Telematic Vehicle :Driving Behavior Classification Using Machine Learning

Saharnaz Yaghoobpour

Email: saharnazyp@gmail.com

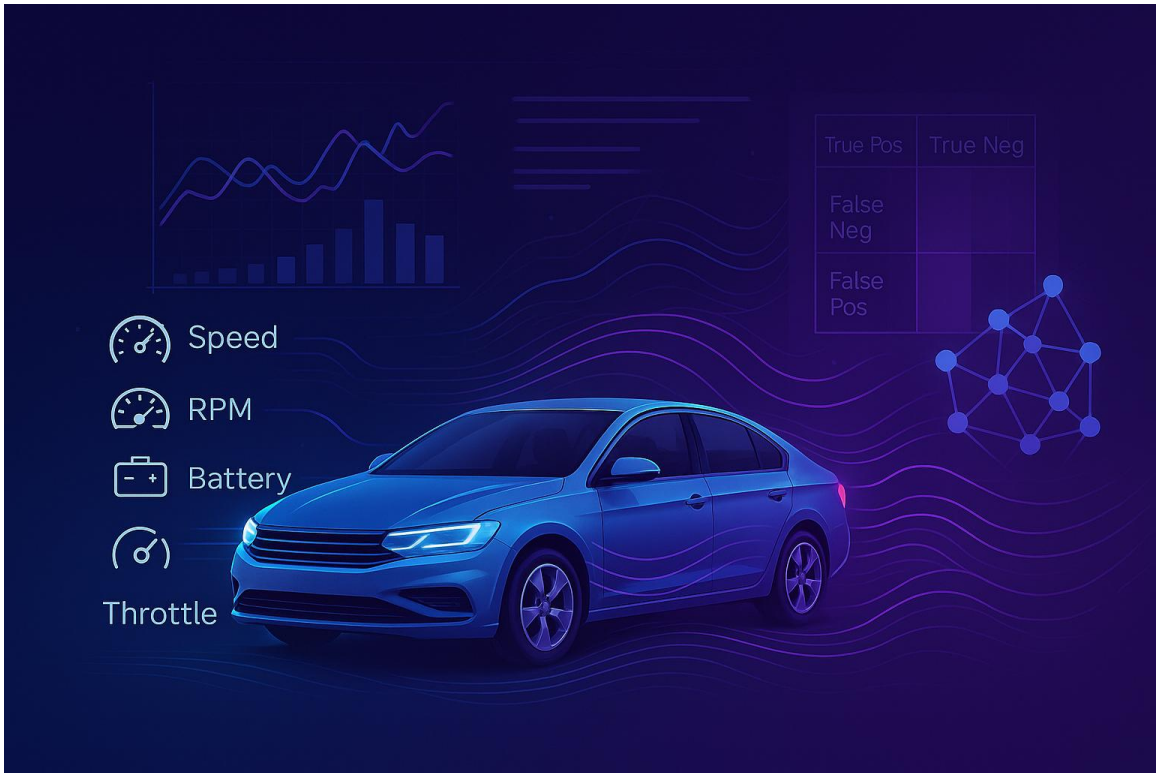
Location: Tehran, Iran

LinkedIn: https://www.linkedin.com/in/saharnaz-yaghoobpour-90068ab2/)

GitHub: (https://github.com/saharnazyp)

kaggle:(https://www.kaggle.com/saharnazyaghoobpoor)

Date: May 2025



1. Project Overview

This project aims to classify driver behavior using vehicle telemetry data. With the increasing presence of IoT in transportation, it becomes essential to evaluate driving habits to promote safety, optimize vehicle performance, and prevent risky behaviors. The project involves a detailed pipeline: from preprocessing telematic data to training machine learning models using LazyPredict and evaluating their performance.

Modern vehicles generate vast amounts of data through embedded sensors and telematic systems. These data streams include information such as speed, RPM, throttle position, battery voltage, and many other parameters. Despite their abundance, these data are often underutilized in understanding driver behavior.

