Lecture 14

Recognition and kNN

Administrative

A3 is out

- Due Feb 21st

A4 is out

- Due Mar 7th

Administrative

Recitation this friday

- Ontologies

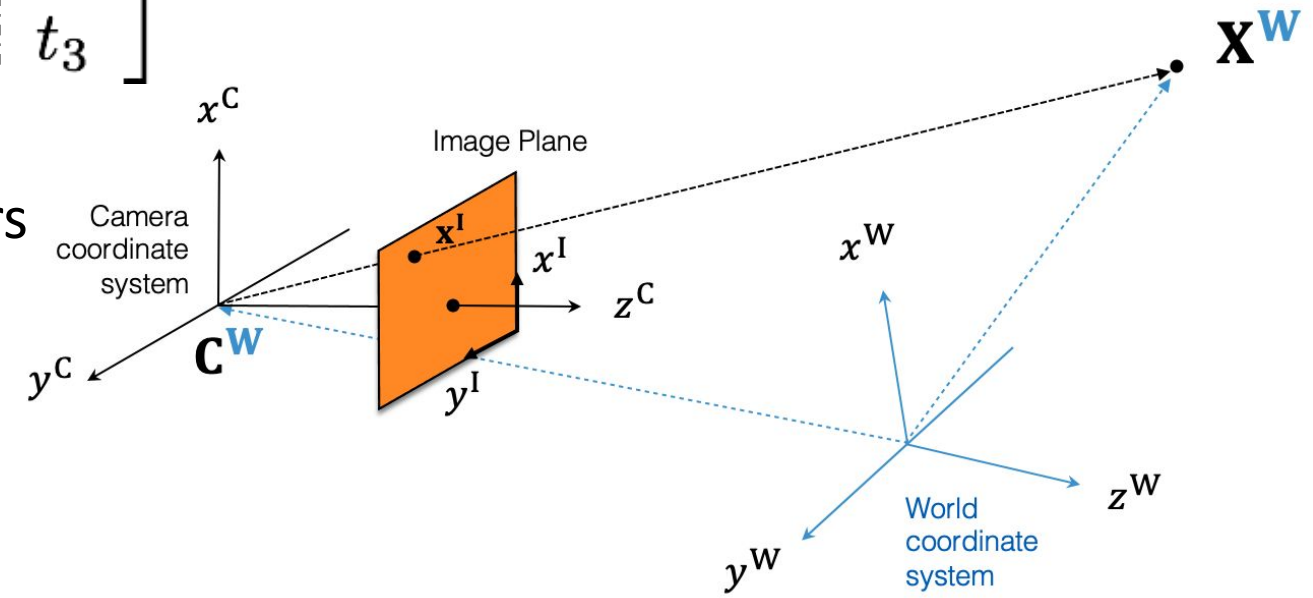
So far: General pinhole camera matrix

$$\mathbf{P} = \mathbf{K}[\mathbf{R}|\mathbf{t}] \qquad \text{where} \qquad \mathbf{t} = -\mathbf{RC}$$

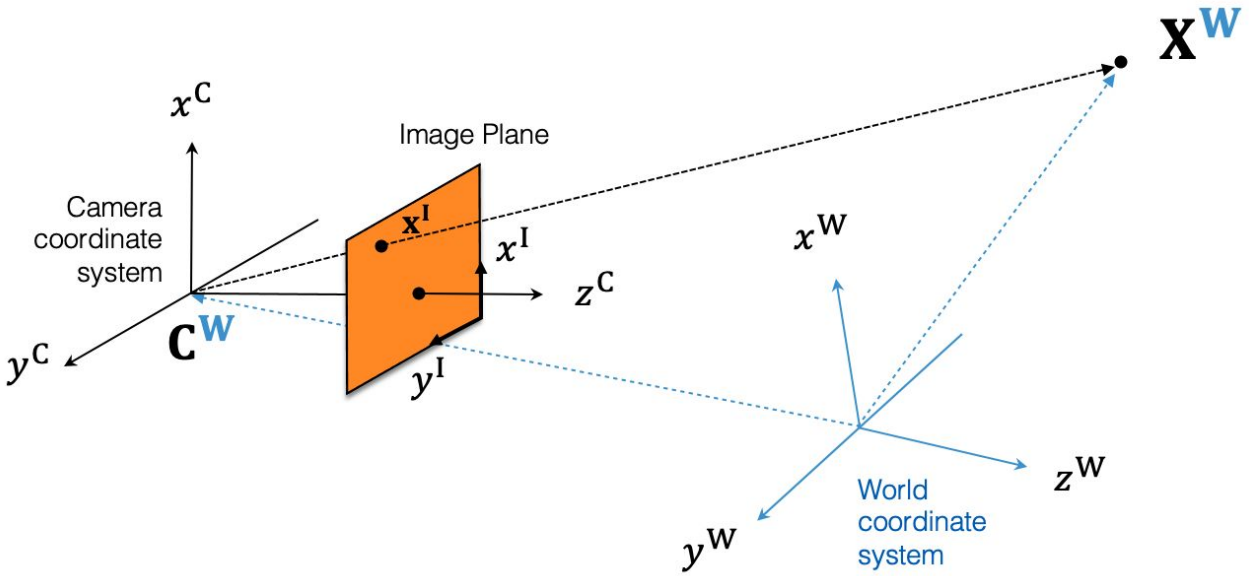
$$\mathbf{P} = \begin{bmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_1 & r_2 & r_3 & t_1 \\ r_4 & r_5 & r_6 & t_2 \\ r_7 & r_8 & r_9 & t_3 \end{bmatrix}$$

intrinsic parameters

extrinsic parameters



So far: The Pinhole Camera Model



$$\tilde{\mathbf{x}}^{\mathrm{I}} \sim \mathbf{P}\tilde{\mathbf{X}}^{\mathrm{W}} \qquad \mathbf{P} = \begin{bmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{bmatrix} [\mathbf{I} \mid \mathbf{0}] \begin{bmatrix} \mathbf{R} & -\mathbf{RC} \\ \mathbf{0} & 1 \end{bmatrix} = \mathbf{K}[\mathbf{R}|\mathbf{t}]$$

intrinsic parameters **K** (3 x 3): correspond to camera internals (image-to-image transformation)

perspective projection (3 x 4): maps 3D to 2D points (camera-to-image transformation)

extrinsic parameters (4 x 4): correspond to camera externals (world-to-camera transformation)

So far: Solving for camera matrix via total least squares

$$\hat{\boldsymbol{x}} = \arg\min_{\boldsymbol{x}} \|\mathbf{A}\boldsymbol{x}\|^2 \text{ subject to } \|\boldsymbol{x}\|^2 = 1$$

$$\mathbf{A} = \begin{bmatrix} \boldsymbol{X}_1^T & 0 & -x_1'\boldsymbol{X}_1^T \\ \mathbf{0} & \boldsymbol{X}_1^T & -y_1'\boldsymbol{X}_1^T \\ \vdots & \vdots & \vdots \\ \boldsymbol{X}_N^T & 0 & -x_N'\boldsymbol{X}_N^T \\ \mathbf{0} & \boldsymbol{X}_N^T & -y_N'\boldsymbol{X}_N^T \end{bmatrix} \qquad \boldsymbol{x} = \begin{bmatrix} \boldsymbol{p}_1 \\ \boldsymbol{p}_2 \\ \boldsymbol{p}_3 \end{bmatrix}$$

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$$

Equivalently, solution ***x*** is the Eigenvector corresponding to the smallest Eigenvalue of **A**

So far: Decomposition of the Camera Matrix

$$\mathbf{P} = \left[\begin{array}{c|c} \overline{\mathbf{P}} & \begin{matrix} p_4 \\ p_8 \\ p_{12} \end{matrix} \end{array}\right] \sim \mathbf{K}\,[\mathbf{R} \mid -\mathbf{RC}]$$

$\overline{\mathbf{P}}^{\mathsf{T}}\overline{\mathbf{P}} \sim \mathbf{K}^{\mathsf{T}}\mathbf{K}$ **with $\mathbf{K}$ upper triangular p.d.**

Obtain $\mathbf{K}$ by Cholesky decomposition of $\overline{\mathbf{P}}^{\mathsf{T}}\overline{\mathbf{P}} = \mathbf{L}\mathbf{L}^{\mathsf{T}}$

$$\mathbf{K} \sim \mathbf{L}^{\mathsf{T}}$$

$$|\mathbf{R}| = 1 \;\Rightarrow \boldsymbol{\lambda} = |\mathbf{K}^{-1}\overline{\mathbf{P}}|^{-1/3}$$

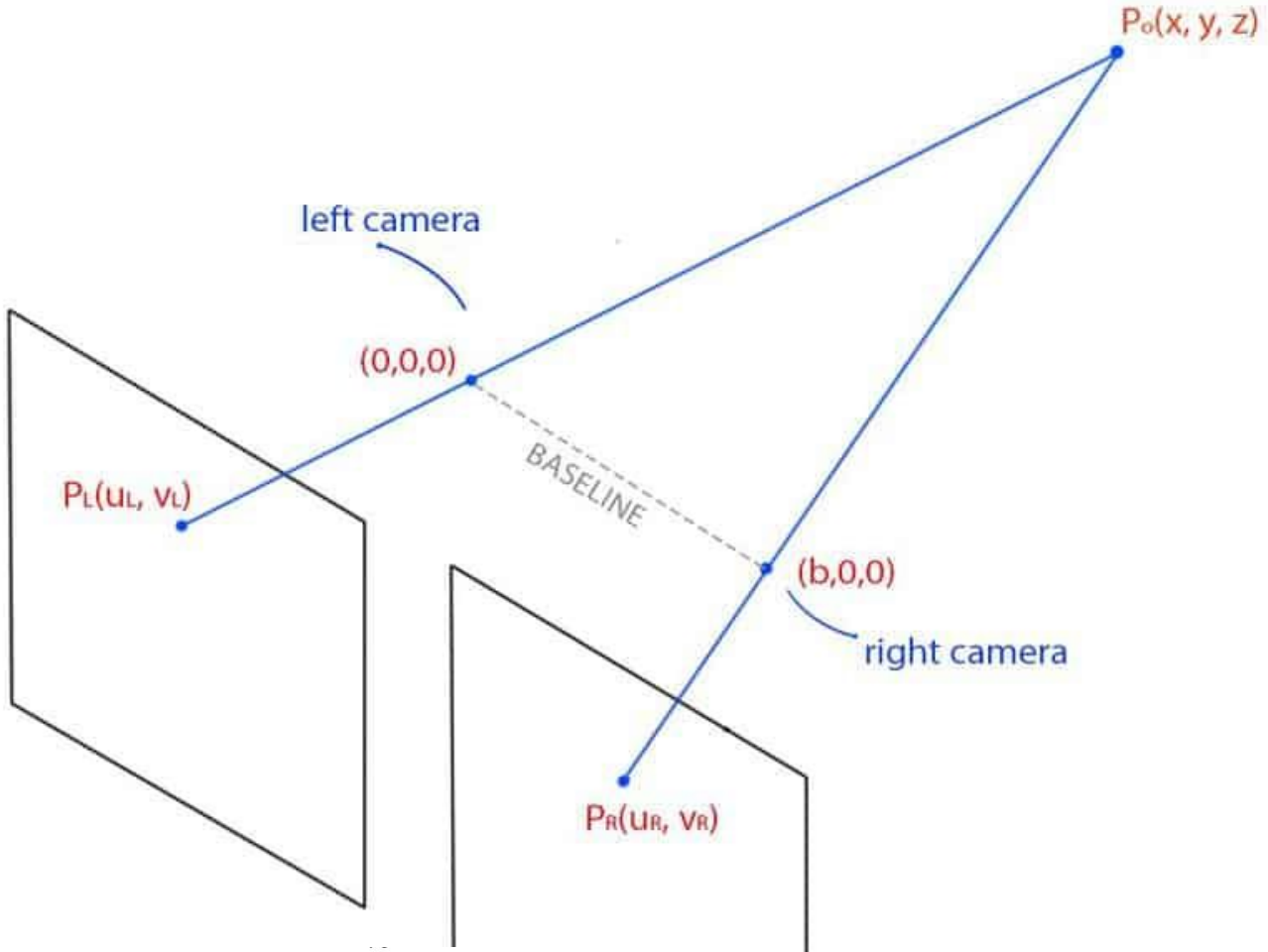
Once $\mathbf{K}$ is known, we can compute $\mathbf{R} = \boldsymbol{\lambda}\mathbf{K}^{-1}\overline{\mathbf{P}}$

Finally, easy to know the camera center: $\mathbf{C} = -\boldsymbol{\lambda}^{-1}\,\mathbf{R}^{\mathsf{T}}\mathbf{K}^{-1}[p_4 \;\; p_8 \;\; p_{12}]^{\mathsf{T}}$

So far: Estimating depth

$$u_L = f\frac{x}{z} + p_x$$

$$v_L = f\frac{y}{z} + p_y$$



$$K = \begin{bmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{bmatrix}$$

$$u_R = f\frac{x-b}{z} + p_x$$

$$v_R = f\frac{y}{z} + p_y$$

Today’s agenda

- Introduction to recognition
- A object recognition pipeline
- Choosing the right features
- A training algorithm: KNN
- Testing an algorithm
- Challenges with kNN
- Dimensionality reduction

Today’s agenda

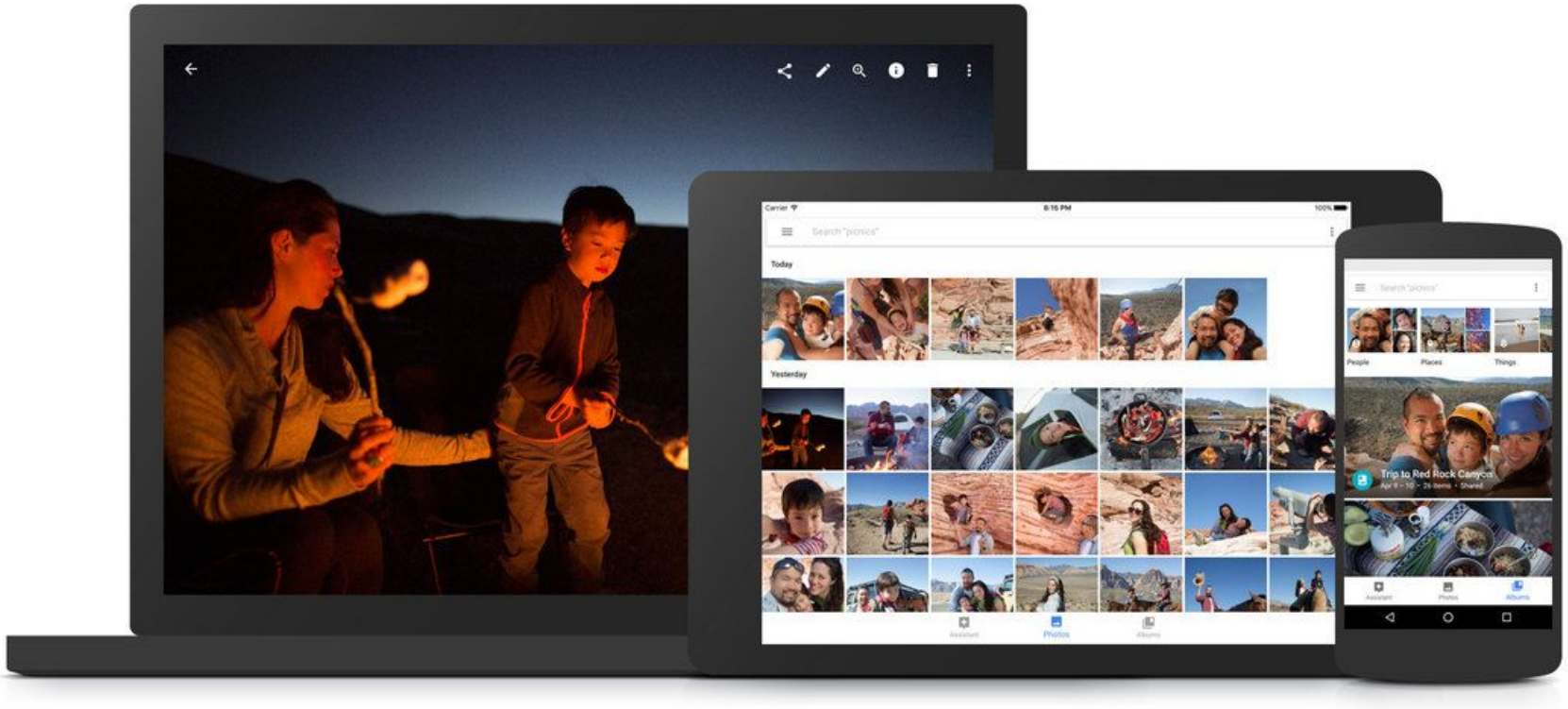
- Introduction to recognition
- A object recognition pipeline
- Choosing the right features
- A training algorithm: KNN
- Testing an algorithm
- Challenges with kNN
- Dimensionality reduction

What do we mean by recognition?

