**Multiple Author Identification Using Source Code**

Balne Thrinath | Pabba Vamshi Krishna | Ganam Giri Prasad |

Gaddam Poojitha | Vepakomma Lakshmi Pranuthanjali

**Abstract**

Authorship analysis has played an important role in identifying the true writers of literary texts. Of late, there is an increased interest in authorship identification of source codes. This is partially due to a rise in anonymous malware, code injections, and digital forensics[1][7]. Here we implemented different models to identify authors of compilable source codes written using Python. The proposed work implements code2seq, word embedding modeling along with SVM, DNN, LSTM and MDA.

**Introduction**

Authorship attribution is about analyzing the text, patterns and ultimately finding the author of code, book etc[8]. This evaluation has heavy applications in forensic science, plagiarism detection and accountability for published work[6]. With the rise in online academic tests, malware, and other code based plagiarism, source code authorship identification has gained importance over time[3].

Different programmers have distinct styles of writing code. Features in Python such as naming the function, modules included, iterator variables etc., are useful while distinguishing one author from another. A machine learning and deep learning based authorship identification approach tries to automatically learn from the data the contribution of different features towards the author identification.

The proposed work uses feature extraction as it plays an important role in this task. The presented model uses a combination of AST path, code2seq, word embeddings for feature extraction and SVM, DNN, LSTM and MDA for authorship recognition.

**Backend work**

Here extracting the best features is the most important thing in machine learning models and for best features data is the most important thing. First we collected some codes from github of authors who wrote codes in python language and in the step of preprocessing we removed the files which are not useful for our feature extraction as we kept the threshold of 20 files for each author and total dataset of 20 authors with 20 files each, so total 400 files of data.

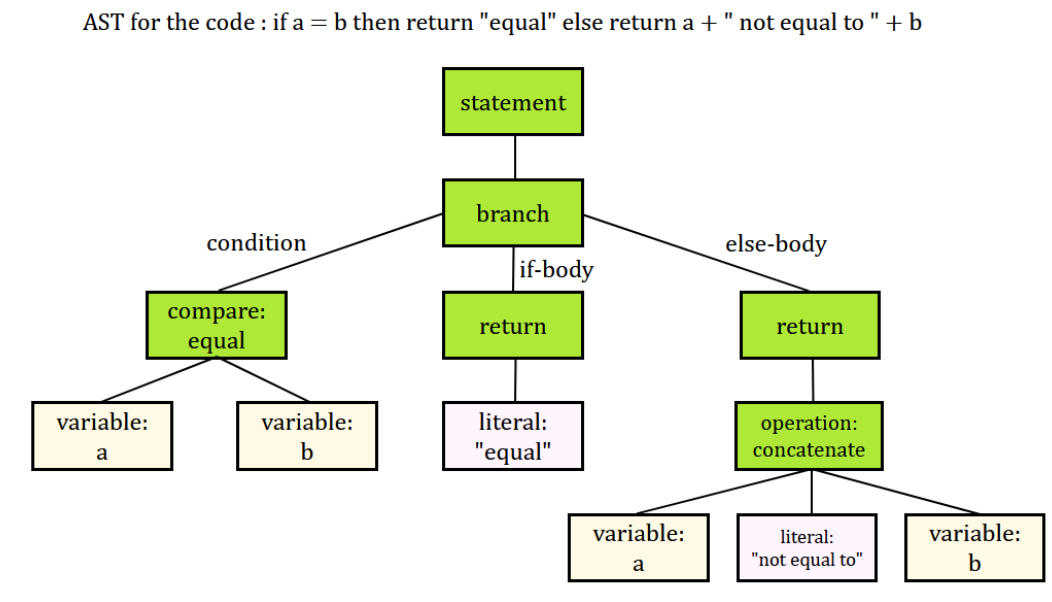
**Proposed Technique:**

Four different models were submitted. Here for all models same features extracted using methods like AST path, code2seq method and models are SVM, DNN, LSTM and MDA[8][4][5].

**Feature Extraction**

**1.Generating AST path:**

Here we generate ast path using ast module. It's a built-in module that provides tools for working with the abstract syntax tree of Python code[10]. The abstract syntax tree is a hierarchical representation of the syntactic structure of a program written in a programming language.



Using the ast module, you can parse Python code and traverse its abstract syntax tree to analyze and manipulate the code's structure[13]. This can be particularly useful for extracting features from code for machine learning purposes, such as code classification, code similarity analysis, code summarization, and more.

