Comparison of Various Machine Learning Methods for Implementing Wearable Stress Detection Systems

Aditya Gading Saputra

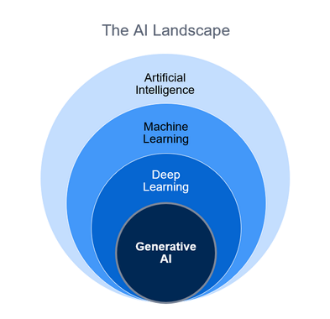
aDepartment of Robotics Engineering, Faculty of Electrical Engineering, Batam State Polytechnic, Riau Islands, Indonesia.

|  |  |  |
| --- | --- | --- |
| KEYWORDS |  | ABSTRACT |
| Machine Learning |  | This paper presents the results of a comparison between several machine learning methods for wearable stress detection, using data sourced from Kaggle. The primary objective of this research is to evaluate and compare the effectiveness of various machine learning, including deep learning and classical machine learning, in classifying stress levels-based data obtained from wearable sensors. The dataset used includes measurements from sensors such as accelerometers, electrocardiograms (ECG), electrodermal activity (EDA), temperature, and respiration, all of which were processed and normalized to serve as input for the machine learning models. Experiments were conducted to assess the performance and efficiency of each machine learning method, with a focus on determining the level of accuracy and size of model. The findings from this research aim to contribute to the development of more precise machine learning methods for stress detection. It is anticipated that these methods can be refined in future studies to improve their practical application. Moreover, this paper seeks to provide readers with a comprehensive understanding of the performance outcomes of each machine learning approach, enabling them to make informed decisions based on the specific requirements of their research, studies, or projects. |

# Introduction

Machine Learning is a branch of artificial intelligence that allows systems to learn from the data provided and make decisions or predictions without being programmed directly. In Machine Learning, algorithms are used to analyze data, identify patterns, and make predictions or decisions based on that data. The main goal of Machine Learning is to give computers the ability to improve their performance as the experience (data) is processed, without the need for direct human intervention in coding and programming. Machine learning relies heavily on data to train models used in making decisions or making predictions. The more data available, the better the algorithm can learn and the more accurate the results will be. On the other hand, the less data is provided in certain conditions or situations, the worse the results will be.

# Material and Methods



**Fig. 1.** Artificial Intelligence World Diagram

Based on **Fig. 1.** Artificial Intelligence (AI) is a branch of computer science that focuses on creating systems or machines capable of performing tasks that typically require human intelligence. These tasks include reasoning, learning, problem-solving, speech recognition, decision-making, visual perception, and language understanding. The ultimate goal of AI is to build systems that can operate independently, adapt to new information, and improve over time without human intervention.

## Part of Artificial Intelligence

The following is an example of learning that include under the category of Artificial Intelligence:

### Machine Learning

Machine Learning is a part of AI that focuses on building systems that can learn and make decisions based on data. ML models improve over time as they are exposed to more data without being explicitly programmed for each task.

The following is an example of types of Machine Learning:

#### Supervised Machine Learning

In this method, the model is trained on labeled data, where the input is paired with the correct output. The goal is to learn the mapping from input to output.

#### Semi-Supervised Machine Learning

This method uses a small amount of labeled data along with a large amount of unlabeled data to improve learning accuracy.

#### Unsupervised Machine Learning

This method involves learning patterns from unlabeled data. The goal is to discover hidden structures, such as clusters or associations, within the data.

#### Reinforcement Machine Learning

In Reinforcement Machine Learning, models learn by interacting with their environment. Models take actions, receive feedback in the form of rewards or penalties, and learn to maximize their cumulative rewards over time.

### Deep Learning

Deep Learning is a subset of Machine Learning that involves neural networks with many layers (hence "deep"). Deep learning models are particularly powerful for complex tasks such as image recognition, speech processing, and natural language understanding.

The following is an example of types of Deep Learning:

#### Convolutional Neural Networks (CNNs)

CNN is specifically designed to process structured data, such as images. They use convolutional layers to automatically detect features in images, such as edges or textures.

#### Recurrent Neural Networks (RNNs)

RNNs are designed to process sequential data, such as time, speech, or text series data. They retain the previous input memory in the sequence.

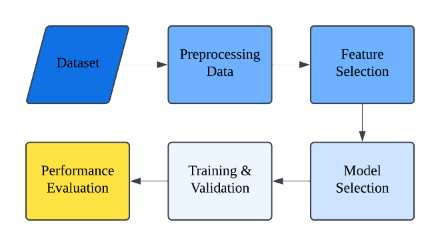
#### Generative Adversarial Networks (GANs)

GAN consists of two neural networks: a generator and a discriminator. They work together to produce new data (such as images or music) that resembles real data.

#### Long Short-Term Memory (LSTM)

The type of RNN designed to capture long-term dependencies in sequential data, preventing the vanishing gradient problem that hinders traditional RNNs from learning long sequences.

## Methods



**Fig. 2.** Stage of Machine Learning process.

Based on **Fig. 2.** The stages of the Machine Learning process cycle can be categorized into six different stages:

### Preprocessing Data

Data preprocessing is an important step in the machine learning workflow, as the quality of the data directly affects the performance of the model. This involves preparing and cleaning raw data before inputting it into the machine learning model. The goal of preprocessing is to ensure that the data is suitable for model training, minimize noise, and address any issues such as missing values or outliers. Common preprocessing techniques include:

#### Handling Missing Data

Data that is missing can happen from various sources, such as incomplete records or errors during data collection. Common strategies for handling missing data include considering the missing values using statistical measures (such as mean, median, or mode imputation), or deleting records with missing values altogether.

#### Data Normalization/Standardization

Normalization and standardization are methods used to scale numerical features so that they have a similar range or distribution. This ensures that no feature dominates the others. Normalization typically scales data to the range [0,1], while standardization transforms data to have a mean of 0 and a standard deviation of 1.

#### Encoding Categorical Variables

Many machine learning algorithms require numerical input. Therefore, categorical variables (such as gender, city names) must be transformed into numerical representations. This is typically done through techniques such as one-hot encoding (creating binary variables for each category) or label encoding (assigning integers to each category).

### Feature Selection

Feature selection is the process of identifying the most relevant features (variables) to be used in a machine learning model. The goal of feature selection is to improve model performance by eliminating irrelevant or excessive features that can cause noise or lead to overfitting. Effective feature selection also helps reduce model complexity, resulting in faster training times and better interpretability.

### Model Selection

Model selection is a critical phase in the machine learning workflow, where appropriate algorithm choices are made based on the problem at hand, data characteristics, and specific analysis objectives. The selection process involves considering various models, each with its own strengths, assumptions, and limitations. For instance, linear models such as linear regression and logistic regression are simple yet powerful techniques suitable for problems where the relationship between variables is approximately linear. These models are often the first choice for tasks such as predicting numerical values or binary classification, especially when interpretability and ease of use are important. On the other hand, more complex models like Support Vector Machines (SVM) can handle non-linear relationships and high-dimensional data, making them effective for tasks involving complex datasets. However, SVMs require careful hyperparameter tuning and can be computationally expensive.

In addition to linear and non-linear models, ensemble methods such as Random Forest and Gradient Boosting are often preferred for their robustness and ability to reduce overfitting. These methods combine several base learners (usually decision trees) to create a strong predictive model by averaging or enhancing predictions. They are particularly useful when dealing with large amounts of data and when individual models do not perform adequately on their own. Deep neural networks, especially for deep learning models, have gained popularity for tasks involving unstructured data such as images, text, or audio, due to their ability to model highly complex patterns. However, deep learning models require substantial computational resources and large datasets to perform well. Ultimately, the model selection process should consider not only model accuracy but also the trade-off between complexity, interpretability, and the computational resources needed for training and deployment. Therefore, it is important to experiment with multiple models, assess their performance through cross-validation, and choose a model that strikes the best balance between generalization ability and efficiency.

### Training and Validation

Training and validation are important stages in the machine learning process that ensure effective learning of the model while generalizing well to new data. During training, the model learns patterns from the training data set by adjusting its internal parameters to minimize the loss function. This process involves iterating through the data multiple times to improve the model's ability to predict or classify accurately.

After training, the model is validated using a separate validation set, which helps assess its performance on unseen data. Validation ensures that the model does not overly fit the training data. Cross-validation, where the data set is divided into several folds for training and validation, is commonly used to enhance the reliability of this evaluation. If overfitting is detected, further steps such as hyperparameter tuning or model simplification may be necessary to improve generalization.

### Performance Evaluation

Performance evaluation is an important step in machine learning that measures how well a model performs on a specific task, ensuring that its predictions or classifications are reliable and accurate. This involves assessing the model's output using specific metrics that measure its effectiveness. Performance evaluation helps understand the strengths and weaknesses of the model, guiding further improvements. Common evaluation metrics for classification tasks include accuracy, precision, recall, and F1 score. Each metric provides different insights into the model's performance, and choosing the appropriate one depends on the specific task at hand, such as whether false positives or false negatives are more costly.

#### Accuracy

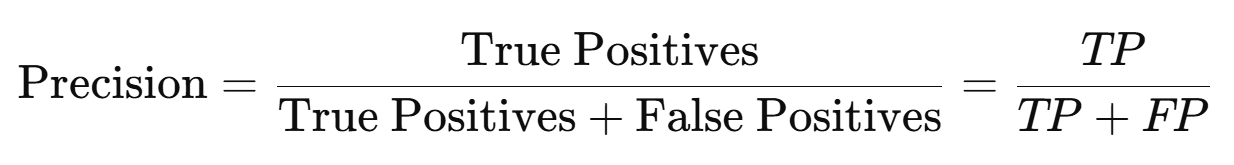
Accuracy is the proportion of correctly predicted instances out of all instances in the dataset. It provides an overall measure of model performance but can be misleading in the case of imbalanced datasets.



**Fig. 3.** Formula to determine accuracy.

#### Precision

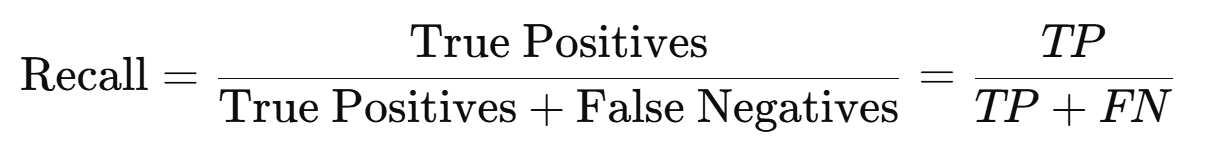
Measuring the accuracy of positive predictions made by the model precisely. This informs us how many positives predicted are actually true. High precision is important when the cost of false positives is high.



**Fig. 4.** Formula to determine precision

#### Recall (Sensitivity or True Positive Rate)

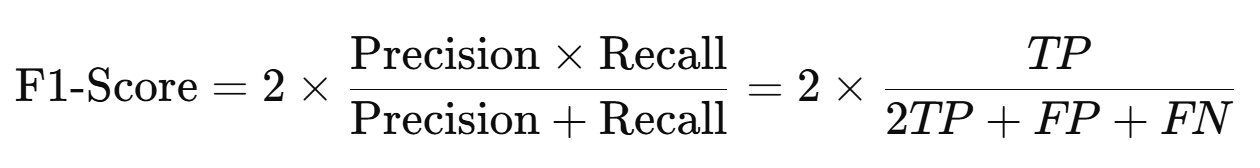
Recall measuring the model's ability to correctly identify all positive examples. It is important when false negative costs are high. High recall means fewer missed positives, but it can lead to more false positives.



**Fig. 5.** Formula to determine recall

#### F-1 Score

The F1 score is the harmonic mean of precision and recall, providing a balance between the two. It is particularly useful when there is an uneven class distribution (when one class occurs more frequently than the others). The F1 score is a better metric to use when you need to balance precision and recall, as it takes into account false positives and false negatives.