Classification: Does this image contain a building? [yes/no]

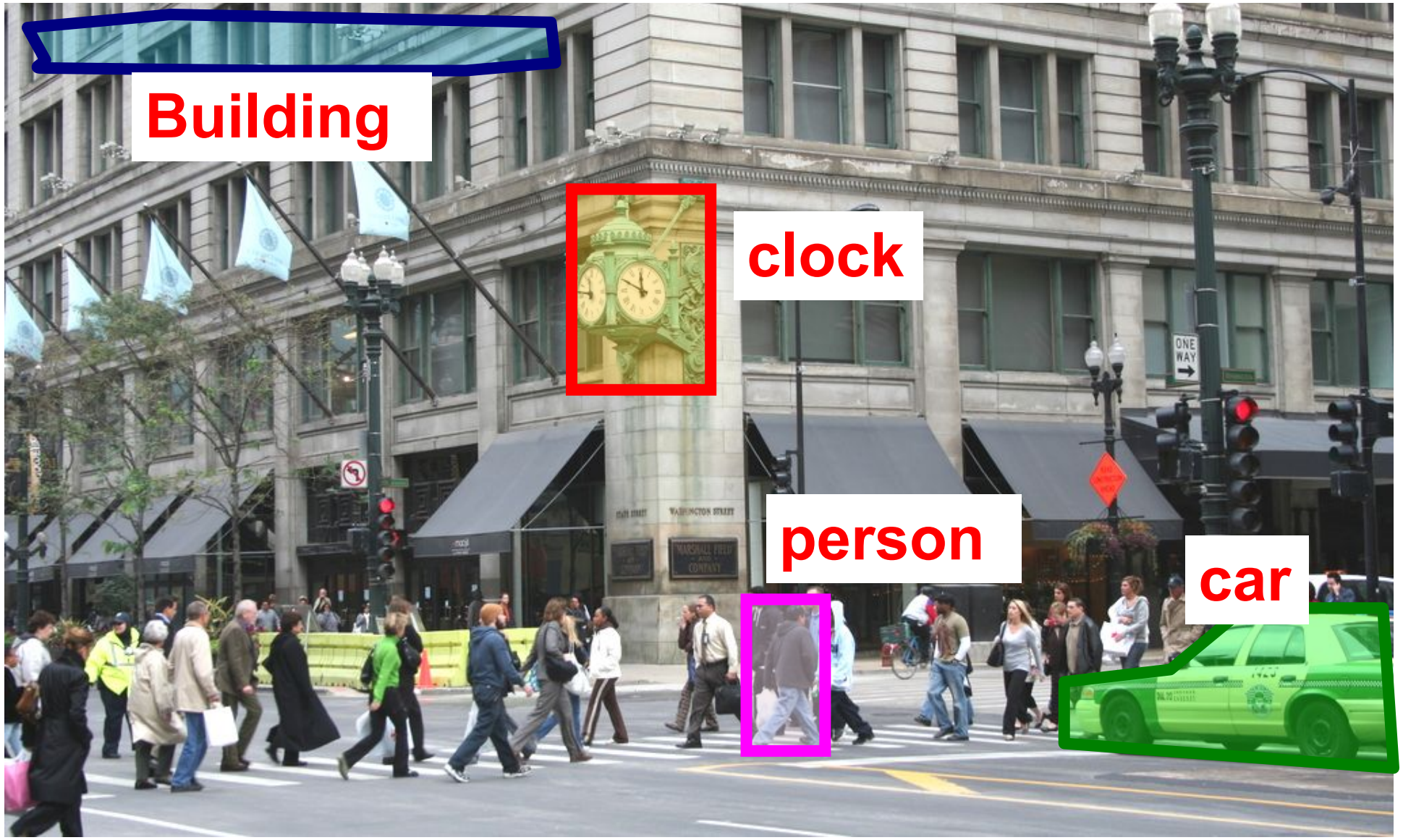
Classification: Is this an beach?

Applications: Image Search & Organizing photo collections



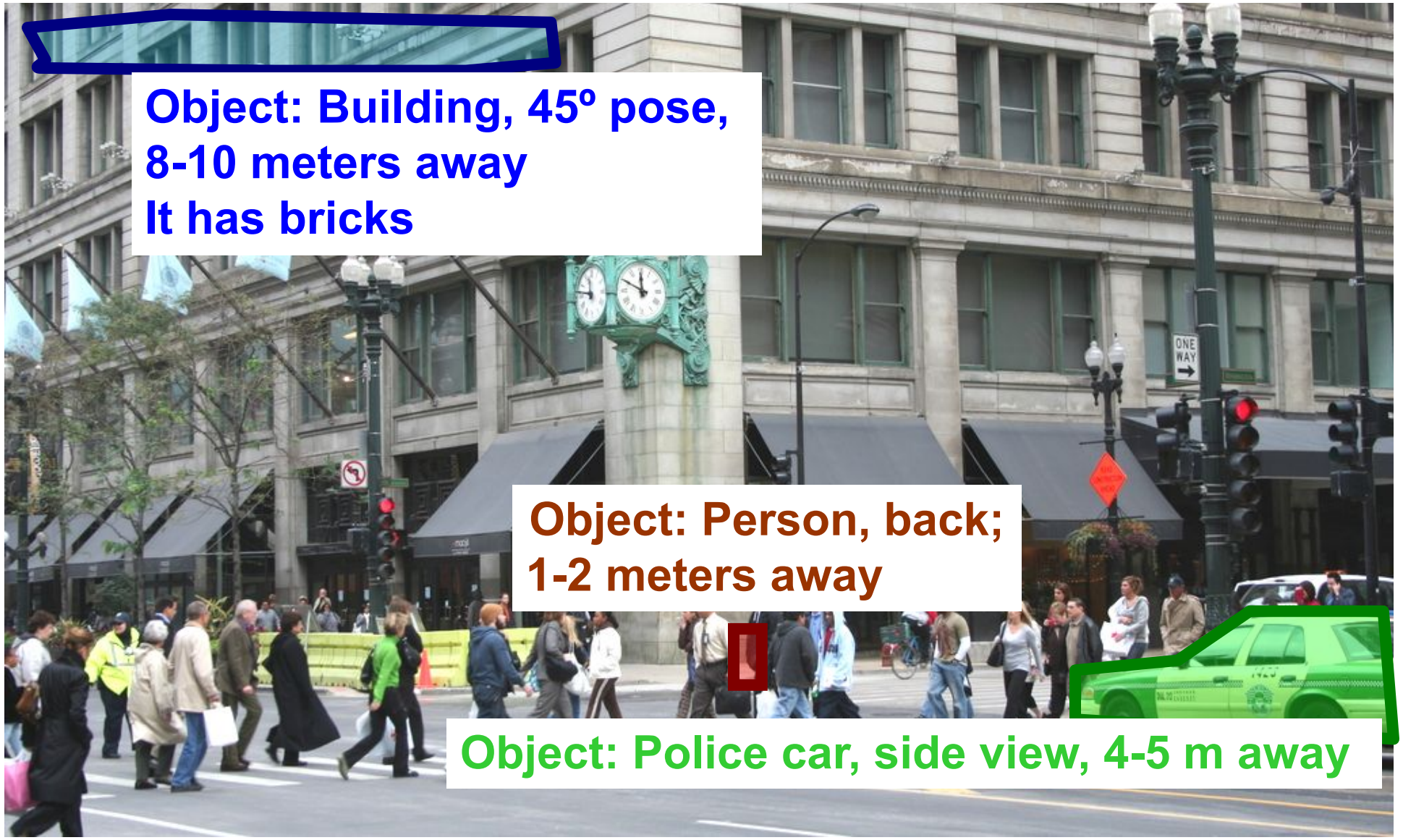
Detection: Does this image contain a car? [where?]

Detection: Which object does this image contain? [where?]



Detection: Accurate localization (segmentation)

Detection: Estimating object semantic & geometric attributes



Levels of recognition: Category-level vs instance-level

Does this image contain the Chicago Macy's building?

Categorization vs Single instance recognition

We have seen a form of single instance categorization already: **Where is the crunchy nut?**

Applications of computer vision

Recognizing landmarks
in mobile devices

Activity recognition: What are these people doing?

Visual Recognition

- Design algorithms that can:
  - Classify images or videos
  - Detect and localize objects
  - Estimate semantic and geometrical attributes
  - Classify human activities and events

Why is this challenging?

How many object categories are there?

Challenges: viewpoint variation

Michelangelo 1475-1564

Challenges: illumination

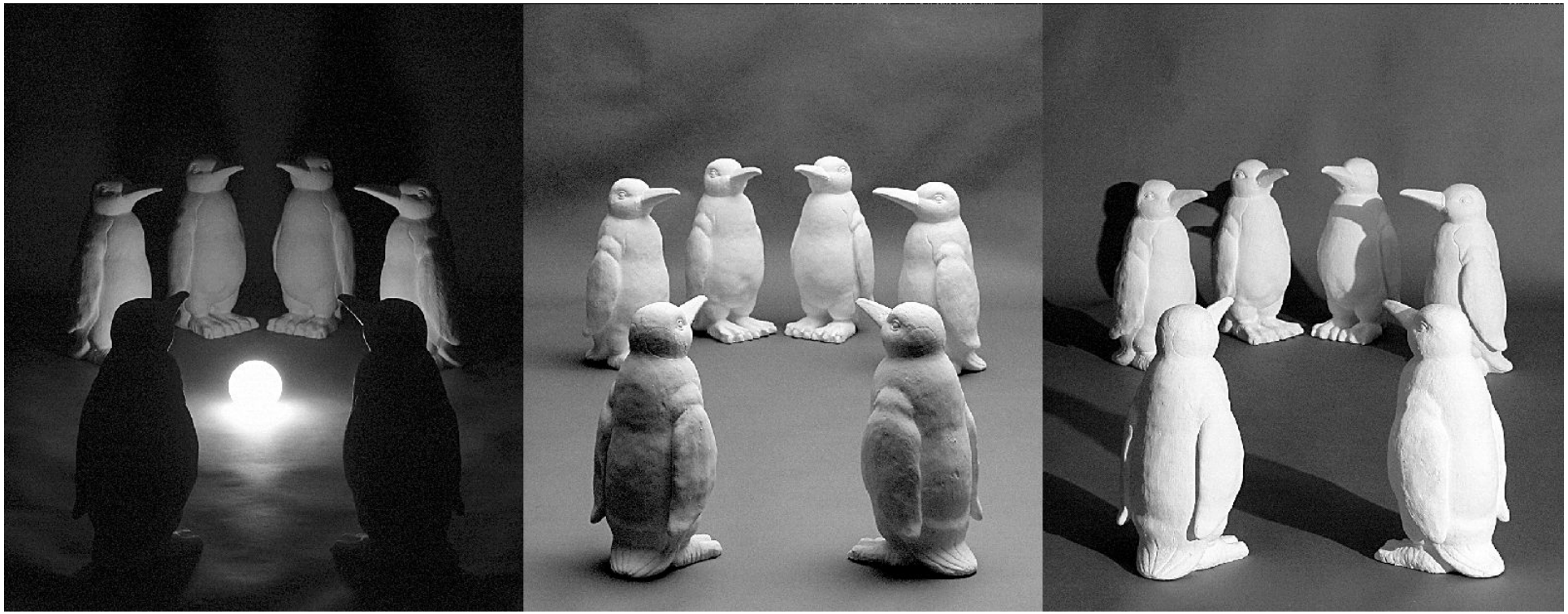
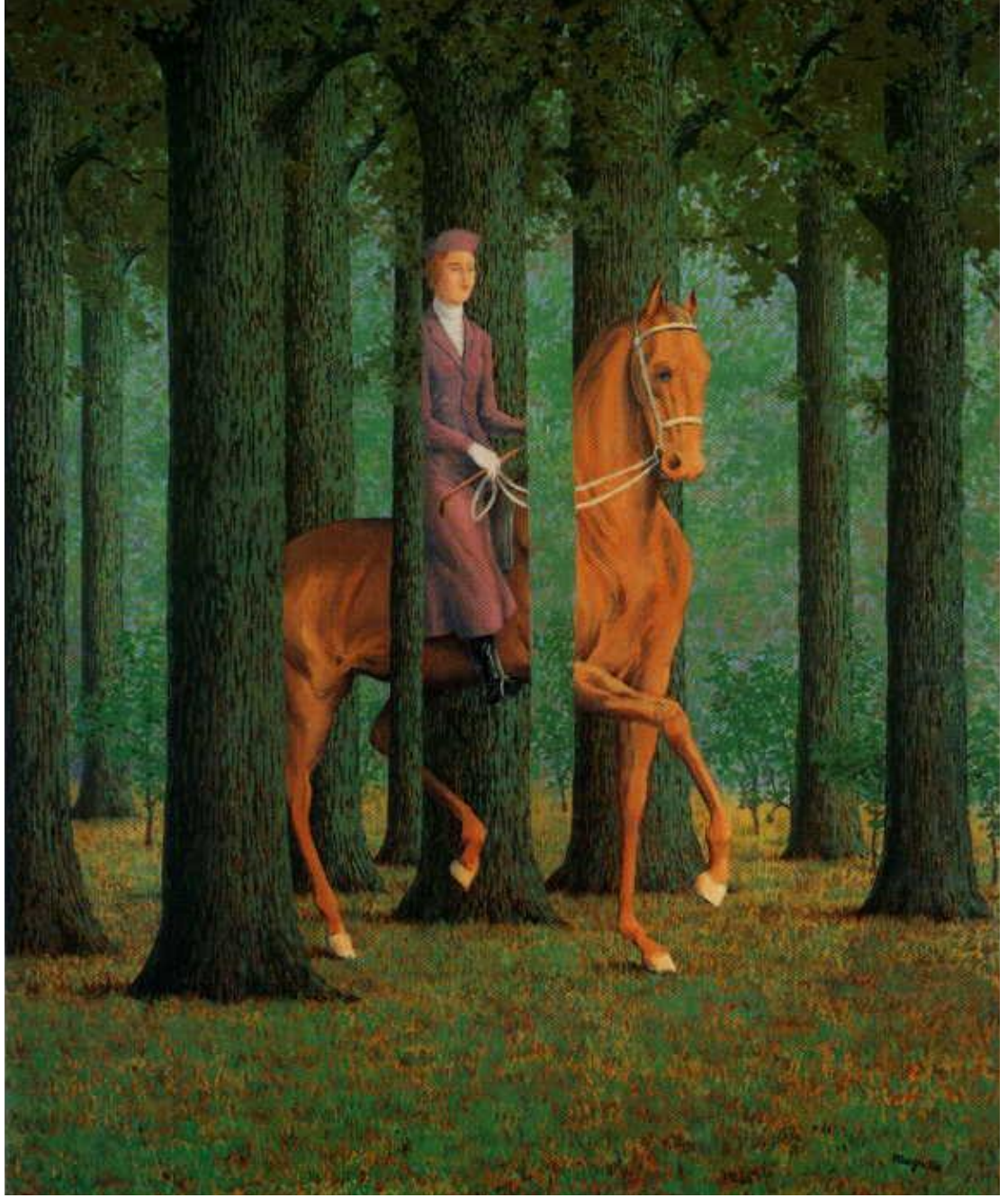


image credit: J. Koenderink

Challenges: scale

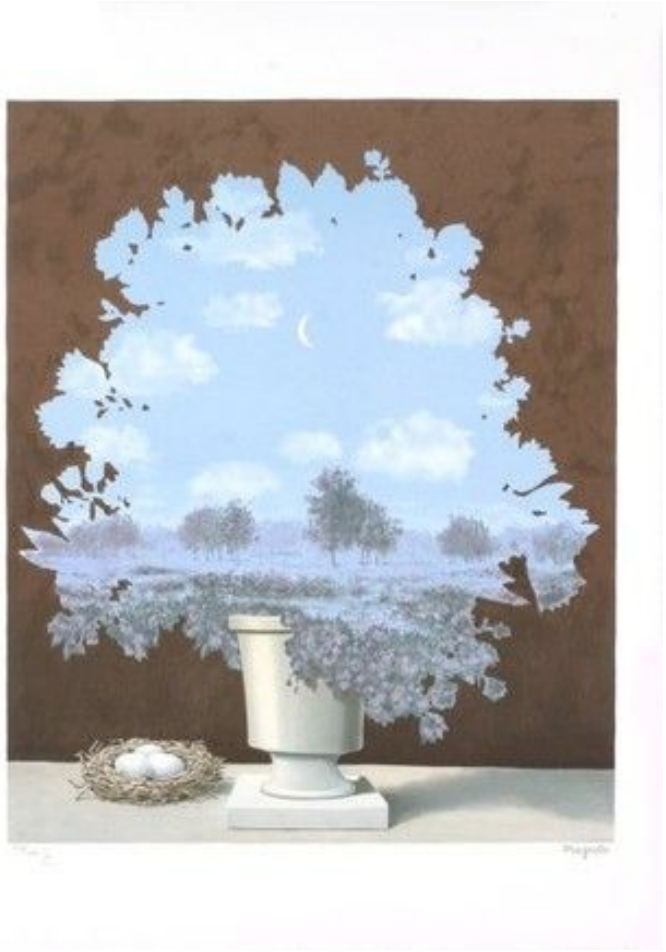
Challenges: deformation

Challenges: occlusion



Magritte, 1957

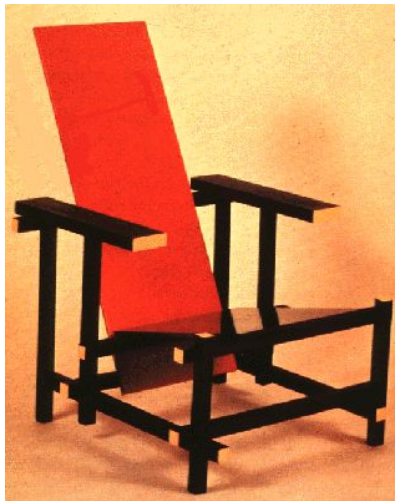
Art Segway - Magritte



Challenges: background clutter

Kilmeny Niland. 1995

Challenges: intra-class variation



Today’s agenda

- Introduction to recognition
- A object recognition pipeline
- Choosing the right features
- A training algorithm: KNN
- Testing an algorithm
- Challenges with kNN
- Dimensionality reduction

Object recognition:
a classification framework

- Apply a prediction function to a feature representation of the image to get the desired output:

f() = “apple”

f() = “tomato”

