Lecture 1: Machine Learning Foundations + PyTorch (start)

CSC 352 / CSC 552 · Fall 2026 · 2026-08-27 · 3:30–4:45 PM · CCE 030

Open in Colab → **File** → **Save a copy in Drive**. Run the install cell first; we look at a table of penguins while it installs.

In [1]:
```
%pip install -q autogluon.tabular
```

```
Requirement already satisfied: autogluon.tabular
```

Look at the data first

Machine learning starts with **examples**, not with a model name.

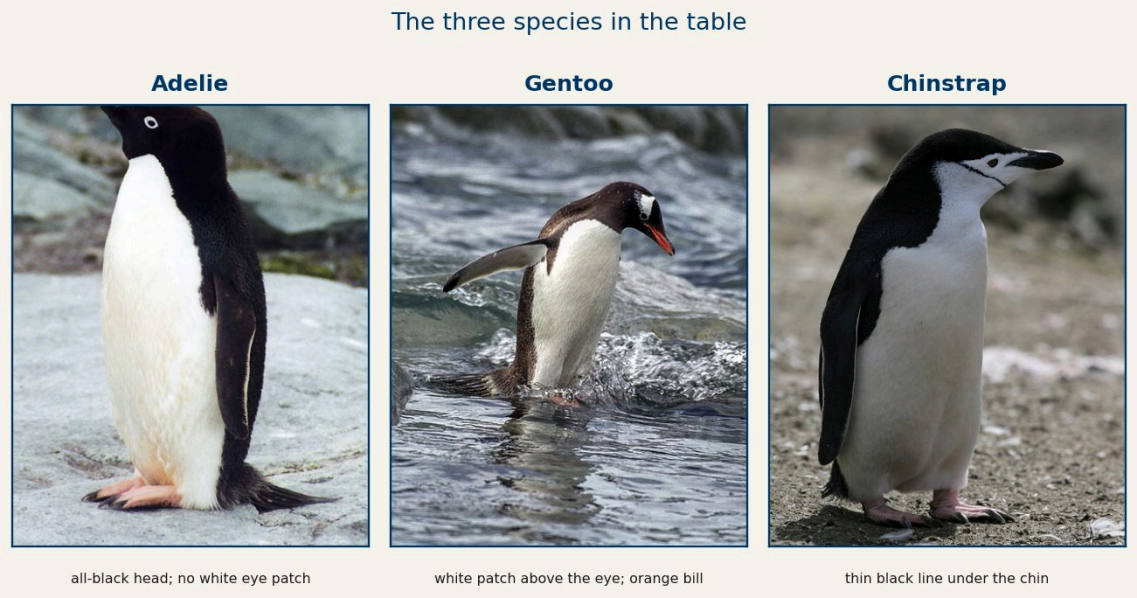
We will use **Palmer Penguins**: real measurements of penguins in Antarctica. Each **row** is one penguin. Each **column** is one fact about that penguin.

Column	What it is, in ordinary language
`species`	The kind of penguin: Adelie, Gentoo, or Chinstrap (three real species)
`island`	Where it was measured
`bill_length_mm`, `bill_depth_mm`	Beak size, in millimeters
`flipper_length_mm`	Flipper length, in millimeters
`body_mass_g`	How heavy it is, in **grams** (3,000 g = 3 kg, about 6.6 pounds)
`sex`	Female or male

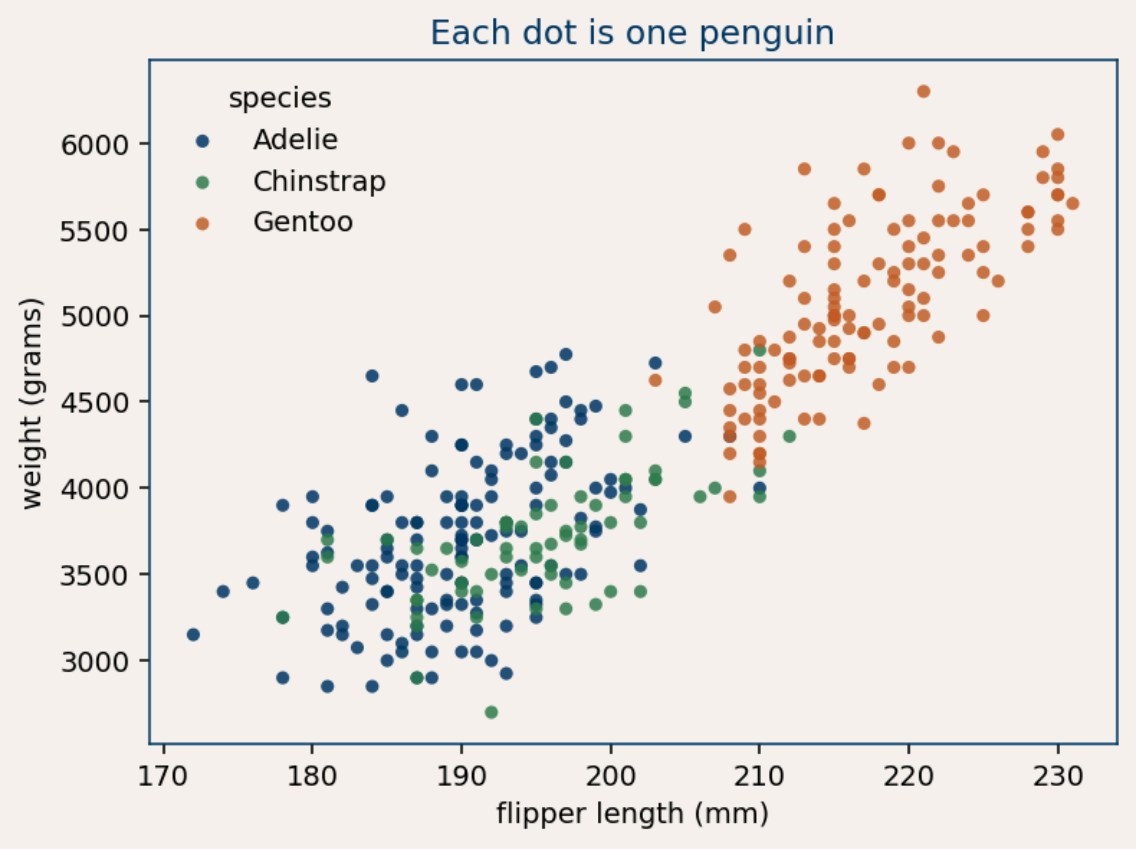
Some rows have a blank cell (a measurement was not taken). `.dropna()` throws those incomplete rows away so we do not have to guess missing numbers today.

This is the same *kind* of file you would download from Kaggle (https://www.kaggle.com) (datasets + competitions as CSV files). We use a public copy so you do not need a Kaggle account in class.

These are the three kinds named in the `species` column:



Adelie, Gentoo, and Chinstrap. Photos: Stan Shebs (CC BY-SA 3.0); Jerzy Strzelecki (CC BY 3.0); Gilad Rom (CC BY 2.0), via Wikimedia Commons.



Each dot is one penguin. Color is the species name already written on that row.

In [4]:
```
import pandas as pd

url = 'https://raw.githubusercontent.com/mwaskom/seaborn-data/master/penguins.csv'
penguins = pd.read_csv(url).dropna().reset_index(drop=True)
print('number of penguins', len(penguins))
print('columns', list(penguins.columns))
print('how many of each species')
print(penguins['species'].value_counts())
penguins.head()
```

```
number of penguins 333
columns ['species', 'island', 'bill_length_mm',
'bill_depth_mm', 'flipper_length_mm', 'body_mass_g', 'sex']
how many of each species
species
Adelie       146
Gentoo       119
Chinstrap     68
Name: count, dtype: int64
```

Out[4]:

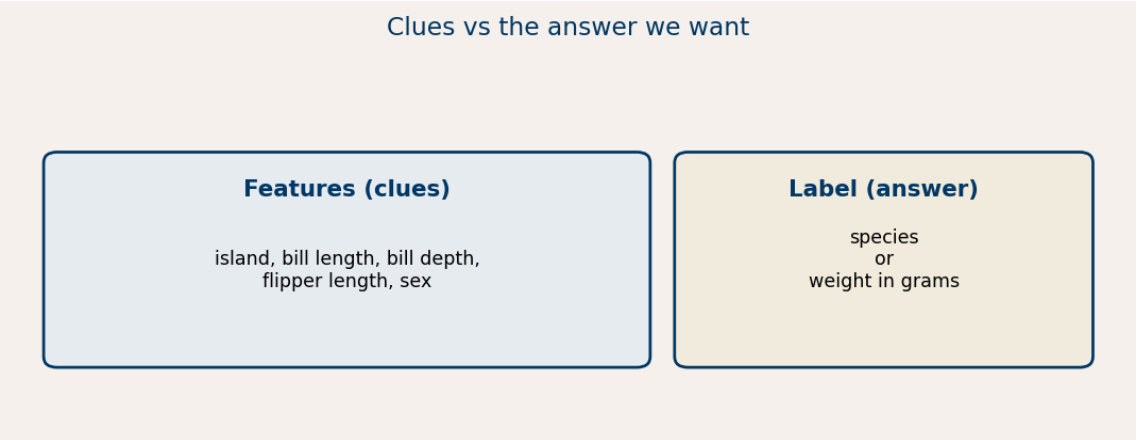
	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	bc
0	Adelie	Torgersen	39.1	18.7	181.0	
1	Adelie	Torgersen	39.5	17.4	186.0	
2	Adelie	Torgersen	40.3	18.0	195.0	
3	Adelie	Torgersen	36.7	19.3	193.0	
4	Adelie	Torgersen	39.3	20.6	190.0	

Read row 0 out loud: one Adelie penguin, on Torgersen island, with a measured beak, flipper, and weight. We are only looking at the whole table.