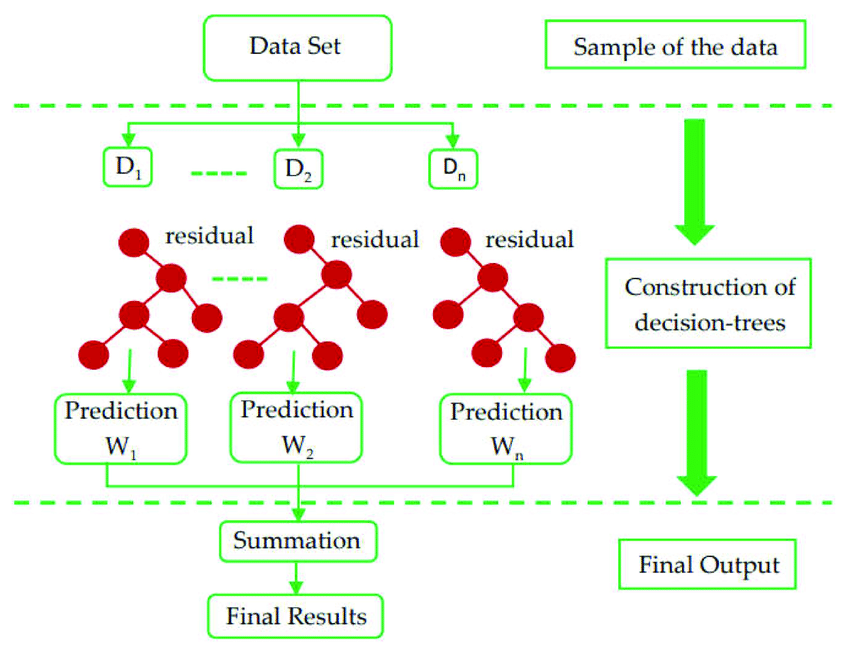
**Fig. 6.** Formula to determine F-1 score

## Machine Learning Used

In this research, Supervised Machine Learning with a classification method is used as a method to process raw data that already has labels to become a machine learning model that can be used. The following is an example of Machine Learning used in this research:

### Extreme Gradient Boosting (XGBoost)

Extreme Gradient Boosting, commonly known as XGBoost, is a machine learning method widely used for processing data through training and evaluation to produce highly accurate machine learning models. The XGBoost method offers two different approaches: classification and regression. This method falls under supervised machine learning and utilizes an ensemble approach that combines several weak models to generate strong predictions.



**Fig. 7.** Extreme Gradient Boosting (XGBoost) architecture.x

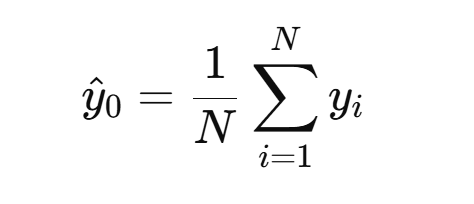
Based on **Fig. 1.** the working principle of Extreme Gradient Boosting or XGBoost involves the use of gradient boosting methods, where the machine learning algorithm creates simple decision trees and iteratively adds more trees to the model. The algorithm corrects errors made by each previous tree, and this process is repeated until the specified number of trees is reached, or the model's performance no longer significantly improves. The following is an explanation of the workflow of XGBoost in processing a dataset until it becomes a Machine Learning model:

#### Initial Dataset

The process begins with an initial set of data consisting of input features (represented as **𝑋**) and corresponding target values (represented as **𝑦**) that we want to predict. This dataset is used to construct the first decision tree and serves as the foundation for all subsequent steps in the improvement process.

#### First Decision Tree

The first decision tree is constructed using the initial dataset. This tree is built to directly predict the target variable. The goal is to create an initial rough estimate of the target value based on features. The tree uses input features to make splits that minimize the loss function, which can be Mean Squared Error for regression or log-loss for classification.

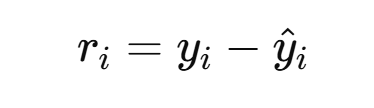


**Fig. 8.** Formula of first decision tree in XGBoost.

where ***N*** is the number of data points in the dataset, and ***yi***​ is the actual value of the target for each data point.

#### Storing Predictions and Residuals

After the first tree is trained, predictions are made on the training data set. These predictions are stored, and residuals (errors) are calculated, indicating how far the predictions deviate from the actual target values. These residuals are crucial for improving the model in the next iteration. Residuals are calculated by subtracting the model's prediction from the actual target value.

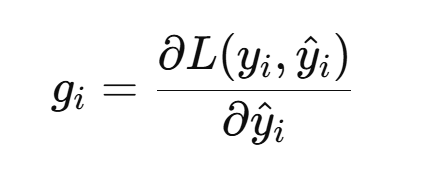


**Fig. 9.** Formula of residuals in XGBoost.

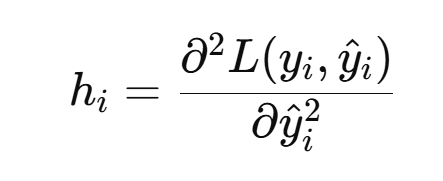
where ***ri*** is the residual for the ***i*-th** data point, ***yi***​ is the true value, and ***y^i*** is the predicted value from the first model. The residuals represent the portion of the target that was not captured by the first model, and this is what the next model (tree) will focus on.

#### Next Decision Tree

The second decision tree is built to predict the residuals from the first model. The goal here is to improve the errors made by the first model by focusing on the residuals. This process is repeated iteratively to gradually enhance the model's performance. The tree is trained to minimize the loss function on the residuals, effectively correcting the errors. The model is trained by calculating the gradient and Hessian of the loss function.

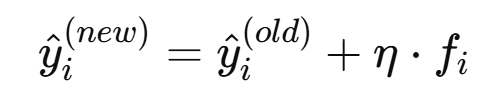


**Fig. 10.** Formula of gradient in XGBoost.



**Fig. 11.** Formula of hessian in XGBoost.

These gradients and Hessians guide how the model should adjust to minimize the residuals in the next decision tree. The updated prediction is given by:



**Fig. 12.** Formula of updated prediction.

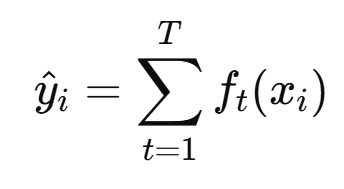
where ***η*** is the learning rate, and ***fi***​ is the output of the new tree for data point ***i***.

#### Iteration Continues

The process of building new decision trees and updating predictions continues repeatedly. With each new tree, the model corrects more errors from the previous tree, refining predictions step by step. Iterations continue until a certain number of trees are built, or until further improvement is no longer significant. At each step, the new tree focuses on predicting the remaining errors (residuals) from the previous model, and predictions are updated with new information. This iterative process leads to a stronger model as each iteration improves upon previous mistakes.a

#### Combining Results

After all decision trees have been constructed, the final prediction is obtained by combining the results from all the trees. The trees work together to create a more accurate final prediction by focusing on various residual aspects at each iteration.

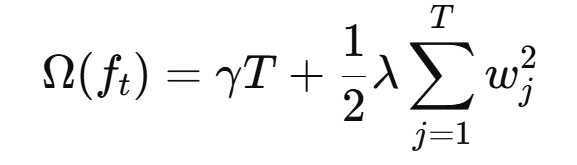


**Fig. 13.** Formula of combining each result.

where ***T*** is the total number of trees, and ***ft(xi)*** is the prediction from tree ***t*** for data point ***i***. This cumulative model is more powerful than any single tree because it gradually corrects errors across iterations.

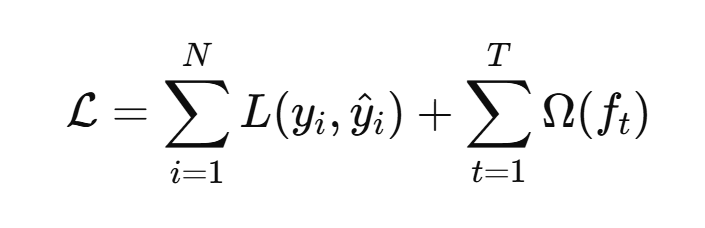
#### Optimization and Regularization

XGBoost incorporates the term regularization in its objective function to prevent overfitting, ensuring that the model does not become too complex and overly adapted to the training data. Regularization penalizes large trees and helps control model complexity. The total objective function consists of two parts: the loss function and the regularization term. The regularization term helps penalize overly complex trees. The regularization term for tree **𝑡** is given by:



**Fig. 14.** Formula of regularization in XGBoost.

where **γ** is the complexity parameter, ***T*** is the number of leaves in the tree, ***wi*** is the weight of each leaf, and **λ** is a regularization parameter controlling the size of the weights. The total loss function to minimize is:



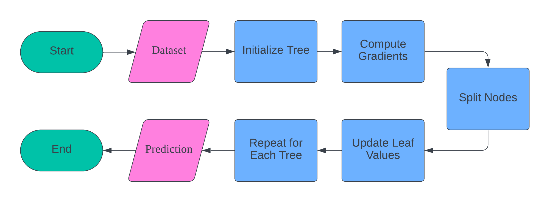
**Fig. 15.** Formula of optimization in XGBoost.

#### Final Result

After all iterations and regularization, the model produces a series of final predictions. The output is a strong and accurate model that can be used to make predictions on unseen new data. The final model is the result of combining all decision trees and optimizing the objective function to minimize loss and prevent overfitting.

The final model prediction is created by summing up the outputs from all decision trees. The final model represents the best estimate of the target variable based on the training data, incorporating all adjustments made throughout the iterations.

### Light Gradient Boosting Machine (LightGBM)



**Fig. 16.** LightGBM architecture

The working principle of LightGBM (Light Gradient Boosting Machine) involves the use of gradient boosting methods, which focus on the iterative creation of decision trees. Unlike traditional gradient boosting, LightGBM employs a histogram-based approach and gradient-based one-side sampling (GOSS) to select data samples and features more efficiently. This results in faster training times and lower memory usage. The following is an explanation of the workflow of LightGBM in processing datasets into a Machine Learning model:

#### Input Dataset

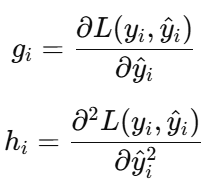
This step involves inputting the data that will be used to train the model. The data must be properly prepared and formatted. This includes handling missing values and normalizing the data if necessary. This data will serve as the foundation for building the LightGBM model. Ensure that the dataset includes relevant features and labels.

#### Initialize Tree

This stage initiates the initialization of the first decision tree in the model. As part of the boosting process, LightGBM will build several trees sequentially. The initialization of this first tree is often performed by making a simple initial guess. Each subsequent tree will correct the errors of the previous tree. The structure of this tree will be determined based on the features and labels provided in the dataset.

#### Compute Gradients

At this stage, we calculate the gradient of the loss function for each data point. The gradient (gi) measures how far the current prediction is from the actual target value. The Hessians (hi) are the second derivatives of the loss function, providing information about the curvature of the loss function. The loss function (L(yi, y^i)) is a metric used to evaluate how well the model performs. This calculation is crucial for determining the direction and step in updating the model.

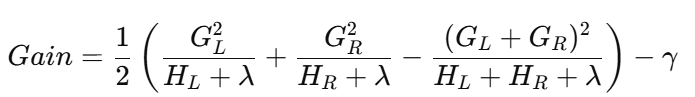


**Fig. 17.** Compute gradients formula

* *gi*​: The gradient of the loss function for the i-th data point.
* *hi*​: The Hessian (second derivative) of the loss function for the i-th data point.
* *L(yi,y^i)*: The loss function that measures how well the prediction *y^i* matches the target *yi*​.

#### Split Nodes

This process involves selecting the best split to separate the data while minimizing the loss function. We calculate the gradient sums (GL, GR) and Hessians (HL, HR) for the left and right branches after the split. Regularization parameters (λ and γ) are used to prevent overfitting by controlling the model's complexity. The best split is the one that provides the greatest reduction in the loss function. This step is repeated until the tree reaches its maximum depth or other stopping criteria are met.

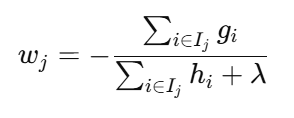


**Fig. 18.** Split nodes formula.

* *GL​, GR*​: The total gradient for the left and right branches.
* *HL, HR*​: The number of Hessians for the left and right branches.
* λ: Regulatory parameters to prevent overfitting.
* γ: The regularization parameter that controls the complexity of the model.

#### Update Leaf Values

After the nodes are split, the values at the leaves are updated. The value at the j leaf (wj) is calculated based on the data index (Ij) that enters that leaf. This update is performed to reduce the overall loss function. These leaf values will be used to make predictions in the next step. This step ensures that the model learns better patterns from the data.



**Fig. 19.** Update leaf values formula

* *wj*​: The value on the j-th leaf.
* *Ij​*: The index of the data that enters leaf j.

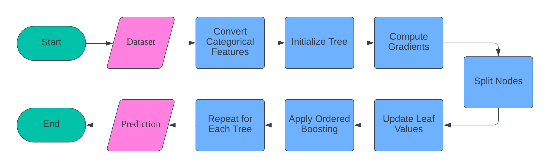
#### Repeat for Each Tree

The previous steps are repeated for each tree in the ensemble. Each new tree is constructed to correct the errors made by the previous trees. This process continues until the specified number of trees is reached or other stopping criteria are met. By adding trees iteratively, the LightGBM model becomes increasingly accurate. This step is crucial for achieving good predictions.

