**Traffic Volume Prediction using Machine Learning and Flask**

**AI/ML Project Report**

**Team ID: LTVIP2025TMID45044**

**Team Size:** 4

**Team Members:**

Seelam Sai Sindhusha

Satvika Chilakala

Pechetti Mohana Nikitha

Pilla Kavya

**1. INTRODUCTION**

**1.1 Project Overview**

The objective of this project is to build an intelligent system that predicts the traffic volume on a city road using machine learning techniques. The model uses weather conditions, date, time, and holiday information as inputs to forecast vehicle volume. A Flask web application is built to make the system accessible and easy to use.

This system is highly useful in urban areas where traffic congestion leads to delays, stress, and economic loss. With the help of traffic prediction, travellers and transport authorities can make informed decisions.

**1.2 Purpose**

\* Predict traffic volume for given environmental and temporal conditions.

\* Help users plan travel and reduce congestion.

\* Learn and implement end-to-end ML deployment using Flask.

**2. IDEATION PHASE**

**2.1 Problem Statement**

Urban cities face increasing traffic congestion, which results in longer commute times and reduced quality of life. Traditional systems lack real-time adaptability. This project proposes a machine learning-based traffic volume prediction model that can forecast traffic patterns using structured data.

**2.2 Empathy Map Canvas**

| **Sees** | **Hears** | **Thinks & Feels** | **Says & Does** | **Pains** | **Gains** |
| --- | --- | --- | --- | --- | --- |
| Congested roads and traffic jams | Radio/FM traffic updates | “Will I be late again?” | Checks Google Maps frequently | Unpredictable delays and missed meetings | Plans travel better using traffic predictions |
| Long queues at signals | Complaints from other passengers or coworkers | Frustration due to slow traffic | Complains about bad traffic management | Wasted time, fuel, and energy | More efficient and relaxed commute |
| Traffic overlays on maps | Voice alerts from navigation apps | Stress during peak travel hours | Takes alternate routes or avoids peak hours | Inconsistent travel duration | Better time management and punctuality |
| Unresponsive or outdated traffic signs | Navigation instructions and alerts | Helplessness due to lack of control | Tries to leave earlier to avoid congestion | No reliable way to predict future traffic | Confidence in planning and timely arrival |

**2.3 Brainstorming Initiatives**

In the ideation phase, multiple ideas were explored to address the problem of traffic congestion prediction effectively. A structured brainstorming session was conducted to evaluate feasible approaches based on criteria such as adaptability, accuracy, scalability, and ease of deployment.

Ideas Considered:

| Approach | Pros | Cons |
| --- | --- | --- |
| Deep Learning Models | High accuracy, can capture complex patterns | Overkill for structured tabular data, requires large data |
| Manual Rule-based Systems | Easy to implement, interpretable | Not adaptable, fails with new/unseen conditions |
| Traditional Statistical Models | Transparent and fast | Lower performance with non-linear patterns |
| Machine Learning (ML) Models | Balanced performance, works well with structured features | Needs preprocessing, requires tuning |
| Hybrid ML + Real-time API | Dynamic, real-time adaptability | Increases system complexity |

After collaborative discussions and weighing the pros and cons, Machine Learning integrated with Flask was selected as the most balanced and effective approach for this project. This combination offers:

* Reliable performance for structured datasets
* Easy web-based deployment using Flask
* Maintainable and extendable codebase
* Quick integration with future enhancements (e.g., real-time APIs)

This decision aligns with the goals of the project: practical, efficient, and user-accessible traffic volume prediction.

**3. REQUIREMENT ANALYSIS**

**3.1 Customer Journey Map**

The customer journey for a user interacting with the traffic prediction system involves several touchpoints.

The map below outlines the experience flow:

| Stage | User Action | System Response | Emotion |
| --- | --- | --- | --- |
| Awareness | Hears about the app or accesses it online | Home page loads with prediction form | Curious, hopeful |
| Consideration | Inputs details: weather, date, time, etc. | Form validation begins | Focused |
| Action | Clicks on 'Predict Traffic' | Backend processes input and calls ML model | Expectant |
| Result | Sees prediction result (Heavy or Normal traffic) | Displays result with suggestions via HTML template | Informed, satisfied |
| Reflection | Plans route or travel based on prediction | Option to refresh or modify input | Empowered, confident |

This journey ensures that the user feels supported and informed at each step, reinforcing trust in the system's prediction accuracy.

