**ABSTRACT**

This report investigates the application of machine learning techniques, specifically Gradient Boosting, to classify epidemic types based on various health-related features. Utilizing a dataset encompassing multiple states, the study categorizes epidemic severity into three types: Low, Moderate, and High, determined by case counts. The methodology includes data preprocessing steps such as handling class imbalance using SMOTE (Synthetic Minority Over-sampling Technique) and employing cross-validation for robust model evaluation. The performance of the Gradient Boosting classifier is assessed through metrics such as accuracy, confusion matrix, and classification report. Results indicate the model's effectiveness in predicting epidemic types, providing valuable insights for public health decision-making and resource allocation.

**INTRODUCTION**

Epidemic outbreaks pose significant challenges to public health systems worldwide, necessitating timely and accurate classification of their severity to inform response strategies. The ability to categorize epidemic types based on various health metrics can enhance resource allocation, improve public health interventions, and facilitate more effective communication with stakeholders.

In recent years, machine learning techniques have emerged as powerful tools for analyzing complex datasets in epidemiology. Among these techniques, Gradient Boosting has gained popularity due to its high predictive accuracy and robustness against overfitting. This report explores the use of Gradient Boosting to classify epidemic types using a dataset that includes critical variables such as case counts, recoveries, mortality rates, and vaccination rates across different states.

The study begins with data preprocessing, including the creation of an Epidemic\_Type variable based on case distributions and the application of SMOTE to address class imbalance. The model's performance is evaluated through cross-validation, ensuring a comprehensive assessment of its predictive capabilities. By leveraging machine learning, this research aims to contribute to the understanding of epidemic dynamics and support public health initiatives in managing outbreaks effectively.

**ALGORITHM DETAILS**

1. **Objective:**
   * The goal of the code is to visualize epidemic data as a heatmap, which helps in identifying geographical areas with high concentrations of reported cases.
2. **Key Components:**
   * **Folium Library:** Used for creating interactive maps.
   * **Pandas Library:** Used for data manipulation and analysis, particularly for handling the CSV file containing epidemic data.
   * **HeatMap Plugin:** A Folium plugin that allows for creating heatmaps based on the density of data points.
3. **Process:**
   * Load the epidemic data from a CSV file into a Pandas DataFrame.
   * Extract relevant columns (latitude, longitude, and number of cases) for heatmap generation.
   * Initialize a Folium map centered at specific coordinates with a defined zoom level.
   * Create a heatmap using the extracted data and add it to the map object.
   * 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

# Create a map object centered at specified latitude and longitude

mapObj = folium.Map(location=[24.2170111233401, 81.0791015625000], zoom\_start=5)

# Load epidemic data from a CSV file into a DataFrame

data = pd.read\_csv("/content/epidemic\_data\_2024.csv")

# Display the first few rows of the DataFrame

data.head()

# Print the columns of the DataFrame to understand its structure

print(data.columns)

# Extract relevant columns: Latitude, Longitude, and Cases

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

# Create a heatmap using the extracted data and add it to the map object

HeatMap(cols).add\_to(mapObj)

# Save the map object to an HTML file

mapObj.save("epidemic\_heatmap.html")

**ALGORITHM DETAILS**

1. **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.
2. **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.
   * **PIL (Pillow):** Used for image processing and creating GIFs.
3. **Process:**
   * Load epidemic data and geographical shape data for India.
   * Merge the datasets based on state names.
   * Fill missing values in the merged data with zeros.
   * Create a plot for each date, visualizing the number of cases for each state.
   * Save each plot as a PNG image in memory.
   * 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

# Load epidemic data from a CSV file into a DataFrame

epidemic\_data = pd.read\_csv('/content/epidemic\_data\_2024\_test.csv')

# Display the columns of the epidemic data

epidemic\_data.columns

# Load geographical shape data for India

india\_shape = gpd.read\_file('/content/gadm41\_IND\_1.json')

print(india\_shape.columns)

print(india\_shape.head())

# Merge the geographic data with the epidemic data based on state names

merged\_data = india\_shape.merge(epidemic\_data, left\_on='NAME\_1', right\_on='State', how='left')