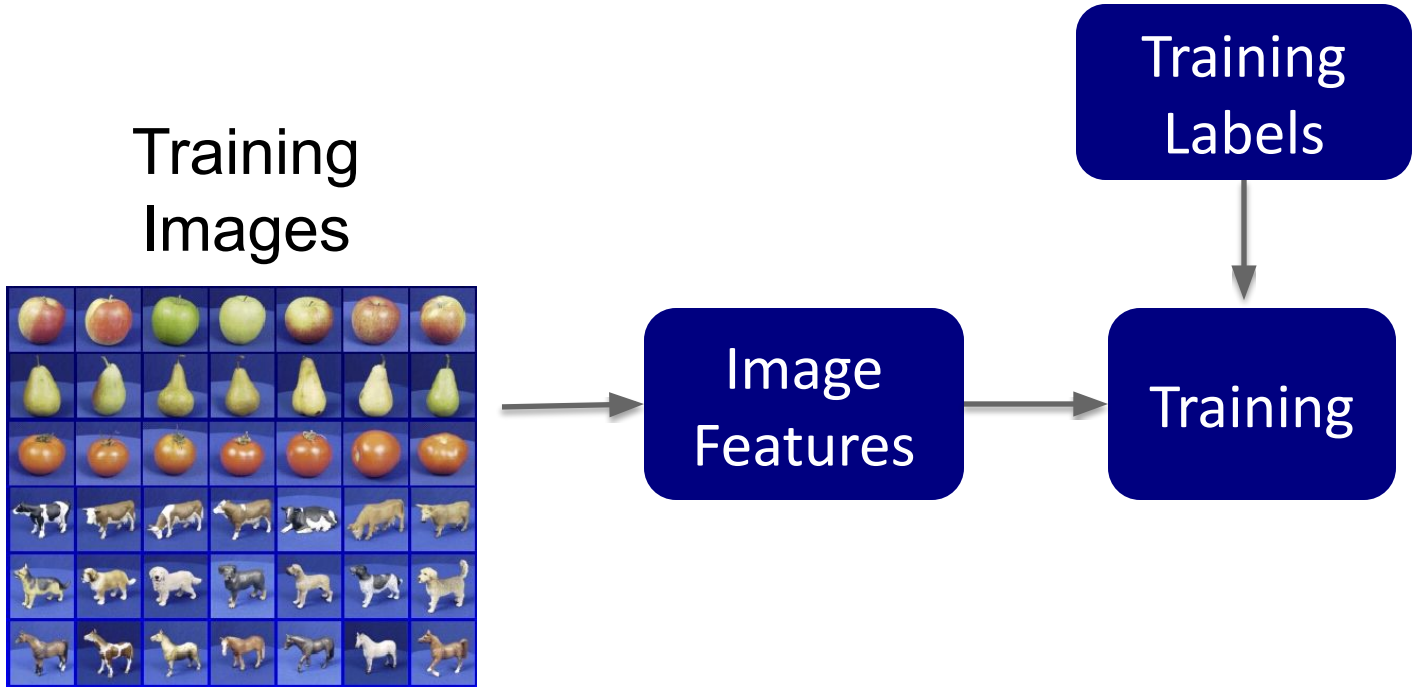
f() = “cow”

A simple pipeline - Training

Training Images

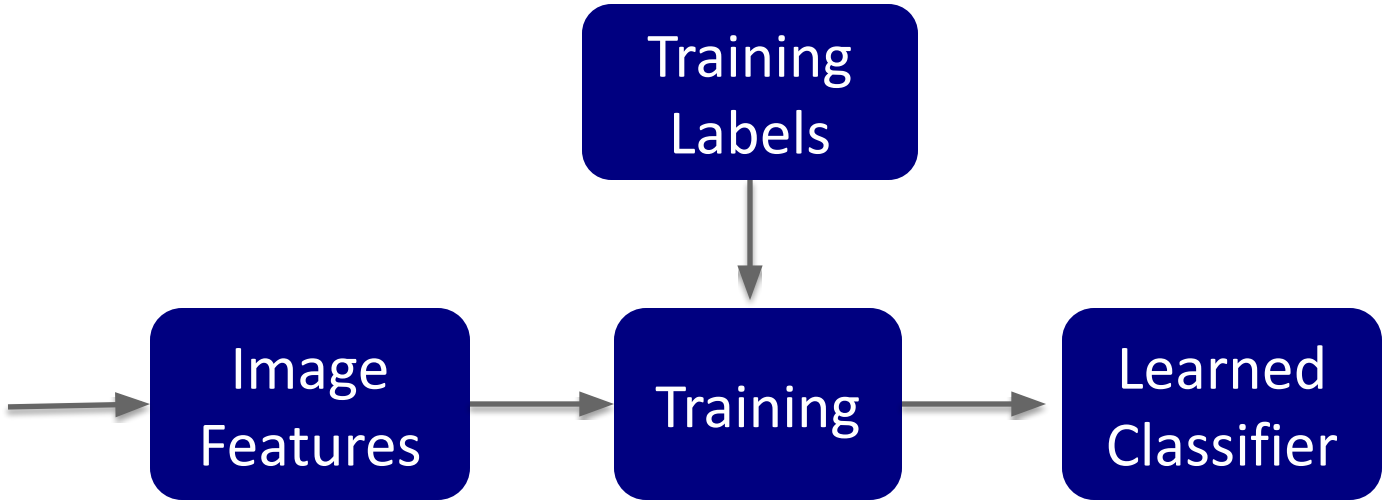
Image Features

A simple pipeline - Training



A simple pipeline - Training

Training Images



A simple pipeline - Training

Training Images

Training Labels

Image Features

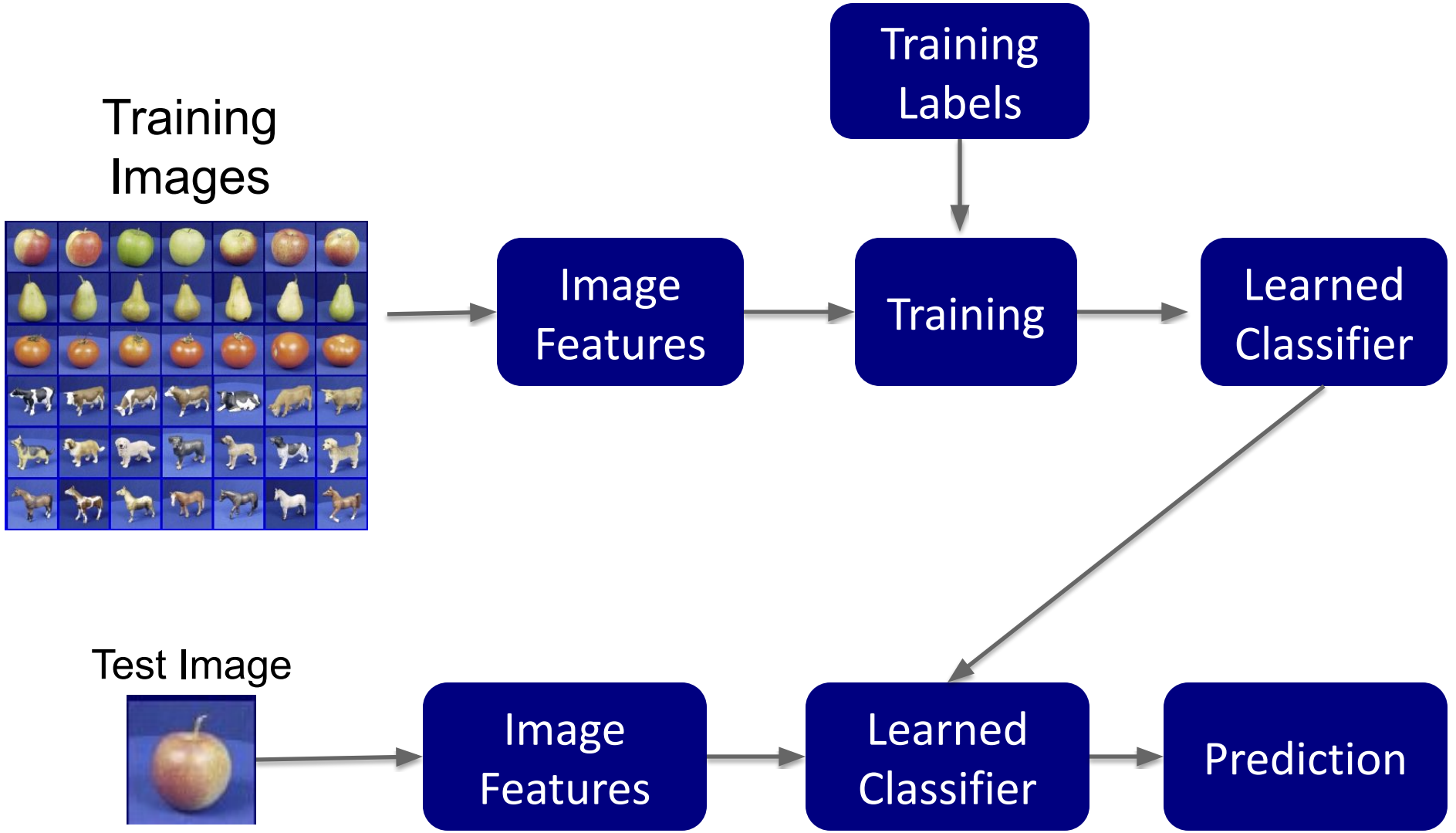
Training

Learned Classifier

Test Image

Image Features

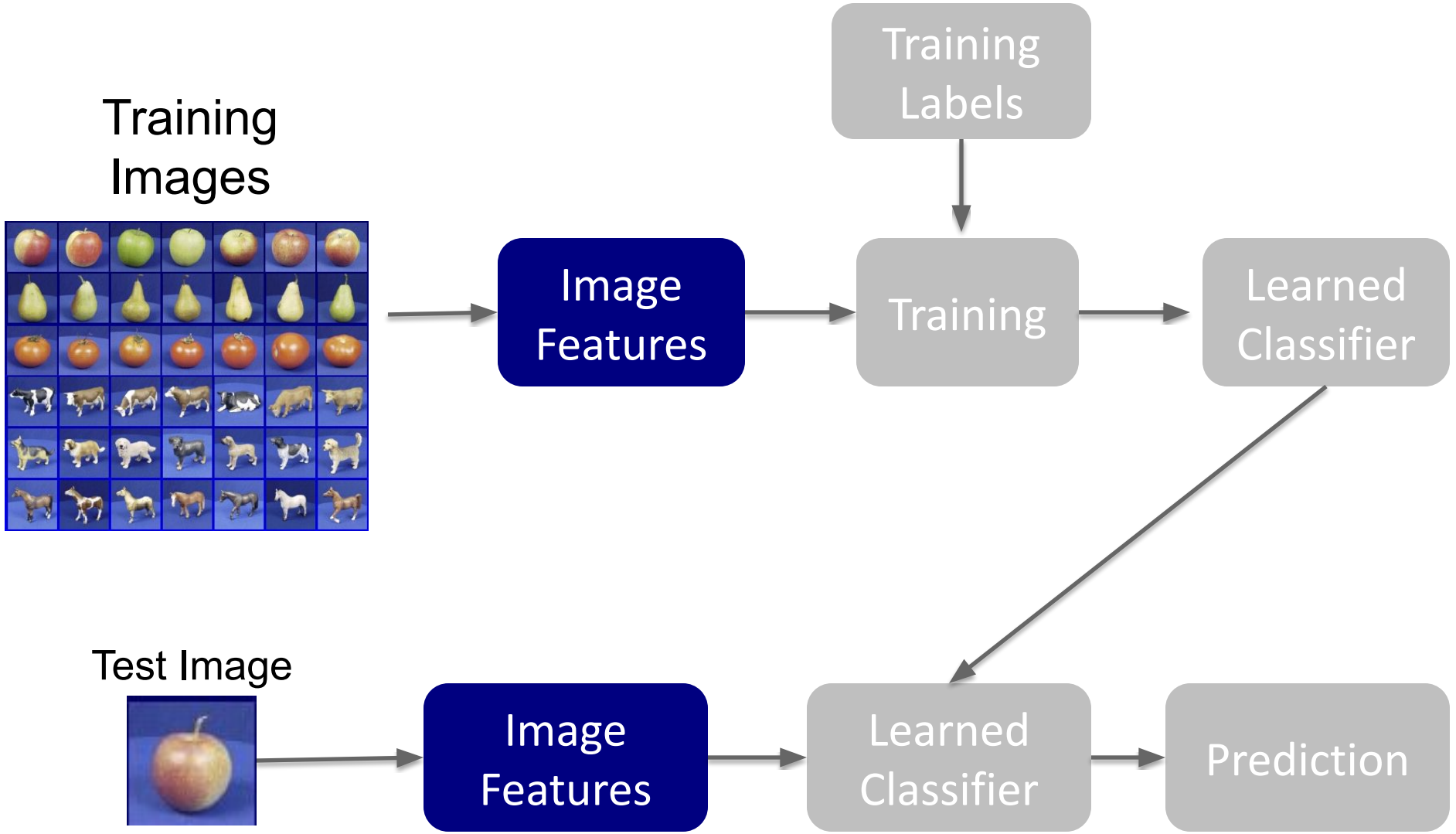
A simple pipeline - Training



What we will learn today?

- Introduction to recognition
- A object recognition pipeline
- Choosing the right features
- A training algorithm: KNN
- Testing an algorithm
- Challenges with kNN
- Dimensionality reduction

A simple pipeline - Training



Choices of features

	Invariances						
	Translation	**Scale**	**Rotation** (relative to camera plane)	**Rotation** (unconstrained)	**Partial Occlusion**	**Illumination**	**Gaussian Noise**
RGB-histogram	**?**						

Choices of features

	Invariances						
	Translation	**Scale**	**Rotation** (relative to camera plane)	**Rotation** (unconstrained)	**Partial Occlusion**	**Illumination**	**Gaussian Noise**
RGB-histogram	✅						

✓ (global color counts don’t change if the image shifts)

Choices of features

	Invariances						
	Translation	**Scale**	**Rotation** (relative to camera plane)	**Rotation** (unconstrained)	**Partial Occlusion**	**Illumination**	**Gaussian Noise**
RGB-histogram	✅	?					

Choices of features

	Invariances						
	Translation	**Scale**	**Rotation** (relative to camera plane)	**Rotation** (unconstrained)	**Partial Occlusion**	**Illumination**	**Gaussian Noise**
RGB-histogram	✅	❌					

✓ (if the *entire* image is uniformly scaled, the color distribution remains the same)

Choices of features

	Invariances						
	Translation	**Scale**	**Rotation** (relative to camera plane)	**Rotation** (unconstrained)	**Partial Occlusion**	**Illumination**	**Gaussian Noise**
RGB-histogram	✅	❌	?	?			

Choices of features

	Invariances						
	Translation	**Scale**	**Rotation** (relative to camera plane)	**Rotation** (unconstrained)	**Partial Occlusion**	**Illumination**	**Gaussian Noise**
RGB-histogram	✅	❌	✅	❌			

✓ (rotating the entire image does not change overall color distribution)
✗ (appearance/colors can change if out-of-plane rotation reveals different surfaces)

Choices of features

	Invariances						
	Translation	**Scale**	**Rotation** (relative to camera plane)	**Rotation** (unconstrained)	**Partial Occlusion**	**Illumination**	**Gaussian Noise**
RGB-histogram	✅	❌	✅	❌	?		

Choices of features

	Invariances						
	Translation	**Scale**	**Rotation** (relative to camera plane)	**Rotation** (unconstrained)	**Partial Occlusion**	**Illumination**	**Gaussian Noise**
RGB-histogram	✅	❌	✅	❌	❌		

✗ (removing part of the image can significantly alter color histogram)

Choices of features

	Invariances						
	Translation	**Scale**	**Rotation** (relative to camera plane)	**Rotation** (unconstrained)	**Partial Occlusion**	**Illumination**	**Gaussian Noise**
RGB-histogram	✅	❌	✅	❌	❌	?	?

Choices of features

	Invariances						
	Translation	**Scale**	**Rotation** (relative to camera plane)	**Rotation** (unconstrained)	**Partial Occlusion**	**Illumination**	**Gaussian Noise**
RGB-histogram	✅	❌	✅	❌	❌	❌	❌

✗ (shifts in illumination change color intensities/distribution)
✗ (noise directly alters pixel distribution)

Choices of features

	Invariances						
	Translation	**Scale**	**Rotation** (relative to camera plane)	**Rotation** (unconstrained)	**Partial Occlusion**	**Illumination**	**Gaussian Noise**
RGB-histogram	✅	❌	✅	❌	❌	❌	❌
HoG	?	?	?	?			

Choices of features

	Invariances						
	Translation	**Scale**	**Rotation** (relative to camera plane)	**Rotation** (unconstrained)	**Partial Occlusion**	**Illumination**	**Gaussian Noise**
RGB-histogram	✅	❌	✅	❌	❌	❌	❌
HoG	❌	❌	❌	❌			

✗ (local bins move)
✗ (needs re-computation at multiple scales)
✗ (oriented gradients are tied to an image grid)
✗ (same reason as the ^)

Choices of features

	Invariances						
	Translation	**Scale**	**Rotation** (relative to camera plane)	**Rotation** (unconstrained)	**Partial Occlusion**	**Illumination**	**Gaussian Noise**
RGB-histogram	✅	❌	✅	❌	❌	❌	❌
HoG	❌	❌	❌	❌	?	?	?

Choices of features

	Invariances						
	Translation	**Scale**	**Rotation** (relative to camera plane)	**Rotation** (unconstrained)	**Partial Occlusion**	**Illumination**	**Gaussian Noise**
RGB-histogram	✅	❌	✅	❌	❌	❌	❌
HoG	❌	❌	❌	❌	❌	✅	❌

✗ (partial occlusion would result in no match)
✓ (gradients are more stable under monotonic intensity changes)
✗ (gradient orientations can be disrupted by significant noise)

Choices of features

	Invariances						
	Translation	**Scale**	**Rotation** (relative to camera plane)	**Rotation** (unconstrained)	**Partial Occlusion**	**Illumination**	**Gaussian Noise**
RGB-histogram	✅	❌	✅	❌	❌	❌	❌
HoG	❌	❌	❌	❌	❌	✅	❌
SIFT	?	?	?	?			

Choices of features

	Invariances						
	Translation	**Scale**	**Rotation** (relative to camera plane)	**Rotation** (unconstrained)	**Partial Occlusion**	**Illumination**	**Gaussian Noise**
RGB-histogram	✅	❌	✅	❌	❌	❌	❌
HoG	❌	❌	❌	❌	❌	✅	❌
SIFT	✅	✅	✅	❌			

✓ (keypoint-based, unaffected by shift)
✓ (built-in scale normalization)
✓ (SIFT normalizes orientation)
✗ (local keypoints might disappear if the object rotates)

Choices of features

	Invariances						
	Translation	**Scale**	**Rotation** (relative to camera plane)	**Rotation** (unconstrained)	**Partial Occlusion**	**Illumination**	**Gaussian Noise**
RGB-histogram	✅	❌	✅	❌	❌	❌	❌
HoG	❌	❌	❌	❌	❌	✅	❌
SIFT	✅	✅	✅	❌	?	?	?

