PRESIDENCY UNIVERSITY

GAIN MORE KNOWLEDGE REACH GREATER HEIGHTS

Private University Estd. in Karnataka State by Act No. 41 of 2013

BANGALORE

PRESIDENCY GROUP OVER 40 YEARS OF ACADEMIC WISDOM

ALGORITHM AND SOURCE CODE DETAILS OF REAL TIME MAPPING OF EPIDEMIC SPREAD

Batch Details

Sl. No.	Roll Number	Student Name
1	20211CSD0038	B.N BHAVANA
2	20211CSD0005	PRARTHANA S.P
3	20211CSD0042	ULLAS GOWDA M
4	20211CSD0150	SANCHIT A

School of Computer Science,

Presidency University, Bengaluru.

Under the guidance of,

Mrs. TINTU VIJITH

School of Computer Science,

Presidency University, Bengaluru

Abstract

This document outlines a comprehensive approach to analyzing and visualizing epidemic data across various states in India. The objective is to utilize multiple algorithms and libraries to explore epidemic trends, predict future cases, and classify epidemic types based on reported data. Key methodologies include the use of heatmaps for geographical visualization, time series forecasting with ARIMA, multivariate analysis using VAR, and machine learning techniques such as Random Forest and XGBoost for classification tasks.
Additionally, a user-friendly front-end interface has been implemented using the index.html file. This interface provides real-time epidemic visualizations, interactive heatmaps, and analytical graphs to enhance accessibility and user engagement. The intuitive design allows users to interact with data through drop-down selections, map views, and analytics summaries, making the insights accessible to policymakers, healthcare workers, and the general public.

Objective Overview

The primary goal of this project is to analyze and visualize epidemic data to identify trends, predict future cases, and classify diseases based on various epidemiological features. This involves leveraging different algorithms and libraries to handle data manipulation, statistical modeling, and visualization.

Key Components

Libraries Used: Pandas, Folium, Geopandas, Matplotlib, Statsmodels, Scikit-learn, XGBoost, Imbalanced-learn.
Front-End: The index.html file provides a user-friendly platform with real-time maps, district-specific analytics, and forecasting visualizations. Features like state/district selectors allow targeted exploration of epidemic trends.

Process Breakdown

1. **Data Loading and Preparation**
    - Load epidemic data from CSV files and clean the data.
    - Create new columns for epidemic types based on cases and mortality rates.
2. **Visualization Techniques**
    - Heatmaps: Display case densities on maps.
    - Animated GIFs: Show epidemic evolution over time.
    - Analytics: Include graphs highlighting disease patterns.
3. **Statistical Modeling and Machine Learning**
    - Forecast future trends using ARIMA and VAR models.
    - Classify epidemic types using Random Forest and XGBoost.
    -

Evaluation and Results

The comprehensive analysis provided actionable insights into epidemic trends. Visualizations and the user-friendly interface further bridge the gap between raw data and decision-making. By combining advanced analytics with accessible visual tools, this work supports informed public health strategies.

ALGORITHM DETAILS

Objective
To analyze and visualize epidemic data effectively, including the integration of interactive front-end features for improved accessibility.

Key Components
- **Folium:** Creates interactive heatmaps.
- **Pandas:** Handles CSV files and prepares data.
- **Geopandas:** Manages geospatial data for map overlays.
- **Matplotlib:** Produces static and animated visualizations.
- **Pillow (PIL):** Creates GIF animations.
- **Front-End:** Epidemic data visualizations and heatmaps are seamlessly integrated into the index.html file. The front-end uses embedded HTML elements like <iframe> for displaying dynamic maps and analytics.

Process
1. Load the epidemic data from a CSV file into a Pandas DataFrame.
2. Extract relevant columns (latitude, longitude, and number of cases) for heatmap generation.
3. Initialize a Folium map centered at specific coordinates with a defined zoom level.
4. Create a heatmap using the extracted data and add it to the map object.
5. Save the final map, including the heatmap, as an HTML file for easy sharing and viewing.

Source Code Details

```
import folium
import pandas as pd
from folium.plugins import HeatMap

mapObj = folium.Map(location=[24.2170111233401, 81.0791015625000],
zoom_start=5)
data = pd.read_csv("/content/epidemic_data_2024.csv")
data.head()
print(data.columns)
cols = data[['Latitude ', 'Longitude ', 'Cases']]
HeatMap(cols).add_to(mapObj)
mapObj.save("epidemic_heatmap.html")
```

Algorithm Details for Animated Visualization

Objective
The goal of this code is to visualize epidemic data for different states in India over time, creating an animated GIF that shows the total cases for each date.

Key Components
- **Geopandas:** Used for handling geospatial data and plotting geographical maps.
- **Pandas:** Used for data manipulation and analysis.

- **Matplotlib:** Used for creating static, interactive, and animated visualizations in Python.
- **Pillow (PIL):** Used for image processing and creating GIFs.

Process

1. Load epidemic data and geographical shape data for India.
2. Merge the datasets based on state names.
3. Fill missing values in the merged data with zeros.
4. Create a plot for each date, visualizing the number of cases for each state.
5. Save each plot as a PNG image in memory.
6. Compile all images into a GIF and save it.

Source Code Details

```
import geopandas as gpd
import pandas as pd
import matplotlib.pyplot as plt
import PIL
import io

epidemic_data = pd.read_csv('/content/epidemic_data_2024_test.csv')
epidemic_data.columns
india_shape = gpd.read_file('/content/gadm41_IND_1.json')
print(india_shape.columns)
print(india_shape.head())
merged_data = india_shape.merge(epidemic_data, left_on='NAME_1',
right_on='State', how='left')

for date in epidemic_data.columns[1:]:
    merged_data[date].fillna(0, inplace=True)

image_frames = []

for date in epidemic_data.columns[1:]:
    fig, ax = plt.subplots(figsize=(15, 15))
    merged_data.plot(column=date, cmap='Reds', legend=True, ax=ax,
edgecolor='black', linewidth=0.5)
    ax.set_title(f'Real Time Mapping of Epidemic\nTotal Cases on
{date}', fontsize=20)
    ax.set_axis_off()
    buf = io.BytesIO()
    plt.savefig(buf, format='png', bbox_inches='tight')
    buf.seek(0)
    image_frames.append(PIL.Image.open(buf))
    plt.close()

image_frames[0].save('output_map_animation.gif', format='GIF',
append_images=image_frames[1:], save_all=True, duration=1000, loop=1)
print("GIF successfully created as 'output_map_animation.gif'")
```

Algorithm Details for Epidemic Forecasting

Objective

The goal is to forecast future daily epidemic cases for a specific state using the ARIMA (AutoRegressive Integrated Moving Average) time series model.

Key Components

- **Pandas**: For data manipulation and analysis.
- **NumPy**: For numerical operations.
- **Matplotlib**: For plotting data visualizations.
- **Statsmodels**: For statistical modeling, including ARIMA.

Process

1. Load the epidemic data from a CSV file and parse dates.
2. Filter the data for a specific state and resample it to get daily case counts.
3. Visualize the daily cases over time.
4. Check for stationarity using the Augmented Dickey-Fuller test (ADF).
5. If the data is non-stationary, apply differencing.
6. Generate ACF (Autocorrelation Function) and PACF (Partial Autocorrelation Function) plots to determine ARIMA parameters (p, d, q).
7. Fit the ARIMA model using the identified parameters.
8. Forecast future cases and visualize the results.