Features and the label

A **feature** is a clue (flipper length, island, …). A **label** is the answer we want the computer to guess (species, or weight).

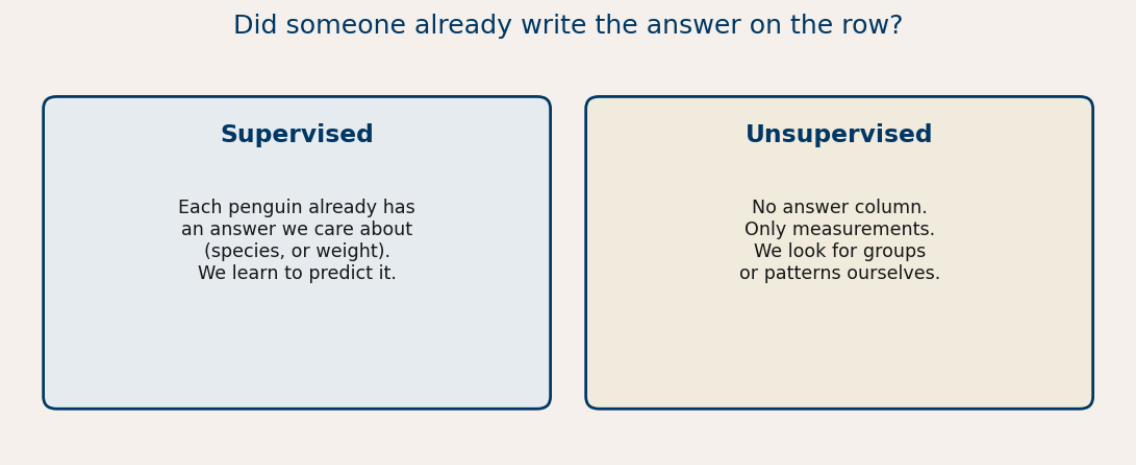
We choose the label. The other columns become features.



We pick one answer column. Everything else is a clue.

Supervised vs unsupervised

Ask: **did a human already write the answer on this row?**



Supervised: the answer is on the row. Unsupervised: no answer column.

Supervised learning: every example already has the answer. We hide that column, try to guess it from the features, and check the guess. Guessing a written-down species is supervised.

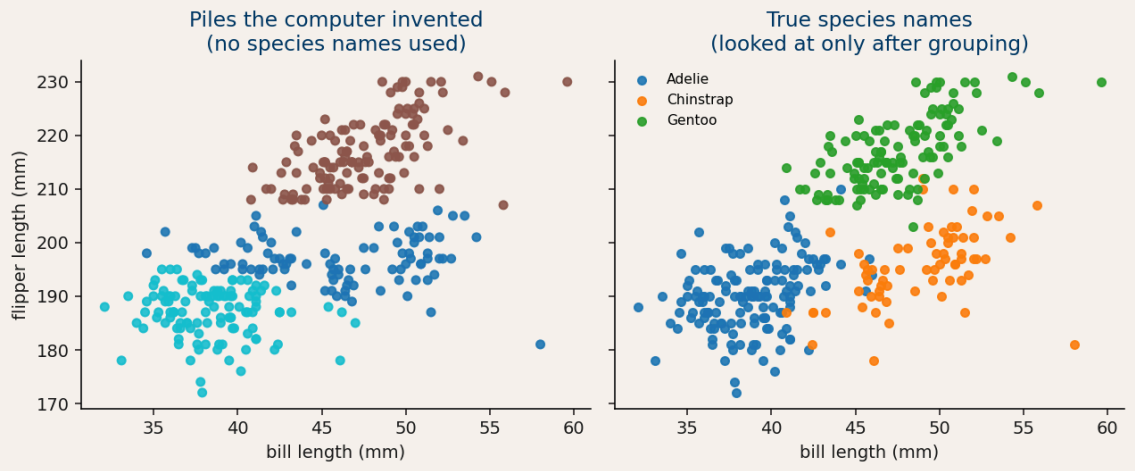
Unsupervised learning: no answer column. Only measurements. You might still notice clumps of similar penguins, but nobody told you the names Adelie / Gentoo.

Self-supervised learning: nobody typed a separate label column, but an answer is already in the data (hide a word, guess it). It is still "guess a hidden answer," so it is a kind of supervised learning. Language-model **training** works this way.

A later, free self-study path if you want more vocabulary: [Google's Machine Learning Crash Course (https://developers.google.com/machine-learning/crash-course)](). Later in this notebook, the supervised penguin jobs use AutoGluon.

Group similar penguins, no names

Hide `species`. Keep two measurements: beak length and flipper length. Ask the computer for **3 piles** of penguins that sit near each other. It does not get Adelie / Gentoo / Chinstrap.



Left: piles with no names. Right: true species, peeked after.

In [11]:
```
from sklearn.cluster import KMeans

xy = penguins[['bill_length_mm',
'flipper_length_mm']].to_numpy()
groups = KMeans(n_clusters=3, random_state=0,
n_init=10).fit_predict(xy)
print(pd.crosstab(groups, penguins['species'], rownames=
['group']))
```

```
species  Adelie  Chinstrap  Gentoo
group
0            38         54       2
1             2          5     117
2           106          9       0
```

Left: piles the computer invented (`0` , `1` , `2`). Right: the real species names, looked at only after grouping.

The piles often line up with species (Gentoo is almost one pile). Adelie and Chinstrap overlap, so they mix. That peek is a **check**, not the training signal: nobody wrote an answer column. This is **unsupervised** clustering. `KMeans` is one recipe for making the piles.

Exercise S — supervised, unsupervised, or something else?

For each story, write one of: **supervised**, **unsupervised**, **self-supervised**, or **generation / inference**.

- **Supervised:** a human (or a spreadsheet) already wrote the answer on the row.
- **Unsupervised:** no answer column. Only measurements. Find groups or "weird" points.
- **Self-supervised:** nobody labeled it, but we **make a fake answer from the data itself** (hide a word and guess it).
- **Generation / inference:** we are *using* an already-trained model to write new tokens. That is not training.

1. Our penguin table, `species` filled in. Guess the species from flipper length.
2. Same measurements, **delete** the `species` column. Cluster penguins that look alike.
3. While **training** an LLM: it reads "the cat sat" and must guess the next token "on". No volunteer labeled the sentence; the next token was already in the text.
4. You open ChatGPT. It writes a story **one token at a time**. Nobody is filling a label column right now.

In [1]:
```
# Exercise S — your attempt
# 1.
# 2.
# 3.
# 4.
```

Solution S

#	Answer	Why
1	**Supervised**	`species` is already on the row. We hide it and try to guess it.
2	**Unsupervised**	No answer column. Only "which penguins look similar?"
3	**Self-supervised**	Next-token **training**. The label is the next token already in the text. That is how GPT-style models pretrain.
4	**Generation (inference), not training**	ChatGPT is *using* a trained model to emit tokens. The *training* that built it was story 3.

This course: the grouping above is **unsupervised**. AutoGluon on penguins is **supervised** (**classification** or **regression**). LLM next-token **training** is **self-supervised**. ChatGPT answering you is **generation / inference**.

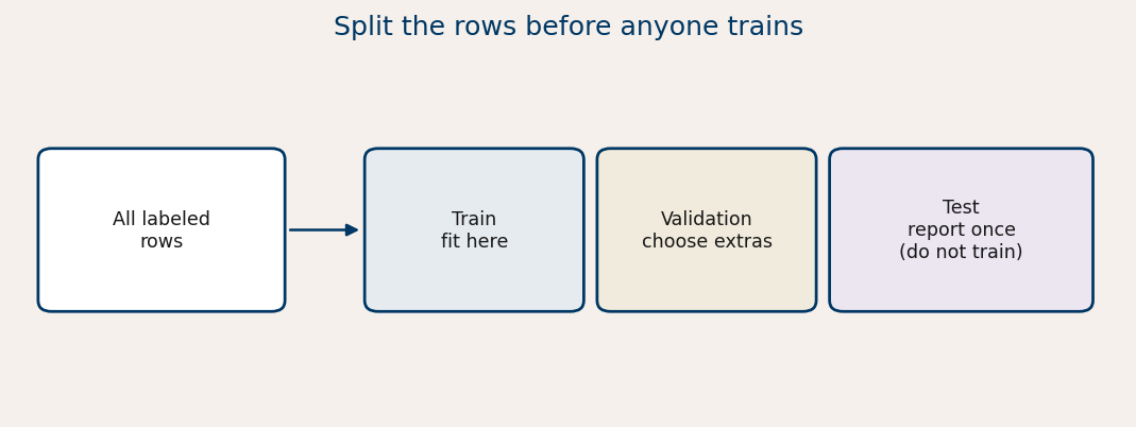
Holding some penguins back

If you study a penguin and then test yourself on **that same penguin**, you might have just memorized it. That does not tell you whether you would recognize a **new** penguin.

So we cut the table **before** we train:

- **Train:** the rows the computer is allowed to study.
- **Validation:** extra rows for choosing settings (when to stop). AutoGluon uses some of these internally.
- **Test:** rows we do not study. We look at the test score **once**. That is the honest number.

Studying the test rows is **leakage**. We will not do it.



Cut the rows first. Do not study the test rows.

Cut the penguin table before any fitting.

