



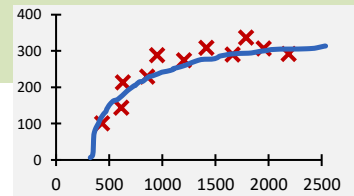
UCCD2063

# Artificial Intelligence Techniques

---

Unit 03:

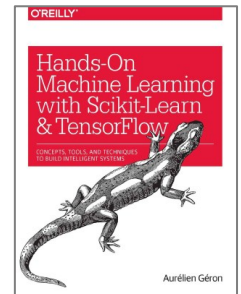
## The Regression Pipeline – 1/2



# Outline

- Regression
- Machine Learning Pipeline for Regression
  1. Look at the Big Picture
  2. Get Data
  3. Explore Data
  4. Prepare Data
  5. Select & train model
  6. Fine-tune the model
  7. Launch, monitor & maintain

## Reference:

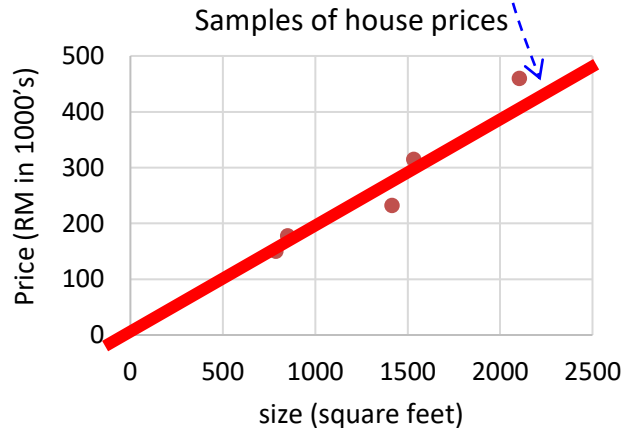


*Géron Chapter 2*

# Regression

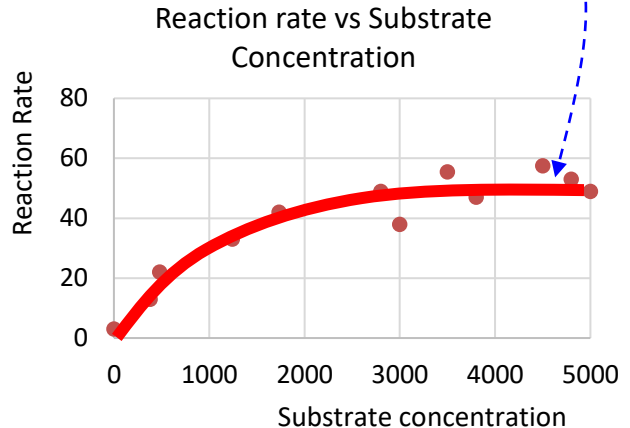
- Learns a function that maps the *input features*  $x$  to a *targeted variable*  $y$  where  $y$  is a **continuous** and scalar value
- Examples:

Output  
variable  
(Continuous)



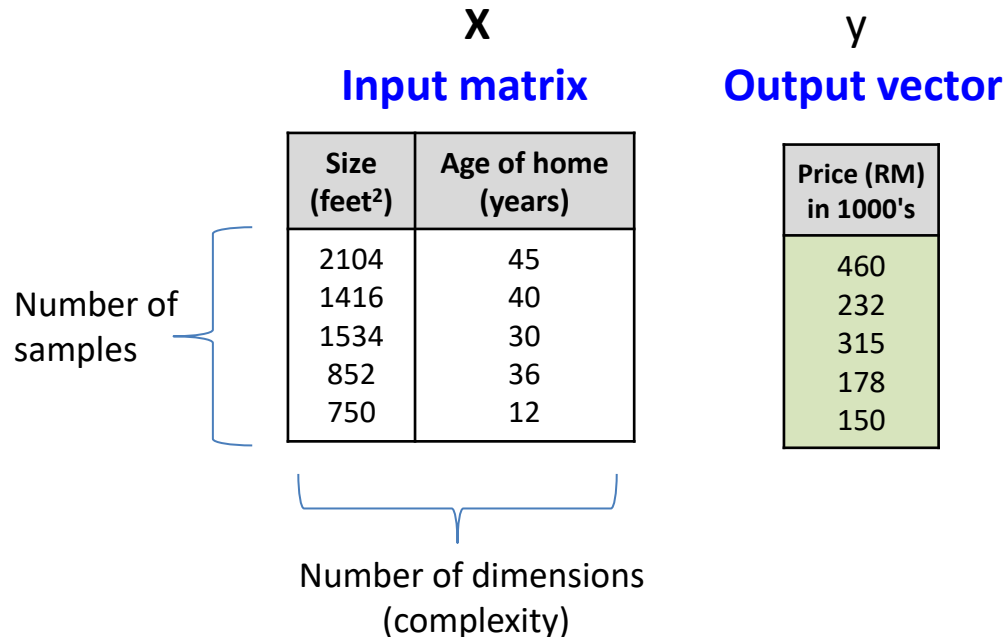
Input variable

Regression aims to find the  
best model (line) to  
represent these points



# Input feature and Targeted output vector

- Input and output of the machine learning are typically in terms of a **matrix** or **vector**:

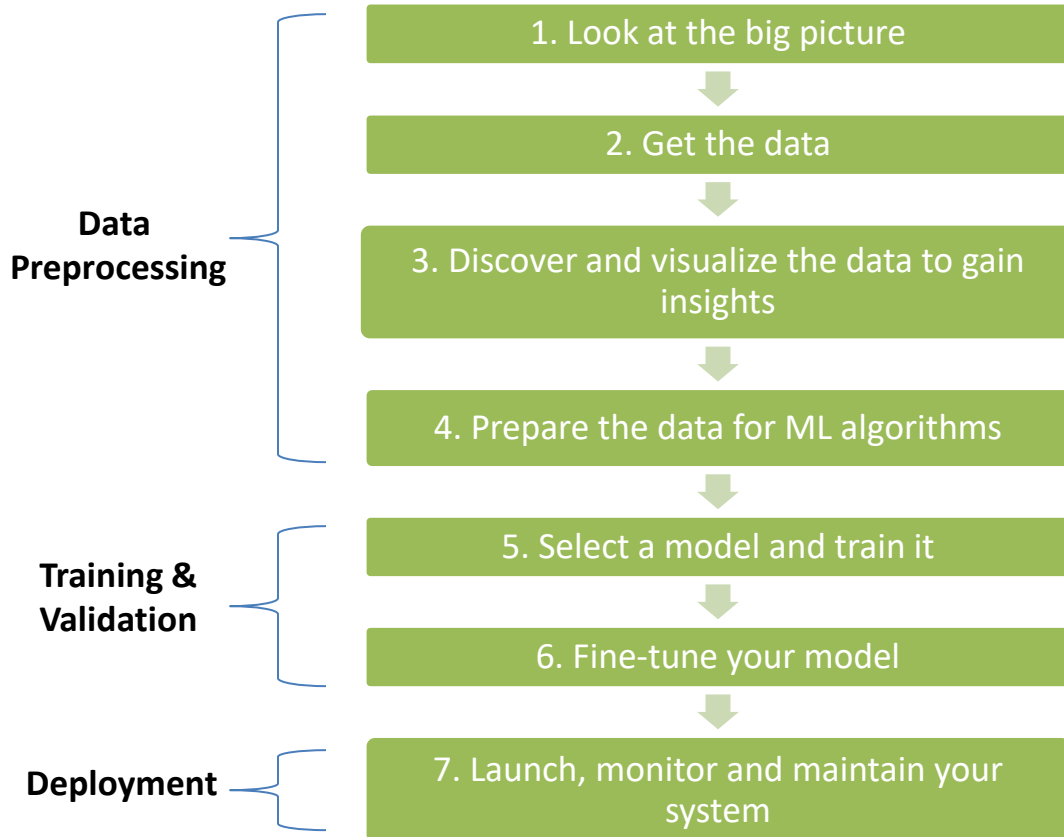


# Python Libraries & Tools

- To learn the machine learning pipeline, we will go through an end-to-end ML project for regression on real dataset using the following Python libraries and tools:
  - Numpy, Pandas, Matplotlib, Scikit-Learn
  - Jupyter Notebook
- Dataset:
  - California Housing Data
    - housing.csv

Will practice the regression pipeline in Lab 4 & 5

# Machine Learning Pipeline – Regression





# Machine Learning Pipeline – Regression

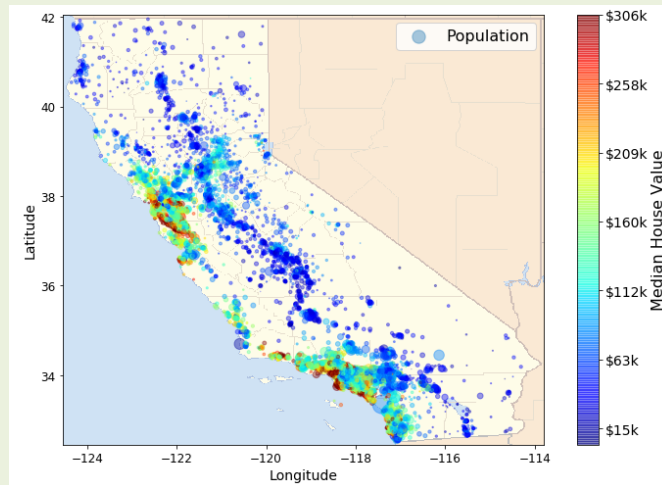
---

- 1. Look at the Big Picture**
2. Get Data
3. Explore Data
4. Prepare Data
5. Select & train model
6. Fine-tune the model
7. Launch, monitor & maintain

# Dataset in this lecture



## California Housing Data:



- Based on data from the 1990 California census from *StatLib* repository
- Contain metrics such as the *population*, *median income*, *location (longitude, latitude)*, and *median housing price* for each district
- Each district has a population of 600 to 3000 people
- **Task:** to predict the *median house price* based on properties of a district



# Phase 1: Look at the big picture

## Look at the big picture

Get data

Explore data

Prepare data

Select and train model

Fine-tune model

Launch, monitor & maintain

### ■ Understand the problem:

- *What is the task?*

Build a model to predict the median housing prices in California, given all other metrics

- *What is the current solution?*

District house prices are currently estimated manually by experts, time-consuming and often estimates were off by more than 20% (a reference performance)

