PROJECT DESCRIPTION

DESCRIPTION OF FILES

1. dashboard.html

Location: Frontend

Purpose: Defines the **main UI** layout and structure of the web application.

Key Sections:

- **Header:** Navigation bar with links (Dashboard, Predict, About)
- **Dashboard Section:** Power BI dashboard preview with .pbix file download
- **Predictor Section:** A form collecting user input (like weather, road type, etc.)
- **About Section:** Project/course/team details
- **Prediction Modal:** Displays prediction result (accident severity & probability)

Why It's Used:

- Acts as the **user-facing interface**
- Renders dropdowns dynamically from Flask
- Makes user input collection and display highly intuitive

2. style.css

Location: Frontend

Purpose: Provides a **modern, responsive, and polished design** to the web app.

Key Styling Elements:

- Color scheme: Professional dark blue with soft highlights
- Form and button designs
- Modal pop-up styling
- Mobile responsiveness (media queries)

Why It's Used:

- Ensures **user-friendliness and aesthetics**
- Maintains visual consistency and modern UI standards

3. app.js

Location: Frontend (Client-side logic)

Purpose: Controls **interactive behavior** of the web application.

Key Functionalities:

- Smooth scroll navigation for single-page layout
- Handles **form submission via AJAX** to Flask backend
- Displays **prediction results** in a modal popup

Why It's Used:

- Enables **real-time prediction** without page reload
- Makes the UI feel smooth, modern, and interactive
- Handles modal logic and formats the input/result beautifully

4. app.py

Location: Backend (Python with Flask)

Purpose: Acts as the **core server-side logic and controller**.

Key Functionalities:

- Loads:
  - Trained Random Forest Model (model.pkl)
  - Label Encoders (encoders.pkl)
  - Target label encoder (target_encoder.pkl)
- **Routes:**
  - / → renders the HTML page with dropdown values
  - /predict → receives form data, encodes it, runs prediction, and returns a JSON result

Why It's Used:

- Core **bridge between frontend and ML model**
- Converts user input into model-readable format
- Converts prediction back to human-friendly output

5. train_model.py

Location: Backend (Model Training Script)

Purpose: Trains the **Random Forest ML model** and prepares it for deployment.

Key Steps:

- Loads traffic_accident_prediction.csv dataset
- Encodes categorical variables using LabelEncoder
- Splits data into features and target
- Trains a RandomForestClassifier model
- Saves:
  - model.pkl: trained model
  - encoders.pkl: encoders for input features
  - target_encoder.pkl: encoder for the severity target

Why It's Used:

- Enables **offline training and reproducibility**
- Ensures deployment-ready ML model and encoders

6. traffic_accident_prediction.csv

Location: Data Source

Purpose: Real-world or simulated dataset used to train the prediction model.

Key Columns:

- Weather, Road_Type, Time_of_Day, Traffic_Density, etc.
- Accident_Severity: Target label (Low, Moderate, High)

Why It's Used:

- This is the **foundation of model training**
- Helps the model learn patterns between road/driver conditions and accident severity

7. Traffic Accident Analysis.pbix

Location: Visualization File (Power BI)

Purpose: Data analysis & visualization using **Power BI**.

Key Dashboards:

- Monthly/Yearly accident distribution
- Road conditions and accident severity heatmap
- Correlation between traffic density and severity

Why It's Used:

- Provides **analytical insights** to complement the predictive tool
- Makes the project industry-ready with rich visual storytelling

Overall Integration

Component	Role
dashboard.html	User Interface (Input, Output display)
style.css	UI Design and Responsiveness
app.js	User Interaction and AJAX logic
app.py	Backend Prediction API via Flask
train_model.py	Model training pipeline
.csv file	Training dataset for the ML model
.pbix file	Visual analytics dashboard in Power BI

train_model.py — MODEL TRAINING SCRIPT (ML BACKEND)

```
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import LabelEncoder
import pickle
```

➞ **Imports**:

- pandas: For reading and manipulating the dataset
- RandomForestClassifier: The chosen ML model
- LabelEncoder: For converting categorical features to numeric
- pickle: For saving model and encoders

```
df = pd.read_csv('traffic_accident_prediction.csv')
```