**3.2 Solution Requirements**

Technical Requirements:

* Python 3.x
* Pandas, NumPy
* Scikit-learn
* Flask
* HTML/CSS for frontend
* Jupyter Notebook for experimentation

Functional Requirements:

* Accept user inputs (date, time, weather, holiday).
* Encode and preprocess features using pipeline components.
* Use trained ML model to predict traffic volume.
* Render prediction results dynamically via Flask routes.
* Allow simple navigation and error handling on the web app.

Non-Functional Requirements:

* Prediction Accuracy: ≥ 85%
* System Latency: Prediction within 2 seconds
* Scalability: Modular backend for future API integration
* Usability: Responsive UI across devices
* Reliability: Model error handling, fallback response on failure

**3.3 Data Flow Diagram (Level 1)**

The data flow of the system begins when the user submits inputs through the web form. These inputs include environmental (weather, temperature) and temporal (date, time, holiday) data. The Flask backend receives this data and applies preprocessing steps such as encoding, imputation, and scaling using saved pipeline objects.

The cleaned and transformed data is then passed to the trained Random Forest model to predict traffic volume. Depending on the predicted value, the system dynamically routes the user to either a "Heavy Traffic" or "Normal Traffic" result page. Logs and error handlers are in place to manage any invalid input or backend failures.

This flow ensures a smooth and reliable process from user interaction to intelligent decision-making.

**Diagram:**

**┌───────────────┐**

**│ User │**

**└─────┬────────┘**

**│ Inputs (Form: date, time, weather)**

**▼**

**┌───────────────┐**

**│ Flask Frontend │**

**└─────┬────────┘**

**│**

**▼**

**┌─────────────────────┐**

**│ Preprocessing │**

**│ Encoder + Imputer + │**

**│ Scaler │**

**└─────┬─────---───────┘**

**│**

**▼**

**┌────────────────────┐**

**│ Trained ML Model │**

**└─────┬─────────────┘**

**│ Prediction Output**

**▼**

**┌────────────────────┐**

**│ Render Result Page │**

**└────────────────────┘**

**3.4 Technology Stack**

| Layer | Technology/Tool | Purpose |
| --- | --- | --- |
| Frontend | HTML5, CSS3, Bootstrap (optional) | UI Design |
| Backend | Flask (Python) | Web server and routing |
| ML Model | Scikit-learn (Random Forest Regressor) | Model training and inference |
| Preprocessing | Pandas, NumPy, Joblib | Data cleaning and pipeline management |
| IDE & Tools | Jupyter Notebook, VS Code | Development and debugging |
| Hosting (Optional) | Heroku / Render / AWS | Cloud deployment |

**4. PROJECT DESIGN**

**4.1 Problem–Solution Fit**

Urban traffic congestion causes daily frustration, delays, and inefficiencies. Traditional systems like static traffic signals or historical data maps fail to adapt in real time. City planners, commuters, and logistics providers lack tools to predict future traffic conditions accurately and conveniently.

Our solution fits this need by providing:

* A machine learning model trained on historical weather, time, and traffic data
* A simple, intuitive Flask-based web application to predict traffic volume
* Real-time user input handling with fast feedback (<2 seconds)

This system offers a data-driven approach to proactively plan travel, reduce congestion, and support smart city initiatives.

**4.2 Proposed Solution**

The solution is designed as a full-stack, modular ML-powered web application:

Core Components:

1. Frontend Interface
   * A web form where users enter environmental and temporal data
   * Responsive and accessible design (HTML/CSS)
2. Backend Logic (Flask)
   * Receives inputs and preprocesses them using saved encoders/scalers
   * Uses a trained Random Forest model to predict traffic volume
3. Output Display
   * Dynamically renders output based on prediction result
   * Shows estimated traffic volume and classifies as "Normal" or "Heavy"
4. Model and Pipelines
   * Scikit-learn model serialized using Joblib
   * Preprocessing pipelines (encoder, imputer, scaler) applied consistently

This architecture ensures that the solution is accurate, fast, and user-friendly while remaining open to future enhancements like API integration or cloud deployment.