### ■ Frame the problem:

- *Supervised, unsupervised, or reinforcement learning?*

**supervised**

- *Regression or classification task?*

**regression**



# Machine Learning Pipeline – Regression

---

1. Look at the Big Picture
- 2. Get Data**
3. Explore Data
4. Prepare Data
5. Select & train model
6. Fine-tune the model
7. Launch, monitor & maintain



Look at the  
big picture

**Get  
data**

Explore  
data

Prepare  
data

Select and  
train  
model

Fine-tune  
model

Launch,  
monitor &  
maintain

- First, let's load the data using **Pandas**
- The dataset is the **csv** file "housing.csv"

## import python libraries

```
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt
```

## Load data set

```
housing = pd.read_csv("housing.csv")
```



# Machine Learning Pipeline – Regression

---

1. Look at the Big Picture
2. Get Data
- 3. Explore Data**
4. Prepare Data
5. Select & train model
6. Fine-tune the model
7. Launch, monitor & maintain

# Phase 3: Explore data

Look at the  
big picture

Get  
data

**Explore  
data**

Prepare  
data

Select and  
train  
model

Fine-tune  
model

Launch,  
monitor &  
maintain

- Get insight of the data
- Identify the necessary data preparation steps
- May sample a smaller subset for exploration if training set is too large
- Activities in this phase include:
  - Take a look at the data structure
  - Check for data quality issues
  - Check the statistics of the data
  - Others

# Take a look at the data structure

- Get a first look to our data and identify the required pre-processing steps and ML algorithm

```
housing.head()
```



	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_proximity
0	-122.23	37.88	41.0	880.0	129.0	322.0	126.0	8.3252	452600.0	NEAR BAY
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.0	8.3014	358500.0	NEAR BAY
2	-122.24	37.85	52.0	1467.0	190.0	496.0	177.0	7.2574	352100.0	NEAR BAY
3	-122.25	37.85	52.0	1274.0	235.0	558.0	219.0	5.6431	341300.0	NEAR BAY
4	-122.25	37.85	52.0	1627.0	280.0	565.0	259.0	3.8462	342200.0	NEAR BAY

## Observations:



- There are 10 attributes (9 inputs + 1 output)
- Observe that `total_rooms` and `total_bedrooms` are in terms of the whole district, not for a household.
- Some values have been scaled or preprocessed (e.g., median income)

***DataFrame.head()***: Look at the top 5 rows of DataFrame



```
housing.info()
```

```
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 20640 entries, 0 to 20639  
Data columns (total 10 columns):  
longitude          20640 non-null float64  
latitude           20640 non-null float64  
housing_median_age  20640 non-null float64  
total_rooms        20640 non-null float64  
total_bedrooms     20433 non-null float64  
population         20640 non-null float64  
households         20640 non-null float64  
median_income      20640 non-null float64  
median_house_value 20640 non-null float64  
ocean_proximity    20640 non-null object  
dtypes: float64(9), object(1)  
memory usage: 1.6+ MB  
None
```

### Observations:

- There are 20,640 samples (fairly small dataset by ML standards)
- total\_bedrooms has 207 missing value (20640-20433)
- All attributes are numerical except for ocean\_proximity

### TODO:

- Need to tackle missing value in total\_bedrooms

**DataFrame.info()** gives a quick description of the data: #rows, all attributes, types of attributes, number of non-null values, memory usage, etc.



## ■ Getting statistics of numerical data

```
housing.describe()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	20640.000000	20640.000000	20640.000000	20640.000000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	1425.476744	499.539680	3.870671	206855.816909
std	2.003532	2.135952	12.585558	2181.615252	421.385070	1132.462122	382.329753	1.899822	115395.615874
min	-124.350000	32.540000	1.000000	2.000000	1.000000	3.000000	1.000000	0.499900	14999.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	787.000000	280.000000	2.563400	119600.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	1166.000000	409.000000	3.534800	179700.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	1725.000000	605.000000	4.743250	264725.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	35682.000000	6082.000000	15.000100	500001.000000

- Mean values shows the scale of an attribute
- Standard deviation (**std**) shows how dispersed the values are within the attribute
- (**min**, **max**) indicate the value range

### Observations:

- The attributes have very different scale

### TODO:

- Perform feature scaling

**DataFrame.describe()** shows a summary of the numerical attributes



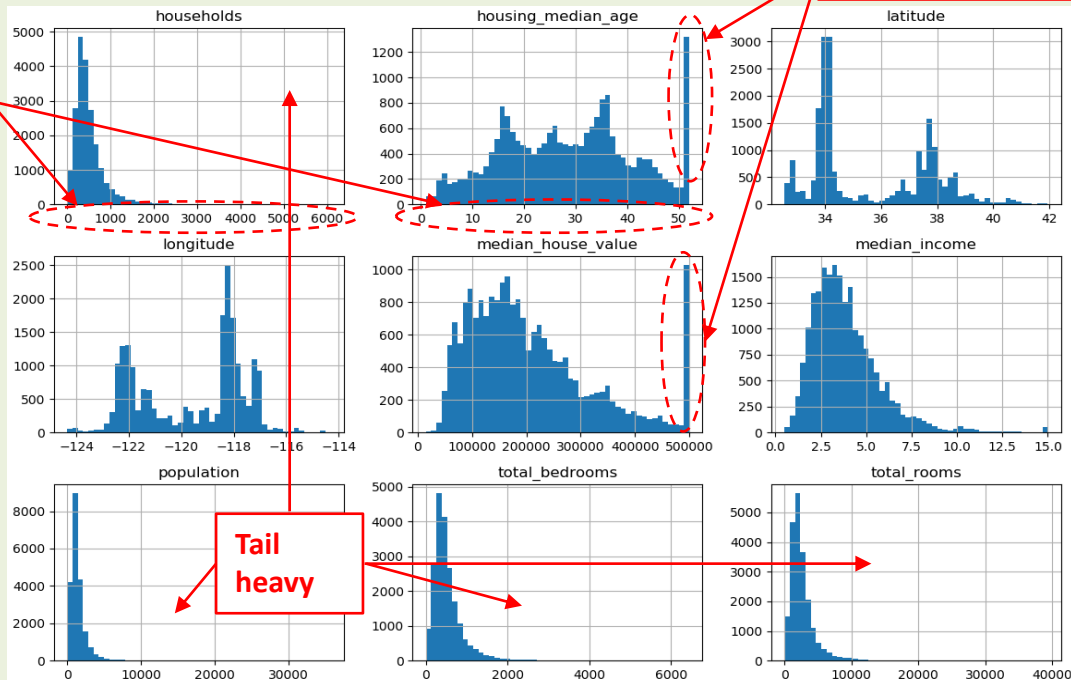
## Visualizing numerical data



