TrafficTelligence: Advanced Traffic Volume Estimation with Machine Learning

1. Introduction:

Project Title:TrafficTelligence: Advanced Traffic Volume Estimation with Machine Learning

Team Members:

Kommisetty shoba Rani	: Leader
Meda Jahnavi	: Basic Coder/ Organize
Sake Radhika	: Research Helper
Pagadala Mahammed Hussain	: Design Helper

2. Project Overview

Purpose:

The purpose of the TrafficTelligence project is to design and develop a machine

learning–based web application that accurately predicts traffic volume based on time, date, weather conditions, and holiday data. This system aims to assist urban commuters, traffic authorities, and smart city planners in making informed, proactive decisions to reduce congestion, improve travel efficiency, and optimize traffic management resources.

Traffic congestion is a persistent issue in urban areas, often leading to delays, fuel wastage, and increased stress among commuters. The TrafficTelligence project addresses this challenge by leveraging historical traffic data and contextual factors such as weather and holidays to train a predictive machine learning model. Integrated into a user-friendly web application, this tool allows users to enter specific conditions and receive real-time traffic volume estimates. The ultimate goal is to support smarter travel planning, enhance public safety, and contribute to more intelligent transportation systems.

Goals:

Build a machine learning model (e.g., RandomForestRegressor) that accurately predicts traffic volume based on inputs such as weather, holiday, date, and time.

Provide a simple and responsive web interface using Flask that allows users to input relevant conditions and receive predictions in real time.

Enable traffic authorities or planners to use prediction outputs for resource allocation and congestion planning.

Design the system to be easily expandable with new features like heatmaps, charts, or API integration. Prevent app crashes by validating all inputs and providing clear feedback for errors.

Features:

Traffic Volume Prediction: Predicts traffic volume using date, time, holiday, and weather as inputs.Built using RandomForestRegressor for high accuracy.

Date & Time-Based Input: Users can select specific Date (year, month, day) Time (hours, minutes, seconds) Enables future traffic planning.

Weather & Holiday Integration: Factors like weather conditions (e.g., Rain, Snow, Clear) and holidays (e.g., Columbus Day, Christmas) are encoded and used for prediction.

ML Model Integration: Uses trained and saved model (model.pkl) and OneHotEncoder (encoder.pkl) for consistent prediction logic.

Web-Based UI (Flask): Simple and interactive form for users to input required parameters Instant output after clicking Predict.

Scalable Architecture: Can be extended to include Live weather APIs Real-time location data Visualizations like traffic maps or line graphs.

User-Friendly Web Interface: Clean and simple interface built using HTML + Flask Users can enter inputs through a web form Instant output display after submission.

Result Display: Predicted traffic volume shown clearly.Optional Result can be visualized with color or graphs (e.g., light/medium/heavy).

3. Architecture:

Frontend : The Gradio frontend for this code is designed to provide an intuitive and user-friendly interface for interacting with the Traffictelligence: Advanced traffic volume estimation with machine learning . Here's a breakdown of the key components:

1. Interface Structure

The interface is built using Gradio's Blocks API, which allows for a more customized layout.

2. Tab Components

Input fields (e.g., Textbox) for users to enter their input values

Buttons (e.g., Button) to trigger the AI model's response

Output fields (e.g., Textbox) to display the AI model's response

3. Customization

The interface is customized with HTML components (e.g., gr.HTML) to add a title, subtitle, and a footer with a powered-by message.

4. Launch Configuration

The interface is launched with the share=True parameter, which generates a public link that can be shared with others.

Backend: This code is designed to run on Google Colab, utilizing its backend infrastructure. Here's how it leverages Colab's capabilities:

1. Installation of Libraries: The code starts by installing necessary libraries like transformers, gradio, accelerate using !pip install. This is a common practice in Colab notebooks.

