NN and SVR implementation for Earthquake prediction

COGS118B 2022FA

Group members:

Andrina / Xiaoxuan Zhang, Alexander Huynh, Victorionna Tran, Nicolas Schaefer

presentation video: https://youtu.be/PW_n3AHvA10

presentation slides: https://docs.google.com/presentation/d/1pluFLWJl29U45WkxpcH_bu0kdAwriR8w6QYsswkTnms/edit

Background

Everything happening with nature has tremendous impacts on human beings. Earthquakes and natural disasters may take away countless people's lives. Earthquake prediction has been around for a while but has yielded no real success and our group wants to work on this topic too.

At the very beginning of the project, our group wanted to work on the time series data of earthquake waveforms, trying to predict how the wave may traverse to predict the earthquake's severity. However, the data we could find online was not very ideal to predict the waveform.

Fortunately, we are able to find a dataset that contains the Data, Time, Latitude, Longitude, Magnitude, Depth, and Region of the earthquake. The core components made us possible to work on the earthquake prediction model.

Research Question

Can we accurately predict the magnitude and depth of an earthquake happening, given the time and location (latitude and longitude) it will occur?

We would like to know the relationship between the time and location of earthquakes The general prediction and patterns on magnitude and depth of earthquakes may also apply to the decision of future urban planning and architect designs With the consideration of these components, the government may design different focuses based on different locations to decrease the effect of earthquakes. With further influence, this work’s importance pertains to the livelihood of people.

Data

In [1]:
```
import pandas as pd
import numpy as np
import datetime
import time
```

```
import matplotlib.pyplot as plt
import geopandas as gpd
```

In [2]:

```
import tensorflow as tf
import keras
```

```
2022-12-06 08:32:32.444173: W tensorflow/stream_executor/platform/default/dso_loa
der.cc:64] Could not load dynamic library 'libcudart.so.11.0'; dlerror: libcudar
t.so.11.0: cannot open shared object file: No such file or directory; LD_LIBRARY_
PATH: /usr/local/nvidia/lib:/usr/local/nvidia/lib64
2022-12-06 08:32:32.444211: I tensorflow/stream_executor/cuda/cudart_stub.cc:29]
Ignore above cudart dlerror if you do not have a GPU set up on your machine.
/opt/conda/lib/python3.9/site-packages/scipy/__init__.py:146: UserWarning: A NumP
y version >=1.16.5 and <1.23.0 is required for this version of SciPy (detected ve
rsion 1.23.5
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
```

In [3]:

```
from sklearn.model_selection import train_test_split, GridSearchCV, RandomizedSea
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPool2D
from keras.wrappers.scikit_learn import KerasClassifier
```

Importing data

In [4]:

```
df = pd.read_csv('clean25kDataset.csv')
```

In [5]:

```
timestamp_list = []
for d, t in zip(df['Date'], df['Time']):
    timestamp = datetime.datetime.strptime(d+' '+t, '%Y-%m-%d %H:%M:%S')
    timestamp_list.append(time.mktime(timestamp.timetuple()))

timeStamp = pd.Series(timestamp_list)
df['Timestamp'] = timeStamp.values
clean_df = df.drop(['Date', 'Time'], axis=1)

clean_df = clean_df.rename(columns={'Lat': 'Latitude', 'Lon': 'Longitude', 'Mag':
clean_df
```

Out[5]:

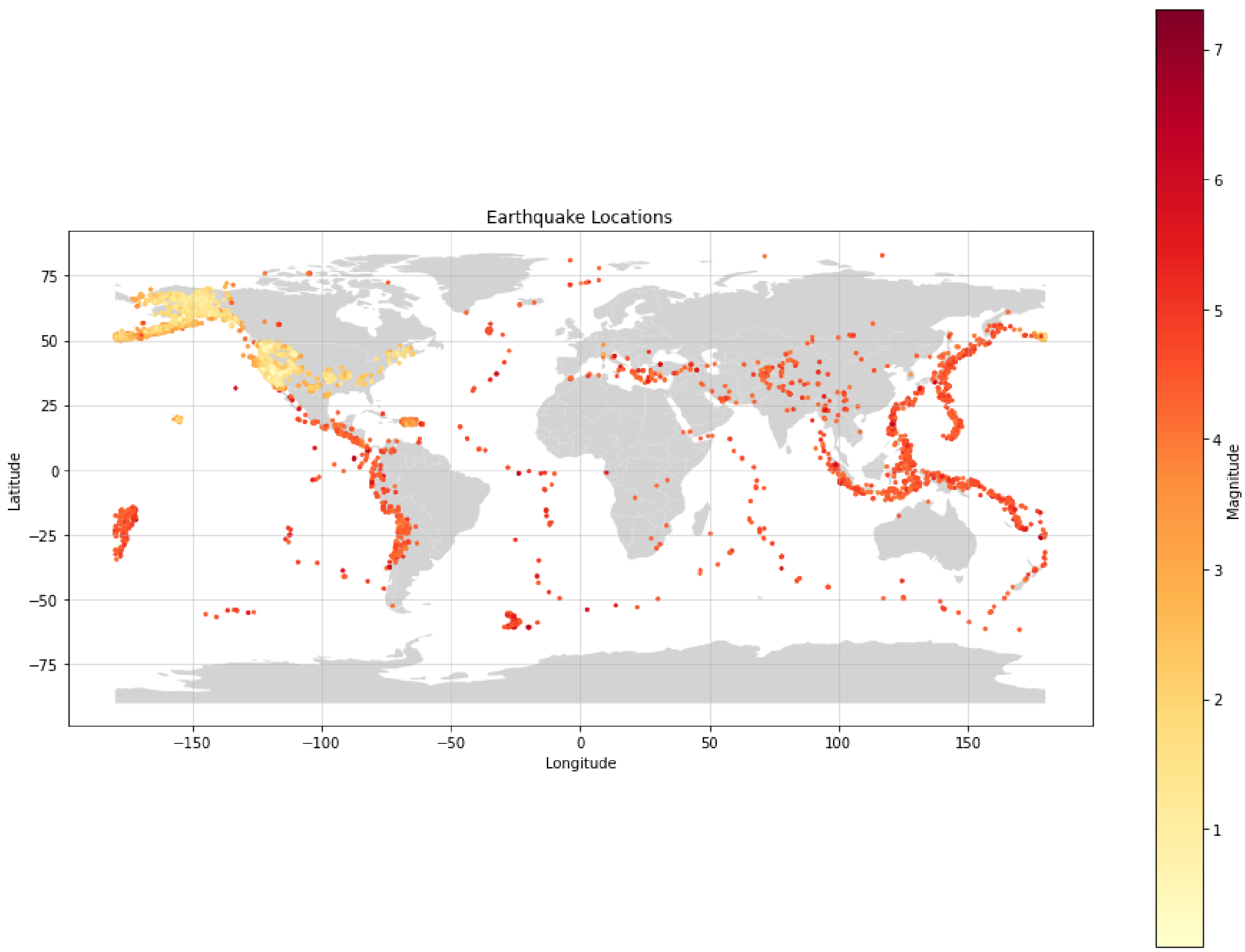
	Latitude	Longitude	Depth	Magnitude	Region	Timestamp
0	60.5758	-147.5620	15.1	2.6	57 km SW of Tatitlek, Alaska	1.670283e+09
1	37.3565	-121.7167	8.2	1.5	10km E of Alum Rock, CA	1.670283e+09
2	60.1315	-153.1349	125.6	1.9	66 km E of Port Alsworth, Alaska	1.670282e+09
3	37.3247	-121.6887	6.9	3.7	13km ESE of Alum Rock, CA	1.670282e+09
4	39.4327	-92.2425	4.7	2.5	5 km SSW of Madison, Missouri	1.670282e+09
...	...	...	...	...	...	...
24995	58.2855	-154.9823	3.8	0.5	85 km NNW of Karluk, Alaska	1.664120e+09
24996	51.3816	142.7739	10.0	4.8	51 km NE of Mgachi, Russia	1.664120e+09
24997	27.7017	56.4543	10.0	4.9	59 km NNE of Bandar Abbas, Iran	1.664120e+09