#### Prediction Output

After all the trees in the ensemble have been trained, the model is ready to make predictions. The predictions are based on the combination of all the trees in the model. These prediction results can then be compared with the target values to measure the model's accuracy. This step is the final output of the LightGBM training process. The trained model can now be used to predict new data.

### CatBoost



**Fig. 20.** CatBoost architecture.

The working principle of CatBoost (Categorical Boosting) involves the use of gradient boosting methods, particularly optimized for categorical features. CatBoost transforms categorical features into numerical values using target-based statistics and then iteratively builds decision trees. Below is an explanation of the workflow of CatBoost in processing datasets into a Machine Learning model:

#### Input Data

This step involves entering the data that will be used to train the model. The data must be prepared and formatted correctly. This includes handling missing values and normalizing the data if necessary. This data will serve as the foundation for building the CatBoost model. Ensure that the dataset includes relevant features and labels.

#### Convert Categorical Features

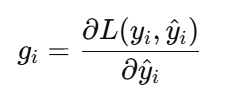
At this stage, the categorical features are converted into numerical ones using target-based statistics. CatBoost employs a special encoding technique that takes into account the target distribution for each category. This assists the model in better understanding the influence of each category on the target. This process is crucial because most machine learning algorithms perform better with numerical data. Thus, this transformation enhances the model's accuracy.

#### Initialize Tree

This stage initiates the initialization of the first decision tree in the model. As part of the boosting process, CatBoost will build several trees sequentially. The initialization of this first tree is often performed by making a simple initial guess. Each subsequent tree will correct the errors of the previous tree. The structure of this tree will be determined based on the features and labels provided in the dataset.

#### Compute Gradients

At this stage, we calculate the gradient of the loss function for each data point. The gradient (gi) measures how far the current prediction is from the actual target value. The loss function (L (yi, y^i)) is a metric used to evaluate how well the model performs. This calculation is crucial for determining the direction and step in updating the model. The gradient provides guidance on how to correct prediction errors.



**Fig. 21.** Compute gradients formula.

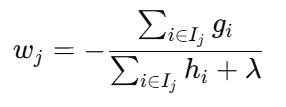
* *gi*​: The gradient of the loss function for the i-th data.
* *L(yi,y^i)*: The loss function measures how well the prediction y^i aligns with the target yi.

#### Split Nodes

This process involves selecting the best split to separate the data while minimizing the loss function. The best split is the one that provides the greatest reduction in the loss function. CatBoost employs specific techniques to ensure that the chosen split does not result in target leakage. The correct selection of splits is key to building an accurate and efficient tree. This step is repeated until the tree reaches maximum depth or other stopping criteria are met.

#### Update Leaf Values

After the nodes are split, the values at the leaves are updated. The value at the j leaf (wj) is calculated based on the data index (Ij) that enters that leaf. This update is performed to reduce the overall loss function. These leaf values will be used to make predictions in the next step. This step ensures that the model learns better patterns from the data.



**Fig. 22.** Update leaf values formula.

* *wj​*: The value on the leaf to j.
* *Ij​*: The data index that enters leaf j.

#### Apply Ordered Boosting

At this stage, CatBoost applies ordered boosting techniques to prevent target leakage. Ordered boosting ensures that each prediction only uses past data to train the model. This is crucial in handling categorical data and maintaining the validity of the model. This technique helps preserve the integrity and accuracy of the model. With ordered boosting, the model can learn more effectively from the data without information leakage.

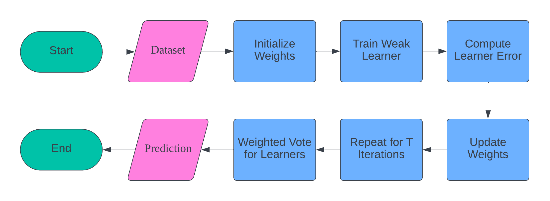
#### Repeat for Each Tree

The previous steps are repeated for each tree in the ensemble. Each new tree is constructed to correct the errors made by the previous trees. This process continues until the specified number of trees is reached or other stopping criteria are met. By adding trees iteratively, the CatBoost model becomes increasingly accurate. This step is crucial for achieving good predictions.

#### Prediction Output

After all the trees in the ensemble have been trained, the model is ready to make predictions. The predictions are based on the combination of all the trees in the model. These prediction results can then be compared with the target values to measure the model's accuracy. This step is the final output of the CatBoost training process. The trained model can now be used to predict new data.

### AdaBoost



**Fig. 23.** AdaBoost architecture.

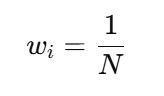
The working principle of AdaBoost (Adaptive Boosting) involves training several weak learners (typically decision trees) sequentially. Each learner focuses on the previous errors by adjusting the sample weights, making it more likely to correct those errors. The following is an explanation of the workflow of AdaBoost in processing the dataset into a Machine Learning model:

#### Input Data

This step involves entering the data that will be used to train the model. The data must be prepared and formatted correctly. This includes handling missing values and normalizing the data if necessary. This data will serve as the foundation for building the AdaBoost model. Ensure that the dataset includes relevant features and labels.

#### Initialize Weights

At this step, the sample weights are initialized uniformly. Each sample in the dataset is assigned the same initial weight, typically 𝑤𝑖=1/𝑁 where 𝑁 is the total number of samples. These weights will be updated during iterations to give more attention to samples that are difficult to predict. Uniform initialization ensures that all samples initially have the same influence in training the model. This step is crucial for starting a balanced learning process.



**Fig. 24.** Initialize weights formula

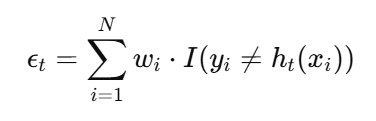
* *wi*​: Sample robot
* *N*: Total number of samples.

#### Train Weak Learner

This process involves training a weak learner model on weighted data. A weak learner is a simple model that can be a decision stump or another model with basic performance. This model is trained using initialized sample weights. The weak learner serves as a building block for the more complex AdaBoost model. Each weak learner will focus on the samples that are difficult to predict in the subsequent iterations.

#### Compute Learner Error

At this step, we calculate the error of the weak learner. The error 𝜖𝑡 is the total number of prediction errors made by the weak learner at the t iteration. The indicator 𝐼(𝑦𝑖≠ℎ𝑡(𝑥𝑖)) is valued at 1 if the prediction is incorrect and 0 if it is correct. Calculating this error is important to determine how well the weak learner performs on the weighted data. This error is then used to update the sample weights.

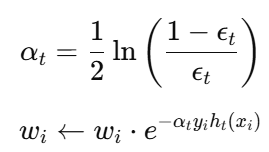


**Fig. 25.** Compute learner error formula

* *ϵt*​: The error of weak learners in the t iteration.
* *I(yi≠ht(xi))*: An indicator that has a value of 1 if the prediction is incorrect, and 0 if it is correct.

#### Update Weights

After calculating the errors of weak learners, the sample weights are updated based on those errors. The weight 𝛼𝑡 of the weak learner at the t iteration is calculated to determine the model's contribution to the final prediction. The sample weights 𝑤𝑖 are updated to give more attention to samples that are predicted incorrectly. This process ensures that the next weak learner will focus on more difficult samples. This update is crucial for iteratively improving the model's accuracy.



**Fig. 26.** Update weights formula.

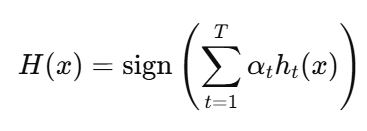
* *αt​*: The weight of weak learners at the t iteration.
* *yi​*: Label the target of sample i.
* *ht(xi)*: Prediction from weak learners at the t iteration for the i sample.

#### Repeat for T Iterations

The previous steps are repeated for T iterations. Each iteration involves training a new weak learner, calculating the error, and updating the sample weights. This process continues until the specified number of iterations is reached. By adding weak learners iteratively, the AdaBoost model becomes increasingly accurate. This step ensures that the model learns from its mistakes and corrects them at each iteration.

#### Weighted Vote for Learners

The previous steps are repeated for T iterations. Each iteration involves training a new weak learner, calculating the error, and updating the sample weights. This process continues until the specified number of iterations is reached. By adding weak learners iteratively, the AdaBoost model becomes increasingly accurate. This step ensures that the model learns from its mistakes and corrects them at each iteration.



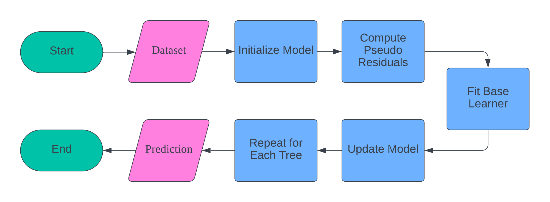
**Fig. 27.** Weighted vote for learner formula.

* *H(x)*: Final prediction of the model.
* *Sign (⋅)*: The function returns +1 if positive and -1 if negative.

#### Prediction Output

After the weighted voting is completed, the model is ready to make predictions. The predictions are based on the results of the weighted voting from all weak learners in the model. These prediction results can then be compared with the target values to measure the model's accuracy. This step is the final output of the AdaBoost training process. The trained model can now be used to predict new data.

### Gradient Boosting Classifier



**Fig. 28.** Gradient boosting classifier architecture.

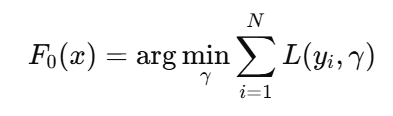
The working principle of the Gradient Boosting Classifier involves the sequential training of an ensemble of decision trees. Each tree is trained on the residual errors of the previous tree to enhance accuracy. Below is an explanation of the workflow of the Gradient Boosting Classifier in processing the dataset into a Machine Learning model:

#### Input Data

This step involves inputting the data that will be used to train the model. The data must be properly prepared and formatted. This includes handling missing values and normalizing the data if necessary. This data will serve as the foundation for building the Gradient Boosting Classifier model. Ensure that the dataset includes relevant features and labels.

#### Initialize Model

At this stage, the model is initialized with the average of the labels. The initial model 𝐹₀(𝑥) is typically the average value of the target 𝑦. The loss function 𝐿(𝑦ᵢ,γ) is used to evaluate how well the initial predictions perform. This initialization provides a starting point for the boosting process. This step is crucial for commencing the model with a reasonable estimate.

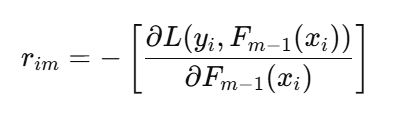


**Fig. 29.** Initialize model formula.

* *F0​(x)*: Early model.
* *L(yi,γ)*: loss function.

#### Compute Pseudo-Residuals

At this stage, we calculate the pseudo-residual based on the previous predictions. The pseudo-residual (rim) for the i sample at the m iteration is the difference between the target yi and the previous model prediction (Fm-1(xi)). This pseudo-residual is used as a temporary label to train the base learner. Calculating the pseudo-residual is important for understanding how far the current model predictions are from the actual target.



**Fig. 30.** Compute pseudo-residuals formula

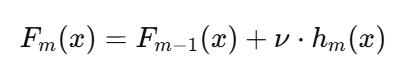
* *rim​*: The pseudo-residual for the i sample in the m iteration.
* *Fm−1(xi)*: The model prediction at iteration m-1 for sample i.

#### Fit Base Learner

This process involves training base learners on pseudo-residuals. Base learners are simple models such as decision stumps that are trained using pseudo-residuals as temporary labels. These base learners assist the model in understanding more complex patterns in the data. Training base learners on pseudo-residuals is the core of the boosting process. This step is crucial for correcting the prediction errors of the previous model.

#### Update Model

After the base learners are trained, the model is updated with the output from these base learners. The model at the m iteration 𝐹𝑚(𝑥) is a combination of the previous model 𝐹𝑚−1(𝑥) and the output from the base learner ℎ𝑚(𝑥) multiplied by the learning rate factor 𝜈. This update helps the model to gradually improve its prediction accuracy. This step is crucial for integrating the contributions from all base learners.



**Fig. 31.** Update model formula

* *Fm​(x)*: Model in the m iteration.
* *ν\nuν*: Learning rate factor.
* *hm(x)*: The output of the basic learner in the miteration.

