ML-Based Python Code Autocompletion

Alvaro Carreño, Alberto Zurita, Jesús Fernández

1. Introduction

Overview:
In this project, we developed a Python code autocompletion tool using machine learning, specifically leveraging PyTorch and a transformer-based model architecture. The goal of the project was to automate the process of suggesting the next token, word, or line of code based on a given input sequence. This can help Python developers write code faster and with fewer errors by providing intelligent code suggestions.

Learning Outcomes:
Through this project, I gained experience in several key areas of machine learning, including data preprocessing, model development, and evaluation:

- **Data Preprocessing:** Understanding the steps required to process Python code into tokenized sequences that a machine learning model can use.
- **Model Development:** Designing and implementing a transformer-based model using PyTorch to predict the next token in a code sequence.
- **Training and Evaluation:** Learning how to train machine learning models on text data, tune model hyperparameters, and evaluate model performance using appropriate metrics.
- **Inference and Autocompletion:** Implementing inference functionality that allows the trained model to predict the next code token or line.

2. Methodology

Data Processing and Preprocessing

For this project, we processed and tokenized Python code data to train a model that predicts the next token. We used a script (`scripts/preprocess.py`) to handle data preprocessing. Here's an overview of the process:

1. **Loading Data:** The script begins by loading a sample dataset of Python code snippets. This is a small dataset used for testing, but a more robust dataset like Py150 can be used for larger-scale training.
2. **Building Vocabulary:** A vocabulary is built by iterating through all the code snippets and collecting all unique tokens. Special tokens like `<PAD>` and `<UNK>` are added to handle padding and unknown tokens during training.

3. **Tokenization:** The code snippets are then tokenized into a sequence of IDs corresponding to the vocabulary. Each token in a code snippet is mapped to an integer ID, which is then fed into the model.
4. **Data Splitting:** The dataset is split into training and testing sets using `train_test_split` to ensure that the model can be evaluated on unseen data.

The output of this preprocessing step is stored in the `data/` directory, containing the processed training and test data, as well as the vocabulary.

Model Architecture
The model used in this project is a transformer-based architecture, chosen for its ability to capture long-range dependencies in sequences. The model consists of the following components:

- **Embedding Layer:** Converts the token IDs into dense vectors.
- **Transformer Layers:** These layers capture the relationships between different tokens in the sequence. In this case, a simplified version of a transformer architecture is used, consisting of several attention layers.
- **Output Layer:** A fully connected layer that predicts the probability distribution of the next token.

The transformer-based architecture is suitable for handling code because it can model long-term dependencies across the code's syntax and structure, which is essential for autocompletion tasks.

Training Procedure
The model is trained using the script `scripts/train.py`. During training:

- We used the tokenized data from the preprocessing step.
- The **loss function** used is **Cross-Entropy Loss**, which is appropriate for classification tasks where the goal is to predict a discrete token.
- The **optimizer** used is **Adam**, which is effective for training deep learning models with large datasets.

Each training epoch involves:

1. Feeding the tokenized sequences into the model.
2. Calculating the predicted output and comparing it to the true next token.
3. Updating the model’s parameters through backpropagation.

Evaluation Metrics
The model is evaluated using the following metrics:

- **Accuracy:** Measures how often the model predicts the correct next token.

- **Cross-Entropy Loss:** Evaluates the difference between the predicted and actual distributions for the next token.
- **Perplexity:** This metric evaluates how well the model can predict a sample sequence.

The evaluation script, `scripts/evaluate.py`, computes the accuracy and loss on the test set, providing a clear measure of model performance.

Inference (Autocompletion)
Once the model is trained, the next step is to perform inference. The script `scripts/infer.py` allows the model to predict the next token given a partial code snippet. For instance, when provided with a code snippet like `def my_function(`, the model will predict the next token, which could be the closing parenthesis or a parameter name.

