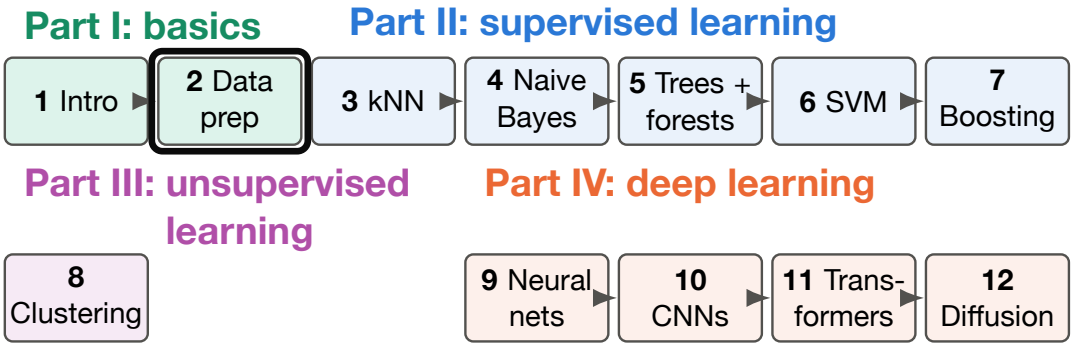
Data preprocessing

229352 Lecture 2: data preprocessing

Course map


Part I: basics
Part II: supervised learning
1 Intro
2 Data prep
3 kNN
4 Naive Bayes
5 Trees + forests
6 SVM
7 Boosting
Part III: unsupervised learning
Part IV: deep learning
8 Clustering
9 Neural nets
10 CNNs
11 Trans-formers
12 Diffusion

Tabular data: a game store

#	Genre	Price	Demo hours	Buy
1	Action	Low	3.0	Yes
2	Sports	High	1.5	No
3	Puzzle	Low	4.0	Yes
4	Action	High	2.0	No
5	Sports	High	3.5	Yes
6	?	Low	2.5	No
7	Puzzle	Low	3.0	Yes
8	Action	High	5.0	Yes
9	Sports	Low	?	No
10	Sports	Low	4.5	Yes

Tabular data: rows and columns.

The table lists ten games of a store, also used in Lecture 4.

Two cells are missing (“?”).

Many models cannot use a row with a missing value.

Made-up data (Lecture 4)

Imputation: fill in each missing value

- numeric: the mean or the median
- categorical: the most common value
- numeric or categorical: a regression on the other columns

#	Genre	Demo hours
1	Action	3.0
2	Sports	1.5
3	Puzzle	4.0
4	Action	2.0
5	Sports	3.5
6	?	2.5
7	Puzzle	3.0
8	Action	5.0
9	Sports	?
10	Sports	4.5

Median of demo hours

#	Demo hours
1	3.0
2	1.5
3	4.0
4	2.0
5	3.5
6	2.5
7	3.0
8	5.0
9	?
10	4.5

Median of the nine known demo hours =

The most common genre

#	Genre	Demo hours
1	Action	3.0
2	Sports	1.5
3	Puzzle	4.0
4	Action	2.0
5	Sports	3.5
6	?	2.5
7	Puzzle	3.0
8	Action	5.0
9	Sports	?
10	Sports	4.5

Known genres:

Action 3, Puzzle 2, Sports 4

Most common: Sports.

Fill in game 6's genre with Sports.

