

Railway accident analytics project presentation prep :

Ai chatbot page :

Okay, let's dissect the `llama_Assistant.py` code step-by-step to understand how this chatbot functions. I'll break it down into sections, explaining each part's role and how they interact.

python

```
import pandas as pd
```

```
import streamlit as st
```

```
import requests
```

- **Imports:** These lines import the necessary Python libraries:
 - `pandas`: Used for data manipulation and analysis, specifically for reading and working with the railway accident data in CSV format.
 - `streamlit`: This is the core library for creating the interactive web application. It allows you to build the user interface with simple Python commands.
 - `requests`: A library for making HTTP requests. In this case, it's used to send requests to the Groq API to get responses from the LLaMA model.

python

```
def show():
```

- **show() function:** This function encapsulates the entire chatbot logic. It's likely called from your `main.py` file to display the chatbot interface on a specific page of your Streamlit app.

python

```
# ✅ API Setup (For general safety queries only)
```

```
GROQ_API_URL = "https://api.groq.com/openai/v1/chat/completions"
```

```
GROQ_API_KEY = "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"
```

```
# Replace with your actual API key
```

- **API Setup:** This section defines the Groq API endpoint and your API key.

- GROQ_API_URL: The URL for the Groq API, which you're using to access the LLaMA language model.
- GROQ_API_KEY: Your secret API key, which authenticates your requests to the Groq API. **Important:** Never hardcode your API key directly in your code if you plan to share it. Use environment variables or a more secure method.

python

```
# ✅ Load Railway Accident Data

@st.cache_data

def load_data():

    try:
```

```
        df = pd.read_csv("C:/Users/yashw/Desktop/6 th sem project
ideas/Indian_Railways_Accidents_Dataset_1902_2024.csv")
```

```
    return df
```

```
except FileNotFoundError:
```

```
    st.error("⚠️ Railway accident dataset not found!")
```

```
    return None
```

```
df = load_data()
```

- **Data Loading:** This section loads the railway accident data from a CSV file using pandas.
 - `@st.cache_data`: This Streamlit decorator caches the output of the `load_data()` function. This means that the data is only loaded once when the app starts, and subsequent calls to `load_data()` will use the cached data, improving performance.
 - `load_data()`: This function attempts to read the CSV file into a pandas DataFrame.
 - **Error Handling:** If the CSV file is not found, it displays an error message in the Streamlit app.
 - `df = load_data()`: This line calls the `load_data()` function and stores the resulting DataFrame (or None if there was an error) in the `df` variable.

python

Query Functions for Dataset

```
def get_accident_stats():
```

```
    """Returns general statistics from the dataset."""
```

```
    return df.describe().to_string() if df is not None else "No data available."
```

```
def get_accidents_by_cause(cause):
```

```
    """Returns accidents related to a specific cause."""
```

```
    if df is not None:
```

```
        filtered_data = df[df["Cause"].str.contains(cause, case=False, na=False)]
```

```
    return filtered_data.to_string(index=False) if not filtered_data.empty else "No records found."
```

```
    return "No data available."
```

```
def get_accidents_by_state(state):
```

```
    """Returns accidents that occurred in a specific state."""
```

```
    if df is not None:
```

```
        filtered_data = df[df["State"].str.contains(state, case=False, na=False)]
```

```
    return filtered_data.to_string(index=False) if not filtered_data.empty else "No records found."
```

```
    return "No data available."
```

```
def get_accidents_by_year(year):
```

```
    """Returns accidents that occurred in a specific year."""
```

```
    if df is not None:
```

```
        filtered_data = df[df["Year"] == int(year)]
```

```
    return filtered_data.to_string(index=False) if not filtered_data.empty else "No records found."
```

```
    return "No data available."
```

- **Query Functions:** These functions define how to extract specific information from the railway accident DataFrame.
 - Each function filters the DataFrame based on a specific criteria (e.g., cause, state, year).
 - If the DataFrame is empty or None, they return a "No data available." message.
 - The results are returned as strings.

python

```
# ✅ AI Chat Function (Updated Model)

def chat_with_ai(prompt):
    # 🔔 Include a summary of the available data in the prompt

    data_summary = f"The available data on Indian railway accidents spans from 1902 to 2024. The model MUST base its answers primarily on this dataset, but can use outside information if required. Do not hallucinate data, instead respond that the data does not exist."

    enhanced_prompt = f"{prompt}. {data_summary}" # Augment the prompt

    headers = {"Authorization": f"Bearer {GROQ_API_KEY}", "Content-Type": "application/json"}

    data = {
        "model": "llama3-8b-8192",
        "messages": [{"role": "user", "content": enhanced_prompt}],
        "max_tokens": 750 # Increased token limit
    }

    response = requests.post(GROQ_API_URL, json=data, headers=headers)

    try:
        response_json = response.json()

        return response_json["choices"][0]["message"]["content"] if "choices" in response_json else f"⚠️ API Error: {response_json}"
    except Exception as e:
```

```
return f"⚠️ Error: {str(e)}"
```