In [17]:
```
train = penguins.sample(frac=0.7, random_state=0)
test = penguins.drop(train.index)
print('train penguins', len(train))
print('test penguins', len(test))
print('species mix in train')
print(train['species'].value_counts(normalize=True).round(3)
)
```

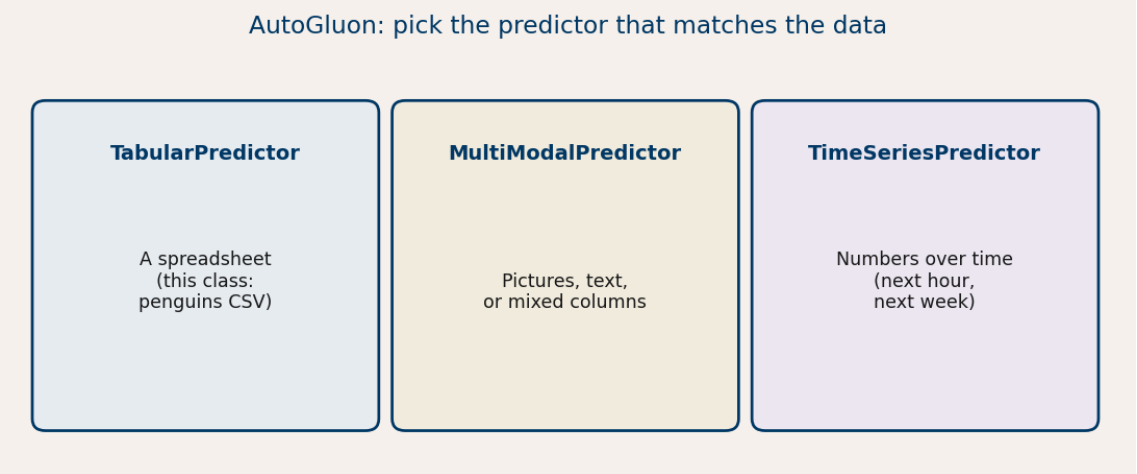
```
train penguins 233
test penguins 100
species mix in train
species
Adelie       0.464
Gentoo       0.352
Chinstrap    0.185
Name: proportion, dtype: float64
```

AutoGluon (AWS)

AutoGluon (https://auto.gluon.ai) is an open-source AutoML library from **AWS AI** (Amazon). The idea: you point at the label column; it trains one or more models and returns predictions. You do not pick "random forest vs neural net" by name today.

Amazon also wraps this in a no-code product (SageMaker Canvas (https://aws.amazon.com/sagemaker/canvas/)). In class we use the Python library, because we will need Python for PyTorch later.

Docs if you want to go further: tabular quick start (https://auto.gluon.ai/stable/tutorials/tabular/tabular-quick-start.html).



Three AutoGluon predictors. Today we only need TabularPredictor.

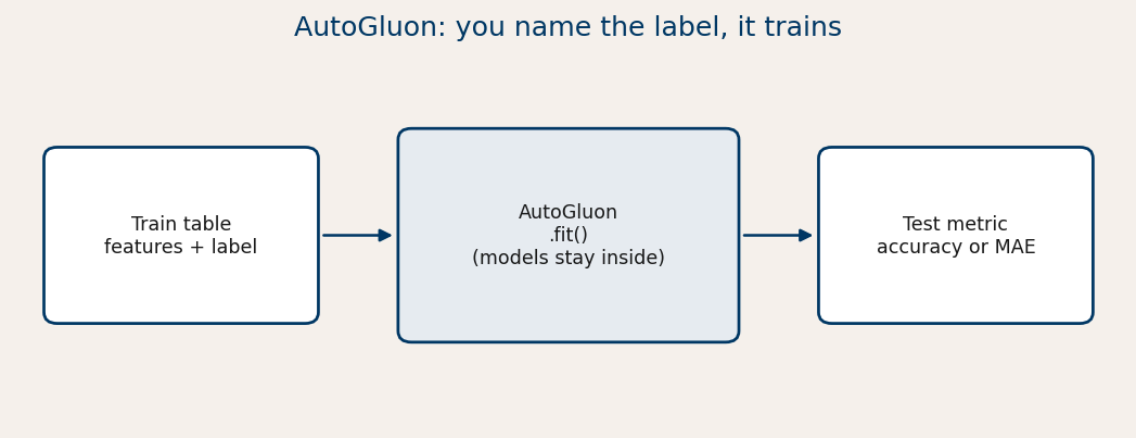
Predictor	What the input looks like	Use when
TabularPredictor	A spreadsheet / CSV	Penguin table (this class)
MultiModalPredictor	Images, text, or mixed columns	A photo of a penguin; a product review
TimeSeriesPredictor	A value that moves over time	Temperature for the next 48 hours

Install `autogluon.tabular` for tables only. `pip install autogluon` pulls the rest (heavier).

The four calls we will use:

1. `TabularPredictor(label=...)` — name the answer column.
2. `.fit(train)` — study the train rows.
3. `.predict(new_rows)` — guesses for new rows.
4. `.evaluate(test)` — the honest score on test rows. Bonus: `.leaderboard(test)` lists the models it tried, best first.

`time_limit=45` stops it after 45 seconds. `hyperparameters={'RF': {}}` means "just train a random forest" so the cell finishes in class. If you omit that, AutoGluon will try several models and combine them (slower, often stronger).



Name the label, fit on train, score on test.

Demo 1 — guess the species (a name)

In [23]:
```
from autogluon.tabular import TabularPredictor

clf = TabularPredictor(
    label='species',
    problem_type='multiclass',
    eval_metric='accuracy',
    path='ag_penguins_clf',
).fit(train, time_limit=45, hyperparameters={'RF': {}})
print('TEST (quote this)', clf.evaluate(test))
print('TRAIN (do not quote this)', clf.evaluate(train))
print(clf.predict(test.head()))
```

```
TEST (quote this) {'accuracy': 1.0, 'balanced_accuracy':
np.float64(1.0), 'mcc': np.float64(1.0)}
TRAIN (do not quote this) {'accuracy': 1.0,
'balanced_accuracy': np.float64(1.0), 'mcc':
np.float64(1.0)}
0     Adelie
9     Adelie
25    Adelie
28    Adelie
31    Adelie
Name: species, dtype: object
```

`eval_metric='accuracy'` means "percent of species names that match." If TRAIN is much higher than TEST, the fit memorized training penguins. We still quote TEST.

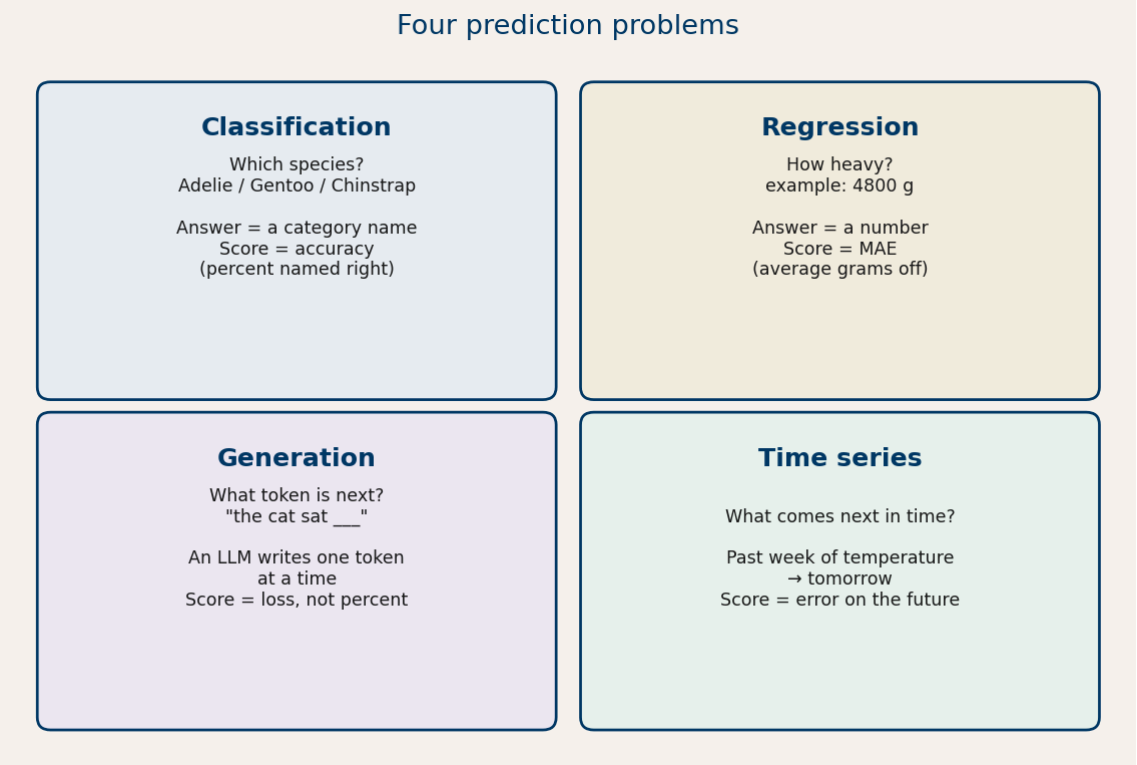
Demo 2 — guess the weight in grams (a number)

In [26]:
```
weight_cols = [c for c in train.columns if c != 'species']
reg = TabularPredictor(
    label='body_mass_g',
    problem_type='regression',
    eval_metric='mean_absolute_error',
    path='ag_penguins_reg',
).fit(train[weight_cols], time_limit=45, hyperparameters=
{'RF': {}})
print('TEST average grams off',
reg.evaluate(test[weight_cols]))
print(reg.predict(test[weight_cols].head()))
```

```
TEST average grams off {'mean_absolute_error':
-256.70667724609376, 'root_mean_squared_error':
np.float64(-330.0935224305818), 'mean_squared_error':
-108961.73355062903, 'r2': 0.8218789869150464, 'pearsonr':
0.9117613530564115, 'median_absolute_error':
np.float64(-214.8333740234375)}
0     3705.833252
9     4077.083252
25    3024.583252
28    3904.916748
31    3961.166748
Name: body_mass_g, dtype: float32
```

`mean_absolute_error` is how many grams off you are, on average. AutoGluon prints it **negative** on purpose (its scores follow "higher is better"). Quote the size: about 257 g off on test, not the minus sign.

