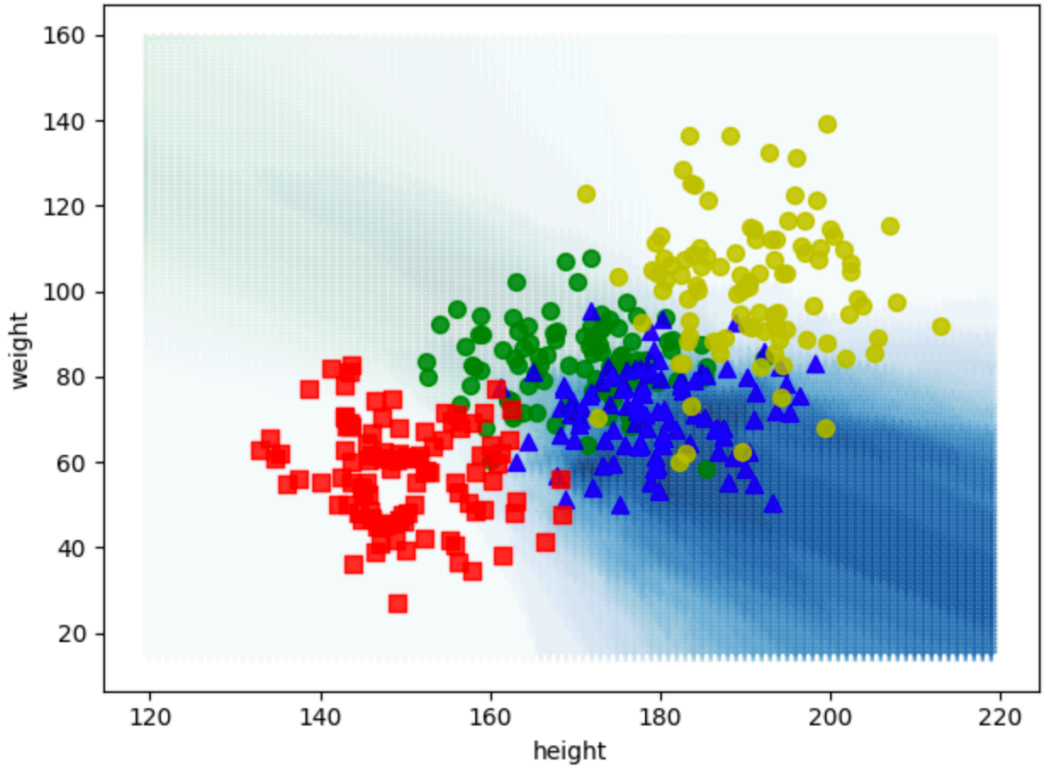
Computer Vision : Insight of object detection algorithms :

Lesson3 KNN to recognize MNIST datasets



By Daniel
Research Scientist
danielshaving@gmail.com

Table of contents

- Interesting Point Extraction (Part II)
  - Hough Transformation
  - Lucas_Kanade Track
- KNN to recognize MNIST data

Hough Transformation: How to extract the lane line?

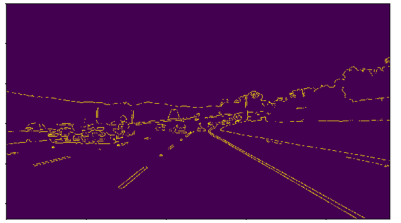
Canny Edge

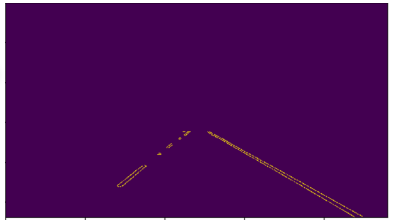
Region of Interest

Hough Transformation

Threshold

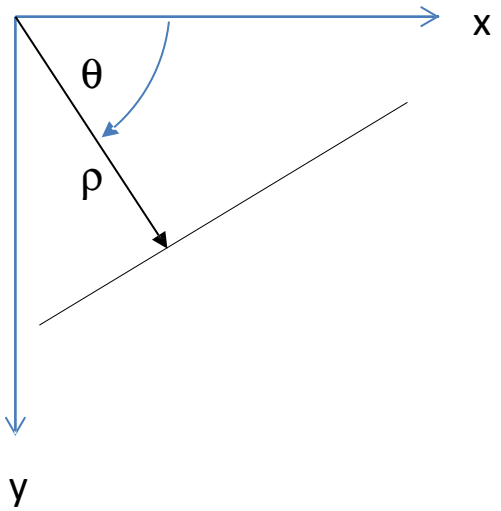
Draw Lines

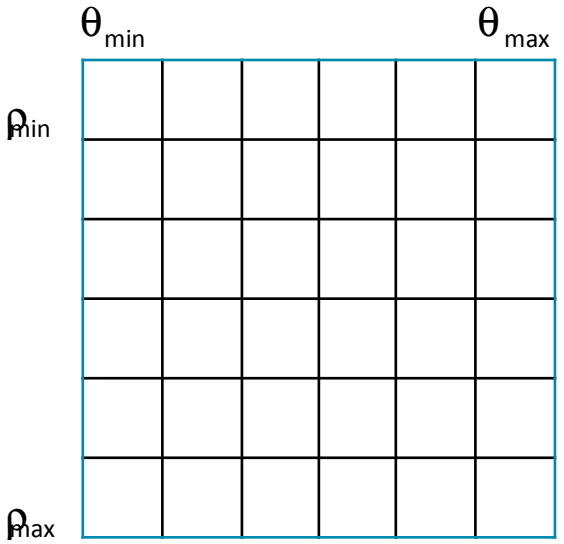




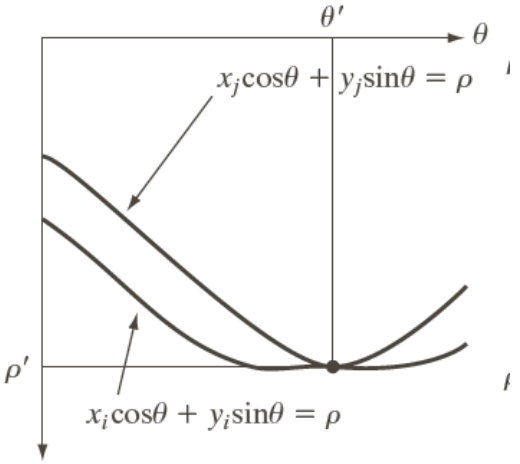
Polar Coordinate Representation of Line

- $\rho = x \cos\theta + y \sin\theta$
  - Avoids infinite slope
  - Constant resolution



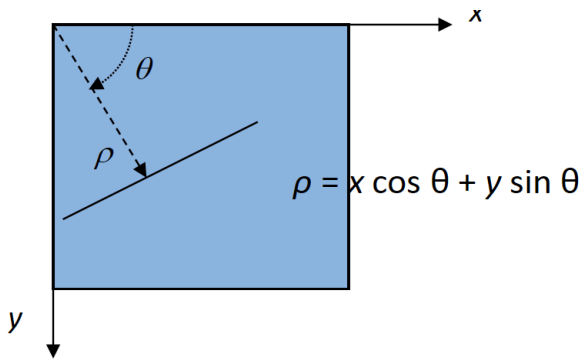


The parameter space transform of a point is a sinusoidal curve

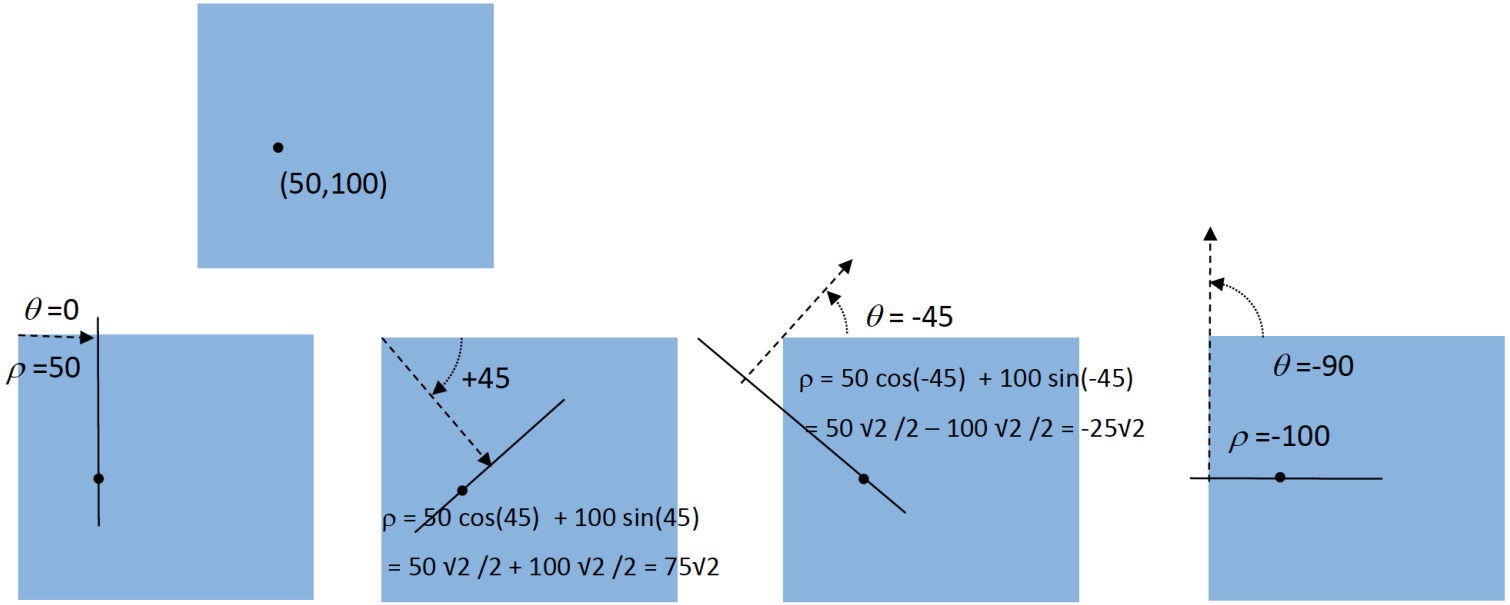


Polar Coordinate Representation of Line

- Angle, axis conventions
  - angle range is -90°..+89°
  - rho range is –dmax..+dmax
    - dmax is the largest possible distance



- Example of a point at (x,y) = (50,100)



