Technical Report

CS5079

Applied AI

Assignment 2: Individual Assessment

Author:
Doroteya Stoyanova

December 19, 2024

Contents

1 Task 1:Gradient Boosting Algorithms: XGBoost vs LightGBM

1.1 Introduction

Situational awareness(SA) is important to improve takeover performance during the transition from automated to manual driving, as it aids in identifying potential hazards promptly and quickly. This task focuses on predicting SA during the takeover transition phase in conditionally automated driving using eye-tracking and self-reported data.

1.2 Exploratory data analysis

1.2.1 Data Import

First of all, there is a need to import the dataset in our environment. That can be done easily by importing the csv file that is provided.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt


data = pd.read_csv('SAdata_allMeasures.csv')
```

1.2.2 Data Summary

The dataset contains 1054 entries with 29 features, of which 28 predictive variables and a target variable 'Y', as it can be seen in Table

Table 1: Categories of the Features

Category	Features
Demographic	age, gender, yearDriving, drivingFrequency
Temporal	temp_length, temp_decisiontime, temp_decision_made, temp_correct_decision, temp_danger, temp_difficulty
Car Placement	CarPlacedLeft, CarPlacedRight
Visual	nums, sAmpMean, sAmpStd, sAmpMax, numf, fMean, 1Std, fMax, backMirror, leftMirror, rightMirror, road, sky
Pupil Dilation	pupilChange, pupilMean, pupilStd

As shown in Table 1, the dataset contains various feature categories. These include demographic features such as age, gender, and driving history. Temporal features capture information about the timing of decisions, including the time taken, decision-making speed, accuracy, perceived danger, and task difficulty. Additionally, visual cues are recorded, such as numerical values, signal amplitudes, measurements from various mirrors (e.g., back, left, and right), along with road and sky observations. Finally, pupil dilation metrics, including changes in pupil size, mean, and standard deviation, are also included in the dataset.

- Initially, the dataset is examined for null values, and since none were found, no preprocessing was necessary.

```
        print(data.isnull().sum())
```

- Secondly, certain features seemed to represent a category, like gender; however, after examining the data with the info command, it was found that among the 29 elements, 11 were floats while the remainder were integers.

```
       data.info()
        dtypes: float64(11), int64(18)
memory usage: 238.9 KB
```

- The data's statistics were also evaluated to determine if scaling was needed for the parameters. It was found that the youngest driver is 22 years old and the oldest is 29. Driving experience ranges from 16 to 22 years. The temp_decisiontime column indicated a minimum of roughly 0.49 seconds and a maximum of around 4.99 seconds for the time taken to make decisions.

```
        data.describe()
```

1.2.3 Visualisations

The mini correlation heatmap in Figure 9 shows the relationships between the following important variables, such as time length, decision time, correct decision time, and driving frequency. These features appear to have the strongest correlations with other variables related to driver behaviour and performance, which are crucial for understanding situational awareness during the transition from automated to manual driving.

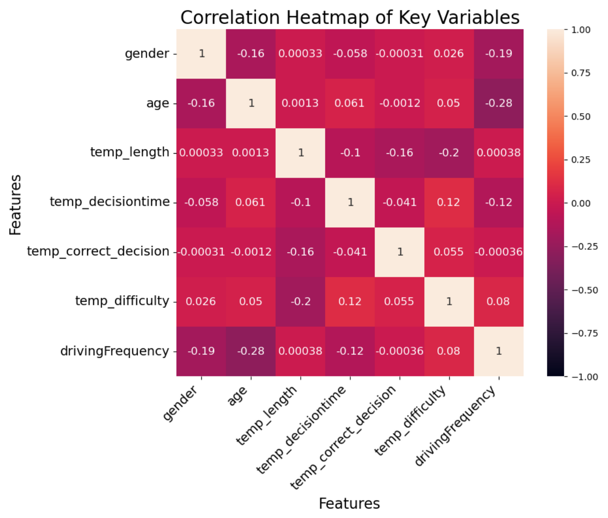
1.3 Data Preprocessing

Given the complexity of the data, a simple Linear Regression may not suffice. Instead, more advanced techniques such as **XGBoost** and **LightGBM** will be employed to create a robust regression model capable of handling intricate patterns and relationships within the dataset.

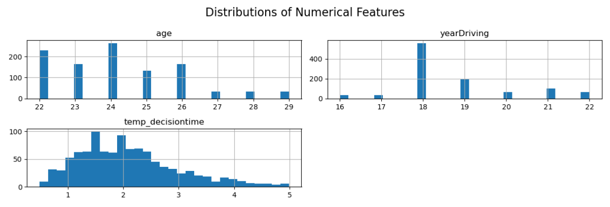
1.3.1 Splitting the data

At first, the features (X) are separated from the target variable, indicated by y. The dataset is split into training, testing, and validation subsets through train_test_split, designating 10% for test data, 70% for training, and 20% for validation approximately.

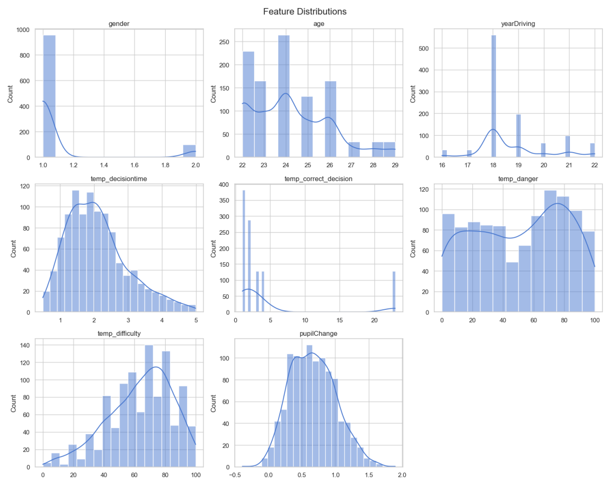
The data was to be preprocessed using StandardScaler, but gradient boosting techniques tend to perform effectively with datasets of different scales, and such processing was not done. As mentioned earlier, it was checked for missing values and categorical data.



(a) Important features



(b) Analysing Age, Years of Driving and Decision Time



(c) Distribution of data

Figure 1: Combined analysis of feature importance, key variables, and data distribution

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
   test_size=0.3, random_state=42)



X_temp, X_test, y_temp, y_test = train_test_split(X, y,
   test_size=0.1, random_state=42)


X_train, X_val, y_train, y_val = train_test_split(X_temp,
   y_temp, test_size=0.22222, random_state=42)

Dataset Sizes:
Training set: 737 samples
Testing set: 106 samples
Val set: 211 samples
```

1.4 Models Development

Gradient boosting is a robust ensemble machine learning method that incrementally constructs predictive models by merging several weak learners, usually decision trees. This approach aims to reduce a defined loss function, enabling it to learn adaptively from the mistakes of earlier models [11].

1.4.1 XGBoost

Model Architecture The architecture of XGBoost is designed to provide a highly efficient, flexible, and accurate gradient-boosting framework. At its core, XGBoost builds an ensemble of decision trees, where each tree corrects the errors made by its predecessors. This sequential boosting approach ensures that the model incrementally improves its predictions by minimizing a specified loss function, such as Mean Squared Error for regression tasks [3].

XGBoost incorporates several innovations to enhance its performance. It employs a regularized objective function that balances model complexity and predictive accuracy by including both L1 (Lasso) and L2 (Ridge) regularization terms. This helps prevent overfitting and promotes better generalization. The framework also uses second-order gradients, which provide a more accurate approximation of the loss function and contribute to faster convergence during training.

