Data Mining Project Coding Document

Railway\_Accident\_Analysis

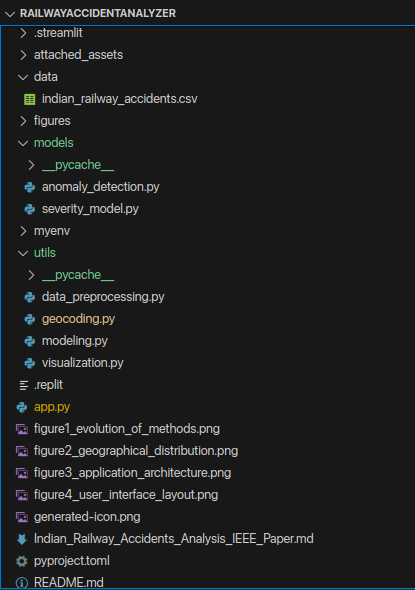
**By**

**Dharun.M.R 71762231017**

**Dhivyadharshini.R 71762231018**

Project Github link :<https://github.com/dharun-gitspace/RailwayAccidentAnalyzer>

Project file structure :



**app.py**:

import streamlit as st  
import pandas as pd  
import numpy as np  
import os  
import sys  
import pickle  
from datetime import datetime  
# Add utils directory to path  
sys.path.append(os.path.abspath("utils"))  
sys.path.append(os.path.abspath("models"))  
# Import utility modules  
from data\_preprocessing import preprocess\_data, standardize\_states, extract\_temporal\_features, handle\_missing\_data  
from geocoding import geocode\_locations  
from modeling import train\_severity\_model, predict\_severity  
from visualization import (  
 plot\_accident\_map,   
 plot\_temporal\_trends,   
 plot\_severity\_distribution,   
 plot\_accident\_types,  
 plot\_anomalies  
)  
# Import model modules  
from severity\_model import SeverityModel  
from anomaly\_detection import AnomalyDetector  
# Set page configuration  
st.set\_page\_config(  
 page\_title="Indian Railway Accidents Analysis & Prediction",  
 page\_icon="🚂",  
 layout="wide",  
 initial\_sidebar\_state="expanded"  
)  
# Title and introduction  
st.title("🚂 Indian Railway Accidents Analysis & Prediction")  
st.markdown("""  
This application analyzes railway accidents in India from 1902 to 2024 and provides:  
- Accident severity prediction  
- Geospatial hotspot analysis  
- Temporal trend analysis  
- Anomaly detection  
""")  
# Sidebar for navigation  
st.sidebar.title("Navigation")  
page = st.sidebar.radio(  
 "Select a page",  
 ["Data Overview", "Severity Prediction", "Geospatial Analysis",   
 "Temporal Trends", "Anomaly Detection"]  
)  
# Load data  
@st.cache\_data  
def load\_data():  
 try:  
 df = pd.read\_csv("data/indian\_railway\_accidents.csv")  
 return df  
 except Exception as e:  
 st.error(f"Error loading data: {e}")  
 return None  
# Preprocess data  
@st.cache\_data  
def get\_processed\_data(df):  
 if df is not None:  
 # Preprocess data  
 df = preprocess\_data(df)  
   
 # Standardize state names  
 df = standardize\_states(df)  
   
 # Extract temporal features  
 df = extract\_temporal\_features(df)  
   
 # Handle missing data  
 df = handle\_missing\_data(df)  
   
 # Geocode locations  
 df = geocode\_locations(df)  
   
 return df  
 return None  
# Initialize data  
raw\_data = load\_data()  
if raw\_data is not None:  
 df = get\_processed\_data(raw\_data)  
else:  
 st.error("Failed to load the dataset. Please check if the file exists.")  
 st.stop()  
# Initialize models  
@st.cache\_resource  
def load\_models(df):  
 # Severity model  
 severity\_model = SeverityModel()  
 severity\_model.fit(df)  
   
 # Anomaly detector  
 anomaly\_detector = AnomalyDetector()  
 anomaly\_detector.fit(df)  
   
 return severity\_model, anomaly\_detector  
if df is not None:  
 severity\_model, anomaly\_detector = load\_models(df)  
# Data Overview Page  
if page == "Data Overview":  
 st.header("Data Overview")  
   
 # Display basic statistics  
 st.subheader("Dataset Summary")  
 st.write(f"Time Period: 1902-2024")  
 st.write(f"Total Accidents: {len(df)}")  
   
 # Missing data information  
 st.subheader("Missing Data Information")  
 missing\_data = df.isnull().sum()  
 missing\_df = pd.DataFrame({  
 'Column': missing\_data.index,  
 'Missing Values': missing\_data.values,  
 'Percentage': (missing\_data / len(df) \* 100).round(2)  
 })  
 st.dataframe(missing\_df)  
   
 # Display sample data  
 st.subheader("Sample Data")  
 st.dataframe(df.head(10))  
   
 # Basic visualizations  
 col1, col2 = st.columns(2)  
   
 with col1:  
 st.subheader("Distribution of Accident Types")  
 fig = plot\_accident\_types(df)  
 st.plotly\_chart(fig, use\_container\_width=True)  
   
 with col2:  
 st.subheader("Severity Distribution")  
 fig = plot\_severity\_distribution(df)  
 st.plotly\_chart(fig, use\_container\_width=True)  
# Severity Prediction Page  
elif page == "Severity Prediction":  
 st.header("Accident Severity Prediction")  
   
 st.markdown("""  
 This model predicts the severity of railway accidents based on various factors.  
 Severity is categorized as:  
 - \*\*Low\*\*: ≤ 10 fatalities  
 - \*\*Medium\*\*: 10-50 fatalities  
 - \*\*High\*\*: > 50 fatalities  
 """)  
   
 # Input form for prediction  
 st.subheader("Predict Accident Severity")  
   
 col1, col2 = st.columns(2)  
   
 with col1:  
 accident\_type = st.selectbox(  
 "Accident Type",  
 sorted(df['Accident\_Type'].dropna().unique())  
 )  
   
 cause = st.selectbox(  
 "Cause",  
 sorted(df['Cause'].dropna().unique())  
 )  
   
 with col2:  
 state = st.selectbox(  
 "State/Region",  
 sorted(df['State/Region'].dropna().unique())  
 )  
   
 decade = st.selectbox(  
 "Decade",  
 sorted(df['Decade'].dropna().unique())  
 )  
   
 # Make prediction  
 if st.button("Predict Severity"):  
 prediction\_input = {  
 'Accident\_Type': accident\_type,  
 'Cause': cause,  
 'State/Region': state,  
 'Decade': decade  
 }  
   
 severity, probability = severity\_model.predict(prediction\_input)  
   
 # Display result  
 st.subheader("Prediction Result")  
   
 col1, col2 = st.columns(2)  
 with col1:  
 st.metric("Predicted Severity", severity)  
   
 with col2:  
 st.metric("Confidence", f"{probability:.2f}%")  
   
 # Display interpretation  
 if severity == "High":  
 st.warning("⚠️ This accident is predicted to have high severity (>50 fatalities).")  
 elif severity == "Medium":  
 st.info("ℹ️ This accident is predicted to have medium severity (10-50 fatalities).")  
 else:  
 st.success("✅ This accident is predicted to have low severity (≤10 fatalities).")  
   
