Source Code Classification

1st Ajay Pawar
Electrical and Computer Engineering
Stevens Institute of Technology
Hoboken, USA
apawar11@stevens.edu

2nd Hena Kharwa
Electrical and Computer Engineering
Stevens Institute of Technology
Hoboken, USA
hkharwa@stevens.edu

3rd Mrunal Pallempati
Electrical and Computer Engineering
Stevens Institute of Technology
Hoboken, USA
mpallemp@stevens.edu

***Abstract*—The objective of this project is to develop an automated system for classifying source code based on programming language and algorithm type using Long-term Recurrent Convolutional Networks (LRCN). The project aims to bridge the gap between code understanding and machine learning by employing LRCN, a model that integrates Convolutional Neural Networks (CNNs) for feature extraction and Recurrent Neural Networks (RNNs) for sequence analysis. This hybrid approach enables the model to efficiently process and classify code samples by recognizing spatial patterns and temporal dependencies inherent in programming languages and algorithms. To achieve this, two classification tasks were tackled: first, classifying source code by programming language (C, C++, Python, Java, JavaScript, C#), followed by algorithm classification. The datasets for both tasks were collected through scraping GitHub repositories, ensuring diverse and representative samples of real-world code. Preprocessing involved tokenizing the code and converting it into embeddings that capture the syntax and semantics of the source code. The LRCN model was then trained and evaluated, achieving 96% accuracy for programming language classification and 88% accuracy for algorithm classification. The results demonstrate the effectiveness of LRCN in automating the process of code classification, achieving high accuracy across both tasks. The model's ability to handle different programming languages and algorithm structures shows its potential for applications in software development, educational tools, and intelligent programming assistants. Future work will focus on expanding the dataset, experimenting with alternative architectures, and enhancing the model's performance for domain-specific tasks. Overall, this project provides a significant contribution to the field of automated code understanding and classification using deep learning techniques.**

Keywords: source code classification, LRCN, programming language classification, algorithm classification, machine learning, GitHub, deep learning, CNN, RNN.

I. Introduction

The field of automated source code classification has gained significant importance in recent years, driven by the increasing complexity and diversity of programming languages and algorithms. This project aims to develop an advanced system for classifying source code based on programming language and algorithm type using Long-term Recurrent Convolutional Networks (LRCN). The goal is to bridge the gap between code understanding and machine learning, leveraging the power of deep learning techniques to automate the process of code analysis and categorization[1].

The project tackles two primary classification tasks: identifying the programming language of a given code snippet and classifying the algorithm implemented in the code. To achieve this, a comprehensive dataset was compiled by scraping GitHub repositories, ensuring a diverse and representative sample of real-world code. The initial dataset included 27,157 files across six programming languages: C, C++, C#, Java, JavaScript, and Python. This dataset was later expanded to include various algorithm implementations, totaling over 65,000 samples across 13 different algorithm types.

The LRCN model, which combines Convolutional Neural Networks (CNNs) for feature extraction and Recurrent Neural Networks (RNNs) for sequence analysis, was chosen as the primary machine learning algorithm for this task. This hybrid approach enables the model to efficiently process and classify code samples by recognizing both spatial patterns and temporal dependencies inherent in programming languages and algorithms. The model architecture includes embedding layers, convolutional layers, max pooling, LSTM layers, and dense layers, carefully designed to capture the nuances of source code structure.

Experimental results demonstrate the effectiveness of the LRCN approach in automating code classification. The model achieved an impressive 96% accuracy for programming language classification and 88% accuracy for algorithm classification. These results highlight the model's ability to handle different programming languages and algorithm structures, showcasing its potential for applications in software development, educational tools, and intelligent programming assistants.

Compared to existing solutions, such as those using Support Vector Machines (SVMs) or traditional natural language processing techniques, the LRCN approach offers several advantages. It eliminates the need for complex language-specific parsing, making it more adaptable to different programming languages. Additionally, by leveraging both CNNs and RNNs, the model can capture both local syntax patterns and broader structural elements of the code, leading to improved classification accuracy. This approach builds upon and extends previous work in the field, addressing limitations of earlier methods that relied heavily on documentation or struggled with limited context in code snippets.

II. Related Work

Ugurel et al. (2002) [1] explored using machine learning, specifically Support Vector Machines (SVMs), to automatically classify source code into application topics and programming languages. Using a dataset from various online archives, they found that SVMs can successfully categorize code based on features extracted from code, comments, and documentation.