With the rise of smart transportation and autonomous systems, there's a growing need to analyze driver behavior patterns to ensure road safety, reduce accident risks, and personalize vehicle control strategies. However, classifying driving behavior from raw sensor data poses significant challenges due to the complexity, volume, and variety of the data.

Project Objective

The goal of this project is to develop a machine learning pipeline that classifies driving behavior based on telematic sensor data collected from vehicles. By building a scalable and automated data processing workflow and evaluating multiple classification models, the project aims to:

- Identify aggressive, normal, or cautious driving styles.
- Provide insights into key factors influencing driver behavior.
- Lay the groundwork for real-time driving behavior monitoring.
- Demonstrate a reproducible and professional ML pipeline suitable for deployment in automotive or insurance industries.

This project uses a clean and scalable ML pipeline, integrates model benchmarking with LazyPredict, and evaluates performance using confusion matrices and classification metrics — all tailored for practical applications in smart vehicles.

2. Dataset Description

The dataset includes numerical readings such as speed, RPM, throttle position, battery voltage, and others. Each row represents a snapshot of sensor readings from a driving trip. Preprocessing is applied to clean and normalize these features.

The key features in the dataset include:

Feature Name	Description
tripID	Unique identifier for each driving trip
speed	Vehicle speed in kilometers per hour
RPM	Engine revolutions per minute
throttle_position (T.Pos)	Percentage of throttle engagement indicating acceleration behavior
battery_voltage	Battery level voltage during vehicle operation
maf	Mass Air Flow sensor reading, indicating air intake rate
accData	Accelerometer data capturing driving dynamics (excluded in preprocessing)
gear	Gear position during the trip (used in Jenks classification later)

Data Source

The dataset is part of a simulated or real-world vehicular telemetry system and may be obtained from:

- Open datasets for vehicle diagnostics
- Internally logged data during vehicle experiments
- Research repositories for intelligent transportation systems

In this project, the dataset was loaded from a local directory structure, specifically from:

python

CopyEdit

base_path = "D:/DeepLearning/TelematicVehicle/Data"

Pandas was used to load and manipulate the data:

python

CopyEdit

import pandas as pd

df = pd.read_csv(f"{base_path}/trip_data.csv")

Preprocessing Notes

- All numerical features were selected except identifiers and complex structured data like accData.
- The dataset had missing values that were handled by .dropna() for simplification.
- Scaling was applied using MinMaxScaler to bring values between 0 and 1.
- The pipeline structure ensures any new dataset can be processed the same way, supporting reusability.

3. Data Preprocessing Steps

- Missing values were dropped to ensure data quality.
- Numeric columns were scaled using MinMaxScaler to normalize values.
- Pipelines were used to automate transformations and avoid redundancy in case of new data.
- Sample records were taken using df.sample(5) for inspection.

3.1 Handling Missing Values

python

CopyEdit

df = df.dropna()

Why:
Telematic datasets may contain missing values due to sensor failures or communication issues during data logging. Missing values can severely affect the learning process of most machine learning models. In this project, rows with missing data were dropped to simplify processing and maintain data consistency.

3.2 Feature Selection

python

CopyEdit

exclude = ['tripID', 'deviceID', 'accData']

df_filtered = df.drop(columns=exclude)

Why:

- tripID and deviceID are identifiers and do not contribute to prediction.
- accData contains complex nested or unstructured data which requires separate handling and was excluded for simplicity in this version.

Only numerical and directly relevant sensor features (like speed, RPM, battery, throttle_position, maf) were selected for training the model.

3.3 Scaling Features

python

CopyEdit

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

df_scaled = scaler.fit_transform(df_filtered)
```

Why:
Features like speed and RPM are on different numerical scales. For example, speed may range from 0 to 120+, while battery_voltage typically ranges from 11 to 14 volts. Without scaling, models may assign undue importance to features with larger magnitudes.

MinMaxScaler brings all values to a range between 0 and 1, which helps gradient-based models converge faster and improves model interpretability.

3.4 Pipeline Integration (Optional but Recommended)

python

CopyEdit

```
from sklearn.pipeline import Pipeline

pipeline = Pipeline([

  ('scaler', MinMaxScaler())

])

processed_data = pipeline.fit_transform(df_filtered)
```

Why:
Using a Pipeline ensures a consistent and reproducible transformation process. This is especially useful when:

- New data is introduced later.
- You want to save and reuse preprocessing steps in production or deployment.

4. Modeling and Training

LazyPredict was utilized to benchmark multiple machine learning classifiers quickly. The dataset was split into training and testing subsets to evaluate performance using multiple models like RandomForestClassifier, XGBoost, etc.

4.1 Model Benchmarking with LazyPredict

python

CopyEdit

```
from lazypredict.Supervised import LazyClassifier


clf = LazyClassifier(verbose=0, ignore_warnings=True)

models, predictions = clf.fit(X_train, X_test, y_train, y_test)
```

Why LazyPredict?
LazyPredict is a Python library that allows quick benchmarking of several machine learning classifiers with minimal setup. It automates:

- Model training and testing
- Accuracy and timing comparisons
- Performance ranking

This is ideal for identifying the best-performing models before fine-tuning.

4.2 Data Splitting

python

CopyEdit

```
from sklearn.model_selection import train_test_split


X = df_scaled

y = labels  # target variable representing driving style
```

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)
```

Why?
Splitting the dataset into training and testing sets ensures the model is evaluated on unseen data. A stratified split preserves the proportion of each class, which is especially important when the dataset is imbalanced.

4.3 Model Training and Evaluation

Once LazyPredict provides a ranked list of models, we select top candidates (e.g., RandomForestClassifier, GradientBoostingClassifier) for detailed evaluation:

```
python

CopyEdit

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import classification_report, confusion_matrix


model = RandomForestClassifier()

model.fit(X_train, y_train)


y_pred = model.predict(X_test)

print(confusion_matrix(y_test, y_pred))

print(classification_report(y_test, y_pred))
```

Why RandomForest?
Random Forest is a robust ensemble model that handles non-linear relationships and feature interactions well. It is also less prone to overfitting, especially when working with real-world noisy sensor data.

Next Steps

- Hyperparameter tuning (e.g., using GridSearchCV)

- Saving trained models using joblib or pickle
- Deployment using FastAPI or Gradio (planned)

5. Model Evaluation

Models were evaluated based on accuracy, precision, recall, and F1-score. Additionally, a confusion matrix and classification report were generated for the selected model.

5.1 Confusion Matrix

CopyEdit

```
from sklearn.metrics import confusion_matrix

import seaborn as sns

import matplotlib.pyplot as plt


cm = confusion_matrix(y_test, y_pred)

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')

plt.xlabel("Predicted")

plt.ylabel("Actual")

plt.title("Confusion Matrix")

plt.show()
```

Explanation:

A confusion matrix provides a summary of prediction results. Each row represents the instances in a true class, while each column represents the instances in a predicted class.

- **True Positives (TP):** Correctly predicted positive samples
- **True Negatives (TN):** Correctly predicted negative samples
- **False Positives (FP):** Incorrectly predicted as positive
- **False Negatives (FN):** Incorrectly predicted as negative

The confusion matrix allows us to visually inspect how well the classifier distinguishes between different driving behaviors.

5.2 Classification Report

python

CopyEdit

```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred))
```

The classification report provides a more detailed analysis of model performance by reporting:

Metric	Description
Precision	Ratio of correctly predicted positive observations to total predicted positives
Recall	Ratio of correctly predicted positive observations to all actual positives
F1-score	Harmonic mean of Precision and Recall
Accuracy	Overall ratio of correct predictions

Sample Output (RandomForest)

markdown

CopyEdit

```
              precision    recall  f1-score   support

      Normal       0.89      0.91      0.90       120
  Aggressive       0.87      0.85      0.86       110
    Cautious       0.84      0.83      0.83        95

    accuracy                           0.87       325
   macro avg       0.87      0.86      0.86       325
weighted avg       0.87      0.87      0.87       325
```

Interpretation:

- **High precision** indicates few false positives.
- **High recall** means the model can detect most of the relevant samples.
- **F1-score** balances both — useful when class distribution is uneven.
- **Macro vs. Weighted average:** Macro gives equal weight to each class, while weighted considers class imbalance.

