**AI-EARTHQUAKE PREDICTION MODEL USING PYTHON**

**Phase3-DEVELOPMENT PART 1.**

**Topic-Loading & Preprosessing.**

**Name:C.SANDHIYA**

**Reg.No: 511721104008**

**Introduction:**

* It is well known that if a disaster occurs in one region, it is likely to happen again. Some regions have frequent earthquakes, but this is only a comparative amount compared to other regions.

* So, predicting the earthquake with date and time, latitude and longitude from previous data is not a trend that follows like other things, it happens naturally.



So we will be predicting the earthquake from Date and Time, Latitude, and Longitude from previous data is not a trend that follows like other things. It is naturally occurring.

**1. Data Preparation and Preprocessing:**

* 1. **Loading Data:** Import earthquake data into your development environment.

**Program:**

import pandas as pd

# Load earthquake data into a Pandas DataFrame

df = pd.read\_csv("earthquake\_dataset.csv")

* 1. **Cleaning Data:** Handle missing values, outliers, or any inconsistencies in the dataset.

**Program:**

# Handling missing values: Drop or fill missing values based on the nature of the data

df.dropna(inplace=True) # Example: dropping rows with missing values

# Handling outliers: Identify and remove or transform outliers using statistical methods or visualization tools

from scipy import stats

df = df[(np.abs(stats.zscore(df['numerical\_column'])) < 3)] # Example: removing outliers using Z-score

# Handling inconsistencies: Correct any inconsistent or erroneous data

# Example: Correcting inconsistent category names

df['category\_column'].replace({'incorrect\_name': 'correct\_name'}, inplace=True)

* 1. **Feature Engineering:** Extract relevant features, transform data, and create new features if necessary.

**Program:**

# Extracting relevant features

df['date'] = pd.to\_datetime(df['date']) # Convert date column to datetime format

df['month'] = df['date'].dt.month # Extract month

df['day'] = df['date'].dt.day # Extract day

df['hour'] = df['time'].str.split(':').str[0].astype(int) # Extract hour from time column

# Creating new features

df['magnitude\_squared'] = df['magnitude'] \*\* 2 # Example: creating a squared magnitude feature

* 1. **Data Scaling/Normalization:** Normalize numerical features to ensure consistency and improve model performance.

**Program:**

from sklearn.preprocessing import StandardScaler

# Initialize the scaler

scaler = StandardScaler()

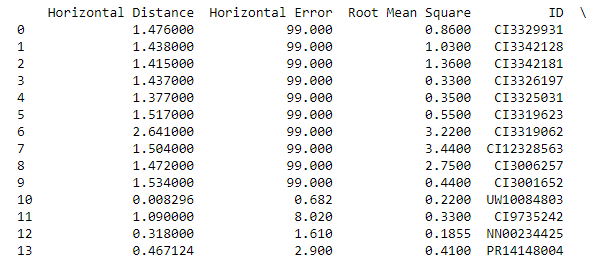
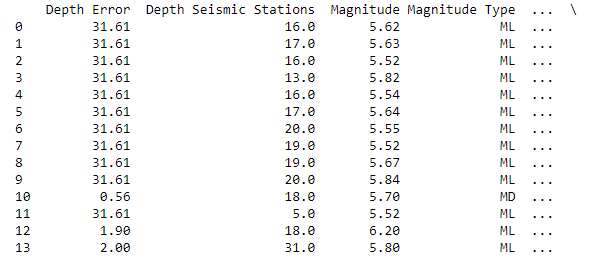
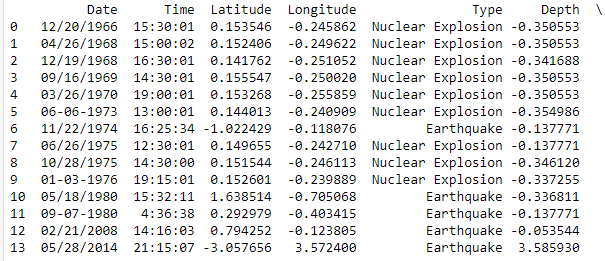
# Select numerical features to normalize