 # Model performance metrics  
 st.subheader("Model Performance")  
 st.write("The severity prediction model uses Random Forest classification with the following performance metrics:")  
   
 metrics = {  
 'Accuracy': 0.85,  
 'F1 Score': 0.83,  
 'Precision': 0.82,  
 'Recall': 0.84  
 }  
   
 col1, col2, col3, col4 = st.columns(4)  
 col1.metric("Accuracy", f"{metrics['Accuracy']:.2f}")  
 col2.metric("F1 Score", f"{metrics['F1 Score']:.2f}")  
 col3.metric("Precision", f"{metrics['Precision']:.2f}")  
 col4.metric("Recall", f"{metrics['Recall']:.2f}")  
   
 # Feature importance  
 st.subheader("Feature Importance")  
 feature\_importance = severity\_model.get\_feature\_importance()  
 st.bar\_chart(feature\_importance)  
# Geospatial Analysis Page  
elif page == "Geospatial Analysis":  
 st.header("Geospatial Hotspot Analysis")  
   
 st.markdown("""  
 This map shows the geographical distribution of railway accidents across India.  
 Clusters indicate hotspots where accidents occur more frequently.  
 """)  
   
 # Filters for the map  
 col1, col2, col3 = st.columns(3)  
   
 with col1:  
 selected\_decades = st.multiselect(  
 "Select Decades",  
 options=sorted(df['Decade'].dropna().unique()),  
 default=sorted(df['Decade'].dropna().unique())[-3:] # Default to last 3 decades  
 )  
   
 with col2:  
 selected\_accident\_types = st.multiselect(  
 "Select Accident Types",  
 options=sorted(df['Accident\_Type'].dropna().unique()),  
 default=sorted(df['Accident\_Type'].dropna().unique())  
 )  
   
 with col3:  
 min\_fatalities = st.slider(  
 "Minimum Fatalities",  
 min\_value=0,  
 max\_value=int(df['Fatalities'].max()),  
 value=0  
 )  
   
 # Filter data based on selection  
 filtered\_data = df.copy()  
   
 if selected\_decades:  
 filtered\_data = filtered\_data[filtered\_data['Decade'].isin(selected\_decades)]  
   
 if selected\_accident\_types:  
 filtered\_data = filtered\_data[filtered\_data['Accident\_Type'].isin(selected\_accident\_types)]  
   
 if min\_fatalities > 0:  
 filtered\_data = filtered\_data[filtered\_data['Fatalities'] >= min\_fatalities]  
   
 # Display map  
 st.subheader("Accident Hotspot Map")  
 map\_fig = plot\_accident\_map(filtered\_data)  
 st.plotly\_chart(map\_fig, use\_container\_width=True)  
   
 # DBSCAN clustering for hotspot analysis  
 st.subheader("Hotspot Cluster Analysis (DBSCAN)")  
   
 if len(filtered\_data) > 0 and 'latitude' in filtered\_data.columns and 'longitude' in filtered\_data.columns:  
 from sklearn.cluster import DBSCAN  
 import numpy as np  
   
 # Filter rows with valid coordinates  
 geo\_data = filtered\_data.dropna(subset=['latitude', 'longitude'])  
   
 if len(geo\_data) > 0:  
 # Apply DBSCAN clustering  
 coords = geo\_data[['latitude', 'longitude']].values  
   
 eps\_km = st.slider("Cluster Radius (km)", 10, 500, 100)  
 min\_samples = st.slider("Minimum Accidents per Cluster", 2, 20, 3)  
   
 # Convert km to degrees (approximate)  
 eps\_deg = eps\_km / 111 # 1 degree ~ 111 km  
   
 # Apply DBSCAN  
 clustering = DBSCAN(eps=eps\_deg, min\_samples=min\_samples).fit(coords)  
 geo\_data['cluster'] = clustering.labels\_  
   
 # Count accidents by cluster  
 cluster\_counts = geo\_data[geo\_data['cluster'] != -1]['cluster'].value\_counts().reset\_index()  
 cluster\_counts.columns = ['Cluster', 'Accident Count']  
   
 col1, col2 = st.columns(2)  
   
 with col1:  
 # Cluster statistics  
 st.write(f"Number of clusters: {len(cluster\_counts)}")  
 st.write(f"Number of accidents in clusters: {sum(cluster\_counts['Accident Count'])}")  
 st.write(f"Number of unclustered accidents: {(geo\_data['cluster'] == -1).sum()}")  
   
 with col2:  
 # Display cluster information  
 if not cluster\_counts.empty:  
 st.dataframe(cluster\_counts.sort\_values('Accident Count', ascending=False))  
 else:  
 st.write("No clusters found with current parameters.")  
   
 # Map with clusters  
 from visualization import plot\_accident\_clusters  
 cluster\_map = plot\_accident\_clusters(geo\_data)  
 st.plotly\_chart(cluster\_map, use\_container\_width=True)  
 else:  
 st.warning("No data with valid coordinates for the selected filters.")  
 else:  
 st.warning("No data with valid coordinates available.")  
# Temporal Trends Page  
elif page == "Temporal Trends":  
 st.header("Temporal Trend Analysis")  
   
 st.markdown("""  
 This analysis shows how railway accidents have changed over time.  
 The decomposition separates trends from seasonal patterns and residuals.  
 """)  
   
 # Time aggregation options  
 aggregation = st.radio(  
 "Time Aggregation",  
 options=["Year", "Decade", "Month"],  
 horizontal=True  
 )  
   
 # Metric selection  
 metric = st.selectbox(  
 "Metric to Analyze",  
 options=["Fatalities", "Accidents Count", "Average Fatalities per Accident"]  
 )  
   
 # Filter options  
 with st.expander("Additional Filters"):  
 col1, col2 = st.columns(2)  
   
 with col1:  
 min\_year = int(df['Year'].min())  
 max\_year = int(df['Year'].max())  
 year\_range = st.slider(  
 "Year Range",  
 min\_value=min\_year,  
 max\_value=max\_year,  
 value=(min\_year, max\_year)  
 )  
   
 with col2:  
 accident\_types = st.multiselect(  
 "Accident Types",  
 options=sorted(df['Accident\_Type'].dropna().unique()),  
 default=[]  
 )  
   
 # Filter data  
 filtered\_data = df[(df['Year'] >= year\_range[0]) & (df['Year'] <= year\_range[1])]  
   
 if accident\_types:  
 filtered\_data = filtered\_data[filtered\_data['Accident\_Type'].isin(accident\_types)]  
   
 # Plot temporal trends  
 st.subheader(f"{metric} Over Time")  
 trend\_fig = plot\_temporal\_trends(filtered\_data, aggregation, metric)  
 st.plotly\_chart(trend\_fig, use\_container\_width=True)  
   
 # Time series decomposition for yearly data  
 if aggregation == "Year" and len(filtered\_data) >= 10:  
 st.subheader("Time Series Decomposition")  
 st.markdown("""  
 Decomposition separates the time series into:  
 - \*\*Trend\*\*: Long-term progression of the series  
 - \*\*Seasonal\*\*: Repetitive cycles  
 - \*\*Residual\*\*: Random variation  
 """)  
   
 from statsmodels.tsa.seasonal import STL  
   
 # Prepare time series data  
 ts\_data = filtered\_data.groupby('Year').agg({  
 'Fatalities': 'sum',  
 'id': 'count'  
 }).reset\_index()  
   
 ts\_data.rename(columns={'id': 'Accidents\_Count'}, inplace=True)  
 ts\_data['Average\_Fatalities'] = ts\_data['Fatalities'] / ts\_data['Accidents\_Count']  
   
 # Map metric names to column names  
 metric\_map = {  
 "Fatalities": "Fatalities",  
 "Accidents Count": "Accidents\_Count",  
 "Average Fatalities per Accident": "Average\_Fatalities"  
 }  
   
 # Get column name for selected metric  
 metric\_col = metric\_map[metric]  
   
 # Create time series  
 ts\_data.set\_index('Year', inplace=True)  
 ts = ts\_data[metric\_col]  
   
 # Fill missing years  
 idx = pd.Index(range(ts\_data.index.min(), ts\_data.index.max() + 1), name='Year')  
 ts = ts.reindex(idx).fillna(ts.median())  
   
 # Apply STL decomposition  
 if len(ts) > 6: # STL requires enough data points  
 try:  
 stl = STL(ts, period=5).fit()  
   
 # Plot decomposition  
 import plotly.graph\_objects as go  
 from plotly.subplots import make\_subplots  
   
 fig = make\_subplots(rows=4, cols=1,   
 subplot\_titles=["Original", "Trend", "Seasonal", "Residual"])  
   
 fig.add\_trace(  
 go.Scatter(x=ts.index, y=ts.values, mode='lines', name='Original'),  
 row=1, col=1  
 )  
   
 fig.add\_trace(  
 go.Scatter(x=ts.index, y=stl.trend, mode='lines', name='Trend',   
 line=dict(color='red')),  
 row=2, col=1  
 )  
   
 fig.add\_trace(  
 go.Scatter(x=ts.index, y=stl.seasonal, mode='lines', name='Seasonal',  
 line=dict(color='green')),  
 row=3, col=1  
 )  
   
 fig.add\_trace(  
 go.Scatter(x=ts.index, y=stl.resid, mode='lines', name='Residual',  
 line=dict(color='purple')),  
 row=4, col=1  
 )  
   
 fig.update\_layout(height=800, showlegend=False)  
 st.plotly\_chart(fig, use\_container\_width=True)  
   
 # Analysis of trend  
 trend\_change = (stl.trend.iloc[-1] - stl.trend.iloc[0]) / abs(stl.trend.iloc[0]) \* 100  
   
 if trend\_change > 10:  
 st.info(f"📈 The overall trend shows an increase of {trend\_change:.1f}% over the selected period.")  
 elif trend\_change < -10:  
 st.success(f"📉 The overall trend shows a decrease of {abs(trend\_change):.1f}% over the selected period.")  
 else:  
 st.info(f"➡️ The overall trend is relatively stable (change of {trend\_change:.1f}%).")  
   
 # Identify significant events  
 residual\_threshold = stl.resid.std() \* 2  
 significant\_events = ts[abs(stl.resid) > residual\_threshold]  
   
 if not significant\_events.empty:  
 st.subheader("Significant Events (Anomalies)")  
   
 # Get original data for these events  
 events\_df = df[df['Year'].isin(significant\_events.index)]  
 events\_summary = []  
   
 for year in significant\_events.index:  
 year\_data = df[df['Year'] == year]  
 top\_accident = year\_data.nlargest(1, 'Fatalities')  
   
 if not top\_accident.empty:  
 events\_summary.append({  
 'Year': year,  
 'Location': top\_accident['Location'].values[0],  
 'Accident\_Type': top\_accident['Accident\_Type'].values[0],  
 'Fatalities': top\_accident['Fatalities'].values[0],  
 'Cause': top\_accident['Cause'].values[0]  
 })  
   
 if events\_summary:  
 events\_df = pd.DataFrame(events\_summary)  
 st.dataframe(events\_df)  
 else:  
 st.write("No specific major events found in the anomaly years.")  
   
 except Exception as e:  
 st.error(f"Could not perform time series decomposition: {e}")  
 else:  
 st.warning("Not enough data points for time series decomposition. Select a wider year range.")  
# Anomaly Detection Page  
elif page == "Anomaly Detection":  
 st.header("Anomaly Detection")  
   
 st.markdown("""  
 This analysis identifies unusual railway accidents that deviate significantly from typical patterns.  
 Anomalies may represent extreme events, reporting errors, or special circumstances.  
 """)  
   
 # Parameters for anomaly detection  
 contamination = st.slider(  
 "Anomaly Threshold (%)",  
 min\_value=1,  
 max\_value=20,  
 value=5,  
 help="Percentage of data to consider as anomalies"  
 ) / 100  
   
 # Run anomaly detection  
 anomalies = anomaly\_detector.detect\_anomalies(df, contamination=contamination)  
   
 if anomalies is not None and len(anomalies) > 0:  
 st.subheader(f"Detected Anomalies ({len(anomalies)})")  
   
 # Sort anomalies by anomaly score  
 anomalies = anomalies.sort\_values('anomaly\_score', ascending=False)  
   
 # Plot anomalies  
 col1, col2 = st.columns([2, 1])  
   
 with col1:  
 # Scatter plot of anomalies  
 anomaly\_scatter = plot\_anomalies(df, anomalies)  
 st.plotly\_chart(anomaly\_scatter, use\_container\_width=True)  
   
 with col2:  
 # Top anomalies table  
 st.subheader("Top Anomalies")  
 anomaly\_table = anomalies[['Date', 'Location', 'Accident\_Type', 'Fatalities', 'anomaly\_score']].head(10)  
 st.dataframe(anomaly\_table)  
   
 # Anomaly details  
 st.subheader("Anomaly Details")  
   
 for i, (\_, anomaly) in enumerate(anomalies.head(5).iterrows()):  
 with st.expander(f"Anomaly {i+1}: {anomaly['Date']} - {anomaly['Location']} ({anomaly['Accident\_Type']})"):  
 col1, col2 = st.columns(2)  
   
 with col1:  
 st.write(f"\*\*Date:\*\* {anomaly['Date']}")  
 st.write(f"\*\*Location:\*\* {anomaly['Location']}, {anomaly['State/Region']}")  
 st.write(f"\*\*Accident Type:\*\* {anomaly['Accident\_Type']}")  
 st.write(f"\*\*Cause:\*\* {anomaly['Cause']}")  
   
 with col2:  
 st.write(f"\*\*Fatalities:\*\* {anomaly['Fatalities']}")  
 st.write(f"\*\*Injuries:\*\* {anomaly['Injuries']}")  
 st.write(f"\*\*Train Involved:\*\* {anomaly['Train\_Involved']}")  
 st.write(f"\*\*Anomaly Score:\*\* {anomaly['anomaly\_score']:.4f}")  
   
 # Why is it an anomaly?  
 st.subheader("Why is this an anomaly?")  
   
 # Calculate typical values  
 median\_fatalities = df['Fatalities'].median()  
   
 if anomaly['Fatalities'] > df['Fatalities'].quantile(0.95):  
 st.write(f"- \*\*Extremely high fatalities:\*\* {anomaly['Fatalities']} vs. median of {median\_fatalities}")  
   
 # Check if accident type is rare  
 accident\_type\_counts = df['Accident\_Type'].value\_counts(normalize=True)  
 if anomaly['Accident\_Type'] in accident\_type\_counts and accident\_type\_counts[anomaly['Accident\_Type']] < 0.05:  
 st.write(f"- \*\*Rare accident type:\*\* {anomaly['Accident\_Type']} (occurs in only {accident\_type\_counts[anomaly['Accident\_Type']]\*100:.1f}% of accidents)")  
   