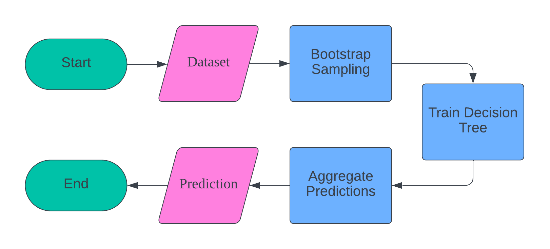
#### Repeat for Each Tree

The previous steps are repeated for each tree in the ensemble. Each iteration involves training a new base learner, calculating pseudo-residuals, and updating the model. This process continues until a specified number of iterations is reached or other stopping criteria are met. By adding trees iteratively, the Gradient Boosting Classifier model becomes increasingly accurate. This step ensures that the model learns from mistakes and corrects them at each iteration.

#### Prediction Output

After all the trees in the ensemble have been trained, the model is ready to make predictions. The predictions are based on the combination of all the trees in the model. These prediction results can then be compared with the target values to measure the model's accuracy. This step is the final output of the Gradient Boosting Classifier training process. The trained model can now be used to predict new data.

### Random Forest



**Fig. 32.** Random forest architecture.

The working principle of Random Forest involves training multiple decision trees on different bootstrap samples of the data and aggregating their predictions. The following is an explanation of the workflow of Random Forest in processing a dataset until it becomes a Machine Learning model:

#### Input Data

This step involves entering the data that will be used to train the model. The data must be prepared and formatted correctly. This includes handling missing values and normalizing the data if necessary. This data will serve as the foundation for building the Random Forest model. Ensure that the dataset includes relevant features and labels.

#### Bootstrap Sampling

At this stage, a random subset of data is created with replacement (sampling with replacement). Bootstrap sampling means that each sample in the original dataset can be selected more than once to be included in the subset. This results in several different subsets, each used to train different decision trees. This process is important for enhancing the diversity of decision trees in the forest.

#### Train Decision Trees

This process involves training decision trees on bootstrapped subsets of data. Each decision tree is trained independently using different subsets of data. Training decision trees on different subsets of data helps in capturing various patterns within the data. This step ensures that each decision tree in the forest contributes to the final prediction in a unique manner.

#### Aggregate Predictions

After all decision trees have been trained, the predictions from each decision tree are combined. The final prediction 𝑦^ is obtained by taking the majority vote of the predictions from all decision trees ℎ𝑡(𝑥). This means that the most common prediction among all decision trees is taken as the final prediction. Combining predictions from all decision trees helps to improve the model's accuracy by reducing overfitting.



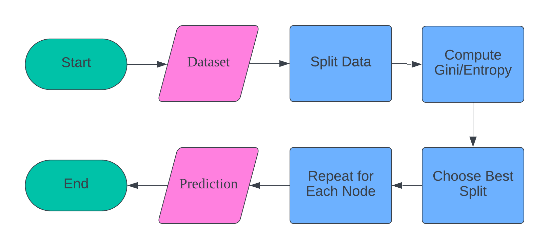
**Fig. 33.** Aggregate predictions formula.

* *y^*​: End prediction.
* *ht(x)*: Prediction from the t decision tree.

#### Prediction Output

This step involves generating predictions based on the majority vote from all decision trees. These predictions are the final output of the Random Forest model. The prediction results can then be compared with the target values to measure the model's accuracy. This step represents the final output of the Random Forest training process. The trained model can now be used to predict new data.

### Decision Tree



**Fig. 34.** Decision tree architecture.

The working principle of Decision Trees involves recursively splitting a dataset into subsets based on the best feature splits that minimize impurity. The following is an explanation of the workflow of Decision Trees in processing a dataset into a Machine Learning model:

#### Input Data

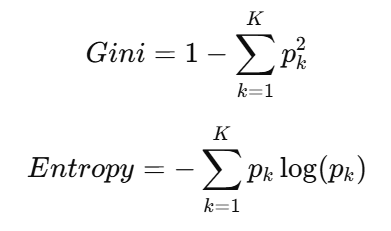
This step involves entering the data that will be used to train the model. The data must be prepared and formatted correctly. This includes handling missing values and normalizing the data if necessary. This data will serve as the foundation for building the Decision Tree model. Ensure that the dataset includes relevant features and labels.

#### Split Data

At this stage, the data is divided into two subsets based on the best split. The splitting process is carried out by selecting features and values that separate the data into two parts in the most optimal way. This best split is identified by measuring how well the data separation performs. The objective of this split is to separate the data in a manner that maximizes the homogeneity of each subset.

#### Compute Gini/Entropy

This step involves calculating impurity (Gini or Entropy) for each split. Impurity is used to measure how well the data is separated at each split. Low impurity indicates a good split, as the resulting subsets are more homogeneous.



**Fig. 35.** Compute Gini/Entropy formula.

* *pk*​: The sample proportion of the k class.

#### Choose Best Split

At this stage, the best split is chosen by minimizing impurity. The best split is one that produces two data subsets with the lowest impurity. Selecting the best split is crucial to ensure that each data partition maximizes the information gained. This step aids in constructing an optimal tree structure.

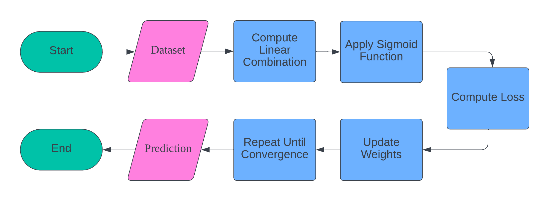
#### Repeat for Each Node

The process of splitting and calculating impurity is repeated for each node until a leaf node is reached. Each node is treated as a new root node, and this process continues recursively. This iteration is carried out until all nodes become leaf nodes or until other stopping criteria are met (such as the maximum depth of the tree or the minimum number of samples in each node). This step is crucial for thoroughly constructing the decision tree.

#### Prediction Output

This step involves generating predictions based on the majority class at the leaf node. Each leaf node represents the final decision based on the majority class of the samples that fall into that node. The prediction results can then be compared with the target values to measure the model's accuracy. This step is the final output of the Decision Tree training process. The trained model can now be used to predict new data.

### Logistic Regression



**Fig. 36.** Logistic regression formula.

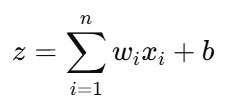
The working principle of Logistic Regression involves adjusting a linear combination of input features to a logistic function to model the probability of a binary outcome. Here is an explanation of the workflow of Logistic Regression in processing a dataset into a Machine Learning model:

#### Input Data

This step involves entering the data that will be used to train the model. The data must be prepared and formatted correctly, including handling missing values and normalizing the data if necessary. This data will serve as the foundation for building the Logistic Regression model. Ensure that the dataset includes relevant features and labels.

#### Compute Linear Combination

At this stage, the linear combination of the input features is calculated. This linear combination is computed using a formula, where 𝑧 is the linear combination of the features, 𝑤𝑖 is the weight of the i feature, 𝑥𝑖 is the i feature, and 𝑏 is the bias. This linear combination will then be used in the sigmoid function to produce probabilities.

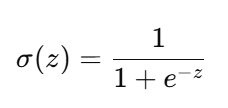


**Fig. 37.** Compute linear combination formula

* *z*: Linear combination of features.
* *wi*​: Weight Feature robot to i.
* *xi*​: Feature to i.
* *b*: Bias.

#### Apply Sigmoid Function

This step involves applying the sigmoid function to transform the output from a linear combination into probabilities. The sigmoid function is expressed as. Where 𝜎(𝑧) is the sigmoid function and 𝑒 is the exponential number. This sigmoid function produces output in the range of 0 to 1, which can be interpreted as probabilities.

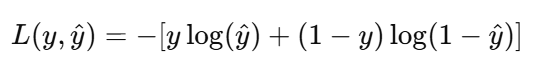


**Fig. 38.**  Apply sigmoid function formula

* *σ(z)*: Sigmoid function.
* *e*: Exponential.

#### Compute Loss

At this stage, the loss function is calculated using cross-entropy. The loss function measures how well the model's predictions align with the target labels. Cross-entropy loss is a measure of the discrepancy between the model's predictions and the actual values, and it is used to guide the weight update process in gradient descent.

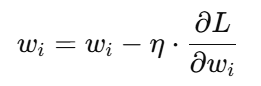


**Fig. 39.** Compute loss formula.

* *L(y,y^​)*: Loss function.
* *y*: Target lable.
* *y^*: Prediction.

#### Update Weights

At this stage, the weights are updated using gradient descent. This update process involves calculating the gradient of the loss function with respect to each weight and updating those weights to minimize the loss function. This update is performed repeatedly to reduce the loss value and improve the model's accuracy.



**Fig. 40.** Update weights formula.

* *η*: Learning rate.
* *∂L/∂wi* ​: Gradien dari fungsi loss terhadap bobot.

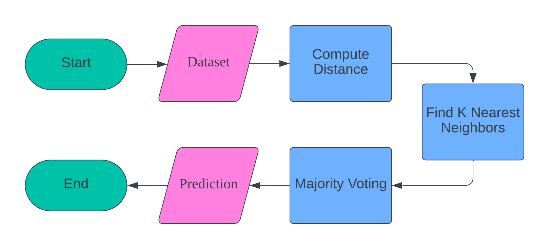
#### Repeat Until Convergence

The process of weight updating and loss calculation is repeated until convergence is achieved. Convergence is reached when the changes in weights become very small or when the loss no longer decreases significantly. This iteration ensures that the model reaches an optimal solution and is ready to be used on new data.

#### Prediction Output

After the model reaches convergence, predictions are generated based on probabilities calculated using the sigmoid function. These probabilities can then be compared with target values to measure the model's accuracy. This step represents the final output of the Logistic Regression training process. The trained model can now be used to predict new data.

### K-Nearest Neighbors (KNN)



**Fig. 41.** KNN architecture

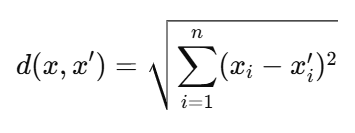
The working principle of K-Nearest Neighbors (KNN) involves classifying samples based on the majority vote of its K nearest neighbors in the training data. The following is an explanation of the workflow of KNN in processing the dataset into a Machine Learning model:

#### Input Data

This step involves inputting the data that will be used to train the K-Nearest Neighbors (KNN) model. The data must be properly prepared and formatted, including handling missing values and normalizing the data if necessary. This data will be used to calculate the distance between new samples and training samples.

#### Compute Distance

At this stage, the distance between the new sample and each sample in the training data is calculated. This distance can be computed using various metrics such as Euclidean, Manhattan, or Minkowski distance. The calculation of this distance is crucial for determining how close the new sample is to the training samples.



**Fig. 42.** Compute distance formula.

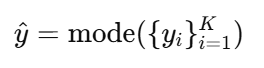
* *d(x,x′)*: The distance between two points x and x′.
* *xi*​: The i feature of sample x.

#### Find K Nearest Neighbors

After the distance is calculated, the next step is to find the K nearest neighbors based on the calculated distance. These nearest neighbors are the training samples that have the shortest distance to the new sample. The number of K nearest neighbors must be determined in advance.

#### Majority Voting

At this stage, a majority vote is conducted to determine the class prediction of the new sample. The class that appears most frequently among the K nearest neighbors will be selected as the class prediction for the new sample. This majority voting ensures that the prediction is based on the majority of the nearest neighbors.



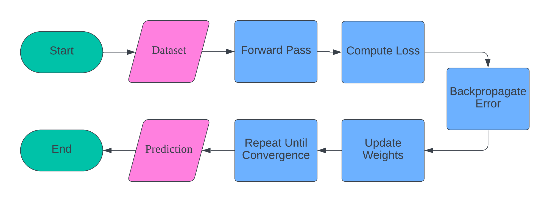
**Fig. 43.** Majority voting formula

* *y^*​: Class prediction.
* *yi* ​: Neighbors lable.

#### Prediction Output

The final step is to generate predictions based on the results of the majority voting. This prediction is the class assigned to a new sample based on its nearest neighbors. This step is the final output of the K-Nearest Neighbors (KNN) process. The trained KNN model can now be used to predict new data.

### Multilayer Perceptron (MLP)



**Fig. 44.** MLP architecture.

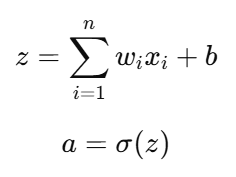
The working principle of the Multilayer Perceptron (MLP) involves several layers of neurons where each neuron is connected to all neurons in the previous layer. The network is trained using backpropagation to minimize the loss function. The following is an explanation of the workflow of MLP in processing the dataset into a Machine Learning model:

#### Input Data

This step involves entering the data that will be used to train the Multilayer Perceptron (MLP) model. The data must be properly prepared and formatted, including handling missing values and normalizing the data if necessary. This data will be used as input for the MLP model training process.