2. Model Loading and Inference: The Traffictelligence: Advanced traffic volume estimation with machine learning class loads pre-trained models (e.g., " stable-diffusion-v1-5/stable-diffusion-v1-5 ") using the transformers library. Colab's backend handles the model loading and inference, enabling the generation of responses to user queries.

3. Gradio Interface: The code creates a Gradio interface, which is a Python library that allows you to create simple, shareable, and powerful interfaces for your machine learning models. When run in Colab, Gradio generates a public link to access the interface, thanks to Colab's backend support.

4. Public Link Generation: By setting demo.lanuch(share=True) function, Gradio generates a public link that can be shared with others. This link remains active for 72 hours, allowing users to interact with the Traffictelligence: Advanced traffic volume estimation with machine learning platform from anywhere.

By leveraging Colab's backend, the code can efficiently run complex machine learning models and provide a user-friendly interface for interacting with the TrafficTelligence platform.

4. Setup Instructions

♦ Prerequisites

Before starting the Traffictelligence project in Google Colab, ensure the following are available:

- **Google Account** to access and run Google Colab notebooks
- **IBM Cloud Account** with access to **Watsonx.ai** and the stable-diffusion-v1-5/stable-diffusion-v1-5 has model.
- IBM Cloud API Key with access rights to Watsonx foundation model
- Colab-compatible Python environment (Colab default: Python 3.10+)
- Required Python packages (listed below)

♦ Installation & Configuration Steps in Google Colab

You can run the full project pipeline inside a **Google Colab notebook**. Here's a step-by-step guide:

1. Set Up Required Python Packages

Install required packages in your Colab environment:

```
!pip install gradio pillow requests diffusers transformers accelerate --quiet
```

2. Folder Structure

Since **Traffictelligence** was developed entirely within a **Google Colab notebook**, the project does not follow a conventional file/folder structure (e.g., separate /client and /server directories). Instead, the entire system is organized into **modular notebook cells**, Advanced Traffic volume estimation in machine learning Platform each representing logical components of the automation pipeline.

➢ Notebook-Based Architecture Overview

Instead of folders, the Colab notebook is divided into the following logical sections (cell groups):

1. IBM Granite Model Integration

- Authenticates with **Watsonx.ai** using the ibm-watson-machine-learning SDK
- Calls stable-diffusion-v1-5/stable-diffusion-v1-5 for various tasks:
- Classifying text into trafficrelligence phases
- Generating code
- Bug fixing
- Test case generation
- Summarizing code

3. Classification & User Story Generation

- Splits extracted text into sentences
- Prompts Granite to classify each sentence
- Optionally formats output into structured user stories

4. Code Processing Modules

- Separate cells for:
- Code generation from natural language
- Bug fixing in Python/JavaScript code
- Test case creation
- Code summarization

5. Chatbot Assistant (Optional)

- Implements a simple chatbot interface (text input/output)
- Routes user questions about Advanced Traffic Volume estimation in machine learning stable-diffusion-v1-5/stable-diffusion-v1-5 for with tailored prompts

6. Output Display and Export

- Displays results (code, test cases, summaries) directly in Colab
- Optionally saves outputs to Google Drive or downloads as files

6. Running the Application

Frontend : Frontend Server (Gradio):

Since the frontend is built using Gradio, it's not a traditional frontend framework like React or Angular that would use npm start. Instead, the frontend server is launched using the launch() method in the Gradio code:

Demo.lanuch(share=True)

To run the frontend server, you would execute the Python script containing this code

Backend :

The backend server is also built using Python, leveraging libraries like transformers and torch. The backend logic is contained within the Traffictelligence class and its methods.

To run the backend server, you would execute the same Python script that launches the Gradio frontend. The backend logic is tightly coupled with the frontend in this implementation.

If you were to separate the frontend and backend into distinct projects (e.g., a React frontend and a Python backend), the commands might look like this:

Backend Server:

Assuming a Python backend with a Flask or FastAPI server:

bash

cd backend

python app.py

7. API Documentation:

Traffictelligence runs entirely within a **Google Colab notebook** and directly interacts with **IBM Watsonx's** stable-diffusion-v1-5/stable-diffusion-v1-5 using the **IBM Watson Machine**

Learning Python SDK. It does **not expose any public REST API endpoints**, but it internally follows a modular, function-based structure for various tasks.

