Project Report on

DisasterShield: Disaster Relief Optimizer

Submitted in partial fulfillment of completion of the course

State Level Hackathon 2025

Submitted by:

Pratiksha Ashok Mohite

Smruti Shripad Deshpande

Under Guidance of:

Dr. Puja Padiya

Mr. Yogesh Raje

edunet foundation

Year 2025

Abstract

Acknowledgement

Team Composition and Workload Division

Table of Contents

Abstract

DisasterShield is a groundbreaking platform designed to transform disaster relief operations for weather-related calamities like floods and earthquakes. Leveraging IBM Watson's AutoAI, Snap ML, and IBM Cloud, it delivers cutting-edge predictive models and real-time insights to empower relief teams with actionable data. By analyzing rainfall patterns, river discharge, and water levels, the system uses logistic regression to predict floods, while Random Forest models provide accurate earthquake magnitude estimates based on geophysical data like depth and location. Integrated with a dynamic Streamlit interface and real-time tracking via IBM Cloud services, DisasterShield bridges the gap between data and decision-making. Its ability to optimize resource allocation and improve preparedness sets a new benchmark for disaster management, ensuring faster responses, reduced impact, and more resilient communities.

Acknowledgement

We would like to express our heartfelt gratitude to all those who contributed to the successful completion of the DisasterShield: Disaster Relief Optimizer project. Firstly, we extend our sincere thanks to Mr. Yogesh Raje Sir and Dr Puja Padiya Ma'am for their invaluable guidance, support, and encouragement throughout the project. Their expertise and constructive feedback were instrumental in shaping our approach and achieving our objectives. We would like to extend our heartfelt gratitude to our mentor and the IBM team for providing us with this incredible opportunity to showcase our skills and creativity. Their guidance, support, and access to IBM's cutting-edge platforms have been instrumental in bringing this vision to life. DisasterShield stands as a testament to the power of collaboration, innovation, and technology in building a safer, more resilient future.

Team Composition and Workload Division

The **DisasterShield project** was a collaborative effort undertaken by a team of two core members, guided by two experienced mentors. The division of workload was thoughtfully planned, ensuring equal contribution while leveraging each member's strengths.

1. **Smruti Deshpande and Pratiksha Mohite**:
    - We worked collaboratively on all aspects of the project, from data collection and preprocessing to model development, integration, and deployment.
    - Tasks were divided based on priority and timelines, with regular brainstorming sessions on Microsoft Teams to align on project goals.
    - Responsibilities included:
        - Designing and implementing predictive models for floods and earthquakes using IBM Watson AutoAI and Snap ML.
        - Developing a user-friendly Streamlit interface for disaster relief teams.
        - Setting up real-time data tracking and visualization using IBM Cloud services.
        - Testing and fine-tuning the models for better accuracy and performance.
2. **Dr. Puja Padiya and Mr. Yogesh Raje (Mentors):**
    - Our mentors provided valuable guidance throughout the project, ensuring we stayed on track and maintained high-quality deliverables.
    - They offered feedback on technical approaches, reviewed our progress, and helped troubleshoot complex challenges.

1. Introduction to Problem

Natural disasters, such as floods and earthquakes, are growing in frequency and intensity due to climate change and rapid urbanization, leaving millions of lives at risk each year. In 2024, over **1.2 million people in India** were impacted by floods, with nearly **1,500 fatalities** reported during the monsoon season. Similarly, earthquake-prone zones across the country continue to face the challenge of unanticipated seismic activities, causing devastation to infrastructure and livelihoods. These disasters not only result in loss of life but also create ripple effects, including displacement, resource shortages, and economic setbacks.

Despite advancements in technology, disaster management operations often struggle with inefficiencies like delayed responses, uneven resource distribution, and an overreliance on reactive measures. Floods, driven by excessive rainfall and rising river levels, frequently catch low-lying areas unprepared, while earthquakes demand precise magnitude predictions to prioritize rescue efforts and mitigate damage. Existing systems lack the predictive power and scalability required to address these challenges effectively.

DisasterShield emerges as a game-changing solution to bridge these gaps. Built on state-of-the-art AI technologies like IBM Watson AutoAI, Snap ML, and IBM Cloud, it integrates real-time environmental data with advanced predictive models. This platform not only forecasts disasters with precision but also enables optimal resource allocation and streamlined relief efforts. Its flood model analyzes parameters like rainfall, water levels, and historical flood patterns, while the earthquake model predicts magnitudes based on geophysical data like depth and location.

Equipped with a user-friendly Streamlit interface, DisasterShield empowers relief agencies with actionable insights in real-time. By leveraging IBM Cloud for seamless data storage, tracking, and visualization, it ensures that decision-makers have the tools they need to act swiftly and effectively. With its innovative approach to disaster-specific modeling, predictive analytics, and real-time resource tracking, DisasterShield is poised to redefine disaster management and improve resilience in the face of nature's uncertainties.

2. Proposed Solution

The proposed solution, **DisasterShield**, is a disaster relief optimization system designed to address the challenges of effective disaster management. By leveraging advanced tools like IBM Watson AutoAI, Snap ML, and IBM Cloud, the system aims to provide accurate predictions and enable efficient resource allocation for disaster response.

Key components of the solution include:

1. Flood Prediction Model:

- Utilizes a logistic regression model built with Snap ML.
- Analyzes critical parameters such as rainfall, river discharge, water levels, and historical flood data to predict the likelihood of flooding.
- Provides timely alerts for proactive resource planning and allocation.

2. Earthquake Magnitude Estimation:

- Employs a Random Forest model from Snap ML to estimate earthquake magnitudes.
- Considers geophysical factors like depth, latitude, and longitude for precise predictions.
- Supports rapid response strategies for mitigating earthquake impacts.

3. User-Friendly Interface:

- Developed using Streamlit, the interface provides an intuitive platform for interacting with the system.
- Offers visual insights, prediction results, and actionable recommendations for disaster management teams.

3.Requirements

3.1 Technology Stack

- **IBM Cloud Services**:
    - **IBM Watson AutoAI**: Used for automating the AI model development process, including model selection, training, and optimization.
    - **Snap ML**: A high-performance machine learning library utilized for building flood prediction and earthquake magnitude estimation models.
    - **IBM Cloud**: Provides scalable cloud infrastructure for real-time data storage, tracking, and visualization. Ensures smooth integration with disaster relief operations.
    - **IBM Watson Studio**: Used for training, validating, and deploying machine learning models, ensuring easy collaboration and model management.
- **Other Services**:
    - **Streamlit**: A framework for building interactive web applications, used to develop the user-friendly interface for DisasterShield.
    - **Python**: The primary programming language for developing the backend models, integrating machine learning, and processing data.
    - **Pandas**: Utilized for data manipulation and analysis, especially for handling datasets related to flood and earthquake predictions.
    - **Scikit-learn**: Used for model training, preprocessing, and evaluation.

3.2 Hardware

- **IBM Cloud Virtual Machines**: Used for hosting the solution, running machine learning models, and storing disaster-related data. The virtual

machines offer flexible scaling based on the system's processing and storage needs.

- **Workstations/Laptops**: Required for the development of the system and initial training of models. Minimum specifications include a CPU with multi-core processing, 8GB RAM, and a stable internet connection for cloud integration.

3.3 Software

- **Python 3.12**: Programming language used for developing the backend models, data processing, and AI integration.
- **IBM Watson Studio**: For model development, training, and optimization on the IBM Cloud platform.
- **Streamlit**: For building the frontend user interface.
- **Scikit-learn**: For building and evaluating machine learning models (Random Forest, Logistic Regression).
- **Pandas, NumPy**: For data preprocessing and manipulation.
- **Matplotlib, Seaborn**: For visualizing the results and insights derived from the models.

3.4 Deployment Environment

- **IBM Cloud**:
    - The solution will be deployed and hosted on IBM Cloud, where the machine learning models will run on virtual machines and leverage cloud storage.
    - IBM Cloud ensures scalability, real-time monitoring, and centralized access to the disaster-related data.
- **Streamlit**:
    - The Streamlit application will be deployed on the cloud to provide users with an interactive web interface for disaster predictions and real-time resource tracking.

    - Deployment through IBM Cloud ensures availability and responsiveness of the application.
- **Version Control**:

    - **Git/GitHub**: For managing the codebase, version control, and collaborative development.