- **AI Chat Function:** This is the core of the chatbot, responsible for interacting with the LLaMA model.
 - `data_summary`: Creates a string summarizing the available data range (1902-2024). Crucially, it also tells the model not to hallucinate data.
 - `enhanced_prompt`: Combines the user's prompt with the `data_summary` to provide context to the model. This helps the model generate more relevant and accurate responses.
 - `headers`: Sets the headers for the HTTP request, including the authorization token.
 - `data`: Constructs the JSON payload that will be sent to the Groq API. It includes:
 - `model`: The name of the LLaMA model you're using.
 - `messages`: A list of messages. In this case, there's only one message, the user's prompt (`enhanced_prompt`).
 - `max_tokens`: The maximum number of tokens the model should generate in its response. This is set to 750 to allow for longer responses.
 - `requests.post()`: Sends a POST request to the Groq API with the specified URL, headers, and data.
 - **Error Handling:** The `try...except` block handles potential errors during the API call. If an error occurs, it returns an error message to the Streamlit app.
 - **Response Parsing:** If the API call is successful, it parses the JSON response and extracts the model's generated content.

python

```
# ✅ Streamlit UI

st.markdown("## AI Chatbot for Railway Accident Analysis")

st.write("Ask any descriptive details about Indian railway accidents and safety measures!")
```

Chat input

```
query = st.text_input("Enter your query:", "")
```

```
if query:
```

```
    ai_response = chat_with_ai(query)  
  
    st.write("### AI Response:")  
  
    st.write(ai_response)
```

- **Streamlit UI:** This section creates the user interface for the chatbot using Streamlit.
 - st.markdown(): Displays a heading for the chatbot.
 - st.write(): Displays a descriptive message to the user.
 - st.text_input(): Creates a text input field where the user can type their query. The value entered by the user is stored in the query variable.
 - Conditional Response: If the user has entered a query (i.e., if query:), the code does the following:
 - ai_response = chat_with_ai(query): Calls the chat_with_ai() function with the user's query to get the model's response.
 - st.write("### AI Response:"): Displays a heading for the AI's response.
 - st.write(ai_response): Displays the AI's response in the Streamlit app.

In summary, here's how the chatbot works end-to-end:

1. **Initialization:** The show() function is called. It loads the necessary libraries and sets up the Groq API credentials.
2. **Data Loading:** The railway accident data is loaded from the CSV file and cached using st.cache_data.
3. **User Input:** The user types a query into the text input field.
4. **AI Interaction:**
 - When the user submits the query, the chat_with_ai() function is called.
 - The user's query is combined with a data_summary to provide context to the model.
 - The enhanced_prompt is sent to the Groq API.
 - The Groq API returns a response generated by the LLaMA model.
5. **Display Response:** The chatbot displays the model's response in the Streamlit app.

The modular structure of the code (with separate functions for data loading, querying, and AI interaction) makes it relatively easy to understand and maintain. The use of Streamlit makes it easy to create an interactive user interface with minimal code.

Predictive model page

Complete Code Dissection: Railway Accident Severity Predictive Model

1. Imports and Constants

python

```
import streamlit as st
import pandas as pd
import numpy as np
import math
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.metrics import mean_absolute_error, r2_score
```

FEATURES = ['Standard Accident Type', 'Deaths', 'Injuries', 'Rescue Time (hrs)']

TARGET_SEVERITY = 'Severity'

CRITICAL_THRESHOLD = 75

MID_LEVEL_THRESHOLD = 50

LOW_LEVEL_THRESHOLD = 25

Explanation:

- **Libraries:**

- streamlit for UI
- pandas and numpy for data handling
- math for mathematical operations
- sklearn modules for model building, preprocessing, and evaluation
- **FEATURES:** The input features used for prediction.
- **TARGET_SEVERITY:** The target variable to predict (severity score).
- **Thresholds:** Define severity levels for classification based on predicted severity.