Source Code Details

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.stattools import adfuller
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

data = pd.read_csv('/content/epidemic_data_2024.csv',
parse_dates=['Date'], index_col='Date')
state_name = 'Uttar Pradesh'
state_data = data[data['State'] == state_name]
daily_cases = state_data.resample('D')['Cases'].sum()

plt.figure(figsize=(12, 6))
plt.plot(daily_cases, label='Daily Cases')
plt.title(f'Daily Cases for {state_name}')
plt.xlabel('Date')
plt.ylabel('Number of Cases')
plt.legend()
plt.show()

result = adfuller(daily_cases)
print('ADF Statistic:', result[0])
print('p-value:', result[1])

daily_cases_diff = daily_cases.diff().dropna()

plot_acf(daily_cases_diff)
plot_pacf(daily_cases_diff)
plt.show()

p, d, q = 1, 1, 1
model = ARIMA(daily_cases, order=(p, d, q))
model_fit = model.fit()
```

```
forecast = model_fit.forecast(steps=10)
print(forecast)

plt.figure(figsize=(12, 6))
plt.plot(daily_cases, label='Observed Cases')
plt.plot(forecast.index, forecast, label='Forecasted Cases',
color='red')
plt.title(f'Forecast of Daily Cases for {state_name}')
plt.xlabel('Date')
plt.ylabel('Number of Cases')
plt.legend()
plt.show()
```

Algorithm Details for Multi-State Analysis

Objective

The goal is to analyze the relationships between epidemic cases across different states using the VAR (Vector AutoRegression) model, which captures linear interdependencies among multiple time series.

Key Components

- **Pandas**: For data manipulation and analysis.
- **NumPy**: For numerical operations.
- **Matplotlib**: For data visualization.
- **Statsmodels**: For statistical modeling, including the VAR model.

Process

1. Load epidemic data from a CSV file and parse dates.
2. Pivot the data to create a multivariate time series format.
3. Check for stationarity of each state’s time series and apply differencing if necessary.
4. Fit the VAR model to the data.
5. Make forecasts for the specified number of future steps.
6. Visualize both observed and forecasted values.

Source Code Details

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.api import VAR
from statsmodels.tsa.stattools import adfuller

data = pd.read_csv('/content/epidemic_data_2024.csv',
parse_dates=['Date'], index_col='Date')
pivot_data = data.pivot_table(index='Date', columns='State',
values='Cases', aggfunc='sum')
pivot_data.fillna(0, inplace=True)
# Check for stationarity and difference if necessary
def check_stationarity(series):
    result = adfuller(series)
    return result[1] <= 0.05  # p-value < 0.05 indicates stationarity

# Check stationarity for each state and apply differencing if needed
for state in pivot_data.columns:
    if not check_stationarity(pivot_data[state]):
        print(f"{state} is not stationary. Differencing...")
        pivot_data[state] = pivot_data[state].diff().dropna()

# Drop any NaN values created by differencing
pivot_data.dropna(inplace=True)

# Fit the VAR model
model = VAR(pivot_data)

# Fit the model with a maximum of 5 lags and use AIC for model
selection
model_fit = model.fit(maxlags=5, ic='aic')

# Print model summary to examine the estimated coefficients and model
fit
print(model_fit.summary())

# Make forecasts
forecast_steps = 10  # Forecast the next 10 days

# Check if model_fit.k_ar is greater than 0 before using it
if model_fit.k_ar > 0:
    forecast = model_fit.forecast(pivot_data.values[-
model_fit.k_ar:], steps=forecast_steps)
else:
    print("Warning: No lags were selected by the model. Using k_ar =
1 for forecasting.")
    model_fit.k_ar = 1
    forecast = model_fit.forecast(pivot_data.values[-
model_fit.k_ar:], steps=forecast_steps)

forecast_index = pd.date_range(start=pivot_data.index[-1] +
pd.Timedelta(days=1), periods=forecast_steps)
forecast_df = pd.DataFrame(forecast, index=forecast_index,
columns=pivot_data.columns)

# Plot observed and forecasted values
plt.figure(figsize=(12, 6))
for state in pivot_data.columns:
    plt.plot(pivot_data[state], label=f'Observed {state}')
    plt.plot(forecast_df[state], label=f'Forecasted {state}',
linestyle='--')

plt.title('Observed and Forecasted Cases')
plt.xlabel('Date')
plt.ylabel('Number of Cases')
plt.legend()
plt.show()
```

Algorithm Details

1. **Objective:**
    - The goal is to predict future epidemic cases for different states using a Random Forest Regressor, which is an ensemble learning method that operates by constructing multiple decision trees during training.

2. **Key Components**:
    - **Pandas**: For data manipulation and analysis.
    - **NumPy**: For numerical operations.
    - **Matplotlib**: For data visualization.
    - **Scikit-learn**: For machine learning utilities, including model training, evaluation, and hyperparameter tuning.

3. **Process**:
    - Load epidemic data from a CSV file.
    - Pivot the data to create a multivariate time series.
    - Create lagged features for the model.
    - Split the data into training and testing sets.
    - Train a Random Forest model for each state.
    - Evaluate the model using Mean Squared Error (MSE) and R² score.
    - Visualize the actual vs. predicted values for a selected state.
    - Perform hyperparameter tuning using Grid Search.

Source Code Details

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.model_selection import GridSearchCV

# Load your dataset
data = pd.read_csv('/content/epidemic_data_2024.csv',
parse_dates=['Date'])

# Pivot the data to create a multivariate time series
pivot_data = data.pivot_table(index='Date', columns='State',
values='Cases', aggfunc='sum')

# Fill missing values (if any)
pivot_data.fillna(0, inplace=True)

# Create features (lagged values) and target
def create_features(df, n_lags=3):
    for lag in range(1, n_lags + 1):
        for state in df.columns:
            df[f'{state}_lag{lag}'] = df[state].shift(lag)
    return df.fillna(0)

# Create features
feature_data = create_features(pivot_data)

# Split the data into features and target
X = feature_data[[col for col in feature_data.columns if '_lag' in
col]]
y = feature_data[pivot_data.columns]

# Train-test split
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Train Random Forest Regressor for each state
models = {}
predictions = {}
for state in y.columns:
    model = RandomForestRegressor(n_estimators=100, random_state=42)
    model.fit(X_train, y_train[state])
    predictions[state] = model.predict(X_test)
    models[state] = model

    # Evaluate the model
    mse = mean_squared_error(y_test[state], predictions[state])
    r2 = r2_score(y_test[state], predictions[state])
    print(f'{state} - MSE: {mse:.2f}, R^2: {r2:.2f}')

# Plotting actual vs predicted values for a selected state
selected_state = 'Uttar Pradesh'  # Replace with your state name
plt.figure(figsize=(12, 6))
plt.plot(y_test[selected_state].values, label='Actual Cases',
color='blue')
plt.plot(predictions[selected_state], label='Predicted Cases',
color='orange', linestyle='--')
plt.title(f'Actual vs Predicted Cases for {selected_state}')
plt.xlabel('Sample Index')
plt.ylabel('Number of Cases')
plt.legend()
plt.show()

# Define the model
rf = RandomForestRegressor(random_state=42)

# Define the parameter grid
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
}

# Perform grid search
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid,
                           scoring='neg_mean_squared_error', cv=3,
n_jobs=-1)

# Fit the grid search
grid_search.fit(X_train, y_train[selected_state])  # Use a specific
state

# Best parameters
print("Best parameters:", grid_search.best_params_)

# Best estimator
best_model = grid_search.best_estimator_

# Predictions with the best model
```

```
best_predictions = best_model.predict(X_test)