	Latitude	Longitude	Depth	Magnitude	Region	Timestamp
24998	35.3747	-118.1223	4.5	1.1	30km NNW of California City, CA	1.664119e+09
24999	53.7207	-162.6143	25.6	1.9	136 km SSE of False Pass, Alaska	1.664119e+09

25000 rows × 6 columns

In [6]:

```
countries = gpd.read_file(gpd.datasets.get_path("naturalearth_lowres"))

# initialize an axis
fig, ax = plt.subplots(figsize=(16,12))

# plot map on axis
countries = gpd.read_file(gpd.datasets.get_path("naturalearth_lowres"))
countries.plot(color="lightgrey",ax=ax)

# plot points
clean_df.plot(x="Longitude", y="Latitude", marker ='.', kind="scatter", c="Magnit
        title="Earthquake Locations", ax=ax)

# add grid
ax.grid(visible=True, alpha=0.5)

plt.show()
```



In [7]:

```
X = clean_df[['Timestamp', 'Latitude', 'Longitude']]
y = clean_df[['Magnitude', 'Depth']]
```

In [8]:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_s
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

```
(20000, 3) (5000, 3) (20000, 2) (5000, 2)
```

Implementing Neural Network

Model-1

Baseline Neural Network

In [9]:

```
def NN(neurons, drop, activation, optimizer, loss):
    model = Sequential()
    model.add(Dense(neurons, activation=activation, input_shape=(3,)))
    model.add(Dense(neurons, activation=activation))
    model.add(Dropout(rate=drop))
    model.add(Dense(2, activation='softmax'))

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

    return model
```

In [16]:

```
test = NN(32, 0.5, 'relu', 'SGD', 'squared_hinge')
test_plot = test.fit(X_train, y_train, batch_size= 64, epochs= 40, verbose=2, val

[test_loss, test_acc] = test.evaluate(X_test, y_test)
print("Evaluation result on Test Data : Loss = {}, accuracy = {}".format(test_los
```

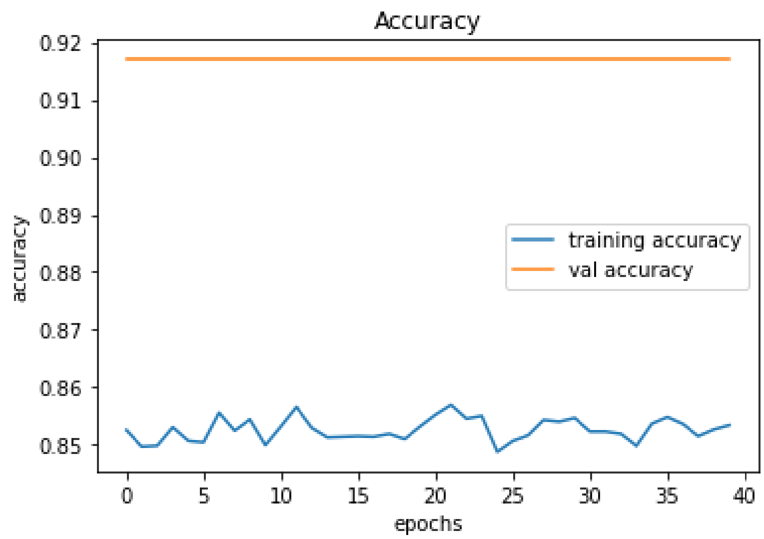
```
Epoch 1/40
313/313 - 1s - loss: 0.6147 - accuracy: 0.8526 - val_loss: 0.6419 - val_accuracy:
0.9170 - 1s/epoch - 4ms/step
Epoch 2/40
313/313 - 1s - loss: 0.6151 - accuracy: 0.8497 - val_loss: 0.6419 - val_accuracy:
0.9170 - 675ms/epoch - 2ms/step
Epoch 3/40
313/313 - 1s - loss: 0.6179 - accuracy: 0.8498 - val_loss: 0.6419 - val_accuracy:
0.9170 - 743ms/epoch - 2ms/step
Epoch 4/40
313/313 - 1s - loss: 0.6150 - accuracy: 0.8530 - val_loss: 0.6419 - val_accuracy:
0.9170 - 779ms/epoch - 2ms/step
Epoch 5/40
313/313 - 1s - loss: 0.6146 - accuracy: 0.8507 - val_loss: 0.6419 - val_accuracy:
0.9170 - 812ms/epoch - 3ms/step
Epoch 6/40
313/313 - 1s - loss: 0.6156 - accuracy: 0.8504 - val_loss: 0.6419 - val_accuracy:
0.9170 - 826ms/epoch - 3ms/step
Epoch 7/40
313/313 - 1s - loss: 0.6168 - accuracy: 0.8555 - val_loss: 0.6419 - val_accuracy:
0.9170 - 777ms/epoch - 2ms/step
Epoch 8/40
```