```
import matplotlib.pyplot as plt
housing.hist(bins=50, figsize=(20,15))
plt.show()
```

Attributes  
have very  
different  
scales

Values have been capped



`DataFrame.hist()` plots the histogram for each numerical attribute



## ■ Analyzing **categorical** data

```
housing["ocean_proximity"].value_counts()
```

```
<1H OCEAN    9136  
INLAND       6551  
NEAR OCEAN   2658  
NEAR BAY     2290  
ISLAND        5  
Name: ocean_proximity, dtype: int64
```

Too few samples

**`Series.value_counts()`** can be used to describe categorical data:

1. what categories exist for a particular attribute
2. how many samples (districts) belong to each category

Observation	Is this an issue?	TODO
<b>median_income</b> attribute not in US\$	Not an issue. Working with preprocessed attributes are common in ML.	Confirm with the team if the value has been scaled.
<b>housing_median_age</b> and <b>median_house_value</b> are capped	<b>median_house_value</b> is the targeted variable – this may be a problem	Check with the team if this is an issue. If a proper value is required, then: <ol style="list-style-type: none"> <li>1. Collect the proper values for the districts whose values were capped</li> <li>2. Remove districts whose values are capped</li> </ol>
Attributes have different scales	Difficult for optimizer to find the best model	Perform feature scaling
Many histograms are tail heavy	Harder for some ML algorithms to detect pattern	Transform these attributes to have more bell-shaped distributions
Too few samples for <b>ISLAND</b> category	Low prediction accuracy for this category	Collect more samples if possible



# Machine Learning Pipeline – Regression

---

1. Look at the Big Picture
2. Get Data
3. Explore Data
- 4. Prepare Data**
5. Select & train model
6. Fine-tune the model
7. Launch, monitor & maintain

# Phase 4: Prepare data

Look at the  
big picture

Get  
data

Explore  
data

**Prepare  
data**

Select and  
train  
model

Fine-tune  
model

Launch,  
monitor &  
maintain

- Data preparation makes the data more suitable for machine learning algorithms.
- Most of the effort involved to build your Machine Learning system likely will happen here
- Major tasks in data preparation:
  - Separate output vector from input features
  - Split the dataset into training and testing set
  - Fix data errors (if any)
  - Handling missing data
  - Perform data transformation
  - Data reduction (if necessary)

# Split Input matrix (**X**) and output vector (**y**)

- Before we preprocess the dataset, we want to separate the output vector (**y**) from the input matrix (**X**)
- For the *housing* dataset, **y** is the 'median\_house\_value' column.

```
X = housing.drop('median_house_value', axis = 1)
y = housing['median_house_value']
```

1=column  
0=row (default)

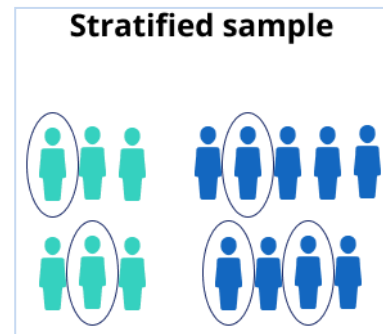
- To show the types and shapes of **X** and **y**

```
print ('Shape of original dataset, train:', housing.shape)
print ('X: shape=', X.shape, 'type=', type(X))
print ('y: shape=', y.shape, 'type=', type(y))
```

```
Shape of original dataset, train: (20640, 10)
X: shape= (20640, 9) type= <class 'pandas.core.frame.DataFrame'>
y: shape= (20640,) type= <class 'pandas.core.series.Series'>
```

# Split Training Set and Test Set

- Set aside some data for **test set**
- Test set evaluates how well our model generalize to novel (unseen) data
- Never fine-tune your model using the test set
- Two types of sampling:
  - **Random sampling**: randomly sample from the whole dataset
  - **Stratified sampling**: randomly sample but maintain the proportion of classes in each split




# Random Sampling

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=30)

print ('full set shape=', X.shape)
print ('train shape=', X_train.shape)
print ('test shape=', X_test.shape)
```

```
full set shape= (20640, 9)
train shape= (16512, 9)
test shape= (4128, 9)
```



random\_state is optional. Used to ensure that we get the same result each time we run the code

