Predicting Housing Prices Using Machine Learning

GitHub Repo

By: Thinh Le, Sandeep Virk, Ellycea Burke, Justin Marsh

Problem

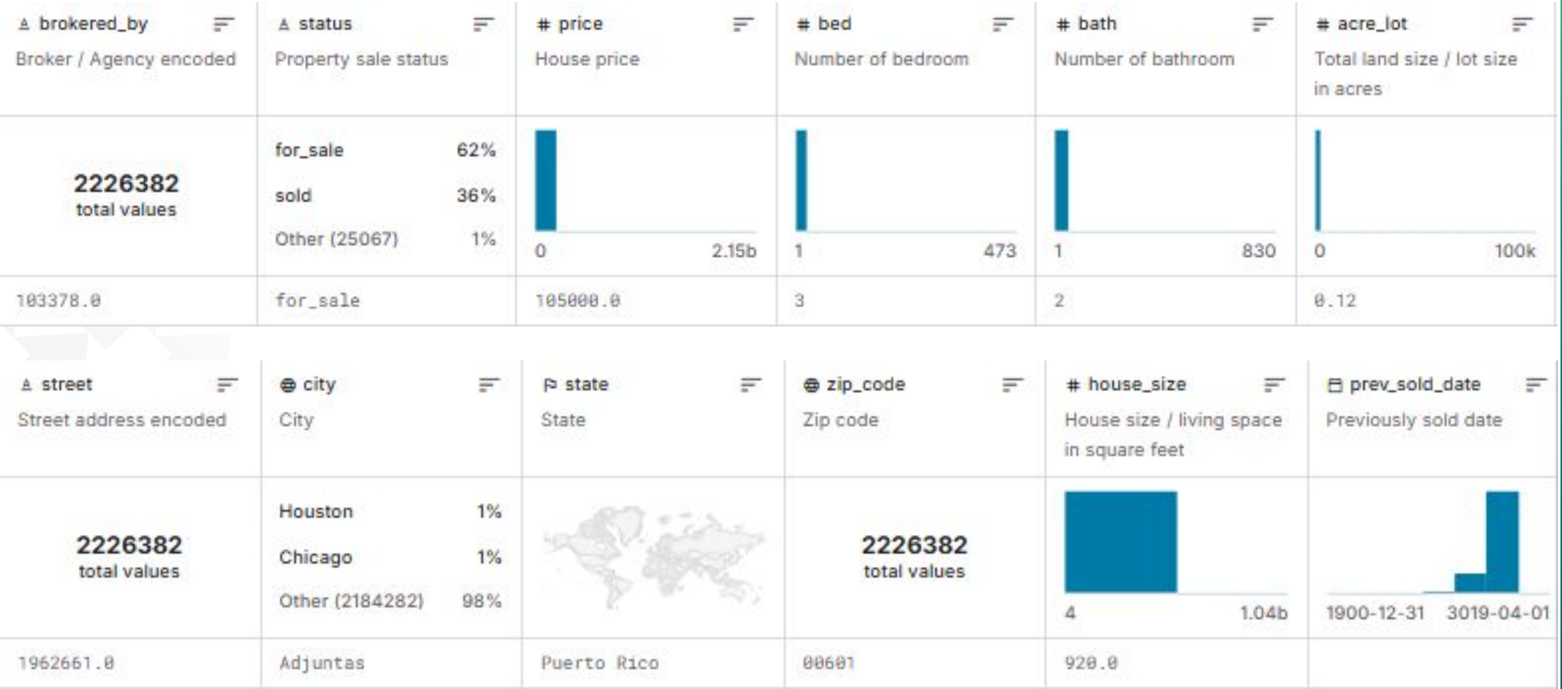
A Comparative Analysis of Traditional Models and Neural Networks

This project aims to predict housing prices based on various features in California such as size, and number of bedrooms; using a dataset of real estate properties. We will implement and compare the performance of traditional regression models (Linear Regression) with a neural network model (Multi-Layer Perceptron). The better performing model will be deployed in our web application, allowing users to input property features and receive a predicted price. This study will provide insights into the effectiveness of neural networks for regression tasks compared to traditional approaches.