2. Data Loading Function

python

```
@st.cache_data

def load_data(uploaded_file):

    try:
        df = pd.read_csv(uploaded_file)
        st.session_state.original_df = df.copy()
    return df

except Exception as e:
    st.error(f"Error loading data: {e}")
return None
```

Explanation:

- Loads CSV data uploaded by the user.
- Caches the data to avoid reloading unnecessarily.
- Stores a copy of the original dataframe in `st.session_state` for later use.
- Handles errors gracefully.

3. Data Preprocessing

python

```
def preprocess_data(df):
    df = df.copy()
```

```
if TARGET_SEVERITY not in df.columns:  
    df[TARGET_SEVERITY] = df['Deaths'] + df['Injuries']  
  
categorical_features = ['Standard Accident Type']  
numerical_features = ['Deaths', 'Injuries', 'Rescue Time (hrs)']  
  
numeric_transformer = Pipeline([  
    ('imputer', SimpleImputer(strategy='mean')),  
    ('scaler', StandardScaler())  
])  
  
categorical_transformer = Pipeline([  
    ('imputer', SimpleImputer(strategy='most_frequent')),  
    ('onehot', OneHotEncoder(handle_unknown='ignore'))  
])  
  
preprocessor = ColumnTransformer([  
    ('num', numeric_transformer, numerical_features),  
    ('cat', categorical_transformer, categorical_features)  
])  
  
X = df[FEATURES]  
X_processed = preprocessor.fit_transform(X)  
  
st.session_state.preprocessor = preprocessor  
st.session_state.feature_names = preprocessor.get_feature_names_out()
```

```
return pd.DataFrame(X_processed, columns=st.session_state.feature_names),  
df[TARGET_SEVERITY]
```

Explanation:

- **Severity Calculation:** If the severity column is missing, it creates it as sum of deaths and injuries.
- **Feature Types:** Defines categorical and numerical features.
- **Preprocessing Pipelines:**
 - Numerical: Imputes missing values with mean and scales features.
 - Categorical: Imputes missing values with mode and one-hot encodes.
- **ColumnTransformer:** Applies appropriate transformations to each feature type.
- **Fitting and transforming:** Fits on the dataset and transforms features.
- Stores the preprocessor and feature names in session state for reuse during prediction.
- Returns processed features and target severity.

4. Model Training

python

```
def train_severity_model(X, y):  
  
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)  
  
  
    model = RandomForestRegressor(random_state=42)  
    model.fit(X_train, y_train)  
  
  
    y_pred = model.predict(X_test)  
    mae = mean_absolute_error(y_test, y_pred)  
    r2 = r2_score(y_test, y_pred)  
  
  
    st.session_state.severity_model = model  
    st.session_state.severity_mae = mae  
    st.session_state.severity_r2 = r2
```

Explanation:

- Splits data into training and testing sets.
- Trains a Random Forest regressor on training data.
- Predicts on test data.
- Calculates MAE and R² to evaluate model performance.
- Stores the trained model and metrics in session state.

5. Severity Classification

python

```
def classify_severity(score):

    if score >= CRITICAL_THRESHOLD:
        return "Critical"

    elif score >= MID_LEVEL_THRESHOLD:
        return "Mid-Level"

    elif score >= LOW_LEVEL_THRESHOLD:
        return "Low-Level"

    else:
        return "Very Low"
```

Explanation:

- Converts continuous severity score into categorical levels using predefined thresholds.
- Helps in actionable risk categorization.

6. Ambulance Estimation

python

```
def estimate_ambulances(deaths, injuries, rescue_time):

    ambulances = math.ceil((deaths * 0.3) + (injuries * 0.5) + (rescue_time * 0.8))

    if ambulances == 0 and (deaths > 0 or injuries > 0 or rescue_time > 0):
        ambulances = 1

    return ambulances
```

Explanation:

- Uses weighted formula to estimate number of ambulances needed.
- Ensures minimum 1 ambulance if any casualty or rescue time exists.

7. Structural Damage Cost Estimation

python

```
def estimate_structural_damage_cost(accident_type, deaths, injuries):  
    base_costs = {  
        'Bombing': 100_000_000,  
        'Fire': 70_000_000,  
        'Derailment': 50_000_000,  
        'Collision': 30_000_000,  
        'Other': 20_000_000  
    }  
  
    base_cost = base_costs.get(accident_type, base_costs['Other'])  
    severity_score = deaths + injuries  
  
  
    if severity_score >= 100:  
        multiplier = 2.0  
    elif 50 <= severity_score < 100:  
        multiplier = 1.5  
    elif 25 <= severity_score < 50:  
        multiplier = 1.2  
    else:  
        multiplier = 1.0  
  
  
    return base_cost * multiplier
```

Explanation:

- Assigns a base cost depending on accident type.