**4.3 Solution Architecture**

[ User Form Input ]

│

▼

[ Flask Route: /predict ]

│

▼

[ Preprocessing Pipeline ]

│

▼

[ Trained ML Model (Random Forest) ]

│

▼

[ Output Decision Logic ]

│

▼

[ Render HTML Template with Result ]

1.User inputs:

\* Weather

\* Temperature

\* Rain/Snow

\* Date & time

\* Holiday type

2. Backend process:

\* Inputs preprocessed (encode + scale)

\* Predict using trained model

\* Decision threshold to classify as High or Low traffic

3. Frontend response:

\* Renders either `chance.html` (heavy traffic) or `noChance.html` (normal traffic)

4.Technologies Used

Flask: Web application server

Scikit-learn:Model training & pipeline

Joblib: Model serialization

Jinja2: Templating

HTML/CSS: UI layer

**5. PROJECT PLANNING**

**5.1 Project Planning**

The project followed a structured 7-day sprint plan to ensure timely development and deployment of the traffic volume prediction system. Tasks were divided across stages such as data preparation, model training, backend integration, and frontend design.

Initial days were focused on data analysis, feature engineering, and model building using machine learning techniques. Once a suitable model was trained and evaluated, the focus shifted to building a Flask-based web application to make the predictions accessible to users. The final days were dedicated to integrating all components, testing functionality, and refining the user interface.

This agile-style planning approach enabled rapid development, early testing, and smooth integration of all system components within a short timeline.

🔹 Week 1 – End-to-End Development Sprint

| Day | Task | Outcome |
| --- | --- | --- |
| Day 1 | Data cleaning and exploratory data analysis | Understand data patterns and identify missing values |
| Day 2 | Feature engineering and transformation | Create meaningful features for prediction |
| Day 3 | Model selection and training (Random Forest) | Train a robust ML model and evaluate accuracy |
| Day 4 | Preprocessing pipeline setup | Create encoder, scaler, imputer, and save them |
| Day 5 | Backend development using Flask | Connect frontend with model via Flask routes |
| Day 6 | Frontend development (HTML/CSS) | Build a user-friendly interface |
| Day 7 | Integration and functional testing | Ensure the system works as a whole |

🔹 Project Milestones

* ✅ Dataset finalization – Day 1
* ✅ Model training and evaluation – Day 3
* ✅ Web interface functional – Day 5
* ✅ Complete system integration – Day 7

This schedule enabled rapid prototyping, iterative feedback, and early validation of the solution. Buffer time was also reserved for fixing bugs, improving UI/UX, and optimizing model performance.

**6. FUNCTIONAL & PERFORMANCE TESTING**

**6.1 Performance Testing**

To ensure the deployed system meets user expectations in terms of speed, efficiency, and robustness, performance testing was conducted on both the machine learning model and the Flask web application.

🔹 Key Performance Metrics

| Component | Metric | Target | Achieved |
| --- | --- | --- | --- |
| Model Inference | Time to predict (input → output) | ≤ 0.5 seconds | ~0.2 seconds |
| App Response | Total response time (form to result) | ≤ 2 seconds | ~1.3 seconds |
| Accuracy | R² Score on Test Data | ≥ 0.85 | 0.88 |
| MAE (Error Margin) | Mean Absolute Error | ≤ 300 vehicles | ~275 vehicles |
| Browser Load Time | Render speed (HTML pages) | ≤ 1 second | ~0.8 seconds (Chrome) |
| Resource Usage | CPU/RAM during prediction | Minimal (local CPU) | Within acceptable range |

🔹 Tools & Methods Used

* Jupyter Notebook: To evaluate model metrics (R², MAE)
* Flask Logs: For timing prediction and rendering
* Browser Dev Tools: For frontend load performance
* Manual Testing: Across devices and browsers (Chrome, Edge, Firefox)

🔹 Observations

* Model predictions are returned almost instantly, even with varied input data.
* Application remains responsive under repeated use.
* Frontend pages (index.html, chance.html, noChance.html) are lightweight and fast-loading.