Dataset

USA Real Estate Dataset

Link: https://www.kaggle.com/datasets/ahmedshahriarsakib/usa-real-estate-dataset



brokered_by	status	price	bed	bath	acre_lot
Broker / Agency encoded	Property sale status	House price	Number of bedroom	Number of bathroom	Total land size / lot size in acres
2226382 total values	for_sale 62% sold 36% Other (25067) 1%	0 – 2.15b	1 – 473	1 – 830	0 – 100k
103378.0	for_sale	105000.0	3	2	0.12

street	city	state	zip_code	house_size	prev_sold_date
Street address encoded	City	State	Zip code	House size / living space in square feet	Previously sold date
2226382 total values	Houston 1% Chicago 1% Other (2184282) 98%		2226382 total values	4 – 1.04b	1900-12-31 – 3019-04-01
1962661.0	Adjuntas	Puerto Rico	00601	920.0	

Data Cleaning

We will pre process the data to ensure it is clean, normalized and ready for modeling.

First we will drop all null values present in the dataset since there are a substantial amount in the null value check. We will also drop any duplicate rows.

```
print(df.isnull().sum())
✓ 0.2s

brokered_by        4533
status                0
price              1541
bed              481317
bath             511771
acre_lot         325589
street            10866
city               1407
state                 8
zip_code            299
house_size       568484
prev_sold_date   734297
dtype: int64
```

```
df = df.dropna().drop(columns=["brokered_by", "status", "prev_sold_date", "street"])

# Display the cleaned DataFrame
print(df)
✓ 0.3s

          price  bed  bath  acre_lot          city           state  \
502    110000.0  7.0   3.0      0.09        Dorado     Puerto Rico
2270   950000.0  5.0   4.0      0.99  Saint Thomas  Virgin Islands
2277  6899000.0  4.0   6.0      0.83  Saint Thomas  Virgin Islands
3409   525000.0  3.0   3.0      0.45        Agawam   Massachusetts
3410   289900.0  3.0   2.0      0.36        Agawam   Massachusetts
```

Data Cleaning

We will also want to remove outliers such as houses with 10 or more bedrooms/bathrooms (these could be buildings, multi-family residentials, luxury mansions, or data entry errors.

```
df = df[(df['bed'] < 10) & (df['bath'] < 10) & (df['price'] > 50000) & (df['price'] < 50000000)]
df
```

	price	bed	bath	acre_lot	city	state	zip_code	house_size
0	105000.0	3.0	2.0	0.12	Adjuntas	Puerto Rico	601.0	920.0
1	80000.0	4.0	2.0	0.08	Adjuntas	Puerto Rico	601.0	1527.0
2	67000.0	2.0	1.0	0.15	Juana Diaz	Puerto Rico	795.0	748.0
3	145000.0	4.0	2.0	0.10	Ponce	Puerto Rico	731.0	1800.0
5	179000.0	4.0	3.0	0.46	San Sebastian	Puerto Rico	612.0	2520.0
...	...	...	...	...	...	...	...	...
2226377	359900.0	4.0	2.0	0.33	Richland	Washington	99354.0	3600.0
2226378	350000.0	3.0	2.0	0.10	Richland	Washington	99354.0	1616.0
2226379	440000.0	6.0	3.0	0.50	Richland	Washington	99354.0	3200.0
2226380	179900.0	2.0	1.0	0.09	Richland	Washington	99354.0	933.0
2226381	580000.0	5.0	3.0	0.31	Richland	Washington	99354.0	3615.0

Pre Processing Data

Now we can start to encode any categorical values to unique numerical values. This will affect State, City and Zip Codes. There is also an option to use embedding layers, however that will limit our prediction inputs. (since the categorical data in the dataset is not fixed) Due to our computation limit, we will only use 43000 entries from the state of California.

