K Nearest Neighbors (KNN)

James G. Shanahan [1,2]

[1]Church and Duncan Group,

[2]School of Informatics, Computing and Engineering, Indiana University

EMAIL: James_DOT_Shanahan_AT_gmail_DOT_com

Outline

- **Introduction**
- **KNN**
  - Introduction
  - KNN
    - Regression
    - Classification
  - Efficient implementations:
    - k-d trees, parallelism
- **Image classification**
  - KNN for image classification in Python
  - Implementation tricks (SKLearn, pilot datasets)
  - Homegrown KNN; CIFAR-10 Kaggle Challenge
- **Hyperparameter selection via crossfold validation**
- **Summary**

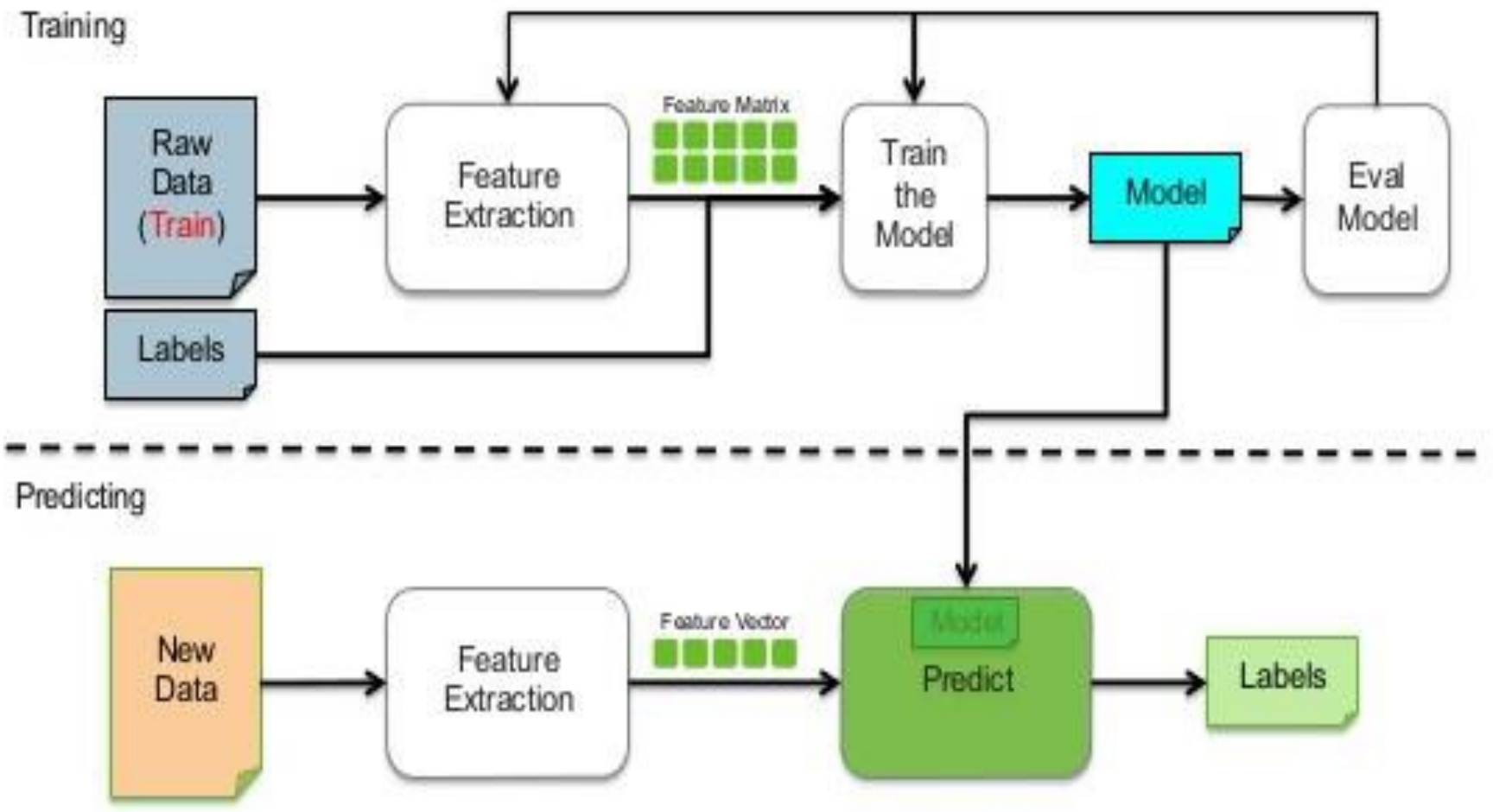
References

- **Python Machine Learning, Sebastian Raschka and Mirjalili Book (Chapter 3)**
- **Wikipedia**
  - https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm
- **https://www.analyticsvidhya.com/blog/2014/10/introduction-k-neighbours-algorithm-clustering/**

Outline

- **Introduction**
- **KNN**
  - Introduction
  - KNN
    - Regression
    - Classification
  - Efficient implementations:
    - k-d trees, parallelism
- **Image classification**
  - KNN for image classification
  - Implementation tricks (SKLearn, pilot datasets)
  - Homegrown KNN; CIFAR-10 Kaggle Challenge
- **Hyperparameter selection via crossfold validation**
- **Summary**

