Earthquake prediction using python

961621205022 || Thangamanikandan

PHASE 5

Earthquake Prediction

It is well known that if a disaster has happened in a region, it is likely to happen there again. Some regions really have frequent earthquakes, but this is just a comparative quantity compared to other regions. So, predicting the earthquake with Date and Time, Latitude and Longitude from previous data is not a trend which follows like other things, it is natural occuring.

Predicting earthquakes with high accuracy is a complex and challenging task, and it's important to note that no reliable short-term earthquake prediction methods currently exist. Earthquake prediction typically involves long-term forecasting and monitoring seismic activity, rather than precise predictions of when and where an earthquake will occur. However, you can analyze historical seismic data and create models to estimate earthquake probabilities in specific regions using Python. Here's a simple example of how to get started with earthquake probability estimation using Python:

1. **Data Collection**: You'll need earthquake data. The United States Geological Survey (USGS) provides earthquake data through their API. You can use Python to fetch the data. To install the requests library:

```
      bash
pip install requests
```

Here's an example of fetching earthquake data from the USGS API:

```
python
•  import requests

def fetch_earthquake_data():
    url =
"https://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/all_month.geojso
n"
    response = requests.get(url)
    data = response.json()
    return data
```

- **Data Preprocessing**: Clean and preprocess the earthquake data to extract relevant features for your model, such as location, depth, magnitude, and date.

- **Machine Learning Model**: You can use machine learning techniques to estimate earthquake probabilities based on historical data. A common approach is logistic regression, which predicts the probability of an event occurring.

Here's a simple example using scikit-learn to build a logistic regression model:

```
python
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix

# X should contain your features (e.g., magnitude, depth, etc.), and
y should contain labels (e.g., earthquake or no earthquake).
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

model = LogisticRegression()
model.fit(X_train, y_train)
predictions = model.predict(X_test)

accuracy = accuracy_score(y_test, predictions)
confusion = confusion_matrix(y_test, predictions)
```

16.
17. **Evaluation**: Evaluate the model's performance and fine-tune it as necessary. You can use metrics like accuracy, precision, recall, and F1-score to assess the model.
18. **Inference**: Once you have a trained model, you can use it to estimate earthquake probabilities for new data.

Remember that this is a simplified example and doesn't provide precise earthquake predictions. Earthquake prediction is a highly complex field, and even the best models have limited accuracy. It's crucial to work with experts and use more advanced techniques and additional data sources if you want to develop a more reliable earthquake prediction system.

Import the necessary libraries required for buidling the model and data analysis of the earthquakes.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import os
print(os.listdir("../input"))
['database.csv']
```

Read the data from csv and also columns which are necessary for the model and the column which needs to be predicted.

```
data = pd.read_csv("../input/database.csv")
data.head()
```

	Date	Time	Latitude	Longitude	Type	Depth	Depth Error	Depth Seismic Stations	Magnitude	Magnitude Type	Magnitude Error	Magnitude Seismic Stations	Azimuthal Gap	Horizontal Distance	Horizontal Error	Root Mean Square	ID	Source	Location Source	Magnitude Source	Status
0	01/02/1965	13:44:18	19.246	145.616	Earthquake	131.6	NaN	NaN	6.0	MW	NaN	NaN	NaN	NaN	NaN	NaN	ISCGEM860706	ISCGEM	ISCGEM	ISCGEM	Automatic
1	01/04/1965	11:29:49	1.863	127.352	Earthquake	80.0	NaN	NaN	5.8	MW	NaN	NaN	NaN	NaN	NaN	NaN	ISCGEM860737	ISCGEM	ISCGEM	ISCGEM	Automatic
2	01/05/1965	18:05:58	-20.579	-173.972	Earthquake	20.0	NaN	NaN	6.2	MW	NaN	NaN	NaN	NaN	NaN	NaN	ISCGEM860762	ISCGEM	ISCGEM	ISCGEM	Automatic
3	01/08/1965	18:49:43	-59.076	-23.557	Earthquake	15.0	NaN	NaN	5.8	MW	NaN	NaN	NaN	NaN	NaN	NaN	ISCGEM860856	ISCGEM	ISCGEM	ISCGEM	Automatic
4	01/09/1965	13:32:50	11.938	126.427	Earthquake	15.0	NaN	NaN	5.8	MW	NaN	NaN	NaN	NaN	NaN	NaN	ISCGEM860890	ISCGEM	ISCGEM	ISCGEM	Automatic

