International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 09 Issue: 04 | April - 2025 SJIF Rating: 8.586 ISSN: 2582-3930

Leveraging Machine Learning for Classifying Assembly Language Snippets to Aid Programmers in Understanding Assembly Code

Devdatt Sonkusare[1], Jaydeep Tayshete[2], Vardhan Bang[3]

[1,]*Dept. of Artificial Intelligence and Data Science, AISSMS IOIT, Pune*
[2]*Dept. of Artificial Intelligence and Data Science, AISSMS IOIT, Pune*
[3]*Dept. of Artificial Intelligence and Data Science, AISSMS IOIT, Pune*

Abstract - In this study, we propose a machine learning-based approach to help programmers learn and understand assembly language by classifying assembly code snippets. Assembly language, while powerful, is often difficult to interpret due to its low-level nature and the lack of high-level context. Our model trains on assembly language snippets labeled with common programming patterns to classify unseen assembly code. By doing so, we offer a tool that can assist in reverse engineering, optimization, and the understanding of compiled programs. We generate a dataset by pairing C code snippets with corresponding assembly instructions, then train a recurrent neural network to classify these snippets. Our results demonstrate the feasibility of this approach, with the model showing strong performance in classifying assembly code types. We also discuss future directions, including generating pseudocode from assembly.

Key Words: Machine learning, compilers, pseudocode, assembly language, deep learning, RNN, LSTM

1. INTRODUCTION

Assembly Language refers to a low level programming language that uses mnemonic codes (eg. “ADD”, “SUB”) to represent machine instructions that provide direct control over computer hardware. It is important for tasks like reverse engineering, performance optimization, and embedded systems development. But the problem is that it is very difficult to understand assembly codes, making it challenging for programmers to map assembly instructions back to their higher-level language codes like C, hindering the debugging and analysis efforts.

To overcome this problem, we propose to use a machine learning based approach that classifies assembly code snippets into common patterns present in programming languages. Our aim is to train a machine learning model on labelled assembly to C code pairs, which will be able to recognize the patterns in assembly sequences, enabling automatic classification into respective code snippets. For this we will be employing Recurrent Neural Network (RNN) which can make sequential predictions based on sequential inputs.

This research aims to make the assembly languages easier to read and understand. Since assembly codes are complex and hard to follow, this machine learning model will help in identifying common patterns, almost like translating it to simpler concepts. This will be useful in tasks such as analyzing unknown software, improving program performance, or helping learn assembly structure.

2. RELATED WORK

A. Role of Assembly Language in Reverse Engineering and Optimization.

Assembly language plays an important role in reverse engineering by giving analysts direct access to binary instructions, making it possible to decompile and study software even without the original source code. Reverse Engineering frequently relies on assembly to uncover malware, analyze software and spot vulnerabilities. Beyond Reverse Engineering, assembly codes are also important for optimization, as it offers precise control over hardware, and developers can fine-tune performance-critical sections manually. This approach has been used for years, especially in embedded systems and high-performance computing, where optimization of speed and time is essential.

B. Machine Learning Applications in Code Analysis and Reverse Engineering.

Machine Learning has revolutionized the process of code analysis and reverse engineering. ML models automate tasks like the vulnerability detection, binary classification, and even converting assembly back into readable high-level code (decompiling). For example, Ding et al. developed a data mining technique to analyze executable behavior statically, helping improve malware detection. Another example is Neutron, a neural decompiler that uses deep learning to turn assembly into human readable code. These tools cut down on manual analysis while boosting the accuracy of disassembly and interpretation.

C. Machine Learning-Based Classification and Interpretation of Assembly Code.

Studies suggest the application of ML to classify and interpret assembly code. One interesting approach, Instruction2Vev, borrows from natural language processing, it treats assembly instructions like words, embedding them as vectors to improve detection in compiled programs. Another method, UniBin, skips traditional disassembly by analyzing raw binary sequences with a transformer model, reducing errors and boosting vulnerability detection. Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs) have also been successful in detecting cloned and malicious code, setting new benchmarks in binary analysis.

3. METHODOLOGY

In this section, we describe the model architecture, how the model was trained and deployed to classify assembly language snippets. Figure-1 provides basic steps of the model building process.

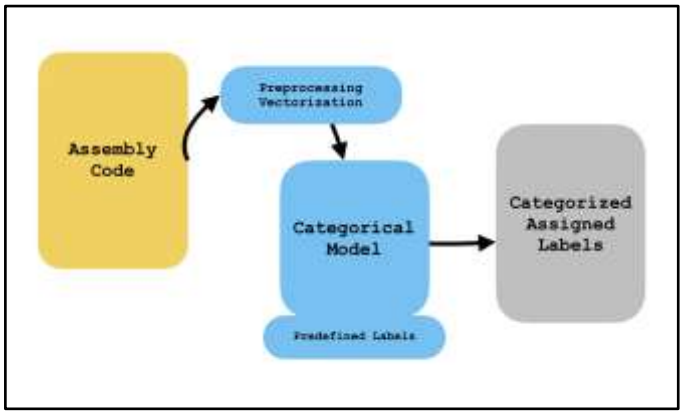

Figure-1: Model Flow

A. Data Generation

The dataset required for training the model was generated in three broad phases. The goal was to map sequences of assembly instructions to the snippet-type of their respective source code.