Another critical feature of the XGBoost architecture is its ability to handle sparse data efficiently. It employs a sparsity-aware algorithm that can automatically detect and handle missing values during training, improving its applicability to real-world datasets. Additionally, XGBoost supports parallel tree construction and optimization techniques, such as column subsampling and histogram-based splitting, to achieve faster training times and scalability.

Finally, the architecture allows for hyperparameter tuning of tree-specific parameters, including the depth of trees, learning rate, subsampling ratios, and the number of estimators, enabling fine-grained control over model performance. Together, these features make XGBoost a powerful and versatile model for a wide range of machine learning tasks (Fig. 2).

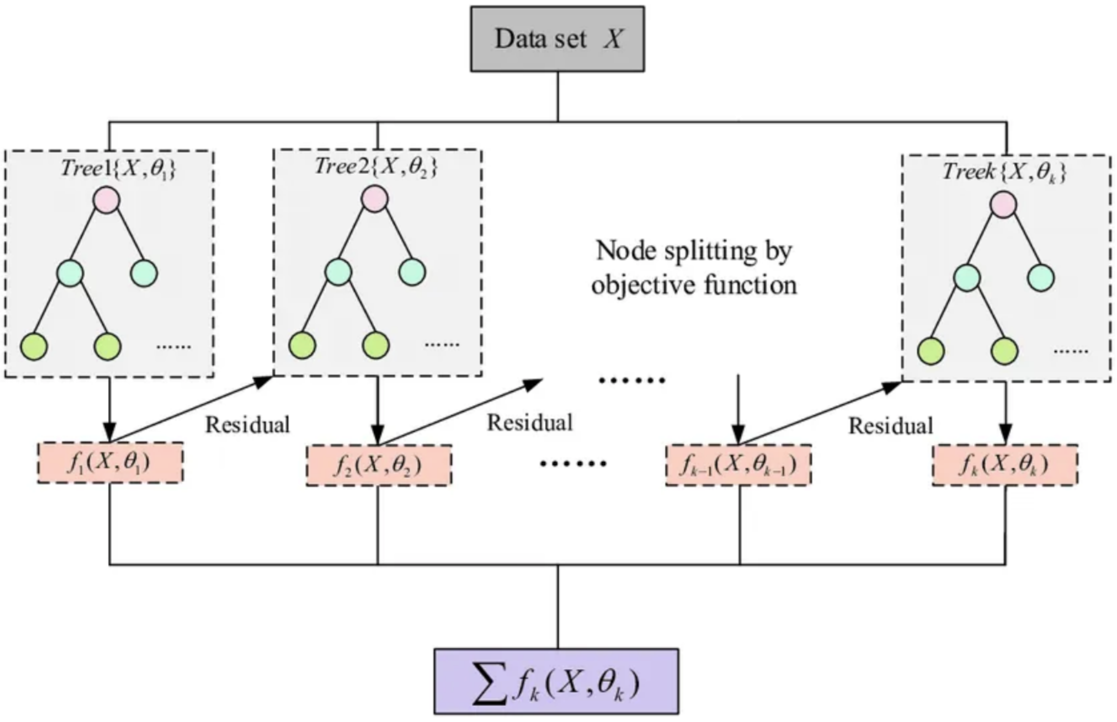


Figure 2: Xgboost Regression Architecture

Implementation The implementation of the XGBoost model involves several stages: data preparation, hyperparameter tuning, training, and evaluation. The process is structured to maximize the predictive performance while optimizing for computation time. The steps are detailed as follows.

Fine-tuning First the data was split in test,train and validation. Afterwards the hyperparameter tuning is performed using a predefined parameter grid, which includes a range of values for key hyperparameters such as learning_rate, max_depth, n_estimators, subsample, colsample_bytree, min_child_weight, gamma, reg_alpha, and reg_lambda. To efficiently search for the best combination of parameters, the RandomizedSearchCV method is employed, significantly reducing computational expense compared to exhaustive grid search. A 10-fold cross-validation strategy is integrated into this process to evaluate the performance of different hyperparameter combinations, ensuring the model is trained on diverse data splits. For training, the optimal hyperparameter configuration identified during the tuning stage is used to fit the XGBoost model. The training process monitors the performance on both training and validation datasets, allowing adjustments to prevent overfitting. The final model is implemented using XGBRegressor, which is configured with an objective function of reg:squarederror for regression tasks, a fixed random state for reproducibility, and n_jobs set to -1 to fully utilize available computational resources.

```
from sklearn.model_selection import RandomizedSearchCV, KFold
from sklearn.metrics import mean_squared_error,
    mean_absolute_error, r2_score,root_mean_squared_error
import xgboost as xgb
import time
```

```
def xgboost_training(X_train, X_val, X_test, y_train, y_val,
   y_test):
    np.random.seed(42)

    param_grid = {
        'learning_rate': [0.01, 0.05, 0.001],
        'max_depth': [2, 3, 4],
        'n_estimators': [200, 500,1000],
        'subsample': [0.6, 0.7, 0.8,0.1,0.2],
        'colsample_bytree': [ 0.5,0.1,0.7,0.5],
        'min_child_weight': [2, 3, 5, 7],
        'gamma': [1,10,5,0.1],
        'reg_alpha': [ 0,0.1],
        'reg_lambda': [ 1, 5]
    }

    kfold = KFold(n_splits=10, shuffle=True, random_state=42)

    xg_reg = xgb.XGBRegressor(
        objective='reg:squarederror',
        random_state=42,
        n_jobs=-1
    )

    random_search = RandomizedSearchCV(
        estimator=xg_reg,
        param_distributions=param_grid,
        n_iter=200,
        cv=kfold,
        scoring='neg_mean_squared_error',
        verbose=2,
        n_jobs=-1,
        random_state=42
    )

    start_training_time = time.time()

    # Fitting RandomizedSearchCV
    random_search.fit(
        X_train, y_train,
        eval_set=[(X_train, y_train), (X_val, y_val)],  #
   Validation set
        verbose=False
    )
```

Model Selection After the Random search the best parameters were identified.

Experimentation Results Given the regression nature of the task, four key metrics were evaluated: Mean Squared Error (MSE), Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and the R^2 Score. Additionally, training and prediction times

Table 2: Hyperparameter Grid for XGBoost Model

Hyperparameter	**Values**
Learning Rate	0.01, 0.05, 0.001
Max Depth	2, 3, 4
Number of Estimators	200, 500, 1000
Subsample	0.6, 0.7, 0.8, 0.1, 0.2
Colsample by Tree	0.5, 0.1, 0.7
Min Child Weight	2, 3, 5, 7
Gamma	1, 10, 5, 0.1
Reg Alpha	0, 0.1
Reg Lambda	1, 5

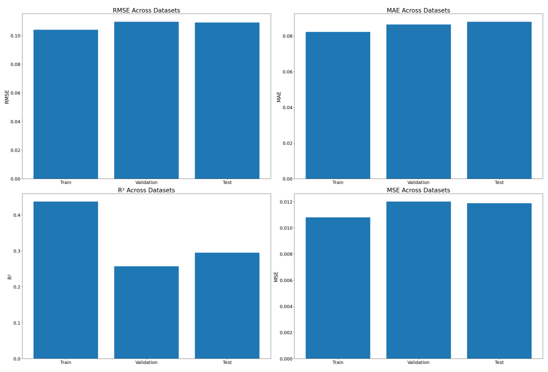
Table 3: Final Hyperparameters for XGBoost Model

Hyperparameter	**Value**
Subsample	0.8
Reg Lambda	5
Reg Alpha	0
Number of Estimators	1000
Min Child Weight	7
Max Depth	4
Learning Rate	0.05
Gamma	0.1
Colsample by Tree	0.5

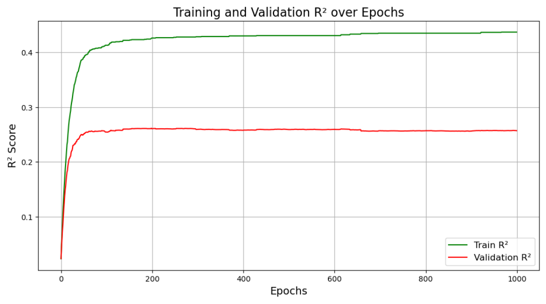
were measured to assess the computational efficiency of the model.