#### Forward Pass

At this stage, the output from each layer of neurons is calculated. The linear combination of the input features is computed first, followed by the application of the activation function to produce the neuron activation. This process is repeated for each layer in the MLP network until the output layer is reached.

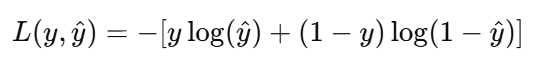


**Fig. 45.** Forward pass formula.

* *z*: Linear combination from feature.
* *wi*​: Weight feature to i.
* *xi*​: Feature to i.
* *b*: Bias.
* *a*: Neuron activation.
* *σ(z)*: Activation function (sigmoid, relu, etc).

#### Compute Loss

This step involves calculating the loss function using cross-entropy or mean squared error (MSE). This loss function measures how well the model's predictions align with the target labels. The calculation of this loss is crucial for guiding the weight update process in the subsequent step.

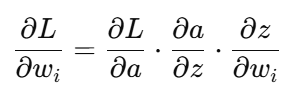


**Fig. 46.** Compute loss formula.

* *L(y,y^​)*: Loss function.
* *y*: Target lable.
* *y^*: Prediction.

#### Backpropagate Error

At this stage, the gradient of the loss function is calculated and propagated back through the network. This process is known as backpropagation, which involves the computation of gradients with respect to weights, activations, and linear combinations at each neuron. These gradients are used to update the weights with the aim of minimizing the loss function.

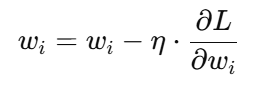


**Fig. 47.** Backpropagate error formula.

* ∂L/∂wi​ ​: The gradient of the loss function with respect to the weights.
* ∂L/∂a: The gradient of the loss function with respect to the activation.
* ∂a/∂z: The gradient of activation with respect to the linear combination.
* ∂z/∂wi: The gradient of the linear combination with respect to the weights.

#### Update Weights

This step involves updating the weights using gradient descent. The weights are updated based on the gradients calculated during the backpropagation step and the predetermined learning rate. The purpose of this update process is to reduce the loss value and improve the model's accuracy.



**Fig. 48.** Update weights formula.

* *η*: Learning rate.

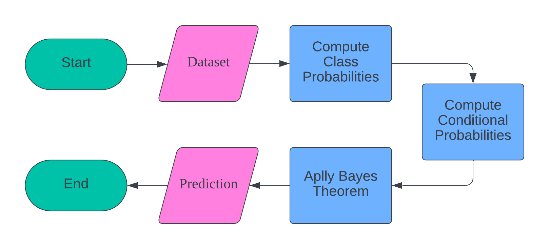
#### Repeat Until Convergence

The process of forward pass, loss calculation, backpropagation, and weight updates is repeated until convergence is achieved. Convergence is reached when the changes in weights become very small or when the loss no longer decreases significantly. This iteration ensures that the model reaches an optimal solution.

#### Prediction Output

After the model reaches convergence, predictions are generated based on the output calculated at the final layer of the network. These predictions can then be compared with the target values to measure the model's accuracy. This step represents the final output of the MLP training process. The trained model can now be used to predict new data.

### Gaussian Naïve Bayes (GaussianNB)



**Fig. 49.** GaussianNB architecture.

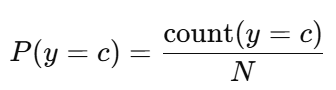
The working principle of Gaussian Naïve Bayes (GaussianNB) involves the use of Bayes' theorem with the assumption of independence among features. Each feature is assumed to follow a Gaussian distribution. The following is an explanation of the workflow of GaussianNB in processing the dataset into a Machine Learning model:

#### Input Data

This step involves inputting the data that will be used to train the Gaussian Naïve Bayes model (GaussianNB). The data must be properly prepared and formatted, including handling missing values and normalizing the data if necessary. This data will be used to calculate class probabilities and conditional probabilities.

#### Compute Class Probabilities

At this stage, the initial probability of each class is calculated. This probability is known as the prior probability and is computed by dividing the number of samples in each class by the total number of samples. This probability provides fundamental information about the class distribution in the data.

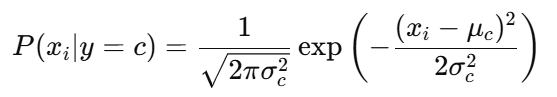


**Fig. 50.** Compute class probabilities formula

* *P(y=c)*: Class C probabilities.
* *count(y=c)*: The number of samples in class c.
* *N*: Total number of samples.

#### Compute Conditional Probabilities

This step involves calculating the conditional probabilities for each feature, given a specific class. The conditional probabilities are computed using the Gaussian distribution, with the mean and standard deviation of the features in each class. This information is crucial for calculating the posterior probabilities using Bayes' Theorem.

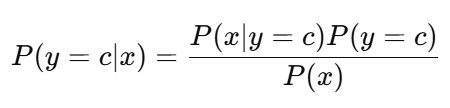


**Fig. 51.** Compute conditional probabilities formula,

* *P(xi∣y=c)*: The probability of feature xi given class c.
* *μc* ​: The average feature in class c.
* *σc* ​: Standard deviation of features in class c.
* *exp*: Exponential function.

#### Apply bayes’ Theorem

At this stage, Bayes' Theorem is used to calculate the posterior probabilities of each class, given the input features. These posterior probabilities are the product of the prior probabilities and the previously calculated conditional probabilities. The posterior probabilities provide the final estimate of the likelihood of a sample belonging to each class.



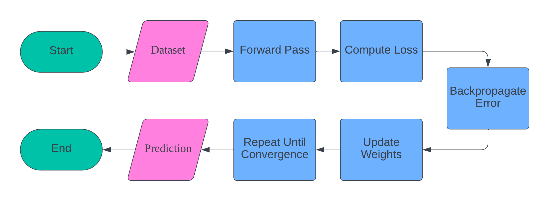
**Fig. 52.** Apply bayes theorem formula.

* *P(y=c∣x)*: The posterior probability of class c.
* *P(x∣y=c)*: The conditional probability of feature x given class c.
* *P(y=c)*: The prior probability of class c.

#### Prediction Output

The final step is to generate predictions based on the calculated posterior probabilities. These predictions represent the class with the highest posterior probability for each sample. This step constitutes the final output of the Gaussian Naïve Bayes (GaussianNB) process. The trained model can now be utilized to predict new data.

### Feedforward Neural Networks (FNN)



**Fig. 53.** FNN architecture.

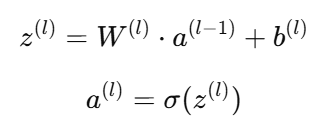
The working principle of a Feedforward Neural Network (FNN) involves several layers of neurons where information flows in one direction from input to output. The network is trained using backpropagation to minimize the loss function. The following is an explanation of the workflow of FNN in processing the dataset into a Machine Learning model:

#### Input Data

This step involves inputting the data that will be used to train the FNN model. The input data must be properly prepared and formatted, including handling missing values and normalizing the data if necessary. This data will be used to train the neural network model.

#### Forward Pass

At this stage, the output from each layer of neurons is calculated by multiplying the weights with the inputs, adding the bias, and applying the activation function. This process begins from the input layer and continues until it reaches the output layer, resulting in the initial predictions of the model.

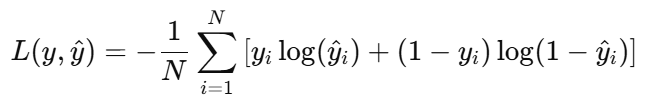


**Fig. 54.** Forward pass formula.

* z(l): Linear combination of features in layer l.
* W(l): The weight on layer l.
* a(l−1): Activation of neurons from the previous layer.
* b(l): Bias on layer l.
* σ: Activation functions (for example, sigmoid, relu).

#### Compute Loss

This step involves calculating the loss function, which is a measure of how well the model's predictions align with the target labels. Common loss functions include cross-entropy for classification and mean squared error (MSE) for regression. This loss provides feedback on the model's performance.



**Fig. 55.** Compute loss formula.

* L(y,y^​): Cross-entropy loss.
* yi: Target label for i sample.
* y^i: Probability prediction for the i sample.

#### Backpropagate Error

At this stage, the gradient of the loss function is calculated and propagated back through the network to update the weights. Backpropagation utilizes the chain rule to compute the gradient from the output layer back to the input layer, allowing the model to learn from its mistakes.



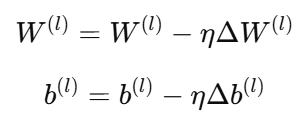


**Fig. 56.** Backpropagate error formula.

* δ(l): Error on layer *l*
* σ′(z(l)): The derivative of the activation function.

#### Update Weights

This step involves updating the network weights using gradient descent. The weights are updated by subtracting the computed gradient multiplied by the learning rate. This process aims to minimize the loss function.



**Fig. 57.** Update weights formula.

* *η*: Learning rate

#### Repeat Until Convergence

The process of forward pass, computing loss, backpropagating error, and updating weights is repeated until the model reaches convergence. Convergence is achieved when the changes in the loss function or weights become very small, indicating that the model has learned the patterns in the data.

#### Prediction Output

The final step is to generate predictions based on the outputs calculated by the neural network model. These predictions can take the form of probabilities for classification or continuous values for regression, depending on the type of problem being addressed by the model.

# Results and Discussion

The following is the result of the research that has been conducted by following the methods used in developing a Machine Learning:

## Dataset

The dataset used in this research was taken from Kaggle and contains sensor data to detect stress levels in humans. Initially, the dataset consisted of 60,807,600 rows and 11 columns. After filtering, irrelevant features such as wrist sensors were removed. The final dataset used consists of 6 main columns: ACC\_chest, ECG\_chest, EDA\_chest, TEMP\_chest, respiration, and stress\_level.

## Preprocessing Data

The dataset undergoes preprocessing which involves data balancing. Data balancing is important to ensure that each column has the same amount of data, aiming to avoid bias in the machine learning model that will be built. Consequently, each feature can contribute proportionally in data analysis and stress level prediction.

## Feature Selection

In this experiment, we chose not to implement feature selection as part of the model training process. The main reason for this decision is that the primary goal of the research is to test the capabilities of each machine learning algorithm in handling the same dataset, without external interventions that could affect their performance. In other words, we want to evaluate each model based on its default parameters and assess its performance objectively and consistently.

## Model Selection

### Parameter of Extreme Gradient Boosting (XGBoost)

The parameter model is the optimized values during the training process to minimize error function or improve prediction accuracy, and they are automatically set by the learning algorithm based on training data. The main function of model parameters is to determine how the model uses input data to produce the desired output, which includes the influence of input on predictions, adaptation to data complexity, and model performance improvement. The following is an example of some model parameters used in XGBoost:

#### booster: ‘gbtree’

Boosters determine the type of boosting model used. Gbtree is a tree-based boosting model commonly used for classification tasks. Default gbtree is chosen for its effectiveness in handling complex and unstructured data.

#### n\_estimators: 100

N\_estimators determine the number of trees that will be built during training. More estimators can improve accuracy but also increase training time. The default value of 100 is chosen to strike a balance between performance and training efficiency.

#### learning\_rate: 0.3

The learning rate controls the size of the steps taken to update the model weights. A lower value means smoother and more stable updates. The default value of 0.3 is chosen to ensure gradual learning of the model, avoiding overshooting, and enabling more stable convergence.

#### max\_depth: 6

Max depth determines the maximum depth of the tree. Deeper depths allow the model to capture more complex patterns. The default value of 6 is chosen to strike a balance between model complexity and the risk of overfitting.

#### min\_child\_weight: 1

The minimum child weight determines the minimum weight required for a leaf node. A lower value allows for more leaves, capturing more details in the data. The default value of 1 is chosen to provide maximum flexibility in constructing the tree.

#### subsample: 1.0

Subsampling determines the proportion of data used for training each tree. A lower value can improve generalization by reducing overfitting. The default value of 1 is chosen to ensure that the model uses all available data for training.

#### colsample\_bytree: 1.0

Colsample bytree determines the proportion of features considered for each tree. A lower value can improve generalization by reducing overfitting. The default value of 1 is chosen to ensure the model uses all available features.

#### gamma: 0

Gamma determines the minimum reduction in impurity required to perform a split. A lower value allows for more splits, capturing more details in the data. The default value of 0 is chosen to provide maximum flexibility in node splitting.

#### n\_jobs: None

N\_jobs determine the number of CPU cores used for training. A value of -1 means using all available cores to maximize training performance. The default value of None means using one core, but users can set a value to accelerate training.

#### objective: ‘binary:logistic’

The objective is to determine the loss function used to measure the model's error. Binary logistic is a suitable loss function for binary classification. Default binary logistic is chosen for its effectiveness in handling binary classification tasks.

