ARTIFICIAL INTELLIGENCE- GROUP 5

PROJECT: EARTHQUAKE PREDICTION MODEL USING PYTHON

PHASE 3: DEVELOPMENT PART 1

MOHAMED SALI HESHAM:963321104035

Building an earthquake prediction model involves several steps, including data loading, preprocessing, feature engineering, model selection, training, and evaluation. Below, I'll outline the steps for building a basic earthquake prediction model:

Data Loading:

Find a reliable dataset containing earthquake-related information. You can search for earthquake datasets on websites like the USGS Earthquake Hazards Program or Kaggle.

Data Preprocessing:

Load the dataset into your chosen programming environment (Python is commonly used for this purpose).

Check for missing values and decide how to handle them (e.g., imputation or removal).

Explore the dataset to understand its structure and the types of information available.

Feature Engineering:

Identify relevant features for predicting earthquakes. These might include geological data (e.g., fault lines, tectonic plate boundaries), historical earthquake data, geographical data, and more.

Extract or create relevant features from the raw data. For example, you might calculate distances to fault lines or identify regions with high seismic activity.

Data Splitting:

Divide the dataset into training, validation, and test sets. The training set is used to train the model, the validation set helps in tuning hyperparameters, and the test set is used for the final evaluation.

Model Selection:

Choose a suitable machine learning algorithm for earthquake prediction. Common choices include Random Forest, Support Vector Machines, Neural Networks, etc. Additionally, consider using time series models if your data includes temporal patterns.

Model Training:

Train your selected model using the training data. Ensure you properly handle the features and target variable.

Model Evaluation:

Evaluate the model using the validation set. Common evaluation metrics for regression tasks (predicting continuous values) include Mean Absolute Error (MAE), Mean Squared Error (MSE), and R-squared.

Hyperparameter Tuning:

If applicable, perform hyperparameter tuning using techniques like grid search or random search to find the best set of hyperparameters for your model.

Final Model Testing:

Once you're satisfied with the model's performance on the validation set, evaluate it on the test set to get a final assessment of its predictive capabilities.

Deployment:

If you plan to deploy the model in a real-world setting, you'll need to set up a deployment environment (e.g., a web application or an API) and ensure it can handle predictions in real-time.

Model Building:

Data Set: https://www.kaggle.com/datasets/usgs/earthquake-database

In this section the following predictive models will be built:

1. KNN
2. Random Forest
3. Gradient Boosted Machines

The aim is to identify the model which best predicts the damage grade on new data given the input features. Within this section, a pre-processing pipeline will be set up to prepare the data for training. A test dataset will be extracted from the main data to give us the opportunity to assess how the models perform on completely new data. The models will be evaluated on the test data using the evaulation metric F1.

KNN:

Using a K-Nearest Neighbors (KNN) model for earthquake prediction involves a different approach compared to traditional machine learning algorithms. KNN is a simple but effective method for classification and regression tasks. In the context of earthquake prediction, you might use KNN for spatial prediction, where you predict seismic activity in a region based on the seismic activity in its neighboring regions.

First load in the data and required packages

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from tqdm import tqdm
from sklearn.model_selection import train_test_split
from sklearn import neighbors
from sklearn import preprocessing
from sklearn.model_selection import cross_val_score
from sklearn.pipeline import make_pipeline
from sklearn.metrics import mean_absolute_error
from sklearn.model_selection import TimeSeriesSplit

# Input data files are available in the "../input/" directory.
# For example, running this (by clicking run or pressing Shift+Enter) will list th
e files in the input directory

import os
print(os.listdir("../input"))

# Any results you write to the current directory are saved as output.
['test', 'sample_submission.csv', 'train.csv']
```

```
## Read in the data
eq = pd.read_csv("../input/train.csv", dtype={'acoustic_data': np.int16,
                          'time_to_failure': np.float64})
## Print the first five lines
print(eq.head(5))
```