1. C snippets generation

A Python script was used to generate 7 types of code snippets in the C language. These snippets feature commonly used patterns in programming. These are:

1. Arithmetic operation
2. Arithmetic operation inside an if-block
3. Arithmetic operation inside a for-loop
4. Arithmetic operation inside an if-block inside an if-block
5. Arithmetic operation inside a for-loop inside an if-block
6. Arithmetic operation inside an if-block inside an for-loop
7. Arithmetic operation inside inside a for-loop inside a for-loop

For simplicity, the snippets were limited to only the main function, with one integer declaration to be used in the subsequent program. The arithmetic operations were limited to addition and subtraction. The snippets were generated with pre-defined template functions, with only the numeric values and arithmetic operations being randomized.

2. Assembly generation and cleanup

The C snippets were compiled using GCC (GNU C Compiler) on a machine with a processor of the x86-64 architecture. The generated assembly files were labelled with the snippet-type of their respective C snippet.

These assembly files were cleaned up to include only the instructions corresponding to the part of the program in the main function, from the integer declaration to the return statement.

3. Data extraction

From the assembly snippets, two types of data are extracted:

1. Main data: A sequence of just the instruction names in the given assembly snippet. Labels are also included as a separate instruction called 'label'.
2. Secondary data: All the numeric values as well as instructions corresponding to arithmetic operations. This data may be used in summary generation (See Future Scope).

Figure-2 shows an example C- snippet through all of the data generation phases.

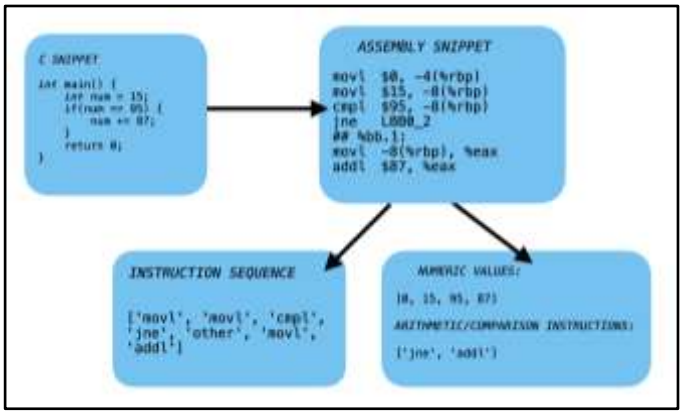

Figure-2: Data Generation

B. Model Architecture

The model architecture designed for classifying assembly language snippets is based on a deep learning approach utilizing RNNs. Recurrent Neural Networks (RNNs) are used for sequential pattern recognition. Figure-2 provides the flow of data from CSV to Model.

Table 2 presents the architecture of the model, which consists of an embedding layer, Long Short-Term Memory (LSTM) layer, dropout layer and two dense layers, along with their output shapes and parameter counts.

Table -2: Model Architecture

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 16, 64)	640
lstm (LSTM)	(None, 64)	33,024
dropout (Dropout)	(None, 64)	0
dense (Dense)	(None, 64)	4,160
dense_1 (Dense)	(None, 7)	455

1. Embedding Layer

The Embedding Layer takes a sequence of tokens (i.e. assembly language instructions) as input and maps each token into a dense vector representation. Each token is represented by

a 64-dimensional vector, allowing the model to capture semantic relationships between assembly language instructions, where similar instructions have similar vector representations.

2. LSTM Layer

The Long Short-Term Memory (LSTM) layer is a type of recurrent neural network (RNN) preferred for sequence-based tasks, like processing assembly code snippets. The LSTM layer learns the relationships between the instructions in a sequence.

3. Dropout Layer

To prevent overfitting and improve generalization during training, a Dropout Layer is included. Dropout works by randomly setting a fraction of neurons to zero during each training step.

4. Dense Layer

A dense layer is a fully connected layer that processes the feature vectors learned by the previous layers.

5. Dense_1 (Final Output Layer)

The final layer consists of 7 units, corresponding to the 7 target classes. Softmax activation function is used to generate the probability distribution over the 7 classes.

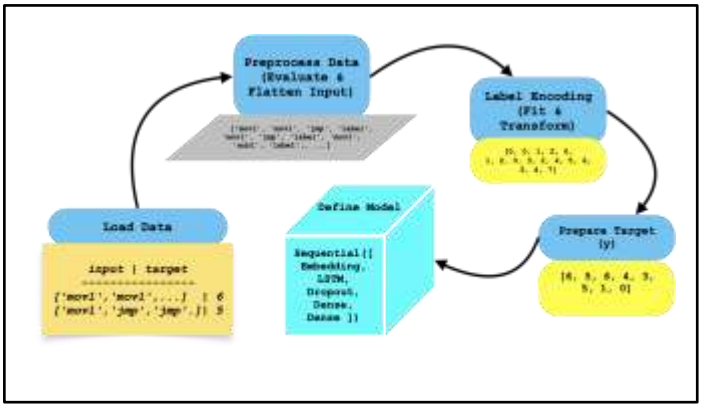

Figure-3: CSV to Model Flow of Data

C. Training

Model was trained on preprocessed labelled dataset which consists of encoded and padded assembly language instruction sequences .Below we explain the key components involved in the training process:

1. **Model Compilation**

Loss Function: The loss function used is Sparse Categorical Cross-Entropy. This loss function calculates the difference between the predicted probabilities and the true labels, helping to increase accuracy.

