PRESIDENCY UNIVERSITY

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

BANGALORE

IMPLEMENTATION 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.

IMPLEMENTATION DETAILS

This implementation guide provides a comprehensive overview of the various algorithms and processes used to analyze and visualize epidemic data. By following these steps, you can effectively leverage data science techniques to gain insights into epidemic trends, forecast future cases, and classify epidemic types based on historical data. Each section outlines the necessary libraries, data sources, and specific code snippets to execute the tasks, making it easier to replicate and adapt for specific needs.
This comprehensive implementation guide provides step-by-step instructions for classifying epidemic types, evaluating model performance, predicting disease types, and counting disease occurrences. Each section includes code snippets and explanations, making it easier to replicate and adapt the processes for specific needs. By following these steps, you can effectively leverage data science techniques to gain insights into epidemic trends and improve public health decision-making.

Visualizing Epidemic Data with Heatmaps

Objective: Visualize geographical areas with high concentrations of reported cases using heatmaps.

Key Components:

- **Libraries:** Folium, Pandas
- **Data Source:** CSV file containing epidemic data.

Process:

Load Data:

```
import pandas as pd

data = pd.read_csv("/content/epidemic_data_2024.csv")
```

Extract Relevant Columns:

```
cols = data[['Latitude', 'Longitude', 'Cases']]
```

Initialize Folium Map:

```
import folium

mapObj = folium.Map(location=[24.2170, 81.0791], zoom_start=5)
```

Create Heatmap:

```
from folium.plugins import HeatMap

HeatMap(cols).add_to(mapObj)
```

Save the Map:

```
mapObj.save("epidemic_heatmap.html")
```

Animating Epidemic Data Over Time

Objective: Create an animated GIF showing total cases for each date.

Key Components:

- **Libraries:** Geopandas, Matplotlib, PIL (Pillow)

- **Data Source:** CSV file and geographical shape data.
-

Process:

Load Data:

```
import pandas as pd
import geopandas as gpd

epidemic_data = pd.read_csv('/content/epidemic_data_2024_test.csv')
india_shape = gpd.read_file('/content/gadm41_IND_1.json')
```

Merge Datasets:

```
merged_data = india_shape.merge(epidemic_data, left_on='NAME_1',
right_on='State', how='left')
```

Create Plots for Each Date:

```
import matplotlib.pyplot as plt

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)
    ax.set_title(f'Real Time Mapping of Epidemic\nTotal Cases on
{date}')
    plt.close(fig)
```

Compile Images into GIF:

```
from PIL import Image

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

Forecasting Future Daily Cases Using ARIMA

Objective: Forecast future cases for a specific state using the ARIMA model.

Key Components:
- **Libraries:** Pandas, NumPy, Statsmodels
- **Data Source:** CSV file containing historical epidemic data.

Process:

Load and Prepare Data:

```
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.arima.model import ARIMA
import matplotlib.pyplot as plt

data = pd.read_csv('/content/epidemic_data_2024.csv',
parse_dates=['Date'], index_col='Date')
```

```
state_data = data[data['State'] == 'Uttar Pradesh']
daily_cases = state_data.resample('D')['Cases'].sum()
```

Stationarity Check:

```
result = adfuller(daily_cases)
```

Fit ARIMA Model:

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

Forecast Future Cases:

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

Plot Results:

```
plt.plot(daily_cases, label='Observed Cases')
plt.plot(forecast.index, forecast, label='Forecasted Cases',
color='red')
plt.legend()
plt.show()
```

Analyzing Relationships Using VAR Model

Objective: Analyze relationships between epidemic cases across states using the VAR model.

Key Components:

- **Libraries:** Pandas, Statsmodels
- **Data Source:** CSV file containing epidemic data.

Process:

Load and Pivot Data:

```
from statsmodels.tsa.api import VAR

pivot_data = data.pivot_table(index='Date', columns='State',
values='Cases', aggfunc='sum')
```

Check Stationarity:

```
for state in pivot_data.columns:
    if not check_stationarity(pivot_data[state]):
        pivot_data[state] = pivot_data[state].diff().dropna()
```

Fit VAR Model:

```
model = VAR(pivot_data)
model_fit = model.fit(maxlags=5, ic='aic')
```

Make Forecasts:

```
forecast = model_fit.forecast(pivot_data.values[-model_fit.k_ar:],
```

```
steps=10)
```

Visualize Observed vs. 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.legend()
plt.show()
```

Predicting Future Epidemic Cases Using Random Forest

Objective: Predict future epidemic cases for different states using the Random Forest model.

