Ira A. Fulton Schools of Engineering
Arizona State University

MAE 598: Introduction to Autonomous Vehicle Engineering

Final Project Report

Predictive Modeling for Tilt Control: Data Science Approach to Enhanced Ride Comfort

Instructor:

Ronald Calhoun

Submitted by:

Sherry Daniel Sajan (1225838460)

Swetha Tirumala (1229953818)

VVS Krishna Vamshi (1229568654)

Table of Contents

Table of Figures

1 Problem Statement

This project aims to develop machine learning models for predicting vehicle tilt angles during maneuvers through curves and turns, enhancing passenger comfort. Additionally, the project seeks to evaluate and compare the performance of these machine learning models to discern optimal methodologies.

2 Introduction

With the growing widespread adoption of autonomous vehicles, it becomes necessary to ensure passengers are comfortable and safe during the rides. Passengers often face motion sickness and uneasiness when the vehicle takes turns due to the lateral acceleration. The concept of curve tilting was introduced to reduce the effect of motion sickness on the passengers. During this, the vehicle is made to tilt by a specific angle to counteract the centrifugal forces acting on the vehicle during curved roads.

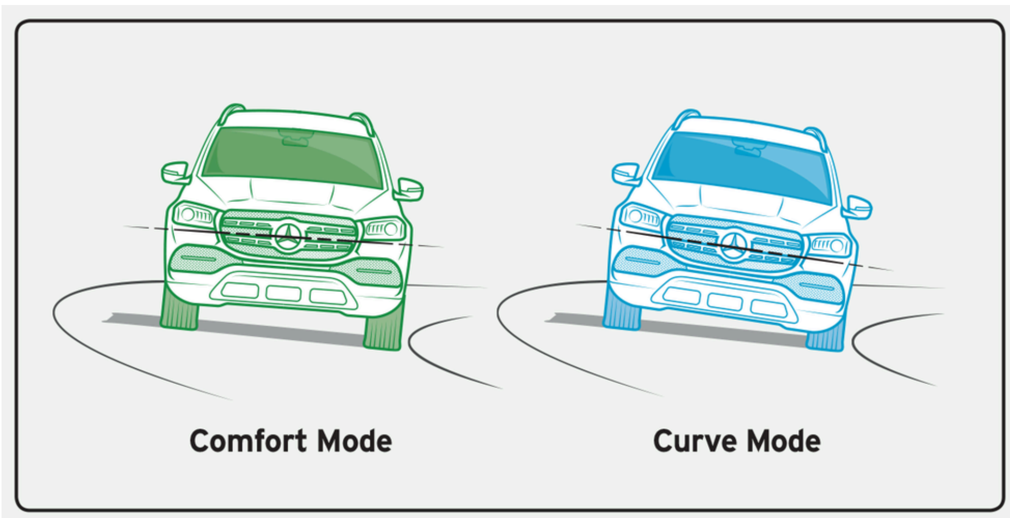

Fig.1. Mercedes curve tilting function

There are various methods to calculate the vehicle's tilt angle, including TGMS (track geometry monitoring systems), which monitors the curvature of the path and generates the optimal tilt angle using the information taken from the sensors for the upcoming curved track. GPS (global positioning system) and GIS (Geographic Information system) provide information which can be combined and tilt angle can be predicted for the upcoming curves. Similarly, certain machine learning algorithms can be used to recognise and understand the pattern and predict the tilt angle value. However, the most fundamental and standard method is to use the vehicle dynamics and calculate the tilt angle mathematically.

3 Advantages

If the formulas give us the absolute tilt angle, it begs the question why would someone use these algorithms in the first place? There are a few reasons why neural networks work better than the dynamics in certain situations.

- Complexity of the relationship: The formula-based tilt angle might be more accurate, but the neural network has the ability to capture the complex relationships between the inputs and the outputs. A neural network might generate more accurate predictions if the relationship between the inputs and the tilt angle becomes highly nonlinear or involves interactions between features that the formula cannot capture.
- Data-Driven Learning: Since neural networks are data-driven, they can adjust to various patterns and trends seen in the training set. In the event that variables are not specifically mentioned in the formula but found in the data affect the tilt angle, a neural network might be able to identify these correlations and make more accurate predictions.
- Generalization: If trained appropriately, neural networks have the capacity to generalize well to previously unknown data. Although the formula might work well given the particular inputs used to develop it, it would not work as well when applied to new, unknown scenarios as a neural network trained on a wider range of data.
- Processing of Noisy Data: Unlike simple algorithms, neural networks are more efficient at processing noisy or faulty data. When noise or uncertainty exists in the data, a neural network may be more resilient to these fluctuations.

4 Solution Approach

In this project, a dataset obtained from the CARLA simulator has been used to train a neural net model and a decision tree model to predict the tilt angle. Furthermore, these algorithms have been analyzed and compared based on different aspects. CARLA(Car Learning to Act) is an open source simulator used in the research and development of autonomous vehicles. It offers an adaptable and realistic setting for exploring and testing different autonomous driving-related models, algorithms, and strategies. CARLA, created by the Barcelona Supercomputing Center (BSC) and the Computer Vision Center (CVC), is becoming popular as a useful tool for developing autonomous vehicles.

5 Methodology

The step-by-step process of the solution approach and the followed methodology is discussed in detail in the following sections.

5.1 Data collection using CARLA

CARLA - an open-source autonomous vehicle simulator was used to collect essential vehicle information. The CARLA simulator works based on Python; large in-built libraries and functions can be performed using the simulator. For this project, the data are the instantaneous velocity, acceleration, and the force acting on the driver/passenger. These are collected using the GNSS (Global Navigation Satellite System) and IMU (Inertial Measurement Unit) sensors integrated into CARLA.