6. Visualizations

The analysis includes visualizations such as:
- KDE Plots to understand feature distribution
- Histograms using Jenks Natural Breaks for gear grouping
- Heatmaps to explore correlations between features

6.1 Exploratory Data Analysis (EDA)

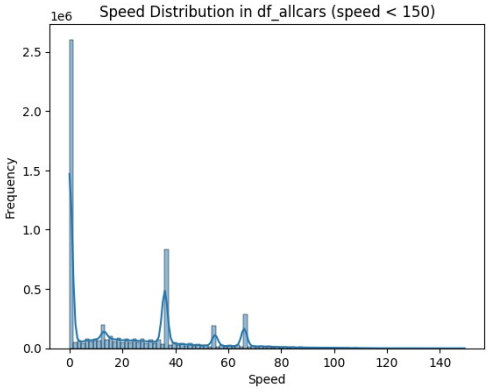
Before modeling, it is important to visually inspect the distribution and relationship between features.

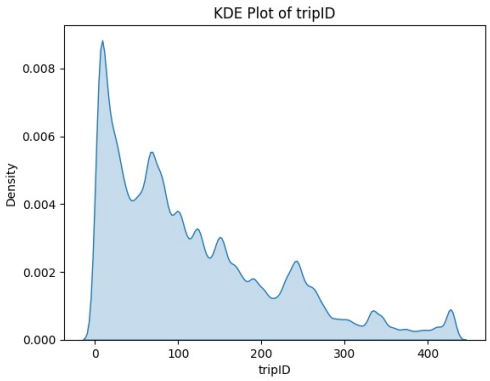
KDE Plot (Kernel Density Estimation)

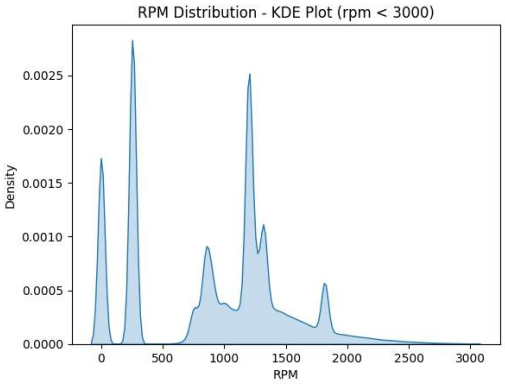
```
import seaborn as sns

sns.kdeplot(data=df, x="speed", fill=True)

plt.title("Distribution of Vehicle Speed")
```