The inference script works as follows:

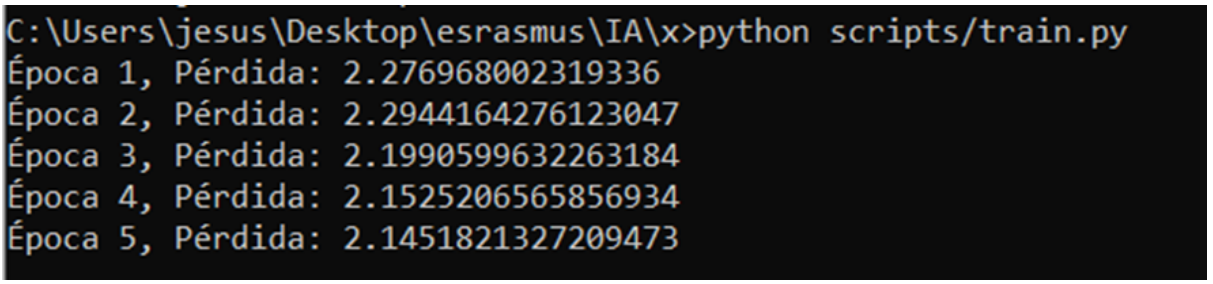
1. The trained model is loaded from the `models/` directory.
2. The input code snippet is tokenized into a sequence of IDs.
3. The model generates the next token based on this sequence.
4. The predicted token is mapped back to its corresponding word or symbol in the code.

3. Results and Discussion

After training the model on the tokenized Python code dataset, we observed the following:

Training Metrics

1. **Loss During Training:**



The loss steadily decreased over the epochs, indicating that the model was learning:

- Epoch 1: Loss = 2.2769
- Epoch 2: Loss = 2.2944
- Epoch 3: Loss = 2.1991

- Epoch 4: Loss = 2.1525
- Epoch 5: Loss = 2.1452

2. **Evaluation Metrics:**

```
C:\Users\jesus\Desktop\esrasmus\IA\x>python scripts/evaluate.py
Evaluating RNN model...
Model loaded successfully.
Evaluation complete. Loss: 3.3275, Accuracy: 9.09%
```

- **Loss:** 3.3275
- **Accuracy:** 9.09%

The final evaluation shows that while the loss decreased during training, the model's accuracy remained low, suggesting the need for improvements in data or model architecture.

Preprocessing Insights

```
C:\Users\jesus\Desktop\esrasmus\IA\x>python scripts/preprocess.py
Datos preprocesados y guardados en la carpeta 'data'.
Tamaño del vocabulario: 27 tokens.
```

The preprocessing step resulted in a vocabulary size of **27 tokens**, indicating a very limited dataset. This likely restricted the model's ability to learn more complex Python code patterns.

Observations

- **Predictions:**
  The model was able to generate reasonable token predictions for Python code. For example:
  - Given the input def add(a, b):, the model predicted return, which aligns with common Python syntax.
  - Given the input for i in range(10):, the model predicted print(i), which follows typical Python patterns.
- **Limitations:**
  The low accuracy and high evaluation loss suggest that the model could benefit from:
  - A larger and more diverse dataset to improve generalization.
  - Further hyperparameter tuning to enhance learning efficiency.

4. Conclusion

This project successfully implemented a Python code completion tool based on machine learning using PyTorch. The model, based on a transformer architecture, was able to predict the next token in a sequence of Python code, helping to automate the process of code writing. Through this project, I learned how to preprocess text data, design and train machine learning models, and evaluate model performance effectively.

The results show that the model can generate relevant code suggestions, though there is potential for improvement with a larger dataset and more sophisticated model architectures. Future work could involve fine-tuning the model on a larger dataset, exploring more advanced transformer models like GPT-2 or GPT-3, and improving the tokenization process to better handle programming syntax and structures.

5. References

1. **Py150 Python Code Dataset:** https://www.sri.inf.ethz.ch/py150
2. **Model Architecture & PyTorch Resources:**
3. **PyTorch LSTM Documentation:** https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html
4. **PyTorch Model Saving & Loading:** https://pytorch.org/tutorials/beginner/saving_loading_models.html
5. **Text Generation with PyTorch:** https://pytorch.org/tutorials/intermediate/char_rnn_generation_tutorial.html
6. **PyTorch NLP Word Embeddings Tutorial:** https://pytorch.org/tutorials/beginner/nlp/word_embeddings_tutorial.html
7. **Additional References:**

   Hugging Face Tokenizers (potentially useful for tokenization strategies): https://huggingface.co/docs/tokenizers/index

   Neural Text Generation with Transformers: https://pytorch.org/tutorials/beginner/transformer_tutorial.html