The GNSS sensor uses satellite signals and provides information about the vehicle's global position, velocity and orientation. The accuracy is based on a few characteristics like noise level, update frequency and drift. The outputs of the GNSS sensor are latitude, longitude, altitude, velocity (speed and direction), and orientation (heading). On the other hand, the IMU sensor consists of measurements from accelerometers and gyroscopes. The data collected from the IMU sensor typically includes linear acceleration (in x, y, and z axes), angular velocity (in roll, pitch, and yaw axes), and orientation (roll, pitch, and yaw angles). This data can be accessed programmatically through the CARLA Python API.

```python
# Add GNSS sensor to ego vehicle.
gnss_bp = world.get_blueprint_library().find('sensor.other.gnss')
gnss_location = carla.Location(0, 0, 0)
gnss_rotation = carla.Rotation(0, 0, 0)
gnss_transform = carla.Transform(gnss_location, gnss_rotation)
gnss_bp.set_attribute("sensor_tick", str(3.0))
ego_gnss = world.spawn_actor(gnss_bp, gnss_transform, attach_to=ego_vehicle, attachment_type=carla.AttachmentType.Rigid)
def gnss_callback(gnss):
    print("GNSS measure:\n" + str(gnss) + '\n')
    with open('gnss_data.csv', mode='a') as file:
        writer = csv.writer(file)
        if file.tell() == 0:
            writer.writerow(['Frame', 'Latitude', 'Longitude', 'Altitude'])
        writer.writerow([gnss.frame, gnss.latitude, gnss.longitude, gnss.altitude])
ego_gnss.listen(lambda gnss: gnss_callback(gnss))

# Add IMU sensor to ego vehicle.
imu_bp = world.get_blueprint_library().find('sensor.other.imu')
imu_location = carla.Location(0, 0, 0)
imu_rotation = carla.Rotation(0, 0, 0)
imu_transform = carla.Transform(imu_location, imu_rotation)
imu_bp.set_attribute("sensor_tick", str(3.0))
ego_imu = world.spawn_actor(imu_bp, imu_transform, attach_to=ego_vehicle, attachment_type=carla.AttachmentType.Rigid)
def imu_callback(imu):
    print("IMU measure:\n" + str(imu) + '\n')
    with open('imu_data.csv', mode='a') as file:
        writer = csv.writer(file)
        if file.tell() == 0:
            writer.writerow(['Frame', 'Acceleration_x', 'Acceleration_y', 'Acceleration_z', 'Angular Velocity_x', 'Angular Velocity_y', 'Angular Velocity_z', 'Compass'])
        writer.writerow([imu.frame, imu.accelerometer.x, imu.accelerometer.y, imu.accelerometer.z, imu.gyroscope.x, imu.gyroscope.y, imu.gyroscope.z, imu.compass])
ego_imu.listen(lambda imu: imu_callback(imu))
```

Fig.2. Code snippet for data collection from GNSS and IMU sensors in CARLA python API

Firstly, the IMU sensor is added to the ego vehicle using CARLA's sensor blueprint, and the initial values of the sensor's location and rotation are set as (0,0,0). 'set_attribute("sensor_tick", str(3.0))' command sets the tick rate to 3Hz which means the data will be collected every ⅓ seconds. The 'imu_callback' function handles the data received from the sensor and prints it to the console every time the function is called. This data is then appended to a CSV file. Whenever the IMU sensor generates new data, the 'imu_callback' function is triggered and the data is added to the CSV file. The code for the GNSS sensor works similarly and thus stores the measurement in a separate CSV file.

5.2 Machine Learning Algorithms

Our tilt prediction model is trained using two machine learning algorithms: the feed-forward neural network and the decision tree. We utilize two different testing models to test each model's performance and determine which offers the best results. Although the dataset used to train the models is the same, their working and performance vary, which will be discussed in greater detail. The following sections provide an in-depth analysis of the structure and workings of each model.

5.2.1 Training the Neural Network

The data collected from the simulator serves as the dataset for training the neural network. The dataset is loaded and split into 80% training and 20% testing data, and the model is trained. The input layer takes a total of 4 features: speed, coefficient of friction, radius of curvature, and mass. There are 5 hidden layers wherein each hidden layer consists of 256 nodes, 128 nodes, and 64 nodes in the first two layers, the third layer, and the remaining three layers respectively as shown in the code. After training the model, the tilt angle is predicted and the accuracy and performance of the model are analyzed.

```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.layers import Dense
from tensorflow.keras.metrics import MeanAbsoluteError

# Load data from the CSV file
df = pd.read_csv('dataset_carla.csv')

# Preprocess the data (e.g., handle missing values, scale features, etc.)
# Example:
df.dropna(inplace=True)  # Drop rows with missing values
X = df.drop('Theta', axis=1)  # Features
y = df['Theta']  # Target variable

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Define neural network architecture
model = Sequential([
    Dense(256, activation='relu', input_shape=(X_train.shape[1],)),  # Input layer
    Dense(256, activation='relu'),
    Dense(128, activation='relu'),
    Dense(64, activation='relu'),
    Dense(64, activation='relu'),
    Dense(64, activation='relu'),# Hidden layers
    Dense(1, activation='linear')  # Output layer
])
```

```python
[2] # Compile the model
metrics=[MeanAbsoluteError()]
model.compile(optimizer='adam', loss='mean_absolute_error', metrics=metrics)

# Train the model
model.fit(X_train, y_train, batch_size=32, epochs=10, validation_data=(X_test, y_test))

# Save the trained model
model.save('tilt_model.h5')

# Function to predict theta value based on mass and radius of curvature
def predict_theta(model,speed, cof,radius_of_curvature, mass):
    # Prepare input data for prediction
    input_data = np.array([[speed, cof,radius_of_curvature,mass]])

    # Make prediction
    prediction = model.predict(input_data)

    return prediction[0][0]  # Assuming single prediction

# Example usage:
# Load the trained model
model = load_model('tilt_model.h5')

# Let's say you want to predict theta for mass=100 kg, radius_of_curvature=10 m, and additional features
mass = 51.6  # kg
radius_of_curvature = 2  # m
cof=0.8
speed=0.348
```