Hough Transformation Example

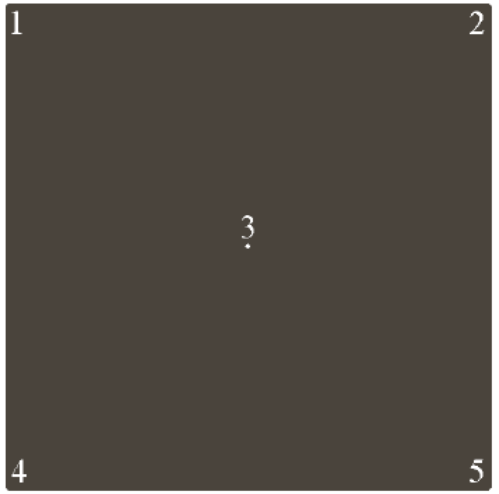
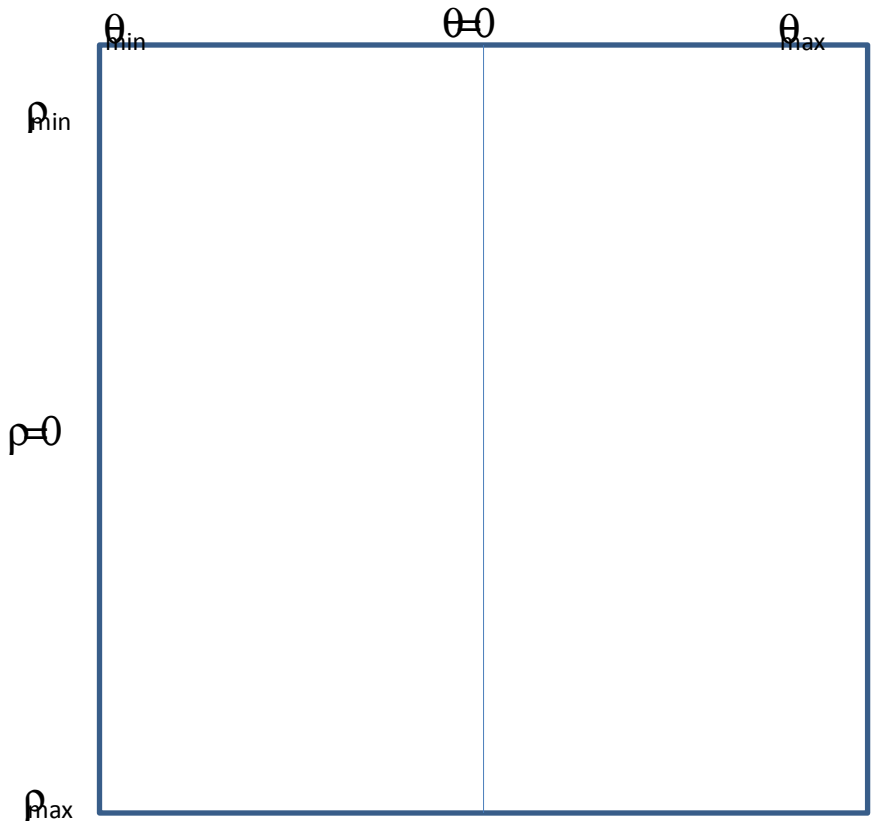
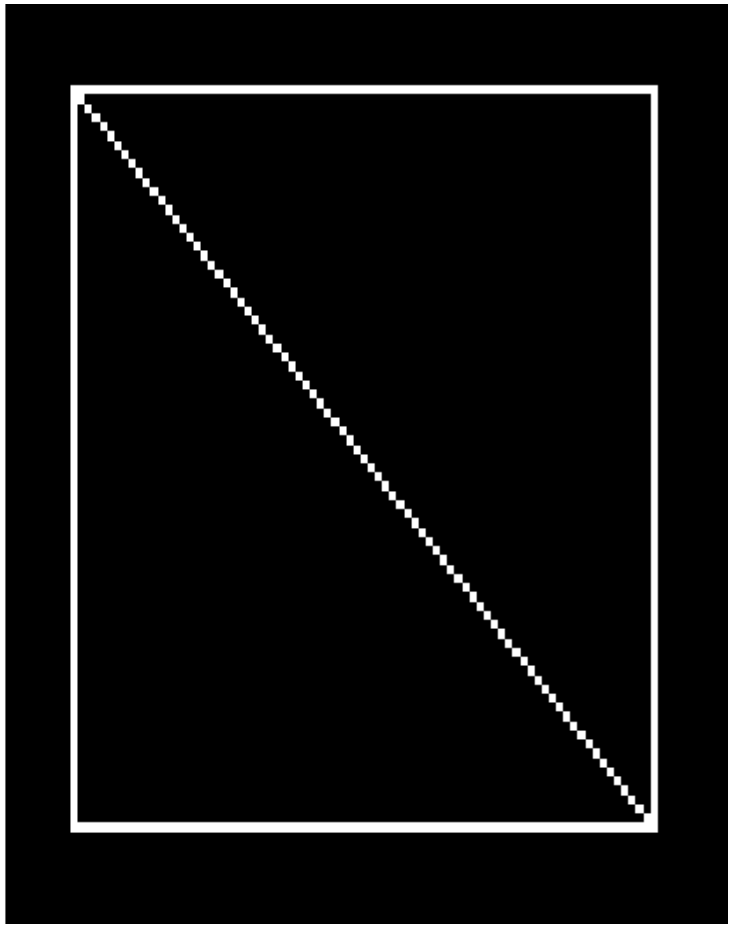
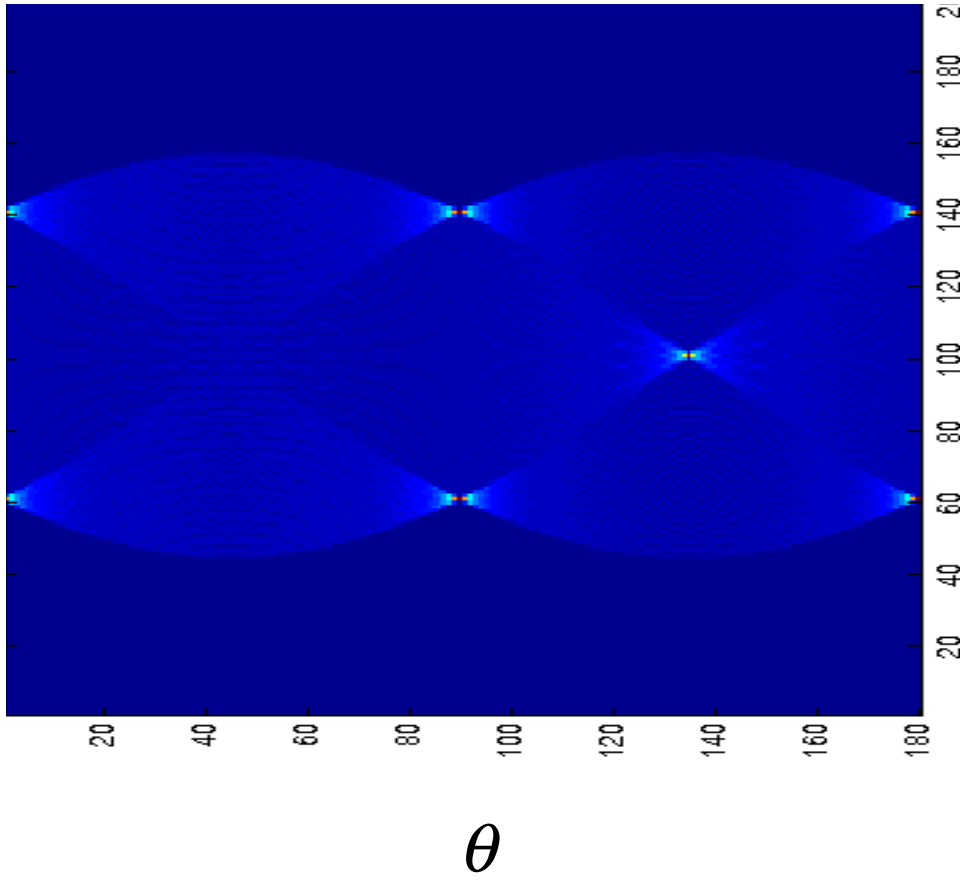


Image of size 100x100, containing 5 points (at the corners)

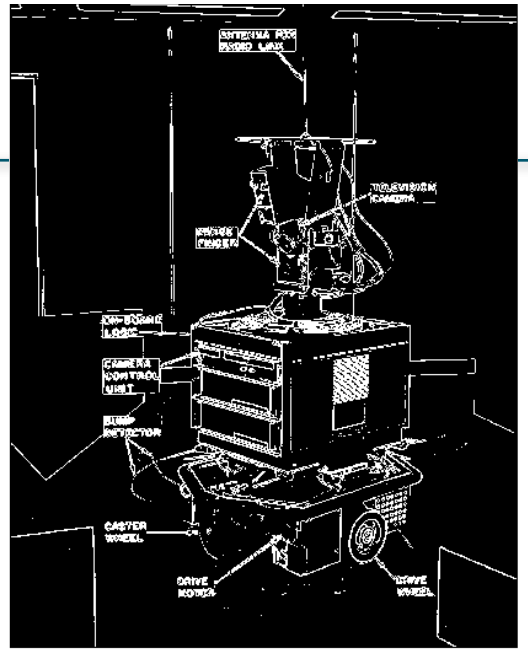


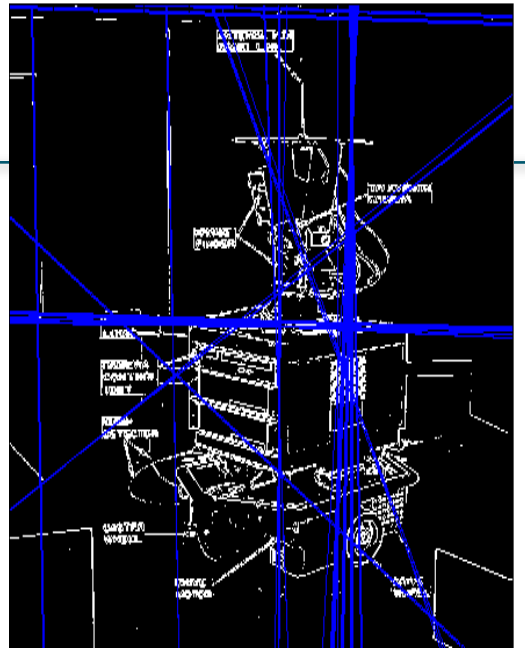
Example

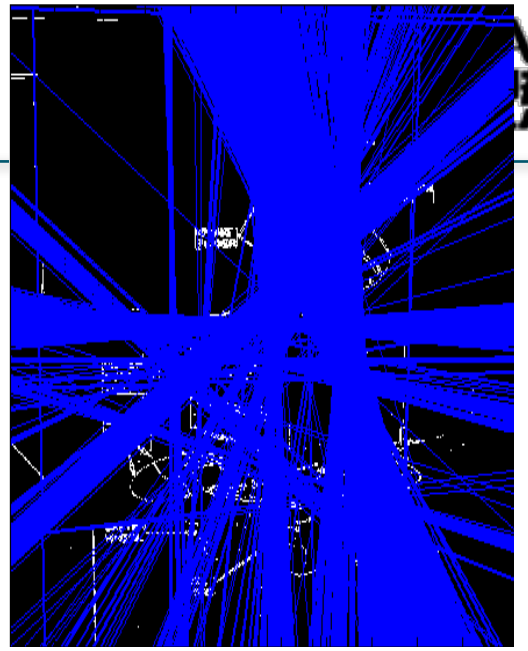


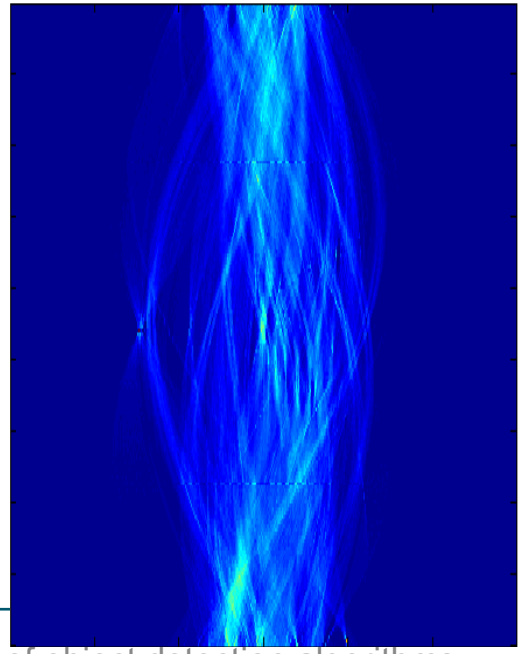


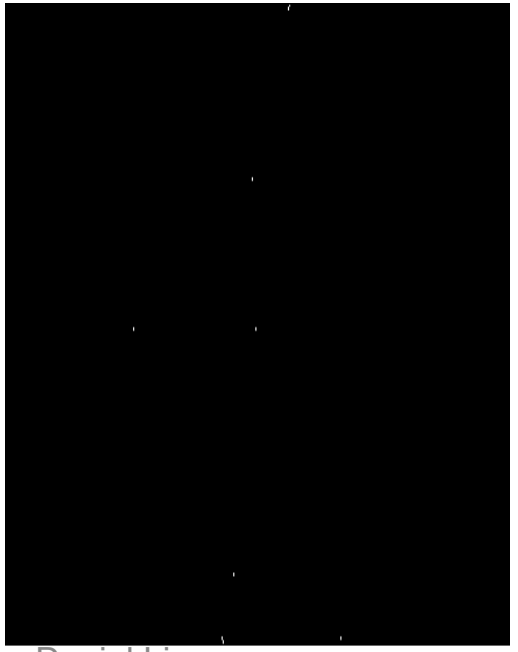
Example











Draw Hough Lines

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

from skimage.transform import hough_line
from skimage.draw import line

img = np.zeros((100, 150), dtype=bool)
img[30, :] = 1
img[:, 65] = 1
img[35:45, 35:50] = 1
rr, cc = line(60, 130, 80, 10)
img[rr, cc] = 1
img += np.random.random(img.shape) > 0.95

out, angles, d = hough_line(img)

fix, axes = plt.subplots(1, 2, figsize=(7, 4))

axes[0].imshow(img, cmap=plt.cm.gray)
axes[0].set_title('Input image')

axes[1].imshow(
    out, cmap=plt.cm.bone,
    extent=(np.rad2deg(angles[-1]), np.rad2deg(angles[0]), d[-1], d[0]))
axes[1].set_title('Hough transform')
axes[1].set_xlabel('Angle (degree)')
axes[1].set_ylabel('Distance (pixel)')