Imputation in scikit-learn

```
import pandas as pd
from sklearn.impute import SimpleImputer

# the 10 games; an empty cell is read as NaN
df = pd.read_csv("games.csv", index_col="game")
med = SimpleImputer(strategy="median")
df[["hours"]] = med.fit_transform(df[["hours"]])
mode = SimpleImputer(strategy="most_frequent")
df[["genre"]] = mode.fit_transform(df[["genre"]])
print(df.loc[6, "genre"], df.loc[9, "hours"])
# -> Sports 3.0
```

Ordinal features

Categories with a natural order:

Customer satisfaction	Education level	T-shirt size
Very unsatisfied	High school	Small
Unsatisfied	Bachelor’s degree	Medium
Neutral	Master’s degree	Large
Satisfied	Doctoral degree	Extra large
Very satisfied		

Nominal and ordinal features

#	Genre	Price	Reviews
1	Action	Low	Good
2	Sports	High	Mixed
3	Puzzle	Low	Good
4	Action	High	Good
5	Sports	High	Good
6	Sports	Low	Mixed
7	Puzzle	Low	Mixed
8	Action	High	Good
9	Sports	Low	Mixed
10	Sports	Low	Good

shaded: filled in by imputation

Genre: **nominal**, no order.

Price: **ordinal**, Low $<$ High.

Reviews: **ordinal**,
Mixed $<$ Good.

Most models need numbers, so we encode the categories.

Ordinal encoding

#	Price	Code	Reviews	Code
1	Low	0	Good	1
2	High	1	Mixed	0
3	Low	0	Good	1
4	High	1	Good	1
5	High	1	Good	1
6	Low	0	Mixed	0
7	Low	0	Mixed	0
8	High	1	Good	1
9	Low	0	Mixed	0
10	Low	0	Good	1

Levels numbered in order:

Low $\rightarrow$ 0, High $\rightarrow$ 1

Mixed $\rightarrow$ 0, Good $\rightarrow$ 1

A model treats the gaps between codes as distances, but the true gaps between levels are unknown.

Ordinal encoding in scikit-learn

```
from sklearn.preprocessing import OrdinalEncoder

# df: the games after imputation
cols = ["price", "reviews"]
order = [["Low", "High"], ["Mixed", "Good"]]
enc = OrdinalEncoder(categories=order)
codes = enc.fit_transform(df[cols])
print(codes[:3].tolist())  # games 1-3
# -> [[0.0, 1.0], [1.0, 0.0], [0.0, 1.0]]
# no categories given: alphabetical order
default = OrdinalEncoder().fit(df[cols])
print([c.tolist() for c in default.categories_])
# -> [['High', 'Low'], ['Good', 'Mixed']]
```

The default order is alphabetical, so High comes before Low.

One-hot encoding

#	Genre	Action	Puzzle	Sports
1	Action	1	0	0
2	Sports	0	0	1
3	Puzzle	0	1	0
4	Action	1	0	0
5	Sports	0	0	1
6	Sports	0	0	1
7	Puzzle	0	1	0
8	Action	1	0	0
9	Sports	0	0	1
10	Sports	0	0	1

shaded: filled in by imputation

Genre has no order; codes 0, 1, 2 would impose one.

One 0/1 column per genre.

Each row has one 1.

Game 6 was in fact Action: imputation can be wrong.

One-hot encoding in scikit-learn

```
from sklearn.preprocessing import OneHotEncoder

# sparse_output=False: a dense array
enc = OneHotEncoder(sparse_output=False)
onehot = enc.fit_transform(df[["genre"]])
print(enc.get_feature_names_out().tolist())
# -> ['genre_Action', 'genre_Puzzle', 'genre_Sports']
print(onehot[:3].astype(int).tolist())  # games 1-3
# -> [[1, 0, 0], [0, 0, 1], [0, 1, 0]]
```

Summary: tabular data

- Fill missing numbers with the mean or the median.
- Fill missing categories with the most common value.
- Ordinal features: integer codes that follow the order of the levels.
- Nominal features: one-hot, one 0/1 column per value.
- scikit-learn: `SimpleImputer`, `OrdinalEncoder`, `OneHotEncoder`.