Four prediction problems



Classification, regression, generation, and time series.

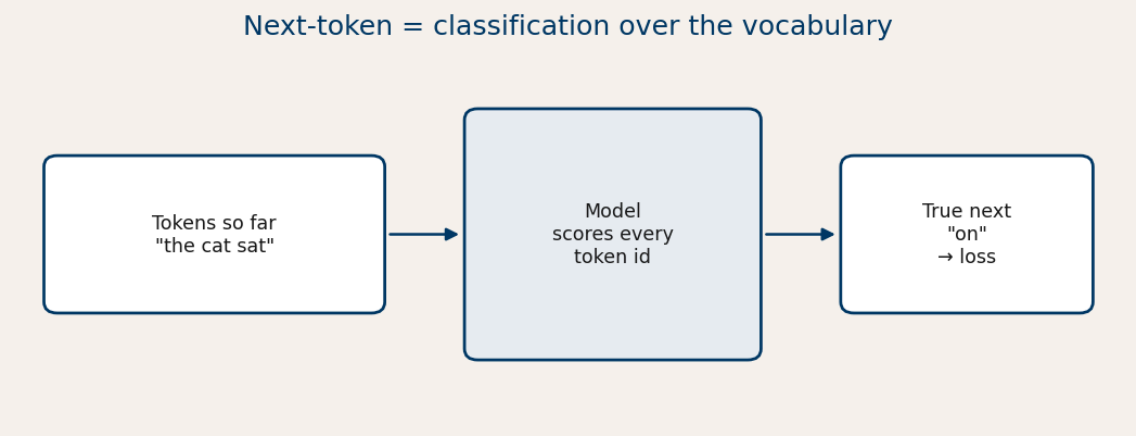
- **Classification:** the answer is a **category name** (Adelie / Gentoo / Chinstrap). Score = **accuracy** = percent named correctly.
- **Regression:** the answer is a **number** (weight in grams). Score = **MAE** = average grams you are off.
- **Generation:** the model writes new tokens one at a time (an LLM completing "the cat sat ___"). Training uses next-token **loss**, not percent exact. When ChatGPT is already trained and you use it, that is **inference / generation**, not a new training run.
- **Time-series prediction:** the answer is **the next value in time** (yesterday's temperatures → tomorrow). The split is by time, not a random shuffle of rows. AutoGluon has `TimeSeriesPredictor`.

Next word

Guessing the next piece of text is the same *kind* of job as guessing a penguin species: pick one name from a list. The list is just huge (the vocabulary).

A **token** is one piece of text the model sees — for today, think "a word or a word-piece." How we split text into tokens is Lecture 3.

We do **not** quote "percent of sentences copied exactly." That number stays near zero even when the model is good. We quote **loss** instead.



Tokens so far → scores for every possible next word → loss.

The model first writes a **score** for every option. Bigger means "I like this one more." Those raw scores are **logits**. They are not percentages yet (they need not add to 1, and they can be negative).

We squeeze logits into **probabilities** that *do* add to 1. That squeeze is called **softmax**. You do not need the formula today; we will use softmax all term.

Loss (here: **cross-entropy**) is one number for "how unhappy we are with the probability on the *true* next token."

- True word got probability 0.8 → loss is small.
- True word got 0.01 → loss is large.

The code uses `-log(probability of the true id)`. Smaller is better. Training later in the course means turning knobs to make this number go down. Today we only read it.

In [32]:
```
import numpy as np

logits = np.array([2.0, 0.5, -1.0])  # bigger = more
confident in that word
probs = np.exp(logits - logits.max())
probs = probs / probs.sum()
true_id = 0
print('probability of each word', np.round(probs, 3))
print('loss (smaller is better)', round(float(-
np.log(probs[true_id])), 3))
print('top guess is the true word?', bool(probs.argmax() ==
true_id))
```

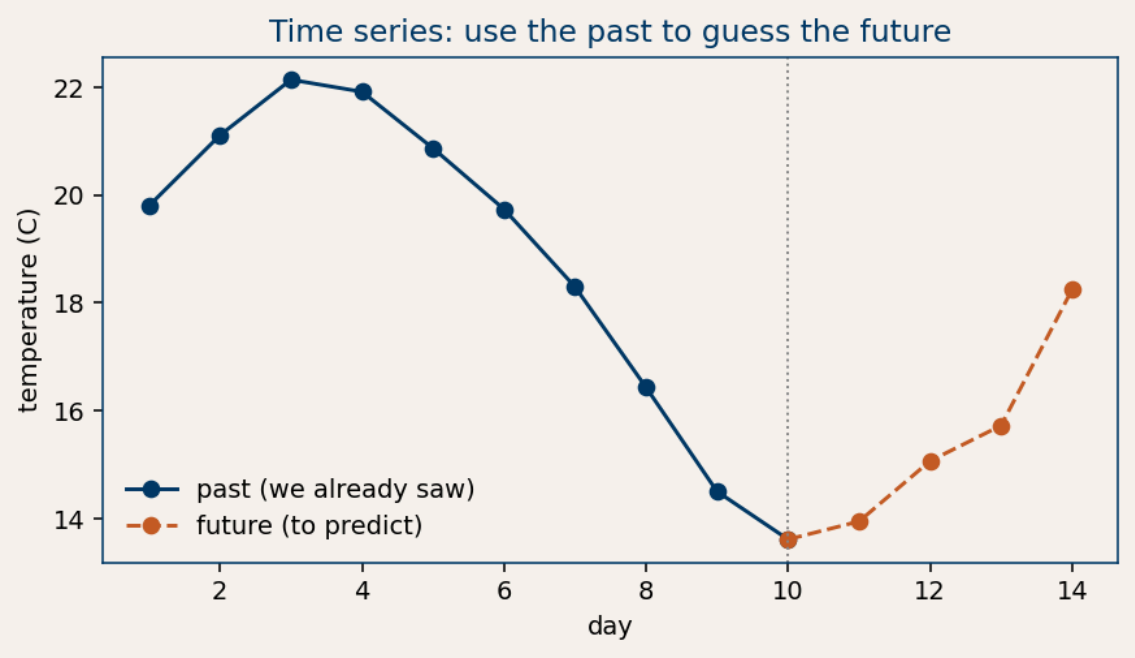
```
probability of each word [0.786 0.175 0.039]
loss (smaller is better) 0.241
top guess is the true word? True
```

Time-series prediction

A **time series** is the same quantity measured again and again in order: temperature each day, passengers each month, a stock price each hour.

The question is not "which penguin is this?" It is **what comes next on the calendar**. You must not shuffle days the way we shuffle penguins: tomorrow cannot leak into yesterday.

Toy example: 10 days of temperature we already saw, 4 future days we pretend we do not know. A naive guess for the future is "same as the last day we saw." That is a **forecast**. Better models (including AutoGluon's `TimeSeriesPredictor`) use the whole past curve.



Solid = past we already saw. Dashed = future we want to predict.

In [35]:
```
import numpy as np

rng = np.random.default_rng(0)
day = np.arange(1, 15)
temp = 18 + 4 * np.sin(day / 2.2) + rng.normal(0, 0.35,
size=len(day))
past, future = temp[:10], temp[10:]
naive = np.full_like(future, past[-1])
mae = float(np.mean(np.abs(naive - future)))
print('last past day (C)', round(float(past[-1]), 2))
print('true future (C)', np.round(future, 2))
print('naive forecast: repeat last day', np.round(naive, 2))
print('MAE on the future (C)', round(mae, 2))
```

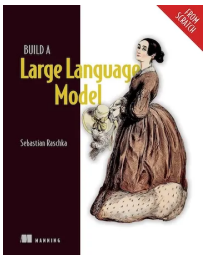
```
last past day (C) 13.61
true future (C) [13.95 15.07 15.72 18.24]
naive forecast: repeat last day [13.61 13.61 13.61 13.61]
MAE on the future (C) 2.13
```

Appendix A.1–A.4 (Raschka)

Official companion: [code-part1 (https://github.com/rasbt/LLMs-from-scratch/blob/main/appendix-A/01_main-chapter-code/code-part1.ipynb)](https://github.com/rasbt/LLMs-from-scratch/blob/main/appendix-A/01_main-chapter-code/code-part1.ipynb). Today we stop after **automatic differentiation**. Next class starts at **A.5 Implementing multilayer neural networks**.

Supplementary code for the Build a Large Language Model From Scratch (http://mng.bz/orYv) book by Sebastian Raschka (https://sebastianraschka.com)

Code repository:
https://github.com/rasbt/LLMs-from-scratch (https://github.com/rasbt/LLMs-from-scratch)



(http://mng.bz/orYv)

Appendix A: Introduction to PyTorch (Part 1)

A.1 What is PyTorch

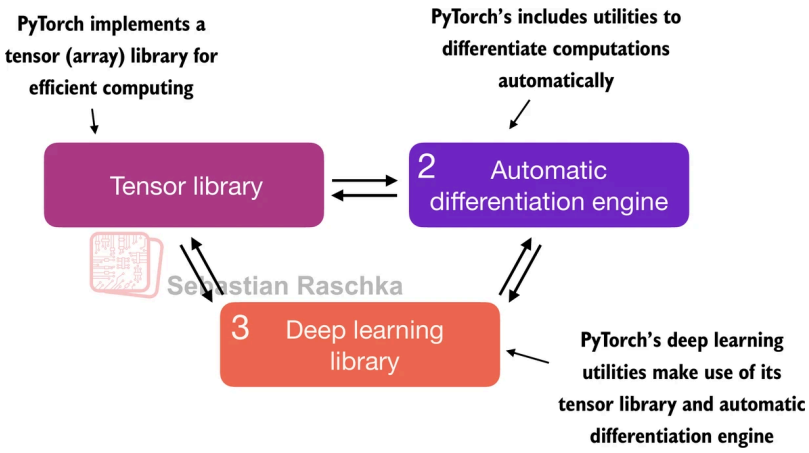
PyTorch is a Python library for boxes of numbers (tensors) and for turning knobs using gradients. The `import torch` cell checks the version. `cuda` is the GPU; `False` means this machine is using the CPU, which is fine for today.

