# Last time:
# Feature Engineering 1

- «Good» / Interest / Key **points** to find:
  - Flat, Edge and Corner
- **Harris** Corner Detection:
  - Taylor, Second moment matrix & Response function
- **SIFT**:
  - Scale invariant, dominant direction, descriptor
- **RANSAC**
  - Sample, model, inliers / outliers

# Last time:
# Feature Engineering 2

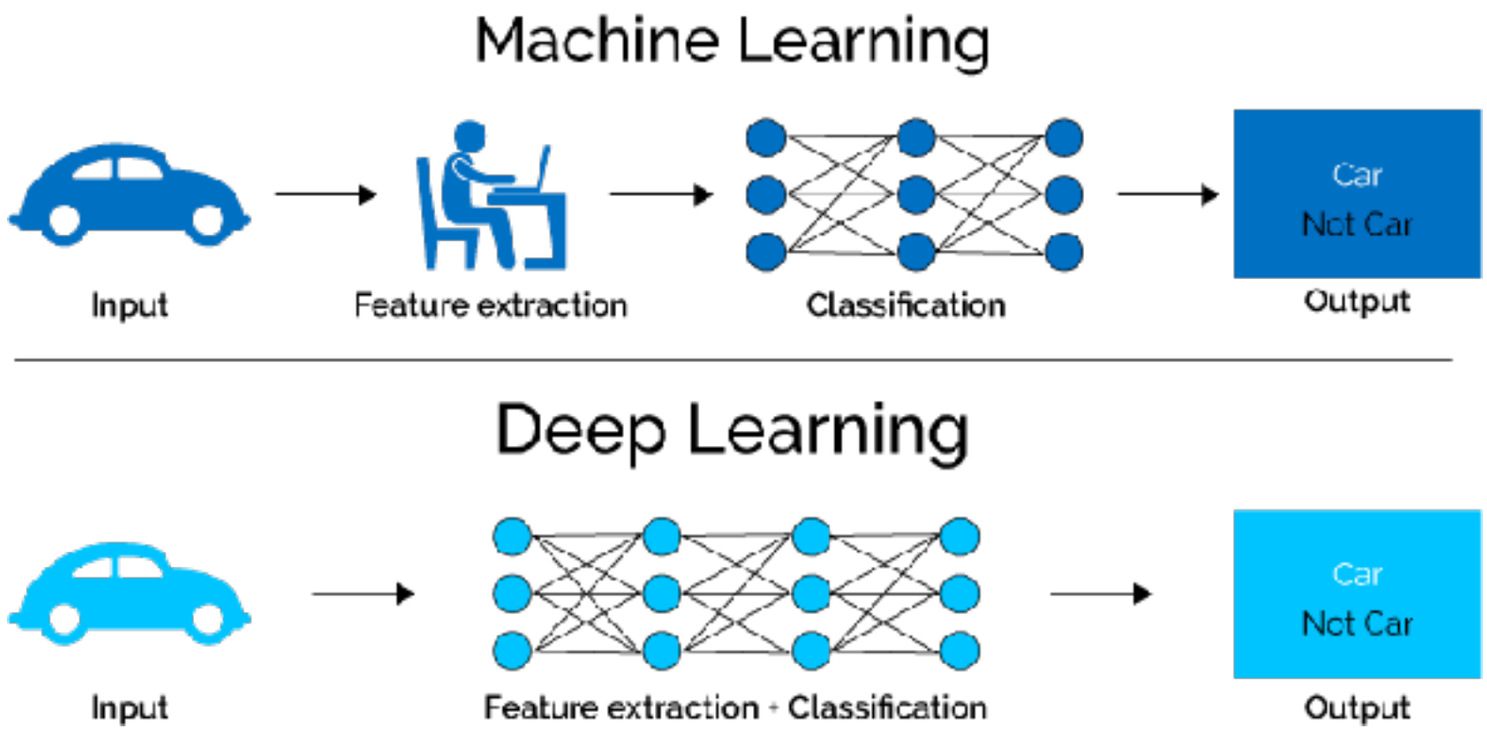
- Viola Jones Orientation, not very relevant for the exam
- HOG
- DPM Orientation, not very relevant for the exam

Orientation, not very relevant for the exam

This time..

# Traditional CV (ML-based): **Classification**

Object recognition



# Agenda

- Intro
- Supervised:
    - kNN - k-Nearest Neighbor
    - SVM - Support Vector Machines
- Unsupervised:
    - K-Means
    - Mean-Shift
    - Principle Component Analysis (**PCA**) & Dimensionality reduction

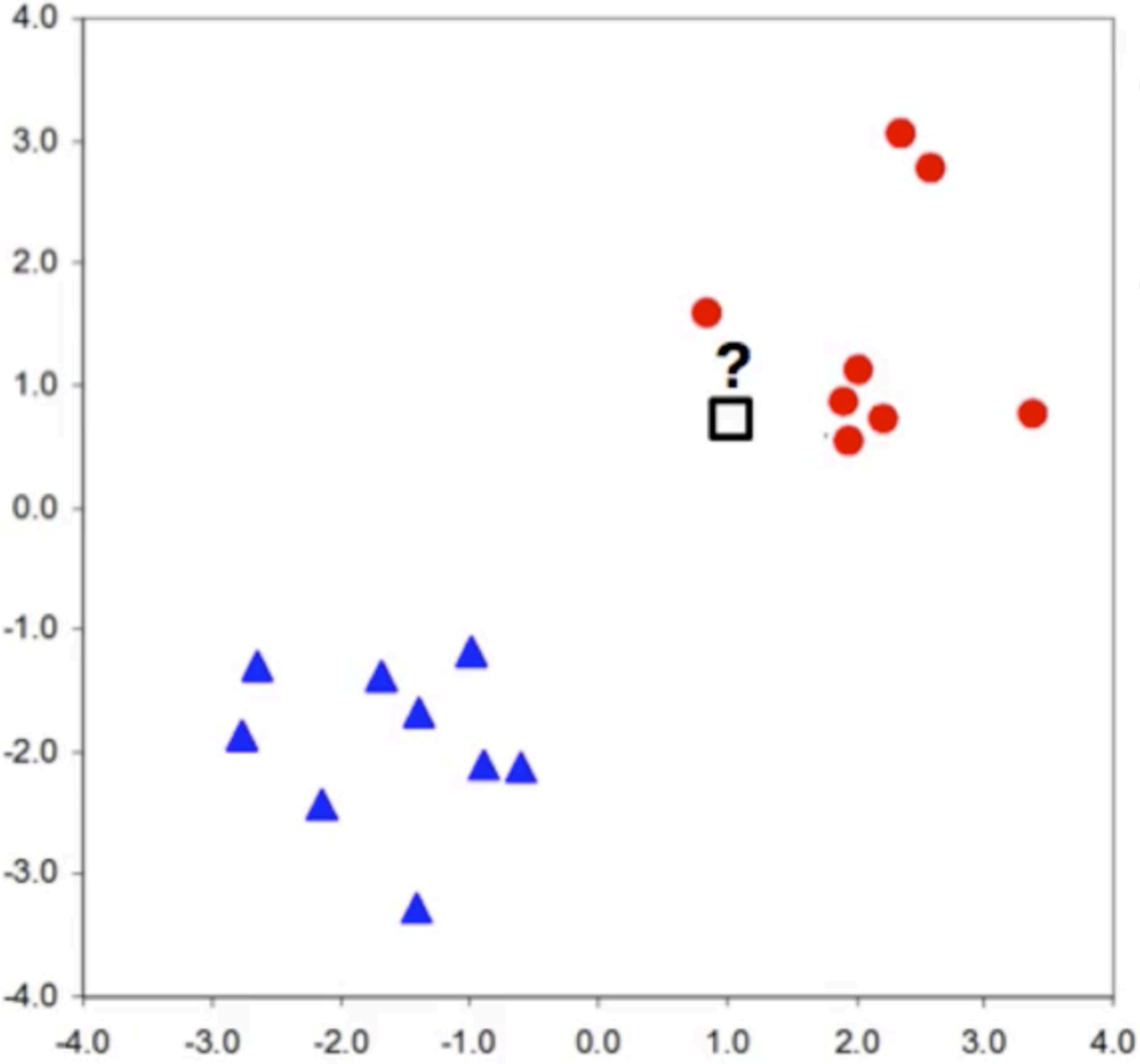
# Supervised

kNN
SVM

# kNN

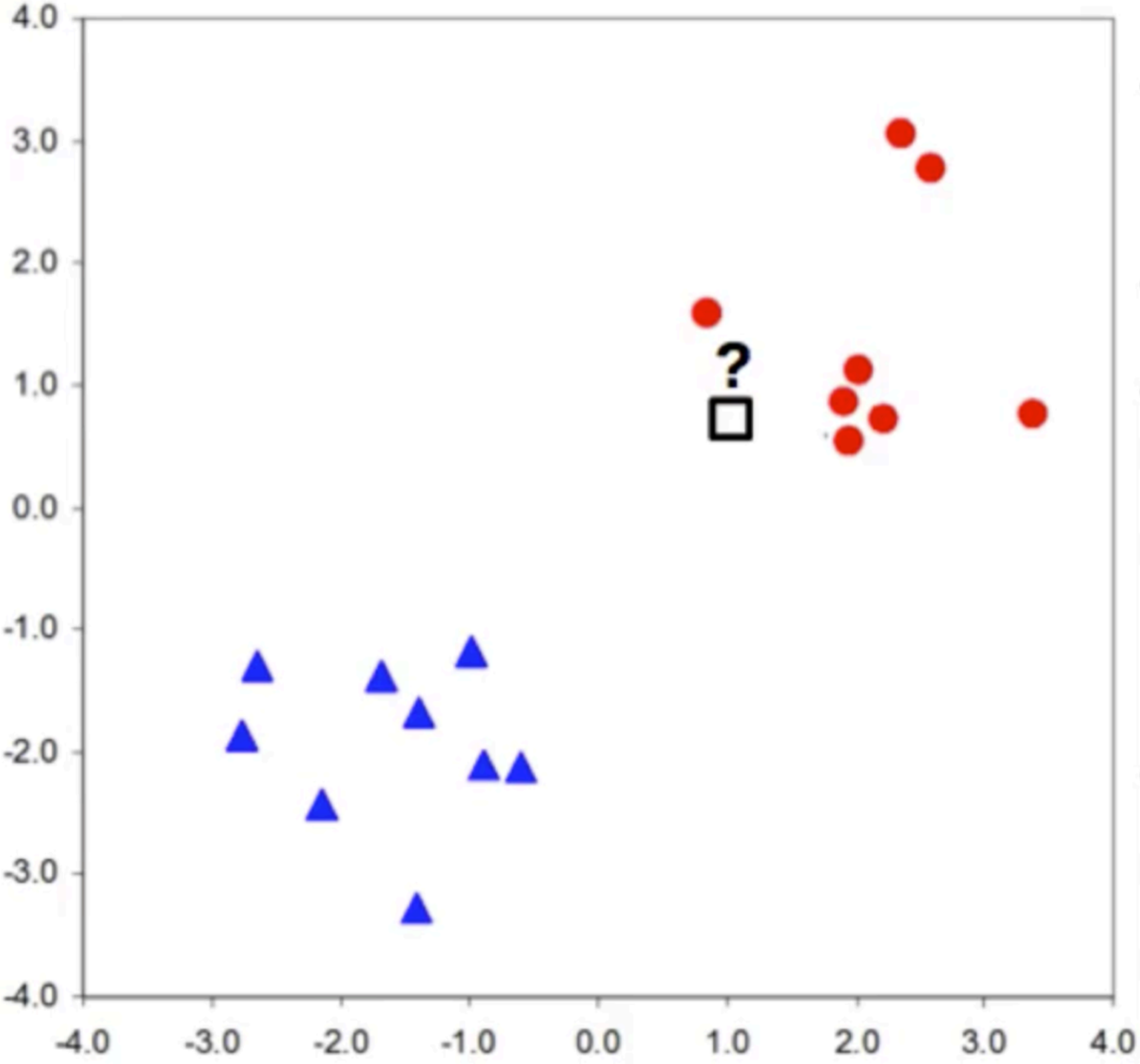
k-Nearest Neighbor (Supervised)

# Intuition for kNN




- set of points (x,y)
  - two classes
- is the box red or blue

# Intuition for kNN



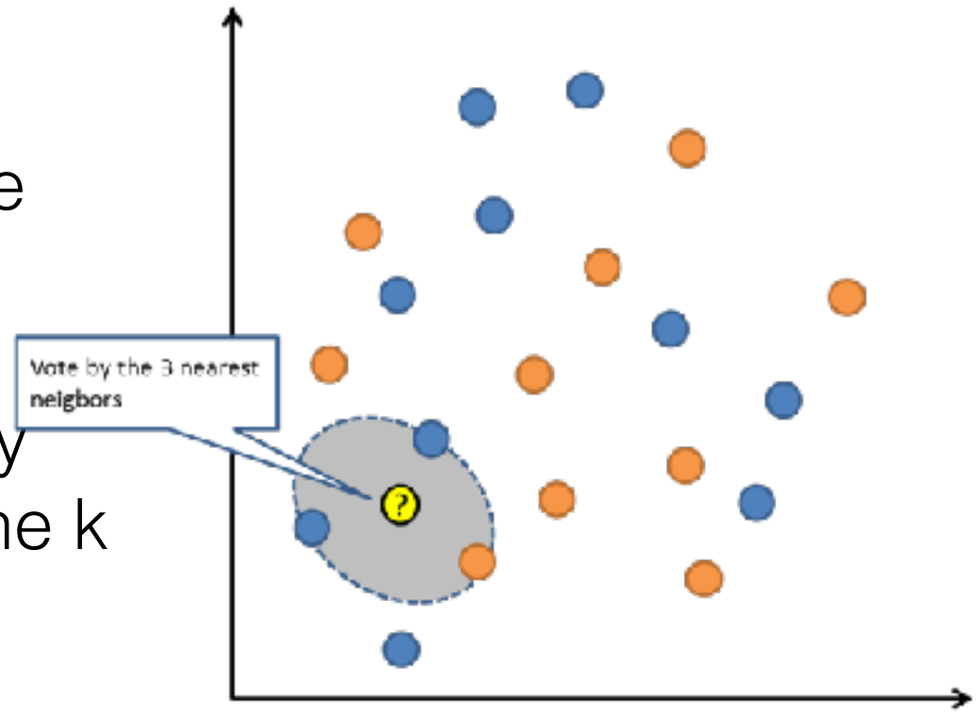

- set of points (x,y)
  - two classes
- is the box red or blue
- how did you do it
  - use Bayes rule?
  - a decision tree?
  - fit a hyperplane?
- nearby points are red
  - use this as a basis for a learning algorithm

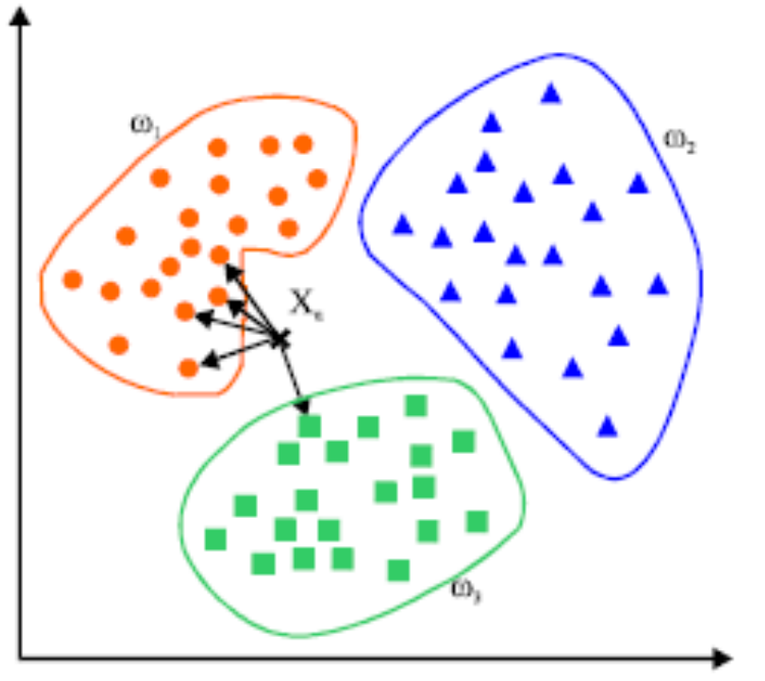
# kNN

- The model representation for kNN is the entire training dataset (i.e. no calculations)

- **Predictions** are made for a new data point by searching through the entire training set for the k most similar instances (the neighbors) and summarizing the output variable for those k instances. For **classification** problems this might be the **mode** (or most common) class value, For **regression** problems this might be the **mean** output variable.

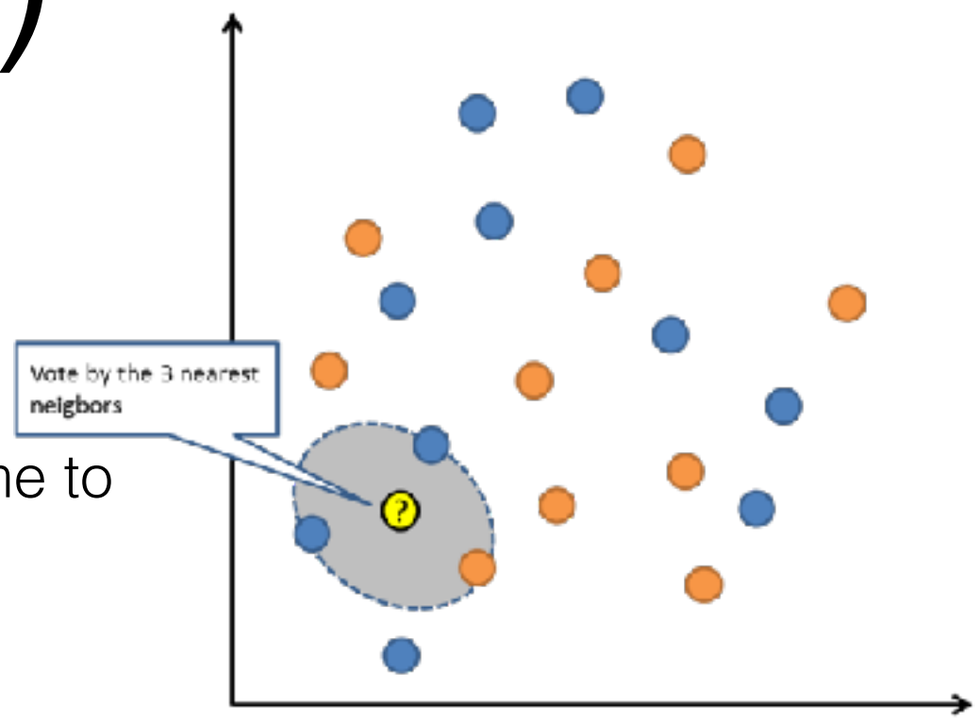
- How to determine **similarity** between the data instances. Use **Euclidean** distance if all attributes are all of the same «scale», e.g. meters.



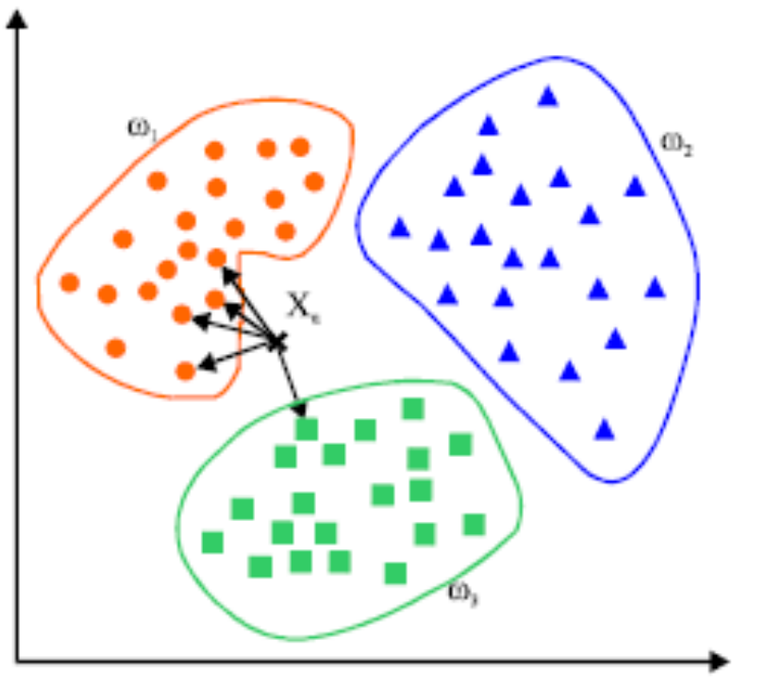



# kNN (2)

- Pros:
  - Very **simple** and very **effective**
  - **Update** and curate your training instances over time to keep predictions accurate.

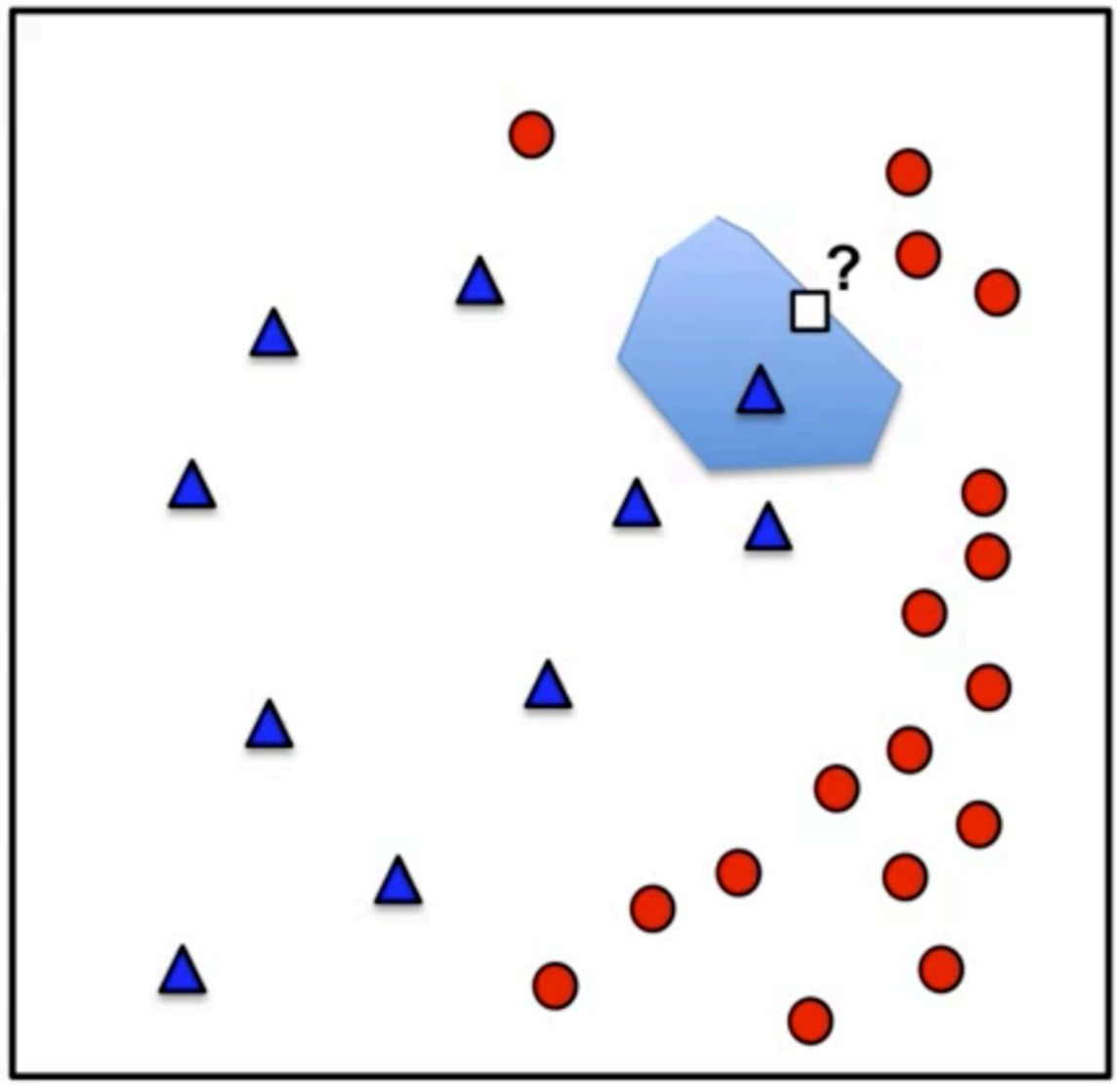



- Cons:
  - Require a lot of **memory** or space to store all of the data
  - All calculations are performs at prediction time, **inference** is potentially slow.
  - The idea of distance or **closeness** can break down in **very high dimensions** which can negatively affect the performance of the algorithm on your problem (only use those input variables that are most relevant to predicting the output variable)



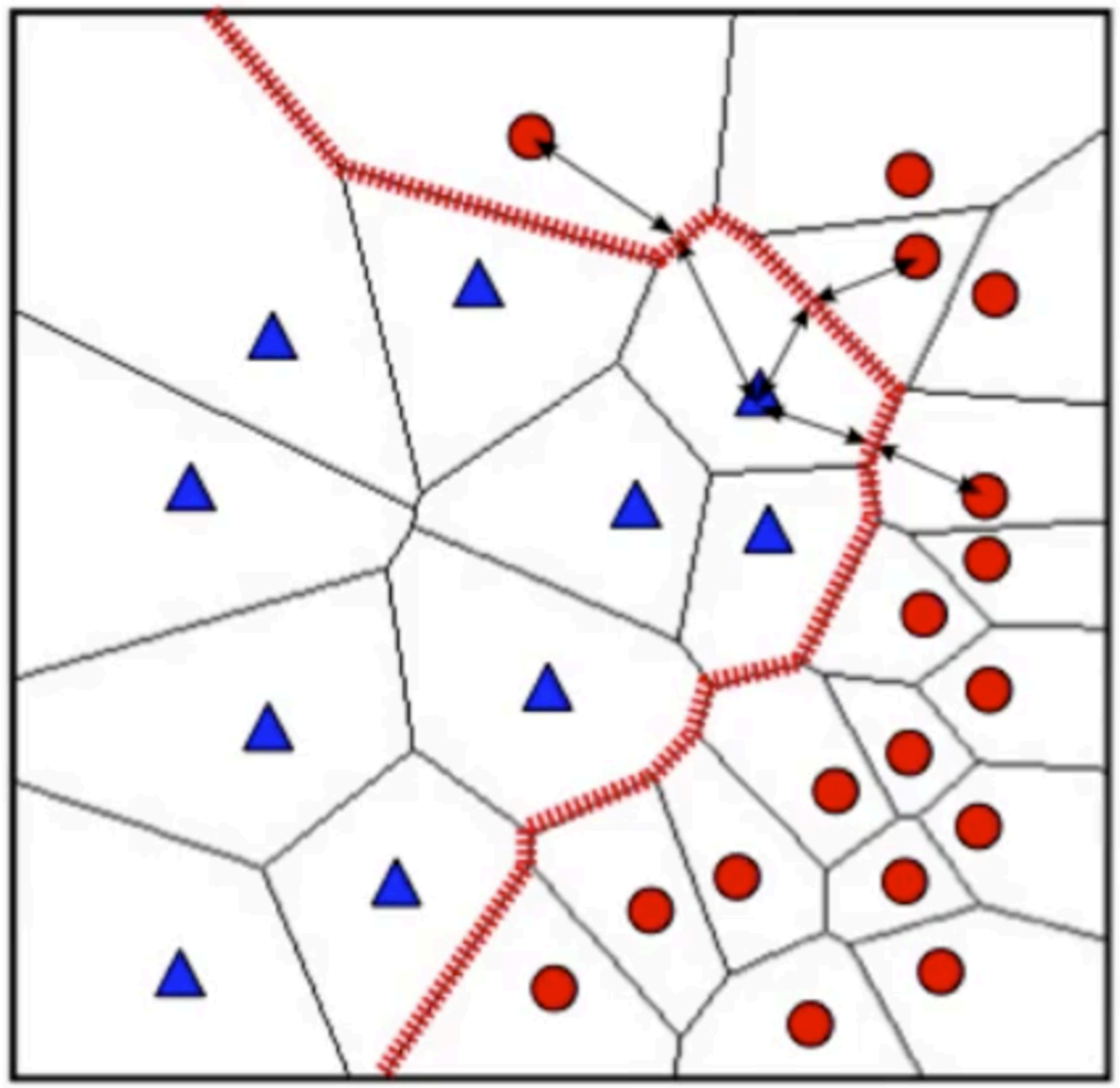
# Nearest-neighbor classification

- Use the intuition to classify a new point x:
  - find the most similar training example x'
  - predict its class y'



# Nearest-neighbor classification

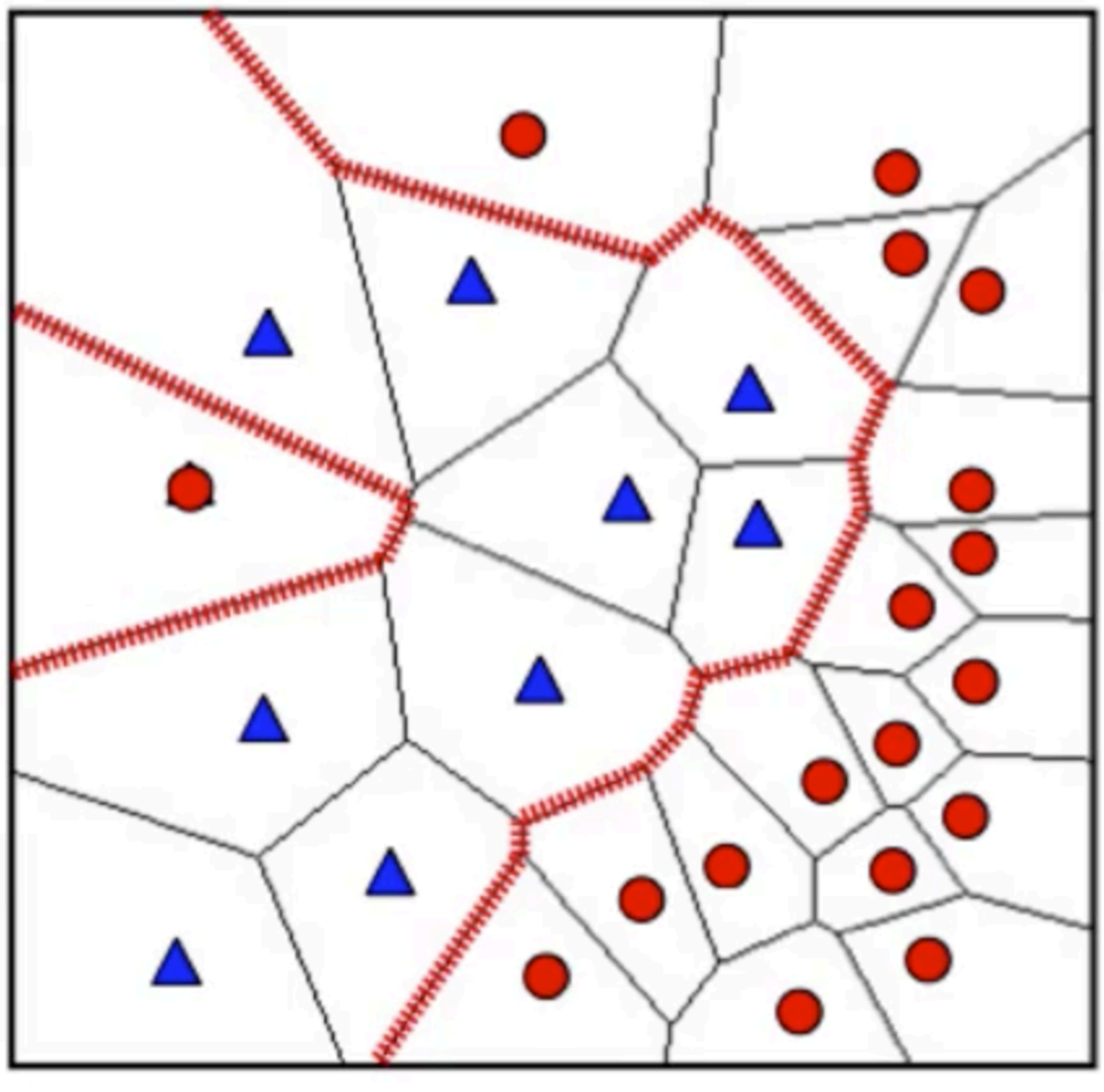
- Use the intuition to classify a new point x:
  - find the most similar training example x'
  - predict its class y'
- Voronoi tesselation
  - partitions space into regions
  - boundary: points at same distance from two different training examples
- decision boundary
  - non-linear, reflects classes well
  - compare to NB, DT, logistic
  - impressive for simple method



# Nearest neighbour: outliers

- Algorithm is sensitive to outliers
  - single mislabeled example dramatically changes boundary

- Idea:
  - use more than one nearest neighbor to make decision
  - count class labels in k most similar training examples
    - many “triangles” will outweigh single “circle” outlier