Choices of features

	Invariances						
	Translation	**Scale**	**Rotation** (relative to camera plane)	**Rotation** (unconstrained)	**Partial Occlusion**	**Illumination**	**Gaussian Noise**
RGB-histogram	✅	❌	✅	❌	❌	❌	❌
HoG	❌	❌	❌	❌	❌	✅	❌
SIFT	✅	✅	✅	❌	✅	✅	✅
Deep learning							

✓ (local keypoints can still match if some are visible)
✓ (gradient-based + normalization)
✓ (SIFT is relatively robust to moderate noise)

Choices of features

	Invariances						
	Translation	**Scale**	**Rotation** (relative to camera plane)	**Rotation** (unconstrained)	**Partial Occlusion**	**Illumination**	**Gaussian Noise**
RGB-histogram	✅	❌	✅	❌	❌	❌	❌
HoG	❌	❌	❌	❌	❌	✅	❌
SIFT	✅	✅	✅	❌	✅	✅	✅
Deep learning	?	?	?	?			

Choices of features

	Invariances						
	Translation	**Scale**	**Rotation** (relative to camera plane)	**Rotation** (unconstrained)	**Partial Occlusion**	**Illumination**	**Gaussian Noise**
RGB-histogram	✅	❌	✅	❌	❌	❌	❌
HoG	❌	❌	❌	❌	❌	✅	❌
SIFT	✅	✅	✅	❌	✅	✅	✅
Deep learning	usually	usually	usually	❌			

~ Deep learning features are usually invariant to translation, scale, and planar rotation if the training data has these translations. It is not invariant to other rotations.

Aside: ImageNet has objects centered in the middle of images. So models trained on ImageNet are not translation or scale invariant.

Choices of features

	Invariances						
	Translation	**Scale**	**Rotation** (relative to camera plane)	**Rotation** (unconstrained)	**Partial Occlusion**	**Illumination**	**Gaussian Noise**
RGB-histogram	✅	❌	✅	❌	❌	❌	❌
HoG	❌	❌	❌	❌	❌	✅	❌
SIFT	✅	✅	✅	❌	✅	✅	✅
Deep learning	usually	usually	usually	❌	?	?	?

Choices of features

	Invariances						
	Translation	**Scale**	**Rotation** (relative to camera plane)	**Rotation** (unconstrained)	**Partial Occlusion**	**Illumination**	**Gaussian Noise**
RGB-histogram	✅	❌	✅	❌	❌	❌	❌
HoG	❌	❌	❌	❌	❌	✅	❌
SIFT	✅	✅	✅	❌	✅	✅	✅
Deep learning	usually	usually	usually	❌	❌	✅	✅

✗ (standard CNNs are not strictly occlusion-invariant; partial robustness depends on training)
✓ (can learn robustness if trained on varied lighting)
✓ (CNNs can learn to be noise-robust with proper training)

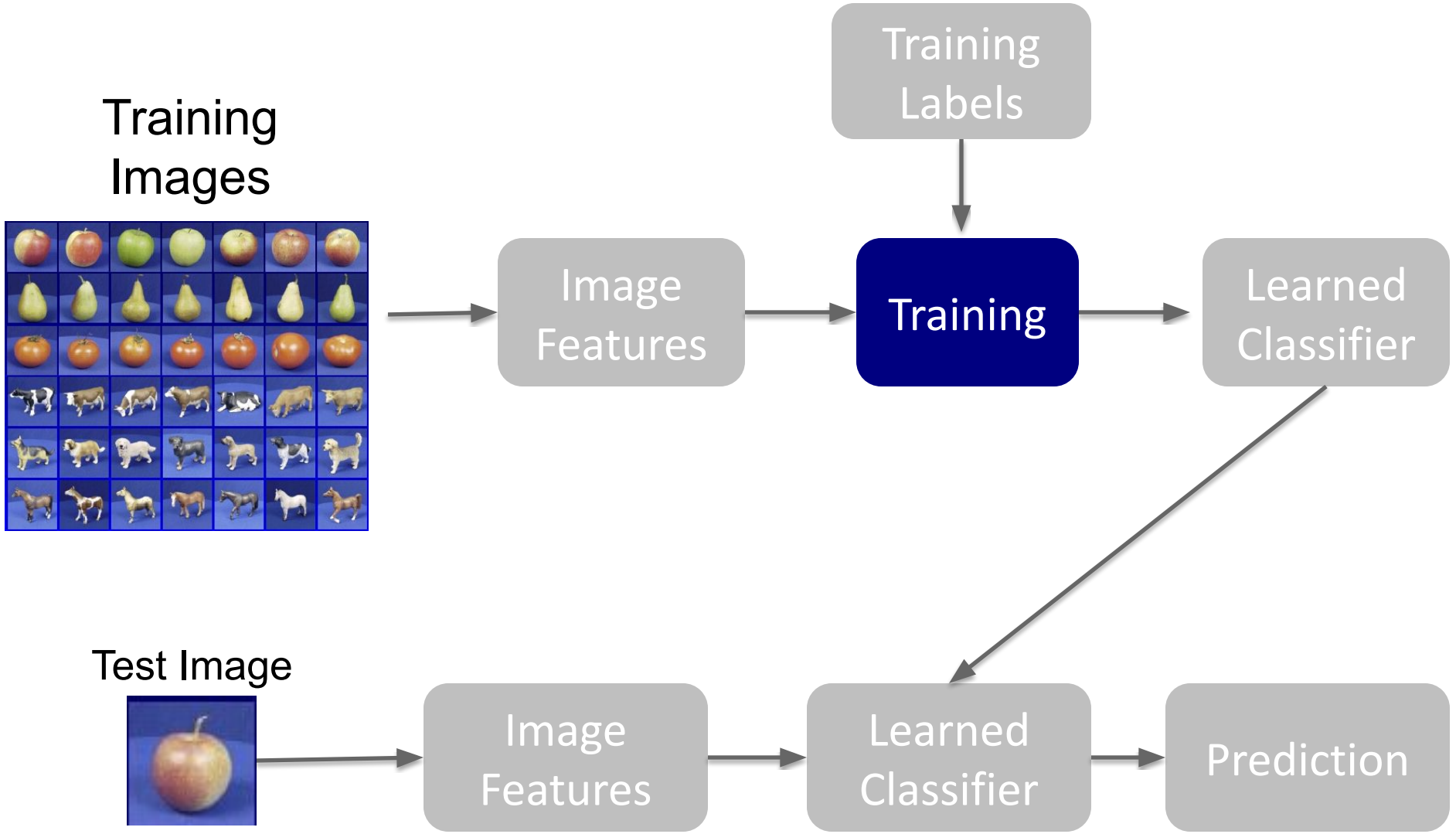
So, which features should we choose?

	Invariances						
	Translation	**Scale**	**Rotation** (relative to camera plane)	**Rotation** (unconstrained)	**Partial Occlusion**	**Illumination**	**Gaussian Noise**
RGB-histogram	✅	❌	✅	❌	❌	❌	❌
HoG	❌	❌	❌	❌	❌	✅	❌
SIFT	✅	✅	✅	❌	✅	✅	✅
Deep learning	usually	usually	usually	❌	❌	✅	✅

What we will learn today?

- Introduction to recognition
- A simple Object Recognition pipeline
- Choosing the right features
- A training algorithm: KNN
- Testing an algorithm
- Challenges with kNN
- Dimensionality reduction

A simple pipeline - Training

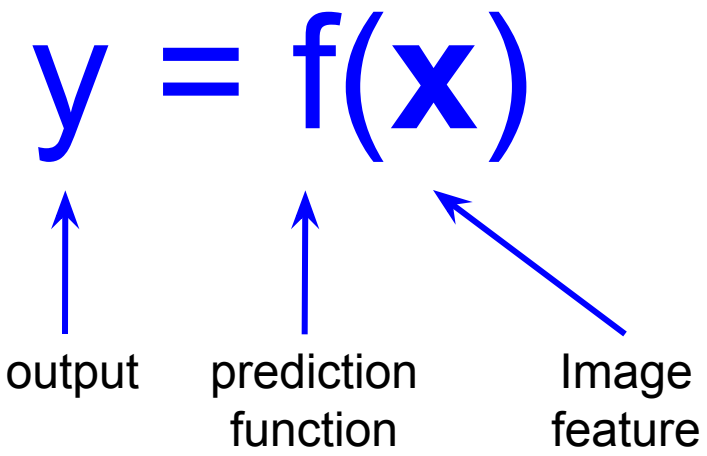


Many classifiers to choose from

- **K-nearest neighbor**
- SVM
- Neural networks
- Naïve Bayes
- Bayesian network
- Logistic regression
- Randomized Forests
- Boosted Decision Trees
- RBMs
- Etc.

Which is the best one?

Learning a classifier to map inputs to outputs



- **Training:** given a *training set* of labeled examples $\{(\mathbf{x}_1,y_1), \ldots, (\mathbf{x}_N,y_N)\}$, estimate the prediction function f by minimizing the prediction error on the training set
- **Testing:** apply f to a never before seen *test example* $\mathbf{x}$ and output the predicted value $y = f(\mathbf{x})$

Slide credit: L. Lazebnik

An example training dataset

Training set (labels known)

Apples

Pear

Tomatos

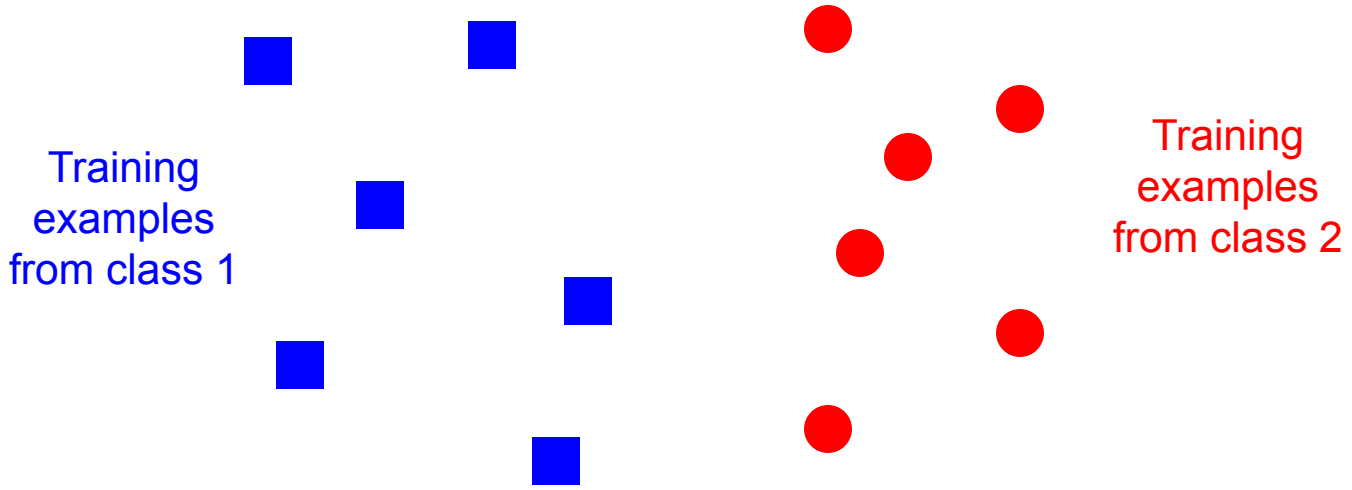
Cow

Dog

Horse

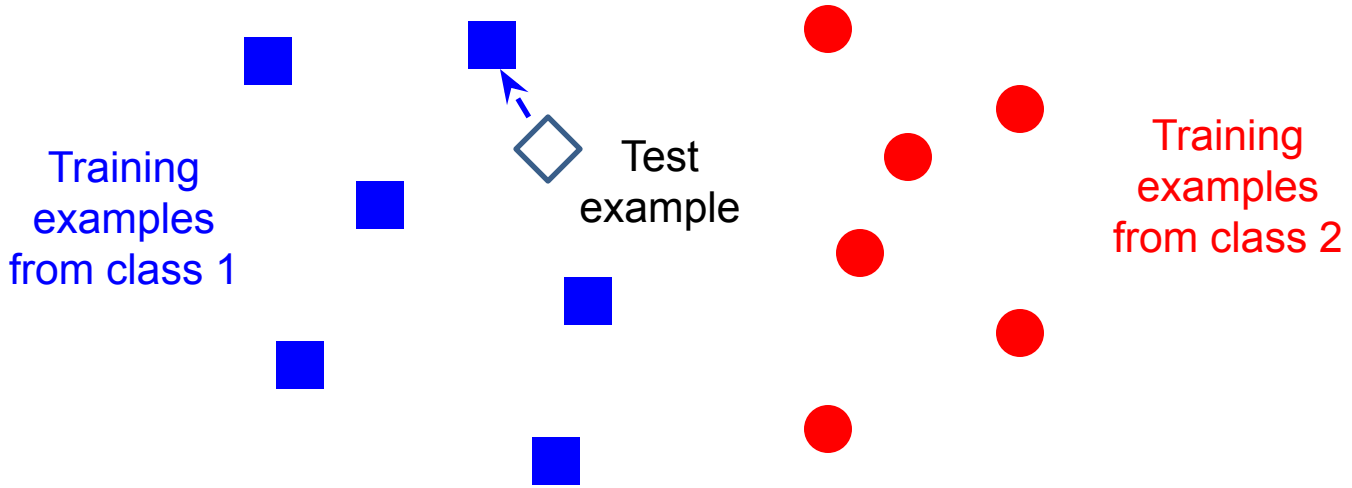
For kNN classifier, training simply means to store all training data.

A stored training set



Slide credit: L. Lazebnik

During testing, we assign the label of the nearest neighbot in feature space

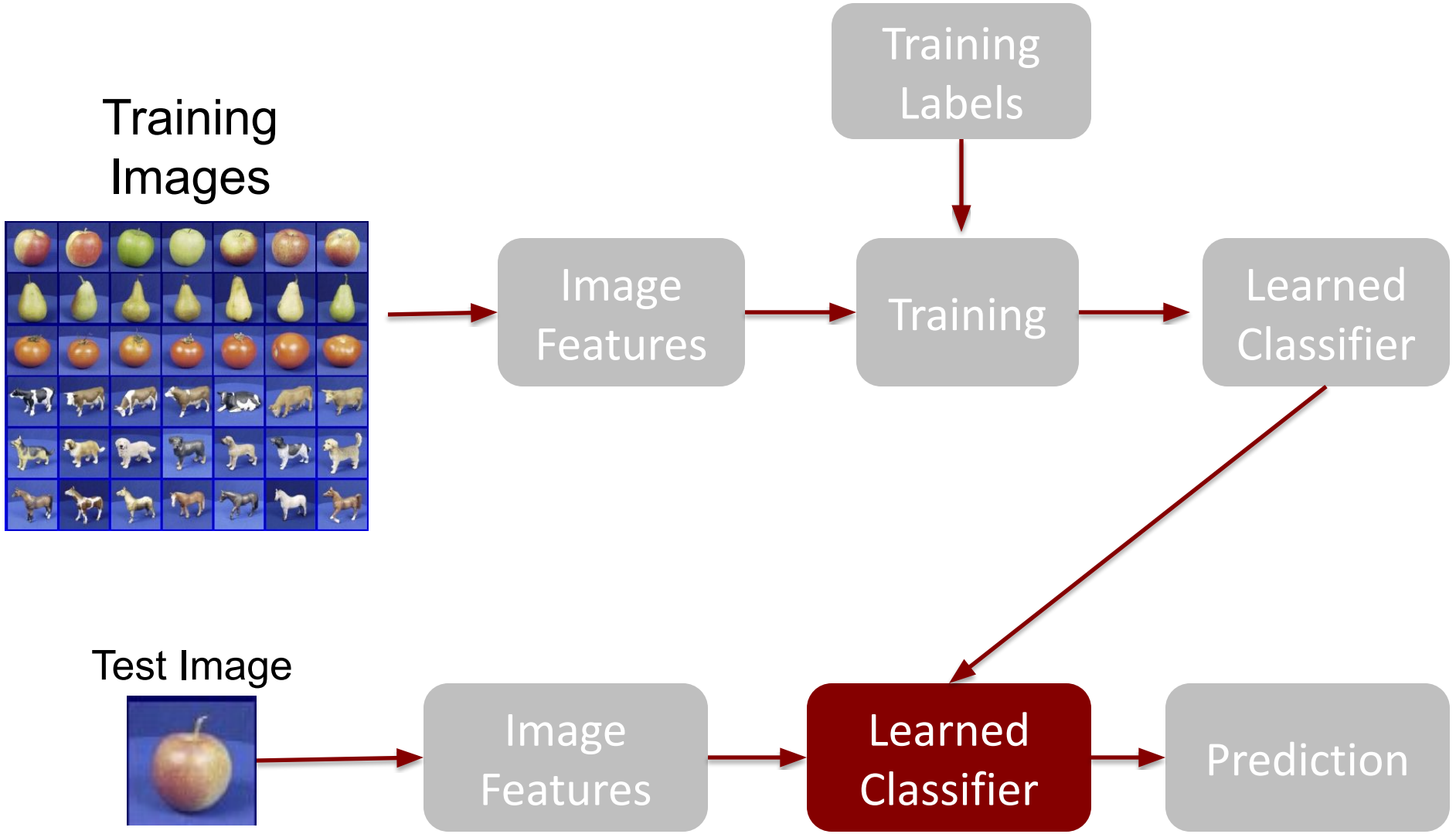


Slide credit: L. Lazebnik

What we will learn today?

- Introduction to recognition
- A simple Object Recognition pipeline
- Choosing the right features
- A training algorithm: kNN
- Testing an algorithm
- Challenges with kNN
- Dimensionality reduction

A simple pipeline - Training



Generalization

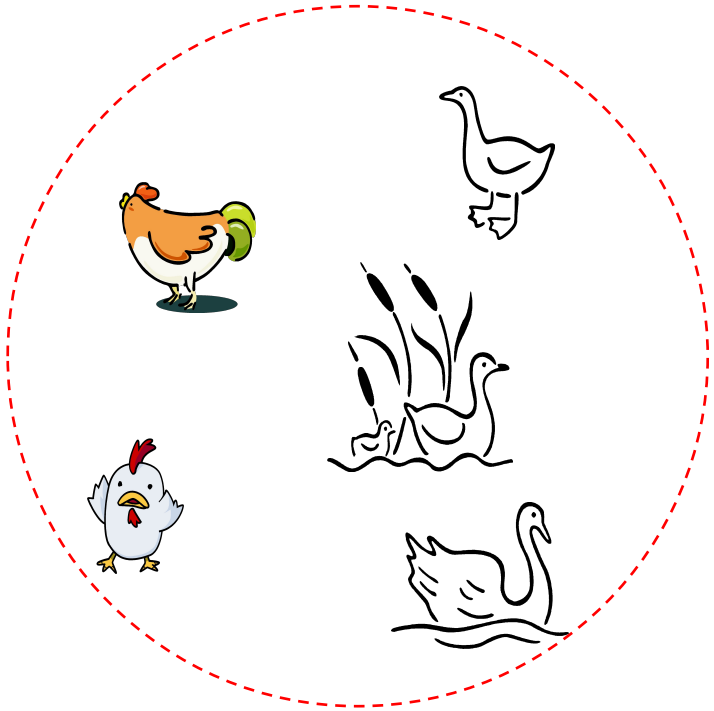
Training set (labels known)

Test set (labels unknown)

- How well does a learned model generalize from the data it was trained on to a new test set?