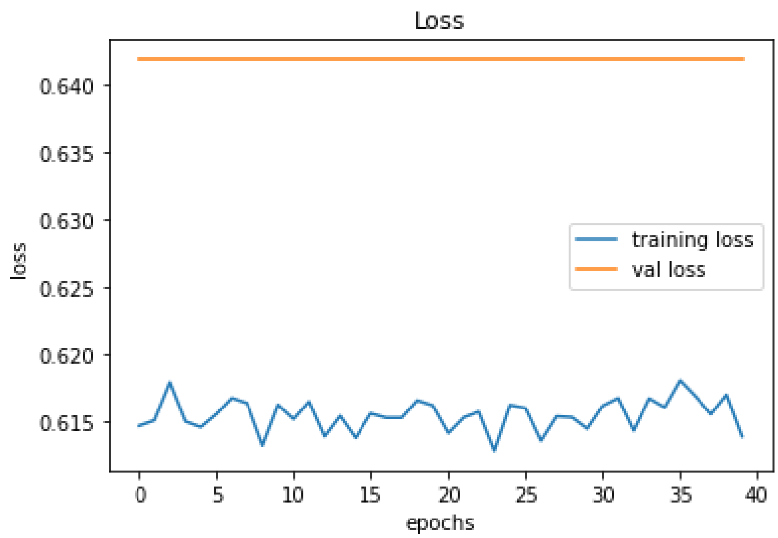
```
313/313 - 1s - loss: 0.6164 - accuracy: 0.8525 - val_loss: 0.6419 - val_accuracy:
0.9170 - 731ms/epoch - 2ms/step
Epoch 9/40
313/313 - 1s - loss: 0.6132 - accuracy: 0.8544 - val_loss: 0.6419 - val_accuracy:
0.9170 - 649ms/epoch - 2ms/step
Epoch 10/40
313/313 - 1s - loss: 0.6163 - accuracy: 0.8500 - val_loss: 0.6419 - val_accuracy:
0.9170 - 757ms/epoch - 2ms/step
Epoch 11/40
313/313 - 1s - loss: 0.6152 - accuracy: 0.8532 - val_loss: 0.6419 - val_accuracy:
0.9170 - 652ms/epoch - 2ms/step
Epoch 12/40
313/313 - 1s - loss: 0.6165 - accuracy: 0.8565 - val_loss: 0.6419 - val_accuracy:
0.9170 - 693ms/epoch - 2ms/step
Epoch 13/40
313/313 - 1s - loss: 0.6139 - accuracy: 0.8529 - val_loss: 0.6419 - val_accuracy:
0.9170 - 723ms/epoch - 2ms/step
Epoch 14/40
313/313 - 1s - loss: 0.6154 - accuracy: 0.8513 - val_loss: 0.6419 - val_accuracy:
0.9170 - 694ms/epoch - 2ms/step
Epoch 15/40
313/313 - 1s - loss: 0.6138 - accuracy: 0.8514 - val_loss: 0.6419 - val_accuracy:
0.9170 - 668ms/epoch - 2ms/step
Epoch 16/40
313/313 - 1s - loss: 0.6156 - accuracy: 0.8515 - val_loss: 0.6419 - val_accuracy:
0.9170 - 657ms/epoch - 2ms/step
Epoch 17/40
313/313 - 1s - loss: 0.6153 - accuracy: 0.8514 - val_loss: 0.6419 - val_accuracy:
0.9170 - 627ms/epoch - 2ms/step
Epoch 18/40
313/313 - 1s - loss: 0.6153 - accuracy: 0.8519 - val_loss: 0.6419 - val_accuracy:
0.9170 - 646ms/epoch - 2ms/step
Epoch 19/40
313/313 - 1s - loss: 0.6166 - accuracy: 0.8510 - val_loss: 0.6419 - val_accuracy:
0.9170 - 803ms/epoch - 3ms/step
Epoch 20/40
313/313 - 1s - loss: 0.6162 - accuracy: 0.8532 - val_loss: 0.6419 - val_accuracy:
0.9170 - 710ms/epoch - 2ms/step
Epoch 21/40
313/313 - 1s - loss: 0.6142 - accuracy: 0.8552 - val_loss: 0.6419 - val_accuracy:
0.9170 - 766ms/epoch - 2ms/step
Epoch 22/40
313/313 - 1s - loss: 0.6153 - accuracy: 0.8569 - val_loss: 0.6419 - val_accuracy:
0.9170 - 849ms/epoch - 3ms/step
Epoch 23/40
313/313 - 1s - loss: 0.6157 - accuracy: 0.8545 - val_loss: 0.6419 - val_accuracy:
0.9170 - 839ms/epoch - 3ms/step
Epoch 24/40
313/313 - 1s - loss: 0.6128 - accuracy: 0.8550 - val_loss: 0.6419 - val_accuracy:
0.9170 - 764ms/epoch - 2ms/step
Epoch 25/40
313/313 - 1s - loss: 0.6162 - accuracy: 0.8488 - val_loss: 0.6419 - val_accuracy:
0.9170 - 706ms/epoch - 2ms/step
Epoch 26/40
313/313 - 1s - loss: 0.6160 - accuracy: 0.8507 - val_loss: 0.6419 - val_accuracy:
0.9170 - 640ms/epoch - 2ms/step
Epoch 27/40
313/313 - 1s - loss: 0.6136 - accuracy: 0.8516 - val_loss: 0.6419 - val_accuracy:
0.9170 - 704ms/epoch - 2ms/step
Epoch 28/40
```

```
313/313 - 1s - loss: 0.6154 - accuracy: 0.8543 - val_loss: 0.6419 - val_accuracy:
0.9170 - 730ms/epoch - 2ms/step
Epoch 29/40
313/313 - 1s - loss: 0.6153 - accuracy: 0.8540 - val_loss: 0.6419 - val_accuracy:
0.9170 - 769ms/epoch - 2ms/step
Epoch 30/40
313/313 - 1s - loss: 0.6145 - accuracy: 0.8547 - val_loss: 0.6419 - val_accuracy:
0.9170 - 717ms/epoch - 2ms/step
Epoch 31/40
313/313 - 1s - loss: 0.6162 - accuracy: 0.8523 - val_loss: 0.6419 - val_accuracy:
0.9170 - 825ms/epoch - 3ms/step
Epoch 32/40
313/313 - 1s - loss: 0.6167 - accuracy: 0.8523 - val_loss: 0.6419 - val_accuracy:
0.9170 - 666ms/epoch - 2ms/step
Epoch 33/40
313/313 - 1s - loss: 0.6143 - accuracy: 0.8519 - val_loss: 0.6419 - val_accuracy:
0.9170 - 716ms/epoch - 2ms/step
Epoch 34/40
313/313 - 1s - loss: 0.6167 - accuracy: 0.8498 - val_loss: 0.6419 - val_accuracy:
0.9170 - 651ms/epoch - 2ms/step
Epoch 35/40
313/313 - 1s - loss: 0.6161 - accuracy: 0.8536 - val_loss: 0.6419 - val_accuracy:
0.9170 - 638ms/epoch - 2ms/step
Epoch 36/40
313/313 - 1s - loss: 0.6181 - accuracy: 0.8548 - val_loss: 0.6419 - val_accuracy:
0.9170 - 758ms/epoch - 2ms/step
Epoch 37/40
313/313 - 1s - loss: 0.6169 - accuracy: 0.8536 - val_loss: 0.6419 - val_accuracy:
0.9170 - 691ms/epoch - 2ms/step
Epoch 38/40
313/313 - 1s - loss: 0.6155 - accuracy: 0.8515 - val_loss: 0.6419 - val_accuracy:
0.9170 - 743ms/epoch - 2ms/step
Epoch 39/40
313/313 - 1s - loss: 0.6170 - accuracy: 0.8526 - val_loss: 0.6419 - val_accuracy:
0.9170 - 691ms/epoch - 2ms/step
Epoch 40/40
313/313 - 1s - loss: 0.6139 - accuracy: 0.8534 - val_loss: 0.6419 - val_accuracy:
0.9170 - 799ms/epoch - 3ms/step
157/157 [==============================] - 0s 742us/step - loss: 0.6419 - accurac
y: 0.9170
Evaluation result on Test Data : Loss = 0.6418639421463013, accuracy = 0.91699999
57084656
```

In [17]:

```
plt.figure(0)
plt.plot(test_plot.history['accuracy'], label='training accuracy')
plt.plot(test_plot.history['val_accuracy'], label='val accuracy')
plt.title('Accuracy')
plt.xlabel('epochs')
plt.ylabel('accuracy')
plt.legend()
plt.show()
plt.figure(1)
plt.plot(test_plot.history['loss'], label='training loss')
plt.plot(test_plot.history['val_loss'], label='val loss')
plt.title('Loss')
plt.xlabel('epochs')
plt.ylabel('loss')
plt.legend()

plt.show()
```





In the graphs we can see as the model is going through more epochs the training and validation are imporoving. We can also see a convergence of the accuracies in the later stages. We see that the loss is getting smaller and converging.

Baseline Neural Network Hyperparameter Tuning

With the parameters we estimate that might be a great input, we already have an decent accuracy. We will try hyper parameter tunning now to see if we can fine tune the model for better accuracy.

In [18]:

```
neurons = [32, 64, 128]
batches = [50, 100, 200]
dropout = [0.1, 0.2, 0.25, 0.5]
activation = ['relu', 'tanh', 'sigmoid', 'linear']
optimizer = ['SGD', 'RMSprop', 'Adagrad', 'Adadelta', 'Adam', 'Adamax', 'Nadam']
loss = ['categorical_crossentropy', 'poisson', 'kl_divergence', 'squared_hinge']
epochs = [40]
search_space = dict(neurons=neurons, batch_size = batches, drop = dropout, epochs
```

These will be the hyperparameters that we are going to tune our Neural network:

- Batches is the batch size of the data, it is the amount data points in a single partition that is going to be passed through our Neural Network, the higher the value the faster it runs.
- Dropout is the dropout rate of the data, it is primarily used to compensate for overfitting, we are going to use 1 dropout rate for hypertuning.
- Filter size is the size of our filter for the neural network, in this case we have to run a smaller filter on the latter half of our NN.
- Epoch is the number of times we are going to train our neural network. It will always tune in hyperparameter tuning, but 40 epochs would be a solid number.

As stated above we decided to tune these four specific hyperparameters: batch size, drop out rate, filter size and epoch. While smaller batch sizes generally give a better result, we did not want to suffer the consequence of a high computation cost. Therefore, we choose these three batch sizes to run with 32, 64 and 128. The dropout rate allows for better accuracy because it can help with overfitting. For this hyperparameter we chose: 0.1, 0.2, 0.25 and 0.5. Lastly, we decided it was best to leave the epoch size to 40. This is because any larger size would greatly increase the runtime and computation load.

In [19]:

```
clf_keras = KerasClassifier(build_fn = NN, verbose = 0)
```

```
/tmp/ipykernel_41859/2273775274.py:1: DeprecationWarning: KerasClassifier is depr
ecated, use Sci-Keras (https://github.com/adriangb/scikeras) instead. See http
s://www.adriangb.com/scikeras/stable/migration.html for help migrating.
  clf_keras = KerasClassifier(build_fn = NN, verbose = 0)
```

We will wrap our tensorflow function into a Scikit learn wrapper in order to do our cross validation and randomized search.

In [20]:

```
rand = RandomizedSearchCV(estimator=clf_keras, param_distributions=search_space,
```

This cross validation will be a randomized search CV, since it will take a lot of computation and time to perform a grid search on the every possible combination of hyperparameters that we have in our dictionary. Utilizing randomized search CV as the cross validation, will allow us to perform a search on all the hyperparameter combinations to find the best ones while still remaining efficient. With five cross validation folds and ten candidates, we will end with fifty runs for our cross validation.