However, the study acknowledges that performance can be affected by the number of examples in each category and the fuzziness of certain categories like "utilities."

Nordström (2015) [2] investigated the classification of Java source code for a Case-Based Reasoning (CBR) system. This system analyzes source code using a combination of type filtering, documentation analysis, syntactic analysis, and semantic analysis. The research found that relying heavily on documentation for classification leads to an imbalanced knowledge base due to the inconsistency of code documentation practices.

Kennedy et al. (2016) [3] proposed using natural language classifiers for software language identification. Their approach leverages the similarities between source code and natural language to achieve accurate classification. This method avoids the need for complex language-specific parsing, making it more adaptable to different programming languages.

Al-Thubaity et al. (2018) [4] developed SCC, a tool for automatically classifying code snippets from Stack Overflow using a Multinomial Naive Bayes (MNB) algorithm. Their research highlights the challenge of classifying code snippets compared to complete source code files due to their limited context. Despite the difficulty, SCC achieved reasonable accuracy, demonstrating the feasibility of applying machine learning to this task.

Hönel et al. (2018) [5] investigated using source code density as a feature to improve the accuracy of automatic commit classification into maintenance activities. They found that incorporating code density, particularly net-size, can significantly enhance classification accuracy. This research emphasizes the value of considering code metrics beyond simple size measures for better understanding code changes.

Phan et al. (2019) [6] presented tree-based approaches, combining TBCNN with kNN and SVMs, for classifying source code. They also introduced techniques for pruning and reconstructing Abstract Syntax Trees (ASTs) to reduce data complexity and improve accuracy. This study highlights the effectiveness of AST-based representations for capturing both structural and semantic information in source code.

Zhang et al. (2022) [7]provided a comprehensive survey of Automatic Source Code Summarization (ASCS) techniques. The survey categorizes and analyzes various ASCS methods, highlighting the evolution from template-based to deep learning-based approaches. The authors emphasize the need for combining generative and search-based methods, and the importance of graph neural networks for representing code in future ASCS systems.

III. Our Solution

We tend to predict various data structure algorithms along with highlighting the programming language, below sections consist of dataset used, the ml models used to train the dataset with and their implementation details.

A. Description of Dataset

Initially, we tried scrapping source files from GitHub Repositories and saving them in separate folders based on six prominent programming languages: C, CPP, C#, Java, JavaScript and Python. Although, Github API has a limit of scrapping 5000 files per hour so we collated total 27157 files. File distribution for each category is shown in the table below:

Language	Number of Files
C	2997
CPP	5896
C#	172
Java	6943
JavaScript	5925
Python	5224

We implemented our initial model mentioned in section III-B2

Post that, we thought to extend the dataset and again, we manually collated the files by scrapping various GitHub's data structures and algorithms repositories for five programming languages: C, CPP, Java, JavaScript and Python. We collated the algorithms mentioned below each having mentioned sample size:

Algorithm Name	**Total Samples**
Bubble Sort	5100
Merge Sort	5100
Quick Sort	5100
Selection Sort	5100
Insertion Sort	5100
Bucket sort	5100
Dijkstra's Algorithm	5100
Bellman-Ford Algorithm	4255
Floyd-Warshall Algorithm	4865
Binary Search	5100
Ternary Search	5100
Depth-First Search	5100
Bredth-First Search	5100

TABLE I. Algorithm Names and Total Samples

Note : Some algorithms i.e. Bellman-Ford and Floyd-Warshall had less than 1020 for some languages. Also, due to Github token limit, we could only scrape 1020 entries at one go. The dataset is on open access at dataset

Preprocess Text Data

Text data requires several preprocessing steps:

1) **Visualizing scraped files:** After collating all the source files, we counted and checked total size for each algorithm and language.
2) **Deleting Extra Files:** After extracting the dataset, inspect the directory structure for redundant files like metadata or system files. These files can clutter the dataset and should be deleted to streamline the analysis.
3) **Encoding Text Data:** As the text data contains characters from various languages or formats,we ensured it is encoded consistently in UTF-8, to prevent issues during processing.
4) **Tokenization:** Break down the text into smaller units, such as words or sentences. Tokenization helps convert unstructured text into a structured format suitable

for analysis. We used Tensorflow's Tokenizer, with below mentioned limits.

```
MAX_VOCAB_SIZE = 20000
# Maximum vocabulary size
MAX_SEQUENCE_LENGTH = 500
# Maximum sequence length
```

5) **Text to sequence**: texts_to_sequences() method takes the text data as input and returns a list of sequences, where each sequence is a list of integers representing the tokens in the corresponding text
6) **Padding**:pad_sequences() pads the sequences to a uniform length as deep learning models require input sequences to have the same length.