Key Components:

- **Libraries:** Pandas, Scikit-learn
- **Data Source:** CSV file containing epidemic data.

Process:

Load Data:

```
import pandas as pd

data = pd.read_csv('/content/epidemic_data_2024.csv',
parse_dates=['Date'])
```

Create Lagged Features:

```
def create_features(df, n_lags=3):
    for lag in range(1, n_lags + 1):
        df[f'lag_{lag}'] = df['Cases'].shift(lag)
    return df.fillna(0)
```

Split Data:

```
from sklearn.model_selection import train_test_split

feature_data = create_features(data)
X = feature_data.drop(['Cases'], axis=1)
y = feature_data['Cases']

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

Train Random Forest Model:

```
from sklearn.ensemble import RandomForestRegressor

model = RandomForestRegressor(n_estimators=100)
model.fit(X_train, y_train)
```

Evaluate Model:

```
from sklearn.metrics import mean_squared_error, r2_score

predictions = model.predict(X_test)
```

```
mse = mean_squared_error(y_test, predictions)
r2 = r2_score(y_test, predictions)
```

Visualize Results:

```
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 6))
plt.plot(y_test.values, label='Actual Cases')
plt.plot(predictions, label='Predicted Cases', color='orange',
linestyle='--')
plt.legend()
plt.show()
```

Classifying Epidemic Types Using XGBoost

Objective: Classify epidemic types based on various features using the XGBoost classifier.

Key Components:

- **Libraries:** Pandas, XGBoost, Scikit-learn
- **Data Source:** CSV file containing epidemic data.

Process:

Load Data and Create Epidemic Type Column:

```
import pandas as pd
import numpy as np

data = pd.read_csv('/content/epidemic_data_2024.csv')
conditions = [data['Cases'] < 100, (data['Cases'] >= 100) &
(data['Cases'] < 500), data['Cases'] >= 500]
choices = ['Low', 'Moderate', 'High']
data['Epidemic_Type'] = np.select(conditions, choices,
default='Unknown')
X = data[['Cases', 'Recoveries', 'Mortality Rate (%)', ...]]
y = data['Epidemic_Type']
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
```

1. Apply SMOTE for Imbalance:

```
from imblearn.over_sampling import SMOTE
smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X_train, y_train)
```

2. Train XGBoost Model:

```
import xgboost as xgb
model = xgb.XGBClassifier()
model.fit(X_resampled, y_resampled)
```

3. Evaluate Model Performance:

```
from sklearn.metrics import classification_report, confusion_matrix
y_pred = model.predict(X_test)
print(classification_report(y_test, y_pred))
```

4. Visualize Predictions:

```
cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, fmt="d", cmap="YlGnBu")
```

Implementation Details

1. Classifying Epidemic Types

Objectives:

- **Classify Epidemic Types:** 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.

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:** For implementing the classification model.
- **Matplotlib & Seaborn:** For data visualization.

Process:

1. **Load the Dataset:**

```
import pandas as pd
data = pd.read_csv('epidemic_data.csv')  # Adjust the path as
necessary
```

2. **Create 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)
```

3. **Visualize Class Distribution Before SMOTE:**

```
import matplotlib.pyplot as plt
import seaborn as sns
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()
```

4. **Loop Through Each State:**

```
from imblearn.over_sampling import SMOTE
```

```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingClassifier
resampled_data = []
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
    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])
```

5. **Convert Resampled Data to DataFrame:**

```
resampled_df = pd.DataFrame(resampled_data, columns=['State',
'Epidemic_Type'])
```