```
   acoustic_data  time_to_failure
0             12           1.4691
1              6           1.4691
2              8           1.4691
3              5           1.4691
4              8           1.4691
```

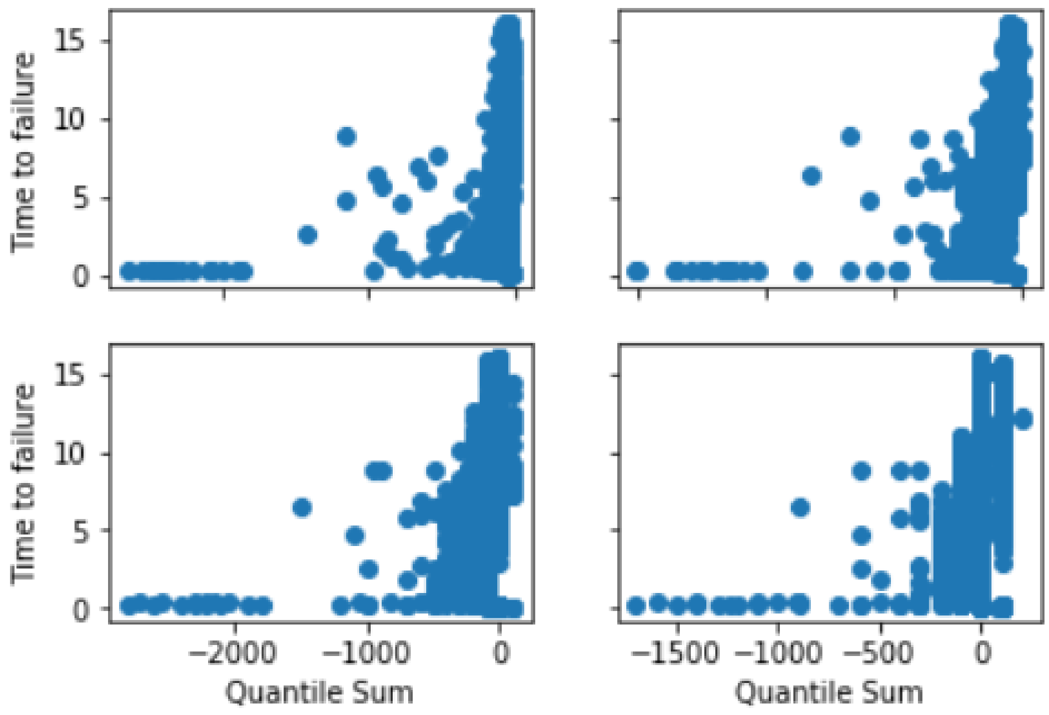
Here we take the raw values of the acoustic data, sort them, and average over ev
ery 100 values.

```
## Number of rows for each segment
rows = 150000
## Get the number of segments
segments = int(np.floor(eq.shape[0] / rows))
## Initialize X values and y values
X_train = pd.DataFrame(index=range(segments), dtype=np.float64, columns=ra
nge(0,1500))
y_train = pd.DataFrame(index=range(segments), dtype=np.float64, columns=['ti
me_to_failure'])
## Create features for each segment
for seg_id in tqdm(range(segments)):
    ## Get segment values
    seg = eq.iloc[range((seg_id*rows),((seg_id+1)*rows))]
    ## Sort the values
    x = np.sort(seg['acoustic_data'].values)
    ## Sum values of the quantile and put in X matrix
    X_train.loc[seg_id,:] = np.reshape(x, (-1, 100)).sum(axis=-1)
    ## Get the time to failure of final observation
    y_train.loc[seg_id, 'time_to_failure'] = seg['time_to_failure'].values[-1];

y_train = np.ravel(y_train);
```

```
100%|██████████| 4194/4194 [02:26<00:00, 28.62it/s]
```

Let's plot some of the quantiles against the time to failure. Each of the quantiles appears to contain some sort of information

```
f, axarr = plt.subplots(2, 2)
axarr[0, 0].scatter(X_train.iloc[:,0], y_train)
axarr[0, 1].scatter(X_train.iloc[:,50], y_train)
axarr[1, 0].scatter(X_train.iloc[:,100], y_train)
axarr[1, 1].scatter(X_train.iloc[:,149], y_train)
for ax in axarr.flat:
    ax.set(xlabel='Quantile Sum', ylabel='Time to failure')
for ax in axarr.flat:
    ax.label_outer()
```



Here we perform 5-fold cross validation of a KNN model after using a standard scaler

```
## Create pipline for cross validation using the standard scaler
knn_pl = make_pipeline(preprocessing.StandardScaler(),
                    neighbors.KNeighborsRegressor(500, weights='uniform', metric='
manhattan'))
## Perform 5 fold cross validation
scores = cross_val_score(knn_pl, X_train, y_train, cv=5, scoring='neg_mean_ab
solute_error')
np.mean(scores)
```

OUTPUT:

```
-2.160673089320833
```

We get a cross validation score of 2.16. Not bad for such a simple model!