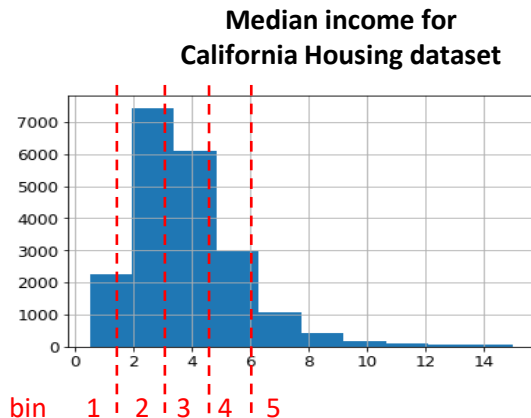
**train\_test\_split()** split the dataset into train & test sets

Note: we will use random sampling splits in the following steps.



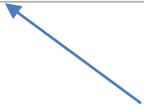
# Stratified Sampling

- Suppose the expert said that **median\_income** is a very important attribute to predict median housing price. Since **median\_income** is tail heavy, we will perform stratified sampling to ensure that we have representative samples in each income range in the split.
- Stratified sampling:
  - Partition **median\_income** into 5 bins, samples with median\_incomes  $> 6$  is grouped to bin 5 to compensate the tail heavy distribution.
  - Sampling while maintaining the distribution of the bins.



# Stratified Sampling

```
housing['income_cat'] = np.ceil(housing['median_income'] / 1.5)  
housing['income_cat'].where(housing['income_cat'] < 5, 5.0, inplace = True)
```



median_income	housing['income_cat']
0 to 1.5	1
1.5 to 3	2
3 to 4.5	3
4.5 to 6	4
6 and above	5

```
from sklearn.model_selection import StratifiedShuffleSplit  
  
splitter = StratifiedShuffleSplit(n_splits = 1, test_size = 0.2, random_state = 42)  
for train_index, test_index in splitter.split(housing, housing['income_cat']):  
    train = housing.loc[train_index]  
    test = housing.loc[test_index]  
print('full set shape:', housing.shape)  
print('train shape:', train.shape)  
print('test shape:', test.shape)
```

**StratifiedShuffleSplit()** split the dataset into two sets using stratified sampling

# Split Numerical and Categorical data

- The required data preprocessing differs for **numerical** and **categorical** features.
- Therefore, it is needed to split the **X** into two sets, one with numerical features only and the other with categorical features only. For the *housing* dataset,
  - Numerical attributes (longitude, latitude, housing\_median\_age, total\_rooms, total\_bedrooms, population, households, median\_income).
  - Categorical attributes (namely ocean\_proximity)

```
X_train_num = X_train.drop('ocean_proximity', axis = 1)
X_train_cat = X_train['ocean_proximity']
```

```
print('X_train_num shape:', X_train_num.shape)
print('X_train_cat shape:', X_train_cat.shape)
```

```
X_train_num shape: (16512, 8)
X_train_cat shape: (16512,)
```

# Handling Missing Values

- Data is not always available
- Missing data may be due to:
  - Equipment malfunction
  - Data is not considered/available at the time of entry
  - Inconsistent with other recorded data and thus deleted
- How to deal with missing data :
  - Remove the record with missing values
  - Fill in the missing values with
    - a global constant (ex. 0)
    - mean or median or mode of each attribute
    - mean or median or mode of each attribute belonging to the same class
    - the value of other similar samples



## Determine missing attributes

- From previous step, the **total\_bedrooms** has missing value
- The code below shows the number of missing values in each column:

```
print('Number of missing values for all columns:')  
X_train_num.isnull().sum()
```

```
Number of missing values for all columns:
```

```
longitude           0  
latitude            0  
housing_median_age  0  
total_rooms         0  
total_bedrooms      162  
population          0  
households          0  
median_income       0  
dtype: int64
```



# Handling missing values

- Most ML algorithms cannot work with missing features
- What can we do?
  1. Remove the samples with missing value (remove row)

```
X_train_num_tr = X_train_num.dropna (subset = ['total_bedrooms'])
```

2. Get rid of the feature (remove column)

```
X_train_num_tr = X_train_num.dropna(axis=1)
```

3. Set the values to some constant value (e.g. zero)

```
X_train_num_tr = X_train_num.fillna(0)
```

**DataFrame.dropna()** is used to drop any rows or columns with missing value

**DataFrame.fillna(value)** or **Series.fillna(value)** is used to replace the value of missing values with a predefined value

# Handling missing values - SimpleImputer



4. Set the values to the **median** or **mean** value. The following code shows how to use *scikit-learn* library to do this:

1. Create an imputer object  
with median strategy

```
from sklearn.impute import SimpleImputer  
  
imputer = SimpleImputer(strategy = 'median')  
imputer.fit(X_train_num)  
X_train_num_tr = imputer.transform(X_train_num)
```

2. Compute (fit) the median for  
each attribute and store it

3. Replace (transform) missing  
values with the median of the  
attribute

The same imputer will be used to handle missing values in the Test set later.