- Applies a multiplier based on severity score.
- Outputs estimated infrastructure damage cost.

8. Main Streamlit UI (show function)

python

```
def show():
    st.title("Railway Accident Severity Analytics")

    uploaded_file = st.file_uploader("Upload your accident data CSV file", type=["csv"])

    if uploaded_file is not None:
        df = load_data(uploaded_file)

        if df is not None:
            st.subheader("Raw Data Preview")
            st.dataframe(df.head())

            X_processed, y = preprocess_data(df)

            st.subheader("Preprocessed Feature Sample")
            st.dataframe(X_processed.head())

        if st.button("Train Severity Prediction Model"):
            train_severity_model(X_processed, y)
            st.success(f"Severity Model Trained! MAE: {st.session_state.severity_mae:.2f}, R2: {st.session_state.severity_r2:.2f}")

    if ('severity_model' in st.session_state and
        'preprocessor' in st.session_state and
        'feature_names' in st.session_state and
```

```
'original_df' in st.session_state):  
  
    st.markdown("---")  
    st.header("Make Predictions")  
  
    accident_types = list(st.session_state.original_df['Standard Accident Type'].unique())  
    col1, col2, col3, col4 = st.columns(4)  
  
with col1:  
    accident_type = st.selectbox("Accident Type", accident_types)  
with col2:  
    deaths = st.number_input("Number of Deaths", min_value=0, step=1)  
with col3:  
    injuries = st.number_input("Number of Injuries", min_value=0, step=1)  
with col4:  
    rescue_time = st.number_input("Rescue Time (hrs)", min_value=0, step=1)  
  
if st.button("Predict Accident Severity and Outcomes"):  
    input_df = pd.DataFrame({  
        'Standard Accident Type': [accident_type],  
        'Deaths': [deaths],  
        'Injuries': [injuries],  
        'Rescue Time (hrs)': [rescue_time]  
    })  
  
    X_input = st.session_state.preprocessor.transform(input_df)  
    X_input_df = pd.DataFrame(X_input, columns=st.session_state.feature_names)
```

```

severity_pred = st.session_state.severity_model.predict(X_input_df)[0]
severity_level = classify_severity(severity_pred)

estimated_ambulances = estimate_ambulances(deaths, injuries, rescue_time)
estimated_damage_cost = estimate_structural_damage_cost(accident_type, deaths,
injuries)

st.markdown("## 🚨 Prediction Results")
col1, col2 = st.columns(2)

with col1:
    st.metric("Predicted Severity Score", f"{severity_pred:.2f}")
    st.metric("Severity Level", severity_level)
    st.metric("Estimated Ambulances Required", f"{estimated_ambulances}")

with col2:
    st.metric("Structural Damage Cost (INR)", f"₹{estimated_damage_cost:.0f}")
    st.markdown("### 📊 Model Performance")
    st.info(f"MAE: {st.session_state.severity_mae:.2f} | R2: {st.session_state.severity_r2:.2f}")

# Math explanations with LaTeX
st.markdown("---")
st.markdown("### 📈 Math Behind the Model")

st.markdown("## Severity Score Calculation:")
st.latex(r"""\text{Severity Score (Input)} = \text{Deaths} + \text{Injuries}""")

```

```

st.markdown("**Model Prediction:**")
st.latex(r"""\text{Predicted Severity Score} = f(\text{Accident Type}, \text{Deaths}, \text{Injuries}, \text{Rescue Time (hrs)})""")
  

  

st.markdown("**Severity Level Assignment:**")
st.latex(r"""
\begin{cases}
\text{Critical}, & \text{if } \text{Predicted Severity Score} \geq 75 \\
\text{Mid-Level}, & \text{if } 50 \leq \text{Predicted Severity Score} < 75 \\
\text{Low-Level}, & \text{if } 25 \leq \text{Predicted Severity Score} < 50 \\
\text{Very Low}, & \text{if } \text{Predicted Severity Score} < 25
\end{cases}
""")  

  

st.markdown("**Ambulance Estimation Formula:**")
st.latex(r"""\text{Estimated Ambulances} = \left\lceil (\text{Deaths} \times 0.3) + (\text{Injuries} \times 0.5) + (\text{Rescue Time (hrs)} \times 0.8) \right\rceil""")
  

  

st.markdown("**Structural Damage Cost:**")
st.markdown("")

**Base Cost by Accident Type:**  


- Bombing: ₹1,00,00,000
- Fire: ₹70,00,000
- Derailment: ₹50,00,000
- Collision: ₹30,00,000
- Other: ₹20,00,000

```

""")

st.markdown("")

st.markdown("")

st.markdown("")

st.latex(r"")

else:

st.info("Upload data and train the severity model to enable predictions.")