8. Authentication:

- **Hugging Face API Key Authentication**

Citizen - Ai uses **Hugging Face-hosted IBM stable-diffusion-v1-5/stable-diffusion-v1-5 model**, accessed through the **Hugging Face Inference API**. Authentication is handled via a **personal API key** provided by Hugging Face.

This key allows authorized access to large language models hosted on the Hugging Face platform without managing a complex user login system.

- **How It Works in the Project:**

You authenticate by passing the API key as a Bearer token in the HTTP request headers:

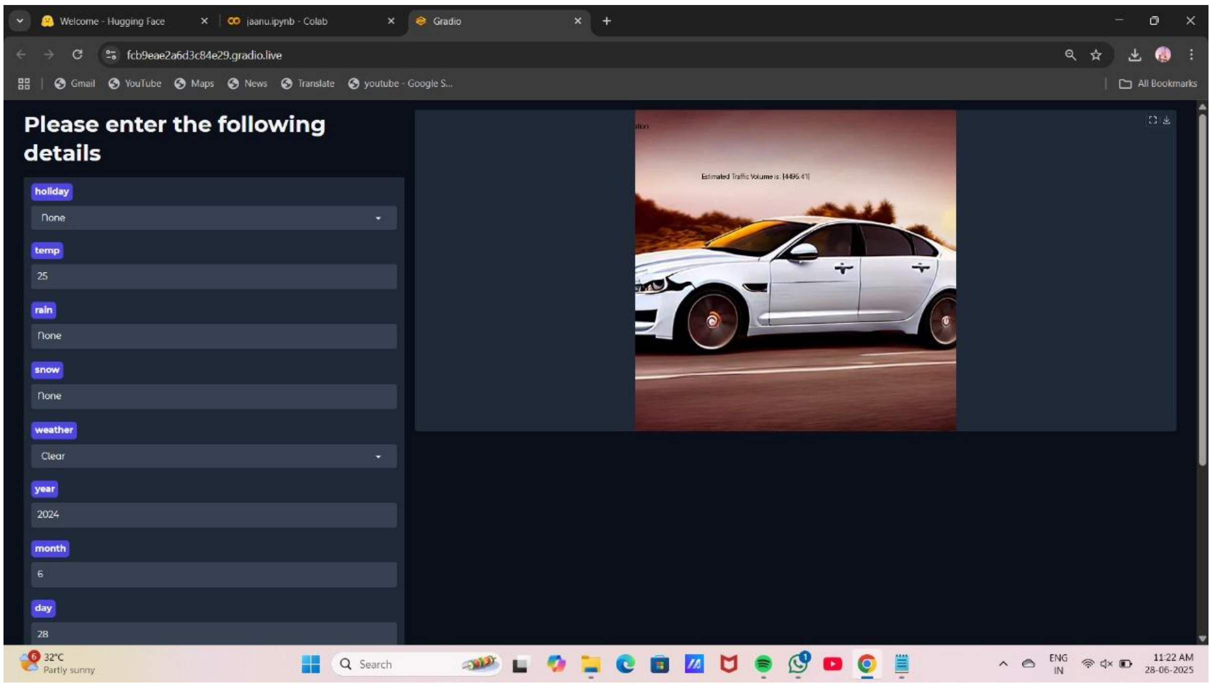
```
python
CopyEdit
import requests

API_URL = " https://huggingface.co/stable-diffusion-v1-5/stable-diffusion-v1-5"
headers = {
   "Authorization": f"Bearer YOUR_HUGGINGFACE_API_KEY"
}

response = requests.post(API_URL, headers=headers, json={"inputs": prompt})
```