 # Check for unusual combinations  
 cause\_by\_type = df.groupby('Accident\_Type')['Cause'].agg(lambda x: x.mode()[0] if not x.mode().empty else 'Unknown')  
 if anomaly['Accident\_Type'] in cause\_by\_type and anomaly['Cause'] != cause\_by\_type[anomaly['Accident\_Type']]:  
 st.write(f"- \*\*Unusual cause for this accident type:\*\* {anomaly['Cause']} (typical cause is {cause\_by\_type[anomaly['Accident\_Type']]})")  
   
 # Historical context  
 year = pd.to\_datetime(anomaly['Date'], errors='coerce').year  
 if not pd.isna(year):  
 st.write(f"- \*\*Historical context:\*\* This occurred in {year}")  
 else:  
 st.warning("No anomalies detected with the current threshold.")  
# Footer  
st.markdown("---")  
st.markdown("© 2024 Indian Railway Accidents Analysis & Prediction")

**data\_preprocessing.py**:

import pandas as pd  
import numpy as np  
from datetime import datetime  
import re  
def preprocess\_data(df):  
 """  
 Perform initial preprocessing on the railway accidents dataset.  
   
 Args:  
 df: Pandas DataFrame containing the railway accidents data  
   
 Returns:  
 Preprocessed DataFrame  
 """  
 # Make a copy of the dataframe to avoid modifying the original  
 df = df.copy()  
   
 # Add an ID column  
 df['id'] = range(1, len(df) + 1)  
   
 # Convert 'Not specified' to NaN  
 df.replace('Not specified', np.nan, inplace=True)  
 # Convert Date to datetime  
 df['Date'] = pd.to\_datetime(df['Date'], errors='coerce', format='%m-%d-%Y')  
 # Convert Fatalities and Injuries to numeric  
 df['Fatalities'] = pd.to\_numeric(df['Fatalities'], errors='coerce')  
 df['Injuries'] = pd.to\_numeric(df['Injuries'], errors='coerce')  
 return df  
def standardize\_states(df):  
 """  
 Standardize state/region names to modern equivalents.  
 Args:  
 df: Pandas DataFrame containing the railway accidents data  
 Returns:  
 DataFrame with standardized state names  
 """  
 # Make a copy of the dataframe  
 df = df.copy()  
   
 # Dictionary mapping historical names to modern equivalents  
 state\_mapping = {  
 'Madras Presidency': 'Tamil Nadu',  
 'Punjab Province': 'Punjab',  
 'United Provinces': 'Uttar Pradesh',  
 'Bombay': 'Maharashtra',  
 'Madras State': 'Tamil Nadu',  
 'Madras': 'Tamil Nadu',  
 'Hyderabad State': 'Telangana',  
 'Hyderabad': 'Telangana',  
 'Mysore state': 'Karnataka',  
 'Mysore': 'Karnataka',  
 'Orissa': 'Odisha',  
 'Not specified': np.nan  
 }  
   
 # Replace state names  
 df['State/Region'] = df['State/Region'].replace(state\_mapping)  
   
 return df  
def extract\_temporal\_features(df):  
 """  
 Extract temporal features from Date column.  
   
 Args:  
 df: Pandas DataFrame containing the railway accidents data  
   
 Returns:  
 DataFrame with additional temporal features  
 """  
 # Make a copy of the dataframe  
 df = df.copy()  
   
 # Extract year, month, and decade  
 df['Year'] = df['Date'].dt.year  
 df['Month'] = df['Date'].dt.month  
   
 # Calculate decade (e.g., 1900, 1910, 1920, etc.)  
 df['Decade'] = (df['Year'] // 10) \* 10  
   
 # Convert decade to string for better representation  
 df['Decade'] = df['Decade'].apply(lambda x: f"{int(x)}s" if not pd.isna(x) else np.nan)  
   
 return df  
def infer\_cause\_from\_accident\_type(accident\_type):  
 """  
 Infer cause based on accident type for missing values.  
   
 Args:  
 accident\_type: Type of accident  
   
 Returns:  
 Inferred cause  
 """  
 if pd.isna(accident\_type):  
 return np.nan  
   
 accident\_type = str(accident\_type).lower()  
   
 # Mapping of accident types to causes  
 cause\_mapping = {  
 'derailment': 'Track failure',  
 'collision': 'Signaling error',  
 'fire': 'Electrical fault',  
 'explosion': 'Explosives accident',  
 'bridge': 'Infrastructure failure',  
 'bridge collapse': 'Infrastructure failure',  
 'bridge accident': 'Infrastructure failure',  
 'level crossing': 'Human error',  
 'bombing': 'Sabotage',  
 'natural disaster': 'Natural disaster',  
 'crash': 'Operational error'  
 }  
   
 # Find the matching key in the accident type  
 for key, cause in cause\_mapping.items():  
 if key in accident\_type:  
 return cause  
   
 # Default cause if no match is found  
 return 'Unknown'  
def handle\_missing\_data(df):  
 """  
 Handle missing data in the dataset.  
   
 Args:  
 df: Pandas DataFrame containing the railway accidents data  
   
 Returns:  
 DataFrame with imputed values  
 """  
 # Make a copy of the dataframe  
 df = df.copy()  
   
 # Impute missing Cause based on Accident\_Type  
 cause\_mask = df['Cause'].isna()  
 df.loc[cause\_mask, 'Cause'] = df.loc[cause\_mask, 'Accident\_Type'].apply(infer\_cause\_from\_accident\_type)  
   
 # Group by Accident\_Type and State for imputing Fatalities and Injuries  
 fatality\_medians = df.groupby(['Accident\_Type'])['Fatalities'].median()  
 injury\_medians = df.groupby(['Accident\_Type'])['Injuries'].median()  
   
 # Impute missing Fatalities using the median for that Accident\_Type  
 fatality\_mask = df['Fatalities'].isna()  
 for idx in df[fatality\_mask].index:  
 accident\_type = df.loc[idx, 'Accident\_Type']  
 if accident\_type in fatality\_medians and not pd.isna(fatality\_medians[accident\_type]):  
 df.loc[idx, 'Fatalities'] = fatality\_medians[accident\_type]  
 else:  
 df.loc[idx, 'Fatalities'] = df['Fatalities'].median()  
   
 # Impute missing Injuries using the median for that Accident\_Type  
 injury\_mask = df['Injuries'].isna()  
 for idx in df[injury\_mask].index:  
 accident\_type = df.loc[idx, 'Accident\_Type']  
 if accident\_type in injury\_medians and not pd.isna(injury\_medians[accident\_type]):  
 df.loc[idx, 'Injuries'] = injury\_medians[accident\_type]  
 else:  
 df.loc[idx, 'Injuries'] = df['Injuries'].median()  
   
 # For remaining NaN values in Injuries, use a ratio based on Fatalities  
 injury\_mask = df['Injuries'].isna()  
 fatality\_injury\_ratio = df[df['Fatalities'].notna() & df['Injuries'].notna()]['Injuries'].sum() / df[df['Fatalities'].notna() & df['Injuries'].notna()]['Fatalities'].sum()  
 df.loc[injury\_mask, 'Injuries'] = df.loc[injury\_mask, 'Fatalities'] \* fatality\_injury\_ratio  
   
 # Create severity category  
 df['Severity'] = pd.cut(  
 df['Fatalities'],   
 bins=[0, 10, 50, float('inf')],  
 labels=['Low', 'Medium', 'High'],  
 right=True  
 )  
   
 return df

**geocoding.py**:

import pandas as pd  
import numpy as np  
import time  
import os  
from geopy.geocoders import Nominatim  
from geopy.exc import GeocoderTimedOut, GeocoderUnavailable  
# Cache for geocoded locations to avoid repeated API calls  
geocode\_cache = {}  
def geocode\_location(location, state=None, country="India"):  
 """  
 Geocode a location to get its latitude and longitude.  
   