```
knn_final = neighbors.KNeighborsRegressor(500, weights='uniform', metric='m
anhattan')
scaler = preprocessing.StandardScaler()
scaler.fit(X_train)
knn_final.fit(scaler.transform(X_train), y_train)
```

```
submission = pd.read_csv('../input/sample_submission.csv', index_col='seg_id')
X_test = pd.DataFrame(dtype = np.float64, index = submission.index,columns=
range(0,1500))
for seg_id in tqdm(X_test.index):
    seg = pd.read_csv('../input/test/' + seg_id + '.csv')
    xc = np.sort(seg['acoustic_data'].values)
    X_test.loc[seg_id,:] = np.reshape(xc, (-1, 100)).sum(axis=-1)

submission['time_to_failure'] = knn_final.predict(scaler.transform(X_test))
print(submission.head(5))

submission.to_csv('knn_std.csv')
```

```
100%|██████████| 2624/2624 [00:51<00:00, 51.40it/s]
```

```
           time_to_failure
seg_id
seg_00030f        4.554798
seg_0012b5        5.317521
seg_00184e        5.456752
seg_003339        8.516055
seg_0042cc        6.379061
```

Random Forest:

Using a Random Forest model for earthquake prediction can be a powerful approach, especially if you have a dataset with diverse features and complex relationships. Random Forest is an ensemble learning method that combines the predictions of multiple decision trees to improve accuracy and reduce overfitting.

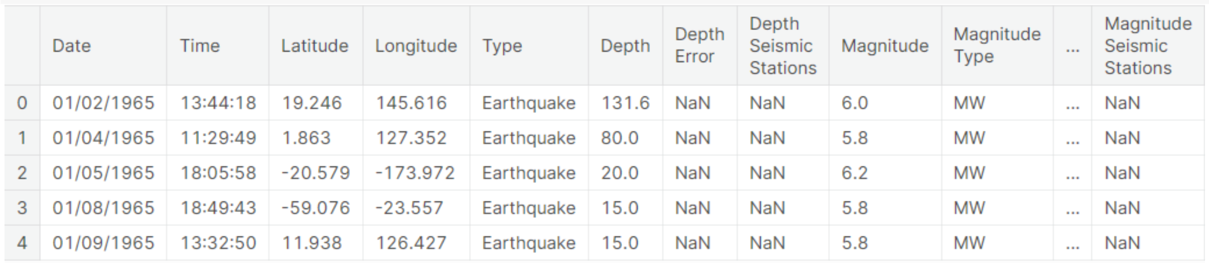
Importing the required packages

```
import numpy as np
import pandas as pd
import datetime
import time
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
import matplotlib.pyplot as plt
import seaborn as sns
```

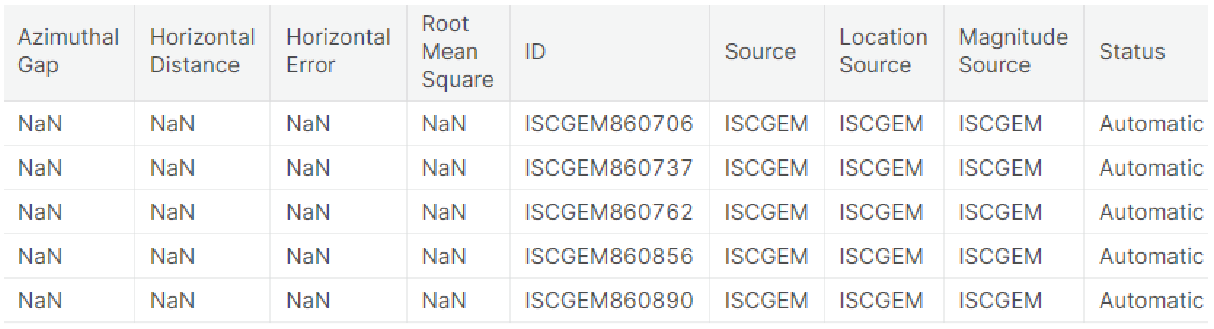
Importing the database

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

OUTPUT:



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



Azimuthal Gap	Horizontal Distance	Horizontal Error	Root Mean Square	ID	Source	Location Source	Magnitude Source	Status
NaN	NaN	NaN	NaN	ISCGEM860706	ISCGEM	ISCGEM	ISCGEM	Automatic
NaN	NaN	NaN	NaN	ISCGEM860737	ISCGEM	ISCGEM	ISCGEM	Automatic
NaN	NaN	NaN	NaN	ISCGEM860762	ISCGEM	ISCGEM	ISCGEM	Automatic
NaN	NaN	NaN	NaN	ISCGEM860856	ISCGEM	ISCGEM	ISCGEM	Automatic
NaN	NaN	NaN	NaN	ISCGEM860890	ISCGEM	ISCGEM	ISCGEM	Automatic

Pre-processing the database:

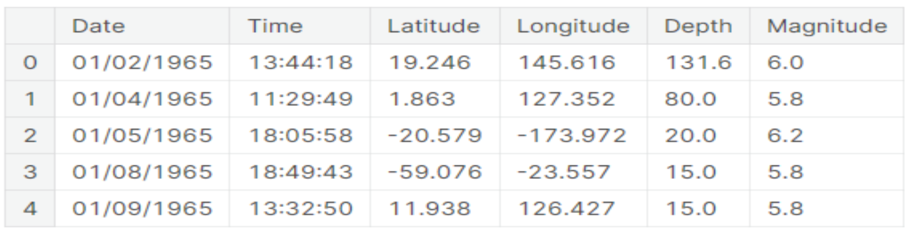
```
data.columns
```

OUTPUT:

```
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')
```

```
data = data[['Date', 'Time', 'Latitude', 'Longitude', 'Depth',
'Magnitude']]data.head()
```

OUTPUT:



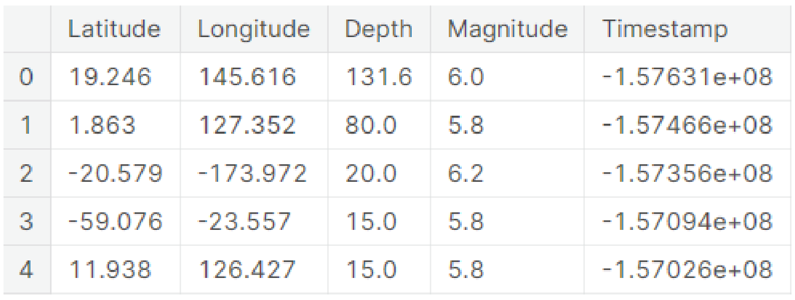
	Date	Time	Latitude	Longitude	Depth	Magnitude
0	01/02/1965	13:44:18	19.246	145.616	131.6	6.0
1	01/04/1965	11:29:49	1.863	127.352	80.0	5.8
2	01/05/1965	18:05:58	-20.579	-173.972	20.0	6.2
3	01/08/1965	18:49:43	-59.076	-23.557	15.0	5.8
4	01/09/1965	13:32:50	11.938	126.427	15.0	5.8

Converting Date & Time to Datestamp for ease of use

```
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()
```

OUTPUT:



	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

Declaring X & Y, and splitting the dataset into tranning & testing model

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

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_st
ate=6) #Random State 6 gives best result for this case
```

```
reg = RandomForestRegressor(random_state=6)
reg.fit(X_train, y_train)
reg.predict(X_test)
```

OUTPUT:

```
array([[ 5.809 , 21.871 ],
       [ 5.899 , 36.615 ],
       [ 5.741 , 40.311 ],
       ...,
       [ 6.176 , 52.2967],
       [ 5.898 , 40.888 ],
       [ 5.769 , 28.978 ]])
```

Accuracy Score:

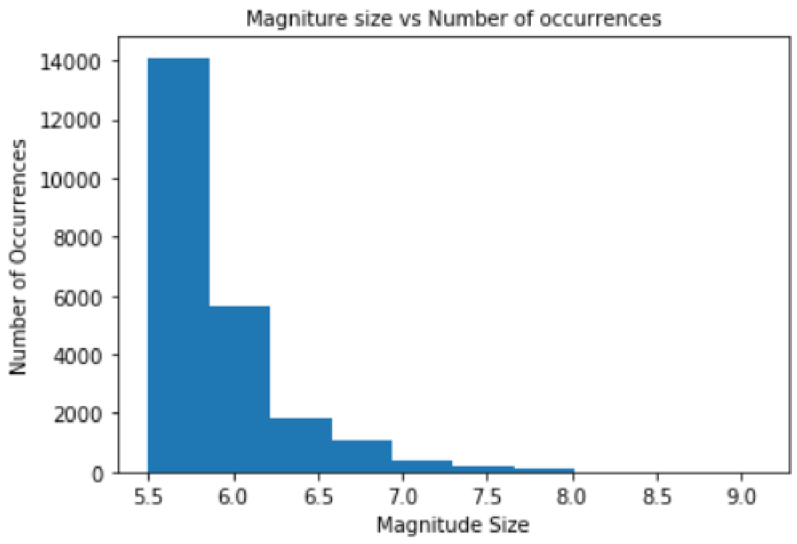
```
reg.score(X_test, y_test)
```

OUTPUT:

```
0.8733937415770104
```

Analytics:

```
plt.hist(data['Magnitude'])
plt.xlabel('Magnitude Size')
plt.ylabel('Number of Occurrences')
plt.title('Magniture size vs Number of occurrences', fontweight = 20, fontsize =
10)
plt.show()
```



