

Earthquake Prediction Model Using Python

I will take you through how to create a model for the task of Earthquake Prediction using Machine Learning and the Python programming language. Predicting earthquakes is one of the great unsolved problems in the earth sciences with the increase in the use of technology, many seismic monitoring stations have increased, so we can use machine learning and other data-driven methods to predict earthquakes.

Earthquake Prediction Model with Machine Learning

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.

I will start this task to create a model for earthquake prediction by importing the necessary python libraries:

```
import numpy as np
```

```
import pandas as pd
```

```
import matplotlib.pyplot as plt
```

	data = pd.read_csv("database.csv")
	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')
```

Now let's see the main characteristics of earthquake data and create an object of these characteristics, 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

Since the data is random, so we need to scale it based on the model inputs. In this, we convert the given date and time to Unix time which is in seconds and a number.

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

Latitude	Longitude	Depth	Magnitude	Timestamp	
0	19.246	145.616	131.6	6.0	- 1.57631e +08

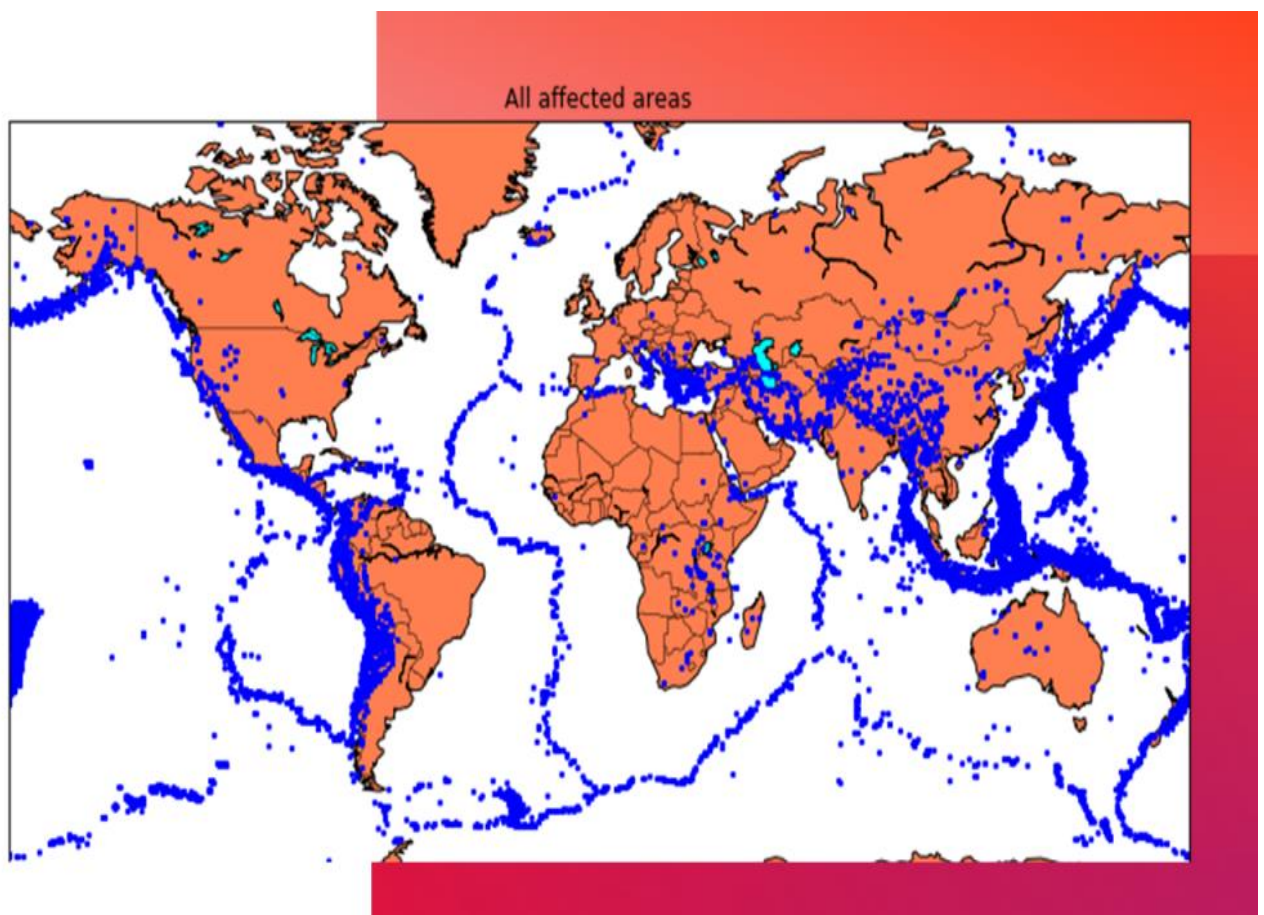
1	1.863	127.352	80.0	5.8	- 1.57466e +08
2	-20.579	-173.972	20.0	6.2	- 1.57356e +08
3	-59.076	-23.557	15.0	5.8	- 1.57094e +08
4	11.938	126.427	15.0	5.8	- 1.57026e +08

Data Visualization

Now, before we create the earthquake prediction model, let's visualize the data on a world map that shows a clear representation of where the earthquake frequency will be more:

	<code>from mpl_toolkits.basemap import Basemap</code>
	<code>m = Basemap(projection='mill',llcrnrlat=-80,urcnrlat=80, llcrnrlon=-180,urcnrlon=180,lat_ts=20,resolution='c')</code>
	<code>longitudes = data["Longitude"].tolist()</code>
	<code>latitudes = data["Latitude"].tolist()</code>
	<code>#m = Basemap(width=12000000,height=9000000,projection='lcc',</code>
	<code>#resolution=None,lat_1=80.,lat_2=55,lat_0=80,lon_0=-107.)</code>
	<code>x,y = m(longitudes,latitudes)</code>

<code>fig = plt.figure(figsize=(12,10))</code>
<code>plt.title("All affected areas")</code>
<code>m.plot(x, y, "o", markersize = 2, color = 'blue')</code>
<code>m.drawcoastlines()</code>
