

# Linear Regression

---

# Loading the Data

Let's start by loading the housing dataset again

```
In [2]: import pandas as pd
import os
fname = os.path.join '..', 'data', 'real_estate.csv'
data = pd.read_csv(fname, sep=',')
data.head() # Head returns the first 5 elements
```

Out[2]:

	house age	dist to MRT	#stores	latitude	longitude	price per area
0	14.8	393.2606	6	24.96172	121.53812	7.6
1	17.4	6488.0210	1	24.95719	121.47353	11.2
2	16.0	4066.5870	0	24.94297	121.50342	11.6
3	30.9	6396.2830	1	24.94375	121.47883	12.2
4	16.5	4082.0150	0	24.94155	121.50381	12.8

- Our goal is to learn a model that can estimate "price per area"
- But how do we achieve that?

**The first step is using Maths to formalize the problem**

# Input, Output, Examples, Targets

**Formally, we say that:**

- All columns except the price represent the **input  $x$**  of our model
  - Inputs are often referred to as **attributes**
- The price represents the **output  $y$**  of our model
- Each row in the table represents one data point, i.e. an **example  $(x_i, y_i)$** 
  - $x_i$  is the input value for the  $i$ -th example
  - $y_i$  is the true output value (or **target**) for the  $i$ -th example

**Our goal is to learn a model  $f$  such that**

- When we feed the input  $x_i$  of each example to it
- ...The output value  $y_i = f(x_i)$  is as close as possible to  $y_i$

**This kind of **task** is known in ML as **supervised learning****

# Supervised Learning and Regression

**Supervised Learning is among the most common forms of ML**

Our **model** is a function  $f(x; \theta)$  with input  $x$  and **parameters**  $\theta$

- If the output is numeric, we speak of **regression**
- ...And we can define the approximation error over the example using, e.g.:

$$MSE(\theta) = \frac{1}{m} \sum_{i=1}^m (f(x_i, ; \theta) - y_i)^2$$

- "MSE" stands for **Mean Squared Error** and it's a common error metric

**Training in a (MSE) regression problem consists in solving**

$$\operatorname{argmin}_{\theta} MSE(\theta)$$

- I.e. choosing the parameters  $\theta$  to minimize approximation error

# Supervised Learning...And Linear Regression

We speak instead of **Linear Regression**

...When  $f$  is defined as a linear combination of features

$$f(x; \theta) = \sum_{i=1}^n \theta_j \phi_j(x)$$

- In general, a feature is any function that can be

**In our case each basis function will correspond to a specific input column**

...Plus a fixed term (think of that as a "1")

$$f(x; \theta) = \theta_0 + \theta_1 \{\text{age}\} + \theta_2 \{\text{MRT dist.}\} + \theta_3 \{\text{\#stores}\} + \theta_4 \{\text{latitude}\} + \theta_5 \{\text{longitude}\}$$

- The fixed term is called the **intercept**

# Supervised Learning...And Linear Regression

We speak instead of **Linear Regression**

...When  $f$  is defined as a linear combination of basis functions

$$f(x; \theta) = \sum_{i=1}^n \theta_j \phi_j(x)$$

**Linear regression is one of the simplest supervised learning approaches**

...But it is still a very good example!

- Since the model itself is relatively simple
- ...It will allow us to focus on the **key challenges** when using ML

# Separating Input and Output

Our first step will be separating our input and output

```
In [3]: cols = data.columns # columns in the dataframe  
X = data[cols[:-1]] # all columns except the last one  
X.head()
```

Out [3]:

	house age	dist to MRT	#stores	latitude	longitude
0	14.8	393.2606	6	24.96172	121.53812
1	17.4	6488.0210	1	24.95719	121.47353
2	16.0	4066.5870	0	24.94297	121.50342
3	30.9	6396.2830	1	24.94375	121.47883
4	16.5	4082.0150	0	24.94155	121.50381

We will focus on predicting the logarithm of the price per area

```
In [5]: import numpy as np  
y = np.log(data[cols[-1]]) # just the last column
```

- In practice, it's like predicting the order of magnitude

# Training and Test Set

**The model we learn should work well on all relevant data**

Formally, the model should generalize well

- How do we check whether this is the case?
- A typical approach: partitioning our dataset

**The basic idea is to split our data in two groups**

- The first group will actually be used for training
  - This will be called the training set
- The second group will be used only for model evaluation
  - This will be called the test set (or holdout set)

