body = {'instances': instances, 'size': size}
response = get_pred_from_model(body, project_name, model_name)
print(response) → Get Results.

Send request to model and receive response.

PREDICTING HOUSE PRICES with REGRESSION USING TENSORFLOW

Features → ~~A~~ Year of sale, age at time of sale, distance from city center, no. of stores in locality, lat, long.

TASK 1 – Import Library

TASK 2.1 → Import Data

```
column_names = ['serial', 'date', 'age' ....]

df = pd.read_csv('data.csv', names = column_names)

df.head()
```

df – data frame variable
pd – pandas
'data.csv' – data without column names
names = column_names – Initializing Column names

TASK 2.2 → Check Missing data.

df.isna() ← will send true if value missing

df.isna().sum() ← will sum the values because we can't see each cell of the large dataset.

TASK 3 – Data Normalisation
↳ Change distribution of different features so that the values are in same ranges.

assumed ~~proud~~ no wrong.

we

False)

name)

back

_/_/_

3.1 Data Normalisation.

```
df = df.iloc[:, 1:]
```

$\underbrace{:}_{\text{All Rows}}$, $\underbrace{1:}_{\text{First Column is ignored.}}$

→ we selected all data except first column i.e the serial no. as it is not necessary.

```
df_norm = (df - df.mean())/df.std()
```

mean → df.mean(), Standard deviation → df.std()

```
df_norm.head()
```
← To view normalised data

3.2 Convert Label Value.

Predicted value will also be normalised. So, we define function to convert normalised predicted value back to original distribution. i.e the predicted price.

```
def convert_label_value(pred):
    return int(pred * y_std + y_mean)
```

Eg ⇒

df not df_norm as it contains ~~normal~~ normalised data

```
y_mean = df['price'].mean()
y_std = df['price'].std()
```

'price' → The ~~pred~~ value to be predicted

→ - - - - -

```
print(convert_label_value(0.35088))
```

TASK-4 Create Training and Test Sets.

_/_/_

4.1 Select Features

We have to remove price from list of features as it is a label (to be predicted) i.e we will take first 6 columns of df_norm.

$$x = \text{df\_norm.iloc}[\underbrace{:}_{\text{all rows}}, \underbrace{:6}_{\text{first 6 columns}}]$$

4.2 Select Label

label

$$y = \text{df\_norm.iloc}[\underbrace{:}_{\text{all rows.}}, \underbrace{-1}_{\text{last column}}]$$

4.3 Feature and label values

x-arr = x.values
y-arr = y.values

It extracts the numeric values i.e x-arr becomes 2-D array and y-arr becomes list as it had one column only

print (x-arr.shape) → (5000, 6)
print (y-arr.shape) → (5000,)

4.4 TRAIN AND TEST SPLIT

x-train, x-test, y-train, y-test = train_test_split(x-arr, y-arr, test_size = 0.05, random-state = 0)

TASK - 5 CREATE THE MODEL

Relu = Rectified Linear Unit.

def get_model():

model = Sequential ([Dense (10, input_shape = (6,), activation = "relu"),

Input to ~~the~~ first layer.

Hidden layer

Dense (20, activation = "relu"),

no. of nodes

Dense (5, activation = "relu")

Dense (1)

Output Layer. Since it is a regression problem we do not need activation function but just a linear output.

It is a Keras Class and cool thing is we can just pass on a list of layers to create model architecture. It is either Sequential or Functional.

The input_shape() automatically defines the hidden layer.

Before using a model we need to compile it

model.compile (
    loss = 'mse'
    optimizer = 'adam'
)

loss function mean square error. It is pretty common for regression problem.

optimization algo It is a variant of stochastic gradient descent called adam. It tries to minimise the loss function.

return model.

get_model().summary() → Summary of model

TASK 6 - MODEL TRAINING

6.1 - MODEL TRAINING

To use Validation loss for Early stopping. It is used on test set and not training set.

Hence it is better when making decision on when to stop training

We can set high epochs and model will stop when it sees no change ~~to~~ or improvement in validation loss.

es_cb = EarlyStopping(monitor='val_loss', patience=5)

Early stopping callback

If validation loss does not decrease for 5 epochs it stops training.

model = get_model()

preds_on_untrained = model.predict(x_test)

Untrained model gives random prediction but we will have something to compare.

history = model.fit(
    x_train, y_train,
    validation_data = (x_test, y_test),
    epochs = 100,
    callbacks = [es_cb]
)

Error/Loss when testing on test data

6.2 PLOT TRAINING LOSS and VALIDATION LOSS

Error/~~test~~Loss when testing on training data.

We will use plot_loss function to take a look at training and validation loss.

plot_loss(history)

It will plot a graph on its own

TASK-7 PREDICTIONS

7.1 PLOT RAW PREDICTIONS

We will use compare_predictions helper function to compare predictions of trained and untrained model.

preds_on_trained = model.predict(x_test)

compare_predictions(preds_on_untrained, preds_on_trained,
Ground Truth → y_test)

Predictions

axis - x → predictions
axis - y → label or ground truth

The function draws a graph on its own

Dotted blue line gives what the ideal model should have given.

7.2 PLOT PRICE PREDICTIONS

price_untrained = [convert_label_value(y) for y in preds_on_untrained]

price_trained = [convert_label_value(y) for y in preds_on_trained]

ground truth

price_test = [convert_label_value(y) for y in y_test]

compare_predictions(price_untrained, price_trained, price_test)

When we split data the first one is used for training the other can be used for

VALIDATION, EVALUATION OR TESTING

VALIDATION ↓ used to tune hyper parameters of model and is done on cross validation set.

EVALUATION OR TESTING ↓ test final performance of algo. and is done on test set.