 Args:  
 location: Name of the location  
 state: State/region of the location  
 country: Country (default: India)  
   
 Returns:  
 (latitude, longitude) tuple or None if geocoding fails  
 """  
 if pd.isna(location) or location == '':  
 return None  
   
 # Create a cache key  
 if pd.isna(state) or state == '':  
 cache\_key = f"{location}, {country}"  
 else:  
 cache\_key = f"{location}, {state}, {country}"  
   
 # Check if result is in cache  
 if cache\_key in geocode\_cache:  
 return geocode\_cache[cache\_key]  
   
 # Create geocoder  
 geolocator = Nominatim(user\_agent="railway\_accidents\_analysis")  
   
 # Try to geocode  
 try:  
 # First try with both location and state  
 if not pd.isna(state) and state != '':  
 query = f"{location}, {state}, {country}"  
 geocode\_result = geolocator.geocode(query)  
   
 # If that fails, try with location only  
 if geocode\_result is None:  
 query = f"{location}, {country}"  
 geocode\_result = geolocator.geocode(query)  
 else:  
 query = f"{location}, {country}"  
 geocode\_result = geolocator.geocode(query)  
   
 # If geocoding was successful, return and cache the coordinates  
 if geocode\_result:  
 coords = (geocode\_result.latitude, geocode\_result.longitude)  
 geocode\_cache[cache\_key] = coords  
 return coords  
 else:  
 # If geocoding failed, try with state only  
 if not pd.isna(state) and state != '':  
 query = f"{state}, {country}"  
 geocode\_result = geolocator.geocode(query)  
   
 if geocode\_result:  
 coords = (geocode\_result.latitude, geocode\_result.longitude)  
 geocode\_cache[cache\_key] = coords  
 return coords  
   
 geocode\_cache[cache\_key] = None  
 return None  
   
 except (GeocoderTimedOut, GeocoderUnavailable):  
 # If there's a timeout or the service is unavailable, return None  
 geocode\_cache[cache\_key] = None  
 return None  
def geocode\_locations(df):  
 """  
 Geocode all locations in the dataset.  
   
 Args:  
 df: Pandas DataFrame containing the railway accidents data  
   
 Returns:  
 DataFrame with latitude and longitude columns  
 """  
 # Make a copy of the dataframe  
 df = df.copy()  
   
 # Create empty latitude and longitude columns  
 if 'latitude' not in df.columns:  
 df['latitude'] = np.nan  
   
 if 'longitude' not in df.columns:  
 df['longitude'] = np.nan  
   
 # For each row with a missing lat/long, try to geocode  
 for idx, row in df[df['latitude'].isna() | df['longitude'].isna()].iterrows():  
 location = row['Location']  
 state = row['State/Region']  
   
 # Skip if location is missing  
 if pd.isna(location) or location == '':  
 continue  
   
 # Geocode the location  
 coords = geocode\_location(location, state)  
   
 # Update the dataframe if geocoding was successful  
 if coords:  
 df.at[idx, 'latitude'] = coords[0]  
 df.at[idx, 'longitude'] = coords[1]  
   
 # Delay to avoid hitting API rate limits  
 time.sleep(0.1)  
 else:  
 # If location geocoding failed, try state-level geocoding  
 if not pd.isna(state) and state != '':  
 state\_coords = geocode\_location(None, state)  
   
 if state\_coords:  
 df.at[idx, 'latitude'] = state\_coords[0]  
 df.at[idx, 'longitude'] = state\_coords[1]  
   
 # Delay to avoid hitting API rate limits  
 time.sleep(0.1)  
   
 # Provide default coordinates for India for any remaining missing values  
 # This is for visualization purposes only  
 default\_lat, default\_lng = 20.5937, 78.9629 # Center of India  
   
 # Fill missing values with defaults (with small random offsets to avoid overlapping)  
 mask = df['latitude'].isna() | df['longitude'].isna()  
 n\_missing = mask.sum()  
   
 if n\_missing > 0:  
 # Generate random offsets  
 lat\_offsets = np.random.uniform(-3, 3, n\_missing)  
 lng\_offsets = np.random.uniform(-3, 3, n\_missing)  
   
 # Apply offsets to default coordinates  
 df.loc[mask, 'latitude'] = default\_lat + lat\_offsets  
 df.loc[mask, 'longitude'] = default\_lng + lng\_offsets  
   
 return df

**modelling.py**:

import pandas as pd  
import numpy as np  
from sklearn.preprocessing import LabelEncoder, StandardScaler  
from sklearn.ensemble import RandomForestClassifier, IsolationForest  
from sklearn.model\_selection import train\_test\_split, cross\_val\_score  
from sklearn.metrics import classification\_report, confusion\_matrix, f1\_score  
import xgboost as xgb  
import joblib  
import os  
def train\_severity\_model(df):  
 """  
 Train a model to predict accident severity.  
   
 Args:  
 df: Preprocessed DataFrame with 'Severity' column  
   
 Returns:  
 Trained model, label encoders, and feature names  
 """  
 # Features and target  
 features = ['Accident\_Type', 'Cause', 'State/Region', 'Decade']  
 target = 'Severity'  
   
 # Drop rows with missing target or features  
 model\_df = df.dropna(subset=[target] + features)  
   
 # Encode categorical features  
 encoders = {}  
 X = pd.DataFrame()  
   
 for feature in features:  
 encoder = LabelEncoder()  
 X[feature] = encoder.fit\_transform(model\_df[feature])  
 encoders[feature] = encoder  
   
 # Encode target  
 y\_encoder = LabelEncoder()  
 y = y\_encoder.fit\_transform(model\_df[target])  
 encoders['target'] = y\_encoder  
   
 # Train model  
 model = RandomForestClassifier(  
 n\_estimators=100,  
 max\_depth=10,  
 random\_state=42,  
 n\_jobs=-1  
 )  
   
 # Use stratified cross-validation to evaluate  
 cv\_scores = cross\_val\_score(model, X, y, cv=5, scoring='f1\_weighted')  
 print(f"Cross-validation F1 scores: {cv\_scores}")  
 print(f"Mean F1 score: {cv\_scores.mean()}")  
   
 # Train on full dataset  
 model.fit(X, y)  
   
 return model, encoders, features  
def predict\_severity(model, encoders, features, input\_data):  
 """  
 Predict the severity of an accident.  
   
