Machine Learning for Robotics

Shafayetul Islam
Robotics and AI Instructor

Contact Me
✉️ Email: shafatsib@gmail.com 📞 Phone: +880178-0404749 📍 Dhaka, Bangladesh
Google Scholar Github Linkedin Youtube

Class 02-
- Dataset

What is a Dataset?

a dataset is just a collection of examples that we use to teach and test a model.

What is a Dataset?

a dataset is just a collection of examples that we use to teach and test a model.

Days	Price
Day 01	100
Day 02	150
Day 03	200
Day 04	250
Day 05	300
Day 06	350

Apple Price Dataset

What is a Dataset?

ID	Size_sqft	Bedrooms	Age_years	Price_USD
1	800	2	15	75,000
2	950	2	10	90,000
3	1200	3	8	120,000
4	1500	3	5	155,000
5	1800	4	3	190,000
6	2200	4	1	230,000

house price dataset

1. Datapoint (also called: sample, example, instance)

A **datapoint** is **one row** from your dataset.
It is one complete example with all its information.

Example row:

Size_sqft	Bedrooms	Age_years	Price_USD
1200	3	8	120,000

This **whole row** is **one datapoint**.

You can say:

- Number of datapoints = number of rows in the dataset.

2. Feature (also called: input, attribute, variable)

A **feature** is **one type of input information** you use to predict something.

In the house table:

- `Size_sqft` → feature
- `Bedrooms` → feature
- `Age_years` → feature

Features are usually the **columns on the left side** that we **use as input** to the model.

For the example row:

- Features = (1200, 3, 8)

So for ML, we often write:

- **x** = features of a datapoint

3. Target (also called: label, output, y)

The **target** is **what we want to predict** using the features.

In the house price example:

- `Price_USD` is the **target**.

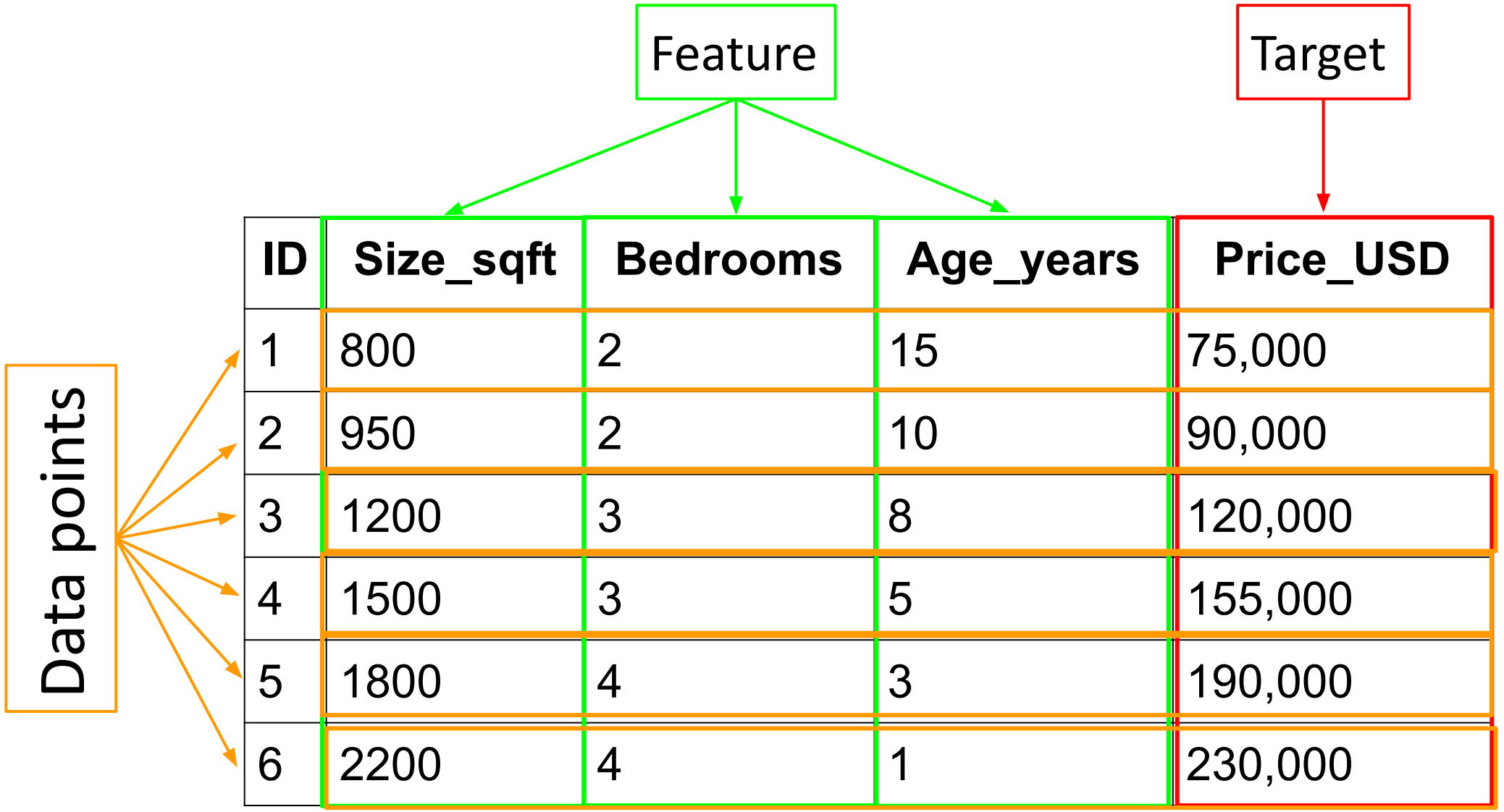
For the same row:

- Target = 120,000

In notation:

- **x** = (1200, 3, 8) → input features
- **y** = 120,000 → target (house price)

Dataset Overview



ID	Size_sqft	Bedrooms	Age_years	Price_USD
1	800	2	15	75,000
2	950	2	10	90,000
3	1200	3	8	120,000
4	1500	3	5	155,000
5	1800	4	3	190,000
6	2200	4	1	230,000

Class 02-
- Numpy

Why Numpy is used in ML?

Numpy is used in ML because it gives fast, memory-efficient operations on large numeric arrays and matrices, with all the linear algebra and statistical tools needed to implement and understand ML algorithms.

Example of few Operations:

- dot products
- matrix multiplication
- transposes
- norms
- eigenvalues / SVD (for PCA, etc.)

0. Importing Numpy

```
import numpy as np
```

1. Creating Np arrays

1D Array:

```
array_1D = [1, 2, 3, 4]
np_1D = np.array(array_1D)
```

2D Array:

```
array_2D =[
                [1, 2, 3, 4],
                [3, 2, 5, 6],
                [10, 12, 11]
            ]
np_2D = np.array(array_2D)
```

2. Inspecting arrays

```
X = np.array([
                [1, 2, 3],
                [4, 5, 6]
            ])

X.shape   # (2, 3)
X.ndim    # 2  (2D array)
X.dtype   # data type (e.g. int64, float64)
```

3. Special arrays

```
np.zeros((3, 4))      # all zeros, shape (3,4)
np.ones((2, 2))       # all ones, shape (2,2)
np.arange(0, 10, 2)   # [0 2 4 6 8]
np.linspace(0, 1, 5)  # [0.  0.25 0.5 0.75 1.]
```

4. Indexing and slicing (taking rows/columns)

```
X = np.array([
    [800,  2, 15],   # datapoint 0
    [950,  2, 10],   # datapoint 1
    [1200, 3,  8],   # datapoint 2
    [1500, 3,  5],   # datapoint 3
])
```

Single element

```
X[0, 0]  # 800 (row 0, column 0)
X[2, 1]  # 3   (row 2, column 1)
```

Row (datapoint)

```
X[0]     # [800   2  15]  → first datapoint
```

Column (feature)

```
X[:, 0]  # [800  950 1200 1500] → all sizes
X[:, 1]  # [2 2 3 3]            → all bedrooms
```

Slicing (ranges)

```
X[0:2]      # first 2 rows
X[1:4, 0:2] # rows 1–3, columns 0–1
```

5. Aggregations (mean, std, etc.) and axis

```
X = np.array([
    [800,  2, 15],
    [950,  2, 10],
    [1200, 3,  8],
    [1500, 3,  5],
])
```

```
X.mean()            # mean of all numbers
X.mean(axis=0)      # mean of each column (feature-wise)
X.mean(axis=1)      # mean of each row (datapoint-wise)

X.std(axis=0)       # standard deviation per feature
X.min(axis=0)       # min of each feature
X.max(axis=0)       # max of each feature
```

6. Reshape and flatten

```
a = np.array([1, 2, 3, 4])
```

```
a_reshaped = a.reshape(2, 2)
# [[1 2]
#  [3 4]]

a_flat = a.flatten()  # [1 2 3 4]
```

7. Concatenation (combine data)

```
A = np.array([    [1, 2],
                  [3, 4]  ])
B = np.array([    [5, 6],
                  [7, 8]  ])
```

```
np.vstack([A, B])
# [[1 2]
#  [3 4]
#  [5 6]
#  [7 8]]

np.hstack([A, B])
# [[1 2 5 6]
#  [3 4 7 8]]
```