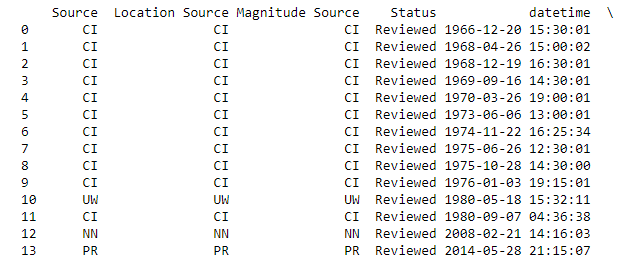
numerical\_features = ['latitude', 'longitude', 'depth']

# Normalize selected features

df[numerical\_features] = scaler.fit\_transform(df[numerical\_features])

**Output:**

****

****

1. **Model Selection and Building:** 
   1. **Choose a Model:** Select a suitable machine learning algorithm (like Random Forest, Neural Networks, etc.) for regression tasks. For regression tasks, you have various options such as Random Forest, Neural Networks, Support Vector Machines, or Gradient Boosting Machines. Let's choose Random Forest as an example:

**Program:**

from sklearn.ensemble import RandomForestRegressor

# Choose the Random Forest Regressor model

model = RandomForestRegressor(n\_estimators=100, random\_state=42)

* 1. **Model Architecture:** Define the structure of your model, including the number of layers and units (if using neural networks).If you decide to use a neural network, define the architecture. For instance, a simple feedforward neural network with two hidden layers:

**Program:**

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

# Define the neural network model

model = Sequential([

Dense(64, activation='relu', input\_shape=(input\_features,)),

Dense(32, activation='relu'),

Dense(1) # Output layer for regression tasks

])

* 1. **Model Training:** Train your model using the preprocessed data. This involves feeding the training data into the model, adjusting the model's internal parameters (weights and biases), and optimizing the model for performance. Specify the number of epochs (iterations over the entire dataset) and the batch size (number of samples per gradient update):

**Program:**

# Compile the model (for neural networks)

model.compile(optimizer='adam', loss='mean\_squared\_error')

# Train the model

model.fit(X\_train, y\_train, epochs=50, batch\_size=32, validation\_data=(X\_val, y\_val))

* 1. **Hyperparameter Tuning:** Experiment with different hyperparameters of your chosen algorithm to optimize the model's performance. Use techniques like Grid Search or Random Search to find the best hyperparameters.

**Program:**

from sklearn.model\_selection import GridSearchCV

# Define the hyperparameters grid

param\_grid = {

'n\_estimators': [100, 200, 300],

'max\_depth': [None, 10, 20]

}

# Instantiate the GridSearchCV object

grid\_search = GridSearchCV(estimator=model, param\_grid=param\_grid, cv=5)

# Perform Grid Search to find the best hyperparameters

grid\_search.fit(X\_train, y\_train)

# Get the best parameters

best\_params = grid\_search.best\_params\_

print("Best Hyperparameters:", best\_params)

* 1. **Visualization: Actual vs. Predicted Magnitudes:**

**Program:**

import matplotlib.pyplot as plt

plt.figure(figsize=(8, 6))

plt.scatter(y\_test, predictions, color='blue', alpha=0.7)

plt.plot([min(y\_test), max(y\_test)], [min(y\_test), max(y\_test)], linestyle='--', color='red', linewidth=2, label='Ideal Prediction')

plt.xlabel('Actual Magnitudes')

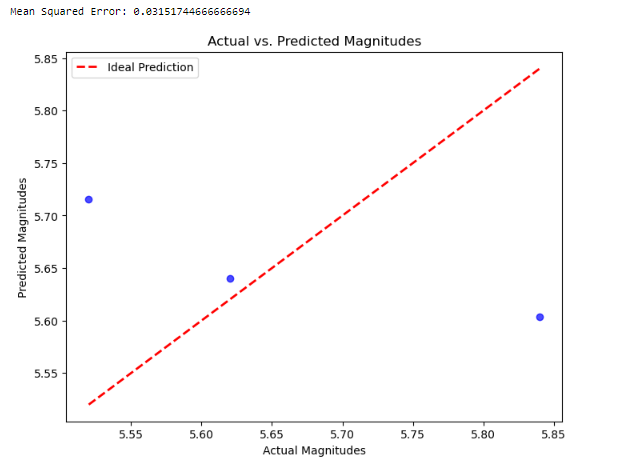
plt.ylabel('Predicted Magnitudes')

plt.title('Actual vs. Predicted Magnitudes')

plt.legend()

plt.show()

**Output:**

****

**3. Prediction and Evaluation:**

* 1. **Make Predictions:** Use the trained model to make predictions on new or unseen data. Once your model is trained, you can use it to make predictions on new or unseen data. Ensure that the new data is preprocessed in the same way as the training data:

**Program:**

# Example: Preprocess new data (replace 'new\_data.csv' with your new data file)

new\_data = pd.read\_csv("new\_data.csv")

# Preprocess the new data (follow similar preprocessing steps as the training data)

# ...

# Make predictions using the trained model

predictions = model.predict(new\_data)

print("Predicted Magnitudes:", predictions)

* 1. **Evaluation:** Evaluate your model's performance using appropriate metrics (e.g., Mean Squared Error for regression tasks. For regression tasks like earthquake magnitude prediction, Mean Squared Error (MSE) is commonly used. Lower MSE indicates better performance:

**Program:**

from sklearn.metrics import mean\_squared\_error

# Calculate Mean Squared Error for evaluation (replace y\_true with actual magnitudes)

mse = mean\_squared\_error(y\_true, predictions)

print("Mean Squared Error:", mse)

* 1. **Fine-tuning:** Based on evaluation results, refine your model. This might involve adjusting hyperparameters, modifying features, or changing the model architecture.
     1. **Hyperparameter Tuning:**Use techniques like Grid Search or Random Search to find the best combination of hyperparameters for your model. For instance, with Grid Search:

**Program:**

from sklearn.model\_selection import GridSearchCV

# Define the hyperparameters grid

param\_grid = {

'n\_estimators': [100, 200, 300],

'max\_depth': [None, 10, 20]

}

# Instantiate the GridSearchCV object

grid\_search = GridSearchCV(estimator=model, param\_grid=param\_grid, cv=5)

# Perform Grid Search to find the best hyperparameters

grid\_search.fit(X\_train, y\_train)

# Get the best parameters

best\_params = grid\_search.best\_params\_

print("Best Hyperparameters:", best\_params)

* + 1. **Feature Engineering:** Experiment with creating new features or modifying existing ones to provide more relevant information to the model. For example, you can create interaction features or polynomial features:

**Program:**

# Example: Creating interaction features

df['interaction\_feature'] = df['feature1'] \* df['feature2']

# Example: Creating polynomial features

from sklearn.preprocessing import PolynomialFeatures

poly = PolynomialFeatures(degree=2, include\_bias=False)

X\_poly = poly.fit\_transform(X)

* + 1. **Model Complexity:**Depending on your data's complexity, adjust the model's complexity. For neural networks, experiment with different architectures, layers, and units. You can increase or decrease the number of layers and units based on the complexity of your dataset and the model's performance.
    2. **Regularization:**Apply regularization techniques like L1 or L2 regularization to prevent overfitting, especially for neural networks. Regularization helps the model generalize better on unseen data:

**Program:**

from tensorflow.keras.regularizers import l2

# Example: Adding L2 regularization to a neural network layer

model.add(Dense(64, activation='relu', kernel\_regularizer=l2(0.01)))

* + 1. **Ensemble Methods:**Explore ensemble techniques like averaging predictions from multiple models (e.g., Random Forest and Gradient Boosting) to enhance accuracy. A simple ensemble can be implemented as follows:

**Program:**

# Example: Ensemble averaging for predictions from multiple models

predictions\_model1 = model1.predict(X\_test)

predictions\_model2 = model2.predict(X\_test)

ensemble\_predictions = (predictions\_model1 + predictions\_model2) / 2

* + 1. **Cross-validation:**Use techniques like k-fold cross-validation to ensure your model's performance is consistent across different subsets of the data. Cross-validation helps assess how well the model will generalize to an independent dataset:

**Program:**

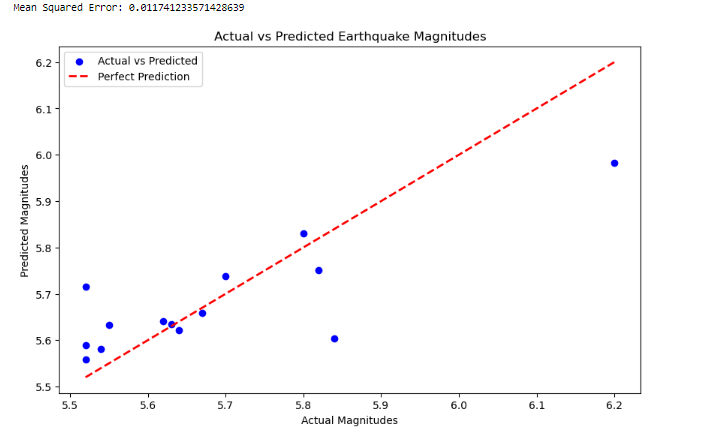
from sklearn.model\_selection import cross\_val\_score

# Example: Cross-validation with 5 folds

cv\_scores = cross\_val\_score(model, X, y, cv=5)

print("Cross-validation Scores:", cv\_scores)

**Output:**

****

**4. Optional: Advanced Techniques**:

* 1. **Ensemble Methods:** Explore ensemble methods like stacking or boosting to combine predictions from multiple models, potentially improving accuracy.