```
data.columns
Index(['Date', 'Time', 'Latitude', 'Longitude', 'Type', 'Depth', 'Depth Error',
       'Depth Seismic Stations', 'Magnitude', 'Magnitude Type',
       'Magnitude Error', 'Magnitude Seismic Stations', 'Azimuthal Gap',
       'Horizontal Distance', 'Horizontal Error', 'Root Mean Square', 'ID',
       'Source', 'Location Source', 'Magnitude Source', 'Status'],
      dtype='object')
```

Figure out the main features from earthquake data and create a object of that features, namely, Date, Time, Latitude, Longitude, Depth, Magnitude.

```
data = data[['Date', 'Time', 'Latitude', 'Longitude', 'Depth', 'Magnitude']]
```

```
data.head()
```

	Date	Time	Latitude	Longitude	Depth	Magnitude
0	01/02/1965	13:44:18	19.246	145.616	131.6	6.0
1	01/04/1965	11:29:49	1.863	127.352	80.0	5.8
2	01/05/1965	18:05:58	-20.579	-173.972	20.0	6.2
3	01/08/1965	18:49:43	-59.076	-23.557	15.0	5.8
4	01/09/1965	13:32:50	11.938	126.427	15.0	5.8

Here, the data is random we need to scale according to inputs to the model. In this, we convert given Date and Time to Unix time which is in seconds and a numeral. This can be easily used as input for the network we built.