Intuition for Nearest Neighbor Classifier

Given a training dataset, simply store each image's features and their corresponding label.

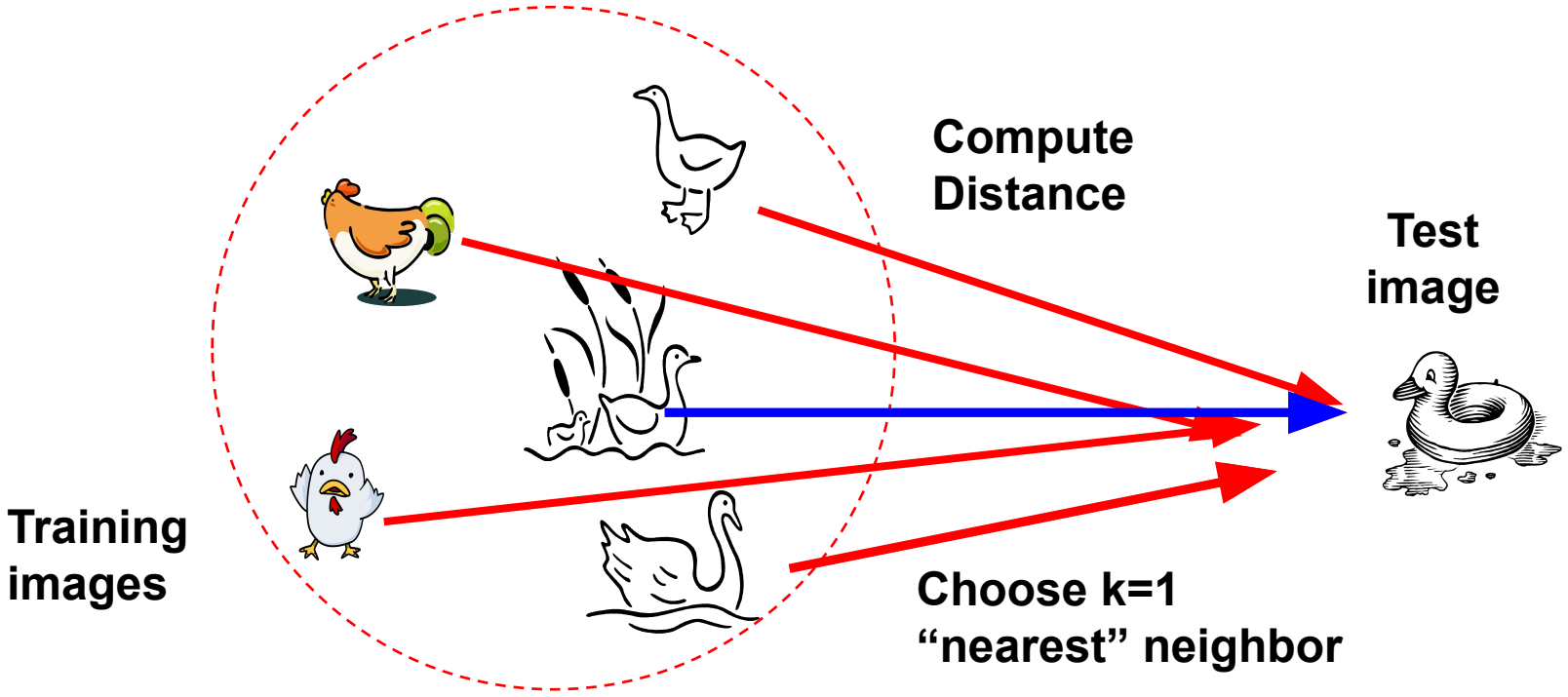


Training images

Source: N. Goyal

Intuition for Nearest Neighbor Classifier

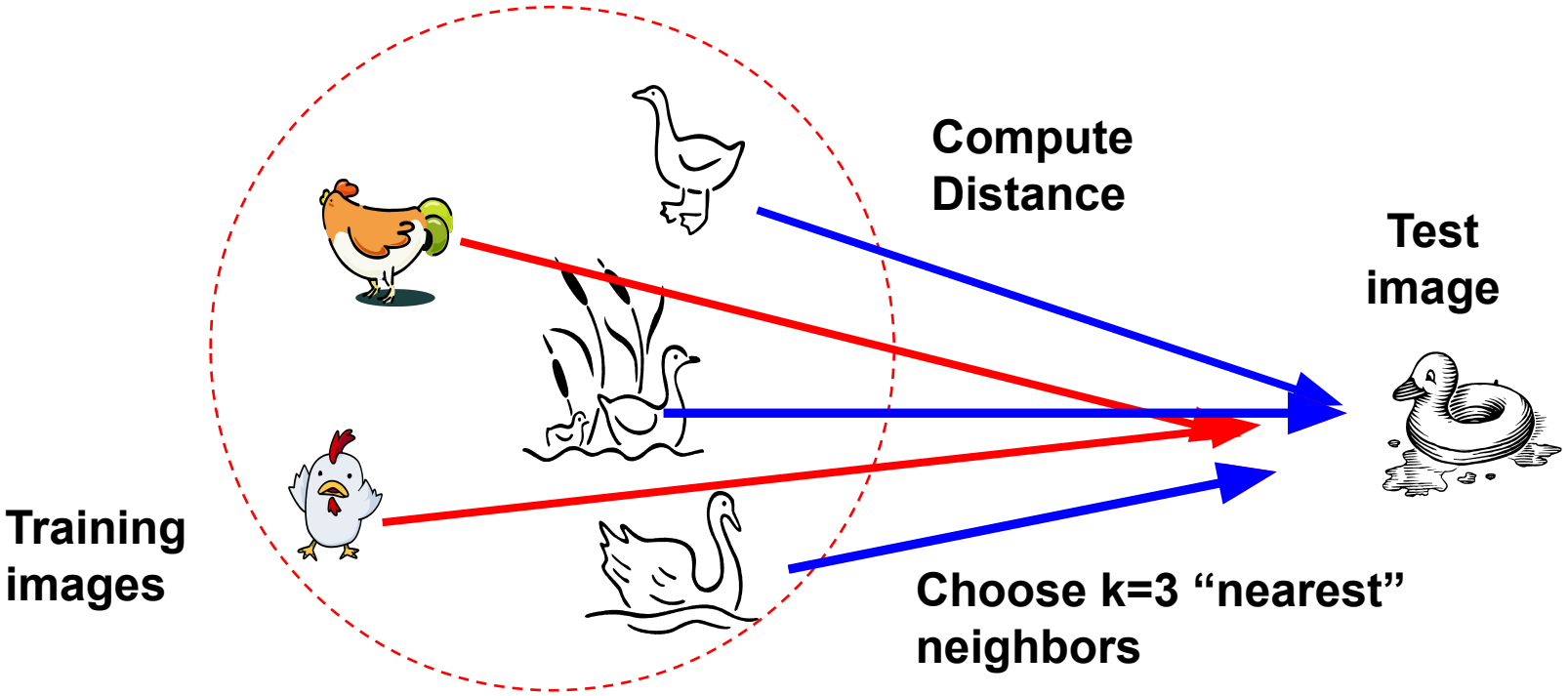
Given a training dataset, simply store each image's features and their corresponding label.



Source: N. Goyal

Nearest Neighbor Classifier

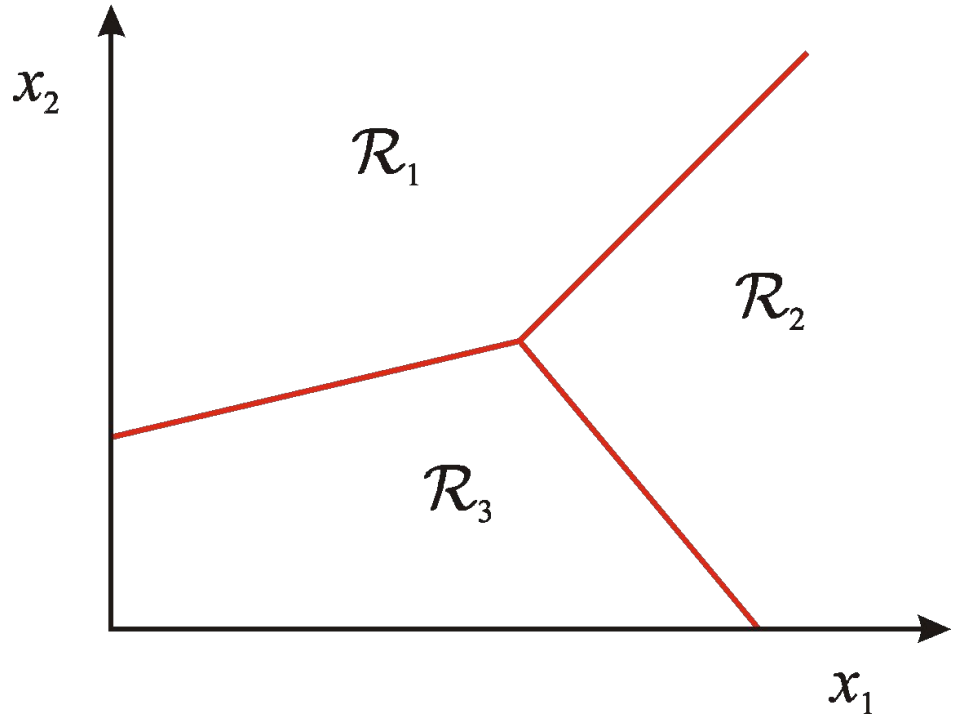
- Assign label of majority of K=3 nearest neighbors



Source: N. Goyal

Classification

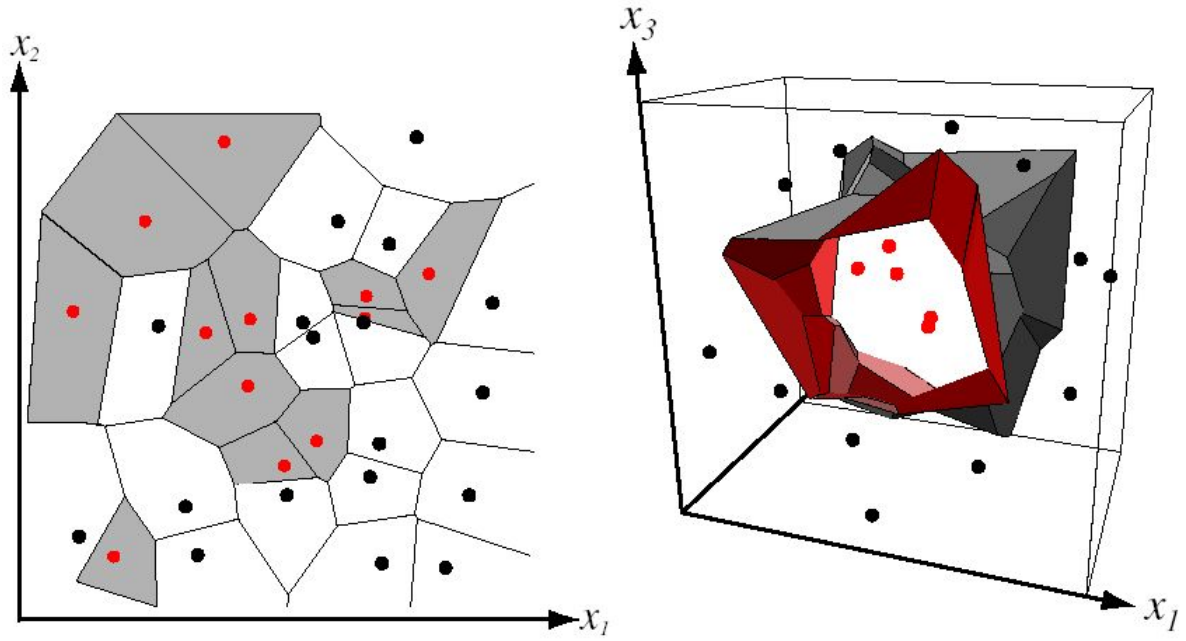
- Assign input vector to one of many classes (categories)
- **Geometric interpretation** of classifiers: A classifier divides input space into *decision regions* separated by *decision boundaries*



Slide credit: S. Lazebnik

Nearest Neighbor Classifier

- Assign label of nearest training data point to each test data point



from Duda *et al.*

Partitioning of feature space
for two-category 2D and 3D data

Source: D. Lowe

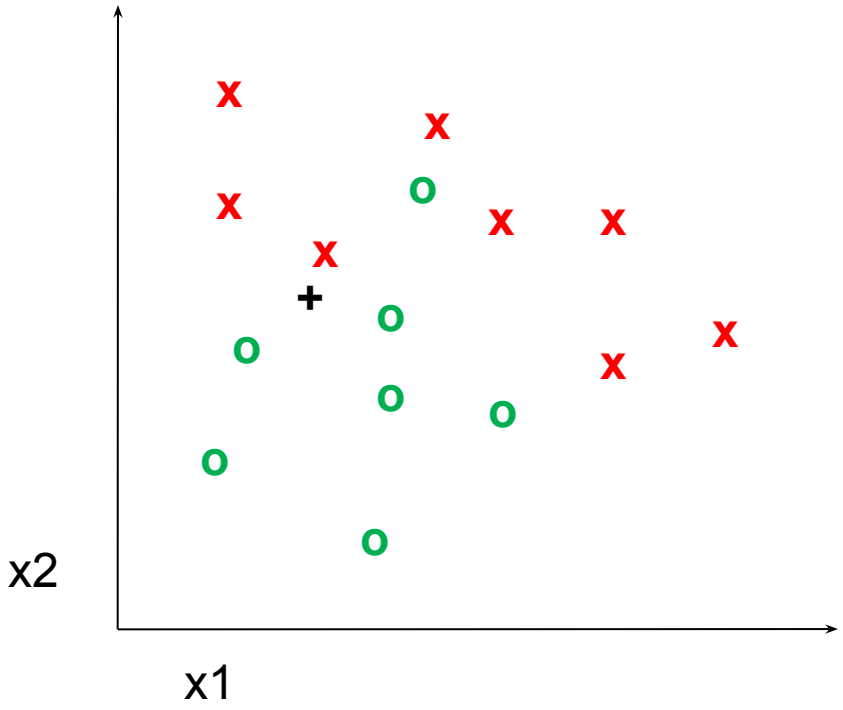
How do we find the nearest neighbors in feature space?

Distance measure (same as the ones from segmentation)

Euclidean:

$$Dist(X^n, X^m) = \sqrt{\sum_{i=1}^{D} (X_i^n - X_i^m)^2}$$

Where X^n and X^m are the n-th and m-th data points



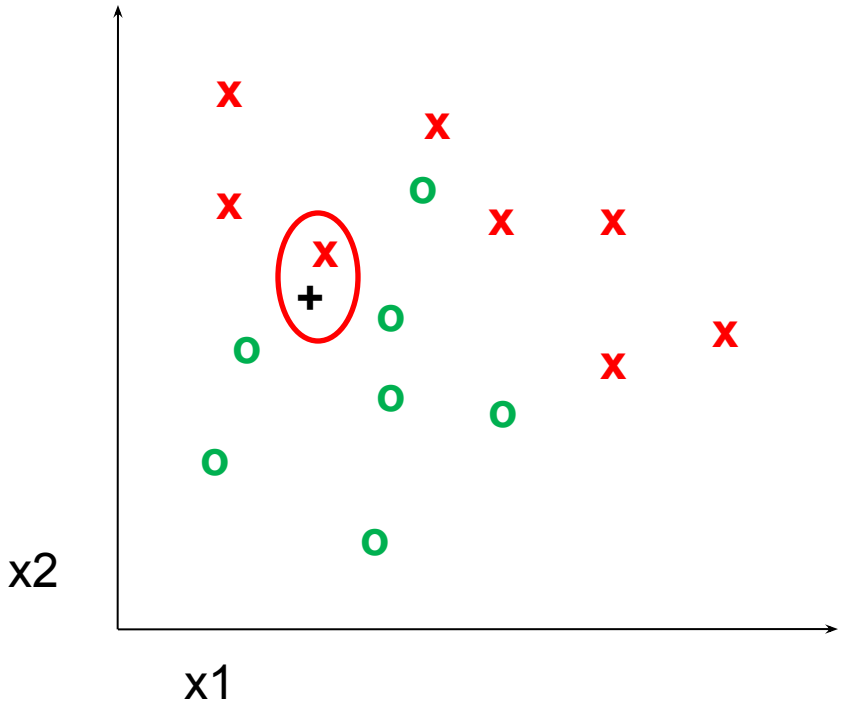
1-nearest neighbor

Distance measure (same as the ones from segmentation)

Euclidean:

$$Dist(X^n, X^m) = \sqrt{\sum_{i=1}^{D} (X_i^n - X_i^m)^2}$$

Where X^n and X^m are the n-th and m-th data points



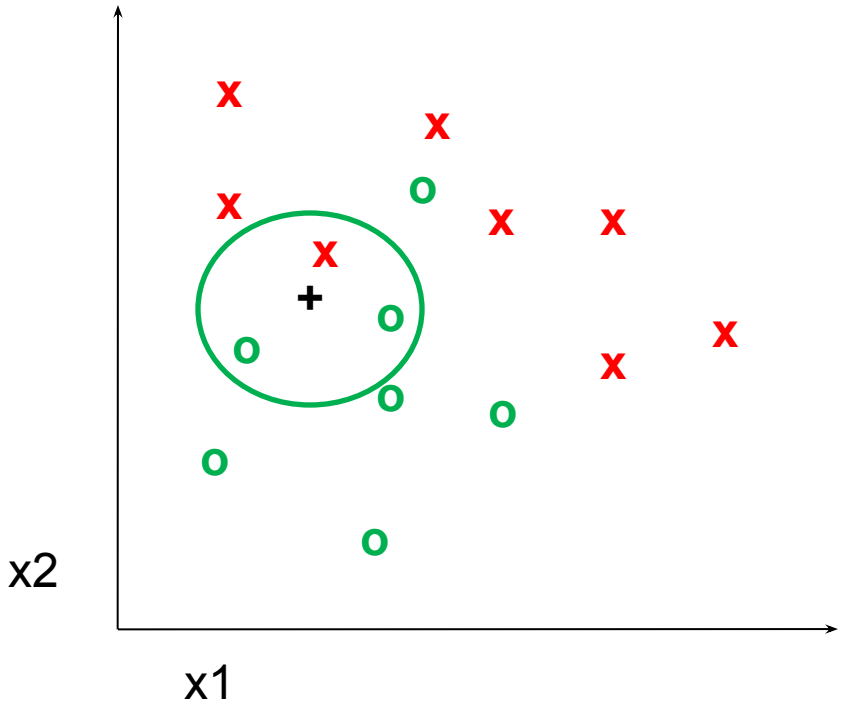
3-nearest neighbor

Distance measure (same as the ones from segmentation)

Euclidean:

$$Dist(X^n, X^m) = \sqrt{\sum_{i=1}^{D} (X_i^n - X_i^m)^2}$$

Where X^n and X^m are the n-th and m-th data points



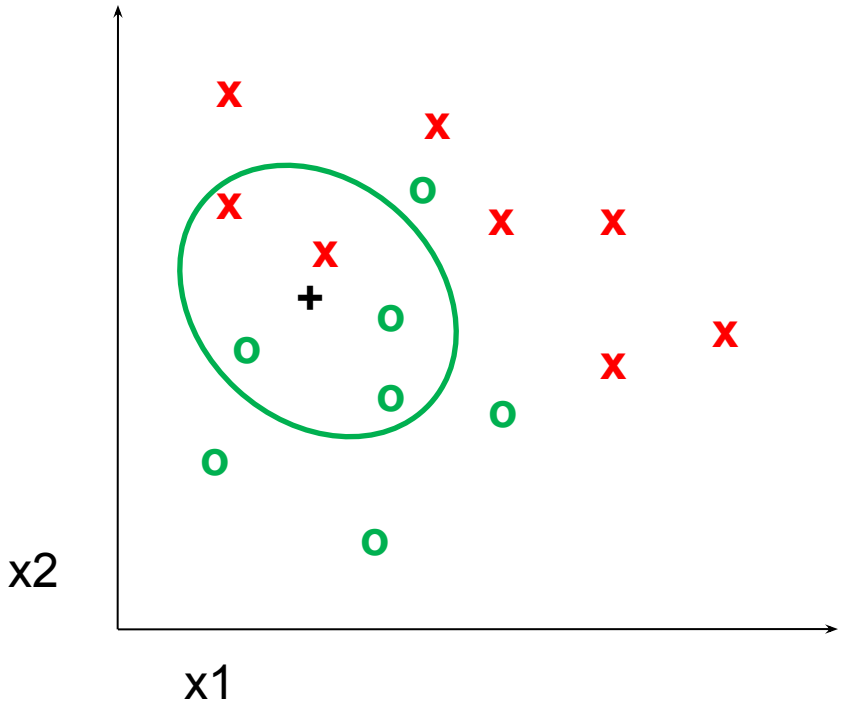
5-nearest neighbor

Distance measure (same as the ones from segmentation)

Euclidean:

$$Dist(X^n, X^m) = \sqrt{\sum_{i=1}^{D} (X_i^n - X_i^m)^2}$$

Where X^n and X^m are the n-th and m-th data points



Choosing the right features is important but dataset-dependent

	Color	$D_x D_y$	Mag-Lap	PCA Masks	PCA Gray	Cont. Greedy	Cont. DynProg	Avg.
apple	57.56%	**85.37%**	80.24%	78.78%	**88.29%**	77.07%	76.34%	77.66%
pear	66.10%	90.00%	85.37%	**99.51%**	**99.76%**	90.73%	91.71%	89.03%
tomato	**98.54%**	94.63%	**97.07%**	67.80%	76.59%	70.73%	70.24%	82.23%
cow	86.59%	82.68%	**94.39%**	75.12%	62.44%	86.83%	86.34%	82.06%
dog	34.63%	62.44%	74.39%	72.20%	66.34%	**81.95%**	**82.93%**	67.84%
horse	32.68%	58.78%	70.98%	77.80%	77.32%	**84.63%**	**84.63%**	69.55%
cup	79.76%	66.10%	77.80%	**96.10%**	**96.10%**	**99.76%**	**99.02%**	87.81%
car	62.93%	**98.29%**	77.56%	**100.0%**	**97.07%**	**99.51%**	**100.0%**	90.77%
total	64.85%	79.79%	82.23%	83.41%	82.99%	86.40%	86.40%	80.87%

Dataset: ETH-80, by B. Leibe, 2003

Results

Category	Primary feature(s)	Secondary feature(s)
apple	PCA Gray	Texture $D_x D_y$
pear	PCA Gray / Masks	
tomato	Color	Texture Mag-Lap
cow	Texture Mag-Lap	Contour / Color
dog	Contour	
horse	Contour	
cup	Contour	PCA Gray / Masks
car	PCA Masks / Contour	Texture $D_x D_y$

Dataset: ETH-80, by B. Leibe, 2003

K-NN: a very useful algorithm

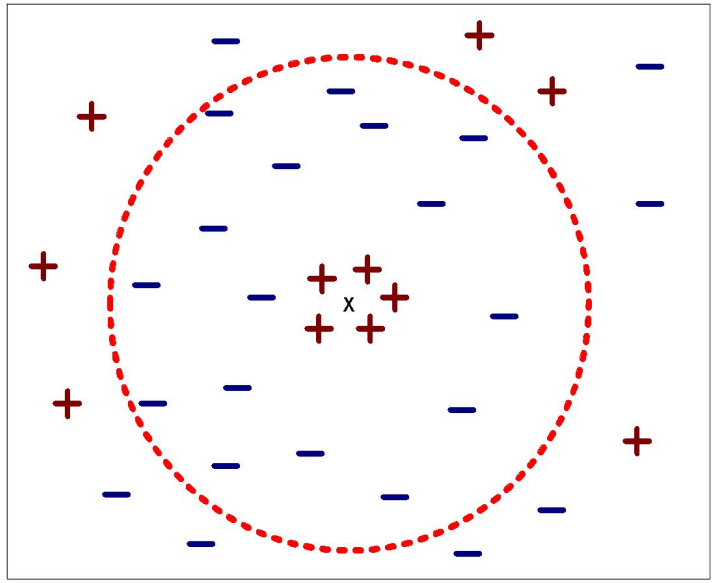
- Simple, a good one to try first
- Very flexible decision boundaries
- With infinite examples, 1-NN has a strong theoretical guarantee (out of scope for this class)

What we will learn today?

- Introduction to recognition
- A simple Object Recognition pipeline
- Choosing the right features
- A training algorithm: kNN
- Testing an algorithm
- Challenges with kNN
- Dimensionality reduction

K-NN: issues to keep in mind

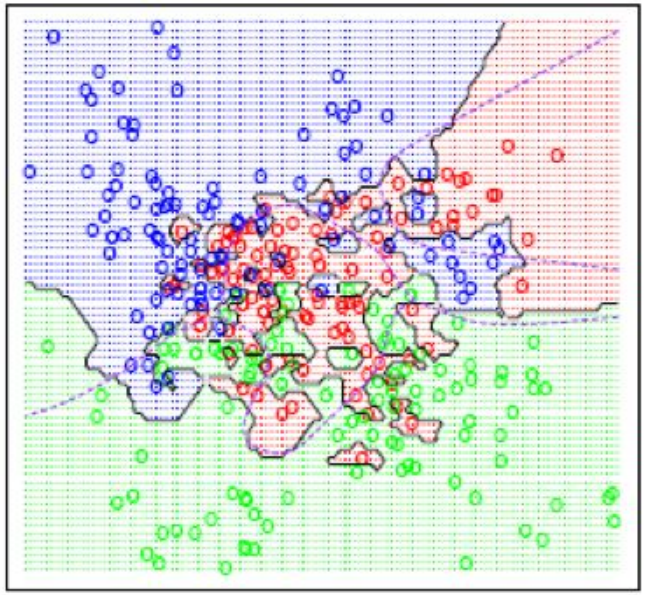
- Choosing the value of k:
  - If too small, sensitive to noise points
  - If too large, neighborhood may include points from other classes



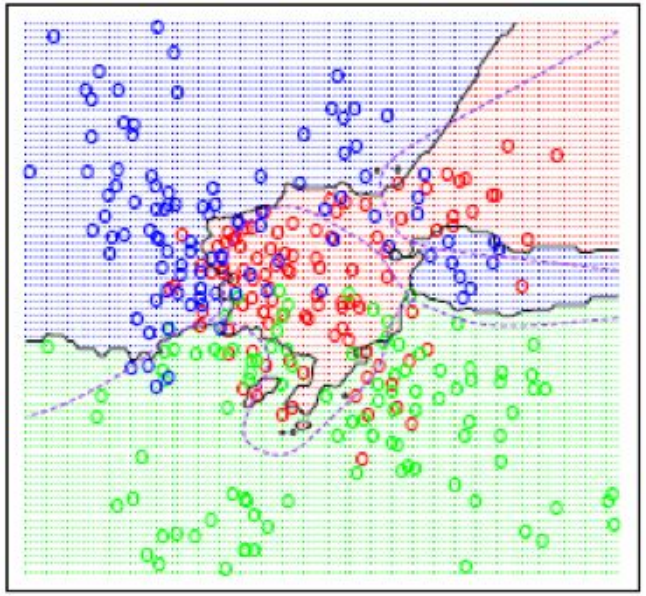
K-NN: issues to keep in mind

- Choosing the value of k:
  - If too small, sensitive to noise points
  - If too large, neighborhood may include points from other classes

K=1

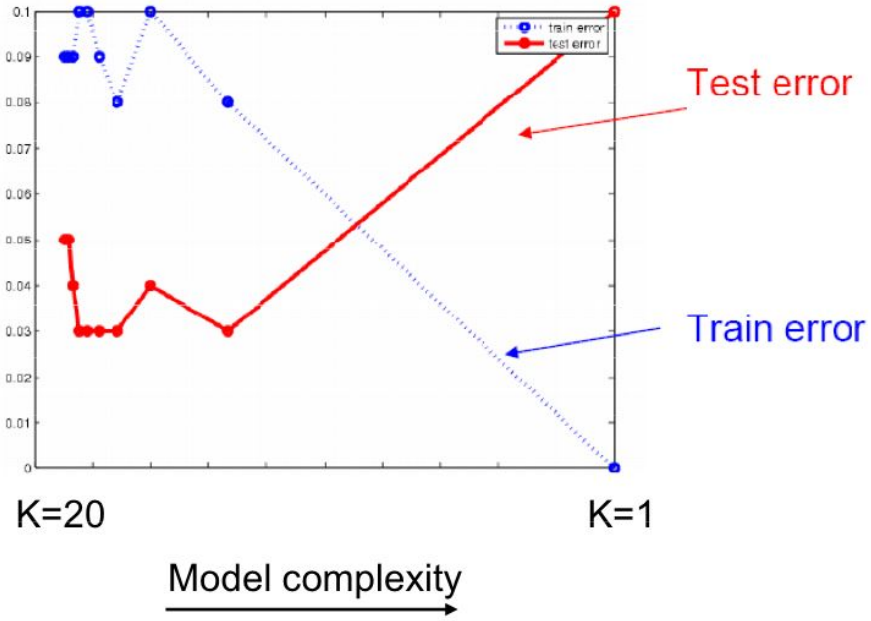


K=15

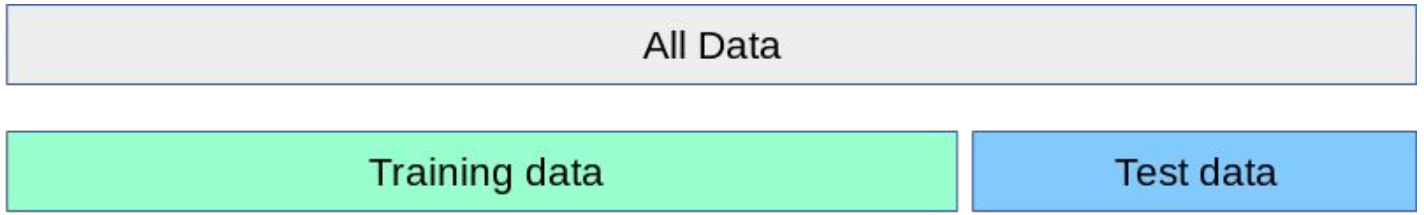


K-NN: issues to keep in mind

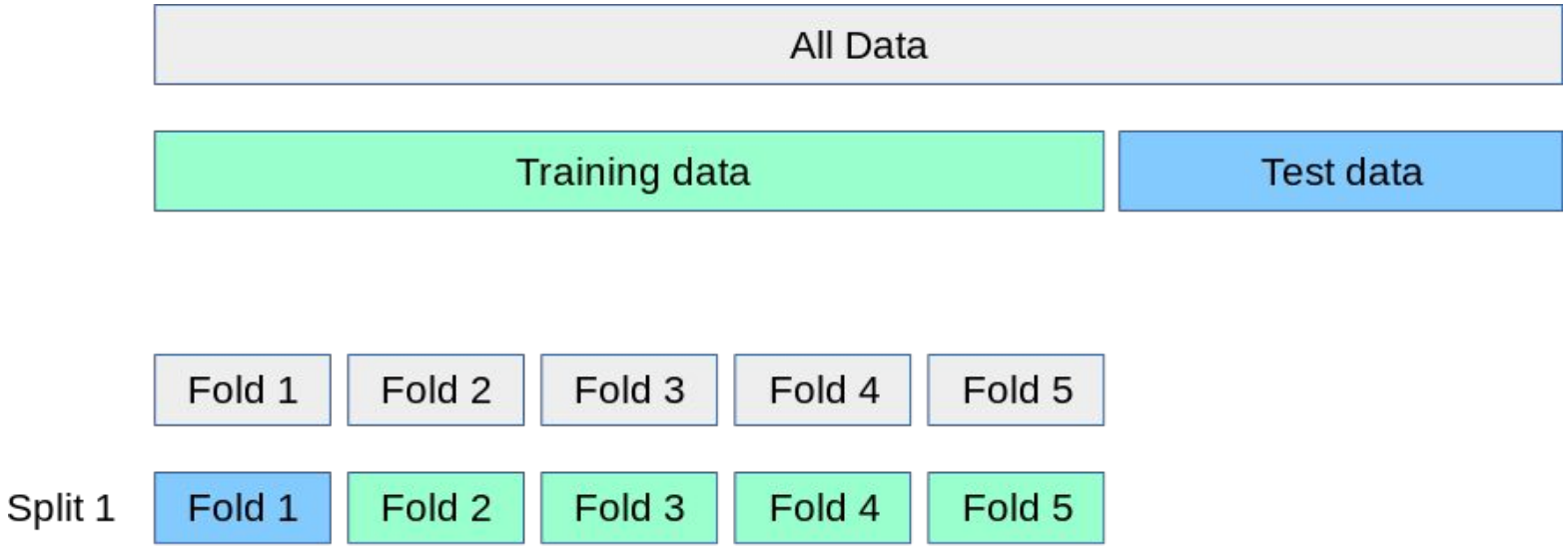
- Choosing the value of k:
  - If too small, sensitive to noise points
  - If too large, neighborhood may include points from other classes
  - Solution: Cross validate



Cross validation

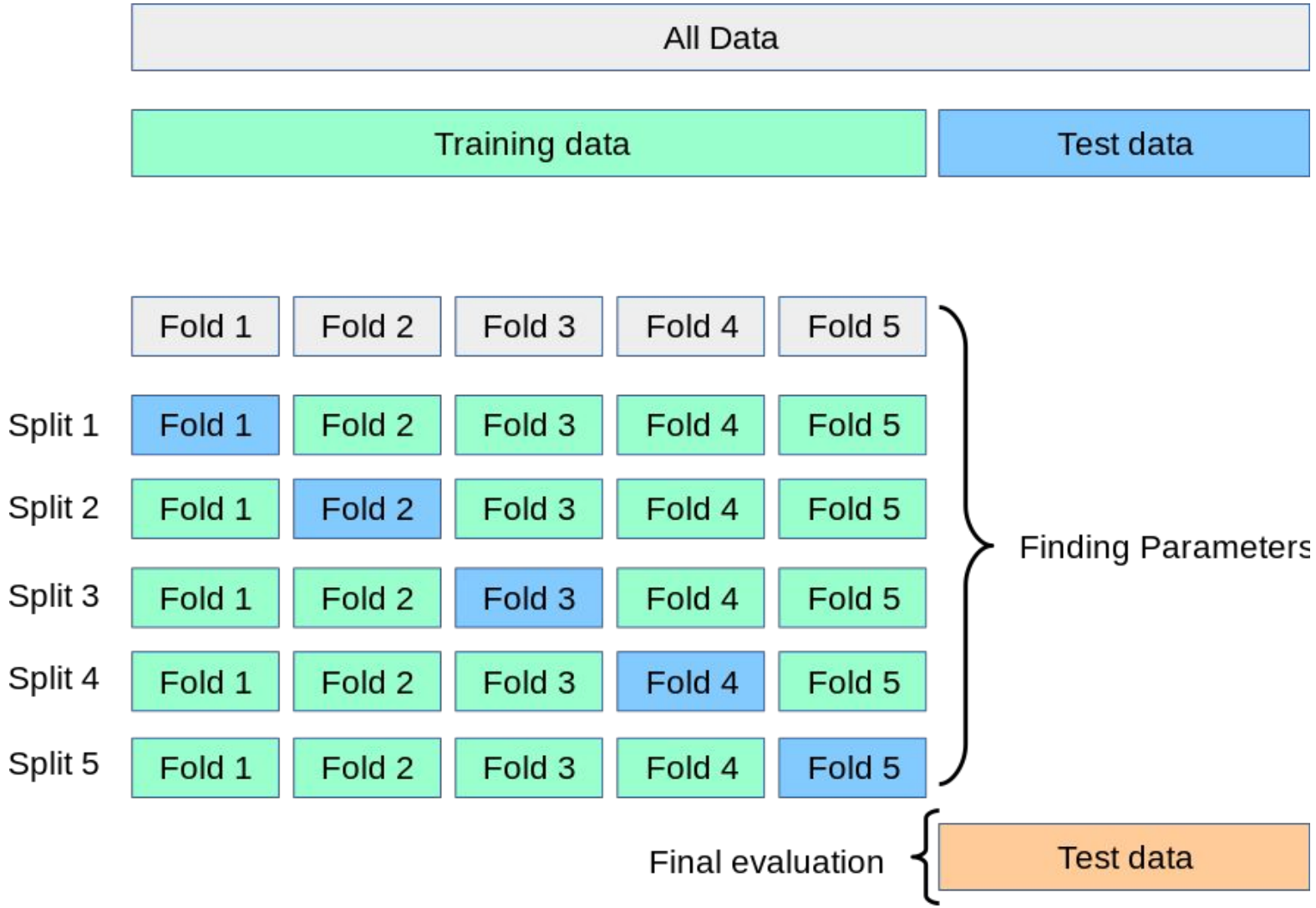


Cross validation



Cross validation

Cross validation



K-NN: issues to keep in mind

- Choosing the value of k:
  - If too small, sensitive to noise points
  - If too large, neighborhood may include points from other classes
  - Solution: cross validate!
- Curse of Dimensionality

Curse of dimensionality

- As the dimensionality increases, the number of data points required for good performance increases exponentially.
- Let's say that for a model to perform well, we need at least 10 data points for each combination of feature values.

Need for Data Points with Increase in Dimensions

1 Binary feature →	2^1 unique values →	$2^1 \times 10 = 20$ data points
2 Binary features →	2^2 unique values →	$2^2 \times 10 = 40$ data points
3 Binary features →	2^3 unique values →	$2^3 \times 10 = 80$ data points
.	.	.
.	.	.
.	.	.
k Binary features →	2^k unique values →	$2^k \times 10$ data points

K-NN: issues to keep in mind

- Choosing the value of k:
  - If too small, sensitive to noise points
  - If too large, neighborhood may include points from other classes
  - Solution: cross validate!
- Curse of Dimensionality
  - Solution: dimensionality reduction

What we will learn today

- Introduction to recognition
- A simple Object Recognition pipeline
- Choosing the right features
- A training algorithm: kNN
- Testing an algorithm
- Challenges with kNN
- Dimensionality reduction

Singular Value Decomposition (SVD)

$$\mathbf{U\Sigma V^T = A}$$

- Where **U** and **V** are rotation matrices, and **Σ** is a scaling matrix. For example:

$$\underset{U}{\begin{bmatrix} -.40 & .916 \\ .916 & .40 \end{bmatrix}} \times \underset{\Sigma}{\begin{bmatrix} 5.39 & 0 \\ 0 & 3.154 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -.05 & .999 \\ .999 & .05 \end{bmatrix}} = \underset{A}{\begin{bmatrix} 3 & -2 \\ 1 & 5 \end{bmatrix}}$$

Singular Value Decomposition (SVD)

- Beyond 2x2 matrices:
  - In general, if **A** is *m* x *n*, then **U** will be *m* x *m*, **Σ** will be *m* x *n*, and $\mathbf{V^T}$ will be *n* x *n*.
  - (Note the dimensions work out to produce *m* x *n* after multiplication)

$$\underset{U}{\begin{bmatrix} -.39 & -.92 \\ -.92 & .39 \end{bmatrix}} \times \underset{\Sigma}{\begin{bmatrix} 9.51 & 0 & 0 \\ 0 & .77 & 0 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} = \underset{A}{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}}$$

Singular Value Decomposition (SVD)

- **U** and **V** are always rotation matrices.
  - Geometric rotation may not be an applicable concept, depending on the matrix. So we call them “unitary” matrices – each column is a unit vector.
- **Σ** is a diagonal matrix
  - The number of nonzero entries = rank of **A**
  - The algorithm always sorts the entries high to low

$$\underset{U}{\begin{bmatrix} -.39 & -.92 \\ -.92 & .39 \end{bmatrix}} \times \underset{\Sigma}{\begin{bmatrix} 9.51 & 0 & 0 \\ 0 & .77 & 0 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} = \underset{A}{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}}$$

SVD Applications

- We've discussed SVD in terms of geometric transformation matrices
- But SVD of an image matrix can also be very useful
- To understand this, we'll look at a less geometric interpretation of what SVD is doing

What is SVD actually doing for images?

$$\underset{U}{\begin{bmatrix} -.39 & -.92 \\ -.92 & .39 \end{bmatrix}} \times \underset{\Sigma}{\begin{bmatrix} 9.51 & 0 & 0 \\ 0 & .77 & 0 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} = \underset{A}{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}}$$

- Look at how the multiplication works out, left to right:
- Column 1 of **U** gets scaled by the first value from **Σ**.

$$\underset{U\Sigma}{\begin{bmatrix} -3.67 & -.71 & 0 \\ -8.8 & .30 & 0 \end{bmatrix}}$$

What is SVD actually doing for images?

$$\underset{U}{\begin{bmatrix} -.39 & -.92 \\ -.92 & .39 \end{bmatrix}} \times \underset{\Sigma}{\begin{bmatrix} 9.51 & 0 & 0 \\ 0 & .77 & 0 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} = \underset{A}{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}}$$

- Look at how the multiplication works out, left to right:
- Column 1 of **U** gets scaled by the first value from **Σ**.

$$\underset{U\Sigma}{\begin{bmatrix} -3.67 & -.71 & 0 \\ -8.8 & .30 & 0 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}}$$

What is SVD actually doing for images?

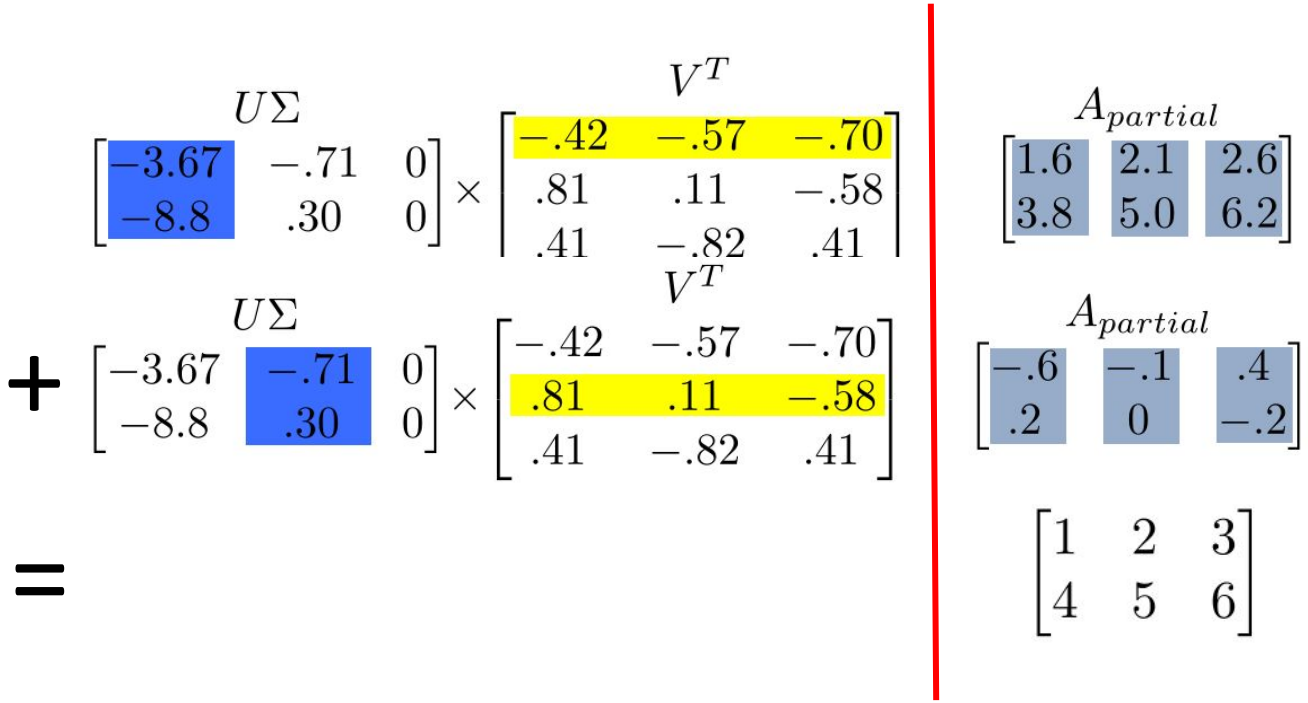
$$\underset{U}{\begin{bmatrix}-.39 & -.92\\ -.92 & .39\end{bmatrix}} \times \underset{\Sigma}{\begin{bmatrix}9.51 & 0 & 0\\ 0 & .77 & 0\end{bmatrix}} \times \underset{V^T}{\begin{bmatrix}-.42 & -.57 & -.70\\ .81 & .11 & -.58\\ .41 & -.82 & .41\end{bmatrix}} = \underset{A}{\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{bmatrix}}$$

- Look at how the multiplication works out, left to right:
- Column 1 of **U** gets scaled by the first value from **Σ**.

$$\underset{U\Sigma}{\begin{bmatrix}-3.67 & -.71 & 0\\ -8.8 & .30 & 0\end{bmatrix}} \times \underset{V^T}{\begin{bmatrix}-.42 & -.57 & -.70\\ .81 & .11 & -.58\\ .41 & -.82 & .41\end{bmatrix}} \quad \underset{A_{partial}}{\begin{bmatrix}1.6 & 2.1 & 2.6\\ 3.8 & 5.0 & 6.2\end{bmatrix}}$$

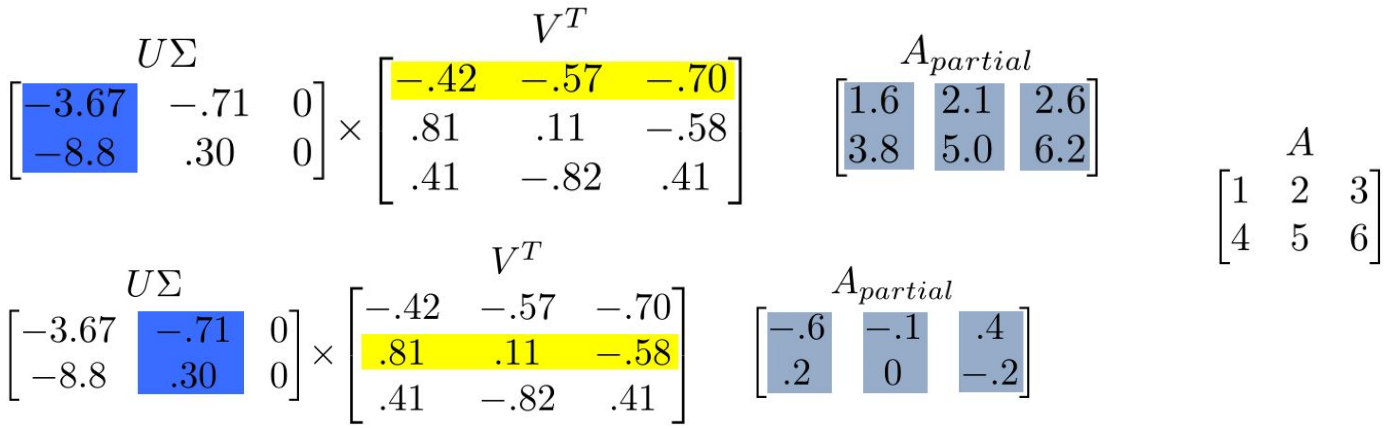
- The resulting vector gets scaled by row 1 of $\mathbf{V^T}$ to produce a contribution to the columns of **A**

SVD is a type dimensionality reduction



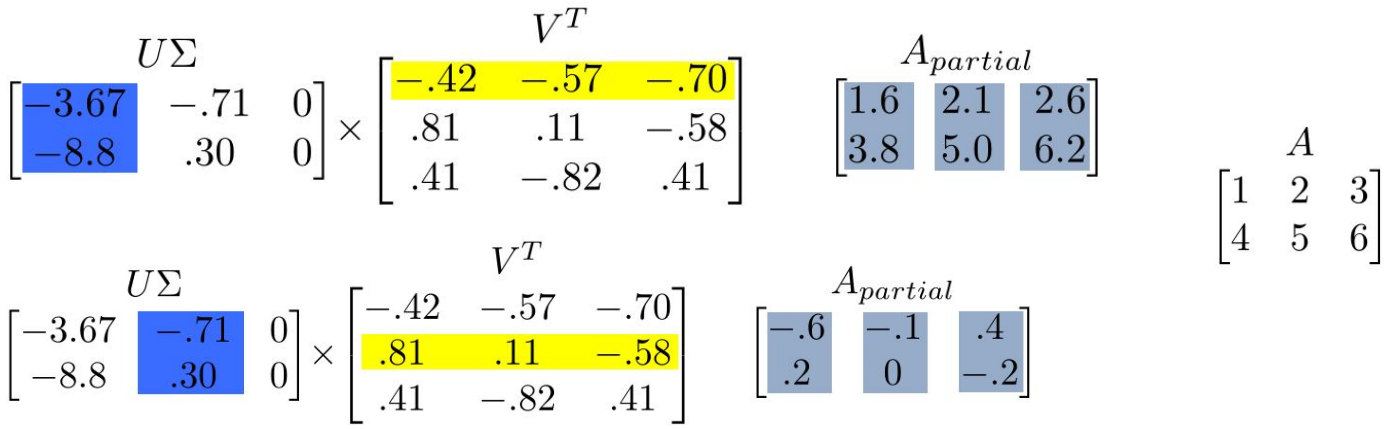
- Each product of (*column i of* ***U***)·(*value i from* ***Σ***)·(*row i of* ***V^T***) produces a component of the final **A**.

SVD is a type dimensionality reduction



- We're building **A** as a linear combination of the columns of ***U***
- Using all columns of ***U***, we'll rebuild the original matrix perfectly
- But, in real-world data, often we can just use the first few columns of ***U*** and we'll get something close (e.g. the first $\boldsymbol{A_{partial}}$, above)

SVD is a type dimensionality reduction



- We can call those first few columns of ***U*** the Principal Components of the data
- They show the major patterns that can be added to produce the columns of the original matrix
- The rows of $\mathbf{V^T}$ show how the *principal components* are mixed to produce the columns of the matrix

SVD is a type dimensionality reduction

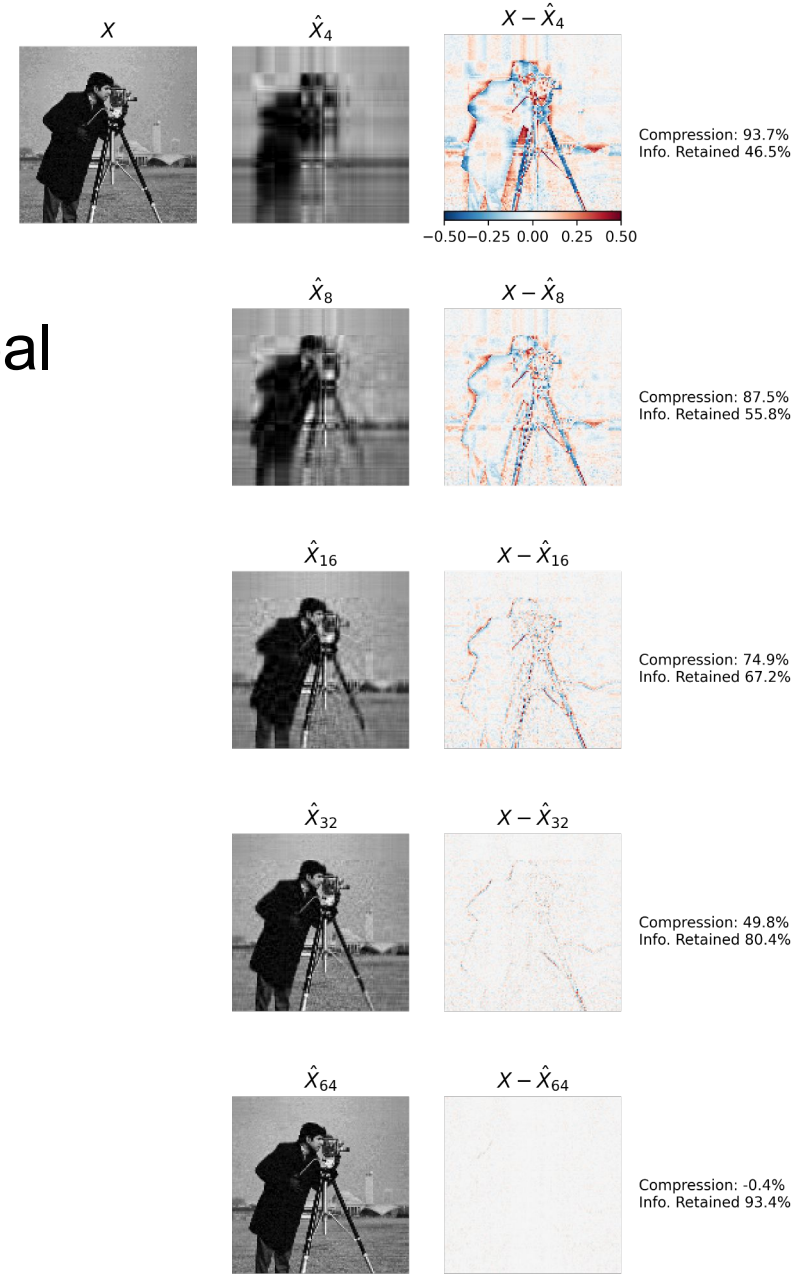
$$\underset{U}{\begin{bmatrix} -.39 & -.92 \\ -.92 & .39 \end{bmatrix}} \times \underset{\Sigma}{\begin{bmatrix} 9.51 & 0 & 0 \\ 0 & .77 & 0 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} = \underset{A}{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}}$$

We can look at **Σ** to see that the first column has a large effect

while the second column has a much smaller effect in this example

Image compression

- For this image, using **only the first 16** of 300 principal components produces a recognizable reconstruction
- Using the first 64 almost perfectly reconstructs the image



SVD for symmetric matrices

- If A is a symmetric matrix, it can be decomposed as the following:

- Compared to a traditional $A = \Phi\Sigma\Phi^T$ V^T and is an orthogonal matrix.

