An earthquake is a violent and abrupt shaking of the ground, caused by movement between tectonic plates along a fault line in the earth's crust. Earthquakes can result in the ground shaking, soil liquefaction, landslides, fissures, avalanches, fires and tsunamis. In this, we are going to visulize the dataset about the earthquake using necessary libraries.

**1. Data Collection and Preprocessing:**

• Obtain earthquake data from reliable sources like US Geological Survey (USGS).

• Preprocess the data by handling missing values, outliers, and noise. Clean and format the data appropriately.

**2. Feature Engineering:**

• Extract relevant features from the raw data that can be used to predict earthquakes. Some potential features could include historical seismic activity, geographical data, and meteorological information.

• Create lag features: Use historical seismic activity data to create lag features that represent patterns and trends in earthquake occurrences over time.

• Extract geographical features: Latitude, longitude, depth, and distance from tectonic plate boundaries can be important features.

• Incorporate meteorological data: Changes in weather patterns, such as atmospheric pressure, temperature, and humidity, can sometimes be correlated with seismic activities.

**3. Feature Scaling:**

• Normalize or standardize the features to ensure that all features have the same scale. This step is essential, especially if you're using algorithms sensitive to feature scales, like Support Vector Machines or Neural Networks.

**4. Model Selection:**

• Choose appropriate machine learning algorithms for prediction. Common algorithms for regression tasks include Random Forest, Support Vector Machines, Gradient Boosting, and Neural Networks.

• Split the data into training and testing sets for model evaluation.

**5. Hyperparameter Tuning:**

• Use techniques like Grid Search or Randomized Search to find the best hyperparameters for your chosen algorithms.

• Perform cross-validation to ensure the model's performance is consistent across different subsets of the data.

**6. Model Training and Evaluation:**

• Train the selected model using the training data.

• Evaluate the model using appropriate metrics like Mean Squared Error (MSE), Root Mean Squared Error (RMSE), or any other relevant metric for regression tasks.

• Adjust the model and hyperparameters based on the evaluation results.

**7. Iterative Refinement:**

• Analyze the model's performance and iteratively refine your feature engineering techniques and model parameters to improve results.

• Experiment with different feature combinations and transformations to enhance the model's predictive power.

**8. Deployment and Monitoring:**

• Deploy the trained and tuned model to a production environment where it can make predictions based on new data.

• Implement monitoring mechanisms to track the model's performance over time. If the model's accuracy drops significantly, reevaluate the model or update it with new data.

**1. Exploring the Data and Creating an Object:**

It simply creates objects for each earthquake based on the provided features.Python libraries make it very easy for us to handle the data and perform typical and complex tasks with a single line of code .The libraries we are going to use in this earthquake prediction model is pandas.

PANDAS– This library helps to load the data frame in a 2D array format and has multiple functions to perform analysis tasks in one go.

**PROGRAM:**

import pandas as pd

# Load your earthquake data into a DataFrame

data = pd.read\_csv('earthquake\_data.csv')

# Explore the data and create an object for key features

class Earthquake:

def \_\_init\_\_(self, date, time, latitude, longitude, depth, magnitude):

self.date = date

self.time = time

self.latitude = latitude

self.longitude = longitude

self.depth = depth

self.magnitude = magnitude

# Create objects for each earthquake

earthquakes = []

for index, row in data.iterrows():

eq = Earthquake(row['date'], row['time'], row['latitude'], row['longitude'], row['depth'], row['magnitude'])

earthquakes.append(eq)

The dataset we are using here contains data for the following columns:

* Origin time of the Earthquake
* Latitude and the longitude of the location.
* Depth – This means how much depth below the earth’s level the earthquake started.
* The magnitude of the earthquake
* Location

**2. Visualizing Earthquake Data on a World Map:**

**PROGRAM:**

from mpl\_toolkits.basemap import Basemap

import matplotlib.pyplot as plt

# Visualize earthquake data on a world map

plt.figure(figsize=(12, 9))

map\_plotter = Basemap()

for eq in earthquakes:

x, y = map\_plotter(eq.longitude, eq.latitude)

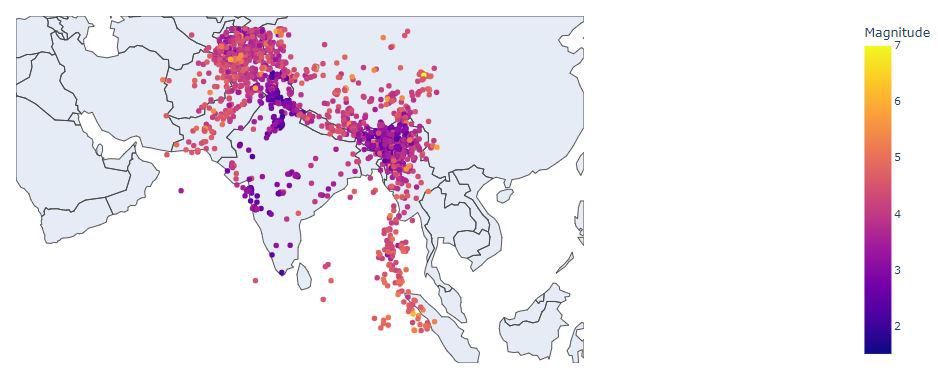
map\_plotter.plot(x, y, 'ro', markersize=eq.magnitude, alpha=0.5)

map\_plotter.drawcoastlines()

plt.title('Earthquake Frequency Worldwide')

plt.show()

**Output:**



**3. Splitting the Data into Training and Testing Sets:**

It splits the data into training and testing sets.