🔹 Bottlenecks Identified

* Initial cold-start time in Flask (first prediction after server launch).
* Potential slowdown if deployed on free-tier cloud hosting.

**7. RESULTS**

**7.1 Model Performance**

Model Used: Random Forest Regressor

R² Score on Test Set:0.88

Mean Absolute Error:\~275 vehicles

**7.2 Sample Output**

Input:

\* Holiday: Veterans Day

\* Temp: 13.5°C

\* Rain: 0 mm

\* Weather: Cloudy

\* Hour: 17

Prediction:

Estimated Traffic Volume:4650 vehicles

Status: Heavy Traffic (redirects to `chance.html`)

**8. ADVANTAGES & LIMITATIONS**

✅ Advantages

* Accurate Predictions
* User-Friendly Interface
* Fast Inference Time
* Modular Code Structure
* End-to-End Deployment
* Lightweight & Local Hosting
* Scalability

❌ Disadvantages

* No Real-Time Data Integration
* Limited to Structured Data
* No Feedback Loop
* Cold Start Time
* Holiday & Weather Generalization

**9. CONCLUSION**

This project successfully demonstrates how machine learning can be applied to predict traffic volume based on structured environmental and temporal data. By integrating a trained Random Forest model with a user-friendly Flask web application, we developed a complete end-to-end solution that is fast, accurate, and practical.

The system empowers users to make informed travel decisions, helping reduce traffic congestion and improve commute efficiency. It also showcases the power of combining data science with web development for real-world impact.

Overall, the project met its objectives and lays the groundwork for further enhancements such as real-time data integration, cloud deployment, and advanced analytics.

**10. FUTURE SCOPE**

\* 🔗 Integration with live weather APIs

\* 📊 Dashboard for visual analytics

\* 🗺️ Google Maps interface for location-based prediction

\* ☁️ Deploy on cloud (AWS, GCP, or IBM Cloud)

\* 📈 Auto-retraining and log monitoring

**11. APPENDICES**

**11.1 Source code**

from flask import Flask, render\_template, request

import numpy as np

import joblib

import datetime

app = Flask(\_\_name\_\_)

# Load all components

model = joblib.load('model.pkl')

encoder = joblib.load('encoder.pkl')

imputer = joblib.load('imputer.pkl')

scaler = joblib.load('scale.pkl')

@app.route('/')

def home():

return render\_template('index.html')

@app.route('/predict', methods=['POST'])

def predict():

try:

# Step 1: Get data from form

holiday = str(request.form['holiday']).strip()

weather = str(request.form['weather']).strip()

temp = float(request.form['temp'])

rain = float(request.form['rain'])

snow = float(request.form['snow'])

hour = int(request.form['hours'])

year = int(request.form['year'])

month = int(request.form['month'])

day = int(request.form['day'])

minute = int(request.form['minutes'])

second = int(request.form['seconds'])

# Step 2: Get day of week

dt = datetime.datetime(year, month, day, hour, minute, second)

dayofweek = dt.weekday()

# Step 3: Prepare features

## 3.1 Categorical → encoder

cat\_features = encoder.transform([[holiday, weather]]).toarray()

## 3.2 Numerical → imputer → scaler

num\_raw = np.array([[temp, rain, snow, hour, dayofweek]])

num\_imputed = imputer.transform(num\_raw)

num\_scaled = scaler.transform(num\_imputed)

## 3.3 Combine final input

final\_input = np.concatenate((cat\_features, num\_scaled), axis=1)

# Step 4: Predict

prediction = model.predict(final\_input)[0]

result = f"Predicted Traffic Volume: {int(prediction)} vehicles"

# Step 5: Render result

if prediction >= 4000:

return render\_template('chance.html', result=result)

else:

return render\_template('noChance.html', result=result)

except Exception as e:

return f"❌ Error: {e}"

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

**11.2 Dataset Overview**

\* Source: UCI ML Repository

\* Records: 48,000+

\* Columns: `holiday`, `temp`, `rain`, `snow`, `weather`, `datetime`, `traffic\_volume`

**11.3 Tools Used**

\* Jupyter Notebook

\* VS Code

\* Flask

\* MS Word

\* GitHub for version control

**11.4 Git hub Link**

https://github.com/satvika1609/TrafficTelligence