Jurnal Informatika Ekonomi Bisnis

http://www.infeb.org

2024 Vol. 6 Iss. 4 Hal: 792-798 e-ISSN: 2714-8491

Comparative Analysis of Deep Learning Architectures for Predicting Software Quality Metrics in Behavior-Driven and Test-Driven Development Approaches

Gregorius Airlangga[1]✉

[1]Atma Jaya Catholic University of Indonesia

gregorius.airlangga@atmajaya.ac.id

Abstract

The impact of software development methodologies on quality metrics is a crucial area of study in empirical software engineering. This research evaluates the performance of three deep learning architectures: Multi-Layer Perceptron (MLP), Convolutional Neural Network (CNN), and Long Short-Term Memory (LSTM), in predicting key software quality indicators, including maintainability index, test coverage, and code complexity, for projects developed using Behavior-Driven Development (BDD) and Test-Driven Development (TDD) approaches. Using a static tabular dataset containing software quality metrics, the models are evaluated based on Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), Mean Absolute Percentage Error (MAPE), and the R^2 coefficient. The MLP achieves the best performance, with the lowest RMSE (6.41) and MAE (6.34) and the highest R^2 value (−4.21), demonstrating its suitability for tabular data. The CNN performs moderately, while the LSTM underperforms due to its reliance on temporal dependencies absent from the dataset. These results emphasize the need for careful architectural alignment with dataset characteristics. The findings contribute to understanding the predictive power of deep learning models in software quality analysis and highlight the potential of MLP as a robust tool for such predictions. Future work can explore hybrid models and domain-specific feature engineering to enhance prediction accuracy.

Keywords: Deep Learning Architectures, Tabular Data, Predictive Modeling, Multi-Layer Perceptron (MLP), Comparative Analysis.

1. Introduction

The ever-evolving landscape of software engineering demands methodologies that not only ensure efficient development cycles but also enhance software quality [1]. [2], [3]. In this pursuit, Behavior-Driven Development (BDD) and Test-Driven Development (TDD) have emerged as two of the most prominent methodologies in contemporary software engineering practices [4]. While both methods advocate a shift-left approach to software quality assurance, they fundamentally differ in their principles and processes. BDD emphasizes collaboration among stakeholders through executable specifications, whereas TDD focuses on rigorous unit testing driven by pre-written test cases [5]. These methodologies have garnered significant attention in the software engineering community, particularly for their potential to enhance maintainability, improve test coverage, and reduce code complexity [6]. However, their relative effectiveness and the contexts in which one may outperform the other remain areas of active research [7].

Existing studies have explored the impacts of BDD and TDD on various software quality metrics. For instance, a research conducted a comparative analysis of BDD and TDD practices in large-scale projects, finding that BDD led to better stakeholder communication but at the cost of increased initial development time [8]. Similarly, another research analyzed the maintainability and test coverage of software modules developed using TDD and concluded that TDD enhanced code reliability but often resulted in higher code complexity due to the extensive focus on unit tests [9]. Despite these insights, the methodologies' effects on holistic quality indicators, such as maintainability indices, code complexity, and test coverage, remain underexplored, particularly in a controlled, data-driven context. Furthermore, the integration of advanced computational models for analysis in these contexts remains limited, leaving an opportunity to deepen understanding through modern techniques [10].

The urgency of this research lies in addressing the growing need for empirical evidence that supports practitioners in selecting the most suitable methodology for their specific development contexts [11]. As the software industry increasingly adopts agile practices, choosing between BDD and TDD has become critical, particularly for projects requiring rapid development and high-quality outcomes [4]. Current research often provides qualitative assessments or limited quantitative analyses, leaving significant gaps in understanding the methodologies' comprehensive impacts. This research aims to fill these gaps by leveraging deep learning models, such as Long Short-Term Memory networks

Accepted: 21-11-2024 | Revision: 04-12-2024 | Publication: 31-12-2024 | doi: 10.37034/infeb.v6i4.1045

(LSTM), Convolutional Neural Networks (CNN), and Multi-Layer Perceptrons (MLP), to analyze and predict software quality metrics [12]. These models are particularly suited for capturing complex patterns and relationships within the dataset, enabling more nuanced insights into the methodologies' effectiveness [13].

The primary goal of this study is to investigate the relative effectiveness of BDD and TDD in influencing software quality metrics, including maintainability, test coverage, and code complexity [14]. By employing deep learning techniques, this study not only compares the methodologies but also demonstrates the potential of advanced computational models in software engineering research [15]. The contribution of this research lies in introducing predictive capabilities to comparative methodology studies, providing actionable insights into quality outcomes based on development practices. Furthermore, the use of LSTM, CNN, and MLP models ensures that both sequential patterns and static relationships in the data are effectively captured, leading to robust and reliable predictions [16].

The remaining structure of this article is as follows: The next section elaborates on the materials and methods used in this study, detailing the dataset characteristics, preprocessing steps, and the architectures of the deep learning models applied. Subsequently, the results and discussion section present a comparative analysis of the performance metrics across different models, highlighting key trends and implications for software development methodologies. Finally, the conclusion summarizes the findings, outlines directions for future research, and emphasizes the practical relevance of this study to the software engineering community.

2. Research Method

The methodology of this study is structured into three main components: dataset description, preprocessing steps, and the architecture of deep learning models employed. This section provides a comprehensive explanation of each component, ensuring a robust and reproducible framework.

2.1. Dataset Description

The dataset utilized in this study provides a comprehensive basis for comparing the Behavior-Driven Development (BDD) and Test-Driven Development (TDD) methodologies and can be downloaded from [17]. Each observation in the dataset represents a software project or module and can be expressed mathematically as a tuple as presented in Equation 1.

$$[X_i = \{x_{i1}, x_{i2}, \dots, x_{if}\}, \quad \text{for } i = 1,2, \dots, n] \tag{1}$$

where (n) is the total number of observations, and (f) denotes the number of features. These features include essential software quality metrics such as the development methodology $((M_i))$, maintainability index $((x_{i1}))$, test coverage $((x_{i2}))$, code complexity $((x_{i3}))$, and development time $((x_{i4}))$. The dataset also contains the target variable (Q_i), which represents the overall software quality. The development methodology feature, (M_i), is a categorical variable defined as follows. If the project follows the BDD methodology, $(M_i = 1)$. If the project follows the TDD methodology, $(M_i = 0)$. The maintainability index, (x_{i1}), is a numerical score that quantifies the ease of maintaining the software. It is computed using Equation 2.

$$\begin{aligned} sx_{i1} &= 171 - 5.2 \cdot \log_2(\text{Cyclomatic Complexity}) \\ &\quad - 0.23 \cdot \text{Lines of Code} - 16.2 \\ &\quad \cdot \log_2(\text{Halstead Volume}) \end{aligned} \tag{2}$$

where the cyclomatic complexity, lines of code, and Halstead volume are derived from the source code metrics of the project. A higher value of (x_{i1}) indicates better maintainability. Test coverage, denoted as (x_{i2}), is a percentage that represents the proportion of code covered by automated tests. It is calculated as presented in Equation 3.

$$x_{i2} = \frac{\text{Lines of Code Tested}}{\text{Total Lines of Code}} \times 100 \tag{3}$$

Code complexity, represented as (x_{i3}), is measured using cyclomatic complexity, which quantifies the number of independent paths through the program's control flow graph. It is defined as presented in Equation 4.

$$x_{i3} = e - n + 2p \tag{4}$$

where (e) is the number of edges in the control flow graph, (n) is the number of nodes in the graph, and (p) is the number of connected components in the graph. Development time, (x_{i4}), is a continuous variable that records the total time required to complete the project, measured in hours. This is computed based on timestamps associated with key milestones during the development lifecycle. The target variable, software quality $((Q_i))$, is a composite metric that integrates maintainability index $((x_{i1}))$, test coverage $((x_{i2}))$, and code complexity $((x_{i3}))$ through a weighted linear combination as presented in Equation 5.

