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

* **pickle**: Used to load the saved ML model and encoders (model.pkl, encoders.pkl, target\_encoder.pkl).
* **numpy**: For reshaping feature vectors into a format the model understands.
* **Flask** components:
  + Flask: Initializes the web app.
  + request: Accesses form data sent by the frontend.
  + render\_template: Renders the HTML page.
  + jsonify: Sends back the prediction as a JSON response.

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

* Creates the Flask application object named app.

**Load Model and Encoders**

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 your **pre-trained Random Forest model**, input feature **label encoders**, and **target encoder** (to decode severity levels).
* All three were saved during training (train\_model.py).

**Define Features**

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'

]

* List of all **input fields** expected from the user.
* Must be in the **same order** used during model training.

**Index Route**

@app.route('/')

def index():

dropdowns = {

'Weather': ['Clear', 'Rainy', 'Foggy', 'Stormy', 'Snowy'],

'Road\_Type': ['City Road', 'Rural Road', 'Highway', 'Mountain Road'],

'Time\_of\_Day': ['Morning', 'Afternoon', 'Evening', 'Night'],

'Traffic\_Density': ['0.0', '1.0', '2.0'],

'Road\_Condition': ['Dry', 'Wet', 'Icy', 'Under Construction'],

'Vehicle\_Type': ['Car', 'Truck', 'Bus', 'Motorcycle'],

'Road\_Light\_Condition': ['Daylight', 'Artificial Light', 'No Light'],

}

return render\_template('dashboard.html', dropdowns=dropdowns)

* Renders the HTML form (dashboard.html).
* Sends dropdown options to populate <select> inputs in the form using Flask templating.

**Prediction Route**

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

def predict():

* Defines the **API endpoint** /predict that receives input data from the form **via POST request**.

**Get JSON Data**

data = request.get\_json()

X = []

* data: Contains user input from the form as a dictionary.
* X: Will hold numeric, model-ready values after processing.

**Loop Over Features**

for feat in FEATURES:

val = data.get(feat, "")

* Retrieves the value for each feature from the incoming data dictionary.

**Categorical Feature Encoding**

if feat in encoders:

try:

val = encoders[feat].transform([val])[0]

except Exception:

val = 0

* If the feature is categorical:
  + Uses its corresponding **LabelEncoder** to transform it into a number.
  + If transformation fails (e.g., unknown value), sets it to 0 as fallback.

**Numeric Conversion**

else:

try:

val = float(val)

except Exception:

val = 0.0

* Converts numeric inputs (e.g., Speed\_Limit) to float.
* If invalid, assigns a default value 0.0.

**Add to Feature Vector**

X.append(val)

* Adds the processed value (encoded or numeric) to the input list.

**Prepare for Prediction**

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

* Converts the list X to a 2D NumPy array of shape (1, 12) — required by scikit-learn models.

**Make Prediction**

pred = model.predict(X)[0]

proba = model.predict\_proba(X).max()

* predict() returns the encoded prediction label (e.g., 2).
* predict\_proba() returns probability scores for all classes. .max() gives the **confidence**.

**Decode Predicted Label**

try:

pred\_label = target\_encoder.inverse\_transform([pred])[0]

except Exception as e:

print("Decoding error:", e)

pred\_label = str(pred)

* Converts the numeric prediction back into the original string label using target\_encoder.
* If decoding fails, fallback to the numeric string.

**Map to Standard Categories**

severity\_map = {

'fatal': 'High',

'major': 'High',

'severe': 'High',

'high': 'High',

'moderate': 'Moderate',

'medium': 'Moderate',

'low': 'Low',

'minor': 'Low',

'no accident': 'Low',

'none': 'Low'

}

severity\_level = severity\_map.get(str(pred\_label).strip().lower(), 'Low')

* Ensures that all prediction labels (e.g., "Severe", "Fatal") are **mapped to consistent categories**: "High", "Moderate", "Low".

**Determine If Accident Occurred**

accident\_occurrence = "Yes" if severity\_level != 'Low' else "No"

* Any prediction other than "Low" means an accident occurred.

**Return Prediction**

return jsonify({

'accident': accident\_occurrence,

'severity': severity\_level,

'probability': f"{proba\*100:.2f}%"

})

* Sends back a JSON response with:
  + Whether accident occurred
  + Severity level
  + Confidence probability (formatted as a percentage)

**Run the App**

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

* Launches the Flask server in debug mode.
* Allows local testing at http://127.0.0.1:5000

**Summary Flow**

User Input → JSON → Flask → Encoders → Model → Prediction → Decode → Format → JSON Response

This architecture is:

* 🔒 Safe (with exception handling)
* 💡 Interpretable (clear labels)
* 💻 Efficient (minimal computation and encoding logic)

## 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.

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

* Ensures responsive design on mobile devices.

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

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

<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**

<header>

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

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

<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**

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

<h2>Traffic Accident Dashboard</h2>

* Heading for the dashboard analytics section.