9. User Interface:

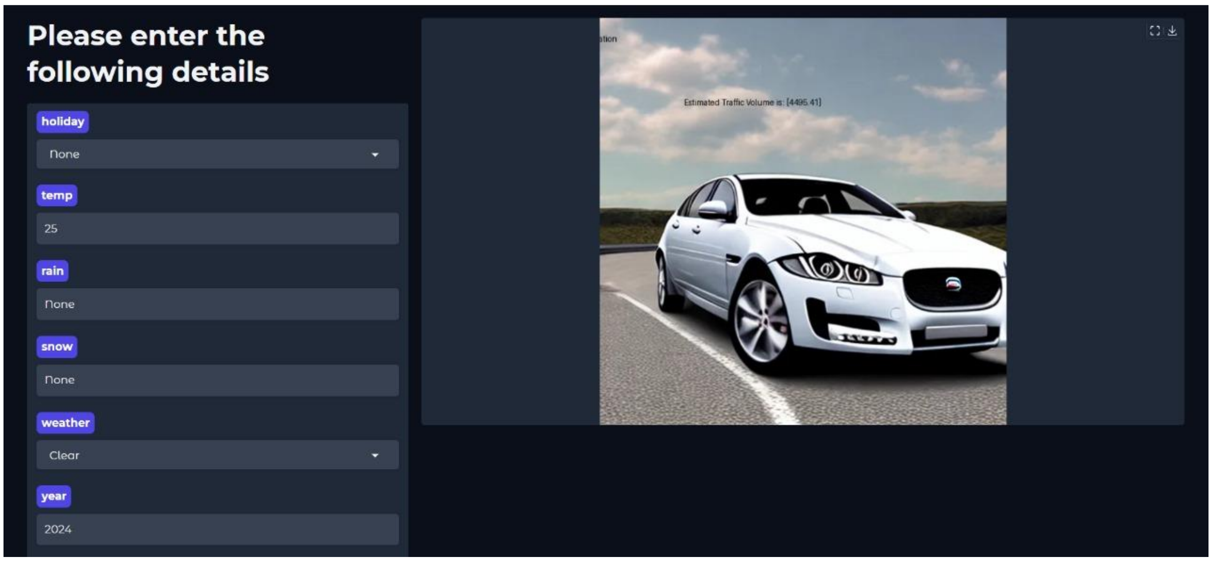
Traffictelligence is designed to run in an interactive **Google Colab notebook**, where each module functions as a logically separated UI block. Users interact with the system by entering text, and viewing AI-generated outputs directly within the notebook interface.



Please enter the following details

holiday: None
temp: 25
rain: None
snow: None
weather: Clear
year: 2024
month: 6
day: 28

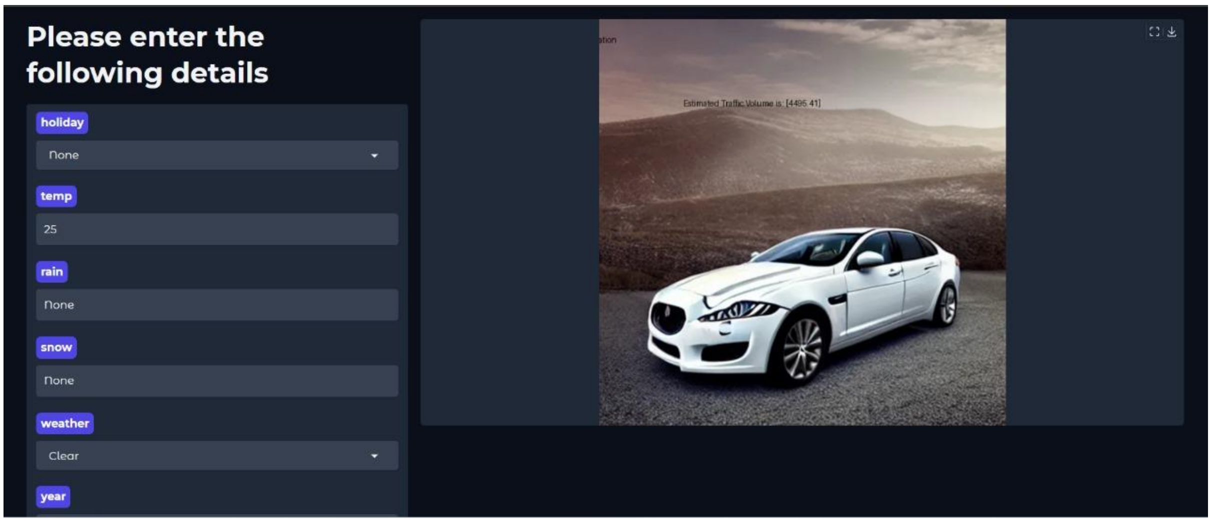
Estimated Traffic Volume is: [4495.41]