$$Q_i = \alpha \cdot x_{i1} + \beta \cdot x_{i2} - \gamma \cdot x_{i3} \tag{5}$$

where (α), (β), and (γ) represent the weights assigned to maintainability, test coverage, and code complexity, respectively. These weights are determined based on the industry's best practices and expert recommendations, ensuring that (Q_i) reflects an accurate and holistic measure of software quality. To summarize, the dataset

can be mathematically represented as presented in Equation 6.

$$X \in R^{n \times f}, \quad y \in R^{n} \tag{6}$$

where (X) is the feature matrix containing (n) observations and (f) features, and (y) is the vector of target quality scores for each project. The data set structure ensures compatibility with advanced deep learning models, facilitating a robust analysis of the impacts of BDD and TDD methodologies on software quality.

2.2. Preprocessing

This pipeline consists of several key stages: data cleaning, feature scaling, categorical encoding, data splitting, and sequence transformation.

2.2.1. Data Cleaning

The raw dataset often contains missing values and outliers, which can adversely affect the training of deep learning models. These issues are addressed through imputation and outlier detection techniques. Missing values in numerical features such as maintainability index $((M))$, test coverage $((C))$, code complexity $((X))$, and development time $((T))$ are imputed using the median value of each feature. Mathematically, the imputed value for a feature (x_{ij}) in row (i) is given in Equation 7.

$$x'_{ij} = \begin{matrix} x_{ij} & \text{if } x_{ij} \neq \text{NaN} \\ \text{median}(x_j) & \text{if } x_{ij} = \text{NaN}. \end{matrix} \tag{7}$$

where $(\text{median}(x_j))$ represents the median of feature (x_j) across all non-missing observations. Outlier detection is performed using the Interquartile Range (IQR) method. The IQR is defined as $\text{IQR} = Q_3 - Q_1$, where (Q_1) and (Q_3) are the 25th and 75th percentiles of the feature (x_j), respectively. Any value (x_{ij}) that satisfies: $x_{ij} < Q_1 - 1.5 \cdot \text{IQR}$ or $x_{ij} > Q_3 + 1.5 \cdot$ IQR is considered an outlier and is capped at the nearest 5th or 95th percentile, respectively in Equation 8 and Equation 9.

$$x'_{ij} \text{ if } x_{ij} \neq \text{NaN} \tag{8}$$

$$\text{median}(x_j) \text{ if } x_{ij} = \text{NaN} \tag{9}$$

Here, $(P_5(x_j))$ and $(P_{95}(x_j))$ denote the 5th and 95th percentiles of feature (x_j), respectively.

2.2.2. Feature Scaling

To standardize the range of numerical features, Min-Max normalization is applied. Each feature (x_{ij}) is scaled to the range $([0,1])$ using the formula:

$x'_{ij} = \frac{x_{ij} - x_{min}}{x_{max} - x_{min}}$, where x_{min} and (x_{max}) are the minimum and maximum values of the feature (x_j), respectively, across all observations. The scaled feature (x'_{ij}) ensures that all inputs to the model have uniform ranges, reducing the risk of dominance by features with larger magnitudes.

2.2.3. Categorical Encoding

The development methodology column, denoted as (M_i), is a categorical variable representing whether the project follows the Behavior-Driven Development (BDD) or Test-Driven Development (TDD) methodology. This column is one-hot encoded into two binary variables, such that $\text{BDD} = [1,0]$, $\text{TDD} = [0,1]$. For each observation (i), the encoded vector is represented as $[1,0]$ if the methodology is BDD and $[0,1]$ if the methodology is TDD.

2.2.4 Data Splitting

The dataset is partitioned into training and testing sets using an 80%-20% split. Let (**X**) and (**y**) represent the feature matrix and the target vector, respectively. The split is performed such that $\boldsymbol{X_{train}}$, $\boldsymbol{y_{train}}$ in $R^{0.8n \times f}$, $\boldsymbol{X}_{\text{ts}}, \boldsymbol{y}_{\text{ts}} \in R^{0.2n \times f}$. During training, the training set is further divided into (k)-folds for cross-validation. For $(k = 10)$, the data is split into 10 mutually exclusive subsets of approximately equal size. For each fold (j), the validation set consists of the (j)-th subset, and the training set consists of the remaining $(k - 1)$ subsets. The objective is to minimize the generalization error by ensuring that each observation is used for validation exactly once.