Fig.3. Code for training the neural net model

5.2.2 Training the decision tree

The data received from the CARLA simulator is fed as a dataset for training the decision tree. The dataset is loaded and preprocessed to remove any missing values and is split into features (X) and target ('Theta'). Then the data is visualized using Seaborn and Matplotlib for scatter plots of features against 'Theta'. The dataset is split into 80% training and 20% testing sets. After this, XGBoost regressor is trained by initializing with the following hyperparameters: no. of estimators = 100, learning rate = 0.1, tree depth = 4. After training, the model is evaluated on the set of test values. The trained model is loaded to predict 'Theta' on a new dataset and the accuracy and performance of the decision tree is analyzed using metrics like Mean absolute error, Mean square error.

```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.layers import Dense

# Load data from the CSV file
#df = pd.read_csv('auto_tiltdataset.csv')
df = pd.read_csv('dataset_carla.csv')

# Preprocess the data (e.g., handle missing values, scale features, etc.)
# Example:
df.dropna(inplace=True)  # Drop rows with missing values
X = df.drop('Theta', axis=1)  # Features
y = df['Theta']  # Target variable
df.head()
```

```python
model = xgb.XGBRegressor(n_estimators=200, learning_rate=0.1, max_depth=4)
model.fit(X_train, y_train)

[ ] # Make predictions on the test set
predictions = model.predict(X_test)

# Calculate mean squared error
mse = mean_squared_error(y_test, predictions)
print("Mean Squared Error:", mse)

Mean Squared Error: 0.00011585458257546127

[ ] mae = mean_absolute_error(y_test, predictions)
print("Mean Absolute Error:", mae)

# Calculate R-squared (R^2)
r2 = r2_score(y_test, predictions)
print("R-squared (R^2):", r2)

Mean Absolute Error: 0.005420124654287623
R-squared (R^2): 0.9992034854330977
```

Fig.4. Code for training the decision tree model

6 Results

Both the models were trained successfully and could predict the required tilt angle for the AV. The decision tree model perhaps performs better than the feed-forward neural network. The following plot shows the predictions made by both the models for a set of 50 inputs given to both the models, compared with the actual or standard value calculated using the vehicle's dynamics for the same set of inputs. The decision tree predictions are closer to the actual values than the ones given by the NN.

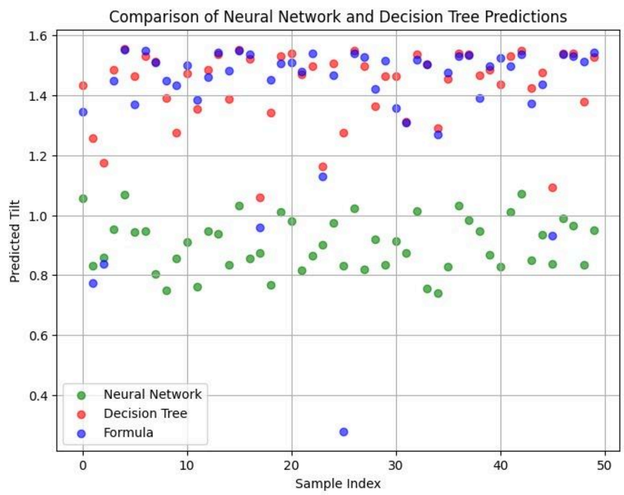

Fig.5. Neural Network Vs. Decision Tree

The absolute error of the predicted values was found and the plot can be observed. The decision tree exhibits fewer errors than the neural network. This could be predicted from the previous observation as the decision tree's outputs are closer to the standard values.