In [21]:

```
rand.fit(X_train,y_train)
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
[CV 1/5] END activation=relu, batch_size=50, drop=0.5, epochs=40, loss=categorica
l_crossentropy, neurons=32, optimizer=Adam;, score=0.741 total time=  23.6s
[CV 2/5] END activation=relu, batch_size=50, drop=0.5, epochs=40, loss=categorica
l_crossentropy, neurons=32, optimizer=Adam;, score=0.753 total time=  22.6s
[CV 3/5] END activation=relu, batch_size=50, drop=0.5, epochs=40, loss=categorica
l_crossentropy, neurons=32, optimizer=Adam;, score=0.744 total time=  23.0s
[CV 4/5] END activation=relu, batch_size=50, drop=0.5, epochs=40, loss=categorica
l_crossentropy, neurons=32, optimizer=Adam;, score=0.741 total time=  22.9s
[CV 5/5] END activation=relu, batch_size=50, drop=0.5, epochs=40, loss=categorica
l_crossentropy, neurons=32, optimizer=Adam;, score=0.747 total time=  22.0s
[CV 1/5] END activation=sigmoid, batch_size=50, drop=0.2, epochs=40, loss=squared
```

```
_hinge, neurons=128, optimizer=Nadam;, score=0.741 total time=  35.2s
[CV 2/5] END activation=sigmoid, batch_size=50, drop=0.2, epochs=40, loss=squared
_hinge, neurons=128, optimizer=Nadam;, score=0.753 total time=  34.8s
[CV 3/5] END activation=sigmoid, batch_size=50, drop=0.2, epochs=40, loss=squared
_hinge, neurons=128, optimizer=Nadam;, score=0.744 total time=  34.4s
[CV 4/5] END activation=sigmoid, batch_size=50, drop=0.2, epochs=40, loss=squared
_hinge, neurons=128, optimizer=Nadam;, score=0.741 total time=  34.6s
[CV 5/5] END activation=sigmoid, batch_size=50, drop=0.2, epochs=40, loss=squared
_hinge, neurons=128, optimizer=Nadam;, score=0.747 total time=  34.6s
[CV 1/5] END activation=tanh, batch_size=50, drop=0.25, epochs=40, loss=categoric
al_crossentropy, neurons=32, optimizer=RMSprop;, score=0.741 total time=  22.4s
[CV 2/5] END activation=tanh, batch_size=50, drop=0.25, epochs=40, loss=categoric
al_crossentropy, neurons=32, optimizer=RMSprop;, score=0.753 total time=  17.4s
[CV 3/5] END activation=tanh, batch_size=50, drop=0.25, epochs=40, loss=categoric
al_crossentropy, neurons=32, optimizer=RMSprop;, score=0.744 total time=  16.0s
[CV 4/5] END activation=tanh, batch_size=50, drop=0.25, epochs=40, loss=categoric
al_crossentropy, neurons=32, optimizer=RMSprop;, score=0.741 total time=  21.5s
[CV 5/5] END activation=tanh, batch_size=50, drop=0.25, epochs=40, loss=categoric
al_crossentropy, neurons=32, optimizer=RMSprop;, score=0.747 total time=  19.0s
[CV 1/5] END activation=sigmoid, batch_size=50, drop=0.25, epochs=40, loss=poisso
n, neurons=32, optimizer=Adagrad;, score=0.259 total time=  18.5s
[CV 2/5] END activation=sigmoid, batch_size=50, drop=0.25, epochs=40, loss=poisso
n, neurons=32, optimizer=Adagrad;, score=0.247 total time=  16.4s
[CV 3/5] END activation=sigmoid, batch_size=50, drop=0.25, epochs=40, loss=poisso
n, neurons=32, optimizer=Adagrad;, score=0.255 total time=  19.0s
[CV 4/5] END activation=sigmoid, batch_size=50, drop=0.25, epochs=40, loss=poisso
n, neurons=32, optimizer=Adagrad;, score=0.259 total time=  18.3s
[CV 5/5] END activation=sigmoid, batch_size=50, drop=0.25, epochs=40, loss=poisso
n, neurons=32, optimizer=Adagrad;, score=0.253 total time=  17.4s
[CV 1/5] END activation=tanh, batch_size=100, drop=0.25, epochs=40, loss=kl_diver
gence, neurons=128, optimizer=RMSprop;, score=0.259 total time=  18.9s
[CV 2/5] END activation=tanh, batch_size=100, drop=0.25, epochs=40, loss=kl_diver
gence, neurons=128, optimizer=RMSprop;, score=0.247 total time=  18.4s
[CV 3/5] END activation=tanh, batch_size=100, drop=0.25, epochs=40, loss=kl_diver
gence, neurons=128, optimizer=RMSprop;, score=0.255 total time=  18.9s
[CV 4/5] END activation=tanh, batch_size=100, drop=0.25, epochs=40, loss=kl_diver
gence, neurons=128, optimizer=RMSprop;, score=0.259 total time=  19.0s
[CV 5/5] END activation=tanh, batch_size=100, drop=0.25, epochs=40, loss=kl_diver
gence, neurons=128, optimizer=RMSprop;, score=0.253 total time=  18.9s
[CV 1/5] END activation=sigmoid, batch_size=200, drop=0.2, epochs=40, loss=kl_div
ergence, neurons=64, optimizer=Adam;, score=0.259 total time=  10.5s
[CV 2/5] END activation=sigmoid, batch_size=200, drop=0.2, epochs=40, loss=kl_div
ergence, neurons=64, optimizer=Adam;, score=0.247 total time=  10.6s
[CV 3/5] END activation=sigmoid, batch_size=200, drop=0.2, epochs=40, loss=kl_div
ergence, neurons=64, optimizer=Adam;, score=0.255 total time=  10.6s
[CV 4/5] END activation=sigmoid, batch_size=200, drop=0.2, epochs=40, loss=kl_div
ergence, neurons=64, optimizer=Adam;, score=0.259 total time=  10.7s
[CV 5/5] END activation=sigmoid, batch_size=200, drop=0.2, epochs=40, loss=kl_div
ergence, neurons=64, optimizer=Adam;, score=0.253 total time=  10.6s
[CV 1/5] END activation=relu, batch_size=100, drop=0.2, epochs=40, loss=poisson,
neurons=64, optimizer=Adamax;, score=0.259 total time=  13.6s
[CV 2/5] END activation=relu, batch_size=100, drop=0.2, epochs=40, loss=poisson,
neurons=64, optimizer=Adamax;, score=0.247 total time=  13.2s
[CV 3/5] END activation=relu, batch_size=100, drop=0.2, epochs=40, loss=poisson,
neurons=64, optimizer=Adamax;, score=0.744 total time=  13.8s
[CV 4/5] END activation=relu, batch_size=100, drop=0.2, epochs=40, loss=poisson,
neurons=64, optimizer=Adamax;, score=0.741 total time=  13.5s
[CV 5/5] END activation=relu, batch_size=100, drop=0.2, epochs=40, loss=poisson,
neurons=64, optimizer=Adamax;, score=0.253 total time=  13.1s
[CV 1/5] END activation=tanh, batch_size=50, drop=0.25, epochs=40, loss=kl_diverg
```

```
ence, neurons=64, optimizer=Adadelta;, score=0.259 total time=  24.2s
[CV 2/5] END activation=tanh, batch_size=50, drop=0.25, epochs=40, loss=kl_diverg
ence, neurons=64, optimizer=Adadelta;, score=0.247 total time=  19.4s
[CV 3/5] END activation=tanh, batch_size=50, drop=0.25, epochs=40, loss=kl_diverg
ence, neurons=64, optimizer=Adadelta;, score=0.255 total time=  21.8s
[CV 4/5] END activation=tanh, batch_size=50, drop=0.25, epochs=40, loss=kl_diverg
ence, neurons=64, optimizer=Adadelta;, score=0.259 total time=  21.2s
[CV 5/5] END activation=tanh, batch_size=50, drop=0.25, epochs=40, loss=kl_diverg
ence, neurons=64, optimizer=Adadelta;, score=0.253 total time=  23.2s
[CV 1/5] END activation=tanh, batch_size=100, drop=0.5, epochs=40, loss=categoric
al_crossentropy, neurons=64, optimizer=Adamax;, score=0.741 total time=  13.6s
[CV 2/5] END activation=tanh, batch_size=100, drop=0.5, epochs=40, loss=categoric
al_crossentropy, neurons=64, optimizer=Adamax;, score=0.753 total time=  12.7s
[CV 3/5] END activation=tanh, batch_size=100, drop=0.5, epochs=40, loss=categoric
al_crossentropy, neurons=64, optimizer=Adamax;, score=0.744 total time=  13.1s
[CV 4/5] END activation=tanh, batch_size=100, drop=0.5, epochs=40, loss=categoric
al_crossentropy, neurons=64, optimizer=Adamax;, score=0.741 total time=  13.0s
[CV 5/5] END activation=tanh, batch_size=100, drop=0.5, epochs=40, loss=categoric
al_crossentropy, neurons=64, optimizer=Adamax;, score=0.747 total time=  13.7s
[CV 1/5] END activation=tanh, batch_size=200, drop=0.1, epochs=40, loss=poisson,
neurons=32, optimizer=Adadelta;, score=0.259 total time=   6.6s
[CV 2/5] END activation=tanh, batch_size=200, drop=0.1, epochs=40, loss=poisson,
neurons=32, optimizer=Adadelta;, score=0.247 total time=   6.2s
[CV 3/5] END activation=tanh, batch_size=200, drop=0.1, epochs=40, loss=poisson,
neurons=32, optimizer=Adadelta;, score=0.255 total time=   6.9s
[CV 4/5] END activation=tanh, batch_size=200, drop=0.1, epochs=40, loss=poisson,
neurons=32, optimizer=Adadelta;, score=0.259 total time=   6.5s
[CV 5/5] END activation=tanh, batch_size=200, drop=0.1, epochs=40, loss=poisson,
neurons=32, optimizer=Adadelta;, score=0.253 total time=   6.5s
```