# Evaluate the best model
best_mse = mean_squared_error(y_test[selected_state],
best_predictions)
best_r2 = r2_score(y_test[selected_state], best_predictions)
print(f'Best Model - {selected_state} - MSE: {best_mse:.2f}, R^2:
{best_r2:.2f}')
```

Algorithm Details

1. **Objective:**
    - The goal is to classify the epidemic type (Low, Moderate, High) based on various features related to the epidemic data using the XGBoost classifier.

2. **Key Components:**
    - **Pandas:** For data manipulation and analysis.
    - **NumPy:** For numerical operations.
    - **Scikit-learn:** For model training, evaluation, and cross-validation.
    - **Imbalanced-learn (imblearn):** For handling class imbalance using SMOTE.
    - **XGBoost:** For the classification model.

3. **Process:**
    - Load the dataset and create a new column Epidemic_Type based on the number of cases.
    - Loop through each state, checking for sufficient data.
    - Define features and target variables.
    - Split the data into training and testing sets.
    - Use SMOTE to address class imbalance.
    - Train an XGBoost model and evaluate it using cross-validation.
    - Make predictions and output results.

Source Code Details

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split,
StratifiedKFold, cross_val_score
from sklearn.metrics import classification_report, confusion_matrix
from imblearn.over_sampling import SMOTE
import xgboost as xgb

# Load your dataset
data = pd.read_csv('/content/epidemic_data_2024.csv')

# Create the Epidemic_Type column based on Cases
low_threshold = data['Cases'].quantile(0.25)  # 25th percentile
high_threshold = data['Cases'].quantile(0.75)  # 75th percentile

conditions = [
    (data['Cases'] < low_threshold),
    (data['Cases'] >= low_threshold) & (data['Cases'] <
high_threshold),
    (data['Cases'] >= high_threshold)
]
choices = [0, 1, 2]  # 0: Low, 1: Moderate, 2: High
data['Epidemic_Type'] = np.select(conditions, choices, default=-1)

# List to store results
results = []

# Loop through each unique state
for state in data['State'].dropna().unique():
    state_data = data[data['State'] == state]

    # Check if there are enough samples to train the model
    if len(state_data) < 10:  # Adjust this threshold as needed
        print(f"Not enough data for {state}, skipping.")
        continue

    # Define features and target variable
    X = state_data[['Cases', 'Recoveries', 'Mortality Rate (%)',
'Population',
                     'Vaccination Rate (%)', 'Testing Rate (per
1,000)',
                     'Infection Rate (%)', 'Healthcare Facilities',
                     'Average Age', 'Urbanization Rate (%)',
                     'Mobility Index', 'Cleanliness Ratio',
                     'Disease Awareness Programs', 'Latitude ',
'Longitude ']]
    y = state_data['Epidemic_Type']

    # Train-test split
    X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

    # Apply SMOTE for handling class imbalance
    smote = SMOTE(random_state=42, k_neighbors=min(5,
y_train.value_counts().min() - 1))
    X_resampled, y_resampled = smote.fit_resample(X_train, y_train)

    # Train XGBoost model with adjusted parameters
    xgb_model = xgb.XGBClassifier(
        n_estimators=30,          # Reduce the number of trees
        max_depth=2,              # Limit the depth of trees
        min_child_weight=3,       # Minimum sum of instance weight
(hessian) needed in a child
        subsample=0.8,            # Use only a fraction of samples
for fitting
        colsample_bytree=0.8,     # Proportion of features to be used
        gamma=1,                  # Minimum loss reduction required
to make a further partition
        random_state=42,
        eval_metric='mlogloss'    # Evaluation metric
    )

    # Cross-validation to evaluate model performance
    skf = StratifiedKFold(n_splits=5)
    cv_scores = cross_val_score(xgb_model, X_resampled, y_resampled,
cv=skf)
    print(f"Cross-validated scores for {state}: {cv_scores}")
    print(f"Mean accuracy: {cv_scores.mean()}")

    # Fit the model on the resampled training data
    xgb_model.fit(X_resampled, y_resampled)

    # Make predictions
    y_pred_xgb = xgb_model.predict(X_test)

    # Print results
    print(f"XGBoost Results for {state}:")
    print(confusion_matrix(y_test, y_pred_xgb))
    print(classification_report(y_test, y_pred_xgb))

    # Output predicted epidemic types with state name
    for index, predicted in zip(X_test.index, y_pred_xgb):
        state_name = state  # Use the state name for the current
iteration
        print(f"Predicted Epidemic Type for {state_name} (index
{index}): {predicted}")
```

This code effectively implements an XGBoost classifier to predict epidemic types based on various features, addressing class imbalance and evaluating the model's performance through cross-validation.

Algorithm Details

1. **Objective**:
   - The goal is to classify the epidemic type (Low, Moderate, High) based on various features related to epidemic data using the Gradient Boosting classifier.

2. **Key Components**:
   - **Pandas**: For data manipulation and analysis.
   - **NumPy**: For numerical operations.
   - **Scikit-learn**: For model training, evaluation, and cross-validation.
   - **Imbalanced-learn (imblearn)**: For handling class imbalance using SMOTE.
   - **Scikit-learn's GradientBoostingClassifier**: For the classification model.

3. **Process**:
   - Load the dataset and create a new column Epidemic_Type based on the number of cases.
   - Loop through each state, checking for sufficient data.
   - Define features and target variables.
   - Split the data into training and testing sets.
   - Use SMOTE to address class imbalance.
   - Train a Gradient Boosting model and evaluate it using cross-validation.
   - Make predictions and output results.

Source Code Details

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split,
StratifiedKFold, cross_val_score
from sklearn.metrics import classification_report, confusion_matrix
from imblearn.over_sampling import SMOTE
from sklearn.ensemble import GradientBoostingClassifier

# Load your dataset
data = pd.read_csv('/content/epidemic_data_2024.csv')

# Create the Epidemic_Type column based on Cases
low_threshold = data['Cases'].quantile(0.25)  # 25th percentile
high_threshold = data['Cases'].quantile(0.75)  # 75th percentile

conditions = [
    (data['Cases'] < low_threshold),
    (data['Cases'] >= low_threshold) & (data['Cases'] <
high_threshold),
    (data['Cases'] >= high_threshold)
]
choices = [0, 1, 2]  # 0: Low, 1: Moderate, 2: High
data['Epidemic_Type'] = np.select(conditions, choices, default=-1)

# List to store results
results = []

# Loop through each unique state
for state in data['State'].dropna().unique():
    state_data = data[data['State'] == state]

    # Check if there are enough samples to train the model
    if len(state_data) < 10:  # Adjust this threshold as needed
        print(f"Not enough data for {state}, skipping.")
        continue

    # Define features and target variable
    X = state_data[['Cases', 'Recoveries', 'Mortality Rate (%)',
'Population',
                     'Vaccination Rate (%)', 'Testing Rate (per
1,000)',
                     'Infection Rate (%)', 'Healthcare Facilities',
                     'Average Age', 'Urbanization Rate (%)',
                     'Mobility Index', 'Cleanliness Ratio',
                     'Disease Awareness Programs', 'Latitude ',
'Longitude ']]
    y = state_data['Epidemic_Type']

    # Train-test split
    X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

    # Apply SMOTE for handling class imbalance
    smote = SMOTE(random_state=42, k_neighbors=min(5,
y_train.value_counts().min() - 1))
    X_resampled, y_resampled = smote.fit_resample(X_train, y_train)

    # Train Gradient Boosting model with adjusted parameters
    gb_model = GradientBoostingClassifier(
        n_estimators=30,          # Number of boosting stages to be
run
        max_depth=2,              # Limit the depth of the individual
estimators
        min_samples_split=2,      # Minimum number of samples
required to split an internal node
        learning_rate=0.1,        # Step size shrinks the
contribution of each tree
        random_state=42
    )

    # Cross-validation to evaluate model performance
    skf = StratifiedKFold(n_splits=5)
    cv_scores = cross_val_score(gb_model, X_resampled, y_resampled,
cv=skf)
    print(f"Cross-validated scores for {state}: {cv_scores}")
    print(f"Mean accuracy: {cv_scores.mean()}")

    # Fit the model on the resampled training data
    gb_model.fit(X_resampled, y_resampled)

    # Make predictions
    y_pred_gb = gb_model.predict(X_test)

    # Print results
    print(f"Gradient Boosting Results for {state}:")
    print(confusion_matrix(y_test, y_pred_gb))
    print(classification_report(y_test, y_pred_gb))

    # Output predicted epidemic types with state name
    for index, predicted in zip(X_test.index, y_pred_gb):
        state_name = state  # Use the state name for the current
iteration
        print(f"Predicted Epidemic Type for {state_name} (index
{index}): {predicted}")
```

Algorithm Details

1. Objectives

- **Classify Epidemic Types:** The primary goal is to classify the epidemic type (Low, Moderate, High) based on various features related to epidemic data.
- **Handle Class Imbalance:** Use SMOTE to address class imbalance in the dataset, ensuring that all classes are adequately represented.

2. Key Components

- **Pandas:** For data manipulation and analysis.
- **NumPy:** For numerical operations.
- **Scikit-learn:** For model training, evaluation, and cross-validation.
- **Imbalanced-learn (imblearn):** For handling class imbalance using SMOTE.
- **Scikit-learn's GradientBoostingClassifier:** For implementing the classification model.
- **Matplotlib & Seaborn:** For data visualization.

3. Process

1. **Load the Dataset:** Import the necessary libraries and load the dataset into a DataFrame.
2. **Create Epidemic_Type Column:** Generate a new column to classify the epidemic type based on the number of cases.
3. **Visualize Class Distribution Before SMOTE:** Create a count plot to visualize the distribution of epidemic types by state before applying SMOTE.
4. **Loop Through Each State:** For each state, check if there's sufficient data to apply SMOTE.
5. **Define Features and Target Variables:** Specify which columns will be used as features and which will be the target variable.
6. **Split the Data:** Use train_test_split to create training and testing datasets.
7. **Apply SMOTE:** Implement SMOTE to address class imbalance in the training set.
8. **Train the Gradient Boosting Model:** Fit the model to the resampled training data.
9. **Evaluate the Model:** Make predictions and evaluate the model using metrics such as accuracy and classification report.
10. **Visualize Class Distribution After SMOTE:** Create a count plot to visualize the distribution of epidemic types by state after applying SMOTE.

Source code details