We should also normalize our numeric features since the values range differ across the entire dataset. This will ensure our data is scaled effectively, and help with performance

city	state	zip_code
0	0	0
1	1	1
1	1	1
2	2	2
2	2	2
...	...	...
1697	46	22030
1697	46	22030
1697	46	22030
1697	46	22030
1697	46	22030

```
#Normalize numerical features
numerical_features = ['acre_lot', 'house_size']
feature_mean = df[numerical_features].mean()
feature_std = df[numerical_features].std()

df[numerical_features] = (df[numerical_features]
                          - feature_mean) / feature_std
```

Pre Processing Data

Now we will convert the inputted location string value into latitude and longitude coordinates. Then we will convert those coordinates into Earth Cartesian coordinates, so that our model can do spatial calculation.

```
def get_cartesian_coordinates(df):
    df_cartesian = df[:]

    # Convert latitude and longitude to radians
    df_cartesian['loc_lat_rad'] = np.radians(df_cartesian['loc_lat'])
    df_cartesian['loc_long_rad'] = np.radians(df_cartesian['loc_long'])

    # Calculate Cartesian coordinates
    df_cartesian['loc_x'] = np.cos(df_cartesian['loc_lat_rad']) * np.cos(df_cartesian['loc_long_rad'])
    df_cartesian['loc_y'] = np.cos(df_cartesian['loc_lat_rad']) * np.sin(df_cartesian['loc_long_rad'])
    df_cartesian['loc_z'] = np.sin(df_cartesian['loc_lat_rad'])

    # Now we can remove all categorical columns and intermediate math columns
    df_out = df_cartesian.drop(columns=['city', 'state', 'zip_code', 'location', 'loc_lat', 'loc_long', 'loc_lat_rad', 'loc_
    return df_out # The Cartesian data is already normalized

df_sample_cartesian = get_cartesian_coordinates(df_sample_longlat)[:]
df_sample_cartesian
```

Model

Now we can start training our model, first we will initialize our Linear & MLP Regression models, then initialize training and testing the Data Loader. After we can train the models and evaluate.

```
class LinearRegressionModel(nn.Module):
    def __init__(self, in_dims):
        super().__init__()
        self.nn = nn.Linear(in_dims, 1)

    def forward(self, x):
        return self.nn(x)
```

```
class MLPRegressionModel(nn.Module):
    def __init__(self, in_dims):
        super().__init__()
        # Define the MLP architecture
        self.nn = nn.Sequential(
            nn.Linear(in_dims, 512),
            nn.ELU(), # Neurons with neg
            nn.Dropout(0.15), # Add drop
            nn.Linear(512, 256),
            nn.ELU(),
            nn.Dropout(0.15),
            nn.Linear(256, 128),
            nn.ELU(),
            nn.Dropout(0.15),
            nn.Linear(128, 64),
            nn.ELU(),
            nn.Dropout(0.15),
            nn.Linear(64, 32),
            nn.ELU(),
            nn.Dropout(0.15),
            nn.Linear(32, 1)
        )

    def forward(self, x):
        return self.nn(x)
```

```
def train(model_class: nn.Module, n_epochs: int, weight_decay=0.0):
    model = model_class(in_dims)
    print(torchinfo.summary(model))

    # Loss function
    # loss_fn = nn.MSELoss() # Mean Squared Error
    loss_fn = nn.HuberLoss(delta=0.8) # less sensitive to outliers

    # Optimizer
    optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weigh

    # Learning rate scheduler: Reduce learning rate when validation
    scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.2,

    # Statistics
    best_train_mse = np.inf # Initialize to infinity
    best_mse = np.inf # Initialize to infinity
    best_weights = None
    history_mse = []
    history_r2 = []
    history_train_mse = []  # To store training MSE
    history_train_r2 = []  # To store training R2

    # Early stopping parameters
    patience = 20 # Number of epochs to wait for improvement
    epochs_without_improvement = 0 # Track epochs without improvemen

    for epoch in range(n_epochs):
        model.train()

        # Variables to accumulate training loss and predictions
        total_train_loss = 0
        y_train_true_all = []
        y_train_pred_all = []
```