plt.tight_layout()
plt.show()
```

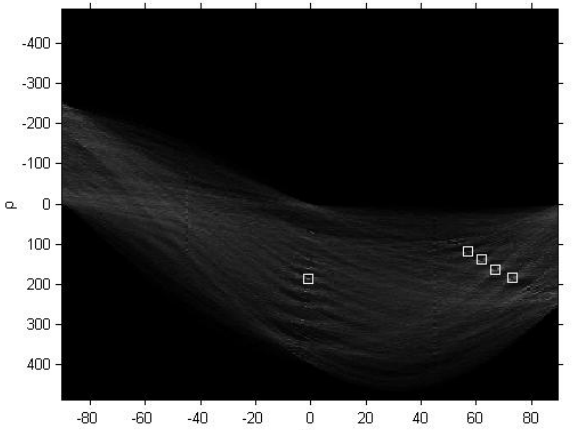
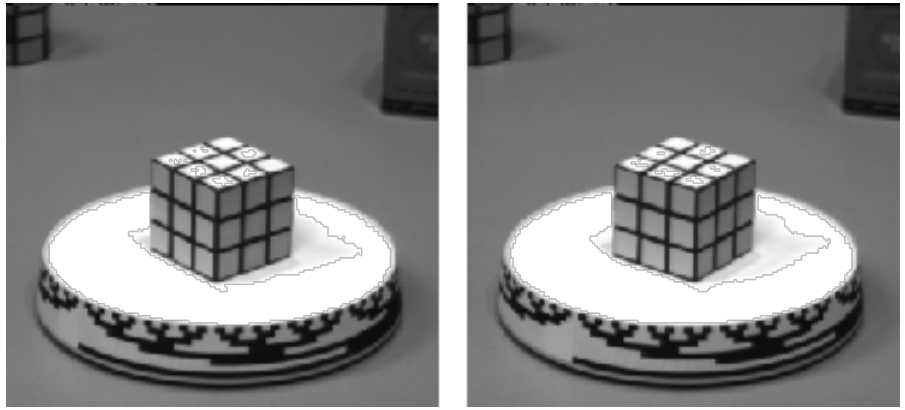
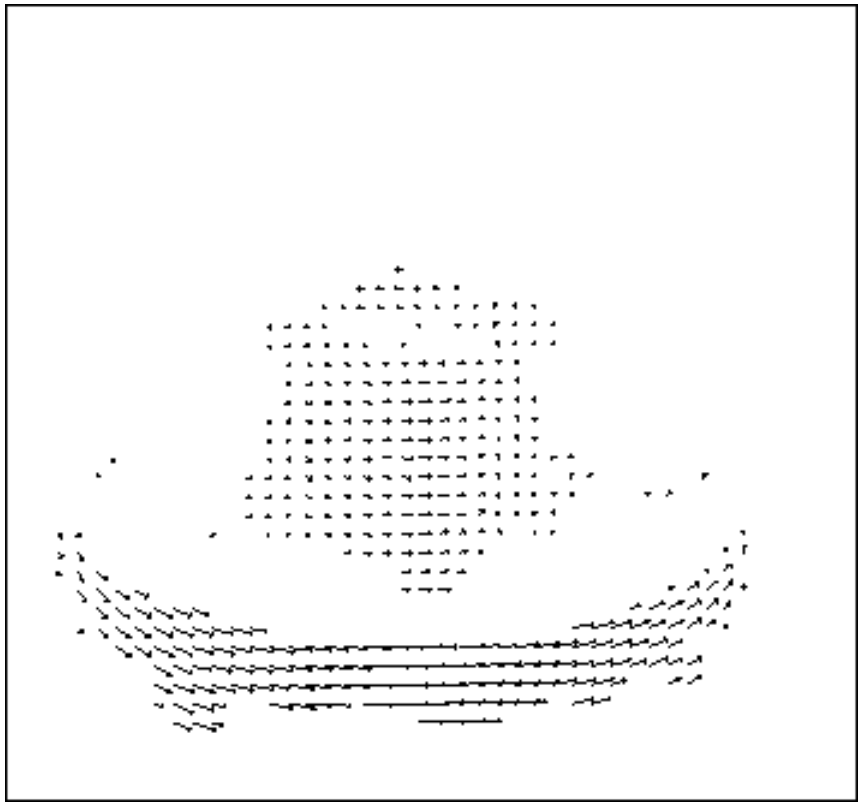


Table of contents

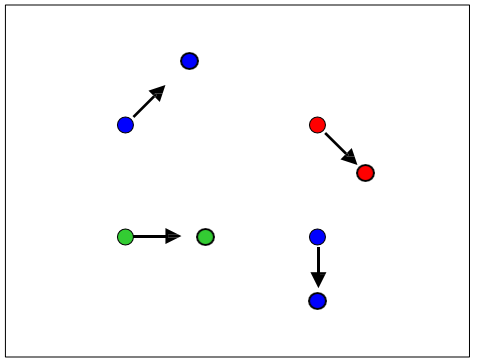
- Interesting Point Extraction (Part II)
  - Hough Transformation
  - Lucas_Kanade Track
- KNN to recognize MNIST data

Optical flow

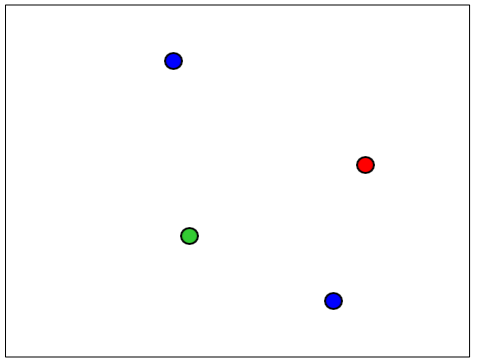




Optical flow about I(x,y,t)



$I(x,y,t-1)$

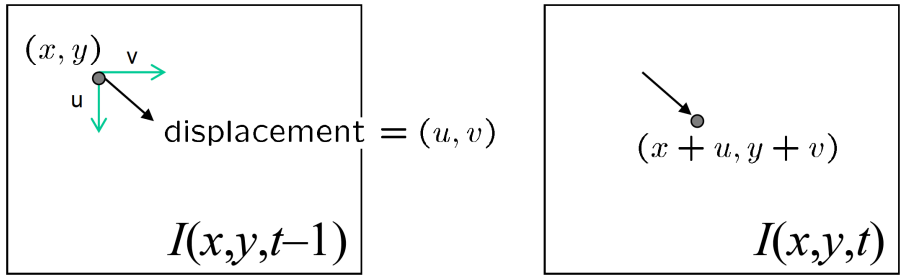


$I(x,y,t)$

Given two subsequent frames, estimate the apparent motion field u(x,y), v(x,y) between them

- Key assumptions
  - **Brightness constancy:** projection of the same point looks the same in every frame
  - **Small motion:** points do not move very far
  - **Spatial coherence:** points move like their neighbors

Optical flow constraints (grayscale images)



Brightness Constancy Equation:

$$I(x, y, t-1) = I(x + u(x, y), y + v(x, y), t)$$

Linearizing the right side using Taylor expansion:

$$I(x+u, y+u, t) \approx I(x, y, t-1) + I_x \cdot u(x, y) + I_y \cdot v(x, y) + I_t$$

$$I(x+u, y+u, t) - I(x, y, t-1) = I_x \cdot u(x, y) + I_y \cdot v(x, y) + I_t$$

Hence, $I_x \cdot u + I_y \cdot v + I_t \approx 0 \quad \rightarrow \nabla \mathrm{I} \cdot \left[\mathrm{u}\ \ \mathrm{v}\right]^{\mathrm{T}} + \mathrm{I}_{\mathrm{t}} = 0$

Lucas-Kanade algorithm

- Brightness constancy
- One equation two unknowns

$$0 = I_t(\mathbf{p_i}) + \nabla I(\mathbf{p_i}) \cdot [u\ v]$$

unknown flow vector

temporal gradient **spatial gradient**

- How to get more equations for a pixel?
  - Basic idea: impose additional constraints
    - one method: pretend the pixel's neighbors have the same (u,v)
      - If we use a 5x5 window, that gives us 25 equations per pixel!

$$\begin{bmatrix} I_x(\mathbf{p_1}) & I_y(\mathbf{p_1}) \\ I_x(\mathbf{p_2}) & I_y(\mathbf{p_2}) \\ \vdots & \vdots \\ I_x(\mathbf{p_{25}}) & I_y(\mathbf{p_{25}}) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} I_t(\mathbf{p_1}) \\ I_t(\mathbf{p_2}) \\ \vdots \\ I_t(\mathbf{p_{25}}) \end{bmatrix}$$

A (25×2) d (2×1) b (25×1)

Lucas-Kanade algorithm

- Now we have more equations than unknowns

$$\underset{25\times2}{A}\ \underset{2\times1}{d} = \underset{25\times1}{b} \longrightarrow \text{minimize } \|Ad - b\|^2$$

  - Solution: solve least squares problem
    - minimum least squares solution given by solution (in d) of:

$$\underset{2\times2}{(A^T A)}\ \underset{2\times1}{d} = \underset{2\times1}{A^T b}$$

$$\begin{bmatrix} \sum I_x I_x & \sum I_x I_y \\ \sum I_x I_y & \sum I_y I_y \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} \sum I_x I_t \\ \sum I_y I_t \end{bmatrix}$$

$A^T A$ $A^T b$

    - The summations are over all pixels in the K x K window
    - This technique was first proposed by Lucas & Kanade (1981)
      - described in Trucco & Verri reading

General situation in LK tracking

$$E(u,v) = \sum [I(x+u,y+v) - T(x,y)]^2$$

template
(model)

current frame

u,v = hypothesized location of
template in current frame

General situation in LK tracking

$$E(u,v) = \sum [I(x+u,y+v) - T(x,y)]^2 \xrightarrow{\text{generalize}} \sum [\, I(W([x,y];P)) - T([x,y])]^2$$

Tyler Expansion:

$$I(W([x,y];P+\Delta P)) \simeq I(W([x,y];P)) + \nabla I \frac{\partial W}{\partial P} \Delta P$$

general equation of Jacobian

$$Let \quad W([x,y];P) = [W_x, W_y]$$

$$\frac{\partial W}{\partial P} = \begin{bmatrix} \frac{\partial W_x}{\partial P_1} & \frac{\partial W_x}{\partial P_2} & \frac{\partial W_x}{\partial P_3} & \cdots & \frac{\partial W_x}{\partial P_n} \\ \frac{\partial W_y}{\partial P_1} & \frac{\partial W_y}{\partial P_2} & \frac{\partial W_y}{\partial P_3} & \cdots & \frac{\partial W_y}{\partial P_n} \end{bmatrix}$$

affine warp function (6 parameters)

$$W([x,y];P) = \begin{pmatrix} 1+p_1 & p_3 & p_5 \\ p_2 & 1+p_4 & p_6 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \Rightarrow \frac{\partial W}{\partial P} = \frac{\partial \begin{bmatrix} x + xP_1 + yP_3 + P_5 \\ xP_2 + y + yP_4 + P_6 \end{bmatrix}}{\partial P}$$

$$= \begin{bmatrix} x & 0 & y & 0 & 1 & 0 \\ 0 & x & 0 & y & 0 & 1 \end{bmatrix}$$

Lucas-Kanade algorithm

Loop Calculation
until P is negligible

Warp I to obtain $I(W([x\ y];P))$

Compute the error image $T(x) - I(W([x\ y];\ P))$

Warp the gradient ∇I with $W([x\ y];\ P)$

Evaluate $\frac{\partial W}{\partial P}$ at $([x\ y];\ P)$ (Jacobian)

Compute steepest descent images $\nabla I \frac{\partial W}{\partial P}$

Compute Hessian matrix $\sum (\nabla I \frac{\partial W}{\partial P})^T (\nabla I \frac{\partial W}{\partial P})$

Compute $\sum (\nabla I \frac{\partial W}{\partial P})^T (T(x,y) - I(W([x,y];P)))$

Compute ΔP

Update $P \longleftarrow P + \Delta P$

CV2.calcOpticalFlowPyrLK

Python: `cv2.calcOpticalFlowPyrLK`(prevImg, nextImg, prevPts[, nextPts[, status[, err[, winSize[, maxLevel[, criteria[, flags[, minEigThreshold]]]]]]]]) → nextPts, status, err

Parameters:

- **prevImg** – first 8-bit input image or pyramid constructed by `buildOpticalFlowPyramid()`.
- **nextImg** – second input image or pyramid of the same size and the same type as `prevImg`.
- **prevPts** – vector of 2D points for which the flow needs to be found; point coordinates must be single-precision floating-point numbers.
- **nextPts** – output vector of 2D points (with single-precision floating-point coordinates) containing the calculated new positions of input features in the second image; when `OPTFLOW_USE_INITIAL_FLOW` flag is passed, the vector must have the same size as in the input.
- **status** – output status vector (of unsigned chars); each element of the vector is set to 1 if the flow for the corresponding features has been found, otherwise, it is set to 0.
- **err** – output vector of errors; each element of the vector is set to an error for the corresponding feature, type of the error measure can be set in `flags` parameter; if the flow wasn't found then the error is not defined (use the `status` parameter to find such cases).
- **winSize** – size of the search window at each pyramid level.
- **maxLevel** – 0-based maximal pyramid level number; if set to 0, pyramids are not used (single level), if set to 1, two levels are used, and so on; if pyramids are passed to input then algorithm will use as many levels as pyramids have but no more than `maxLevel`.
- **criteria** – parameter, specifying the termination criteria of the iterative search algorithm (after the specified maximum number of iterations `criteria.maxCount` or when the search window moves by less than `criteria.epsilon`.
- **flags** –

  operation flags:

  - **OPTFLOW_USE_INITIAL_FLOW** uses initial estimations, stored in `nextPts`; if the flag is not set, then `prevPts` is copied to `nextPts` and is considered the initial estimate.
  - **OPTFLOW_LK_GET_MIN_EIGENVALS** use minimum eigen values as an error measure (see `minEigThreshold` description); if the flag is not set, then L1 distance between patches around the original and a moved point, divided by number of pixels in a window, is used as a error measure.
- **minEigThreshold** – the algorithm calculates the minimum eigen value of a 2x2 normal matrix of optical flow equations (this matrix is called a spatial gradient matrix in [Bouguet00]), divided by number of pixels in a window; if this value is less than `minEigThreshold`, then a corresponding feature is filtered out and its flow is not processed, so it allows to remove bad points and get a performance boost.

Tracking Results

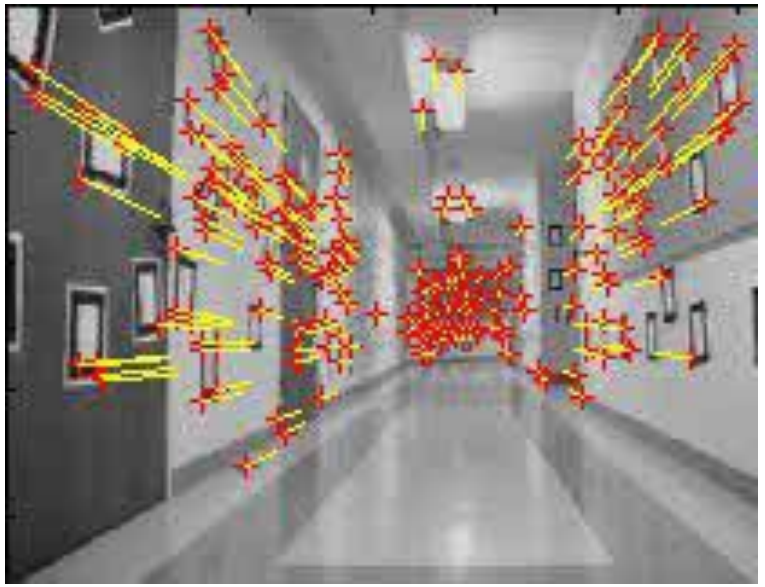
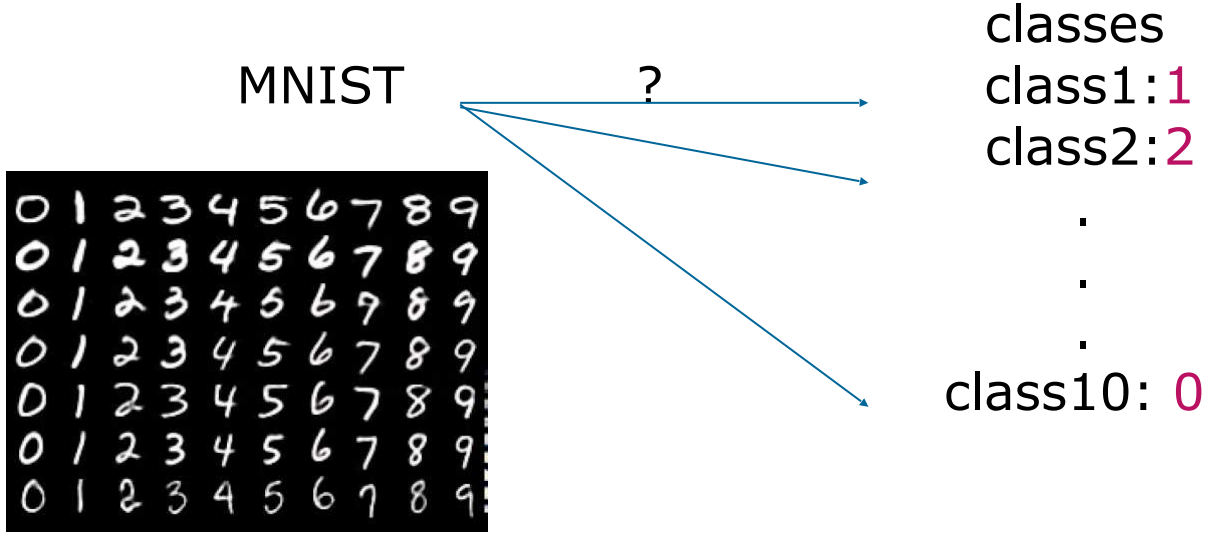


Table of contents

- Interesting Point Extraction (Part II)
- KNN to recognize MNIST data
  - MNIST datasets
  - KNN Algorithms
  - KD-Tree model
  - SKLearn Functions

Problem: classification of a simple image

- Text classification (text categorization): assign documents to one or more predefined categories



MNIST

MNIST is a dataset consists of handwriting training and test images, labeled by 0-9

http://yann.lecun.com/exdb/mnist/