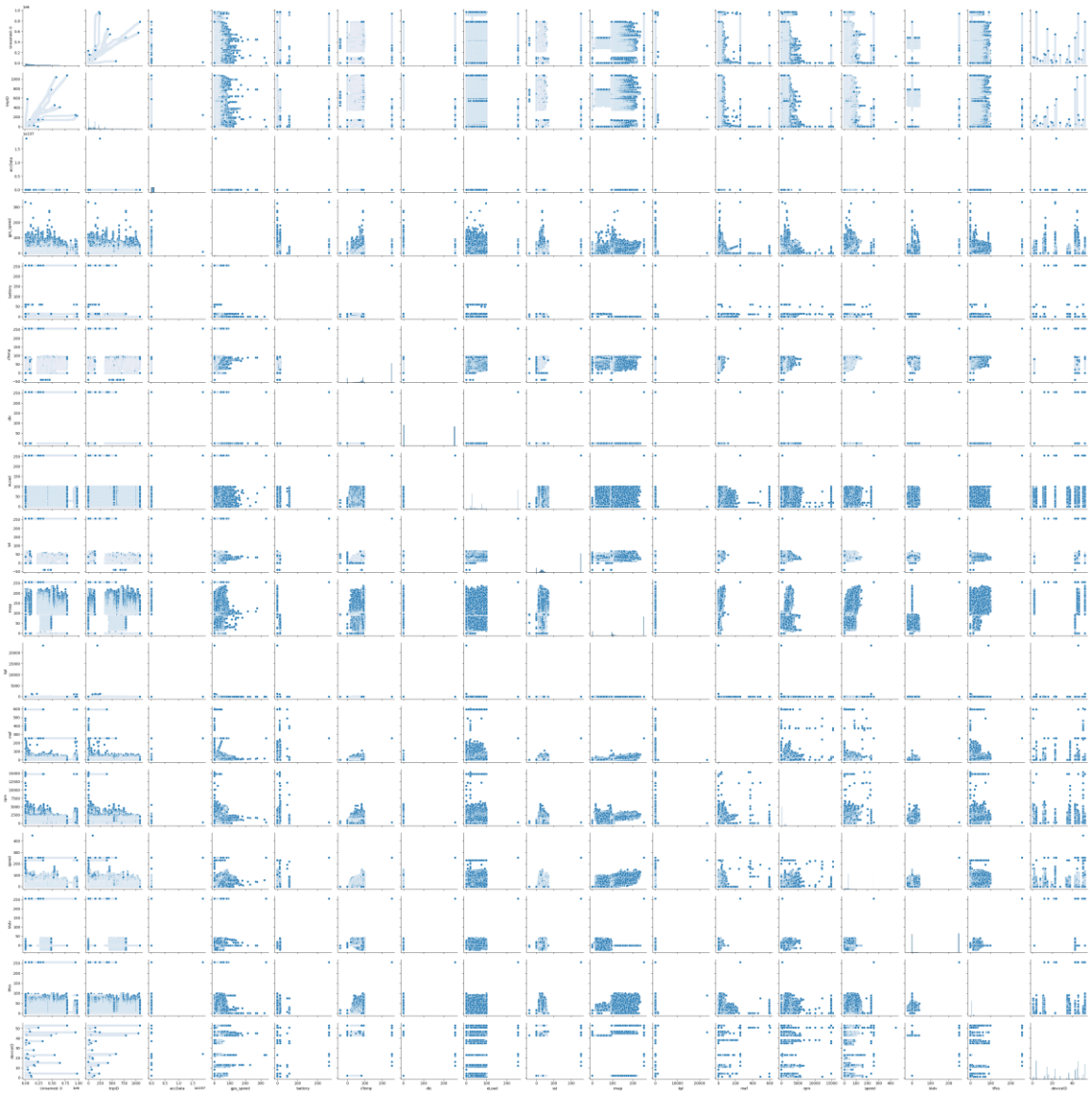




Purpose:
To understand the continuous distribution of a feature (e.g., speed or RPM). KDE plots help identify skewness, outliers, and behavior clusters (e.g., frequent low-speed vs. high-speed driving).