2.2.5. Sequence Transformation

For sequential models, such as Long Short-Term Memory (LSTM) networks and Convolutional Neural Networks (CNNs), the data must be reshaped into a three-dimensional tensor. The feature matrix ($\boldsymbol{X} \in R^{n \times f}$) is transformed into $X' \in R^{n \times t \times f}, where(n)$ is the number of samples, $(t = 1)$ represents the time step (since the data is static for each project), and (f) is the number of features. Each reshaped sample can be expressed as presented in Equation 10.

$$X_i' = \left[x_{i1}, x_{i2}, \ldots, x_{if}\right]^{\top} \in R^{1 \times f} \quad (10)$$

This transformation enables sequential models to process the input as a time series, even if each sample represents a static observation. The preprocessing steps described above ensure that the dataset is clean, consistent, and appropriately formatted for deep learning models. By addressing missing values, outliers, and scaling inconsistencies, and transforming the data for sequence-based architectures, the pipeline ensures that the input data optimally supports the learning process.

2.3. Deep Learning Model Architectures

In this study, three deep learning architectures are utilized: Multi-Layer Perceptron (MLP), Long Short-Term Memory (LSTM), and Convolutional Neural Network (CNN). Each architecture is specifically designed to leverage unique characteristics of the dataset and capture different aspects of its structure. The MLP model is a fully connected feedforward network optimized for processing tabular data. The MLP model is a feature vector ($X_i \in R^f$), where (f) is the number of input features for each sample. The input layer directly accepts the (f)-dimensional feature vector (X_i). This is followed by two hidden layers, each represented as a dense transformation $h_1 = \sigma(W_1 X_i + b_1), \quad h_2 = \sigma(W_2 h_1 + b_2)$, where ($W_1 \in R^{256 \times f}$), ($W_2 \in R^{128 \times 256}$) are weight matrices, ($b_1$) and ($b_2$) are the corresponding bias terms, and ($\sigma(\cdot)$) is the ReLU activation function defined as $\sigma(z) = \max(0, z)$

To prevent overfitting, Dropout regularization with a dropout probability ($p = 0.3$) is applied to each hidden layer, effectively setting a fraction of the layer's activations to zero during training. The output layer is a single neuron with a linear activation function, producing the final prediction ($\hat{y}_i$) for the (i)-th sample $\hat{y}_i = w_3^{\top} h_2 + b_3$, where ($w_3 \in R^{128}$) is the weight vector and (b_3) is the bias term. The model is optimized using the Mean Squared Error (MSE) loss function $\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$, where ($y_i$) is the true target value for the (i)-th sample, and ($\hat{y}_i$) is the predicted value.

Furthermore, the LSTM model is designed to capture temporal dependencies and long-range interactions, even though the dataset represents static data. The input to the LSTM model is a sequence tensor ($X \in R^{n \times t \times f}$), where ($n$) is the number of samples, (t) is the time step (here ($t = 1$)), and (f) is the number of features. The LSTM model consists of two stacked layers. Each LSTM layer processes the input tensor sequentially and produces an output (h_t) at each time step (t). For a given LSTM layer, the computations are as presented in the Equation 11 to Equation 16.

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \quad (11)$$

$$f_t = \sigma\left(W_f x_t + U_f h_{t-1} + b_f\right) \quad (12)$$

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \quad (13)$$

$$g_t = \tanh\left(W_g x_t + U_g h_{t-1} + b_g\right) \quad (14)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \quad (15)$$

$$h_t = o_t \odot \tanh(c_t) \quad (16)$$

where (i_t, f_t, o_t)represent the input, forget, and output gates, respectively, (c_t) is the cell state, and (h_t) is the hidden state. ($\sigma(\cdot)$) is the sigmoid activation, and ($\odot$) denotes element-wise multiplication. The first LSTM layer has 128 units, while the second layer has 64 units. Both layers use ReLU activation for their outputs, followed by Dropout regularization with ($p = 0.3$). The final output is passed to a dense layer with a single neuron and a linear activation function to produce the prediction ($\hat{y}_i$). The LSTM model is trained using the Adam optimizer with a learning rate of 0.001, and the MSE loss function is minimized.