Table 4: Summary of Error Metrics and Computational Times

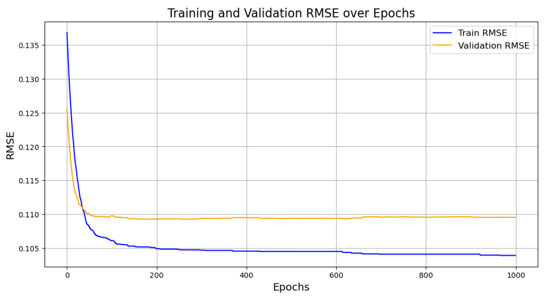
Metric/Aspect	**Description**
MSE	Measures the average squared difference between predicted and actual values, emphasizing larger deviations.
MAE (Mean Absolute Error)	Represents the average absolute difference between predicted and actual values, providing an interpretable scale.
RMSE	The square root of MSE, expressed in the same units as the target variable, improving interpretability for practical applications.
R2	Indicates the proportion of variance explained by the model, offering a measure of predictive power.
Training Time	Efficient learning during hyperparameter tuning and cross-validation.
Prediction Time	Demonstrates the model's scalability for real-time applications.



(a) Error Scores for Train, Validation, and Test Sets



(b) R2 Scores for Train and Validation over Epoch



(c) RMSE Scores for Train and Validation over Epoch

Figure 3: Error Metrics (MAE, MSE, RMSE, R^2) for XGBoost Across Train, Validation, and Test Sets

1.4.2 LightGBM

LightGBM (Light Gradient Boosting Machine) is a high-performance, distributed, and scalable implementation of gradient boosting, specifically designed for efficient training of large-scale datasets. Unlike traditional gradient boosting methods, LightGBM optimizes both memory usage and computation time, making it suitable for tasks involving large datasets, such as machine learning competitions and production environments. There will be two strategies for Lightgbm: creating a distributed version using Dask, and developing a standard model. The general model will follow a similar approach as the Xgboost method, in terms of methodology, with the only difference that the hyperparameter tuning is done with different parameters.

Architecture LightGBM employs a leaf-wise tree growth strategy instead of the traditional level-wise approach. This means that it grows trees by selecting the leaf node that provides the maximum reduction in the loss function, leading to deeper and more expressive trees. This method allows for faster training as it expands fewer nodes compared to level-wise growth, which splits all nodes at the same level before moving deeper. However, this can increase the risk of overfitting, which can be mitigated through regularization techniques (Fig. 4).
In regression tasks, LightGBM builds an ensemble of decision trees sequentially. Each tree is trained to predict the residuals of the previous trees' predictions, thereby improving overall accuracy. Gradient-based One-Side Sampling (GOSS) technique is one of the

Table 5: Training, Validation, and Test Metrics for XGBoost

Aspect	Train	Validation	Test	Overall
MSE	0.0108	0.0120	0.0119	–
RMSE	0.1039	0.1095	0.1090	–
MAE	0.0822	0.0864	0.0879	–
R^2 Score	0.4370	0.2570	0.2948	–
Training Time (s)	–	–	–	297.66
Prediction Time (s)	–	–	–	0.04

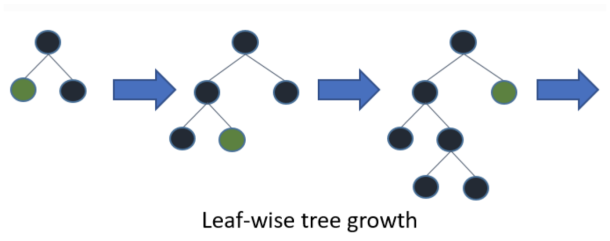


Figure 4: Light GBM approach

architectures available in the framework. It is an innovation in LightGBM that optimizes training by selectively sampling data points based on their gradients.
During training, instances with small gradients are typically well-trained and thus retain less information for further learning. In contrast, those with large gradients are under-trained and more informative. GOSS keeps all instances with large gradients and randomly samples a subset of instances with small gradients, thereby maintaining the distribution of the data while reducing computational overhead. This allows LightGBM to focus on the most relevant data points for improving model accuracy.

Implementation The implementation follows a comprehensive approach similar to XGBoost, involving data preparation, hyperparameter tuning, training, and evaluation, except that the parameters to be tuned were different, such as boosting type,

```
    from sklearn.model_selection import RandomizedSearchCV, KFold
from sklearn.metrics import mean_squared_error,
   mean_absolute_error, r2_score
import lightgbm as lgb
import time

def lightgbm_training(X_train, X_val, X_test, y_train, y_val,
   y_test):
    np.random.seed(42)

    param_grid = {
    'boosting_type': ['goss'],
    'n_estimators': [50, 100, 200],
    'num_leaves': [31, 50, 75,100, 200, 300],
    'learning_rate': [0.05, 0.01, 0.001],
    #'feature_fraction': [0.9, 0.8],
    #'bagging_freq': [5, 7, 8],
}


    kfold = KFold(n_splits=10, shuffle=True, random_state=42)

    lgb_reg = lgb.LGBMRegressor(
        objective='regression',
        metrics= ['l2', 'l1'],
        random_state=42,
        n_jobs=-1
    )

    random_search = RandomizedSearchCV(
        estimator=lgb_reg,
        param_distributions=param_grid,
        n_iter=200,
        cv=kfold,
        scoring='neg_mean_squared_error',
        verbose=2,
        n_jobs=-1,
        random_state=42
    )

    start_training_time = time.time()

    random_search.fit(
        X_train, y_train,
        eval_set=[(X_train, y_train), (X_val, y_val)]
    )


    training_time = time.time() - start_training_time

    best_model = random_search.best_estimator_


    start_pred_time = time.time()
```

The RandomSearchCV was applied with K-fold in a manner similar to the earlier approach. Following the training, the ultimate parameters and the most effective method

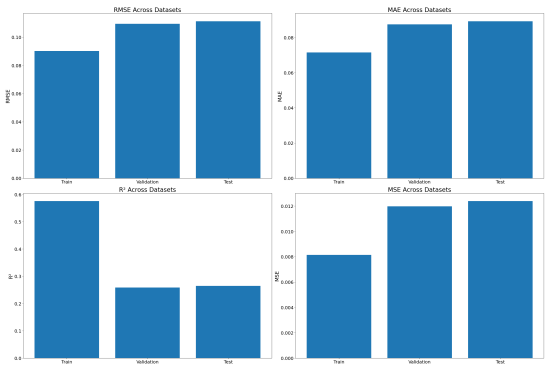
Parameter	**Values**
boosting_type	'goss'
n_estimators	50, 100, 200
num_leaves	31, 50, 75, 100, 200, 300
learning_rate	0.05, 0.01, 0.001

Table 6: Hyperparameters for LightGBM

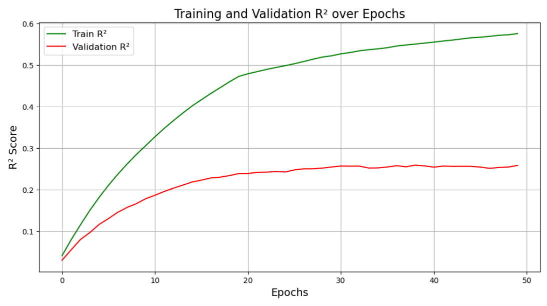
were determined. Although additional parameters were evaluated, a research paper on a related subject indicated that these specific parameters yielded the best results [19].

