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_feature
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("  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 follo

    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_

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 fil
    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().r
        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 Co
            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=Fals
            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", "Residu

        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
                if trend_change > 10:
                    st.info(f" The overall trend shows an increase of {trend_change:.1f}% o
                elif trend_change < -10:
                    st.success(f" The overall trend shows a decrease of {abs(trend_change):
                else:
                    st.info(f" The overall trend is relatively stable (change of {trend_cha

                # 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
# Anomaly Detection Page
elif page == "Anomaly Detection":
    st.header("Anomaly Detection")

    st.markdown("""
    This analysis identifies unusual railway accidents that deviate significantly from typic
    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', 'a
            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']} ({an
                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.

                # 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
                    st.write(f"- **Rare accident type:** {anomaly['Accident_Type']} (occurs

                # Check for unusual combinations
                cause_by_type = df.groupby('Accident_Type')['Cause'].agg(lambda x: x.mode()
                if anomaly['Accident_Type'] in cause_by_type and anomaly['Cause'] != cause_b
                    st.write(f"- **Unusual cause for this accident type:** {anomaly['Cause']

                # 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

    # 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
    df.loc[injury_mask, 'Injuries'] = df.loc[injury_mask, 'Fatalities'] * fatality_injury_r

    # 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]])
        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.ma

        # 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=Fal

        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]])
                except:
                    # Use the most frequent class if the value was not seen during training
                    encoded_value = self.encoders[feature].transform([self.encoders[feature
            else:
                # Use the most frequent class if the feature is missing
                encoded_value = self.encoders[feature].transform([self.encoders[feature].cla

            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
```