<code>m.fillcontinents(color='coral',lake_color='aqua')</code>
<code>m.drawmapboundary()</code>
<code>m.drawcountries()</code>
<code>plt.show()</code>



Splitting the Dataset

Now, to create the earthquake prediction model, we need to divide the data into Xs and ys which respectively will be entered into the model as inputs to receive the output from the model.

Here the inputs are Timestamp, Latitude and Longitude and the outputs are Magnitude and Depth. I'm going to split the xs and ys into train and test with validation. The training set contains 80% and the test set contains 20%:

<code>X = final_data[['Timestamp', 'Latitude', 'Longitude']]</code>
<code>y = final_data[['Magnitude', 'Depth']]</code>
<code>from sklearn.cross_validation import train_test_split</code>
<code>X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)</code>
<code>print(X_train.shape, X_test.shape, y_train.shape, X_test.shape)</code>

Neural Network for Earthquake Prediction

Now I will create a neural network to fit the data from the training set. Our neural network will consist of three dense layers each with 16, 16, 2 nodes and reread. Relu and softmax will be used as activation functions:

<code>from keras.models import Sequential</code>
<code>from keras.layers import Dense</code>

<code>def create_model(neurons, activation, optimizer, loss):</code>
<code>model = Sequential()</code>
<code>model.add(Dense(neurons, activation=activation, input_shape=(3,)))</code>
<code>model.add(Dense(neurons, activation=activation))</code>
<code>model.add(Dense(2, activation='softmax'))</code>
<code>model.compile(optimizer=optimizer, loss=loss, metrics=['accuracy'])</code>
<code>return model</code>

Now we need to find the best fit of the above model and get the mean test score and standard deviation of the best fit model:

<code>grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1)</code>
<code>grid_result = grid.fit(X_train, y_train)</code>
<code>print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))</code>
<code>means = grid_result.cv_results_['mean_test_score']</code>
<code>stds = grid_result.cv_results_['std_test_score']</code>
<code>params = grid_result.cv_results_['params']</code>
<code>for mean, stdev, param in zip(means, stds, params):</code>
<code>print("%f (%f) with: %r" % (mean, stdev, param))</code>

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': 'Adadelata'} 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': 'Adadelata'}
```