Parameter	**Best Value**
num_leaves	31
n_estimators	50
learning_rate	0.05
boosting_type	'goss'

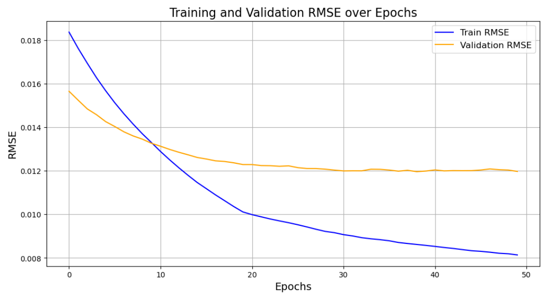
Table 7: Best Hyperparameters for LightGBM and Results



(a) Error Scores for Train, Validation, and Test Sets



(b) R2 Scores for Train and Validation over Epoch



(c) RMSE Scores for Train and Validation over Epoch

Figure 5: Error Metrics (MAE, MSE, RMSE, R^2) for LightGBM Across Train, Validation, and Test Sets

Table 8: Training, Validation, and Test Metrics for LightGBM

Aspect	**Train**	**Validation**	**Test**	**Overall**
MSE	0.0081	0.01197	0.01239	–
RMSE	0.0902	0.1094	0.1113	–
MAE	0.0714	0.0875	0.0892	–
R^2 Score	0.5754	0.2586	0.2648	–
Training Time (s)	–	–	–	468.09
Prediction Time (s)	–	–	–	0.01

Distributed Version When utilizing only a CPU-machine, the LightGBM technique may perform more slowly, according to research [15]. A LightGBM distributed version was employed with the Dask library. Dask is a Python framework for parallel computing

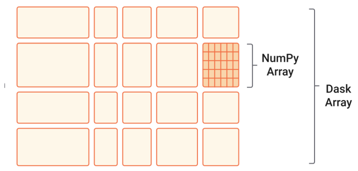


Figure 6: Dask Array

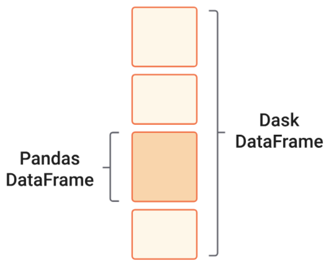


Figure 7: Dask DataFrame

that allows for scalable and distributed data processing. It is especially advantageous for handling extensive datasets and executing intricate calculations on multicore systems or throughout machine clusters. Dask offers a collection of high-level interfaces for handling parallelized data structures and computations, such as Dask arrays and dataframes.

- Dask Arrays implement a subset of the NumPy ndarray interface, allowing operations on large arrays that may not fit into memory. They achieve this by breaking arrays into smaller chunks that can be processed in parallel across multiple cores or distributed systems. This lazy evaluation model means computations are executed only when explicitly requested (e.g., by calling .compute())(Fig.6).

- Dask DataFrames extend the capabilities of pandas DataFrames by partitioning data into smaller pandas DataFrames (known as partitions) that can be processed in parallel. Like Dask Arrays, they support lazy evaluation, meaning operations are not computed until a compute request is made. This structure enables users to perform typical pandas operations—such as filtering, aggregating, and joining—on datasets that are too large to fit into memory while retaining a familiar API (Fig.7.)

Distributed LightGBM Implementation Implementing Distributed LightGBM on a MacBook required several considerations due to hardware limitations. The MacBook lacks a dedicated GPU or high-performance cluster, so the model had to be adapted to fit the available resources while still enabling efficient training.

To optimize CPU usage, Dask was employed to facilitate parallel computing across multiple threads. However, due to the MacBook's hardware constraints, the handling of data during training needed to be adjusted. Although DaskLGBMRegressor is designed to work directly with Dask DataFrames, certain operations, such as cross-validation using KFold, required conversion of Dask DataFrames into pandas DataFrames for compatibility. This conversion was essential because KFold expects pandas or NumPy arrays, not Dask DataFrames.

After generating the cross-validation folds, the data was converted back into Dask DataFrames for training with LightGBM. This dual approach—using pandas for cross-validation and Dask for model training—was chosen to balance the memory and processing limitations of the MacBook while still leveraging distributed computing capabilities.

This method, while not fully utilizing Dask's potential for distributed computation, allowed for efficient model training and evaluation on the available hardware, ensuring that large datasets could still be processed without memory overflow.

For the implementation of distributed training with LightGBM, the Dask-ML integration, specifically the DaskLGBMRegressor, was used. This integration allows LightGBM to utilize parallel computing resources, speeding up the training process and enabling the model to handle larger datasets than a single machine could process. The Dask cluster was initialized with 4 workers, each with 2 threads, leveraging the MacBook's multi-core architecture. The dataset was split into training, validation, and test sets using pandas, and Dask DataFrames were created from these pandas DataFrames, partitioning the data into smaller chunks for parallel processing. Cross-validation was performed using KFold with 10 splits, where the model was trained on the training subset and evaluated on the validation subset. Metrics such as RMSE, MSE, MAE, and R^2 were calculated for both the training and validation sets, and training and prediction times for each fold were recorded.

Metrics	**Dask LightGBM**
Average Training Metrics	
MSE	0.00856
RMSE	0.0925
MAE	0.0731
R^2	0.5533
Average Validation Metrics	
MSE	0.0135
RMSE	0.1159
MAE	0.0911
R^2	0.2823
Average Training Time per Fold	0.64 seconds
Average Prediction Time per Fold	0.12 seconds
Test Metrics	
MSE	0.0117
RMSE	0.1082
MAE	0.0862
R^2	0.3058