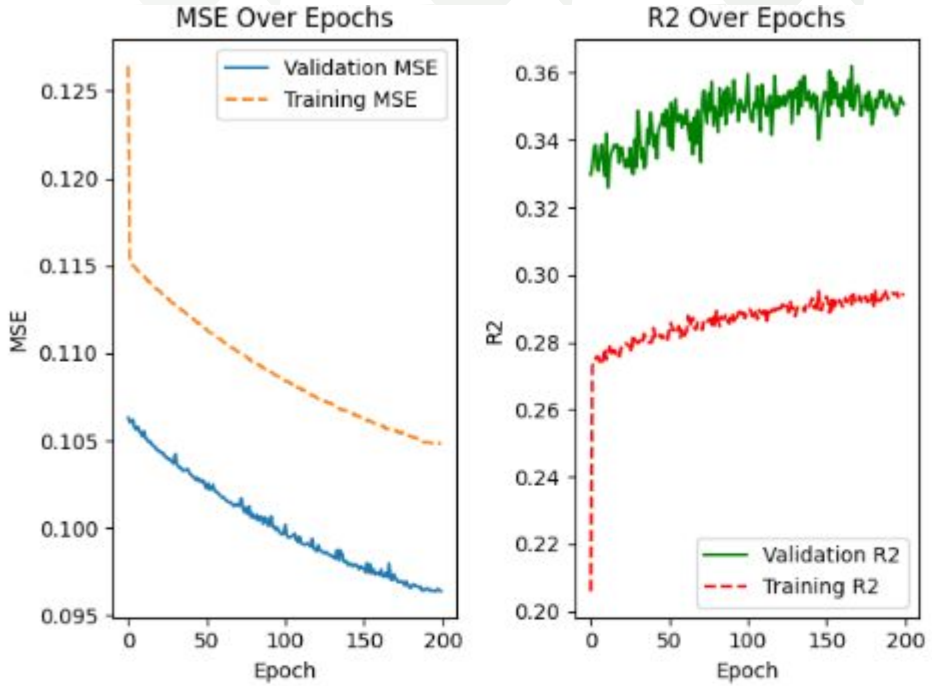
```
with tqdm.tqdm(train_loader, unit="batch", mininterval=0, disable=True) as bar:
    bar.set_description(f"Epoch {epoch}")
    for X_batch, y_batch in bar:
        # Forward pass
        y_pred = model(X_batch)
        loss = loss_fn(y_pred, y_batch)
        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        # Update weights
        optimizer.step()
        # Print progress
        bar.set_postfix(mse=float(loss))

        total_train_loss += loss.item()
        y_train_true_all.append(y_batch.cpu().numpy())
        y_train_pred_all.append(y_pred.cpu().detach().numpy())

# Calculate training MSE
train_mse = total_train_loss / len(train_loader)
if train_mse < best_train_mse: best_train_mse = train_mse
history_train_mse.append(train_mse)
# Flatten training true and predicted values for R2 calculation
y_train_true_all = np.concatenate(y_train_true_all, axis=0)
y_train_pred_all = np.concatenate(y_train_pred_all, axis=0)
# Calculate training R2
train_r2 = r2_score(y_train_true_all, y_train_pred_all)
history_train_r2.append(train_r2)

# Evaluate accuracy at the end of each epoch
model.eval()
total_loss = 0
y_true_all = []
y_pred_all = []

with torch.no_grad():
    for X_batch, y_batch in test_loader:
        y_pred = model(X_batch)
        loss = loss_fn(y_pred, y_batch)
        total_loss += loss.item()

        # Store true and predicted values for R2 calculation
        y_true_all.append(y_batch.cpu().numpy())
        y_pred_all.append(y_pred.cpu().numpy())

mse = total_loss / len(test_loader)
mse = float(mse)
history_mse.append(mse)

# Flatten true and predicted values for R2 calculation
y_true_all = np.concatenate(y_true_all, axis=0)
y_pred_all = np.concatenate(y_pred_all, axis=0)
# Calculate R2
r2 = r2_score(y_true_all, y_pred_all)
history_r2.append(r2)
```

Model

After running our model, we can see that the simple linear regression model is too simple and underfitting for this dataset.

```
lr_model = train(LinearRegressionModel, 200)
```

```
=================================================================
Layer (type:depth-idx)                   Param #
=================================================================
LinearRegressionModel                    --
├─Linear: 1-1                            8
=================================================================
Total params: 8
Trainable params: 8
Non-trainable params: 0
=================================================================
Epoch 00187: reducing learning rate of group 0 to 2.0000e-04.
Train MSE: 0.10
Train RMSE: 0.32
Train R2: 0.30
MSE: 0.10
RMSE: 0.31
R2: 0.36
```