**Imputer** is a **transformer** provided by Scikit-Learn to handle missing values.

# Scikit-Learn Design

In Scikit-Learn, all ML objects share a consistent and simple interface.

- **Estimators**: Any object that can estimate some parameters based on a dataset. All estimator maintains
  - `.fit(X, y)` estimates (train) some parameters based on X (i.e., input feature) and optionally y (i.e., output)
- **Transformers** is a kind of *estimator* that can transform a dataset. It additionally has the following functions:
  - `.transform(X)` transforms the dataset X using the parameters that was learnt using `fit()`.
  - `.fit_transform(X)` first performs `.fit(X)` followed by `.transform(X)`
- **Predictors** is a kind of *estimator* that are capable of making predictions given a dataset. It additionally has the following functions:
  - `.predict(X)` predicts the label  $h$  of each samples in X
  - `.predict_proba(X)` predicts the probability of all possible classes for a given record (classification only)



# Scikit-Learn Design

- Although Scikit-Learn library accepts both numpy arrays and pandas dataFrames as input, they typically returns a **numpy array** as output.

```
from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy = 'median')
imputer.fit(X_train_num)
X_train_num_tr = imputer.transform(X_train_num)
```

Numpy array

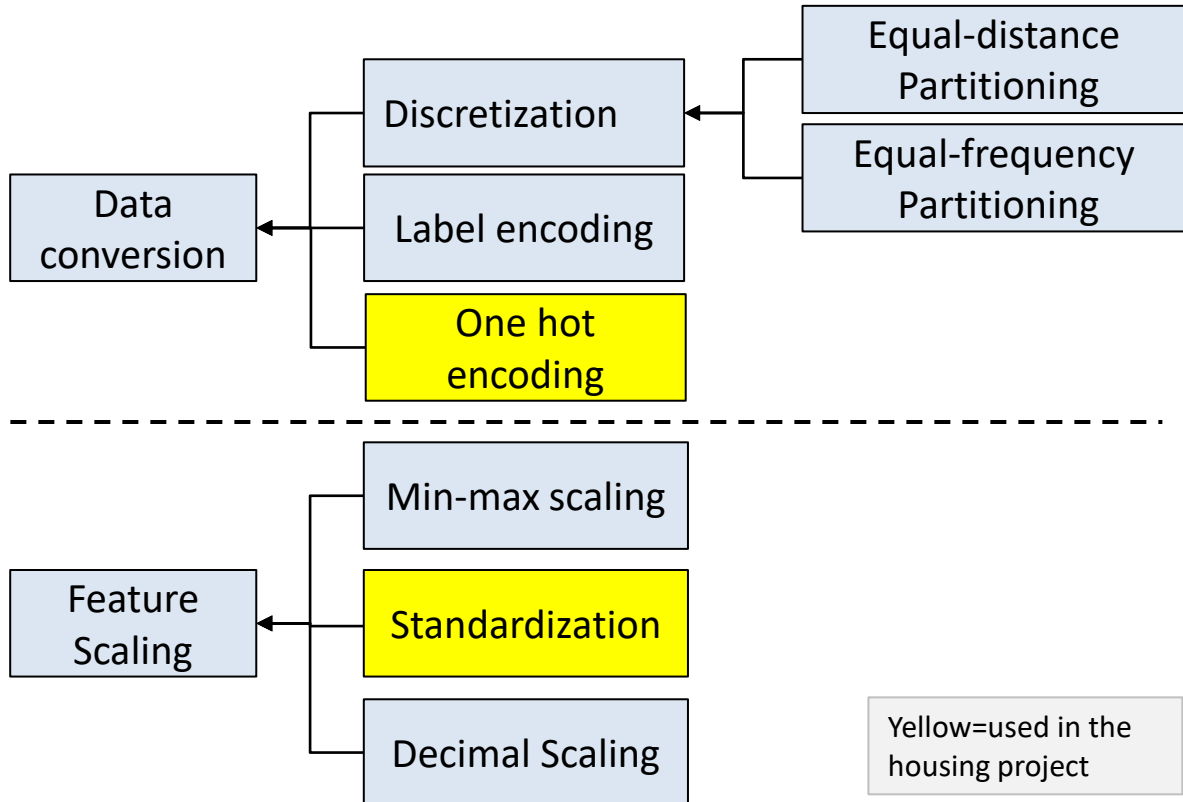
DataFrame

```
print('Type of X_train_num:', type(X_train_num))
print('Type of X_train_num_tr:', type(X_train_num_tr))
```

```
Type of X_train_num: <class 'pandas.core.frame.DataFrame'>
Type of X_train_num_tr: <class 'numpy.ndarray'>
```

# Data Transformation

- Common data transformation :



# Processing Numerical Data – Feature scaling

---

- Feature scaling changes raw feature vectors into a representation that is more suitable for the downstream estimators
- Most machine algorithm may behaves badly if the features are not scaled properly
- Two common scaling strategies:
  - **Min-max scaling**
  - **Standardization**