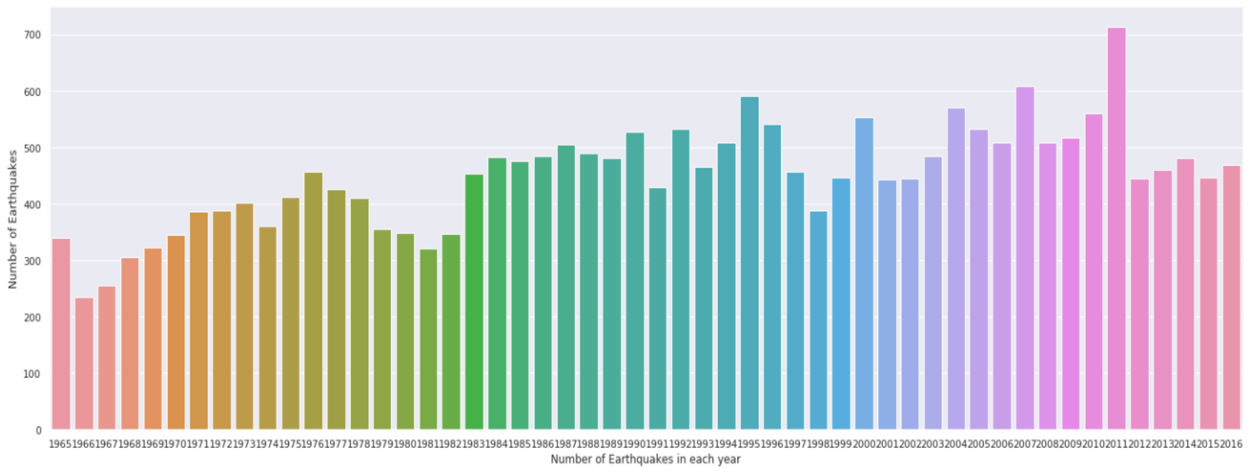
```
data['date']=data['Date'].apply(lambda x: pd.to_datetime(x))
data['year']=data['date'].apply(lambda x:str(x).split('-')[0])
```

Number of Earthquakes each year

```
plt.figure(figsize=(25,8))
sns.set(font_scale=1.0)
sns.countplot(x="year",data=data)
plt.ylabel('Number of Earthquakes')
plt.xlabel('Number of Earthquakes in each year')
```