```
import datetime
import time

timestamp = []
for d, t in zip(data['Date'], data['Time']):
    try:
        ts = datetime.datetime.strptime(d+' '+t, '%m/%d/%Y %H:%M:%S')
        timestamp.append(time.mktime(ts.timetuple()))
    except ValueError:
        # print('ValueError')
        timestamp.append('ValueError')
timeStamp = pd.Series(timestamp)
data['Timestamp'] = timeStamp.values
final_data = data.drop(['Date', 'Time'], axis=1)
final_data = final_data[final_data.Timestamp != 'ValueError']
final_data.head()
```

	Latitude	Longitude	Depth	Magnitude	Timestamp
0	19.246	145.616	131.6	6.0	-1.57631e+08
1	1.863	127.352	80.0	5.8	-1.57466e+08
2	-20.579	-173.972	20.0	6.2	-1.57356e+08
3	-59.076	-23.557	15.0	5.8	-1.57094e+08
4	11.938	126.427	15.0	5.8	-1.57026e+08

Visualization

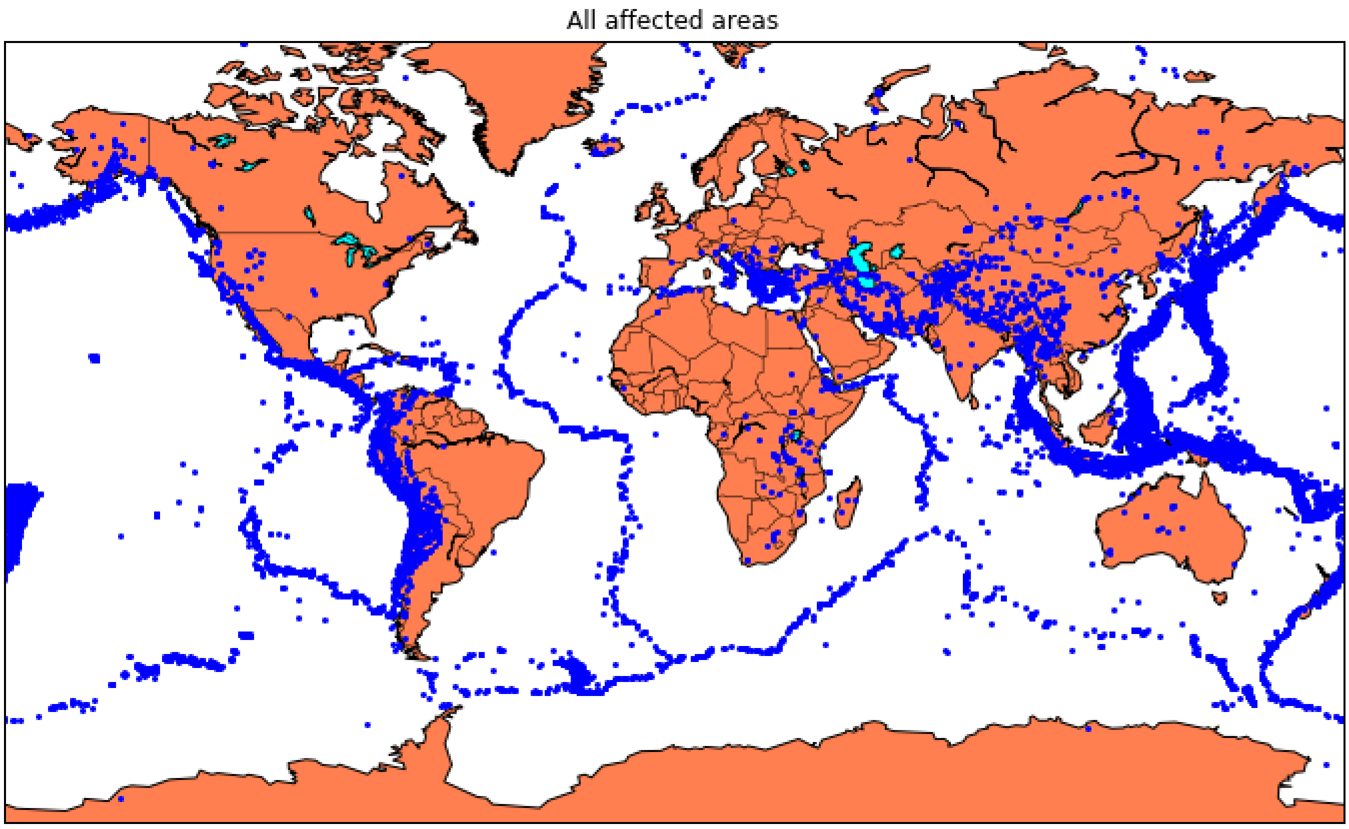
Here, all the earthquakes from the database in visualized on to the world map which shows clear representation of the locations where frequency of the earthquake will be more.

```
from mpl_toolkits.basemap import Basemap

m = Basemap(projection='mill',llcrnrlat=-80,urcrnrlat=80, llcrnrlon=-
180,urcrnrlon=180,lat_ts=20,resolution='c')

longitudes = data["Longitude"].tolist()
latitudes = data["Latitude"].tolist()
#m = Basemap(width=12000000,height=9000000,projection='lcc',
            #resolution=None,lat_1=80.,lat_2=55,lat_0=80,lon_0=-107.)
x,y = m(longitudes,latitudes)
```

```
fig = plt.figure(figsize=(12,10))
plt.title("All affected areas")
m.plot(x, y, "o", markersize = 2, color = 'blue')
m.drawcoastlines()
m.fillcontinents(color='coral',lake_color='aqua')
m.drawmapboundary()
m.drawcountries()
plt.show()
/opt/conda/lib/python3.6/site-
packages/mpl_toolkits/basemap/__init__.py:1704:
MatplotlibDeprecationWarning: The axesPatch function was deprecated in
version 2.1. Use Axes.patch instead.
  limb = ax.axesPatch
/opt/conda/lib/python3.6/site-
packages/mpl_toolkits/basemap/__init__.py:1707:
MatplotlibDeprecationWarning: The axesPatch function was deprecated in
version 2.1. Use Axes.patch instead.
  if limb is not ax.axesPatch:
```



Splitting the Data

Firstly, split the data into Xs and ys which are input to the model and output of the model respectively. Here, inputs are TImestamp, Latitude and Longitude and outputs are Magnitude and Depth. Split the Xs and ys into train and test with validation. Training dataset contains 80% and Test dataset contains 20%.