4. User Requirements

1. Predictive Analytics

- **Flood Prediction**:
    - Users need a reliable system to predict the likelihood of floods based on historical and real-time data (rainfall, river discharge, water levels, etc.).
    - Predictions should be accurate, timely, and easy to interpret for early intervention planning.
- **Earthquake Magnitude Estimation**:
    - Users require the ability to estimate earthquake magnitudes based on geophysical parameters such as depth, location, and seismic activity.
    - The system must provide real-time earthquake predictions to aid in preparedness and response.

2. User Interface and Accessibility

- **Streamlined User Interface**:
    - Users need an intuitive, easy-to-use interface to interact with the system.
    - The platform should be accessible from various devices (desktop, tablet, mobile) to ensure flexibility during disaster response operations.
    - The interface must display visualizations such as graphs and maps for better understanding of the predictions and real-time data.

3. Scalability and Performance

- The system must be able to scale effectively with growing amounts of disaster-related data, especially during periods of intense activity (e.g., during large storms or earthquakes).
- It should perform consistently, with minimal latency in predictions and real-time data tracking.

4. Resource Allocation Optimization

- Users need a system that not only predicts disasters but also provides actionable insights for optimizing resource allocation.
- The system should recommend the most critical resources to deploy (e.g., medical teams, food, water, equipment) based on real-time disaster predictions and affected areas.

5. System Reliability and Availability

- The system should be highly reliable, with minimal downtime, especially during disaster events.
- The solution should be hosted on a scalable cloud infrastructure (IBM Cloud) that ensures high availability and redundancy in case of failure.

6. Easy Deployment and Maintenance

- The system should be easy to deploy and update with minimal technical overhead.
- Regular updates and maintenance should be supported to improve accuracy, incorporate new data sources, and refine models.

5. Implementation Details

1. **Data Collection and Preparation**
    - **Flood Dataset**: Historical flood data including rainfall (mm), river discharge (m^3/s), water levels (m), and previous flood occurrences.
    - **Earthquake Dataset**: Geophysical parameters such as latitude, longitude, depth, magnitude station count (magNst), and earthquake magnitudes.
    - Data was collected from trusted sources like the Indian Meteorological Department (IMD), Central Water Commission (CWC), and seismic monitoring agencies.
    - Preprocessing involved cleaning missing values, normalizing features, and creating training and testing datasets.
2. **Model Development**
    - **Flood Prediction Model**:
        - Algorithm: **Logistic Regression (Snap ML)**.
        - Inputs: Rainfall, river discharge, water level, and historical flood data.
        - Output: Binary classification (Flood Occurred: 1 or No Flood: 0).
        - Trained using high-performance Snap ML for faster computation.
    - **Earthquake Prediction Model**:
        - Algorithm: **Random Forest (Snap ML)**.
        - Inputs: Latitude, longitude, depth, and magnitude station count.
        - Output: Magnitude of the earthquake (regression task).
3. **IBM Watson AutoAI and Snap ML**
    - IBM AutoAI was used to automate the training and hyperparameter optimization of both models.
    - Snap ML ensured high-performance machine learning by leveraging GPU acceleration for faster model training and deployment.
4. **Data Integration**
    - **IBM Cloud Object Storage**:
        - Used to store real-time data, including environmental parameters and disaster-specific information.
    - **Watsonx.ai Studio**:
        - Enabled building and deploying AI models seamlessly.
    - **Watsonx Runtime**:
        - Provided the runtime environment for executing predictive models efficiently.
    - **Deployment Space**:

        - Managed and deployed models for real-time accessibility.
5. **Web Interface**

    - **Streamlit** was used to create an intuitive, interactive user interface for:
        - Uploading data and configuring prediction parameters.
        - Visualizing real-time predictions and resource allocation insights.
        - Displaying detailed disaster impact forecasts with graphs and metrics.
6. **Resource Allocation Optimization**

    - Integrated predictive insights with a custom algorithm to optimize resource distribution based on severity, region-specific needs, and real-time data.
    - The system ensured that relief materials were allocated effectively, reducing overstocking in some areas and shortages in others.
7. **End-to-End Workflow**

    - Data is collected in real time from weather and seismic monitoring systems.
    - Predictive models analyze the data to forecast disasters.
    - Results are displayed on the Streamlit interface, enabling relief teams to take informed, proactive measures.
    - IBM Cloud services ensure seamless data processing, storage, and model deployment, supporting real-time decision-making.
8. **Testing and Evaluation**

    - Models were validated using cross-validation techniques to ensure high accuracy and robustness.
    - Performance metrics like F1-score, precision, recall (for floods), and RMSE (for earthquakes) were used to evaluate the models.

6. Deployment

The deployment of DisasterShield involves setting up an AI-powered disaster relief optimization system using IBM Cloud services, ensuring accessibility, scalability, and efficiency.

1.Deployment Infrastructure

The system is deployed on IBM Cloud, leveraging its AI and machine learning services for predictive analytics. The key components of the deployment infrastructure include:

- IBM Watson Studio: Used for model training, testing, and deployment.
- IBM Cloud Object Storage: Stores disaster data, historical datasets, and processed results.
- IBM Watsonx Runtime: Provides an execution environment for AI models, ensuring fast predictions.
- IBM Cloud Virtual Machines (VMs): Hosts the backend processing of predictive models.
- Streamlit Web Application: Deployed on IBM Cloud to provide an interactive and accessible frontend.

2. Deployment Process

1. Model Deployment on IBM Watsonx.ai
    - The flood prediction (Logistic Regression) and earthquake magnitude estimation (Random Forest) models are trained and deployed using IBM Watson AutoAI and Snap ML.
    - The trained models are registered in IBM Watson Machine Learning (WML) for inference.
    - API endpoints are created for accessing predictions via IBM Watsonx Runtime.
2. Data Storage and Integration
    - Data, such as rainfall levels, seismic activity, and river discharge, is ingested into IBM Cloud Object Storage.
    - Historical datasets are preprocessed and stored for model retraining and validation.
3. Frontend Deployment using Streamlit
    - The web application is built using Streamlit and hosted on IBM Cloud.
    - The frontend allows users to input parameters, visualize predictions, and receive resource allocation insights.

4. Scalability and Performance Optimization
    - The deployment ensures horizontal scaling to handle increased user requests during disaster events.
    - IBM Cloud Service is used for managing containerized deployments, ensuring reliability.
    - The system is optimized for low-latency predictions, reducing response time for disaster .

3.Monitoring and Maintenance

- IBM Cloud Monitoring is used to track model performance, uptime, and API usage.
- Continuous model improvement is ensured through automated retraining based on updated datasets.
- Security protocols are implemented to protect sensitive disaster-related data from unauthorized access.

7. Future Scope

1. Expansion to Other Natural Disasters:

- The system can be expanded to predict and manage additional types of natural disasters, such as wildfires, tsunamis, and hurricanes, by incorporating new datasets and predictive models.
- This would make the system more comprehensive, offering an all-encompassing solution for disaster response teams.

2. Improved Predictive Accuracy

- **Real-Time Data from IoT Devices**:
    - Leveraging Internet of Things (IoT) devices, such as weather sensors, water-level gauges, and seismic activity detectors, to collect real-time data for more accurate and timely predictions.
    - Incorporating satellite data and remote sensing technologies to improve the spatial resolution and coverage of the predictions.

3. Geographic Expansion and Regional Customization

- **Local Data Integration**:
    - Extending the system to cater to regional or local requirements by integrating region-specific datasets, such as soil moisture data for flood predictions or localized seismic activity data for earthquakes.
    - Implementing localized models that are tailored to the unique characteristics of specific regions, improving prediction accuracy and relevance.
- **Global Coverage**:
    - Expanding the geographic coverage of the system to include disaster-prone areas worldwide, providing predictions and insights for international disaster relief efforts.

4. AI-Powered Rescue Coordination:

- Leveraging AI to help coordinate rescue operations, suggesting the most efficient routes for emergency teams and tracking real-time progress through live data feeds.

5. Integration with Social Media and Crowdsourced Data

- **Crowdsourcing Disaster Data**:

    - Integrating crowdsourced data from social media platforms, such as tweets or posts during disasters, to provide real-time situational awareness and verify disaster events.
    - Combining machine learning models with natural language processing (NLP) techniques to analyze social media feeds for on-the-ground disaster reports.