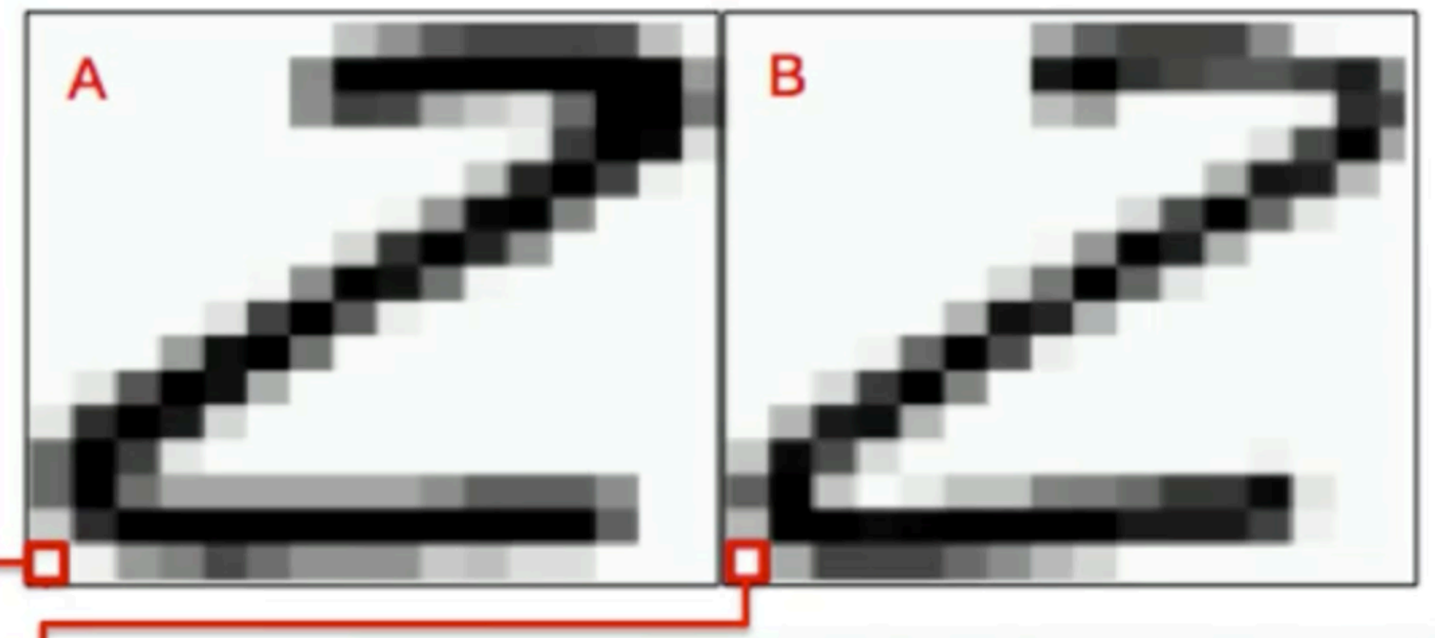
# kNN classification algorithm

- Given:
  - training examples $\{x_i, y_i\}$
    - $x_i$ … attribute-value representation of examples
    - $y_i$ … class label: {ham,spam}, digit {0,1,…9} etc.
  - testing point $x$ that we want to classify
- Algorithm:
  - compute distance $D(x,x_i)$ to every training example $x_i$
  - select $k$ closest instances $x_{i1}…x_{ik}$ and their labels $y_{i1}…y_{ik}$
  - output the class $y^*$ which is most frequent in $y_{i1}…y_{ik}$

# Example: handwritten digits

- 16x16 bitmaps
- 8-bit grayscale
- Euclidian distance
  - over raw pixels

$$D(A,B)=\sqrt{\sum_r \sum_c \left(A_{r,c}-B_{r,c}\right)^2}$$



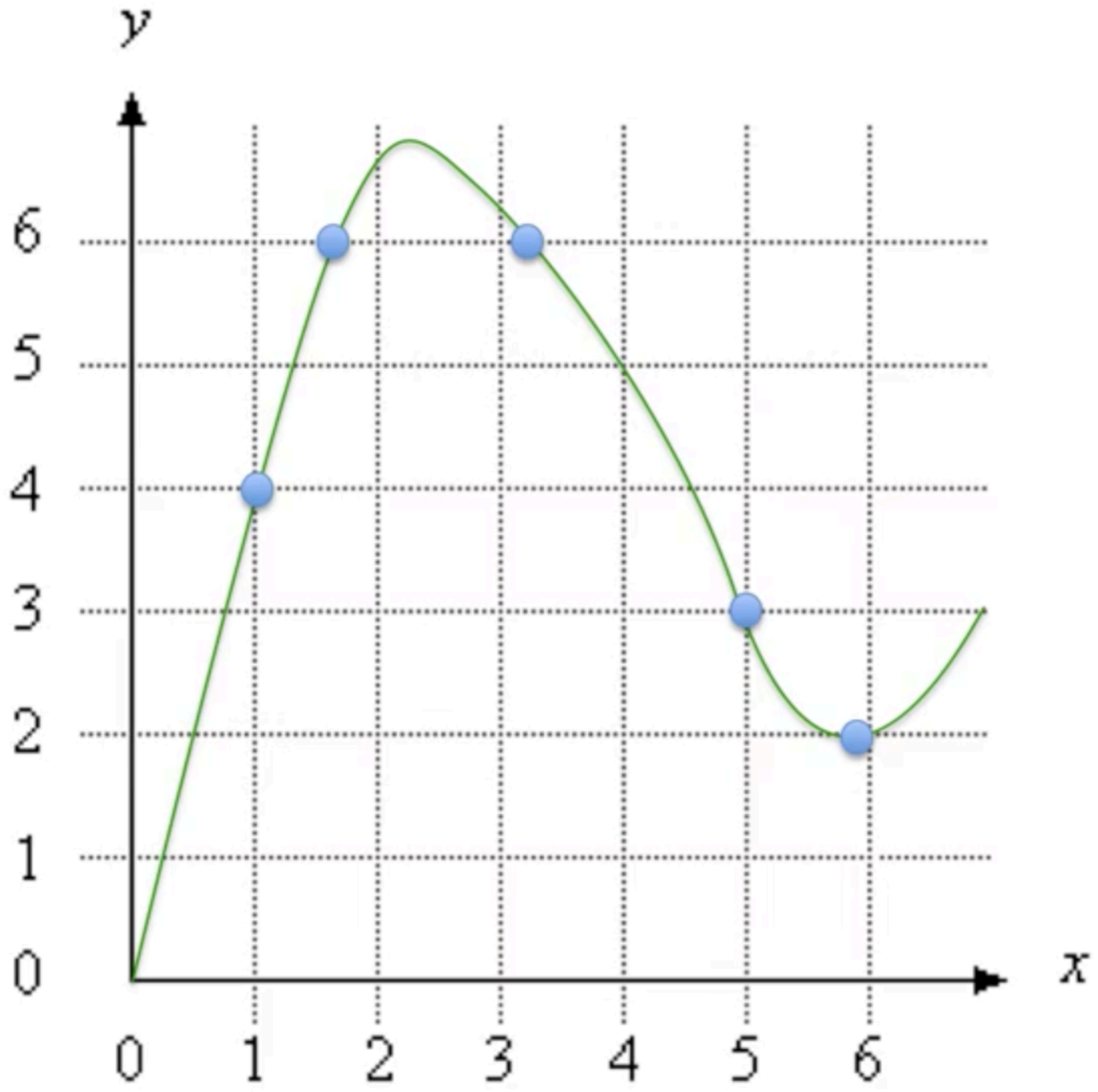

- Accuracy:
  - 7-NN ~ 95.2%
  - SVM ~ 95.8%
  - humans ~ 97.5%

# kNN regression algorithm

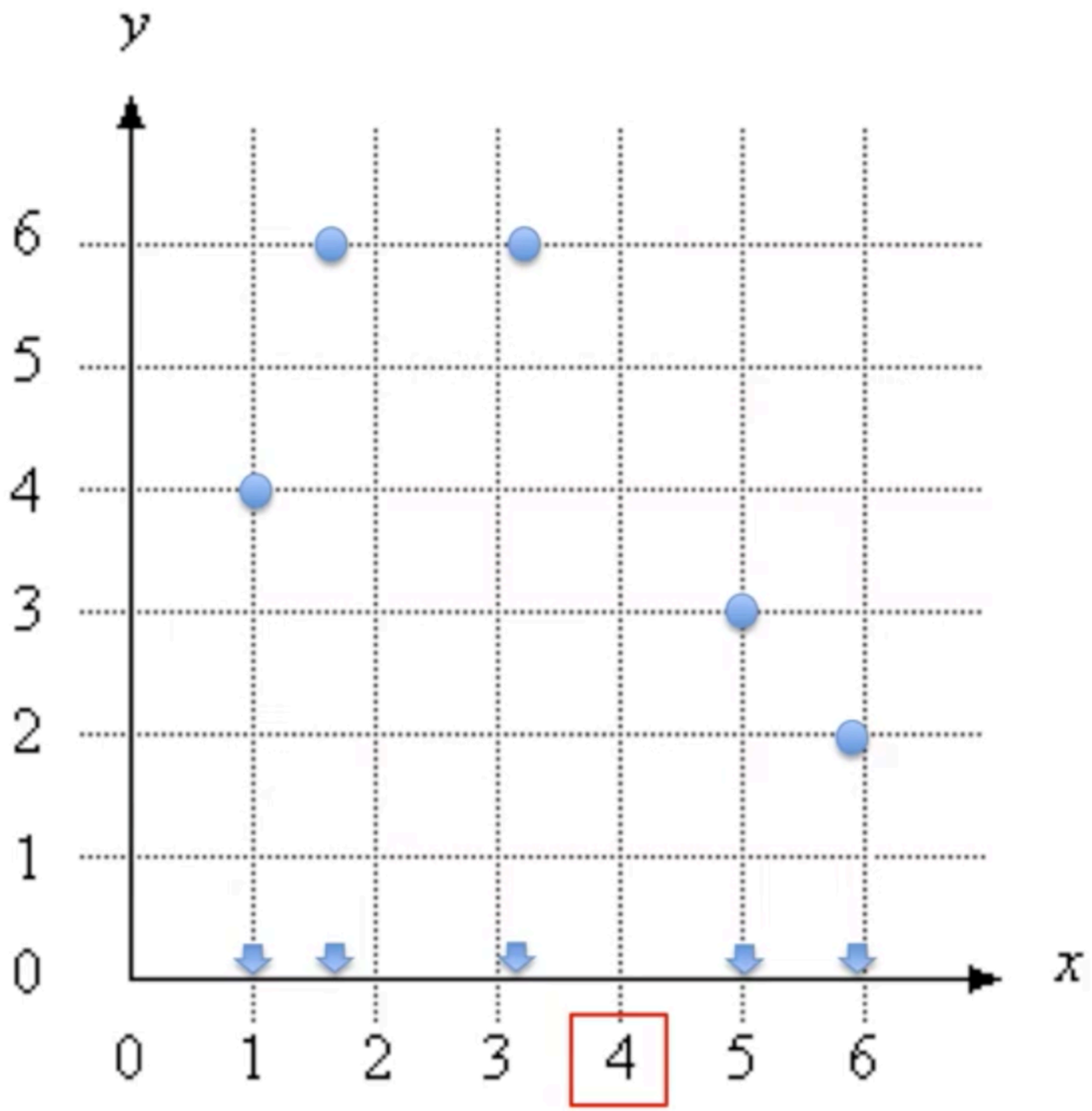
- Given:
  - training examples $\{x_i, y_i\}$
    - $x_i$ … attribute-value representation of examples
    - $y_i$ … real-valued target (profit, rating on YouTube, etc)
  - testing point x that we want to predict the target
- Algorithm:
  - compute distance $D(x,x_i)$ to every training example $x_i$
  - select $k$ closest instances $x_{i1}…x_{ik}$ and their labels $y_{i1}…y_{ik}$
  - output the mean of $y_{i1}…y_{ik}$ :

$$\hat{y} = f(x) = \frac{1}{k}\sum_{j=1}^{k} y_{i_j}$$

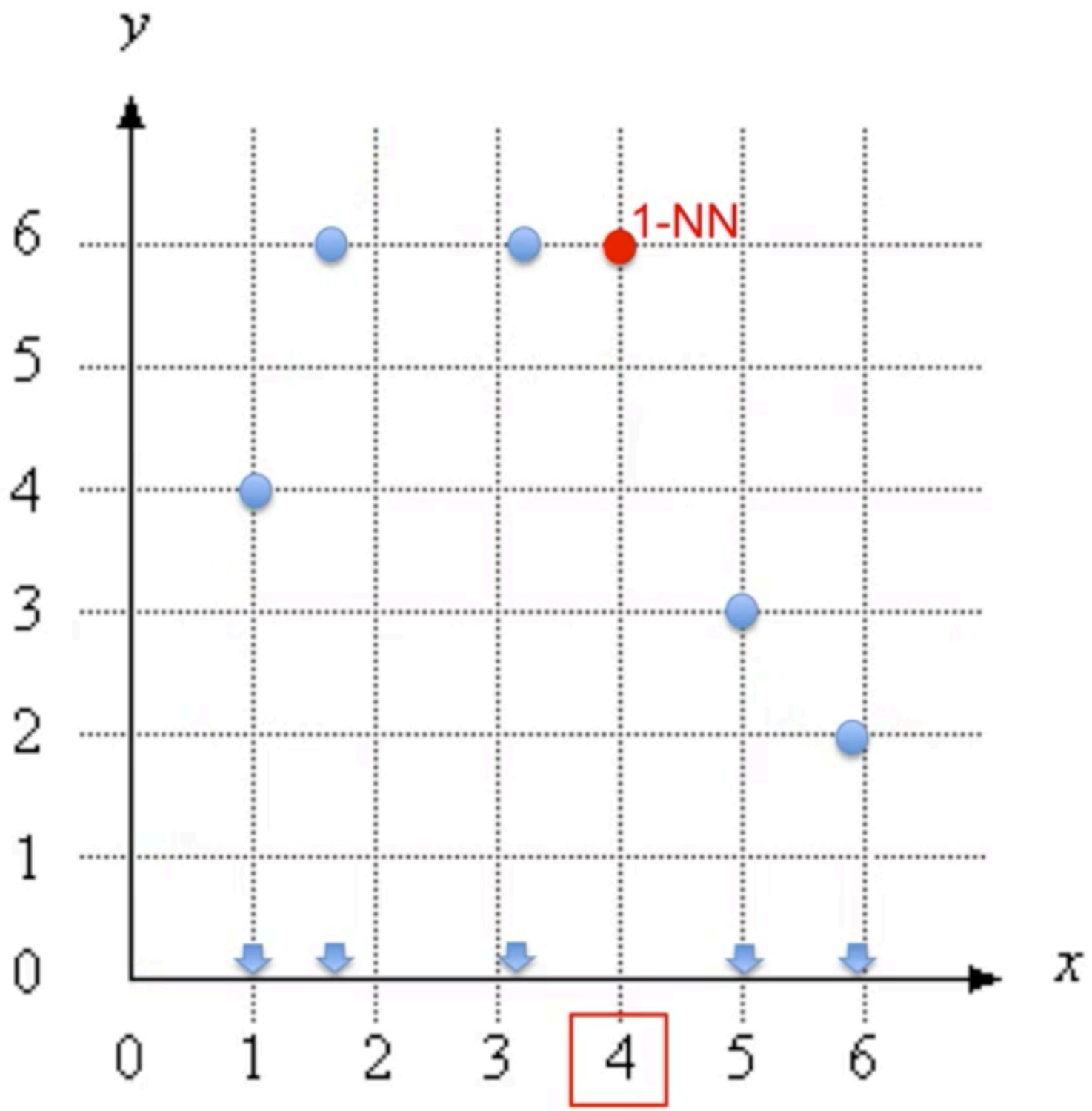
# Example: kNN regression in 1-d



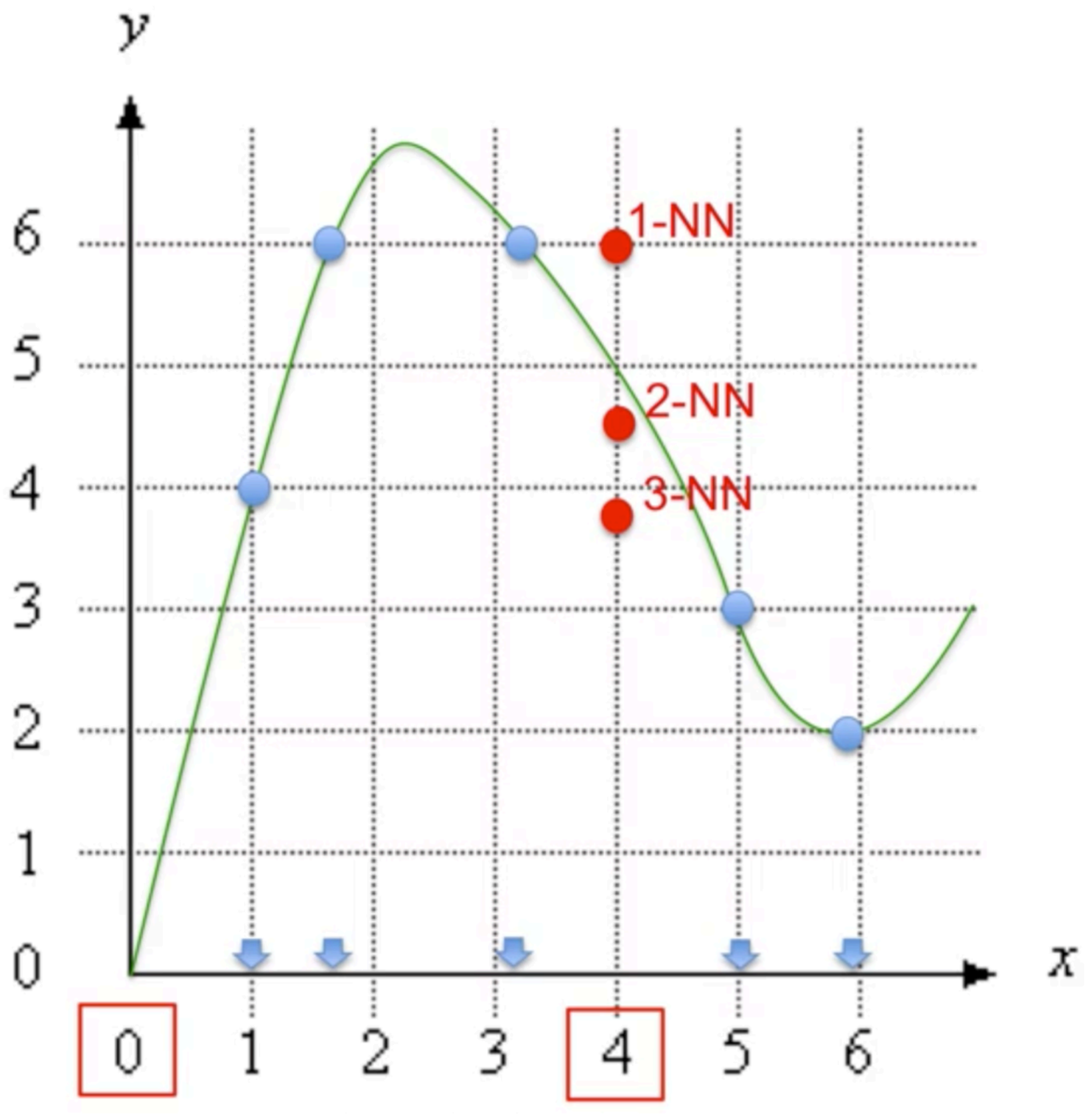
# Example: kNN regression in 1-d



# Example: kNN regression in 1-d

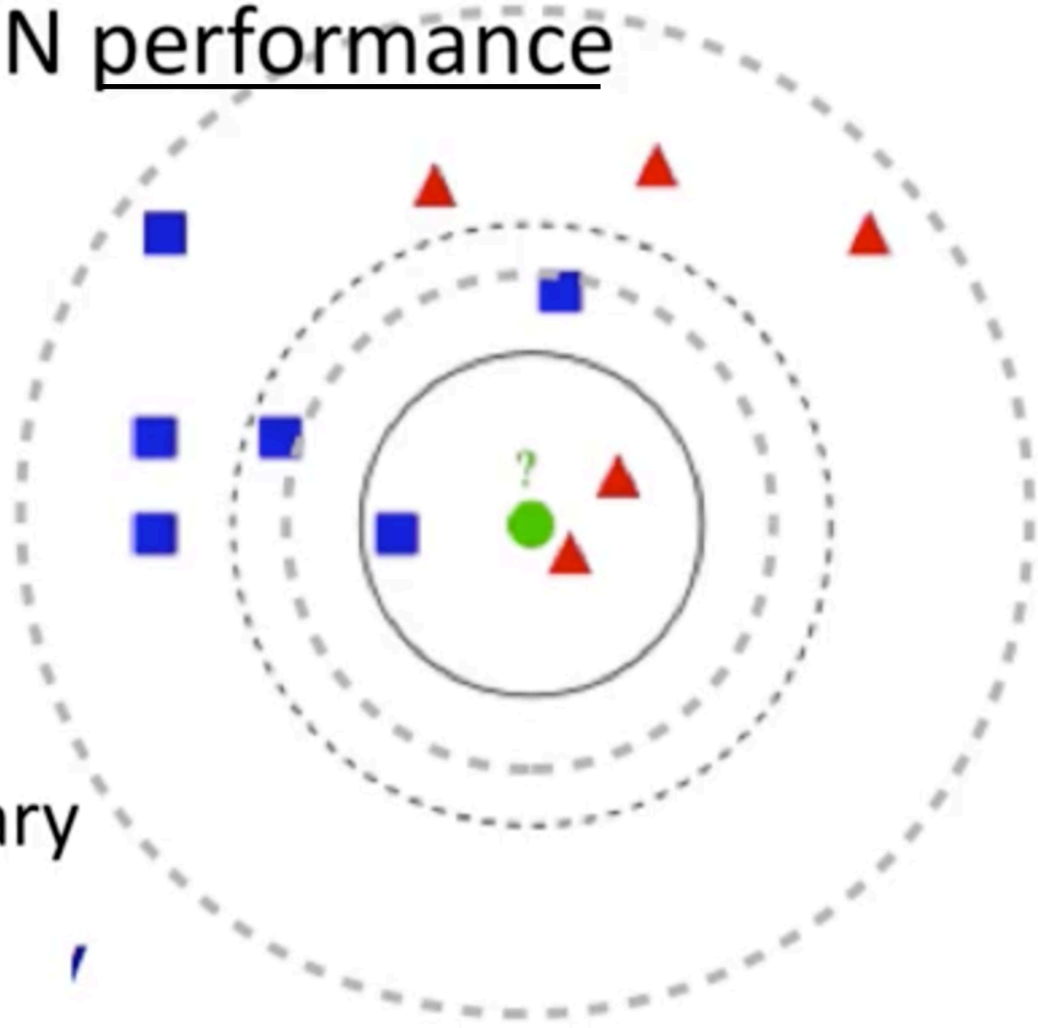


# Example: kNN regression in 1-d



# Choosing the value of k

- Value of k has strong effect on kNN performance
  - large value → everything classified as the most probable class: P(y)
  - small value → highly variable, unstable decision boundaries
    - small changes to training set → large changes in classification
  - affects "smoothness" of the boundary



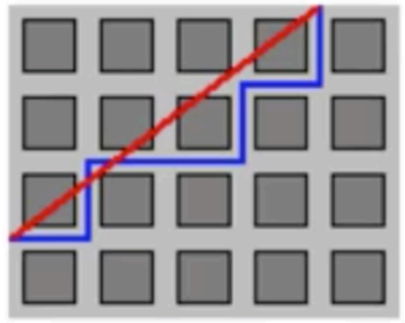

- Selecting the value of k
  - set aside a portion of the training data (validation set)
  - vary k, observe training → validation error

# Distance measures

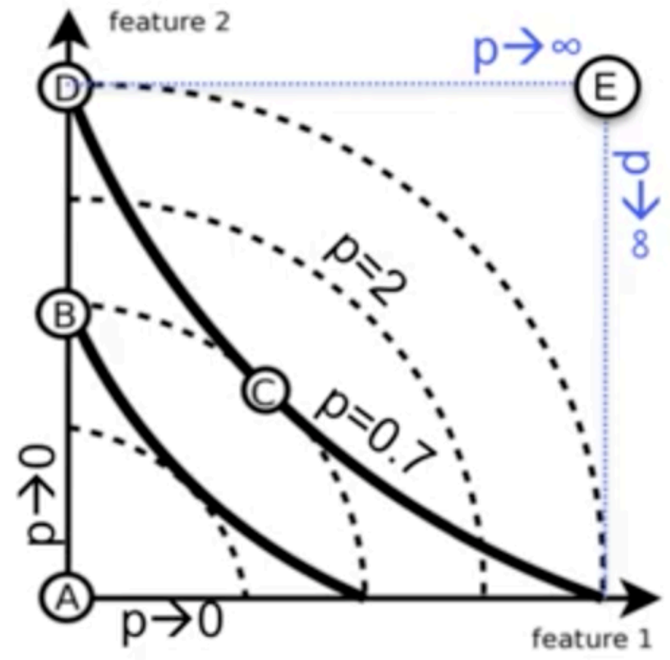
- Key component of the kNN algorithm
  - defines which examples are similar & which aren't
  - can have strong effect on performance
- Euclidian (numeric attributes): $D(x,x') = \sqrt{\sum_d |x_d - x'_d|^2}$
  - symmetric, spherical, treats all dimensions equally
  - sensitive to extreme differences in single attribute
    - behaves like a "soft" logical OR
- Hamming (categorical attributes): $D(x,x') = \sum_d 1_{x_d \neq x'_d}$
  - number of attributes where *x*, *x'* differ

# Distance measures (2)

- Minkowski distance (*p*-norm): $D(x,x') = \sqrt[p]{\sum_d |x_d - x'_d|^P}$
  - $p = 2$: Euclidian
  - $p = 1$: Manhattan



  - $p \to \infty$: $\mathbf{max}_d\ |x_d - x'_d|$ … logical OR
  - $p \to 0$: … logical AND



# kNN: practical issues

- Resolving ties:
  - equal number of positive/negative neighbours
  - use odd k (doesn't solve multi-class)
  - breaking ties:
    - random: flip a coin to decide positive / negative
    - prior: pick class with greater prior
    - nearest: use 1-nn classifier to decide
- Missing values
  - have to "fill in", otherwise can't compute distance
  - key concern: should affect distance as little as possible
  - reasonable choice: average value across entire dataset

# kNN pros and cons

- Almost no assumptions about the data
  - smoothness: nearby regions of space → same class
  - assumptions implied by distance function (only locally!)
  - non-parametric approach: "let the data speak for itself"
    - nothing to infer from the data, except *k* and possibly D()
    - easy to update in online setting: just add new item to training set
- Need to handle missing data: fill-in or create a special distance
- Sensitive to class-outliers (mislabeled training instances)
- Sensitive to lots of irrelevant attributes (affect distance)
- Computationally expensive:
  - space: need to store all training examples
  - time: need to compute distance to all examples: O(*nd*)
    - *n* … number of training examples, *d* … cost of computing distance
    - *n* grows → system will become slower and slower
    - expense is at *testing*, not *training* time (bad)

# SVM

Suport Vector Machine

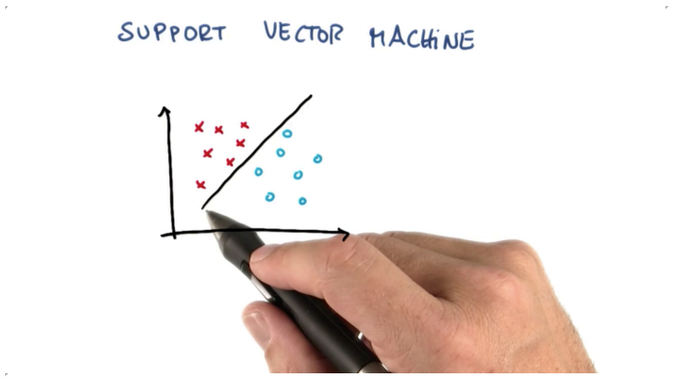
# SVM (for Classification)

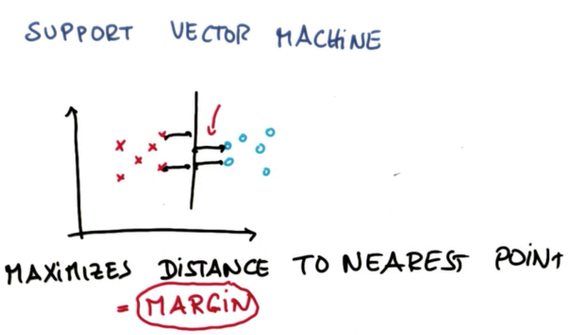
- **Given**: many feature vectors, **Find**: decision boundary (line or hyperplane). **Margin** (robustness).

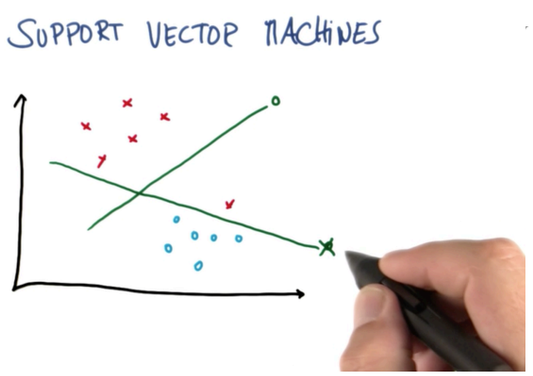
- **Difficult** data (correct first). **Outliers** (robust)

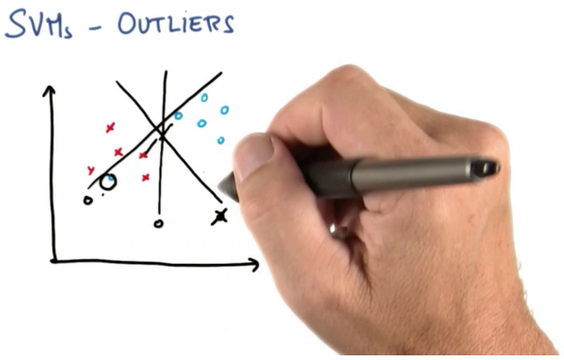
- **Non-Linearly separable**, new variable

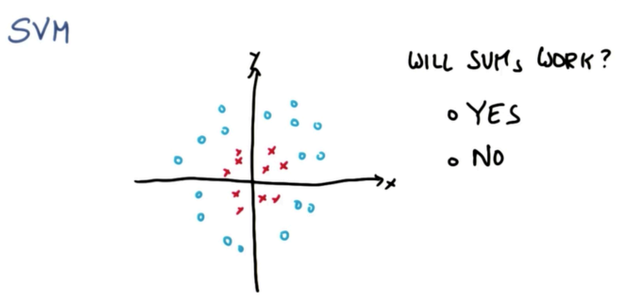
- **Kernel-trick** and non-linear SVMs, parameters and overfitting