Table 9: Dask LightGBM Model Performance Metrics per fold

```
    from dask.distributed import Client, LocalCluster
import dask.dataframe as dd
from sklearn.metrics import mean_squared_error, r2_score,
   mean_absolute_error,root_mean_squared_error
from lightgbm.dask import DaskLGBMRegressor
from sklearn.model_selection import train_test_split, KFold
import time
import numpy as np
import matplotlib.pyplot as plt

# Initializing dask client and cluster
cluster = LocalCluster(n_workers=4, threads_per_worker=2)
client = Client(cluster)


# 80% train and 20% test
X_train, X_test, y_train, y_test = train_test_split(X, y,
   test_size=0.2, random_state=42)


# split off the test set (10%)
X_temp, X_test, y_temp, y_test = train_test_split(X, y,
   test_size=0.1, random_state=42)

# split the remaining data into training (70%) and validation
   (20%)
X_train, X_val, y_train, y_val = train_test_split(X_temp,
   y_temp, test_size=0.22222, random_state=42)


# Convert pandas to to Dask DataFrames
X_train_dask = dd.from_pandas(X_train.reset_index(drop=True),
   npartitions=100)
y_train_dask = dd.from_pandas(y_train.reset_index(drop=True),
   npartitions=100)
X_val_dask = dd.from_pandas(X_val.reset_index(drop=True),
   npartitions=100)
y_val_dask = dd.from_pandas(y_val.reset_index(drop=True),
   npartitions=100)
X_test_dask = dd.from_pandas(X_test.reset_index(drop=True),
   npartitions=100)
y_test_dask = dd.from_pandas(y_test.reset_index(drop=True),
   npartitions=100)

def calculate_metrics(y_true, y_pred):
    mse = mean_squared_error(y_true, y_pred)
    rmse = root_mean_squared_error(y_true, y_pred)
    mae = mean_absolute_error(y_true, y_pred)
    r2 = r2_score(y_true, y_pred)
    return {'MSE': mse, 'RMSE': rmse, 'MAE': mae, 'R2': r2}

def distributed_lightgbm_training(X_train_dask, y_train_dask):
    params = {
        'boosting_type': 'goss',
        'n_estimators': 50,
        'num_leaves': 31,
        'learning_rate': 0.05,
        'objective': 'regression',
        'metric': 'rmse',
        'tree_learner': "data_parallel",
        'random_state': 42,
    }

    lgb_reg = DaskLGBMRegressor(**params)

    kf = KFold(n_splits=10, shuffle=True, random_state=42)
    train_metrics_list = []
    val_metrics_list = []
    training_times = []
    prediction_times = []

    eval_results_list = []  # store evaluation results for each
fold

    # Convert Dask DataFrames to Pandas for KFold splitting
    X_train_pd = X_train_dask.compute()
    y_train_pd = y_train_dask.compute()

    for fold, (train_index, val_index) in
enumerate(kf.split(X_train_pd)):
        print(f"Training fold {fold + 1}")

        # Use iloc to select rows based on indices
        X_train_fold = X_train_pd.iloc[train_index]
        y_train_fold = y_train_pd.iloc[train_index]
        X_val_fold = X_train_pd.iloc[val_index]
        y_val_fold = y_train_pd.iloc[val_index]

        # Convert back to Dask DataFrames for training
        X_train_fold_dask =
dd.from_pandas(X_train_fold.reset_index(drop=True),
npartitions=4)
        y_train_fold_dask =
dd.from_pandas(y_train_fold.reset_index(drop=True),
npartitions=4)
        X_val_fold_dask =
dd.from_pandas(X_val_fold.reset_index(drop=True),
npartitions=4)
        y_val_fold_dask =
dd.from_pandas(y_val_fold.reset_index(drop=True),
npartitions=4)

        start_training_time = time.time()

        # Fit the model and store eval results
        lgb_reg.fit(X_train_fold_dask,
                    y_train_fold_dask,
                    eval_set=[(X_val_fold_dask,
y_val_fold_dask)],
                    eval_metric='rmse')
        training_time = time.time() - start_training_time
        training_times.append(training_time)

        start_pred_time = time.time()
```

1.4.3 LightGBM vs Distributed LightGBM

When applying the distributed Dask version, the best LightGBM metrics were selected, and tested with the same k-folds. While the metric results are the same, the parallel method is faster in terms of speed.

1.5 Discussion

Fine-tuning parameters for both XGBoost and LightGBM was essential to balance the trade-off between model complexity and generalization, which directly impacts performance. For XGBoost, key parameters like `max_depth`, `learning_rate`, `subsample`, and `colsample_bytree` were fine-tuned to optimize tree complexity and control overfitting. A higher `max_depth` allows the model to capture more intricate patterns but increases the risk of overfitting, particularly on smaller datasets. By limiting the depth and introducing subsampling (e.g., `subsample` and `colsample_bytree`), the model was forced to focus on subsets of data, improving generalization. Regularization terms (`reg_alpha` and `reg_lambda`) further constrained model complexity, addressing overfitting by penalizing large weights 10.

For LightGBM, parameters such as `num_leaves`, `learning_rate`, and `boosting_type` were critical in fine-tuning its performance. The leaf-wise tree growth strategy in LightGBM inherently risks overfitting, especially with high `num_leaves` values, as it allows deeper and more specific splits. To mitigate this, lower `num_leaves` values were explored, coupled with a smaller `learning_rate` to ensure incremental updates and stability in training. The `boosting_type` parameter was set to GOSS to prioritize informative data points, reducing computational overhead while maintaining accuracy.

The observed overfitting in both models, despite fine-tuning, can be logically justified by their architectural tendencies and dataset characteristics. XGBoost's level-wise growth tends to overfit when the dataset is small and the trees are deep, even with regularization. Similarly, LightGBM's leaf-wise approach can lead to overly complex trees, especially when the `num_leaves` parameter is not sufficiently constrained. This overfitting highlights the importance of further parameter tuning, such as adjusting learning rates and introducing early stopping, to improve generalization.

The experimental results highlighted distinct strengths and weaknesses of both models. XGBoost achieved slightly lower RMSE and MSE on the test set, demonstrating strong generalization capabilities. However, its validation R^2 score was slightly lower than LightGBM.

The results reveal distinct strengths and weaknesses for XGBoost and LightGBM in the task. XGBoost achieved better generalization, as evidenced by its lower test MSE (0.0119) and RMSE (0.1090) compared to LightGBM (MSE: 0.0124, RMSE: 0.1113), alongside a higher test R^2 score (0.2948 vs. 0.2648). This performance can be attributed to XGBoost's controlled tree depth (`max_depth: 4`), robust regularization (`reg_alpha` and `reg_lambda`), and gradual learning through a small `learning_rate: 0.05` combined

Parameter	Model	Role	Impact on Overfitting
`max_depth`	XGBoost	Limits the depth of trees to control complexity.	Higher values risk overfitting by modeling noise; lower values reduce the risk but may underfit.
`learning_rate`	Both	Controls step size for updates in boosting.	Smaller values prevent overfitting by ensuring gradual learning but require more estimators.
`subsample`	XGBoost	Randomly samples rows for each tree.	Reduces correlation among trees, lowering the risk of overfitting.
`colsample_bytree`	XGBoost	Randomly samples columns (features) for each tree.	Introduces randomness, improving generalization.
`reg_alpha`	XGBoost	L1 regularization to penalize large coefficients.	Prevents overfitting by enforcing sparsity in the model.
`reg_lambda`	XGBoost	L2 regularization to penalize large weights uniformly.	Reduces model complexity to avoid overfitting.
`n_estimators`	Both	Specifies the number of boosting iterations (trees).	More estimators risk overfitting unless regularized effectively.
`num_leaves`	LightGBM	Sets the maximum number of leaf nodes in a tree.	Higher values capture finer patterns but increase overfitting risk.
`boosting_type`	LightGBM	Specifies the boosting strategy (e.g., GOSS for gradient-based sampling).	GOSS emphasizes informative samples, which can mitigate overfitting on noisy data.
`min_data_in_leaf`	LightGBM	Minimum number of samples required in a leaf.	Higher values prevent overly specific splits, reducing overfitting.
`lambda_l1`	LightGBM	L1 regularization to penalize leaf complexity.	Helps prevent overfitting by enforcing sparsity.
`lambda_l2`	LightGBM	L2 regularization to penalize overall tree weights.	Constrains model flexibility to reduce overfitting.