The CNN model is designed to extract spatial patterns from the dataset, treating the input as a one-dimensional sequence. The input to the CNN is a tensor ($X \in R^{n \times t \times f}$), like the LSTM. The first layer is a one-dimensional convolutional layer that applies ($k = 64$) filters, each with a kernel size of 3. For a given input sequence ($X_i \in R^{t \times f}$), the convolutional operation produces feature maps (F_j) as $F_j = \sigma\left(W_j * X_i + b_j\right)$, where ($*$) denotes the convolution operation, ($W_j \in R^{3 \times f}$) is the filter for the (j)-th feature map, and (b_j) is the bias term. The activation function ($\sigma(\cdot)$) is ReLU. A MaxPooling1D layer with a pool size of 2 is applied to downsample the feature maps, reducing their dimensionality by half. The resulting tensor is flattened into a one-dimensional vector $v = \text{Flatten}(F)$, where (F) represents the pooled feature maps. This flattened vector is passed through a dense layer with 128 units, ReLU activation, and Dropout regularization (($p = 0.3$)), followed by a final dense layer with a single neuron and a linear activation function to produce the prediction ($\hat{y}_i$). The CNN model is optimized using the Adam optimizer, with the MSE loss function serving as the objective to minimize. These three architectures: MLP, LSTM, and CNN are designed to leverage different structural characteristics of the data. The MLP captures static relationships, the LSTM explores sequential dependencies, and the CNN identifies spatial patterns, ensuring a comprehensive analysis of the dataset. All models are trained using backpropagation with the Adam optimizer and evaluated using the Mean Squared Error loss function.

2.4. Model Training and Evaluation

The training of all models is conducted using a batch size of 32 to ensure efficient computation and stable gradient updates. An early stopping mechanism is employed to mitigate overfitting, halting training when the validation loss does not improve for 10 consecutive epochs. This is implemented using a patience parameter \(p = 10\), ensuring that the best weights are restored after training concludes. Additionally, the learning rate is dynamically adjusted using the ReduceLROnPlateau callback, which reduces the learning rate by a factor of 0.5 whenever the validation loss plateaus for five epochs. This mechanism ensures that the optimization process converges smoothly and avoids overshooting the minima. The performance of each trained model is evaluated using several metrics that capture different aspects of predictive accuracy and error characteristics. These metrics include Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), Mean Absolute Percentage Error (MAPE), and the coefficient of determination ((R^2)). The Root Mean Squared Error (RMSE) is defined as presented in Equation 17.

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}, \tag{17}$$

where (y_i) represents the true target value for the (i)-th sample, $(\hat{y}_i)$ is the predicted target value, and (n) is the total number of samples. RMSE emphasizes larger errors due to the quadratic term, making it sensitive to outliers. The Mean Absolute Error (MAE) is calculated as presented in Equation 18.

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|, \tag{18}$$

which measures the average magnitude of errors in predictions without considering their direction. MAE provides an intuitive interpretation of the average prediction error in the same units as the target variable. The Mean Absolute Percentage Error (MAPE) quantifies the prediction error as a percentage of the true value as presented in Equation 19.

$$\text{MAPE} = \frac{100}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \tag{19}$$

where $\left(\left|\frac{y_i - \hat{y}_i}{y_i}\right|\right)$ represents the relative error for each sample. MAPE is particularly useful for understanding the error in the context of the target variable's scale. The coefficient of determination $((R^2))$ evaluates the proportion of variance in the target variable that is captured by the model. It is defined as presented in Equation 20.