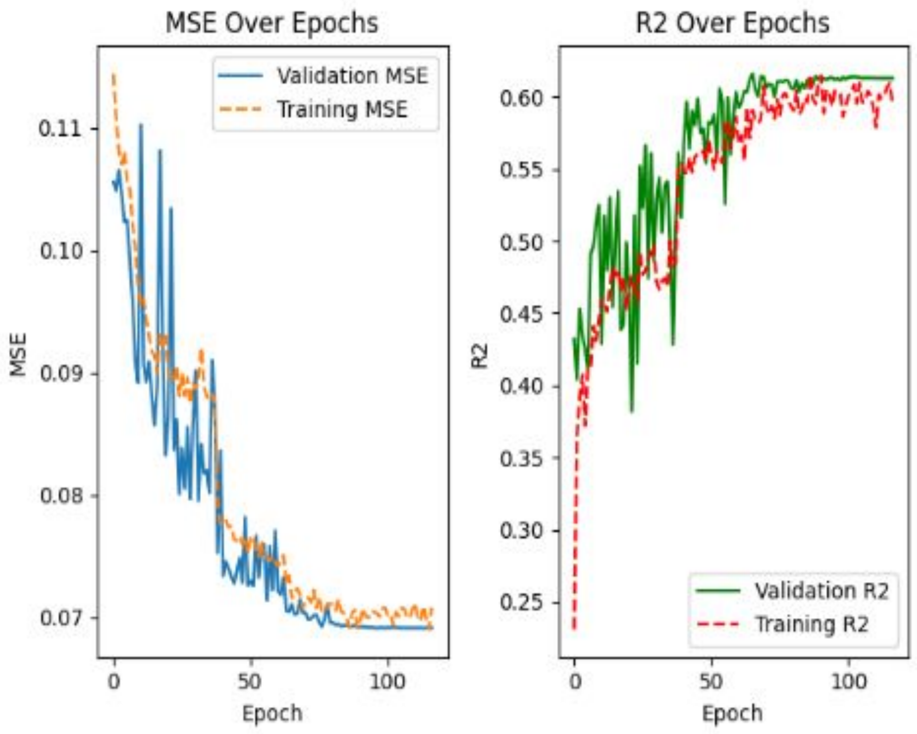


Model

However, when using the MLP model with more layers will allow fine-tuning and improve performance. And such will be used as our model in our web application.

```
mlp_model = train(MLPRegressionModel, 200)
```

```
=================================================================
Layer (type:depth-idx)                   Param #
=================================================================
MLPRegressionModel                       --
├─Sequential: 1-1                        --
│    └─Linear: 2-1                       4,096
│    └─ELU: 2-2                          --
│    └─Dropout: 2-3                      --
│    └─Linear: 2-4                       131,328
│    └─ELU: 2-5                          --
│    └─Dropout: 2-6                      --
│    └─Linear: 2-7                       32,896
│    └─ELU: 2-8                          --
│    └─Dropout: 2-9                      --
│    └─Linear: 2-10                      8,256
│    └─ELU: 2-11                         --
│    └─Dropout: 2-12                     --
│    └─Linear: 2-13                      2,080
│    └─ELU: 2-14                         --
│    └─Dropout: 2-15                     --
│    └─Linear: 2-16                      33
=================================================================
Total params: 178,689
Trainable params: 178,689
Non-trainable params: 0
=================================================================
Epoch 00038: reducing learning rate of group 0 to 2.0000e-04.
Epoch 00063: reducing learning rate of group 0 to 4.0000e-05.
Epoch 00083: reducing learning rate of group 0 to 8.0000e-06.
Epoch 00089: reducing learning rate of group 0 to 1.6000e-06.
Epoch 00103: reducing learning rate of group 0 to 3.2000e-07.
Epoch 00109: reducing learning rate of group 0 to 6.4000e-08.
Epoch 00115: reducing learning rate of group 0 to 1.2800e-08.
Early stopping at epoch 117
Train MSE: 0.07
Train RMSE: 0.26
Train R2: 0.62
MSE: 0.07
RMSE: 0.26
R2: 0.62
```