In the step below, the best-fit parameters are used for the same model to calculate the score with the training data and the test data:

	<code>model = Sequential()</code>
	<code>model.add(Dense(16, activation='relu', input_shape=(3,)))</code>
	<code>model.add(Dense(16, activation='relu'))</code>
	<code>model.add(Dense(2, activation='softmax'))</code>
	<code>model.compile(optimizer='SGD', loss='squared_hinge', metrics=['accuracy'])</code>
	<code>model.fit(X_train, y_train, batch_size=10, epochs=20, verbose=1, validation_data=(X_test, y_test))</code>
	<code>[test_loss, test_acc] = model.evaluate(X_test, y_test)</code>
	<code>print("Evaluation result on Test Data : Loss = {}, accuracy = {}".format(test_loss, test_acc))</code>

Evaluation result on Test Data : Loss = 0.5038455790406056, accuracy = 0.9241777017858995

So we can see in the above output that our neural network model for earthquake prediction performs well. I hope you liked this article on how to create an earthquake prediction model with machine learning and the Python programming language.

Feature engineering

Feature engineering means identifying the relationships between independent and dependent features. Then the identified relationships we can add as polynomial or interaction features.

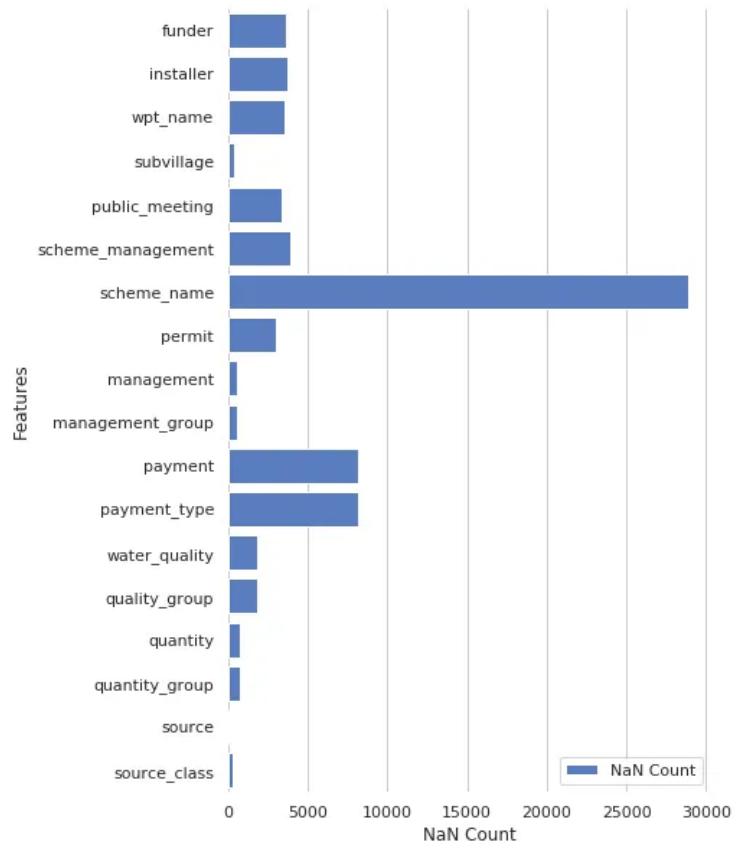
Feature engineering step is the point of entry for successive iterations. This is a critical step and plays a greater role in predictions as compared to model validation.

We can either drop columns with high cardinality and NaN values if needed. Also for numerical features, we can fill the NaN values with its mean, median or mode instead of dropping it. In the same way for categorical features, we can categorize the NaN as a separate category.

Also based on the water table data set, we can engineer few features like:

- NaN values in **longitude** and **latitude** updated with its mean.

Both numerical and categorical features have NaN



```
# Apply mean for missing values of latitude  
and longitude
```

```
mean_longitude =  
df['longitude'].mean()  
df['longitude'] =  
df['longitude'].apply(lambda x:  
mean_longitude if round(x, 2) == 0  
else x)  
mean_latitude = df['latitude'].mean()  
df['latitude'] =  
df['latitude'].apply(lambda x:  
mean_latitude if round(x, 2) == 0  
else x)
```

- Binary feature to identify if **funder** has at least funded 5 pumps.
In the same was identify if **installer** has installed at least 5 pumps.

	# Identify items who have funded atleast 5 pumps
	if str(row) == "nan":
	return np.nan
	value_count = value_count_series.get(row)
	if value_count < count:
	return 0
	else:
	return 1
	# Create a column to indicate funder with atleast 5 pumps maintained.
	value_count_funder = df.funder.value_counts()
	df['funder_aleast_5'] = df['funder'].apply(atleast,
	args=(value_count_funder,))
	# Create a column to indicate installer with atleast 5 pumps maintained.
	value_count_installer = df.installer.value_counts()
	df['installer_aleast_5'] = df['installer'].apply(atleast,
	args=(value_count_installer,))

- Split the **date_recorded** into **year_recorded** and **month_recorded**. Even group in different bins.

	# split year from date_recorded
	return int(row.split('-')[0])
	def compute_month_recorded(row):
	# split year from date_recorded
	return int(row.split('-')[1])
	# Fetch Year and Month of date recorded
	df['year_recorded'] = df['date_recorded'].apply(compute_year_recorded)
	df['month_recorded'] = df['date_recorded'].apply(compute_month_recorded)

<ul style="list-style-type: none"> Compute the age of pump based on construction_year and year_recorded. 	