Table 10: Key Parameters for XGBoost and LightGBM, Their Roles, and Impact on Overfitting

with a high number of estimators (`n_estimators: 1000`). These settings effectively balanced the bias-variance tradeoff, preventing overfitting while capturing relevant patterns in the data. In contrast, LightGBM demonstrated faster prediction times (0.01 seconds vs. 0.04 seconds for XGBoost) and higher training R^2 (0.5754 vs. 0.4370), reflecting its ability to model complex relationships. However, LightGBM's leaf-wise growth strategy and a high number of leaves (`num_leaves:31`) made it more prone to overfitting, as shown by its slightly higher test errors. LightGBM also had longer training times (468.09 seconds vs.297.66 seconds for XGBoost), likely due to its deeper tree structure and fewer estimators (`n_estimators:50`), and the fact that XgBoost could have better time-perfromance when trained on a CPU. The distributed LightGBM was really fast - less than 10 seconds for both training time and prediction time. The experimental results for XGBoost and LightGBM models reveal suboptimal performance on the dataset, primarily due to its limited size and high dimensionality. Both models exhibited signs of overfitting for the squared R, a common challenge when dealing with datasets characterized by a high feature-to-sample ratio. XGBoost demonstrated marginally superior generalization capabilities, achieving a test R^2 of 0.2948 compared to LightGBM's 0.2648. This slight advantage suggests XGBoost was somewhat more adept at handling the dataset's inherent complexities. However, the relatively low R^2 scores for both models indicate their limited ability to explain the variance in the target variable. The overfitting issue is particularly evident when comparing training and test performance. LightGBM, for instance, showed a substantial performance degradation from a training R^2 of 0.5754 to a test R^2 of 0.2648, underscoring a significant reduction in predictive power on unseen data. These findings highlight the challenges posed by the dataset's characteristics and emphasize the need for careful model selection and parameter tuning when working with high-dimensional, small-sample datasets in regression tasks. 2.

1.5.1 Model Overfitting prevention strategies

To further enhance the performance of the models, several strategies can be employed. First, more extensive **hyperparameter tuning** is crucial, especially for key parameters such as `num_leaves` in LightGBM and `max_depth` in XGBoost. For LightGBM, exploring a broader range of `num_leaves` values (e.g., 31, 50, 75, 100, 200, 300) will help strike the optimal balance between model complexity and generalization.Similarly, for XGBoost, experimenting with various `max_depth` values (e.g., 2, 3, 4) will help control tree depth and prevent overfitting by limiting the model's capacity to memorize data.

Utilizing **feature importance** methods available in both methods can also enhance model performance. By identifying and focusing on the most influential features, it is possible to reduce the dimensionality of the data and potentially eliminate irrelevant or redundant features. This simplification can help the model focus on the most predictive aspects of the data, improving both performance and interpretability.

Finally, **ensemble methods** can further improve performance by combining their strengths. Techniques like *simple averaging* or *stacking* can leverage the complementary advantages of both models, potentially leading to a more robust final prediction. By blending the models' predictions, we can reduce individual model biases and variance, ultimately improving overall performance.

2 Task 2:Explaimable AI using SHAP

2.1 Introduction

The second portion of the assignment entails using Explainable AI methodology, like SHAP (SHapley Additive exPlanations) to clarify the predictions generated by two machine learning models: Ridge Regression and LightGBM. The dataset given for this analysis shows a non-linear characteristic, which requires the use of sophisticated interpretability methods such as SHAP to accurately comprehend the model results.

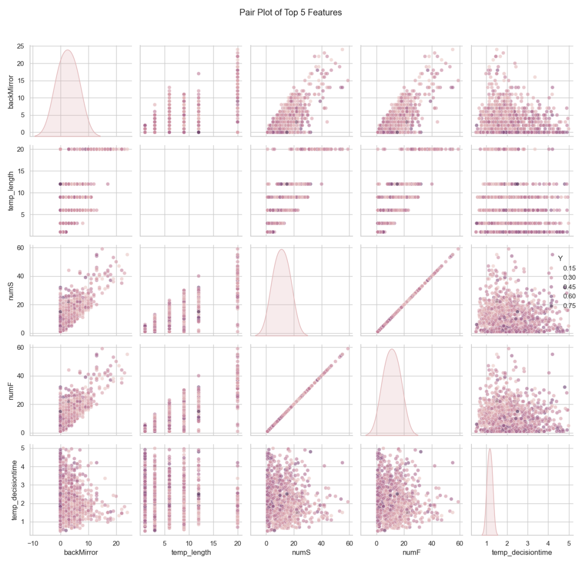
```
data = pd.read_csv('SAdata_allMeasures.csv')
```

```
print(data.head())
print(data.isnull().sum())
data.info()
X = data.drop(columns=['Y'])
y = data['Y']
```

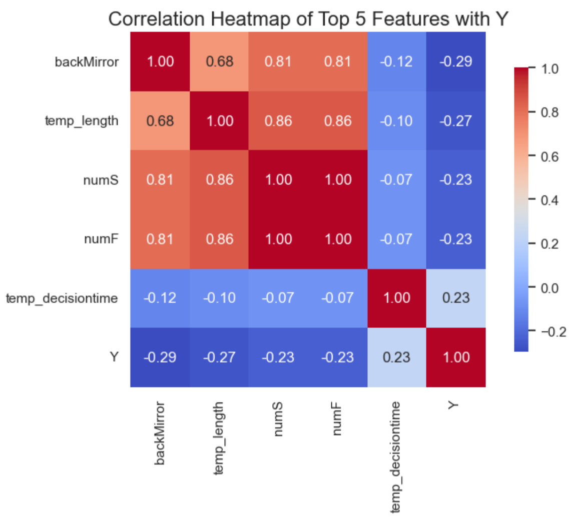
2.2 Exploratory data analysis

2.2.1 Data Visualisations

In this iteration of the analysis, the focus remains on the same dataset from Task 1, continuing with null value checks and information verification, but with a heightened emphasis on preprocessing. The preprocessing strategy centres on sophisticated feature selection techniques, identifying the top 20 features based on their correlation with the target variable (Y).



(a) Pair plot of top features



	backMirror	temp_length	numS	numF	temp_decisiontime	Y
backMirror	1.00	0.68	0.81	0.81	-0.12	-0.29
temp_length	0.68	1.00	0.86	0.86	-0.10	-0.27
numS	0.81	0.86	1.00	1.00	-0.07	-0.23
numF	0.81	0.86	1.00	1.00	-0.07	-0.23
temp_decisiontime	-0.12	-0.10	-0.07	-0.07	1.00	0.23
Y	-0.29	-0.27	-0.23	-0.23	0.23	1.00

(b) Correlation heatmap of Top 5 Features

Figure 8: Exploratory data analysis visualizations

```
    correlation =
   data.corr()['Y'].abs().sort_values(ascending=False)
print(correlation)
top_features = correlation[1:20].index.tolist()
X = data[top_features]
y = data['Y']
# Step 1: Split off the test set (20%)
X_temp, X_test, y_temp, y_test = train_test_split(X, y,
   test_size=0.2, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_temp,
   y_temp, test_size=0.125, random_state=42)
# Scale the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_val = scaler.transform(X_val)
```

```
X_test = scaler.transform(X_test)
# Print the shapes to verify the split
print("Training set shape:", X_train.shape)
print("Validation set shape:", X_val.shape)
print("Test set shape:", X_test.shape)
```

2.2.2 Preprocessing

To scale the dataset after splitting it into training (70%), validation (10%), and test (20%) sets, first train_test_split was used to separate the data, then fit a StandardScaler on the training set and transform both the validation and test sets using the same scaler.