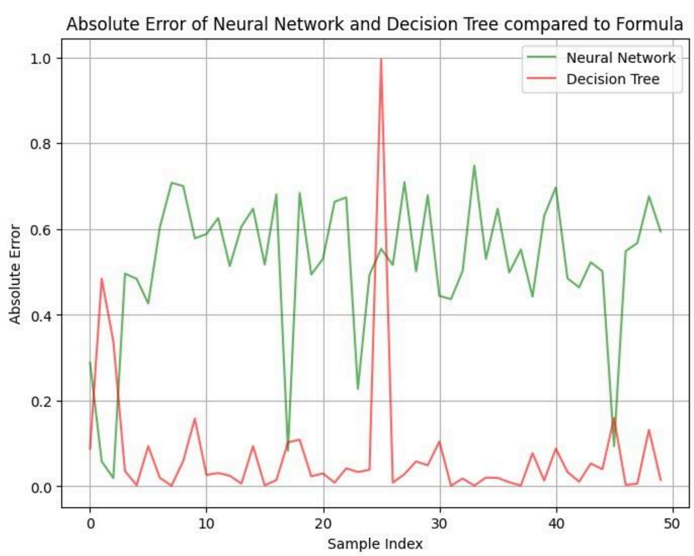

Fig.6. Absolute Error - NN vs. Decision Tree

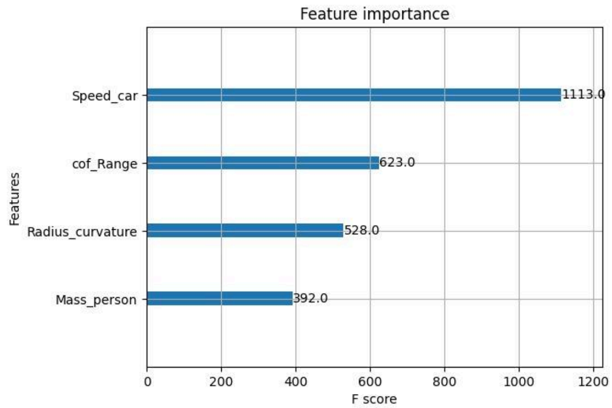

Fig.7. Feature importance graph - Decision Tree

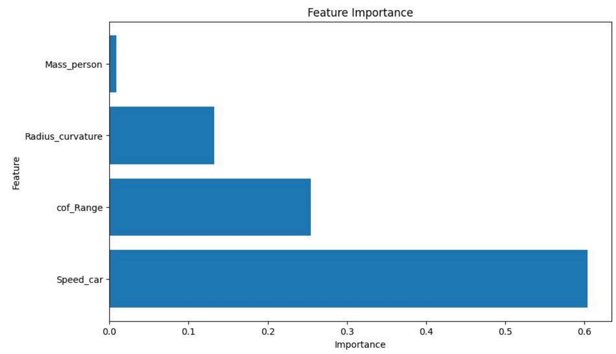

Fig.8. Feature importance graph - Decision Tree (Normalised)

A feature importance graph visually represents the importance of different features or input variables in a predictive model. It helps understand which features significantly influence the model's predictions and can aid in feature selection, interpretation, and model optimization. The car's speed and coefficient of friction are the factors that primarily affect the tilt angle in the decision tree model as seen in the figure above, followed by the radius of curvature and mass of the passenger. The figure shows the actual weights of each factor while the other figure is normalized and is within the range of 0-1. Similarly in the fig below, the coefficient of friction is the most significant factor in the NN model, followed by the car’s speed, passenger mass, and curvature radius. The difference between the outputs of both models could be due to the difference between the most significant factors and how they are computed.

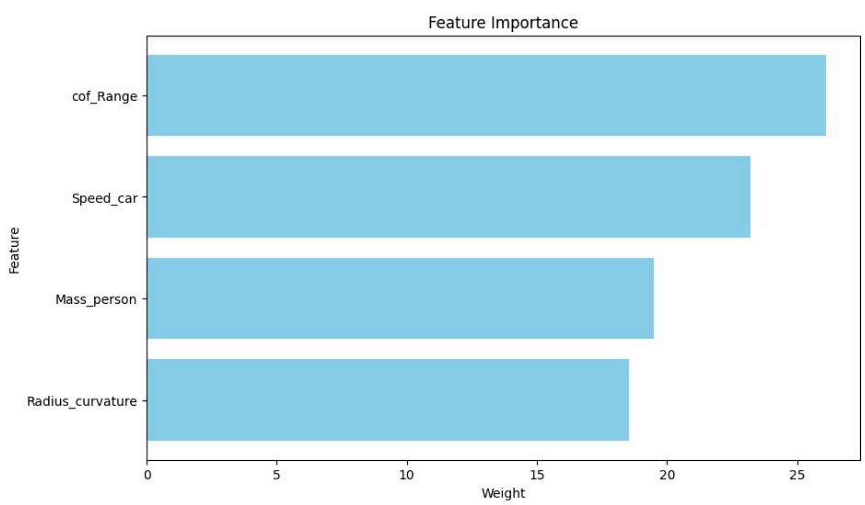

Fig.9. Feature importance graph - NN

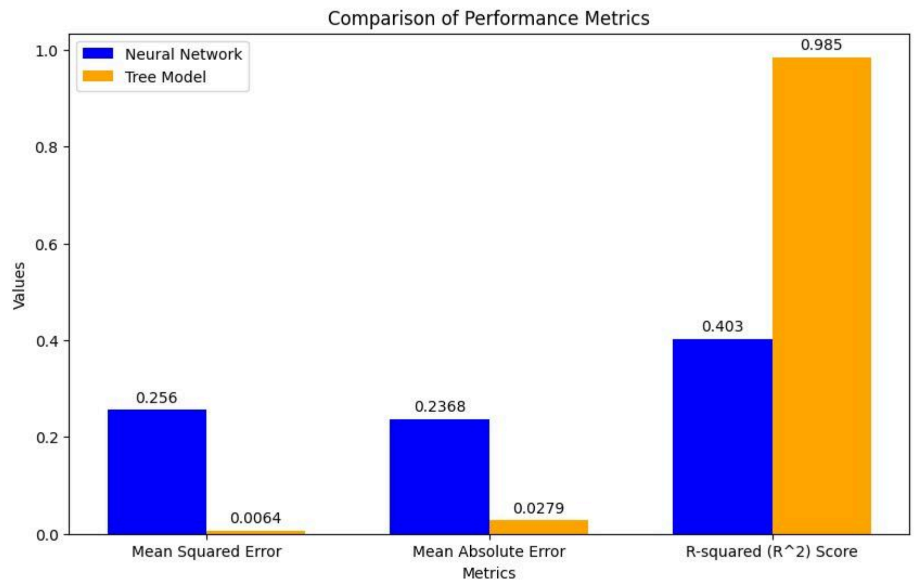

Fig.10. Performance Characteristics of the ML Models

Various performance characteristics of the models have been found and plotted as seen from the figure above. Namely mean squared error (MSE), mean absolute error, and R-squared score are studied. The MAE represents the average absolute differences between predicted and actual values. It measures the average magnitude of errors in a set of predictions, without considering their direction. Lower MAE values indicate better model performance, implying smaller errors between predictions and actual values. The MAE of NN was 0.2368, whereas that of the decision tree was 0.0279. It's almost a one-tenth difference. So, this indicates that the tree model is superior to the NN in terms of MAE.