# Min-max Scaling

- Min-max scaling linearly scales the features such that it lie between a given minimum and maximum value

$$x' = \left[ \frac{x - \min}{\max - \min} \right] \times (\text{newmax} - \text{newmin}) + \text{newmin}$$

- Note that if the new range is [0..1], then this simplifies to

$$x' = \left[ \frac{x - \min}{\max - \min} \right]$$

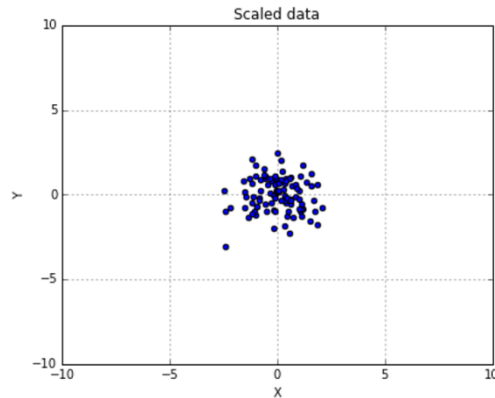
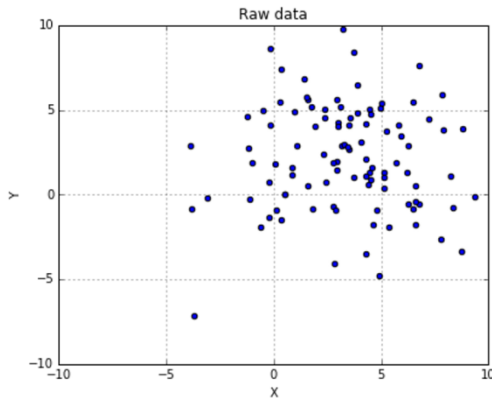
Example: Transform \$30000 between [10000..45000] into [0..1]

$$x' = \left[ \frac{30000 - 10000}{45000 - 10000} \right] = 0.5714$$

# Standardization

- Transforms the data such that they look like a standard normally distributed data: Gaussian with **zero mean** and **unit variance**

$$x' = \frac{(x - \text{mean})}{\text{stddev}}$$



# Standardization - Example

- Example: normalizing the “Humidity” attribute:

$$x' = \frac{(x - \text{mean})}{\text{stddev}}$$

Humidity
85
90
78
96
80
70
65
95
70
80
70
90
75
80



Mean = 80.3  
Stdev = 9.84



Humidity
0.48
0.99
-0.23
1.6
-0.03
-1.05
-1.55
1.5
-1.05
-0.03
-1.05
0.99
-0.54
-0.03

Mean = 0.0  
Stdev = 1.0

# Standardization



The following code shows how to perform standardization in scikit-learn

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaler.fit(X_train_num_tr)
X_train_num_tr = scaler.transform(X_train_num_tr)

print(X_train_num_tr.mean (axis=0))
print(X_train_num_tr.std (axis=0))
```

Compute the *mean* and *std* for each attribute and store it

Standardize all attributes using the computed *mean* and *std* value

```
[ -4.35310702e-15  2.28456358e-15 -4.70123509e-17  7.58706190e-17
  1.36061489e-16 -3.70074342e-17  2.07897868e-17 -2.10210832e-16]
[ 1.  1.  1.  1.  1.  1.  1.  1.]
```

**StandardScaler** is used to perform standardization on the dataset

# Processing Categorical Data – Encoding

- Why label encoding?

Most machine learning algorithm only accepts numerical input.

- **Label encoding** – convert categorical value to a number

- Label encoding is suitable for **ordinal data types**:

High → 2

Medium → 1

Low → 0

Very good → 5

Good → 4

Average → 3

Poor → 2

Very poor → 1

- Not suitable for **nominal data types** where order does not matter.

- { INLAND, NEAR OCEAN, NEAR BAY, ISLAND, <1H OCEAN }
- { *Science, Maths, History, Geography, English, Literature* }
- Doesn't make sense to say that *Science* is bigger or smaller than *Maths*



# One-hot encoding

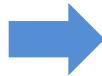
- **One-hot encoding** is used to encode nominal data
- **One-hot encoding:**
  - Create one binary attribute for each category
  - For example, 3 categories in  $\{Science, Math, Arts\}$ . So, create 3 new attributes, one for each category
    - $Science \rightarrow 0 \ 0 \ 1$
    - $Math \rightarrow 0 \ 1 \ 0$
    - $Arts \rightarrow 1 \ 0 \ 0$
  - For feature with only two categories (e.g., spam / not spam). Simply use convert to single binary attribute (1/0).

# One-hot encoding

- Example:

Original

Sample	Weather	Haze	Humidity
1	Sunny	High	85
2	Sunny	Low	95
3	Overcast	Medium	90
4	Rainy	High	78
5	Overcast	High	65
6	Rainy	Medium	73
7	Sunny	Low	66



One hot encoding

Sample	Weather			Haze	Humidity
	Sunny	Overcast	Rainy		
1	1	0	0	2	85
2	1	0	0	0	95
3	0	1	0	1	90
4	0	0	1	2	78
5	0	1	0	2	65
6	0	0	1	1	73
7	1	0	0	0	66

## One-hot encoding with LabelBinarizer



```
from sklearn.preprocessing import LabelBinarizer

lb_encoder = LabelBinarizer()
X_train_cat_tr = lb_encoder.fit_transform(X_train_cat)
X_train_cat_tr
```

```
array([[1, 0, 0, 0, 0],
       [1, 0, 0, 0, 0],
       [0, 0, 0, 0, 1],
       ...,
       [0, 1, 0, 0, 0],
       [1, 0, 0, 0, 0],
       [0, 0, 0, 1, 0]], dtype=int32)
```

**LabelBinarizer** converts **categorical attributes** directly into **one-hot vector**.

# Finalizing the Training set



## Finalizing the training set

- Now we are ready to combine the transformed categorical and numerical.
- Also, convert `y_train` from Pandas series to Numpy array for subsequence model training process.