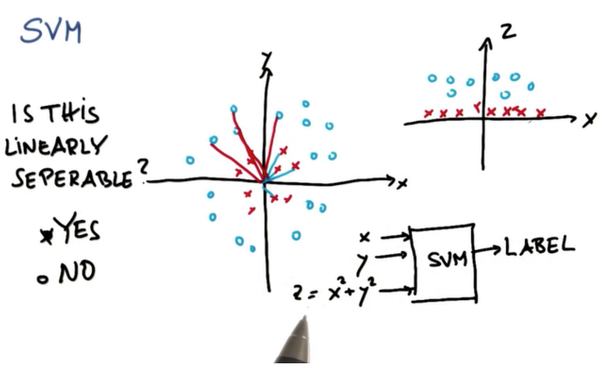


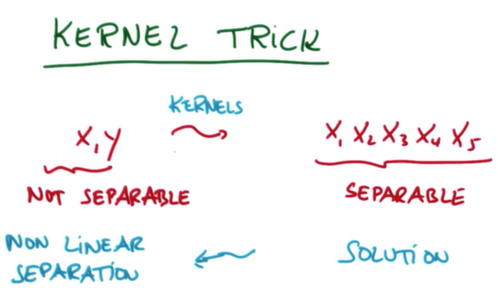



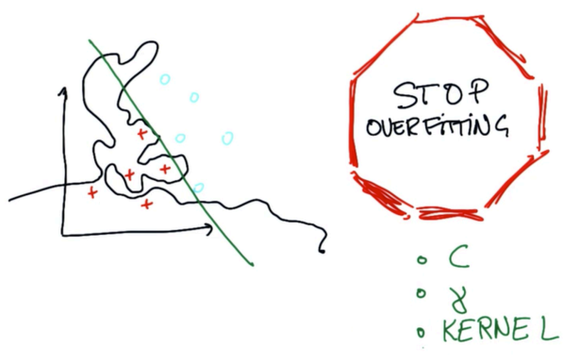



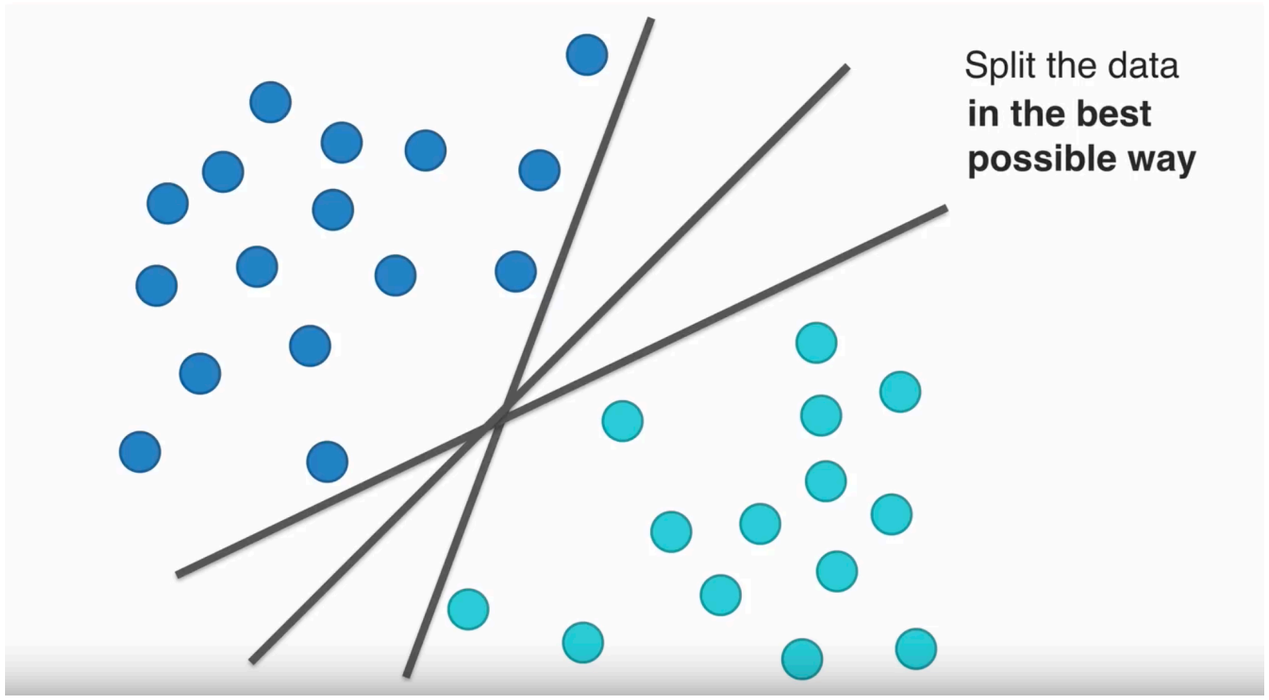



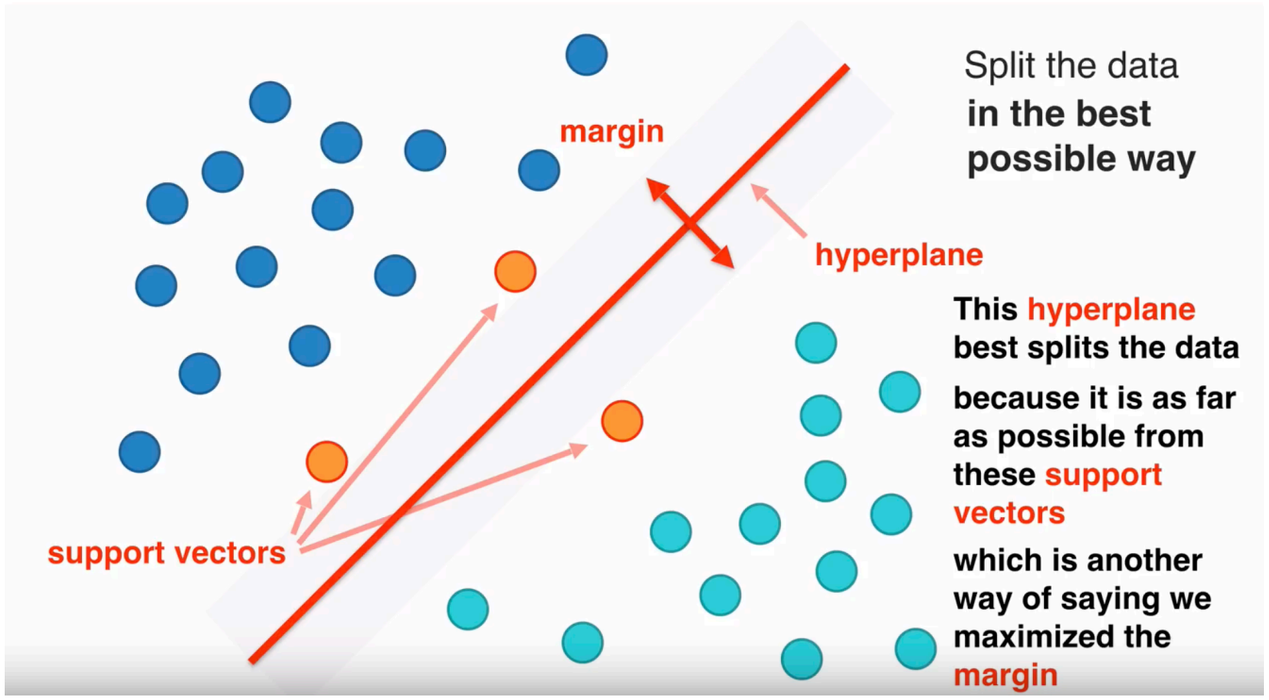



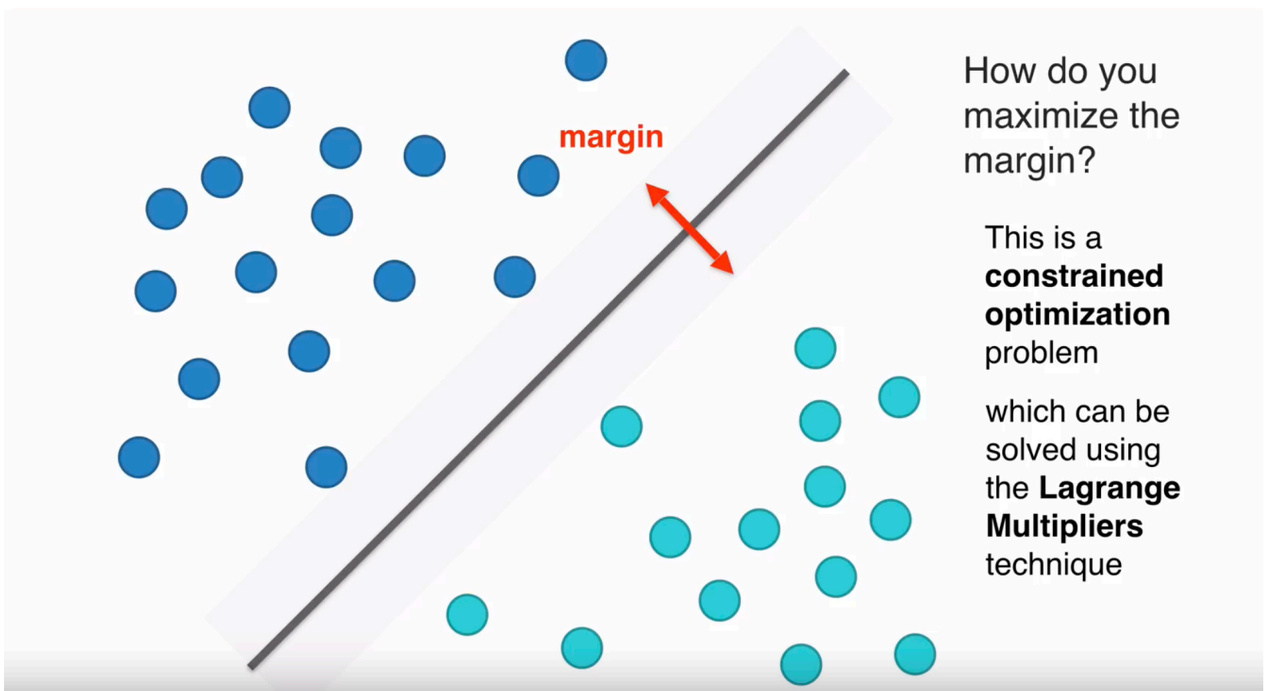



# SVM: **Basics**


Split the data
in the best
possible way

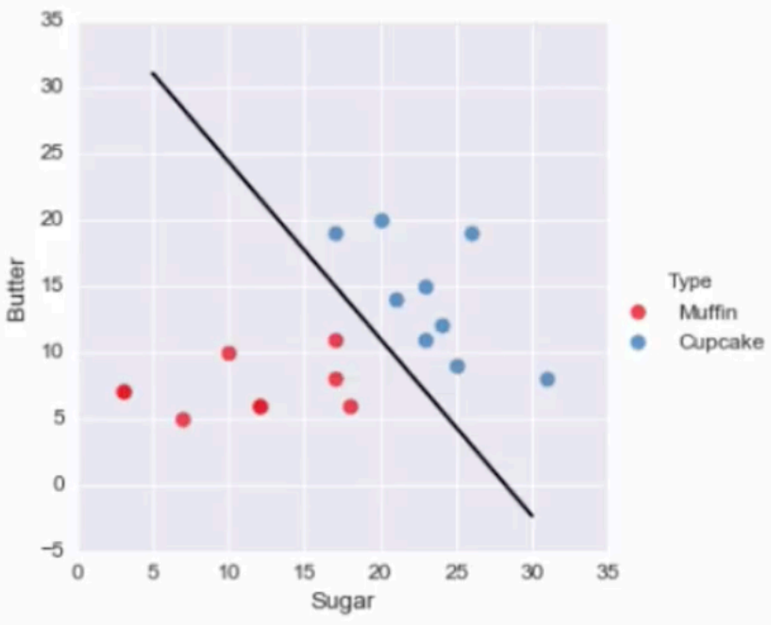

margin
hyperplane
support vectors
Split the data
in the best
possible way
This hyperplane
best splits the data
because it is as far
as possible from
these support
vectors
which is another
way of saying we
maximized the
margin


margin
How do you
maximize the
margin?
This is a
constrained
optimization
problem
which can be
solved using
the Lagrange
Multipliers
technique

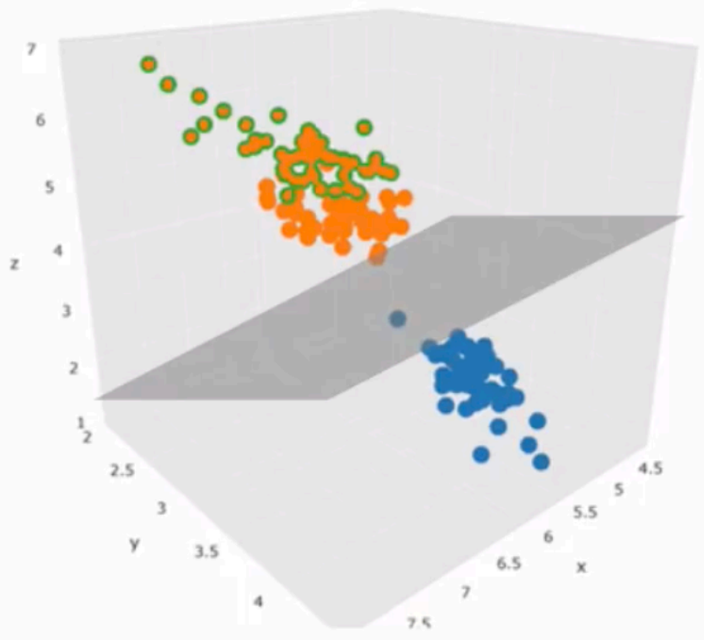
# SVM: ND - **Hyperplane**

## Higher Dimensions: Visual

2D: Separate with Line




3D: Separate with Plane



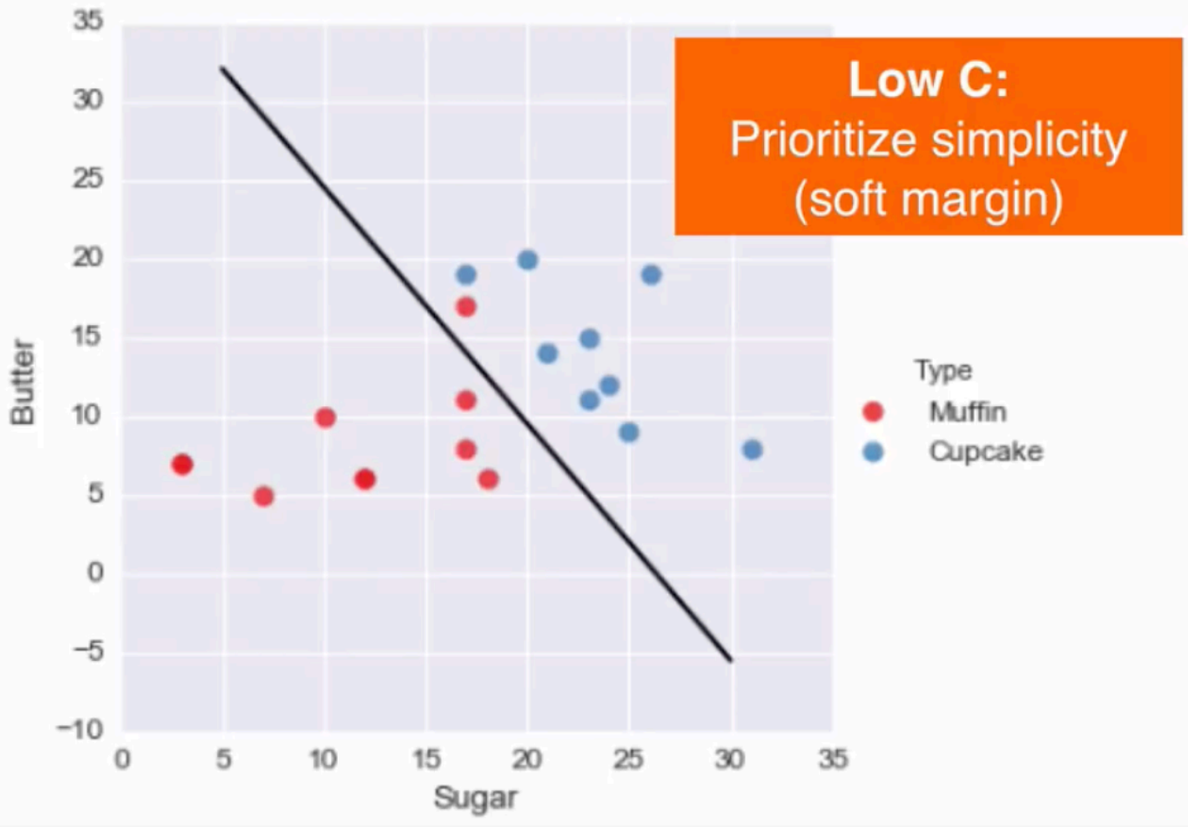

4D+: Separate with Hyperplane

**Hard to Visualize**

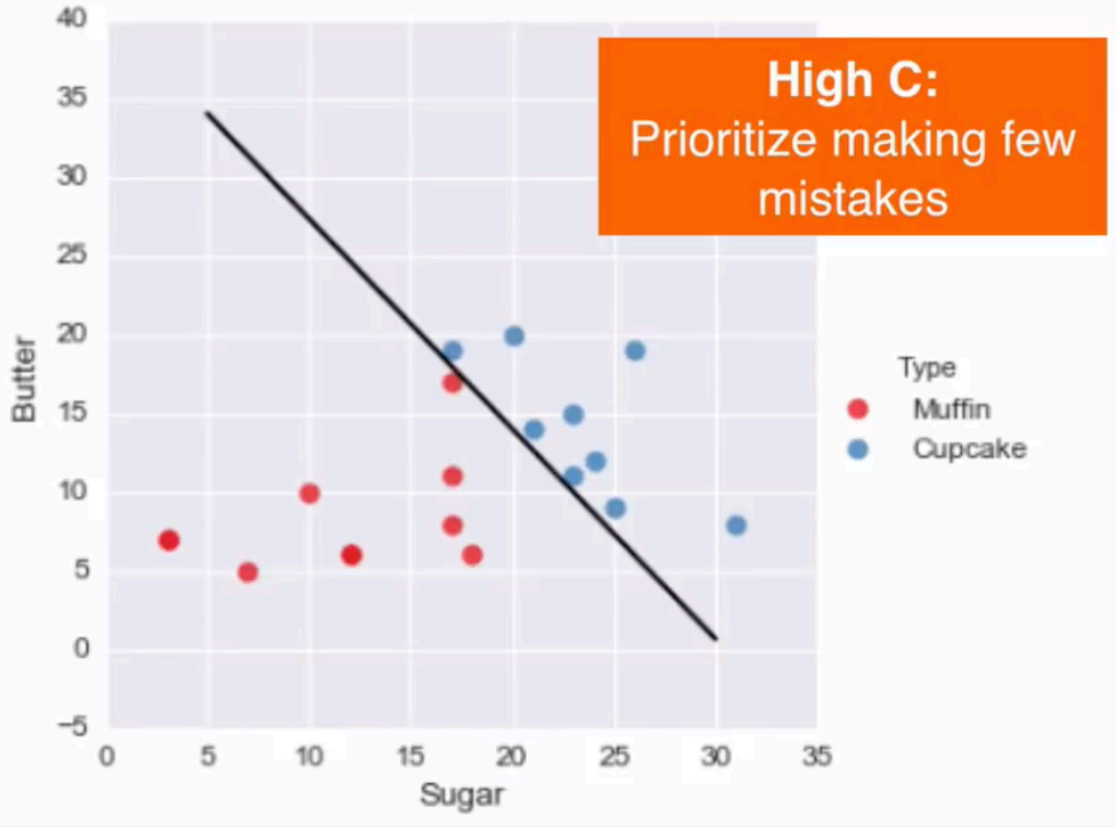
# SVM: **C-parameter**

## C Parameter: Comparison

```
# Fit the SVM model with a LOW C
model = svm.SVC(kernel='linear', C=2**-5)
model.fit(sugar_butter, type_label)
```

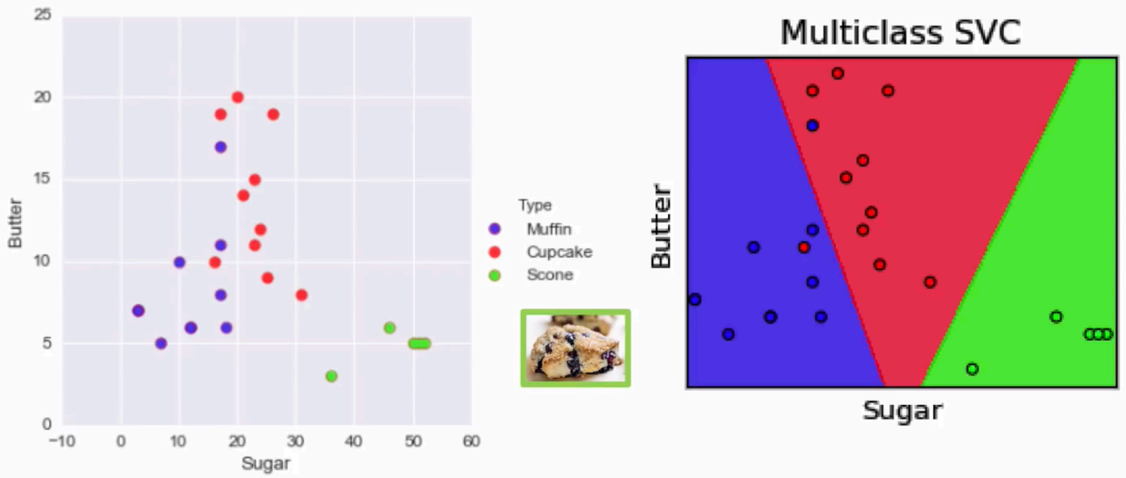



```
# Fit the SVM model with a HIGH C
model = svm.SVC(kernel='linear', C=2**5)
model.fit(sugar_butter, type_label)
```



# SVM: **Multiple Classes**

## Multiple Classes: Visual









## Multiple Classes: Code

Original Code (2 classes)

```
# Fit the SVM model
model = svm.SVC(kernel='linear')
model.fit(sugar_butter, type_label)
```

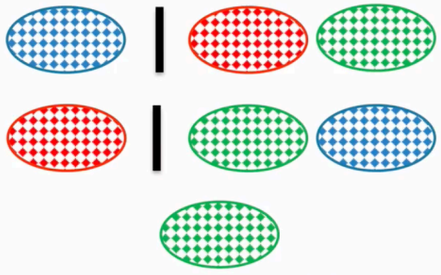
Updated Code (3+ classes)

```
# Fit the SVM model for more than 2 classes
model = svm.SVC(kernel='linear', decision_function_shape='ovr')
model.fit(sugar_butter, type_label)
```



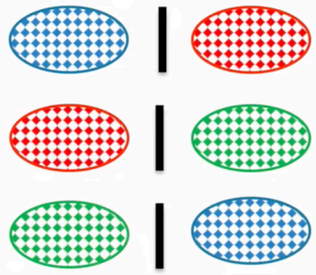
## Multiple Classes: Comparison

**OVR: One vs Rest**



**Pros: Fewer classifications**
**Cons: Classes may be imbalanced**

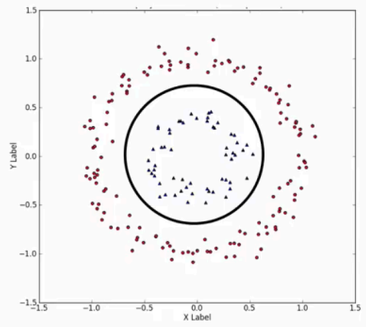
**OVO: One vs One**



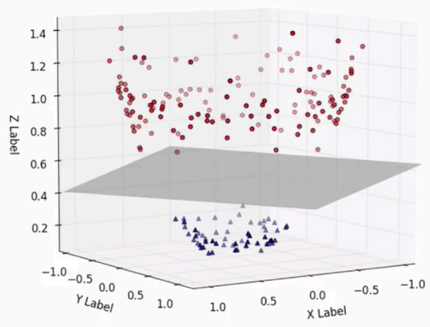
**Pros: Less sensitive to imbalance**
**Cons: More classifications**

# SVM: **Kernel Trick**

## Kernel Trick: Visual




Kernel




Kernel Options
- Linear
- Radial Basis Function
- Polynomial
- Sigmoid



## Kernel Trick: Code

Original Code (linear)

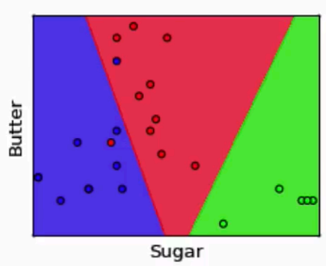
```
# Fit basic SVC model (linear kernel)
model = svm.SVC(kernel='linear')
model.fit(sugar_butter, type_label)
```

Updated Code (RBF)

```
# Fit the SVC model with radial kernel
model = svm.SVC(kernel='rbf', C=1, gamma=2**-5)
model.fit(sugar_butter, type_label)
```

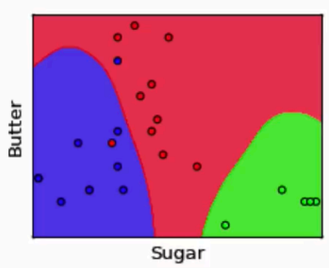


## Kernel Trick: Comparison



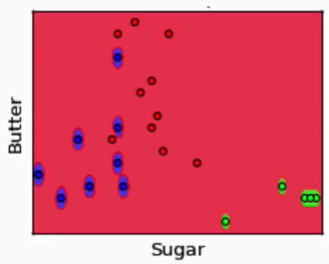

**Kernel**: Linear
**C**: 1

Muffin
Cupcake
Scone




**Kernel**: RBF
**C**: 1
**Gamma**: 2^-5

**Small Gamma:**
Less complexity




**Kernel**: RBF
**C**: 1
**Gamma**: 2^1

**Large Gamma:**
More complexity

# SVM: **Pros & Cons**

- **Pros**:
  - Good at dealing with high dimensional data
  - Works well on small data sets
- **Cons**:
  - Picking the right kernel and parameters can be computationally intensiv

# Unsupervised (Clustering / Grouping)

K-Means
Mean-Shift
PCA

# Clustering (Grouping)

- Grouping of **data points**
- Given a set of data points, use a clustering algorithm to **classify** each data point into a specific group
- **Same** group **similar** properties / features, **different** groups **dissimilar** properties / features
- Gain valuable insights by seeing what **groups** the data points fall into.

# K-Means

## Clustering (Unsupervised)

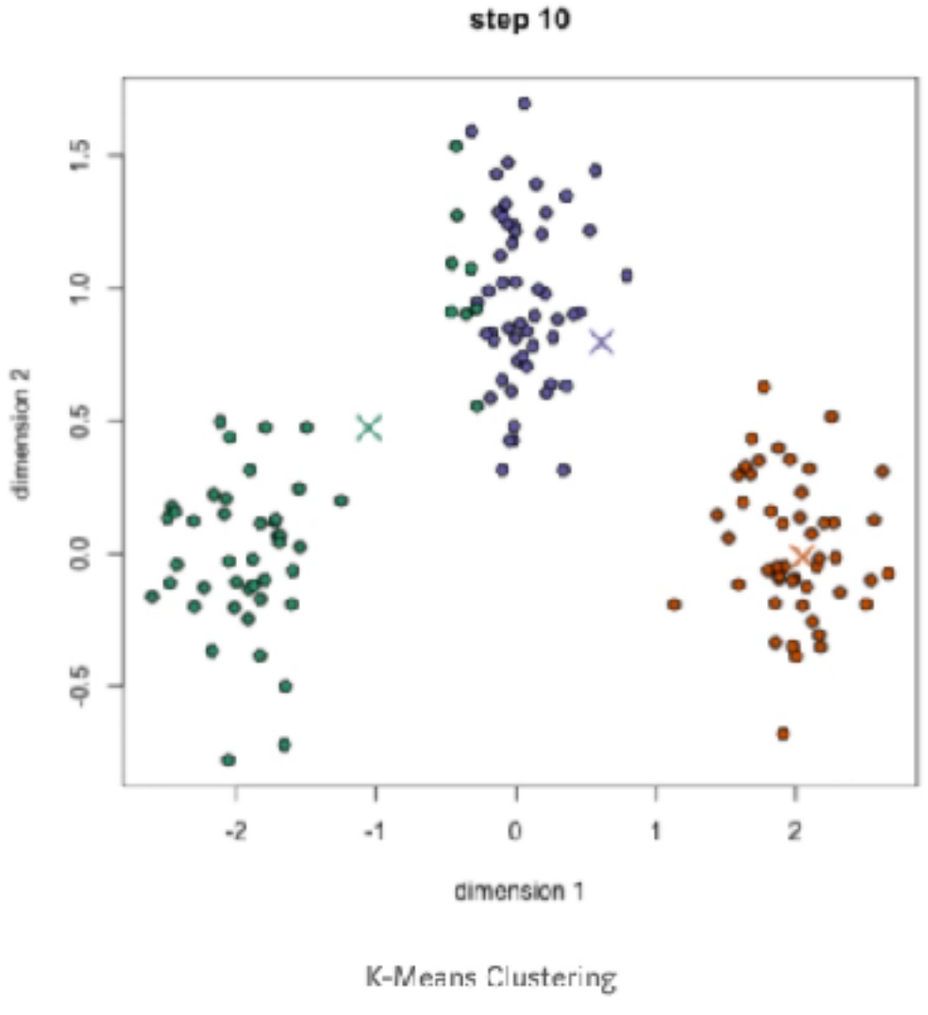
# K-Means

- Select **number** of classes / groups and **randomly** initialize their respective center points. Center points have same **length** as data points (X to left)

- Each data point is classified based on **distance** to each group center.

- **Recompute** the group center by taking the mean of all the vectors in the group.

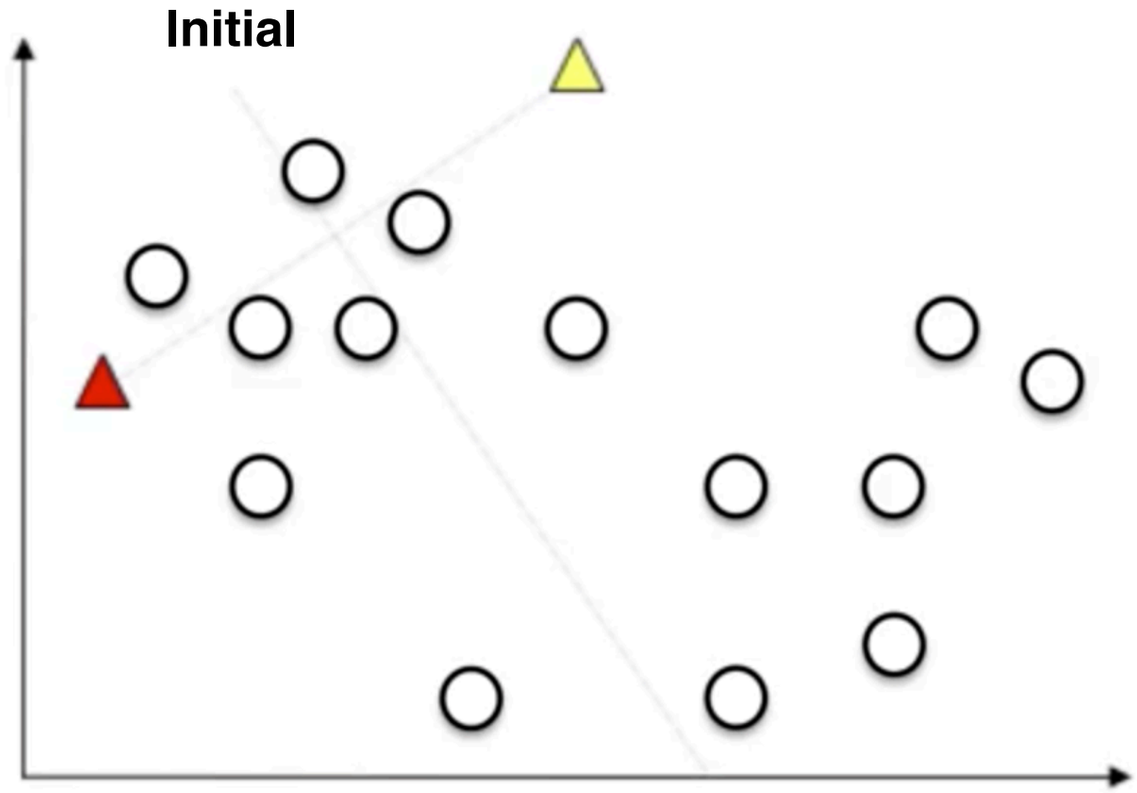
- **Repeat** a fixed number of **iterations** or until no change in group centers.



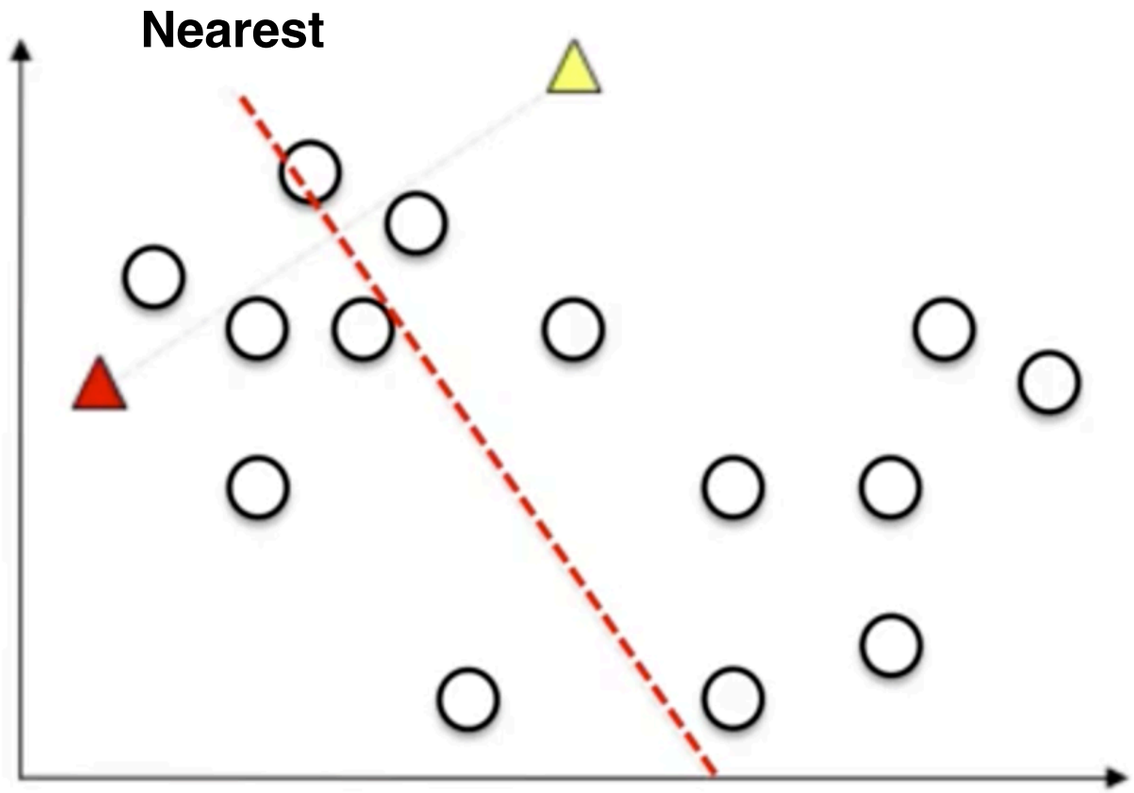
# K-Means (2)

- **Pros**:
  - pretty fast, only computes distances between points and group centers, linear complexity O(n)
- **Cons**:
  - have to select how many groups/classes there are
  - starts with a random choice of cluster centers (different results on different runs)
- **K-Medians**: use median vector of the group, less sensitive to outliers, slower due to sorting