### Parameter of Light Gradient Boosting Machine (LightGBM)

#### num\_leaves: 31

The number of leaves determines the maximum number of leaves on each tree. More leaves allow the model to capture more complex patterns. The default value of 31 is chosen because it strikes a balance between model complexity and the risk of overfitting.

#### learning\_rate: 0.1

The learning rate controls the size of the steps taken to update the model weights. A lower value means smoother and more stable updates. The default value of 0.1 is chosen to ensure gradual learning of the model, avoiding overshooting, and enabling more stable convergence.

#### n\_estimators: 100

N\_estimators determines the number of trees that will be built during training. More estimators can improve accuracy but also increase training time. The default value of 100 is chosen to strike a balance between performance and training efficiency.

#### max\_depth: -1

Max depth determines the maximum depth of the tree. A value of -1 means there is no depth limit, allowing the tree to grow according to the data's needs. The default value of -1 is chosen to provide maximum flexibility in tree construction.

#### min\_child\_weight: 1e-3

The minimum child weight determines the minimum total sample weight in a leaf node for splitting. A lower value allows for more splits, capturing more details in the data. The default value of 1e-3 is chosen to provide maximum flexibility in tree construction.

#### subsample: 1.0

Subsampling determines the proportion of data used for training each tree. A lower value can improve generalization by reducing overfitting. The default value of 1.0 is chosen to ensure that the model uses all available data for training.

#### colsample\_bytree: 1.0

Colsample bytree determines the proportion of features considered for each tree. A lower value can improve generalization by reducing overfitting. The default value of 1.0 is chosen to ensure the model uses all available features.

#### reg\_alpha: 0.0

Reg alpha determines the penalty on L1 for feature weights, preventing overfitting by encouraging some feature weights to become zero. Default 0.0 is chosen because it provides minimal penalty, ensuring the model has flexibility to learn all relevant features.

#### reg\_lambda: 0.0

The lambda parameter determines the L2 penalty on feature weights, preventing overfitting by adding a penalty to large weights. The default value of 0.0 is chosen because it provides minimal penalty, ensuring that the model has flexibility to learn all relevant features.

#### random\_state: None

Random state ensures reproducible results by setting consistent random values. This setting is important for validation and experiments. Default None means random values will be generated automatically, but users can set fixed values for reproducibility.

### Parameter of CatBoost

#### iterations: 1000

Iterations determine the number of trees that will be built during training. More iterations allow the model to learn more deeply from the data, but it can increase the risk of overfitting. The default value of 1000 is chosen to strike a balance between sufficient learning time and avoiding overfitting.

#### learning\_rate: 0.03

The learning rate controls the size of the steps taken to update the model weights. A lower value means smoother and more stable updates. The default value of 0.03 is chosen to ensure gradual learning of the model, avoiding overshooting, and enabling more stable convergence.

#### depth: 6

Depth determines the maximum depth of the decision tree in the model. Deeper depth allows the model to capture more complex feature interactions. The default depth of 6 is chosen because it strikes a balance between model complexity and the risk of overfitting – it is deep enough to capture important patterns but not too deep to overfit.

#### l2\_leaf\_reg: 3.0

L2 leaf regularization is a regularization parameter that adds a penalty to large weights to prevent overfitting. The default value of 3.0 is chosen to provide a moderate level of regularization, sufficient to control model complexity without sacrificing performance.

#### loss\_function: 'Logloss'

The loss function determines the loss function used to measure the model's error. Logloss is suitable for binary classification, optimizing prediction probabilities to approach the actual target value. Default Logloss is chosen for its effectiveness in handling binary classification tasks.

#### custom\_metric: []

Custom metrics allow users to add additional evaluation metrics that are specific to their needs. This provides flexibility in monitoring model performance based on relevant metrics. Default [] means no custom metrics are used unless specified by the user.

#### thread\_count: -1

Thread count determines the number of CPU cores used for training. A value of -1 means using all available cores to maximize training performance. This default is chosen to ensure training proceeds as quickly as possible by utilizing all available resources.

#### random\_seed: None

Random seed ensures reproducible training results by setting consistent random values. This configuration is important for validation and experiments. Default None means random values will be generated automatically, but users can set fixed values for reproducibility.

#### border\_count: 254

Border count determines the number of divisions (borders) used to transform numerical features into categorical features. More borders allow for capturing finer variations in features. The default value of 254 is chosen to provide sufficient resolution in handling numerical features.

#### bagging\_temperature: 1.0

Bagging temperature controls the diversity in the model by determining the probability of sample selection for each iteration. A higher value increases diversity and reduces overfitting. The default value of 1.0 is chosen to strike a balance between model diversity and training stability.

### Parameter of AdaBoost

#### n\_estimators: 50

N\_estimators determines the number of trees to be built in the ensemble. More estimators can improve accuracy but also increase training time. The default value of 50 is chosen to strike a balance between performance and training efficiency.

#### learning\_rate: 1.0

The learning rate controls the impact of each tree on the final prediction. A lower value reduces the contribution of each tree, allowing for more stable training. The default value of 1.0 is chosen to provide significant contribution from each tree, accelerating convergence.

#### algorithm: 'SAMME.R'

The algorithm determines the boosting method to be used. SAMME.R uses prediction probabilities to improve accuracy and efficiency. Default SAMME.R is chosen because of its faster convergence compared to standard SAMME.

#### random\_state: None

Random state ensures reproducible results by setting consistent random values. This setting is important for validation and experiments. Default None means random values will be generated automatically, but users can set fixed values for reproducibility.

#### base\_estimator: DecisionTreeClassifier()

The base estimator determines the basic model used in boosting. Decision trees are a common choice due to their simplicity and ability to handle non-linear data. The default DecisionTreeClassifier() is chosen for its efficiency in simple classification tasks.

#### base\_estimator\_\_max\_depth: None

The maximum depth of the base estimator determines the maximum depth of the tree. Deeper depths allow the model to capture more complex patterns. The default value of None means there is no depth limit, allowing the tree to grow according to the data's needs.

#### min\_samples\_split: 2

The minimum samples split determines the minimum number of samples required to split a node. A lower value allows for more divisions, capturing more details in the data. The default value of 2 is chosen to provide maximum flexibility in building the tree.

#### min\_samples\_leaf: 1

The minimum samples leaf determines the minimum number of samples that must be present in a tree leaf. A lower value allows for smaller leaves, capturing more details in the data. The default value of 1 is chosen to provide maximum flexibility in tree construction.

#### max\_features: None

Max features determine the maximum number of features considered for each split. A value of None means that all features will be considered, allowing for maximum utilization of feature information. Default None is chosen to ensure the use of all available features.

#### criterion: 'gini'

Criterion determines the function used to measure the quality of division. Gini impurity is a common choice for classification tasks due to its efficiency in evaluating class separation. Default gini is selected for its speed and accuracy in separating classes.

### Parameter of Gradient Boosting Classifier

#### n\_estimators: 100

N\_estimators determines the number of trees that will be built during training. More estimators can improve accuracy but also increase training time. The default value of 100 is chosen to strike a balance between performance and training efficiency.

#### learning\_rate: 0.1

The learning rate controls the size of the steps taken to update the model weights. A lower value means smoother and more stable updates. The default value of 0.1 is chosen to ensure gradual learning of the model, avoiding overshooting, and enabling more stable convergence.

#### max\_depth: 3

Max depth determines the maximum depth of the tree. Deeper depths allow the model to capture more complex patterns. The default value of 3 is chosen to strike a balance between model complexity and the risk of overfitting.

#### min\_samples\_split: 2

The minimum samples split determines the minimum number of samples required to split a node. A lower value allows for more divisions, capturing more details in the data. The default value of 2 is chosen to provide maximum flexibility in building the tree.

#### min\_samples\_leaf: 1

The minimum samples leaf determines the minimum number of samples that must be present in a tree leaf. A lower value allows for smaller leaves, capturing more details in the data. The default value of 1 is chosen to provide maximum flexibility in tree construction.

#### subsample: 1.0

Subsampling determines the proportion of data used for training each tree. A lower value can improve generalization by reducing overfitting. The default value of 1.0 is chosen to ensure that the model uses all available data for training.

#### max\_features: None

Max features determine the maximum number of features considered for each split. A value of None means that all features will be considered, allowing for maximum utilization of feature information. Default None is chosen to ensure the use of all available features.

#### random\_state: None

Random state ensures reproducible results by setting consistent random values. This setting is important for validation and experiments. Default None means random values will be generated automatically, but users can set fixed values for reproducibility.

#### loss: 'deviance'

Loss determines the loss function used to measure the model's error. Deviance is the log-loss loss function suitable for binary classification. Default deviance is chosen for its effectiveness in handling binary classification tasks.

#### criterion: 'friedman\_mse'

Criterion determines the function used to measure the quality of partitioning. Friedman MSE is an adapted criterion for boosting, reducing the estimation variance in predictions. Default friedman\_mse is chosen to enhance model stability and accuracy.

### Parameter of Random Forest

#### n\_estimators: 100

The N\_estimators determine the number of trees to be built in the forest. More trees can improve accuracy but also increase training time. The default value of 100 is chosen to strike a balance between performance and training efficiency.

#### criterion: 'gini'

Criterion determines the function used to measure the quality of division. Gini impurity is a common choice for classification tasks due to its efficiency in evaluating class separation. Default gini is chosen for its speed and accuracy in separating classes.

#### max\_depth: None

Max depth determines the maximum depth of the tree. Deeper depths allow the model to capture more complex patterns. Default None means there is no depth limit, allowing the tree to grow according to the data's needs.

#### min\_samples\_split: 2

The minimum samples split determines the minimum number of samples required to split a node. A lower value allows for more divisions, capturing more details in the data. The default value of 2 is chosen to provide maximum flexibility in building the tree.

#### min\_samples\_leaf: 1

The minimum samples leaf determines the minimum number of samples that must be present in a tree leaf. A lower value allows for smaller leaves, capturing more details in the data. The default value of 1 is chosen to provide maximum flexibility in tree construction.

#### min\_weight\_fraction\_leaf: 0.0

The minimum weight fraction leaf determines the minimum fraction of the total weight (for weighted samples) required for leaf nodes. A default of 0.0 means there are no weight restrictions, allowing full flexibility in leaf distribution.

#### max\_features: 'auto'

Max features determine the maximum number of features considered for each split. Auto selects the square root of the total number of features, providing a balance between complexity and the risk of overfitting. The default auto selection is made to ensure optimal use of feature information without overfitting.

#### random\_state: None

A random state ensures reproducible results by setting consistent random values. This setting is important for validation and experimentation. A default of None means that random values will be generated automatically, but users can set a fixed value for reproducibility.

#### bootstrap: True

Bootstrap determines whether bootstrap sampling is used when constructing the tree. The use of bootstrap sampling can enhance diversity within the tree and reduce overfitting. The default value of True is selected to ensure model diversity and improved performance.

#### oob\_score: False

The out-of-bag score determines whether the out-of-bag score is used to evaluate the accuracy of the model. The use of the out-of-bag score can provide a more objective accuracy estimate without requiring an additional test set. The default is set to False to save computation time unless out-of-bag evaluation is necessary.

### Parameter of Decision Tree

#### criterion: 'gini'

Criterion determines the function used to measure the quality of division. Gini impurity is a common choice for classification tasks due to its efficiency in evaluating class separation. Default gini is chosen for its speed and accuracy in separating classes.

#### splitter: 'best'

The splitter determines the strategy used to select the division for each node. It is best to choose the optimal division based on the specified criteria. By default, the best division is selected to ensure quality at each step.

#### max\_depth: None

The maximum depth determines the maximum depth of the tree. A deeper depth allows the model to capture more complex patterns. The default value of None means there is no depth limitation, allowing the tree to grow as needed by the data.

#### min\_samples\_split: 2

The minimum samples split determines the minimum number of samples required to split a node. A lower value allows for more splits, capturing more detail in the data. The default of 2 is chosen to provide maximum flexibility in building the tree.

#### min\_samples\_leaf: 1

The minimum samples leaf determines the minimum number of samples that must be present in a tree leaf. A lower value allows for smaller leaves, capturing more details in the data. The default value of 1 is chosen to provide maximum flexibility in tree construction.

#### min\_weight\_fraction\_leaf: 0.0

The minimum weight fraction of leaves determines the minimum fraction of the total weight (for weighted samples) required for a leaf node. A default value of 0.0 means there is no weight constraint, allowing full flexibility in leaf allocation.

#### max\_features: None

Max features determine the maximum number of features considered for each split. A value of None means that all features will be considered, allowing for maximum utilization of feature information. Default None is chosen to ensure the use of all available features.