```
train-images-idx3-ubyte.gz:  training set images (9912422 bytes)
train-labels-idx1-ubyte.gz:  training set labels (28881 bytes)
t10k-images-idx3-ubyte.gz:   test set images (1648877 bytes)
t10k-labels-idx1-ubyte.gz:   test set labels (4542 bytes)
```

Split

- mnist.**train**.images
- mnist.**train**.labels

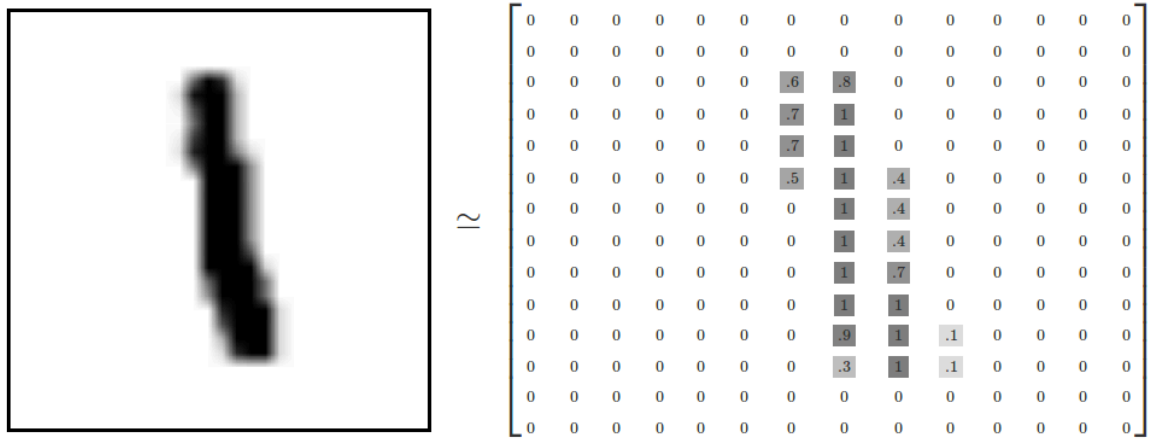
- mnist.**validation**.images
- mnist.**validation**.labels

- mnist.**test**.images
- mnist.**test**.labels

MNIST dataset Images

Features:

28 matrix



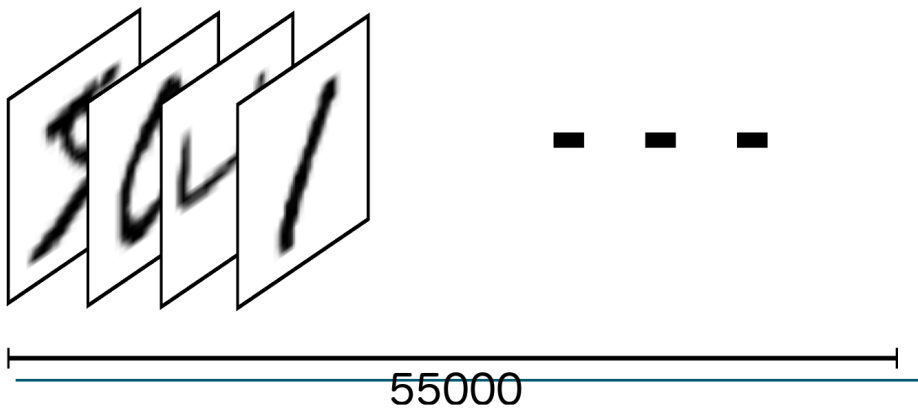
≈

Flatten

784

784-dim vector

mnist.train.xs



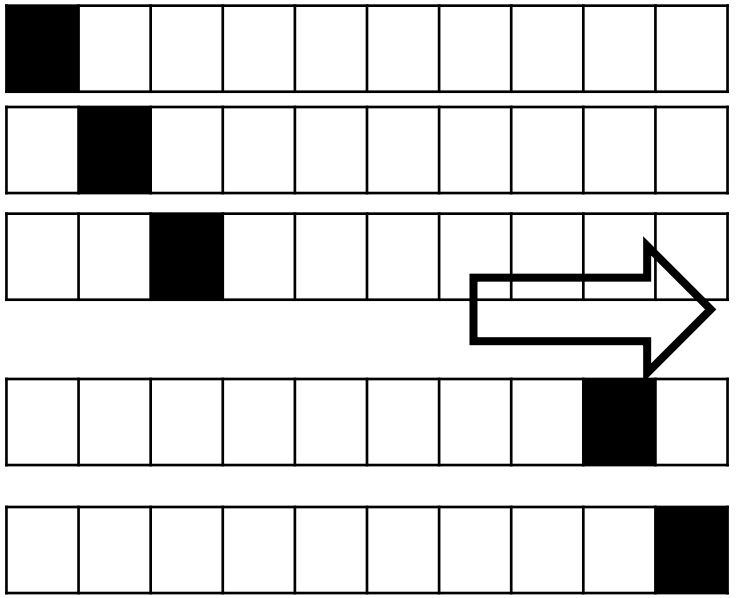
55000

MNIST dataset Labels

Labels:

0
1
2
…
8
9

One-hot vector



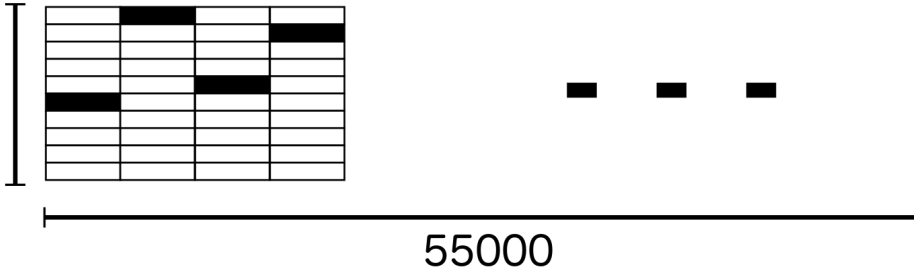
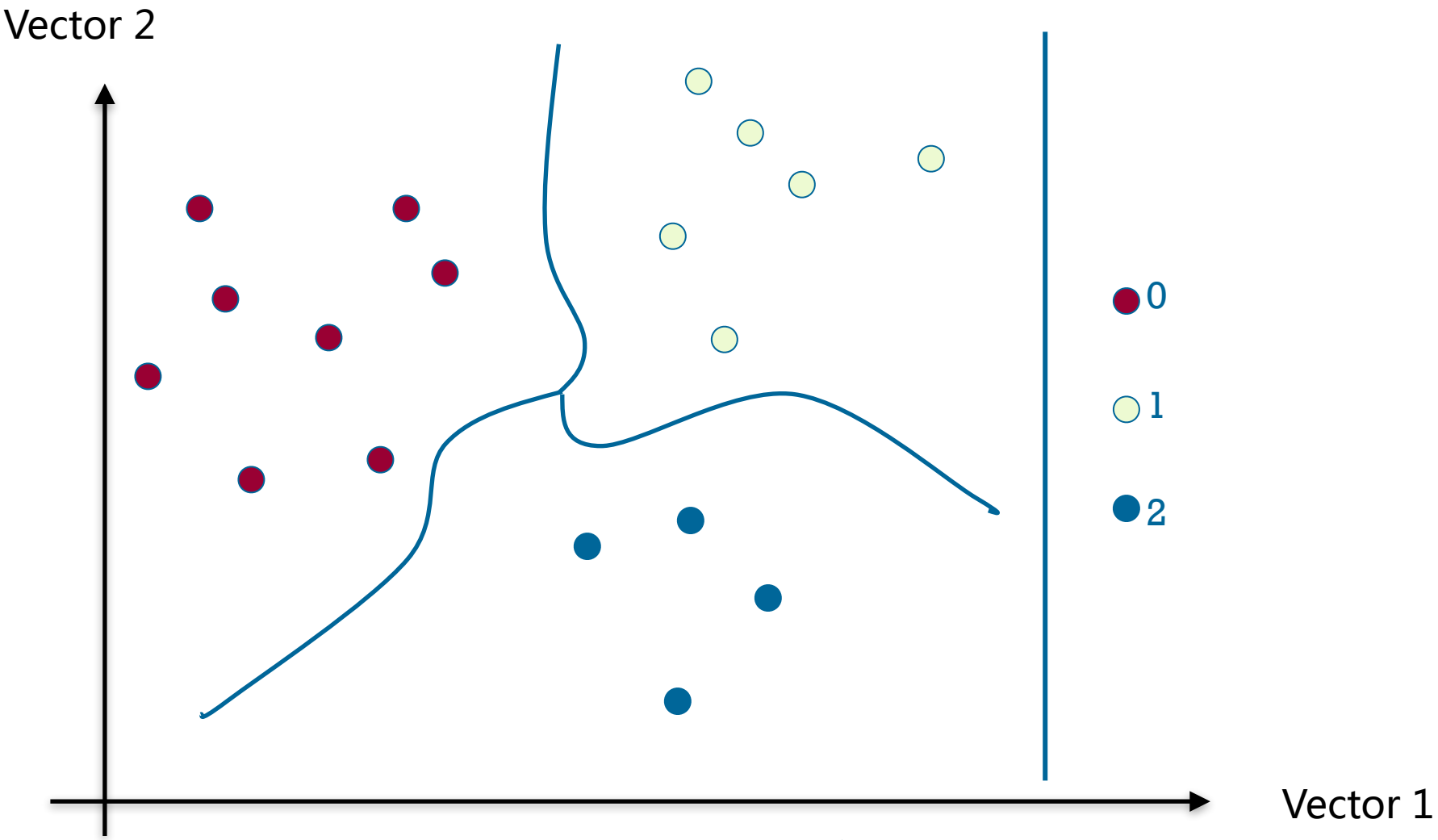


Table of contents

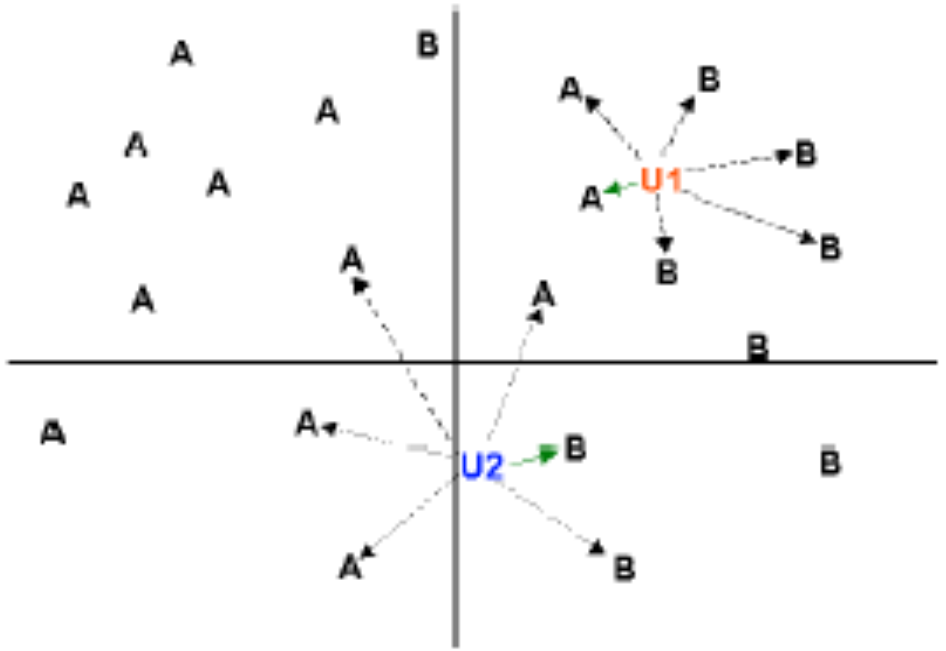
- Interesting Point Extraction (Part II)
- KNN to recognize MNIST data
  - MNIST datasets
  - KNN Algorithms
  - KD-Tree model
  - SKLearn Functions

Illustration of Handwriting Classification



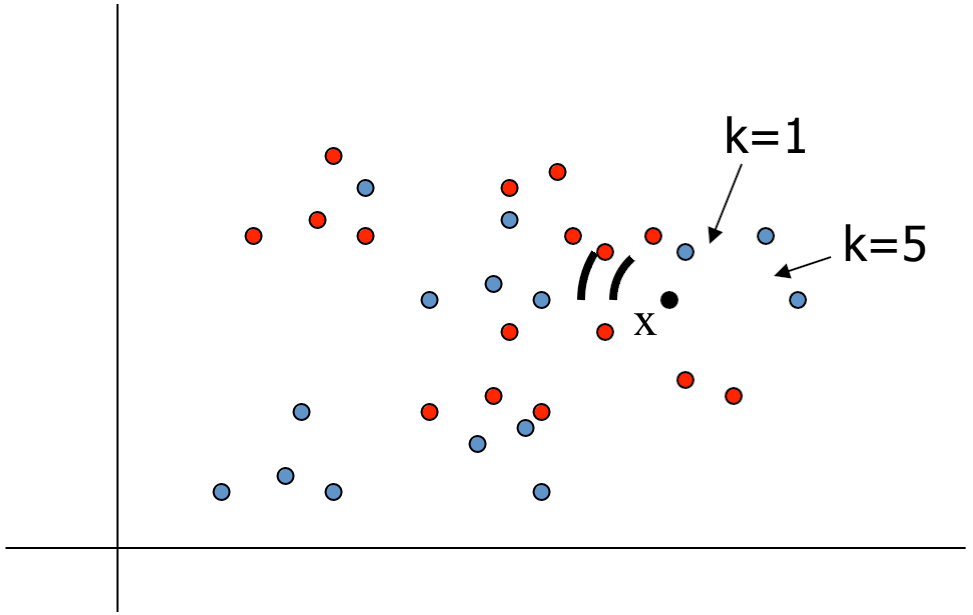
K-Nearest-Neighbor Algorithm

- Principle: points (documents) that are close in the space belong to the same class



K-Nearest-Neighbor Algorithm

- To classify a new input vector x, examine the k-closest training data points to x and assign the object to the most frequently occurring class



common values for k: 3, 5

K-Nearest Neighbour

- Notation: object with p measurements

$$x^i = (x_1^i, x_2^i, \ldots, x_p^i)$$

- Most common distance metric is *Euclidean* distance:

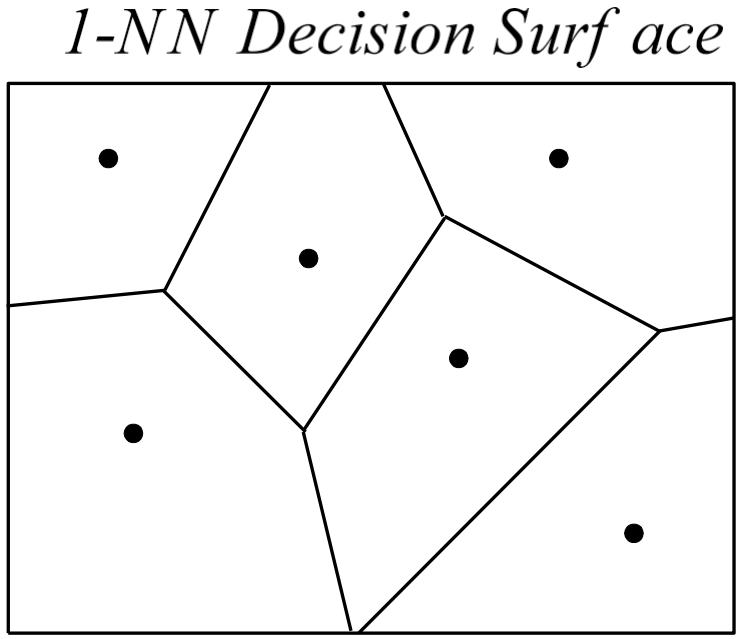
$$d_E(x^i, x^j) = \left(\sum_{k=1}^{p} \left(x_k^i - x_k^j \right)^2 \right)^{\frac{1}{2}}$$

- ED makes sense when different measurements are commensurate; each is variable measured in the same units.

- If the measurements are different, say length and weight, it is not clear.