Checkpoint 1

Which encoder for each new column?

platform: PC, PlayStation, Switch

age rating: 3+, 7+, 12+, 18+

Checkpoint 1

Which encoder for each new column?

platform: PC, PlayStation, Switch

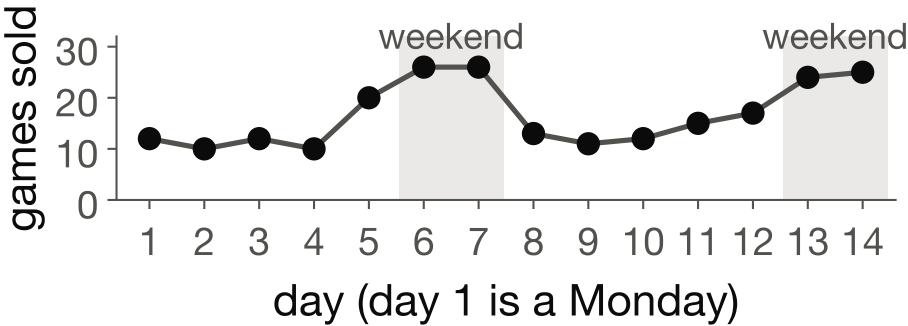
age rating: 3+, 7+, 12+, 18+

platform: one-hot, because there is no order

age rating: ordinal, with the order 3+, 7+, 12+, 18+

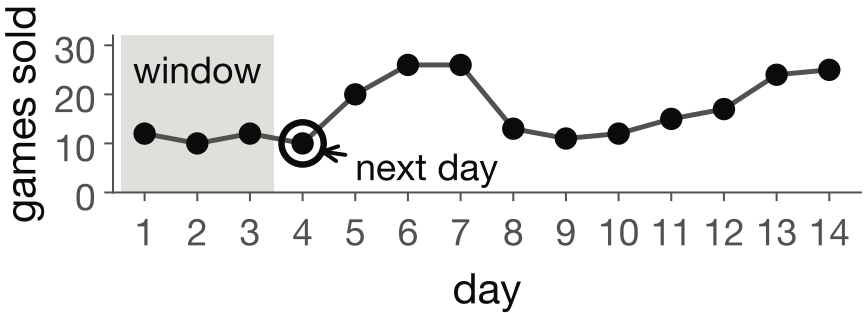
(or numeric, since the ratings are minimum ages;
alphabetical order would give 12+, 18+, 3+, 7+)

Time series data: simulated daily sales



On weekends the store sells about 25 games a day,
on weekdays about 13.

The sliding window



A model needs rows, each with inputs and a target.

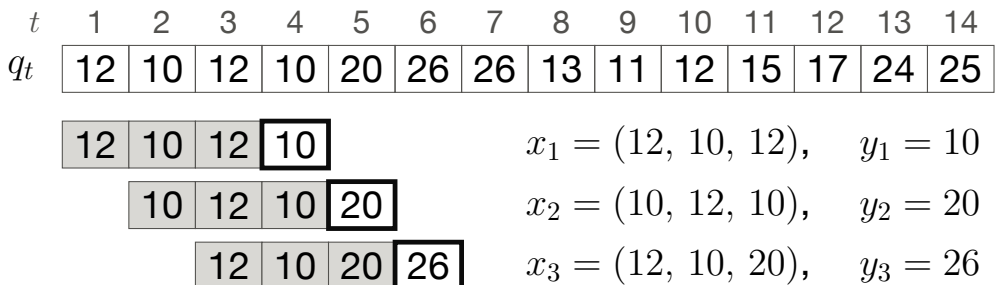
Past days help predict the next day:

the inputs are the last three days,
and the target is the next day.

Rows from the sliding window

q_t: games sold on day t. Row i: $x_i = (q_i, q_{i+1}, q_{i+2})$, $y_i = q_{i+3}$.

x_{ij}: feature j of row i. The window moves one day at a time.



Sliding window: write the rows

t	1	2	3	4	5	6	7	8	9	10	11	12	13	14
q_t	12	10	12	10	20	26	26	13	11	12	15	17	24	25

Windows of three days: write every row x_i, y_i.