```
# Import necessary libraries
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from imblearn.over_sampling import SMOTE
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import classification_report, accuracy_score

# Load your dataset
data = pd.read_csv('epidemic_data.csv')  # Adjust the path as 
necessary

# Create the Epidemic_Type column
def classify_epidemic(row):
    if row['Cases'] < 100:
        return 'Low'
    elif 100 <= row['Cases'] < 500:
        return 'Moderate'
    else:
        return 'High'

data['Epidemic_Type'] = data.apply(classify_epidemic, axis=1)

# Before SMOTE: Visualize class distribution by state
plt.figure(figsize=(12, 6))
sns.countplot(data=data, x='Epidemic_Type', hue='State', 
palette='viridis')
plt.title('Class Distribution by State (Before SMOTE)')
plt.xlabel('Epidemic Type')
plt.ylabel('Count')
plt.legend(title='State', bbox_to_anchor=(1, 1))
plt.show()

# Initialize lists to collect resampled data for all states
resampled_data = []

# Loop through each state to apply SMOTE and store results
for state in data['State'].dropna().unique():
    state_data = data[data['State'] == state]

    if len(state_data) < 10:
        continue  # Skip states with fewer than 10 samples

    # Define features and target
    X = state_data[['Cases', 'Recoveries', 'Mortality Rate (%)',
'Population',
                    'Vaccination Rate (%)', 'Testing Rate (per
1,000)',
                    'Infection Rate (%)', 'Healthcare Facilities',
                    'Average Age', 'Urbanization Rate (%)',
                    'Mobility Index', 'Cleanliness Ratio',
                    'Disease Awareness Programs', 'Latitude ',
'Longitude ']]
    y = state_data['Epidemic_Type']

    # Train-test split
    X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

    # Apply SMOTE with dynamic k_neighbors based on the smallest
class size
    minority_class_size = y_train.value_counts().min()
    smote = SMOTE(random_state=42, k_neighbors=min(5,
minority_class_size - 1))
    X_resampled, y_resampled = smote.fit_resample(X_train, y_train)

    # Store the resampled data along with the state name
    resampled_data.extend([(state, epidemic_type) for epidemic_type
in y_resampled])

# Convert the collected resampled data to a DataFrame for plotting
resampled_df = pd.DataFrame(resampled_data, columns=['State',
'Epidemic_Type'])

# After SMOTE: Visualize combined distribution for all states
plt.figure(figsize=(12, 6))
sns.countplot(data=resampled_df, x='Epidemic_Type', hue='State',
palette='viridis')
plt.title('Class Distribution by State (After SMOTE)')
plt.xlabel('Epidemic Type')
plt.ylabel('Count')
plt.legend(title='State', bbox_to_anchor=(1, 1))
plt.show()

# Train the Gradient Boosting Model
model = GradientBoostingClassifier(random_state=42)
model.fit(X_resampled, y_resampled)
```

```
# Evaluate the model
y_pred = model.predict(X_test)
print("Accuracy:", accuracy_score(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

Algorithm Details

1. Objectives

- **Evaluate Model Performance:** The primary goal is to assess the performance of the Gradient Boosting classifier across different states using cross-validation.
- **Handle Class Imbalance:** Ensure that the model is trained on a balanced dataset by applying SMOTE, which synthesizes minority class samples.

2. Key Components

- **Pandas:** For data manipulation and analysis.
- **NumPy:** For numerical operations.
- **Scikit-learn:** For model training, evaluation, and cross-validation.
- **Imbalanced-learn (imblearn):** For handling class imbalance using SMOTE.
- **GradientBoostingClassifier:** The machine learning model used for classification.
- **StratifiedKFold:** A cross-validation strategy that maintains the proportion of classes in each fold.
- **Matplotlib & Seaborn:** For data visualization.

3. Process

1. **Load the Dataset:** Import the necessary libraries and load the dataset into a DataFrame.
2. **Create Epidemic_Type Column:** Generate a new column to classify the epidemic type based on the number of cases.
3. **Loop Through Each State:** For each state, check if there's sufficient data to apply SMOTE.
4. **Define Features and Target Variables:** Specify which columns will be used as features and which will be the target variable.
5. **Train-Test Split:** Split the data into training and testing sets.
6. **Apply SMOTE:** Implement SMOTE to balance the classes in the training set.
7. **Initialize the Gradient Boosting Model:** Set up the model for training.
8. **Set Up Stratified K-Folds:** Create a cross-validation strategy that maintains class distribution.
9. **Perform Cross-Validation:** Calculate cross-validation scores for the model on the resampled data.
10. **Convert Scores to DataFrame:** Prepare the scores for visualization.
11. **Visualize Cross-Validation Scores:** Create a boxplot to visualize the distribution of cross-validation scores by state.

Source Code Details

```
# Import necessary libraries
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import cross_val_score, StratifiedKFold
from sklearn.ensemble import GradientBoostingClassifier
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split

# Load your dataset
data = pd.read_csv('epidemic_data.csv')  # Adjust the path as
necessary

# Create the Epidemic_Type column
def classify_epidemic(row):
    if row['Cases'] < 100:
        return 'Low'
    elif 100 <= row['Cases'] < 500:
        return 'Moderate'
    else:
        return 'High'

data['Epidemic_Type'] = data.apply(classify_epidemic, axis=1)

# Initialize a dictionary to store cross-validation scores per state
cv_scores_dict = {}

# Loop through each state to apply SMOTE and store results
for state in data['State'].dropna().unique():
    state_data = data[data['State'] == state]

    if len(state_data) < 10:
        continue  # Skip states with fewer than 10 samples

    # Define features and target
    X = state_data[['Cases', 'Recoveries', 'Mortality Rate (%)',
'Population',
                    'Vaccination Rate (%)', 'Testing Rate (per
1,000)',
                    'Infection Rate (%)', 'Healthcare Facilities',
                    'Average Age', 'Urbanization Rate (%)',
                    'Mobility Index', 'Cleanliness Ratio',
                    'Disease Awareness Programs', 'Latitude ',
'Longitude ']]
    y = state_data['Epidemic_Type']

    # Train-test split
    X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

    # Apply SMOTE
    smote = SMOTE(random_state=42)
    X_resampled, y_resampled = smote.fit_resample(X_train, y_train)

    # Initialize the Gradient Boosting model
    gb_model = GradientBoostingClassifier(random_state=42)

    # Set up StratifiedKFold for cross-validation
    skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
    # Store cross-validation scores per state
    cv_scores_dict[state] = cross_val_score(gb_model, X_resampled,
y_resampled, cv=skf)

# Convert to DataFrame for easy plotting
cv_df = pd.DataFrame(cv_scores_dict).melt(var_name='State',
value_name='Cross-Validation Score')

# Plotting the cross-validation scores
plt.figure(figsize=(14, 8))
sns.boxplot(data=cv_df, x='State', y='Cross-Validation Score',
palette='plasma')
plt.xticks(rotation=90)
plt.title('Cross-Validation Scores by State')
plt.ylabel('Score')
plt.xlabel('State')
plt.show()
```

Algorithm Details

1. Objectives
   - Evaluate Model Predictions: The main goal is to visualize the performance of the Gradient Boosting classifier by displaying a confusion matrix for each state.
   - Understand Misclassifications: The confusion matrix helps identify how well the model predicts each class and where it makes errors.

2. Key Components
   - Pandas: For data manipulation and analysis.
   - NumPy: For numerical operations.
   - Scikit-learn: For model training, evaluation, and metrics.
   - Imbalanced-learn (imblearn): For handling class imbalance using SMOTE.
   - GradientBoostingClassifier: The machine learning model used for classification.
   - Confusion Matrix: A table used to evaluate the performance of a classification model.
   - Seaborn & Matplotlib: For data visualization, specifically for displaying the confusion matrix.