2.3 Explainable AI (XAI) methodologies

Understanding model interpretation is essential for analyzing the decision-making processes of machine learning models. Analytical methods such as feature importance, LIME, SHAP, ICE curves, and Anchors enable researchers to gain insights into model predictions, feature contributions, and overall transparency [18, 4].

- **Feature importance** quantifies the influence of features on predictions. Tree-based models provide relative importance scores, while regression models use coefficients to indicate feature direction and strength [14]. These insights help prioritize impactful features.
- **LIME (Local Interpretable Model-agnostic Explanations)** generates interpretable approximations of model behavior around specific data points, making it especially useful for complex models like XGBoost and random forests [7].
- **SHAP** offers a game-theoretic framework to calculate feature contributions to predictions, providing both local and global interpretability. Its visualizations, such as Summary and Force Plots, simplify complex relationships [8].
- **ICE (Individual Conditional Expectation) Curve** visualize the effect of a single feature on predictions by plotting outputs for individual instances while keeping other features constant. This highlights heterogeneous and non-linear relationships in the data [13].
- **Anchors** generate “if-then” rules to explain predictions by identifying consistent conditions under which the model’s outputs remain unchanged, offering intuitive and reliable explanations [9].

2.3.1 Advantages of SHAP

SHAP stands out for its foundation in Shapley values, ensuring fair and consistent feature contribution measurements across diverse models. It excels in sensitive applications like finance and healthcare, where interpretability is critical [16]. SHAP’s dual interpretability—providing local explanations and global insights—enhances understanding of both individual predictions and overall model behavior [8]. Its model-agnostic nature and intuitive visualizations further increase its accessibility and versatility, making it a valuable tool for uncovering feature interactions and complex model dynamics [9].

2.4 Model Selection

For this task, Ridge Regression is selected to compare against LightGBM. While Ridge Regression is inherently a linear model, it can effectively approximate non-linear relationships through feature transformations, such as polynomial features or kernel mappings. Ridge Regression introduces an L2 regularization term that minimizes overfitting by penalizing large coefficients, resulting in a more robust model for datasets with multicollinearity or noisy features.The model is particularly effective in scenarios where the features are highly correlated, as the regularization term minimizes the impact of multicollinearity and improves model stability.It can also handles high-dimensional data efficiently, ensuring reliable performance even when the number of features is large relative to the number of observations.

2.5 Implementation

The sklearn's Ridge Regression model was implemented with hyperparameter tuning using `RandomizedSearchCV`. The hyperparameter grid for the `alpha` parameter was defined using a logarithmic scale from 10^{-4} to 10^4. A `KFold` cross-validation strategy with 10 splits was employed to ensure proper model evaluation. Once the model was trained, predictions were made on the training, validation, and test datasets. The performance of the model was evaluated using multiple metrics, including Mean Squared Error (MSE), Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and the R2 score. These metrics were computed for each dataset to assess the model's performance and generalization capability. `RandomizedSearchCV` was used to randomly sample from the hyperparameter grid, and the best model was selected based on the negative mean squared error.

Metrics	Train	Validation	Test
MSE	0.01442	0.01455	0.01491
RMSE	0.1201	0.1206	0.1221
MAE	0.0958	0.0982	0.0971
R^2	0.2477	0.0991	0.1149
Best α	65.7933		
Training Time (s)	6.43		
Prediction Time (s)	0.00		

Table 11: Performance Metrics for Ridge Regression Model

```
def ridge_regression_training(X_train, X_val, X_test, y_train,
   y_val, y_test):
    np.random.seed(42)


    param_grid = {
        'alpha': np.logspace(-4, 4, 100)
    }

    kfold = KFold(n_splits=10, shuffle=True, random_state=42)
    ridge_reg = Ridge(random_state=42)
    # RandomizedSearchCV for hyperparameter tuning
    random_search = RandomizedSearchCV(
     Same as the previous implementations
    )
    start_training_time = time.time()
    # Fitting RandomizedSearchCV
    random_search.fit(X_train, y_train)
```

2.6 SHAP analysis

First of all for the development of SHAP, the python library shap was used. To check ridge regression, LinearExplainer was used, while for the LightGBM - TreeExplainer.

```
    import shap
# Ensure model and X_train_scaled are defined
explainer = shap.LinearExplainer(best_model, X_train)
shap_values = explainer.shap_values(X_test)
explainer = shap.TreeExplainer(best_model, X_train)
shap_values = explainer.shap_values(X_test)
shap_values
shap_values
```

To visualize and interpret the impact of features on the model's predictions, a SHAP summary plot is used. Additionally, to quantify and display the global importance of features, a SHAP (SHapley Additive exPlanations) bar plot is employed.

```
  X_test_scaled_df = pd.DataFrame(X_test, columns=feature_names)
shap.summary_plot(shap_values, X_test_scaled_df, max_display=29)
```

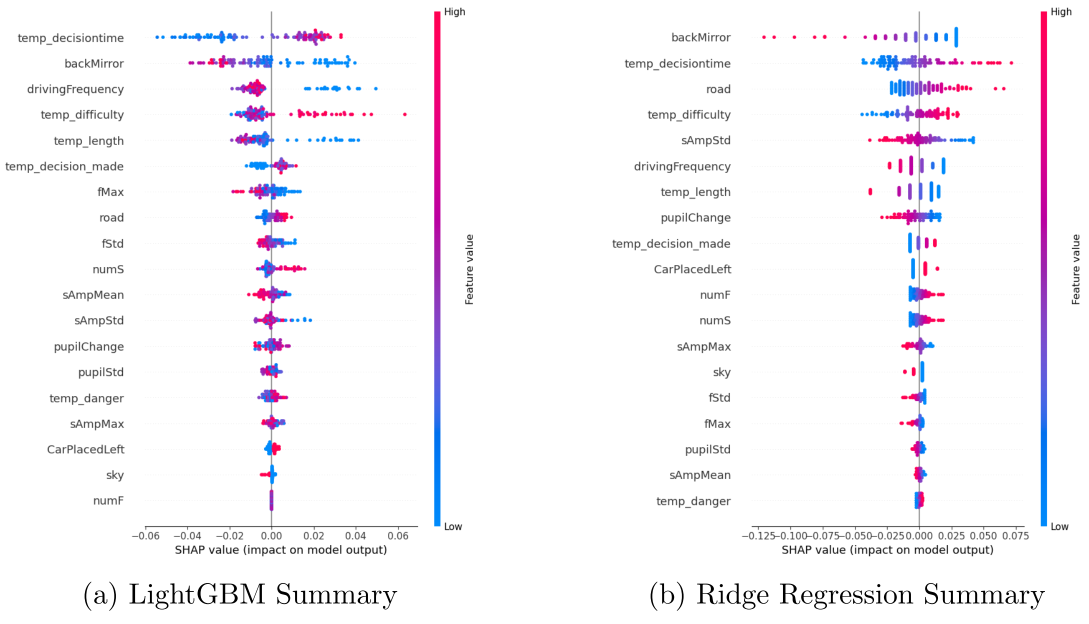


Figure 9: Comparative analysis with SHAP for LightGBM and Ridge Regression models.

