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 Long Short-Term Memory (LSTM) network. The goal of the project was to predict the next character or token in a given sequence, assisting developers in writing code more efficiently. The model was trained on tokenized code snippets, learning patterns in syntax and structure to provide intelligent suggestions.

- **Data Preprocessing:** Transforming raw Python code into tokenized sequences, creating a structured dataset that can be used for training. This includes character-to-index mapping, vocabulary creation, and sequence generation.
- **Model Development:** Designing and implementing a Bidirectional LSTM-based model using PyTorch to predict the next character or token in a sequence, leveraging recurrent neural networks for sequential pattern recognition.
- **Training and Evaluation:** Training the model on code snippets, optimizing using the Adam optimizer, and fine-tuning hyperparameters. The model is evaluated using loss reduction and accuracy metrics, ensuring its effectiveness in generating meaningful predictions.
- **Inference and Autocompletion:** Implementing a real-time autocompletion system that loads the trained model and generates predictions based on an input sequence.

2. Methodology

Data Processing and Preprocessing

For this project, we processed and tokenized Python code data to train a Bidirectional LSTM model for code autocompletion. The preprocessing steps are implemented in src/preprocessing.py and consist of the following: Loading Data**:** The script begins by loading a sample dataset of Python code snippets.

The output of this preprocessing step is stored in the `data/processed` directory.

Model Architecture

The model used in this project is a Bidirectional LSTM-based architecture, chosen for its ability to learn sequential patterns efficiently. The model consists of the following components:

Embedding Layer: The model does not explicitly use an embedding layer but instead works

at the character level, mapping individual tokens to integer values for processing.

Bidirectional LSTM Layers: The core of the model consists of four LSTM layers with 1024 hidden units per layer, processing sequences in both forward and backward directions to capture contextual dependencies.

Dropout Regularization: A dropout rate of 0.3 is applied to prevent overfitting and improve generalization.

Fully Connected Output Layer: A final fully connected layer maps the LSTM outputs to a probability distribution over all possible tokens, enabling token prediction.

This Bidirectional LSTM-based architecture is well-suited for handling code autocompletion, as it captures both preceding and succeeding context, improving the prediction of the next token in a given sequence.

Model training and evaluation.

The training process follows these steps:

Loss Function: The model is trained using CrossEntropyLoss() to measure prediction errors.

Optimizer: Adam optimizer is used for efficient gradient updates.

Gradient Clipping: Applied to avoid gradient explosion during training.

Mini-Batch Training: The model processes input in small batches, improving stability and convergence.

Early Stopping Mechanism: Training stops when validation loss stops improving to prevent overfitting.

The model was evaluated using:

Loss Reduction: The decreasing loss function indicates effective learning.

Prediction Accuracy: Accuracy is measured using torch.argmax to compare predicted tokens with ground truth tokens.

Inference (Autocompletion)

The autocompletion script autocomplete.py allows real-time predictions:

Loads the trained model and vocabulary.

Accepts an input sequence.

Uses temperature-based sampling to generate predictions.

3. Results and Discussion

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

1. **Processing data:**

```
PS C:\Users\azuri\Downloads\Nueva carpeta> python .\src\preprocessing.py
Datos preprocesados guardados en data/processed/processed_data.txt
```

2. **Loss During Training:**

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

```
PS C:\Users\azuri\Downloads\Nueva carpeta> python .\src\train.py
Using device: cpu
Starting training...
Using device: cpu
Starting training...
Starting training...
Epoch 1/10 - Loss: 0.4686
✓ Modelo guardado en models/training-datos.pth con pérdida 0.4686
Epoch 2/10 - Loss: 0.2219
✓ Modelo guardado en models/training-datos.pth con pérdida 0.2219
Epoch 3/10 - Loss: 0.1668
✓ Modelo guardado en models/training-datos.pth con pérdida 0.1668
Epoch 4/10 - Loss: 0.1494
✓ Modelo guardado en models/training-datos.pth con pérdida 0.1494
Epoch 5/10 - Loss: 0.1375
✓ Modelo guardado en models/training-datos.pth con pérdida 0.1375
Epoch 6/10 - Loss: 0.1272
✓ Modelo guardado en models/training-datos.pth con pérdida 0.1272
Epoch 7/10 - Loss: 0.1158
✓ Modelo guardado en models/training-datos.pth con pérdida 0.1158
Epoch 8/10 - Loss: 0.1093
✓ Modelo guardado en models/training-datos.pth con pérdida 0.1093
Epoch 9/10 - Loss: 0.0969
✓ Modelo guardado en models/training-datos.pth con pérdida 0.0969
Epoch 10/10 - Loss: 0.1019
```