OUTPUT:

```
Text(0.5, 0, 'Number of Earthquakes in each year')
```

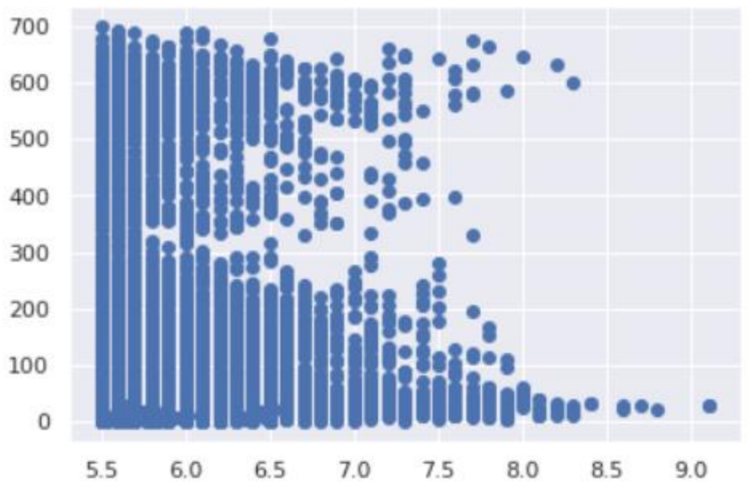


Magnitude vs Depth

```
plt.scatter(data["Magnitude"],data["Depth"])
```

OUTPUT:

```
<matplotlib.collections.PathCollection at 0x7f15eaf3d250>
```

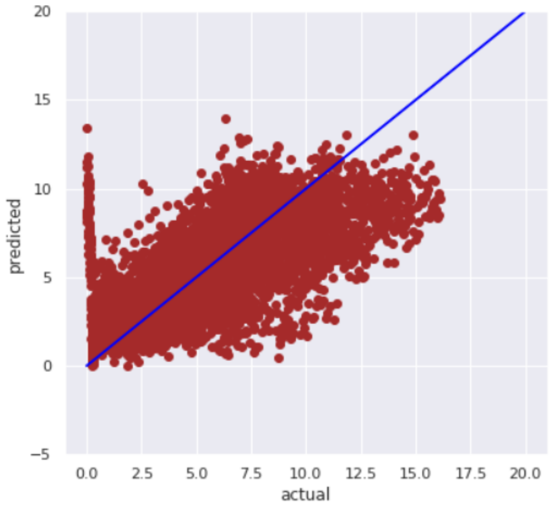


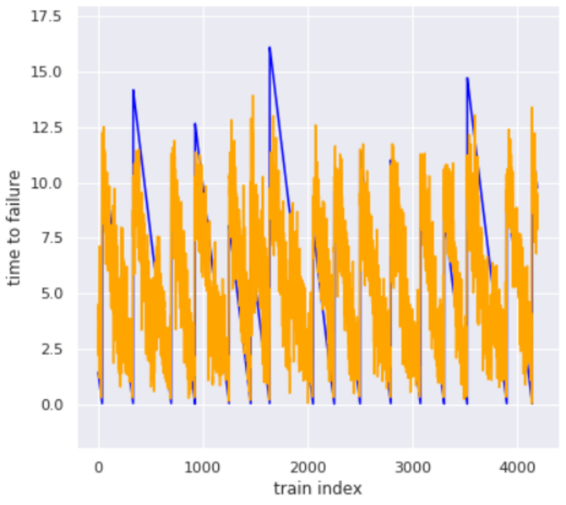
Gradient Boosted Machines:

Using Gradient Boosted Machines (GBM) for earthquake prediction is another powerful approach. GBM is an ensemble learning method that builds multiple decision trees in a sequential manner, each tree correcting the errors of the previous one. It's particularly effective when dealing with complex relationships and non-linear patterns in the data.This model achieve a F1 score of 88.3%, which under performs against the random forest but performs strongly against the KNN model

```
fixed_params = {
    'objective': 'regression_l1',
    'boosting': 'gbdt',
    'verbosity': -1,
    'random_seed': 19,
    'n_estimators': 80000,
    'max_depth': -1
}

param_grid = {
    'learning_rate': [0.1, 0.05, 0.01, 0.005],
    'num_leaves': list(range(2, 60, 2)),
    'feature_fraction': [0.8, 0.85, 0.9, 0.95, 1],
    'subsample': [0.8, 0.9, 0.95, 1],
    'lambda_l1': [0, 0.2, 0.4, 0.6, 0.8, 2, 5],
    'lambda_l2': [0, 0.2, 0.4, 0.6, 0.8, 2, 5],
    'min_data_in_leaf': [10, 20, 40, 60, 100],
    'min_gain_to_split': [0, 0.001, 0.01, 0.1],
}

best_score = 999
dataset = lgb.Dataset(X_train, label=target)  # no need to scale features

for i in range(600):
    params = {k: random.choice(v) for k, v in param_grid.items()}
    params.update(fixed_params)
    result = lgb.cv(params, dataset, nfold=10, early_stopping_rounds=100,
                    stratified=False)

    if result['l1-mean'][-1] < best_score:
        best_score = result['l1-mean'][-1]
        best_params = params
        best_nrounds = len(result['l1-mean'])
```

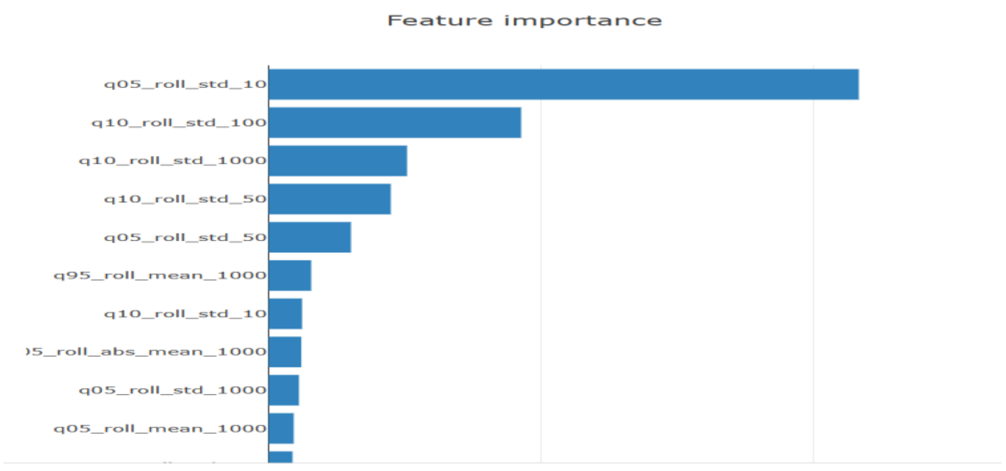
```
print("Best mean score: {:.4f}, num rounds: {}".format(best_score, best_nrounds
))
print(best_params)
gb_pipe = make_pipeline(lgb.LGBMRegressor(**best_params))
gb_oof = make_predictions(X_train, target, gb_pipe)

Best mean score: 2.0047, num rounds: 92
{'learning_rate': 0.05, 'num_leaves': 8, 'feature_fraction': 0.95, 'subsample': 0.8, '
lambda_l1': 5, 'lambda_l2': 5, 'min_data_in_leaf': 40, 'min_gain_to_split': 0.1, 'o
bjective': 'regression_l1', 'boosting': 'gbdt', 'verbosity': -1, 'random_seed': 19, 'n_
estimators': 80000, 'max_depth': -1}
```