The sliding window in Python

```
import numpy as np

sales = np.array([12, 10, 12, 10, 20, 26, 26,  # 14 days
                  13, 11, 12, 15, 17, 24, 25])

def make_windows(q, days):
    X, y = [], []
    for t in range(len(q) - days):
        X.append(q[t:t + days])  # the window
        y.append(q[t + days])    # the next day: the target
    return np.array(X), np.array(y)

X, y = make_windows(sales, 3)
print(X.shape, y.shape, X[0].tolist(), y[0])
# -> (11, 3) (11,) [12, 10, 12] 10
```

Multivariate sliding window

q_t	12	10	12	10	20	26	26	13	11	12	15	17	24	25
v_t	113	98	124	119	155	205	244	110	103	122	114	162	241	214

v_t: visitors on day t. Write the rows

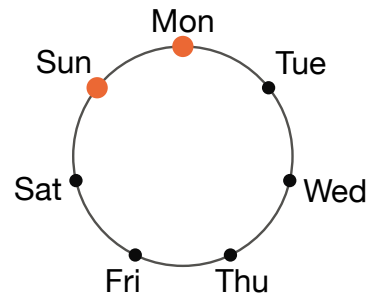
$x_i = (q_i, \ldots, q_{i+2},\ v_i, \ldots, v_{i+2}),\ \ y_i = q_{i+3}$:

Cyclical encoding

u: the day of the week, 0 (Mon) to 6 (Sun).

day	Mon	Tue	Wed	Thu	Fri	Sat	Sun
u	0	1	2	3	4	5	6

$$u \longmapsto \left(\sin \frac{2\pi u}{7}, \cos \frac{2\pi u}{7} \right)$$



Sun and Mon differ by 6 but are neighbors on the circle.

Electric load forecasting

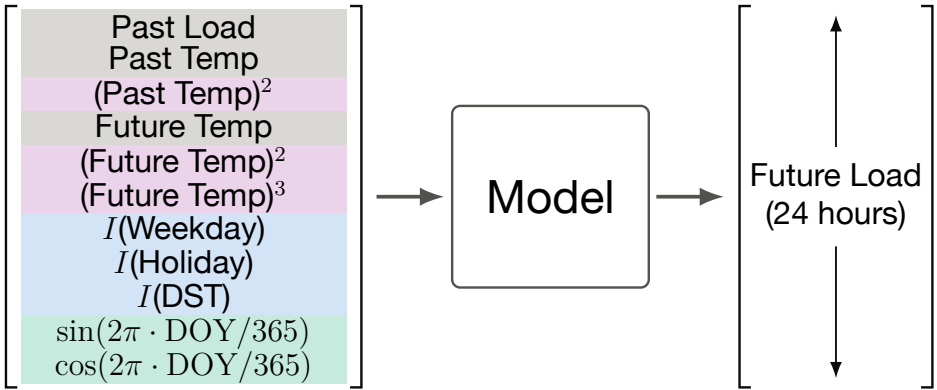
Data: the electric load and the temperature of each hour, 2008 to 2016, from the US grid operator PJM.

Task: predict the load of each of the 24 hours of the next day.

date	hour	load	temperature (°F)
2008-01-01	1:00	1.477	35.4
2008-01-01	2:00	1.414	37.3
2008-01-01	3:00	1.364	39.0
2008-01-01	4:00	1.335	39.0
⋮	⋮	⋮	⋮

Data: PJM, via P. L. Donti, B. Amos, J. Z. Kolter, NeurIPS 2017 code (Apache-2.0)

Encodings in load forecasting



After P. L. Donti, B. Amos, J. Z. Kolter, NeurIPS 2017, Fig. 3

Summary: time series

- Sliding window: the inputs are the last few days, and the target is the next day.
- Move the window one day at a time: one row per window.
- With several series, one row contains all their windows.
- Cyclical encoding of the day u of the week: $u \longmapsto \big(\sin(2\pi u/7),\ \cos(2\pi u/7)\big)$.
- Load forecasting: the inputs combine raw values, powers, indicators $I(\cdot)$ and cyclical features.