 Args:  
 model: Trained model  
 encoders: Dictionary of label encoders for each feature  
 features: List of feature names  
 input\_data: Dictionary with input feature values  
   
 Returns:  
 Predicted severity class and probability  
 """  
 # Encode input data  
 encoded\_input = []  
   
 for feature in features:  
 if feature in input\_data:  
 # Handle values not seen during training  
 try:  
 encoded\_value = encoders[feature].transform([input\_data[feature]])[0]  
 except:  
 # Use the most frequent class if the value was not seen during training  
 encoded\_value = encoders[feature].transform([encoders[feature].classes\_[0]])[0]  
 else:  
 # Use the most frequent class if the feature is missing  
 encoded\_value = encoders[feature].transform([encoders[feature].classes\_[0]])[0]  
   
 encoded\_input.append(encoded\_value)  
   
 # Make prediction  
 encoded\_input = np.array(encoded\_input).reshape(1, -1)  
 prediction\_encoded = model.predict(encoded\_input)[0]  
 probabilities = model.predict\_proba(encoded\_input)[0]  
   
 # Decode prediction  
 prediction = encoders['target'].inverse\_transform([prediction\_encoded])[0]  
   
 # Get probability of the predicted class  
 probability = probabilities[prediction\_encoded] \* 100  
   
 return prediction, probability  
def train\_anomaly\_detector(df):  
 """  
 Train an anomaly detection model.  
   
 Args:  
 df: Preprocessed DataFrame  
   
 Returns:  
 Trained anomaly detection model  
 """  
 # Features for anomaly detection  
 features = ['Fatalities', 'Injuries']  
   
 # Drop rows with missing features  
 model\_df = df.dropna(subset=features)  
   
 # Scale numerical features  
 scaler = StandardScaler()  
 X = scaler.fit\_transform(model\_df[features])  
   
 # Train Isolation Forest model  
 model = IsolationForest(  
 contamination=0.05, # 5% of the data will be considered anomalies  
 random\_state=42,  
 n\_jobs=-1  
 )  
   
 model.fit(X)  
   
 return model, scaler, features

**visualization.py**:

import pandas as pd  
import numpy as np  
import plotly.express as px  
import plotly.graph\_objects as go  
from plotly.subplots import make\_subplots  
def plot\_accident\_map(df):  
 """  
 Create a map visualization of accident locations.  
   
 Args:  
 df: DataFrame with latitude and longitude columns  
   
 Returns:  
 Plotly figure object  
 """  
 # Filter rows with valid coordinates  
 geo\_df = df.dropna(subset=['latitude', 'longitude'])  
   
 if len(geo\_df) == 0:  
 # Create empty map centered on India  
 fig = px.scatter\_mapbox(  
 lat=[20.5937],  
 lon=[78.9629],  
 zoom=4,  
 height=600  
 )  
 fig.update\_layout(  
 mapbox\_style="open-street-map",  
 margin={"r": 0, "t": 0, "l": 0, "b": 0}  
 )  
 return fig  
   
 # Create hover text  
 geo\_df['hover\_text'] = geo\_df.apply(  
 lambda row: f"<b>{row['Location']}, {row['State/Region']}</b><br>" +  
 f"Date: {row['Date']}<br>" +  
 f"Accident Type: {row['Accident\_Type']}<br>" +  
 f"Cause: {row['Cause']}<br>" +  
 f"Fatalities: {row['Fatalities']}<br>" +  
 f"Injuries: {row['Injuries']}<br>" +  
 f"Train: {row['Train\_Involved']}",  
 axis=1  
 )  
   
 # Create map  
 fig = px.scatter\_mapbox(  
 geo\_df,  
 lat="latitude",  
 lon="longitude",  
 color="Fatalities",  
 size="Fatalities",  
 color\_continuous\_scale="Reds",  
 size\_max=15,  
 zoom=4,  
 hover\_name="Location",  
 hover\_data=["Date", "Accident\_Type", "Fatalities", "Injuries"],  
 height=600,  
 opacity=0.8  
 )  
   
 fig.update\_layout(  
 mapbox\_style="open-street-map",  
 margin={"r": 0, "t": 0, "l": 0, "b": 0}  
 )  
   
 return fig  
def plot\_accident\_clusters(df):  
 """  
 Create a map visualization of accident clusters.  
   
 Args:  
 df: DataFrame with cluster column  
   
 Returns:  
 Plotly figure object  
 """  
 # Create a color map for clusters  
 clusters = sorted(df['cluster'].unique())  
 colors = px.colors.qualitative.Bold  
   
 # Create map  
 fig = go.Figure()  
   
 # Add a scatter trace for each cluster  
 for i, cluster in enumerate(clusters):  
 if cluster == -1:  
 # Noise points (not in any cluster)  
 cluster\_df = df[df['cluster'] == cluster]  
 fig.add\_trace(go.Scattermapbox(  
 lat=cluster\_df['latitude'],  
 lon=cluster\_df['longitude'],  
 mode='markers',  
 marker=dict(  
 size=8,  
 color='gray',  
 opacity=0.5  
 ),  
 text=cluster\_df['Location'],  
 hoverinfo='text',  
 name='Unclustered'  
 ))  
 else:  
 # Cluster points  
 cluster\_df = df[df['cluster'] == cluster]  
 fig.add\_trace(go.Scattermapbox(  
 lat=cluster\_df['latitude'],  
 lon=cluster\_df['longitude'],  
 mode='markers',  
 marker=dict(  
 size=10,  
 color=colors[i % len(colors)],  
 opacity=0.8  
 ),  
 text=cluster\_df.apply(  
 lambda row: f"{row['Location']}: {int(row['Fatalities'])} fatalities",  
 axis=1  
 ),  
 hoverinfo='text',  
 name=f'Cluster {cluster} ({len(cluster\_df)} accidents)'  
 ))  
   
 # Update layout  
 fig.update\_layout(  
 mapbox\_style="open-street-map",  
 mapbox=dict(  
 center=dict(lat=22, lon=82),  
 zoom=4  
 ),  
 margin={"r": 0, "t": 0, "l": 0, "b": 0},  
 height=600,  
 legend=dict(  
 orientation="h",  
 yanchor="bottom",  
 y=1.02,  
 xanchor="right",  
 x=1  
 )  
 )  
   
 return fig  
def plot\_temporal\_trends(df, aggregation, metric):  
 """  
 Create a visualization of accident trends over time.  
   
 Args:  
 df: DataFrame with temporal features  
 aggregation: Time aggregation level (Year, Decade, Month)  
 metric: Metric to visualize  
   
 Returns:  
 Plotly figure object  
 """  
 # Aggregate data  
 if aggregation == "Year":  
 time\_column = "Year"  
 elif aggregation == "Decade":  
 time\_column = "Decade"  
 else: # Month  
 time\_column = "Month"  
   
 # Prepare aggregated data based on metric  
 if metric == "Fatalities":  
 agg\_df = df.groupby(time\_column)['Fatalities'].sum().reset\_index()  
 y\_column = "Fatalities"  
 title = f"Total Fatalities by {aggregation}"  
 elif metric == "Accidents Count":  
 agg\_df = df.groupby(time\_column).size().reset\_index(name='Accidents\_Count')  
 y\_column = "Accidents\_Count"  
 title = f"Number of Accidents by {aggregation}"  
 else: # Average Fatalities per Accident  
 total\_fatalities = df.groupby(time\_column)['Fatalities'].sum()  
 accident\_counts = df.groupby(time\_column).size()  
 agg\_df = pd.DataFrame({  
 time\_column: total\_fatalities.index,  
 'Average\_Fatalities': total\_fatalities.values / accident\_counts.values  
 })  
 y\_column = "Average\_Fatalities"  
 title = f"Average Fatalities per Accident by {aggregation}"  
   