**With this trick, we can assess our model performance on unseen data**



# Training and Test Set

## There are a couple of catches

For this to work:

- The examples in the training set and the test set should be similar
- The test data should be a good match for the data we'll use for real

Ideally, we should have that:

**The training data should be representative of the true population**

This is the golden rule for building a training set

- Sometimes that's relatively easy to do
- ...But sometimes it may be difficult or impossible

# Training and Test Set

For learning our model, we will use scikit-learn

...Which provides a function to handle **shuffling** and **training/test splitting**:

```
In [6]: from sklearn.model_selection import train_test_split

X_tr, X_ts, y_tr, y_ts = train_test_split(X, y, test_size=0.34, random_state=42)

print(f'Size of the training set: {len(X_tr)}')
print(f'Size of the test set: {len(X_ts)}')
```

```
Size of the training set: 273
Size of the test set: 141
```

The function `train_test_split`

- Randomly shuffles the data (optionally with a fixed seed `random_state`)
- Puts a fraction `test_size` of the data in the test set
- ...And the remaining data in the training set
- Both the input and the output data is processed in this fashion

# Training and Test Set

Using separate test set is **extremely important**

...Because we want our model to work on **new data**

- We have no use for a model that **learns the input data perfectly**
- ...But that **behaves poorly on unseen data**
- In these cases, we say that the model **does not generalize**

By keeping a separate test set we can simulate this evaluation

**However, beware of exceptions!**

Sometimes, you it impossible to guarantee train/test similarity

- E.g. when making forecasts over time, the **historical** system behavior
- ...Can be different from the **future** system behavior
- In that case, the train/test split should simulate the expected difference

The trick is to think of what the train and test data will be **at deployment time**

# Fitting the Model

We can now train a linear model

```
In [7]: from sklearn.linear_model import LinearRegression  
  
m = LinearRegression()  
m.fit(X_tr, y_tr)
```

Out[7]:

```
▼ LinearRegression ⓘ ?  
LinearRegression()
```

We obtain the estimated output via the **predict** method:

```
In [8]: y_pred_tr = m.predict(X_tr)  
        y_pred_ts = m.predict(X_ts)
```

- The predictions (unlike the targets) are not guaranteed to be integers
- ...But that is still fine, since it's easy to interpret them

# Evaluation

## Finally, we need to evaluate the prediction quality

A common approach is using metrics. Here are a few examples:

- The **Mean Absolute Error** is given by:

$$MAE = \frac{1}{m} \sum_{i=1}^m |f(x_i) - y_i|$$

- The **Root Mean Squared Error** is given by:

$$RMSE = \sqrt{\frac{1}{m} \sum_{i=1}^m (f(x_i) - y_i)^2}$$

Both the RMSE and MAE are relatively easy to read

- They are expressed in the same unit as the original variable

# Evaluation

- The coefficient of determination ( $R^2$  coefficient) is given by:

$$R^2 = 1 - \frac{\sum_{i=1}^m (f(x_i) - y_i)^2}{\sum_{i=1}^m (y_i - \tilde{y})^2}$$

where  $\tilde{y}$  is the average of the  $y$  values

**The coefficient of determination is a useful, but more complex metric:**

- Its maximum is 1: an  $R^2 = 1$  implies perfect predictions
- Having a known maximum make the metric very readable
- It can be arbitrarily low (including negative)
- It can be subject to a lot of noise if the targets  $y$  have low variance

**Using the MSE directly for evaluation is usually a bad idea**

...Since it is a square, and therefore not easy to parse for a human

# Evaluation

Let's see the values for our example

```
In [9]: from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error

print(f'MAE on the training data: {mean_absolute_error(y_tr, y_pred_tr):.3}')
print(f'MAE on the test data: {mean_absolute_error(y_ts, y_pred_ts):.3}')
print(f'RMSE on the training data: {np.sqrt(mean_squared_error(y_tr, y_pred_tr)):.3}')
print(f'RMSE on the test data: {np.sqrt(mean_squared_error(y_ts, y_pred_ts)):.3}')
print(f'R2 on the training data: {r2_score(y_tr, y_pred_tr):.3}')
print(f'R2 on the test data: {r2_score(y_ts, y_pred_ts):.3}')
```

```
MAE on the training data: 0.143
MAE on the test data: 0.177
RMSE on the training data: 0.207
RMSE on the test data: 0.253
R2 on the training data: 0.691
R2 on the test data: 0.645
```