Out[21]:

```
RandomizedSearchCV(cv=5,
                   estimator=<keras.wrappers.scikit_learn.KerasClassifier object
at 0x7f4e34580610>,
                   param_distributions={'activation': ['relu', 'tanh',
                                                       'sigmoid', 'linear'],
                                        'batch_size': [50, 100, 200],
                                        'drop': [0.1, 0.2, 0.25, 0.5],
                                        'epochs': [40],
                                        'loss': ['categorical_crossentropy',
                                                 'poisson', 'kl_divergence',
                                                 'squared_hinge'],
                                        'neurons': [32, 64, 128],
                                        'optimizer': ['SGD', 'RMSprop',
                                                      'Adagrad', 'Adadelta',
                                                      'Adam', 'Adamax',
                                                      'Nadam']},
                   verbose=3)
```

After tuning the fitting of our randomized search, we came up with the best score below.

In [22]:

```
rand.best_score_
```

Out[22]:

```
0.7453999996185303
```

These are the parameters that resulted in the best score:

In [23]:

```
params = rand.best_params_
params
```

Out[23]:
```
{'optimizer': 'Adam',
 'neurons': 32,
 'loss': 'categorical_crossentropy',
 'epochs': 40,
 'drop': 0.5,
 'batch_size': 50,
 'activation': 'relu'}
```

In [29]:
```
hyper = NN(neurons=32, drop= 0.5, activation='relu', optimizer='Adam', loss='cate
hyper_plot = hyper.fit(X_train, y_train, batch_size= 50, epochs= 40, validation_d

[test_loss, test_acc] = hyper.evaluate(X_test, y_test)
print("Evaluation result on Test Data : Loss = {}, accuracy = {}".format(test_los
```

```
Epoch 1/40
400/400 [==============================] - 1s 3ms/step - loss: 15735300096.0000 -
accuracy: 0.8802 - val_loss: 39592742912.0000 - val_accuracy: 0.9170
Epoch 2/40
400/400 [==============================] - 1s 2ms/step - loss: 96205340672.0000 -
accuracy: 0.8601 - val_loss: 111738462208.0000 - val_accuracy: 0.9170
Epoch 3/40
400/400 [==============================] - 1s 3ms/step - loss: 250909458432.0000
- accuracy: 0.8124 - val_loss: 178581749760.0000 - val_accuracy: 0.9170
Epoch 4/40
400/400 [==============================] - 1s 2ms/step - loss: 525025705984.0000
- accuracy: 0.7572 - val_loss: 185003507712.0000 - val_accuracy: 0.9170
Epoch 5/40
400/400 [==============================] - 1s 2ms/step - loss: 1023428984832.0000
- accuracy: 0.6779 - val_loss: 196580605952.0000 - val_accuracy: 0.9170
Epoch 6/40
400/400 [==============================] - 1s 2ms/step - loss: 1764850532352.0000
- accuracy: 0.6406 - val_loss: 163681370112.0000 - val_accuracy: 0.9170
Epoch 7/40
400/400 [==============================] - 1s 2ms/step - loss: 2715518500864.0000
- accuracy: 0.5880 - val_loss: 108186075136.0000 - val_accuracy: 0.9170
Epoch 8/40
400/400 [==============================] - 1s 2ms/step - loss: 3911046660096.0000
- accuracy: 0.5418 - val_loss: 66726080512.0000 - val_accuracy: 0.9170
Epoch 9/40
400/400 [==============================] - 1s 2ms/step - loss: 5032604860416.0000
- accuracy: 0.5052 - val_loss: 914012504064.0000 - val_accuracy: 0.0830
Epoch 10/40
400/400 [==============================] - 1s 2ms/step - loss: 6945942011904.0000
- accuracy: 0.4554 - val_loss: 1668484694016.0000 - val_accuracy: 0.0830
Epoch 11/40
400/400 [==============================] - 1s 2ms/step - loss: 8394176987136.0000
- accuracy: 0.4380 - val_loss: 3432445640704.0000 - val_accuracy: 0.0830
Epoch 12/40
400/400 [==============================] - 1s 2ms/step - loss: 10474702766080.000
0 - accuracy: 0.4092 - val_loss: 6480138338304.0000 - val_accuracy: 0.0830
Epoch 13/40
400/400 [==============================] - 1s 2ms/step - loss: 14016737968128.000
0 - accuracy: 0.3776 - val_loss: 9491389087744.0000 - val_accuracy: 0.0830
Epoch 14/40
400/400 [==============================] - 1s 2ms/step - loss: 16842905092096.000
0 - accuracy: 0.3444 - val_loss: 11745008549888.0000 - val_accuracy: 0.0830
Epoch 15/40
400/400 [==============================] - 1s 2ms/step - loss: 19394082111488.000
0 - accuracy: 0.3451 - val_loss: 15939719397376.0000 - val_accuracy: 0.0830
Epoch 16/40
```

```
400/400 [==============================] - 1s 2ms/step - loss: 23386426179584.000
0 - accuracy: 0.3221 - val_loss: 20530394562560.0000 - val_accuracy: 0.0830
Epoch 17/40
400/400 [==============================] - 1s 2ms/step - loss: 27870189584384.000
0 - accuracy: 0.2946 - val_loss: 24110501462016.0000 - val_accuracy: 0.0830
Epoch 18/40
400/400 [==============================] - 1s 2ms/step - loss: 32211220824064.000
0 - accuracy: 0.2778 - val_loss: 27881631645696.0000 - val_accuracy: 0.0830
Epoch 19/40
400/400 [==============================] - 1s 2ms/step - loss: 36148059570176.000
0 - accuracy: 0.2600 - val_loss: 32047796060160.0000 - val_accuracy: 0.0830
Epoch 20/40
400/400 [==============================] - 1s 2ms/step - loss: 42987157454848.000
0 - accuracy: 0.2361 - val_loss: 36730203799552.0000 - val_accuracy: 0.0830
Epoch 21/40
400/400 [==============================] - 1s 2ms/step - loss: 47986348392448.000
0 - accuracy: 0.2304 - val_loss: 42174255202304.0000 - val_accuracy: 0.0830
Epoch 22/40
400/400 [==============================] - 1s 2ms/step - loss: 54186859298816.000
0 - accuracy: 0.2144 - val_loss: 50222243774464.0000 - val_accuracy: 0.0830
Epoch 23/40
400/400 [==============================] - 1s 2ms/step - loss: 62446681194496.000
0 - accuracy: 0.2120 - val_loss: 59800004067328.0000 - val_accuracy: 0.0830
Epoch 24/40
400/400 [==============================] - 1s 2ms/step - loss: 71441840078848.000
0 - accuracy: 0.1978 - val_loss: 67673111134208.0000 - val_accuracy: 0.0830
Epoch 25/40
400/400 [==============================] - 1s 2ms/step - loss: 82009758105600.000
0 - accuracy: 0.1967 - val_loss: 81084997959680.0000 - val_accuracy: 0.0830
Epoch 26/40
400/400 [==============================] - 1s 2ms/step - loss: 94413497827328.000
0 - accuracy: 0.1808 - val_loss: 93322819403776.0000 - val_accuracy: 0.0830
Epoch 27/40
400/400 [==============================] - 1s 2ms/step - loss: 107220637319168.00
00 - accuracy: 0.1732 - val_loss: 105991379091456.0000 - val_accuracy: 0.0830
Epoch 28/40
400/400 [==============================] - 1s 2ms/step - loss: 123049957392384.00
00 - accuracy: 0.1652 - val_loss: 126742933012480.0000 - val_accuracy: 0.0830
Epoch 29/40
400/400 [==============================] - 1s 2ms/step - loss: 140142903820288.00
00 - accuracy: 0.1537 - val_loss: 141803898535936.0000 - val_accuracy: 0.0830
Epoch 30/40
400/400 [==============================] - 1s 2ms/step - loss: 157767939653632.00
00 - accuracy: 0.1408 - val_loss: 157948412166144.0000 - val_accuracy: 0.0830
Epoch 31/40
400/400 [==============================] - 1s 2ms/step - loss: 179400767176704.00
00 - accuracy: 0.1368 - val_loss: 181812626194432.0000 - val_accuracy: 0.0830
Epoch 32/40
400/400 [==============================] - 1s 2ms/step - loss: 203780930928640.00
00 - accuracy: 0.1289 - val_loss: 198738823872512.0000 - val_accuracy: 0.0830
Epoch 33/40
400/400 [==============================] - 1s 2ms/step - loss: 229554408914944.00
00 - accuracy: 0.1221 - val_loss: 221941176729600.0000 - val_accuracy: 0.0830
Epoch 34/40
400/400 [==============================] - 1s 2ms/step - loss: 248806750814208.00
00 - accuracy: 0.1163 - val_loss: 246812711583744.0000 - val_accuracy: 0.0830
Epoch 35/40
400/400 [==============================] - 1s 3ms/step - loss: 280908292685824.00
00 - accuracy: 0.1124 - val_loss: 273016978145280.0000 - val_accuracy: 0.0830
Epoch 36/40
```