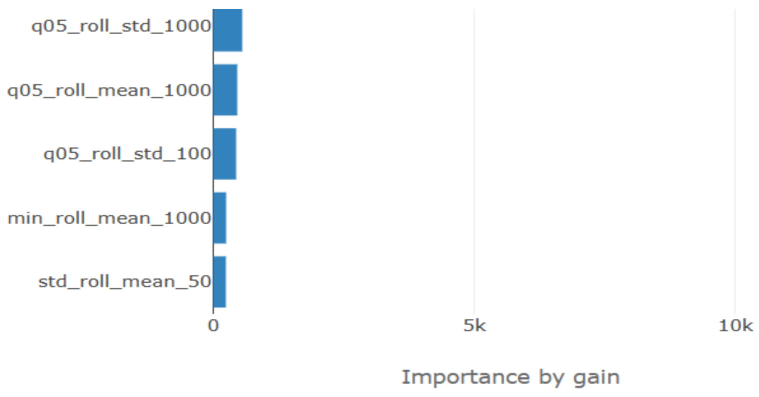




Now let's have a look at the **feature importance**:

```
def plot_feature_importance(x, y, columns):
    importance_frame = pd.DataFrame()
    for (train_index, valid_index) in fold_generator(x, y):
        reg = lgb.LGBMRegressor(**best_params)
        reg.fit(x.iloc[train_index], y.iloc[train_index],
                early_stopping_rounds=100, verbose=False,
                eval_set=[(x.iloc[train_index], y.iloc[train_index]),
                         (x.iloc[valid_index], y.iloc[valid_index])])
        fold_importance = pd.DataFrame()
        fold_importance["feature"] = columns
        fold_importance["gain"] = reg.booster_.feature_importance(importance_ty
pe='gain')
        #fold_importance["split"] = reg.booster_.feature_importance(importance_t
ype='split')
        importance_frame = pd.concat([importance_frame, fold_importance], axis
=0)
  mean_importance = importance_frame.groupby('feature').mean().reset_index()
    mean_importance.sort_values(by='gain', ascending=True, inplace=True)
    trace = go.Bar(y=mean_importance.feature, x=mean_importance.gain,
                  orientation='h', marker=dict(color='rgb(49,130,189)'))

    layout = go.Layout(
        title='Feature importance', height=800, width=600,
        showlegend=False,
        xaxis=dict(
            title='Importance by gain',
            titlefont=dict(size=14, color='rgb(107, 107, 107)'),
            domain=[0.15, 1]
        ),
    )
  fig = go.Figure(data=[trace], layout=layout)
    iplot(fig)
    plot_feature_importance(X_train, target, X_train.columns)
```





Data Processing Pipelines:

This subsection is foccused on preparing the data to give the models the best opportunity of identifying the signal in the data. The following steps will be applied:

1.Label Encode Target Variable

2.Ensure variables have the correct data types

3.Covert nomial variables to numeric with onehot encoding

4.Restrain outliers

5.Centre and scale all numerical variables

6.Remove variables with either no or minimal variance

Firstly we encode the objects y_train and y_test to be value labels ranging from 0 to 4, corresponding to Grades 1 - 5.

```
# ------------ Target Processing ------------

# Target Variable Transformer
preprocessor_tar = LabelEncoder()
y_train = preprocessor_tar.fit_transform(y_train)
y_test = preprocessor_tar.fit_transform(y_test)

# Visualise Counts
pd.DataFrame(y_train).value_counts(normalize = True)

# Clean objects
del preprocessor_tar
```