# Evaluation

## Let's see the values for our example

```
In [9]: from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error

print(f'MAE on the training data: {mean_absolute_error(y_tr, y_pred_tr):.3}')
print(f'MAE on the test data: {mean_absolute_error(y_ts, y_pred_ts):.3}')
print(f'RMSE on the training data: {np.sqrt(mean_squared_error(y_tr, y_pred_tr)):.3}')
print(f'RMSE on the test data: {np.sqrt(mean_squared_error(y_ts, y_pred_ts)):.3}')
print(f'R2 on the training data: {r2_score(y_tr, y_pred_tr):.3}')
print(f'R2 on the test data: {r2_score(y_ts, y_pred_ts):.3}')
```

```
MAE on the training data: 0.143
MAE on the test data: 0.177
RMSE on the training data: 0.207
RMSE on the test data: 0.253
R2 on the training data: 0.691
R2 on the test data: 0.645
```

- In general, we have better predictions on the training set than on the test set
- This is symptomatic of some **overfitting**
- I.e. we are learning patterns that don't translate to unseen data

Later on, we will see some techniques to deal with this situation

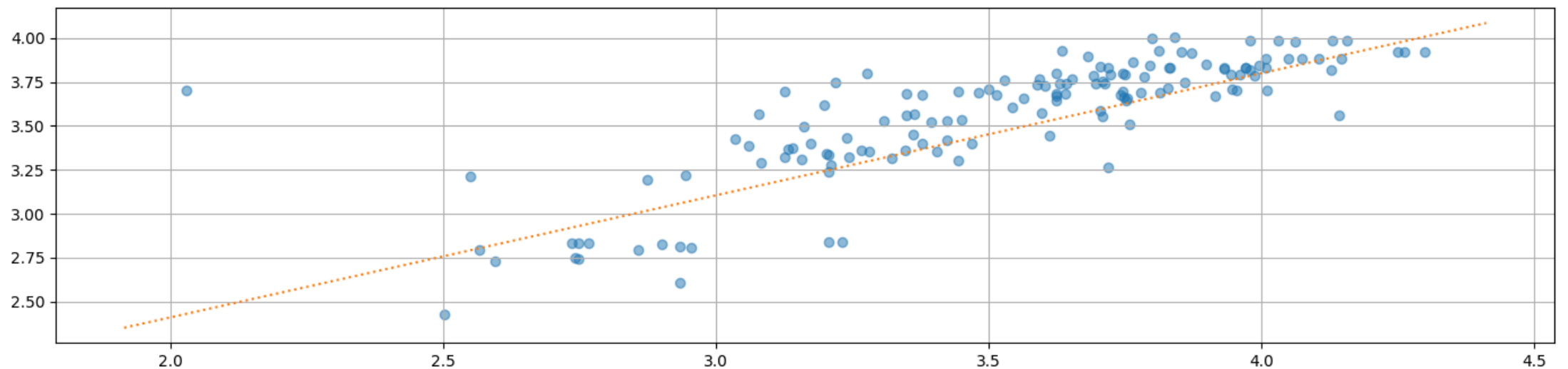


# Evaluation

As an (important!) alternative to metrics, we can use **scatter plots**

We can show the true vales on the x-axis, the predictions on the y-axis

```
In [10]: from matplotlib import pyplot as plt
plt.figure(figsize=figsize)
plt.scatter(y_ts, y_pred_ts, alpha=0.5)
plt.plot(plt.xlim(), plt.ylim(), linestyle=':', color='tab:orange')
plt.tight_layout(); plt.grid(':')
```



This gives us a better idea of which kind of mistakes the model is making

# Conclusions and Take-Home Messages

- Basic formulation of supervised learning
  - I.e. learning a model from available examples
  - ...When the examples contain values for both the input and the output
- Basic linear regression model
  - One the simplest approaches for supervised learning
  - I.e. the output is a linear combination of the input values
  - Regression = we estimate a numeric quantity
- Train/test set split
  - Needed to evaluate our model on unseen data (generalization)
- Evaluation of regression models
  - Make sure to compare the performance on both training and test data
  - Metrics (e.g. RMSE, MAE) provide a compact evaluation
  - Scatter plot for a more fine-grained evaluation