* **Parsing:** The ast.parse() function is used to parse a string of Python code and generate an abstract syntax tree. It returns a top-level ast.Module node that represents the entire module.
* **Node Classes:** The ast module provides various classes that correspond to different types of nodes in the AST. For example:

1. **ast.Module:** Represents the root node of the AST for a Python module.
2. **ast.FunctionDef:** Represents a function definition.
3. **ast.ClassDef:** Represents a class definition.
4. **ast.Assign:** Represents an assignment statement.
5. **ast.For, ast.While**: Represent loop constructs.
6. **ast.If, ast.Else:** Represent conditional constructs.

Here from abstract tree we generate useful features like comment density, code duplicate etc.,

Here before generating to code2seq we can generate few features from ast path but to extract features like Lexical, Syntactic and Semantic features we need to convert ast to code2seq.

**2. Ast to code2seq:**

**Steps in code2seq to extract features:**

Here first we initialize the neural network architecture that will be used to process the input AST path[10].

**Initialization and Forward Pass:**

* **self.embedding (Embedding Layer):**

**Purpose:** The embedding layer converts input tokens (node types from the AST path) into dense vectors. It learns a mapping from token indices to continuous vector representations. These embeddings capture semantic relationships between different tokens.

**Usage:** The embedded vectors are fed into the GRU layer for sequential processing.

* **self.gru (Gated Recurrent Unit**):

**Purpose:** The GRU layer processes the embedded input sequence in a sequential manner. It captures contextual information by considering the previous tokens in the sequence.

**Usage:** The GRU produces hidden states that capture the context of the input sequence. These hidden states are then used to extract different types of features.

* **self.fc (Fully Connected Layer for Main Task):**

**Purpose:** The fully connected layer produces the main output of the model. It can be used for tasks like author classification, where the goal is to predict a specific label or class for the given input sequence.

**Usage:** The output of this layer provides predictions for the main task.

* **hidden:** The final hidden state of the GRU is extracted.
* **output:** The main task output, author classification is obtained from the fully connected layer.

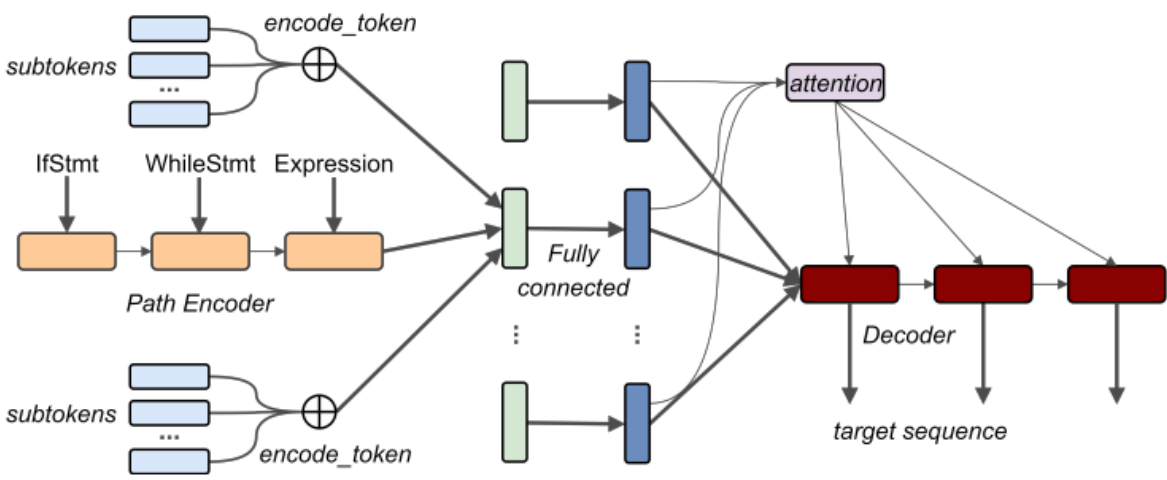
**Activation Functions**

* In GRU layer we use Rectified Linear Unit (ReLU) activation function

f(x) = max(0, x)

* In FC layer we use Tanh function.

f(x) = (e^x - e^(-x)) / (e^x + e^(-x))



**self.lexical\_fc, self.syntactic\_fc, self.semantic\_fc, self.layout\_fc (Feature Extraction Layers):**

**Purpose**: These fully connected layers are used to extract different types of features from the hidden state of the GRU. Each layer extracts features that capture specific aspects of the code.