6. **Visualize Class Distribution After SMOTE:**

```
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()
```

7. **Train the Gradient Boosting Model:**

```
model = GradientBoostingClassifier(random_state=42)
model.fit(X_resampled, y_resampled)
```

8. **Evaluate the Model:**

```
y_pred = model.predict(X_test)
from sklearn.metrics import classification_report, accuracy_score
print("Accuracy:", accuracy_score(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

---

2. Evaluating Model Performance with Cross-Validation

Objectives:

- **Evaluate Model Performance:** 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.

Key Components:

- **StratifiedKFold:** A cross-validation strategy that maintains the proportion of classes in each fold.

Process:

1. **Load the Dataset:** Same as above.

2. **Create Epidemic_Type Column:** Same as above.

3. **Initialize a Dictionary to Store Cross-Validation Scores:**

```
cv_scores_dict = {}
```

4. **Loop Through Each 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']
    # 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)
```

5. **Convert Scores to DataFrame for Visualization:**

```
cv_df = pd.DataFrame(cv_scores_dict).melt(var_name='State',
value_name='Cross-Validation Score')
```

6. **Visualize 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()
```

---

3. Visualizing Model Predictions with Confusion Matrices

Objectives:

- **Evaluate Model Predictions:** Visualize the performance of the Gradient Boosting classifier by displaying a confusion matrix for each state.
- **Understand Misclassifications:** Identify how well the model predicts each class and where it makes errors.

Process:

1. **Load the Dataset:** Same as above.

2. **Create Epidemic_Type Column:** Same as above.

3. **Loop Through Each 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']

    # 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
    from sklearn.metrics import 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()
```

4.Evaluating Feature Contribution

Objectives:

- **Evaluate Feature Contribution:** Visualize the importance of each feature in the Gradient Boosting model.
- **Inform Feature Selection:** Guide decisions on which features to keep or remove.

Process:

1. **Calculate Feature Importances:**

```
importances = gb_model.feature_importances_
feature_names = X.columns
```

2. **Prepare Data for Visualization:**

```
feature_importance_df = pd.DataFrame({'Feature': feature_names,
'Importance': importances})
feature_importance_df =
feature_importance_df.sort_values(by='Importance', ascending=False)
```

3. **Visualize 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()
```

---

5. Predicting Disease Types

Objectives:

- **Predict Disease Types:** 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.

Process:

1. **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']]
y = df['Disease_type']
```

2. **Split Data:**

```
from sklearn.model_selection import train_test_split

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

3. **Model Training:**

```
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier(random_state=42)
model.fit(X_train, y_train)
```

4. **Make Predictions:**

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

5. **Evaluate Model:**

```
from sklearn.metrics import classification_report, confusion_matrix

print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

6. **Predict for All States:**

```
df['Predicted Disease Type'] = model.predict(X)
```

7. **Display Predictions:**

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

6.Counting Disease Occurrences

Objectives:

- **Count Disease Occurrences:** Count how many times each disease occurs in each state.
- **Identify Most Common Disease:** Determine the most frequently reported disease for each state.

Process:

1. **Group Data:**

```
disease_counts = df.groupby(['State',
'Disease_type']).size().reset_index(name='Count')
```

2. **Find Most Common Disease:**

```
most_common_disease =
disease_counts.loc[disease_counts.groupby('State')['Count'].idxmax()]
```

3. **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.")
```

7.Counting Predicted Disease Occurrences

Objectives:

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

Process:

1. **Initialize Data Structures:**

```
from collections import defaultdict, Counter
state_disease_counts = defaultdict(Counter)
```

2. **Count Diseases:**

```
predictions = {
    "Andhra Pradesh": ["Hepatitis B", "Tuberculosis (TB)", ...],  #
Add all predicted diseases here
    "Bihar": ["Hepatitis B", "Leptospirosis", ...],
    # Add other states...
}

for state, diseases in predictions.items():
    state_disease_counts[state] = Counter(diseases)
```

3. **Output 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}")
```