. Print a slice of a dataset of names of users from Great Britain and Italy, sampled from a leaked, now-public dataset.
   (a) Be aware that data protected by companies can be leaked publicly.
   (b) Reflect critically on the legitimacy of using such a dataset.
2. Change the `generate_text` function so that instead of generating a text of a fixed length, it generates text until a newline character is produced. Train a model with $N=3$ on the dataset of names and use it to generate names.
   (a) Understand how a model can "automatically" decide when to stop.
   (b) Experience the generation of texts of different lengths, as chatbots do.
3. Change the `generate_text` function to start with a user-supplied prompt. Ask the user for a prompt of length $N-1=2$, and pass it to the function.
   (a) Understand that a prompt is simply the beginning of the text to generate.
   (b) Experience how a prompt significantly "steers" the output of a model.
   (c) Connect the prompt entered by the user with the input given to chatbots.
4. Train a "sequence" of models from $N=1$ to $N=8$.
   (a) Understand the need to fall back to a shorter prefix when we cannot estimate the probability of a complete one.
5. Extend the `generate_text` function to fall back to shorter prefixes until one is found. Implement a loop that keeps asking the user for a prompt and each time generates a name starting with that prompt.
   (a) See how a model can generate text from an arbitrary user prompt.
   (b) Experience how an Italian-sounding prompt will likely generate an Italian-sounding name, and a British-sounding prompt a British-sounding name.

4.6 Programming Activity 6: Tic-Tac-Toe

1. Print a slice of a dataset containing tic-tac-toe games. Each game is represented as a sequence of moves, such as `^35241`.
   (a) Understand how text does not have to contain only natural language.
   (b) Be able to reason about a simple textual encoding of a game.
2. Train a language model and use it multiple times to generate the next character (instead of a whole text) on the partial game `^5`.
   (a) Use a language model to produce non-natural language.
   (b) Relate the generation of text to the generation of a game move.
3. Use a `render_board` function to produce a graphic corresponding to the board state after a sequence of moves.
   (a) Understand that the textual output of a model can be post-processed by external tools to produce more human-friendly representations.
4. Implement a function `ask_number_between` to repeatedly ask the user for a number between 1 and 3, until they input a valid number. Implement a function `rc_to_move` that turns a row and a column into a move.
   (a) Understand that more human-friendly interfaces can be used to create the text fed as a textual input to a language model.
5. Implement a tic-tac-toe game loop: in every round, the user is first asked for a move, and then the language model generates the next move. After each move, the model is also used to predict the possible end of the game.
   (a) Experience how a simple model trained on the right data can exhibit "intelligent" behavior, reflecting critically on the meaning of intelligence.
   (b) Understand that context is the portion of text considered by the model to generate further text.
   (c) Experience how a reduction of the context length can significantly affect the performance of the model.
   (d) Be aware of the utmost importance of the training data in determining the performance of a language model.

5 Conclusion

This short paper presented a sequence of programming activities that adopt the *programming to learn* approach to language models, following two key design principles that make them suitable for secondary education. To show that the activities indeed follow the approach, we provided a fine-grained mapping of each programming task to a set of learning objectives related to language models.

On a more theoretical level, this paper serves as an example of how programming activities can be aligned to the learning objectives of a specific topic to demonstrate the successful application of the *programming to learn* approach.

Future work should rigorously evaluate the extent to which students achieve the proposed learning goals after working through the activities. Although the work presented here is purely conceptual, these activities have already been used to train informatics teachers in Switzerland. Several teachers deemed them appropriate for the secondary level and intend to implement them in their classrooms, providing a valuable platform for future empirical studies.

References

1. Abelson, H., Sussman, G.J.: Structure and Interpretation of Computer Programs. The MIT Press
2. Bootstrap Community: Bootstrap:AI, https://www.bootstrapworld.org/materials/courses/ai/
3. Chiodini, L.: Teaching Introductory Programming Using Graphics as a Domain, https://n2t.net/ark:/12658/srd1332702
4. Farrugia-Roberts, M., Jeffries, B., Søndergaard, H.: Programming to Learn: Logic and Computation from a Programming Perspective. In: Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1. pp. 311–317. https://doi.org/10.1145/3502718.3524814
5. Goldberg, Y.: The unreasonable effectiveness of Character-level Language Models, https://nbviewer.org/gist/yoavg/d76121dfde2618422139
6. Hauswirth, M.: Learn to program? Program to learn! **1**(119), https://bulletin.eatcs.org/index.php/beatcs/article/view/424/404
7. Hielscher, M.: Soekiagpt - ein didaktisches sprachmodell . https://doi.org/10.18420/ibis-01-01-04
8. Karpathy, A.: Neural Networks: Zero to Hero, https://github.com/karpathy/nn-zero-to-hero
9. Long, D., Magerko, B.: What is AI Literacy? Competencies and Design Considerations. In: Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems. pp. 1–16. CHI '20, Association for Computing Machinery. https://doi.org/10.1145/3313831.3376727
10. Mendelsohn, P., Green, T.R.G., Brna, P.: Programming languages in education: The search for an easy start. In: Psychology of Programming, pp. 175–200. Elsevier, https://www.sciencedirect.com/science/article/pii/B9780123507723500161
11. Miller, J.A.: Promoting computer literacy through programming Python, https://hdl.handle.net/2027.42/124137
12. Norvig, P.: The Effectiveness of Generative Language Models, https://github.com/norvig/pytudes/blob/main/ipynb/Goldberg.ipynb
13. Papert, S.: Mindstorms: Children, Computers, and Powerful Ideas. Basic Books, Inc.
14. Sussman, G.J.: The legacy of computer science. In: Computer Science: Reflections on the Field, Reflections from the Field, pp. 180–183. The National Academies Press
15. Touretzky, D.: Markov Chain Demo, https://ai4k12.org/markov-chain-demo/
16. Touretzky, D., Gardner-McCune, C., Martin, F., Seehorn, D.: Envisioning AI for K-12: What Should Every Child Know about AI? **33**(01), 9795–9799. https://doi.org/10.1609/aaai.v33i01.33019795
17. Wenger, A.: Learning to Program, Programming to Learn, https://continuations.com/learning-to-program,-programming-to-learn