6. Mobile App Development

- **Mobile Platform**:
    - Developing a mobile application that provides users with disaster predictions, real-time alerts, and resource optimization information on their smartphones for faster access and on-the-go updates.
    - Integrating geolocation-based services to deliver personalized notifications and disaster response recommendations based on the user's location.

7. Post-Disaster Recovery and Impact Assessment

- **Damage Assessment**:
    - Using AI to analyze satellite images and drone footage to assess the damage caused by disasters, enabling more accurate impact assessments for recovery planning.
    - Incorporating machine learning techniques to evaluate the effectiveness of disaster response and recovery operations.
- **Long-Term Impact Predictions**:
    - Extending the system's capabilities to predict the long-term environmental and economic impacts of disasters, helping governments and organizations plan for long-term recovery and resilience-building strategies.

8. Adoption of Blockchain for Data Security

- **Blockchain for Transparency and Security**:
    - Implementing blockchain technology to secure disaster-related data, ensuring transparency and traceability for resource allocation and management.
    - Using blockchain for verifying data authenticity, especially in crowdsourced disaster reports, to ensure that the information is reliable and accurate.

8. Conclusion

DisasterShield represents a significant step forward in leveraging AI for disaster management, addressing the critical challenges of delayed responses and inefficient resource allocation in flood and earthquake relief operations. By combining IBM Watson's AutoAI, Snap ML, and IBM Cloud services with real-time data integration, the system empowers disaster relief teams with predictive analytics and actionable insights.

The project demonstrates how AI-driven solutions can transform disaster preparedness by accurately forecasting calamities, streamlining resource distribution, and enabling proactive decision-making. With its user-friendly Streamlit interface and robust modeling techniques, DisasterShield bridges the gap between data and action, ensuring timely interventions that can save lives and reduce the impact of natural disasters.

As climate change continues to exacerbate the frequency and intensity of such events, DisasterShield provides a scalable and adaptable framework for addressing diverse disaster scenarios. This project sets a precedent for the application of cutting-edge technology in creating resilient and efficient disaster management systems, ultimately contributing to a safer and more sustainable future.