Below figure shows heat map distribution of final dataset:

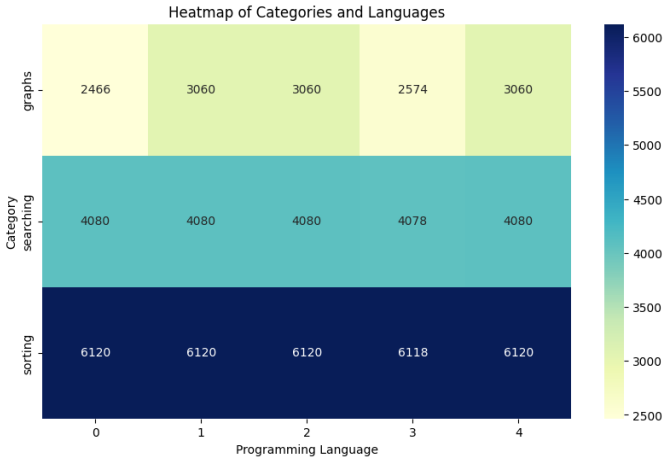

Fig. 1. Heatmap Distribution of Final Dataset

B. Proposed Model

The algorithm used is LRCN(Long-term Recurrent Convolutional Network)[8], a type of neural network architecture that combines the strengths of Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) for classification.

1) Algorithm Selection and Justification: For this project, we selected an LSTM-based neural network. LSTMs are particularly well-suited for sequence data due to their ability to capture long-term dependencies, which is critical for tasks involving text or time-series data. The primary motivation for using LSTM is its effectiveness in understanding context within sequences, making it an ideal choice for source code classification tasks, where the order of tokens significantly impacts the semantics.

Traditional recurrent neural networks (RNNs) model temporal dynamics using the following recurrence equations:

- $h_t = g(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$
- $z_t = g(W_{hz}h_t + b_z)$
- where:
  - g is an element-wise non-linearity, such as a sigmoid or hyperbolic tangent.
  - x_t is the input at time t.
  - $h_t \in \mathbb{R}^N$ is the hidden state at time t with N hidden units.
  - z_t is the output at time t.
  - W_{xh}, W_{hh}, and W_{hz} are weight matrices.
  - b_h and b_z are bias vectors.
- For an input sequence $x_1, x_2, \ldots, x_T$, the updates are computed sequentially, starting with h_1 (given $h_0 = 0$).

Long Short-Term Memory (LSTM) units address the vanishing gradient problem in RNNs. The LSTM updates at time step t given inputs x_t, h_{t-1}, and c_{t-1} are:

- $i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i)$
- $f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f)$
- $o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o)$
- $g_t = \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c)$
- $c_t = f_t \odot c_{t-1} + i_t \odot g_t$
- $h_t = o_t \odot \tanh(c_t)$
- where:
  - $\sigma(x) = (1 + e^{-x})^{-1}$ is the sigmoid non-linearity.
  - $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1$ is the hyperbolic tangent non-linearity.
  - $x \odot y$ denotes the element-wise product of vectors x and y.
- In addition to the hidden state $h_t \in \mathbb{R}^N$, the LSTM has:
  - an input gate $i_t \in \mathbb{R}^N$.
  - a forget gate $f_t \in \mathbb{R}^N$.
  - an output gate $o_t \in \mathbb{R}^N$.
  - an input modulation gate $g_t \in \mathbb{R}^N$.
  - a memory cell $c_t \in \mathbb{R}^N$.
- The memory cell c_t combines the previous memory cell c_{t-1} modulated by f_t, and g_t modulated by the input gate i_t. The gates learn to selectively forget previous memory or consider the current input.

2) Network Structure: As stated earlier, we have used two different LRCN models for initial dataset and its extension.