3. Process
   1. Load the Dataset: Import the necessary libraries and load the dataset into a DataFrame.
   2. Create Epidemic_Type Column: Generate a new column to classify the epidemic type based on the number of cases.
   3. Loop Through Each State: For each state, check if there's sufficient data.
   4. Define Features and Target Variables: Specify which columns will be used as features and which will be the target variable.
   5. Train-Test Split: Split the data into training and testing sets.
   6. Apply SMOTE: Implement SMOTE to balance the classes in the training set.
   7. Initialize and Fit the Model: Set up and train the Gradient Boosting model on the resampled data.
   8. Predict on the Test Set: Use the trained model to make

predictions on the test set.

9. Calculate Confusion Matrix: Compute the confusion matrix to evaluate model performance.
10. Visualize the Confusion Matrix: Create a heatmap to visualize the confusion matrix.

Source Code Details

```
# Import necessary libraries
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingClassifier
from imblearn.over_sampling import SMOTE
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# Load your dataset
data = pd.read_csv('epidemic_data.csv')  # Adjust the path as
necessary

# Create the Epidemic_Type column
def classify_epidemic(row):
    if row['Cases'] < 100:
        return 'Low'
    elif 100 <= row['Cases'] < 500:
        return 'Moderate'
    else:
        return 'High'

data['Epidemic_Type'] = data.apply(classify_epidemic, axis=1)

# Loop through each state to fit the model and display confusion
matrix
for state in data['State'].dropna().unique():
    state_data = data[data['State'] == state]
    if len(state_data) < 10:
        continue  # Skip states with fewer than 10 samples

    # Define features and target
    X = state_data[['Cases', 'Recoveries', 'Mortality Rate (%)',
'Population',
                    'Vaccination Rate (%)', 'Testing Rate (per
1,000)',
                    'Infection Rate (%)', 'Healthcare Facilities',
                    'Average Age', 'Urbanization Rate (%)',
                    'Mobility Index', 'Cleanliness Ratio',
                    'Disease Awareness Programs', 'Latitude ',
'Longitude ']]
    y = state_data['Epidemic_Type']

    # Train-test split
    X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

    # Apply SMOTE
    smote = SMOTE(random_state=42)
    X_resampled, y_resampled = smote.fit_resample(X_train, y_train)

    # Initialize the Gradient Boosting model
    gb_model = GradientBoostingClassifier(random_state=42)

    # Fit the model on the resampled data
    gb_model.fit(X_resampled, y_resampled)

    # Predict on the test set
    y_pred_gb = gb_model.predict(X_test)

    # Calculate confusion matrix
    cm = confusion_matrix(y_test, y_pred_gb)

    # Display confusion matrix
    plt.figure(figsize=(6, 5))
    sns.heatmap(cm, annot=True, fmt="d", cmap="YlGnBu", cbar=False,
                xticklabels=gb_model.classes_,
yticklabels=gb_model.classes_)
    plt.title(f'Confusion Matrix for {state}')
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.show()
```

Algorithm Details

1. Objectives

- Evaluate Feature Contribution: The goal is to visualize the importance of each feature in the Gradient Boosting model, helping to understand which features contribute most to the model's predictions.
- Inform Feature Selection: This analysis can guide decisions on which features to keep or remove in future model iterations.

2. Key Components

- Pandas: For data manipulation and analysis.
- NumPy: For numerical operations.
- Scikit-learn: For model training and evaluation.
- Matplotlib & Seaborn: For data visualization, specifically for displaying feature importance.

3. Process

1. Calculate Feature Importances: Extract the feature importances from the trained Gradient Boosting model.
2. Prepare Data for Visualization: Create a DataFrame containing feature names and their corresponding importances.
3. Sort the DataFrame: Sort the DataFrame by importance in descending order to highlight the most important features.
4. Visualize Feature Importance: Use a bar plot to visualize the importance of each feature.

Source Code Details

```
# Assuming gb_model is already trained and X is defined
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Calculate feature importances
importances = gb_model.feature_importances_
feature_names = X.columns

# Create a DataFrame for feature importances
feature_importance_df = pd.DataFrame({'Feature': feature_names,
'Importance': importances})
feature_importance_df =
feature_importance_df.sort_values(by='Importance', ascending=False)

# Plotting feature importance
plt.figure(figsize=(12, 8))
sns.barplot(data=feature_importance_df, x='Importance', y='Feature',
palette='coolwarm')
plt.title('Feature Importance in Gradient Boosting Model')
plt.show()
```

Algorithm Details

1. Objectives

- Predict Epidemic Types: The goal is to predict epidemic types for different states in India using a Gradient Boosting model.
- Visualize Predictions: The predictions are visualized on a geographical map to show the distribution of predicted epidemic types across states.

2. Key Components

- Pandas: For data manipulation and analysis.
- NumPy: For numerical operations.
- Scikit-learn: For model training, evaluation, and metrics.
- Imbalanced-learn (imblearn): For handling class imbalance using SMOTE.
- GradientBoostingClassifier: The machine learning model used for classification.
- Geopandas: For handling geographical data and plotting maps.
- Matplotlib & Seaborn: For data visualization.

3. Process

1. Feature and Target Definition: Define the features (X) and target variable (y) for the model.
2. Store State Information: Keep track of the state information for later use in predictions.
3. Train-Test Split: Split the dataset into training and testing sets while retaining the state information.
4. Dynamic SMOTE Application: Use SMOTE to balance the training set, dynamically adjusting the number of neighbors based on the smallest class size.
5. Model Training: Train the Gradient Boosting model on the resampled data.
6. Predictions: Make predictions on the test set.
7. Store Predictions: Create a DataFrame with predictions for each state and append it to a list.
8. Concatenate All Predictions: Combine all state predictions into a single DataFrame.
9. Geographical Visualization: Plot the geographical distribution

of predicted epidemic types using a map of India.

Source Code Details

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingClassifier
from imblearn.over_sampling import SMOTE
import geopandas as gpd

# Assuming `data` is your main DataFrame containing the data

all_predictions = []

# Loop through each unique state
for state in data['State'].dropna().unique():
    state_data = data[data['State'] == state]
    if len(state_data) < 10:
        continue  # Skip states with fewer than 10 samples

    # Define features and target
    X = state_data[['Cases', 'Recoveries', 'Mortality Rate (%)', 
'Population',
                    'Vaccination Rate (%)', 'Testing Rate (per 
1,000)',
                    'Infection Rate (%)', 'Healthcare Facilities',
                    'Average Age', 'Urbanization Rate (%)',
                    'Mobility Index', 'Cleanliness Ratio',
                    'Disease Awareness Programs', 'Latitude ', 
'Longitude ']]
    y = state_data['Epidemic_Type']
    states = state_data['State']

    # Train-test split, keeping the state column
    X_train, X_test, y_train, y_test, states_train, states_test = 
train_test_split(X, y, states, test_size=0.2, random_state=42)

    # Apply SMOTE with dynamic k_neighbors based on the smallest 
class size
    minority_class_size = y_train.value_counts().min()
    smote = SMOTE(random_state=42, k_neighbors=min(5, 
minority_class_size - 1))

    # Resample the data
    X_resampled, y_resampled = smote.fit_resample(X_train, y_train)

    # Initialize and fit the Gradient Boosting model
    gb_model = GradientBoostingClassifier(random_state=42)
    gb_model.fit(X_resampled, y_resampled)

    # Make predictions on the test set
    y_pred_gb = gb_model.predict(X_test)

    # Create DataFrame with predictions and append to the list
    state_predictions = pd.DataFrame({
        'State': states_test.values,
        'Latitude': X_test['Latitude '].values,
        'Longitude': X_test['Longitude '].values,
        'Predicted_Epidemic_Type': y_pred_gb
    })
    all_predictions.append(state_predictions)

# Concatenate all state predictions into a single DataFrame
predicted_data = pd.concat(all_predictions, ignore_index=True)