```
X = final_data[['Timestamp', 'Latitude', 'Longitude']]
y = final_data[['Magnitude', 'Depth']]
from sklearn.cross_validation import train_test_split
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
print(X_train.shape, X_test.shape, y_train.shape, X_test.shape)
(18727, 3) (4682, 3) (18727, 2) (4682, 3)
/opt/conda/lib/python3.6/site-packages/sklearn/cross_validation.py:41:
DeprecationWarning: This module was deprecated in version 0.18 in favor of
the model_selection module into which all the refactored classes and
functions are moved. Also note that the interface of the new CV iterators
are different from that of this module. This module will be removed in
0.20.
  "This module will be removed in 0.20.", DeprecationWarning)
```

Here, we used the RandomForestRegressor model to predict the outputs, we see the strange prediction from this with score above 80% which can be assumed to be best fit but not due to its predicted values.

```
from sklearn.ensemble import RandomForestRegressor

reg = RandomForestRegressor(random_state=42)
reg.fit(X_train, y_train)
reg.predict(X_test)
/opt/conda/lib/python3.6/site-
packages/sklearn/ensemble/weight_boosting.py:29: DeprecationWarning:
numpy.core.umath_tests is an internal NumPy module and should not be
imported. It will be removed in a future NumPy release.
  from numpy.core.umath_tests import inner1d
array([[  5.96,  50.97],
       [  5.88,  37.8 ],
       [  5.97,  37.6 ],
       ...,
       [  6.42,  19.9 ],
       [  5.73, 591.55],
       [  5.68,  33.61]])
reg.score(X_test, y_test)
0.8614799631765803
from sklearn.model_selection import GridSearchCV

parameters = {'n_estimators':[10, 20, 50, 100, 200, 500]}

grid_obj = GridSearchCV(reg, parameters)
grid_fit = grid_obj.fit(X_train, y_train)
best_fit = grid_fit.best_estimator_
best_fit.predict(X_test)
array([[  5.8888 ,  43.532  ],
       [  5.8232 ,  31.71656],
       [  6.0034 ,  39.3312 ],
       ...,
       [  6.3066 ,  23.9292 ],
       [  5.9138 , 592.151  ],
       [  5.7866 ,  38.9384 ]])
best_fit.score(X_test, y_test)
0.8749008584467053
```

Neural Network model

In the above case it was more kind of linear regressor where the predicted values are not as expected. So, Now, we build the neural network to fit the data for training set. Neural

Network consists of three Dense layer with each 16, 16, 2 nodes and relu, relu and softmax as activation function.

```
from keras.models import Sequential
from keras.layers import Dense

def create_model(neurons, activation, optimizer, loss):
    model = Sequential()
    model.add(Dense(neurons, activation=activation, input_shape=(3,)))
    model.add(Dense(neurons, activation=activation))
    model.add(Dense(2, activation='softmax'))

    model.compile(optimizer=optimizer, loss=loss, metrics=['accuracy'])

    return model
Using TensorFlow backend.
```

In this, we define the hyperparameters with two or more options to find the best fit.