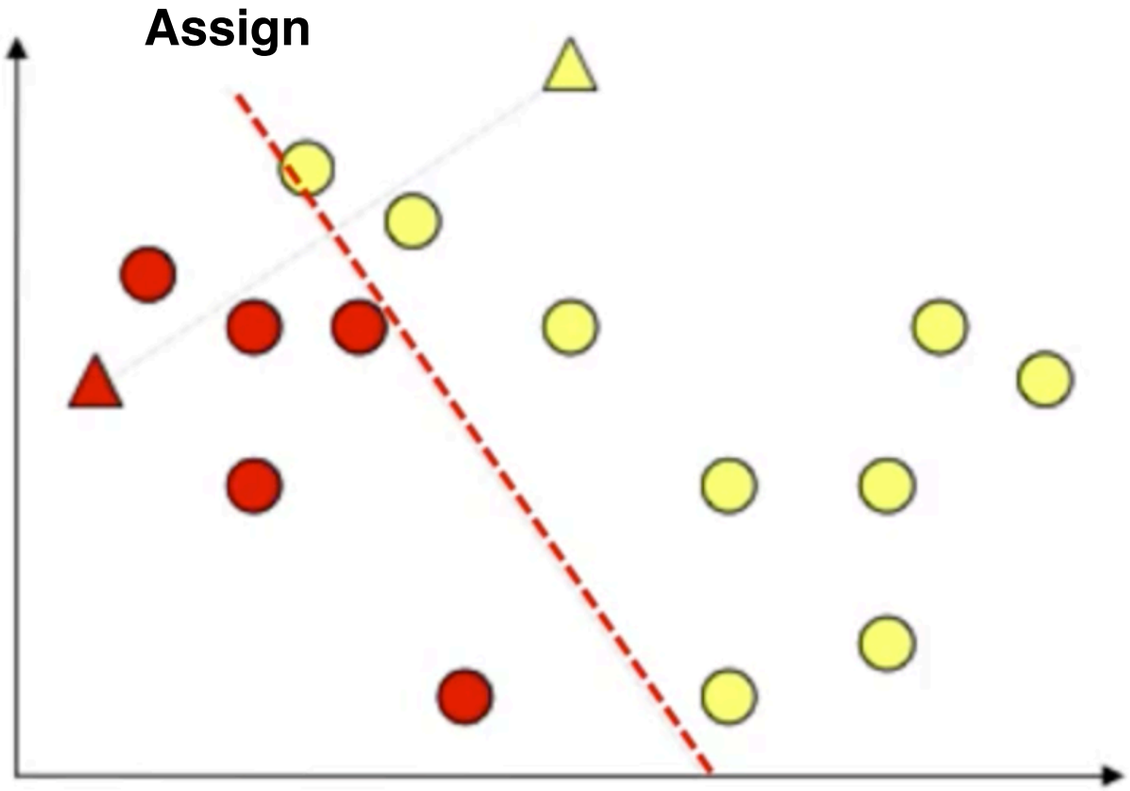
# K-means clustering example



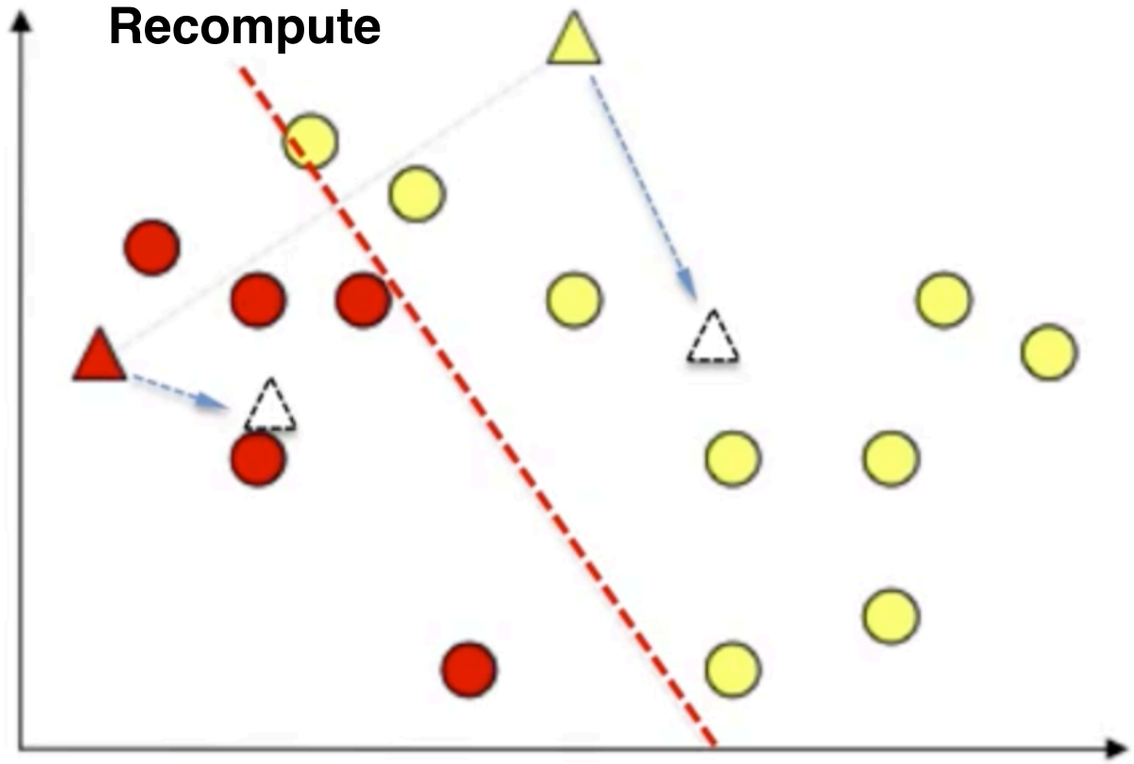
# K-means clustering example



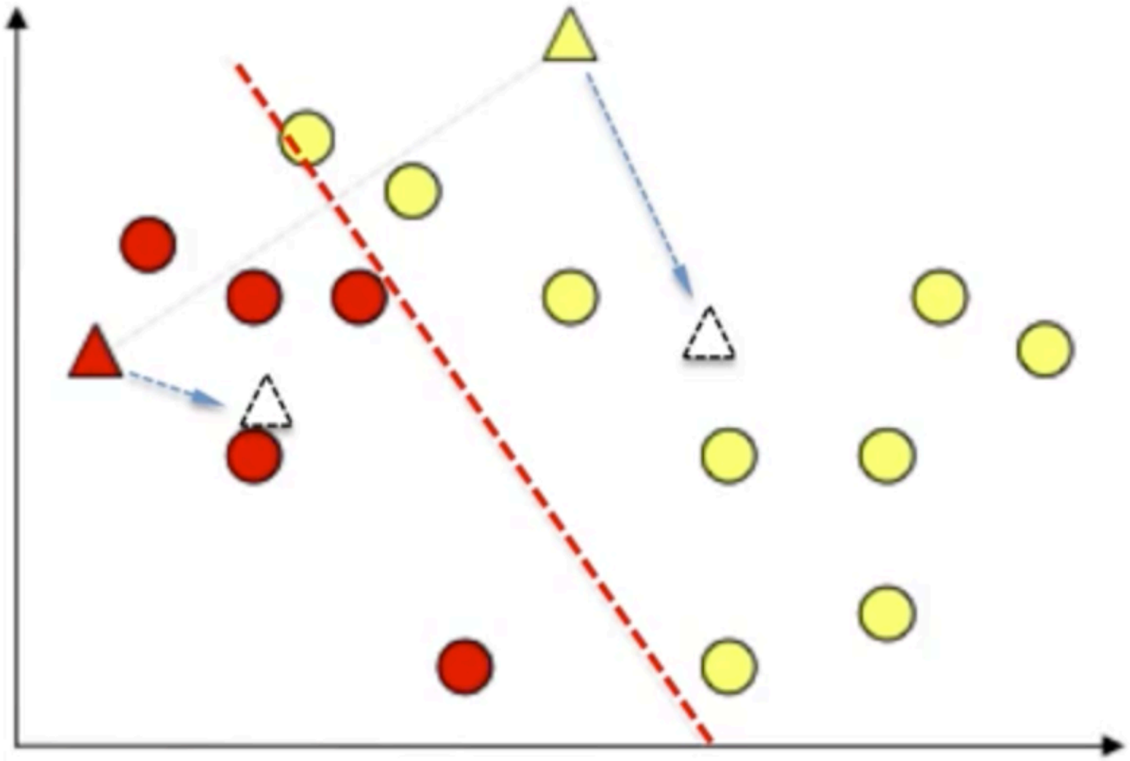
# K-means clustering example



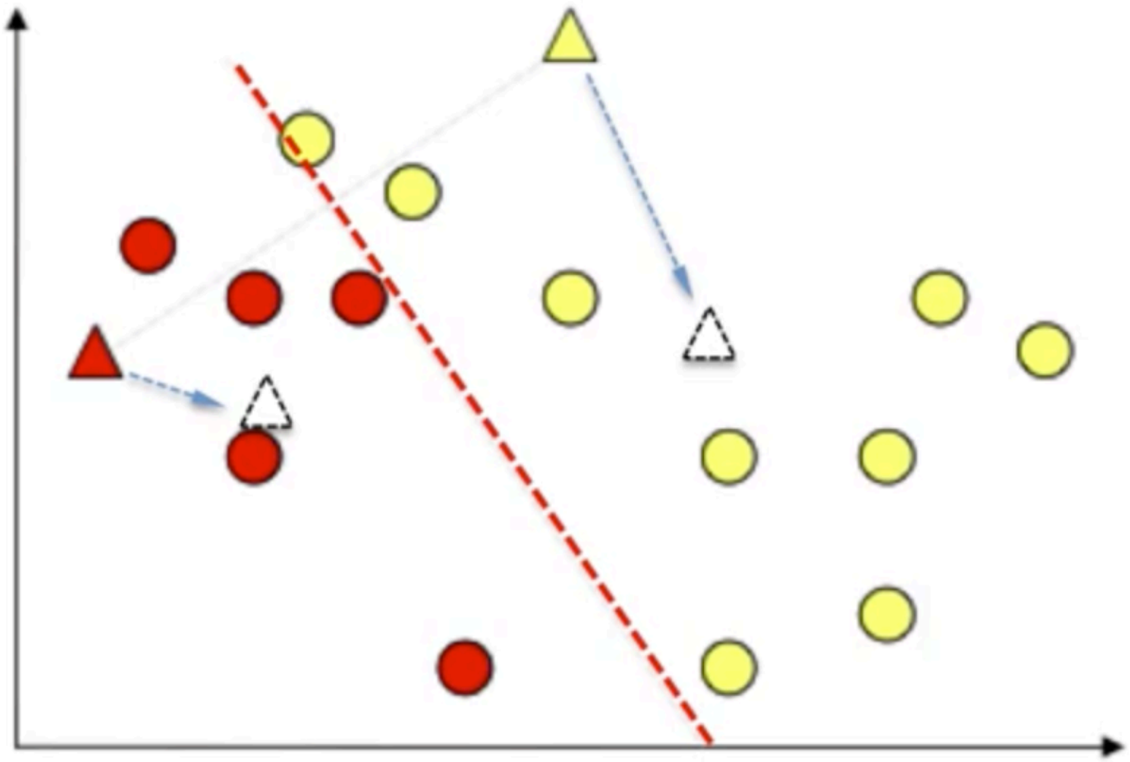
# K-means clustering example

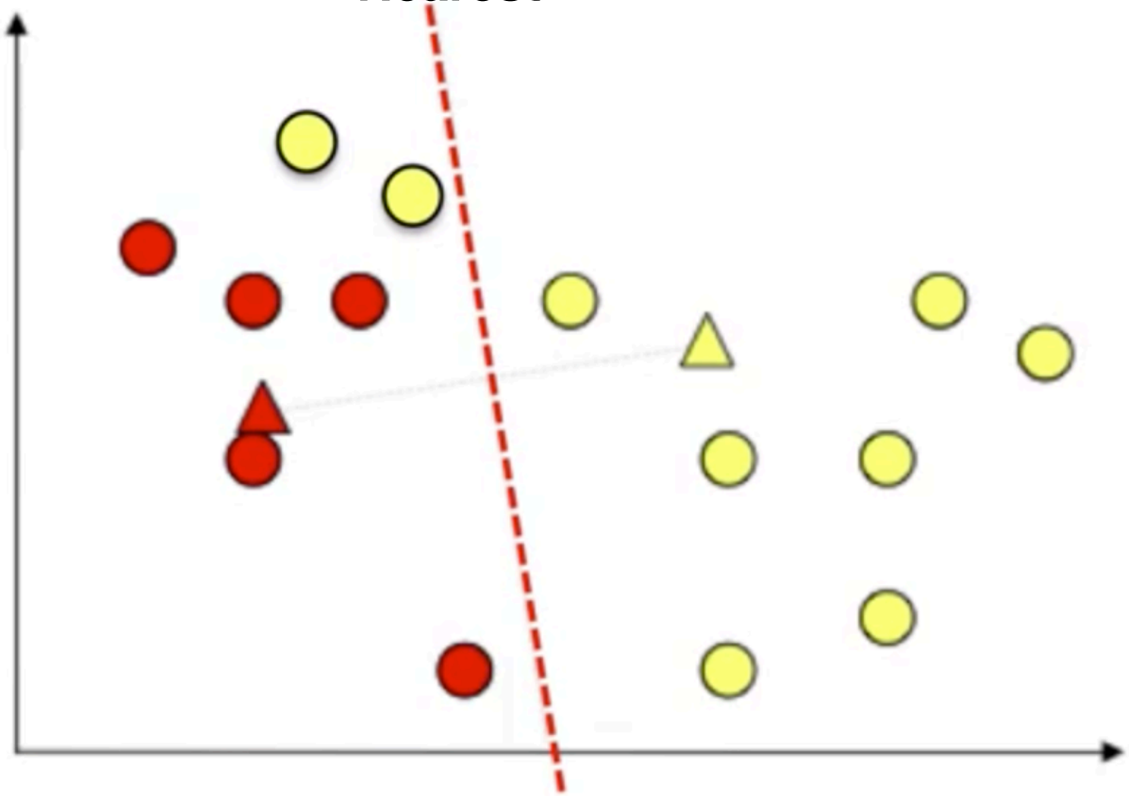


# K-means clustering example

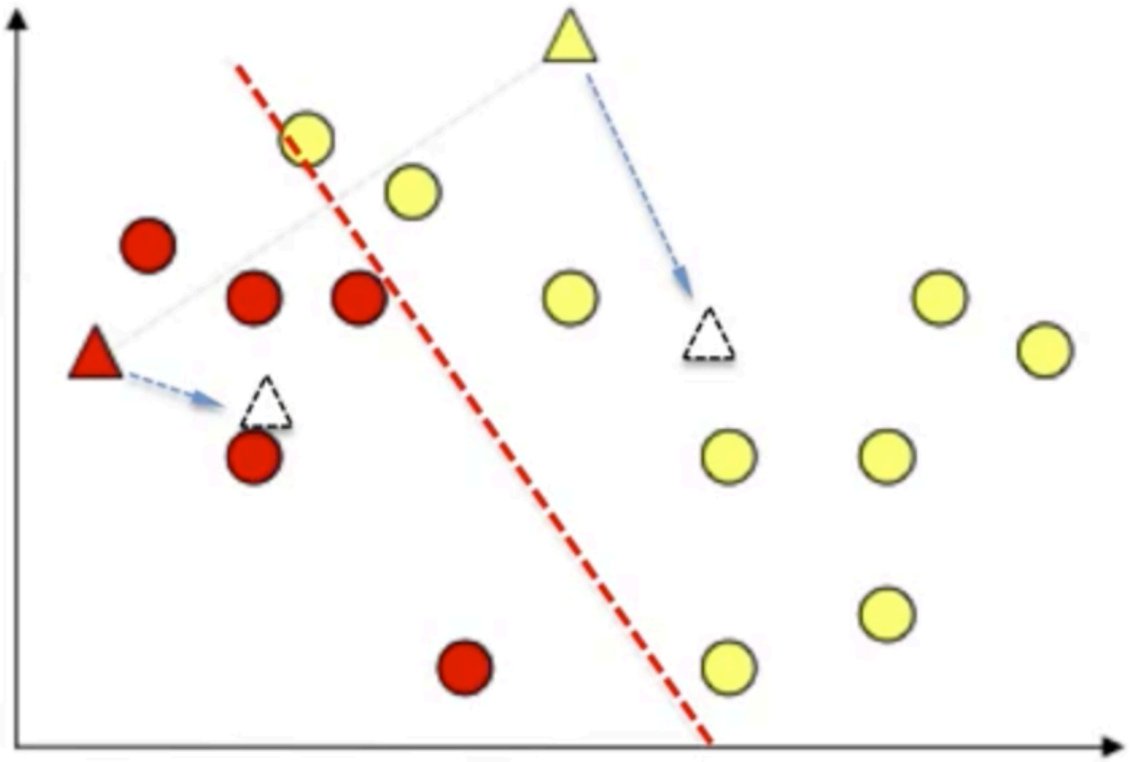


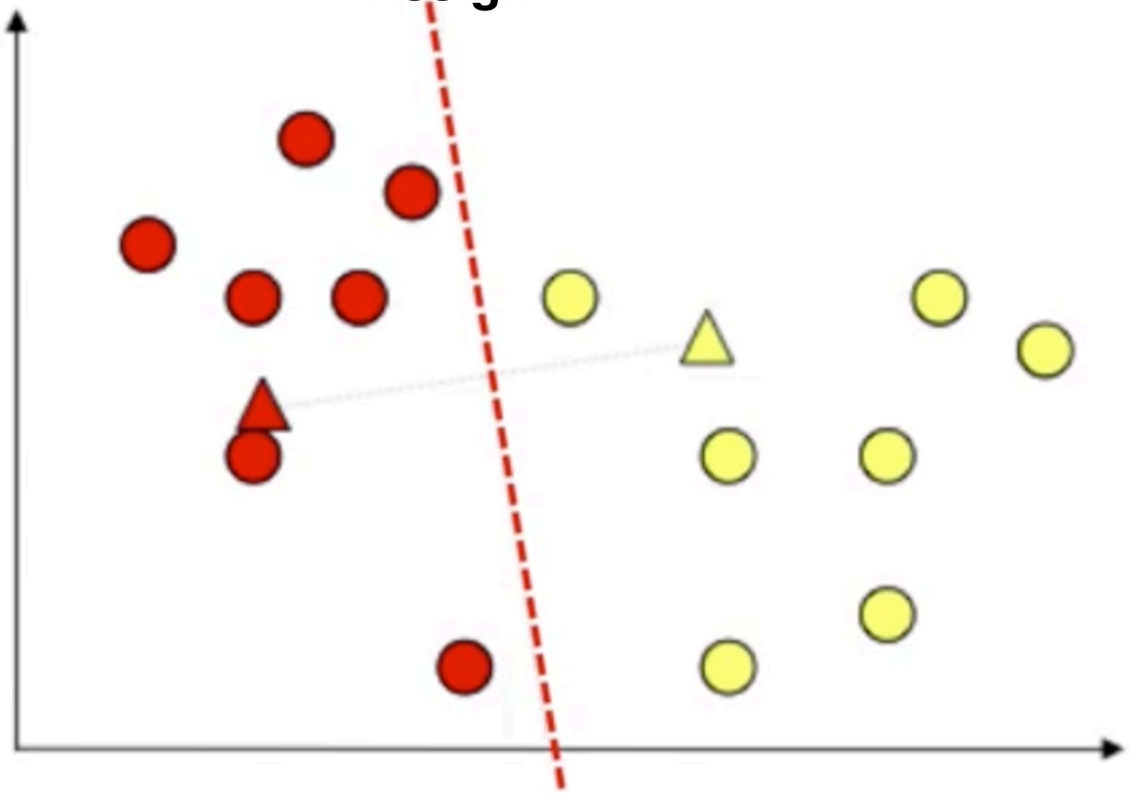
# K-means clustering example



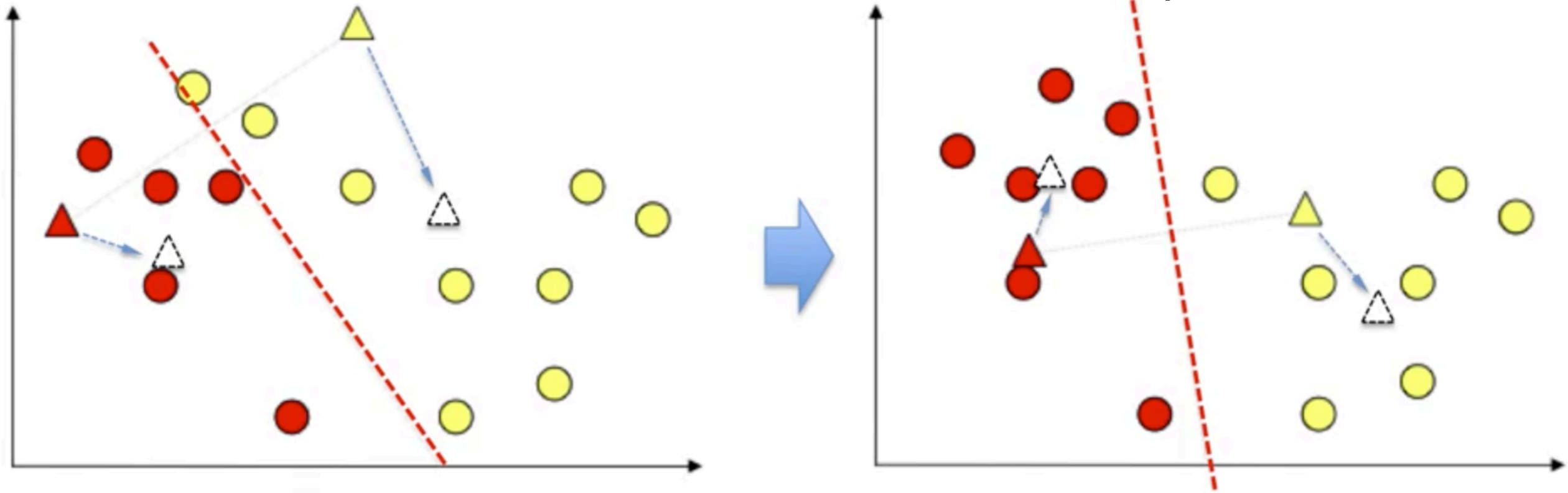


# K-means clustering example

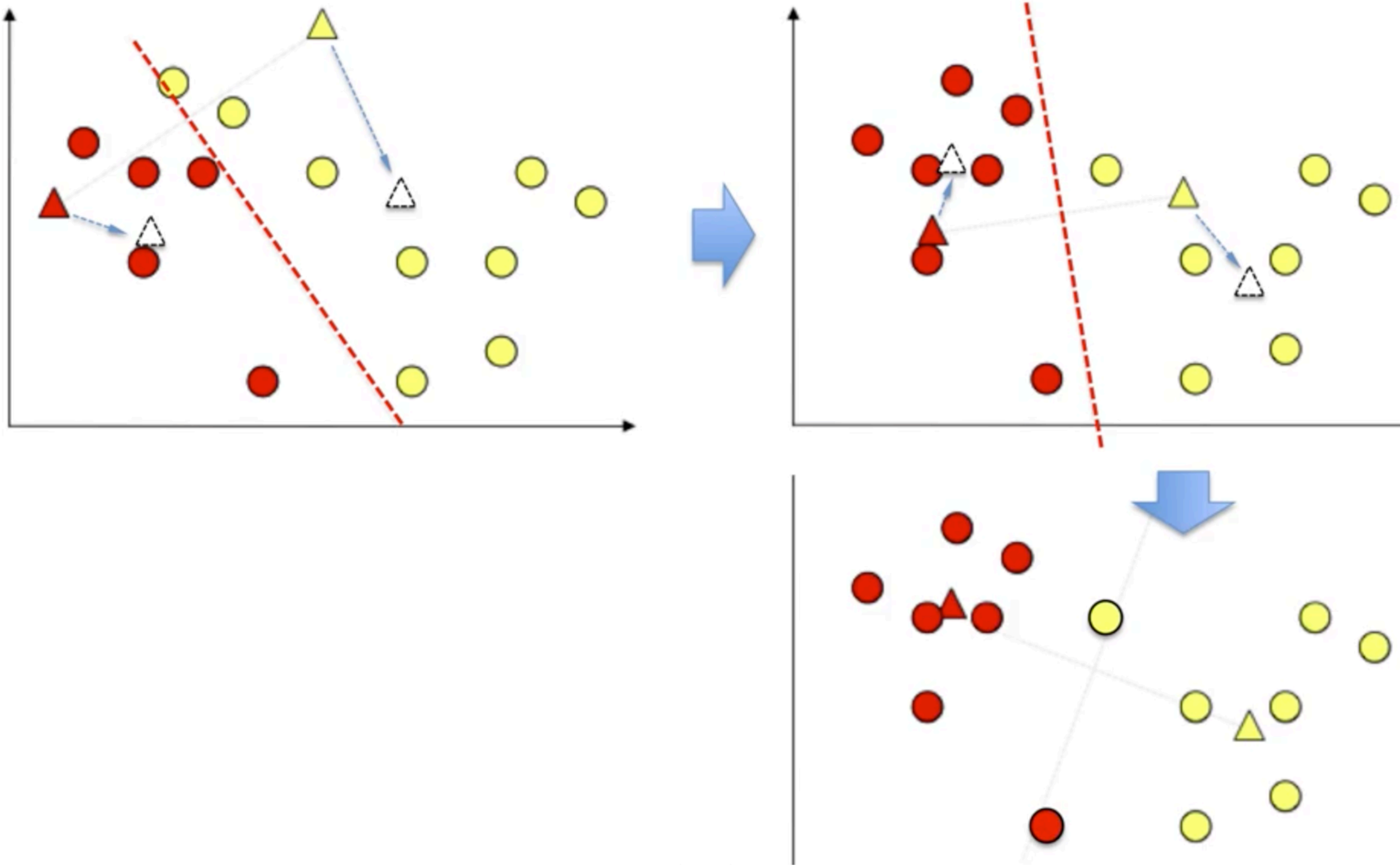




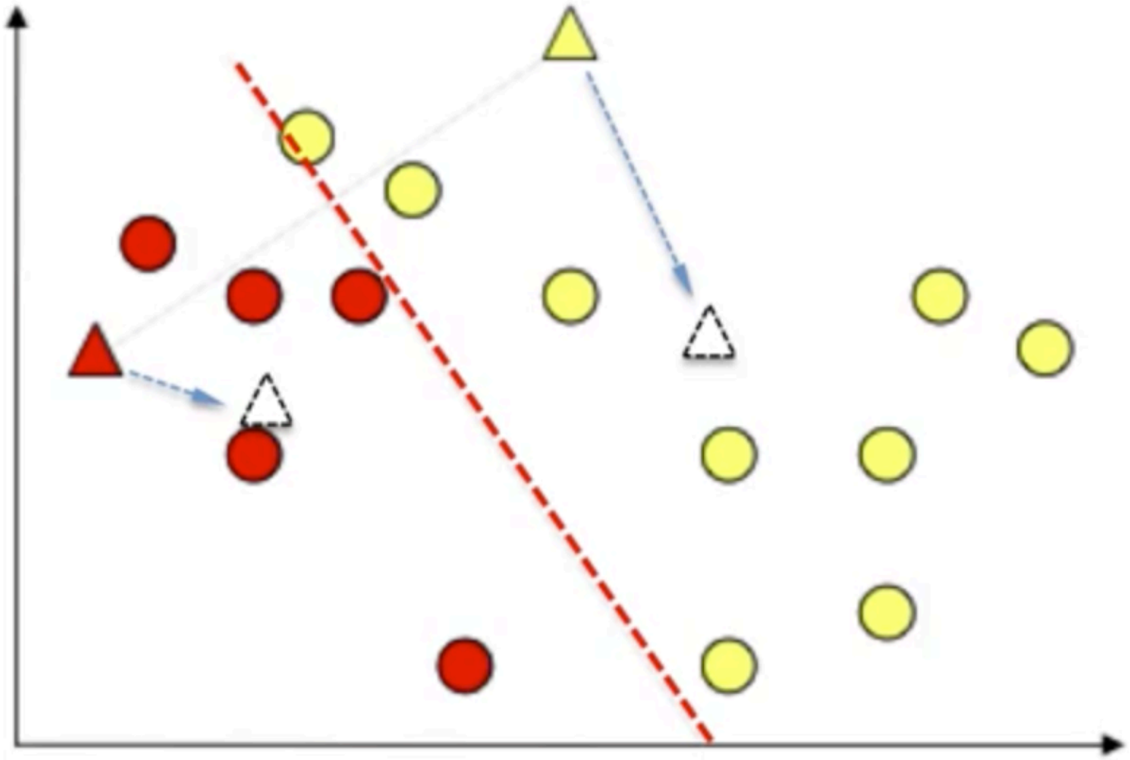
# K-means clustering example

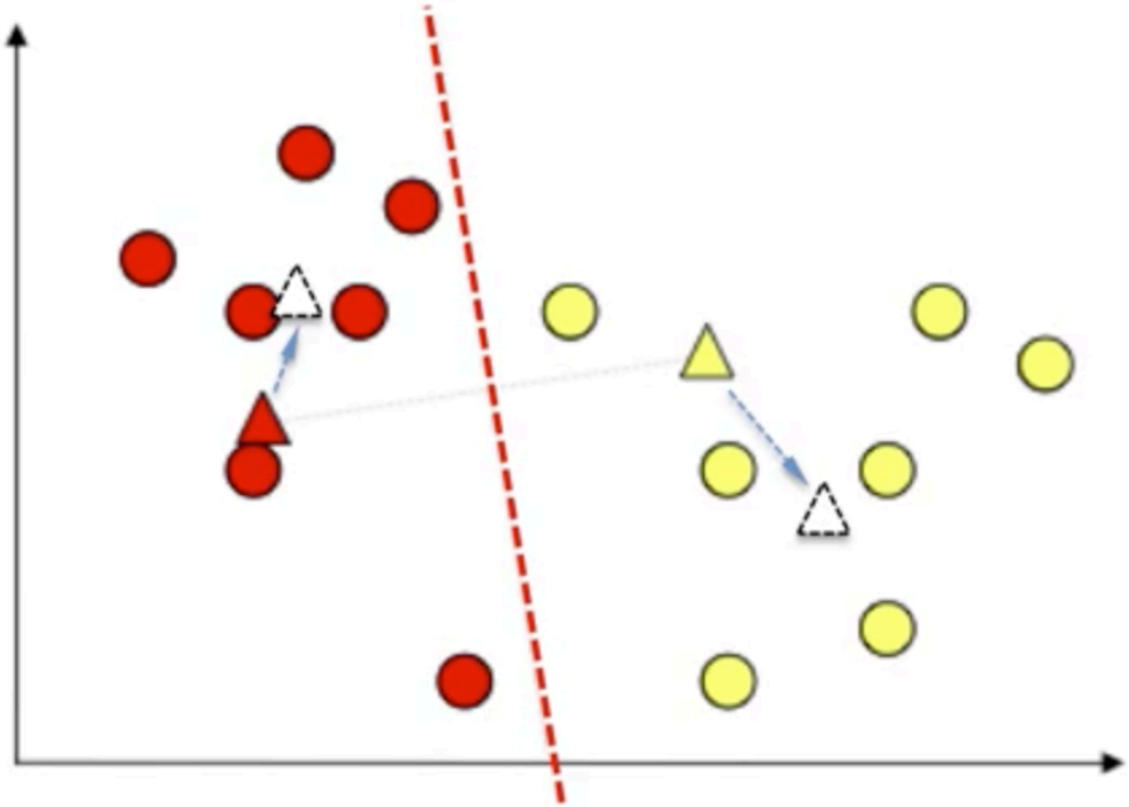


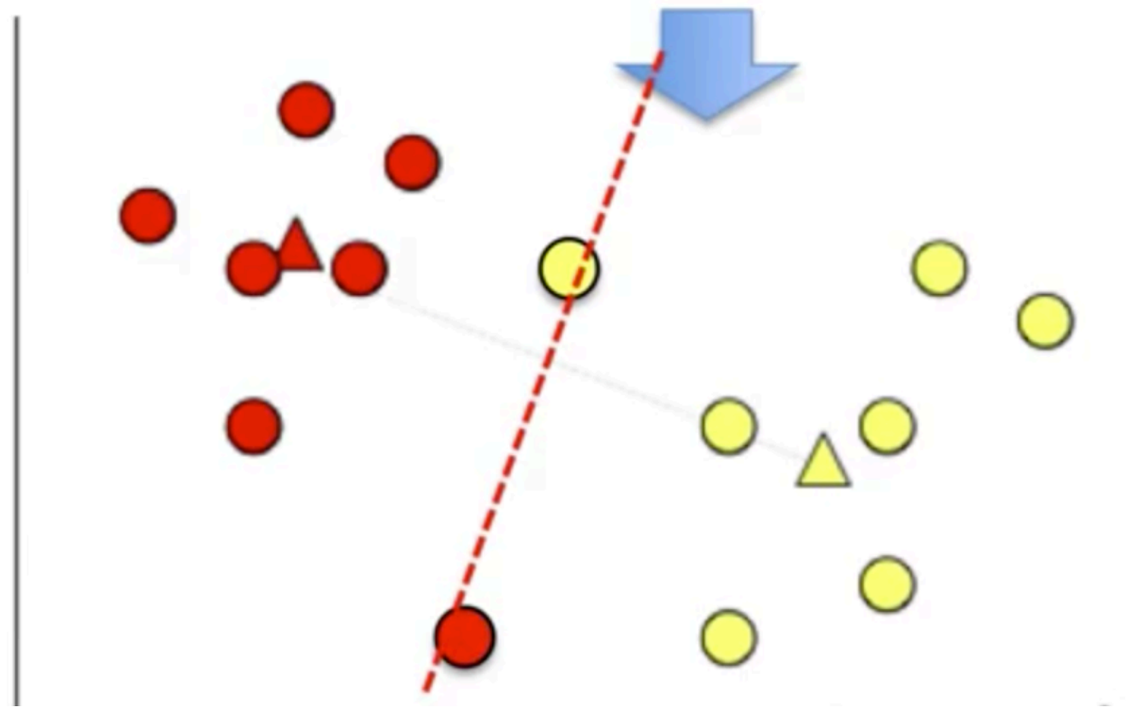
# K-means clustering example



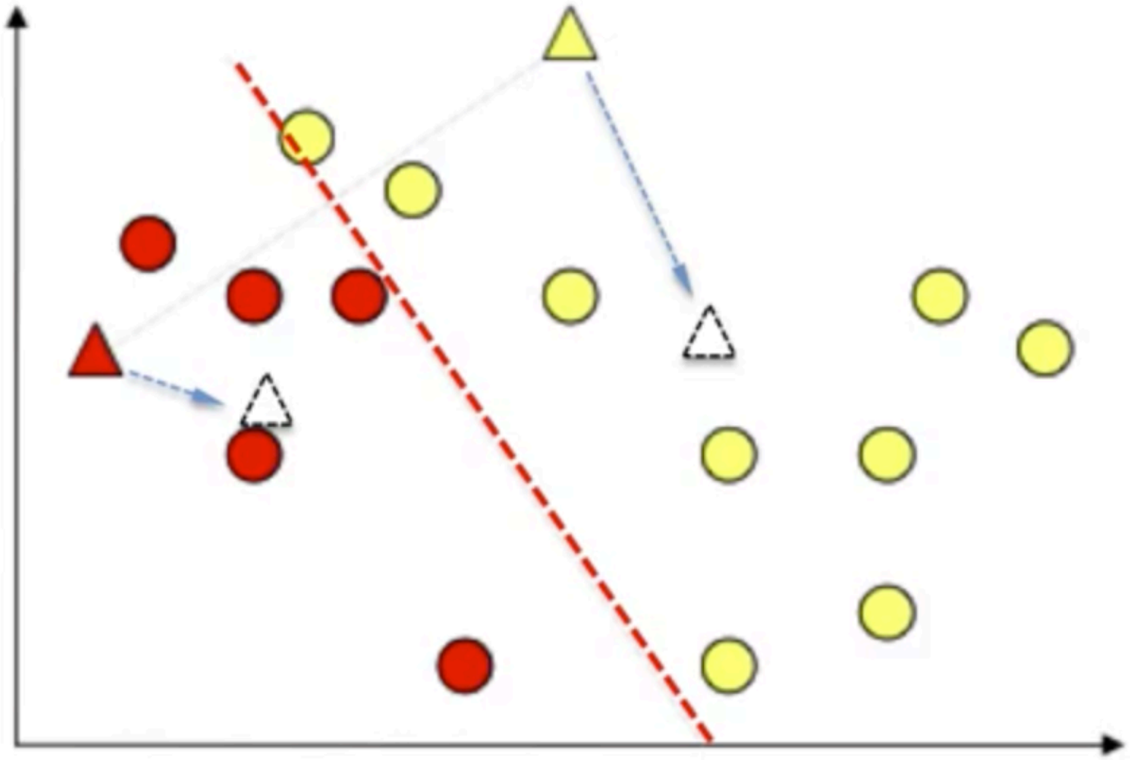
# K-means clustering example

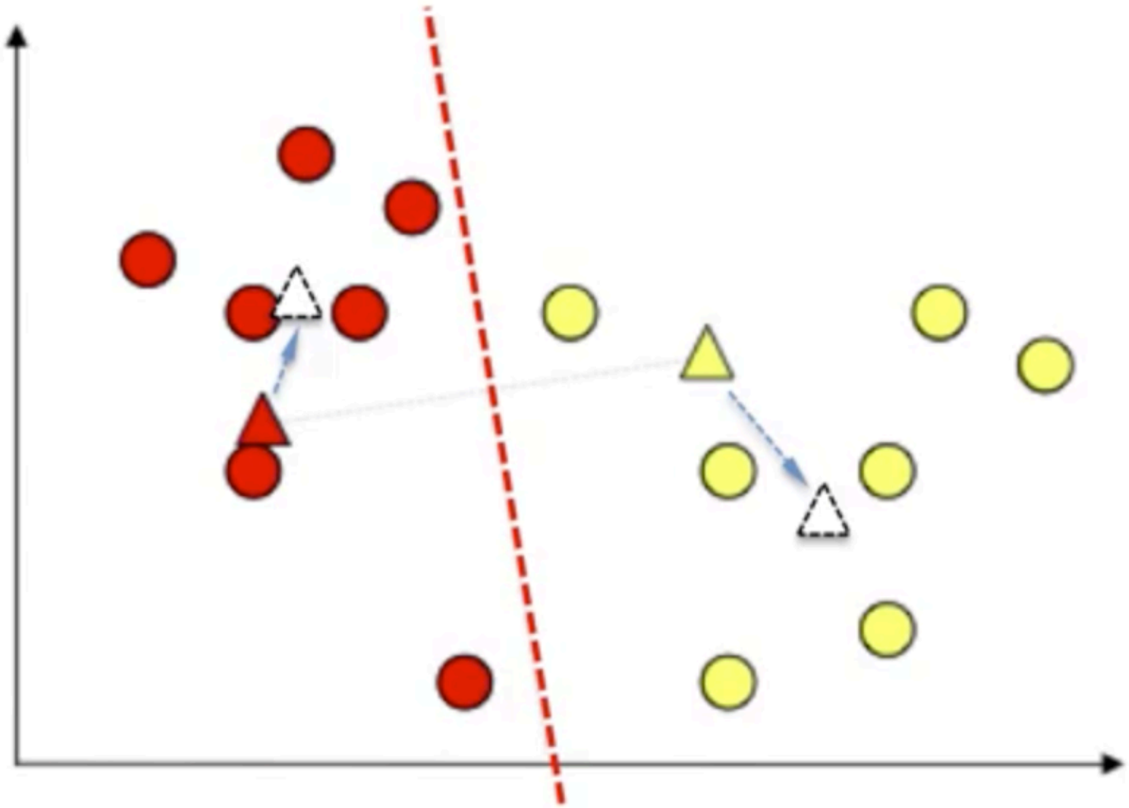


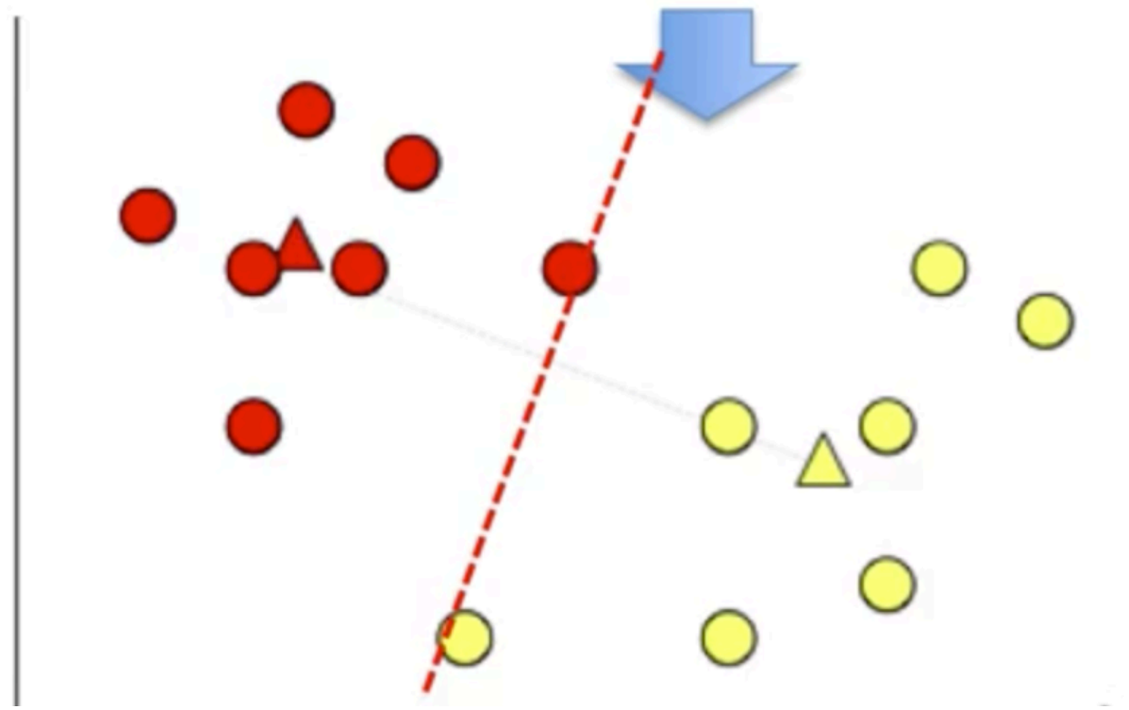




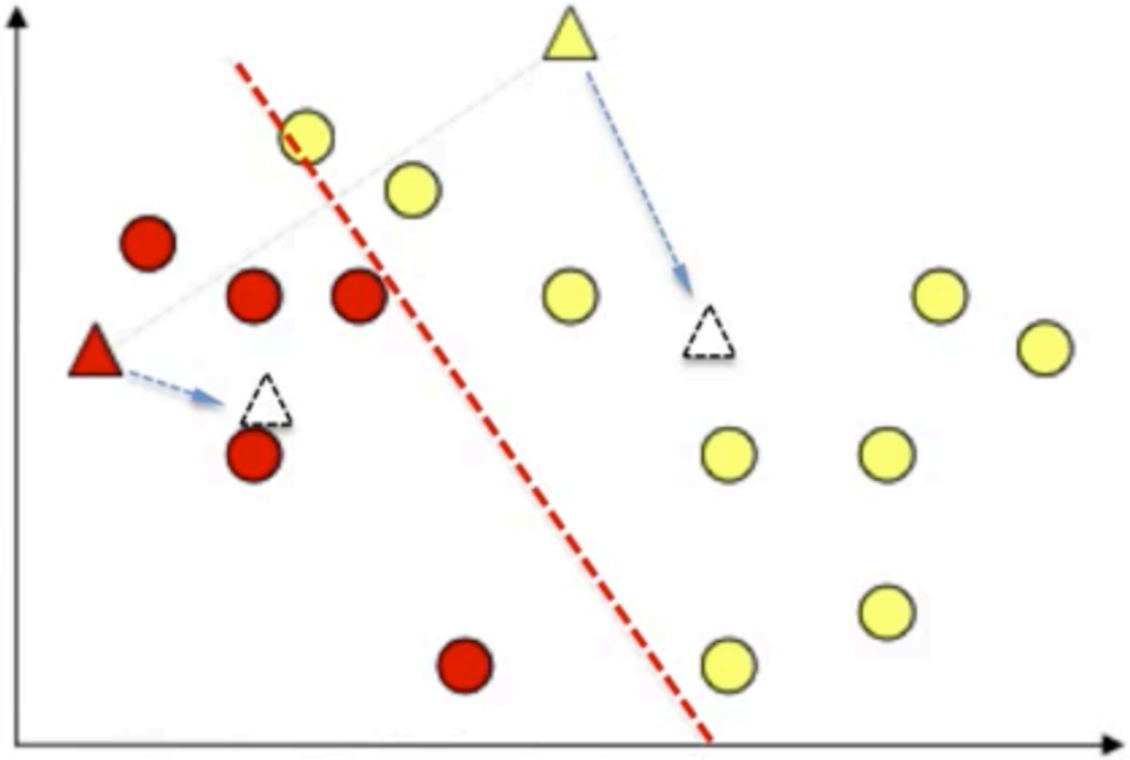
# K-means clustering example

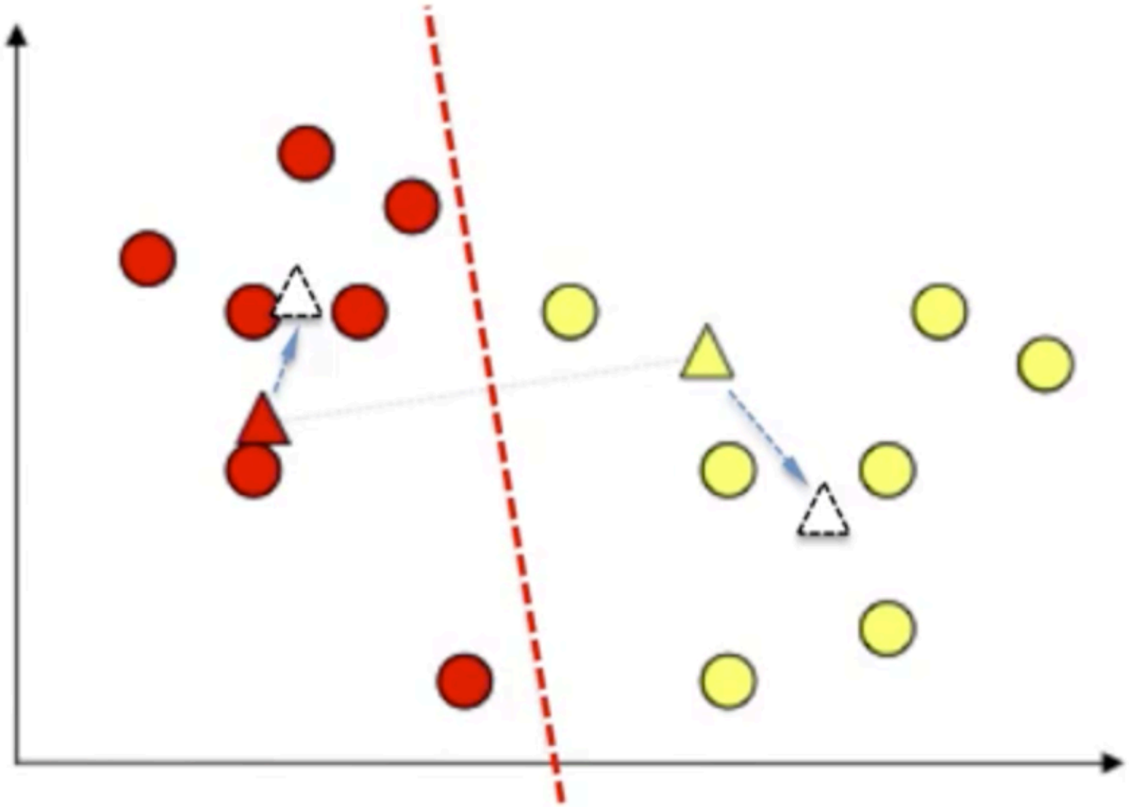


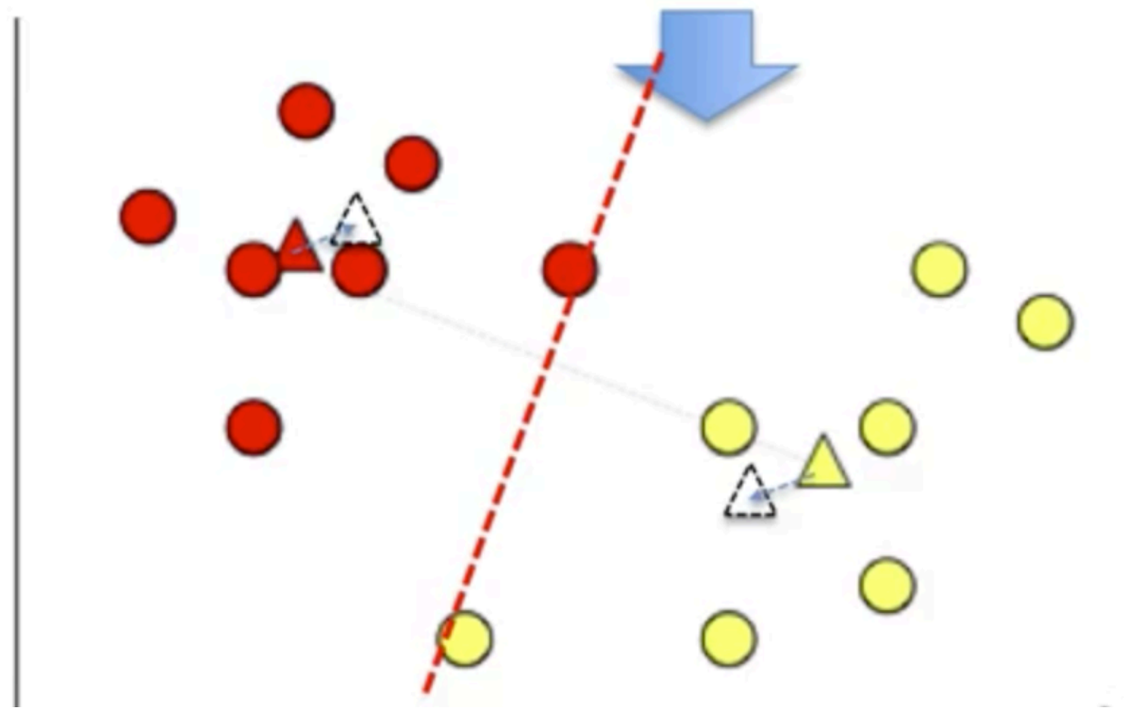




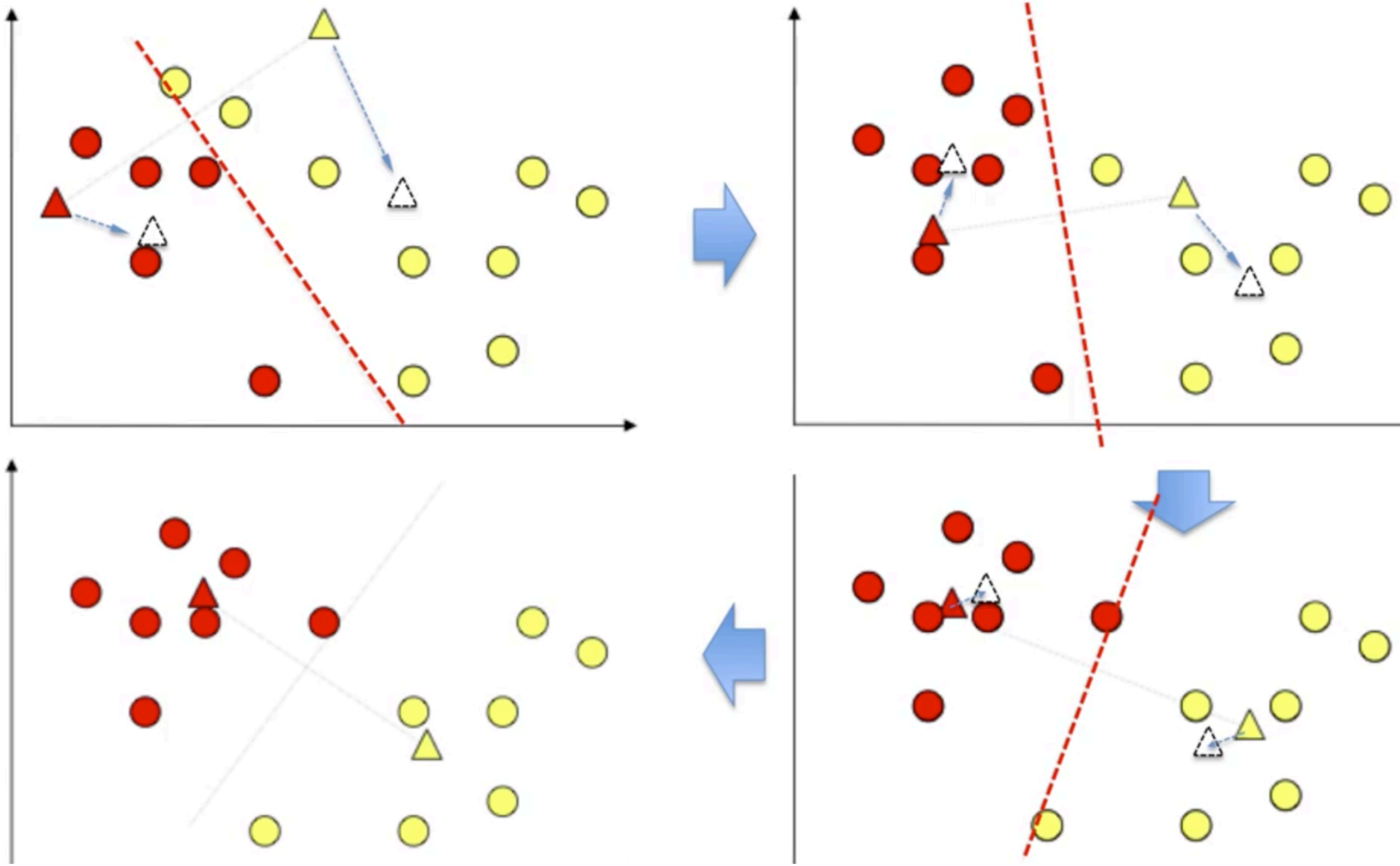
# K-means clustering example



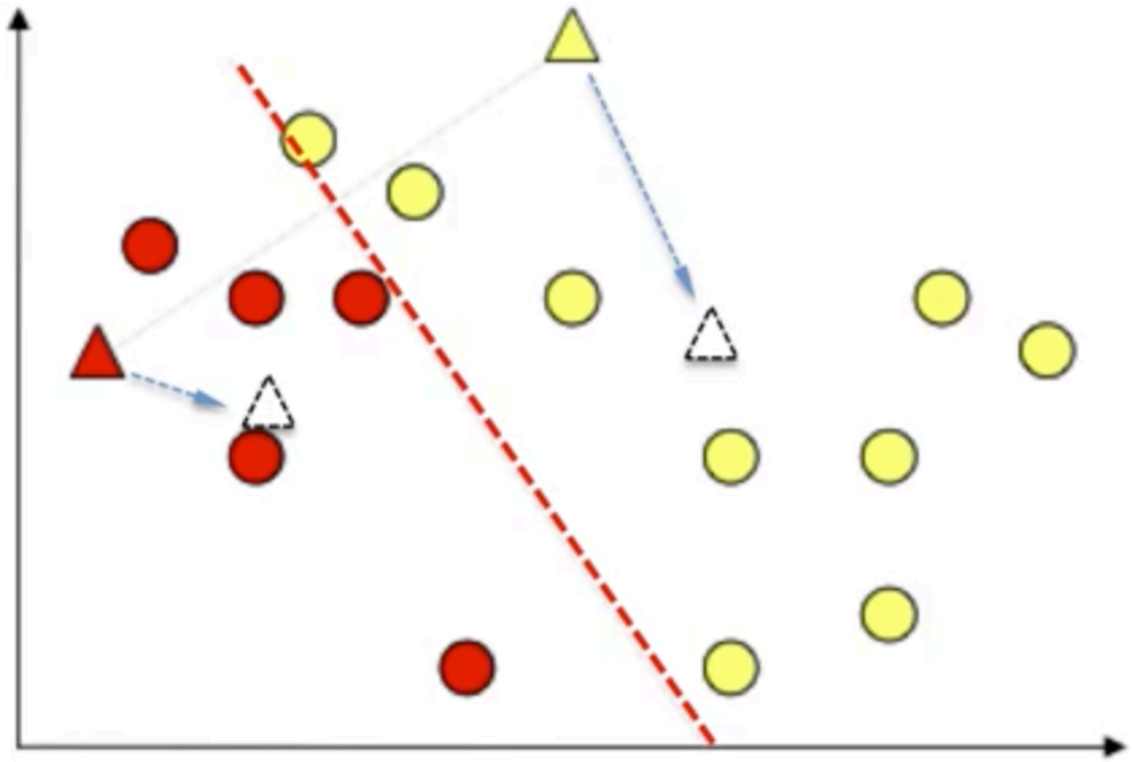


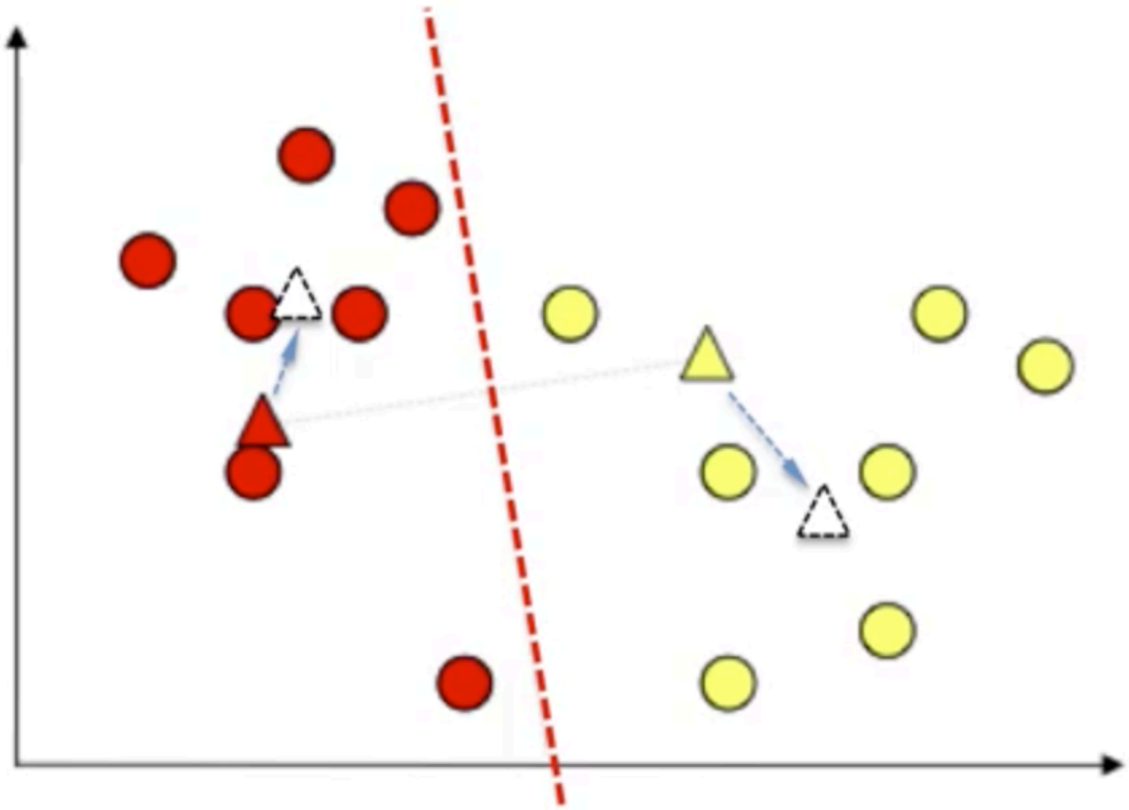


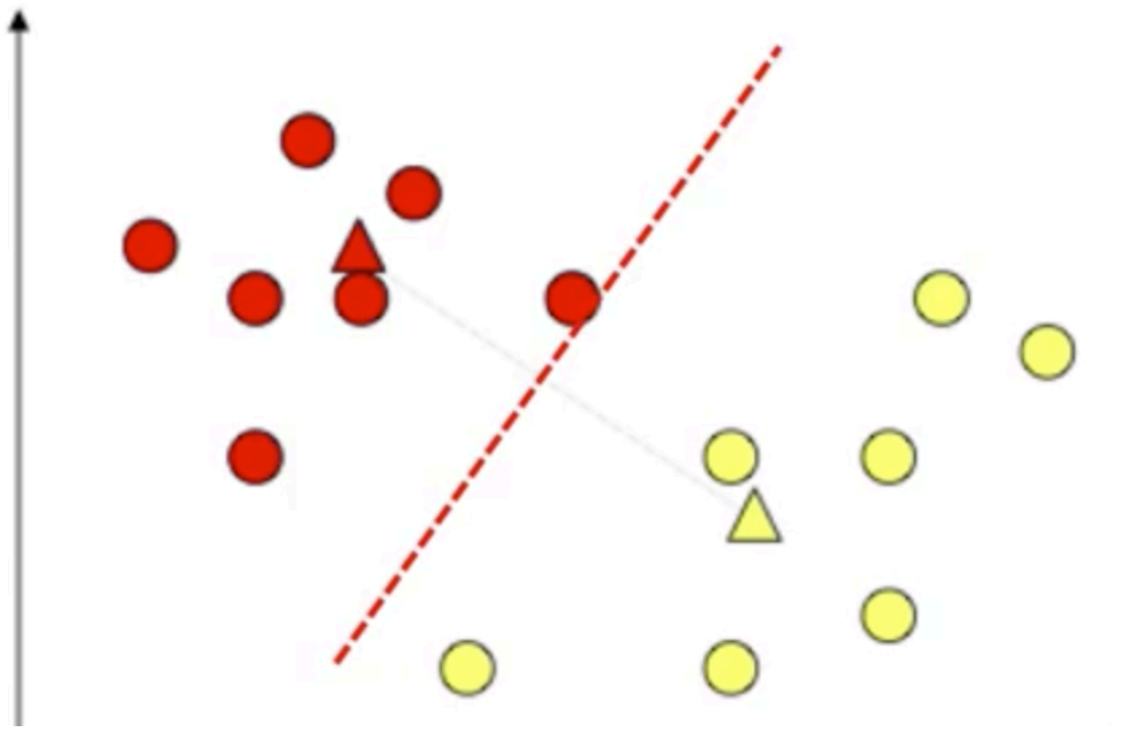