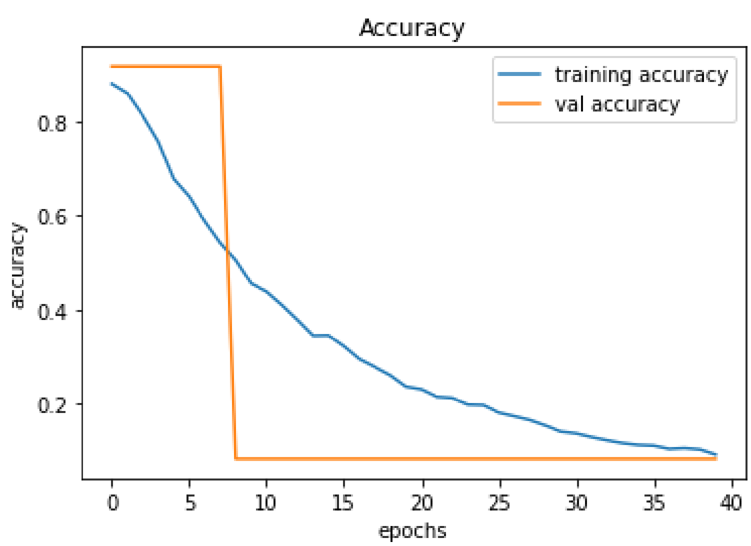
```
400/400 [==============================] - 1s 2ms/step - loss: 305380731125760.00
00 - accuracy: 0.1110 - val_loss: 304967978057728.0000 - val_accuracy: 0.0830
Epoch 37/40
400/400 [==============================] - 1s 2ms/step - loss: 348521463021568.00
00 - accuracy: 0.1040 - val_loss: 334336494665728.0000 - val_accuracy: 0.0830
Epoch 38/40
400/400 [==============================] - 1s 2ms/step - loss: 376112098050048.00
00 - accuracy: 0.1057 - val_loss: 374958328905728.0000 - val_accuracy: 0.0830
Epoch 39/40
400/400 [==============================] - 1s 2ms/step - loss: 413446369705984.00
00 - accuracy: 0.1028 - val_loss: 421071681486848.0000 - val_accuracy: 0.0830
Epoch 40/40
400/400 [==============================] - 1s 2ms/step - loss: 464967589429248.00
00 - accuracy: 0.0921 - val_loss: 459555192438784.0000 - val_accuracy: 0.0830
157/157 [==============================] - 0s 1ms/step - loss: 459554991112192.00
00 - accuracy: 0.0830
Evaluation result on Test Data : Loss = 459554991112192.0, accuracy = 0.082999996
84095383
```

In [30]:

```
plt.plot(hyper_plot.history['accuracy'], label='training accuracy')
plt.plot(hyper_plot.history['val_accuracy'], label='val accuracy')
plt.title('Accuracy')
plt.xlabel('epochs')
plt.ylabel('accuracy')
plt.legend()
```

Out[30]:

```
<matplotlib.legend.Legend at 0x7f4e047789a0>
```



The fine tuned parameters work not as good as the parameters we designed to input might be caused by our limitation of data or the processing power, since we used RandomizedSearchCV instead of GridSearchCV.

Model-2

SVR

In [31]:

```
from sklearn.metrics import mean_squared_error
import matplotlib.cm as cm
```

In [32]:
```
df = pd.read_csv('clean25kDataset.csv')
```

In [33]:
```
timestamp_list = []
for d, t in zip(df['Date'], df['Time']):
    timestamp = datetime.datetime.strptime(d+' '+t, '%Y-%m-%d %H:%M:%S')
    timestamp_list.append(time.mktime(timestamp.timetuple()))

timeStamp = pd.Series(timestamp_list)
df['Timestamp'] = timeStamp.values
clean_df = df.drop(['Date', 'Time'], axis=1)

clean_df = clean_df.rename(columns={'Lat': 'Latitude', 'Lon': 'Longitude', 'Mag':
clean_df
```

Out[33]:

	Latitude	Longitude	Depth	Magnitude	Region	Timestamp
0	60.5758	-147.5620	15.1	2.6	57 km SW of Tatitlek, Alaska	1.670283e+09
1	37.3565	-121.7167	8.2	1.5	10km E of Alum Rock, CA	1.670283e+09
2	60.1315	-153.1349	125.6	1.9	66 km E of Port Alsworth, Alaska	1.670282e+09
3	37.3247	-121.6887	6.9	3.7	13km ESE of Alum Rock, CA	1.670282e+09
4	39.4327	-92.2425	4.7	2.5	5 km SSW of Madison, Missouri	1.670282e+09
...	...	...	...	...	...	...
24995	58.2855	-154.9823	3.8	0.5	85 km NNW of Karluk, Alaska	1.664120e+09
24996	51.3816	142.7739	10.0	4.8	51 km NE of Mgachi, Russia	1.664120e+09
24997	27.7017	56.4543	10.0	4.9	59 km NNE of Bandar Abbas, Iran	1.664120e+09
24998	35.3747	-118.1223	4.5	1.1	30km NNW of California City, CA	1.664119e+09
24999	53.7207	-162.6143	25.6	1.9	136 km SSE of False Pass, Alaska	1.664119e+09