Machine Learning Pipeline

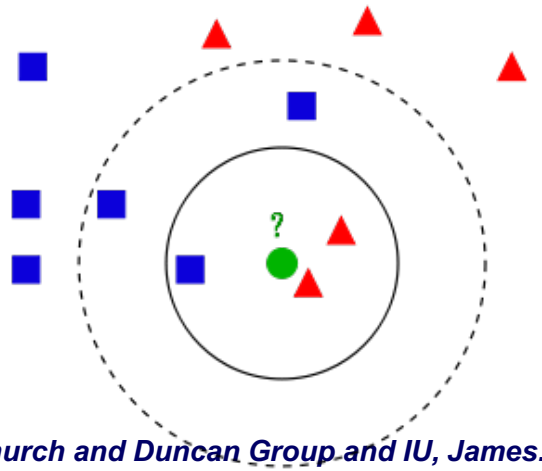


KNN: a non-parametric method

- In machine learning, the k-nearest neighbors algorithm (k-NN) is a non-parametric method used for classification and regression.
- In both cases, the input consists of the *k* closest training examples in the feature space.
- Case-based or instance based

KNN Algo for Classification

- The training examples are vectors in a multidimensional feature space, each with a class label.
- The training phase of the algorithm consists only of storing the feature vectors and class labels of the training samples.
- In the classification phase, *k* is a user-defined constant, and an unlabeled vector (a query or test point) is classified by assigning the label which is most frequent among the *k* training samples nearest to that query point.



Outline

- **Introduction**
- **KNN**
  - Introduction
  - KNN
    - Regression
    - Classification
  - Efficient implementations:
    - k-d trees, parallelism
- **Image classification**
  - KNN for image classification
- **Hyperparameter selection via crossfold validation**
- **Summary**

Machine Learning: Regression

Machine Learning (ML):"*a computer program that improves its performance at some task through experience*" [Mitchell 1997]