# K-means clustering example

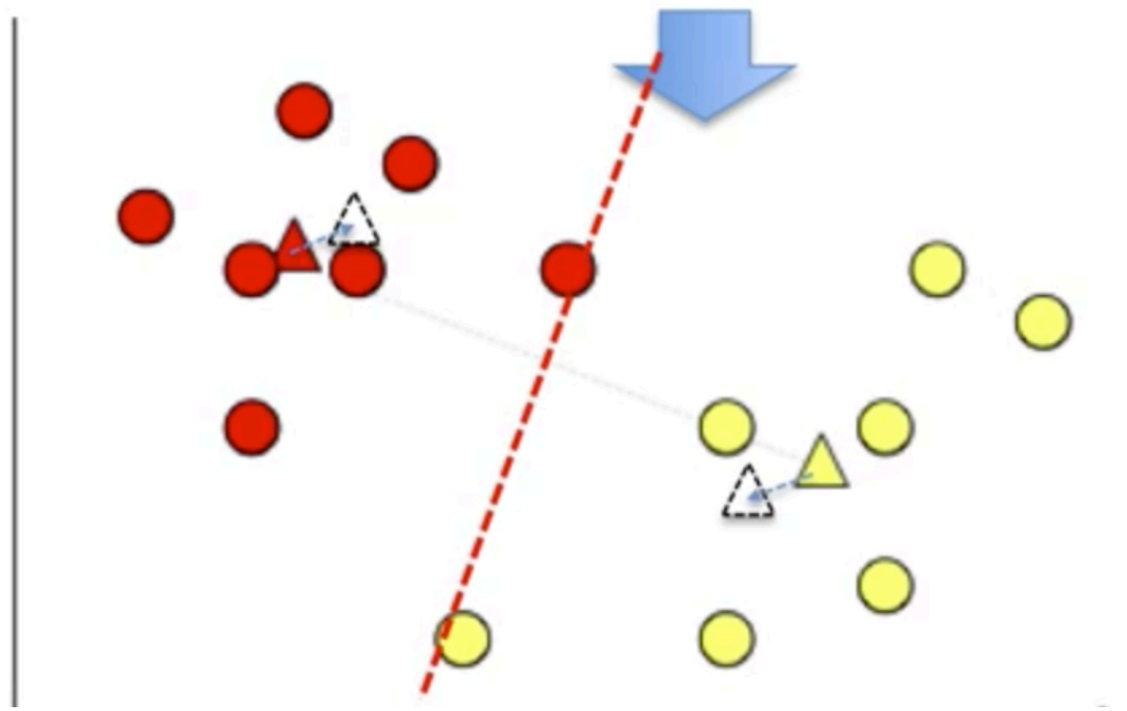


# K-means clustering example









# Mean-Shift

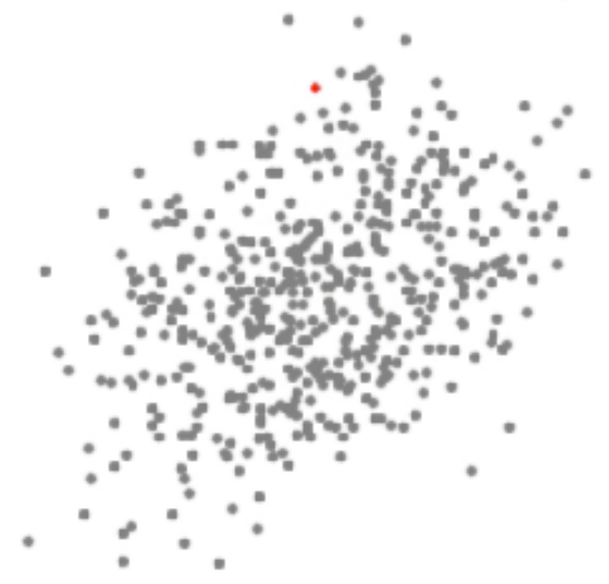
## Clustering (Unsupervised)

# Mean-Shift

- a **sliding**-window-based algorithm that attempts to find **dense** areas of data points

- It is a **centroid**-based algorithm that locate the **center** points of each group/ class (**update** the candidate center points to be the mean of the points within the sliding-window).

- The candidate windows are then **filtered** in a post-processing stage to eliminate near-duplicates, forming the final set of center points and their corresponding groups

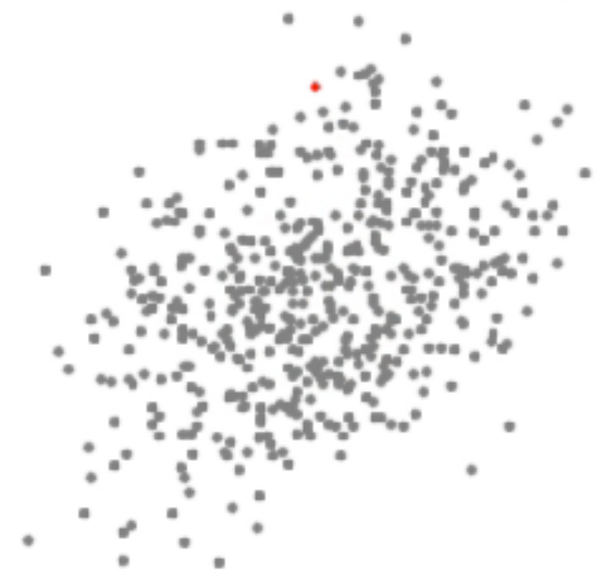


Mean-Shift Clustering for a single sliding window

# Mean-Shift (2)

- Explanation: set of points in 2D space.
  **Circular** sliding window **centered** at a point C (randomly selected) and having **radius** r as the kernel.

- **Hill climbing** algorithm: At every iteration the sliding window is **shifted** towards regions of **higher density** by shifting the center point to the **mean** of the points within the window (hence the name). Density within the sliding window is **proportional** to the number of points inside it, by shifting to the **mean** of the points in the window it will gradually move towards areas of higher point density.