8. Boolean indexing (filtering rows)

```
prices = np.array([75000, 90000, 120000, 155000])
```

```
mask = prices > 100000

mask
# [False False  True  True]

prices[mask]
# [120000 155000]
```

9. Scalar operations

```
X = np.array([1, 2, 3, 4])
```

```
# Add a scalar
X_plus_10 = X + 10      # [11 12 13 14]

# Multiply by a scalar
X_times_2 = X * 2       # [2 4 6 8]

# Divide by a scalar
X_div_2 = X / 2         # [0.5 1.  1.5 2.]

# Power
X_squared = X ** 2      # [1 4 9 16]
```

10. Array / matrix operations

A. Elementwise operations (same shape)

If two arrays have the same shape, operators like `+`, `-`, `*`, `/`, `**` work **elementwise**.

```
a = np.array([1, 2, 3])
b = np.array([10, 20, 30])

print(a + b)   # [11 22 33]
print(a * b)   # [10 40 90]
print(a / b)   # [0.1 0.1 0.1]
```

10. Array / matrix operations

For 2D (matrices):

```
A = np.array([[1, 2],
              [3, 4]])
B = np.array([[10, 20],
              [30, 40]])

print(A + B)
# [[11 22]
#  [33 44]]

print(A * B)   # elementwise multiply, NOT matrix multiply!
# [[ 10  40]
#  [ 90 160]]
```

10. Array / matrix operations

B. Matrix multiplication (dot product) – `@` or `np.dot`

This is the **linear algebra** operation used in ML models.

1D dot product (vector · vector)

```
w = np.array([1, 2, 3])      # weights
x = np.array([4, 5, 6])      # input

dot = np.dot(w, x)           # 1*4 + 2*5 + 3*6 = 32
# or: dot = w @ x
```

10. Array / matrix operations

2D: matrix · vector

This matches the **linear model** idea.

```
# X: (n_samples, n_features)
X = np.array([
    [1, 2, 3],   # sample 1
    [4, 5, 6],   # sample 2
])

w = np.array([0.1, 0.2, 0.3])  # (n_features,)
b = 0.5

y_pred = X @ w + b
# shape: (2,) → prediction for each sample
```

10. Array / matrix operations

2D: matrix · matrix

```
A = np.array([[1, 2],
              [3, 4]])

B = np.array([[5, 6],
              [7, 8]])

C = A @ B       # matrix multiplication
# or: C = np.dot(A, B)
```

Important:

- `A * B` → elementwise
- `A @ B` or `np.dot(A, B)` → matrix multiply

10. Array / matrix operations

C. Useful matrix operations

```
A = np.array([[1, 2, 3],
              [4, 5, 6]])

A.T          # transpose → shape (3, 2)

A.sum(axis=0)   # sum of each column
A.sum(axis=1)   # sum of each row
```

11. Shuffling data using Numpy

1. `np.random.shuffle` – shuffles in-place

Changes the array itself (no return value).

```
arr = np.array([1, 2, 3, 4, 5])
np.random.shuffle(arr)
print(arr)   # e.g. [3 5 1 4 2]
```

11. Shuffling data using Numpy

For 2D arrays, it shuffles the **rows**:

```
X = np.array([
    [1, 10],
    [2, 20],
    [3, 30],
])

np.random.shuffle(X)
print(X)
# rows are in random order, e.g.
# [[3 30]
#  [1 10]
#  [2 20]]
```

11. Shuffling data using Numpy

2. `np.random.permutation` – returns a shuffled copy (or shuffled indices)

(a) Shuffled copy of an array

```
arr = np.array([1, 2, 3, 4, 5])
shuffled = np.random.permutation(arr)

print(arr)       # original unchanged
print(shuffled)  # shuffled version
```

11. Shuffling data using Numpy

(b) Shuffled indices (very useful for X, y together)

```
X = np.array([[1, 10],
              [2, 20],
              [3, 30]])
y = np.array([0, 1, 0])

indices = np.random.permutation(len(X))

X_shuffled = X[indices]
y_shuffled = y[indices]

print(indices)       # e.g. [2 0 1]
print(X_shuffled)    # X rows shuffled
print(y_shuffled)    # y shuffled in same way
```

ANY QUESTIONS ?

THANK YOU

Contact Me

✉️ Email: shafatsib@gmail.com 📞 Phone: +880178-0404749 📍 Dhaka, Bangladesh

Google Scholar Github Linkedin Youtube