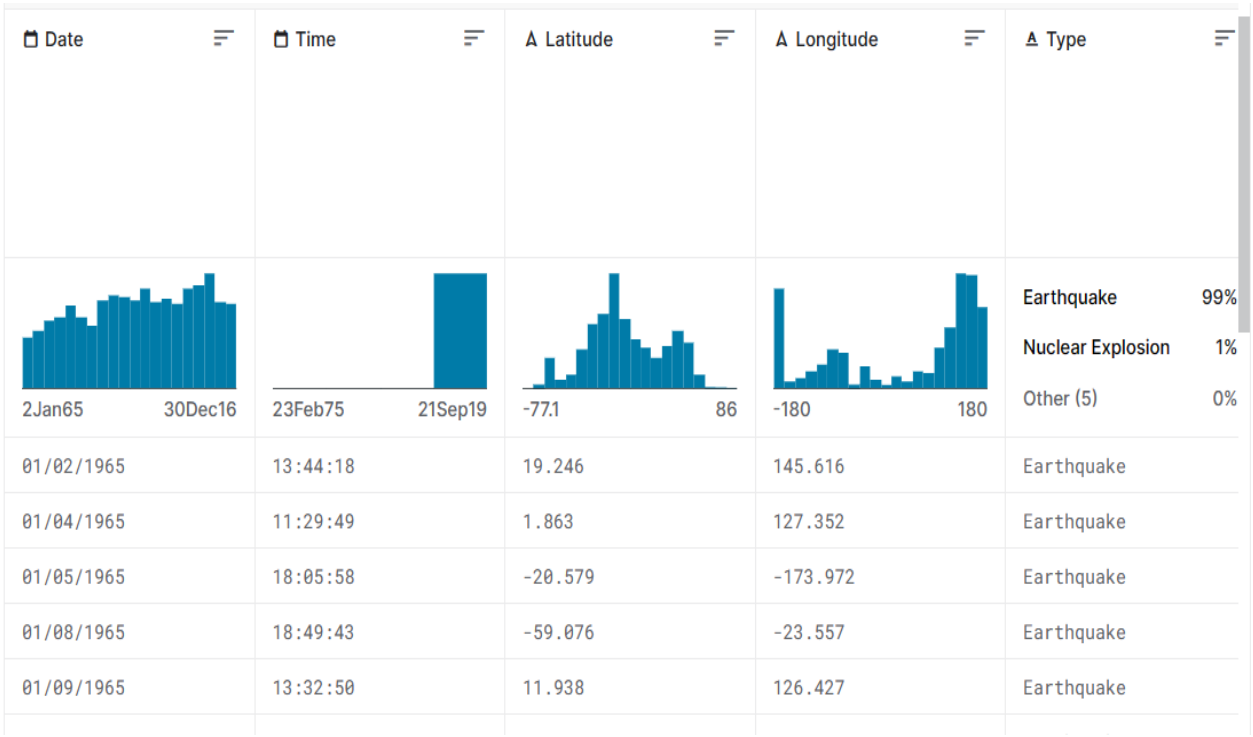
EARTHQUAKE PREDICTION MODEL USING PYTHON

DATASETS:

When a specific field is researched in terms of machine learning, the first question is where to find data. As for earthquake datasets, various organizations and research institutions are constantly monitoring seismic activity of all over the world. The structure of earthquake datasets are usually presented in the form of a table, each record of which corresponds to a certain seismic event. The sets of attributes are different for data published in different datasets.

Dataset Link: https://www.kaggle.com/datasets/usgs/earthquake-database



Date	Time	Latitude	Longitude	Type
2Jan65 – 30Dec16	23Feb75 – 21Sep19	-77.1 – 86	-180 – 180	Earthquake 99%, Nuclear Explosion 1%, Other (5) 0%
01/02/1965	13:44:18	19.246	145.616	Earthquake
01/04/1965	11:29:49	1.863	127.352	Earthquake
01/05/1965	18:05:58	-20.579	-173.972	Earthquake
01/08/1965	18:49:43	-59.076	-23.557	Earthquake
01/09/1965	13:32:50	11.938	126.427	Earthquake

HYPER PARAMETER TUNING:

1) OPTIMIZING LINEAR REGRESSION:

The linear regression model might be the simplest predictive model that learns from data .The model has one coefficient for each input and the predicted output is simply the weights of some inputs and coefficients. In this section, we will optimize the coefficients of a linear regression model.

First, let's define a synthetic regression problem that we can use as the focus of optimizing the model.