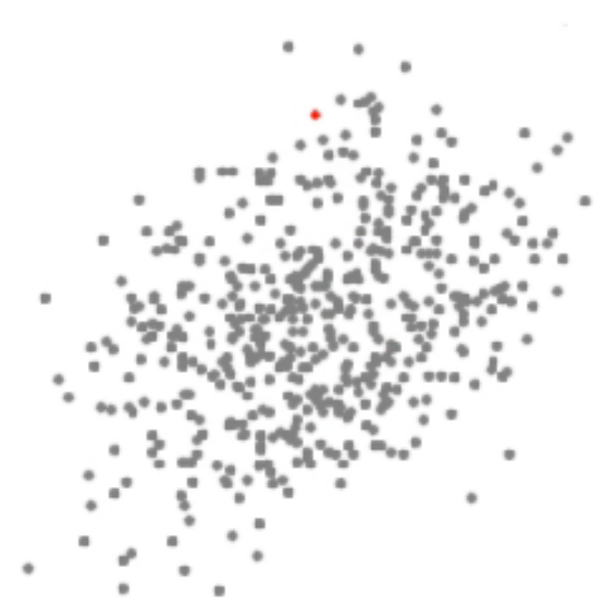


Mean-Shift Clustering for a single sliding window

# Mean-Shift (3)

- **Continue shifting** the sliding window according to the **mean** until there is no **direction** at which a shift can **accommodate** more points inside the kernel (keep moving the circle until we no longer are increasing the density, i.e number of points in the window)

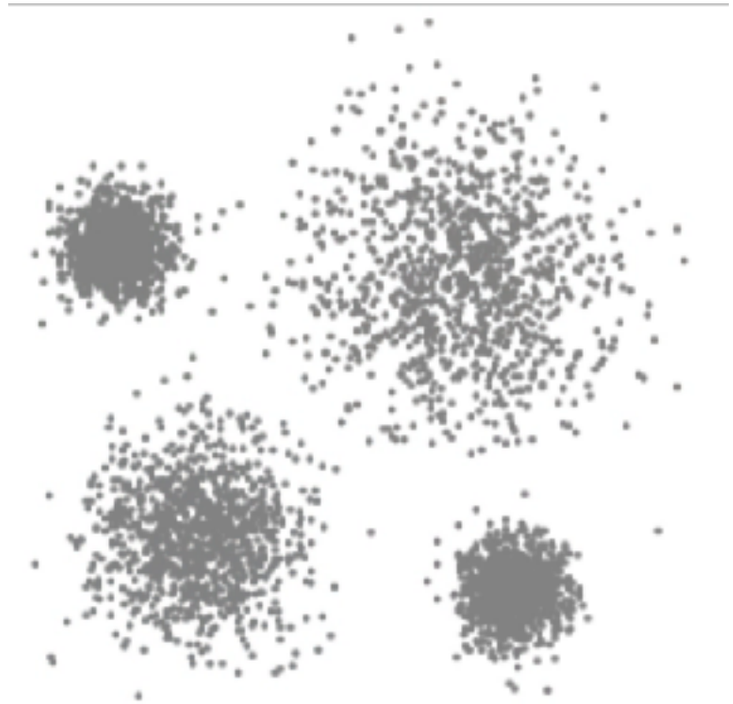
- This process is done with **many** sliding windows. When multiple sliding windows **overlap** the window containing the most points is preserved. The data points are then **clustered** according to the sliding window in which they reside.



Mean-Shift Clustering for a single sliding window

# Mean-Shift (4)

- An illustration of the entire process with **all** the sliding windows is show to the right. Each **black** dot represents the centroid of a sliding window and each **gray** dot is a data point.



The entire process of Mean-Shift Clustering

- Pros:
  - no need to select the number of clusters as mean-shift automatically discovers this
  - the fact that the cluster **centers** converge towards the points of **maximum density** is also quite desirable as it is quite intuitive to understand and fits well in a naturally data-driven sense
- Cons:
  - the selection of window size/radius “r” can be non-trivial.

# PCA:
# Principal Components Analysis

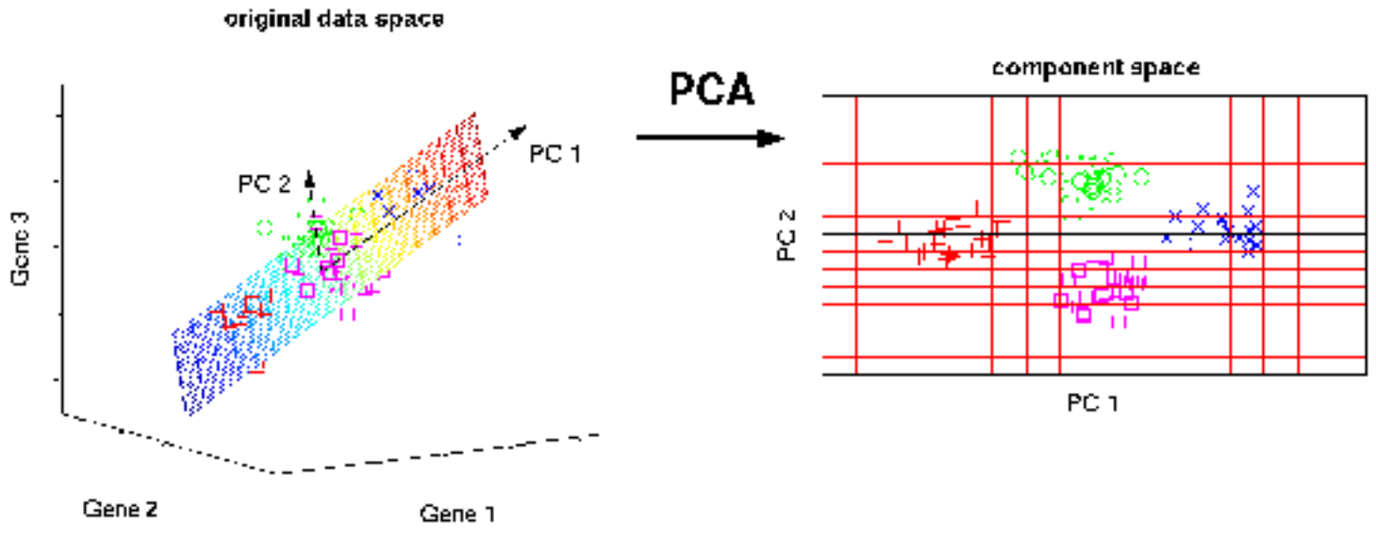
Dimensionality reduction & recognition

# PCA

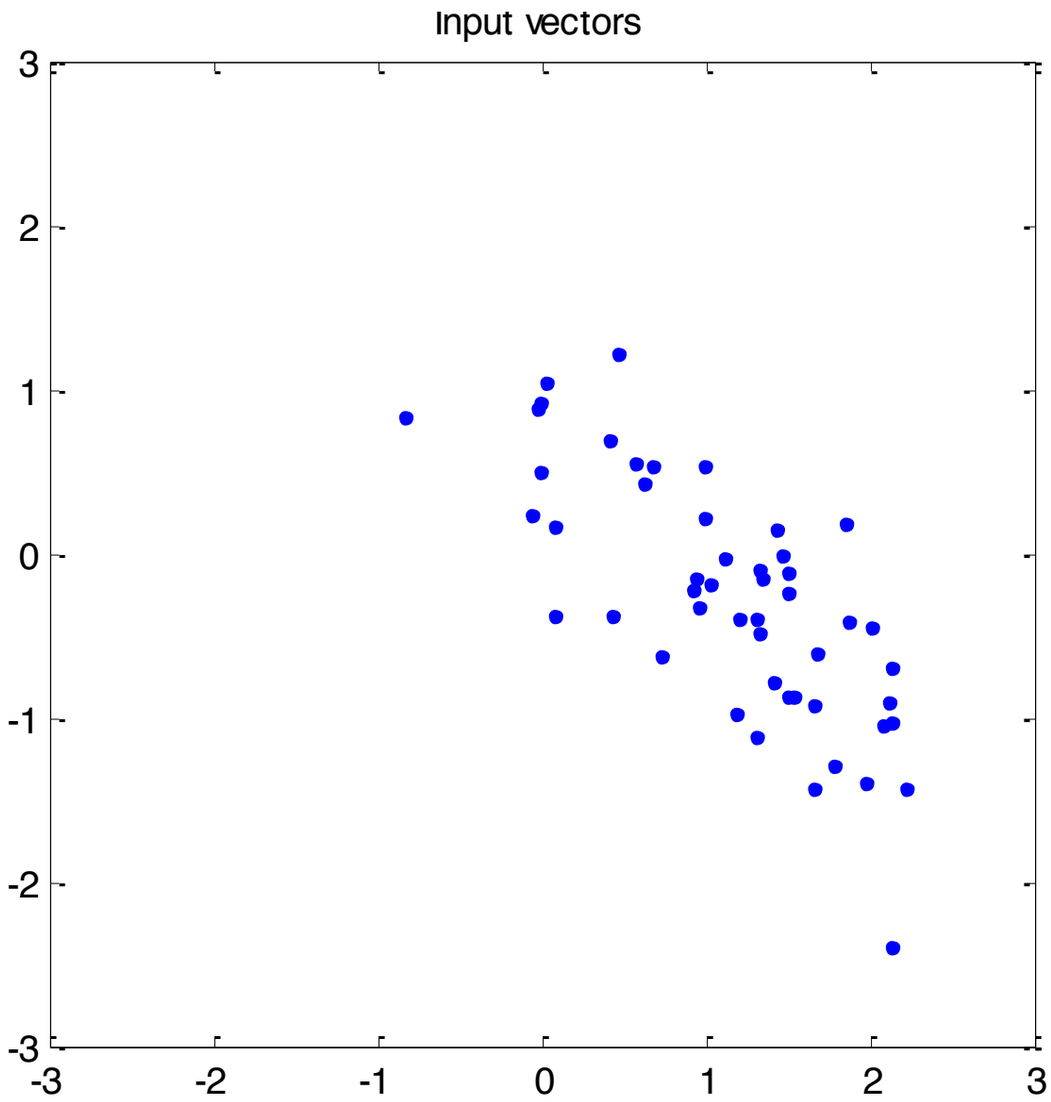
- Unsupervised method to understand **global properties** of a dataset consisting of vectors

- The **covariance matrix** of the data points is analyzed here to understand what **dimensions** are more **important** (i.e. have high variance).

- One way to think of the top **PCs** (principal components) of a matrix is to think of its **eigenvectors** with highest **eigenvalues**



# PCA: Intro





**Project:**

$$\mathbf{y} = \mathbf{A}\left(\mathbf{x}_{in} - \mathbf{m}_{x}\right)$$

$$\mathbf{y} = \mathbf{A}_{k}\left(\mathbf{x} - \mathbf{m}_{\mathbf{x}}\right)$$

**Reconstruct:**

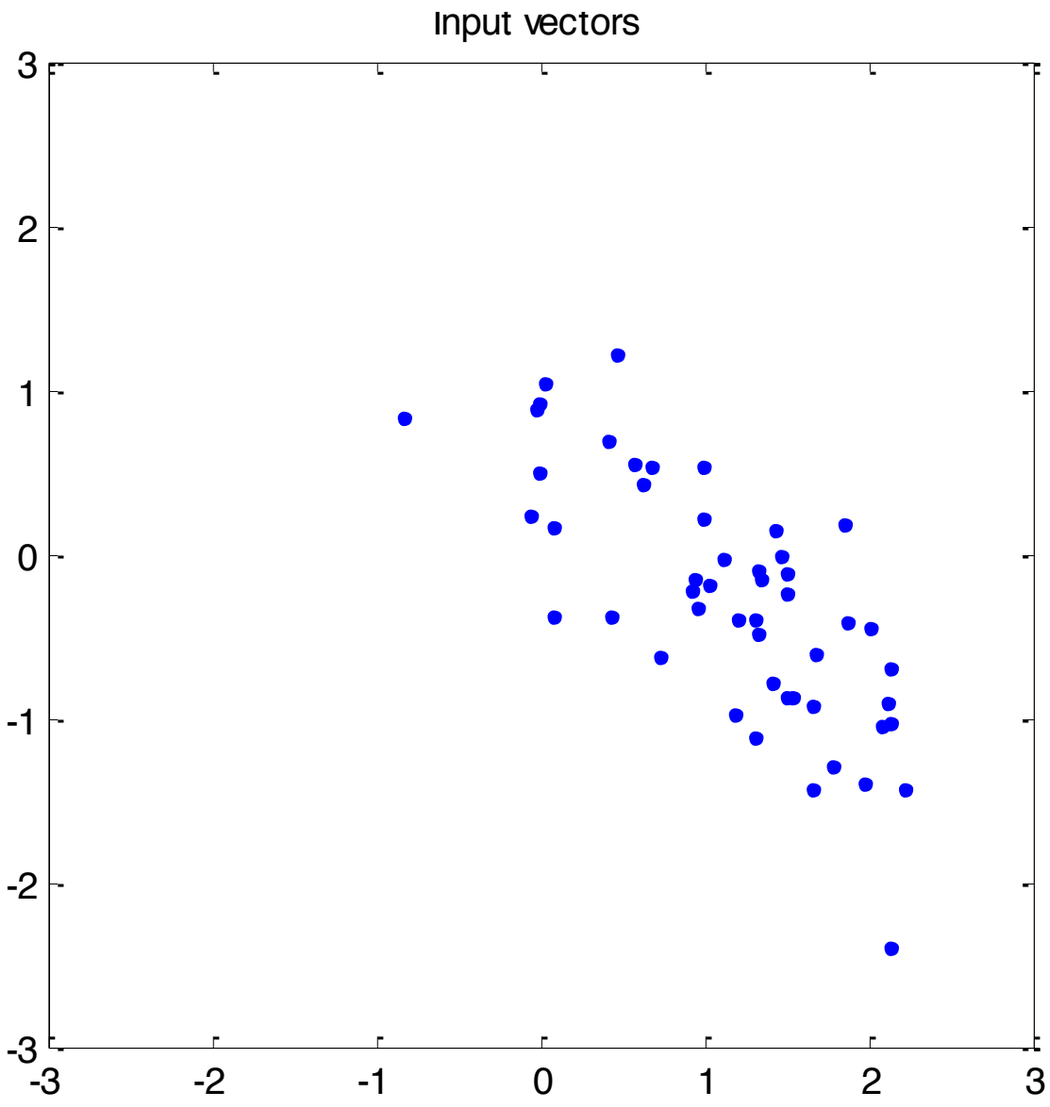
$$\mathbf{x} = \mathbf{A}^{T}\mathbf{y} + \mathbf{m}_{\mathbf{x}}$$

$$\mathbf{x}' = \mathbf{A}_{k}^{T}\mathbf{y} + \mathbf{m}_{\mathbf{x}}$$

# Principal Components

**(Mathematical background: Simple example)**

- Assume we have a population of vectors x
- Example: create a set of input data points (vectors in 2D)




**Plotted 2D vectors**

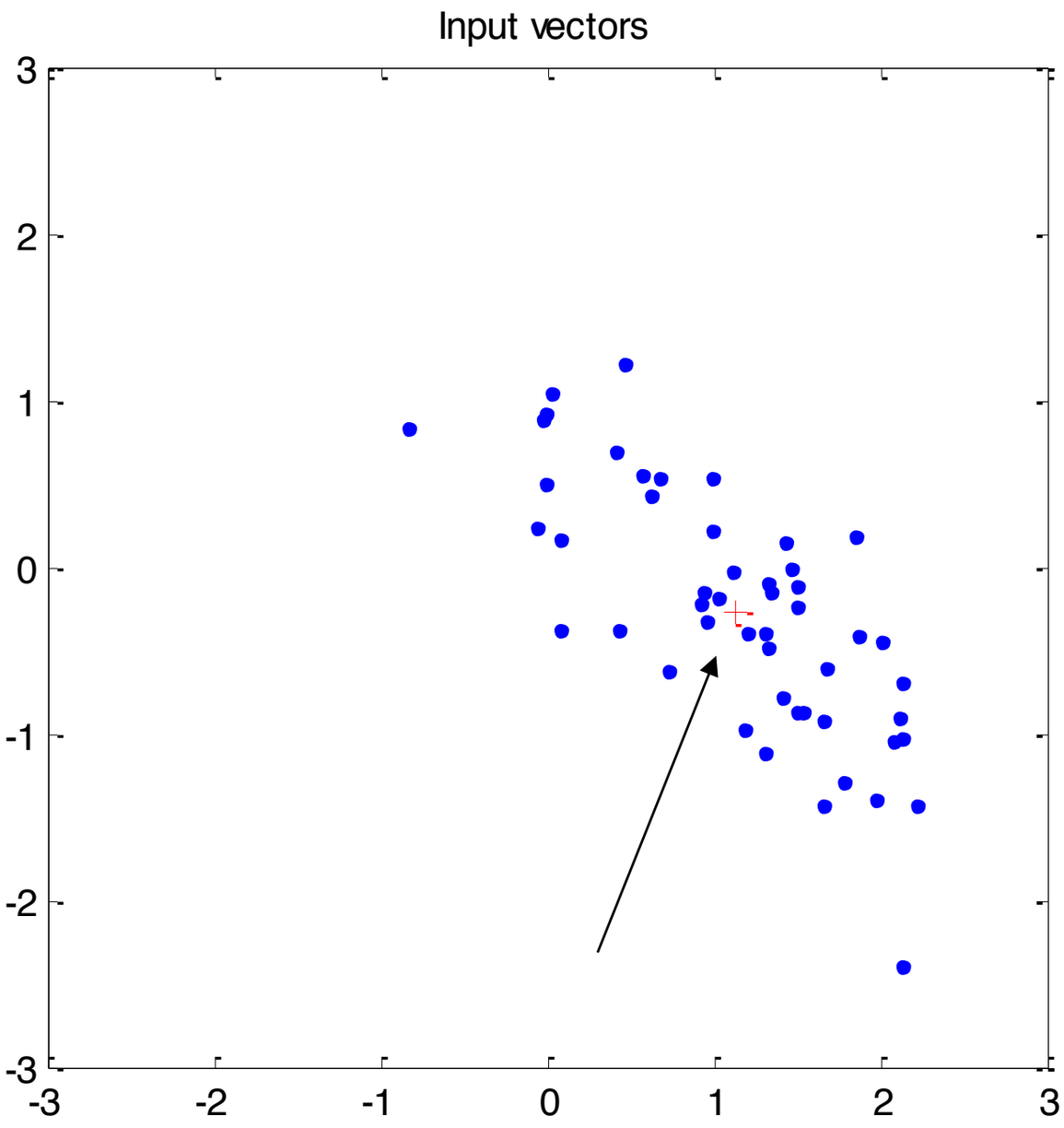
$$\mathbf{x}_{in} = \begin{bmatrix} x \\ y \end{bmatrix}$$

```
clear all
close all

randn('state',0);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Make sample data
N = 50;

xIn(:,1) = randn(N,1) + 1;
xIn(:,2) = 0.5*randn(N,1) + 0.5;

theta = -0.707;
R = [ cos(theta)   sin(theta);
     -sin(theta)   cos(theta)];

xIn = xIn*R;
figure, plot(xIn(:,1), xIn(:,2), '.');
title('Input vectors');
axis equal
axis([-3.0 3.0 -3.0 3.0]);
```

**Generated 2D vectors**

# Example (continued)

- The mean is $\mathbf{m}_x = E[\mathbf{x}]$

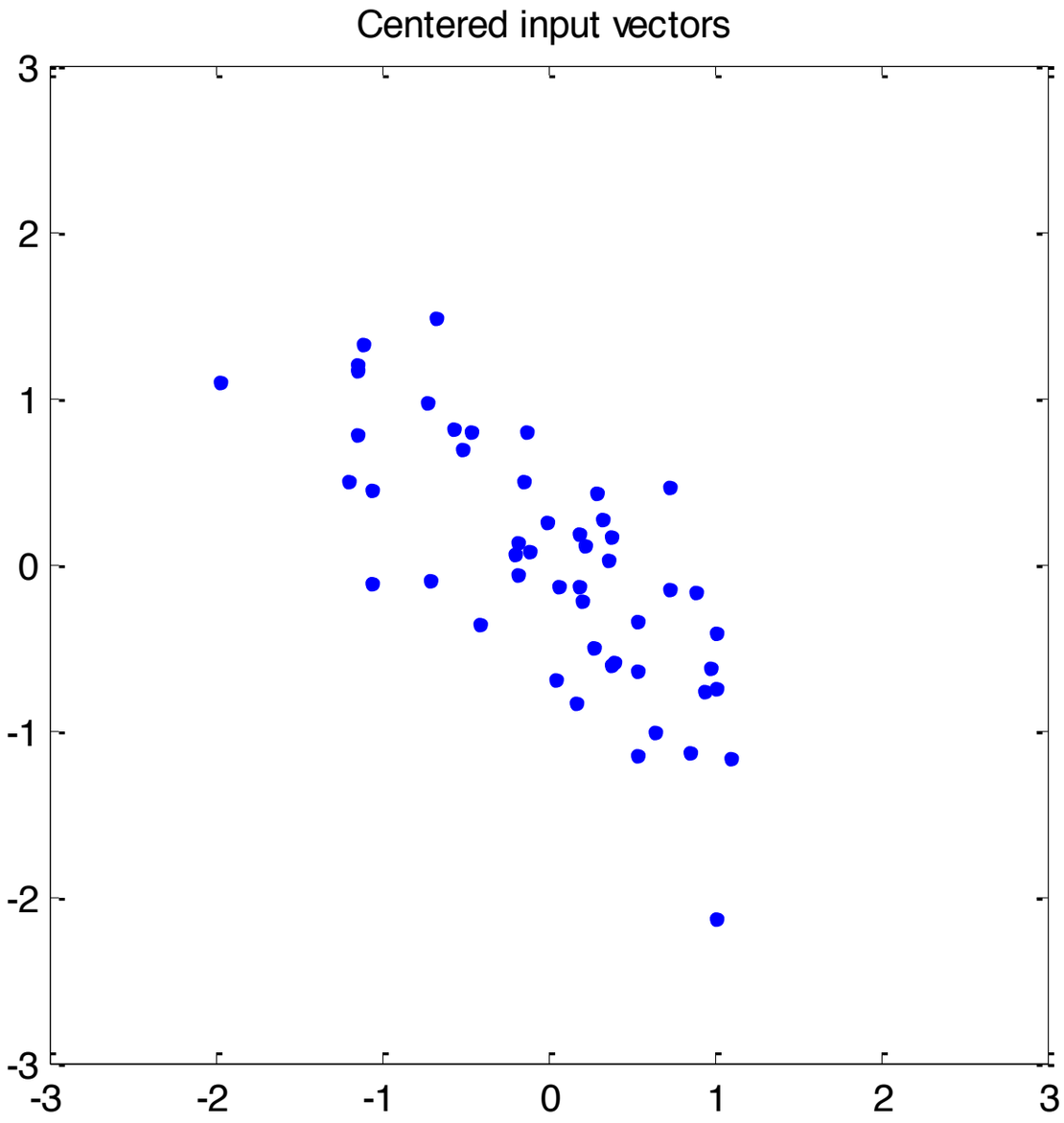



$$\mathbf{m}_x = E\{\mathbf{x}_{in}\} = \frac{1}{K}\sum_{k=1}^{K}\mathbf{x}_{in(k)}$$

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
ux = mean(xIn);        % mean of input vectors
hold on
plot( ux(1), ux(2), '+r');
hold off
pause
```

# Example (continued)

- The covariance matrix is $\mathbf{C}_x = E[\,(\mathbf{x}-\mathbf{m}_x)(\mathbf{x}-\mathbf{m}_x)^T\,]$
  - To compute covariance we first subtract off the mean of data points




$$\mathbf{x} = \mathbf{x}_{in} - \mathbf{m}_x$$

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
x = xIn - repmat(ux,N,1);     % subtract off mean

figure, plot(x(:,1), x(:,2), '.');
title('Centered input vectors');
axis equal
axis([-3.0 3.0 -3.0 3.0]);
```

# Example (continued)

- Find covariance matrix

$$\mathbf{C}_{\mathbf{x}} = E\left[(\mathbf{x}-\mathbf{m}_{\mathbf{x}})(\mathbf{x}-\mathbf{m}_{\mathbf{x}})^T\right]$$
$$= \frac{1}{K}\sum_{k=1}^{K}\mathbf{x}_k\mathbf{x}_k^T - \mathbf{m}_{\mathbf{x}}\mathbf{m}_x^T$$

$$\mathbf{C}_{\mathbf{x}} = \begin{bmatrix} \sigma_x^2 & \sigma_{xy}^2 \\ \sigma_{xy}^2 & \sigma_x^2 \end{bmatrix}$$

```
Covariance of input
    0.5353   -0.4118
   -0.4118    0.5600
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Covariance of input vectors
Cx = cov(x);
disp('Covariance of input');
disp(Cx);
```

- The covariance matrix is real and symmetric
- If dimensions $x_1$,$x_2$ are uncorrelated, their covariance (the off diagonal terms) are zero

# Principal Components

- Consider the eigenvectors and eigenvalues of $\mathbf{C_x}$

$$\mathbf{C_x e}_i = \lambda_i \mathbf{e}_i$$

  - $\mathbf{e}_i$ are the eigenvectors
  - $\lambda_i$ are the corresponding eigenvalues

- By definition, eigenvectors are orthonormal
  - The magnitude (or length) of each eigenvector equals 1
  - The dot product of any pair of vectors is 0 (i.e., they are perpendicular to each other)

$$|\mathbf{e}_i| = 1 \qquad \mathbf{e}_i \cdot \mathbf{e}_j = 0, \quad \text{if } i \neq j$$

- We sort the eigenvectors and eigenvalues in descending order

$$\lambda_1 \geq \lambda_2 \geq \lambda_3 \geq \ldots \geq \lambda_N$$

- The eigenvectors are called "principal components"

# Principal Components

- Let **A** be the matrix whose rows are the eigenvectors of $\mathbf{C}_x$
- We can use **A** as a transformation matrix that maps **x** into **y**

  $\mathbf{y} = \mathbf{A}(\mathbf{x} - \mathbf{m}_x)$

  – This is called the Hotelling transform, or principal components transform

$$\mathbf{A} = \begin{bmatrix} \mathbf{e}_1^T \\ \mathbf{e}_2^T \\ \vdots \\ \mathbf{e}_N^T \end{bmatrix}$$

- The covariance matrix of **y** is

$$\mathbf{C}_\mathbf{y} = E\left[(\mathbf{y} - \mathbf{m}_\mathbf{y})(\mathbf{y} - \mathbf{m}_\mathbf{y})^T\right]$$

- But $\mathbf{m}_\mathbf{y} = 0$ because

$$E[\mathbf{y}] = E[\mathbf{A}(\mathbf{x} - \mathbf{m}_x)] = \mathbf{A}(E[\mathbf{x}] - E[\mathbf{m}_x]) = 0$$

**Equal: def & sub. of mean**

- So

$$\begin{aligned}\mathbf{C}_\mathbf{y} &= E\left[\mathbf{y}\mathbf{y}^T\right] = E\left[\{\mathbf{A}(\mathbf{x} - \mathbf{m}_x)\}\{\mathbf{A}(\mathbf{x} - \mathbf{m}_x)\}^T\right] \\ &= E\left[\mathbf{A}(\mathbf{x} - \mathbf{m}_x)(\mathbf{x} - \mathbf{m}_x)^T\mathbf{A}^T\right] \\ &= \mathbf{A}E\left[\mathbf{x}\mathbf{x}^T\right]\mathbf{A}^T = \mathbf{A}\mathbf{C}_\mathbf{x}\mathbf{A}^T\end{aligned}$$

# Principal Components

- We have

$$\mathbf{C}_\mathbf{y} = \mathbf{A}\mathbf{C}_\mathbf{x}\mathbf{A}^T$$

- where

$$\mathbf{A} = \begin{bmatrix} \mathbf{e}_1^T \\ \mathbf{e}_2^T \\ \vdots \\ \mathbf{e}_N^T \end{bmatrix}, \quad \text{and } \mathbf{A}^T = \begin{bmatrix} \mathbf{e}_1 & \mathbf{e}_2 & \cdots & \mathbf{e}_N \end{bmatrix}$$

- Now, since the columns of $\mathbf{A}^T$ are the eigenvectors of $\mathbf{C}_\mathbf{x}$, then multiplying $\mathbf{A}^T$ by $\mathbf{C}_\mathbf{x}$ just gives us the eigenvectors back (times the eigenvalues)

$$\mathbf{C}_\mathbf{x}\mathbf{A}^T = \mathbf{C}_\mathbf{x}\begin{bmatrix} \mathbf{e}_1 & \mathbf{e}_2 & \cdots & \mathbf{e}_N \end{bmatrix} = \begin{bmatrix} \lambda_1\mathbf{e}_1 & \lambda_2\mathbf{e}_2 & \cdots & \lambda_N\mathbf{e}_N \end{bmatrix}$$

**Definition**

- So

$$\mathbf{C}_\mathbf{y} = \mathbf{A}\mathbf{C}_\mathbf{x}\mathbf{A}^T = \begin{bmatrix} \mathbf{e}_1^T \\ \mathbf{e}_2^T \\ \vdots \\ \mathbf{e}_N^T \end{bmatrix} \begin{bmatrix} \lambda_1\mathbf{e}_1 & \lambda_2\mathbf{e}_2 & \cdots & \lambda_N\mathbf{e}_N \end{bmatrix} = \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & & 0 \\ \vdots & & \ddots & \\ 0 & 0 & & \lambda_N \end{pmatrix}$$

**Orthonormal**

# Principal Components

- $\mathbf{C_y}$ is a diagonal matrix

$$\mathbf{C}_y = \begin{pmatrix} \lambda_1 & & & 0 \\ & \lambda_2 & & \\ & & \ddots & \\ 0 & & & \lambda_n \end{pmatrix}$$

*So the dimensions of the* **y***'s are uncorrelated*

  - where the $\lambda_i$ are the eigenvalues of $\mathbf{C_x}$
  - $\mathbf{C_x}$ and $\mathbf{C_y}$ have the same eigenvalues

- Again, the eigenvectors of $\mathbf{C_x}$ (the rows of the matrix **A**) are called "principal components"
  - They are orthogonal and orthonormal
  - As a result, $\mathbf{A}^{-1} = \mathbf{A}^T$

- We can express any vector x as a linear combination of the principal components

$$\mathbf{y} = \mathbf{A}(\mathbf{x} - \mathbf{m}_\mathbf{x}) \longrightarrow \mathbf{A}^T\mathbf{y} = \mathbf{A}^T\mathbf{A}(\mathbf{x} - \mathbf{m}_\mathbf{x}) \longrightarrow \mathbf{x} = \mathbf{A}^T\mathbf{y} + \mathbf{m}_\mathbf{x}$$

**Given y, reconstruct x (lin. comb.)**

# Principal Components Example

- Find eigenvalues and eigenvectors

$$\mathbf{C_x e} = \lambda \mathbf{e}$$

```
Eigenvector e1:
   -0.7176
   -0.6964
Eigenvector e2:
   -0.6964
    0.7176
Eigenvalue d1:
    0.1357
Eigenvalue d2:
    0.9597
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Get eigenvalues and eigenvectors of Cx
% Produces V,D such that Cx*V = V*D.
% So the eigenvectors are the columns of V.
[V,D] = eig(Cx);
e1 = V(:,1);
disp('Eigenvector e1:'), disp(e1);
e2= V(:,2);
disp('Eigenvector e2:'), disp(e2);
d1 = D(1,1);
disp('Eigenvalue d1:'), disp(d1);
d2 = D(2,2);
disp('Eigenvalue d2:'), disp(d2);
```

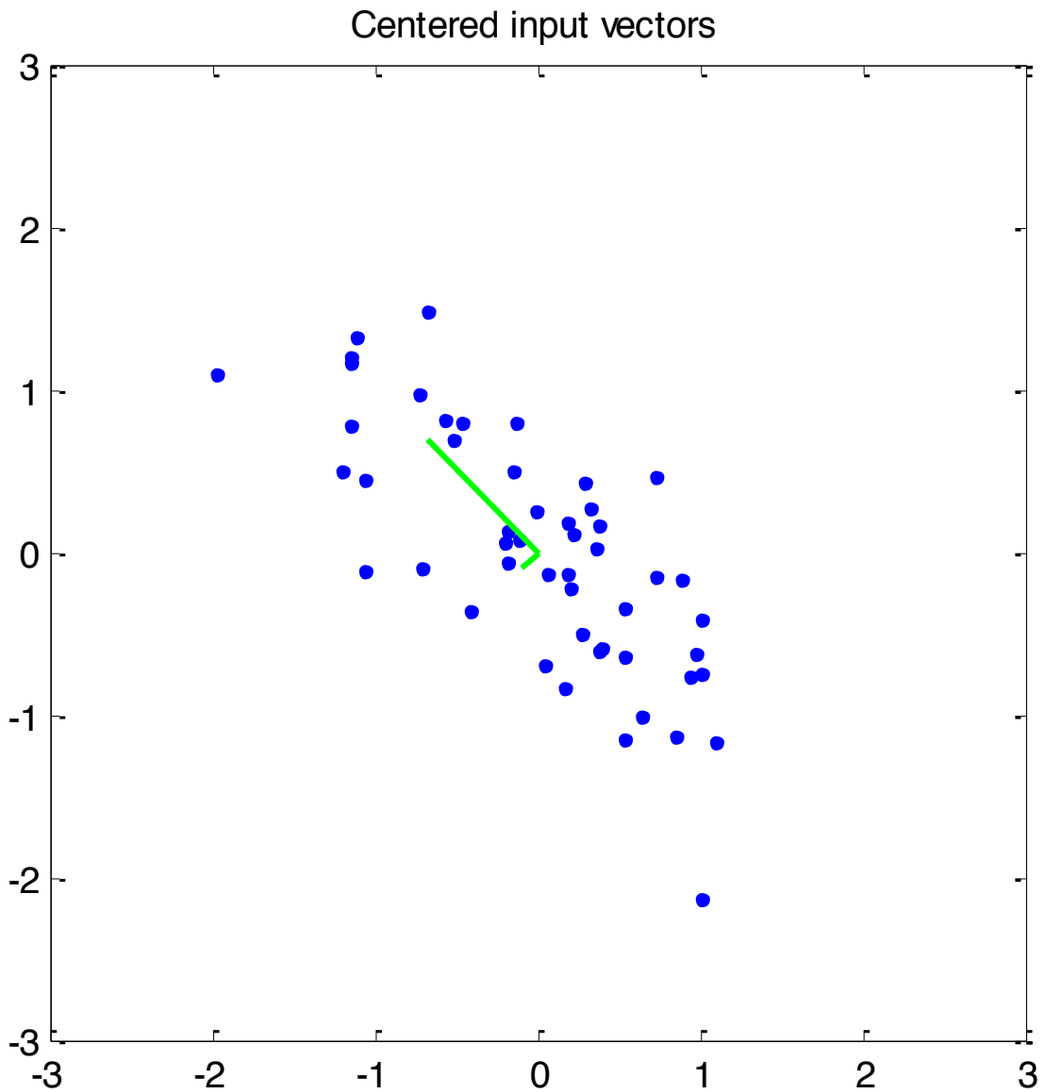
```
% Verify eigenvalues and eigenvectors
disp('Cx*e1 = '), disp(Cx*e1);
disp('d1*e1 = '), disp(d1*e1);

disp('Cx*e2 = '), disp(Cx*e2);
disp('d2*e2 = '), disp(d2*e2);
```