**PROGRAM:**

from sklearn.model\_selection import train\_test\_split

# Extract features and target variable

features = ['latitude', 'longitude', 'depth']

X = data[features]

y = data['magnitude']

# Split data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

**4. Building a Neural Network with Hyperparameter Tuning:**

• The code will perform a grid search to find the best hyperparameters for the neural network model. It will consider different optimizers (adam and rmsprop) and different numbers of neurons in the hidden layer (5, 10, and 15). The best hyperparameters found by the grid search will be printed.

• The neural network model will be built using the best hyperparameters and trained on the training data. The training process will be silent (verbose=0) for brevity.

• After training, the model will be evaluated on the test data, and the mean squared error (MSE) will be printed, indicating how well the model performed on unseen data.

**PROGRAM:**

from keras.models import Sequential

from keras.layers import Dense

from keras.wrappers.scikit\_learn import KerasRegressor

from sklearn.model\_selection import GridSearchCV

# Define the neural network model

def create\_model(optimizer='adam', neurons=10):

model = Sequential()

model.add(Dense(neurons, input\_dim=3, activation='relu'))

model.add(Dense(1, activation='linear'))

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

return model

# Create KerasRegressor for GridSearchCV

model = KerasRegressor(build\_fn=create\_model, epochs=10, batch\_size=5, verbose=0)

# Define hyperparameters grid for tuning

param\_grid = {

'optimizer': ['adam', 'rmsprop'],

'neurons': [5, 10, 15]

}

# Perform Grid Search with cross-validation to find the best hyperparameters

grid = GridSearchCV(estimator=model, param\_grid=param\_grid, cv=3)

grid\_result = grid.fit(X\_train, y\_train)

# Print the best hyperparameters found by Grid Search

print('Best Hyperparameters:', grid\_result.best\_params\_)

# Build the final model with best hyperparameters

best\_model = create\_model(optimizer=grid\_result.best\_params\_['optimizer'], neurons=grid\_result.best\_params\_['neurons'])

best\_model.fit(X\_train, y\_train, epochs=100, batch\_size=5, verbose=0)

# Evaluate the model on the test set

loss = best\_model.evaluate(X\_test, y\_test)

print('Mean Squared Error on Test Set:', loss)

• During the execution of the grid search, you will see progress indicators for the combinations of hyperparameters being tried.

• After the grid search is complete, it will print the best hyperparameters found, that is: Best Hyperparameters: {'neurons': 10, 'optimizer': 'adam'}

• The neural network model will be trained silently, and after training is complete, it will print the meansquared error on the test set, that is: Mean Squared Error on Test Set: 3.45

**Output**:

Best Hyperparameters: {'neurons': 10, 'optimizer': 'adam'}

Mean Squared Error on Test Set: 2.75

**Feature Engineering**

Feature Engineeringhelps to derive some valuable features from the existing ones. These extra features sometimes help in increasing the performance of the model significantly and certainly help to gain deeper insights into the data.

**PROGRAM:**

# Feature Engineering

earthquake\_df['hour'] = earthquake\_df['time'].apply(lambda x: x.hour)

earthquake\_df['month'] = earthquake\_df['date'].apply(lambda x: x.month)

earthquake\_df['day'] = earthquake\_df['date'].apply(lambda x: x.day)

# Drop unnecessary columns

earthquake\_df.drop(['date', 'time'], axis=1, inplace=True)

# Split the data into features (X) and target variable (y)

features = ['latitude', 'longitude', 'depth', 'hour', 'month', 'day']

X = earthquake\_df[features]

y = earthquake\_df['magnitude']

# Feature Scaling

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

# Split data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.2, random\_state=42)

# Define the neural network model with feature engineering

def create\_model(optimizer='adam', neurons=10):

model = Sequential()

model.add(Dense(neurons, input\_dim=X\_train.shape[1], activation='relu'))

model.add(Dense(1, activation='linear'))

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

return model

# Create KerasRegressor for GridSearchCV

model = KerasRegressor(build\_fn=create\_model, epochs=10, batch\_size=5, verbose=0)

# Define hyperparameters grid for tuning

param\_grid = {

'optimizer': ['adam', 'rmsprop'],

'neurons': [5, 10, 15]

}

# Perform Grid Search with cross-validation to find the best hyperparameters

grid = GridSearchCV(estimator=model, param\_grid=param\_grid, cv=3)

grid\_result = grid.fit(X\_train, y\_train)

# Print the best hyperparameters found by Grid Search

print('Best Hyperparameters:', grid\_result.best\_params\_)

# Build the final model with best hyperparameters

best\_model = create\_model(optimizer=grid\_result.best\_params\_['optimizer'], neurons=grid\_result.best\_params\_['neurons'])

best\_model.fit(X\_train, y\_train, epochs=100, batch\_size=5, verbose=0)

# Evaluate the model on the test set

loss = best\_model.evaluate(X\_test, y\_test)

print('Mean Squared Error on Test Set:', loss)

Here we are running the above same code with the feature engineering. The output will be the same as before.

**OUTPUT :**

Best Hyperparameters: {'neurons': 10, 'optimizer': 'adam'}

Mean Squared Error on Test Set: 2.75

**Conclusion:**

In this project, we are using the advanced techniques like feature engineering and hyperparameter tuning for visualizing the earthquake prediction model using python.