<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**

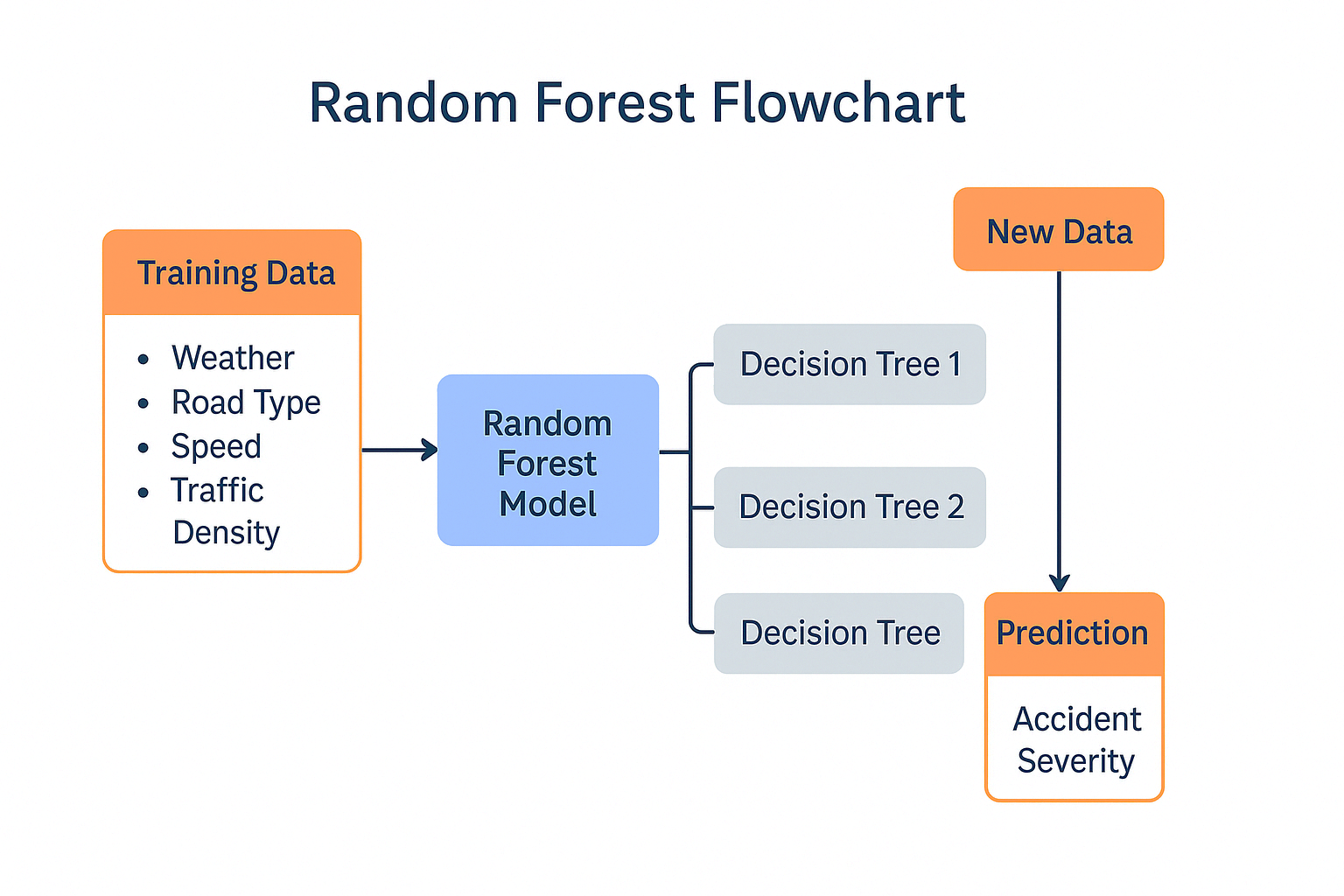
Imagine asking **100 doctors** about a diagnosis.

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”.

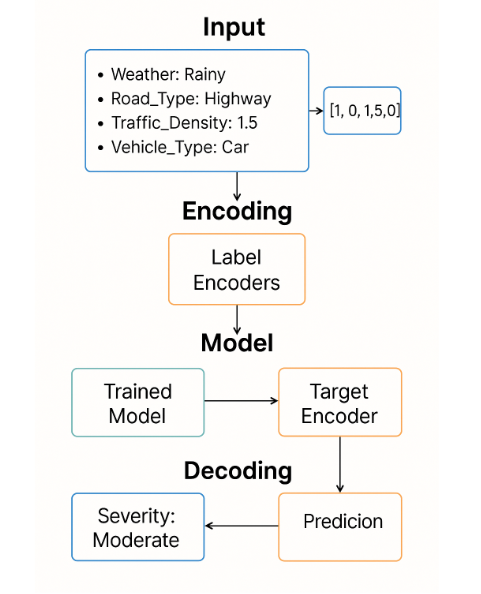
And **pickle** is like a **USB drive**:

* 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:

