Lecture 15 — Machine Learning for Computer Vision

Lecture 15

Kaveh Fathian
Computer Science Department, Colorado School of Mines

Learning Objectives

By the end of this lecture, you should be able to:

- Name the main recognition tasks (classification, detection, semantic and instance segmentation) and explain why recognition is hard.
- Describe the data-driven approach, and distinguish supervised, unsupervised, and self-supervised learning.
- Explain underfitting, overfitting, and the bias–variance trade-off, and why more data helps.
- Use training, validation, and test sets to fit parameters, choose hyperparameters, and report a final error.
- Choose an evaluation metric that fits the task, evaluate a classifier with accuracy and the confusion matrix, and explain why accuracy can mislead when classes are imbalanced.
- Represent an image as a feature vector, compare images with the L2 distance, classify them with KNN, and explain the limitations of KNN on raw pixels.
- Cluster data with k-means, and use it to segment an image by color.
- Compute PCA, interpret its components and reconstructions, and use PCA coordinates as compact features.
- Explain linear classifiers, the SVM margin, soft margins, kernels, and multi-class SVMs, and train an SVM with OpenCV.

Fundamental Equations of Computer Vision

1. Image Filtering	$$y[m,n] = \sum_{k,l} h[k,l]\, I[m+k, n+l]$$
2. Optical Flow	$$I(x,y,t) = I(x+\Delta x,\ y+\Delta y,\ t+\Delta t)$$
3. Camera Geometry	$$\lambda\, \mathbf{x} = \mathbf{K} \begin{bmatrix} {}^{c}\mathbf{R}_{w} & {}^{c}\mathbf{t}_{w} \end{bmatrix} {}^{w}\mathbf{X} \qquad \mathbf{x}^{T}\mathbf{F}\mathbf{x}' = 0$$
4. Machine Learning	$$\arg\min_{\mathcal{S}} \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} \left\| \mathbf{x} - \boldsymbol{\mu}_i \right\|^2 \qquad y = \varphi\left(\mathbf{w}^{T}\mathbf{x} + b\right)$$

Recognition Tasks in Computer Vision

Classification



Detection



Semantic segmentation

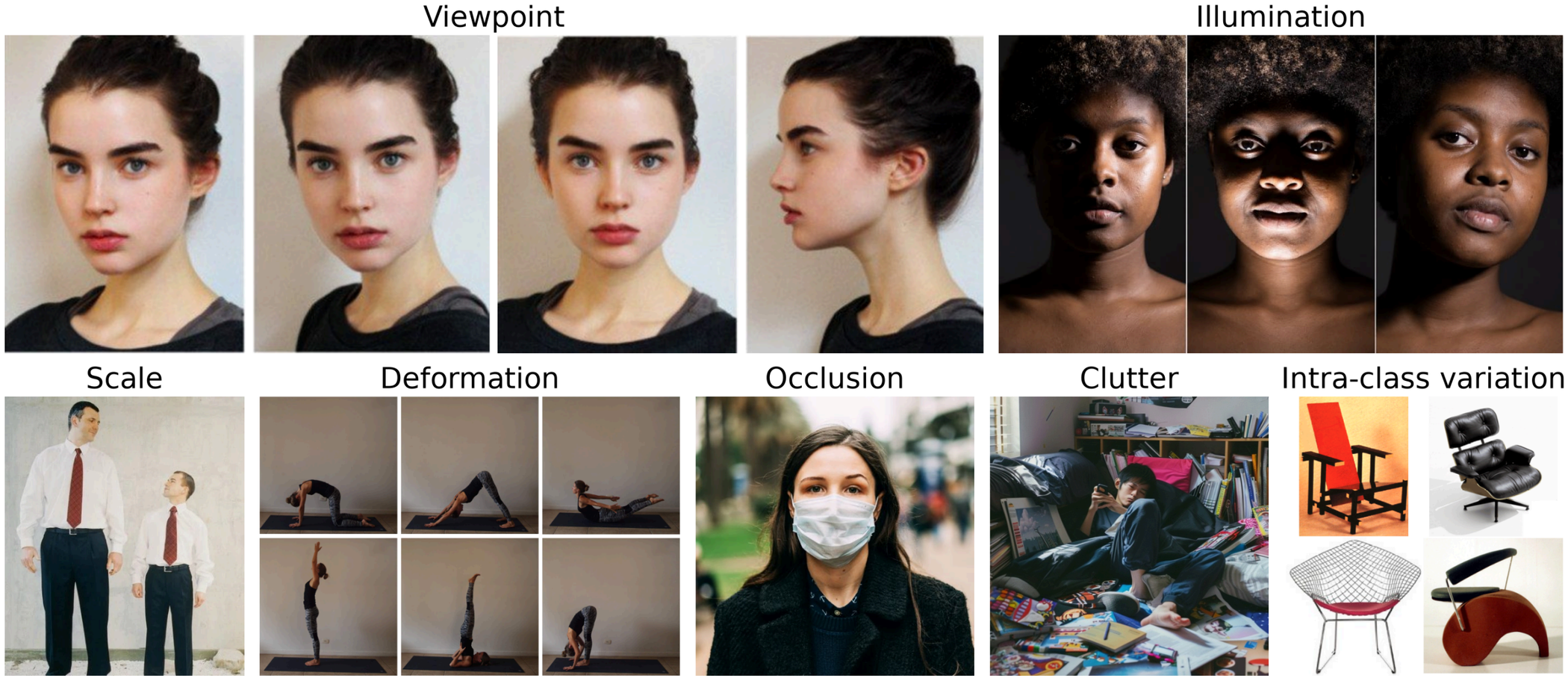


Instance segmentation



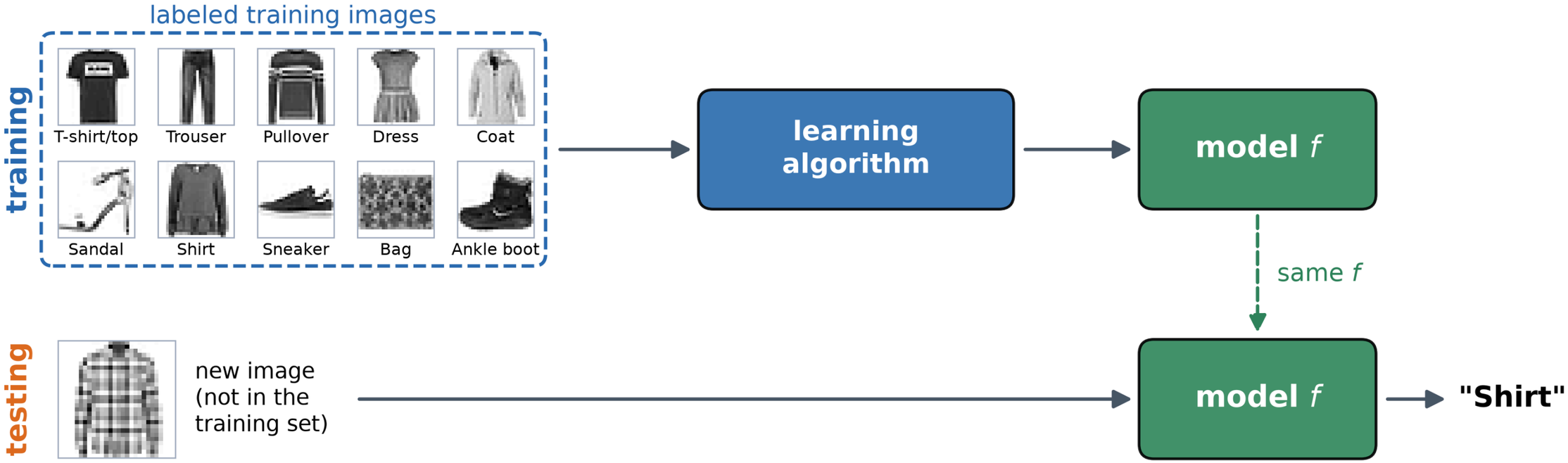
- **Classification:** is there a person? $\rightarrow$ one label for the image.
- **Detection:** where is each person? $\rightarrow$ a box per person.
- **Semantic segmentation:** which pixels are "person"? $\rightarrow$ a label per pixel.
- **Instance segmentation:** which pixels belong to which person? $\rightarrow$ a mask per person.
- Others: scene categorization, activity/event recognition, …

Why Is Recognition Hard?



- How many different object classes are there? Roughly 10,000 to 30,000.
- Objects of the same class can have very different pixels.

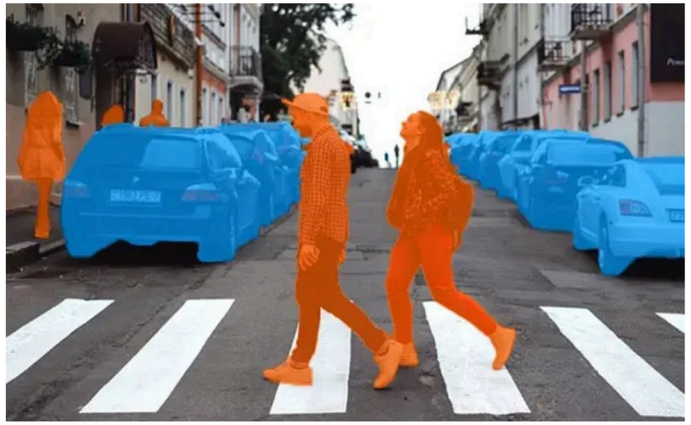
The Data-Driven Approach



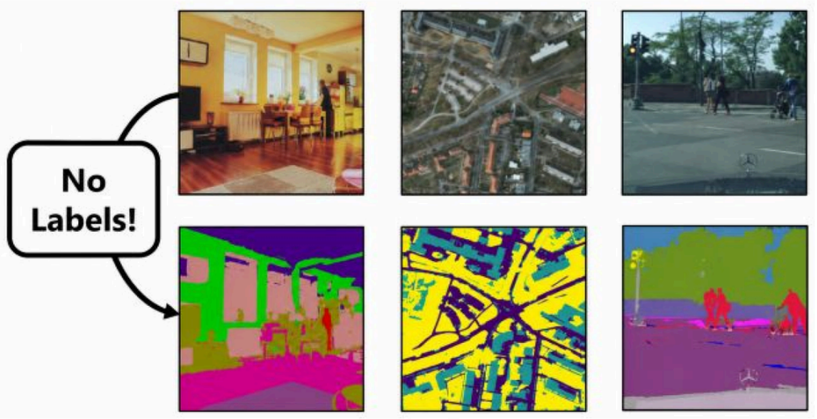
1. **Training set**: Images with ground truth labels.
2. **Learning**: A learning algorithm fits a **model** f to data: $f(\text{image}) \approx \text{label}$.
3. **Inference**: Use f on new images. What matters is how well f does on images it has never seen.

Types of Learning

supervised:
every pixel is labeled



unsupervised:
no labels, find groups



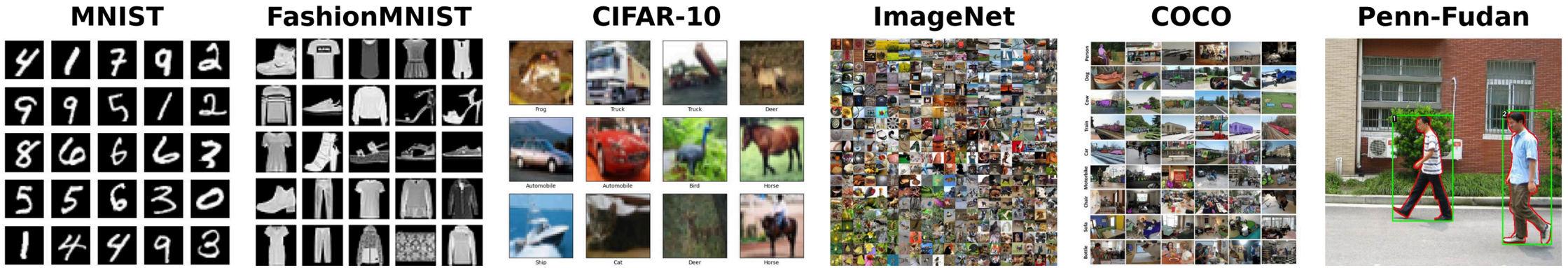
self-supervised:
predict the hidden part



Type	Training data	Vision examples
Supervised	images **and** labels	classify clothing; detect pedestrians
Unsupervised	images only	group pixels by color (clustering); compress images (PCA)
Self-supervised	images only; the labels come from the data itself	predict hidden patches; match images to their captions

- Labels cost effort, more so for richer labels: tag < box < mask.

Common Machine Learning Datasets for Computer Vision



Dataset	Year	Images	Classes	Image size	Task
MNIST	1998	60,000 train + 10,000 test	10 digits	28 × 28, gray	classification
FashionMNIST	2017	60,000 train + 10,000 test	10 clothing types	28 × 28, gray	classification
CIFAR-10	2009	50,000 train + 10,000 test	10 objects and animals	32 × 32, color	classification
ImageNet	2012	1.28 M train + 50,000 val	1,000	varies, color	classification
COCO	2014	118,000 train + 5,000 val	80	varies, color	detection, segmentation, captions
Penn-Fudan	2007	170	pedestrians	varies, color	detection, segmentation

- These are **benchmarks** that are widely used to compare methods on the same data.
- ImageNet's size made deep learning practical (2012).

FashionMNIST: Our Running Example

	Images	Per class
Training	60,000	6,000
Test	10,000	1,000

- **28 × 28** gray images: 784 pixels, values 0 to 255.
- Same format as MNIST, but harder: T-shirt, pullover, coat, and shirt look alike.
- Small enough to train on a laptop CPU.
- We compare our classifiers on this benchmark.



A Toy Learning Problem

- **Task:** predict y from x, using only the noisy points.
- **Model:** a polynomial of degree M:

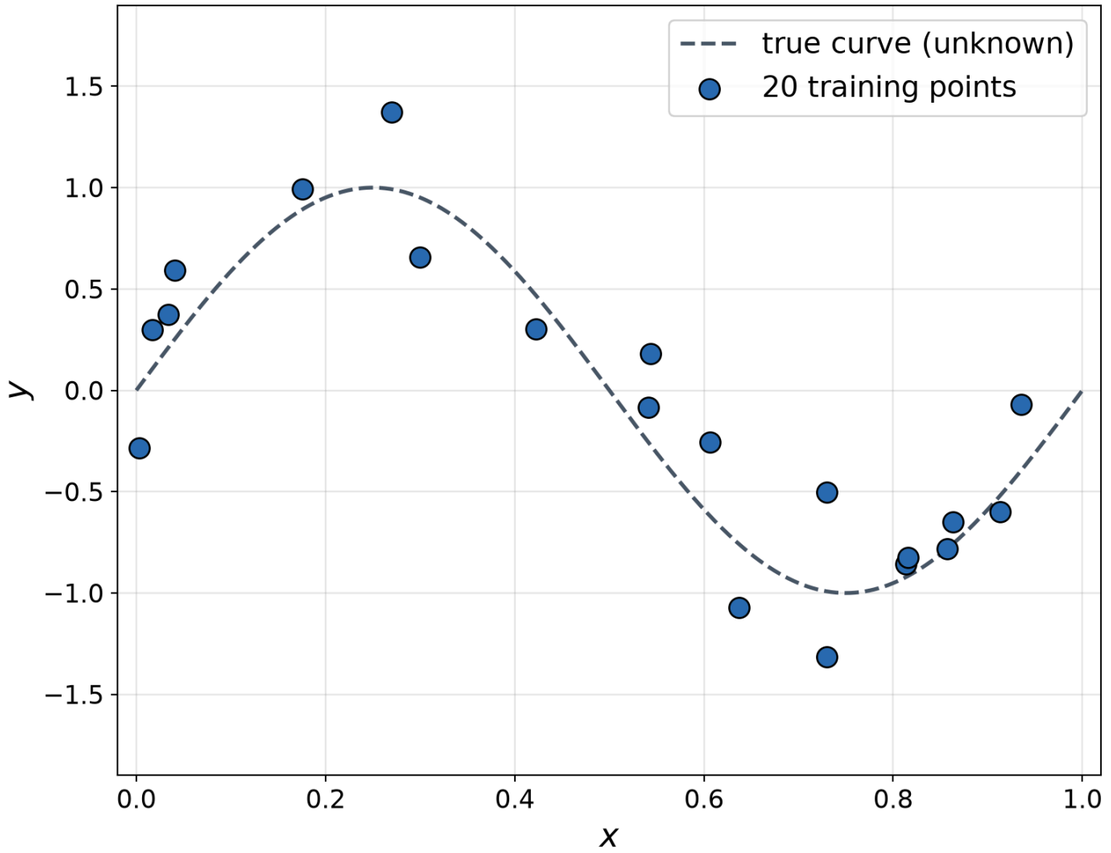
$$\hat{y} = w_0 + w_1 x + w_2 x^2 + \cdots + w_M x^M$$

polynomial model

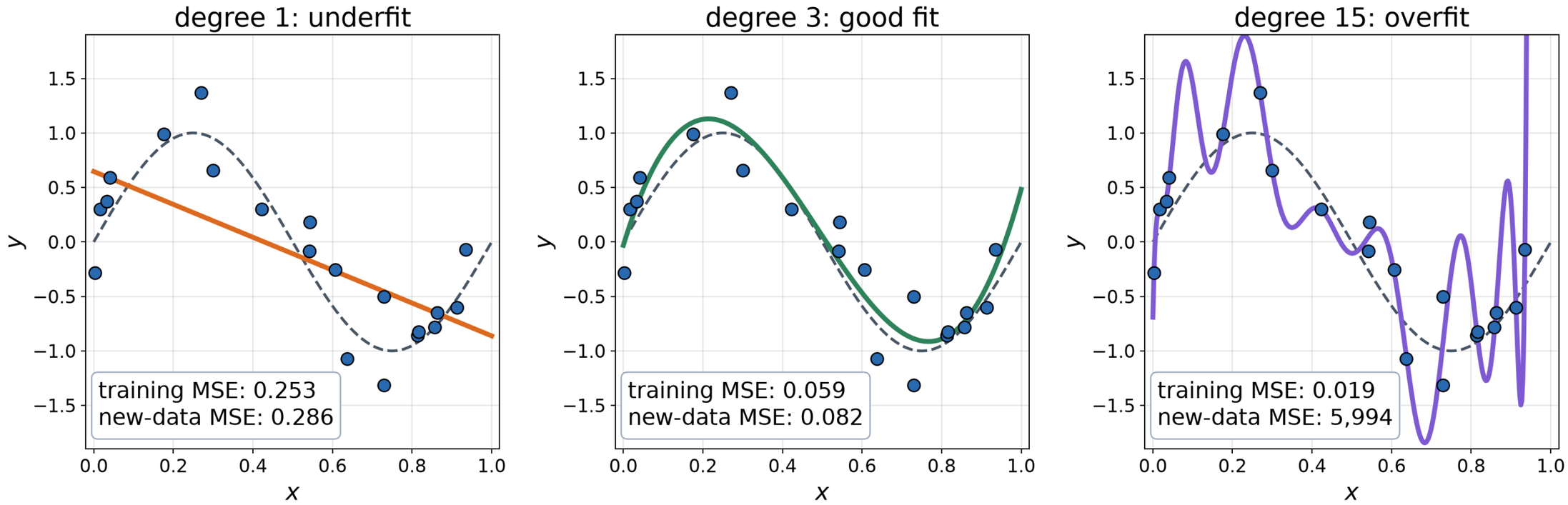
- x: input; $\hat{y}$: prediction
- $w_0, \ldots, w_M$: **parameters** learned from the data.
- **Learning:** pick the w's with the smallest **mean squared error** (MSE) on the training points:

$$\text{MSE} = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

- y_i: true value; $\hat{y}_i$: prediction for point i; N: number of points.



Model Complexity: Underfitting vs. Overfitting

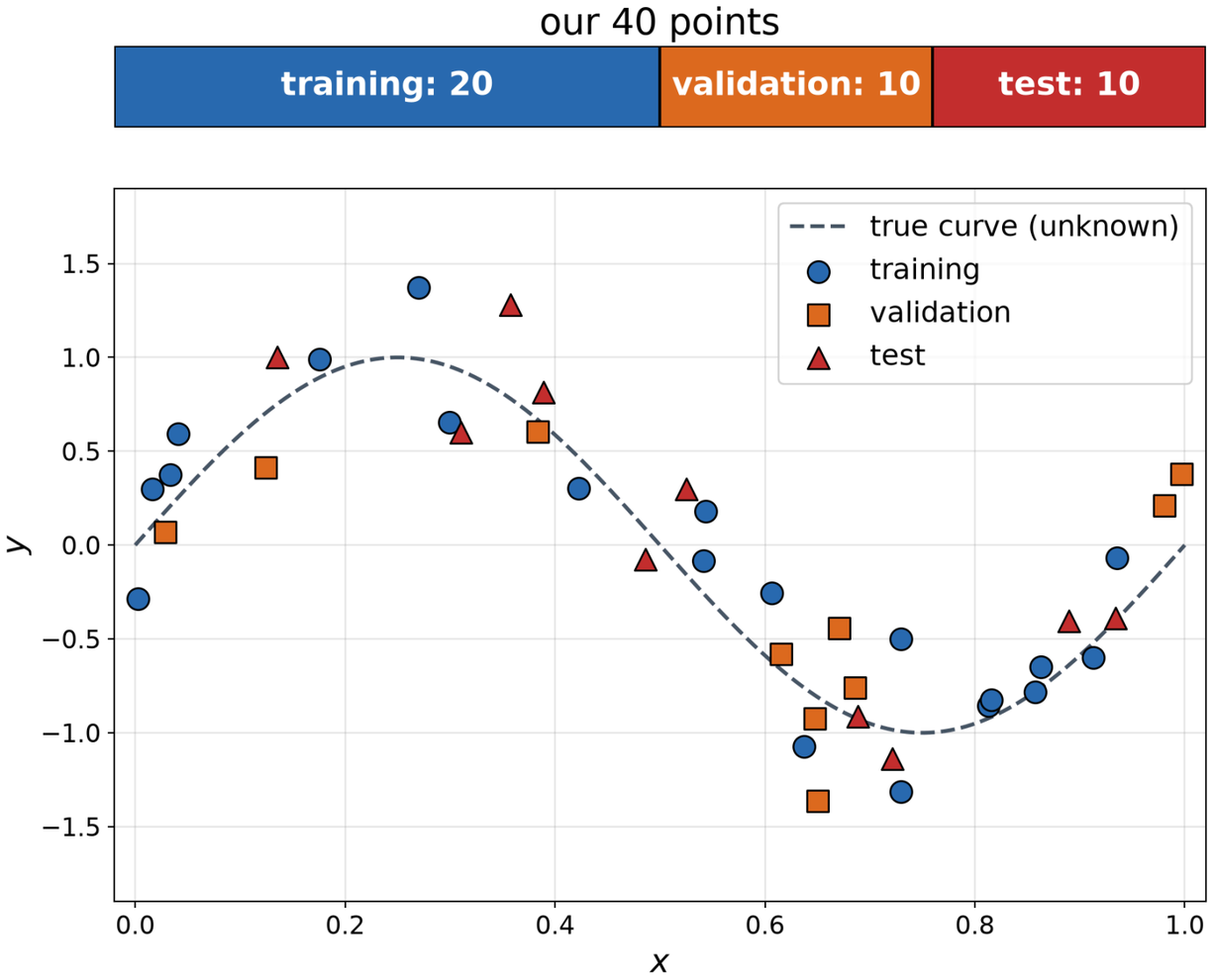


New data: 5,000 fresh points from the same curve and noise, never used for fitting.

- **Underfit:** too simple to follow the pattern $\rightarrow$ high error everywhere.
- **Overfit:** so flexible that it fits the **noise** $\rightarrow$ lowest training error, huge error on new data.
- The goal is **generalization**: low error on data the model has **not** seen. Training error alone does not tell us that.

Train / Validation / Test

- **Training set:** fit the **parameters** $(w_0, \ldots, w_M)$.
- **Validation set:** choose the **hyperparameters**: settings we pick, not learned (here, the degree M).
- **Test set:** measure the final error **once**, at the end. Never use it to make choices.

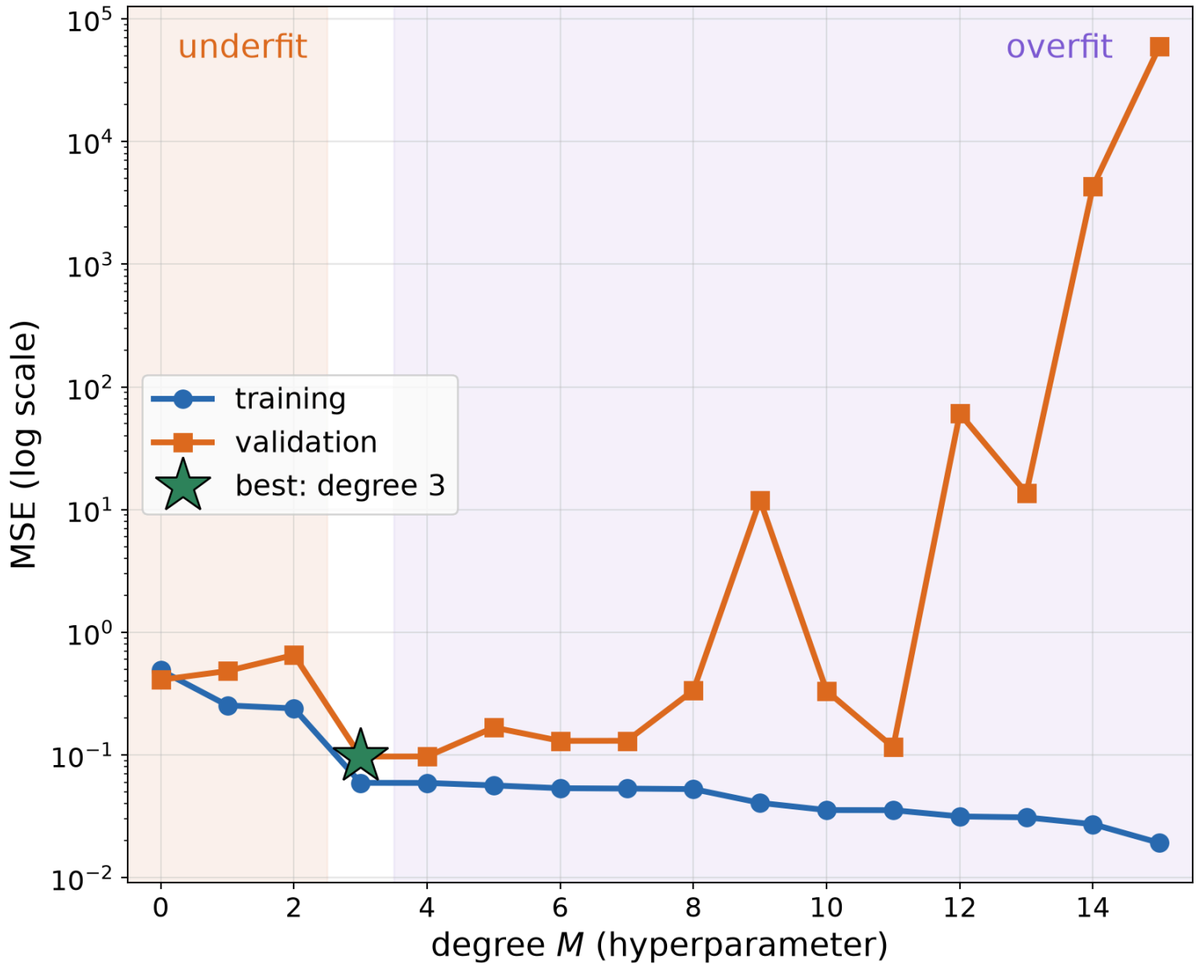


Choosing the Degree

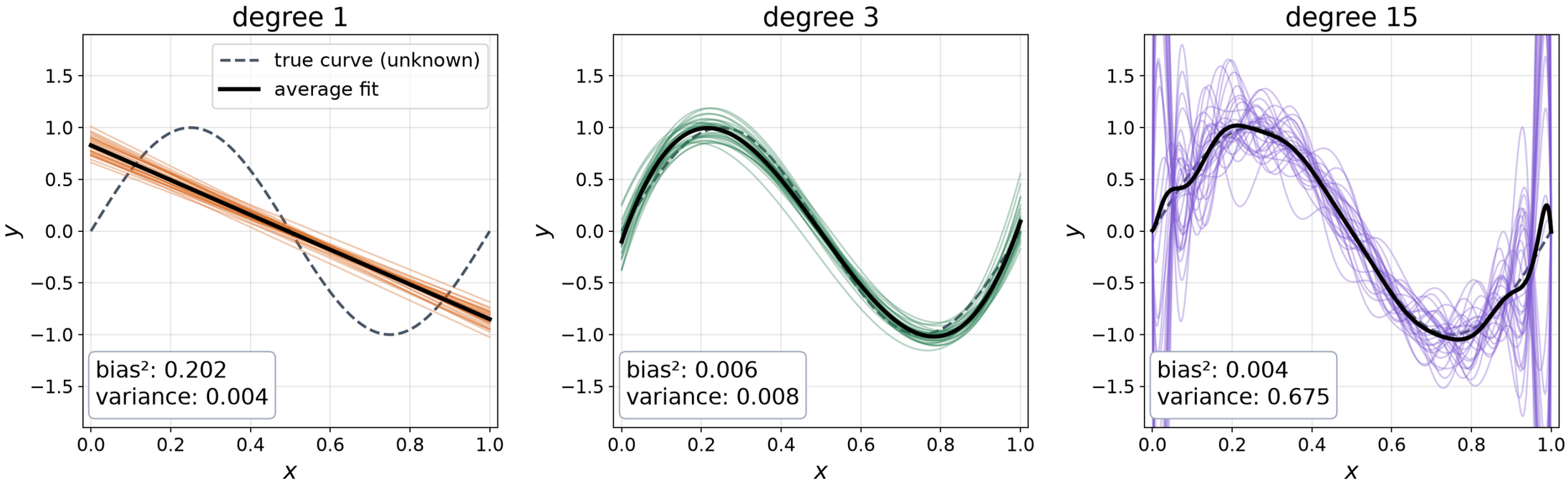
Choosing a Hyperparameter

1. For each candidate M: fit on the **training** set.
2. Measure the error on the **validation** set.
3. Keep the M with the lowest validation error.
4. Report the **test** error of that model, once.

- Training error keeps falling as M grows. Validation error falls, then rises.
- Best: M = **3** (validation MSE 0.097). Its test MSE: **0.072**.

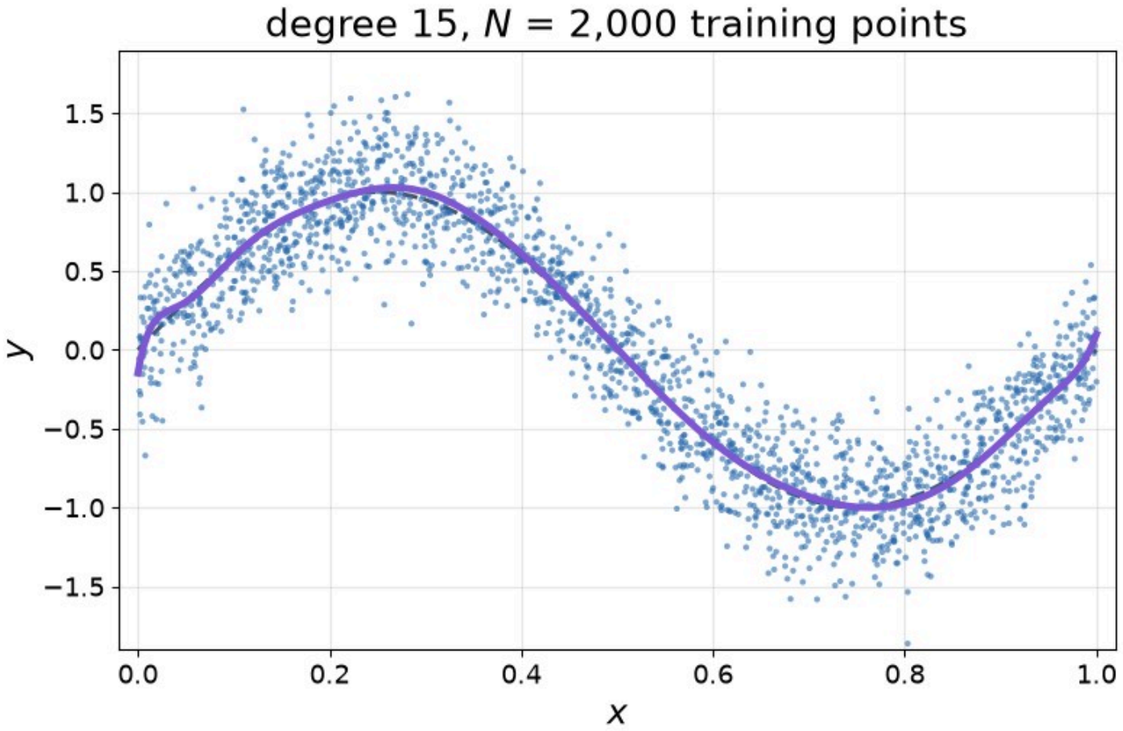


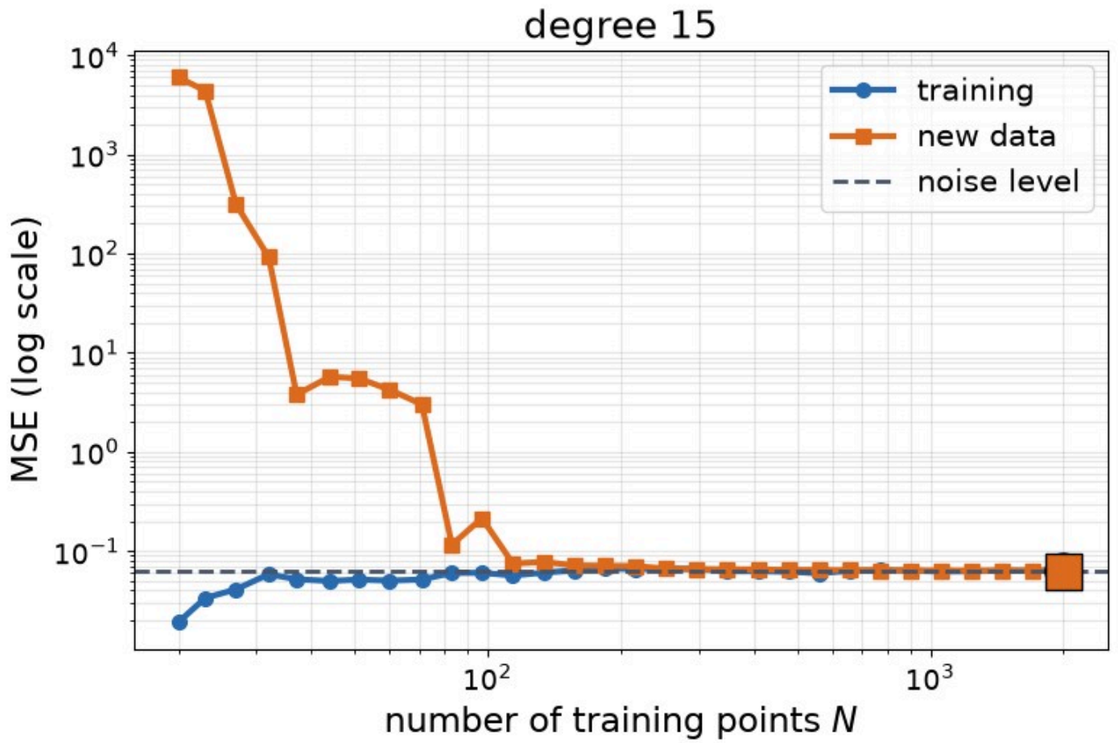
Bias–Variance Trade-Off



- Each thin line: a fit to a **different training set** (same x, new noise). Black: their average.
- **Bias:** the average fit misses the truth $\rightarrow$ the model is **too simple**.
- **Variance:** the fit changes a lot from one training set to the next $\rightarrow$ the model is **too flexible**.
- New-data error $\approx$ bias^2 + variance + noise. The best complexity balances the two.

More Data Helps



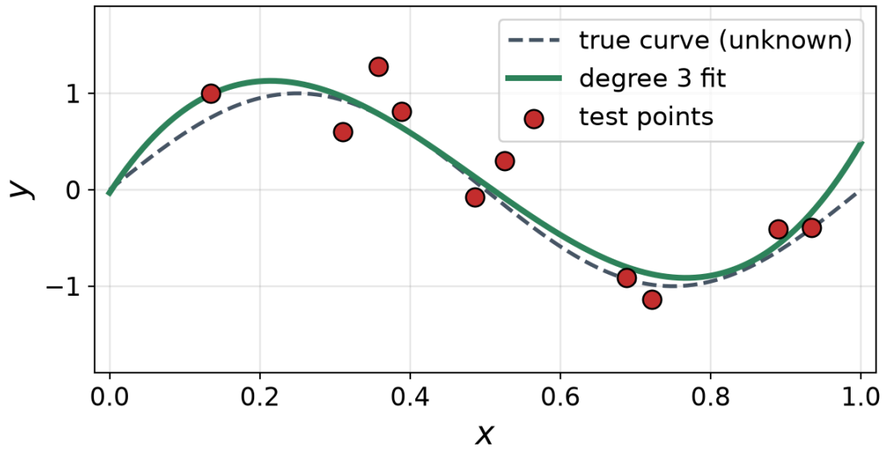


- With few points, the flexible model fits the noise. With many, it **cannot** fit the noise, so it follows the true curve.
- Training and new-data errors meet at the noise level, which no model can beat.
- **More data** is often the best cure for overfitting.

Polynomial Fitting in Code

- `Polynomial.fit(x, y, deg)`: finds the w's with the smallest training MSE (least squares).
- `model(x)`: evaluates the fitted polynomial at `x`.
- `mse(...)`: the mean squared error from before.
- `min(val_mse, key=val_mse.get)`: the degree with the lowest validation error.

On our data: best M = **3**, test MSE = **0.072**.




```python
import numpy as np
from numpy.polynomial import Polynomial

def mse(model, x, y):
    return np.mean((model(x) - y) ** 2)

# (x_tr, y_tr), (x_val, y_val), (x_te, y_te): splits
val_mse = {}
for M in range(16):
    model = Polynomial.fit(x_tr, y_tr, deg=M)
    val_mse[M] = mse(model, x_val, y_val)

best_M = min(val_mse, key=val_mse.get)
model = Polynomial.fit(x_tr, y_tr, deg=best_M)
print(best_M, mse(model, x_te, y_te))   # test: once
```

Polynomial Fitting: Full Code

```python
import numpy as np
import matplotlib.pyplot as plt
from numpy.polynomial import Polynomial

# 1. Data: noisy samples of y = sin(2 pi x). Split 20 / 10 / 10.
rng = np.random.default_rng(0)
x = rng.uniform(0, 1, 40)
y = np.sin(2 * np.pi * x) + rng.normal(0, 0.25, 40)
x_tr, y_tr = x[:20], y[:20]
x_val, y_val = x[20:30], y[20:30]
x_te, y_te = x[30:], y[30:]

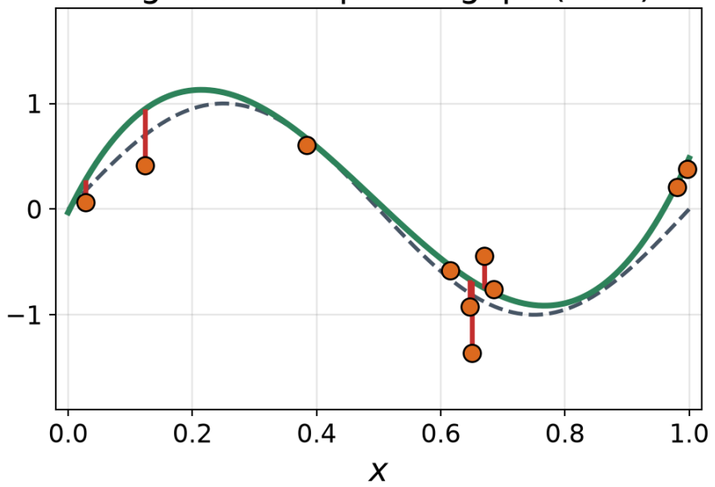
def mse(model, x, y):
    return np.mean((model(x) - y) ** 2)

# 2. Fit every degree on the training set; measure training and validation error.
degrees = range(16)
train_mse, val_mse = [], []
for M in degrees:
    model = Polynomial.fit(x_tr, y_tr, deg=M)
    train_mse.append(mse(model, x_tr, y_tr))
    val_mse.append(mse(model, x_val, y_val))

# 3. Choose the degree on the validation set; report the test error once.
best_M = int(np.argmin(val_mse))
model = Polynomial.fit(x_tr, y_tr, deg=best_M)
print("best degree:", best_M)
```

Evaluation Metrics Depend on the Task

regression: squared gaps (MSE)



classification: fraction correct

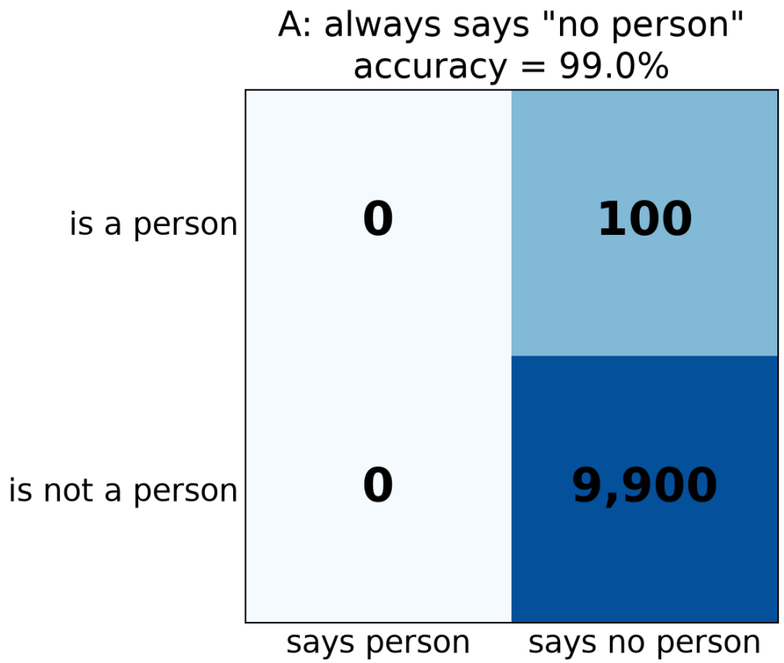
$$\text{accuracy} = \frac{\#\,\text{correct}}{\#\,\text{images}}$$

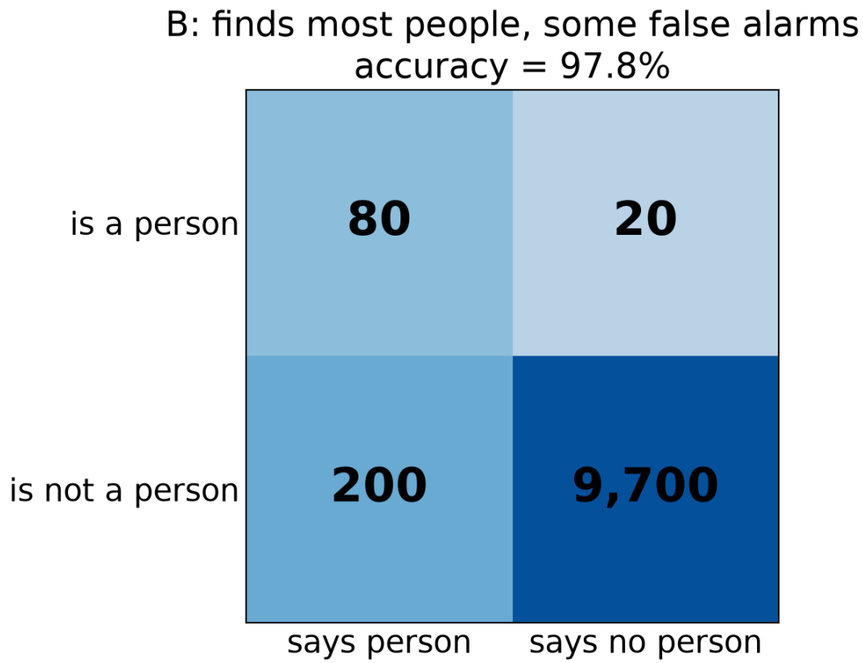
detection: box overlap, IoU = 0.41



Task	Output	Common metrics
Regression	number(s)	mean squared error (MSE)
Classification	a category	accuracy; confusion matrix (which classes get mixed up)
Detection	a box and class per object	intersection over union (IoU); precision and recall; average precision (AP)
Segmentation	a label per pixel	IoU per class

When Accuracy Misleads





- 10,000 images, only 1% contain a person. Method **A** finds nobody, yet has a higher accuracy than method **B** that finds most people (99.0% vs. 97.8%).
- Need to look at both precision and recall. **Precision**: fraction of correct detections (A: undefined (no detections), B: 29%). **Recall**: fraction of people found (A: 0%, B: 80%).
- Accuracy is fine when the classes are **balanced** (as in FashionMNIST).

Supervised Learning: KNN

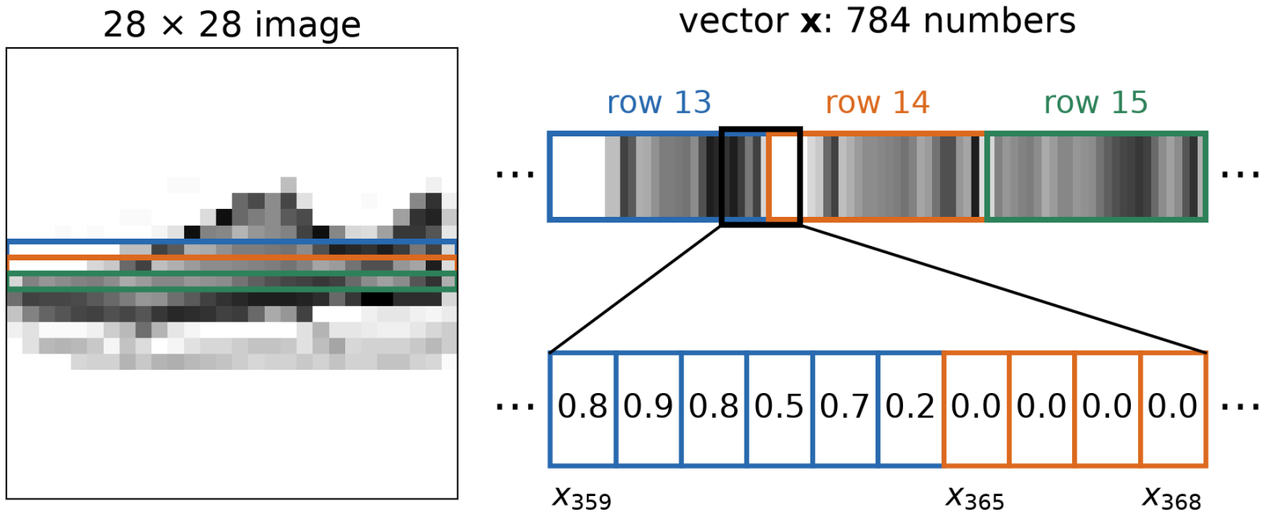
An Image Is a Vector

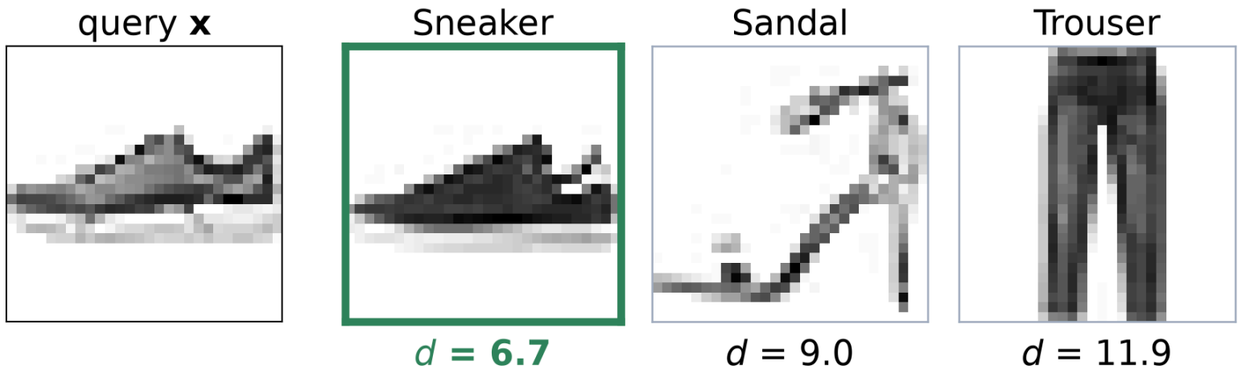
- A 28 × 28 gray image is **784 numbers**. Stack its rows into one long vector $\mathbf{x} = (x_1, x_2, \ldots, x_{784})$.
- Compare two images pixel by pixel:

$$d(\mathbf{x}, \mathbf{x}') = \sqrt{\sum_{j=1}^{784} \left(x_j - x'_j\right)^2}$$

L2 distance

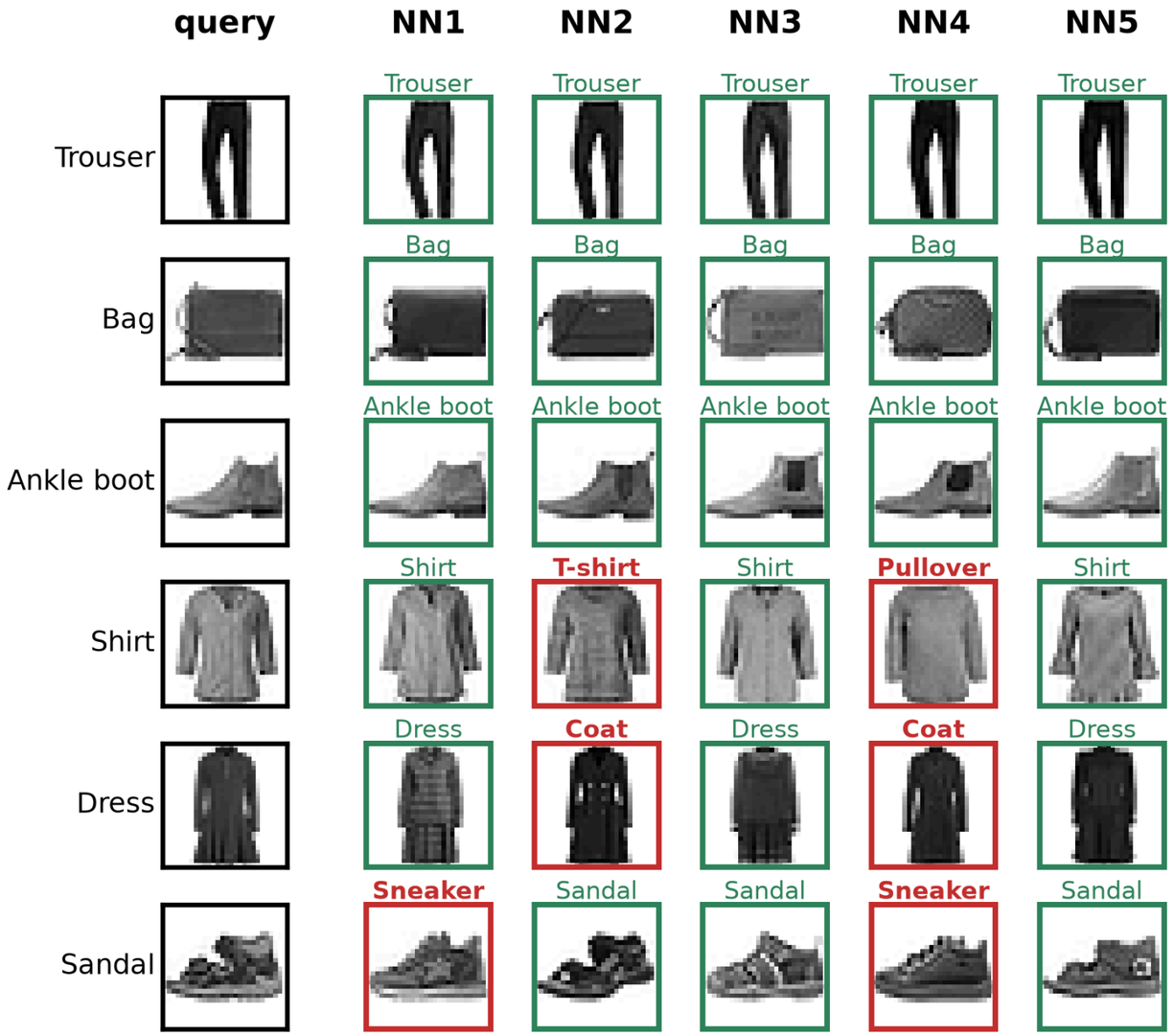
- x_j, x'_j: value of pixel j (0 to 1) in images $\mathbf{x}$ and $\mathbf{x}'$.
- Small d: similar images. Large d: different images.
- The sneaker is closest to the other sneaker (d = 6.7) and farthest from the trouser (d = 11.9).





Examples of Nearest Neighbors on FashionMNIST

- **Query** (left column): a test image.
- **Neighbors:** the 5 training images with the smallest L2 distance to the query, from NN1 (closest) to NN5.
- Green frame: same class as the query. Red frame: a different class.
- Mistakes come from look-alike classes: the shirt has T-shirt and pullover neighbors, and the dress has coat neighbors.
- The sandal's closest neighbor is a sneaker, but 3 of its 5 neighbors are sandals.
- Simple idea: **classify a new image based on the majority class among its k nearest neighbors.**



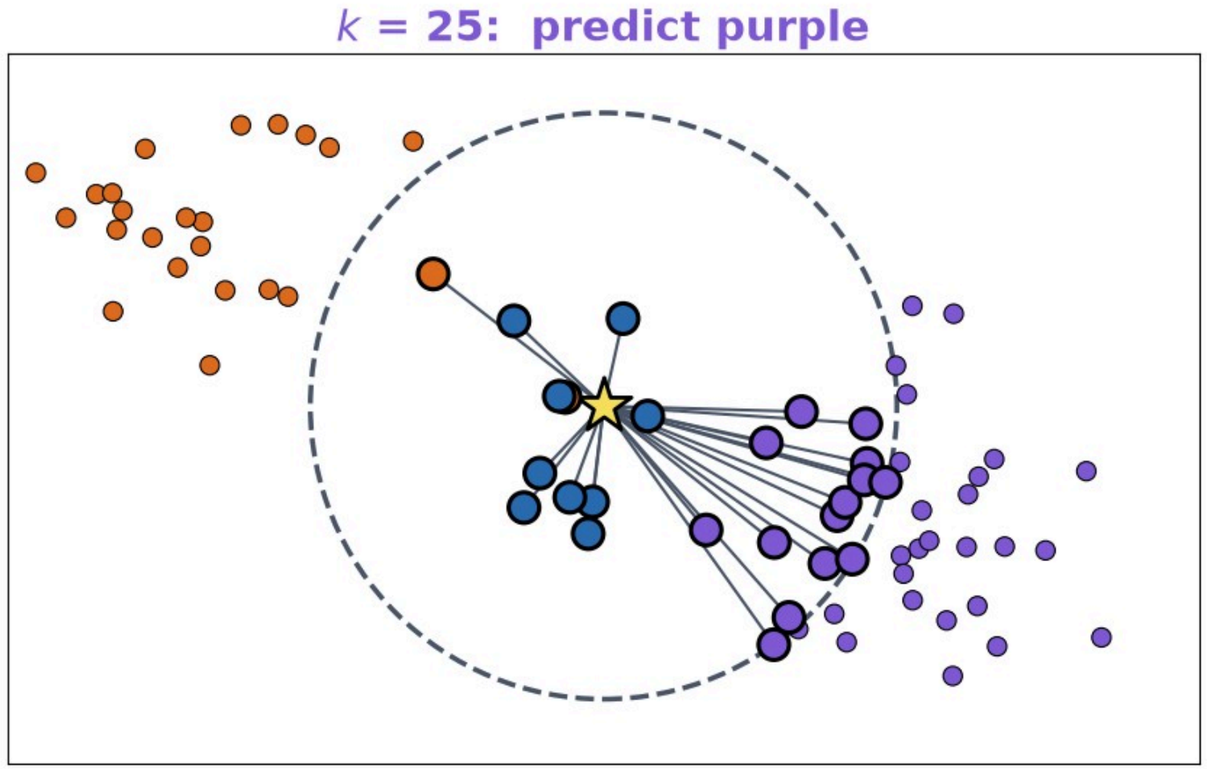
k-Nearest Neighbors (KNN) Classification

k-Nearest Neighbors (KNN)

Training: store all training images and their labels.

Prediction for a new image $\mathbf{x}$:

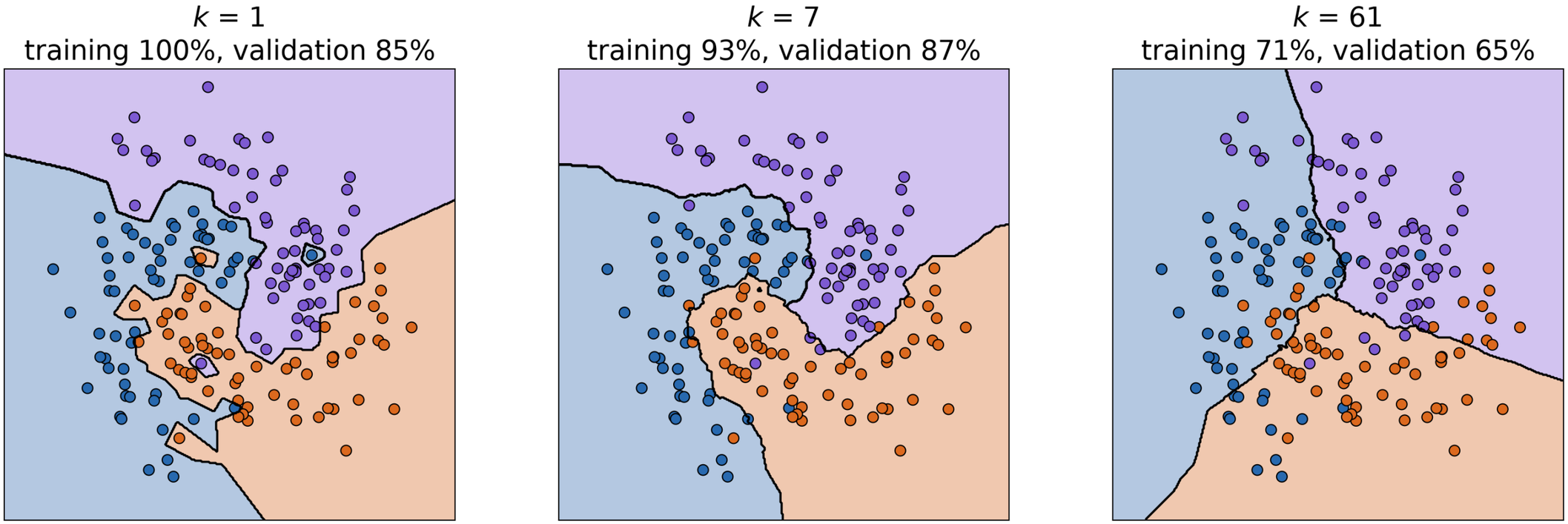
1. Compute the distance from $\mathbf{x}$ to every training image.
2. Find the k closest training images.
3. Predict the most common label among them (**majority vote**).





- 2D points represent images: each point is a vector with 2 numbers instead of 784.
- **Small k:** one noisy neighbor decides.
- **Large k:** far-away points join, and the big class wins.

Decision Boundaries Change with k

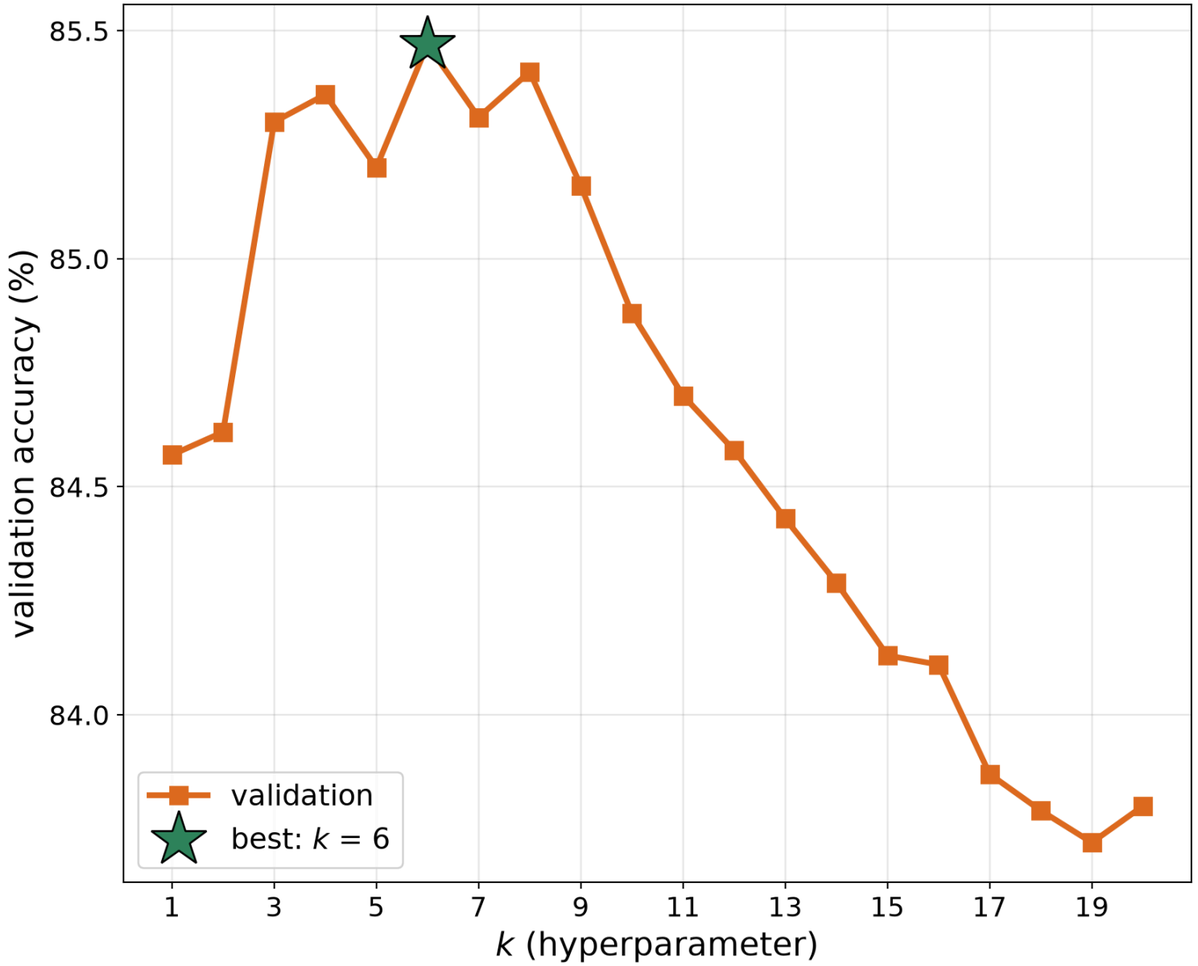


- **Decision boundary:** where the predicted class changes. Each location is colored by its KNN prediction.
- **Small k:** jagged boundary with islands around single points. It fits the noise (**overfitting**).
- **Large k:** smooth boundary, but the arms of the spiral get washed out (**underfitting**).
- k is a **hyperparameter**: choose it on validation data, like the polynomial degree.

Choosing k on FashionMNIST

- Split the 60,000 training images: **50,000 training** + **10,000 validation**. Keep the 10,000 **test** images aside.
- Same recipe as choosing the polynomial degree: for each $k = 1, \ldots, 20$, predict the validation images with KNN and measure the accuracy.
- Best: $k = \mathbf{6}$ (validation accuracy 85.5%).
- Test accuracy with $k = 6$, measured once: **85.0%**.





KNN in Code

- `KNearest_create()`: an empty KNN model.
- `train(X, cv2.ml.ROW_SAMPLE, y)`: stores the data. `X`: one image per row (`float32`); `y`: the labels.
- `findNearest(X, k)`: KNN prediction for each row of `X`. Returns the predicted labels, the labels of the `k` neighbors, and their distances.

- On our data ($k = 6$): test accuracy **85.0%**, about **1 ms** per test image.

```python
import cv2
import numpy as np

# X_tr: (50000, 784) float32, y_tr: labels 0-9
# X_te: (10000, 784) float32, y_te: labels 0-9
knn = cv2.ml.KNearest_create()
knn.train(X_tr, cv2.ml.ROW_SAMPLE,
          y_tr.astype(np.float32))

# best_k: chosen on the validation set
_, pred, neighbors, dist = knn.findNearest(
    X_te, k=best_k)
pred = pred.ravel().astype(int)
print("test accuracy:", np.mean(pred == y_te))
```

KNN predictions on test images (green: correct, red: wrong)



KNN: Full Code

```python
import gzip
from pathlib import Path

import cv2
import numpy as np
import matplotlib.pyplot as plt

# 1. Data: FashionMNIST as 784-vectors. Split 50,000 training / 10,000 validation; 10,000 test.
data_dir = Path("assets/datasets/fashion_mnist")    # folder with the four .gz files

def read_idx(name, offset):
    with gzip.open(data_dir / name) as f:
        return np.frombuffer(f.read(), np.uint8, offset=offset)

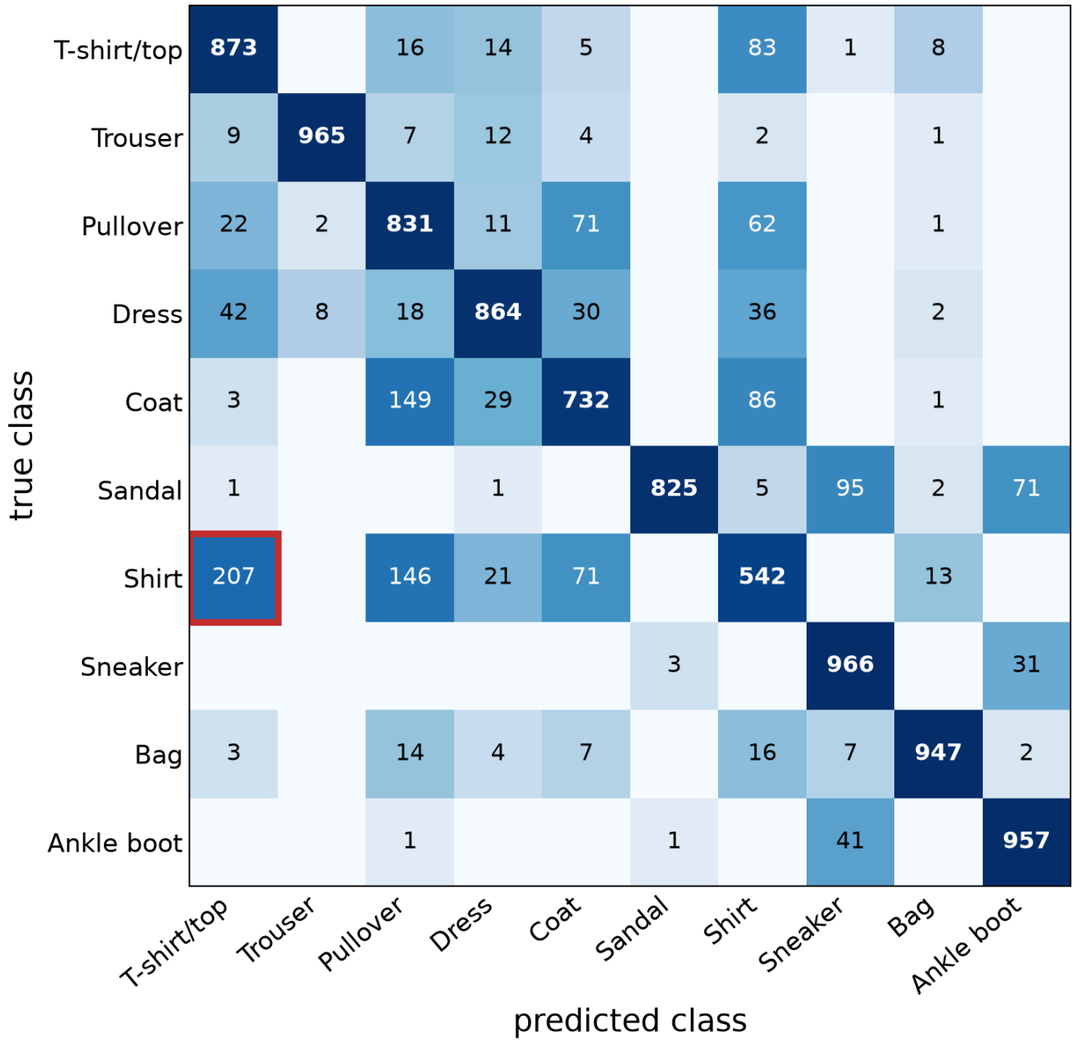
X = read_idx("train-images-idx3-ubyte.gz", 16).reshape(-1, 784).astype(np.float32)
y = read_idx("train-labels-idx1-ubyte.gz", 8).astype(int)
X_te = read_idx("t10k-images-idx3-ubyte.gz", 16).reshape(-1, 784).astype(np.float32)
y_te = read_idx("t10k-labels-idx1-ubyte.gz", 8).astype(int)
X_tr, y_tr, X_val, y_val = X[:50000], y[:50000], X[50000:], y[50000:]

# 2. "Training": store the training images.
knn = cv2.ml.KNearest_create()
knn.train(X_tr, cv2.ml.ROW_SAMPLE, y_tr.astype(np.float32))

# 3. Choose k on the validation set. One search for the 20 nearest neighbors
#    (closest first), then a majority vote among the first k of them.
ks = range(1, 21)
```

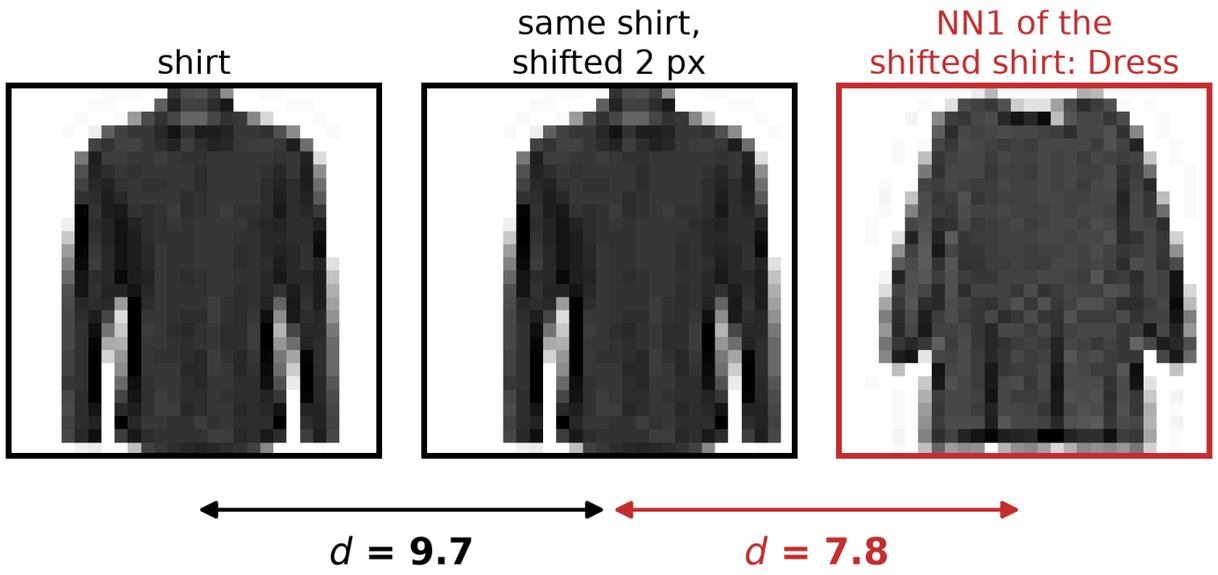
Confusion Matrix and Accuracy

- **Confusion matrix:** entry (row i, column j) = number of test images of class i predicted as class j.
- **Diagonal:** correct predictions. Off-diagonal: mistakes, and which classes get mixed up.
- $\text{accuracy} = \dfrac{\text{sum of the diagonal}}{\text{sum of all entries}}$
- KNN ($k = 6$): 8,502 / 10,000 correct = **85.0%**.
- Easiest: Sneaker (97% correct). Hardest: **Shirt** (54%), most often called T-shirt/top (207 times).



Limitations of KNN

- **Slow prediction:** compares each new image with all 50,000 training images (about 1.4 ms per new image on a CPU, for all 50,000 comparisons).
- **Stores all the data:** 50,000 × 784 values = 157 MB. More data: more memory and slower predictions.
- **Pixel distance is not semantic:** a 2-pixel shift is invisible to us, but changes many pixels.
  - The shifted shirt is **9.7** away from the original, but a dress is only **7.8** away.
  - Shifting all test images by 2 pixels drops the accuracy from 85.0% to **67.0%**.
- Next: better features and better classifiers.



the 6 nearest neighbors of the shifted shirt: KNN says Bag



Unsupervised Learning: Clustering and PCA

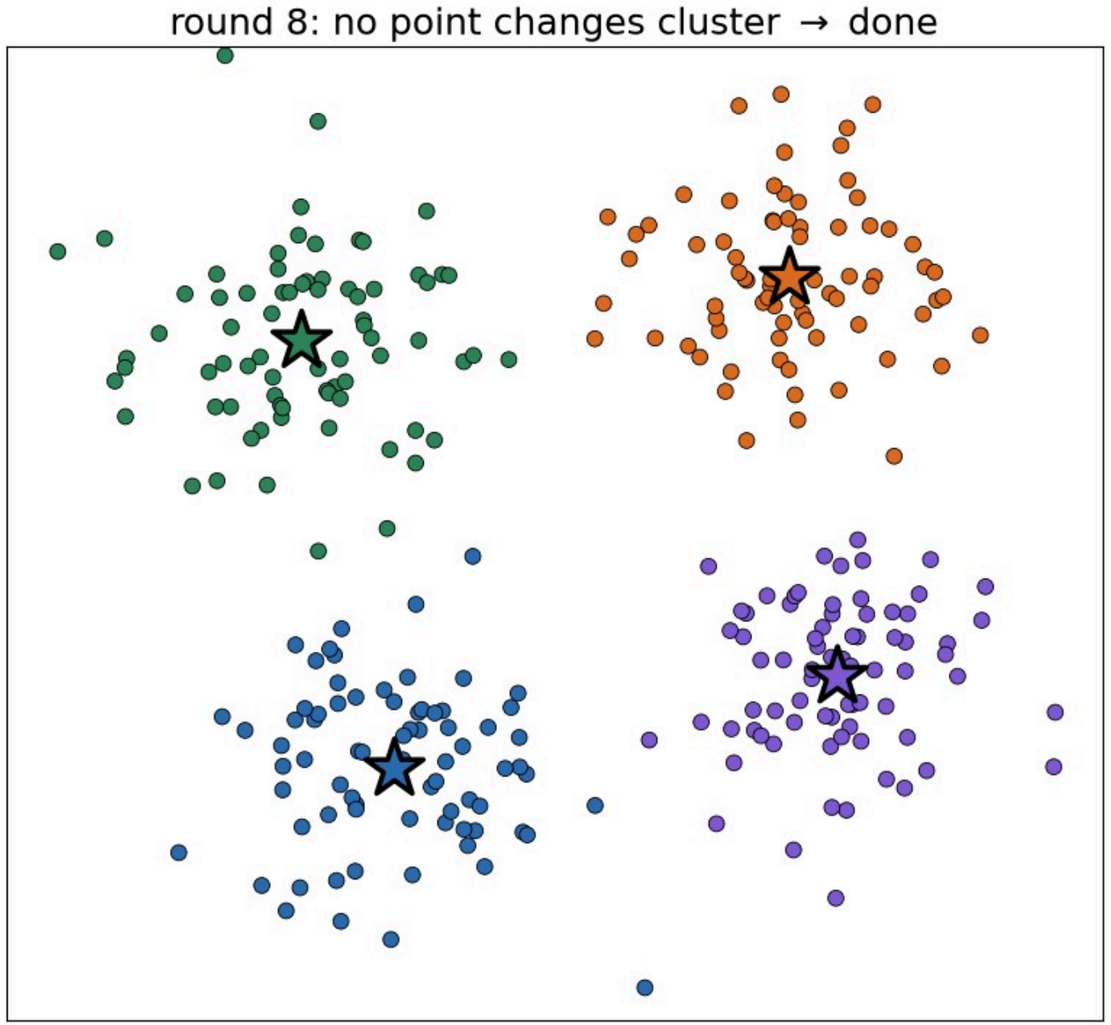
k-Means Clustering

- **Unsupervised:** points without labels. Goal: group them into k **clusters** of nearby points.
- Each cluster has a **center**. Each point belongs to the nearest center.

k-Means Clustering

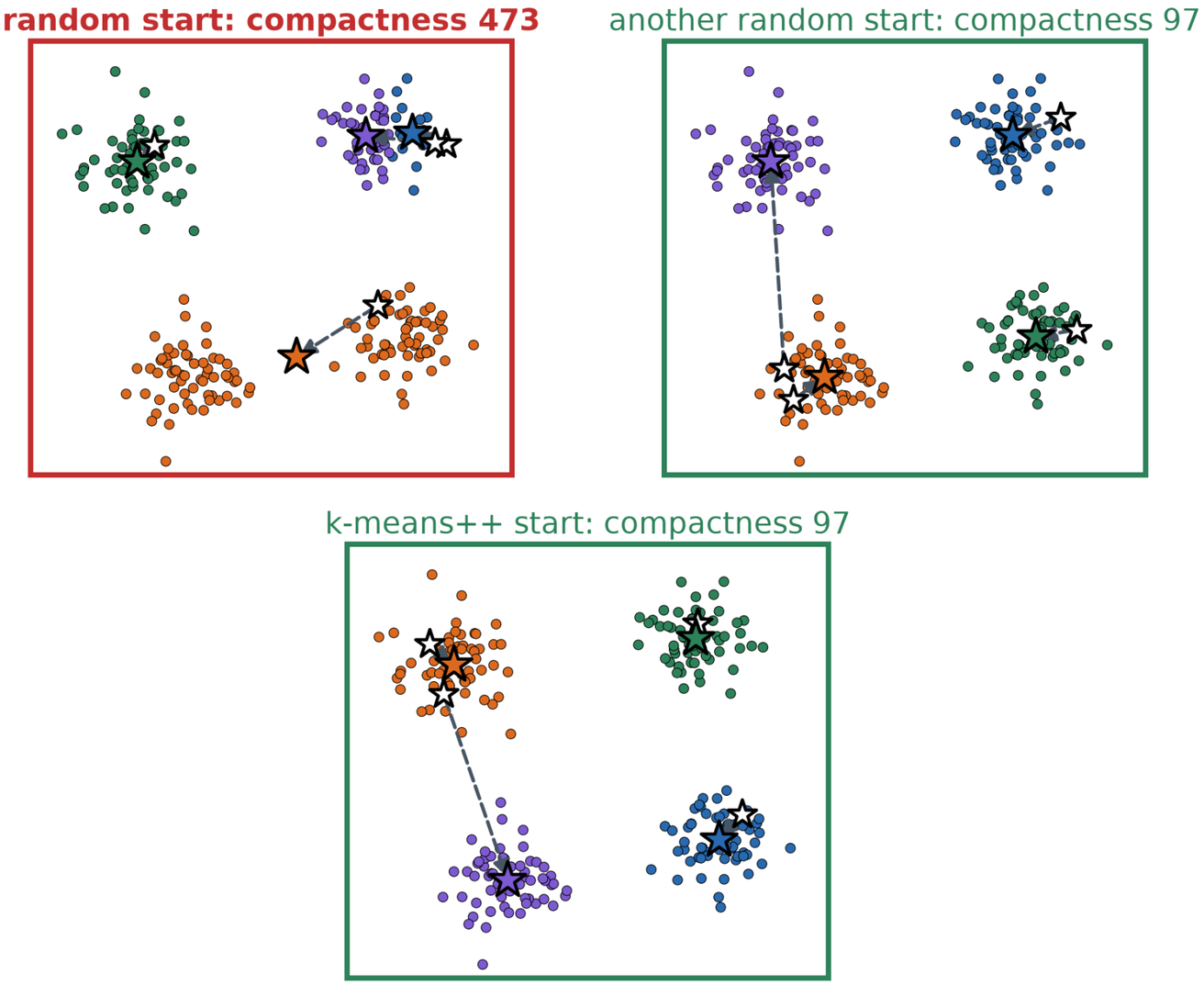
1. Pick k starting centers (e.g., k random points).
2. **Assign:** each point joins its nearest center.
3. **Update:** move each center to the mean of its points.
4. Repeat 2–3 until the assignments stop changing.

- k is a hyperparameter.
- Different starting centers can lead to different final clusters.



Starting Centers Matter

- A bad start can **get stuck** in a bad result (top left: two blobs share one center; white stars: starting centers; colored stars: final centers).
- **Compactness:** sum of squared distances from each point to its center. Lower is better.
- **Several attempts:** run from different starts; keep the lowest compactness.
- **k-means++:** first center at random; each next center **far from the chosen ones**.



Color Segmentation with k-Means

- Each pixel is a point (R, G, B) in 3D color space.
- k-means with k clusters groups the pixels into k groups, each represented by a color (the center).
- Pixels of similar color land in one cluster, even on different objects.
- No labels needed: the colors group by themselves.

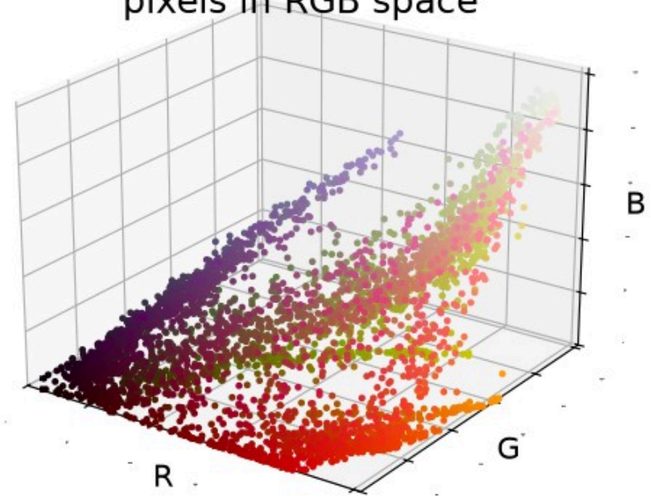
photo



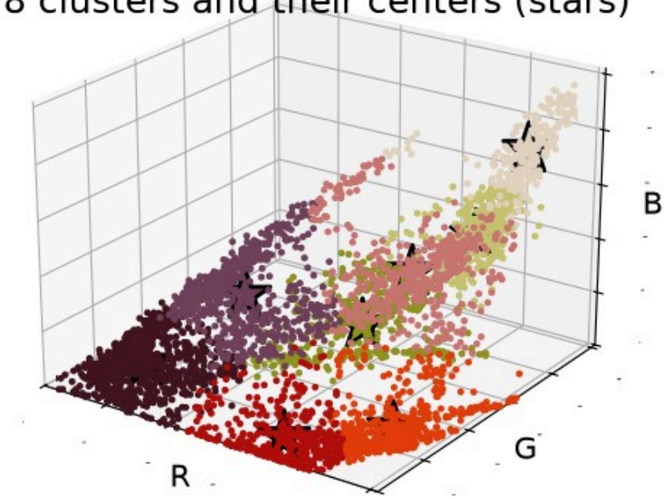
each pixel → its center color ($k = 8$)



pixels in RGB space



8 clusters and their centers (stars)



k-Means in Code

- `cv2.kmeans(data, K, bestLabels, criteria, attempts, flags)`:
  - `data`: one point per row (`float32`).
  - `K`: the number of clusters k.
  - `bestLabels`: starting labels (`None`: not given).
  - `criteria`: stop after 20 rounds or when the centers move < 1.0.
  - `attempts`: 5 runs from different starts; keep the lowest compactness.
  - `flags`: starting centers.
    `KMEANS_PP_CENTERS`: k-means++;
    `KMEANS_RANDOM_CENTERS`: random points.
- Returns `compactness`, `labels` (cluster of each point), `centers` ($k \times 3$).
- `centers[labels]`: each pixel → its center color.

```python
import cv2
import numpy as np

image = cv2.imread("fruits.jpg")        # H x W x 3 (BGR)
pixels = image.reshape(-1, 3).astype(np.float32)

stop = cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER
criteria = (stop, 20, 1.0)   # 20 rounds or moves < 1.0
compactness, labels, centers = cv2.kmeans(
    pixels, 4, None, criteria, 5, cv2.KMEANS_PP_CENTERS)

segmented = centers[labels.ravel()].reshape(image.shape)
cv2.imwrite("fruits_k4.png", segmented.astype(np.uint8))
```

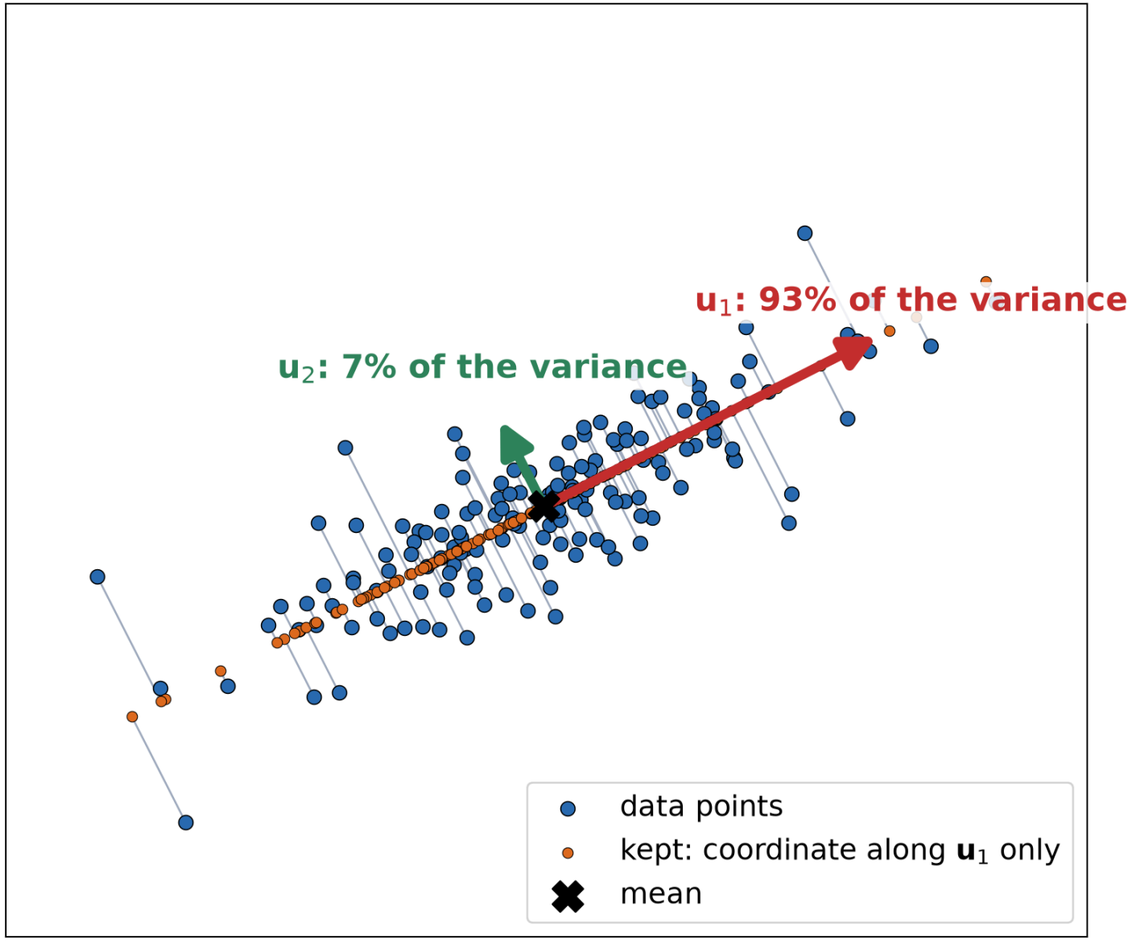
PCA: The Directions of Largest Variance

- **PCA** (principal component analysis): find the directions in which the data varies the most.

PCA

1. Subtract the mean from every point.
2. **First component $\mathbf{u}_1$**: the direction of largest variance.
3. **Next components $\mathbf{u}_2, \mathbf{u}_3, \ldots$**: largest remaining variance, perpendicular to the previous ones.
4. Keep the first k. Describe each point by its k coordinates along them.

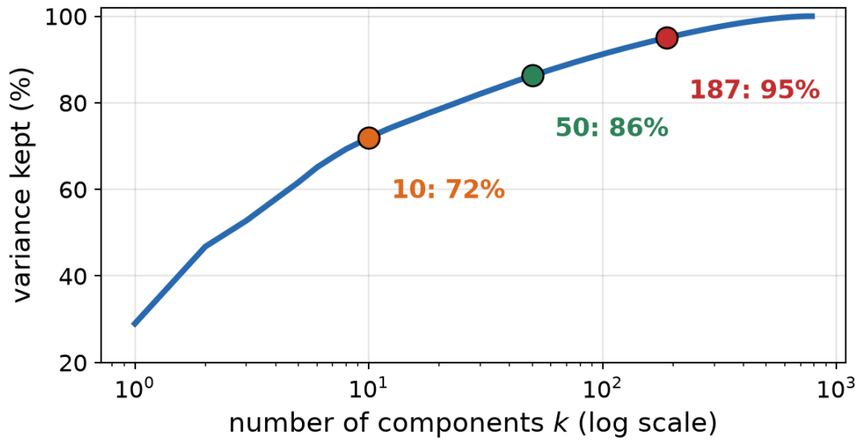
- The components are the **eigenvectors** of the data's **covariance matrix**.

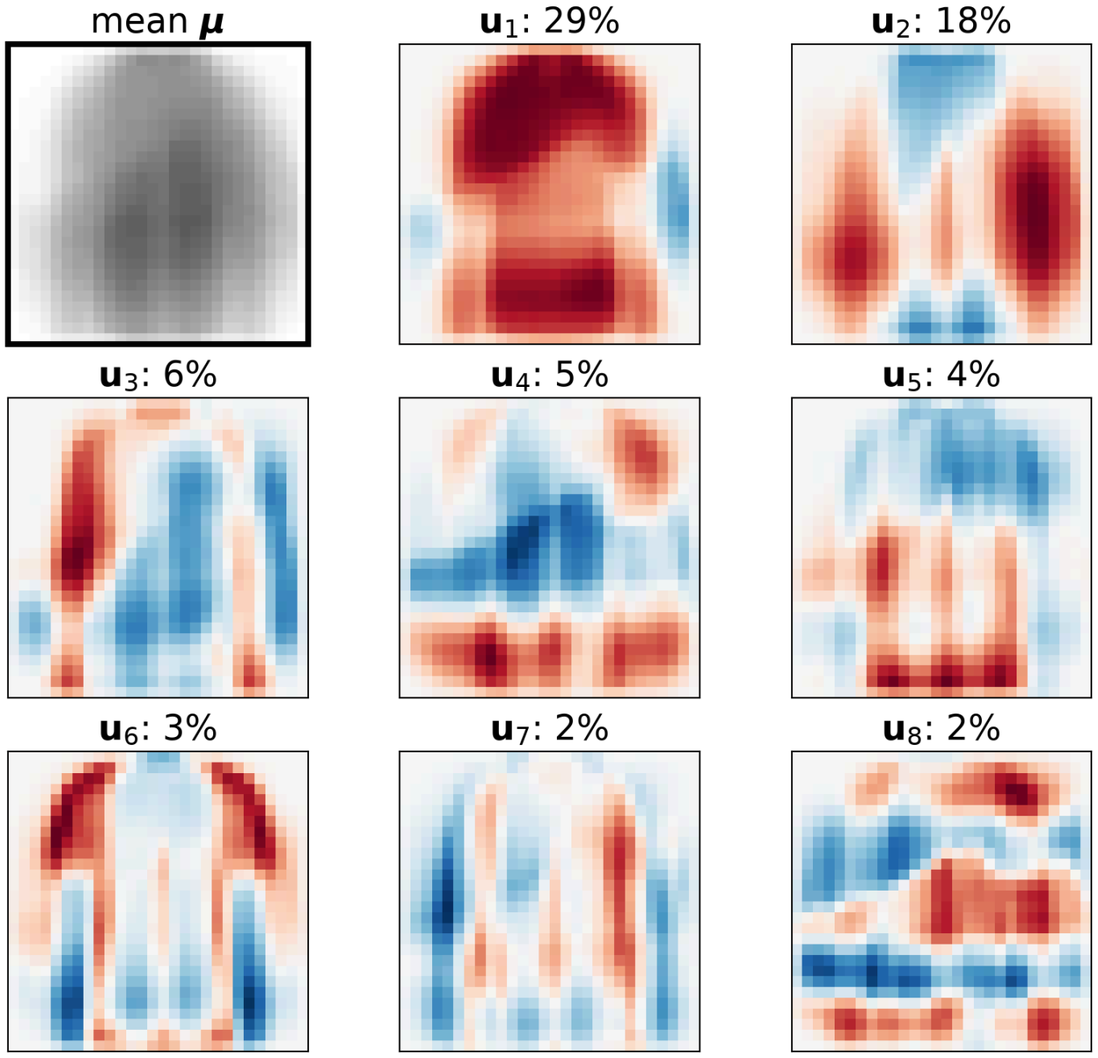


$\mathbf{u}_1$ holds **93%** of the variance. Keep only the coordinate along $\mathbf{u}_1$: 1 number instead of 2, and the points barely move.

PCA on FashionMNIST

- PCA of the 50,000 training images, each a vector of 784 numbers.
- Mean image $\boldsymbol{\mu}$ and components $\mathbf{u}_j$ are 784-vectors too: show them as images.
- Red: add ink to the mean. Blue: remove ink. Early components: coarse shapes. Later: finer details.
- Variance kept: 72% with 10 components, 86% with 50. **187** of the 784 components keep 95%.





Reconstruction with k Components

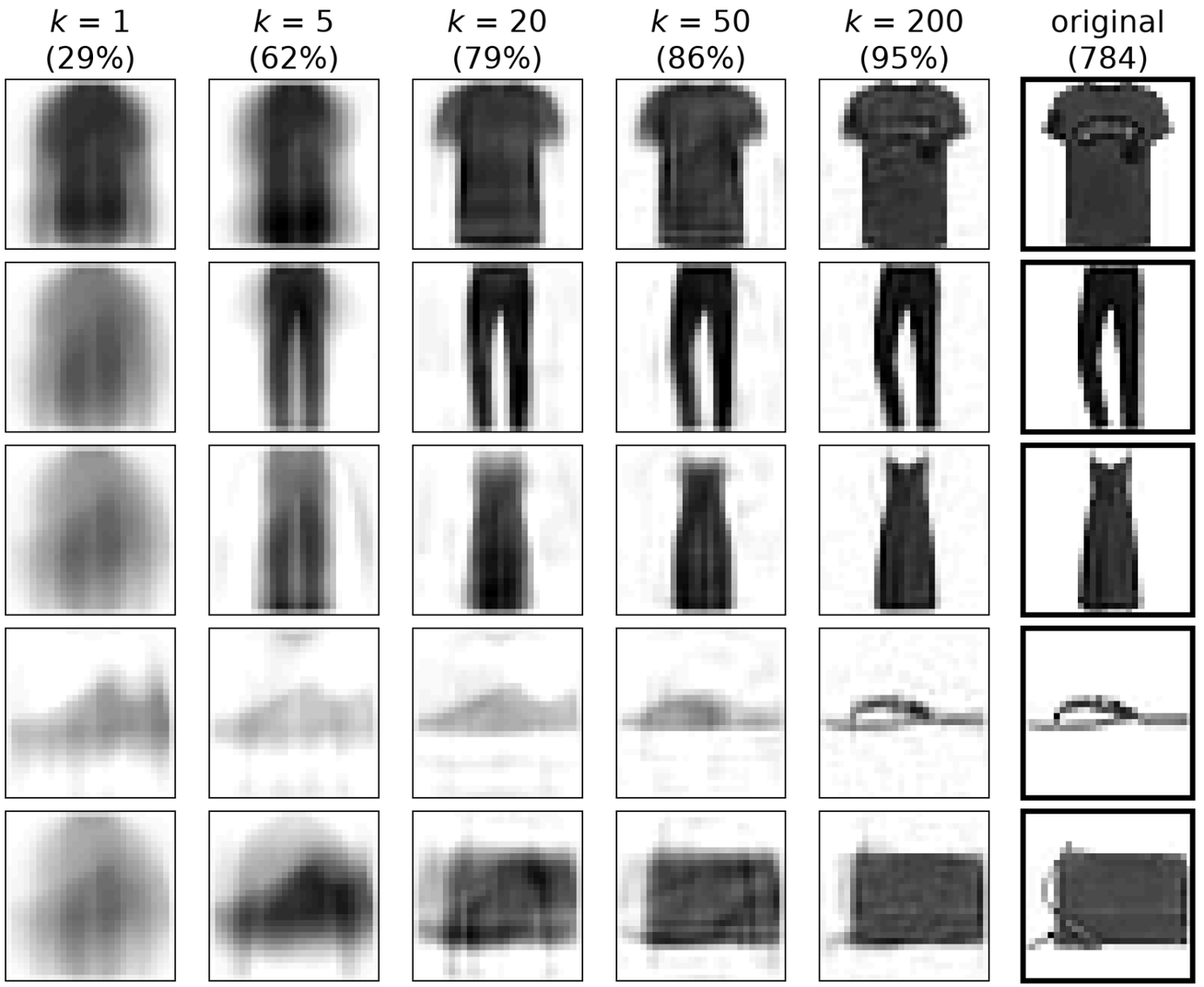
- Any image is the mean plus a weighted sum of components:

$$\hat{\mathbf{x}} = \boldsymbol{\mu} + a_1\mathbf{u}_1 + \cdots + a_k\mathbf{u}_k$$

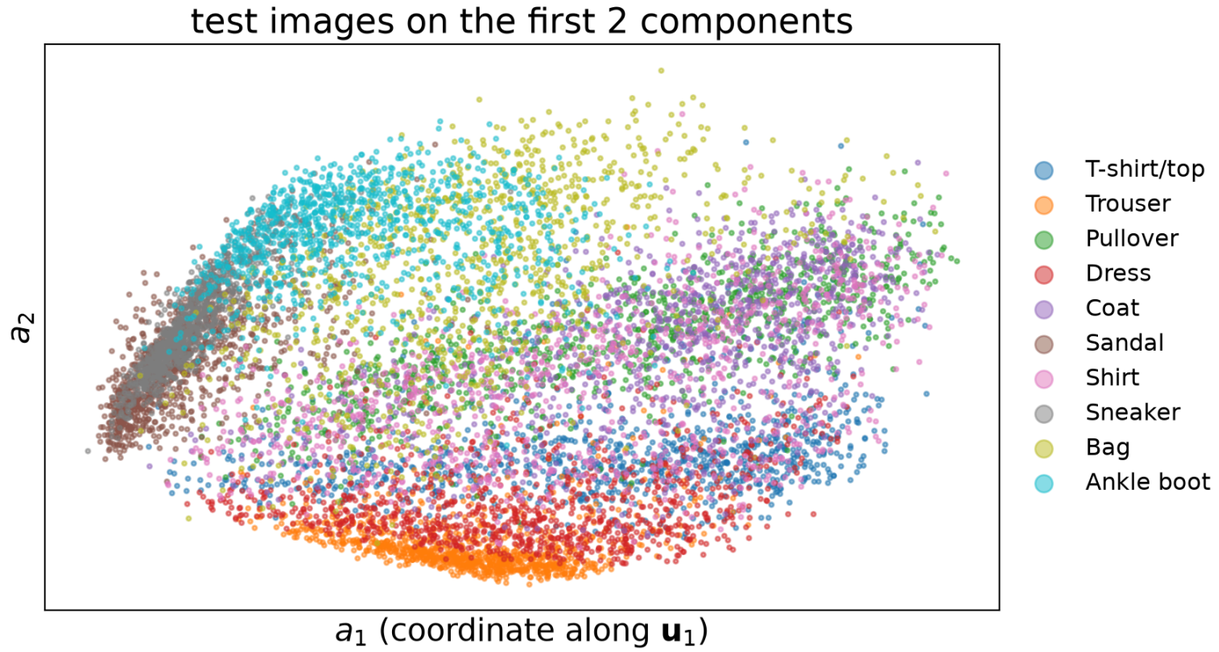
PCA reconstruction

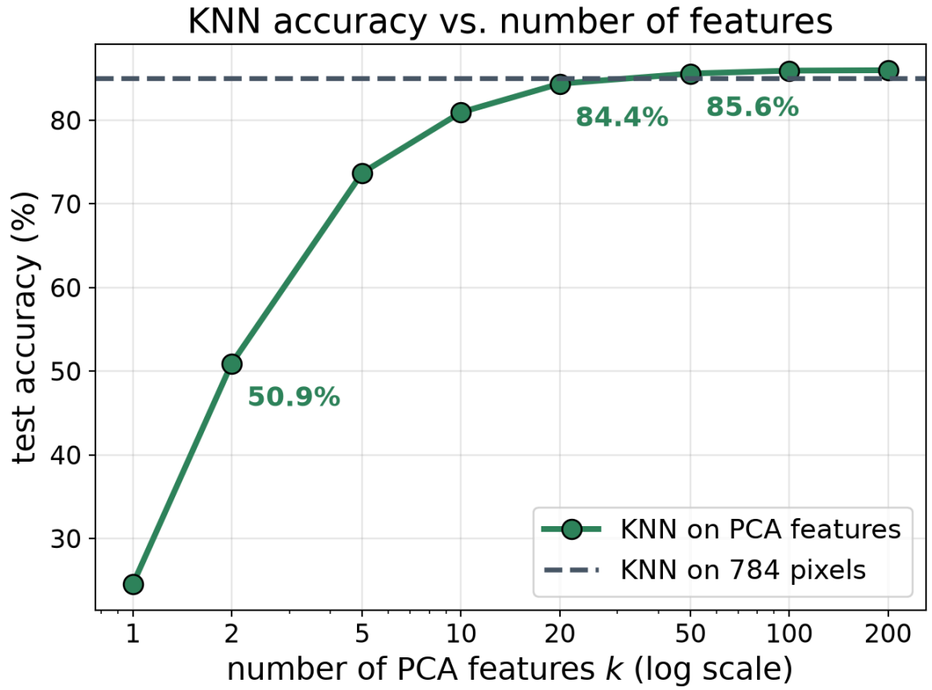
- $\boldsymbol{\mu}$: mean image; $\mathbf{u}_j$: component j.
- $a_j = \mathbf{u}_j \cdot (\mathbf{x} - \boldsymbol{\mu})$: coordinate of image $\mathbf{x}$ along $\mathbf{u}_j$.

- With k = 50 numbers instead of 784, the garment is still easy to recognize.
- So the coordinates $(a_1, \ldots, a_k)$ are compact **features**: an input for a classifier.



PCA Features for Recognition





- **Left:** PCA uses no labels, yet with 2 numbers per image many classes form their own regions; look-alike classes overlap.
- **Right:** 20 features: within 1 point of the 784 pixels (84.4% vs. 85.0%). From 50 on, slightly better (the small components are mostly pixel noise).
- Still pixel-based: a 2-pixel shift drops KNN on 50 PCA features to 66.0%.

PCA in Code

- `PCACompute(data, mean, maxComponents)`: `data` has one image per row. `mean=None`: compute the mean. Returns the mean $\boldsymbol{\mu}$ (1×784) and the components $\mathbf{u}_j$ (one per row: 50×784).
- `PCAProject(X, mean, vecs)`: the coordinates $a_1, \ldots, a_k$ of each row of `X`.
- `PCABackProject(A, mean, vecs)`: the reconstructions $\boldsymbol{\mu} + a_1\mathbf{u}_1 + \cdots + a_k\mathbf{u}_k$.

- **KNN on PCA features** ($k = 6$ neighbors, test set):

Input	Numbers	Accuracy	Time / image
pixels	784	85.0%	1.4 ms
PCA	50	85.6%	0.1 ms

- About the same accuracy, **11× faster**, **16× less memory**.
- Next: a better classifier on the same features.

```python
import cv2
import numpy as np

# X_tr: (50000, 784), X_te: (10000, 784) float32
mu, U = cv2.PCACompute(X_tr, mean=None, maxComponents=50)
F_tr = cv2.PCAProject(X_tr, mu, U)     # (50000, 50)
F_te = cv2.PCAProject(X_te, mu, U)     # (10000, 50)

X_hat = cv2.PCABackProject(F_te, mu, U)    # (10000, 784)
print("MSE:", np.mean((X_hat - X_te) ** 2))

# Any classifier can use the features, e.g., KNN:
knn = cv2.ml.KNearest_create()
knn.train(F_tr, cv2.ml.ROW_SAMPLE, y_tr.astype(np.float32))
_, pred, _, _ = knn.findNearest(F_te, k=best_k)
```

Supervised Learning: SVM

Linear Classifiers

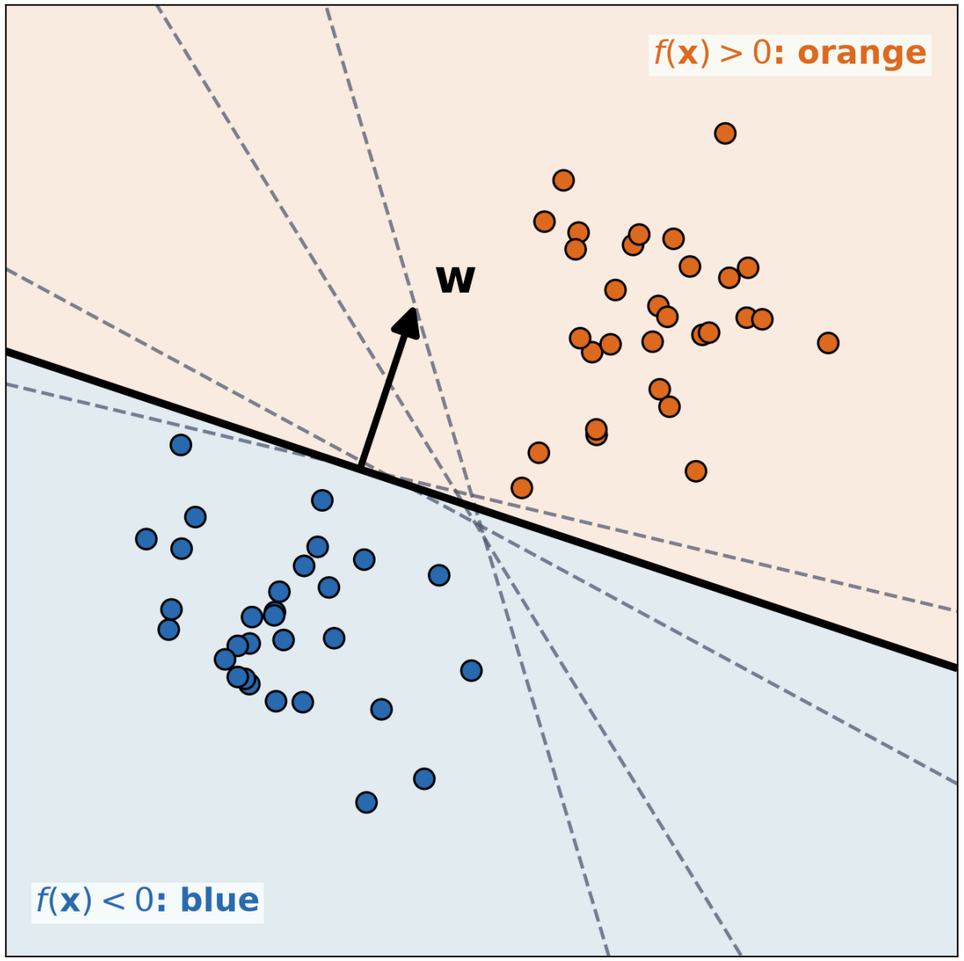
- A **linear classifier** scores an input with a weighted sum:

$$f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b$$

linear classifier

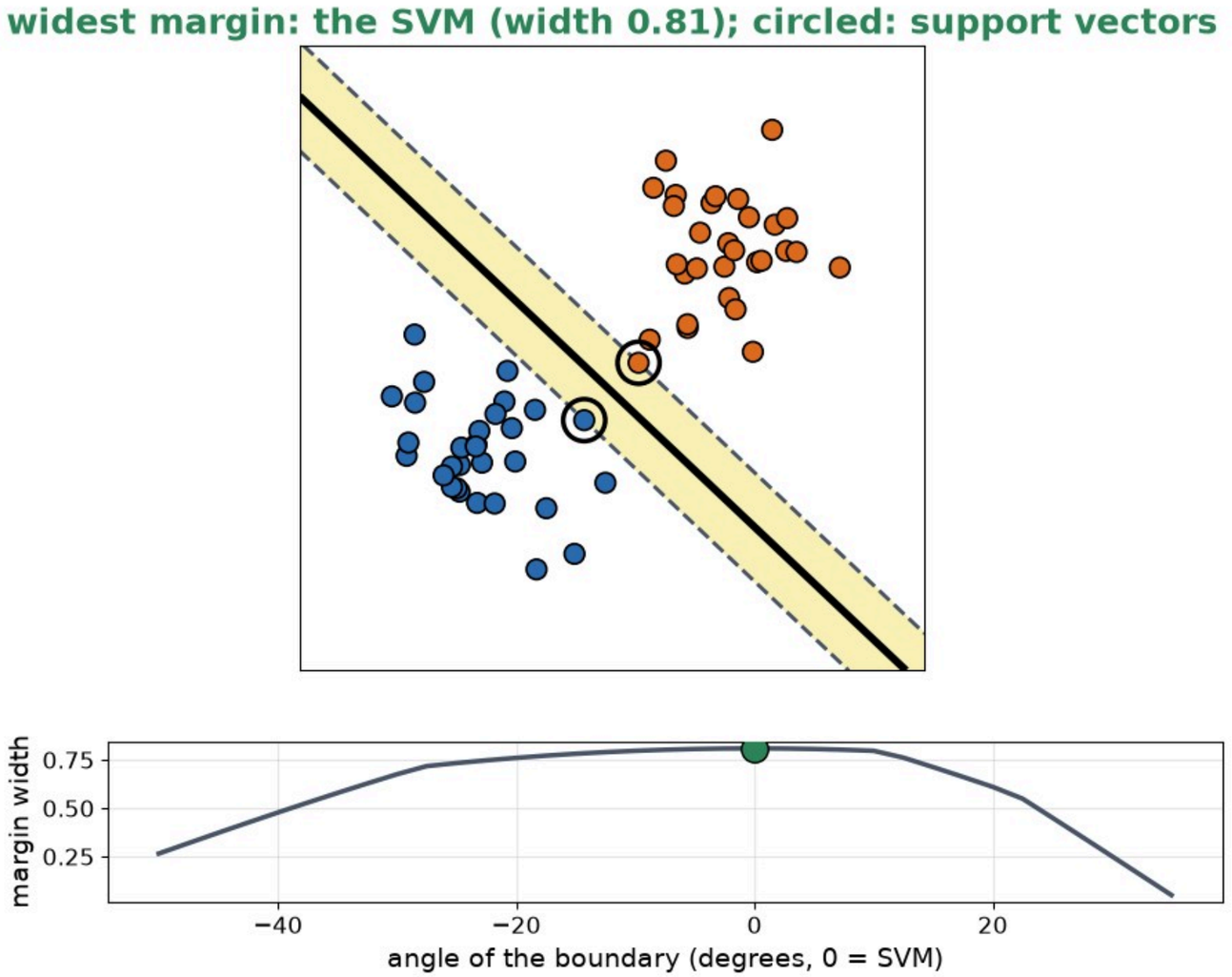
- $\mathbf{x}$: input point coordinates.
- $\mathbf{w}$: weights; b: bias. Both learned.
- Predict orange if $f(\mathbf{x}) > 0$, blue otherwise.
- The **decision boundary** $f(\mathbf{x}) = 0$ is a line (a "hyperplane" in more dimensions).
- Many lines separate the training points (dashed). **Which one is best?**

black: one linear classifier $f(\mathbf{x}) = 0$; dashed: other separating lines



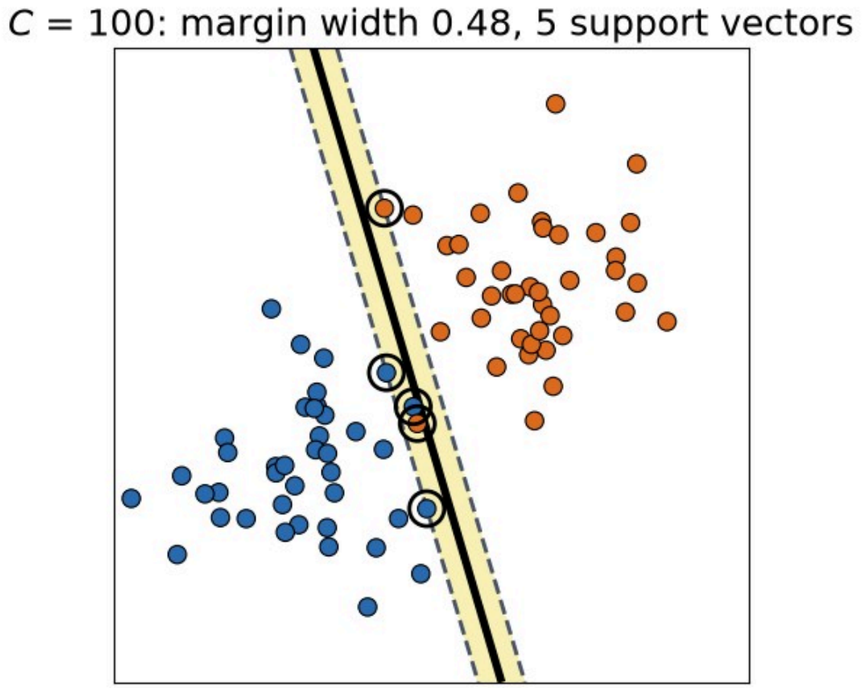
Support Vector Machine (SVM)

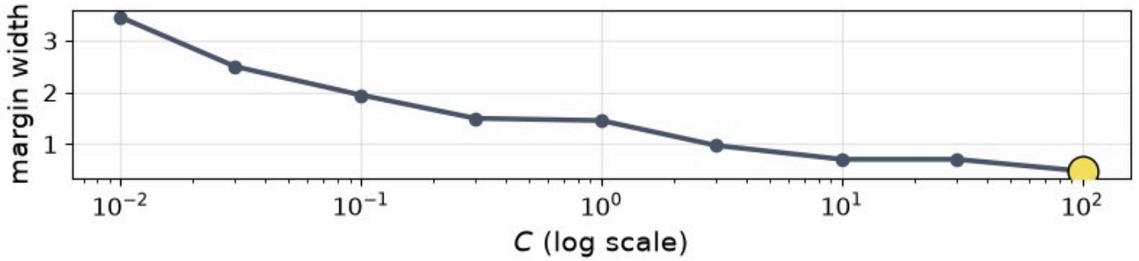
- **SVM** (support vector machine): the linear classifier with the **largest margin**. A wide gap is safer for new points.
- **Margin:** the distance from the boundary to the closest training points.
- **Support vectors:** the points on or inside the margin. They alone fix the boundary; moving any other point changes nothing.



Soft Margin

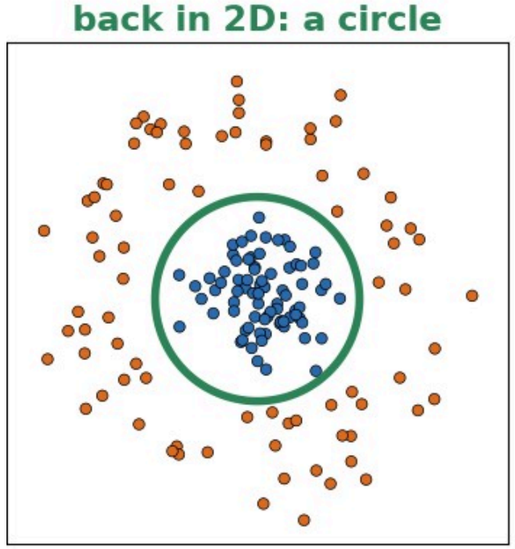
- Overlapping classes: some points must be inside the margin or on the wrong side (**soft margin**).
- Hyperparameter C: the penalty for those points.
  - Small C: wide margin, many points inside.
  - Large C: narrow margin, fewer points inside.
- Like k, choose C on validation data.