Summary

- **Data loading and caching** for efficiency.
- **Preprocessing pipelines** handle missing data, scaling, and encoding.
- **Random Forest regression** predicts severity score.
- **Severity classification** translates scores into actionable levels.
- **Rule-based ambulance and damage cost estimation** provide practical resource and financial impact insights.
- **Professional UI** with clear input forms, metrics, and math explanations rendered with LaTeX.
- **Session state** maintains model and preprocessing objects across user interactions.

If you want me to explain any specific part in more detail or help with further enhancements, just ask!

Share

Mathematics Behind Random Forest Algorithm

Certainly! Here's a clear breakdown of **all the mathematical parts** in your finalized predictive model, including the core machine learning algorithm (Random Forest regression), the feature engineering, and the resource estimation formulas:

1. Severity Score Calculation (Feature Engineering)

Before modeling, the **severity score** is calculated as:

$$\text{Severity Score (Input)} = \text{Deaths} + \text{Injuries}$$

This simple sum gives a quick measure of the human impact of an accident.

2. Random Forest Regression: Mathematical Foundation

How Random Forest Works

- Random Forest is an ensemble of decision trees, each trained on a random subset of the data and features.
- For **regression**, each tree predicts a value, and the final prediction is the **average** of all tree predictions¹³⁷.

Mathematical Steps

a. Bootstrap Sampling:

For each tree, generate a random sample (with replacement) from the dataset.

b. Random Feature Selection:

At each split in a tree, randomly select a subset of features (for regression, typically $m=p/3$, where p is the total number of features)¹³.

c. Tree Construction:

Each tree is grown to its maximum depth (no pruning). At each split, the feature and threshold that minimize the **Mean Squared Error (MSE)** are chosen:

$$MSE = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

where y_i is the true value, and \hat{y}_i is the predicted value for sample i .

d. Prediction Aggregation:

For a new input, predict using each tree, then average:

$$\hat{y}_{RF} = \frac{1}{T} \sum_{t=1}^T \hat{y}(t)$$

where T is the number of trees, and $\hat{y}(t)$ is the prediction from tree t ¹³⁷.

3. Severity Level Assignment (Thresholding)

The predicted severity score is mapped to a qualitative level using thresholds:

{Critical,if Predicted Severity Score \geq 75
Mid-Level,if $50 \leq$ Predicted Severity Score <75
Low-Level,if $25 \leq$ Predicted Severity Score <50
Very Low,if Predicted Severity Score <25
} {Critical,
Mid-Level,Low-
Level,Very Low,if Predicted Severity Score \geq 75
if $50 \leq$ Predicted Severity Score <75
if $25 \leq$ Predicted Severity Score <50
if Predicted Severity Score <25

4. Ambulance Estimation Formula

Resource needs are estimated using a weighted sum:

$$\text{Estimated Ambulances} = \lceil (\text{Deaths} \times 0.3) + (\text{Injuries} \times 0.5) + (\text{Rescue Time (hrs)} \times 0.8) \rceil$$
$$\text{Estimated Ambulances} = \lceil (\text{Deaths} \times 0.3) + (\text{Injuries} \times 0.5) + (\text{Rescue Time (hrs)} \times 0.8) \rceil$$

- $\lceil \cdot \rceil$ is the ceiling function (round up to nearest integer).
- Each factor reflects the expected demand for ambulances per casualty and per hour of rescue.

5. Structural Damage Cost Estimation

The cost is estimated as:

$$\text{Estimated Cost} = \text{Base Cost} \times \text{Multiplier}$$
$$\text{Estimated Cost} = \text{Base Cost} \times \text{Multiplier}$$

- **Base Cost** depends on accident type:
 - Bombing: ₹1,00,00,000
 - Fire: ₹70,00,000
 - Derailment: ₹50,00,000
 - Collision: ₹30,00,000
 - Other: ₹20,00,000
- **Multiplier** depends on severity score:
 - ≥ 100 : $\times 2.0$
 - 50–99: $\times 1.5$
 - 25–49: $\times 1.2$
 - < 25 : $\times 1.0$

6. Model Performance Metrics

- **Mean Absolute Error (MAE):**

$$MAE = \frac{1}{N} \sum_{i=1}^N |y^i - \hat{y}_i|$$

- **R-squared (R²):**

$$R^2 = 1 - \frac{\sum_{i=1}^N (y_i - \hat{y}_i)^2}{\sum_{i=1}^N (y_i - \bar{y})^2}$$

where \bar{y} is the mean of the observed data.