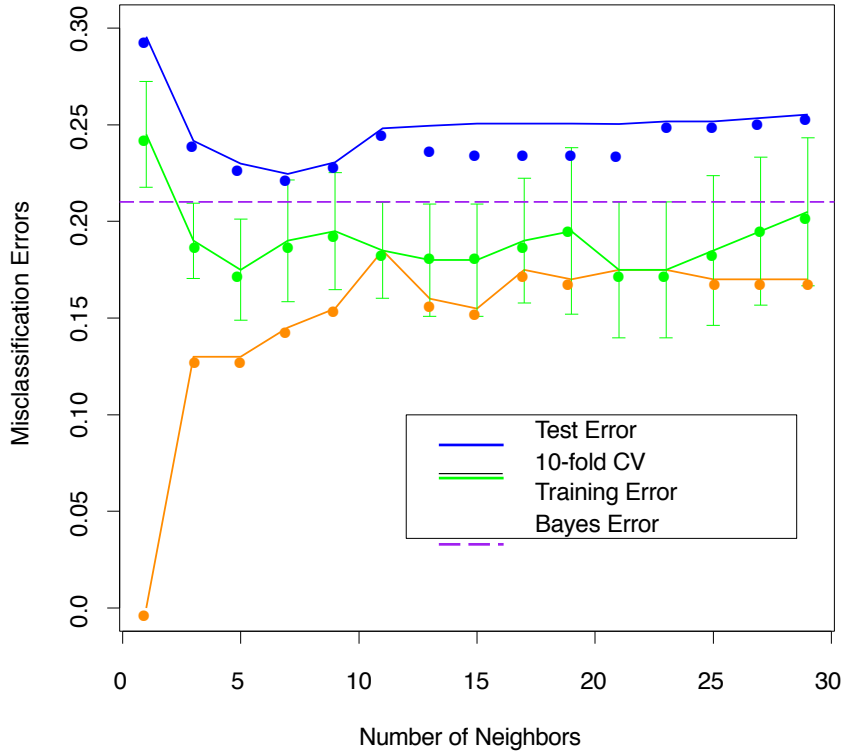
Decision Boundaries

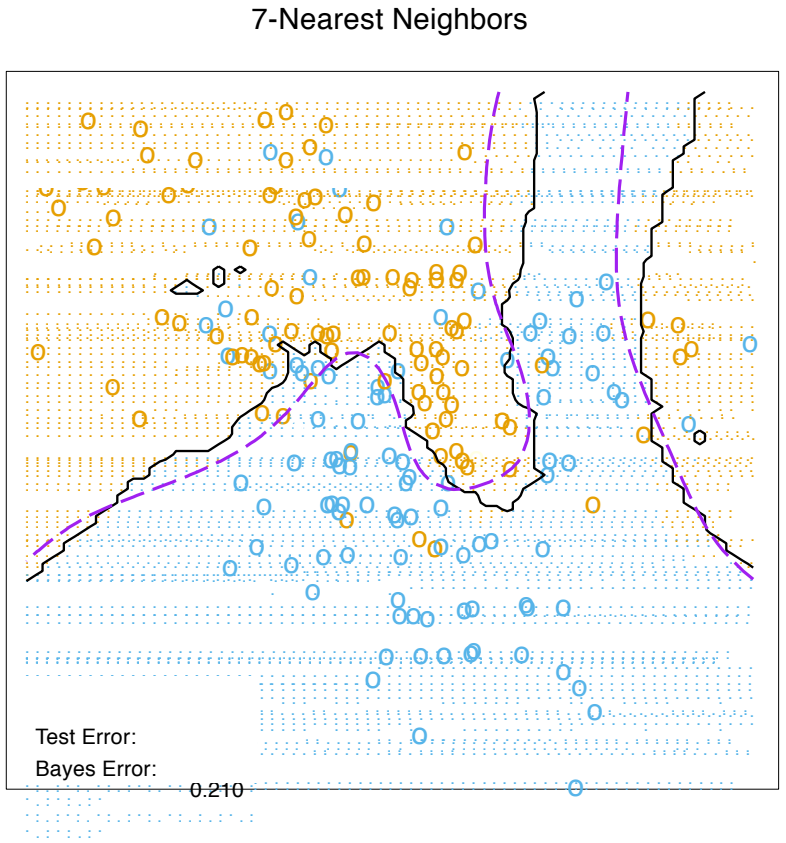
- The nearest neighbor algorithm does not explicitly compute decision boundaries. However, the decision boundaries form a subset of the Voronoi diagram for the training data.



The more examples that are stored, the more complex the decision boundaries can become

Decision boundary

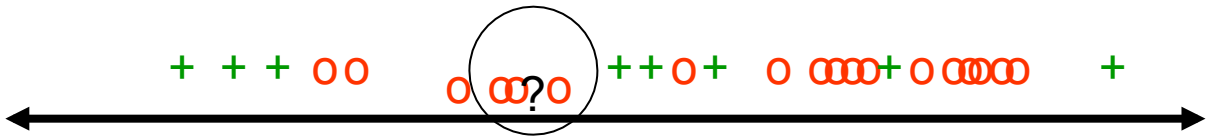




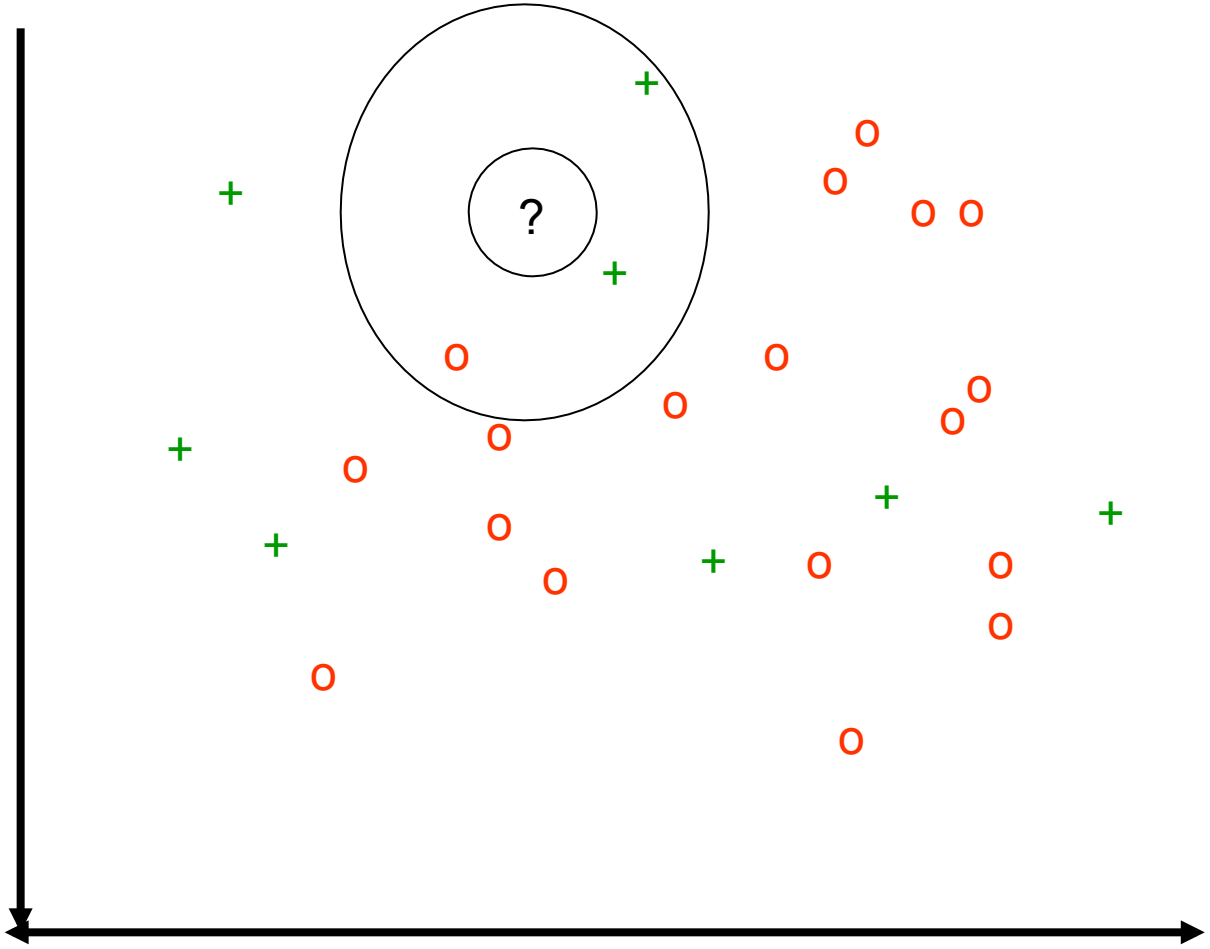
Nearest Neighbour algorithm

- Nearest neighbor breaks down in high-dimensional spaces because the “neighborhood” becomes very large.
- Suppose we have 5000 points uniformly distributed in the unit hypercube and we want to apply the 5-nearest neighbor algorithm.
- Suppose our query point is at the origin.
  - 1D –
    - On a one dimensional line, we must go a distance of 5/5000 = 0.001 on average to capture the 5 nearest neighbors
  - 2D –
    - In two dimensions, we must go sqrt(0.001) to get a square that contains 0.001 of the volume
  - D –
    - In D dimensions, we must go $(0.001)^{1/D}$

D = 1 and D = 2

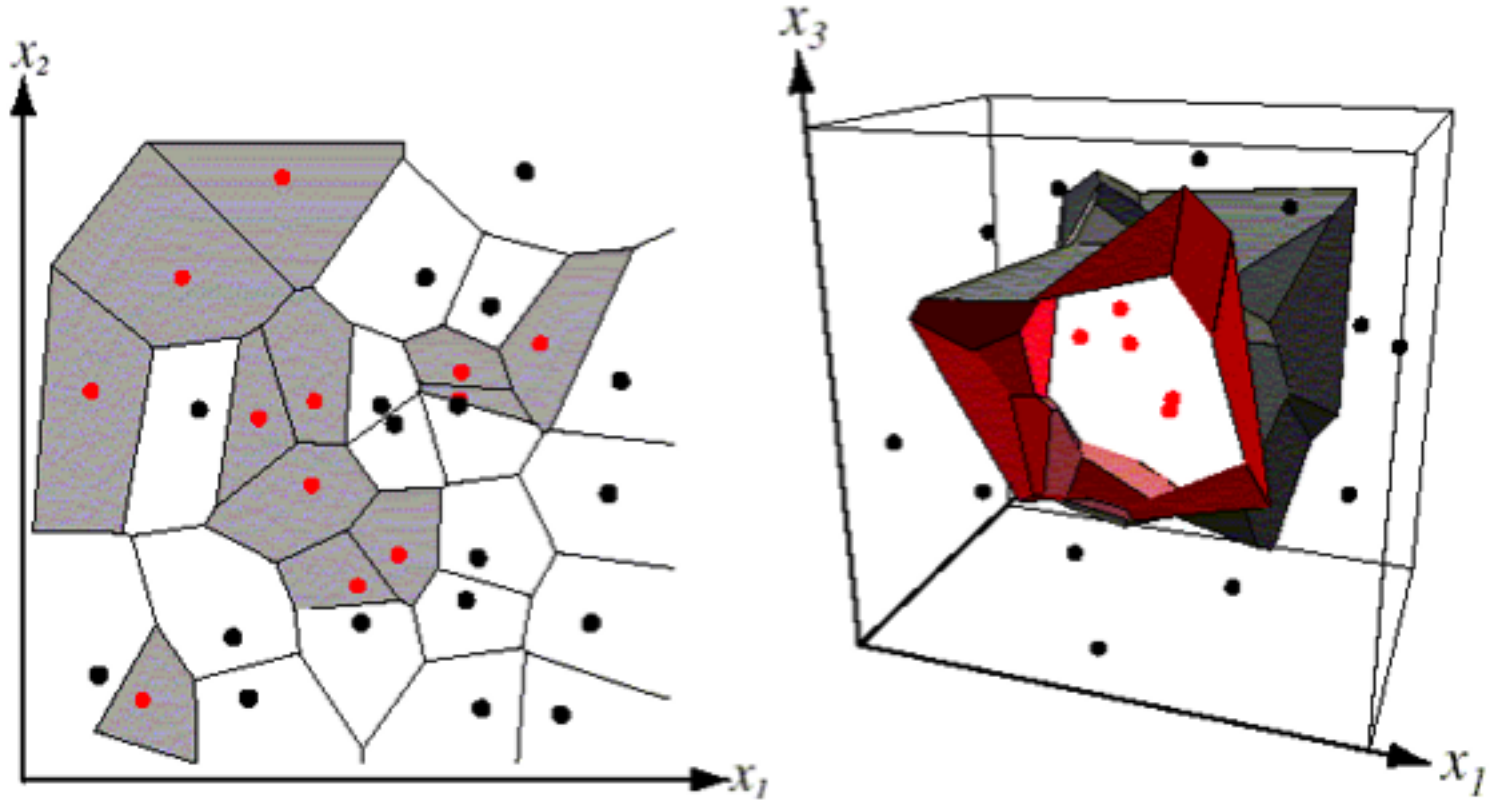


D = 1



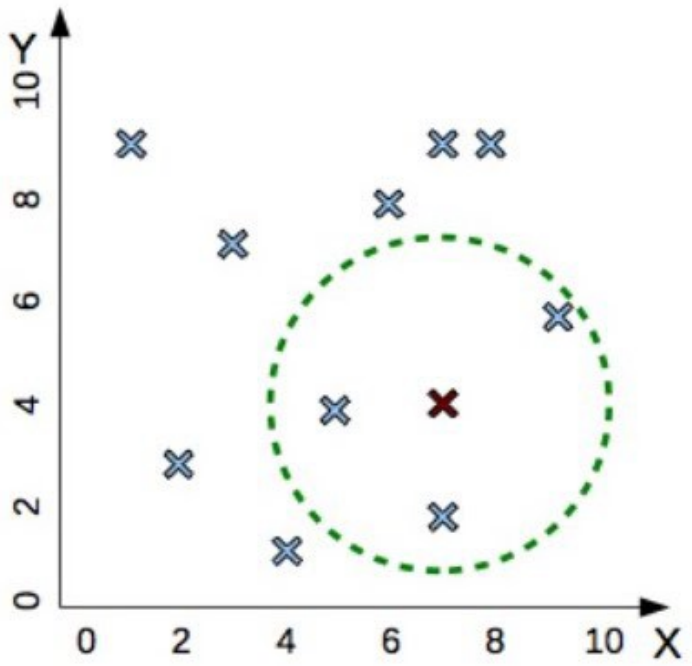
D = 2

D = 2 and D = N



K Nearest Neighbours: Linear Scan

What you see



Find nearest neighbors
of the testing point (red)

What algorithm sees

- Training set:

  {(1,9), (2,3), (4,1),
  (3,7), (5,4), (6,8),
  (7,2), (8,8), (7,9), (9,6)}

- Testing instance:

  (7,4)

- Nearest neighbors?

  compare one-by-one
  to each training instance

- n comparisons
- each takes d operations

Speed of KNN

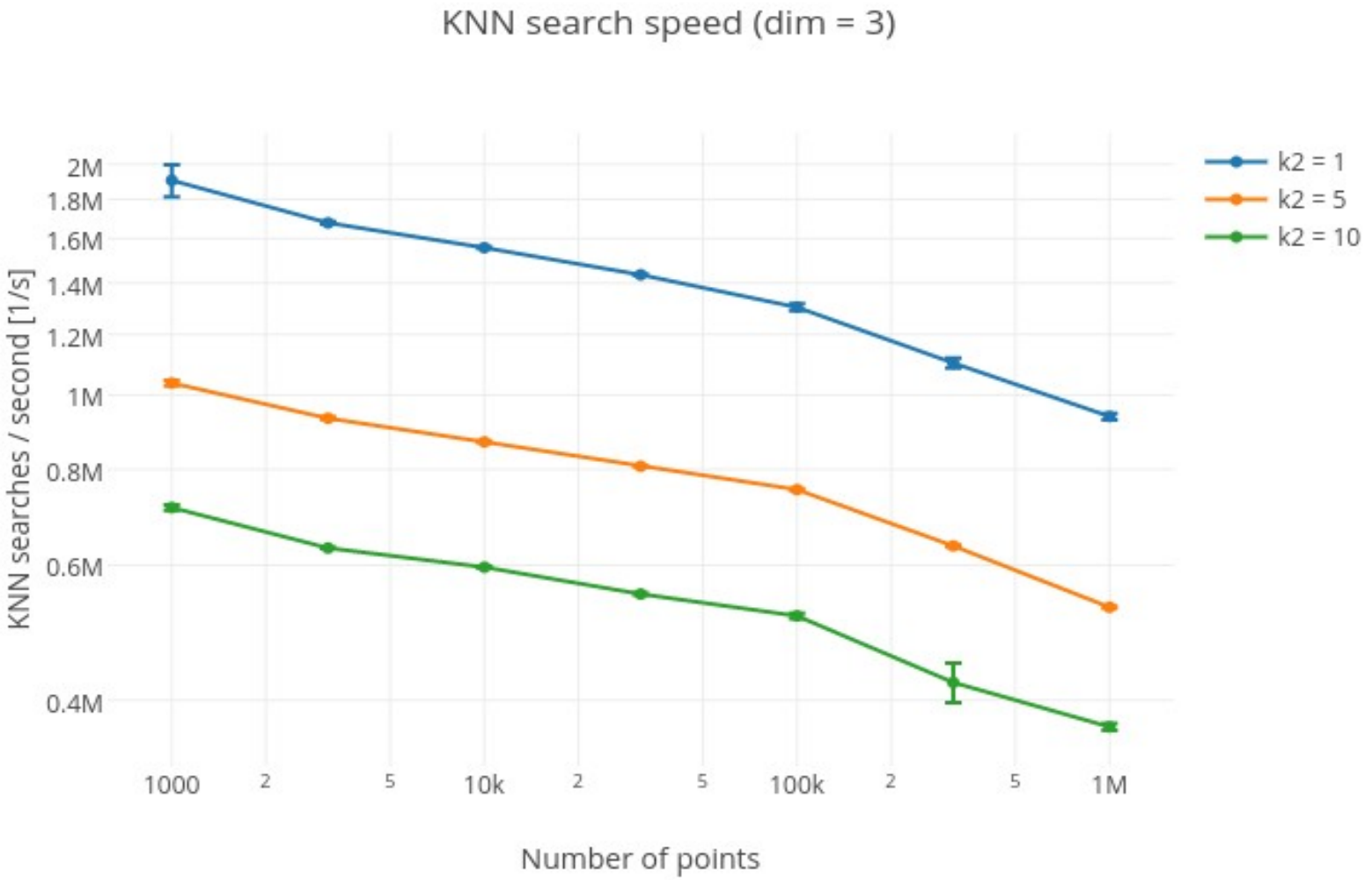
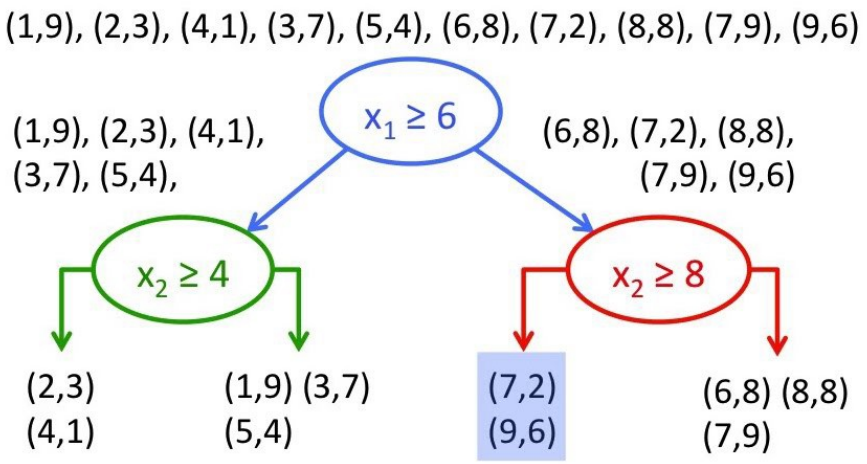

KNN search speed (dim = 3)
KNN searches / second [1/s]
2M
1.8M
1.6M
1.4M
1.2M
1M
0.8M
0.6M
0.4M
1000
2
5
10k
2
5
100k
2
5
1M
Number of points
k2 = 1
k2 = 5
k2 = 10