Appendix A

Project Code

GitHub Link for our project:- Smruti0109/DisasterShield

```
import streamlit as st
import requests
import pandas as pd
import folium
from geopy.distance import geodesic
from streamlit_folium import st_folium
import json
from PIL import Image

# Disaster options
DISASTER_TYPES = ["Flood", "Earthquake"]

# Initialize session state
if "lat" not in st.session_state:
    st.session_state["lat"] = None
if "lon" not in st.session_state:
    st.session_state["lon"] = None
if "disaster_type" not in st.session_state:
    st.session_state["disaster_type"] = None
if "current_stock" not in st.session_state:
    st.session_state["current_stock"] = None

# Predefined stock locations
stock_locations = [
    {"name": "Delhi", "coords": (28.7041, 77.1025), "resources": {"Food": 500, "Water": 1000, "Medical Kits": 300}},
    {"name": "Mumbai", "coords": (19.0760, 72.8777), "resources": {"Food": 700, "Water": 1200, "Medical Kits": 400}},
```

```
    {"name": "Chennai", "coords": (13.0827, 80.2707), "resources": {"Food": 450, "Water": 900, "Medical Kits": 200}},
    {"name": "Kolkata", "coords": (22.5726, 88.3639), "resources": {"Food": 600, "Water": 1100, "Medical Kits": 350}},
]
import requests
def predict_flood(inputs):
    # IBM Watson API key and token generation
    API_KEY = "SHzusG7In1o2YosuocN02UTXbRxnShE0pYGax9NS9Eqx"
    token_response = requests.post(
        'https://iam.cloud.ibm.com/identity/token',
        data={"apikey": API_KEY, "grant_type": 'urn:ibm:params:oauth:grant-type:apikey'}
    )
    mltoken = token_response.json().get("access_token")
    if not mltoken:
        st.error("Failed to obtain token. Check your API key.")
        return "Error in prediction"

    header = {
        'Content-Type': 'application/json',
        'Authorization': 'Bearer ' + mltoken
    }

    # Prepare the payload with the user's input
    payload_scoring = {
        "input_data": [{
            "fields": ["Latitude", "Longitude", "Rainfall (mm)", "River Discharge (m³/s)", "Water Level (m)", "Historical Floods"],
            "values": [list(inputs.values())]
        }]
    }
```

```
    # Send the request to the flood prediction model endpoint
    response_scoring = requests.post(

'https://eu-de.ml.cloud.ibm.com/ml/v4/deployments/9c34421c-e85b-4f19-9c52-672a03e77d69/predictions?version=2021-05-01',
        json=payload_scoring,
        headers=header
    )
    # Log the response details
    if response_scoring.status_code == 200:
        result = response_scoring.json()
        # st.write(result)  # Log the full response for debugging
        flood_occurred = result.get("predictions", [{}])[0].get("values", [[0]])[0][0]
        return "Yes" if flood_occurred == 1 else "No"
    else:
        st.error(f"API request failed with status code {response_scoring.status_code}: {response_scoring.text}")
        return "Error in prediction"
def predict_earthquake(inputs):
    # IBM Watson API key and token generation
    API_KEY = " w9EwMCdVlZX2gj_9zKkda0sTjLyzKW94vS42niPMy6gv"
    token_response = requests.post(
        'https://iam.cloud.ibm.com/identity/token',
        data={"apikey": API_KEY, "grant_type": 'urn:ibm:params:oauth:grant-type:apikey'}
    )
    mltoken = token_response.json()["access_token"]

    header = {
        'Content-Type': 'application/json',
        'Authorization': 'Bearer ' + mltoken
```

```
    }

    # Prepare the payload with the user's input
    payload_scoring = {
        "input_data": [{
            "fields": ["depth", "magNst", "latitude", "longitude"],
            "values": [list(inputs.values())]
        }]
    }

    # Send the request to the earthquake prediction model endpoint
    response_scoring = requests.post(
'https://eu-de.ml.cloud.ibm.com/ml/v4/deployments/59c280a2-4624-4145-8820-48382e1a4e5a/predictions?version=2021-05-01',
        json=payload_scoring,
        headers=header
    )

    if response_scoring.status_code == 200:
        result = response_scoring.json()
        try:
            magnitude = result.get("predictions", [{}])[0].get("values", [[None]])[0][0]
            if magnitude is None:
                raise ValueError("Magnitude value is missing")
            return round(float(magnitude), 2)
        except (TypeError, ValueError):
            st.error("Failed to parse magnitude from API response.")
            return "Error in prediction"
    else:
```

```
        st.error(f"API request failed with status code {response_scoring.status_code}: {response_scoring.text}")
        return "Error in prediction"


def get_earthquake_category(magnitude):
    """Classify earthquake magnitude into categories."""
    try:
        magnitude = float(magnitude)  # Ensure magnitude is a float
    except (ValueError, TypeError):
        return "Invalid magnitude value"


    if magnitude < 2.0:
        return "Micro Earthquake"
    elif 2.0 <= magnitude < 4.0:
        return "Minor Earthquake"
    elif 4.0 <= magnitude < 6.0:
        return "Light Earthquake"
    elif 6.0 <= magnitude < 7.0:
        return "Strong Earthquake"
    elif 7.0 <= magnitude < 8.0:
        return "Major Earthquake"
    elif 8.0 <= magnitude < 10.0:
        return "Great Earthquake"
    else:
        return "Massive Earthquake"


# Function to set custom background with revolving Earth
def set_background():
    background_style = """
    <style>
    .stApp {
```

```
        background: radial-gradient(circle at center, #0d1117, #0d1117);
        color: white;
        font-family: 'Arial', sans-serif;
        position: relative;
        overflow: hidden;
        min-height: 100vh;
    }
    .earth {
        position: absolute;
        top: 50%;
        left: 50%;
        transform: translate(-50%, -50%);
        width: 500px;
        height: 500px;
        background:
url('https://upload.wikimedia.org/wikipedia/commons/9/97/The_Earth_seen_from_Apollo_17.jpg') no-repeat center;
        background-size: cover;
        border-radius: 50%;
        animation: spin 20s linear infinite;
        box-shadow: 0 0 50px rgba(255, 255, 255, 0.5);
    }
    @keyframes spin {
        from { transform: translate(-50%, -50%) rotate(0deg); }
        to { transform: translate(-50%, -50%) rotate(360deg); }
    }

    .sidebar .sidebar-content {
        background: linear-gradient(to bottom, #3a3a3a, #6a11cb);
        color: white;
    }
```

```
.stButton > button {
    background: linear-gradient(to right, #43cea2, #185a9d);
    color: white;
    border: none;
    border-radius: 8px;
    font-size: 18px;
    padding: 10px 20px;
    transition: transform 0.2s;
}
.stButton > button:hover {
    transform: scale(1.1);
    background: linear-gradient(to right, #185a9d, #43cea2);
}
.stDropdown {
    background: linear-gradient(to bottom right, #ffffff, #e0e0e0);
    border-radius: 5px;
    font-size: 16px;
}
.square-box {
    width: 300px;
    height: 150px;
    background-color: #161b22;
    border: 1px solid #30363d;
    border-radius: 10px;
    display: flex;
    align-items: center;
    justify-content: center;
    color: #c9d1d9;
    font-size: 16px;
    margin: 20px;
```

```
        text-align: center;
        padding: 10px;
        box-shadow: 0 4px 8px rgba(0, 0, 0, 0.2);
    }
    </style>
    <div class="earth"></div>
    """
    st.markdown(background_style, unsafe_allow_html=True)

# Apply custom background
set_background()

# Streamlit Sidebar
st.sidebar.title("Navigation")
menu = st.sidebar.selectbox(
    "Go to", ["Home", "Disaster Monitoring & Prediction", "Resource Tracking"],
    format_func=lambda x: f"🌟 {x}"
)

# Main Dashboard
st.title("🌍 DisasterShield: Disaster Relief System")

if menu == "Home":
    st.subheader("Welcome!")
    st.markdown("""
        DisasterShield is an disaster relief system designed to optimize disaster response efforts. It leverages predictive analytics, and resource tracking to provide actionable insights for disaster management.

        Explore:

        - *Disaster Monitoring & Prediction*: Track weather conditions and predict disaster impacts.
```

```
    - *Resource Tracking*: Monitor and optimize resource allocation.
    """)

    # Corrected Code

    col1, col2 = st.columns(2)

    with col1:
        st.markdown("<h3 style='text-align: center;'>Flood Statistics</h3>", unsafe_allow_html=True)
        st.markdown("<p style='text-align: center;'>Monitor rainfall data and assess flood risks.</p>", unsafe_allow_html=True)
        flood_image = Image.open("C:\\Users\\Pratiksha\\Documents\\VScode\\disaster_relief\\india-flood-prone-areas-map-2021.jpg")  # Replace with your actual path
        st.image(flood_image, caption="Flood Monitoring", use_container_width=True)
    with col2:
        st.markdown("<h3 style='text-align: center;'>Earthquake Statistics</h3>", unsafe_allow_html=True)
        st.markdown("<p style='text-align: center;'>Track seismic activity and evaluate earthquake impacts.</p>", unsafe_allow_html=True)
        earthquake_image = Image.open("C:\\Users\\Pratiksha\\Documents\\VScode\\disaster_relief\\Seismic-Map-of-India.png")  # Replace with your actual path
        st.image(earthquake_image, caption="Earthquake Monitoring", use_container_width=True)



elif menu == "Disaster Monitoring & Prediction":
    st.subheader("⛈️ Disaster Monitoring & AI Predictions")
    # Disaster type selection
    disaster_type = st.selectbox("Select Disaster Type", DISASTER_TYPES)
    st.session_state["disaster_type"] = disaster_type
```

```
if disaster_type == "Flood":
    # Inputs for flood prediction
    lat = st.number_input("Enter Latitude", step=0.01)
    lon = st.number_input("Enter Longitude", step=0.01)
    rainfall = st.number_input("Enter Rainfall (mm)", value=0.0)
    river_discharge = st.number_input("Enter River Discharge (m³/s)", value=0.0)
    water_level = st.number_input("Enter Water Level (m)", value=0.0)
    historical_floods = st.selectbox("Historical Floods in Area", ["Yes", "No"])

    if st.button("Predict Flood Impact"):
        inputs = {
            "Latitude": lat,
            "Longitude": lon,
            "Rainfall (mm)": rainfall,
            "River Discharge (m³/s)": river_discharge,
            "Water Level (m)": water_level,
            "Historical Floods": historical_floods
        }
        st.session_state["lat"] = lat  # Store latitude in session state
        st.session_state["lon"] = lon  # Store longitude in session state
        flood_result = predict_flood(inputs)
        st.markdown(f"*Flood Prediction Result:*\n- Flood Occurred: {flood_result}")

elif disaster_type == "Earthquake":
    # Inputs for earthquake prediction
    lat = st.number_input("Enter Latitude", value=20.5937, step=0.01)
    lon = st.number_input("Enter Longitude", value=78.9629, step=0.01)
    depth = st.number_input("Enter Depth (km)", value=10.0, step=0.1)
    magNst = st.number_input("Enter Magnitude (magNst)", value=4.0, step=0.1)
```

```
        if st.button("Predict Earthquake Impact"):
            inputs = {
                "latitude": lat,
                "longitude": lon,
                "depth": depth,
                "magNst": magNst
            }
            magnitude = predict_earthquake(inputs)
            category = get_earthquake_category(magnitude)
            st.markdown(f"**Prediction Result:**\n- Magnitude: {magnitude}\n- Category: {category}")
            st.session_state["lat"] = lat
            st.session_state["lon"] = lon


    # Display selected location on map only if lat and lon are valid
    if st.session_state["lat"] is not None and st.session_state["lon"] is not None:
        st.markdown("*Location Map:*")
        m = folium.Map(location=[st.session_state["lat"], st.session_state["lon"]], zoom_start=6)
        folium.Marker([st.session_state["lat"], st.session_state["lon"]],
                      tooltip=f"Disaster Location: ({st.session_state['lat']}, {st.session_state['lon']})").add_to(m)
        st_folium(m, width=800, height=400)


elif menu == "Resource Tracking":
    st.subheader("🚚 Resource Tracking")


    st.markdown(
        """
        <style>
        .large-font {
```

```
            font-size:20px !important;
        }
        .medium-font {
            font-size:18px !important;
        }
        </style>
        """,
        unsafe_allow_html=True
    )

    # Load the resources.csv file
    resources_file = "C:\\Users\\Pratiksha\\Documents\\VScode\\disaster_relief\\resources_data.csv"
    resources_df = pd.read_csv(resources_file)

    # Check for missing or invalid data
    if resources_df[['Latitude', 'longitude']].isnull().any().any():
        st.error("Error: The dataset contains missing latitude or longitude values. Please check your resources.csv file.")
    else:
        if "lat" in st.session_state and "lon" in st.session_state and st.session_state["lat"] is not None and st.session_state["lon"] is not None:
            user_coords = (st.session_state["lat"], st.session_state["lon"])

            # Check if user input coordinates are valid
            if not all(map(lambda x: x is not None and isinstance(x, (int, float)), user_coords)):
                st.error("Invalid input latitude and longitude. Please enter valid coordinates.")
            else:
                # Calculate the geodesic distance
                resources_df['Distance'] = resources_df.apply(
```

```
            lambda row: geodesic(user_coords, (row['Latitude'],
row['longitude'])).km, axis=1
        )
        closest_stock = resources_df.loc[resources_df['Distance'].idxmin()]

        # Ask if the user wants to allocate resources (default set to "No")
        allocate = st.radio("Do you want to allocate resources?", ("Yes", "No"),
index=1)

        # Display available resources table
        st.markdown(f"Closest Stock Location (lat: {closest_stock['Latitude']}, lon:
{closest_stock['longitude']}):")
        st.markdown("### Available Resources")
        stock_info = closest_stock[["Food And Water", "Clothing", "Shelter",
"Medical Suppliers"]]
        stock_df = pd.DataFrame(stock_info).reset_index()
        stock_df.columns = ["Resource", "Available Quantity"]
        st.table(stock_df)

        if allocate == "Yes":
            # Help allocate resources
            st.markdown("### Allocate Resources")
            allocations = {}
            for resource in stock_info.index:
                allocations[resource] = st.number_input(
                    f"Allocate {resource.replace('_', ' ').capitalize()}",
                    min_value=0, max_value=int(stock_info[resource]), step=1
                )

            if st.button("Update Stock After Allocation"):
                # Update the resources in the DataFrame
                for resource, allocated_quantity in allocations.items():
```

```
                    closest_stock[resource] -= allocated_quantity


                # Save the updated data back to CSV
                resources_df.loc[resources_df['Distance'].idxmin()] = closest_stock
                resources_df.drop(columns=["Distance"], inplace=True)  # Clean up before saving
                resources_df.to_csv(resources_file, index=False)


                st.success("Stock updated successfully!")


                # Display updated stock
                updated_stock_info = closest_stock[["Food And Water", "Clothing", "Shelter", "Medical Suppliers"]]
                updated_stock_df = pd.DataFrame(updated_stock_info).reset_index()
                updated_stock_df.columns = ["Resource", "Remaining Quantity"]
                st.markdown("### Updated Stock Levels")
                st.table(updated_stock_df)
            else:
                st.info("You chose not to allocate resources. The available resources are displayed above.")
        else:
            st.warning("Please enter latitude and longitude in the Disaster Monitoring tab first.")
```