7. Summary Table		
Step	Formula/Math	Purpose
Severity Score	Deaths + Injuries	Quantify human impact
Random Forest Output	$\hat{y}_{RF} = \frac{1}{T} \sum_{t=1}^T \hat{y}^{(t)}$	Predict severity using ensemble learning
Severity Level	See threshold formula above	Categorize risk
Ambulance Estimation	$\lceil (Deaths \times 0.3) + (Injuries \times 0.5) + (Rescue\ Time\ (hrs) \times 0.8) \rceil$	Estimate resources
Damage Cost	Base Cost × Multiplier	Estimate financial impact
MAE	$\frac{1}{N} \sum_{i=1}^N \hat{y}_i - y_i $	
R ²	$1 - \frac{\sum (y_i - \hat{y}_i)^2}{\sum (y_i - \bar{y})^2}$	Model fit (variance explained)

✓ Why I Used Random Forest Regression:

I chose **Random Forest Regression** because it provides a powerful and robust approach for predicting continuous outcomes like **accident severity scores**, especially when the data includes both **numerical** and **categorical** features. It handles **non-linear relationships**, reduces **overfitting** through ensemble averaging, and performs well even when the dataset contains noise or missing values. In my case, Random Forest allowed for accurate severity predictions based on diverse accident factors such as **deaths**, **injuries**, **rescue time**, and **accident type**, making it an ideal choice for reliable real-world modeling.

Explanation of OneHotEncoding for the "Standard Accident Type" feature:

In your code, the feature "**Standard Accident Type**" is a **categorical variable**, meaning it contains categories like "Bombing," "Fire," "Derailment," etc. Machine learning algorithms like **Random Forest** require numerical input, so **OneHotEncoding** is applied to convert these categorical values into a numeric format.

❖ How OneHotEncoding Works:

OneHotEncoding creates **binary (0 or 1)** columns for each unique category in the original feature. For example:

- **Original column: Standard Accident Type**
 - Categories: ["Bombing", "Fire", "Derailment", "Collision"]
- **OneHotEncoded columns:**
 - Bombing: 1 or 0
 - Fire: 1 or 0
 - Derailment: 1 or 0
 - Collision: 1 or 0

Example:

If you have an accident of type "Fire", the OneHotEncoding transformation would look like this:

Bombing Fire Derailment Collision

0 1 0 0

- "Fire" is represented as a **1** in the **Fire** column, and all other columns have **0**.
-

💡 Why It's Used in Your Code:

- The **Random Forest model** works with **numerical inputs**, so OneHotEncoding is applied to the **"Standard Accident Type"** to convert it into a numeric format.
- This allows the model to process categorical data while keeping the relationships intact.

Response to the Database Question:

In my project, I have used CSV files as the source for accident data. The data is uploaded by the user in real-time via Streamlit's file uploader, and it's processed and stored temporarily within the session state using Pandas DataFrames. This approach is suitable for the scope of the project, where we analyze and predict accident severity based on uploaded datasets. For more robust, large-scale

applications, a relational database like MySQL or PostgreSQL, or a NoSQL database like MongoDB could be considered.

Why It's Challenging:

To integrate a **real-time cloud database** (like Firebase, AWS DynamoDB, or Google BigQuery), you'd need access to:

- **Live accident data feeds** or
- **Official APIs** from Indian Railways or government transport authorities.

But:

- Most of this data isn't **publicly accessible in real-time**.
 - Getting **permissions** from government departments involves **formal approval**, often for research or enterprise use.
 - There are also **data privacy and security concerns** when working with sensitive or real-world incident data.
-

What You Can Say If Asked:

Real-time cloud integration was considered for this project. However, due to the **lack of open-access live accident data** and the complexity of obtaining **official permissions**, we opted for static CSV-based analysis. This allowed us to simulate real-world scenarios while maintaining the flexibility to deploy the application as a proof-of-concept. Future versions could integrate live APIs if access is granted by relevant authorities.

explain random forest regression ml model in context to my project predictive model code

Answer

Certainly! Here's an explanation of the Random Forest Regression model, within the context of the Predictive_Model.py code you provided.

Random Forest Regression in Predictive_Model.py

1. **Purpose:** The primary goal of the Predictive_Model.py is to predict the severity of railway accidents. The Random Forest Regression model is used to estimate a continuous 'Severity' score based on various input features.