The MSE on the other hand calculates the average of the squares of the differences between predicted values and actual values. MSE penalizes larger errors more heavily than smaller ones, making it sensitive to outliers. Lower MSE values indicate better model performance, reflecting smaller deviations between predictions and actual values. The MSE of the NN is 0.256 and for the tree model, it is 0.0064. Again, the tree model stands superior to the NN.

R-squared is a statistical measure that represents the proportion of variance in the dependent variable that is explained by the independent variables in a regression model. R-squared measures the model's goodness of fit to the observed data. It ranges from 0 to 1, where 1 indicates a perfect fit. Higher R-squared values indicate better model performance, as they signify that the model explains a larger proportion of the variance in the target variable. The tree model is observed to have an R-squared value of 0.985 while the NN model has 0.403. The higher R-squared value indicates that the model has a good fit.

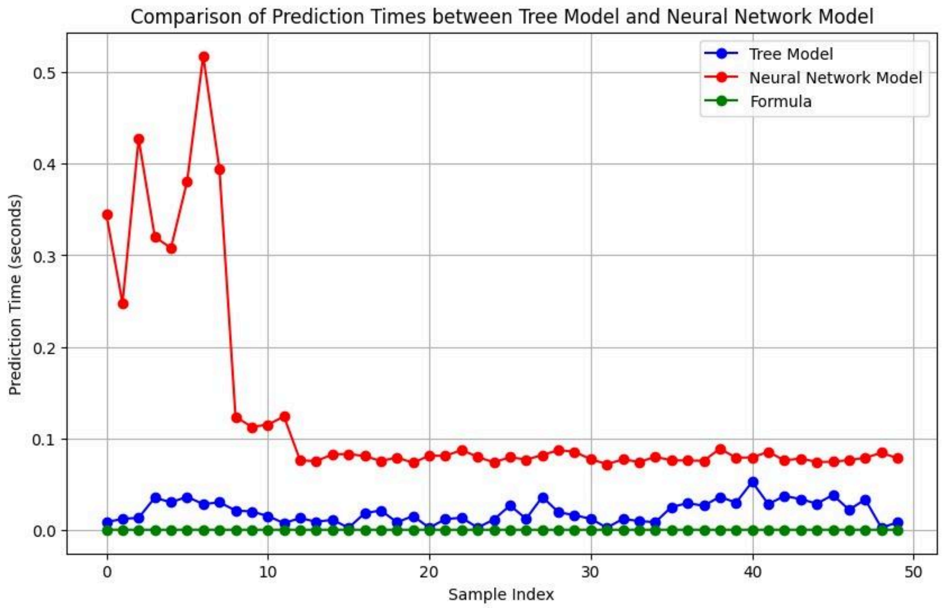

Fig.11. Prediction time comparison plots

Finally, the time taken by all the models for each prediction is plotted to find the fastest model. Turns out, the decision tree model is better even in this case. The average time taken by the tree model to predict the tilt angle is about 0.0194601 seconds while the NN model takes 0.1281326 seconds. And, the calculation of the tilt angle directly from the dynamics equations is 0.000000653 seconds, practically the values are given instantly. The reason for using an Ml algorithm was explained in the previous sections.

7 Conclusion

The evaluation of both the neural network and decision tree models reveals valuable insights into their performance in predicting the tilt angle of a vehicle. While both models were trained successfully, the decision tree model emerged as the preferred choice due to its superior prediction accuracy and robustness. The decision tree's predictions closely align with the actual values, as evidenced by the comparison plots, indicating better accuracy and reliability. Moreover, the error analysis highlights fewer errors in the decision tree model compared to the neural network, further affirming its superiority. Feature importance analysis elucidates the significant factors influencing tilt angle prediction, with the decision tree model effectively capturing the relationships between input features and the target variable. Performance metrics such as Mean Squared Error (MSE), Mean Absolute Error (MAE), and R-squared (R^2) score further validate the decision tree's superior performance, demonstrating lower error rates and higher explanatory power. In conclusion, the decision tree model exhibits superior predictive capabilities and represents a promising approach for accurately predicting vehicle tilt angles, with potential for further optimization and refinement.

8 Improvements and Future Scope

Implementing this predictive model includes rigorous testing and validation of results to further enhance the predictive modeling for tilt control in autonomous vehicles and ensure enhanced ride comfort. This is crucial to ensure the model's reliability and effectiveness in real-world scenarios. Additionally, improving feedback mechanisms is pivotal. This enables the vehicle to dynamically adapt its tilt angle in response to unexpected obstacles or changes in road conditions, ensuring passenger comfort and safety are maintained.

Expanding the use of simulators can play a significant role in this development process. By generating vast amounts of data covering a wide range of driving scenarios, simulators offer a cost-effective and efficient means to train and test models more comprehensively. This extensive simulation can help identify and address potential issues in the model's performance under diverse conditions, leading to more robust predictive capabilities.