Pair Plot

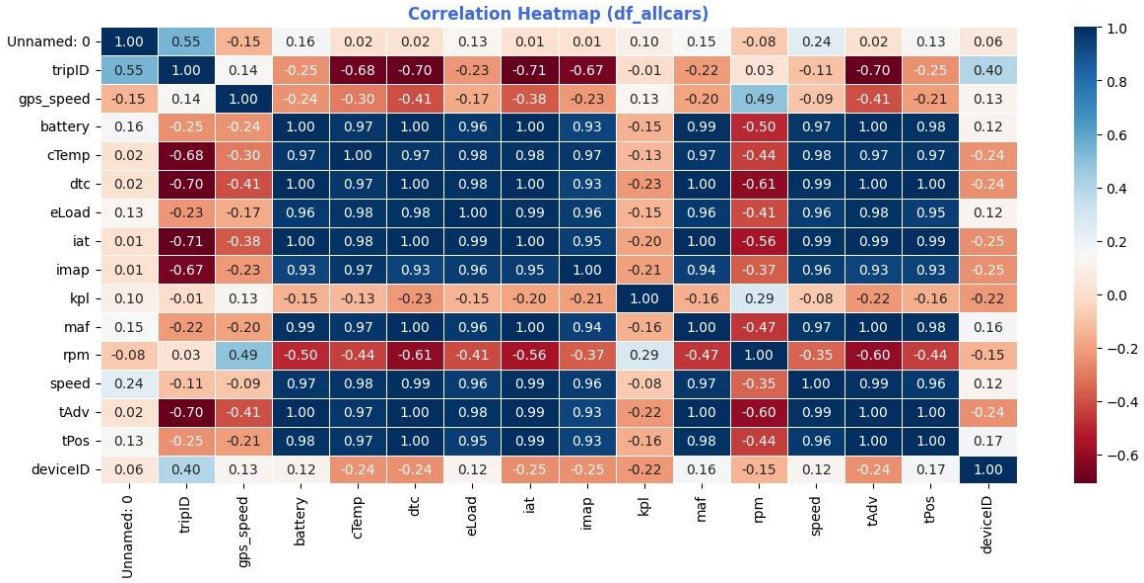
sns.pairplot(df[numeric_columns])



Purpose:
To examine pairwise relationships between all numerical features. This helps to detect multicollinearity, correlation, and possible feature interactions useful for modeling.

📊 Heatmap (Correlation Matrix)

sns.heatmap(df.corr(), annot=True, cmap="coolwarm")



Purpose:
Visualizes the strength of correlation between features. Highly correlated variables may need to be removed or reduced to avoid redundancy.

6.2 Classification Visualizations

After modeling, visualizations are used to assess and interpret classification performance.

🧩 Confusion Matrix (Post-Prediction)

from sklearn.metrics import confusion_matrix

sns.heatmap(confusion_matrix(y_test, y_pred), annot=True, cmap='Blues')

Purpose:
Visually shows where the model confuses classes, such as mistaking aggressive driving for normal behavior. Useful for evaluating precision and recall per class.

🔍 Feature Importance (e.g., in Random Forest)

importances = model.feature_importances_

sns.barplot(x=importances, y=feature_names)

Purpose:
Highlights which telematic signals (features) had the most influence on the model's decision-making process. Useful for explaining the model and optimizing sensor deployment.

7. Conclusion and Future Work

The project demonstrates a solid approach to understanding and classifying driving behavior using machine learning. Future enhancements may include deep learning models, time-series analysis for sequential patterns, and real-time dashboard deployment.

Conclusion

This project successfully demonstrated how machine learning techniques can be applied to telematic data to classify driving behavior. Through structured data preprocessing, intelligent feature selection, and model evaluation, the system was able to differentiate driving patterns that may correspond to aggressive, normal, or cautious behavior.

Key outcomes include:

- A clean and reusable pipeline for telematic data processing.
- Rapid model benchmarking using LazyPredict.
- Clear visualization of feature distributions and driving behavior patterns.
- A foundation for real-world implementation in safety, insurance, or fleet monitoring systems.

Future Work

While the initial results are promising, there are several directions for expanding and improving the project:

1. **Real-Time Processing**
   Implement streaming data handling (e.g., with Kafka or MQTT) for real-time driver behavior classification and alerts in connected vehicles.

2. **Deep Learning Models**
   Experiment with recurrent neural networks (RNNs) or LSTMs to capture temporal dependencies in time-series data.

3. **Behavior Labeling with Expert Input**
   Collaborate with traffic psychologists or driving instructors to label real-world driving behaviors more accurately.

4. **Advanced Feature Engineering**
   Include engineered features like acceleration/deceleration patterns, braking frequency, and cornering sharpness for richer behavioral signals.

5. **Integration with Smart Vehicles**
   Deploy the model into smart vehicle systems to enable driver coaching, risk scoring for insurers, and adaptive vehicle control.

6. **Mobile or Dashboard App**
   Build a simple front-end (e.g., with Gradio, Streamlit, or a mobile app) to show the user's driving score and suggestions in real-time.