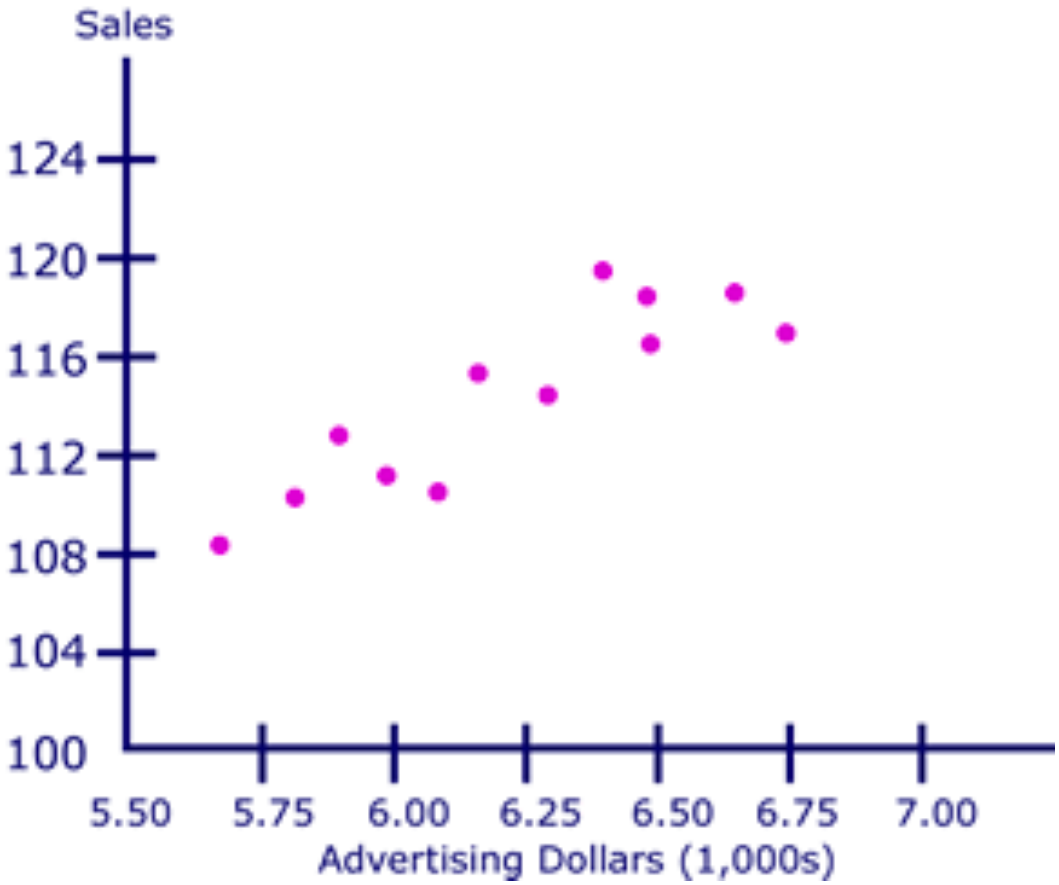
GIVEN: Input data is a table of attribute values and associated class values (in the case of supervised learning)
GOAL: Approximate $f(x_1,\ldots,x_n)$->y

Mimimize $$MSE = \frac{1}{n}\sum \text{Residual}^i = \frac{1}{n}\sum \left(WX^i - y^i\right)^2$$