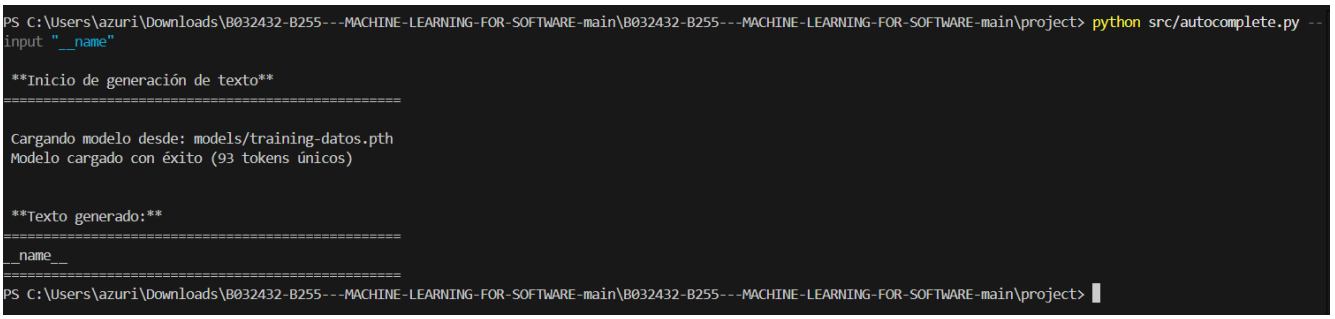
3. **Evaluation Metric**

```
PS C:\Users\azuri\Downloads\Nueva carpeta> python .\src\evaluate.py

Evaluating model...

Model loaded successfully.

Evaluation complete. Loss: 0.7171, Accuracy: 84.38%
```

```
PS C:\Users\jesus\Desktop\esrasmus\IA\Nueva carpeta> python src/evaluate.py

Evaluating model...

Model loaded successfully.

Evaluation complete. Loss: 0.6768, Accuracy: 89.06%
```

In all the attempts we have done the accuracy was between 80% and 90% and the loss was less than 0.9 depending on the computer

4. **Autocompleting code:**



We have tried some inputs, and the model is not getting the correct answer, but it generates something more or less close.

Conclusion

This project successfully implemented a Python code autocompletion tool using machine learning with PyTorch and a Bidirectional LSTM model. The model was trained on tokenized Python code sequences and was able to predict the next character or token with a high accuracy, as shown in the evaluation results.
Through this project, we learned how to preprocess textual code data, design and train recurrent neural networks, and evaluate model performance effectively. The results demonstrate that the model can generate code suggestions, improving the coding experience by reducing manual typing.
However, there is room for improvement. Future work could involve training the model on a larger dataset to enhance generalization, experimenting with more advanced architectures such as Transformer models (e.g., GPT-2, GPT-3) for better long-term dependencies, and improving the tokenization process to better handle programming syntax, functions, and indentation structures.

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. **Machine learning professor notebooks:** https://github.com/fpinell/mlsa/blob/main/notebooks/Lecture_4_Deep_Learning.ipynb
5. **PyTorch Model Saving & Loading:** https://pytorch.org/tutorials/beginner/saving_loading_models.html
6. **Text Generation with PyTorch:** https://pytorch.org/tutorials/intermediate/char_rnn_generation_tutorial.html
7. **ChatGptO1 model** https://openai.com/o1/

8. **PyTorch NLP Word Embeddings Tutorial:**
   https://pytorch.org/tutorials/beginner/nlp/word_embeddings_tutorial.html