➞ Loads the dataset into a DataFrame df.

```
categorical_cols = ['Weather', 'Road_Type', 'Time_of_Day',
'Road_Condition', 'Vehicle_Type', 'Road_Light_Condition']
```

➞ Lists all categorical columns to be encoded.

```
encoders = {}
for col in categorical_cols:
    le = LabelEncoder()
    df[col] = df[col].astype(str)  # Ensures consistent datatype
    df[col] = le.fit_transform(df[col])  # Applies encoding
    encoders[col] = le  # Saves encoder for deployment
```

➞ Encodes each categorical column using LabelEncoder and stores the encoders in a dictionary.

```
target_le = LabelEncoder()
df['Accident_Severity'] =
target_le.fit_transform(df['Accident_Severity'])
y = df['Accident_Severity']
```

➞ Encodes the **target column** (Accident_Severity) and stores it in y.

```
with open('target_encoder.pkl', 'wb') as f:
    pickle.dump(target_le, f)
```

➞ Saves the target encoder to reuse in the Flask app for decoding predictions.

```
FEATURES = [
    'Weather', 'Road_Type', 'Time_of_Day', 'Traffic_Density',
'Speed_Limit',
    'Number_of_Vehicles', 'Driver_Alcohol', 'Road_Condition',
    'Vehicle_Type', 'Driver_Age', 'Driver_Experience',
'Road_Light_Condition'
]
X = df[FEATURES]
```

➞ Defines the input features and creates X, the training data.

```
model = RandomForestClassifier()
model.fit(X, y)
```

➞ Trains the Random Forest model using the features and target labels.

```
with open('model.pkl', 'wb') as f:
    pickle.dump(model, f)
```

➞ Saves the trained model to a .pkl file.

```
with open('encoders.pkl', 'wb') as f:
    pickle.dump(encoders, f)
```

➞ Saves the dictionary of encoders for reuse in the Flask prediction app.

app.py — FLASK BACKEND

```
import pickle
import numpy as np
from flask import Flask, request, render_template, jsonify
```

➞ Imports libraries for:

- Loading models
- Working with arrays
- Handling Flask routes and JSON responses

```
app = Flask(__name__)
```

➞ Initializes a Flask web app.

```
with open('model.pkl', 'rb') as f:
    model = pickle.load(f)
with open('encoders.pkl', 'rb') as f:
    encoders = pickle.load(f)
with open('target_encoder.pkl', 'rb') as f:
    target_encoder = pickle.load(f)
```

➞ Loads the previously saved model and encoders.

```
FEATURES = [...]
```

➞ Lists all feature names to extract from the user form input.

```
@app.route('/')
def index():
    dropdowns = { ... }
    return render_template('dashboard.html', dropdowns=dropdowns)
```

➞ Renders the main HTML page and passes default dropdown values.

```
@app.route('/predict', methods=['POST'])
def predict():
    data = request.get_json()
    X = []
```

➞ Accepts prediction request via POST method and reads JSON input.

```
for feat in FEATURES:
    val = data.get(feat, "")
    if feat in encoders:
```

```
        try:
            val = encoders[feat].transform([val])[0]
        except Exception:
            val = 0
    else:
        try:
            val = float(val)
        except Exception:
            val = 0.0
    X.append(val)
```

➞ Encodes categorical inputs and converts numerics to float. Adds to X.

```
X = np.array(X).reshape(1, -1)
pred = model.predict(X)[0]
proba = model.predict_proba(X).max()
```

➞ Reshapes input, performs prediction, and gets confidence probability.

```
pred_label = target_encoder.inverse_transform([pred])[0]
accident_occurrence = "Yes" if pred_label.lower() not in ['none', 'no accident', 'no'] else "No"
severity_map = { ... }
severity_level = severity_map.get(pred_label.lower(), pred_label)
```

➞ Translates predicted label into readable accident result and severity level.

```
return jsonify({ ... })
```

➞ Sends back prediction result as JSON to frontend.

```
if __name__ == '__main__':
    app.run(debug=True)
```

➞ Starts Flask server when executed.

dashboard.html — Main Web Page Template

This is the **frontend template** rendered by Flask. It is a complete HTML5 page that contains the UI for dashboard preview, prediction form, and project details.