The designed network for initial model consists of the following layers:

- **Input Layer**: Accepts tokenized and padded sequences of code snippets.
- **Embedding Layer**: Converts input tokens into dense vector representations.
- **Convolutional Layer (Conv1D)**: Extracts local features from the embedded sequences.
- **Max Pooling Layer (MaxPooling1D)**: Reduces the spatial dimensions of the convolutional output.
- **LSTM Layer**: Processes the sequence data, capturing long-term dependencies.

- **Dense Layer**: Fully connected layer for final classification.
- **Output Layer**: Produces class probabilities for the different programming languages.

Below figure depicts initial model:

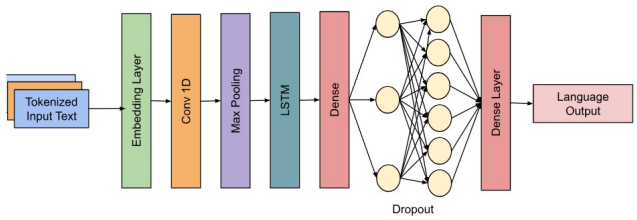

Fig. 2. LRCN Architecture for Initial Dataset

Next, we design a new network structure to fit the second dataset, the description of various layers are mentioned below:

- **Input Layer**: The input layer accepts preprocessed sequences of tokens derived from source code.
- **Embedding Layer**: Converts tokens into dense vector representations, enabling the model to learn relationships between tokens.
- **Bidirectional LSTM Layer**: A bidirectional LSTM layer is used to capture dependencies in both forward and backward directions, enhancing the model's understanding of context.
- **Dense Layers**: Fully connected layers are employed to aggregate features extracted by the LSTM. These layers include dropout for regularization and batch normalization for improved convergence.
- **Output Layer**: A final dense layer with a softmax activation function outputs the probabilities for each class corresponding to different categories of source code.

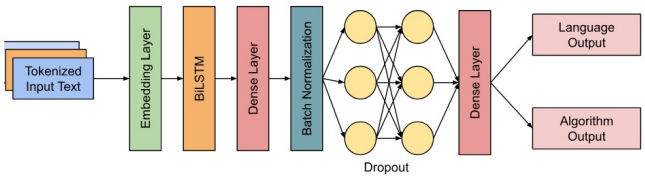

Fig. 3. LRCN Architecture for Final Dataset

3) Model Summary: : Firstly, below is the initial model summary:

Model: "functional"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 500)	0
embedding (Embedding)	(None, 500, 128)	2,560,000
conv1d (Conv1D)	(None, 498, 128)	49,280
max_pooling1d (MaxPooling1D)	(None, 249, 128)	0
lstm (LSTM)	(None, 128)	131,584
dense (Dense)	(None, 64)	8,256
dropout (Dropout)	(None, 64)	0
dense_1 (Dense)	(None, 6)	390

Total params: 2,749,510 (10.49 MB)

Trainable params: 2,749,510 (10.49 MB)

Non-trainable params: 0 (0.00 B)

Fig. 4. Initial Model Summary

Secondly, the figure below shows the final model summary:

Model: "functional"

Layer (type)	Output Shape	Param #	Connected to
input (InputLayer)	(None, 500)	0	-
embedding (Embedding)	(None, 500, 128)	2,560,000	input[0][0]
bidirectional_lstm (Bidirectional)	(None, 256)	263,168	embedding[0][0]
dense_1 (Dense)	(None, 128)	32,896	bidirectional_ls…
batch_norm_1 (BatchNormalizatio…	(None, 128)	512	dense_1[0][0]
dropout_1 (Dropout)	(None, 128)	0	batch_norm_1[0][…
dense_2 (Dense)	(None, 128)	16,512	dropout_1[0][0]
algorithm_output (Dense)	(None, 13)	1,677	dropout_1[0][0]
language_output (Dense)	(None, 5)	645	dense_2[0][0]

Total params: 2,875,410 (10.97 MB)

Trainable params: 2,875,154 (10.97 MB)

Non-trainable params: 256 (1.00 KB)

Fig. 5. Final Model Summary

C. Implementation Details

This subsection details the implementation of the source code classification system, focusing on training performance testing, hyperparameter tuning, and techniques used to improve model performance. The code is saved in .ipynb jupyter notebooks at dataset location

1) Training: : Now as the models are ready, we set the training parameters to fit the datasets, and test their performance:

Parameter Initialization: The initial choices for the key parameters are as follows:

- **Embedding Dimension:** 128
- **LSTM Units:** 128
- **Dropout Rate:** 0.5
- **Learning Rate:** 0.001 (using the Adam optimizer)
- **Batch Size:** 32
- **Sequence Length:** 500

For second model, below mentioned parameters were chosen:

- **Embedding Dimension**: 128
- **LSTM Units**: 256
- **Dropout Rate**: 0.5
- **Learning Rate**: 0.001 (using the Adam optimizer)
- **Batch Size**: 32
- **Sequence Length**: 500 (maximum sequence length after padding)

The Adam optimizer (**A**daptive **M**oment Estimation) was selected for the following reasons:

- **Adaptive Learning Rates:** Adam dynamically adjusts the learning rate for each parameter based on the first and second moments of the gradient. This leads to faster convergence and better performance compared to fixed learning rate optimizers.
- **Efficient Computation:** The computational cost of Adam is low, making it suitable for training deep learning models with large datasets.
- **Robustness:** Adam performs well in a variety of scenarios, including sparse gradients and noisy data.
- **Hyperparameter Tuning:** Adam works well with its default parameters (learning rate = 0.001, $\beta_1 = 0.9$, $\beta_2 = 0.999$), reducing the need for extensive hyperparameter tuning.
- **Suitability for NLP Tasks:** Adam is widely used in natural language processing tasks as it effectively handles non-stationary objectives and stochastic optimization problems.

These parameters are subject to tuning during the training process to achieve optimal performance. The network is trained using a categorical cross-entropy loss function, appropriate for multi-class classification tasks.

2) Testing and Validation: The dataset is split into training and testing subsets using an 80-20 split. The testing set is held out during training and used exclusively for evaluating model performance. The primary evaluation metric is accuracy, supplemented by precision, recall, and F1-score to provide a comprehensive assessment of the model.

To ensure robustness, cross-validation is applied during hyperparameter tuning. This technique helps prevent overfitting by validating the model across multiple folds of the data set.Below is the description of training parameters of **Initial Model:**

- **Data Splitting:**
  - `train_test_split()` is used to split the dataset into training and validation sets.
  - Inputs (`X_padded`) and labels (`y_categorical`) are divided with 80% of the data used for training (`X_train`, `y_train`) and 20% for validation (`X_val`, `y_val`).
  - The parameter `random_state=42` ensures reproducibility by using a fixed random seed for splitting.
- **Model Training:**
  - The `model.fit()` function trains the deep learning model using the training data (`X_train`, `y_train`).
  - Validation is performed on the validation set (`X_val`, `y_val`) at the end of each epoch to monitor the model's performance on unseen data.
  - **Epochs:** The training process runs for 10 epochs, allowing the model to update its weights iteratively.
  - **Batch Size:** A batch size of 32 means the model processes 32 samples at a time before updating its weights, balancing memory efficiency and model performance.
- **Output:**
  - The function returns a `history` object that stores metrics such as training and validation loss and accuracy for each epoch.
  - These metrics can be visualized to analyze the training process and identify potential overfitting or underfitting.

Next, is the description of training parameters of **Final Model:**

- **Training the Model:**
  - The `model.fit()` function is used to train the deep learning model.
  - The model is trained on inputs (`X_train`) and two separate outputs:
    - `'algorithm_output'`: Targets (`y_train_algo`) for predicting algorithms.
    - `'language_output'`: Targets (`y_train_lang`) for predicting programming languages.

- **Validation Split:**
  - `validation_split=0.2` allocates 20% of the training data for validation, ensuring the model is evaluated on unseen data at the end of each epoch.
- **Training Configuration:**
  - **Epochs:** The model is trained for 10 epochs, iteratively updating weights to minimize loss.
  - **Batch Size:** A batch size of 32 processes data in small chunks, optimizing memory usage and computational efficiency.
- **Callbacks:**
  - `early_stopping`: Monitors the validation loss to stop training early if no improvement is detected, preventing overfitting and saving computation time.
  - `lr_scheduler`: Adjusts the learning rate dynamically during training, potentially improving convergence by reducing the rate when nearing the optimal solution.
- **Output:**
  - The `history` object stores training and validation metrics (e.g., loss and accuracy) for both outputs across epochs.
  - These metrics can be analyzed to evaluate the model's performance and detect issues like overfitting or underfitting for either output.