Appendix B

Screenshot of Project

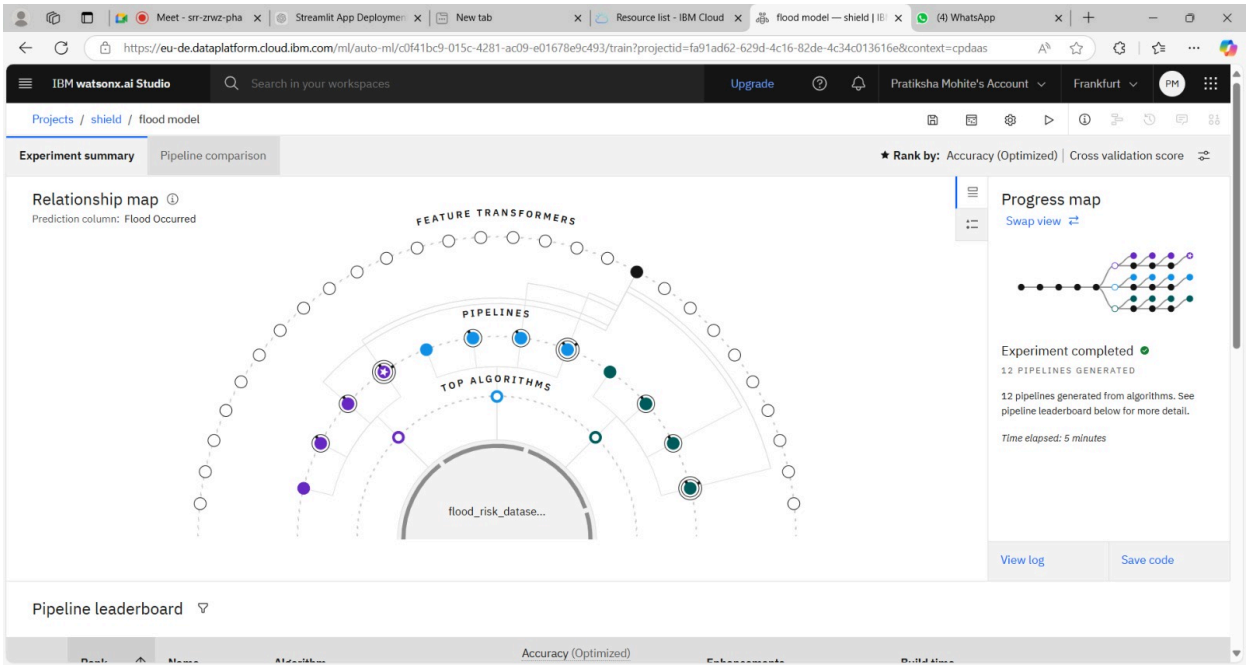


Fig 1: Relationship Map for flood model

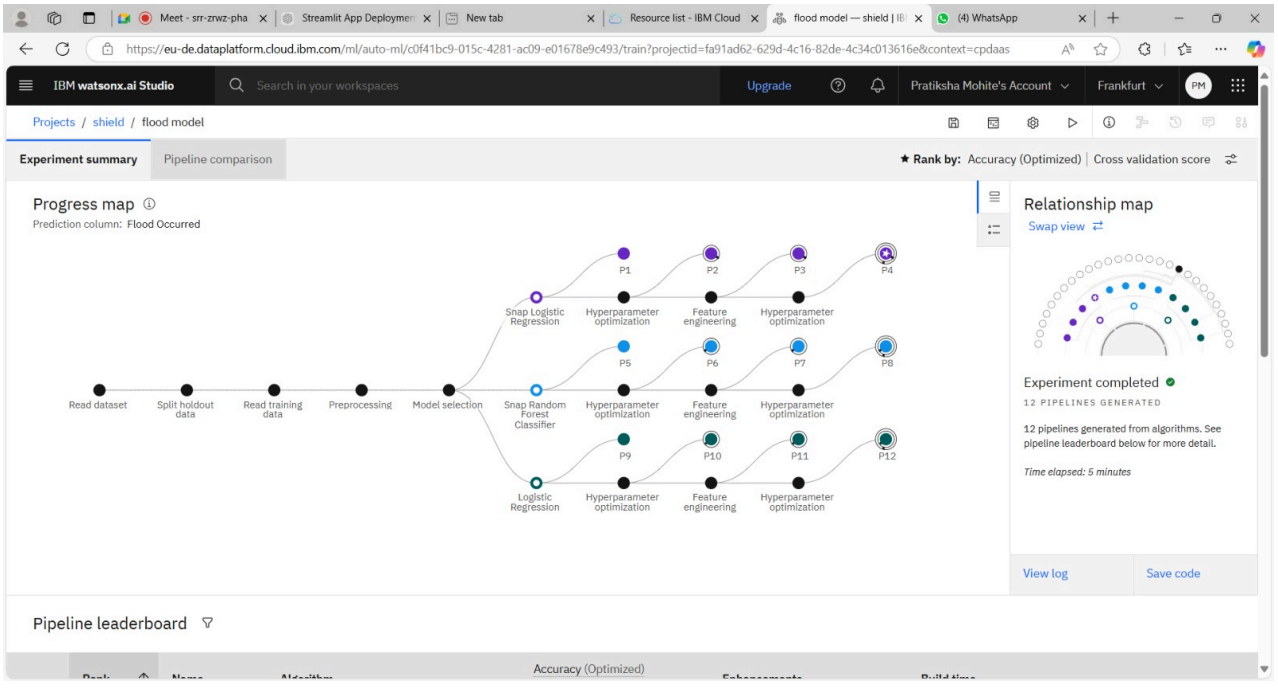


Fig 2: Pipeline Comparison for flood model

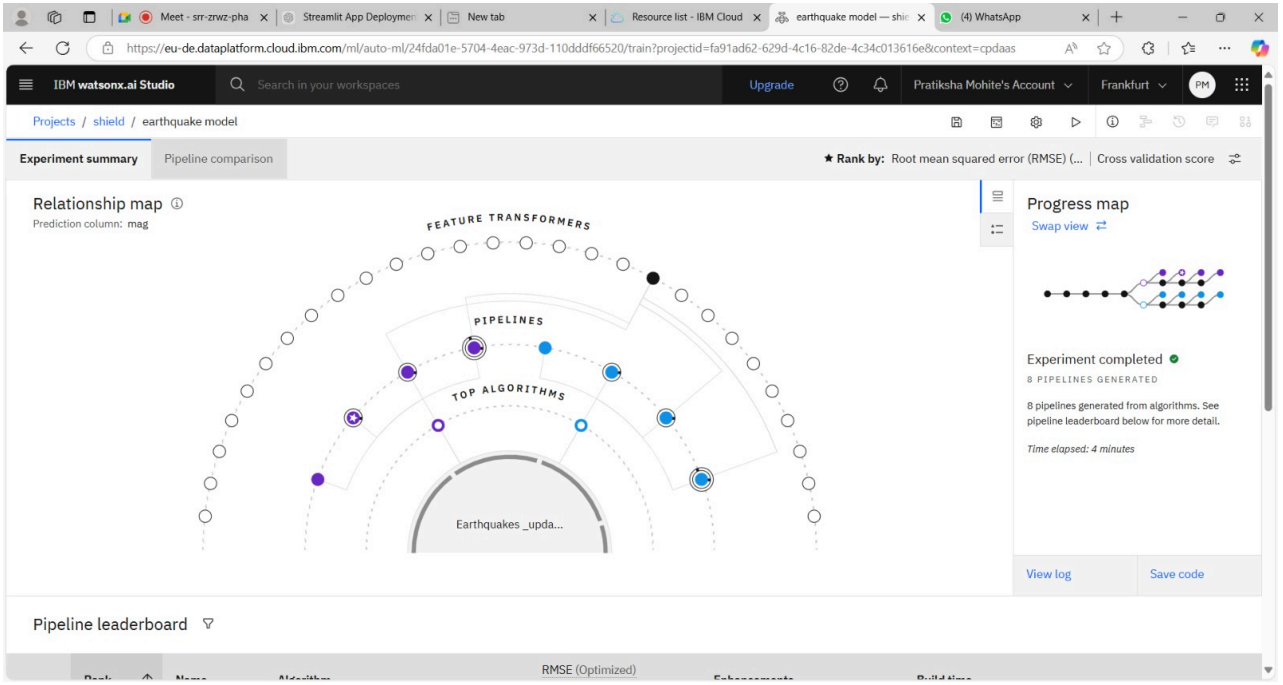


Fig 3: Relationship Map for earthquake model

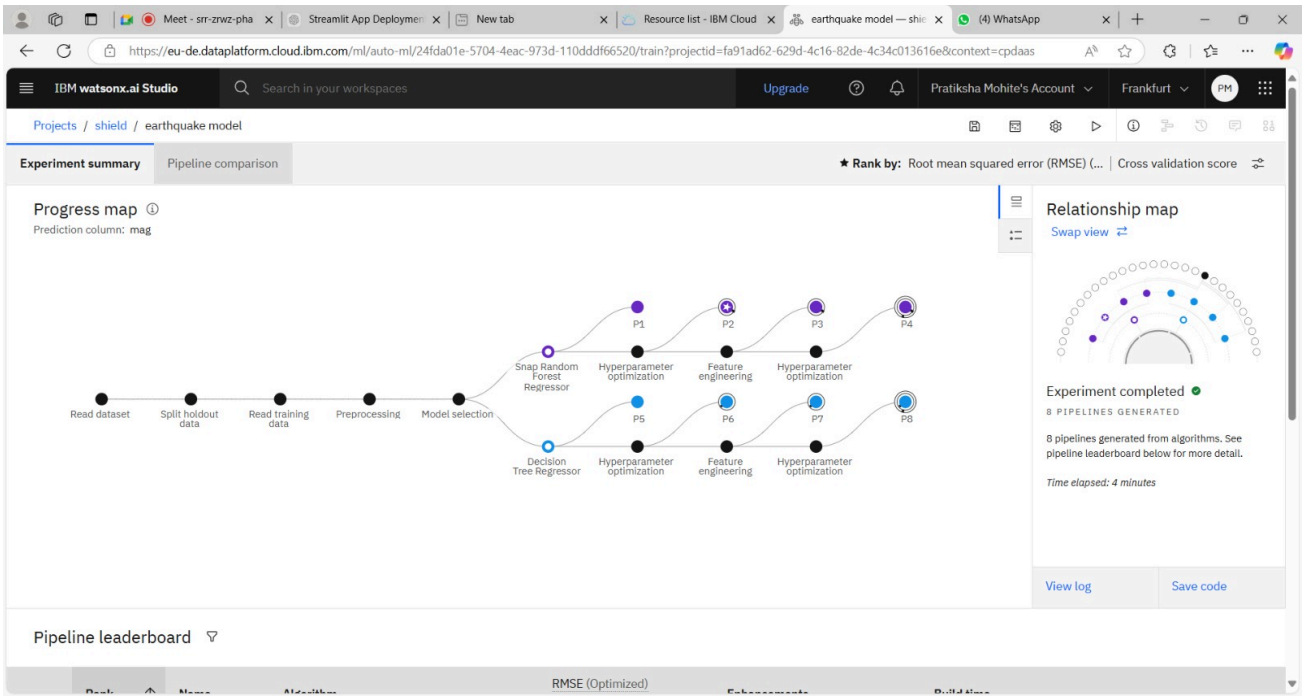


Fig 2: Pipeline Comparison for earthquake model

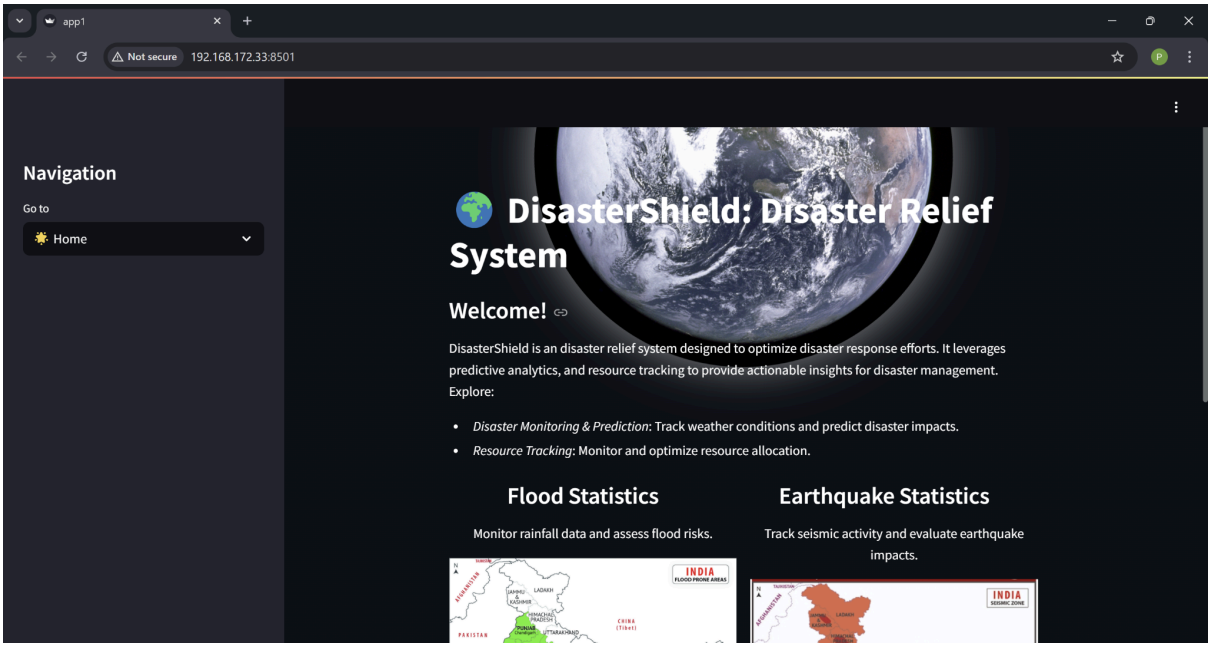


Fig 5 :- This is Home page of our streamlit interface