# Load geographical data for India
region_map = gpd.read_file("/content/gadm41_IND_1.json")

# Plot the map of India with all predictions
plt.figure(figsize=(12, 10))
base = region_map.plot(color='white', edgecolor='black')
sns.scatterplot(data=predicted_data, x='Longitude', y='Latitude',
hue='Predicted_Epidemic_Type',
                palette='YlOrRd', s=100, edgecolor='k',
legend='full', ax=base)
plt.title("Geographic Distribution of Predicted Epidemic Types in
India")
plt.xlabel("Longitude")
plt.ylabel("Latitude")
plt.legend(title='Epidemic Type')
plt.show()
```

Algorithm Details

1. Objectives

- Classify Disease Types: The goal is to classify the type of disease based on the number of cases and mortality rates in the dataset.
- Data Enrichment: Add a new column to the dataset that indicates the predicted disease type for each entry.

2. Key Components

- Pandas: For data manipulation and analysis.
- Custom Function: A function to determine the disease type based on specific conditions related to cases and mortality rates.
- DataFrame Operations: Use of apply to iterate through the DataFrame rows and assign disease types.

3. Process

1. Load Dataset: Read the epidemic data from a CSV file.
2. Define Classification Logic: Create a function that uses conditional statements to classify the disease type based on the values in the Cases and Mortality Rate (%) columns.
3. Apply Function: Use the apply method to apply the classification function to each row of the DataFrame.
4. Save Modified Data: Write the modified DataFrame with the new disease type column back to a CSV file.
5. Display Relevant Information: Print selected columns from the modified DataFrame for review.

Source Code Details

```
import pandas as pd

# Load the dataset
df = pd.read_csv('/content/epidemic_data_2024.csv')

def assign_disease_type(row):
    if row['Cases'] > 20000:
        return 'COVID-19'
    elif row['Cases'] > 15000 and row['Mortality Rate (%)'] > 4:
        return 'Cholera'
    elif row['Cases'] > 10000 and row['Mortality Rate (%)'] < 4:
        return 'Dengue Fever'
    elif row['Cases'] > 10000 and row['Mortality Rate (%)'] < 2 and
row['Vaccination Rate (%)'] < 50:
        return 'Measles'
    elif row['Cases'] < 2000 and row['Mortality Rate (%)'] > 5:
        return 'Ebola'
    elif row['Cases'] > 1000 and row['Cases'] < 10000 and
row['Mortality Rate (%)'] < 1:
        return 'Zika Virus'
    elif row['Cases'] > 10000 and row['Mortality Rate (%)'] >= 2:
        return 'Tuberculosis (TB)'  # Combined condition for TB
    elif row['Cases'] > 50000:
        return 'HIV/AIDS'
    elif row['Cases'] > 5000 and row['Cases'] < 20000:
        if row['Mortality Rate (%)'] >= 2:
            return 'Hepatitis B'
        else:
            return 'Typhoid Fever'  # Adjusted logic for Typhoid
based on mortality
    elif row['Cases'] >= 2000 and row['Cases'] <= 10000:
        if row['Mortality Rate (%)'] > 5:
            return 'Meningitis'
        else:
            return 'Leptospirosis'  # More inclusive for lower
mortality rates
    elif row['Cases'] >= 1000 and row['Cases'] <= 10000 and
row['Mortality Rate (%)'] > 5:
        return 'Yellow Fever'
    else:
        return 'Flu'

# Apply function to create new column
df['Disease_type'] = df.apply(assign_disease_type, axis=1)

# Save the modified DataFrame to a new CSV file
df.to_csv('modified_epidemic_data_2024.csv', index=False)

# Print selected columns from the modified DataFrame
print(df[['State', 'Date', 'Cases', 'Disease_type']])
```

Algorithm Details

1. Objectives

- Predict Disease Types: The goal is to predict the disease type based on various epidemiological features using a Random Forest model.

- Model Evaluation: Assess the performance of the model through confusion matrices and classification reports.

2. Key Components

- Pandas: For data manipulation and analysis.
- Scikit-learn: For machine learning model training, evaluation, and metrics.
- RandomForestClassifier: The machine learning model used for classification.
- Train-Test Split: To evaluate model performance on unseen data.

3. Process

1. Define Features and Target Variable: Specify the features (X) and target variable (y) for the model.
2. Split Data: Divide the dataset into training and testing sets to evaluate the model's performance.
3. Model Training: Train the Random Forest classifier on the training set.
4. Make Predictions: Use the trained model to make predictions on the test set.
5. Evaluate Model: Print the confusion matrix and classification report to assess model performance.
6. Predict for All States: Use the model to predict disease types for the entire dataset.
7. Display Predictions: Print the predicted disease type for each state.

Source Code Details

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, confusion_matrix

# Assuming df is already defined and contains the necessary features

# Define features and target variable
X = df[['Cases', 'Recoveries', 'Mortality Rate (%)', 'Population',
         'Vaccination Rate (%)', 'Testing Rate (per 1,000)',
         'Infection Rate (%)', 'Healthcare Facilities', 'Average
Age',
         'Urbanization Rate (%)', 'Mobility Index', 'Cleanliness
Ratio',
         'Disease Awareness Programs', 'Latitude ', 'Longitude ']]  #
Features

# Assuming you have a categorical target variable 'Disease_type'
y = df['Disease_type']  # Target variable

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Create and train the Random Forest model
model = RandomForestClassifier(random_state=42)
model.fit(X_train, y_train)
```

```
# Make predictions on the test set
y_pred = model.predict(X_test)

# Evaluate the model
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))
print("\nClassification Report:")
print(classification_report(y_test, y_pred))

# Predict disease type for all entries in the original dataset
df['Predicted Disease Type'] = model.predict(X)

# Display predicted risks for each state
for index, row in df.iterrows():
    print(f"In {row['State']}, the predicted disease type is:
{row['Predicted Disease Type']}.")
```

Algorithm Details

1. Objectives

- Count Disease Occurrences: The goal is to count how many times each disease occurs in each state.
- Identify Most Common Disease: Determine the most frequently reported disease for each state.

2. Key Components

- Pandas: For data manipulation and analysis.
- GroupBy: To aggregate data based on state and disease type.
- idxmax: To identify the index of the maximum count for each state.

3. Process

1. Group Data: Use groupby to count the occurrences of each disease type in each state.
2. Find Most Common Disease: Use idxmax to find the disease with the highest occurrence for each state.
3. Display Results: Print the most common disease along with its count for each state.

Source Code Details

```
import pandas as pd

# Assuming df is your DataFrame containing 'State' and 'Disease_type'
columns

# Count occurrences of each disease in each state
disease_counts = df.groupby(['State',
'Disease_type']).size().reset_index(name='Count')

# Identify the most common disease for each state
most_common_disease =
disease_counts.loc[disease_counts.groupby('State')['Count'].idxmax()]

# Display results
for index, row in most_common_disease.iterrows():
    print(f"In {row['State']}, the most common disease is:
```

```
{row['Disease_type']} with {row['Count']} occurrences.")
```

Algorithm Details

1. Objectives

- Count Disease Occurrences: The goal is to count how many times each disease is predicted for each state.
- Display Results: Print the counts of predicted diseases for each state.

2. Key Components

- defaultdict: To create a dictionary that automatically initializes a Counter for each state.
- Counter: To count occurrences of each disease within each state.
- Looping: Iterate through the predictions to populate the counts and then display the results.

3. Process

1. Initialize Data Structures: Use defaultdict to hold Counter objects for counting diseases.
2. Count Diseases: For each state in the predictions, update the corresponding Counter with the diseases listed.
3. Output Results: Print the counts of each disease for each state.