What we have learned today?

- Introduction to recognition
- A simple Object Recognition pipeline
- Choosing the right features
- A training algorithm: kNN
- Testing an algorithm
- Challenges with kNN
- Dimensionality reduction

Next lecture

PCA + LDA

Extra slides (out of scope)

for those of you curious about how SVD is calculated
and what PCA is usually used for outside of computer vision

Principal Component Analysis

$$\underset{U\Sigma}{\begin{bmatrix} -3.67 & -.71 & 0 \\ -8.8 & .30 & 0 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} \quad \underset{A_{partial}}{\begin{bmatrix} 1.6 & 2.1 & 2.6 \\ 3.8 & 5.0 & 6.2 \end{bmatrix}}$$

- Remember, columns of ***U*** are the *Principal Components* of the data: the major patterns that can be added to produce the columns of the original matrix
- One use of this is to construct a matrix where each column is a separate data sample
- Run SVD on that matrix, and look at the first few columns of ***U*** to see patterns that are common among the columns
- This is called *Principal Component Analysis* (or PCA) of the data samples

Principal Component Analysis

$$\underset{U\Sigma}{\begin{bmatrix} -3.67 & -.71 & 0 \\ -8.8 & .30 & 0 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} \quad \underset{A_{partial}}{\begin{bmatrix} 1.6 & 2.1 & 2.6 \\ 3.8 & 5.0 & 6.2 \end{bmatrix}}$$

- Often, raw data samples have a lot of redundancy and patterns
- PCA can allow you to represent data samples as weights on the principal components, rather than using the original raw form of the data
- By representing each sample as just those weights, you can represent just the “meat” of what’s different between samples.
- This minimal representation makes machine learning and other algorithms much more efficient

How is SVD computed?

- For this class: tell PYTHON to do it. Use the result.
- But, if you're interested, one computer algorithm to do it makes use of Eigenvectors!

Eigenvector definition

- Suppose we have a square matrix **A**. We can solve for vector x and scalar λ such that Ax= λx
- In other words, find vectors where, if we transform them with **A**, the only effect is to scale them with no change in direction.
- These vectors are called eigenvectors (German for “self vector” of the matrix), and the scaling factors λ are called eigenvalues
- An *m* x *m* matrix will have ≤ *m* eigenvectors where λ is nonzero

Finding eigenvectors

- Computers can find an x such that $Ax = \lambda x$ using this iterative algorithm:
  - X = random unit vector
  - while(x hasn't converged)
    - X = Ax
    - normalize x
- x will quickly converge to an eigenvector
- Some simple modifications will let this algorithm find all eigenvectors

Finding SVD

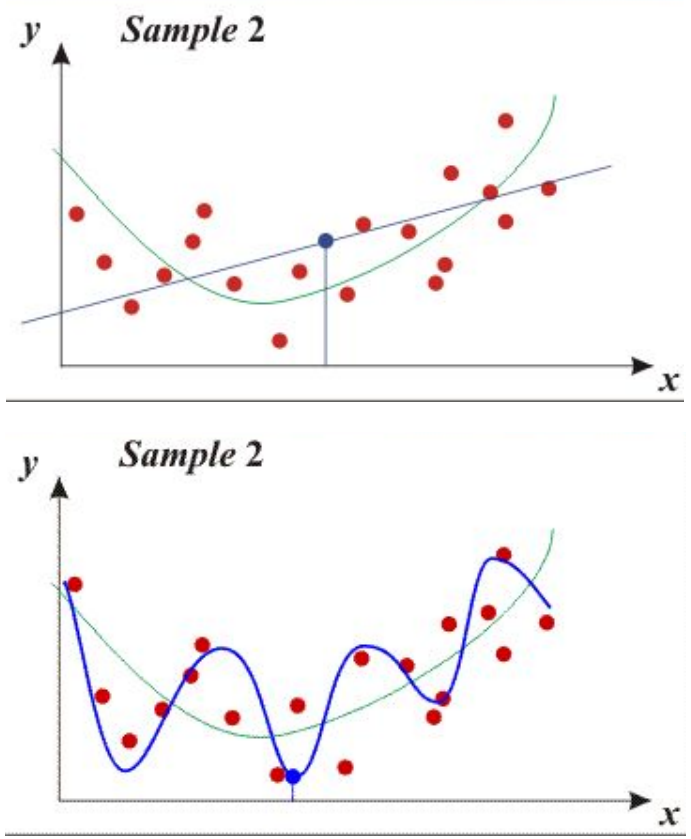
- Eigenvectors are for square matrices, but SVD is for all matrices
- To do svd(A), computers can do this:
  - Take eigenvectors of AA^T (matrix is always square).
    - These eigenvectors are the columns of **U**.
    - Square root of eigen*values* are the singular values (the entries of **Σ**).
  - Take eigenvectors of A^TA (matrix is always square).
    - These eigenvectors are columns of **V** (or rows of $\mathbf{V^T}$)

Finding SVD

- Moral of the story: SVD is fast, even for large matrices
- It's useful for a lot of stuff
- There are also other algorithms to compute SVD or part of the SVD
  - Python's np.linalg.svd() command has options to efficiently compute only what you need, if performance becomes an issue

A detailed geometric explanation of SVD is here:
http://www.ams.org/samplings/feature-column/fcarc-svd

Bias-Variance Trade-off



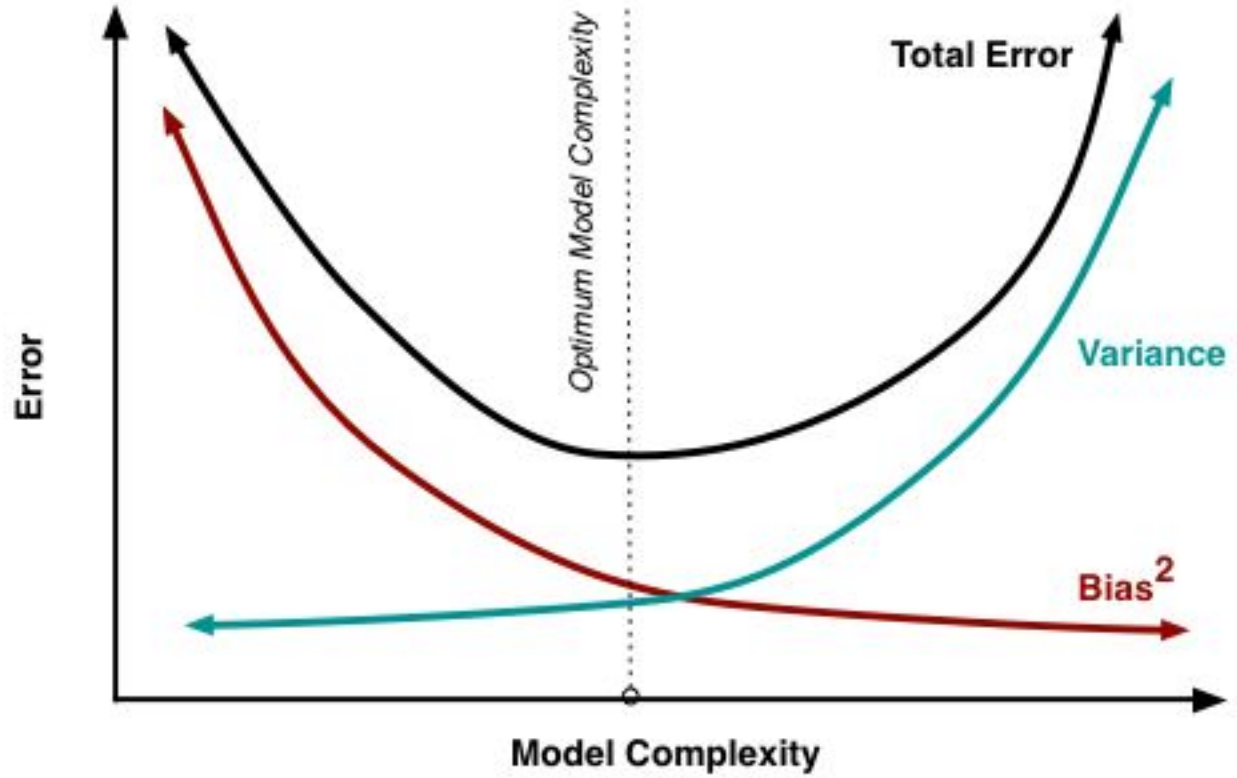
- Models with too few parameters are inaccurate because of a large bias (not enough flexibility).
- Models with too many parameters are inaccurate because of a large variance (too much sensitivity to the sample).

Slide credit: D. Hoiem

Bias versus variance

- Components of generalization error
  - **Bias:** how much the average model over all training sets differ from the true model?
    - Error due to inaccurate assumptions/simplifications made by the model
  - **Variance:** how much models estimated from different training sets differ from each other
- **Underfitting:** model is too “simple” to represent all the relevant class characteristics
  - High bias and low variance
  - High training error and high test error
- **Overfitting:** model is too “complex” and fits irrelevant characteristics (noise) in the data
  - Low bias and high variance
  - Low training error and high test error

Bias versus variance trade off



No Free Lunch Theorem

In a supervised learning setting, we can't tell which classifier will have best generalization

Slide credit: D. Hoiem

Remember…

- No classifier is inherently better than any other: you need to make assumptions to generalize

- Three kinds of error
  - Inherent: unavoidable
  - Bias: due to over-simplifications
  - Variance: due to inability to perfectly estimate parameters from limited data

Slide credit: D. Hoiem

How to reduce variance?

- Choose a simpler classifier
- Regularize the parameters
- Get more training data

How do you reduce bias?

Slide credit: D. Hoiem

Last remarks about applying machine learning methods to object recognition

- There are machine learning algorithms to choose from
- Know your data:
  - How much supervision do you have?
  - How many training examples can you afford?
  - How noisy?
- Know your goal (i.e. task):
  - Affects your choices of representation
  - Affects your choices of learning algorithms
  - Affects your choices of evaluation metrics
- Understand the math behind each machine learning algorithm under consideration!

PCA by SVD

- An alternative manner to compute the principal components, based on singular value decomposition
- Quick reminder: SVD
  - Any real n x m matrix (n>m) can be decomposed as

$$A = M \Pi N^T$$

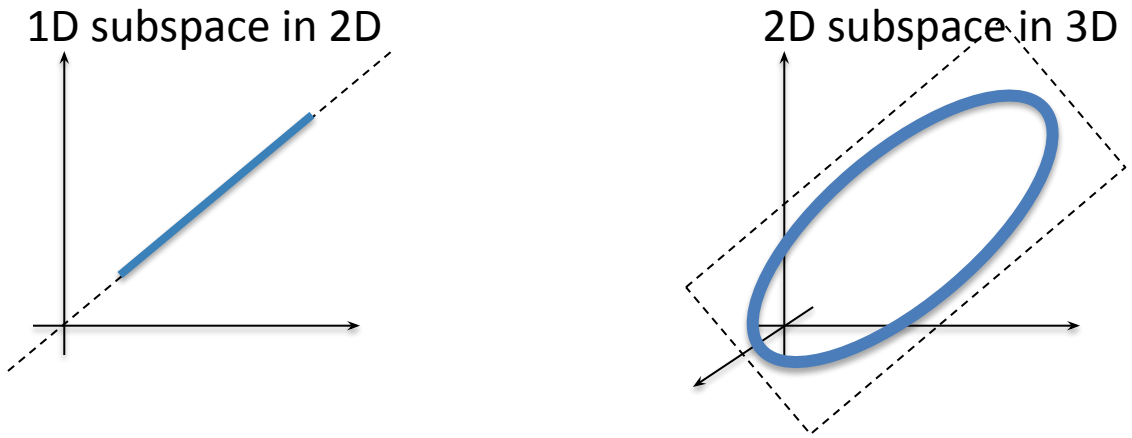
  - Where M is an (n x m) column orthonormal matrix of left singular vectors (columns of M)
  - Π is an (m x m) diagonal matrix of singular values
  - N^T is an (m x m) row orthonormal matrix of right singular vectors (columns of N)

$$M^T M = I \qquad N^T N = I$$

Slide inspired by N. Vasconcelos

PCA Formulation

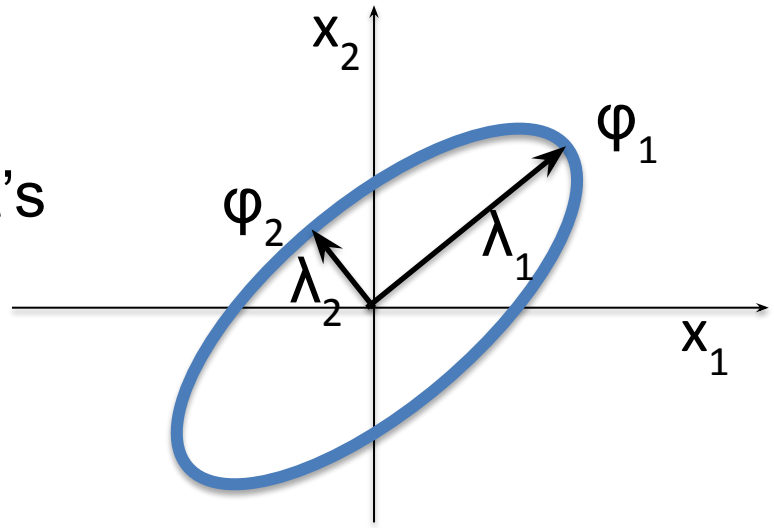
- Basic idea:
  - If the images (or their features) live in a subspace, it is going to look very flat when viewed from the full feature space, e.g.



Slide inspired by N. Vasconcelos

Alternative PCA Formulation

- Assume images x is Gaussian with covariance Σ.



- Recall that a gaussian is defined with it's mean and variance:

$$\mathbf{X} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$$

- Recall that μ and Σ of a gaussian are defined as:

$$\boldsymbol{\mu} = \mathrm{E}[\mathbf{X}]$$

$$\boldsymbol{\Sigma} =: \mathrm{E}[(\mathbf{X} - \boldsymbol{\mu})(\mathbf{X} - \boldsymbol{\mu})^{\mathrm{T}}] = [\mathrm{Cov}[X_i, X_j]; 1 \leq i, j \leq k]$$

Alternative PCA formulation

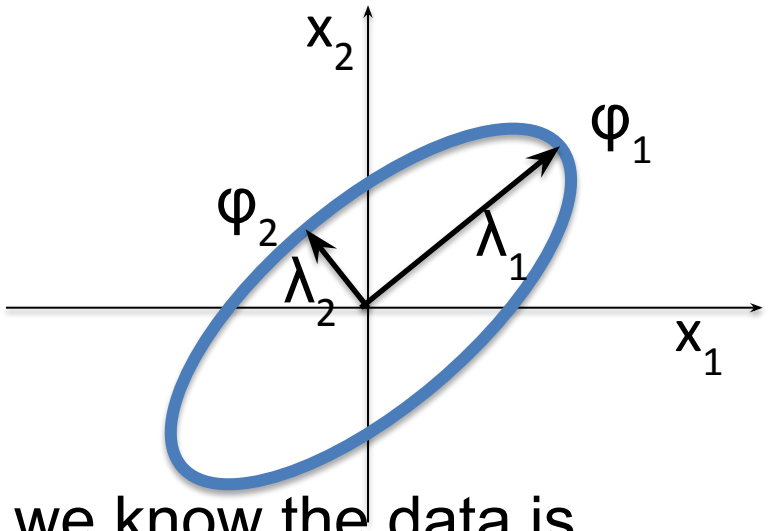
- Since gaussians are symmetric, it's covariance matrix is also a symmetric matrix. So we can express it as:
  - $\mathbf{\Sigma} = \mathbf{U\Lambda U}^T = \mathbf{U\Lambda}^{1/2}(\mathbf{U\Lambda}^{1/2})^T$

$$\mathbf{X} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \iff \mathbf{X} \sim \boldsymbol{\mu} + \mathbf{U}\boldsymbol{\Lambda}^{1/2}\mathcal{N}(0, \mathbf{I})$$

$$\iff \mathbf{X} \sim \boldsymbol{\mu} + \mathbf{U}\mathcal{N}(0, \boldsymbol{\Lambda}).$$

Alternative PCA Formulation

- If x is Gaussian with covariance Σ,
  - Principal components φ_i are the eigenvectors of Σ
  - Principal lengths λ_i are the eigenvalues of Σ



- by computing the eigenvalues we know the data is
  - Not flat if $\lambda_1 \approx \lambda_2$
  - Flat if $\lambda_1 >> \lambda_2$

Alternative PCA Algorithm (training)

- Given sample $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, $x_i \in \mathcal{R}^d$

  - compute sample mean: $\hat{\mu} = \frac{1}{n}\sum_i(\mathbf{x}_i)$
  - compute sample covariance: $\hat{\Sigma} = \frac{1}{n}\sum_i(\mathbf{x}_i - \hat{\mu})(\mathbf{x}_i - \hat{\mu})^T$
  - compute eigenvalues and eigenvectors of $\hat{\Sigma}$

    $$\hat{\Sigma} = \Phi\Lambda\Phi^T, \ \Lambda = diag(\sigma_1^2, \ldots, \sigma_n^2) \ \Phi^T\Phi = I$$

  - order eigenvalues $\sigma_1^2 > \ldots > \sigma_n^2$
  - if, for a certain k, $\sigma_k << \sigma_1$ eliminate the eigenvalues and eigenvectors above k.

Alternative PCA Algorithm (testing)

- Given principal compoenents $\phi_i, i \in 1, \ldots, k$ and a test sample $\mathcal{T} = \{\mathbf{t}_1, \ldots, \mathbf{t}_n\}, \ t_i \in \mathcal{R}^d$

  - subtract mean to each point $\mathbf{t}'_i = \mathbf{t}_i - \widehat{\mu}$

  - project onto eigenvector space $\mathbf{y}_i = \mathbf{A}\mathbf{t}'_i$ where

$$\mathbf{A} = \begin{bmatrix} \phi_1^T \\ \vdots \\ \phi_k^T \end{bmatrix}$$

  - use $\mathcal{T}' = \{\mathbf{y}_1, \ldots \mathbf{y}_n\}$ to estimate class conditional densities and do all further processing on $\mathbf{y}$.