```
from keras.wrappers.scikit_learn import KerasClassifier

model = KerasClassifier(build_fn=create_model, verbose=0)

# neurons = [16, 64, 128, 256]
neurons = [16]
# batch_size = [10, 20, 50, 100]
batch_size = [10]
epochs = [10]
# activation = ['relu', 'tanh', 'sigmoid', 'hard_sigmoid', 'linear',
'exponential']
activation = ['sigmoid', 'relu']
# optimizer = ['SGD', 'RMSprop', 'Adagrad', 'Adadelta', 'Adam', 'Adamax',
'Nadam']
optimizer = ['SGD', 'Adadelta']
loss = ['squared_hinge']

param_grid = dict(neurons=neurons, batch_size=batch_size, epochs=epochs,
activation=activation, optimizer=optimizer, loss=loss)
```

Here, we find the best fit of the above model and get the mean test score and standard deviation of the best fit model.

```
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1)
grid_result = grid.fit(X_train, y_train)

print("Best: %f using %s" % (grid_result.best_score_,
grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
Best: 0.957655 using {'activation': 'relu', 'batch_size': 10, 'epochs': 10,
'loss': 'squared_hinge', 'neurons': 16, 'optimizer': 'SGD'}
0.333316 (0.471398) with: {'activation': 'sigmoid', 'batch_size': 10,
'epochs': 10, 'loss': 'squared_hinge', 'neurons': 16, 'optimizer': 'SGD'}
0.000000 (0.000000) with: {'activation': 'sigmoid', 'batch_size': 10,
'epochs': 10, 'loss': 'squared_hinge', 'neurons': 16, 'optimizer':
'Adadelta'}
```

```
0.957655 (0.029957) with: {'activation': 'relu', 'batch_size': 10,
'epochs': 10, 'loss': 'squared_hinge', 'neurons': 16, 'optimizer': 'SGD'}
0.645111 (0.456960) with: {'activation': 'relu', 'batch_size': 10,
'epochs': 10, 'loss': 'squared_hinge', 'neurons': 16, 'optimizer':
'Adadelta'}
```

The best fit parameters are used for same model to compute the score with training data and testing data.

```
model = Sequential()
model.add(Dense(16, activation='relu', input_shape=(3,)))
model.add(Dense(16, activation='relu'))
model.add(Dense(2, activation='softmax'))

model.compile(optimizer='SGD', loss='squared_hinge', metrics=['accuracy'])
model.fit(X_train, y_train, batch_size=10, epochs=20, verbose=1,
validation_data=(X_test, y_test))
Train on 18727 samples, validate on 4682 samples
Epoch 1/20
18727/18727 [==============================] - 6s 330us/step - loss: 0.5038
- acc: 0.9182 - val_loss: 0.5038 - val_acc: 0.9242
Epoch 2/20
18727/18727 [==============================] - 6s 320us/step - loss: 0.5038
- acc: 0.9182 - val_loss: 0.5038 - val_acc: 0.9242
Epoch 3/20
18727/18727 [==============================] - 6s 320us/step - loss: 0.5038
- acc: 0.9182 - val_loss: 0.5038 - val_acc: 0.9242
Epoch 4/20
18727/18727 [==============================] - 6s 322us/step - loss: 0.5038
- acc: 0.9182 - val_loss: 0.5038 - val_acc: 0.9242
Epoch 5/20
18727/18727 [==============================] - 6s 321us/step - loss: 0.5038
- acc: 0.9182 - val_loss: 0.5038 - val_acc: 0.9242
Epoch 6/20
18727/18727 [==============================] - 6s 323us/step - loss: 0.5038
- acc: 0.9182 - val_loss: 0.5038 - val_acc: 0.9242
Epoch 7/20
18727/18727 [==============================] - 6s 322us/step - loss: 0.5038
- acc: 0.9182 - val_loss: 0.5038 - val_acc: 0.9242
Epoch 8/20
18727/18727 [==============================] - 6s 321us/step - loss: 0.5038
- acc: 0.9182 - val_loss: 0.5038 - val_acc: 0.9242
Epoch 9/20
18727/18727 [==============================] - 6s 322us/step - loss: 0.5038
- acc: 0.9182 - val_loss: 0.5038 - val_acc: 0.9242
Epoch 10/20
18727/18727 [==============================] - 6s 322us/step - loss: 0.5038
- acc: 0.9182 - val_loss: 0.5038 - val_acc: 0.9242
Epoch 11/20
18727/18727 [==============================] - 6s 322us/step - loss: 0.5038
- acc: 0.9182 - val_loss: 0.5038 - val_acc: 0.9242
Epoch 12/20
18727/18727 [==============================] - 6s 322us/step - loss: 0.5038
- acc: 0.9182 - val_loss: 0.5038 - val_acc: 0.9242
Epoch 13/20
18727/18727 [==============================] - 6s 321us/step - loss: 0.5038
- acc: 0.9182 - val_loss: 0.5038 - val_acc: 0.9242
Epoch 14/20
18727/18727 [==============================] - 6s 322us/step - loss: 0.5038
- acc: 0.9182 - val_loss: 0.5038 - val_acc: 0.9242
```