- Verify

```
Cx*e1 =          Cx*e2 =
   -0.0974          -0.6684
   -0.0945           0.6887
d1*e1 =          d2*e2 =
   -0.0974          -0.6684
   -0.0945           0.6887
```

# Principal Components Example

- Draw eigenvectors (principal components)



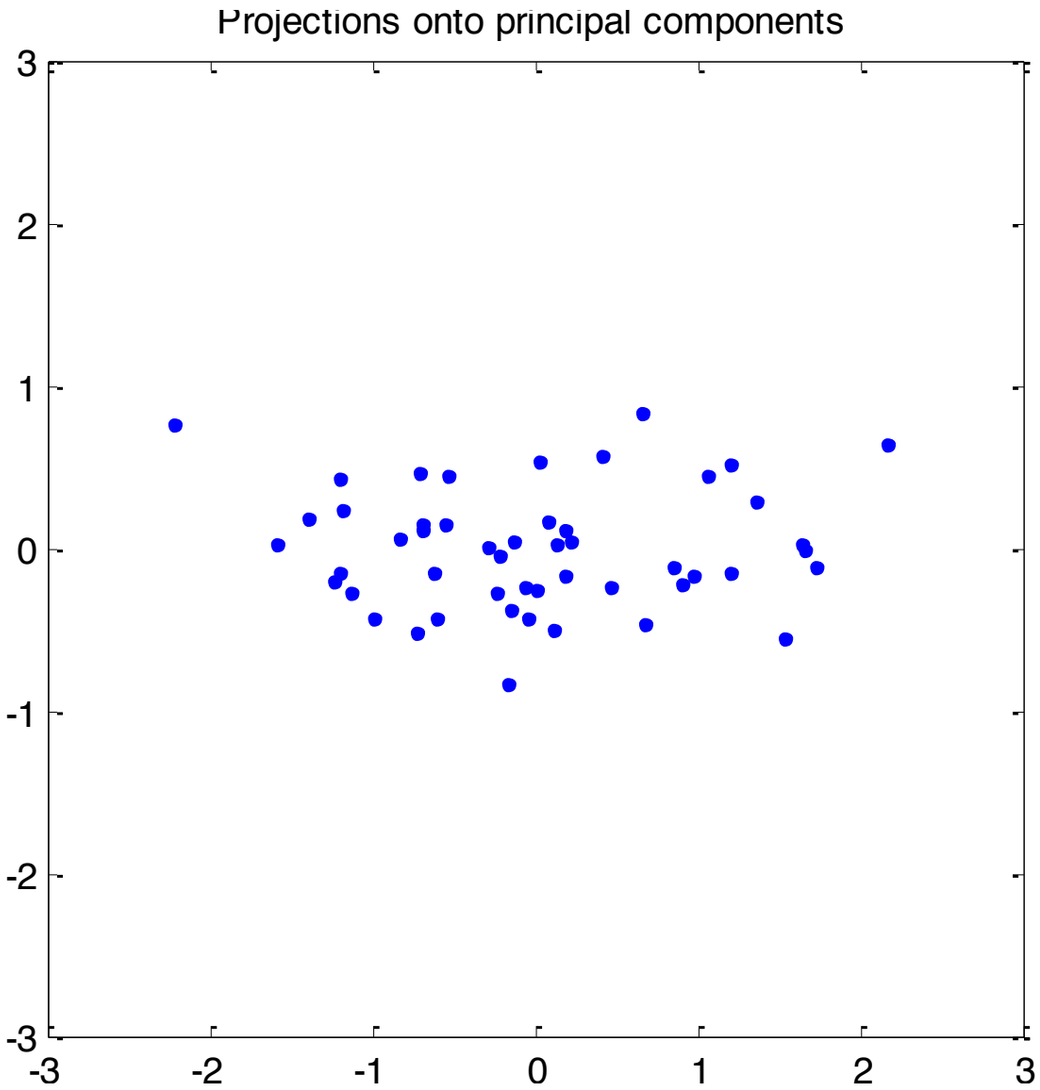

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Draw eigenvectors
hold on
line([0 d1*e1(1)], [0 d1*e1(2)], 'Color', …
                'g', 'LineWidth', 2);
line([0 d2*e2(1)], [0 d2*e2(2)], 'Color', …
                'g', 'LineWidth', 2);
```

# Principal Components Example

- Project input data onto principal components

$$\mathbf{y} = \mathbf{A}\left(\mathbf{x}_{in} - \mathbf{m}_{x}\right)$$

  - where A is the matrix whose rows are eigenvectors of Cx




```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Project input data onto principal components
y = [e2'; e1']*x';

figure, plot(y(1,:),y(2,:), '.');
title('Projections onto principal components');
axis equal
axis([-3.0 3.0 -3.0 3.0]);
```

# Principal Components

- We can reconstruct **x** from the **y**'s

  $\mathbf{x} = \mathbf{A}^T\mathbf{y} + \mathbf{m}_x$

- Instead of using all the eigenvectors of $\mathbf{C}_x$, we can take only the eigenvectors corresponding to the *k* largest eigenvalues => $\mathbf{A}_k$

- We can reconstruct an approximation of **x** from only a few principal components (PC's):

  $\mathbf{x'} = \mathbf{A}_k^T\mathbf{y} + \mathbf{m}_x$

- So to represent the input data approximately, we just need to use *k* principal components and the value of the projections onto those components
- The mean squared error is

$$e_{ms} = E\left\{\left(\mathbf{x} - \hat{\mathbf{x}}\right)^2\right\} = \sum_{j=1}^{n}\lambda_j - \sum_{j=1}^{k}\lambda_j$$

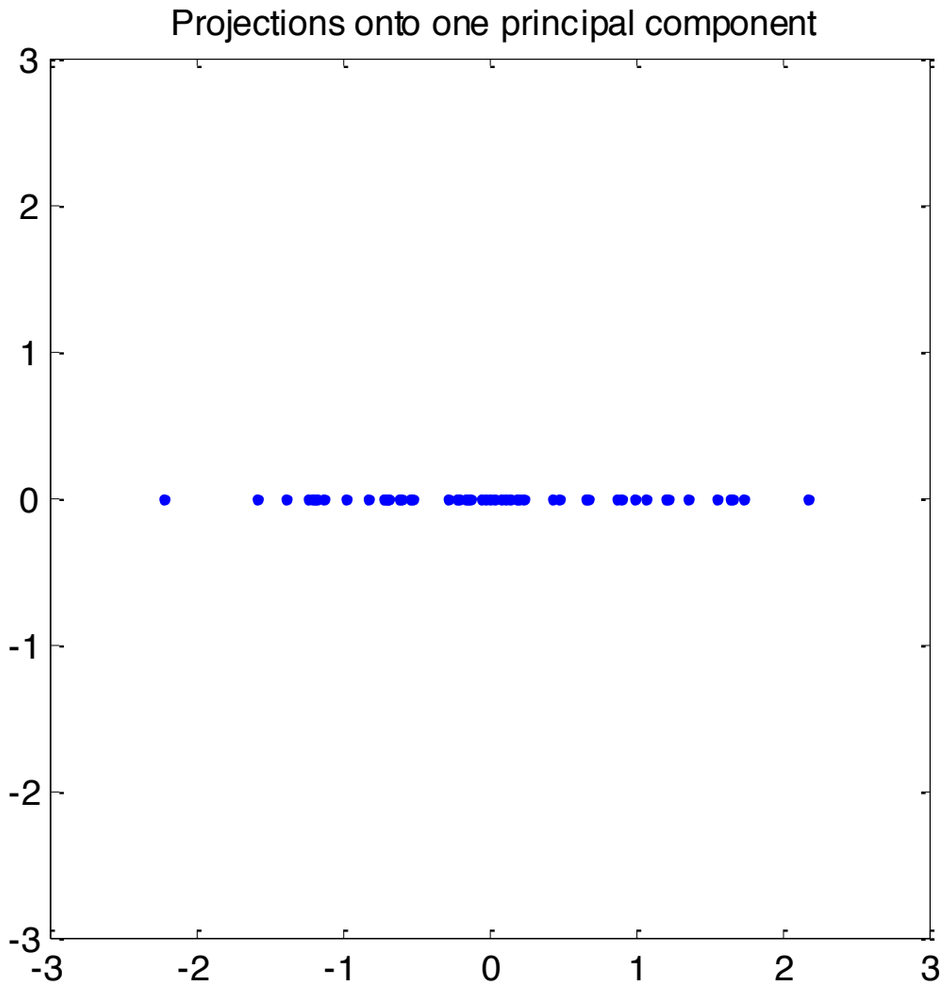
$$= \sum_{j=k+1}^{n}\lambda_j$$

# Principal Components Example

- Project input data onto only k principal components

$$\mathbf{y} = \mathbf{A}_k \left( \mathbf{x} - \mathbf{m}_\mathbf{x} \right)$$

  - where $\mathbf{A}_k$ is the matrix whose rows are the first $k$ eigenvectors of $\mathbf{C}_\mathbf{x}$




```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Project input data using only one principal
component
y = [e2']*x';
figure, plot(y(1,:),zeros(1,length(y)), '.');
title('Projections onto one principal component');
axis equal
axis([-3.0 3.0 -3.0 3.0]);
```

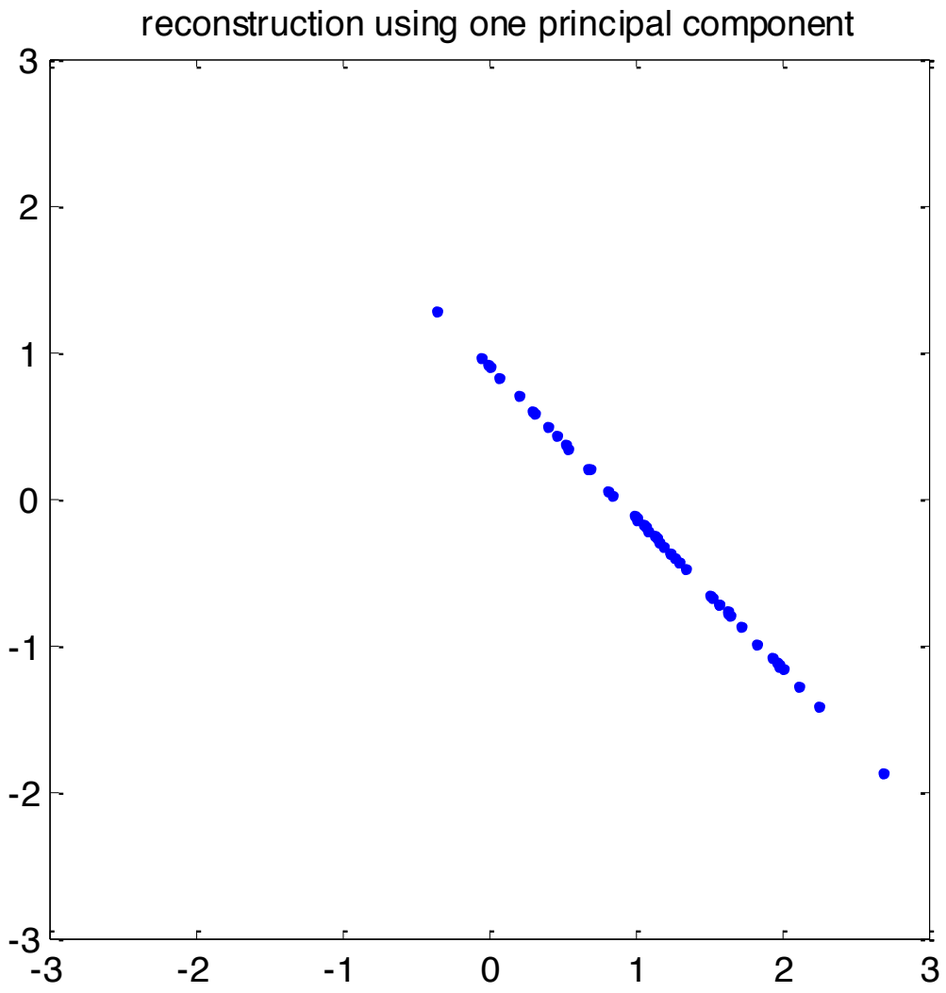
**Simple 2D case using only one of the two eigenvectors (k=1)**

# Principal Components Example

- Reconstruction of input data using only *k* principal components

$$\mathbf{x} = \mathbf{A}_k^T \mathbf{y} + \mathbf{m}_\mathbf{x}$$

  - where $\mathbf{A}_k$ is the matrix whose rows are the first *k* eigenvectors of $\mathbf{C}_\mathbf{x}$




```
% Reconstruct
xx = e2*y + repmat(ux',1,length(y));