Text data: three reviews

1 Great game, great fun.

2 Boring game.

3 Excellent game, fun music.

A bag of words for each review



review 1



review 2



review 3

Text data: bag of words

Three reviews:

1 Great game, great fun.

2 Boring game.

3 Excellent game, fun music.

Count each word in each review; ignore case, punctuation and word order.

	review		
word	**1**	**2**	**3**
boring	0	1	0
excellent	0	0	1
fun	1	0	1
game	1	1	1
great	2	0	0
music	0	0	1

In the data, review i is row x_i and each word is a column.

Bag of words in scikit-learn

```
from sklearn.feature_extraction.text import CountVectorizer

reviews = ["Great game, great fun.", "Boring game.",
           "Excellent game, fun music."]
vec = CountVectorizer()
X = vec.fit_transform(reviews)  # one row per review
print(*vec.get_feature_names_out())  # the vocabulary
# -> boring excellent fun game great music
print(X.toarray())
# -> [[0 0 1 1 2 0]
#     [1 0 0 1 0 0]
#     [0 1 1 1 0 1]]
```

TF-IDF

TF-IDF: term frequency times inverse document frequency.

$$\mathrm{tfidf}_{w,D} = \mathrm{tf}_{w,D} \times \mathrm{idf}_w, \qquad \mathrm{idf}_w = \ln \frac{1+N}{1+\mathrm{df}_w} + 1$$

w: a word; D: a document; N: the number of documents

$\mathrm{tf}_{w,D}$: the count of w in D

df_w: the number of documents that contain w

$\ln$: the natural logarithm

TF-IDF: an example

$N = 2$ documents. Fill in idf and tf$\times$idf.

Word	tf		idf	tf$\times$idf	
	Doc 1	Doc 2		Doc 1	Doc 2
see	1	0			
cat	1	1			
and	2	1			
dog	1	1			
bird	0	1			

TF-IDF in scikit-learn

```
from sklearn.feature_extraction.text import TfidfVectorizer

# reviews: the three reviews of the bag-of-words slide
vec = TfidfVectorizer()
X = vec.fit_transform(reviews)
print(*vec.get_feature_names_out())
# -> boring excellent fun game great music
print(X.toarray()[0].round(3))  # review 1
# -> [0.    0.    0.343 0.266 0.901 0.   ]
```

These values differ from the formula;
the next slide explains why.

TF-IDF of review 1

Review 1: "Great game, great fun."
($N = 3$ documents: the three reviews)

word	tf	tf$\times$idf	divided by length
fun	1	1.288	0.343
game	1	1.000	0.266
great	2	3.386	0.901

Length of the row before dividing: 3.758.

Summary: bag of words and TF-IDF

- Bag of words: a count for each word of the vocabulary.
- TF-IDF: $\mathrm{tf} \times \mathrm{idf}$; rare words get more weight.
- Bag of words and TF-IDF both ignore word order.

Image data: pixels

0	0	9	14	8	1	0	0
0	0	12	14	14	12	0	0
0	0	9	10	0	15	4	0
0	0	3	16	12	14	2	0
0	0	4	16	16	2	0	0
0	3	16	8	10	13	2	0
0	1	15	1	3	16	8	0
0	0	11	16	15	11	1	0

A grayscale image is a grid of numbers.

The digit has 8×8 pixels with values 0 (white) to 16 (black).

The feature vector lists the 64 numbers row by row.

Digit: scikit-learn `load_digits` (UCI, E. Alpaydin and C. Kaynak), CC BY 4.0

Color images: RGB



Each pixel has red, green and blue values from 0 to 255.

Channels drawn in gray: 0 black, 255 white.

This photo has 400 rows and 600 columns of pixels.