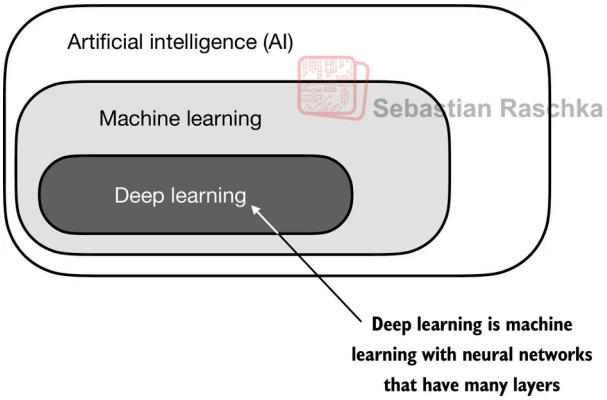
In [40]:
```
import torch

print(torch.__version__)
```

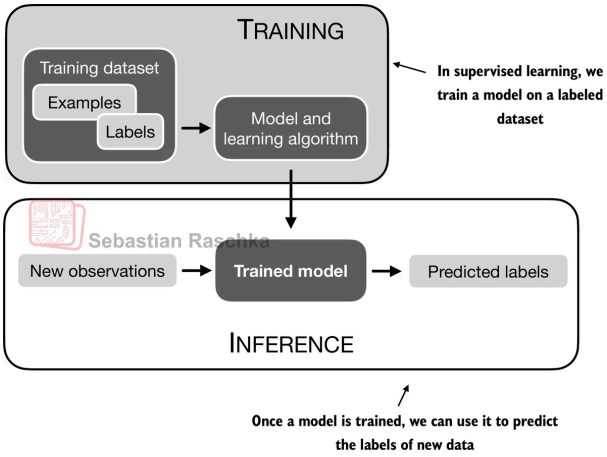
```
2.8.0
```

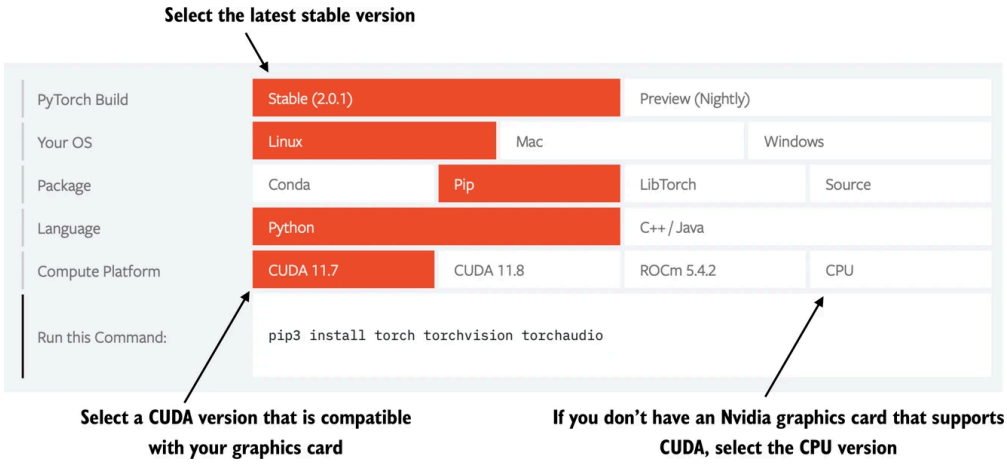
In [41]:
```
print(torch.cuda.is_available())
```

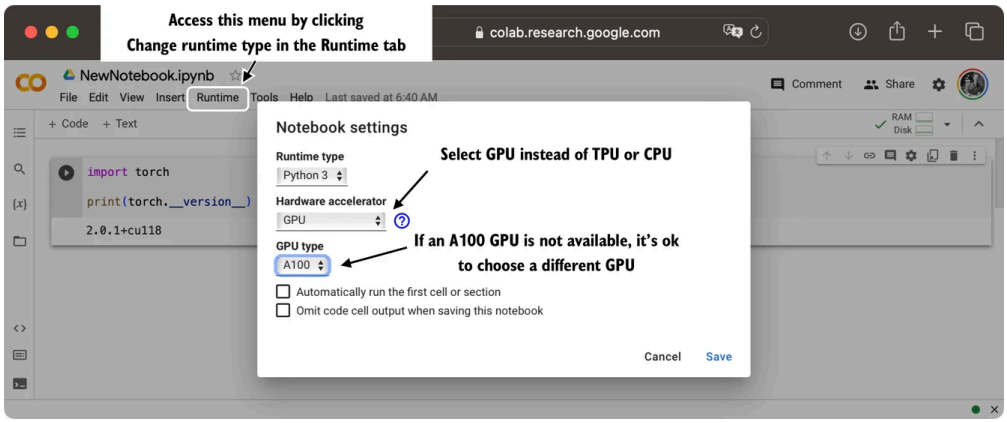
```
False
```




Artificial intelligence (AI)
Machine learning
Deep learning
Sebastian Raschka
Deep learning is machine learning with neural networks that have many layers

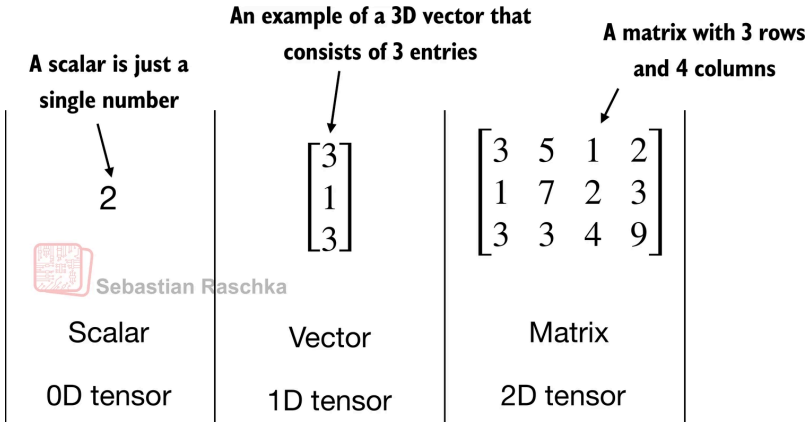

TRAINING
Training dataset
Examples
Labels
Model and learning algorithm
In supervised learning, we train a model on a labeled dataset
Sebastian Raschka
New observations
Trained model
Predicted labels
INFERENCE
Once a model is trained, we can use it to predict the labels of new data


Select the latest stable version
PyTorch Build
Stable (2.0.1)
Preview (Nightly)
Your OS
Linux
Mac
Windows
Package
Conda
Pip
LibTorch
Source
Language
Python
C++ / Java
Compute Platform
CUDA 11.7
CUDA 11.8
ROCm 5.4.2
CPU
Run this Command:
pip3 install torch torchvision torchaudio
Select a CUDA version that is compatible with your graphics card
If you don't have an Nvidia graphics card that supports CUDA, select the CPU version



A.2 Understanding tensors

A **tensor** is PyTorch's box of numbers: one number, a list, a table, or a stack of tables. Same idea as a NumPy array. **`shape`** is how the box is laid out (rows × columns, …). **`dtype`** is the kind of number (integer vs float). We will use these operations when we build the net.



A.2.1 Scalars, vectors, matrices, and tensors

In [47]:
```
import torch
import numpy as np

# create a 0D tensor (scalar) from a Python integer
tensor0d = torch.tensor(1)

# create a 1D tensor (vector) from a Python list
tensor1d = torch.tensor([1, 2, 3])

# create a 2D tensor from a nested Python list
tensor2d = torch.tensor([[1, 2],
                         [3, 4]])

# create a 3D tensor from a nested Python list
tensor3d_1 = torch.tensor([[[1, 2], [3, 4]],
                           [[5, 6], [7, 8]]])

# create a 3D tensor from NumPy array
ary3d = np.array([[[1, 2], [3, 4]],
                  [[5, 6], [7, 8]]])
tensor3d_2 = torch.tensor(ary3d)  # Copies NumPy array
tensor3d_3 = torch.from_numpy(ary3d)  # Shares memory with
NumPy array
```

In [48]:
```
ary3d[0, 0, 0] = 999
print(tensor3d_2) # remains unchanged
```

```
tensor([[[1, 2],
         [3, 4]],

        [[5, 6],
         [7, 8]]])
```

In [49]:
```
print(tensor3d_3) # changes because of memory sharing
```

```
tensor([[[999,   2],
         [  3,   4]],

        [[  5,   6],
         [  7,   8]]])
```

A.2.2 Tensor data types

In [51]:
```
tensor1d = torch.tensor([1, 2, 3])
print(tensor1d.dtype)
```

```
torch.int64
```

In [52]:
```
floatvec = torch.tensor([1.0, 2.0, 3.0])
print(floatvec.dtype)
```

```
torch.float32
```

In [53]:
```
floatvec = tensor1d.to(torch.float32)
print(floatvec.dtype)
```

```
torch.float32
```

A.2.3 Common PyTorch tensor operations

This 2 × 3 table is the running example: 2 rows, 3 columns.

In [1]:
```
tensor2d = torch.tensor([[1, 2, 3],
                         [4, 5, 6]])  # 2 rows, 3 columns
tensor2d
```

Out[1]:
```
tensor([[1, 2, 3],
        [4, 5, 6]])
```

`shape` is (rows, columns). Here that is `(2, 3)`.