**self.lexical\_fc:** Captures lexical (word-level) information in the code.

**self.syntactic\_fc:** Captures syntactic (structural) information in the code.

**self.semantic\_fc:** Captures semantic (meaning) information in the code.

**self.layout\_fc:** Captures layout information (spatial arrangement) in the code.

**Usage:** The output of these layers provides feature vectors that can be used for downstream analysis or tasks.

The purpose of these layers is to create a multi-faceted representation of the input code. By extracting various types of features, the model aims to capture different dimensions of information present in the code, including the meaning of tokens, their relationships, the structural organization, and even the spatial layout.

**Encoding Phase**

The input sequence (AST path or any other sequence) is processed through the model's encoder layers, which can include embedding layers, RNN layers (such as GRU or LSTM), and possibly other fully connected layers.The encoder layers transform the input sequence into a fixed-size encoded representation (hidden state) that captures relevant information from the input.

**Decoding Phase**

After the encoding phase, the encoded representation is used to generate the output sequence. In the case of Code2Seq, the generated output could be a code snippet, a sequence of tokens, or any other relevant information.

**Output Generation**

The decoder typically starts with a special token (e.g., <START>) and generates tokens step by step.At each decoding step, the model takes the previous generated token and the current state to predict the next token in the sequence.This process continues until an end token (e.g., <END>) is generated, or until a predetermined maximum sequence length is reached.

**Models Used for identification**

**1.Support Vector Machine(SVM):**

**STEP 1: Data Loading and Feature Extraction**

We start by loading the dataset from a CSV file using the pandas library. The dataset contains code features that describe the characteristics of code snippets, along with information about the authors who contributed them.

**STEP 2: Feature Engineering**

**Numeric Features:** We extract numeric features such as Code Complexity, Cyclomatic Complexity, Lines of Code, and more, which provide insights into the structural and complexity aspects of the code.

**Non-Numeric Features:** Code style and structure are captured by non-numeric features like Lexical Features, Syntactic Features, Semantic Features, and Layout Features. We convert these features into a suitable format using one-hot encoding.

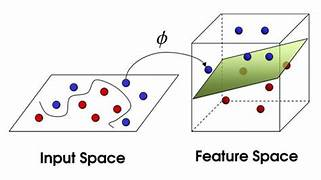
**Textual Features:** Features like Word Frequencies and Character N-grams capture the textual characteristics of the code[14]. We process these features by summing their values, offering insights into the specific keywords and n-gram patterns used by different authors.

**STEP 3: Data Preparation**

**Imputation and Normalization:** To handle any missing values, we employ the SimpleImputer to fill in the mean of each column. Next, we normalize the features using the StandardScaler to ensure all features are on the same scale.

**Label Encoding:** The target column represents the authors of the code snippets. We convert these author labels into numeric format using LabelEncoder.

**Train-Test Split:** We partition the dataset into a training set (80%) and a testing set (20%) using train\_test\_split to enable model evaluation.



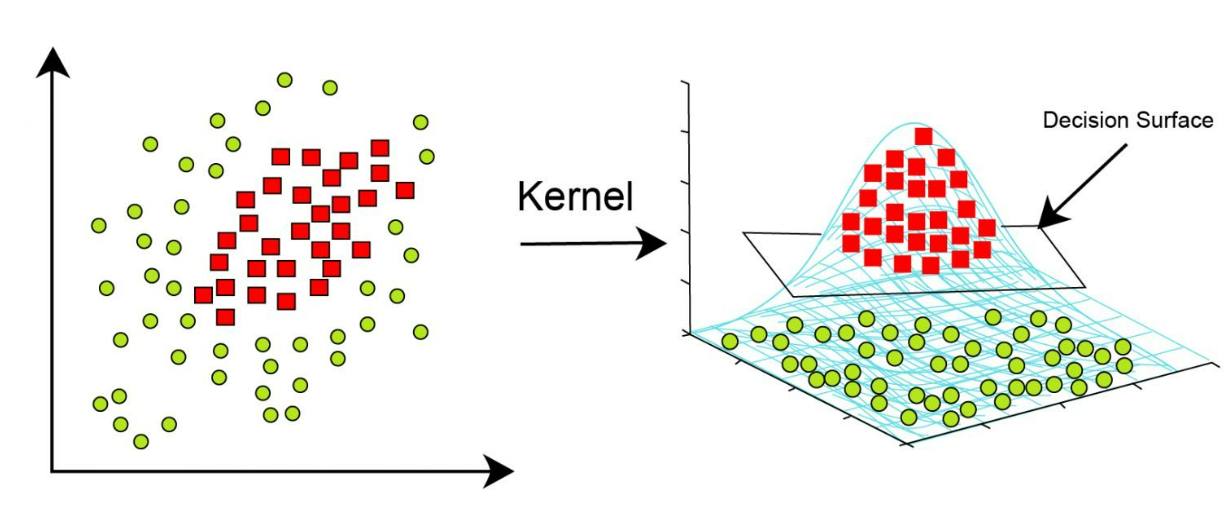
**STEP 4: Model Building and Hyperparameter Tuning**