2. Key Components and Functionality:

- **Feature Selection:**
 - FEATURES = ['Standard Accident Type', 'Deaths', 'Injuries', 'Rescue Time (hrs)']
 - These features are used as input to the model.
- **Preprocessing:**
 - OneHotEncoder is used on 'Standard Accident Type'
 - StandardScaler is used on numerical features
- **Model Training:**
 - train_test_split splits the data into training and testing
 - RandomForestRegressor algorithm is used to train the model and predict values
 - Mean absolute error and r2 score are computed as the model metrics
- **Prediction**
 - classify_severity function converts the predicted values into critical, medium, low or very low values.

3. Implementation Details:

- **Data Preprocessing (preprocess_data):**
 - **Severity Calculation:** The code first computes the accident 'Severity' if it's not already present in the dataset, calculating it as the sum of 'Deaths' and 'Injuries'.
 - **Feature Transformation:** It uses a ColumnTransformer to handle different feature types: numerical and categorical.
 - Numerical features ('Deaths', 'Injuries', 'Rescue Time (hrs)') are scaled using StandardScaler after imputing missing values with the mean.
 - Categorical features ('Standard Accident Type') are one-hot encoded using OneHotEncoder after imputing missing values with the most frequent value.
- **Model Training (train_severity_model):**
 - The preprocessed data is split into training and testing sets.

- A RandomForestRegressor is initialized with a random_state for reproducibility.
- The model is trained using the training data.
- Predictions are made on the test set, and performance metrics like Mean Absolute Error (MAE) and R-squared score (R2) are calculated.

- **Performance Metrics:**

- The code calculates the Mean Absolute Error (mae) and R-squared score (r2) to evaluate the model's performance. These metrics are stored in the session state for later use.

4. Random Forest Regression Explanation:

- **Ensemble Learning:** Random Forest is an ensemble learning method, meaning it combines multiple individual models to make a final prediction.
- **Decision Trees:** It consists of a collection of decision trees, each trained on a random subset of the data and features.
- **Randomness:** The randomness in feature and data selection helps to create diverse trees, reducing overfitting and improving generalization.
- **Regression:** For regression tasks, each tree predicts a continuous value, and the final prediction is the average of all tree predictions.

5. Use in Project:

- The trained Random Forest model is used to predict accident severity. These predictions can provide insights to help with rapid resource allocation and improved safety protocols.

6. Code Snippet (Key Parts):

```
python
```

```
# Model training

model = RandomForestRegressor(random_state=42)

model.fit(X_train, y_train)
```

```
# Prediction
```

```
y_pred = model.predict(X_test)
```

```
# Metrics
```

```
mae = mean_absolute_error(y_test, y_pred)  
r2 = r2_score(y_test, y_pred)
```

In summary, the Random Forest Regression model in Predictive_Model.py plays a crucial role in estimating accident severity by leveraging an ensemble of decision trees trained on preprocessed accident data. This enables the system to support decision-making related to resource allocation and safety improvements.

provide a justification why i used random forest regression model only

Justification for Random Forest Regression

1. **Non-linear Relationships:** Railway accident data often involves complex, non-linear relationships between various factors. Random Forest Regression excels at capturing these intricate interactions because it's based on decision trees that can partition the feature space in non-linear ways.
2. **Feature Importance:** Random Forest provides measures of feature importance, allowing you to identify which accident characteristics (e.g., 'Standard Accident Type', 'Deaths', 'Injuries', 'Rescue Time (hrs)') are most influential in determining severity.
3. **Ensemble Learning:** The ensemble nature of Random Forest (multiple decision trees) reduces the risk of overfitting compared to a single decision tree or a linear model. Each tree is trained on a random subset of the data and features, making the model more robust and improving its generalization ability.
4. **Handles Mixed Data Types:** Random Forest can handle both numerical and categorical features without requiring extensive preprocessing (although the code does use OneHotEncoder for the categorical feature). It works effectively with a mix of data types, which can be useful in real-world accident datasets that may contain both continuous and discrete variables.
5. **Robustness to Outliers:** Decision trees are less sensitive to outliers than linear models. Outliers have a limited impact on the tree structure, reducing their influence on the final prediction.
6. **Prediction Accuracy:** Random Forest Regression models are recognized for their high predictive accuracy in diverse domains.
7. **Interpretability:** Although Random Forest is an ensemble method, it provides some level of interpretability through feature importance scores. These scores can provide insights into the factors driving accident severity.