<head> Section

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Traffic Accident Severity Predictor</title>
```

- Declares document type as HTML5.
- Sets the page language to English.
- Sets the browser tab title.

```html
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

- Ensures responsive design on mobile devices.

```html
    <link rel="stylesheet" href="/static/style.css">
```

- Loads external **CSS file** (style.css) for custom styling.

```html
    <link href="https://fonts.googleapis.com/css2?family=Montserrat:wght@400;700&display=swap" rel="stylesheet">
```

- Loads **Montserrat** font from Google Fonts to give a modern look.

<body> Section

Header

```html
<header>

    <div class="logo">🚦 Traffic Insight Pro</div>
```

- Displays project name/logo in the top navigation bar.

```html
  <nav>

        <a href="#dashboard-section" class="nav-link">Dashboard</a>

        <a href="#predictor-section" class="nav-link">Predict</a>

        <a href="#about-section" class="nav-link">About</a>

    </nav>

</header>
```

- Top navigation links using anchor IDs.
- Clicking scrolls to sections smoothly (enabled via JavaScript).

Dashboard Section

```html
<section class="dashboard-section" id="dashboard-section">

    <h2>Traffic Accident Dashboard</h2>
```

- Heading for the dashboard analytics section.

```html
    <div class="dashboard-image-container">

        <a href="/static/Traffic Accident Analysis.pbix" target="_blank" download>
```

```
            <img  src="/static/powerbi_dashboard.jpg"  alt="Power  BI 
Dashboard" class="dashboard-image" />
        </a>
        <p  class="dashboard-caption">Click  the  image  to  open  the 
interactive Power BI dashboard (.pbix file)</p>
    </div>
</section>
```

- Shows an image (preview) of Power BI dashboard.
- Allows user to download the actual .pbix file.

Predictor Form Section

```
<section class="predictor-section" id="predictor-section">
    <h2>Predict Accident Severity</h2>
    <form id="predict-form" autocomplete="off">
```

- A structured input form where users submit data to get predictions.

```
<div class="form-group">
    <label for="Weather">Weather</label>
    <select id="Weather" name="Weather">
        {% for val in dropdowns['Weather'] %}
        <option value="{{val}}">{{val}}</option>
        {% endfor %}
    </select>
</div>
```

- Uses Flask templating ({% for ... %}) to populate dropdowns dynamically.
- One such block is repeated for every input field: Road_Type, Time_of_Day, etc.

```
<button type="submit" class="predict-btn">Predict</button>
```

- Submit button that sends form data to Flask backend (/predict).

```
<div id="prediction-result" class="result-box"></div>
```

- Placeholder to show result (accident chance/severity) returned via JS.

About Section

```
<section class="about-section" id="about-section">
    <h2>About</h2>
```

```
    <div class="about-content">
        <p><strong>Programme Name:</strong> B. Tech in CSE - DS</p>
        ...
    </div>
</section>
```

- Lists course name, team members, and project code.

Footer

```
<footer>
    &copy; 2025 Traffic Insight Pro. All rights reserved.
</footer>
```

- Footer with copyright.

Modal Popup Template

```
<div id="result-modal" class="modal">
    <div class="modal-content">
        <span class="close-btn" id="close-modal">&times;</span>
        <h3>Prediction Result</h3>
        <ul id="input-params-list"></ul>
        <div id="modal-prediction-output"></div>
    </div>
</div>
```

- Hidden by default.
- JS shows this after prediction to neatly display inputs and results.

```
<script src="/static/app.js"></script>
```

- Links the JavaScript file which handles form submission and UI behavior.

app.js — Client-side JavaScript Logic

This script controls form handling, AJAX calls, and modal UI.

```
document.addEventListener('DOMContentLoaded', function() {
```

- Waits until the DOM is fully loaded before executing any script.

Navigation Scroll

```
document.querySelectorAll('.nav-link').forEach(link => {
```

```
    link.addEventListener('click', function(e) {
        ...
    });
});
```

- Enables **smooth scrolling** to sections when navigation items are clicked.