Y is real valued

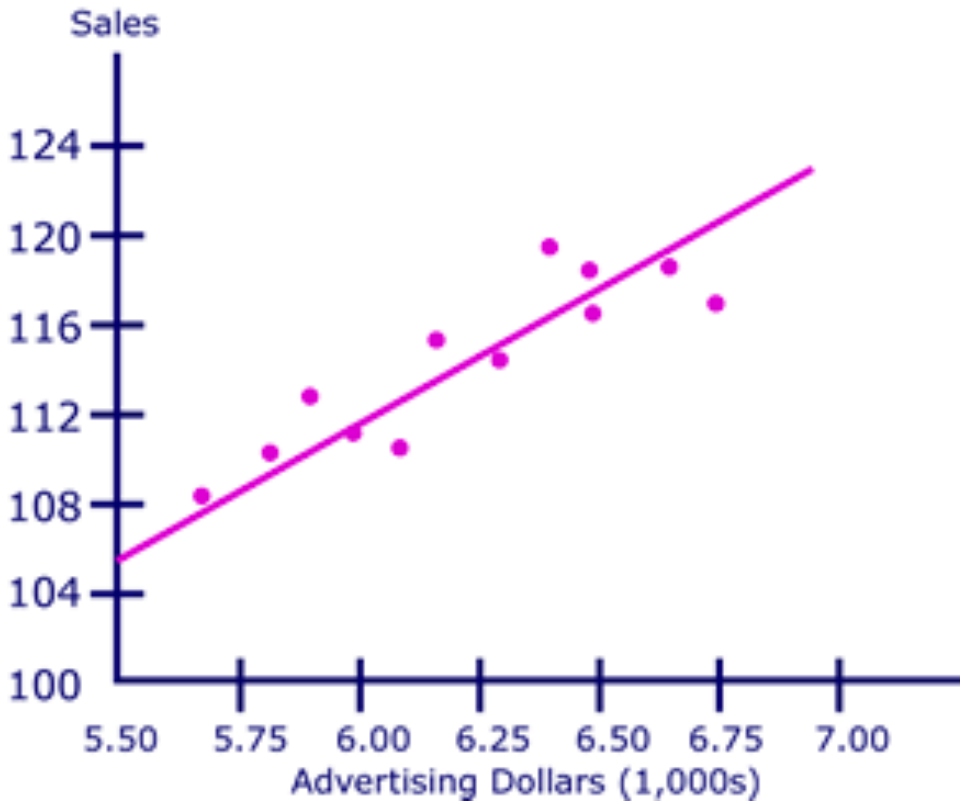
Instance\Attr	x_1	x_2	…	x_n	y
1	3	0	..	7	73
2					76
…	…	…	…	…	…
L (aka m)	0	4	...	8	97