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}, \tag{20}$$

where $(\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i)$ is the mean of the true target values. The numerator $(\sum_{i=1}^{n}(y_i - \hat{y}_i)^2)$ represents the residual sum of squares (unexplained variance), while the denominator $(\sum_{i=1}^{n}(y_i - \bar{y})^2)$ is the total sum of squares (total variance). An (R^2) value close to 1 indicates that the model explains most of the variability in the target variable. To provide a comprehensive evaluation metric, a custom scoring function integrates RMSE, MAE, and (R^2). The custom score is defined as presented in Equation 21.

$$\text{Score} = R^2 - \frac{\text{RMSE} + \text{MAE}}{2} \tag{21}$$

This function balances the goodness of fit (measured by (R^2) against the magnitude of errors (measured by RMSE and MAE). By penalizing higher errors, the custom score ensures that models are evaluated holistically, considering both accuracy and robustness. This rigorous training and evaluation framework ensures that the models not only achieve high predictive performance but also generalize well to unseen data. By leveraging multiple metrics and a custom scoring function, the approach provides a nuanced understanding of each model's strengths and limitations. These insights are critical for selecting the most effective architecture for the dataset.

3. Results and Discussion

The performance of the three deep learning models: Convolutional Neural Network (CNN), Long Short-Term Memory (LSTM), and Multi-Layer Perceptron (MLP) were evaluated across several metrics, including Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), Mean Absolute Percentage Error (MAPE), (R^2) (coefficient of determination), and a custom score that integrates RMSE, MAE, and (R^2). These metrics collectively provide a nuanced view of the predictive accuracy, robustness, and generalization ability of each model as presented in the Table 1 and Figure 1.

Table 1. Deep Learning Performance

Model	RMSE	MAE	MAPE	R^2	Custom Score
CNN	7.83	7.27	27.79%	-6.88	-14.43
LSTM	8.81	8.46	32.83%	-17.71	-26.34
MLP	6.41	6.34	25.63%	-4.21	-10.59

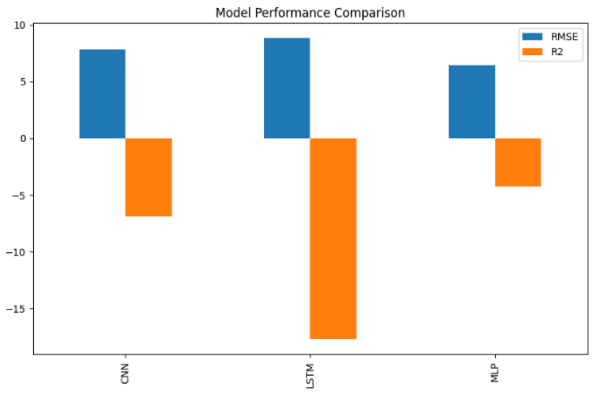

Figure 1. Deep Learning Model Comparison

The MLP model demonstrates the best overall performance across all metrics. It achieves the lowest RMSE of (6.41), indicating that its predictions are closest to the true values in terms of squared error magnitude. Additionally, its MAE of (6.34) reflects minimal average deviation from the true values, and its MAPE of (25.63%) signifies the smallest percentage error, which is crucial in understanding the relative accuracy of predictions. The (R^2) score for MLP is (-4.21), which, while negative, still suggests it explains more variance in the data compared to CNN and LSTM. The custom score of (-10.59), which balances (R^2), RMSE, and MAE, highlights the MLP as the most robust model among the three.

The CNN model performs moderately well, sitting between the MLP and LSTM in terms of accuracy. Its RMSE of (7.83) and MAE of (7.27) are higher than those of the MLP, indicating less accurate predictions. The CNN's MAPE of (27.79\%) is slightly worse than the MLP, demonstrating higher relative errors in its predictions. The (R^2) value of (-6.88) shows that the CNN explains less variance in the data than the MLP. Its custom score of (-14.43) reinforces this relative underperformance. However, CNN's ability to extract spatial or feature-level patterns likely contributes to its intermediate performance, making it a viable option when MLP is unavailable.