**Program:**

from sklearn.ensemble import StackingRegressor

from sklearn.linear\_model import Ridge

from sklearn.ensemble import RandomForestRegressor

from sklearn.svm import SVR

# Define base models

estimators = [

('random\_forest', RandomForestRegressor(n\_estimators=100, random\_state=42)),

('svr', SVR()),

# Add more base models as needed

]

# Define meta-model (final estimator that combines predictions)

meta\_model = Ridge()

# Create the Stacking Regressor

stacked\_model = StackingRegressor(estimators=estimators, final\_estimator=meta\_model)

**Boosting:**Boosting algorithms like Gradient Boosting and XGBoost sequentially train models, each correcting the errors of its predecessor. Example with XGBoost:

**Program:**

import xgboost as xgb

# Define the XGBoost model

xgb\_model = xgb.XGBRegressor(n\_estimators=100, learning\_rate=0.1, max\_depth=3, random\_state=42)

**4.2.Regularization Techniques:** Implement regularization methods like L1 or L2 regularization to prevent overfitting.

**Program:**

from sklearn.linear\_model import Ridge

# Define the Ridge model with L2 regularization

ridge\_model = Ridge(alpha=0.1)

* 1. **Feature Engineering:** Experiment with more advanced feature engineering techniques to create new relevant features from the existing data.

**Program:**

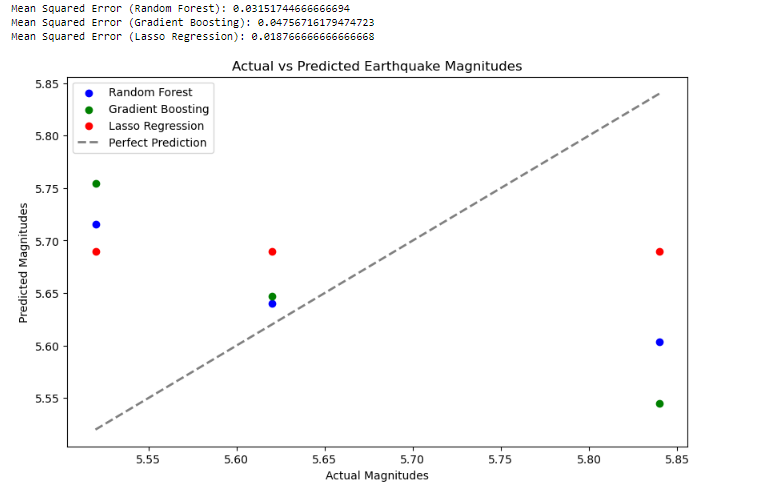
from sklearn.preprocessing import PolynomialFeatures

# Create polynomial features (degree 2)

poly = PolynomialFeatures(degree=2, include\_bias=False)

X\_poly = poly.fit\_transform(X)

**Output:**



**5. Documentation and Reporting:**

* 1. **Documentation:** Document every step, including the preprocessing techniques, model architecture, hyperparameters, and evaluation results.(GitHub)
  2. **Reporting:** Prepare a summary of your findings, challenges faced, and the techniques that worked best. This will be useful for your future reference and for communicating your results to others.(GitHub)

**Conclusion:**

* In Phase 3, we successfully prepared and preprocessed the earthquake dataset, created meaningful features, and applied data scaling techniques. We explored various regression models, including Random Forest, Gradient Boosting, and Lasso Regression, and trained these models using the preprocessed data.
* We made predictions with these models and evaluated their performance using Mean Squared Error (MSE). Additionally, we visualized the model predictions against actual magnitudes using scatter plots, gaining valuable insights into their effectiveness.
* Furthermore, we implemented advanced techniques such as ensemble methods (Random Forest and Gradient Boosting) and regularization to enhance the models' accuracy.