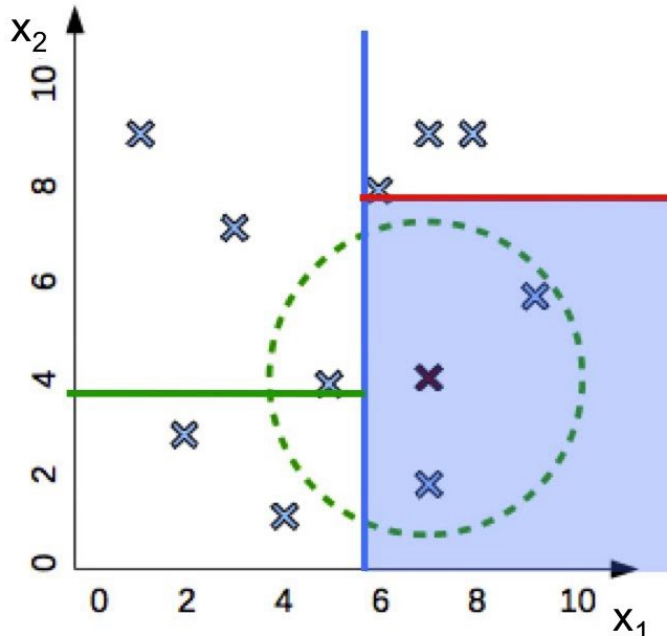
Table of contents

- Interesting Point Extraction (Part II)
- KNN to recognize MNIST data
  - MNIST datasets
  - KNN Algorithms
  - KD-Tree model
  - SKLearn Functions

Improvement over KNN: KD-Tree

- Building a K-D tree from training data:
  - pick random dimension, find median, split data, repeat
- Find NNs for new point (7,4)
  - find region containing (7,4)
  - compare to all points in region

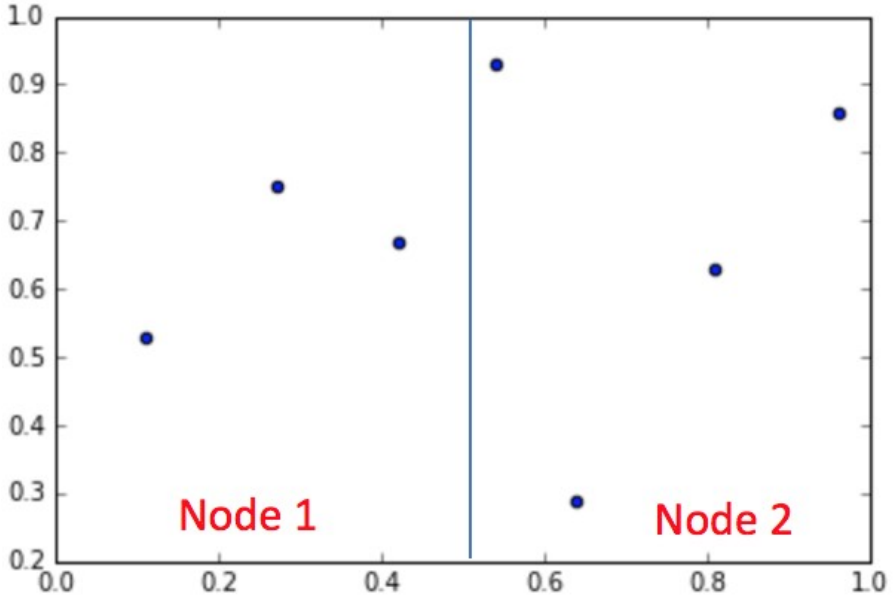


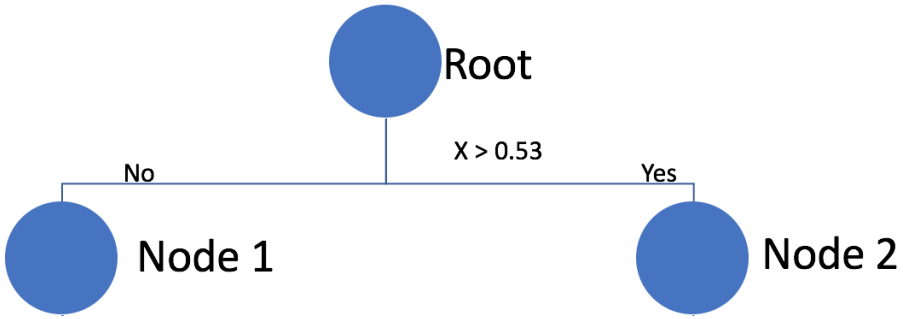


KD-Tree Example

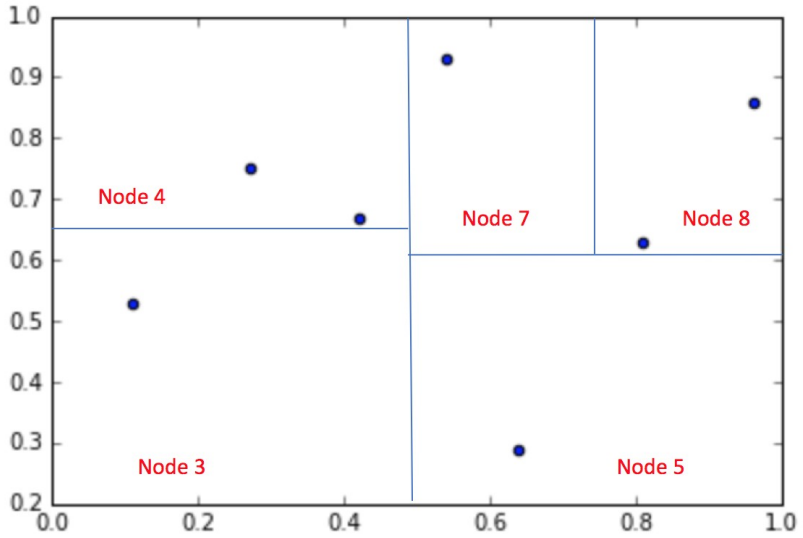
	X	Y
Data point 0	0.54	0.93
Data point 1	0.96	0.86
Data point 2	0.42	0.67
Data point 3	0.11	0.53
Data point 4	0.64	0.29
Data point 5	0.27	0.75
Data point 6	0.81	0.63

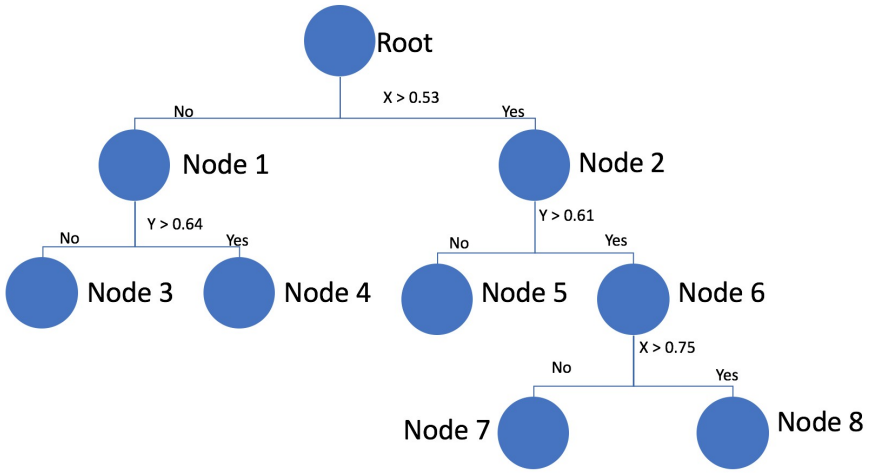
Split the Data into Two Groups



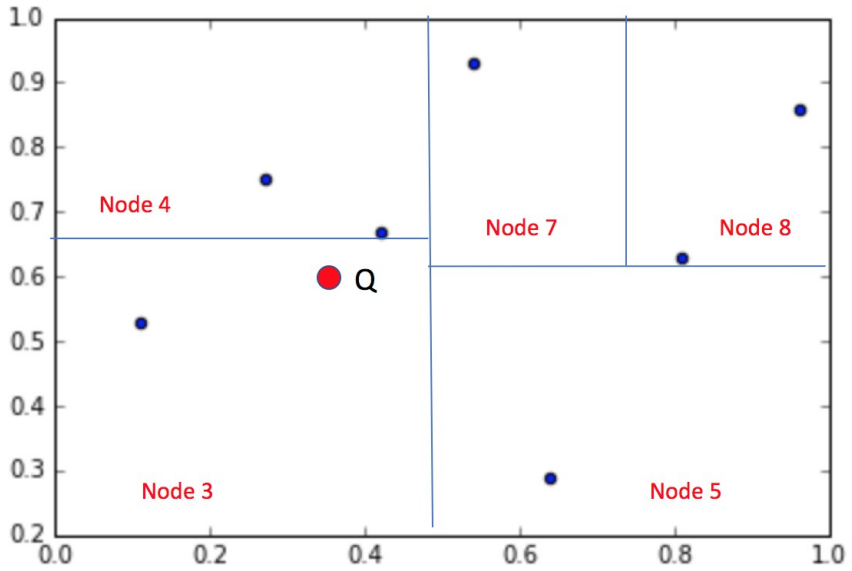


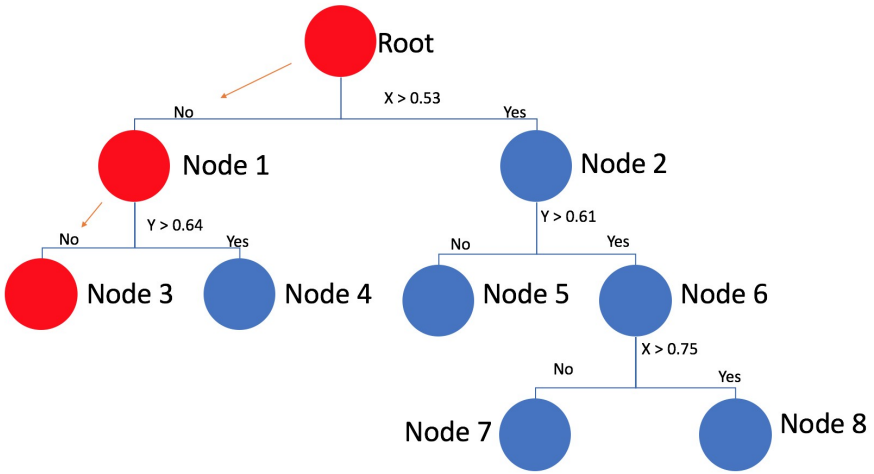
The Constructed KD-Tree



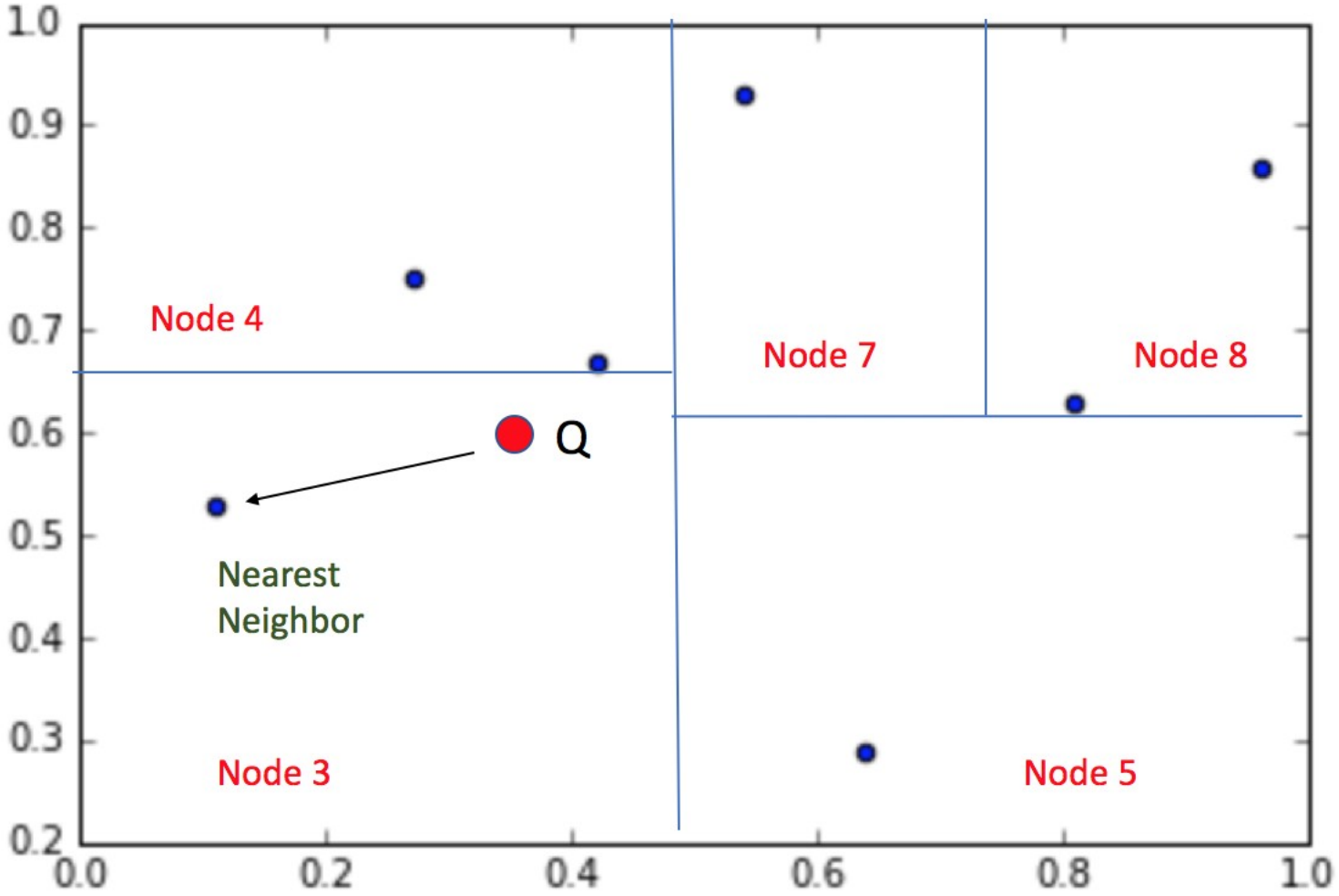


Query a KD-Tree



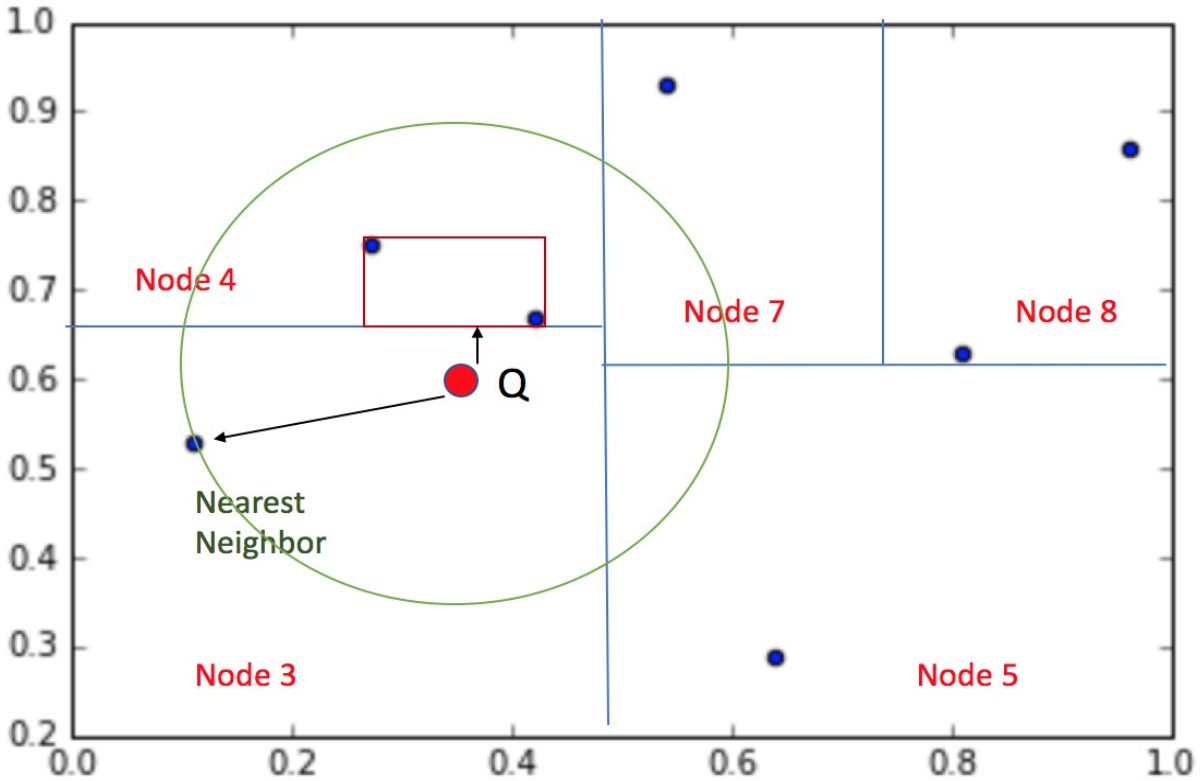


D = 1 and D = 2

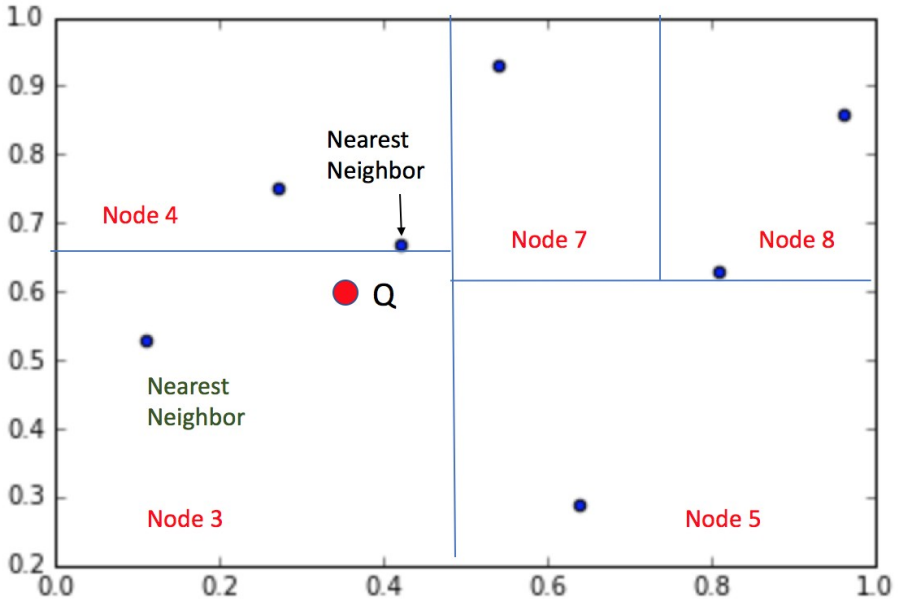

1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.0
0.2
0.4
0.6
0.8
1.0
Node 4
Node 7
Node 8
Q
Nearest
Neighbor
Node 3
Node 5

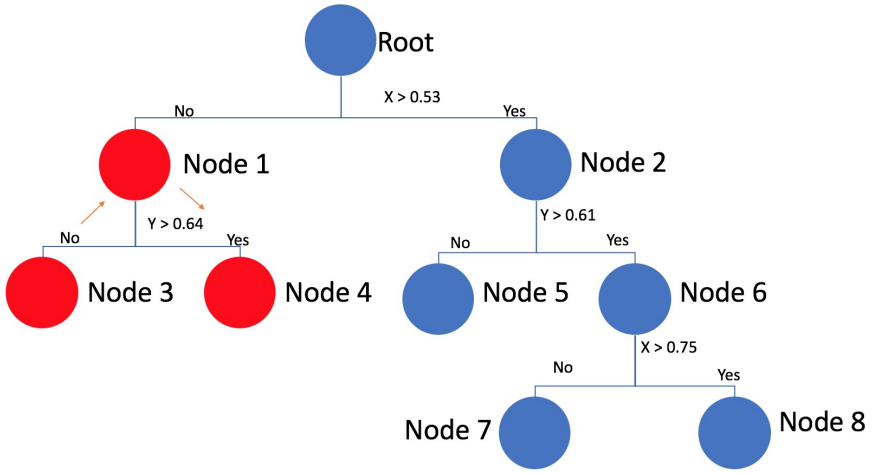
The Nearest Neighbour in Node 3

- Do we need to inspect all remaining data points in Node 1?
- We can check this by checking if the tightest box containing all the points of Node 4 is closer than the current near point or not.



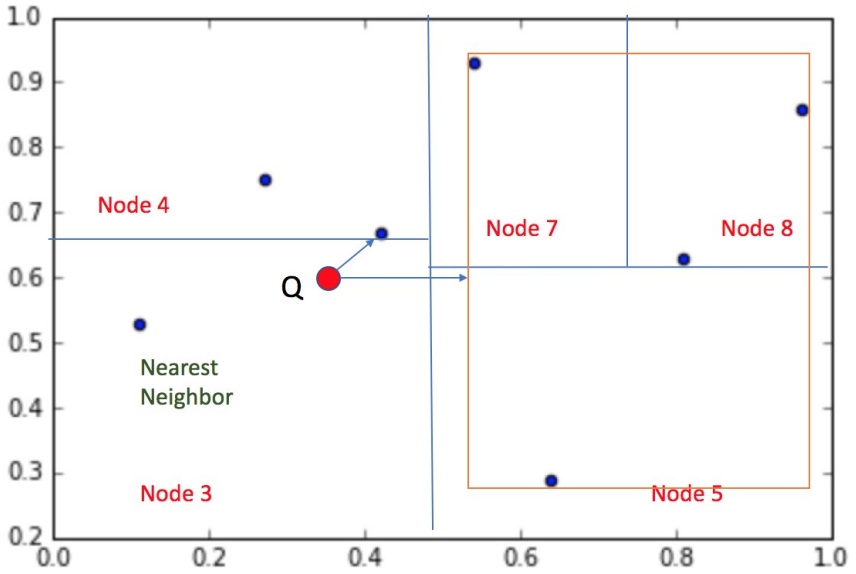
Traverse One Level Up to Node 1

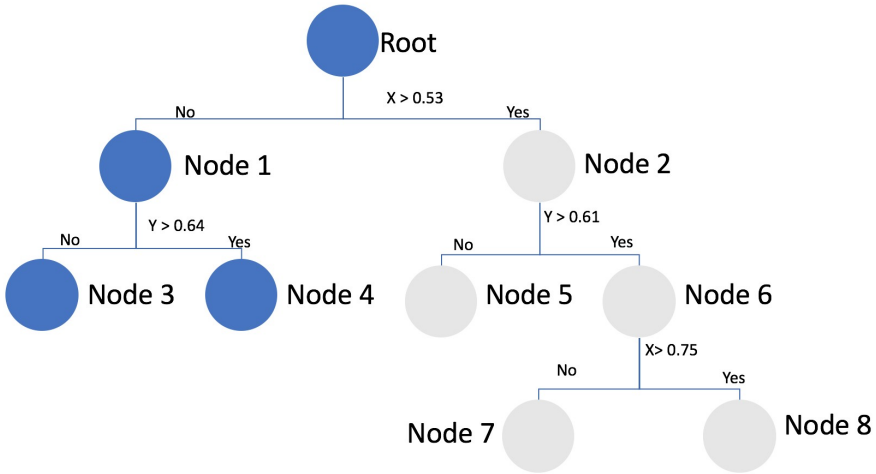




Traverse One Level Up to Node 1

- Do we need to inspect all remaining data points in Node 2?
- We can check this by checking if the tightest box containing all the points of Node 2 is closer than the current Near point or not.