Source code details

```
from collections import defaultdict, Counter

# Initialize a dictionary to hold counts for each state
state_disease_counts = defaultdict(Counter)

# Define the predictions for each state based on your provided data
predictions = {
    "Andhra Pradesh": [
        "Hepatitis B", "Tuberculosis (TB)", "Dengue Fever",
"Leptospirosis", "Leptospirosis",
        "Dengue Fever", "Dengue Fever", "Tuberculosis (TB)",
"Hepatitis B", "Dengue Fever",
        "Leptospirosis", "Dengue Fever", "Dengue Fever", "Hepatitis
B", "Hepatitis B",
        "Flu", "Hepatitis B", "Leptospirosis", "Dengue Fever",
"Leptospirosis",
        "Hepatitis B", "Hepatitis B", "Hepatitis B", "Hepatitis B",
"Leptospirosis",
        "Leptospirosis", "Dengue Fever", "Hepatitis B", "Hepatitis
B", "Dengue Fever",
        "Dengue Fever", "Dengue Fever", "Dengue Fever", "Dengue
Fever", "Hepatitis B",
        "Tuberculosis (TB)", "Hepatitis B", "Dengue Fever",
"Cholera", "Dengue Fever"
    ],
    "Bihar": [
        "Hepatitis B", "Leptospirosis", "Cholera", "Dengue Fever",
"Cholera",
        "Dengue Fever", "Dengue Fever", "Flu", "Tuberculosis (TB)",
"Hepatitis B",
        "Cholera", "Typhoid Fever", "Dengue Fever", "Typhoid Fever",
"Hepatitis B",
        "Dengue Fever", "Dengue Fever", "Dengue Fever", "Dengue
Fever", "Leptospirosis",
        "Dengue Fever", "Dengue Fever", "Leptospirosis", "Typhoid
Fever", "Hepatitis B",
        "Leptospirosis", "Leptospirosis", "Dengue Fever",
"Leptospirosis", "Hepatitis B",
        "Dengue Fever", "Tuberculosis (TB)", "Leptospirosis",
"Hepatitis B", "Cholera",
        "Tuberculosis (TB)", "Tuberculosis (TB)", "Leptospirosis",
"Dengue Fever",
        "Leptospirosis", "Cholera", "Typhoid Fever", "Leptospirosis",
"Dengue Fever",
        "Flu", "Dengue Fever", "Leptospirosis", "Hepatitis B",
"Leptospirosis",
        "Dengue Fever", "Dengue Fever"

    ],
    "Delhi": [
      "Dengue Fever", "Dengue Fever", "Leptospirosis", "Hepatitis B",
"Dengue Fever",
      "Leptospirosis", "Typhoid Fever", "Dengue Fever", "Dengue
Fever", "Dengue Fever",
      "Dengue Fever", "Typhoid Fever", "Flu", "Dengue Fever",
"Tuberculosis (TB)",
      "Tuberculosis (TB)", "Dengue Fever", "Dengue Fever", "Dengue
Fever", "Leptospirosis",
      "Hepatitis B", "Dengue Fever", "Cholera", "Dengue Fever",
"Hepatitis B",
      "Leptospirosis", "Tuberculosis (TB)", "Dengue Fever", "Dengue
Fever",
      "Tuberculosis (TB)", "Leptospirosis", "Hepatitis B", "Cholera",
"Leptospirosis",
      "Leptospirosis"

    ],
    "Gujarat": [
       "Dengue Fever", "Dengue Fever", "Cholera", "Leptospirosis",
"Dengue Fever",
       "Typhoid Fever", "Dengue Fever", "Dengue Fever",
"Leptospirosis", "Leptospirosis",
       "Leptospirosis", "Leptospirosis", "Cholera", "Dengue Fever",
"Typhoid Fever",
       "Leptospirosis", "Hepatitis B", "Hepatitis B", "Dengue Fever",
"Leptospirosis",
       "Leptospirosis", "Hepatitis B", "Dengue Fever", "Flu", "Dengue
Fever",
       "Dengue Fever", "Dengue Fever", "Dengue Fever", "Dengue
Fever", "Hepatitis B",
       "Leptospirosis", "Cholera", "Hepatitis B", "Dengue Fever",
"Dengue Fever",
       "Dengue Fever", "Dengue Fever", "Dengue Fever"

    ],
    "Haryana": [
        "Hepatitis B", "Dengue Fever", "Dengue Fever", "Dengue
Fever", "Tuberculosis (TB)",
        "Typhoid Fever", "Hepatitis B", "Dengue Fever", "Hepatitis
B", "Typhoid Fever",
        "Hepatitis B", "Dengue Fever", "Dengue Fever",
"Leptospirosis", "Dengue Fever",
        "Dengue Fever", "Dengue Fever", "Flu", "Dengue Fever",
"Hepatitis B",
        "Hepatitis B", "Leptospirosis", "Flu", "Dengue Fever",
"Leptospirosis",
        "Dengue Fever", "Dengue Fever", "Dengue Fever", "Dengue
Fever", "Leptospirosis",
        "Hepatitis B", "Dengue Fever", "Hepatitis B"

    ],
    "Himachal Pradesh": [
       "Hepatitis B", "Leptospirosis", "Leptospirosis", "Dengue
Fever", "Hepatitis B",
       "Typhoid Fever", "Dengue Fever", "Typhoid Fever", "Dengue
Fever", "Hepatitis B",
       "Dengue Fever", "Tuberculosis (TB)", "Hepatitis B", "Flu",
"Leptospirosis",
       "Hepatitis B", "Flu", "Flu", "Tuberculosis (TB)", "Dengue
Fever", "Hepatitis B",
       "Dengue Fever", "Dengue Fever", "Dengue Fever", "Hepatitis B",
"Tuberculosis (TB)",
       "Tuberculosis (TB)", "Cholera", "Typhoid Fever", "Tuberculosis
(TB)",
       "Typhoid Fever", "Hepatitis B", "Dengue Fever", "Dengue
Fever", "Dengue Fever",
       "Cholera", "Leptospirosis", "Dengue Fever", "Leptospirosis",
"Hepatitis B",
       "Cholera", "Typhoid Fever", "Cholera", "Hepatitis B", "Typhoid
Fever",
       "Leptospirosis", "Typhoid Fever", "Dengue Fever", "Cholera",
"Hepatitis B",
       "Dengue Fever", "Hepatitis B"

    ],
    "Karnataka": [
       "Tuberculosis (TB)", "Dengue Fever", "Leptospirosis", "Typhoid
Fever", "Hepatitis B",
       "Dengue Fever", "Dengue Fever", "Hepatitis B",
"Leptospirosis", "Dengue Fever",
       "Typhoid Fever", "Leptospirosis", "Typhoid Fever",
"Leptospirosis", "Hepatitis B",
       "Cholera", "Dengue Fever", "Typhoid Fever", "Hepatitis B",
"Cholera",
       "Hepatitis B", "Flu", "Dengue Fever", "Dengue Fever", "Typhoid
Fever",
       "Dengue Fever", "Hepatitis B", "Leptospirosis", "Dengue
Fever", "Typhoid Fever",
       "Dengue Fever", "Dengue Fever", "Cholera", "Dengue Fever",
"Dengue Fever",
      "Tuberculosis (TB)", "Dengue Fever", "Dengue Fever",
"Tuberculosis (TB)",
      "Tuberculosis (TB)", "Dengue Fever", "Dengue Fever", "Hepatitis
B",
       "Hepatitis B", "Leptospirosis", "Dengue Fever", "Hepatitis B",
       "Tuberculosis (TB)", "Hepatitis B", "Dengue Fever", "Dengue
```

```
Fever",
      "Hepatitis B", "Leptospirosis"

    ],
    "Kerala": [
       "Dengue Fever", "Cholera", "Hepatitis B", "Hepatitis B",
"Tuberculosis (TB)",
       "Dengue Fever", "Dengue Fever", "Leptospirosis", "Dengue
Fever", "Dengue Fever",
       "Cholera", "Leptospirosis", "Hepatitis B", "Dengue Fever",
"Dengue Fever",
       "Dengue Fever", "Hepatitis B", "Leptospirosis", "Dengue
Fever", "Hepatitis B",
      "Leptospirosis", "Cholera", "Hepatitis B", "Leptospirosis",
"Hepatitis B",
      "Hepatitis B", "Dengue Fever", "Dengue Fever", "Dengue Fever",
"Dengue Fever",
      "Hepatitis B", "Hepatitis B", "Leptospirosis", "Leptospirosis",
"Dengue Fever",
      "Dengue Fever", "Hepatitis B", "Tuberculosis (TB)",
"Leptospirosis",
      "Hepatitis B", "Hepatitis B", "Dengue Fever", "Hepatitis B",
      "Hepatitis B", "Dengue Fever"

    ],
    "Maharashtra":[
        "Leptospirosis", "Dengue Fever", "Dengue Fever", "Dengue
Fever", "Leptospirosis",
        "Leptospirosis", "Hepatitis B", "Leptospirosis",
"Tuberculosis (TB)", "Dengue Fever",
        "Leptospirosis", "Leptospirosis", "Dengue Fever", "Cholera",
"Dengue Fever",