Modal Setup

```
const modal = document.getElementById('result-modal');
const closeBtn = document.getElementById('close-modal');
const paramsList = document.getElementById('input-params-list');
const modalOutput = document.getElementById('modal-prediction-output');
```

- Gets references to modal HTML elements.

```
function showModal(inputs, prediction) {
    ...
}
```

- Dynamically builds the modal content:
  - Lists all input fields and values
  - Shows prediction result

Close Modal Logic

```
closeBtn.onclick = function() {
    modal.style.display = 'none';
};
window.onclick = function(event) {
    if (event.target === modal) {
        modal.style.display = 'none';
    }
};
```

- Allows modal to close on "X" click or clicking outside the box.

Form Submission Logic

```
const form = document.getElementById('predict-form');
form.addEventListener('submit', function(e) {
```

```
    e.preventDefault();
    ...
});
```

- Prevents normal form submission.
- Gathers input values and sends them as JSON via fetch.

```
fetch('/predict', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(data)
})
.then(resp => resp.json())
.then(res => {
    showModal(data, res);
})
```

- Sends input to Flask /predict route.
- Receives prediction and passes to showModal.

```
.catch(() => {
    showModal({}, {
        accident: "<span style='color:red;'>Prediction failed.</span>",
        severity: "",
        probability: ""
    });
});
```

- If the server fails, shows a graceful error message.

style.css — Styling for Web App

Defines all visual and layout rules.

Base Styles

```
body {
    margin: 0;
    font-family: 'Montserrat', sans-serif;
    background: #f2f4f8;
```

```
    color: #222;
}
```

- Removes default margins and applies the modern Montserrat font.

Header & Navigation

```
header {
    background: #232946;
    color: #fff;
    ...
}
nav a:hover, nav a.active {
    color: #eebbc3;
    text-decoration: underline;
}
```

- Creates a sticky dark-blue topbar with hover styles.

Sections & Layout

```
main {
    max-width: 1200px;
    margin: 2rem auto;
    padding: 1rem;
}
section {
    scroll-margin-top: 80px;
}
```

- Responsive layout with padding and spacing for each section.

Dashboard & Form Cards

```
.dashboard-section, .predictor-section, .about-section {
    background: #fff;
    border-radius: 16px;
    ...
}
```

- Gives each section a **card-like appearance** with padding and shadow.

Form Inputs

```
.form-group input,
.form-group select {
    padding: 0.5rem;
    ...
}
```

- Beautiful, readable input fields and dropdowns.

Predict Button

```
.predict-btn {
    background: #232946;
    color: #fff;
    ...
}
.predict-btn:hover {
    background: #eebbc3;
    color: #232946;
}
```

- Bold button that inverts colors on hover.

Modal Styling

```
.modal {
    display: none;
    ...
}
.modal-content {
    background-color: #fff;
    ...
}
```

- Styles for modal: background overlay, animated pop-up effect, close button.

Responsive Design

```
@media (max-width: 900px) {
    .form-row {
```

```
        flex-direction: column;
    }
}
```

- Makes layout stack vertically on smaller devices.

Summary

File	Role	Core Use
train_model.py	Training the ML model	Offline pre-deployment
app.py	Backend logic & prediction API	Bridges UI & model
dashboard.html	UI structure and rendering template	Form & layout
app.js	AJAX & modal logic	Dynamic interaction
style.css	Visual styling & responsiveness	Modern UX

What is a Modal in JavaScript?

A **modal** in JavaScript is a **popup dialog box** or **overlay** that appears on top of the main page content, often used to:

- Display messages or alerts,
- Show forms or predictions (like in your project),
- Confirm actions from the user.

It **blocks interaction** with the rest of the page until the user closes or submits it — this is why it's often called a **modal window** or **modal dialog**.

Characteristics of a Modal

- Appears above everything else (usually with z-index)
- Usually includes a **background overlay**
- Can be dismissed with a close button (X) or by clicking outside
- Doesn't require a new page to open (unlike alert boxes)

Example from Project (app.js)

In app.js, this block defines the modal logic:

```
const modal = document.getElementById('result-modal');   // Get modal element

const closeBtn = document.getElementById('close-modal'); // Get close button

function showModal(inputs, prediction) {

    // Fill content dynamically

    modal.style.display = 'block';  // Show modal
```

```
}
closeBtn.onclick = function() {
    modal.style.display = 'none';   // Close on 'X' click
}
```

So here, the **modal is used to display the prediction result** (like severity level, chance of accident) in a clean popup without reloading or navigating away.

In Summary:

- A **modal** is a custom popup overlay.
- It's usually built with HTML (structure), CSS (visibility & animation), and JavaScript (functionality).
- It improves UX by avoiding page reloads and highlighting important information clearly.

Absolutely! Let's dive into a **clear and thorough explanation of Random Forest** and all its **related concepts** as used in your **Traffic Insight Pro project** — from the basics to the way it's implemented in your code.

What is Random Forest?

Definition:

Random Forest is an **ensemble machine learning algorithm** that builds **multiple decision trees** and **combines their outputs** to improve prediction accuracy and reduce overfitting.

It can be used for:

- **Classification** (like predicting accident severity),
- **Regression** (like predicting house prices).

Why Random Forest?

In the project, the goal is to **predict accident severity** based on features like weather, traffic, speed, etc. You use **Random Forest** because:

- It's robust to noisy data.
- It works well with mixed data types (categorical + numeric).
- It handles overfitting better than a single decision tree.
- It gives good accuracy **without needing much hyperparameter tuning**.

Key Concepts of Random Forest

1. Decision Tree

A decision tree makes decisions by **asking questions** like:

"Is speed > 60?"

"Is road wet?"

"Is it nighttime?"

And then **splits the data** into branches until it reaches a prediction.

Problem: One tree might **overfit** (memorize training data).

2. Random Forest

A random forest:

- **Builds many decision trees** on **random subsets** of the data and features.
- **Averages their outputs** (majority vote for classification).

It avoids overfitting by:

- Using **bagging** (Bootstrap Aggregation)
- Using **random feature selection**

How It's Implemented in Your Project

📄 **File: train_model.py**

Importing the Random Forest classifier

```
from sklearn.ensemble import RandomForestClassifier
```

This loads the RandomForestClassifier from scikit-learn.

Feature Engineering

```
FEATURES = [ ... ]
X = df[FEATURES]
y = df['Accident_Severity']
```

- X contains your **input features**: weather, road type, time of day, etc.
- y is the **target** you want to predict: Accident_Severity.

Training the Model

```
model = RandomForestClassifier()
model.fit(X, y)
```

- RandomForestClassifier() creates the model with default settings.
- .fit(X, y) trains the model on your dataset.

Each tree in the forest is trained on **a random sample of data** with **a random subset of features**. This ensures **diversity** in the trees.

Saving the Model

```
import pickle
with open('model.pkl', 'wb') as f:
    pickle.dump(model, f)
```

This saves the trained model to a .pkl file for deployment in your Flask app (app.py).

📄 **File: app.py**

Predicting with the Model

```
X = np.array(X).reshape(1, -1)
```

```
pred = model.predict(X)[0]
proba = model.predict_proba(X).max()
```

- model.predict() gives the predicted severity class (e.g., Low, Moderate, High).
- model.predict_proba() returns the **probabilities for each class**, and .max() gives the confidence level.

Related Concepts

1. Label Encoding

from sklearn.preprocessing import LabelEncoder

Used to **convert categorical values to numeric** (e.g., "Rainy" → 2). Required for ML models to process string features.

2. Pickle

Used to **serialize (save) the model and encoders** so they can be reused without retraining every time.

3. Inference vs Training

- **Training** = train_model.py: Fit the model once.
- **Inference** = app.py: Use the trained model to predict new data.

How Random Forest Works Internally (Visualized)

Let's say you have 100 training examples.

1. **Tree 1**: Randomly selects 60 samples + 5 features
2. **Tree 2**: Selects a different 60 samples + 5 features
3. **...** Builds 100 trees like this
4. **Final Prediction**: Takes the majority vote from all trees

This is called **ensemble learning**.

Pros of Random Forest (Why You Chose It)

- High accuracy
- Robust to overfitting
- Works well even without much hyperparameter tuning
- Handles missing values and noisy data
- Can rank feature importance (optional)

Limitations (To Keep in Mind)

- Slower for very large datasets (many trees)

- Harder to interpret compared to a single decision tree
- Size of model can be large (but not a big issue here)

Real-World Analogy

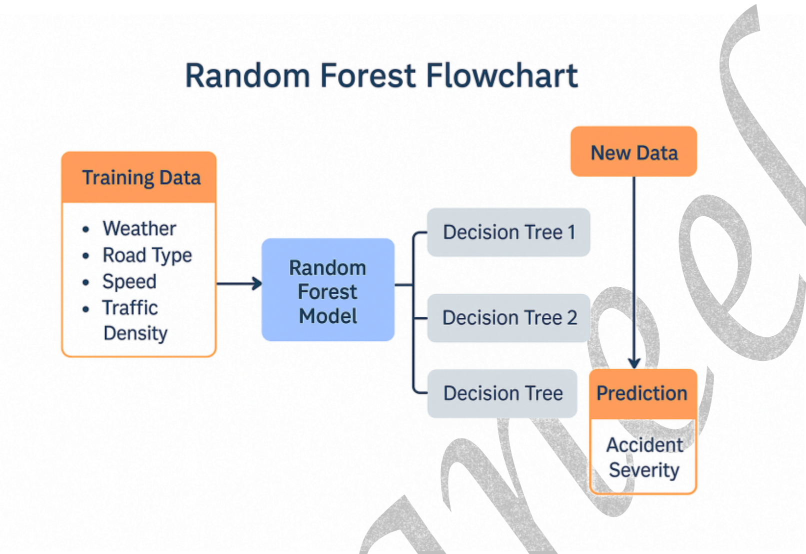
Imagine asking **100 doctors** about a diagnosis.

Each one gives their opinion after looking at **slightly different symptoms**.

You go with the **majority vote**.

That's what Random Forest does!

A **visual flowchart** of how Random Forest works:



What Are Encoders?

Definition:

Encoders are **conversion tools** that **transform categorical (text) data into numeric values** — because machine learning models can only understand numbers.

For example:

Original	Encoded
Clear	0
Rainy	1
Stormy	2

This mapping is done using **LabelEncoder()** from sklearn.preprocessing.

Why You Need Encoders in This Project

Your dataset has several **categorical fields**:

- Weather: Clear, Rainy, Foggy, Stormy...
- Vehicle_Type: Car, Truck, Motorcycle...
- Road_Type: City Road, Highway, etc.

If you send these raw text labels to the ML model, it will **crash** or produce **wrong results**. You need to convert them to numbers first — that's where encoders come in.

Where and How Are Encoders Used?

In train_model.py — During Training

```
from sklearn.preprocessing import LabelEncoder

categorical_cols = ['Weather', 'Road_Type', 'Time_of_Day', 'Road_Condition', 'Vehicle_Type', 'Road_Light_Condition']

encoders = {}

for col in categorical_cols:

    le = LabelEncoder()

    df[col] = le.fit_transform(df[col])

    encoders[col] = le
```

Explanation:

- For each categorical column, you:
    - Create a LabelEncoder
    - .fit_transform() it (learns mapping + applies it)
    - Store that encoder in a dictionary called encoders

So, if Weather = Stormy, encoders["Weather"] knows how to convert "Stormy" → 2 (say).

Saved as Pickle:

```
with open('encoders.pkl', 'wb') as f:

   pickle.dump(encoders, f)
```

This saves **all trained label encoders** in a file for future use.

What Is a Pickle?

Definition:

Pickle is Python's way to **serialize (save) Python objects to disk** so you can load them later exactly as they were.

In Your Project:

You save 3 pickle files:

File	What It Stores
`model.pkl`	Trained Random Forest model
`encoders.pkl`	Dictionary of feature-wise LabelEncoders
`target_encoder.pkl`	Encoder for Accident_Severity (Low, Moderate...)