3) Hyperparameter Tuning: Key hyperparameters are tuned to optimize performance:

- **Embedding Dimension**: Initially set to 128, varied between 64 and 256.
- **LSTM Units**: Tested with 128, 256, and 512 units to balance performance and computational cost.
- **Dropout Rate**: Adjusted between 0.3 and 0.5 to mitigate overfitting.
- **Learning Rate**: Experimented with values ranging from 0.001 to 0.0001 using the Adam optimizer.
- **Maximum Sequence length**: Experimented with values ranging from 100 to 500, selecting 500 as the optimal length.
- **Vocabulary Size**: Experimented with values ranging from 5000 to 20000 values of tokens , selecting 20000 as the optimal number of tokens.

Hyperparameters	Values
Epochs	10
Batch Size	32
Maximum Sequence Length	500
Vocabulary Length	20000
Learning Rate	0.001
Loss	Categorical Crossentropy
Optimizer	Adam

TABLE II. FINE-TUNED HYPERPARAMETERS FOR INITIAL MODEL

Hyperparameters	Values
Epochs	20
Batch Size	32
Patience	10
Factor	0.5
Learning Rate	0.001
Loss	Sparse Categorical Crossentropy
Optimizer	Adam

TABLE III. FINE-TUNED HYPERPARAMETERS FOR FINAL MODEL

The best-performing configuration includes an embedding dimension of 128, 256 LSTM units, a dropout rate of 0.5, and a learning rate of 0.001.

4) Performance Improvement Techniques: Several techniques are employed to enhance the model's performance:

- **Bidirectional LSTM**: Captures dependencies in both forward and backward directions, improving context understanding.
- **Batch Normalization**: Applied after dense layers to stabilize and accelerate training.
- **Dropout**: Reduces overfitting by randomly deactivating neurons during training.
- **Early Stopping**: Monitors validation loss and stops training when performance ceases to improve.

IV. RESULTS

The initial and final models achieve an accuracy of 94.65% and 88% on the test set. The accuracy trends during training for both models are shown in the figures below, respectively:

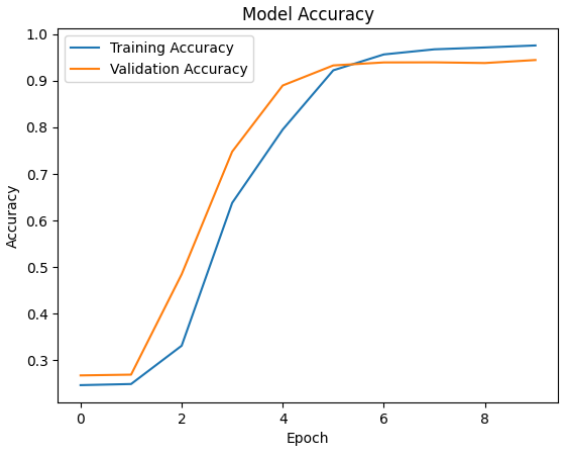

Fig. 6. Initial Model Accuracy

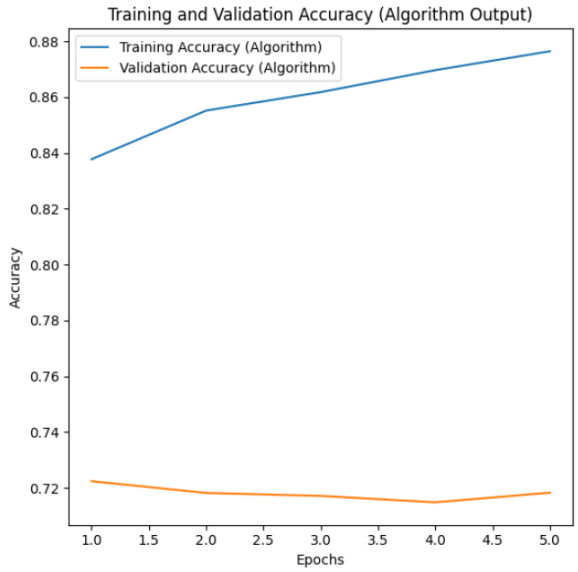

Fig. 7. Final Model Accuracy - Algorithm

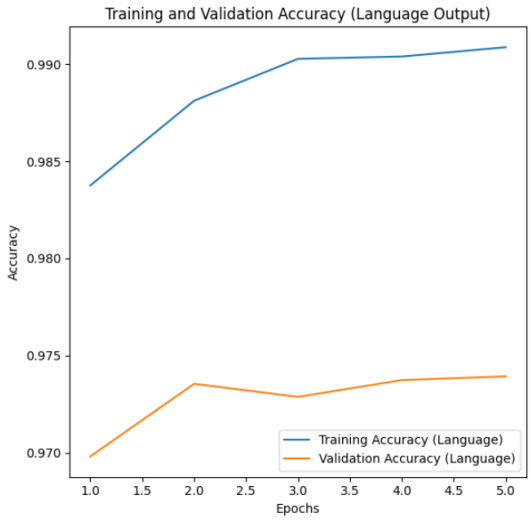

Fig. 8. Final Model Accuracy - Language

The images below depict the Loss vs Epochs for the initial and final models:

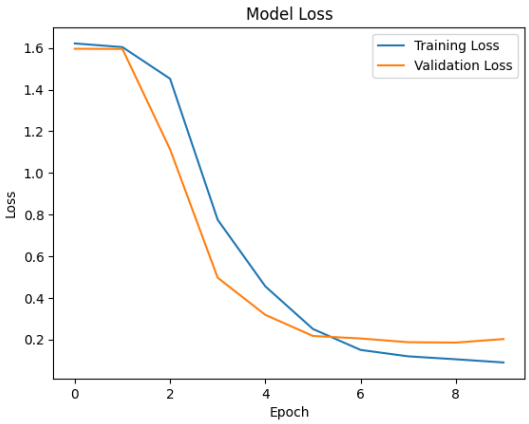

Fig. 9. Initial Model Loss vs Epochs

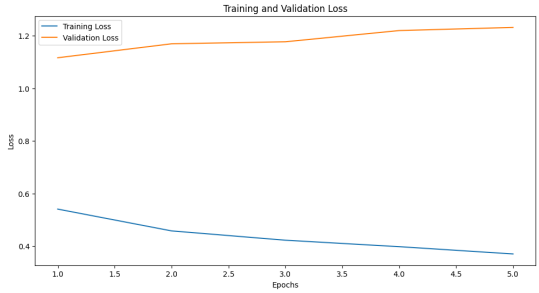

Fig. 10. Final Model Loss vs Epochs

Below are the confusion matrices for the initial and final models:

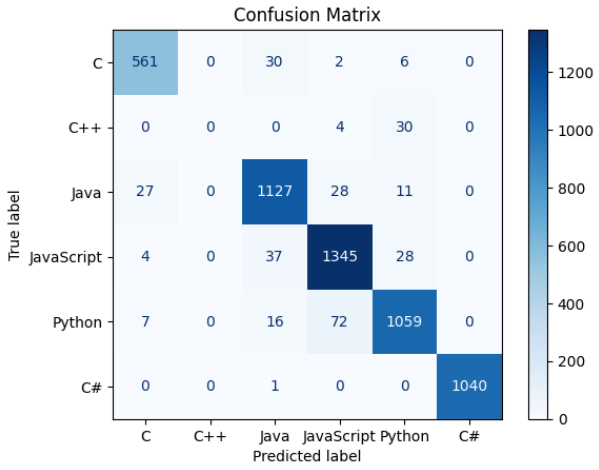

Fig. 11. Initial Model Confusion Matrix

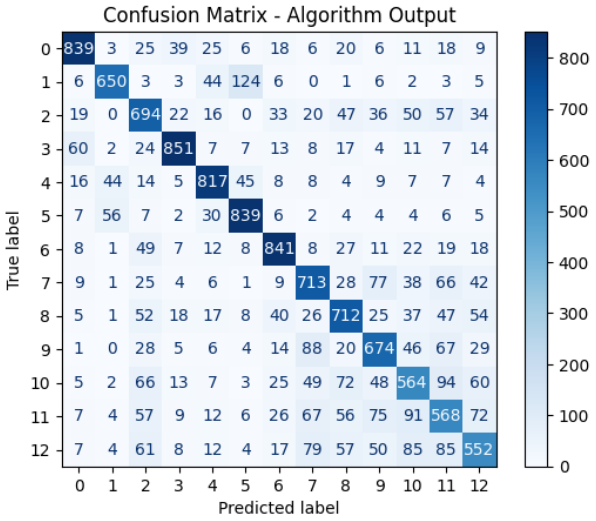

Fig. 12. Final Model Confusion Matrix - Algorithm

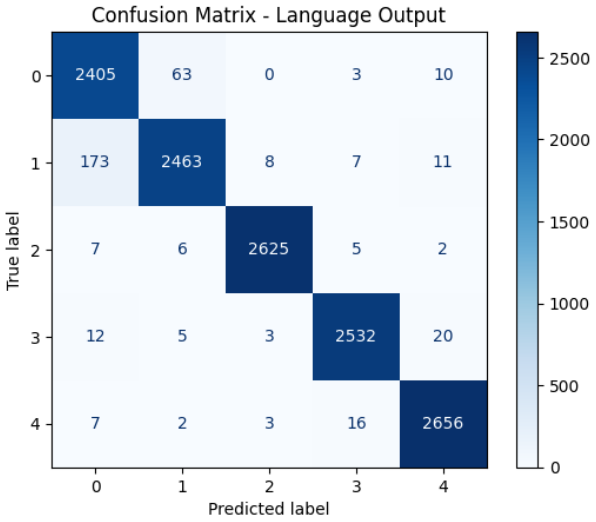

Fig. 13. Final Model Confusion Matrix - Language

V. Conclusion

This project demonstrates the successful implementation of Long-term Recurrent Convolutional Networks (LRCN) for classifying programming languages and algorithms, highlighting its potential for automated code analysis. Initially, we focused on programming language classification, achieving an impressive accuracy of 96% across six programming languages: C, C++, Python, Java, JavaScript, and C#. We then extended our approach to classify algorithms, training the LRCN model on a larger dataset and achieving an accuracy of 88%. Both datasets were obtained through scraping GitHub repositories, ensuring a comprehensive and diverse set of code samples for training and evaluation.

The results affirm the robustness and effectiveness of the LRCN model in handling source code classification tasks, capturing both spatial and temporal dependencies within the code. This work contributes to the broader field of automated code understanding and algorithm classification, offering applications in areas such as software maintenance, educational tools, and intelligent programming assistants. Future work will focus on improving classification accuracy by incorporating larger, more varied datasets, experimenting with other architectures, and fine-tuning the model for domain-specific tasks. The success of this project paves the way for advanced, scalable systems in the realm of software development and code analysis.

VI. Team Efforts

The task break for all the team members is given in below image

Task	Name
Data Scraping	Ajay Pawar
Preprocessing and Visualization	Ajay Pawar, Mrunal Pallempati
Preparing Report	Ajay Pawar, Mrunal Pallempati, Hena Kharwa
LRCN Implementation for Language Classification	Mrunal Pallempati
LRCN Implementation for Algorithm Classification	Hena Kharwa
Fine-tuning both models	Hena Kharwa

Fig. 14. Task Break-up

References

[1] S. Ugurel, R. Krovetz, and C. L. Giles, "What's the code? automatic classification of source code archives," in *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 632–638. [Online]. Available: https://doi.org/10.1145/775047.775141

[2] M. Nordström, "Automatic source code classification: Classifying source code for a case-based reasoning system," 2015.

[3] J. K. Van Dam and V. Zaytsev, "Software language identification with natural language classifiers," in *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 624–628.

[4] K. Alreshedy, D. Dharmaretnam, D. M. German, V. Srinivasan, and T. A. Gulliver, "Scc: automatic classification of code snippets," *arXiv preprint arXiv:1809.07945*, 2018.

[5] S. Hönel, M. Ericsson, W. Löwe, and A. Wingkvist, "Using source code density to improve the accuracy of automatic commit classification into maintenance activities," *Journal of Systems and Software*, vol. 168, p. 110673, 2020.

[6] A. V. Phan, P. N. Chau, M. Le Nguyen, and L. T. Bui, "Automatically classifying source code using tree-based approaches," *Data & Knowledge Engineering*, vol. 114, pp. 12–25, 2018.

[7] C. Zhang, J. Wang, Q. Zhou, T. Xu, K. Tang, H. Gui, and F. Liu, "A survey of automatic source code summarization," *Symmetry*, vol. 14, no. 3, p. 471, 2022.

[8] J. Donahue, L. Anne Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell, "Long-term recurrent convolutional networks for visual recognition and description," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 2625–2634.