We employ a machine learning model to recognize the patterns in the code features and predict the authors behind the code snippets. Specifically, we use a Support Vector Machine (SVM) classifier for this purpose.

**Hyperparameter Tuning:** Employing GridSearchCV, we explore different hyperparameter combinations (e.g., C for penalty parameter, kernel for kernel function) to identify the optimal configuration for our SVM model. This step ensures that the model performs optimally by cross-validating on the training data.

The penalty parameter 'C' in a Support Vector Machine (SVM) is a hyperparameter that controls the trade-off between achieving a low training error and a low testing error. In other words, it balances the desire to minimize the misclassification of training samples with the goal of having a simpler decision boundary that generalizes well to unseen data.

The 'C' parameter can have different effects based on whether you're working with a linear SVM or a kernel SVM:



**Linear SVM:**

* For a linear SVM, 'C' determines the degree to which the model tolerates misclassification of training examples.
* A smaller 'C' value encourages a wider margin and allows some training examples to be misclassified (soft margin). This can help the model generalize better to unseen data and avoid overfitting.
* A larger 'C' value enforces a stricter margin, aiming to correctly classify more training examples (hard margin). This might lead to overfitting if the data is noisy or not linearly separable.

**Kernel SVM:**

* In a kernel SVM, the 'C' parameter still plays a role in controlling the trade-off between misclassification and the complexity of the decision boundary.
* However, the influence of 'C' is combined with the effect of the chosen kernel function. The kernel function maps the data into a higher-dimensional space, where a linear decision boundary might become nonlinear.
* With a larger 'C', the model becomes more sensitive to individual data points, potentially leading to overfitting if the kernel is too complex or if the data is noisy.

**STEP 5: Model Evaluation**

We assess the trained model's performance on the testing set to measure its accuracy in correctly attributing code snippets to authors. The accuracy score, computed using the accuracy\_score function from sklearn.metrics, quantifies the model's ability to correctly identify code authors.

**STEP 6: Conclusion**

In our project, our focus was on developing a machine learning pipeline for identifying authors of code snippets. With combination of numeric, non-numeric, and textual features, and incorporating advanced techniques like hyperparameter tuning, we have built a model for author identification.

**2.DEEP NEURAL NETWORK(DNN):**

**STEP 1: Data Loading and Preprocessing:**

* We have loaded the dataset from the CSV file that consisting the extracted features using pandas.
* Processed numeric features, non-numeric features, Word Frequencies, and Character N-grams[14].
* Combined all features into a single DataFrame.
* Imputed missing values with the mean using SimpleImputer.
* We have normalized the features using StandardScaler.

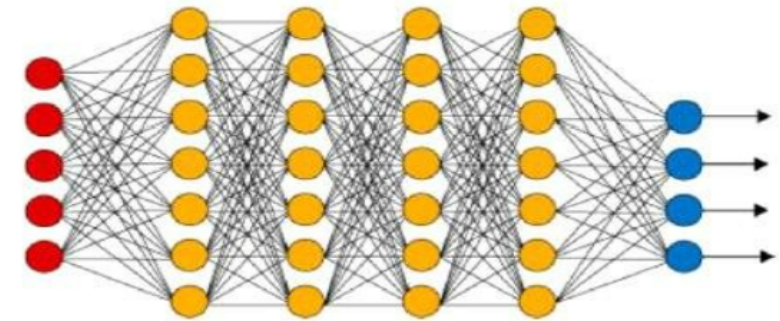
**STEP 2: Feature Conversion:**

* Converted the features and labels into PyTorch tensors using torch.tensor.
* Converted labels using LabelEncoder to encode categorical labels as integers.