The Nearest Neighbour Has Been Found

Since weve traversed the whole tree, we are done: data point marked is indeed the true nearest neighbour of the query.

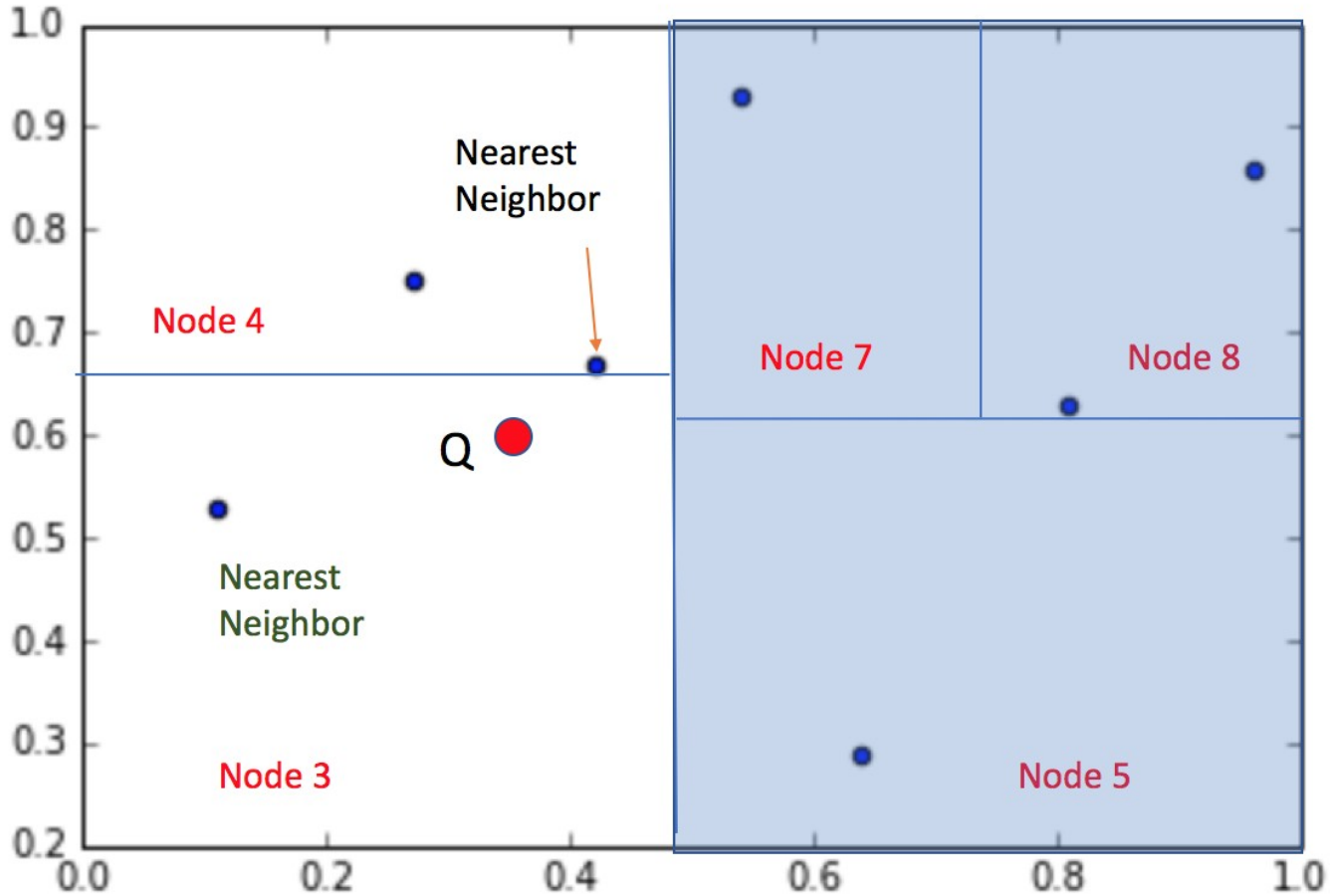


Table of contents

- Interesting Point Extraction (Part II)
- KNN to recognize MNIST data
  - MNIST datasets
  - KNN Algorithms
  - KD-Tree model
  - SKLearn KNN Functions

Sklearn.model_selection.train_test_split

Parameters:	***arrays : *sequence of indexables with same length / shape[0]*** Allowed inputs are lists, numpy arrays, scipy-sparse matrices or pandas dataframes. **test_size : *float, int or None, optional (default=0.25)*** If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the test split. If int, represents the absolute number of test samples. If None, the value is set to the complement of the train size. By default, the value is set to 0.25. The default will change in version 0.21. It will remain 0.25 only if `train_size` is unspecified, otherwise it will complement the specified `train_size`. **train_size : *float, int, or None, (default=None)*** If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the train split. If int, represents the absolute number of train samples. If None, the value is automatically set to the complement of the test size. **random_state : *int, RandomState instance or None, optional (default=None)*** If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random. **shuffle : *boolean, optional (default=True)*** Whether or not to shuffle the data before splitting. If shuffle=False then stratify must be None. **stratify : *array-like or None (default=None)*** If not None, data is split in a stratified fashion, using this as the class labels.
Returns:	**splitting : *list, length=2 * len(arrays)*** List containing train-test split of inputs. *New in version 0.16:* If the input is sparse, the output will be a `scipy.sparse.csr_matrix`. Else, output type is the same as the input type.