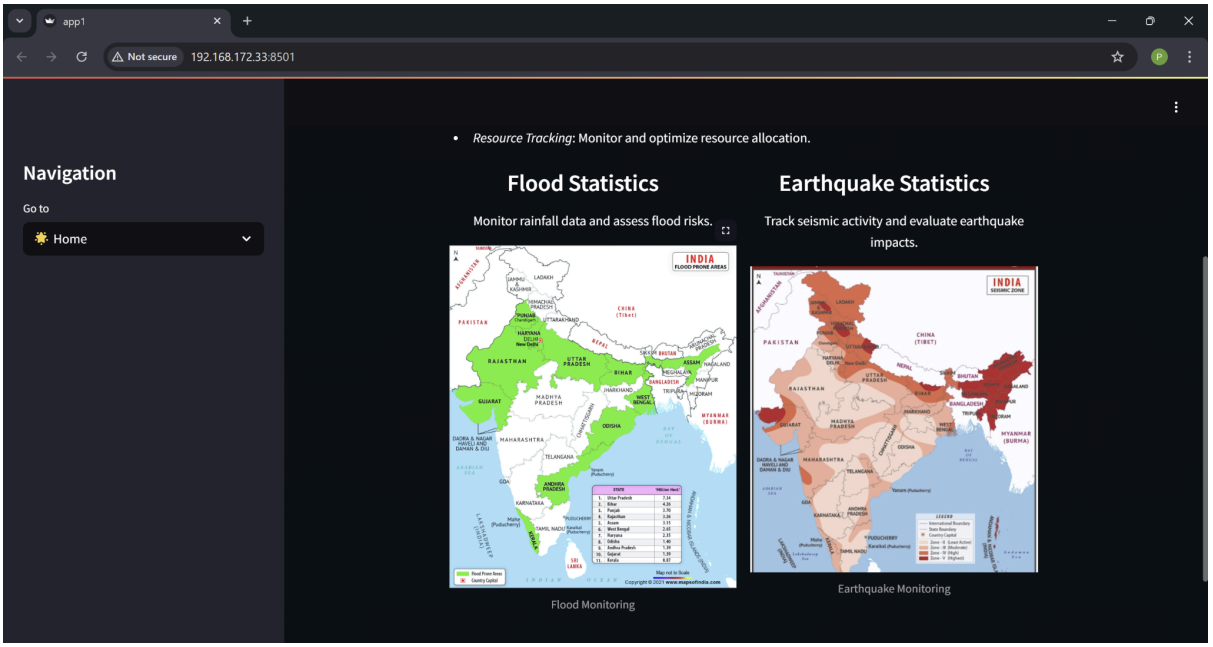


Fig 6:- Finding whether flood has occurred or not, through the form

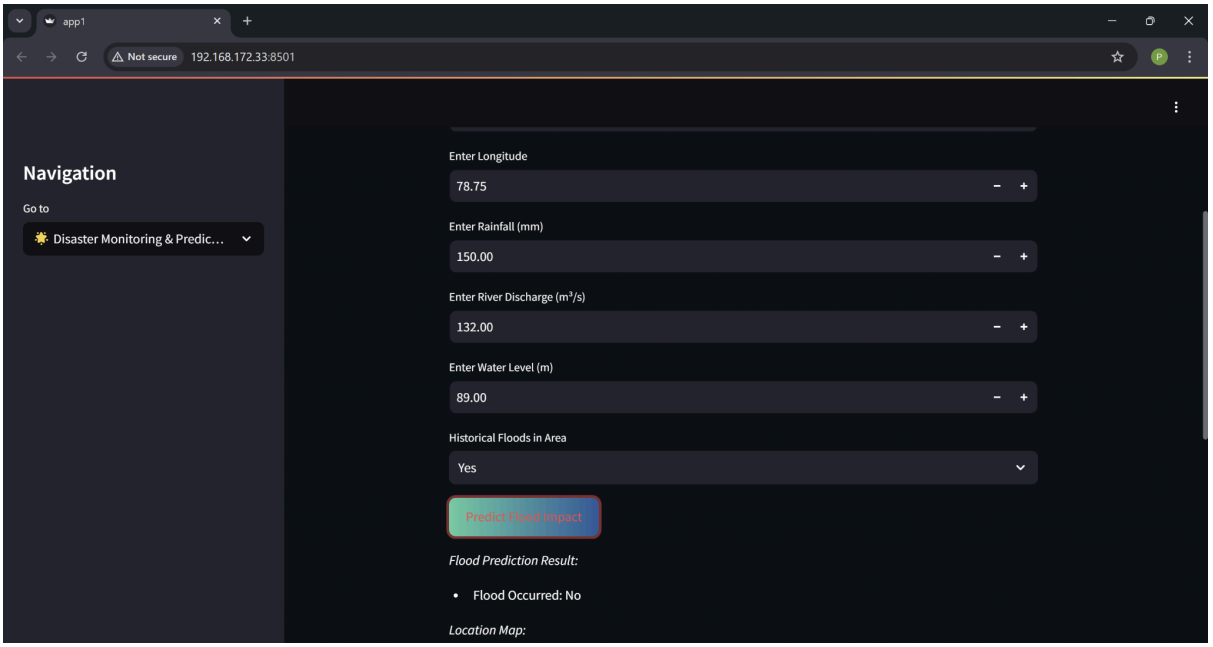


Fig 7:- Got the output for flood

Fig 8:- Finding whether earthquake has occurred or not, through the form

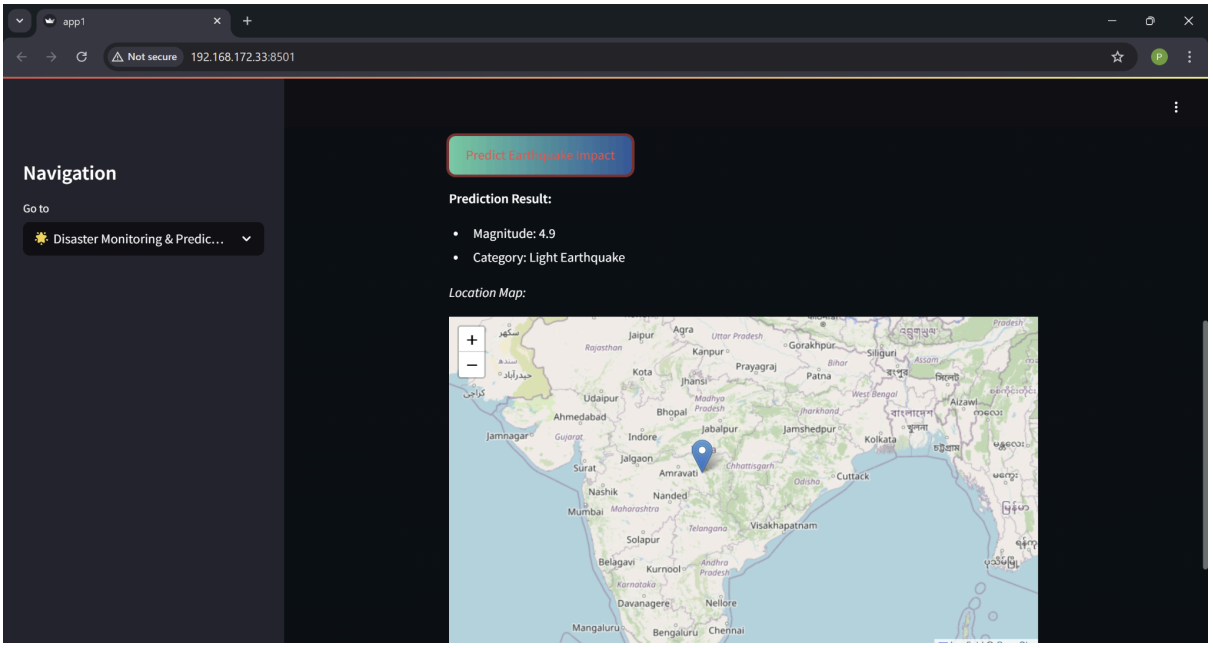


Fig 9:- Got the output for flood

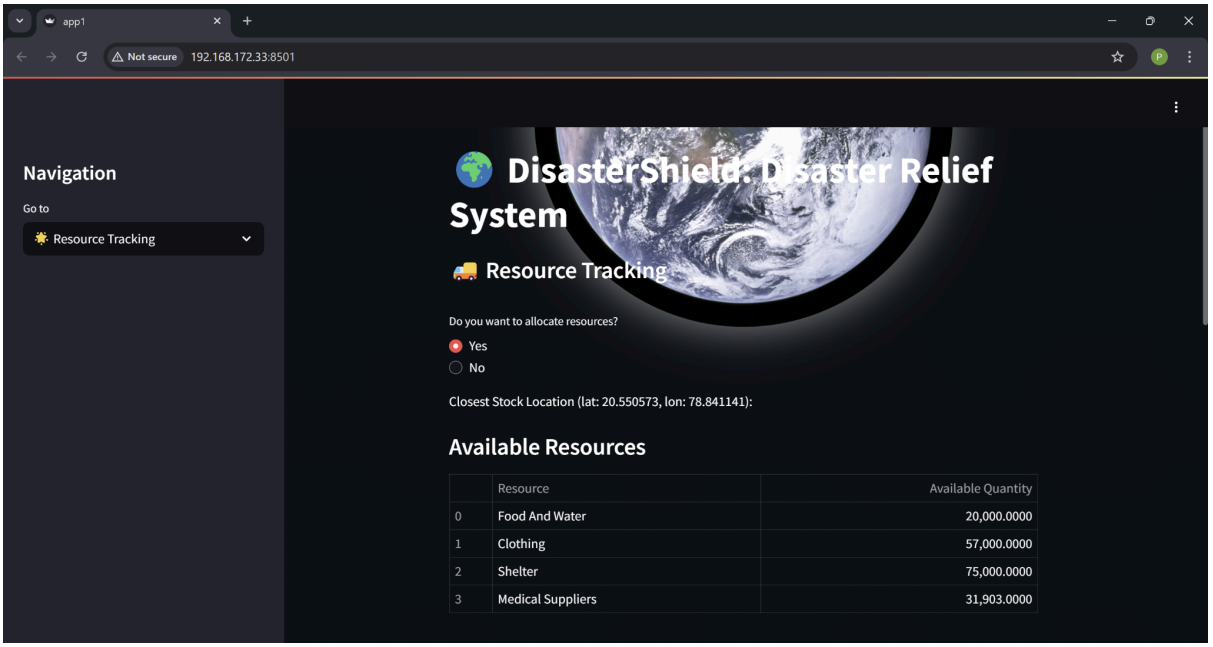
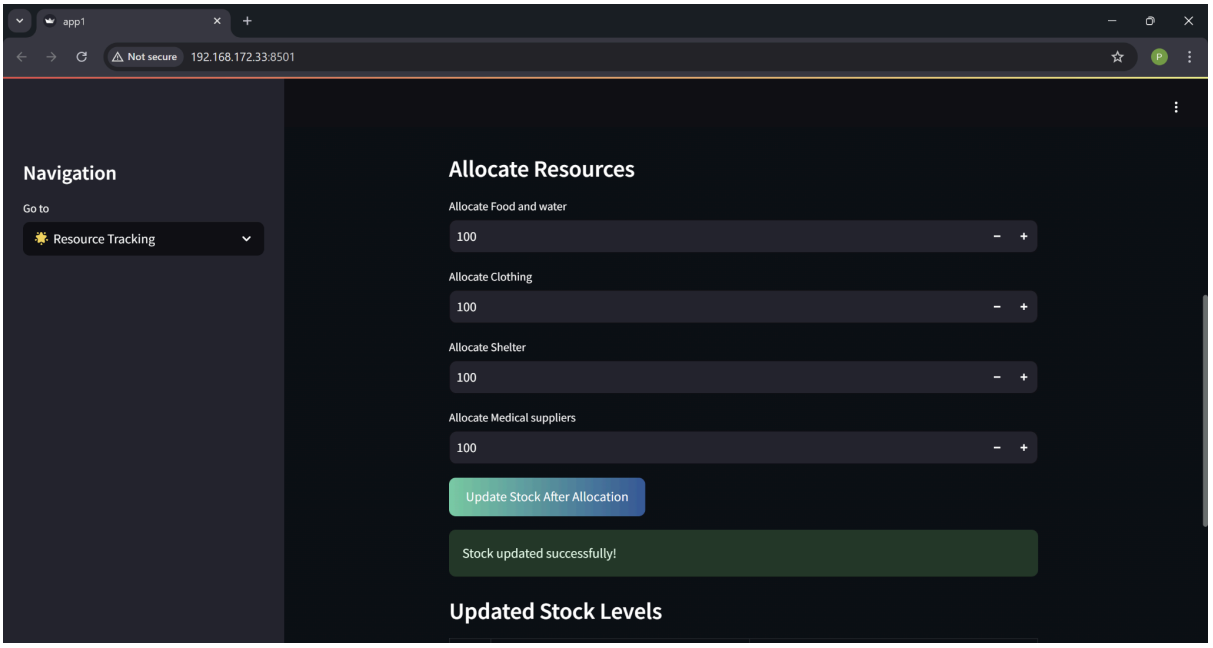


Fig 10:- If the end user want to allocate the resources