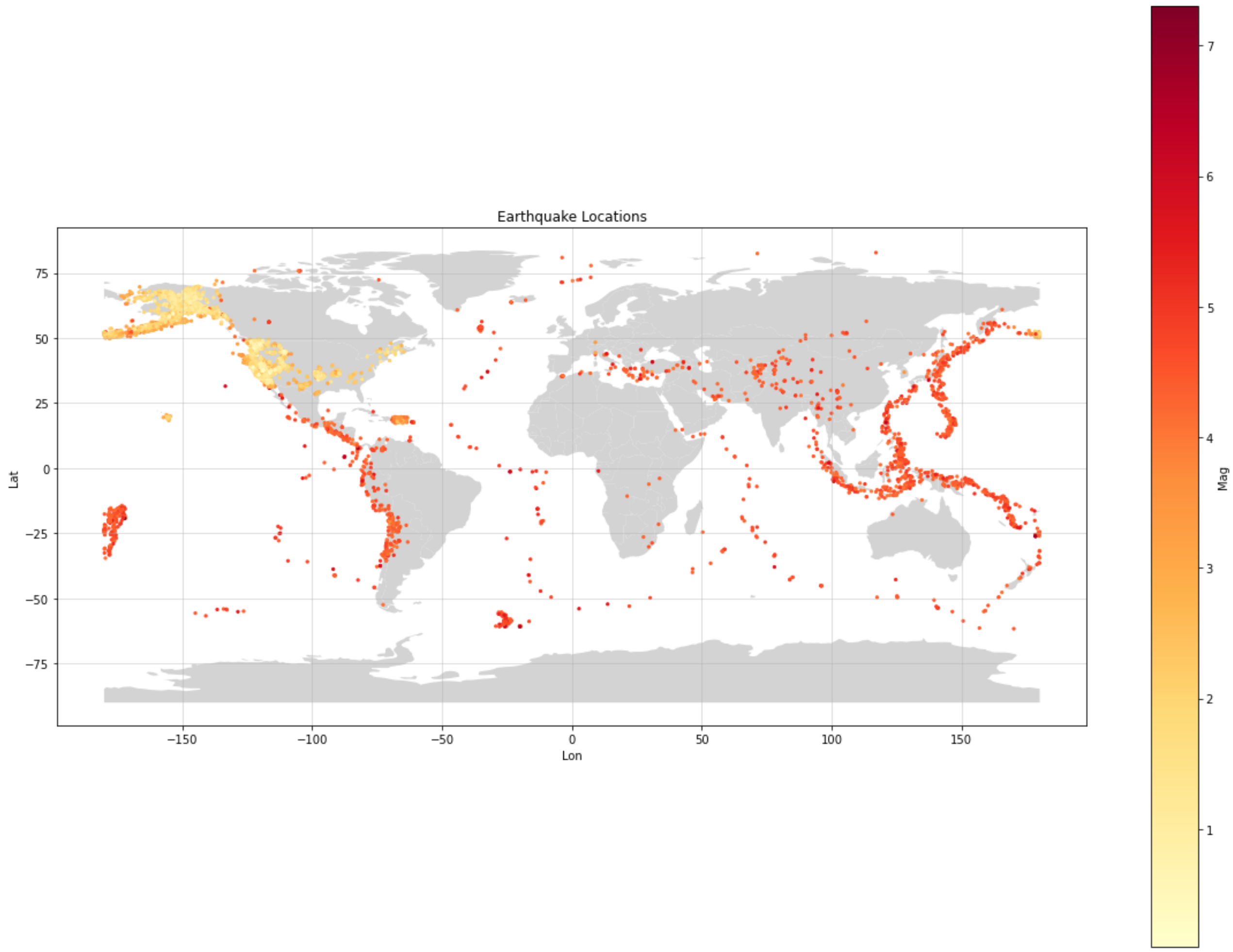
25000 rows × 6 columns

In [34]:
```
max(clean_df['Magnitude'])
```

Out[34]:
```
7.3
```

In [35]:
```
countries = gpd.read_file(gpd.datasets.get_path("naturalearth_lowres"))

# initialize an axis
fig, ax = plt.subplots(figsize=(20,15))

# plot map on axis
countries = gpd.read_file(gpd.datasets.get_path("naturalearth_lowres"))
countries.plot(color="lightgrey",ax=ax)

# plot points
df.plot(x="Lon", y="Lat", marker ='.', kind="scatter", c="Mag", colormap="YlOrRd"
        title="Earthquake Locations", ax=ax)

# add grid
```

```
ax.grid(visible=True, alpha=0.5)

plt.show()
```



In [36]:

```
X = clean_df[['Timestamp', 'Latitude', 'Longitude']]
y = clean_df[['Magnitude']]
X = X.to_numpy()
y = y.to_numpy().reshape(-1, 1)
```

In [37]:

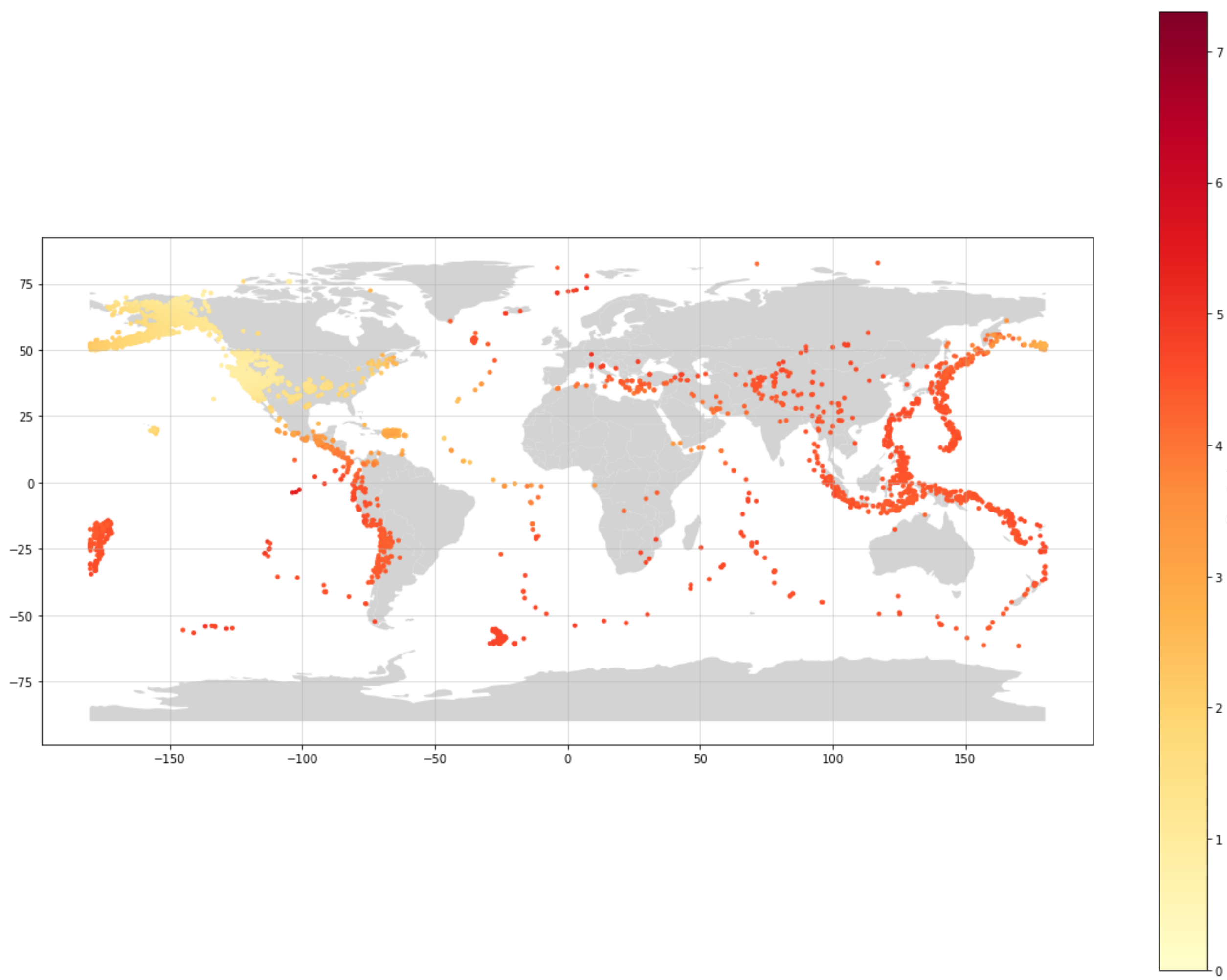
```
from sklearn.svm import SVR
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

sc_X = StandardScaler()
sc_y = StandardScaler()
X = sc_X.fit_transform(X)
y = sc_y.fit_transform(y)
```