Optimizer: The Adaptive Moment Estimation (Adam) optimizer updates network weights iteratively based on training data.

Evaluation Metric: The model's performance is evaluated using accuracy, which measures the proportion of correct predictions made by the model.

2. Model Training

The model was trained on the encoded assembly language snippets (X_train) and their corresponding target labels (y_train). The training process was carried out over different numbers of epochs, where each epoch involves passing the entire training dataset through the model once and updating the model's weights.

Batch size was set to 32, meaning the model's weights are updated after every 32 samples. To validate the model's performance during training, the validation data (X_val, y_val) was used.

3. Training History

The model's performance is stored in the history object, which records the loss and accuracy values for both training and validation datasets. The history is used for evaluation graphs.

4. Training Results

The model achieved a test accuracy of 100% (refer Figure-3) on the test set, confirming its ability to classify assembly language instructions into one of seven target classes. The training and validation accuracy curves (shown in Figure 1) indicate that the model successfully learned to classify the assembly code snippets.

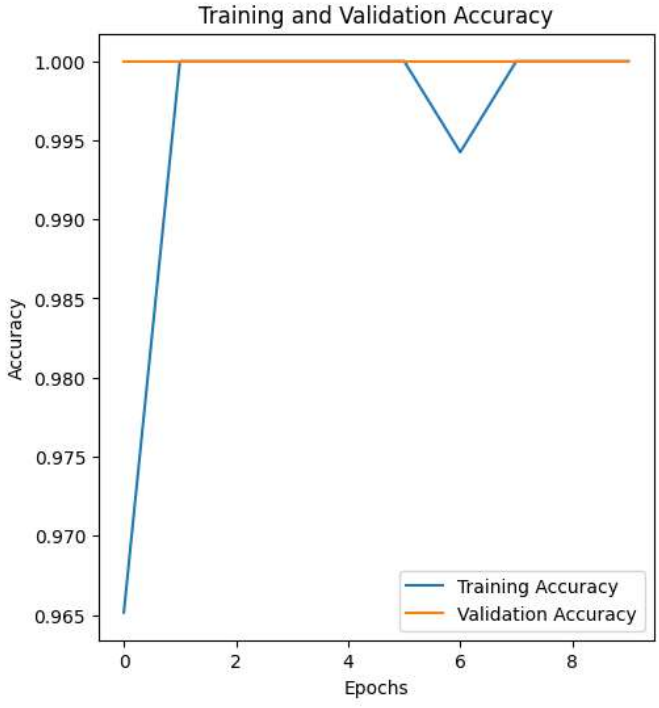

Figure-3: Training and Validation Accuracy

D. Evaluation

The model's performance was evaluated on the test dataset (X_test, y_test). This evaluation gives a final measure of how well the model is likely to perform when deployed in real-world scenarios with unseen assembly language snippets.

E. Deployment and Web Interface

A Web Application was made which allows end-users to interact with the model by providing assembly code snippets through a web interface. The system processes the input and classifies the snippet based on the trained model's predictions. The provided input is preprocessed through pipeline and uses the encoding provided during training the model by loading the encoders. The workflow is as follow:

System Workflow:

1. User Input: The user submits an assembly code snippet via the web form.
2. Data Preprocessing: The code snippet is tokenized, encoded, and padded.
3. Model Prediction: The preprocessed data is passed to the trained model for prediction.
4. Display Results: The predicted class label is displayed on the webpage.

4. CONCLUSION

Training a deep learning model to identify common programming patterns from assembly code is a potential method to make understanding of machine languages easier. It can be used as a foundation to build upon more complicated summary/pseudocode generation tools.

4. FUTURE SCOPE

Some ways in which this project can be improved:

1. Support for multiple processor architectures (ARM, RISC-V) as well as multiple compilers (clang, TCC).
2. Support for more languages.
3. Using Fine Tuned Transformers(BERT, GPT, BART, etc.).
4. Using extracted literal data (integers) as well information about arithmetic and comparison instructions to generate accurate summaries.

REFERENCES

1. Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8), 1735–1780.
2. Katz, O., Olshaker, A., Goldberg, Y., & Yahav, E. (2019). Towards Neural Decompilation. *arXiv preprint arXiv:1905.08325*.
3. Alex Sherstinsky, Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network, Physica D: Nonlinear Phenomena, Volume 404, 2020, 132306, ISSN 0167-2789, https://doi.org/10.1016/j.physd.2019.132306.
4. D. S. Katz, J. Ruchti and E. Schulte, "Using recurrent neural networks for decompilation," *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Campobasso, Italy, 2018, pp. 346-356, doi: 10.1109/SANER.2018.8330222.