In [2]:
```
tensor2d.shape  # torch.Size([2, 3]) means 2 rows × 3 columns
```

Out[2]:
```
torch.Size([2, 3])
```

`reshape` and **`view`** put the **same numbers** into a new layout. They do not change the values. `view` needs the numbers to sit in one contiguous block of memory; `reshape` will copy if it has to. For this table they match.

```
In [3]: tensor2d.reshape(3, 2)  # same six numbers, now 3 rows × 2 columns
Out[3]: tensor([[1, 2],
                [3, 4],
                [5, 6]])
```

`view` does the same rearrangement for this table:

```
In [4]: tensor2d.view(3, 2)  # same result as reshape here
Out[4]: tensor([[1, 2],
                [3, 4],
                [5, 6]])
```

`.T` **transposes** the table: rows become columns. `(2, 3)` → `(3, 2)`. We need this before matrix multiply, because the inner sizes have to match.

```
In [5]: tensor2d.T  # columns of tensor2d become rows
Out[5]: tensor([[1, 4],
                [2, 5],
                [3, 6]])
```

Matrix multiply is not "multiply matching entries."

- Python `*` on tensors is **element-wise**: each pair of numbers in the **same position** is multiplied. Both sides need the **same shape**.
- `.matmul(...)` and `@` (the operator is read **"at"**) are **matrix multiplication**. Each output number is a **dot product**: take one **row of the left** table, one **column of the right** table, multiply those pairs, then add.

Shape rule: `(m × n) @ (n × p)` → `(m × p)`. The inner `n` must match.

`tensor2d` is 2 × 3, so `tensor2d @ tensor2d` is illegal (3 ≠ 2). `tensor2d.T` is 3 × 2, so `tensor2d @ tensor2d.T` is 2 × 3 times 3 × 2 → **2 × 2**.

Top-left entry of that 2 × 2: first row of `tensor2d` is `[1, 2, 3]`, first column of `tensor2d.T` is also `[1, 2, 3]`:

`1·1 + 2·2 + 3·3 = 14`.

In [6]:
```
# matrix multiply: (2×3) @ (3×2) → (2×2)
# top-left 14 = 1*1 + 2*2 + 3*3   (row 0 · row 0, because .T
turns row 0 into column 0)
# top-right 32 = 1*4 + 2*5 + 3*6
# bottom-left 32 = 4*1 + 5*2 + 6*3
# bottom-right 77 = 4*4 + 5*5 + 6*6
tensor2d.matmul(tensor2d.T)
```

Out[6]:
```
tensor([[14, 32],
        [32, 77]])
```

`@` is the same operation as `.matmul`. People write `@` because it is shorter. Both are matrix multiply. Neither is `*`.

In [7]:
```
tensor2d @ tensor2d.T  # identical to
tensor2d.matmul(tensor2d.T)
```

Out[7]:
```
tensor([[14, 32],
        [32, 77]])
```

`*` needs the same shape, so here both sides stay 2 × 3. Each matching pair is multiplied. That is **not** the 2 × 2 matrix product above.

In [8]:
```
tensor2d * tensor2d  # element-wise: [[1, 4, 9], [16, 25,
36]]
```

Out[8]:
```
tensor([[ 1,  4,  9],
        [16, 25, 36]])
```

On two **vectors** of the same length, `@` is exactly the **dot product** (one number). That is the same arithmetic as **one entry** of a matrix multiply. `torch.dot` is the same thing for 1-D tensors.

In [9]:
```
u = torch.tensor([1, 2, 3])
v = torch.tensor([1, 2, 3])
print('u @ v (dot product)', u @ v)          # 1*1 + 2*2 +
3*3 = 14
print('torch.dot(u, v)     ', torch.dot(u, v))  # same 14
print('u * v (element-wise)', u * v)          # [1, 4, 9],
still a vector
```

```
u @ v (dot product) tensor(14)
torch.dot(u, v)      tensor(14)
u * v (element-wise) tensor([1, 4, 9])
```

Exercise (matrix multiply)

`t` is already built. Print `t.shape`, `t.T`, and the **matrix product** `t @ t.T` (same as `t.matmul(t.T)`). Check that the top-left entry equals `1*1 + 2*2 + 3*3`.

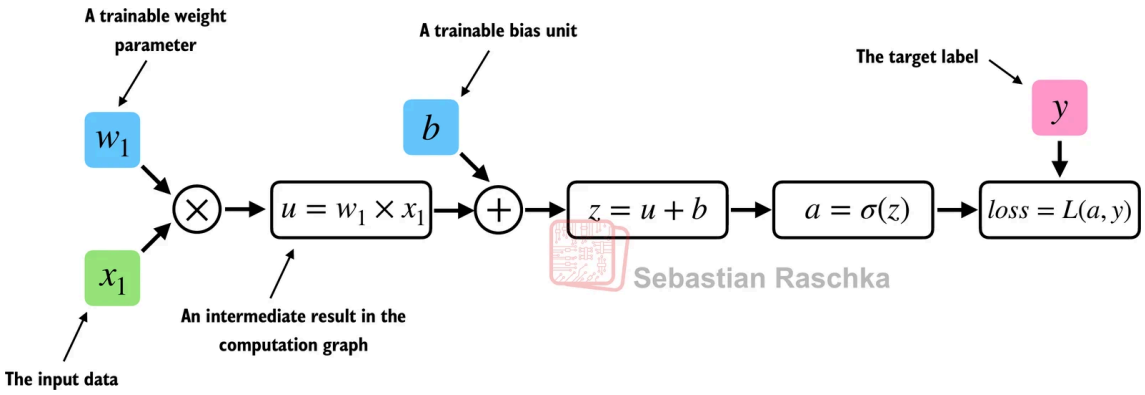
In [1]:
```
# Exercise 3 — your attempt
import torch

t = torch.tensor([[1, 2, 3], [4, 5, 6]])
print()  # shape: rows × columns
print()  # transpose: .T flips rows and columns
print()  # matrix multiply t @ t.T  (not element-wise *)
```

Solution (matrix multiply)

In [1]:
```
# Solution 3 — shape is rows × columns. .T flips them.
# @ (and .matmul) is matrix multiply: each entry is a
row·column dot product.
import torch

t = torch.tensor([[1, 2, 3], [4, 5, 6]])
print('shape', tuple(t.shape))
print(t.T)
print(t @ t.T)
print('top-left by hand', 1*1 + 2*2 + 3*3)
```

```
shape (2, 3)
tensor([[1, 4],
        [2, 5],
        [3, 6]])
tensor([[14, 32],
        [32, 77]])
top-left by hand 14
```

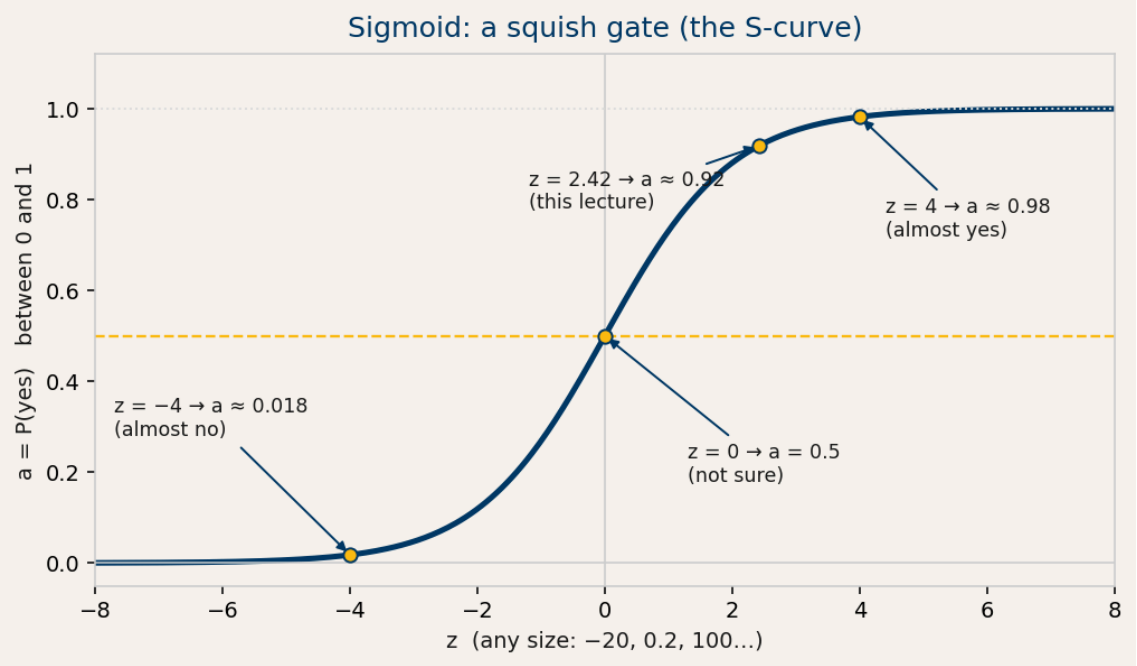
A.3 Seeing models as computation graphs



The picture is **one neuron**: a tiny mixing desk with two knobs.

- The **ingredient** is the input `x1`.
- `w1` is a **volume knob** (the **weight**): how loud that ingredient is.
- `b` is a **nudge** (the **bias**): a constant push, even if `x1` is zero.
- **Mix:** `z = x1 * w1 + b`. `z` can be 100, 0.2, or –20.
- **Squish: sigmoid** forces *any* mix into a number between **0 and 1**. Call it `a`. Read it as **P(yes)**. The curve is an **S**.

Look at that S first, then a few numbers through it.



Sigmoid is the S-shaped squish gate: any z becomes a number between 0 and 1.