2.6.1 Feature importance and analysis

Analyzing the SHAP value plots reveals key influential features in driving decision-making. The `temp_decisiontime` variable shows high importance across both models, highlighting how decision timing crucially impacts driving safety. This makes sense as decision time directly measures how quickly a driver reacts to a situation, with quicker decision-making often correlating with better reaction times and potentially safer driving. The `backMirror` feature demonstrates significant influence, particularly with extreme negative values in the non-tree model, emphasizing the critical nature of mirror-checking behavior. Mirror usage is a key safety measure, as frequent checks of the rear-view mirror are indicative of a driver's situational awareness. This aligns with the intuition that drivers who check their mirrors more often are likely to make safer decisions. `DrivingFrequency` and `temp_difficulty` also show substantial impact, indicating that both experience and situation complexity affect decisions. Drivers who have more experience are likely to make better decisions in complex situations, which is reflected in the importance of these variables. The difficulty of a driving task also affects decision-making, as more difficult situations demand more careful and deliberate decisions. These variables are consistent with the broader understanding of factors influencing driver behaviour, as they reflect both mental and physical states, as well as external driving conditions [**fig:combined˙analysis**]. Comparing the models shows interesting differences: the non-tree-based model assigns more extreme SHAP values (0.1250.125 to 0.0750.075) versus LightGBM's narrower range (0.060.06 to 0.060.06), and places higher emphasis on `backMirror` and road conditions. The non-tree model appears more sensitive to feature variations, which could result from its linear approach. In contrast, LightGBM shows smoother value distributions, reflecting its ability to capture non-linear relationships in the data. These variations highlight how different modeling approaches can provide complementary insights into driving behavior patterns. While the non-tree model may overemphasize certain features like `backMirror`, LightGBM's ability to model more complex relationships between features allows it to offer a broader and potentially more nuanced understanding of driving behavior.

3 Task 3: Healthcare data condsiderations

3.1 Ethical Considerations

Ethical factors for data scientists managing medical datasets need thoughtful focus on several crucial aspects. The primary concern is patient privacy and confidentiality—data scientists are required to adhere rigorously to regulations such as GDPR and HIPAA when managing sensitive medical data [6]. This involves appropriate data anonymization and safe management of personal health data [5]. Informed consent is a vital aspect, requiring patients to fully comprehend and consent to the usage of their data while ensuring their right to revoke that consent is honored [10].

Data security represents another essential ethical factor, necessitating strong protocols for storage, transmission, and access management. Data quality and integrity should be upheld by consistent validation and verification procedures [10].

3.2 The steps to unbiased model and the fairness requirements:

To maintain fairness standards and guarantee an impartial model, various essential measures need to be taken [12]. Thorough pre-processing is crucial to tackle inherent biases in the data, for example demographic disparities in heart disease occurrence across age categories or genders [1]. Resampling methods can assist in forming a more balanced dataset, and consistent performance assessments among demographic groups can reveal any variations in accuracy [1]. Integrating explainable AI techniques aids in elucidating decision-making processes, making sure that predictions are not excessively swayed by isolated factors such as age,sex or ethnicity [18].

3.3 Should a model reduce bias to achieve fair results? Or not?

In healthcare, machine learning models must reduce bias to achieve fair results. Skewed predictions can worsen health inequalities, resulting in incorrect diagnoses or insufficient treatment suggestions for marginalized groups [17]. For instance, if a heart disease model places too much importance on aspects like age or specific demographic traits, it may unjustly forecast a greater risk for particular groups, possibly leading to discrimination [17]. Nevertheless, in certain situations, a managed bias might be permissible—for example, if particular age demographics are statistically more susceptible to heart disease, concentrating on these demographics could be sensible. Nonetheless, any application of bias must be thoroughly validated and supported by medical evidence rather than by assumptions, guaranteeing that the model's predictions accurately represent risk factors while maintaining fairness [2].
Word Count for task 3:346

References

[1] Riccardo Cau et al. "Addressing hidden risks: Systematic review of artificial intelligence biases across racial and ethnic groups in cardiovascular diseases". In: *European Journal of Radiology* (2024), p. 111867.

[2] Richard J Chen et al. "Algorithmic fairness in artificial intelligence for medicine and healthcare". In: *Nature biomedical engineering* 7.6 (2023), pp. 719–742.

[3] Tianqi Chen and Carlos Guestrin. "Xgboost: A scalable tree boosting system". In: *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 2016, pp. 785–794.

[4] Rudresh Dwivedi et al. "Explainable AI (XAI): Core ideas, techniques, and solutions". In: *ACM Computing Surveys* 55.9 (2023), pp. 1–33.

[5] Khaled El Emam and Luk Arbuckle. *Anonymizing health data: case studies and methods to get you started.* " O'Reilly Media, Inc.", 2013.

[6] Nehal Ettaloui, Sara Arezki, and Taoufiq Gadi. "An Overview of blockchain-based electronic health record and compliance with GDPR and HIPAA". In: (2023), pp. 405–412.

[7] Damien Garreau and Ulrike Luxburg. "Explaining the explainer: A first theoretical analysis of LIME". In: *International conference on artificial intelligence and statistics*. PMLR. 2020, pp. 1287–1296.

[8] Anshul Goel. "Model Exploitability using SHAP (SHapley Additive exPlanations) and LIME (Local Interpretable Model-agnostic Explanations)". In: *Medium* (2021). URL: https://medium.com/@anshulgoel991/model-exploitability-using-shap-shapley-additive-explanations-and-lime-local-interpretable-cb4f5594fc1a.

[9] Andreas Holzinger et al. "Explainable AI methods-a brief overview". In: *International workshop on extending explainable AI beyond deep models and classifiers*. Springer. 2022, pp. 13–38.

[10] Katya Kaplow et al. "Data professionals' attitudes on data privacy, sharing, and consent in healthcare and research". In: *Digital health* 10 (2024), p. 20552076241290964.

[11] Andrei V Konstantinov and Lev V Utkin. "Interpretable machine learning with an ensemble of gradient boosting machines". In: *Knowledge-Based Systems* 222 (2021), p. 106993.

[12] John P Lalor et al. "Should fairness be a metric or a model? a model-based framework for assessing bias in machine learning pipelines". In: *ACM Transactions on Information Systems* 42.4 (2024), pp. 1–41.

[13] Abhishek Maheshwarappa. "Explainable AI with ICE (Individual Conditional Expectation Plots)". In: *Medium* (2021). URL: https://abhishek-maheshwarappa.medium.com/explainable-ai-with-ice-individual-conditional-expectation-plots-c71e8fc1f1c2.

[14] Machine Learning Mastery. *How to Calculate Feature Importance With Python*. Accessed: 2024-12-15. Aug. 2020. URL: https://machinelearningmastery.com/calculate-feature-importance-with-python/.

[15] Neptune.ai. *XGBoost vs LightGBM: A Detailed Comparison*. Accessed: 2024-12-19. 2024. URL: https://neptune.ai/blog/xgboost-vs-lightgbm.

[16] Yasunobu Nohara et al. "Explanation of machine learning models using shapley additive explanation and application for real data in hospital". In: *Computer Methods and Programs in Biomedicine* 214 (2022), p. 106584.

[17] Oriel Perets et al. "Inherent Bias in Electronic Health Records: A Scoping Review of Sources of Bias". In: *medRxiv* (2024).

[18] Feiyu Xu et al. "Explainable AI: A brief survey on history, research areas, approaches and challenges". In: *Natural language processing and Chinese computing: 8th cCF international conference, NLPCC 2019, dunhuang, China, October 9–14, 2019, proceedings, part II 8*. Springer. 2019, pp. 563–574.

[19] Feng Zhou, X Jessie Yang, and Joost CF De Winter. "Using eye-tracking data to predict situation awareness in real time during takeover transitions in conditionally automated driving". In: *IEEE Transactions on Intelligent Transportation Systems* 23.3 (2021), pp. 2284–2295.