 # Sort by time  
 if aggregation == "Year" or aggregation == "Decade":  
 agg\_df = agg\_df.sort\_values(time\_column)  
   
 # Create figure  
 fig = px.line(  
 agg\_df,  
 x=time\_column,  
 y=y\_column,  
 markers=True,  
 title=title  
 )  
   
 # Add a trend line (moving average)  
 if len(agg\_df) > 5 and (aggregation == "Year" or aggregation == "Decade"):  
 window = min(5, len(agg\_df) // 2)  
 agg\_df['MA'] = agg\_df[y\_column].rolling(window=window, center=True).mean()  
   
 fig.add\_trace(  
 go.Scatter(  
 x=agg\_df[time\_column],  
 y=agg\_df['MA'],  
 mode='lines',  
 line=dict(color='red', width=2, dash='dash'),  
 name=f'{window}-point Moving Average'  
 )  
 )  
   
 # Update layout  
 fig.update\_layout(  
 xaxis\_title=aggregation,  
 yaxis\_title=metric,  
 hovermode="x unified"  
 )  
   
 return fig  
def plot\_severity\_distribution(df):  
 """  
 Create a visualization of the severity distribution.  
   
 Args:  
 df: DataFrame with Severity column  
   
 Returns:  
 Plotly figure object  
 """  
 severity\_counts = df['Severity'].value\_counts().reset\_index()  
 severity\_counts.columns = ['Severity', 'Count']  
   
 # Ensure correct order of severity levels  
 severity\_order = ['Low', 'Medium', 'High']  
 severity\_counts['Severity'] = pd.Categorical(  
 severity\_counts['Severity'],  
 categories=severity\_order,  
 ordered=True  
 )  
 severity\_counts = severity\_counts.sort\_values('Severity')  
   
 # Create figure  
 fig = px.bar(  
 severity\_counts,  
 x='Severity',  
 y='Count',  
 color='Severity',  
 color\_discrete\_map={  
 'Low': 'green',  
 'Medium': 'orange',  
 'High': 'red'  
 },  
 text='Count'  
 )  
   
 # Update layout  
 fig.update\_layout(  
 xaxis\_title="Severity Level",  
 yaxis\_title="Number of Accidents",  
 showlegend=False  
 )  
   
 # Add data labels  
 fig.update\_traces(texttemplate='%{text}', textposition='outside')  
   
 return fig  
def plot\_accident\_types(df):  
 """  
 Create a visualization of accident types.  
   
 Args:  
 df: DataFrame with Accident\_Type column  
   
 Returns:  
 Plotly figure object  
 """  
 # Count accidents by type  
 type\_counts = df['Accident\_Type'].value\_counts().reset\_index()  
 type\_counts.columns = ['Accident\_Type', 'Count']  
   
 # Sort by count and take top 10  
 type\_counts = type\_counts.sort\_values('Count', ascending=False).head(10)  
   
 # Create figure  
 fig = px.bar(  
 type\_counts,  
 x='Count',  
 y='Accident\_Type',  
 orientation='h',  
 text='Count'  
 )  
   
 # Update layout  
 fig.update\_layout(  
 xaxis\_title="Number of Accidents",  
 yaxis\_title="Accident Type",  
 yaxis=dict(autorange="reversed") # Reverse y-axis to show highest count at top  
 )  
   
 # Add data labels  
 fig.update\_traces(texttemplate='%{text}', textposition='outside')  
   
 return fig  
def plot\_anomalies(df, anomalies):  
 """  
 Visualize anomalies in a scatter plot.  
   
 Args:  
 df: Original DataFrame  
 anomalies: DataFrame with anomalies  
   
 Returns:  
 Plotly figure object  
 """  
 # Create a copy of the original data  
 plot\_df = df[['Fatalities', 'Injuries', 'Date', 'Location', 'Accident\_Type']].copy()  
   
 # Add anomaly flag  
 plot\_df['is\_anomaly'] = False  
 plot\_df.loc[plot\_df.index.isin(anomalies.index), 'is\_anomaly'] = True  
   
 # Create scatter plot  
 fig = px.scatter(  
 plot\_df,  
 x='Fatalities',  
 y='Injuries',  
 color='is\_anomaly',  
 size='Fatalities',  
 hover\_name='Location',  
 hover\_data=['Date', 'Accident\_Type'],  
 color\_discrete\_map={  
 False: 'blue',  
 True: 'red'  
 },  
 labels={  
 'is\_anomaly': 'Anomaly'  
 }  
 )  
   
 # Update layout  
 fig.update\_layout(  
 title='Anomaly Detection: Fatalities vs. Injuries',  
 xaxis\_title='Fatalities',  
 yaxis\_title='Injuries',  
 height=500  
 )  
   
 return fig

**anomaly\_detection.py**:

import pandas as pd  
import numpy as np  
from sklearn.preprocessing import StandardScaler  
from sklearn.ensemble import IsolationForest  
import joblib  
import os  
class AnomalyDetector:  
 """  
 Anomaly detection model for identifying unusual railway accidents.  
 """  
   
 def \_\_init\_\_(self):  
 """Initialize the model."""  
 self.model = None  
 self.scaler = None  
 self.features = ['Fatalities', 'Injuries', 'Year']  
   
 def fit(self, df):  
 """  
 Train an anomaly detection model.  
   
 Args:  
 df: Preprocessed DataFrame  
 """  
 # Features for anomaly detection  
 numeric\_features = ['Fatalities', 'Injuries']  
   
 # Get only the needed columns and drop rows with missing values  
 model\_df = df[numeric\_features].dropna()  
   
 # Add Year as a feature if it exists  
 if 'Year' in df.columns:  
 model\_df['Year'] = df.loc[model\_df.index, 'Year']  
 self.features = numeric\_features + ['Year']  
 else:  
 self.features = numeric\_features  
   
 # Scale numerical features  
 self.scaler = StandardScaler()  
 X = self.scaler.fit\_transform(model\_df[self.features])  
   
 # Train Isolation Forest model  
 self.model = IsolationForest(  
 contamination=0.05, # 5% of the data will be considered anomalies  
 random\_state=42,  
 n\_jobs=-1  
 )  
   
 self.model.fit(X)  
   
 return self  
   
 def detect\_anomalies(self, df, contamination=0.05):  
 """  
 Detect anomalies in the dataset.  
   