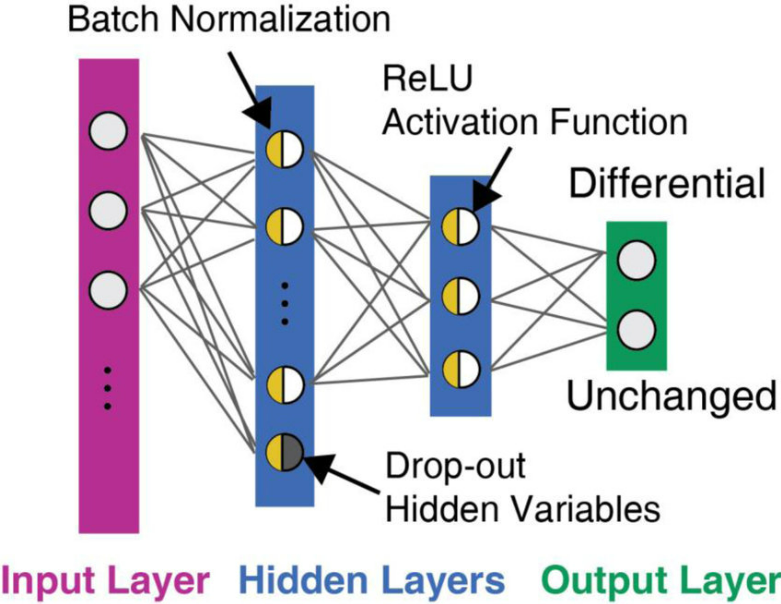
**STEP 3: Train-Test Split:**

* Spliting the dataset into training and testing sets using train\_test\_split.

**STEP 4: Define Deep Neural Network (DNN) Model:**

****

* We have defined a custom DNN model using PyTorch's nn.Module class.
* The model has two fully connected layers with a ReLU activation function in between.
* **nn.Linear:** Creates fully connected layers that perform mathematical transformations on the input data.
* **nn.ReLU:** Represents the Rectified Linear Unit activation function, adding non-linearity to the model.



* The forward method defines how data flows through the layers during inference:
* The input data is passed through the first fully connected layer (self.fc1), which transforms the data based on the specified weights and biases.
* The output of the first layer is passed through the ReLU activation function (self.relu), which introduces non-linearity by replacing negative values with zeros.
* The result is then passed through the second fully connected layer (self.fc2), which produces the final output. This output will be used for making predictions.
* This DNN architecture allows the model to capture intricate relationships in the data, enabling it to learn and predict authors based on the provided features.

**STEP 5: Hyperparameters and Training Setup:**

* We have set the hyperparameters like input\_size, hidden\_size, num\_classes, num\_epochs, and learning\_rate.
* Initialized the DNN model, defined the loss function (CrossEntropyLoss), and the optimizer (Adam).

**Defining the Loss Function (CrossEntropyLoss):**

**criterion = nn.CrossEntropyLoss() :** creates an instance of the Cross-Entropy loss function. Cross-Entropy loss is commonly used for multi-class classification problems.

**Defining the Optimizer (Adam):**

**optimizer = torch.optim.Adam(model.parameters(), lr=learning\_rate)** creates an instance of the Adam optimizer. It's responsible for updating the model's parameters based on the computed gradients during backpropagation.

model.parameters() provides the parameters (weights and biases) of the model that need to be optimized.

**lr=learning\_rate** specifies the learning rate, which controls the step size for parameter updates.

**Step 6 : Training Loop:**

* We have train the DNN model using a loop over a specified number of epochs.
* Zero the gradients, compute the model's outputs, calculate the loss, backpropagate, and update the model's parameters using the optimizer.
* Before computing the gradients during the backward pass, it's essential to set the gradients to zero. This prevents gradients from accumulating across multiple iterations, ensuring that only the gradients of the current batch contribute to parameter updates.
* Printing the loss every few epochs.

**Step 7: Model Evaluation:**

* Evaluated the trained model on the test set.
* Switch the model to evaluation mode and used torch.no\_grad() to disable gradient calculation.
* with torch.no\_grad(): creates a context where gradient calculation is disabled. This reduces memory usage and speeds up computation because gradients are not needed during evaluation and prediction.
* Computed the accuracy of the model's predictions using accuracy\_score.