Furthermore, investigating more advanced machine learning algorithms or considering hybrid models that combine multiple algorithms' strengths may significantly improve prediction accuracy and robustness. Exploring these advanced methodologies could uncover new ways to enhance the model's ability to predict vehicle tilt angles accurately, even in complex and unpredictable driving scenarios. This pursuit of improved methodologies underscores the commitment to leveraging cutting-edge technology to enhance autonomous vehicle engineering and passenger experience.

9 References

1. Y. Zheng, B. Shyrokau, T. Keviczky, M. A. Sakka and M. Dhaens, "Curve Tilting With Nonlinear Model Predictive Control for Enhancing Motion Comfort," in IEEE Transactions on Control Systems Technology, vol. 30, no. 4, pp. 1538-1549, July 2022, doi: 10.1109/TCST.2021.3113037.
2. CARLA Simulator - https://carla.org/
3. CARLA Documentation - https://carla.readthedocs.io/en/latest/

10 Appendix

10.1 Neural Network Model Code

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential, load_model
```

```
from tensorflow.keras.layers import Dense
from tensorflow.keras.metrics import MeanAbsoluteError
# Load data from the CSV file
df = pd.read_csv('dataset_carla.csv')
df.dropna(inplace=True)  # Drop rows with missing values
X = df.drop('Theta', axis=1)  # Features
y = df['Theta']  # Target variable
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Define neural network architecture
model = Sequential([
    Dense(256, activation='relu', input_shape=(X_train.shape[1],)),  # Input layer
    Dense(256, activation='relu'),
    Dense(128, activation='relu'),
    Dense(64, activation='relu'),
    Dense(64, activation='relu'),
    Dense(64, activation='relu'),# Hidden layers
    Dense(1, activation='linear')  # Output layer
])
# Compile the model
metrics=[MeanAbsoluteError()]
model.compile(optimizer='adam', loss='mean_absolute_error', metrics=metrics)
# Train the model
model.fit(X_train, y_train, batch_size=32, epochs=10, validation_data=(X_test, y_test))
# Save the trained model
model.save('tilt_model.h5')
# Function to predict theta value based on mass and radius of curvature
def predict_theta_NN(model,speed, cof,radius_of_curvature, mass):
    # Prepare input data for prediction
    input_data = np.array([[speed, cof,radius_of_curvature,mass]])
    # Make prediction
    prediction = model.predict(input_data)
    return prediction[0][0]  # Assuming single prediction
# Load the trained model
model1 = load_model('tilt_model.h5')
mass = 51.6  # kg
radius_of_curvature = 2  # m
cof=0.8
speed=0.348
```