```
X_train_tr = np.hstack([X_train_num_tr, X_train_cat_tr])
y_train = y_train.values; #convert to numpy array

print('X_train_tr shape:', X_train_tr.shape)
print('y_train shape:', y_train.shape)
```

```
X_train_tr shape: (16512, 13)
y_train shape: (16512,)
```

# Discretization

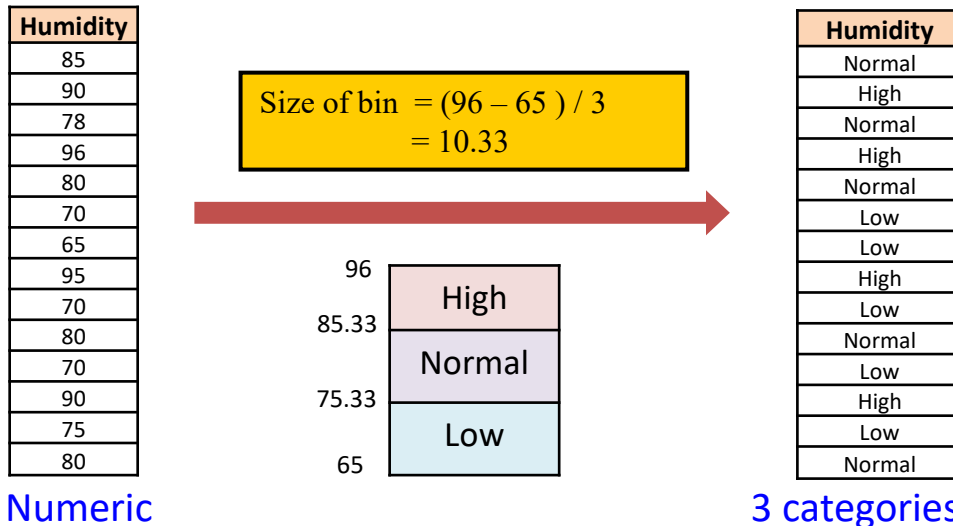
- Sometimes we need to convert a numeric attribute into a categorical one
- Why discretize?
  - Transform a *regression* into a *classification* problem
  - Change data distribution (e.g. skewed -> even distribution)
  - Assist interpretation (e.g. 2.3, 5.6, 8.7 -> low, median, high)
- Data discretization methods:
  - Binning
    - Equal distance partitioning
    - Equal frequency partitioning
  - Clustering analysis

We do not use  
discretization here.

# Equal-distance partitioning

- Divides the range into N intervals of equal size: uniform grid
- To get the intervals

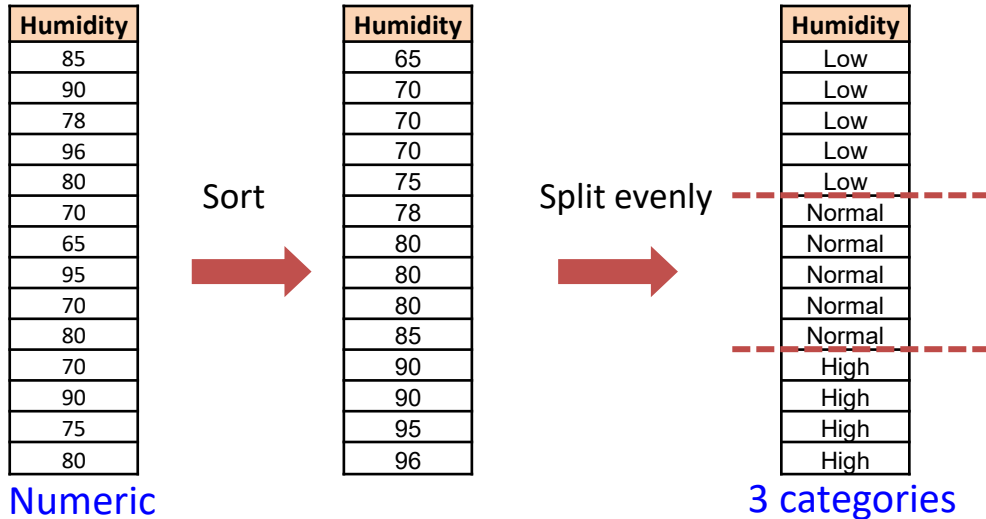
$$\text{size\_of\_bin} = (\text{max\_value} - \text{min\_value}) / \text{num\_categories}$$



- Does not handle skewed data very well

# Equal-frequency partitioning

- Divides the range into N intervals, each containing approximately same number of samples



- Good data scaling
- Less susceptible to noise compared to equal-distance partitioning



Next:

## **The Regression Pipeline – 2/2**