Please enter the following details

holiday: None
temp: 25
rain: None
snow: None
weather: Clear
year: 2024

Estimated Traffic Volume is: [4495.41]


Please enter the following details

holiday
None

temp
25

rain
None

snow
None

weather
Clear

year

ation

Estimated Traffic Volume is: [4495.41]

10. Testing:

➢ **Testing Strategy**

Traffictelligence is tested through **manual testing and functional validation** using the Gradio interface. Each module is designed as a user-facing interactive block, allowing rapid iteration and debugging.

➢ **Tools Used:**

- Google Colab runtime
- Gradio’s live UI for visual verification
- Print/log statements for model loading/debug fallback

11.Demo:

Demo Link:
https://drive.google.com/file/d/16zj2kUTtcbxn6w3hAmt6c7hDOuRi1Cne/view?usp=drivesdk

12. Known Issues:

Here are some known bugs or limitations that may affect users or developers:

- **Model Load Time**:
  The stable-diffusion-v1-5/stable-diffusion-v1-5 model is large and can take considerable time to load in Colab, especially on free-tier runtimes.
- **Memory Constraints in Google Colab**:
  Large models may cause memory overflows or slow performance on limited resources.
- **Fallback Model Simplicity**:
  The fallback model (DialoGPT-medium) is significantly less capable and may produce generic or unrelated outputs.
- **Response Variability**:
  Generative AI outputs may vary between runs and may include irrelevant or partially complete code.
- **No Persistent Storage**:
  There's no backend database or file-saving mechanism, so all results are lost when the session ends unless manually saved.
- **No Authentication**:
  The app does not currently restrict access or protect the Hugging Face API key, which can be a security risk in shared environments.

13. Future Enhancements:

To improve usability, scalability, and feature richness, the following enhancements are recommended:

Functional Enhancements:

- **Add More Modules**:
- Bug Fixing
- Test Case Generation
- Code Summarization
- Deployment Suggestions
- **Improve Prompt Engineering**:
  Use structured prompt templates and dropdowns to tailor AI responses more precisely.

Application & UI Improvements:

- **Persistent Storage**:
  Add integration with cloud storage or databases (e.g., Firebase or MongoDB) to save user inputs and outputs.
- **Export Functionality**:
  Allow users to download generated code and requirement summaries in .txt, .py, or .pdf format.
- **Code Execution Cell**:
  Let users run generated Python code inside the app using secure sandboxing (e.g., Code Interpreter or subprocess).
- **Enhanced Error Handling**:
  Provide clearer error messages, especially for model loading or API rate limits.

Security Enhancements:

- **API Key Protection**:
  Store the Hugging Face API key in environment variables or use OAuth-secured proxy to avoid exposure in notebooks.
- **User Authentication (Optional)**:
  Add login functionality if deployed publicly, especially when saving user data.

Deployment Options:

- **Dockerize the App**:
  Containerize with Docker for easier deployment on IBM Cloud or other cloud platforms.
- **Host on a Web Server**:
  Convert the app from Colab-based to a fully hosted app using Flask + Gradio, deployed on IBM Cloud or Hugging Face Spaces.