In [38]:

```
svr = SVR().fit(X, y.ravel())
yfit = svr.predict(X)
```

In [39]:
```
X = sc_X.inverse_transform(X)
y = sc_y.inverse_transform(y)
yfit = sc_y.inverse_transform(yfit)
```

In [40]:
```
countries = gpd.read_file(gpd.datasets.get_path("naturalearth_lowres"))

# initialize an axis
fig, ax = plt.subplots(figsize=(20,15))

# plot map on axis
countries = gpd.read_file(gpd.datasets.get_path("naturalearth_lowres"))
countries.plot(color="lightgrey",ax=ax)

# plot points
plt.scatter(X[:,2], X[:,1], marker ='.', c=yfit)
plt.set_cmap("YlOrRd")
plt.colorbar(label="Magnitude", orientation="vertical")
plt.clim(0,7.3)

# add grid
ax.grid(visible=True, alpha=0.5)

plt.show()
```

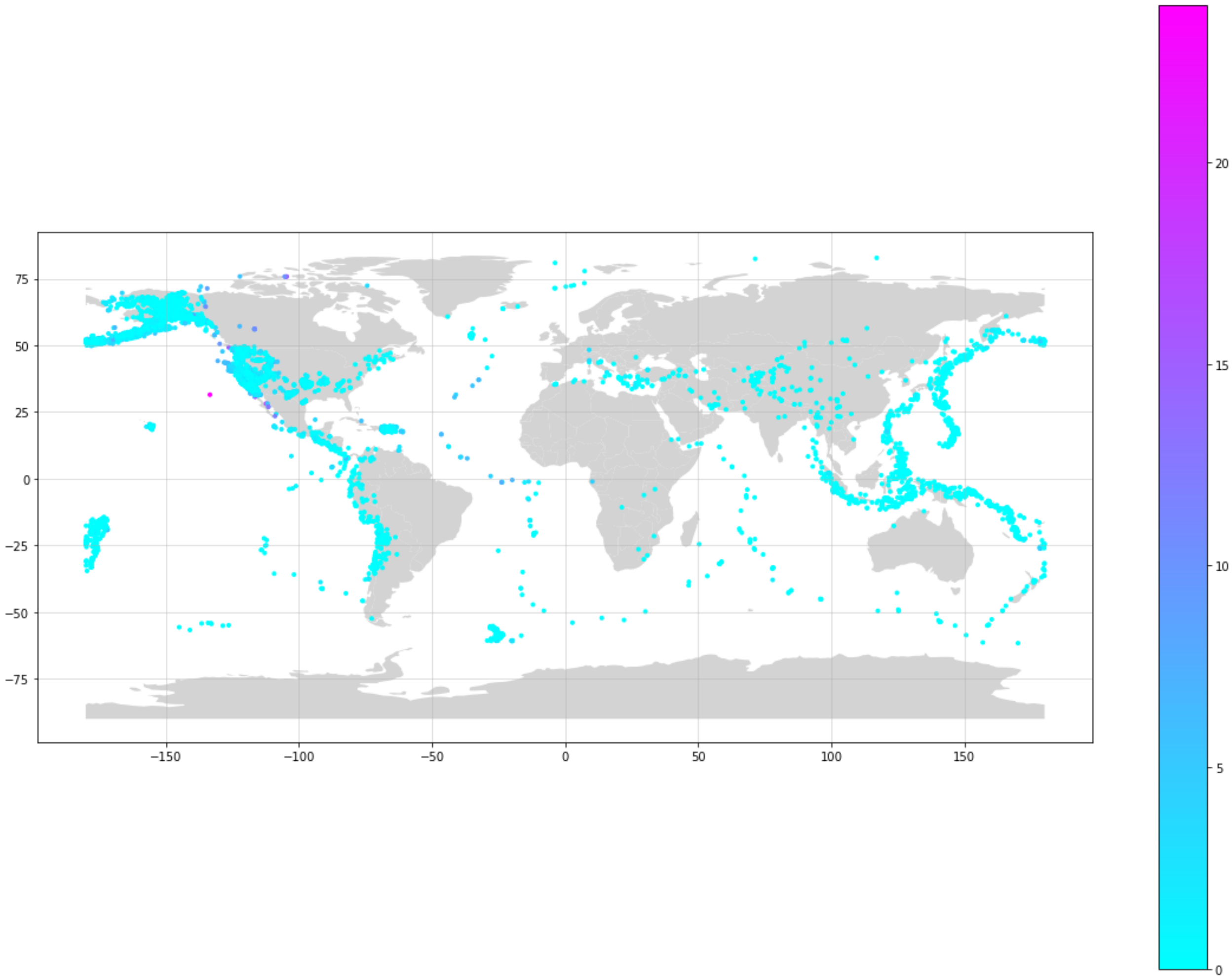


In [41]:
```
score = svr.score(X,y[:,0])
print("R-squared:", score)
print("MSE:", mean_squared_error(y[:,0], yfit))
```

```
R-squared: -0.10852553290586431
MSE: 0.37817285756014696
```

In [42]:
```
yfit = yfit.reshape(-1,1)
newy = y-yfit
newy = np.power(newy,2)
newy = newy.reshape(-1,1)
```

In [43]:
```
countries = gpd.read_file(gpd.datasets.get_path("naturalearth_lowres"))

# initialize an axis
fig, ax = plt.subplots(figsize=(20,15))

# plot map on axis
countries = gpd.read_file(gpd.datasets.get_path("naturalearth_lowres"))
countries.plot(color="lightgrey",ax=ax)

# plot points
plt.scatter(X[:,2], X[:,1], marker ='.', c=newy)
plt.set_cmap("cool")
plt.colorbar(label="Error", orientation="vertical")

# add grid
ax.grid(visible=True, alpha=0.5)

plt.show()
```



Results

From the graph and the verbose of fitting the data into our model, it looks like our hyperparameter tuned model is OVERFITTING/UNDERFITTING to the training data. We can see the intersection of the two accuracy values within our graph. Our training accuracy is higher than the validation accuracy. It is a semi accurate model, with a INSERT PERCENT validation accuracy. We can try implementing our model with fewer epochs in order to combat the OVERFITTING of training data.

Compared to the base model of our hyperparameter tuned model, the hyper-tuned model overfitted with the training data set. The validation accuracy of the base model is INSERT PERCENT with a training accuracy of INSERT PERCENT, while the hyper tuned model has a validation accuracy of INSERT PERCENT with a INSERT PERCENT training accuracy. Overall the base NN model would be the better performer out of the two models. We could have done a grid search, to find the best true hyperparameters, but that would be computationally expensive.

One of the reasons, our baseline model did better was due to the custom parameters we choose were not optimal for the dataset to begin with.

Discussion

What did we learn?

- Data Sets
    - Machine learning prediction models have the better accuracy when working with very large datasets. The first two datasets we used were quite small and resulted in poor outcomes or did not go work with the model at all.
- Models
    - Initially we attempted to use LeNetas our model and we got it to work but the results were poor. This was most likely due to the fact that LeNet are mainly used as a models for image prediction and classification, which is not what we were doing. #### How can we improve?
- Data
    - More data. Our dataset collects the earthquakes from September 25th, 2022 to December 5th, 2022 (the moment we collected the data). So the dataset was small and might because of seasonal effects on tectonic features, the data isn’t representative enough.
    - With more processing power we could use larger datasets for a more generalized picture
- Time
    - One major limitation was time and processing power
    - Because of the processing power and the numbers of the hyper-parameters we wanted to test out, it takes too long for us to utilize the GridSearchCV in order to find better hyper-parameters than purely using RandomizedSearchCV.

Team Contributions

- Andrina / Xiaoxuan Zhang: Finding data, Model implementation, Testing & tuning the algorithm, Slides & Video presentation, Video Editing
- Alexander Huynh: Researching models, Creating model, Tuning the algorithm, Slides & Video presentation
- Victorionna Tran: Data cleaning & wrangling & visualization, Slides & Video presentation
- Nicolas Schaefer: Data collecting & cleaning, Slides & Video presentation