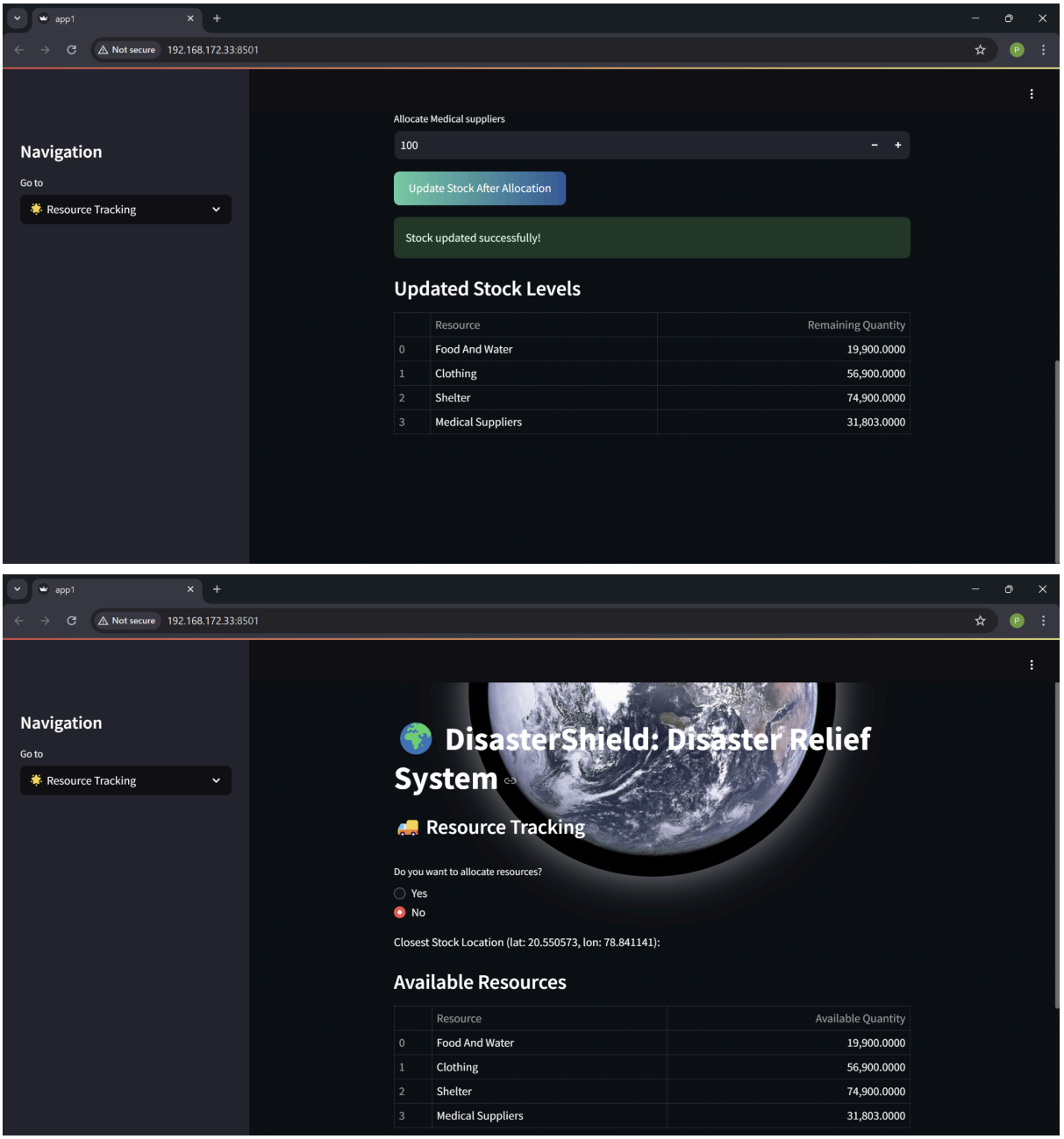


Fig 11:- If the end user does not want to allocate the resources

Appendix C

Abbreviation

AI: Artificial Intelligence

CWC: Central Water Commission

F1-Score: F1 Measure (Harmonic Mean of Precision and Recall)

GPU: Graphics Processing Unit

IoT: Internet of Things

IMD: Indian Meteorological Department

magNst: Magnitude Station Count

ML: Machine Learning

NLP: Natural Language Processing

RMSE: Root Mean Square Error

Snap ML: Scalable and High-Performance Machine Learning

References

IBM Watson AutoAI
IBM Watson AutoAI Documentation. (2024). Retrieved from https://www.ibm.com/watson/autoai

Snap ML
Snap ML: High-Performance Machine Learning Library. IBM Research. (2024). Retrieved from https://www.ibm.com/research/snapml

Streamlit
Streamlit: The fastest way to build data apps. (2024). Retrieved from https://streamlit.io/

IBM Cloud
IBM Cloud Services. (2024). Retrieved from https://www.ibm.com/cloud