First, we need to split the dataset into train and test sets. It is important to hold back some data not used in optimizing the model so that we can prepare a reasonable estimate of the performance of the model when used to make predictions on new data.

```
# optimize linear regression coefficients for regression dataset
from numpy.random import randn
from numpy.random import rand
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# linear regression
def predict_row(row, coefficients):
# add the bias, the last coefficient
result = coefficients[-1]
# add the weighted input
for i in range(len(row)):
result += coefficients[i] * row[i]
return result

# use model coefficients to generate predictions for a dataset of rows
def predict_dataset(X, coefficients):
yhats = list()
for row in X:
# make a prediction
yhat = predict_row(row, coefficients)
# store the prediction
yhats.append(yhat)
return yhats

# objective function
def objective(X, y, coefficients):
# generate predictions for dataset
yhat = predict_dataset(X, coefficients)
# calculate accuracy
score = mean_squared_error(y, yhat)
```

```
return score

# hill climbing local search algorithm
def hillclimbing(X, y, objective, solution, n_iter, step_size):
# evaluate the initial point
solution_eval = objective(X, y, solution)
# run the hill climb
for i in range(n_iter):
# take a step
candidate = solution + randn(len(solution)) * step_size
# evaluate candidate point
candidte_eval = objective(X, y, candidate)
# check if we should keep the new point
if candidte_eval <= solution_eval:
# store the new point
solution, solution_eval = candidate, candidte_eval
# report progress
print('>%d %.5f' % (i, solution_eval))
return [solution, solution_eval]

# define dataset
X, y = make_regression(n_samples=1000, n_features=10,
n_informative=2, noise=0.2, random_state=1)
# split into train test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33)
# define the total iterations
n_iter = 2000
# define the maximum step size
step_size = 0.15
# determine the number of coefficients
n_coef = X.shape[1] + 1
# define the initial solution
solution = rand(n_coef)
# perform the hill climbing search
coefficients, score = hillclimbing(X_train, y_train, objective, solution,
n_iter, step_size)
print('Done!')
print('Coefficients: %s' % coefficients)
print('Train MSE: %f' % (score))
# generate predictions for the test dataset
```

```
yhat = predict_dataset(X_test, coefficients)
# calculate accuracy
score = mean_squared_error(y_test, yhat)
print('Test MSE: %f' % (score))
```

2)DECISION TREE:

AVOIDING OVERFITTING:

The techniques for preventing overfitting remain largely the same as for decision tree classifiers. However, it seems that not many people actually take the time to prune a decision tree for regression, but rather they elect to use a random forest regressor (a collection of decision trees) which are less prone to overfitting and perform better than a single optimized tree. The common argument for using a decision tree over a random forest is that decision trees are easier to interpret, you simply look at the decision tree logic. However, in a random forest, you're not going to want to study the decision tree logic of 500 different trees. Luckily for us, there are still ways to maintain interpretability within a random forest without studying each tree manually.

IMPLEMENTATION:

```
from sklearn.tree import DecisionTreeRegressor

start2 = time.time()

regressor = DecisionTreeRegressor(random_state = 40)

regressor.fit(X_train,y_train)

ans2 = regressor.predict(X_test)

end2 = time.time()

t2 = end2-start2
```

3)KNN MODEL:

A k-nearest neighbors is algorithm used for classification and regression. It classifies a new data point by finding the k-nearest points in the training dataset and assigns it the majority class among those neighbors.

Machine learning algorithms have hyperparameters that allow you to tailor the behavior of the algorithm to your specific dataset. Hyperparameters Tuning can

improve model performance by about 20% to a range of 77% for all evaluation matrices. Hyperparameter tuning in k-nearest neighbors (KNN) is important because it allows us to optimize the performance of the model. The KNN algorithm has several hyperparameters that can significantly affect the accuracy of the model, such as the number of nearest neighbors to consider (k), the distance metric used to measure similarity, and the weighting scheme used to aggregate the labels of the nearest neighbors.

Required Libraries:

- NumPy
- Pandas
- Scikit-learn
- Matplotlib.

Things to keep in mind when performing turning:

1. **Understand the parameters:** The main hyperparameter to tune in k-nearest neighbors is k, the number of neighbors to consider. Other parameters include distance metrics, weights, and algorithm types.

2. **Select a distance metric:** Choose the right distance metric to measure the similarity between the data points. Common distance metrics include Euclidean, Manhattan, and cosine distance.

3. **Select an appropriate value for k:** Selecting a value for k is crucial in k-nearest neighbors. A larger value of k provides a smoother decision boundary but may not be suitable for all datasets. A smaller value of k may lead to overfitting.

4. **Choose an algorithm type:** k-nearest neighbors has two algorithm types: brute-force and tree-based. Brute-force algorithm computes the distances between all pairs of points in the dataset while tree-based algorithm divides the dataset into smaller parts.

5. **Cross-validation:** Cross-validation is a technique used to validate the performance of the model. It involves splitting the dataset into training and testing sets and evaluating the model's performance on the testing set.

6. **Grid search:** Grid search is a hyperparameter tuning technique that involves testing a range of values for each hyperparameter to find the best combination of values.

7. **Random search:** Random search is another hyperparameter tuning technique that randomly selects a combination of hyperparameter values to test.

8. **Bias-variance tradeoff:** k-nearest neighbors is prone to overfitting due to the high variance in the model. Regularization techniques such as L1 and L2 regularization can be used to mitigate overfitting.

9. **Data preprocessing:** Data preprocessing plays a crucial role in k-nearest neighbors. Scaling the data using techniques such as normalization and standardization can improve the model's performance. Outlier removal and feature selection can also help improve the model's performance.

IMPLEMENTATION:

```
from sklearn.neighbors import KNeighborsRegressor

start3 = time.time()

knn = KNeighborsRegressor(n_neighbors=6)

knn.fit(X_train, y_train)

ans3 = knn.predict(X_test)

end3 = time.time()

t3 = end3-start3
```