The LSTM model performs the worst among the three architectures. Its RMSE of (8.81) and MAE of (8.46) indicate substantial errors in its predictions. With a MAPE of (32.83%), the LSTM produces the largest relative deviations from the true values. The (R^2) score of (-17.71) is significantly lower than the other models, highlighting its inability to explain the variance in the dataset. The custom score of (-26.34) underscores the compounded impact of these errors, rendering the LSTM unsuitable for this dataset. The poor performance of the LSTM can be attributed to the static nature of the data, which lacks the sequential dependencies that LSTMs are designed to leverage. This mismatch between the dataset structure and the model's architecture severely impacts its predictive capabilities.

The RMSE, which emphasizes larger errors due to its quadratic term, shows that the MLP minimizes significant deviations more effectively than the CNN and LSTM. The CNN's higher RMSE compared to the MLP indicates it struggles more with larger prediction errors, while the LSTM's even higher RMSE reveals its pronounced difficulty in aligning predictions with true values. The MAE, a measure of average absolute errors, aligns with the RMSE trends. The MLP's MAE of (6.34) confirms that it achieves the smallest average error. The CNN's MAE of (7.27) and the LSTM's MAE of (8.46) indicate progressively worse average prediction accuracy. The MAPE provides additional context by normalizing the absolute errors as a percentage of the true values. The MLP's MAPE of (25.63%) demonstrates its ability to generalize well, producing smaller relative errors. In contrast, the CNN's MAPE of (27.79%) and the LSTM's MAPE of (32.83%) reveal their poorer performance in handling relative deviations.

The (R^2) metric, which assesses the proportion of variance captured by the model, shows that the MLP explains the most variance, albeit with a negative value $((-4.21))$. The CNN's (R^2) of (-6.88) suggests it is less effective at capturing data variability, while the LSTM's (R^2) of (-17.71) confirms its inability to model the dataset effectively. The custom score, which combines (R^2), RMSE, and MAE, provides a holistic evaluation of each model. The MLP achieves the least negative score $((-10.59))$, confirming its superior balance of accuracy and robustness. The CNN follows with a score of (-14.43), and the LSTM performs the worst with a score of (-26.34). The results demonstrate that the MLP is the most suitable architecture for the given dataset, as it achieves the lowest errors and the highest relative accuracy. The CNN's moderate performance suggests that it can extract meaningful patterns, though it is outperformed by the MLP due to the dataset's tabular nature. The LSTM's poor performance highlights the importance of aligning the model architecture with the dataset characteristics. Since the data lacks sequential relationships, the LSTM's temporal modeling capabilities are redundant, leading to overparameterization and poor generalization.

4. Conclusion

This study evaluates the effectiveness of three machine learning architectures: Multi-Layer Perceptron (MLP), Convolutional Neural Network (CNN), and Long Short-Term Memory (LSTM) for detecting Alzheimer's Disease using a dataset encompassing demographic, lifestyle, medical, and cognitive features. Through 10-fold cross-validation, the CNN emerged as the most effective model, achieving the highest metrics (accuracy: 88.65%, F1-score: 88.62%) due to its ability to capture complex spatial patterns. The MLP demonstrated moderate performance (accuracy: 84.41%, F1-score: 84.17%), while LSTM, more suited for temporal data, struggled with the tabular nature of the

dataset, achieving lower metrics (accuracy: 75.57%, F1-score: 75.28%). These findings underscore the importance of aligning model architecture with data characteristics, with CNNs proving highly effective for complex feature interactions. This study highlights the potential of machine learning models as diagnostic tools in data-driven healthcare, suggesting future exploration of hybrid models, interpretability techniques, and validation on larger datasets to improve early detection and management of Alzheimer's Disease.

References

[1] Pargaonkar, S. (2023). Synergizing requirements engineering and quality assurance: A comprehensive exploration in software quality engineering. International Journal of Science and Research (IJSR), 12(8), 2003–2007.