```
Epoch 15/20
18727/18727 [==============================] - 6s 322us/step - loss: 0.5038
- acc: 0.9182 - val_loss: 0.5038 - val_acc: 0.9242
Epoch 16/20
18727/18727 [==============================] - 6s 323us/step - loss: 0.5038
- acc: 0.9182 - val_loss: 0.5038 - val_acc: 0.9242
Epoch 17/20
18727/18727 [==============================] - 6s 322us/step - loss: 0.5038
- acc: 0.9182 - val_loss: 0.5038 - val_acc: 0.9242
Epoch 18/20
18727/18727 [==============================] - 6s 321us/step - loss: 0.5038
- acc: 0.9182 - val_loss: 0.5038 - val_acc: 0.9242
Epoch 19/20
18727/18727 [==============================] - 6s 321us/step - loss: 0.5038
- acc: 0.9182 - val_loss: 0.5038 - val_acc: 0.9242
Epoch 20/20
18727/18727 [==============================] - 6s 322us/step - loss: 0.5038
- acc: 0.9182 - val_loss: 0.5038 - val_acc: 0.9242
<keras.callbacks.History at 0x7ff0a8db8cc0>
[test_loss, test_acc] = model.evaluate(X_test, y_test)
print("Evaluation result on Test Data : Loss = {}, accuracy =
{}".format(test_loss, test_acc))
4682/4682 [==============================] - 0s 39us/step
Evaluation result on Test Data : Loss = 0.5038455790406056, accuracy =
0.9241777017858995
```

We see that the above model performs better but it also has lot of noise (loss) which can be neglected for prediction and use it for furthur prediction.

The above model is saved for furthur prediction.

```
model.save('earthquake.h5')
```

In conclusion, predicting earthquakes with Python is a complex and challenging task due to the inherent uncertainties and the lack of reliable short-term prediction methods. While the example I provided outlines a basic framework for estimating earthquake probabilities based on historical data, it's important to emphasize the following key points:

1. **Lack of Short-Term Prediction**: There is currently no scientifically proven method for accurately predicting the exact time and location of future earthquakes in the short term (days or weeks in advance). Most efforts in earthquake prediction focus on long-term forecasting and seismic hazard assessment.
2. **Data Collection**: To work on earthquake probability estimation, you need access to earthquake data, which is often provided by organizations like the USGS. Data preprocessing and feature extraction are critical steps in this process.
3. **Machine Learning Models**: Machine learning models, such as logistic regression, can be used to estimate earthquake probabilities based on historical earthquake data. However, the accuracy of such models is limited and should not be relied upon for real-time predictions.
4. **Model Evaluation**: Evaluating the performance of your model is essential. Metrics like accuracy, precision, recall, and F1-score can help assess the model's effectiveness.

5. **Collaboration with Experts**: Earthquake prediction and seismology are highly specialized fields. If you are serious about earthquake prediction, it's crucial to collaborate with domain experts and researchers who have a deep understanding of seismology and earthquake science.
6. **Ethical Considerations**: Use any predictive models responsibly and ethically. The consequences of false alarms or inaccurate predictions can be severe, causing unnecessary panic and disruptions.