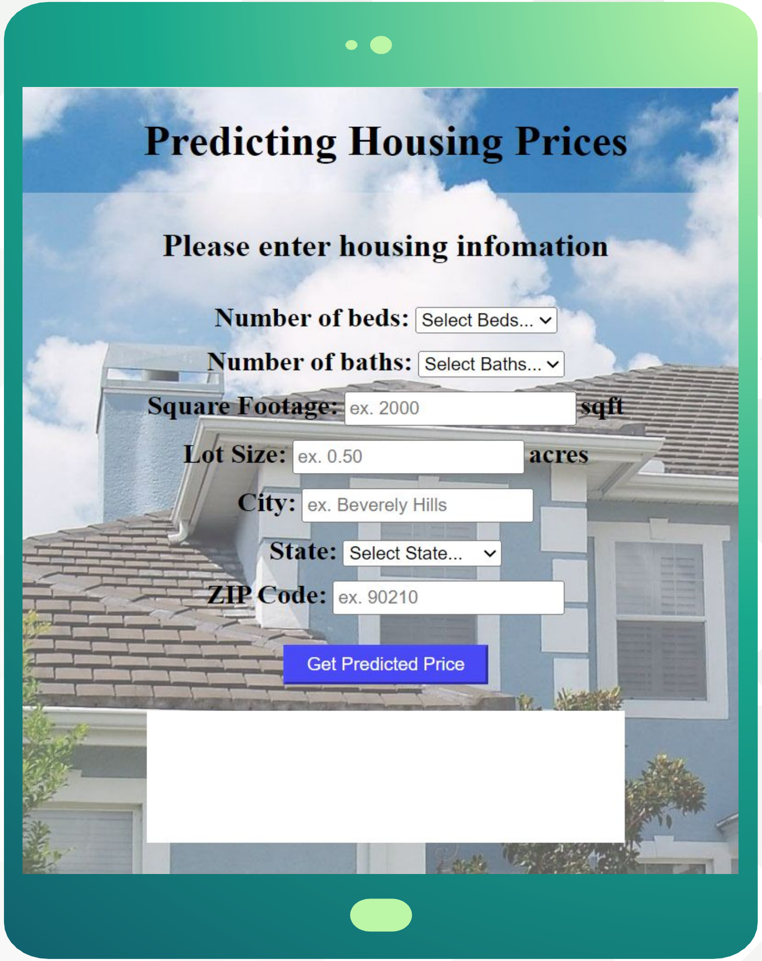


Deployment

GitHub Repository: https://github.com/ThinhLe188/csci4050-final-project.git

Frontend application

Users will enter housing information that lines up with the dataset we used. Once input is submitted, they will receive their predicted housing price.



Backend

Modules used in the backend:

- Math
- Joblib
- Numpy
- Onnxruntime
- FastAPI
- BaseModel
- Geopy
- CORSMiddleware from Starlette

app > server.py > ...

```python
# run command:
# uvicorn app.server:app --reload
import joblib
import numpy as np
import onnxruntime as ort
from fastapi import FastAPI
from pydantic import BaseModel
from geopy.geocoders import Nominatim
from starlette.middleware.cors import CORSMiddleware
import math
```

Backend

Data comes from FastAPI, it is then normalized. City, state and address are taken as location inputs and converted to cartesian coordinates to make all data numerical. Model then predicts based on data and sends it back to the frontend.

```python
def get_coordinates(location):
    try:
        loc = geolocator.geocode(location.split(',')[-1].strip()) # try with only the zip code
        if loc:
            return convert_to_cartesian(loc.latitude, loc.longitude)
        else:
            loc = geolocator.geocode(location)
            if loc:
                return convert_to_cartesian(loc.latitude, loc.longitude)
            else:
                return None, None, None
    except Exception as e:
        return None, None, None
```

```python
@app.get("/")
def read_root():
    return {"message": "Welcome to the MLP Model API for predicting housing prices"}


@app.post("/predict")
def predict(data: InputData):
    location = f'{data.address}, {data.city}, {data.state}, {data.zip_code}'
    loc_x, loc_y, loc_z = get_coordinates(location)

    if loc_x and loc_y and loc_z:
        raw_inputs = np.array([data.num_bed, data.num_bath, data.acre_lot, data.house_size])
        norm_inputs = f_scaler.transform([raw_inputs])
        input_data = np.append(norm_inputs, [loc_x, loc_y, loc_z])
        outputs = ort_session.run(None, {"input": np.expand_dims(input_data, axis=0).astype(np.float32)})
        norm_price = outputs[0][0].item()
        original_price = p_scaler.inverse_transform(np.array([[norm_price]]))[0][0]
        return {"price": round(original_price), "prediction": norm_price}
    else:
        return {"error": "cannot find location"}
```

Thanks!