This way, you don't retrain every time. You just **load and predict**.

How They Work Together in app.py

```
with open('model.pkl', 'rb') as f:
```

```
    model = pickle.load(f)
with open('encoders.pkl', 'rb') as f:
    encoders = pickle.load(f)
with open('target_encoder.pkl', 'rb') as f:
    target_encoder = pickle.load(f)
```

These lines **reload everything** that was trained earlier.

Using Encoders at Prediction Time

```
for feat in FEATURES:
    val = data.get(feat, "")
    if feat in encoders:
        val = encoders[feat].transform([val])[0]  # Convert text to number
    ...
```

- This loop takes form input like "Stormy" and uses the encoder to convert it to 2.
- Without this conversion, your model won't work.

Using target_encoder for Output

```
pred_label = target_encoder.inverse_transform([pred])[0]
```

Your model predicts a **numeric class** (e.g., 0, 1, 2).

But you want to show the **human-readable label** ("Low", "Moderate", "High").

This line converts numeric → label using `target_encoder`.

Real-World Analogy

Think of **encoders** as a **translator**:

- When you train, it learns that "Stormy" = 2.
- When you predict, it helps convert "Stormy" back to 2.
- When the model outputs "1", it helps convert it back to "Moderate".

And **pickle** is like a **USB drive**:

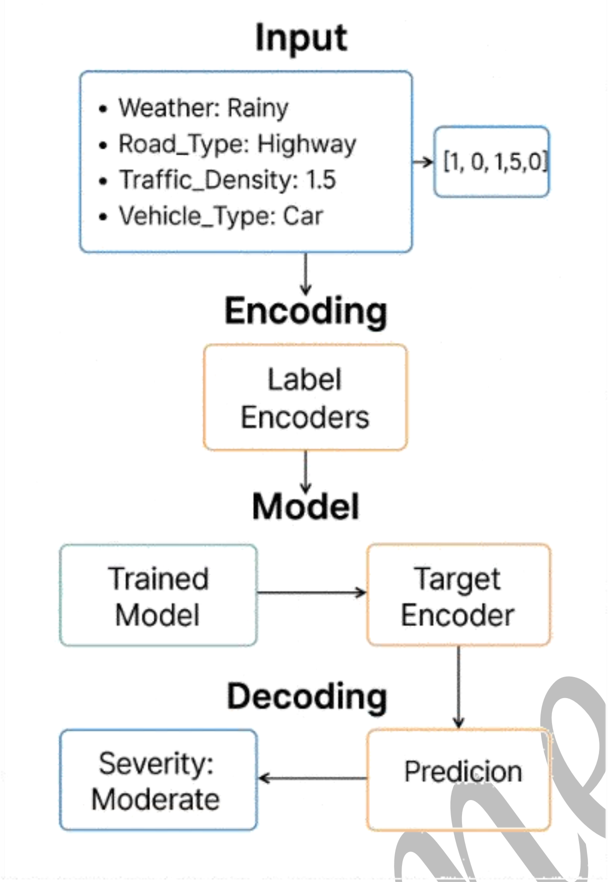
- You store the translator (encoder) and trained model on it.
- You can plug it in any time without re-learning from scratch.

Summary Table

Concept	***Purpose***	***Used In***
LabelEncoder	Convert text features to numeric	train_model.py, app.py
encoders.pkl	Stores all input encoders	train_model.py (created), app.py (used)
target_encoder	Converts severity label (Low/High) ↔ number	train_model.py, app.py

model.pkl	Stores trained ML model	Used for inference
pickle module	Saves and loads models/encoders to disk	Everywhere

A visual diagram showing how input → encoding → model → decoding



What is a Target Encoder?

In your project, the **target encoder** is a specific LabelEncoder used to **convert the target column — Accident_Severity — from text to numbers and vice versa**.

Why It's Called "Target" Encoder?

Because it encodes and decodes the **target variable** (i.e., the output label that your model is trying to predict).