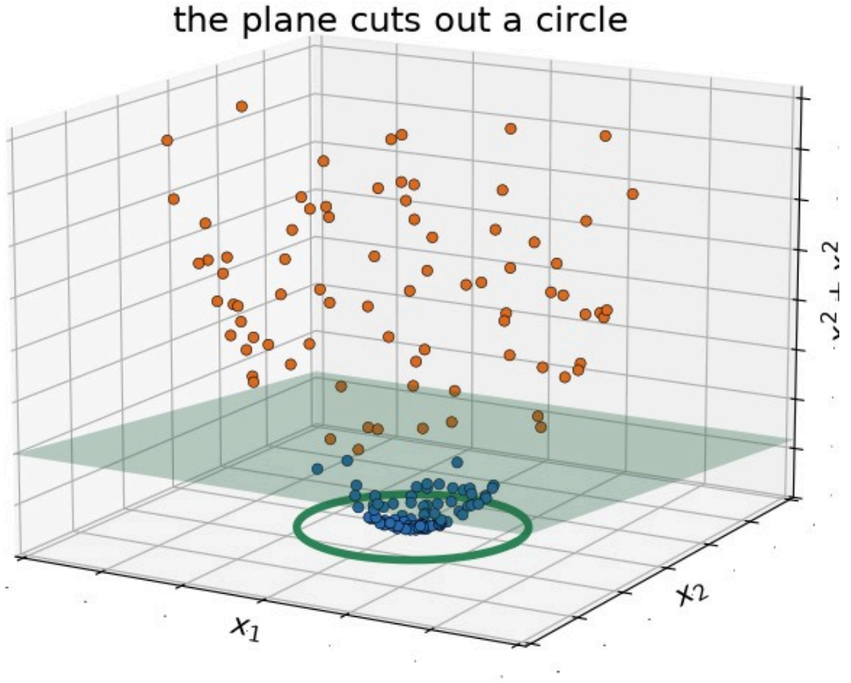




Kernel SVM: Nonlinear Boundaries

- No line separates blue and orange points.
- Idea: **map** each point to a higher-dimensional space, where a **plane** separates them. Here: $(x_1, x_2) \mapsto (x_1, x_2, x_1^2 + x_2^2)$.
- A linear SVM in the new space is a nonlinear boundary in the original space (here: a circle).
- In practice, a **kernel** (e.g., **RBF**) builds the mapping inside the SVM; its hyperparameter γ sets how curvy the boundary can be.





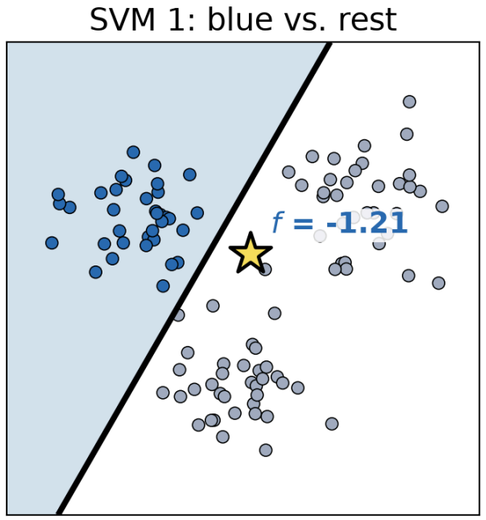
Multi-Class SVM: One vs. Rest

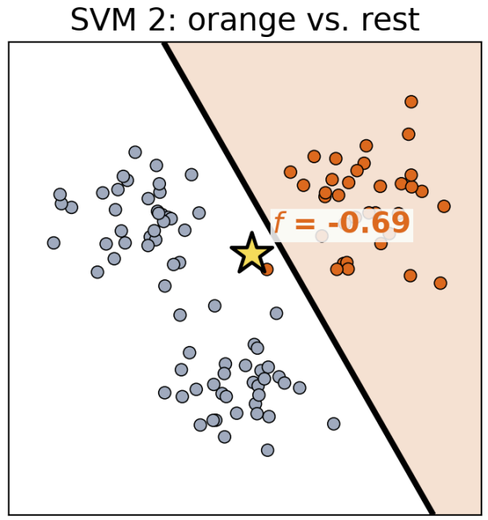
- Unfortunately, there is no definitive multi-class SVM. In practice, we combine multiple two-class SVMs.

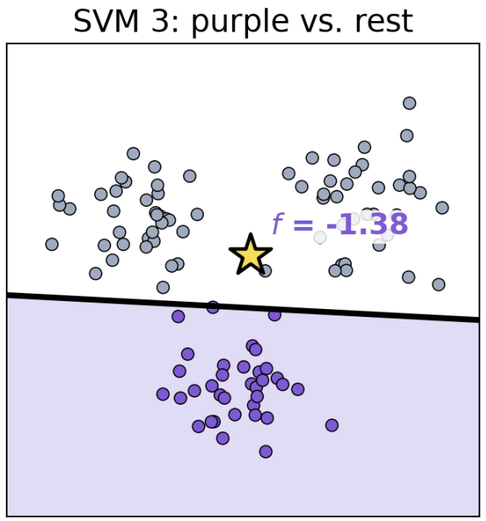
One vs. Rest

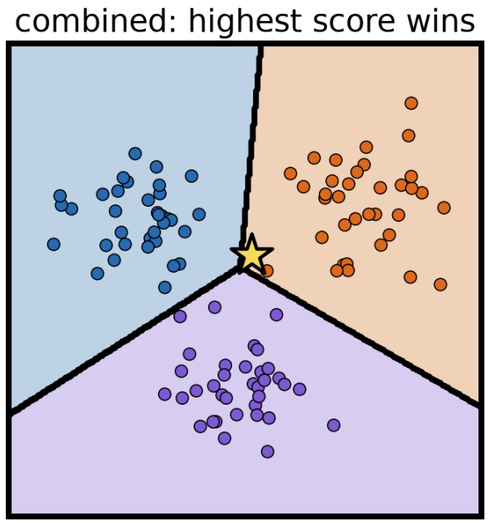
1. For each class c: train a binary SVM, class c (+1) vs. all other classes (−1).
2. New image: compute all class scores $f_c(\mathbf{x})$.
3. Predict the class with the **highest score**.

- 3 classes: 3 SVMs. 10 FashionMNIST classes: 10 binary SVMs.
- Star: f_{blue} = -1.21, f_{orange} = -0.69, f_{purple} = -1.38 $\rightarrow$ predict **orange**.









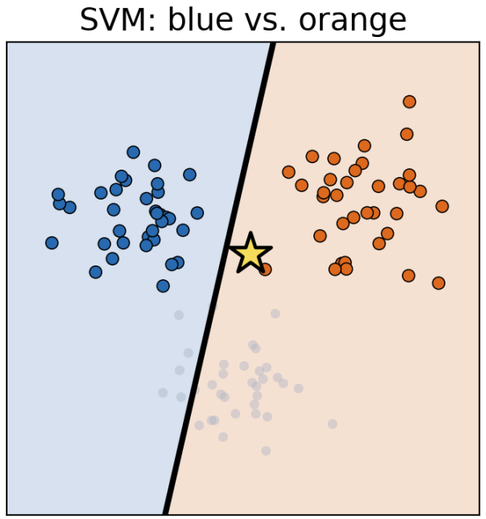
Multi-Class SVM: One vs. One

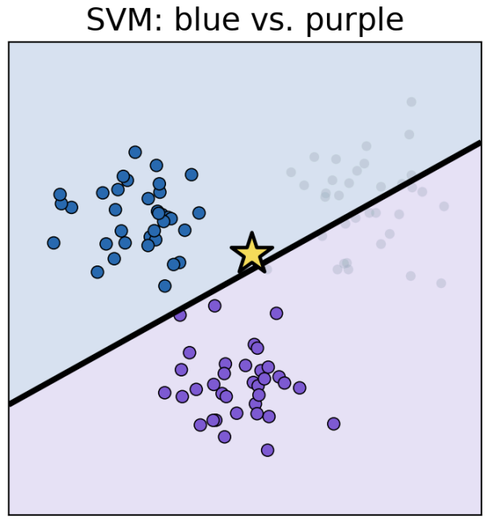
One vs. One

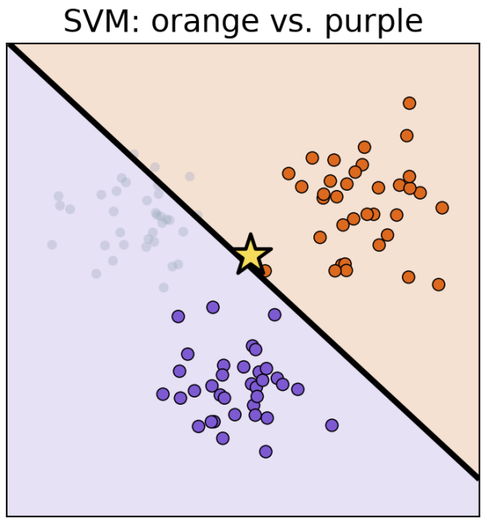
1. For each pair of classes: train a binary SVM on the points of those two classes only.
2. New image: each SVM votes for one of its two classes.
3. Predict the class with the **most votes**.

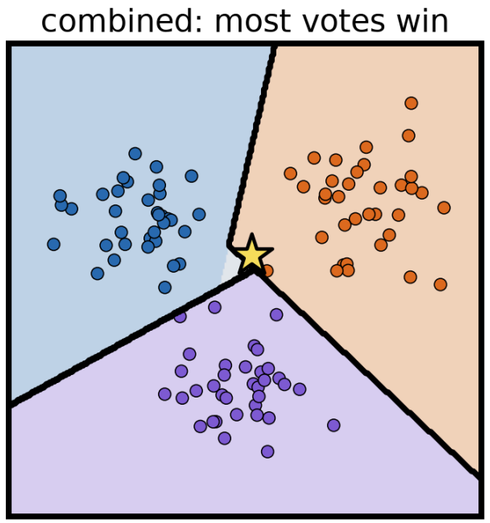
- Star: votes 1 blue, 2 orange, 0 purple $\rightarrow$ predict **orange**.
- OpenCV's SVM handles multiple classes by itself, with **one vs. one**: one SVM per pair of classes, 45 SVMs for 10 classes, then a majority vote.

	One vs. rest	One vs. one
SVMs for 3 classes	3	3
SVMs for 10 classes	10	45
Data per SVM	all images	images of 2 classes
Prediction	highest score	most votes









SVM in Code

- `setType(SVM_C_SVC)`: classification with a soft margin.
- `setKernel(SVM_RBF)`, `setC(...)`, `setGamma(...)`: kernel and hyperparameters.
- `train(F, ROW_SAMPLE, y)`: `F` one feature vector per row (`float32`); `y` labels (`int32`).
- `predict(F)`: predicted labels.

- Choose C and γ on the validation set (training on 10,000 images to save time). Validation accuracy:

	γ = **0.01**	γ = **0.03**	γ = **0.1**
C = 1	84.3%	85.8%	86.1%
C = 10	86.4%	**86.9%**	86.3%
C = 100	86.7%	85.9%	86.2%

- Final model on all 50,000 training images: test accuracy **88.9%**, trained in 17 s.

```python
import cv2
import numpy as np

# F_tr, F_te: PCA features (50 per image), scaled
svm = cv2.ml.SVM_create()
svm.setType(cv2.ml.SVM_C_SVC)
svm.setKernel(cv2.ml.SVM_RBF)
svm.setC(best_C)           # chosen on validation data
svm.setGamma(best_gamma)
svm.train(F_tr, cv2.ml.ROW_SAMPLE, y_tr.astype(np.int32))
# 10 classes: OpenCV trains 45 one-vs-one SVMs inside

_, pred = svm.predict(F_te)
print("test accuracy:", np.mean(pred.ravel() == y_te))
```

PCA + SVM: Full Code

```python
import gzip
import time
from pathlib import Path

import cv2
import numpy as np

# 1. Data: FashionMNIST as 784-vectors. Split 50,000 training / 10,000 validation; 10,000 test.
data_dir = Path("assets/datasets/fashion_mnist")    # folder with the four .gz files

def read_idx(name, offset):
    with gzip.open(data_dir / name) as f:
        return np.frombuffer(f.read(), np.uint8, offset=offset)

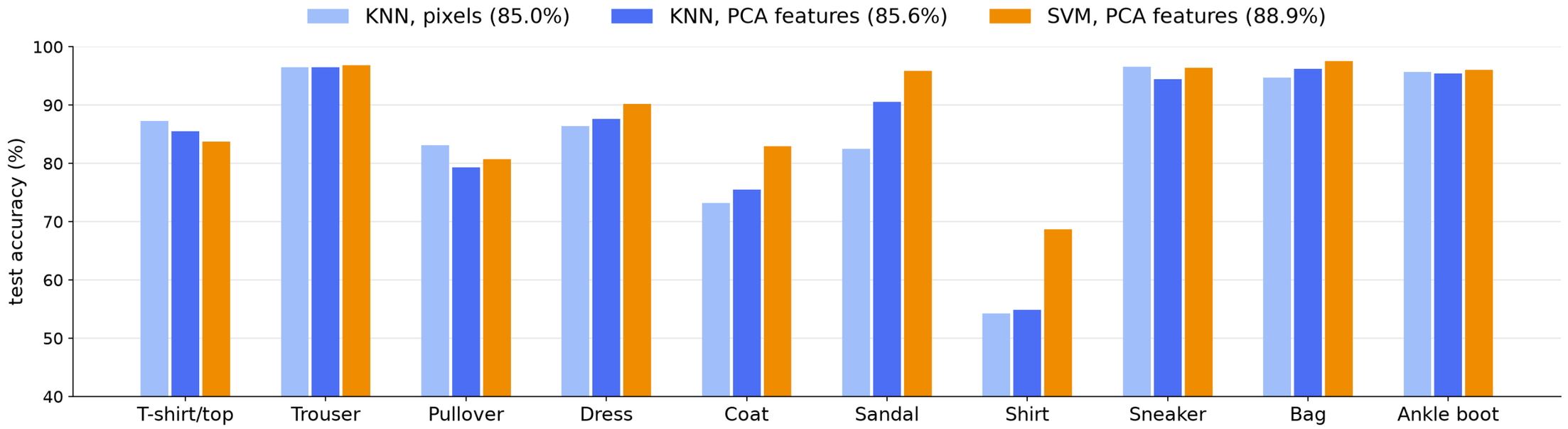
X = read_idx("train-images-idx3-ubyte.gz", 16).reshape(-1, 784).astype(np.float32)
y = read_idx("train-labels-idx1-ubyte.gz", 8).astype(np.int32)
X_te = read_idx("t10k-images-idx3-ubyte.gz", 16).reshape(-1, 784).astype(np.float32)
y_te = read_idx("t10k-labels-idx1-ubyte.gz", 8).astype(np.int32)
X_tr, y_tr, X_val, y_val = X[:50000], y[:50000], X[50000:], y[50000:]

# 2. PCA features (learned on the training set only), scaled to unit spread.
mu, U = cv2.PCACompute(X_tr, mean=None, maxComponents=50)
F_tr = cv2.PCAProject(X_tr, mu, U)
scale = F_tr.std()
F_tr = F_tr / scale
F_val = cv2.PCAProject(X_val, mu, U) / scale
F_te = cv2.PCAProject(X_te, mu, U) / scale
```

Our FashionMNIST Benchmark

Method	Input	Test accuracy	Training time	Time per image	Model size
KNN ($k = 6$)	784 pixels	85.0%	none	1.4 ms	157 MB
KNN ($k = 6$)	50 PCA features	85.6%	PCA only	0.1 ms	10 MB
SVM (RBF)	50 PCA features	**88.9%**	16.5 s	0.1 ms	3.1 MB

- SVM: best so far, **+3.9 points** over KNN on pixels.
- All three methods use pixel-based features. Neural networks learn their own features; we will compare them later.



Key Takeaways

- **Recognition tasks** differ in their output: a label per image (classification), a box per object (detection), a label per pixel (segmentation).
- **Supervised** learning uses images and labels; **unsupervised** uses images only.
- Training error is not the goal: a model that is **too simple underfits**, one that is **too flexible overfits**. Fit **parameters** on the training set, choose **hyperparameters** on the validation set, and report the **test** error once.
- Pick the **metric for the task**. With imbalanced classes, accuracy alone misleads.
- **KNN** predicts the majority label of the k nearest training images. It is simple, but **slow**, and pixel distance is **not semantic**.
- **k-means** groups unlabeled points into k clusters. **PCA** finds the directions of largest variance and gives compact **features**.
- The **SVM** picks the boundary with the **largest margin**. C trades margin width for training mistakes; kernels give nonlinear boundaries.
- All these methods use **pixel-based features**; neural networks learn their own features.