Each `z` in, one `a` out. Big negative mix → almost no. Mix near 0 → coin flip. Big positive mix → almost yes. `z = 2.42` is the mix we get from `x1 = 1.1` and `w1 = 2.2` in the next cells.

In [2]:
```
# a few mixes through the squish gate
z = torch.tensor([-20.0, -4.0, 0.0, 2.42, 4.0, 100.0])
a = torch.sigmoid(z)
print('   z (mix)     a = P(yes)')
for zi, ai in zip(z, a):
    print(f'{float(zi):10.2f}   {float(ai):.6f}')
```

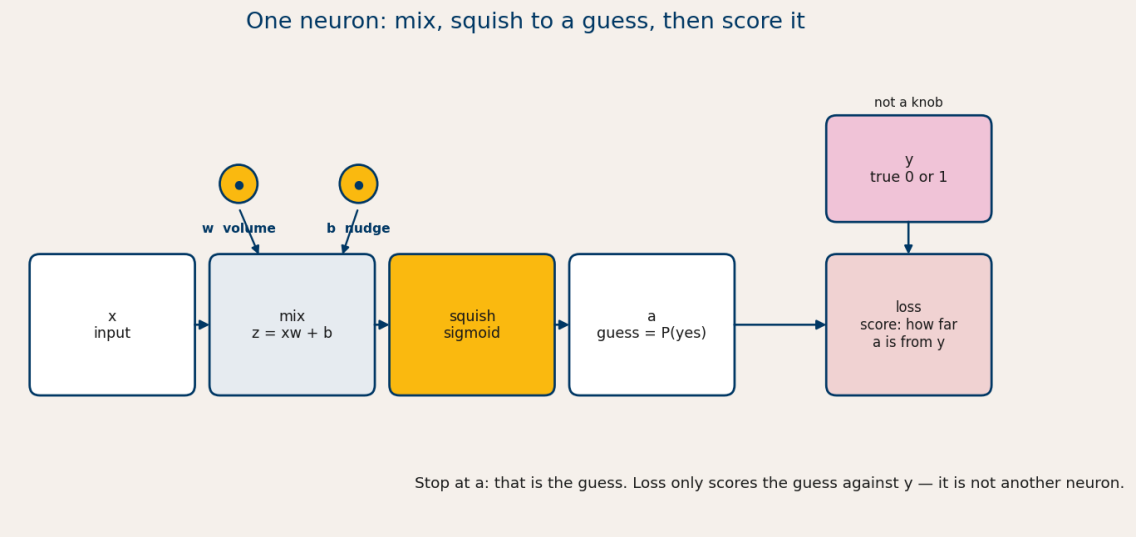
```
   z (mix)     a = P(yes)
    -20.00   0.000000
     -4.00   0.017986
      0.00   0.500000
      2.42   0.918340
      4.00   0.982014
    100.00   1.000000
```

The printed `a` **is the model's guess** — not the true answer.

In Raschka's picture (and the mixing-desk picture next):

- **`a`** = the guess, `P(yes)`, after sigmoid.
- **`y`** = the true 0 or 1 (the pink box). It is **not** a knob.
- **`loss`** = a **score** for that guess: how far `a` is from `y`. Loss is not another neuron and not another mix.

The desk **stops at `a`**. Loss only scores the guess.



The desk outputs a guess a. y is the true 0 or 1. loss scores how far a is from y — it is not another neuron.

Binary cross-entropy is the yes/no version of the next-word **cross-entropy** above. Same question: *how much probability did we put on the true answer?* Then take `-log` of that number (natural log — the same `log` PyTorch uses). Small `-log` means we were not surprised; large `-log` means we almost missed.

- True answer is **yes** (`y = 1`): loss = `-log(a)` . We wanted `a` near 1.
  - `a = 0.9` → `-log(0.9)` ≈ `0.105` (small)
  - `a = 0.1` → `-log(0.1)` ≈ `2.30` (large — we said "probably no" when it was yes)
- True answer is **no** (`y = 0`): loss = `-log(1 - a)` . We wanted `a` near 0.

One line that covers both cases:

```
loss = -(y * log(a) + (1 - y) * log(1 - a))
```

`F.binary_cross_entropy(a, y)` computes that. Next-word uses the many-class cousin: `-log` of the **true word's** probability. Smaller is better.

In [3]:
```
import torch.nn.functional as F

y = torch.tensor([1.0])   # true label: 1 = yes
x1 = torch.tensor([1.1])  # input feature
w1 = torch.tensor([2.2])  # weight (a knob)
b = torch.tensor([0.0])   # bias (the other knob)

z = x1 * w1 + b           # net input; can be any real
number
a = torch.sigmoid(z)      # squash z into (0, 1) = P(yes)

# y is 1, so this is -log(a). Same idea as next-word: -
log(prob on the true answer).
loss = F.binary_cross_entropy(a, y)
print('a = P(yes)', a)
print('loss     ', loss)
print('-log(a)  ', -torch.log(a))  # matches loss because y
= 1
```

```
a = P(yes) tensor([0.9183])
loss       tensor(0.0852)
-log(a)    tensor([0.0852])
```

The printed `loss` is already small here because this made-up `w1` pushes sigmoid close to 1 (the true label). A worse guess makes the same formula large:

In [4]:
```
# same formula, two made-up probabilities, true answer still
yes (y = 1)
print('good guess a=0.9, y=1  loss', round(float(-
torch.log(torch.tensor(0.9))), 3))
print('bad guess  a=0.1, y=1  loss', round(float(-
torch.log(torch.tensor(0.1))), 3))
# if the true answer had been no (y = 0), we would use -
log(1 - a) instead
```

```
good guess a=0.9, y=1  loss 0.105
bad guess  a=0.1, y=1  loss 2.303
```

Exercise (binary cross-entropy)

True answer is yes (`y = 1`). Using `-log(a)` , print the loss for `a = 0.8` and for `a = 0.2` . Which guess is worse, and by roughly how much?

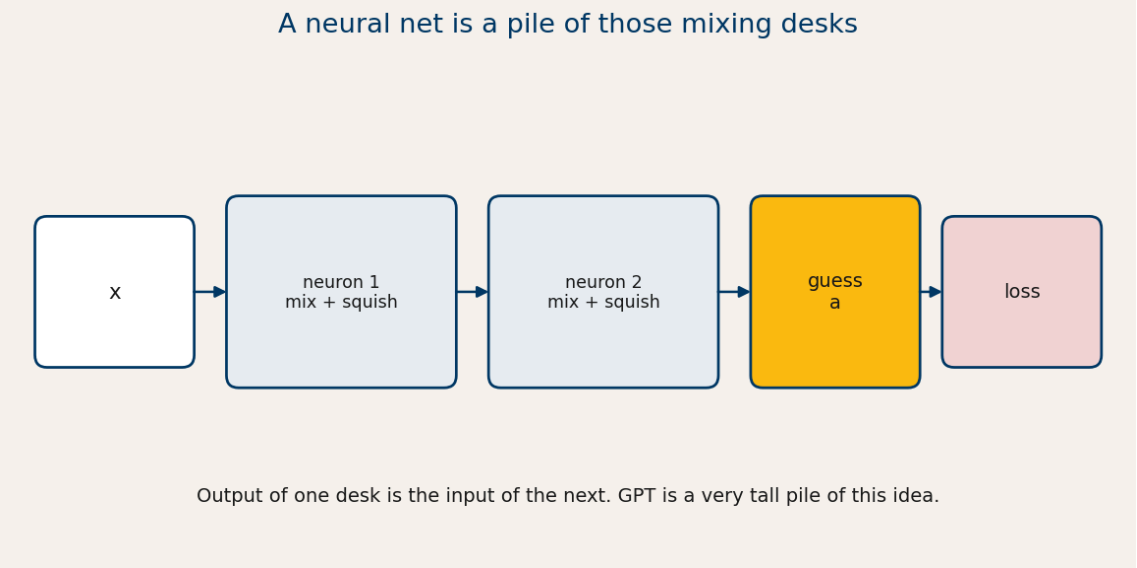
In [1]:
```
# Exercise (binary cross-entropy) — your attempt
import torch

a_good = torch.tensor(0.8)
a_bad = torch.tensor(0.2)
# loss = -log(a) because y = 1
loss_good = ...
loss_bad = ...
print('a=0.8 loss', loss_good)
print('a=0.2 loss', loss_bad)
```

Solution (binary cross-entropy)

In [5]:
```
# Solution — y = 1, so loss is -log(a). Smaller a on the
true answer → larger loss.
import torch

a_good = torch.tensor(0.8)
a_bad = torch.tensor(0.2)
loss_good = float(-torch.log(a_good))
loss_bad = float(-torch.log(a_bad))
print('a=0.8 loss', round(loss_good, 3))
print('a=0.2 loss', round(loss_bad, 3))
print('worse is a=0.2; about', round(loss_bad / loss_good,
1), 'times as large')
```