The second step is to create a pipeline of processing steps which can be applied to the predictor features. These are all combined into a single pipe which easily allows for the same processing steps to be applied to different dataframes.The print out below shows that with have significantly increased the dimensionality of our dataset as a result of one hot encoding the categorical variables. I did not venture into it this time, but it would have been sensible to apply some dimensionality reduction to the columns created by one hot encoding.

```
# Print x train shape before processing
print("Before preprocessing there were {} rows and {} columns".format(*x_train
.shape))

# ------------ Predictor Processing ------------
# Identify columns
fts_cvt_obj = ['district_id', 'vdcmun_id', 'ward_id']
fts_outlier = ['age_building']
fts_cat = df_stru.drop(fts_cvt_obj, axis = 1).select_dtypes(include=['object']).dr
op('damage_grade', axis = 1).columns
```

```
fts_num = df_stru.select_dtypes(np.number).columns

# Convert to object Transformer
def covert_to_object(x):
    '''Converts a column to object'''
    return pd.DataFrame(x).astype(object)
trans_to_object = Pipeline(steps = [('convert_to_object', FunctionTransformer(c
overt_to_object))])

# Outlier Restriction
trans_outlier = Pipeline(steps = [('Outlier_scaler', RobustScaler(quantile_range
= (0,0.9)))])

# Categorical Transformer
trans_cat = Pipeline(steps = [('onehot', OneHotEncoder(handle_unknown='ignor
e'))])

# Numerical Transformer
trans_num = Pipeline(steps = [('scaler', StandardScaler()),
                              ('MinMax', MinMaxScaler())])

# Zero or Near Zero variance
trans_nzv = Pipeline(steps = [('nzv', VarianceThreshold(threshold = 0.01))])

# Create a single Preprocessing step for predictors
preprocessor_preds = ColumnTransformer(
    transformers=[
        ('convert_to_object', trans_to_object, fts_cvt_obj), # Convert data types
        ('Outlier', trans_outlier, fts_outlier), # Outlier treatment
        ('num', trans_num, fts_num), # Centre and scale
        ('cat', trans_cat, fts_cat), # One Hot encode
        ('nzv', trans_nzv,[]) # One Hot encode
    ])

# Apply the transformations to both train and test
x_train = preprocessor_preds.fit_transform(x_train)
x_test = preprocessor_preds.fit_transform(x_test)

# Print x train shape before processing
print("After preprocessing there are {} rows and {} columns".format(*x_train.sh
ape))

# Clean objects
```

```
del fts_cvt_obj, fts_outlier, fts_cat, fts_num, covert_to_object, trans_to_object, t
rans_outlier, trans_cat, trans_num, trans_nzv
```

```
Before preprocessing there were 609675 rows and 31 columns
After preprocessing there are 609675 rows and 69 columns
```

Cross Validation Strategy:

5 fold cross validation will be used as the strategy, which will provide confidenc e that we are not overfitting and allow us to make fair comparisons between co mpeting models. Ultimately this approach splits the data into 5 random partition s and builds 5 models, for each model it uses 4 partitions of data to train the mo dels and the fifth for testing.

```
# Store the Kfold object
kfold = KFold(n_splits=5, random_state=1989, shuffle = True)
```

Model Building and Comparison

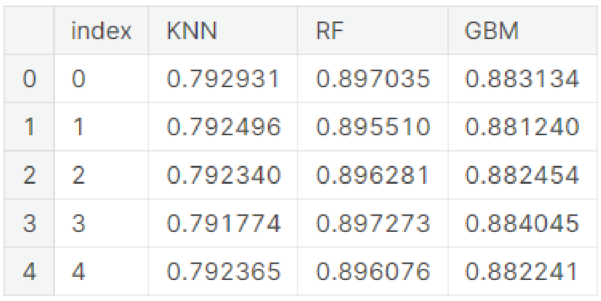
The three models are built across the same 5 folds of data and the output below shows how each model performs using the F1 score evaluation metric. We can see that the models are blah blah blah. The stats present a summarised evaluation, indicating how the model performed generally across all of the folds.

- KNN model: It performed the worst achieving a F1 score of 79.2%, the low standard deviation score indiates that similar perfomance was achieve across the 5 folds
- Random Forest model: This model achieved a F1 score of 90%, just over a 10% point increase relative to the simple KNN model. The standard deviation score for this model also indcates similar performance across the folds.
- Gradient Boosted Machine model: This model achieve a F1 score of 88.3%, which under performs against the random forest but performs strongly against the KNN model

```
    # List of classification models
    classifiers = [('KNN', KNeighborsClassifier(3)),
    ('RF', RandomForestClassifier()),
        ('GBM', GradientBoostingClassifier())]
# Evaluate each model
        results = []
        names = []
        for name, model in classifiers:
        cv_results = cross_val_score(model, x_train, y_train, cv=kfold, scoring='f
        1_micro')
        results.append(cv_results)
        names.append(name)
        print("%s: %f (%f)" % (name, cv_results.mean(), cv_results.std()))
```

```
KNN: 0.792381 (0.000370)
RF: 0.896435 (0.000644)
GBM: 0.882623 (0.000935
```

```
# Summarise scores
pd.DataFrame(np.transpose(results), columns = names).reset_index()
```



	index	KNN	RF	GBM
0	0	0.792931	0.897035	0.883134
1	1	0.792496	0.895510	0.881240
2	2	0.792340	0.896281	0.882454
3	3	0.791774	0.897273	0.884045
4	4	0.792365	0.896076	0.882241

Conclusion

This task was used as an opportunity to explore python to build a predictive supervised learning model. There were many aspects of the analysis which could have been extended such as a more extensive feature engineering, applying dimensionality reduction and hyperparameter turning. That said, without this an f1 score of 89.4% isn't that bad. If you have made it this far, I hope you enjoyed reading it, I certainly have enjoyed learning.