Photo: scikit-image `coffee`, Rachel Michetti, CC0

The values of a 4×4 patch

R: red				G: green				B: blue			
226	224	225	221	137	136	137	132	49	51	54	50
227	228	227	219	136	140	141	130	50	55	58	47
228	228	227	222	139	140	141	132	51	55	58	52
229	227	227	225	140	139	141	135	54	55	58	53

the 16 pixels at the center of the white box

An orange area has much red, some green and little blue.

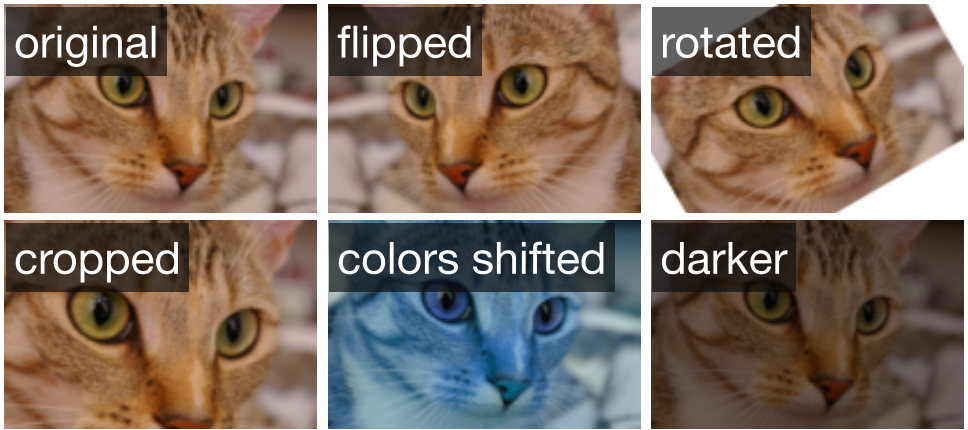
Importing images

```
import numpy as np
from PIL import Image

im = Image.open("coffee.png")  # a photo saved as PNG
x = np.array(im)
print(x.shape, x.dtype)  # height, width, colors
# -> (400, 600, 3) uint8
print(x[0, 0].tolist())  # the top-left pixel: R, G, B
# -> [21, 13, 8]
```

An RGB photo is an array of shape (height, width, 3).

Image augmentation



One image gives many training images, all labeled cat

Photo: scikit-image `chelsea`, Stefan van der Walt, CC0

Summary: images

- Grayscale image: a grid of gray levels.
- Color image: three grids, R, G and B.
- As features: all values in one vector (64 for a digit).
- Augmentation: flipped, rotated, cropped, recolored or darkened copies.

Standardize the features

x_{ij}: feature j of row i. mean_j, sd_j: the mean and the standard deviation (sd) of column j. For each column j:

$$x_{ij} \longmapsto \frac{x_{ij} - \text{mean}_j}{\text{sd}_j}$$

Then every column has mean 0 and sd 1.

Why standardize?

- k-nearest neighbors (Lecture 3) and k-means (Lecture 8) compare rows by their distance.
- A feature with a large scale then dominates the distance.
- Gradient descent (Lecture 7) converges faster when the features have similar scales.

Features in different units

#	Demo hours	Price (baht)
1	3.0	390
2	1.5	1,590
3	4.0	290
4	2.0	1,290
5	3.5	1,890
6	2.5	590
7	3.0	690
8	5.0	1,490
10	4.5	420

Nine games have known demo hours.

Prices are in baht.

Distance between two games:

$$\sqrt{(\Delta \text{hours})^2 + (\Delta \text{baht})^2}$$

Δ: the difference between them

Prices: made-up data

Distances in different units

#	Demo hours	Price (baht)
1	3.0	390
2	1.5	1,590
3	4.0	290
4	2.0	1,290
5	3.5	1,890
6	2.5	590
7	3.0	690
8	5.0	1,490
10	4.5	420

From game 1 to the shaded games:

game 10: 30.0

game 7: 300.0

Nearest: game 10, though its demo hours differ by 1.5.