```
predicted_theta = np.abs(predict_theta_NN(model, speed, cof, radius_of_curvature,mass))
print("Predicted Theta:", predicted_theta)
import numpy as np
f_seat = mass*9.8
f_friction = f_seat*cof
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
formula_theta = np.arctan(f_seat*(speed**2) - f_friction*9.8*radius_of_curvature/(f_seat+f_friction*speed**2))
error = round(np.abs(formula_theta - predicted_theta),2)
percent_error = round(error*100/formula_theta,2)
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
# Make predictions on the test set
predictions_NN = model.predict(X_test)
# Calculate Mean Squared Error (MSE)
mse_NN = mean_squared_error(y_test, predictions_NN)
print("Mean Squared Error (NN):", mse_NN)
# Calculate Mean Absolute Error (MAE)
mae_NN = mean_absolute_error(y_test, predictions_NN)
print("Mean Absolute Error (NN):", mae_NN)
# Calculate R-squared (R^2) Score
r2_NN = r2_score(y_test, predictions_NN)
print("R-squared (R^2) Score (NN):", r2_NN)
print(f'Predicted theta: {predicted_theta}')
print(f'Formula theta: {formula_theta}')
print(f'Absolute error: {error}')
print(f'Percent error = {percent_error}%')
```

10.2 Decision Tree Model Code

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.layers import Dense

# Load data from the CSV file
#df = pd.read_csv('auto_tiltdataset.csv')
df = pd.read_csv('dataset_carla.csv')
```

```
df.dropna(inplace=True)  # Drop rows with missing values
X = df.drop('Theta', axis=1)  # Features
y = df['Theta']  # Target variable
df.head()
import seaborn as sns
import matplotlib.pyplot as plt
temp = df.head(4000)
features = list(temp.columns)[:-1]
# Create subplots
fig, axes = plt.subplots(nrows=len(features), ncols=1, figsize=(8, 6 * len(features)))
# Plot each numerical column against the label
for i, column in enumerate(features):
    sns.scatterplot(x=temp[column], y=temp['Theta'], ax=axes[i])
    axes[i].set_xlabel(column)
    axes[i].set_ylabel('Theta')
plt.tight_layout()
plt.show()
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
import xgboost as xgb
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.model_selection import GridSearchCV
model = xgb.XGBRegressor(n_estimators=200, learning_rate=0.1, max_depth=4)
model.fit(X_train, y_train)
# Make predictions on the test set
predictions = model.predict(X_test)
# Calculate mean squared error
mse = mean_squared_error(y_test, predictions)
print("Mean Squared Error:", mse)
mae = mean_absolute_error(y_test, predictions)
print("Mean Absolute Error:", mae)
# Calculate R-squared (R^2)
r2 = r2_score(y_test, predictions)
print("R-squared (R^2):", r2)
import pickle
# Save the trained model to a file
with open('xgboost_model.pkl', 'wb') as file:
    pickle.dump(model, file)
# Load the model from the file
import pickle
```

```
#import pandas as pd
#from sklearn.model_selection import train_test_split
with open('xgboost_model.pkl', 'rb') as file:
    loaded_model = pickle.load(file)
# Make predictions using the loaded model
predictions_loaded_model = loaded_model.predict(X_test)
X_test
mass = 51.6  # kg
radius_of_curvature = 2  # m
cof=0.8
speed=0.348
import numpy as np
f_seat = mass*9.8
f_friction = f_seat*cof
formula_theta = np.arctan(f_seat*(speed**2) - f_friction*9.8*radius_of_curvature/(f_seat+f_friction*speed**2))
dummy_data = {
    'Speed_car': [speed],
    'cof_Range': [cof],
    'Radius_curvature': [radius_of_curvature],
    'Mass_person': [mass]
}
# Create a DataFrame from the dictionary
dummy_df = pd.DataFrame(dummy_data)
prediction = np.abs(loaded_model.predict(dummy_df))
prediction
formula_theta
from sklearn.metrics import mean_squared_error
# Extract the 'Theta' values from the ground truth DataFrame
y_true = formula_theta
# Extract the predicted 'Theta' values from the prediction output DataFrame
y_pred = prediction[0]
# Calculate mean absolute error
error = round(np.abs(y_true - y_pred),2)
percent_error = round(error*100/formula_theta,2)
print(f'Predicted theta: {y_pred}')
print(f'Formula theta: {formula_theta}')
print(f'Absolute error: {error}')
print(f'Percent error = {percent_error}%')
```

```
from ipywidgets import interact, widgets
def predict_and_calculate_error(mass_value,cof_value):
    # Update the mass value in the dummy DataFrame
    dummy_df['Mass_person'] = mass_value
    dummy_df['cof_Range'] = cof_value
    # Make predictions with your XGBoost model
    # Extract the 'Theta' values from the ground truth DataFrame
    f_seat = mass_value*9.8
    f_friction = f_seat*cof_value
formula_theta_var = np.arctan(f_seat*(speed**2) - f_friction*9.8*radius_of_curvature/(f_seat+f_friction*speed**2))
    y_true = formula_theta_var
    # Extract the predicted 'Theta' values from the prediction output DataFrame
    prediction_var = np.abs(loaded_model.predict(dummy_df))
    y_pred_var = prediction_var[0]
    # Calculate mean squared error
    #mse = mean_squared_error(y_true, y_pred)
    error = round(np.abs(y_true - y_pred_var),2)
    # Print the mean squared error
    print("Mass Value:", mass_value)
    print(f'Predicted theta: {y_pred_var}')
    print(f'Formula theta: {formula_theta_var}')
    print("Error:", error)
mass_slider = widgets.FloatSlider(value=70, min=50, max=120, step=0.01, description='Mass Value')
cof_slider = widgets.FloatSlider(value=0.9, min=0.6, max=1, step=0.01, description='Friction coeff.')
# Define an interactive widget
interactive_plot = interact(predict_and_calculate_error, mass_value=mass_slider, cof_value = cof_slider)
```