Parameters :

n_neighbors : ***int, optional (default = 5)***
Number of neighbors to use by default for `kneighbors` queries.
weights : ***str or callable, optional (default = 'uniform')***
weight function used in prediction. Possible values:

- 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
- 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
- [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

algorithm : ***{'auto', 'ball_tree', 'kd_tree', 'brute'}, optional***
Algorithm used to compute the nearest neighbors:

- 'ball_tree' will use `BallTree`
- 'kd_tree' will use `KDTree`
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit` method.

Note: fitting on sparse input will override the setting of this parameter, using brute force.
leaf_size : ***int, optional (default = 30)***

p : ***integer, optional (default = 2)***
Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.
metric : ***string or callable, default 'minkowski'***
metric_params : ***dict, optional (default = None)***
Additional keyword arguments for the metric function.
n_jobs : ***int or None, optional (default=None)***
The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See Glossary for more details.
Doesn't affect `fit` method.

sklearn.neighbors.KNeighborsClassifier (Methods)

Methods

fit(X, y)	Fit the model using X as training data and y as target values
get_params([deep])	Get parameters for this estimator.
kneighbors([X, n_neighbors, return_distance])	Finds the K-neighbors of a point.
kneighbors_graph([X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbors for points in X
predict(X)	Predict the class labels for the provided data
predict_proba(X)	Return probability estimates for the test data X.
score(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.