In Your Dataset

The target column looks like this (raw data):

Accident_Severity
Low
Moderate
High
Severe

ML models can't process strings like "Moderate" — so we convert them to numbers:

Accident_Severity	Encoded
Low	0

Moderate	1
High	2

This is done using:

```
from sklearn.preprocessing import LabelEncoder

target_le = LabelEncoder()

df['Accident_Severity'] = target_le.fit_transform(df['Accident_Severity'])
```

The fitted encoder is saved as:

```
pickle.dump(target_le, open('target_encoder.pkl', 'wb'))
```

Where is the Target Encoder Used?

In train_model.py

To **convert accident severity to numbers** so the model can be trained:

```
df['Accident_Severity'] = target_le.fit_transform(df['Accident_Severity'])
```

In app.py

After prediction, the model returns a **number** (e.g., 1). You need to **decode** that back into "Moderate" to display to the user:

```
pred_label = target_encoder.inverse_transform([pred])[0]
```

So, internally:

Model says: 2

Target encoder says: "High"

Real-Life Analogy

Imagine your target encoder is like a **legend on a map**:

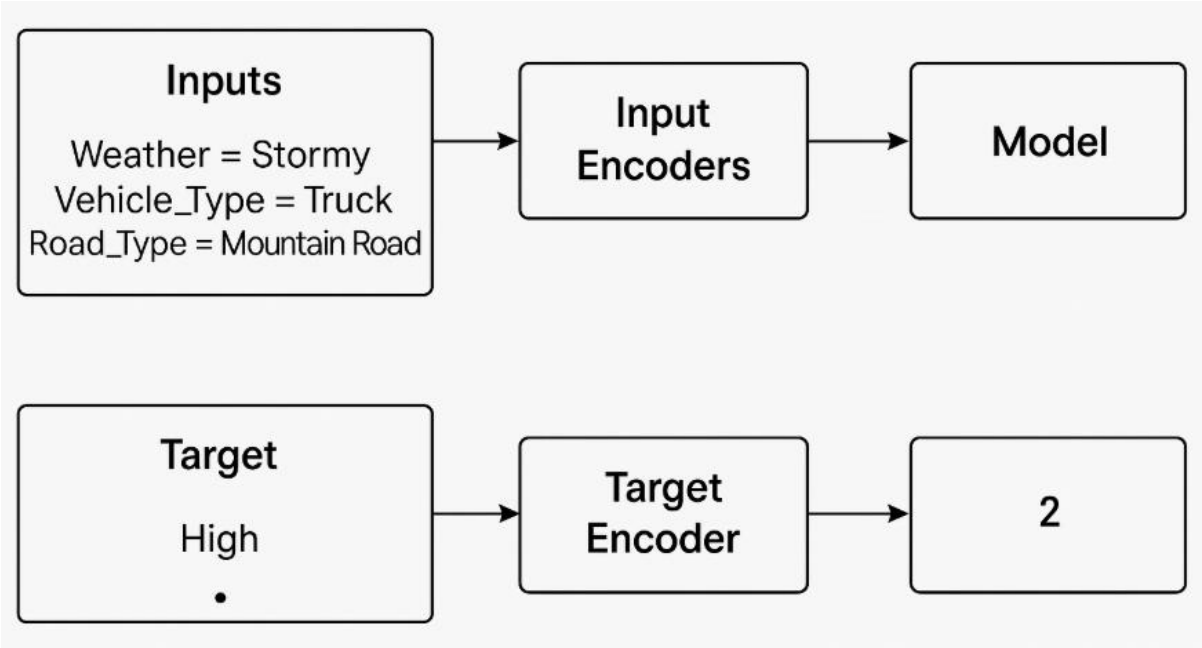
- Model says: Zone = 1
- Legend tells you: Zone 1 = "Moderate severity"

Without the target encoder, the app would show the user a raw number, not a human-friendly label.

Summary Table

Aspect	Encoder
Used for inputs	encoders.pkl (dict of encoders for features)
Used for outputs	target_encoder.pkl (LabelEncoder for severity)
Encodes from	"Low" → 0
Decodes back to	0 → "Low"

A **chart** that compares input encoders vs target encoder visually:


Inputs
Weather = Stormy
Vehicle_Type = Truck
Road_Type = Mountain Road
Input Encoders
Model
Target
High
Target Encoder
2