10.3 Code for data collection in CARLA

```
import glob
import os
import sys
import time
import cv2
import numpy as np
from PIL import Image
```

```
import csv
try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass
import carla
import argparse
import logging
import random
# Global variables
frame_rate = 30.0  # Frame rate of the video
image_size = (1920, 1080)  # Image size (width, height)
video_writer = None  # Video writer object
def save_image_and_video(image, video_filename):
    global video_writer
    global frame_rate
    global image_size
    if 'video_writer' not in globals():
        video_writer = None
    # Save the image
    image.save_to_disk('~/tutorial/output/%.6d.jpg' % image.frame)
    # Convert the image to OpenCV format (BGR)
    cv_image = cv2.cvtColor(np.array(image.raw_data), cv2.COLOR_RGBA2BGR)
    # Initialize video writer if not already initialized
    if video_writer is None:
            video_writer = cv2.VideoWriter(video_filename, cv2.VideoWriter_fourcc(*'XVID'), 
frame_rate, image_size)

    # Write the frame to the video
    video_writer.write(cv_image)
def main():
    argparser = argparse.ArgumentParser(
        description=__doc__)
    argparser.add_argument(
        '--host',
        metavar='H',
        default='127.0.0.1',
```

```
    help='IP of the host server (default: 127.0.0.1)')
argparser.add_argument(
    '-p', '--port',
    metavar='P',
    default=2000,
    type=int,
    help='TCP port to listen to (default: 2000)')
args = argparser.parse_args()
logging.basicConfig(format='%(levelname)s: %(message)s', level=logging.INFO)
client = carla.Client(args.host, args.port)
client.set_timeout(10.0)
# Video file name
video_filename = '~/tutorial/output/video_output.avi'
try:
    world = client.get_world()
    ego_vehicle = None
    ego_cam = None
    ego_col = None
    ego_lane = None
    ego_obs = None
    ego_gnss = None
    ego_imu = None
    # Start recording
    client.start_recorder('~/tutorial/recorder/recording01.log')
    # Spawn ego vehicle
    ego_bp = world.get_blueprint_library().find('vehicle.tesla.model3')
    ego_bp.set_attribute('role_name', 'ego')
    print('\nEgo role_name is set')
    ego_color = random.choice(ego_bp.get_attribute('color').recommended_values)
    ego_bp.set_attribute('color', ego_color)
    print('\nEgo color is set')
    spawn_points = world.get_map().get_spawn_points()
    number_of_spawn_points = len(spawn_points)
    if number_of_spawn_points > 0:
        random.shuffle(spawn_points)
        ego_transform = spawn_points[0]
        ego_vehicle = world.spawn_actor(ego_bp, ego_transform)
        print('\nEgo is spawned')
    else:
        logging.warning('Could not find any spawn points')
```

```
# Add a RGB camera sensor to ego vehicle.
cam_bp = world.get_blueprint_library().find('sensor.camera.rgb')
cam_bp.set_attribute("image_size_x", str(1920))
cam_bp.set_attribute("image_size_y", str(1080))
cam_bp.set_attribute("fov", str(105))
cam_location = carla.Location(x=2.5, y=0, z=1.7)
cam_rotation = carla.Rotation(pitch=8, yaw=180, roll=0)
cam_transform = carla.Transform(cam_location, cam_rotation)
ego_cam = world.spawn_actor(cam_bp, cam_transform, attach_to=ego_vehicle,
attachment_type=carla.AttachmentType.Rigid)
ego_cam.listen(lambda image: save_image_and_video(image, video_filename))
# Add collision sensor to ego vehicle.
col_bp = world.get_blueprint_library().find('sensor.other.collision')
col_location = carla.Location(0, 0, 0)
col_rotation = carla.Rotation(0, 0, 0)
col_transform = carla.Transform(col_location, col_rotation)
ego_col = world.spawn_actor(col_bp, col_transform, attach_to=ego_vehicle,
attachment_type=carla.AttachmentType.Rigid)
def col_callback(colli):
  print("Collision detected:\n" + str(colli) + '\n')
  with open('collision_data.csv', mode='a') as file:
    writer = csv.writer(file)
    if file.tell() == 0:
      writer.writerow(['Frame', 'Actor Id', 'Other Actor Id'])
    writer.writerow([colli.frame, colli.actor.id if colli.actor else "None", colli.other_actor.id
if colli.other_actor else "None"])
ego_col.listen(lambda colli: col_callback(colli))
# Add GNSS sensor to ego vehicle.
gnss_bp = world.get_blueprint_library().find('sensor.other.gnss')
gnss_location = carla.Location(0, 0, 0)
gnss_rotation = carla.Rotation(0, 0, 0)
gnss_transform = carla.Transform(gnss_location, gnss_rotation)
gnss_bp.set_attribute("sensor_tick", str(3.0))
    ego_gnss = world.spawn_actor(gnss_bp, gnss_transform, attach_to=ego_vehicle,
attachment_type=carla.AttachmentType.Rigid)
def gnss_callback(gnss):
  print("GNSS measure:\n" + str(gnss) + '\n')
  with open('gnss_data.csv', mode='a') as file:
    writer = csv.writer(file)
    if file.tell() == 0:
```

```
                writer.writerow(['Frame', 'Latitude', 'Longitude', 'Altitude'])
            writer.writerow([gnss.frame, gnss.latitude, gnss.longitude, gnss.altitude])
        ego_gnss.listen(lambda gnss: gnss_callback(gnss))
        # Add IMU sensor to ego vehicle.
        imu_bp = world.get_blueprint_library().find('sensor.other.imu')
        imu_location = carla.Location(0, 0, 0)
        imu_rotation = carla.Rotation(0, 0, 0)
        imu_transform = carla.Transform(imu_location, imu_rotation)
        imu_bp.set_attribute("sensor_tick", str(3.0))
        ego_imu = world.spawn_actor(imu_bp, imu_transform, attach_to=ego_vehicle,
attachment_type=carla.AttachmentType.Rigid)
        def imu_callback(imu):
            print("IMU measure:\n" + str(imu) + '\n')
            with open('imu_data.csv', mode='a') as file:
                writer = csv.writer(file)
                if file.tell() == 0:
                    writer.writerow(['Frame', 'Acceleration_x', 'Acceleration_y', 'Acceleration_z',
'Angular Velocity_x', 'Angular Velocity_y', 'Angular Velocity_z', 'Compass'])
                writer.writerow([imu.frame, imu.accelerometer.x, imu.accelerometer.y,
imu.accelerometer.z, imu.gyroscope.x, imu.gyroscope.y, imu.gyroscope.z, imu.compass])
        ego_imu.listen(lambda imu: imu_callback(imu))
        # Place spectator on ego spawning
        spectator = world.get_spectator()
        world_snapshot = world.wait_for_tick()
        spectator.set_transform(ego_vehicle.get_transform())
        # Enable autopilot for ego vehicle
        ego_vehicle.set_autopilot(True)
        # Timer for stopping after 90 seconds
        start_time = time.time()
        while time.time() - start_time < 90:
            world_snapshot = world.wait_for_tick()
    finally:
        # Stop recording and destroy actors
        client.stop_recorder()
        if ego_vehicle is not None:
            if ego_cam is not None:
                ego_cam.stop()
                ego_cam.destroy()
            if ego_col is not None:
                ego_col.stop()
```

```
            ego_col.destroy()
        if ego_lane is not None:
            ego_lane.stop()
            ego_lane.destroy()
        if ego_obs is not None:
            ego_obs.stop()
            ego_obs.destroy()
        if ego_gnss is not None:
            ego_gnss.stop()
            ego_gnss.destroy()
        if ego_imu is not None:
            ego_imu.stop()
            ego_imu.destroy()
        ego_vehicle.destroy()
    # Release the video writer
    if video_writer is not None:
        video_writer.release()
    print('\nDone with tutorial_ego.')
if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        pass
    finally:
        print('\nDone with tutorial_ego.')
```