#### random\_state: None

A random state ensures reproducible results by setting consistent random values. This setting is important for validation and experimentation. A default of None means that random values will be generated automatically, but users can set a fixed value for reproducibility.

#### max\_leaf\_nodes: None

The maximum leaf nodes determine the maximum number of leaf nodes in a tree. This restriction can reduce overfitting by preventing the tree from becoming too complex. The default value of None means there is no limitation, allowing the tree to grow as needed by the data.

#### min\_impurity\_decrease: 0.0

The minimum impurity decrease determines the minimum reduction in impurity required to perform a split. A lower value allows for more splits, capturing more detail in the data. A default of 0.0 is chosen to provide maximum flexibility in node splitting.

### Parameter of Logistic Regression

#### penalty: 'l2'

The penalty determines the type of regularization used to prevent overfitting. L2 regularization adds a penalty to the square of the weights, encouraging smaller weights. The default L2 is chosen due to its effectiveness in controlling model complexity without compromising performance.

#### dual: False

Dual determines whether the primal or dual problem will be solved. Default False means that the primal problem will be solved, which is more efficient for a larger number of samples than features. Default False is chosen for computational efficiency in most cases.

#### tol: 0.0001

To determine tolerance for termination criteria. A lower value implies stricter convergence criteria, allowing for more precise solutions. The default value of 0.0001 is chosen to ensure stable convergence and high accuracy.

#### C: 1.0

C determines the strength of regularization, with lower values indicating stronger regularization. The default value of 1.0 is chosen to strike a balance between regularization and the model's ability to learn from the data.

#### fit\_intercept: True

The fit intercept determines whether the intercept will be added to the model. The use of an intercept is important to accommodate bias in the data. The default value of True is chosen to ensure that the model can adjust for any offset in the data.

#### intercept\_scaling: 1

Intercept scaling determines the scale for the intercept in the model. A default value of 1 means no additional scale is applied. Defaulting to 1 is chosen to maintain the interpretability of the intercept without any additional modifications.

#### solver: 'lbfgs'

The solver determines the algorithm used to optimize the model. L-BFGS is an efficient algorithm for large problems and is suitable for L2 regularization. Default L-BFGS is chosen for its efficiency and stability in optimizing logistic regression models.

#### max\_iter: 100

Max iter determines the maximum number of iterations for the solver. More iterations allow the solver to achieve better convergence. The default value of 100 is chosen to strike a balance between convergence accuracy and computational efficiency.

#### multi\_class: 'auto'

Multi-class determines the strategy used for multi-class classification. Auto automatically selects the best strategy based on the data. Default auto is chosen to ensure flexibility and optimal performance in various classification scenarios.

#### random\_state: None

Random state ensures reproducible results by setting consistent random values. This setting is important for validation and experiments. Default None means random values will be generated automatically, but users can set fixed values for reproducibility.

### Parameter of K-Nearest Neighbors (KNN)

#### n\_neighbors: 5

N\_neighbors determines the number of nearest neighbors considered for classifying data points. Having more neighbors can enhance prediction stability but may also reduce sensitivity to local patterns. The default value of 5 is chosen to strike a balance between sensitivity and prediction stability.

#### weights: 'uniform'

Weights determine how weight is assigned to neighbors. Uniform assigns the same weight to all neighbors, while distance assigns greater weight to closer neighbors. Default uniform is chosen to provide balanced contributions from all neighbors.

#### algorithm: 'auto'

The algorithm determines the algorithm used to calculate the nearest neighbors. Auto automatically selects the best algorithm based on the data. Default auto is chosen to ensure optimal efficiency and performance in various scenarios.

#### leaf\_size: 30

Leaf size determines the size of the leaves in KD-trees or Ball trees used by the algorithm. A larger size can improve search speed but also increases memory usage. The default value of 30 is chosen to strike a balance between speed and memory usage.

#### p: 2

P determines the norm strength used to calculate the distance between data points. The default value of 2 means using the Euclidean norm (L2), which is commonly used and intuitive. Default 2 is chosen because it provides a common and efficient distance calculation.

#### metric: 'minkowski'

Metric determines the distance function used to calculate the distance between data points. Minkowski is a generalization of several distance metrics, including Euclidean and Manhattan. Default Minkowski is chosen for flexibility in handling various types of data and scenarios.

#### metric\_params: None

Metric parameters allow for additional customization of the distance metric used. This provides extra flexibility in calculating distances. The default value of "None" means that no additional parameters are specified, and the default settings for the chosen metric are used.

#### n\_jobs: None

N\_jobs determines the number of CPU cores used for calculating nearest neighbors. A value of -1 means using all available cores to maximize search performance. The default value of None means using one core, but users can set a value to speed up the search.

#### random\_state: None

Random state ensures reproducible results by setting consistent random values. This setting is important for validation and experiments. Default None means random values will be generated automatically, but users can set fixed values for reproducibility.

#### leaf\_size: 30

Leaf size determines the size of the leaves in the KD-tree or Ball tree used by the algorithm. A larger size can improve search speed but also increase memory usage. Default 30 is chosen to provide a balance between speed and memory usage.

### Parameter of Multilayer Perceptron (MLP)

#### hidden\_layer\_sizes: (100,)

The number of units in each hidden layer. Default (100,) means one hidden layer with 100 units. This size provides a balance between model complexity and its ability to learn from data.

#### activation: 'relu'

The activation function for the hidden layer. The default 'relu' (Rectified Linear Unit) is used because it is fast and helps address the vanishing gradient problem.

#### solver: 'adam'

The algorithm to optimize weights. The default 'adam' is an efficient optimization method and works well with various types of data.

#### alpha: 0.0001

The L2 regularization parameter (penalty). The default value of 0.0001 helps reduce overfitting by adding a penalty to large weights.

#### batch\_size: 'auto'

The sample size per batch. The default 'auto' selects the batch size as min (200, n\_samples), providing a balance between computational efficiency and training stability.

#### learning\_rate: 'constant'

Scheduling the learning rate. The default 'constant' uses a fixed learning rate during training, providing stability in the learning process.

#### learning\_rate\_init: 0.001

The initial learning rate is used. The default value of 0.001 is commonly used and works well in many situations.

#### max\_iter: 200

The maximum number of iterations. The default value of 200 provides sufficient opportunity for the model to converge during training.

#### shuffle: True

Randomizing the data after each epoch. The default value of True helps prevents the model from memorizing the data sequence, which can lead to overfitting.

#### random\_state: None

The seed is for reproducibility. Default None means that random values will be generated automatically, but users can set a fixed value for consistency.

### Parameter of Gaussian Naïve Bayes (GaussianNB)

#### priors: None

The prior class probabilities. Default None means that the class probabilities will be calculated from the training data, allowing the model to adapt to the data distribution.

#### var\_smoothing: 1e-9

Part of the largest variant of all features that will be added for numerical stability. The default 1e-9 helps prevent division by zero issues and improves computational stability.

### Parameter of FNN

#### units: 128

The number of neurons in each layer. The default 128 provides a balance between model complexity and the ability to learn from data.

#### activation: 'relu'

The activation function is used for every layer except the output layer. The default 'relu' is used because it is fast and helps address the vanishing gradient problem.

#### optimizer: 'adam'

The optimizer used for training. The default 'adam' is an efficient optimization method that works well with various types of data.

#### loss: 'sparse\_categorical\_crossentropy'

The function 'loss' is used for multi-class classification problems. The default 'sparse\_categorical\_crossentropy' is used because it is effective in calculating loss for multi-class classification.

#### metrics: ['accuracy']

The metric used for model evaluation. The default 'accuracy' is used to measure the performance of the classification model.

#### batch\_size: 32

The number of samples per batch. Default 32 provides a balance between computational efficiency and training stability.

#### epochs: 50

The total number of training iterations on the dataset. The default value of 10 provides sufficient opportunity for the model to converge during training. But in this case the training using 50 epochs.

#### validation\_split: 0.2

The proportion of training data used as validation data. Default 0.2 means 20% of the data is used for validation, aiding in monitoring overfitting.

## Training and Validation

### Data Splitting

At this stage, the dataset undergoes two main separations. The first separation involves dividing the dataset into x and y. Here, x contains all the feature values extracted from the columns of the dataset, while y holds the labels, specifically the stress levels, which are the target values the model aims to predict. This is the initial step in preparing the data for machine learning training, where the features (x) will be used to train the model, and the labels (y) will guide the learning process.

Following this, the dataset is further split into training and testing subsets, where 85% of the data is allocated for training, and 15% is reserved for testing. The training data (both x\_train and y\_train) will be used to train the model, allowing it to learn the relationship between the input features and the target labels. On the other hand, the testing data (both x\_test and y\_test) will serve as a validation set to evaluate the performance of the model. This division ensures that the model can be trained on one portion of the data and tested on a separate portion that the model has never seen before, helping to assess its generalizability and avoid overfitting.

### Train Model

During the training phase, 85% of the dataset is used to train the model. This data serves to provide information to the model about the relationship between features (independent variables) and the target (dependent variable). The model uses this training data to learn and adjust its internal parameters in order to predict the target accurately. In this process, the model processes the data, tries to recognize patterns or correlations between input and output, and optimizes its weights and biases based on the loss function.

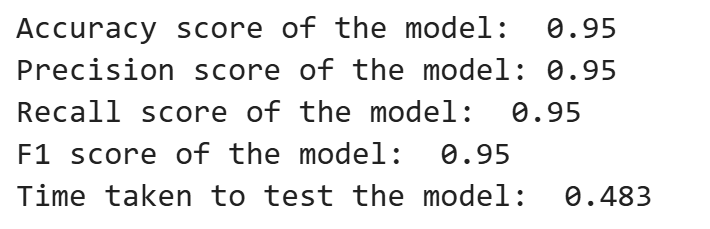
### Evaluate Model

After the model has been trained using 85% of the training data, the next step is testing (evaluation) using the remaining 15% of the dataset that has not been used during training. This testing data serves to evaluate how well the trained model can predict unseen data. This data is considered out-of-sample, meaning the model has no prior information about it during the training process.

## Performance Evaluation

All Machine Learning models are evaluated using several common metrics: Accuracy, Precision, Recall, F1-Score, and the time required for testing. These metrics provide a comprehensive understanding of how well the model performs in predicting stress levels based on the dataset.

### Extreme Gradient Boosting (XGBoost)

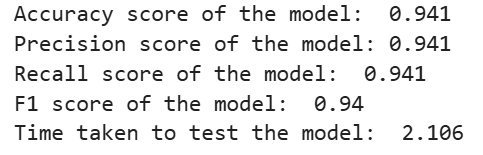


**Fig. 58.** XGBoost performance evaluation

Based on **Fig. 58.** the performance of the XGBoost model was evaluated using several metrics, all of which indicate strong results. This model achieved an accuracy of 0.95, meaning it correctly predicts 95% of total instances, demonstrating overall excellent performance. Both precision and recall, also at 0.95, highlight the model's ability to accurately identify true positives while minimizing false positives and false negatives. The F1 score of 0.95 indicates a balanced performance between precision and recall, ensuring that the model performs well even in imbalanced data cases. Additionally, the model only took 0.483 seconds to generate predictions on test data, showcasing its computational efficiency. Overall, these results indicate that the XGBoost model is highly effective, reliable, and efficient in predicting stress levels.

|  |  |  |
| --- | --- | --- |
| **Fig. 59.** Normalized Confusion Matrix of XGBoost | **Fig. 60.** Non-Normalized Matrix of XGBoost | **Fig. 61.** ROC Curve of XGBoost |

### Light Gradient Boosting Machine (LightGBM)

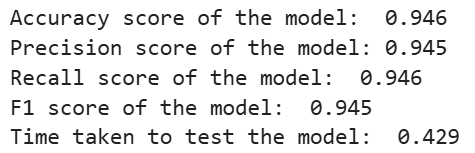


**Fig. 62.** LightGBM performance evaluation

The performance of the LightGBM model was evaluated using several metrics, all of which indicate strong results. This model achieved an accuracy of 0.941, meaning it correctly predicts 94.1% of total instances, demonstrating overall excellent performance. Both precision and recall, both at 0.941, highlight the model's ability to accurately identify true positives while minimizing false positives and false negatives. The F1 score of 0.94 indicates a balanced performance between precision and recall, ensuring that the model performs well even in imbalanced data cases. Additionally, this model only took 2.106 seconds to generate predictions on the test data, showcasing its computational efficiency. Overall, these results demonstrate that the LightGBM model is highly effective, reliable, and efficient in predicting stress levels.

|  |  |  |
| --- | --- | --- |
| **Fig. 63.** Normalized Confusion Matrix of LightGBM | **Fig. 64.** Non-Normalized Confusion Matrix of LightGBM | **Fig. 65.** ROC Curve of LightGBM |