**Step 8 : Hyperparameter Tuning with Grid Search:**

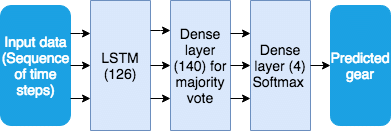
* Defined a parameter grid param\_grid for hyperparameters.
* Created an MLPClassifier model.
* MLPClassifier stands for Multi-Layer Perceptron Classifier. It is a type of artificial neural network model designed for classification tasks. The Multi-Layer Perceptron (MLP) is a fundamental building block of deep learning and is widely used for various machine learning and pattern recognition tasks.
* Used GridSearchCV to perform a grid search over different hyperparameter combinations.
* Found the best hyperparameters and the corresponding model.

**Step 9 : Model Evaluation with Tuned Hyperparameters:**

* Evaluating the best model with tuned hyperparameters on the test set.
* Calculated and print the accuracy of the best model.

**3. Long Short-Term Memory (LSTM):**

LSTM is a type of recurrent neural network (RNN) architecture designed to address the vanishing gradient problem and capture long-range dependencies in sequential data[12]. It uses a more complex structure in its hidden units, allowing it to remember information over longer sequences without the vanishing gradient problem.



**Step 1: Data Loading and Preprocessing**

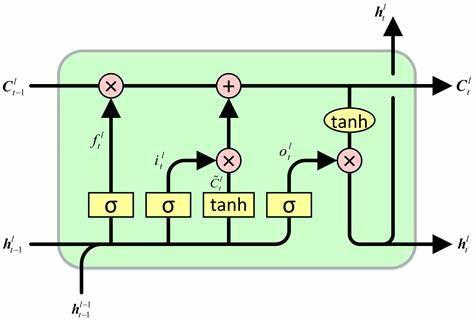
* Loaded the dataset from a CSV file of extracted features using pandas.
* Processed numeric features, non-numeric features.
* Combined all features into a single DataFrame.
* Normalized the features using StandardScaler.

**Step 2: Feature Conversion and Train-Test Split**

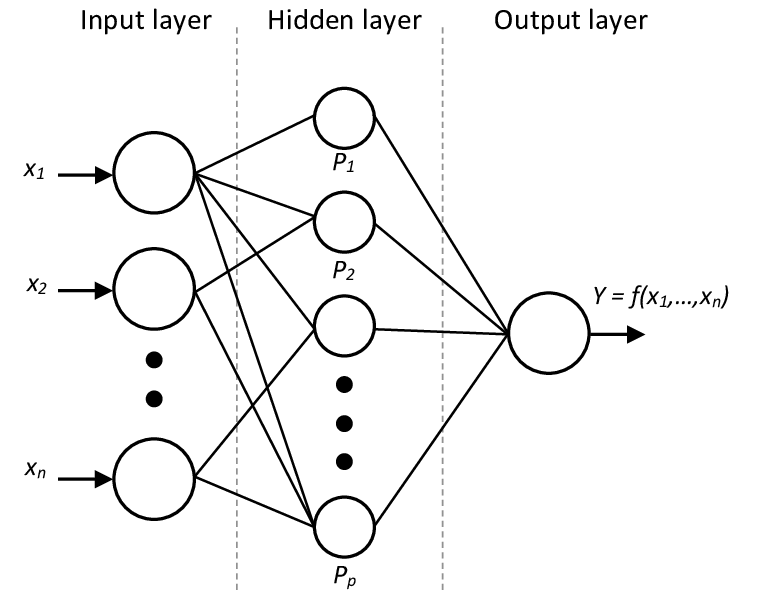
* Converted the features and labels into PyTorch tensors using torch.tensor.
* Converted labels using LabelEncoder to encode categorical labels as integers.
* Splited the dataset into training and testing sets using train\_test\_split.

**Step 3: Define LSTM Model Class(LSTM) Model:**

* **Sequential():** Defines a sequential model from keras module, used to create a linear stack of layers, where each layer has exactly one input tensor and one output tensor. The model allows to build a neural network by stacking layers linearly.
* We have defined the LSTM layer from keras.layers. The **units** parameter specifies the number of neurons in the layer. The **input\_shape** parameter defines the shape of your input data for this layer. Added the LSTM layer to the sequential model.
* LSTM cells maintain two types of internal states: the hidden state (h) and the cell state (c).
* The hidden state captures the relevant information and context of the sequence processed so far.
* The cell state stores and carries information across time steps.
* The output corresponding to the last time step is used for classification



* Lstm layer is used for processing sequential data and for capturing long-range dependencies.
* We have defined the fully connected Dense layer from keras module and added it to sequential model**,** the **units** parameter specifies the number of neurons and the **softmax** activation function which is common in multi-class classification to produce class probabilities.
* Dense layer is used to learn complex relationship between features.