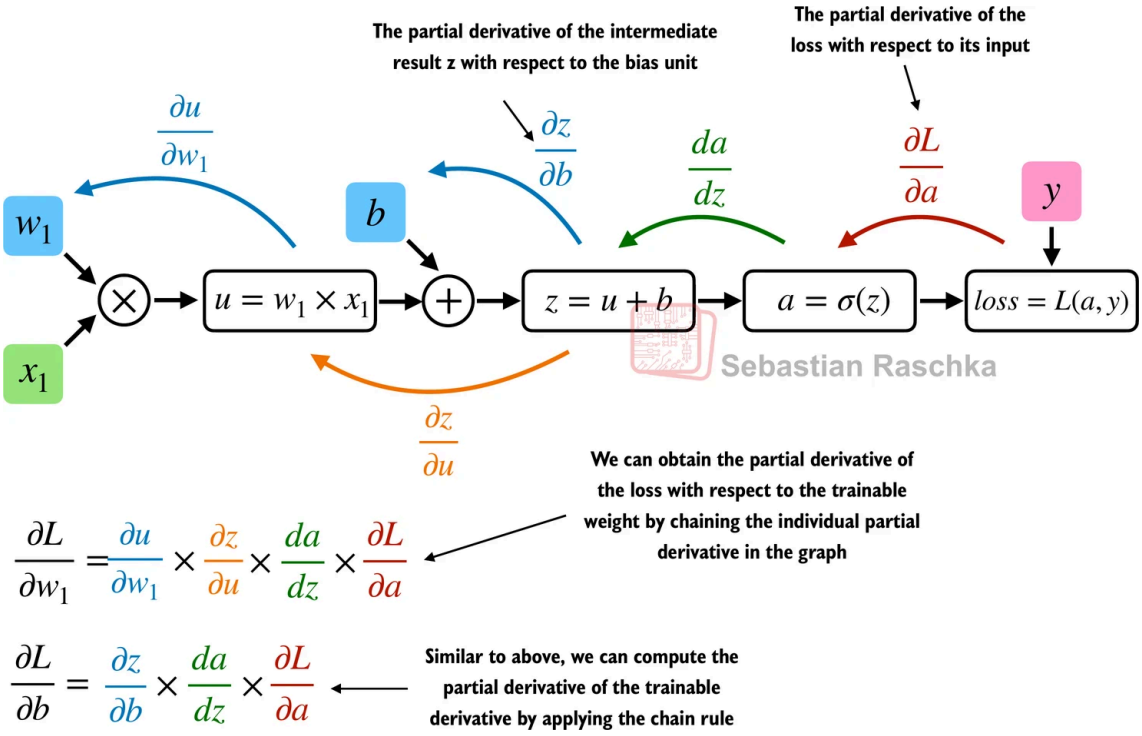
```
a=0.8 loss 0.223
a=0.2 loss 1.609
worse is a=0.2; about 7.2 times as large
```

A **neural net** is a **pile** of these mixing desks. The output of one is the input of the next. GPT is a very tall pile of the same idea. We just ran **one** desk. Stacking them as `nn.Module` is Appendix A.5.



A neural net is a pile of mixing desks. Output of one is the input of the next.

A.4 Automatic differentiation made easy



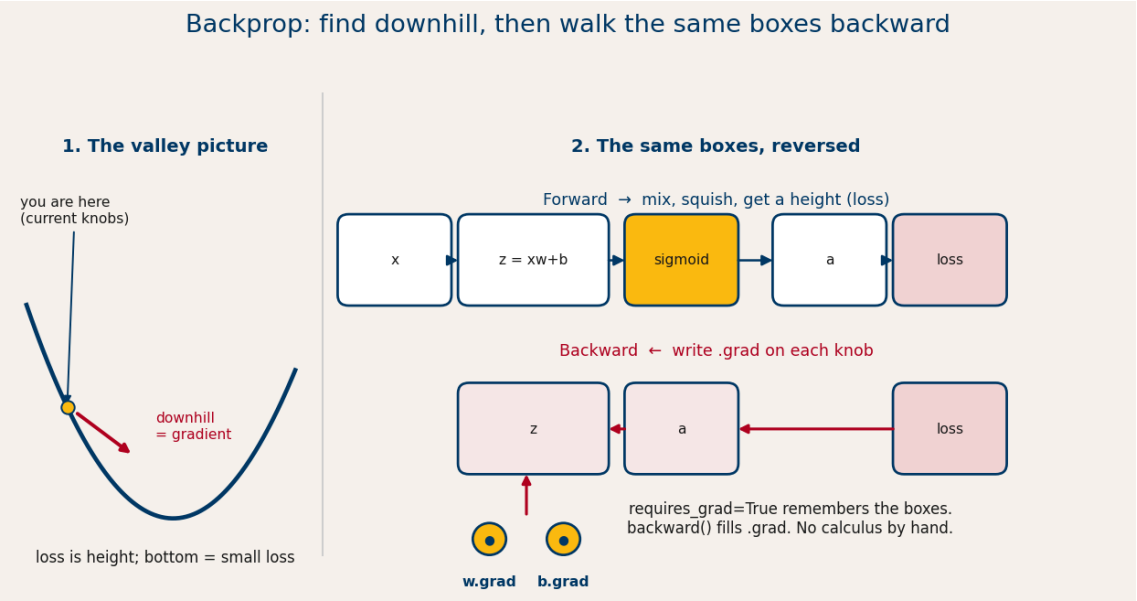
Two pictures for the same move.

The valley. Loss is **height**. We stand on a slope. The **gradient** is the arrow “downhill is that way.” Later we take a **small step** along that arrow. We do not jump across the valley.

The chain of boxes. Forward is left to right: mix, squish, then a height (`loss`). `requires_grad=True` means “remember this box.” **Backpropagation** — `loss.backward()` — walks **the same boxes in reverse** (loss → a → z → knobs) and writes `.grad` on each knob.

- Negative `.grad` : turning the knob **up** would make the height **smaller** (locally).
- Positive `.grad` : turning the knob **up** would make the height **larger**.

PyTorch walks the chain. We do not differentiate by hand.



Left: loss is height; the red arrow is downhill. Right: forward left to right, backward writes .grad on each knob.

In [6]:
```
import torch.nn.functional as F
from torch.autograd import grad

y = torch.tensor([1.0])
x1 = torch.tensor([1.1])
w1 = torch.tensor([2.2], requires_grad=True)
b = torch.tensor([0.0], requires_grad=True)

z = x1 * w1 + b
a = torch.sigmoid(z)

loss = F.binary_cross_entropy(a, y)  # height of the valley
(still -log(a) because y = 1)

grad_L_w1 = grad(loss, w1, retain_graph=True)  # which way
is downhill for w1?
grad_L_b = grad(loss, b, retain_graph=True)    # which way
is downhill for b?

print(grad_L_w1)
print(grad_L_b)
```

```
(tensor([-0.0898]),)
(tensor([-0.0817]),)
```

`loss.backward()` fills `.grad` on every tensor that had `requires_grad=True`. Same slopes as `grad(...)` above, stored on the knobs themselves.

In [7]:
```
loss.backward()  # walk the same boxes backward; fills .grad
on every knob

print(w1.grad)  # same number as grad(...) above; negative →
turning w1 up lowers loss
print(b.grad)
```

```
tensor([-0.0898])
tensor([-0.0817])
```

Exercise (backward)

`w1.grad` printed as about `-0.09` (negative). If we **increase** `w1` a little, does the loss go **up** or **down**? One sentence.

In [1]:
```
# Exercise (backward) — your attempt
# w1.grad was about -0.09
# If we increase w1, loss goes: "up" or "down"?
answer = "..."
print(answer)
```

Solution (backward)

In [8]:
```
# Solution — a negative slope means: locally, bigger w1 → smaller loss.
# Downhill on the valley is toward increasing w1.
print('down')
print('negative .grad: increasing the knob lowers loss (here)')
```

```
down
negative .grad: increasing the knob lowers loss (here)
```

Comprehensive exercises

These two pull the first half of class together (penguins + next-word). Try the starter cell. The solution is the next cell.

Exercise 1. Ignore AutoGluon. Always guess the most common `species` in `train`. What percent of `test` penguins is that right? Compare to the TEST number from Demo 1.

In [1]:
```
# Exercise 1 — your attempt
majority = train['species'].mode()[0]
# fraction of test rows whose species equals majority
acc = ...
print('always guess', majority)
print('percent correct on test', acc)
```

Solution 1

In [9]:
```
# Solution 1 — mode() is the most common value; compare that name to the whole test column.
majority = train['species'].mode()[0]
acc = float((test['species'] == majority).mean())
print('always guess', majority)
print('percent correct on test', round(acc, 3))
```

```
always guess Adelie
percent correct on test 0.38
```

Exercise 2. `logits = np.array([0.0, 3.0, 0.2])`, true word id `1`. Print probabilities and loss. Why quote loss for next-word instead of percent exact?

In [1]:
```python
# Exercise 2 — your attempt
import numpy as np

logits = np.array([0.0, 3.0, 0.2])
true_id = 1
# softmax so the three numbers add to 1; loss = -
log(probs[true_id])
probs = ...
loss = ...
print('probs', probs)
print('loss', loss)
```

Solution 2

In [10]:
```python
# Solution 2 — softmax turns scores into probabilities; loss
is -log of the true word's probability.
import numpy as np

logits = np.array([0.0, 3.0, 0.2])
true_id = 1
probs = np.exp(logits - logits.max())
probs = probs / probs.sum()
loss = float(-np.log(probs[true_id]))
print('probs', np.round(probs, 3))
print('loss', round(loss, 3))
print('top guess id', int(probs.argmax()))
```

```
probs [0.045 0.9   0.055]
loss 0.105
top guess id 1
```

Takeaways

1. Classification → accuracy. Regression → MAE / MSE. Next-token → a huge classification; report **cross-entropy loss**, not percent exact.
2. No answer column: group similar rows. That is **unsupervised** clustering.
3. Split first. The number you believe is **test**. AutoGluon: name the label, `.fit(train)` , `.evaluate(test)` .
4. One **neuron** is a mixing desk: knobs (`weight` , `bias`), mix, **sigmoid** squish into 0–1. A **neural net** is a pile of those desks.
5. **Cross-entropy** is `-log` of the probability on the true answer. **Backprop** walks the same boxes backward; `.grad` says which way is downhill for that knob.
6. `*` is element-wise. `.matmul` and `@` (“at”) are **matrix multiply** (each entry is a **dot product** of a left-row and a right-column). Inner shapes must match.
7. Tuesday: Appendix **A.5–A.9** — `nn.Module` , DataLoader, the training loop (the small step along the arrow).