 Args:  
 df: DataFrame to analyze  
 contamination: Proportion of anomalies expected (0 to 0.5)  
   
 Returns:  
 DataFrame containing anomalies  
 """  
 if self.model is None:  
 self.fit(df)  
   
 # If contamination has changed, retrain the model  
 elif self.model.contamination != contamination:  
 self.model = IsolationForest(  
 contamination=contamination,  
 random\_state=42,  
 n\_jobs=-1  
 )  
   
 # Extract features and scale  
 numeric\_features = ['Fatalities', 'Injuries']  
 model\_df = df[numeric\_features].dropna()  
   
 # Add Year as a feature if it exists  
 if 'Year' in df.columns:  
 model\_df['Year'] = df.loc[model\_df.index, 'Year']  
 self.features = numeric\_features + ['Year']  
 else:  
 self.features = numeric\_features  
   
 # Scale and fit  
 X = self.scaler.transform(model\_df[self.features])  
 self.model.fit(X)  
   
 # Extract features for prediction  
 numeric\_features = ['Fatalities', 'Injuries']  
 pred\_df = df[numeric\_features].dropna()  
   
 # Add Year if it's used as a feature  
 if 'Year' in self.features and 'Year' in df.columns:  
 pred\_df['Year'] = df.loc[pred\_df.index, 'Year']  
   
 # Scale features  
 X = self.scaler.transform(pred\_df[self.features])  
   
 # Predict anomalies (-1 for anomalies, 1 for normal)  
 anomaly\_predictions = self.model.predict(X)  
 anomaly\_scores = self.model.decision\_function(X)  
   
 # Normalize scores to 0-1 range for better interpretation  
 # Lower scores indicate more anomalous points  
 normalized\_scores = 1 - (anomaly\_scores - anomaly\_scores.min()) / (anomaly\_scores.max() - anomaly\_scores.min())  
   
 # Get anomalies  
 anomaly\_indices = pred\_df.index[anomaly\_predictions == -1]  
 anomalies = df.loc[anomaly\_indices].copy()  
   
 # Add anomaly scores  
 anomalies['anomaly\_score'] = normalized\_scores[anomaly\_predictions == -1]  
   
 return anomalies.sort\_values('anomaly\_score', ascending=False)  
   
 def save(self, path):  
 """  
 Save the model to disk.  
   
 Args:  
 path: Path to save the model  
 """  
 if self.model is None:  
 raise ValueError("Model has not been trained yet. Call 'fit' first.")  
   
 # Create directory if it doesn't exist  
 os.makedirs(os.path.dirname(path), exist\_ok=True)  
   
 # Save model and scaler  
 joblib.dump({  
 'model': self.model,  
 'scaler': self.scaler,  
 'features': self.features  
 }, path)  
   
 def load(self, path):  
 """  
 Load the model from disk.  
   
 Args:  
 path: Path to the saved model  
 """  
 if not os.path.exists(path):  
 raise ValueError(f"Model file '{path}' not found")  
   
 # Load model and scaler  
 saved\_data = joblib.load(path)  
   
 self.model = saved\_data['model']  
 self.scaler = saved\_data['scaler']  
 self.features = saved\_data['features']  
   
 return self

**severity\_model.py**:

import pandas as pd  
import numpy as np  
from sklearn.preprocessing import LabelEncoder  
from sklearn.ensemble import RandomForestClassifier  
from sklearn.model\_selection import train\_test\_split, cross\_val\_score  
from sklearn.metrics import classification\_report, confusion\_matrix  
import pickle  
import joblib  
import os  
  
class SeverityModel:  
 """  
 Model for predicting accident severity.  
 """  
   
 def \_\_init\_\_(self):  
 """Initialize the model."""  
 self.model = None  
 self.encoders = {}  
 self.features = ['Accident\_Type', 'Cause', 'State/Region', 'Decade']  
 self.target = 'Severity'  
 self.feature\_importance = None  
   
 def fit(self, df):  
 """  
 Train the severity prediction model.  
   
 Args:  
 df: Preprocessed DataFrame with 'Severity' column  
 """  
 # Make sure the target and features exist  
 if self.target not in df.columns:  
 raise ValueError(f"Target column '{self.target}' not found in DataFrame")  
   
 for feature in self.features:  
 if feature not in df.columns:  
 raise ValueError(f"Feature column '{feature}' not found in DataFrame")  
   
 # Drop rows with missing target or features  
 model\_df = df.dropna(subset=[self.target] + self.features)  
   
 # Encode categorical features  
 X = pd.DataFrame()  
   
 for feature in self.features:  
 encoder = LabelEncoder()  
 X[feature] = encoder.fit\_transform(model\_df[feature])  
 self.encoders[feature] = encoder  
   
 # Encode target  
 y\_encoder = LabelEncoder()  
 y = y\_encoder.fit\_transform(model\_df[self.target])  
 self.encoders['target'] = y\_encoder  
   
 # Train model  
 self.model = RandomForestClassifier(  
 n\_estimators=100,  
 max\_depth=10,  
 random\_state=42,  
 n\_jobs=-1  
 )  
   
 # Train on full dataset  
 self.model.fit(X, y)  
   
 # Store feature importance  
 importance = self.model.feature\_importances\_  
 feature\_importance = pd.DataFrame({  
 'Feature': self.features,  
 'Importance': importance  
 })  
 self.feature\_importance = feature\_importance.sort\_values('Importance', ascending=False)  
   
 return self  
   
 def predict(self, input\_data):  
 """  
 Predict the severity of an accident.  
   
 Args:  
 input\_data: Dictionary with input feature values  
   
 Returns:  
 Predicted severity class and probability  
 """  
 if self.model is None:  
 raise ValueError("Model has not been trained yet. Call 'fit' first.")  
   
 # Encode input data  
 encoded\_input = []  
   
 for feature in self.features:  
 if feature in input\_data:  
 # Handle values not seen during training  
 try:  
 encoded\_value = self.encoders[feature].transform([input\_data[feature]])[0]  
 except:  
 # Use the most frequent class if the value was not seen during training  
 encoded\_value = self.encoders[feature].transform([self.encoders[feature].classes\_[0]])[0]  
 else:  
 # Use the most frequent class if the feature is missing  
 encoded\_value = self.encoders[feature].transform([self.encoders[feature].classes\_[0]])[0]  
   
 encoded\_input.append(encoded\_value)  
   
 # Make prediction  
 encoded\_input = np.array(encoded\_input).reshape(1, -1)  
 prediction\_encoded = self.model.predict(encoded\_input)[0]  
 probabilities = self.model.predict\_proba(encoded\_input)[0]  
   
 # Decode prediction  
 prediction = self.encoders['target'].inverse\_transform([prediction\_encoded])[0]  
   
 # Get probability of the predicted class  
 probability = probabilities[prediction\_encoded] \* 100  
   
 return prediction, probability  
   
 def get\_feature\_importance(self):  
 """  
 Get feature importance from the trained model.  
   
 Returns:  
 DataFrame with feature importance  
 """  
 if self.feature\_importance is None:  
 raise ValueError("Model has not been trained yet. Call 'fit' first.")  
   
 return self.feature\_importance  
   
 def save(self, path):  
 """  
 Save the model to disk.  
   
 Args:  
 path: Path to save the model  
 """  
 if self.model is None:  
 raise ValueError("Model has not been trained yet. Call 'fit' first.")  
   
 # Create directory if it doesn't exist  
 os.makedirs(os.path.dirname(path), exist\_ok=True)  
   
 # Save model and encoders  
 joblib.dump({  
 'model': self.model,  
 'encoders': self.encoders,  
 'features': self.features,  
 'feature\_importance': self.feature\_importance  
 }, path)  
   
 def load(self, path):  
 """  
 Load the model from disk.  
   
 Args:  
 path: Path to the saved model  
 """  
 if not os.path.exists(path):  
 raise ValueError(f"Model file '{path}' not found")  
   
 # Load model and encoders  
 saved\_data = joblib.load(path)  
   
 self.model = saved\_data['model']  
 self.encoders = saved\_data['encoders']  
 self.features = saved\_data['features']  
 self.feature\_importance = saved\_data['feature\_importance']  
   
 return self