**Step 4: Hyperparameters and Training Setup:**

Hyperparameters defining, tuning and learning of LSTM is similar to DNN.

**Step 5: Training Loop:**

* Since the LSTM layer expects a 3D input shape, the **X\_train** data is reshaped to **(samples, time steps, features)**, where **sample** is the number of training samples, **time steps** is set to 1 as the data is not time-series, and **feature** is the number of features in your data.
* Performed validation with 10% of the training data.

**Step 6: Model Evaluation**

* Model evaluation is an essential step to assess how well the trained LSTM model performs on unseen data.
* The testing data **X\_test** is reshaped to match the format required by the LSTM layer. It's reshaped from a 2D matrix to a 3D tensor with dimensions **(number\_of\_samples, time\_steps, number\_of\_features)**. This matches the format of the training data that the model was trained on.
* The model.evaluate() function is used to evaluate the model's performance on the testing data.
* It gives the loss and acuracy of the model along with classification report.

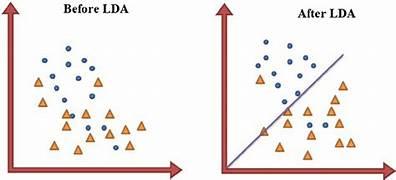
**Step 7: Conclusion**

This model demonstrates the process of loading, preprocessing, building, training, and evaluating a neural network model for multi-class classification. The inclusion of an LSTM layer allows the model to capture sequential dependencies, making it suitable for tasks involving sequential data like time series or text data. Through this model, we’ve gained insights into how to structure and train neural networks using Keras, paving the way for further exploration and optimization in future projects.

**4.Multivariate Discriminate analysis(MDA):**

The goal of LDA is to maximize the separation between classes while minimizing the variation within each class. LDA achieves this by calculating the mean vector and covariance matrix for each class.

It then calculates the "between-class scatter matrix" and the "within-class scatter matrix[4]. "The "between-class scatter matrix" measures the spread between the class means, and the "within-class scatter matrix" measures the spread within each class.LDA then finds a projection vector that maximizes the ratio of the determinant of the between-class scatter matrix to the determinant of the within-class scatter matrix. This projection aims to maximize the separation between classes.The projection vector is used to transform the original feature space into a lower-dimensional space, where classes are well-separated.



**Step 1: Data Loading and Preprocessing**

We start by loading the dataset from a CSV file using the pandas library. This dataset contains various code features and information about the authors who contributed them.

**Step 2: Preprocessing Numeric Features**

We preprocess the numeric features like Code Complexity, Cyclomatic Complexity, Lines of Code, etc.

Missing values are handled using SimpleImputer to fill with the mean of each column.

**Step 3: Preprocessing Non-Numeric Features**

We preprocess non-numeric features like Lexical Features, Syntactic Features, Semantic Features, Layout Features, Word Frequencies, and Character N-grams[14].

For list-like features, we calculate the mean value to represent each feature.

**Step 4: Combining Preprocessed Features**

We concatenate the preprocessed numeric and non-numeric features to create the final feature matrix.

**Step 5: Splitting Data for Training and Testing**

The dataset is divided into a training set (80%) and a testing set (20%) using the train\_test\_split function from sklearn.model\_selection.

**Step 6: Standardizing Features**

We standardize the features by using StandardScaler to ensure they are on the same scale.

**Step 7: Training the Linear Discriminant Analysis (LDA) Model**

We initialize and train the LinearDiscriminantAnalysis (LDA) model using the training data.

**Step 8: Making Predictions and Model Evaluation**

We use the trained LDA model to make predictions on the standardized test set.

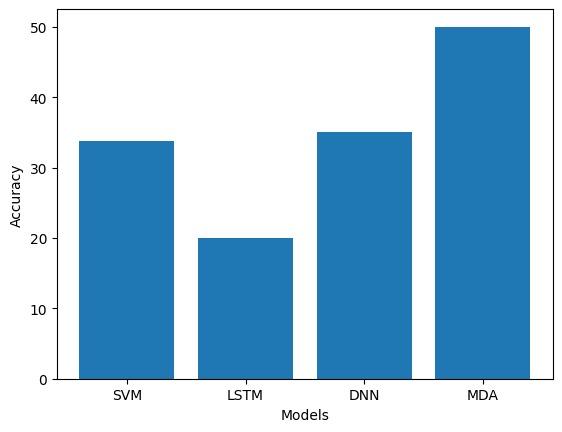
We calculate the accuracy of the model by comparing predicted labels to the actual labels.