[2] Al-Baik, O., Abu Alhija, M., Abdeljaber, H., & Ovais Ahmad, M. (2024). Organizational debt—Roadblock to agility in software engineering: Exploring an emerging concept and future research for software excellence. PLOS ONE, 19(11), e0308183. https://doi.org/10.1371/journal.pone.0308183

[3] Gupta, M. L., Puppala, R., Vadapalli, V. V., Gundu, H., & Karthikeyan, C. V. S. S. (2024). Continuous integration, delivery and deployment: A systematic review of approaches, tools, challenges and practices. In International Conference on Recent Trends in AI Enabled Technologies (pp. 76–89). Springer. https://doi.org/10.1007/978-3-031-59114-3_7

[4] Cui, J. (2024). A comparative study on the impact of test-driven development (TDD) and behavior-driven development (BDD) on enterprise software delivery effectiveness. arXiv preprint arXiv:2411.04141. https://doi.org/10.48550/arXiv.2411.04141

[5] Natarajan, T., & Pichai, S. (2024). Behaviour-driven development and metrics framework for enhanced agile practices in scrum teams. Information and Software Technology, 170, 107435. https://doi.org/10.1016/j.infsof.2024.107435

[6] Rahman, S., & Nadia, F. (2024). Pioneering testing technologies: Advancing software quality through innovative methodologies and frameworks. Journal of Artificial Intelligence and Machine Learning in Management, 8(2), 44–70.

[7] Yuan, X., & Tang, X. (2024). Relative effectiveness of morphological analysis training and context clue training on multidimensional vocabulary knowledge. The Journal of Genetic Psychology, 185(2), 77–90. https://doi.org/10.1080/00221325.2024.1234567

[8] Irshad, M., Britto, R., & Petersen, K. (2021). Adapting behavior-driven development (BDD) for large-scale software systems. Journal of Systems and Software, 177, 110944. https://doi.org/10.1016/j.jss.2021.110944

[9] Parsa, S., Zakeri-Nasrabadi, M., & Turhan, B. (2025). Testability-driven development: An improvement to the TDD efficiency. Computer Standards & Interfaces, 91, 103877. https://doi.org/10.1016/j.csi.2025.103877

[10] Razavi, S. (2021). Deep learning, explained: Fundamentals, explainability, and bridgeability to process-based modelling. Environmental Modelling & Software, 144, 105159. https://doi.org/10.1016/j.envsoft.2021.105159

[11] Thesing, T., Feldmann, C., & Burchardt, M. (2021). Agile versus waterfall project management: Decision model for selecting the appropriate approach to a project. Procedia Computer Science, 181, 746–756. https://doi.org/10.1016/j.procs.2021.12.094

[12] Ahmed, S. (2023). A software framework for predicting the maize yield using modified multi-layer perceptron. Sustainability, 15(4), 3017. https://doi.org/10.3390/su15043017

[13] Shu, X., & Ye, Y. (2023). Knowledge discovery: Methods from data mining and machine learning. Social Science Research, 110, 102817. https://doi.org/10.1016/j.ssresearch.2023.102817

[14] Smart, J. F., & Molak, J. (2023). BDD in action: Behavior-driven development for the whole software lifecycle. Simon and Schuster.

[15] Krzywanski, J., Sosnowski, M., Grabowska, K., Zylka, A., Lasek, L., & Kijo-Kleczkowska, A. (2024). Advanced computational methods for modeling, prediction and optimization—a review. Materials, 17(14), 3521. https://doi.org/10.3390/ma17143521

[16] Ahmed, S. F., Alam, M. S. B., Hassan, M., Rozbu, M. R., Ishtiak, T., Rafa, N., Mofijur, M., Shawkat Ali, A. B. M., & Gandomi, A. H. (2023). Deep learning modelling techniques: Current progress, applications, advantages, and challenges. Artificial Intelligence Review, 56(11), 13521–13617. https://doi.org/10.1007/s10462-023-10420-9

[17] Yogi. (2024). TDD and BDD comparison dataset. Kaggle. Retrieved November 19, 2024, from https://www.kaggle.com/datasets/yogi2727/tdd-and-bdd-comparison