Price dominates the distance.

After standardizing

#	standardized Demo hours	Price
1	−0.21	−1.00
2	−1.59	1.10
3	0.72	−1.17
4	−1.13	0.58
5	0.26	1.63
6	−0.67	−0.65
7	−0.21	−0.47
8	1.64	0.93
10	1.18	−0.95

Hours: mean 3.22, sd 1.08

Baht: mean 960, sd 570

From game 1 to the shaded games:

game 10: 1.39; game 7: 0.53

Nearest: game 7.

Scaling images and text

- p: a pixel value. Photos: $p \mapsto p/255$; digits: $p \mapsto p/16$. Gradient descent works well with inputs in $[0, 1]$.
- Text vectors: divide each row by its length, so long and short documents become comparable:

$$x_i \longmapsto \frac{x_i}{\|x_i\|_2}, \qquad \|x_i\|_2 = \sqrt{x_{i1}^2 + \cdots + x_{id}^2}$$

Summary: scaling

- Standardize each column: $(x_{ij} - \text{mean}_j)/\text{sd}_j$.
- Standardize when the features have different units and the method uses distances or gradient descent.
- Photos: divide by 255; text rows: divide by the length.

Training and test sets

- Training set: the rows used for fitting.
- Test set: rows kept aside, as new data for scoring.
- Split at random,
  but for time series the test set is the last days.

Data leakage

1. Split the data first.
2. Fit the standardizer (`StandardScaler`) on the training set only.
3. Transform the test set with the training mean_j and sd_j.

If test rows are used in the fit, the test score no longer measures performance on new data: **data leakage**.

Wrong: scale, then split

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

rng = np.random.default_rng(0)
hours = rng.normal(3, 1, size=(20, 1))  # 20 games
scaler = StandardScaler()
hours_std = scaler.fit_transform(hours)  # all 20 rows
train, test = train_test_split(hours_std, test_size=0.25,
                               random_state=0)
print(scaler.mean_.round(2))  # the mean of all 20 rows
# -> [2.82]
```

The standardizer was fitted on all 20 rows. So the test rows influence the scaling, and the test score can be optimistic.

Correct: split, then scale

```
# hours: the 20 simulated games of the previous slide
train, test = train_test_split(hours, test_size=0.25,
                               random_state=0)
scaler = StandardScaler()
train_std = scaler.fit_transform(train)  # 15 rows
test_std = scaler.transform(test)  # training mean, sd
print(scaler.mean_.round(2))  # the 15 training rows
# -> [2.76]
print(test.mean().round(2))  # the 5 test rows
# -> 3.0
```

The standardizer uses the training mean 2.76, not 2.82.

Summary: data leakage

- Split first, then fit each step on the training set.
- Apply the fitted step, unchanged, to the test set.
- The same holds for imputation and for the idf of TF-IDF; earlier slides fit on all rows only to explain the methods.
- Time series: the test set is the last days.

Checkpoint 2

Demo hours of the training set: mean 3.2, sd 1.2

Demo hours of the test set: mean 4.0, sd 2.0

A test game has 5.6 demo hours.

What is its standardized value?

Checkpoint 2

Demo hours of the training set: mean 3.2, sd 1.2

Demo hours of the test set: mean 4.0, sd 2.0

A test game has 5.6 demo hours.

What is its standardized value?

$(5.6 - 3.2)/1.2 = 2.0$: the training mean and sd

(not $(5.6 - 4.0)/2.0 = 0.8$)

Preprocessing in scikit-learn

```
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OrdinalEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer

SimpleImputer(strategy="median")  # numbers
SimpleImputer(strategy="most_frequent")  # categories
OrdinalEncoder(categories=[["Low", "High"]])  # ordered
OneHotEncoder()  # no order: one 0/1 column per value
CountVectorizer()  # text: word counts
TfidfVectorizer()  # text: TF-IDF, rows of length 1
# every fit uses the training set only
StandardScaler()  # standardize
```