We generate a classification report that provides more detailed metrics on the model's performance, including precision, recall, and F1-score for each class (author).

**Step 9: Conclusion**

This project focuses on using Linear Discriminant Analysis (LDA) to identify code authors based on various code features.

**Analysis and Results**



We have worked on LSTM, DNN, MDA and SVM. The accuracies of both the LSTM and DNN models range from 15% to 25% and 35% to 40%. This range indicates that the models are performing quite poorly on the given task. Accuracies in this range suggest that the models are not effectively capturing the patterns and relationships present in the data. It's possible that these models are struggling to learn meaningful representations from the data, leading to random or incorrect predictions[2]. The SVM and MDA models, on the other hand, has an accuracy of 33.75% and 50%. The accuracies of SVM, DNN and MDA are notably higher compared to the LSTM model. An accuracy of 50% suggests that the MDA model is performing better than random guessing, but it might still not be performing well enough for certain applications. Depending on the context of the problem, this accuracy could be considered low or moderate.

**Future Work:**

So far we have made the models to predict the actual author of a code. It will work with code written by single author. But in future we can add multiple code snippets in a single code of different authors, such that the model has to predict how many authors involved in the code.

**References:**

[1] Saed Alrabaee, Paria Shirani, Mourad Debbabi, and Lingyu Wang. 2017. On the feasibility of malware authorship attribution. arXiv:1701.02711.

[2] Steven Burrows, Alexandra L. Uitdenbogerd, and Andrew Turpin. 2014. Comparing techniques for authorship attribution of source code. Software: Practice and Experience 44, 1 (Aug. 2014), 1–32.

[3] Muqaddas Gull, Tehseen Zia, and Muhammad Ilyas. 2017. Source code author attribution using author’s programming style and code smells. International Journal of Intelligent Systems and Applications 9, 5, 27.

[4] Jane Huffman Hayes. 2008. Authorship attribution: A principal component and linear discriminant analysis of the consistent programmer hypothesis. International Journal on Computers and Their Applications 15, 2 (2008), 79–99.

[5] Paul W. Oman and Curt R. Cook. 1989. Programming style authorship analysis. In Proceedings of the 17th ACM Annual Computer Science Conference (CSC’89). 320–326.

[6] Philip J. Sallis, Asbjorn Aakjaer, and Stephen G. Macdonell. 1996. Software forensics: Old methods for a new science. In Proceedings of the International Conference Software Engineering: Education and Practice. IEEE, Los Alamitos, CA, 481–485.

[7] Eugene H. Spafford and Stephen A. Weeber. 1993. Software forensics: Can we track code to its authors? Computers and Security 12, 6 (1993), 585–595.

[8] VAIBHAVI KALGUTKAR, RATINDER KAUR, HUGO GONZALEZ, NATALIA STAKHANOVA, and ALINA MATYUKHINA, University of New Brunswick. Code Authorship Attribution: Methods and Challenges.

[9] Abdul Mannan Omi 1, Monir Hossain 2, Md Nahidul Islam 3 and Tanni Mittra 4 Department of Computer Science and Engineering, East West University Dhaka, Bangladesh. Multiple Authors Identification from Source Code Using Deep Learning Model.

[10] Uri Alon, Shaked Brody, Omer Levy, Eran Yahav. CODE2SEQ: GENERATING SEQUENCES FROM STRUCTURED REPRESENTATIONS OF CODE.

[11] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, pages 249–256, 2010.

[12] Sepp Hochreiter and Jurgen Schmidhuber. Long short-term memory. ̈ Neural Comput., 9(8):1735–1780, November 1997. ISSN 0899-7667. doi:10.1162/neco.1997.9.8.1735. URL <http://dx.doi.org/10.1162/neco.1997.9.8.1735>.

[13] Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code genera-

tion and semantic parsing. In Proceedings of the 55th Annual Meeting of the Association for

Computational Linguistics (Volume 1: Long Papers), pages 1139–1149. Association for Com-

putational Linguistics, 2017. doi:10.18653/v1/P17-1105. URL <http://www.aclweb.org/anthology/P17-1105>.

[14] S. Burrows and S. M. M. Tahaghoghi. 2007. Source code authorship attribution using N-grams. In Proceedings of the 12th Australasian Document Computing Symposium. 32–39.