# Fill missing values in the merged data with zeros

for date in epidemic\_data.columns[1:]: # Assuming first column is state names

merged\_data[date].fillna(0, inplace=True)

image\_frames = []

# Loop through date columns (from column 2 onwards in your dataset)

for date in epidemic\_data.columns[1:]:

# Create a new figure for each frame

fig, ax = plt.subplots(figsize=(15, 15))

# Plot the map for this date

merged\_data.plot(column=date, cmap='Reds', legend=True, ax=ax,

edgecolor='black', linewidth=0.5)

# Set the title and axis off

ax.set\_title(f'Real Time Mapping of Epidemic\nTotal Cases on {date}', fontsize=20)

ax.set\_axis\_off()

# Save each frame as a PNG image in memory

buf = io.BytesIO()

plt.savefig(buf, format='png', bbox\_inches='tight')

buf.seek(0)

image\_frames.append(Image.open(buf)) # Append to the list of images

# Close the plot to prevent overlaps

plt.close()

# Save all frames as a GIF

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**

1. **Objective:**
   * The goal is to forecast future daily epidemic cases for a specific state using the ARIMA (AutoRegressive Integrated Moving Average) time series model.
2. **Key Components:**
   * **Pandas:** For data manipulation and analysis.
   * **NumPy:** For numerical operations.
   * **Matplotlib:** For plotting data visualizations.
   * **Statsmodels:** For statistical modeling, including ARIMA.
3. **Process:**
   * Load the epidemic data from a CSV file and parse dates.
   * Filter the data for a specific state and resample it to get daily case counts.
   * Visualize the daily cases over time.
   * Check for stationarity using the Augmented Dickey-Fuller test (ADF).
   * If the data is non-stationary, apply differencing.
   * Generate ACF (Autocorrelation Function) and PACF (Partial Autocorrelation Function) plots to determine ARIMA parameters (p, d, q).
   * Fit the ARIMA model using the identified parameters.
   * Forecast future cases and visualize the results.

**SOURCE 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

# Load your dataset

data = pd.read\_csv('/content/epidemic\_data\_2024.csv', parse\_dates=['Date'], index\_col='Date')

# Filter for a specific state

state\_name = 'Uttar Pradesh' # Replace with the state you're interested in

state\_data = data[data['State'] == state\_name]

# Group by date and sum cases

daily\_cases = state\_data.resample('D')['Cases'].sum()

# Plot the data

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()

# Check for stationarity

result = adfuller(daily\_cases)

print('ADF Statistic:', result[0])

print('p-value:', result[1])

# If not stationary, differencing may be needed

daily\_cases\_diff = daily\_cases.diff().dropna()

# ACF and PACF plots

plot\_acf(daily\_cases\_diff)

plot\_pacf(daily\_cases\_diff)

plt.show()

# Fit ARIMA model (replace p, d, q with identified values)

p, d, q = 1, 1, 1 # Example values; use your analysis to determine these

model = ARIMA(daily\_cases, order=(p, d, q))

model\_fit = model.fit()

# Make predictions

forecast = model\_fit.forecast(steps=10) # Forecast the next 10 days

print(forecast)

# Plot the 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**

1. **Objective:**
   * The goal is to analyze the relationships between epidemic cases across different states using the VAR model, which captures linear interdependencies among multiple time series.
2. **Key Components:**
   * **Pandas:** For data manipulation and analysis.
   * **NumPy:** For numerical operations.
   * **Matplotlib:** For data visualization.
   * **Statsmodels:** For statistical modeling, including the VAR model.
3. **Process:**
   * Load epidemic data from a CSV file and parse dates.
   * Pivot the data to create a multivariate time series format.
   * Check for stationarity of each state’s time series and apply differencing if necessary.
   * Fit the VAR model to the data.
   * Make forecasts for the specified number of future steps.
   * 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

# Load your dataset

data = pd.read\_csv('/content/epidemic\_data\_2024.csv', parse\_dates=['Date'], index\_col='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)

# 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}")

**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}")