        "Hepatitis B", "Tuberculosis (TB)", "Dengue Fever",
"Leptospirosis", "Leptospirosis",
        "Hepatitis B", "Dengue Fever", "Typhoid Fever",
"Leptospirosis", "Leptospirosis",
        "Hepatitis B", "Hepatitis B", "Leptospirosis",
"Leptospirosis", "Tuberculosis (TB)",
        "Leptospirosis", "Hepatitis B", "Hepatitis B", "Dengue
Fever", "Flu",
        "Leptospirosis", "Hepatitis B", "Dengue Fever",
"Leptospirosis", "Cholera",
        "Dengue Fever", "Dengue Fever", "Flu", "Hepatitis B", "Dengue
Fever",
        "Hepatitis B", "Hepatitis B", "Typhoid Fever", "Hepatitis B",
"Dengue Fever",
        "Leptospirosis", "Hepatitis B", "Leptospirosis", "Dengue
Fever", "Dengue Fever",
        "Dengue Fever"

    ],
    "Odisha":[
        "Hepatitis B", "Hepatitis B", "Dengue Fever", "Dengue Fever",
"Leptospirosis",
        "Dengue Fever", "Hepatitis B", "Dengue Fever", "Cholera",
"Dengue Fever",
       "Flu", "Dengue Fever", "Hepatitis B", "Cholera",
"Leptospirosis",
       "Dengue Fever", "Dengue Fever", "Typhoid Fever", "Dengue
Fever", "Hepatitis B",
       "Leptospirosis", "Dengue Fever", "Hepatitis B", "Dengue
Fever", "Dengue Fever",
       "Dengue Fever", "Leptospirosis", "Dengue Fever", "Hepatitis
B", "Cholera",
       "Typhoid Fever", "Dengue Fever", "Dengue Fever",
"Leptospirosis",
       "Cholera", "Leptospirosis", "Dengue Fever", "Dengue Fever",
"Flu",
       "Dengue Fever", "Tuberculosis (TB)", "Hepatitis B", "Dengue
Fever",
       "Dengue Fever", "Dengue Fever"

    ],
    "Punjab":[
        "Hepatitis B", "Leptospirosis", "Hepatitis B", "Tuberculosis
(TB)", "Leptospirosis",
        "Hepatitis B", "Cholera", "Dengue Fever", "Hepatitis B",
"Dengue Fever",
        "Flu", "Dengue Fever", "Cholera", "Leptospirosis", "Dengue
Fever",
        "Flu", "Hepatitis B", "Typhoid Fever", "Typhoid Fever",
"Cholera",
        "Tuberculosis (TB)", "Tuberculosis (TB)", "Leptospirosis",
"Leptospirosis",
       "Dengue Fever", "Cholera", "Leptospirosis", "Flu", "Dengue
Fever",
       "Typhoid Fever", "Cholera", "Tuberculosis (TB)"

    ],
    "Rajasthan":[
        "Dengue Fever", "Dengue Fever", "Typhoid Fever", "Dengue
Fever", "Dengue Fever",
        "Dengue Fever", "Dengue Fever", "Typhoid Fever", "Typhoid
Fever", "Hepatitis B",
        "Hepatitis B", "Tuberculosis (TB)", "Flu", "Flu", "Hepatitis
B",
        "Hepatitis B", "Leptospirosis", "Hepatitis B", "Dengue
Fever", "Cholera",
        "Dengue Fever", "Typhoid Fever", "Typhoid Fever", "Dengue
Fever", "Flu",
        "Dengue Fever", "Cholera", "Hepatitis B", "Typhoid Fever",
"Hepatitis B",
        "Hepatitis B", "Leptospirosis", "Typhoid Fever",
"Leptospirosis",
        "Hepatitis B", "Dengue Fever", "Hepatitis B", "Dengue Fever",
"Hepatitis B",
        "Dengue Fever", "Dengue Fever"

    ],
    "Tamil Nadu":[
        "Dengue Fever", "Tuberculosis (TB)", "Leptospirosis",
"Leptospirosis", "Leptospirosis",
        "Dengue Fever", "Dengue Fever", "Dengue Fever", "Dengue
Fever", "Dengue Fever",
        "Dengue Fever", "Leptospirosis", "Cholera", "Hepatitis B",
"Leptospirosis",
        "Hepatitis B", "Dengue Fever", "Hepatitis B",
"Leptospirosis", "Leptospirosis",
        "Typhoid Fever", "Flu", "Tuberculosis (TB)", "Dengue Fever",
"Tuberculosis (TB)",
        "Hepatitis B", "Hepatitis B", "Dengue Fever",
"Leptospirosis", "Dengue Fever",
        "Tuberculosis (TB)", "Typhoid Fever", "Dengue Fever",
"Tuberculosis (TB)",
        "Leptospirosis", "Typhoid Fever", "Dengue Fever",
"Tuberculosis (TB)",
        "Hepatitis B", "Hepatitis B"

    ],
    "Telegana":[
        "Leptospirosis", "Dengue Fever", "Flu", "Dengue Fever",
"Dengue Fever",
        "Hepatitis B", "Dengue Fever", "Dengue Fever", "Dengue
Fever", "Dengue Fever",
        "Dengue Fever", "Leptospirosis", "Hepatitis B", "Hepatitis
B", "Typhoid Fever",
        "Leptospirosis", "Cholera", "Dengue Fever", "Dengue Fever",
"Hepatitis B",
        "Leptospirosis", "Flu", "Leptospirosis", "Dengue Fever",
"Dengue Fever",
        "Hepatitis B", "Dengue Fever", "Dengue Fever", "Hepatitis B",
"Typhoid Fever",
        "Dengue Fever", "Dengue Fever", "Dengue Fever", "Hepatitis
B", "Tuberculosis (TB)",
        "Dengue Fever", "Leptospirosis", "Hepatitis B", "Hepatitis
B", "Dengue Fever",
        "Hepatitis B", "Dengue Fever"

    ],
    "Uttar Pradesh":[
        "Cholera", "Hepatitis B", "Dengue Fever", "Leptospirosis",
"Dengue Fever",
        "Leptospirosis", "Dengue Fever", "Hepatitis B", "Dengue
Fever", "Dengue Fever",
        "Dengue Fever", "Dengue Fever", "Cholera", "Leptospirosis",
"Dengue Fever",
        "Typhoid Fever", "Hepatitis B", "Dengue Fever", "Cholera",
"Leptospirosis",
        "Dengue Fever", "Hepatitis B", "Hepatitis B", "Hepatitis B",
"Dengue Fever",
        "Dengue Fever", "Typhoid Fever", "Tuberculosis (TB)",
"Typhoid Fever",
        "Typhoid Fever", "Leptospirosis", "Hepatitis B", "Flu",
"Dengue Fever",
        "Hepatitis B", "Dengue Fever",

    ],
    "Uttarakhand":[
        "Dengue Fever", "Hepatitis B", "Dengue Fever", "Hepatitis B",
"Dengue Fever",
        "Dengue Fever", "Dengue Fever", "Dengue Fever", "Dengue
Fever", "Dengue Fever",
        "Leptospirosis", "Hepatitis B", "Leptospirosis", "Dengue
Fever", "Typhoid Fever",
        "Dengue Fever", "Dengue Fever", "Cholera", "Dengue Fever",
"Dengue Fever",
        "Flu", "Leptospirosis", "Dengue Fever", "Dengue Fever",
"Typhoid Fever",
        "Hepatitis B", "Hepatitis B", "Dengue Fever", "Cholera",
"Dengue Fever",
        "Leptospirosis", "Dengue Fever", "Hepatitis B", "Dengue
Fever", "Hepatitis B",
        "Dengue Fever", "Leptospirosis", "Leptospirosis", "Dengue
Fever", "Typhoid Fever",
        "Dengue Fever", "Dengue Fever", "Dengue Fever"

    ],
    "West Bengal":[
        "Leptospirosis", "Typhoid Fever", "Hepatitis B",
"Leptospirosis", "Typhoid Fever",
        "Typhoid Fever", "Dengue Fever", "Leptospirosis",
"Leptospirosis", "Flu",
        "Dengue Fever", "Dengue Fever", "Leptospirosis",
"Tuberculosis (TB)", "Hepatitis B",
        "Dengue Fever", "Tuberculosis (TB)", "Hepatitis B",
"Leptospirosis", "Cholera",
        "Dengue Fever", "Dengue Fever", "Cholera", "Dengue Fever",
"Dengue Fever",
        "Leptospirosis", "Flu", "Dengue Fever", "Hepatitis B",
"Leptospirosis",
        "Cholera", "Flu", "Leptospirosis", "Dengue Fever", "Dengue
Fever",
        "Dengue Fever", "Hepatitis B", "Dengue Fever"

    ]

}

# Count occurrences of each disease for each state
for state, diseases in predictions.items():
    state_disease_counts[state] = Counter(diseases)

# Output the results
for state, diseases in state_disease_counts.items():
    print(f"\nIn {state}, the predicted diseases are:")
    for disease, count in diseases.items():
        print(f"{disease}: {count}")
```