figure, plot(xx(1,:),xx(2,:), '.');
title('reconstruction using one principal
component');
axis equal
axis([-3.0 3.0 -3.0 3.0]);
```

# Use of PCA for description

- We can use principal components analysis (PCA) to
  - represent images more concisely
  - help with recognition and matching

- We'll look at two ways to use PCA on images
  - If we have a multidimensional image (e.g., color RGB)
    - We have a collection of vectors, each represents a pixel (e.g., R,G,B values)
    - There is no notion of spatial position ... the set is a "bag of pixels"
    - We can potentially represent each pixel using fewer dimensions
    - We'll call the principal components "eigenpixels"

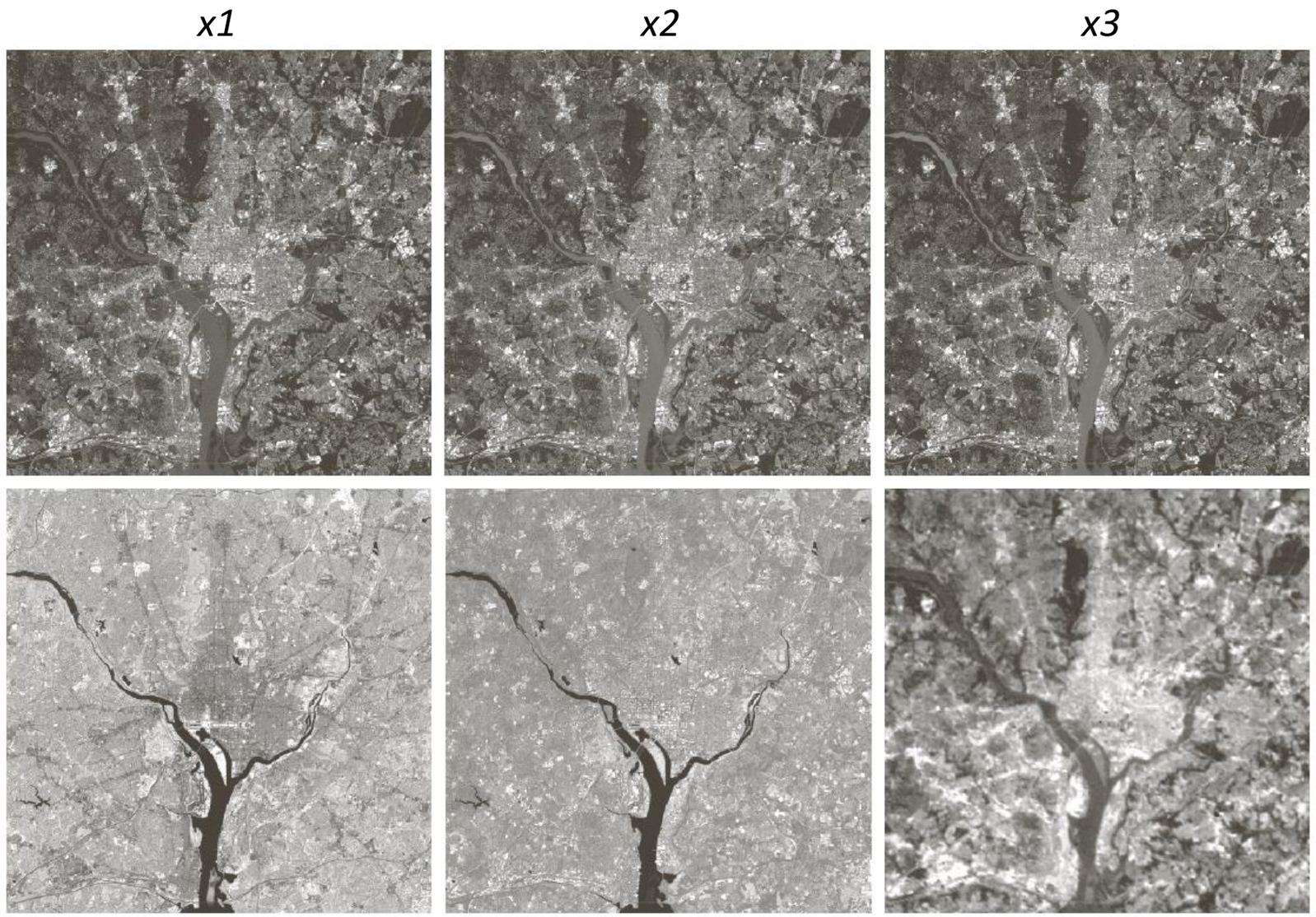
  - If we have a set of images (monochrome)
    - We have a collection of vectors, each represents an entire image
    - We can potentially represent each image using a linear combination of "basis" images
    - The number of basis images can be much smaller than our original collection
    - We'll call the principal components "eigenimages"

One

Bands

# Eigenpixel example

- We have an image where each pixel is composed of 6 values **(bands)** [*x1,x2,x3,x4,x5,x6*]
- But some of these values may be redundant (e.g., middle wavelength infrared may be very similar to long wavelength infrared)
- We may be able to represent each pixel using fewer than 6 values




a b c
d e f

**FIGURE 11.38** Multispectral images in the (a) visible blue, (b) visible green, (c) visible red, (d) near infrared, (e) middle infrared, and (f) thermal infrared bands. (Images courtesy of NASA.)

# Example

- Each pixel is a 6-element vector
- Image size is 564x564 = 318,096
- So we have this many vectors

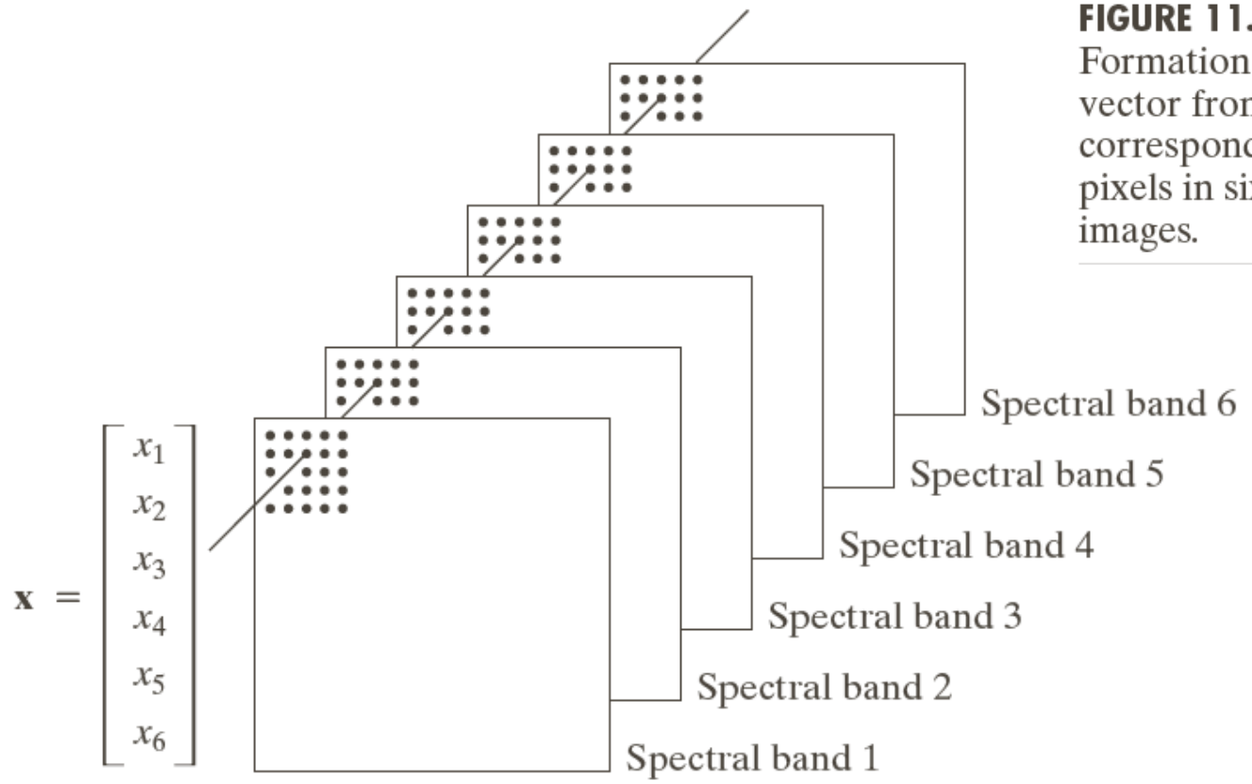



**FIGURE 11.39** Formation of a vector from corresponding pixels in six images.

- Compute mean, covariance, eigenvalues and eigenvectors

**TABLE 11.6** Eigenvalues of the covariance matrices obtained from the images in Fig. 11.38.

| $\lambda_1$ | $\lambda_2$ | $\lambda_3$ | $\lambda_4$ | $\lambda_5$ | $\lambda_6$ |
|---|---|---|---|---|---|
| 10344 | 2966 | 1401 | 203 | 94 | 31 |

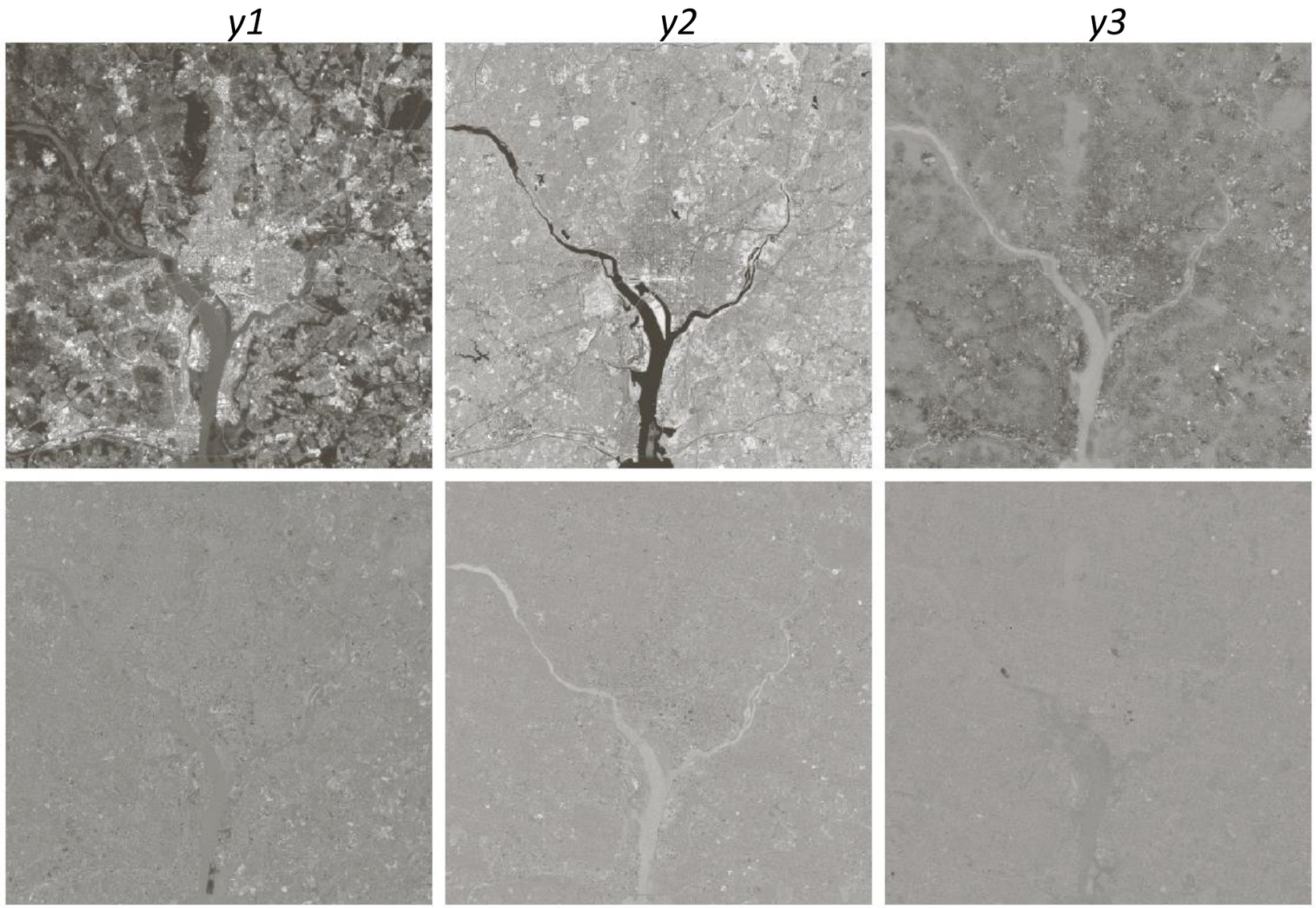
# Coefficients of the **y**-vectors

$$\mathbf{y} = \mathbf{A}(\mathbf{x} - \mathbf{m}_x)$$

- Note that *y1* has the most variation
  - This is to be expected because $\lambda_1$ is the variance of *y1*

$$\mathbf{C}_y = \begin{pmatrix} \lambda_1 & & 0 \\ & \lambda_2 & \\ & & \ddots \\ 0 & & \lambda_n \end{pmatrix}$$

  - and $\lambda_1$ is the largest eigenvalue




a b c
d e f

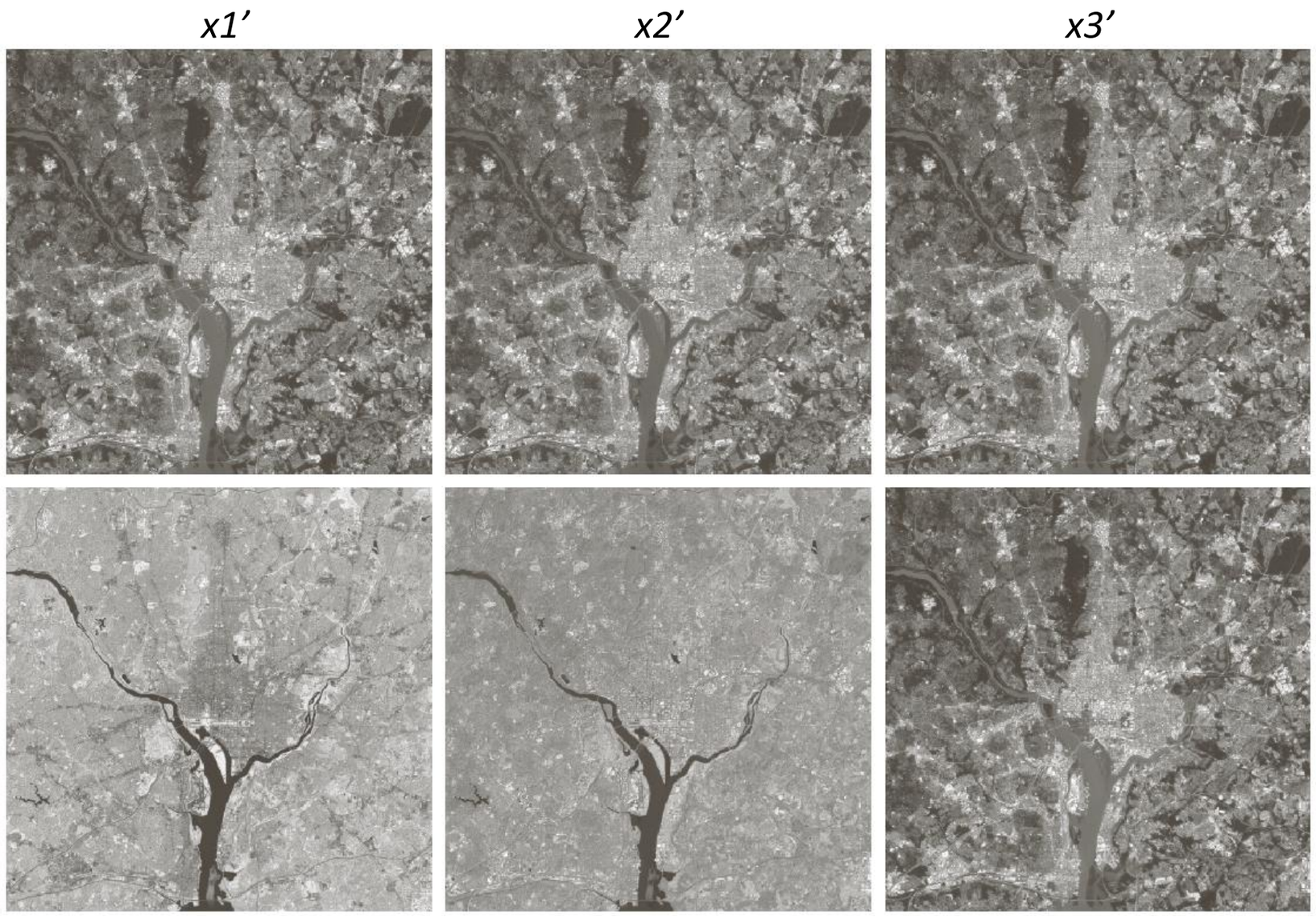
**FIGURE 11.40** The six principal component images obtained from vectors computed using Eq. (11.4-6). Vectors are converted to images by applying Fig. 11.39 in reverse.

# Example

- Reconstruction of all 6 values, using the top 2 principal components

- We only use [*y1,y2*]

- $\mathbf{x' = A_k^T y + m_x}$

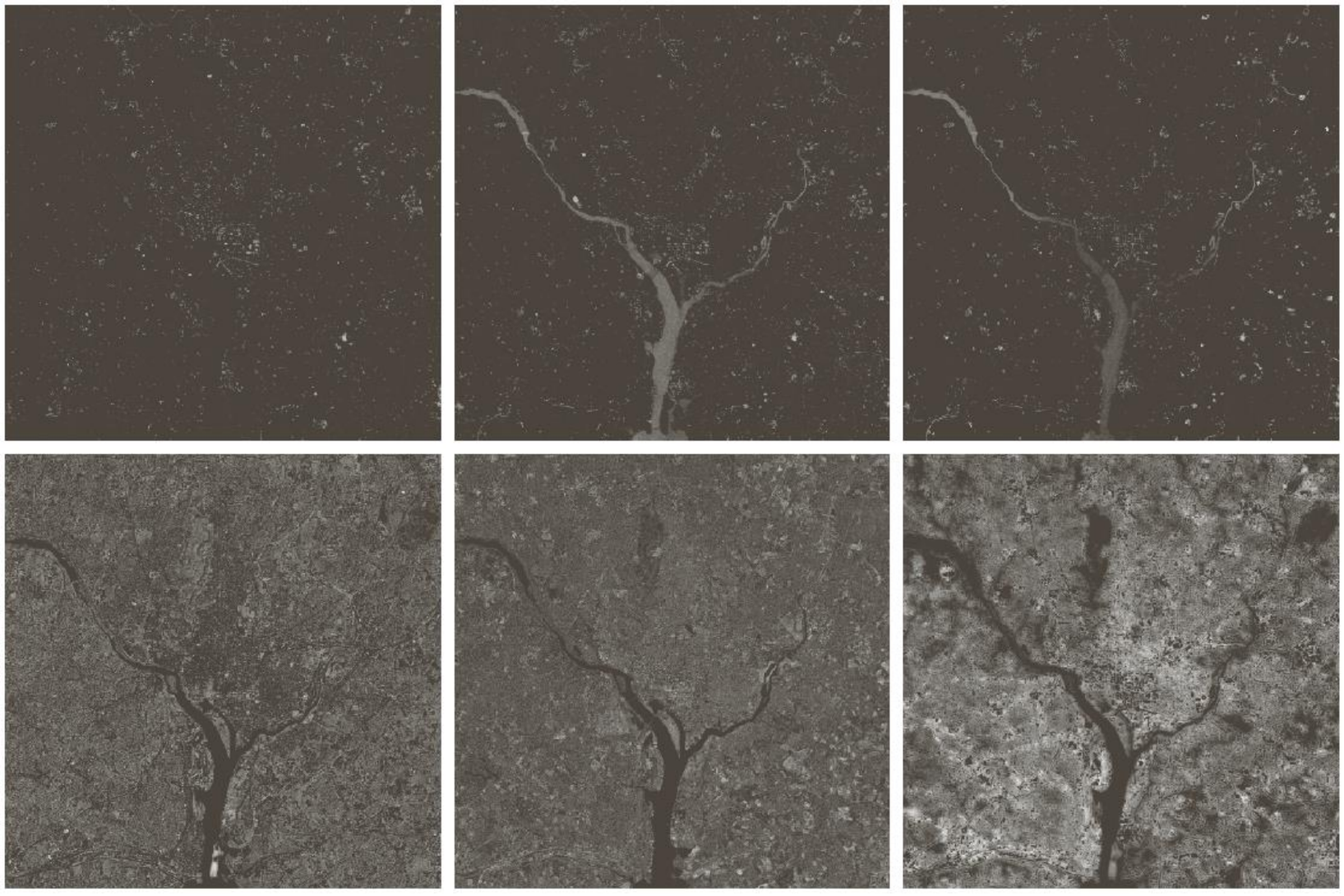



a b c
d e f

**FIGURE 11.41** Multispectral images reconstructed using only the two principal component images corresponding to the two principal component images with the largest eigenvalues (variance). Compare these images with the originals in Fig. 11.38.

# Example

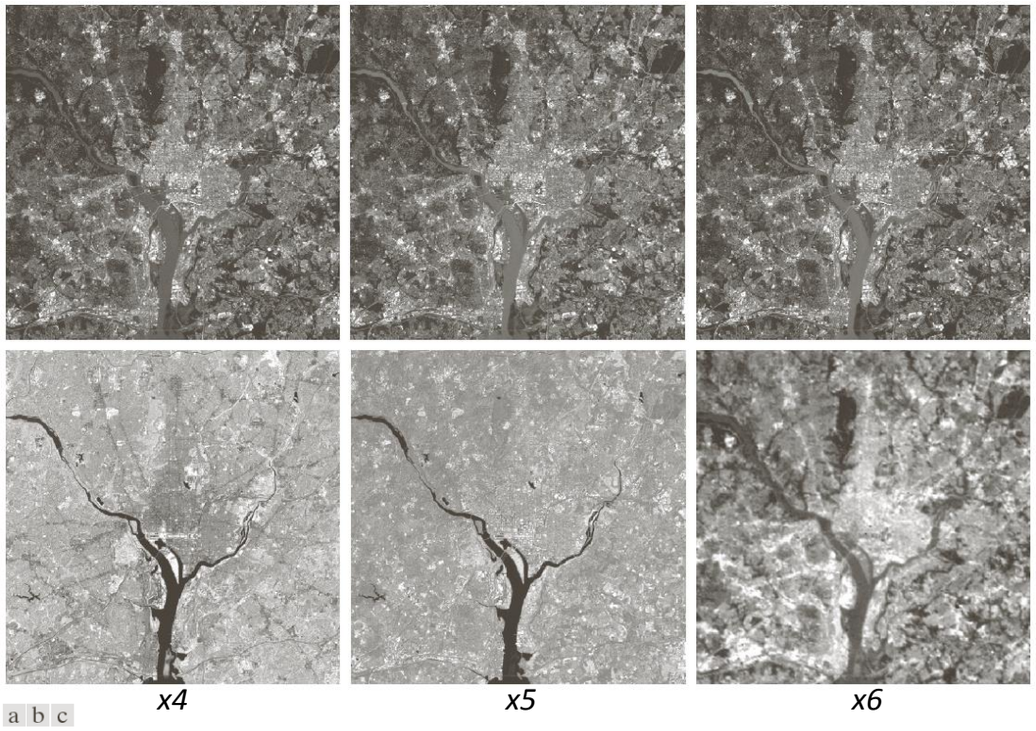
- Error between reconstructed images and original images



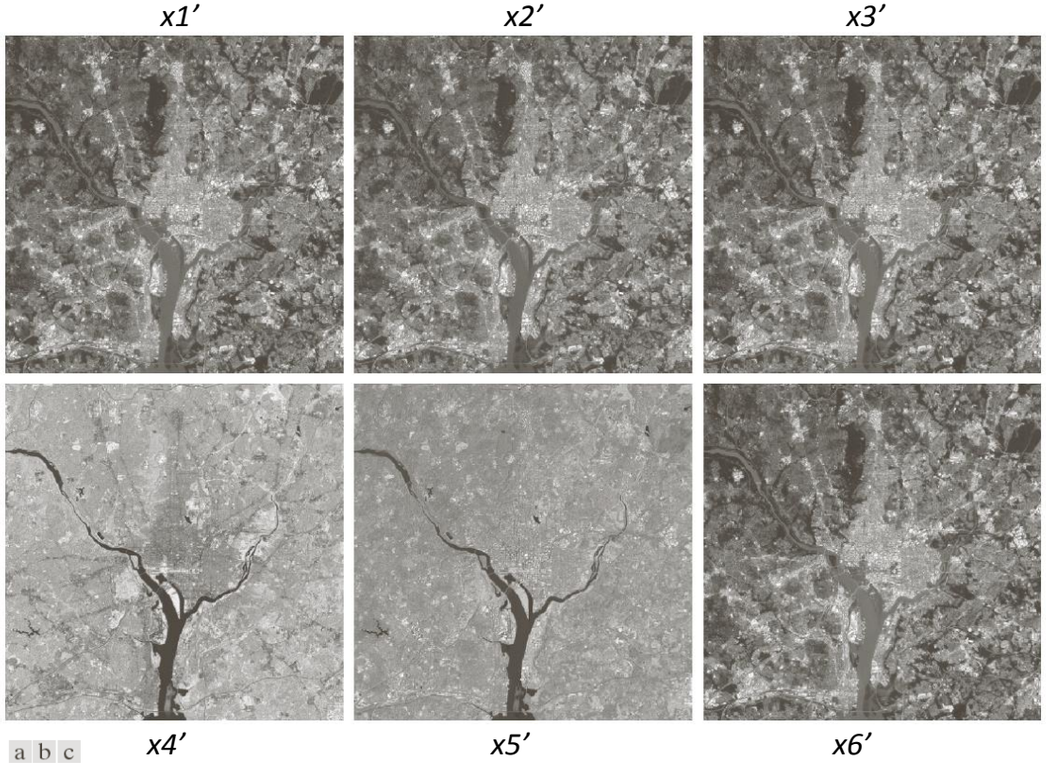
a b c
d e f

**FIGURE 11.42** Differences between the original and reconstructed images. All difference images were enhanced by scaling them to the full [0, 255] range to facilitate visual analysis.

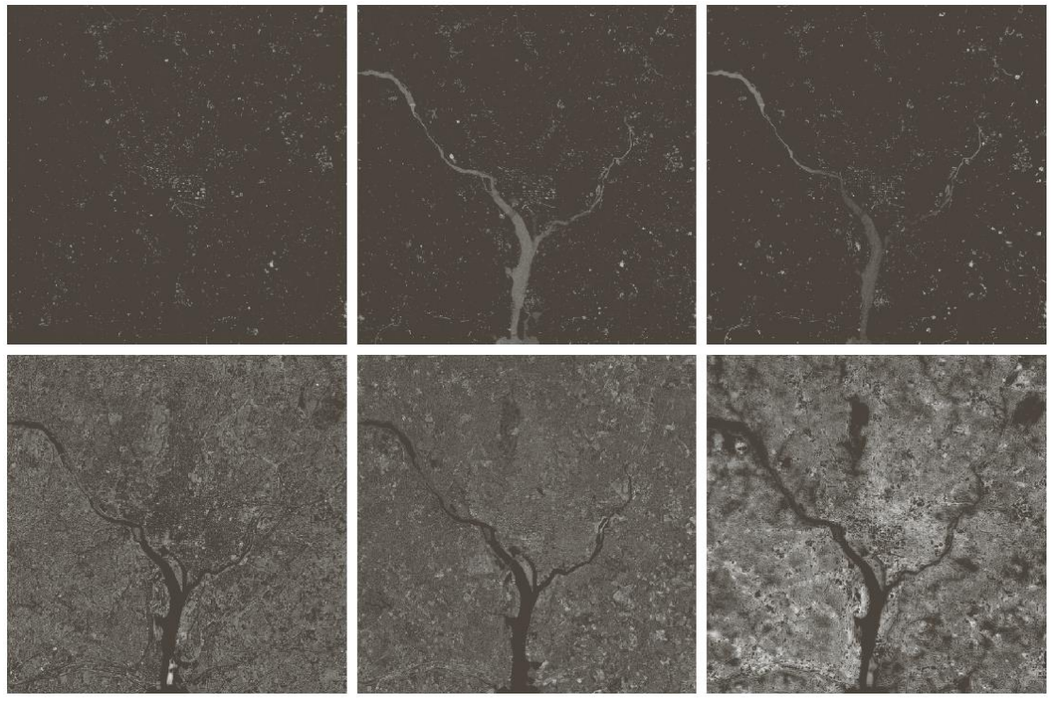
## Original




## Approximation




## Error

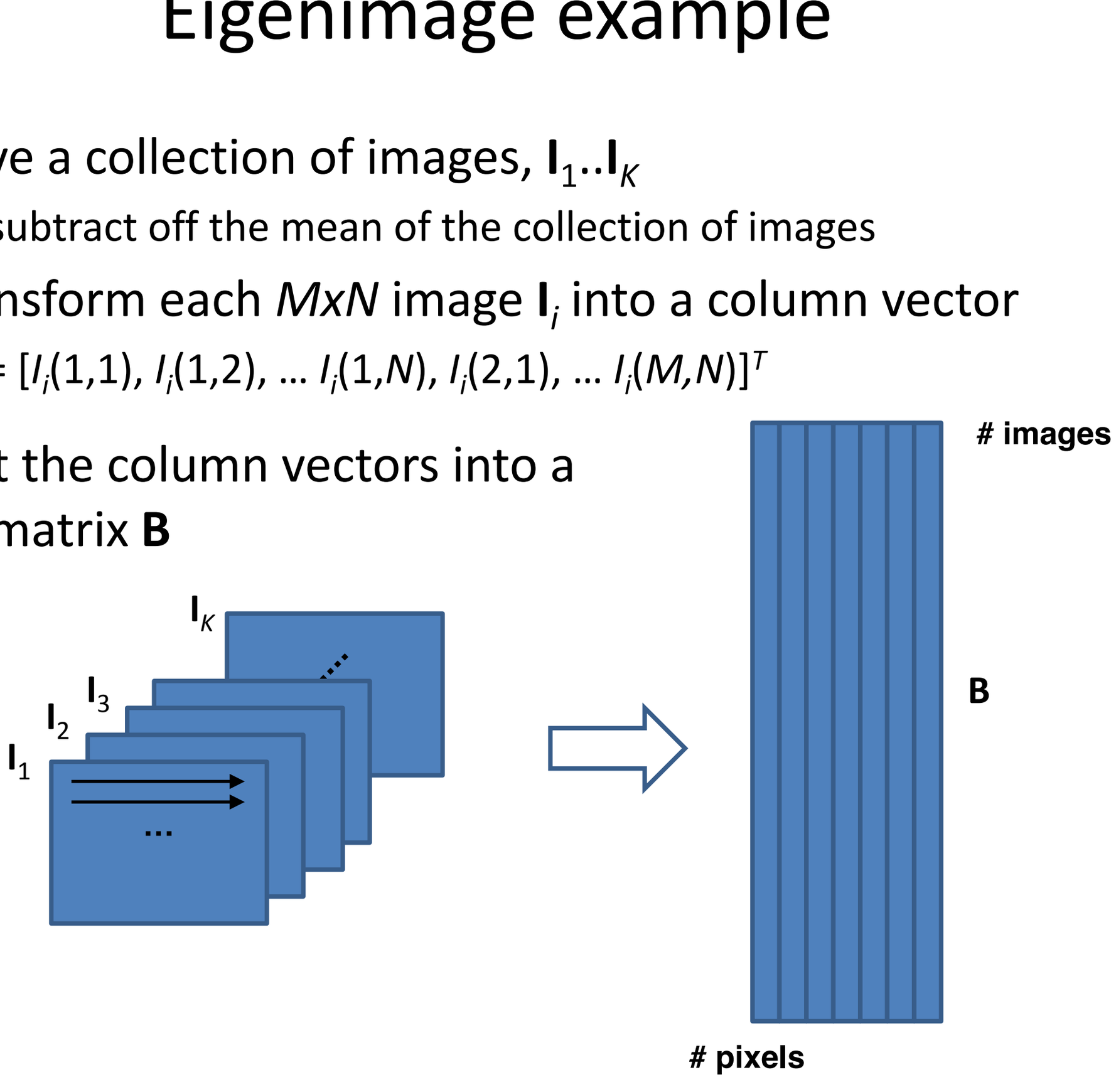


# Eigenimage example

- We have a collection of images, $\mathbf{I}_1..\mathbf{I}_K$
  - We subtract off the mean of the collection of images
- We transform each *MxN* image $\mathbf{I}_i$ into a column vector

$$\mathbf{x}_i = [I_i(1,1), I_i(1,2), \ldots I_i(1,N), I_i(2,1), \ldots I_i(M,N)]^T$$

- We put the column vectors into a single matrix **B**



# Eigenimage example

- Covariance of our vectors

$$\mathbf{C}_\mathbf{x} = E\left[(\mathbf{x} - \mathbf{m}_\mathbf{x})(\mathbf{x} - \mathbf{m}_\mathbf{x})^T\right]$$
$$= \frac{1}{K}\sum_{k=1}^{K}\mathbf{x}_k\mathbf{x}_k^T - \mathbf{m}_\mathbf{x}\mathbf{m}_x^T$$

  - since we have subtracted off the mean from the input vectors, $\mathbf{m}_\mathbf{x} = 0$
  - So

$$\mathbf{C}_\mathbf{x} = \frac{1}{K}\sum_{k=1}^{K}\mathbf{x}_k\mathbf{x}_k^T = \frac{1}{K}\mathbf{B}\mathbf{B}^T$$

- As before, let **A** be the matrix whose rows are the eigenvectors of $\mathbf{C}_\mathbf{x}$
  - The eigenvectors (principal components) represent “basis” images
  - We’ll call them “eigenimages”

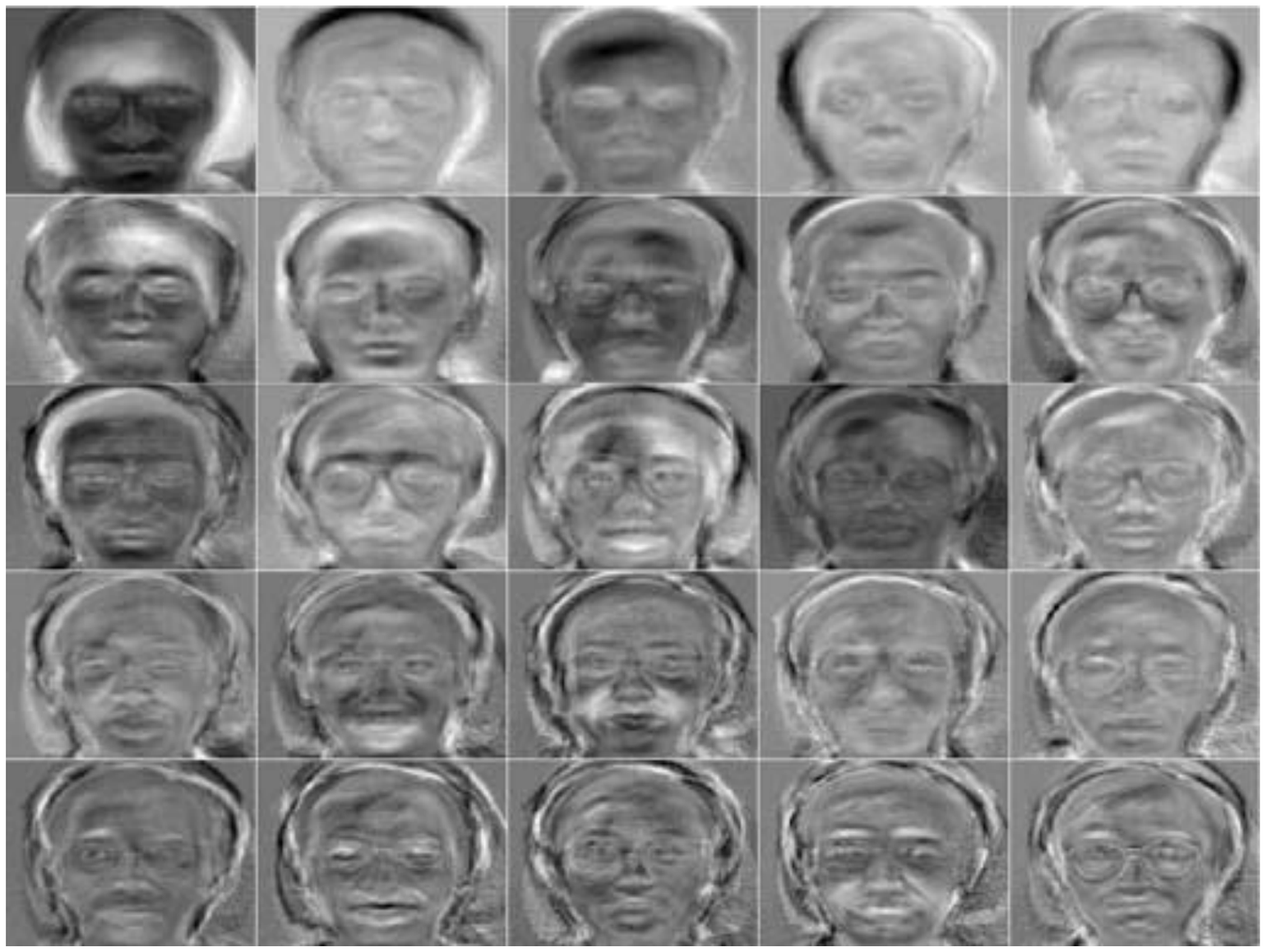
# Application - Face images

Input images (Yale database)

**Multiple persons Multiple views for each person**

*from: http://www.cs.princeton.edu/~cdecoro/eigenfaces/*

# Eigenfaces



- 10 PC's capture 36% of the variance
- 25 PC's capture 56% of the variance

**Represent any face as the mean + a lin. comb. of these eigenfaces**

# Reconstruction with a small number of Principal Components

**Mean (average) face**

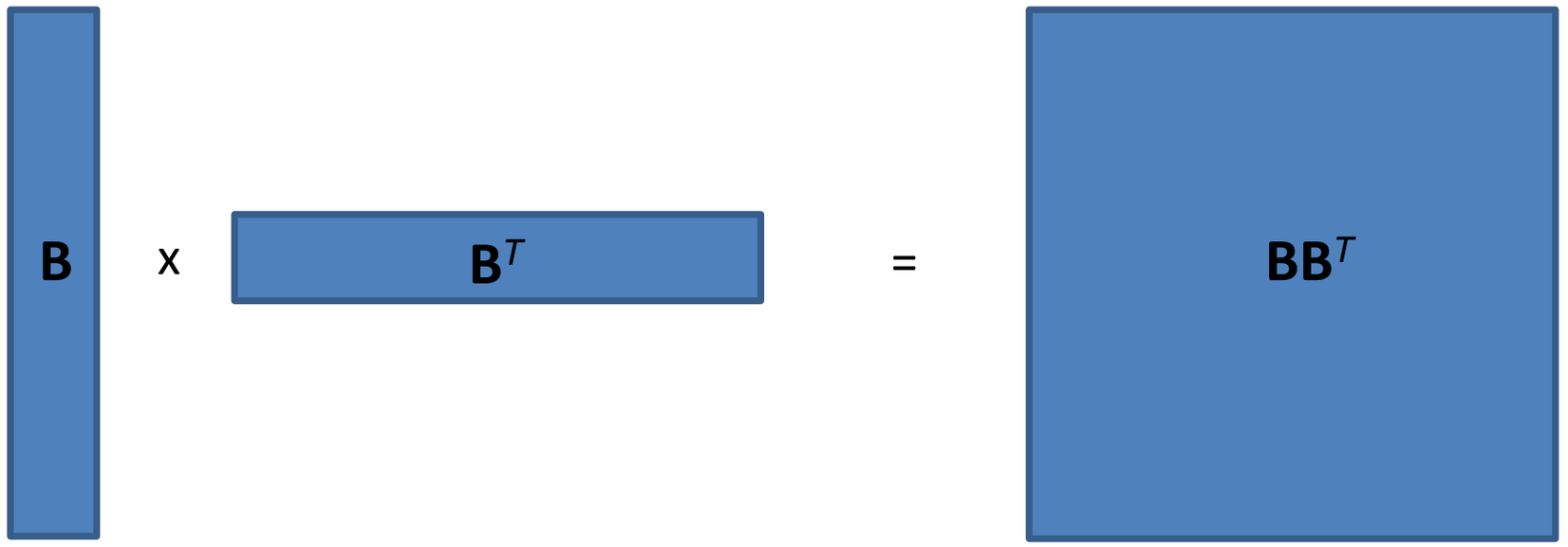
**Target face**

Each new image from left to right corresponds to using 1 additional PC for reconstruction

Each new image from left to right corresponds to using 8 additional PC for reconstruction

# Note on computational expense

- Size of **B**
  - Number of rows in **B** = the number of pixels in the image
  - Number of columns in **B** = the number of images in our collection
  - Typically #rows >> #columns
- So $\mathbf{B}\mathbf{B}^T$ is a large matrix
  - It is square (size = number of pixels in the image squared)



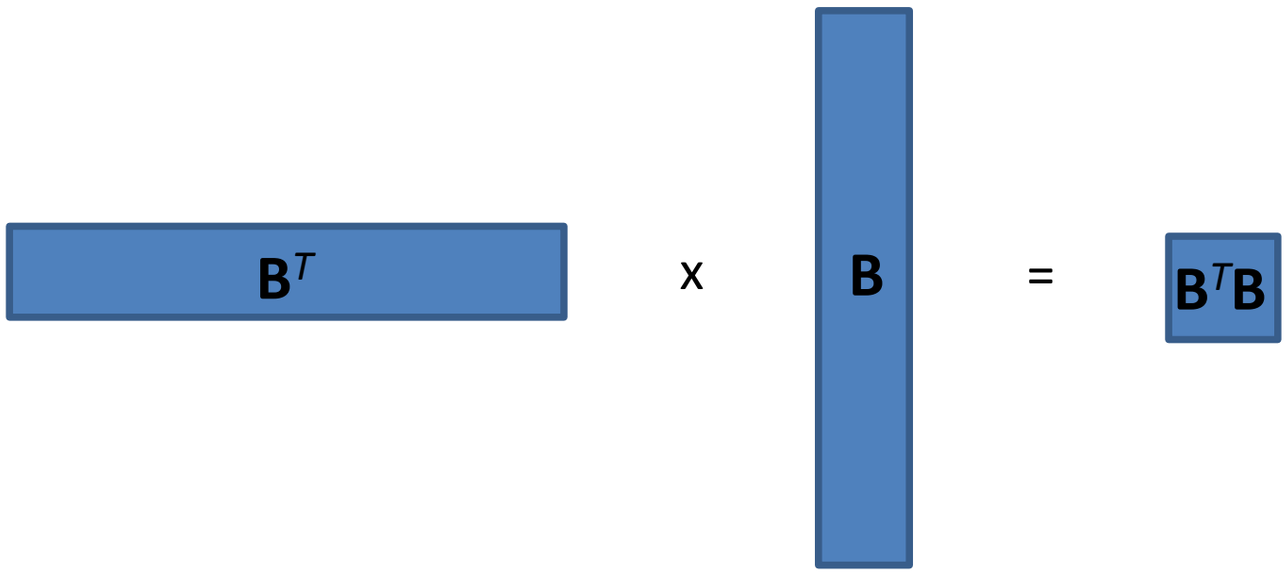

- We want to compute the eigenvectors of $\mathbf{B}\mathbf{B}^T$; ie find the vectors $\mathbf{u}_i$ such that

  $(\mathbf{B}\mathbf{B}^T)\mathbf{u}_i = \lambda_i \mathbf{u}_i$

  - It can be expensive to compute eigenvectors of a large matrix

# Note on computational expense (continued)

- Look at $\mathbf{B}^T\mathbf{B}$, a much smaller matrix
  - It is square (size = number of images squared)




- We compute the eigenvectors of $\mathbf{B}^T\mathbf{B}$; ie find the vectors $\mathbf{v}_i$ such that

  $(\mathbf{B}^T\mathbf{B})\mathbf{v}_i = \lambda_i \mathbf{v}_i$
- If we multiply each side by B

  $\mathbf{B}\,(\mathbf{B}^T\mathbf{B})\mathbf{v}_i = \mathbf{B}\lambda_i \mathbf{v}_i \quad \rightarrow \quad (\mathbf{B}\mathbf{B}^T)\,(\mathbf{B}\,\mathbf{v}_i) = \lambda_i\,(\mathbf{B}\,\mathbf{v}_i)$
- So the eigenvectors of $\mathbf{B}\mathbf{B}^T$ are the vectors $\mathbf{B}\,\mathbf{v}_i$ (we still need to scale them so that they are unit vectors)
  - This is a cheaper way to get the eigenvectors of $\mathbf{B}\mathbf{B}^T$ (at least the first *K* eigenvectors)

# Finding Matches in Eigenspace

- Say we want to find a match for an image $\mathbf{I}_1$
- One way to find a match is to look for an image $\mathbf{I}_2$ such that the sum-of-squared differences is small
  - This is the (squared) Euclidean distance between the two image vectors $\mathbf{x}_1$ and $\mathbf{x}_2$

- For large images, computing this is expensive because $\mathbf{x}_1$ and $\mathbf{x}_2$ are large vectors

- However, it is equivalent to computing the (squared) Euclidean distance between the vectors in eigenspace
  - This is much faster because the $\mathbf{y}$ vectors are small

$$\|\mathbf{x}_1 - \mathbf{x}_2\|^2$$

$$= \left\| \sum_{i=1}^{n} y_{1i}\mathbf{e}_i - \sum_{i=1}^{n} y_{2i}\mathbf{e}_i \right\|^2$$

$$\approx \left\| \sum_{i=1}^{k} y_{1i}\mathbf{e}_i - \sum_{i=1}^{k} y_{2i}\mathbf{e}_i \right\|^2$$

$$= \left\| \sum_{i=1}^{k} (y_{1i} - y_{2i})\mathbf{e}_i \right\|^2$$

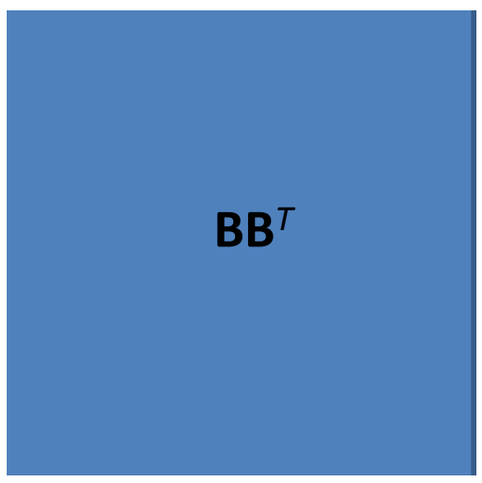
$$= \sum_{i=1}^{k} (y_{1i} - y_{2i})^2$$

$$= \|\mathbf{y}_1 - \mathbf{y}_2\|^2$$

# Approach

- Training phase
  - We read in a set of training images $\mathbf{x}_1,\ldots,\mathbf{x}_n$ and put them in vector form
  - We compute the mean of the vectors, **m**, and subtract off the mean from each vector
  - We put the vectors in the columns of the matrix **B**
  - We compute the eigenvectors **v** of $\mathbf{B}^T\mathbf{B}$ **(small, # images)**
  - We compute the principal components **u**, using **U=BV**

- Testing phase
  - We read in a new image **x** and subtract off **m**
  - We project that onto the space of PCs, using $\mathbf{y} = \mathbf{U}^T\mathbf{x}$
  - We find the closest match of **y** to the other images in the database

# AT&T Face Database

- Located at
  - http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html
  - Download att_faces.zip
- Extract to a directory called “images”

- Code on next slides
  - Loads images of a subset of people
  - Displays the images
  - (This code was developed by Chris Ostrum, CSM student)

# Load and Display

```
%% Chris Ostrum Eigenface recognition
% This algorithm uses the eigenface system (based on pricipal component
% analysis - PCA) to recognize faces.
clear all; close all;

directory = 'att_faces/s'; ext = '.pgm'; person_max = 40; face_max = 10;

person_count = 9;     up to person_max
face_count   = 9;   up to face_max-1; leaving at least one unknown for recognizing

% Principal components to keep
N = ceil(.25 * (person_count*face_count));
fprintf('Keep only the top %d principal components out of %d\n', N,
person_count*face_count);

if( person_count > person_max || face_count > face_max-1 )
    fprintf('Count values cannot exceed maximums\n');
    return;
end

[faces,irow,icol,resize,pidx,fidx] = ...
    load_faces(directory,person_count,person_max,face_count,face_max,ext);

if person_count <= 10
    display_faces(faces,irow,icol,person_count,face_count);
end
```

```
function [faces,irow,icol,resize,pidx,fidx] = load_faces(directory,person_count,person_max,face_count,face_max,ext)
% This function loads the face images
% Chris Ostrum
resize = false;
fidx = 1;

% test image size
img = imread(strcat(directory,'1/1',ext));
img = img(:,:,1);
[irow,icol] = size(img);
if( size(img,1) > 400 )
    img = imresize(img,[200, 200]);
    resize = true;
end

% setup variables
[irow,icol] = size(img);
img_size = irow*icol;
v=zeros(img_size,person_count*face_count);
w=zeros(img_size,person_count*face_count);

% create a random permutation of the people to pick from
people = randperm(person_max);
% choose a random person as our recognition person
rand_idx = mod(round(person_count*rand),person_count)+1;
% pidx is now the randomly selected person index
pidx = people(rand_idx);

for i=1:person_count
    % similarly, create a random permutation of possible faces
    faces = randperm(face_max);

    for j=1:face_count
        file = strcat(directory,num2str(people(i)),'/',num2str(faces(j)),ext);

        % pre-processing
        img = imread(file);
        if( resize ), img=imresize(img,[irow icol]); end
        img = img(:,:,1);

        w(:,(i-1)*face_count+j)=reshape(img,img_size,1);
    end

    % select our random face that is not in the training set
    if( i == rand_idx )
        fidx = faces(face_count+1);
    end
end

faces = uint8(w);
end
```

Function to load face images

# Function to display faces

```
function display_faces(face_space,irow,icol,sub_row,sub_col)
% Face Display
%   Chris Ostrum
[H W] = size(face_space);

%figure('Name','Face Display', 'NumberTitle','off', 'MenuBar', 'none')
figure('Name','Face Display')
for i = 1:W
    subplot(sub_row,sub_col,i);
    face = reshape(face_space(:,i),irow,icol);
    imshow(face,[]);
end
```

```
end
```

# faces / views

# persons

# Pick a testing image

```
%% Choose recognition image
% We choose an image not in the training set as our image to identify
% Face to recognize
file = strcat(directory,num2str(pidx),'/',num2str(fidx),ext);

recognize = imread(file);
if( resize ), recognize = imresize(recognize,[irow icol]); end
recognize = recognize(:,:,1);

figure, imshow(recognize,[]), title('Face to recognize');
```

# Calculate the mean image

- Mean

```
%% Calculate the mean
m=uint8(mean(faces,2));

figure, imshow(reshape(m,irow,icol),[]), title('Mean face');
```

- Subtract off mean from all faces

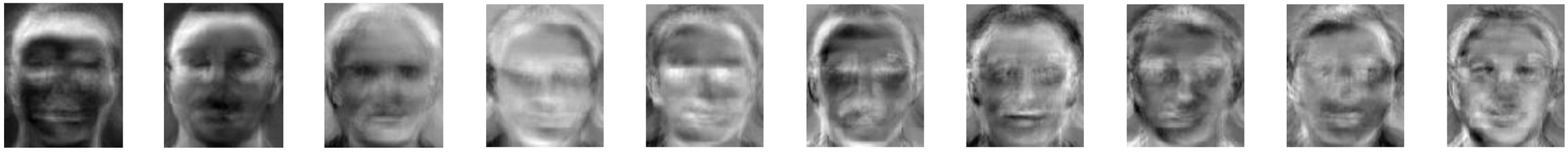
```
faces_mean = faces - uint8( single(m)*single( uint8(ones(1,size(faces,2)) ) ));
```

# Calculate Eigenfaces

```
%% Calculating eigenvectors
% L = A'A
L=single(faces_mean)'*single(faces_mean);
[V,D]=eig(L);

% Plot the eigenvalues.  Matlab's "eig" function sorts them from low to
% high, so let's reverse the order for display purposes.
eigenvals = diag(D);
figure, plot(eigenvals(end:-1:1)), title('Eigenvalues');

% Look at the mean squared error, as we increase the number of PCs to
keep.
figure, plot(sum(eigenvals) - cumsum(eigenvals(end:-1:1)));
title('Mean squareed error vs number of PCs');


% Here are the principal components.
PC=single(faces_mean)*V;

% Pick the top N eigenfaces
PC=PC(:,end:-1:end-(N-1));

display_faces(PC(:,1:10),irow,icol,1,10);  % Display first 10 eigenfaces
```

$B^TB$

$BB^T$



# Calculate Eigenspace Coefficients

- Project each of the input database images **x** onto the space of eigenimages, and get the coefficients **y** = $\mathbf{U}^T\mathbf{x}$
- Save the y's for each database image

```
%% Calculate image signature
signatures=zeros(size(faces,2),N);
for i=1:size(faces,2);
    signatures(i,:)=single(faces_mean(:,i))'*PC; % Each row is an image signature
end
figure, imshow(signatures, [], 'InitialMagnification', 300);
title('Signatures of images in database');
```

# Recognition

```
%% Recognition
%  Now run the algorithm to see if we are able to match the new face
figure('Name','Result', 'NumberTitle','off', 'MenuBar','none')
subplot(231),imshow(recognize,[]),title('Face to recognize');

% Prepare the recognition face
rec=reshape(recognize,irow*icol,1)-m;
rec_weighted=single(rec)'*PC;

fprintf('Here is the signature of the new (input) face.\n');
fprintf('These are the coefficients of the face projected onto the %d PCs:\n', N);
disp(rec_weighted);

scores=zeros(1,size(signatures,1));
for i=1:size(faces,2)
    % calculate Euclidean distance as score
    scores(i)=norm(signatures(i,:)-rec_weighted,2);
end

% display results
[C,idx] = sort(scores,'ascend');
fprintf('Top 3 scores: %f, %f, %f\n', C(1), C(2), C(3));

subplot(234);
imshow(reshape(faces(:,idx(1)),irow,icol),[]),title('Best match');
subplot(235);
imshow(reshape(faces(:,idx(2)),irow,icol),[]),title('2nd best');
subplot(236);
imshow(reshape(faces(:,idx(3)),irow,icol),[]),title('3rd best');
```

- Put the test image into vector form, find its eigenspace coefficients
- Find the distance between its vector of coefficients, against each image in the database


Result
Recognition
Best match
2nd best
3rd best

# Summary / Questions

- In principal components analysis (PCA), we map a set of vectors into another coordinate system.
- The axes of that coordinate system are called “principal components” (PCs).
  - The PCs are sorted so that the first PC corresponds to the direction of the most variation in the input data, the second PC corresponds to the direction of the second most variation, and so on.
  - If we use only the top *k* PCs for reconstruction, we can still get a good approximation to the input data.

- What two ways to use PCA on images were demonstrated?