### CatBoost

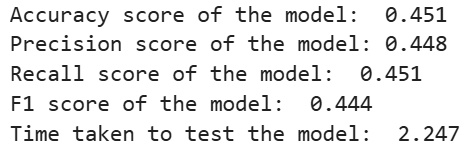


**Fig. 66.** CatBoost performance evaluation

The performance of the CatBoost model was evaluated using several metrics, all of which showed impressive results. This model achieved an accuracy of 0.946, meaning it correctly predicted 94.6% of total instances, demonstrating overall excellent performance. Both precision and recall, at 0.945 and 0.946 respectively, highlight the model's ability to accurately identify true positives while minimizing false positives and false negatives. An F1 score of 0.945 indicates a balanced performance between precision and recall, ensuring that the model performs well even in imbalanced data cases. Additionally, this model only took 0.429 seconds to generate predictions on test data, showcasing its computational efficiency. Overall, these results indicate that the CatBoost model is highly effective, reliable, and efficient in predicting stress levels.

|  |  |  |
| --- | --- | --- |
| **Fig. 67.** Normalized Confusion Matrix of CatBoost | **Fig. 68.** Non-Normalized Confusion Matrix of CatBoost | **Fig. 69.** ROC Curve of CatBoost |

### AdaBoost

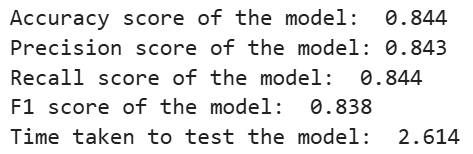


**Fig. 70.** AdaBoost performance evaluation

The AdaBoost model's performance was evaluated using several metrics, and the results indicate that the model is not optimal. The model achieved an accuracy of 0.451, which means it correctly predicts 45.1% of total instances, falling below the acceptable threshold for reliable performance. Precision and recall, at 0.448 and 0.451 respectively, reflect the model's limited ability to accurately identify true positives while minimizing false positives and false negatives. The F1 score of 0.444 further highlights this imbalance, indicating that the model struggles to maintain a good balance between precision and recall. Additionally, this model takes 2.247 seconds to generate predictions on test data, showing relatively slower computational efficiency compared to other models. Overall, these results suggest that the AdaBoost model is less effective and less reliable in predicting stress levels.

|  |  |  |
| --- | --- | --- |
| **Fig. 71.** Normalized Confusion Matrix of AdaBoost | **Fig. 72.** Non-Normalized Confusion Matrix of AdaBoost | **Fig. 73.** ROC Curve of AdaBoost |

### Gradient Boosting Classifier

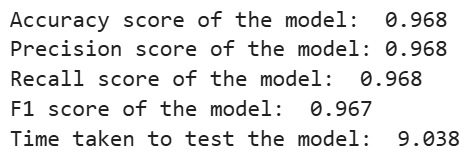


**Fig. 74.** Gradient Boosting Classifier performance evaluation

The Gradient Boosting Classifier performance was evaluated using several metrics, resulting in strong outcomes. This model achieved an accuracy of 0.844, meaning it correctly predicted 84.4% of total instances, demonstrating overall solid performance. Precision and recall, at 0.843 and 0.844 respectively, highlight the model's ability to accurately identify true positives while minimizing false positives and false negatives. The F1 score of 0.838 indicates a balanced performance between precision and recall, ensuring that the model handles imbalanced data fairly well. Additionally, the model took 2.614 seconds to generate predictions on test data, showing relatively slower computational efficiency compared to some other models. Overall, these results indicate that the Gradient Boosting Classifier model is effective, reliable, and reasonably efficient in predicting stress levels.

|  |  |  |
| --- | --- | --- |
| **Fig. 75.** Normalized Confusion Matrix of Gradient Boosting Classifier | **Fig. 76.** Non-Normalized Confusion Matrix of Gradient Boosting Classifier | **Fig. 77.** ROC Curve of Gradient Boosting Classifier |

### Random Forest

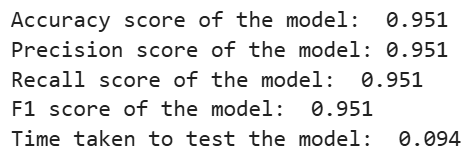


**Fig. 78.** Random Forest performance evaluation

The Random Forest model was evaluated using several metrics, and the results indicate excellent performance. This model achieved an accuracy of 0.968, meaning it correctly predicts 96.8% of total instances, demonstrating outstanding overall performance. Both precision and recall, both at 0.968, highlight the model's ability to accurately identify true positives while minimizing false positives and false negatives. The F1 score of 0.967 further confirms the balance between precision and recall, ensuring that the model performs very well even with potentially imbalanced data. However, the model takes 9.038 seconds to generate predictions on test data, indicating that it is computationally intensive and slower compared to other models. Overall, these results show that the Random Forest model is highly effective and reliable in predicting stress levels, although it may require more computational resources.

|  |  |  |
| --- | --- | --- |
| **Fig. 79.** Normalized Confusion Matrix of Random Forest | **Fig. 80.** Non-Normalized Confusion Matrix of Random Forest | **Fig. 81.** ROC Curve of Random Forest |

### Decision Tree

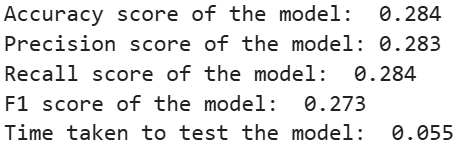


**Fig. 82.** Decision Tree performance evaluation

The Decision Tree model was evaluated using several metrics, showing strong results. The model achieved an accuracy of 0.951, indicating that it correctly predicts 95.1% of total instances, reflecting overall excellent performance. Precision and recall were also at 0.951, highlighting the model's ability to accurately identify true positives while minimizing false positives and false negatives. An F1 score of 0.951 demonstrates a balanced performance between precision and recall, ensuring effectiveness even in imbalanced data cases. Additionally, this model only took 0.094 seconds to generate predictions on test data, showcasing its efficiency and computational speed. Overall, these results indicate that the Decision Tree model is highly effective, reliable, and efficient in predicting stress levels.

|  |  |  |
| --- | --- | --- |
| **Fig. 83.** Normalized Confusion Matrix of Decision Tree | **Fig. 84.** Non-Normalized Confusion Matrix of Decision Tree | **Fig. 85.** ROC Curve of Decision Tree |

### Logistic Regression

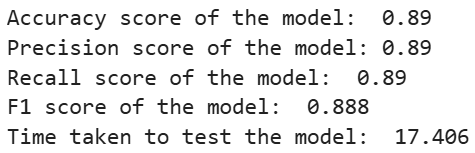


**Fig. 86.** Logistic Regression performance evaluation

The performance of the Logistic Regression model was evaluated using several metrics, and the results indicate that the model's performance is not optimal. This model achieved an accuracy of 0.284, which means it correctly predicts only 28.4% of total instances, significantly below the acceptable threshold for reliable performance. Precision and recall are both at 0.283 and 0.284, respectively, reflecting the model's limited ability to accurately identify true positives while minimizing false positives and false negatives. The F1 score of 0.273 further highlights this imbalance, indicating that the model struggles to maintain a good balance between precision and recall. However, the model only takes 0.055 seconds to make predictions on test data, demonstrating high computational efficiency and speed. Despite the fast prediction time, these results suggest that the Logistic Regression model is not effective or reliable in prediction for this particular dataset.

|  |  |  |
| --- | --- | --- |
| **Fig. 87.** Normalized Confusion Matrix of Logistic Regression | **Fig. 88.** Non-Normalized Confusion Matrix of Logistic Regression | **Fig. 89.** ROC Curve of Logistic Regression |

### K-Nearest Neighbors (KNN)

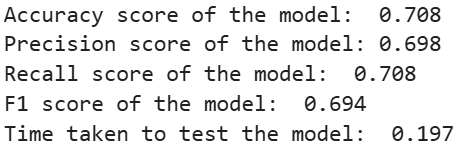


**Fig. 90.** KNN performance evaluation

The performance of the K-Nearest Neighbors (KNN) model was evaluated using several metrics, showing solid results. This model achieved an accuracy of 0.89, meaning it correctly predicted 89% of total instances, demonstrating strong overall performance. Precision and recall were also at 0.89, highlighting the model's ability to accurately identify true positives while minimizing false positives and false negatives. The F1 score of 0.888 indicates a balanced performance between precision and recall, ensuring effective handling of imbalanced data. However, this model took 17.406 seconds to generate predictions on the test data, indicating that it is computationally intensive and relatively slower compared to other models. Overall, these results suggest that the KNN model is effective and reliable in predicting stress levels, although it may require more computational resources and time to achieve results.

|  |  |  |
| --- | --- | --- |
| **Fig. 91.** Normalized Confusion Matrix of KNN | **Fig. 92.** Non-Normalized Confusion Matrix of KNN | **Fig. 93.** ROC Curve of KNN |

### Multilayer Perceptron (MLP)

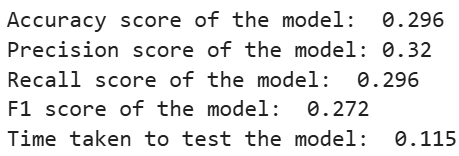


**Fig. 94.** MLP performance evaluation

The performance of the Multi-Layer Perceptron (MLP) model was evaluated using several metrics, showing moderate results. The model achieved an accuracy of 0.708, which means it correctly predicts 70.8% of total instances, indicating overall fair performance. Precision is at 0.698, demonstrating the model's ability to correctly identify true positives from all positive predictions. Recall is at 0.708, highlighting the model's ability to detect true positive examples, although some false negatives may still occur. The F1 score of 0.694 provides a balance between precision and recall, indicating that this model performs reasonably well but can be improved. The model takes 0.197 seconds to generate predictions on test data, showing good computational efficiency and speed. Overall, these results suggest that while the MLP model is effective in predicting stress levels, there is room for improvement to enhance precision and recall.

|  |  |  |
| --- | --- | --- |
| **Fig. 95.** Normalized Confusion Matrix of MLP | **Fig. 96.** Non-Normalized Confusion Matrix of MLP | **Fig. 97.** ROC Curve of MLP |

### Gaussian Naïve Bayes (GaussianNB)

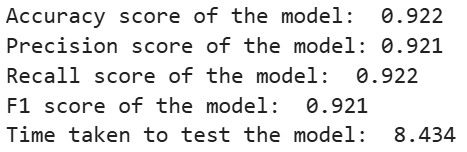


**Fig. 98.** GaussianNB performance evaluation

The performance of the Gaussian Naive Bayes (GaussianNB) model was evaluated using several metrics, and the results indicate that the model's performance is not optimal. This model achieved an accuracy of 0.296, which means it correctly predicted only 29.6% of total instances, significantly below the acceptable threshold for reliable performance. The precision is at 0.32, indicating that 32% of positive predictions are correct. The recall is at 0.296, highlighting the model's limited ability to detect true positives while missing a significant number of them. The F1 score of 0.272 further illustrates this imbalance, showing that the model struggles to maintain a good balance between precision and recall. However, the model only requires 0.115 seconds to make predictions on test data, demonstrating high computational efficiency and speed. Despite its fast prediction time, these results suggest that the GaussianNB model is not effective or reliable.

|  |  |  |
| --- | --- | --- |
| **Fig. 99.** Normalized Confusion Matrix of GaussianNB | **Fig. 100.** Non-Normalized Confusion Matrix of GaussianNB | **Fig. 101.** ROC Curve of GaussianNB |

### Tensorflow FNN



**Fig. 102.** Tensorflow FNN performance evaluation

The performance of the TensorFlow Sequential Feedforward Neural Network (FNN) model was evaluated using several metrics, and the results indicate strong performance. The model achieved an accuracy of 0.922, meaning it correctly predicted 92.2% of total instances, demonstrating overall excellent performance. Precision and recall were both at 0.921 and 0.922, highlighting the model's ability to accurately identify true positives while minimizing false positives and false negatives. An F1 score of 0.921 indicates a balanced performance between precision and recall, ensuring the effectiveness of this model even with potentially imbalanced data. However, this model took 8.434 seconds to make predictions on test data, which is relatively slower compared to other models, reflecting the computational complexity of training and inference for neural networks. Overall, these results show that the Sequential TensorFlow FNN model is highly effective and reliable.

|  |  |  |
| --- | --- | --- |
| **Fig. 103.** Normalized Confusion Matrix of FNN | **Fig. 104.** Non-Normalized Confusion Matrix of FNN | **Fig. 105.** ROC Curve of FNN |