8. **Metrics.** The metrics used to measure Random Forest, provides clear metrics which can be used to show the accuracy of the models like MAE and R-squared metrics

Alternative Models:

- Linear Regression would assume a linear relationship between accident severity and various causal factors, but relationships are rarely linear.
- Neural networks, while powerful, may be overkill for the dataset's size and require extensive hyperparameter tuning and computational resources.

Explanation of Insights_and_Analysis.py

This Streamlit application provides visual and statistical insights into railway accident data using the train_accident_analysis-dataset.csv dataset.

1. **Objective:** The primary goal is to explore and present key patterns and trends in the dataset to help identify risk factors and improve railway safety.
2. **Data Source:**
 - The application uses train_accident_analysis-dataset.csv, which contains information about railway accidents, including details such as the environment, train type, casualties, causes, time, and region.
 - The data is loaded using pd.read_csv().
 - The first few rows of the dataset are:

text

```
X,environment,train_name,injured,killed,triggering_factor,time,railway_division,cause,env  
1,open rail track|over 2000 attackers,Sabarmati Express,43,58,attack by a mob at Godhra  
station in Gujarat and four coaches were set on fire,27-02-2002 08:30,w,attack,open  
2,open rail track,Shramjeevi Express,80,12, crash occurred when sabotage derailed the  
Shramjeevi express at Jaunpur,13-05-2002 03:00,n,attack,open  
3,collision on a manned level crossing,Kanpur Kasganj Express,29,49,Express collided with a  
passenger bus near the town of Kasganj,04-06-2002 13:00,ne,unclear track,busy
```

3. **Key Processes:**

- **show() Function:** This function generates the Streamlit UI.
 - **Data Loading:** Reads the dataset using pd.read_csv().
 - **Dropdown Selection:** Provides a dropdown to select a column for analysis.

- **Descriptive Findings:** Displays the most frequent value and its count for the selected column.
- **Additional Analysis:** Displays total casualties and regions, causes of accidents.

4. Core Functionalities:

- **Data Visualization:** Uses matplotlib.pyplot and seaborn to create visualizations.
 - **Bar Plots:** Displays the distribution of accident counts for different categories (e.g., daytime, season, region, environment, train type, cause).
 - **Casualties by Region:** Displays the total injured and killed by region using a bar plot.

5. Code Snippets:

python

Bar plot of accident counts

```
sns.barplot(x=df[col].value_counts().index, y=df[col].value_counts().values, ax=ax,
palette='viridis')
```

```
ax.set_xlabel(col.replace('_', ' ').title(), fontsize=10)
```

```
ax.set_ylabel("Accident Count", fontsize=10)
```

```
ax.set_title(question, fontsize=12)
```

```
plt.xticks(rotation=45, ha='right', fontsize=8)
```

```
plt.yticks(fontsize=8)
```

```
plt.tight_layout()
```

```
st.pyplot(fig)
```

python

Casualties by region

```
region_casualties = df.groupby('region')[['injured', 'killed']].sum()
```

python

Most frequent causes of accidents

```
cause_counts = df['cause'].value_counts()
```

6. Flowchart Integration:

- **Process & Visualize Data:** The flowchart shows the main action as "Process & Visualize Data."
- This includes loading the data, creating visualizations, and presenting findings.
- The flowchart element, then displays results, showing data distributions and related statistics.

7. Insights and Outputs:

- For each selected column, the application provides interpretations and actionable insights.
- Examples:
 - **Daytime:** Accidents during night and early morning may be linked to reduced visibility.
 - **Season:** Accidents in certain seasons may be due to weather conditions.
 - **Region:** Specific regional challenges may contribute to higher accident rates.
 - **Environment:** Understanding the accident environment can highlight risks such as unmanned crossings.
 - **Train Type:** Accidents involving certain train types may point to maintenance issues.

8. Streamlit UI:

- The Streamlit UI facilitates easy data exploration and interpretation.
- Users can select a column, view visualizations, and read descriptive findings.

9. Purpose & Relation to Project

- The "Insights and Analysis" page provides valuable insights into trends, and regional patterns, extracted from the historical accident data. The visualization aids in identifying the predominant variables that cause accidents.
- These findings assist in creating better and more reliable Predictive models, or improve the Groq AI model by specifying its limitations

Summary:

`Insights_and_Analysis.py` leverages Streamlit, pandas, matplotlib, and seaborn to analyze and visualize railway accident data. By presenting data distributions, total casualties, and top causes of accidents, the application supports informed decision-making for enhancing railway safety protocols. The flowchart accurately represents this process as “Process & Visualize Data”, highlighting the role of data exploration and visual representation.