Training data

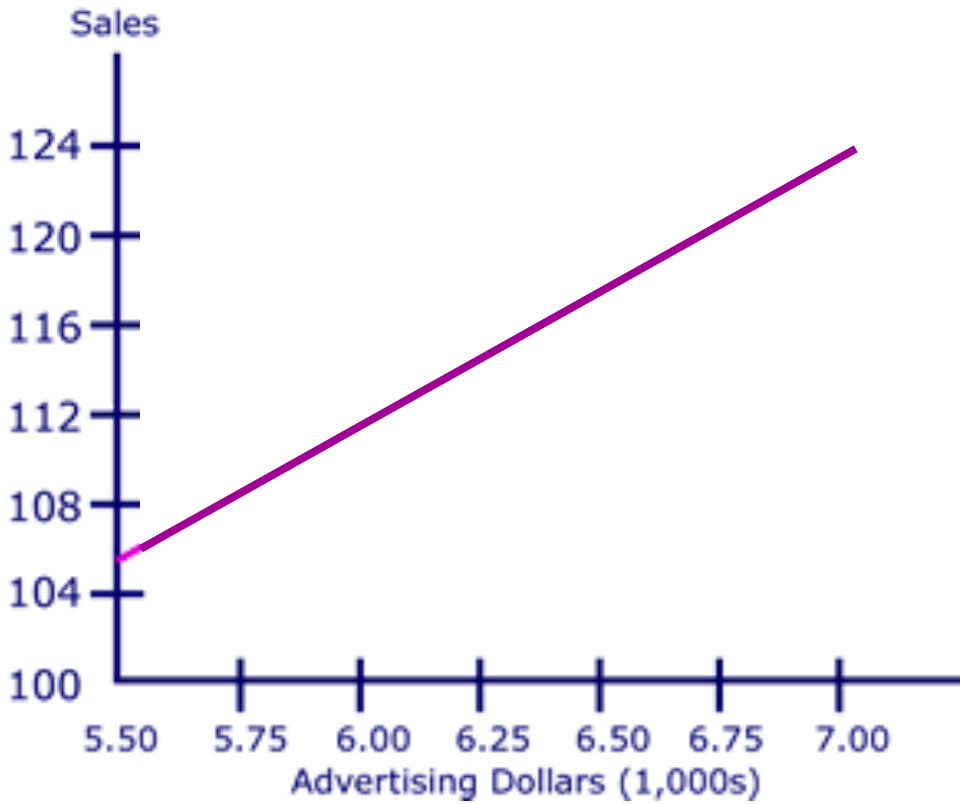


http://ci.columbia.edu/ci/premba_test/c0331/s7/s7_6.html

Learn a regression model



Deploy a regression model

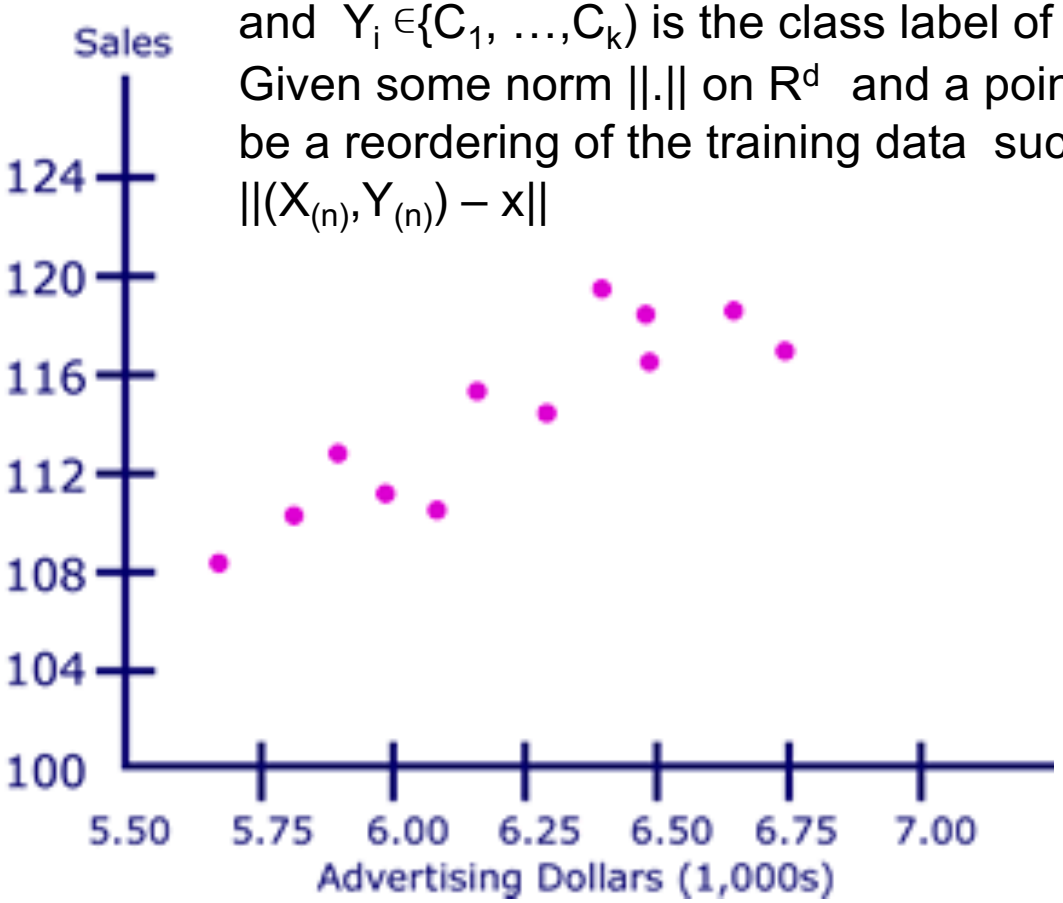


y = mx + b

Replace date with
Slope and intercept

Deploy a KNN

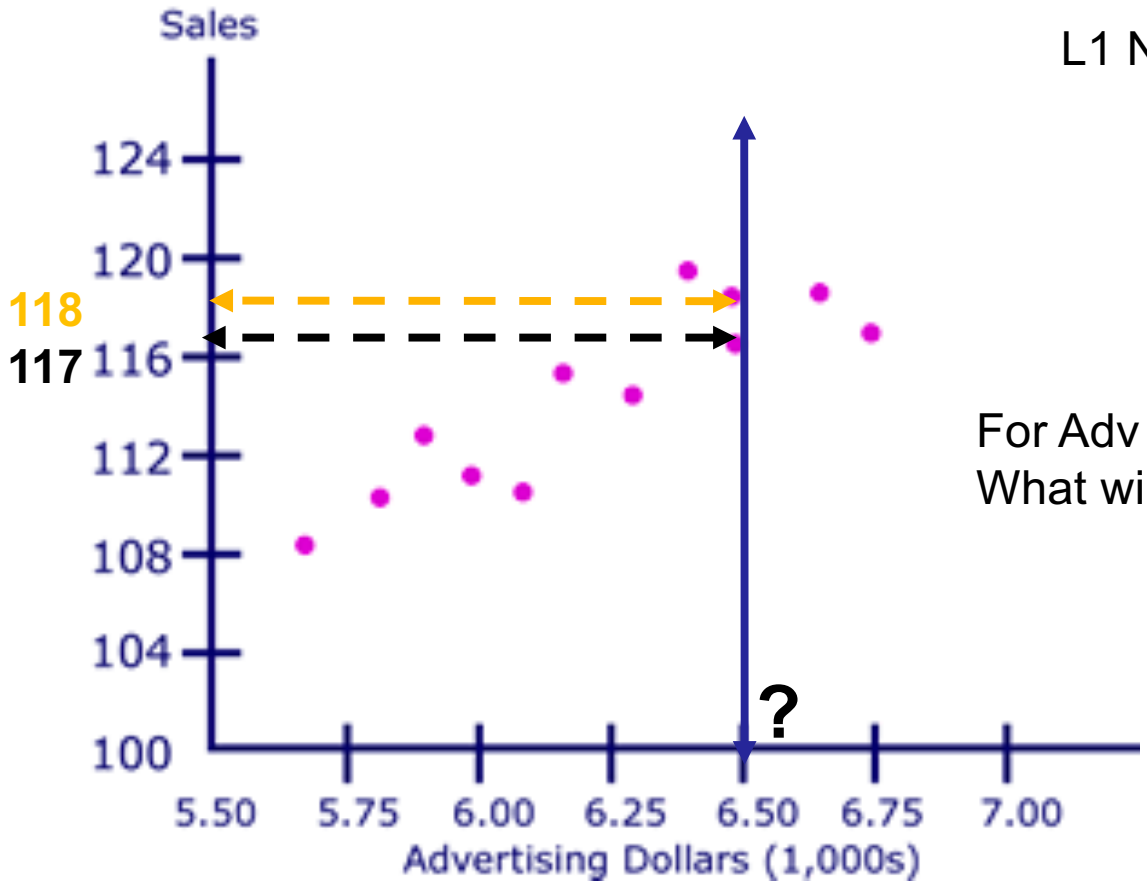
Suppose we have pairs $(X_1,Y_1), \ldots, (X_n,Y_n)\}$, where X_i takes values in R^d, and $Y_i \in \{C_1, \ldots, C_k)$ is the class label of X_i.
Given some norm $||.||$ on R^d and a point $x \in R^d$, let $(X_{(1)},Y_{(1)}), \ldots, (X_{(n)},Y_{(n)})$ be a reordering of the training data such that $||(X_{(1)},Y_{(1)}) - x|| \leq, \ldots, \leq ||(X_{(n)},Y_{(n)}) - x||$



For Adv = $65,000
What will be the value of sales

http://ci.columbia.edu/ci/premba_test/c0331/s7/s7_6.html

Deploy a KNN



L1 Norm: $|Adv_i - Adv_?|$

For Adv = $65,000
What will be the value of sales

http://ci.columbia.edu/ci/premba_test/c0331/s7/s7_6.html

NN Regression (with K =??)

Distance functions

Euclidean $\sqrt{\sum_{i=1}^{k}(x_i - y_i)^2}$

Manhattan $\sum_{i=1}^{k}|x_i - y_i|$

Minkowski $\left(\sum_{i=1}^{k}(|x_i - y_i|)^q\right)^{1/q}$

NN For regression with K=??

Consider the following data concerning House Price Index or HPI. Age and Loan are two numerical variables (predictors) and HPI is the numerical target.

Age	Loan	House Price Index	Distance	
25	$40,000	135	102000	
35	$60,000	256	82000	
45	$80,000	231	62000	
20	$20,000	267	122000	
35	$120,000	139	22000	2
52	$18,000	150	124000	
23	$95,000	127	47000	
40	$62,000	216	80000	
60	$100,000	139	42000	3
48	$220,000	250	78000	
33	$150,000	264	8000	1
48	**$142,000**	**?**		

$$D = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$$

K=3 For regression

Consider the following data concerning House Price Index or HPI. Age and Loan are two numerical variables (predictors) and HPI is the numerical target.

Age	Loan	House Price Index	Distance	
25	$40,000	135	102000	
35	$60,000	256	82000	
45	$80,000	231	62000	
20	$20,000	267	122000	
35	$120,000	139	22000	2
52	$18,000	150	124000	
23	$95,000	127	47000	
40	$62,000	216	80000	
60	$100,000	139	42000	3
48	$220,000	250	78000	
33	$150,000	264	8000	1
48	**$142,000**	**?**		

We can now use the training set to classify an unknown case (Age=33 and Loan=$150,000) using Euclidean distance K=1 then the nearest neighbor is the last case in the training set with HPI=264.

$$D = \text{Sqrt}[(48-33)^2 + (142000-150000)^2] = 8000.01 \;>>\; HPI = 264$$

By having K=3, the prediction for HPI is equal to the average of HPI for the top three neighbors.

$$HPI = (264+139+139)/3 = 180.7$$

Standardized Distance

One major drawback in calculating distance measures directly from the training set is in the case where variables have different measurement scales or there is a mixture of numerical and categorical variables. For example, if one variable is based on annual income in dollars, and the other is based on age in years then income will have a much higher influence on the distance calculated. One solution is to standardize the training set as shown below.

Age	Loan	House Price Index	Distance
0.125	0.11	135	0.7652
0.375	0.21	256	0.5200
0.625	0.31	231	0.3160
0	0.01	267	0.9245
0.375	0.50	139	0.3428
0.8	0.00	150	0.6220
0.075	0.38	127	0.6669
0.5	0.22	216	0.4437
1	0.41	139	0.3650
0.7	1.00	250	0.3861
0.325	0.65	264	0.3771
0.7	**0.61**	**?**	

$$X_s = \frac{X - Min}{Max - Min}$$

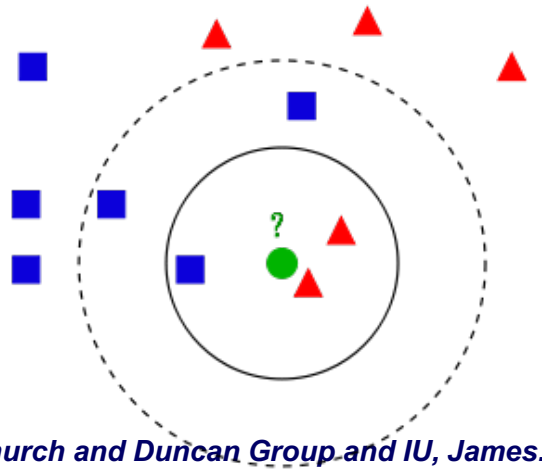
As mentioned in KNN Classification using the standardized distance on the same training set, the unknown case returned a different neighbor which is not a good sign of robustness.

Outline

- **Introduction**
- **KNN**
  - Introduction
  - KNN
    - Regression
    - Classification
  - Efficient implementations:
    - k-d trees, parallelism
- **Image classification**
  - KNN for image classification
- **Hyperparameter selection via crossfold validation**
- **Summary**

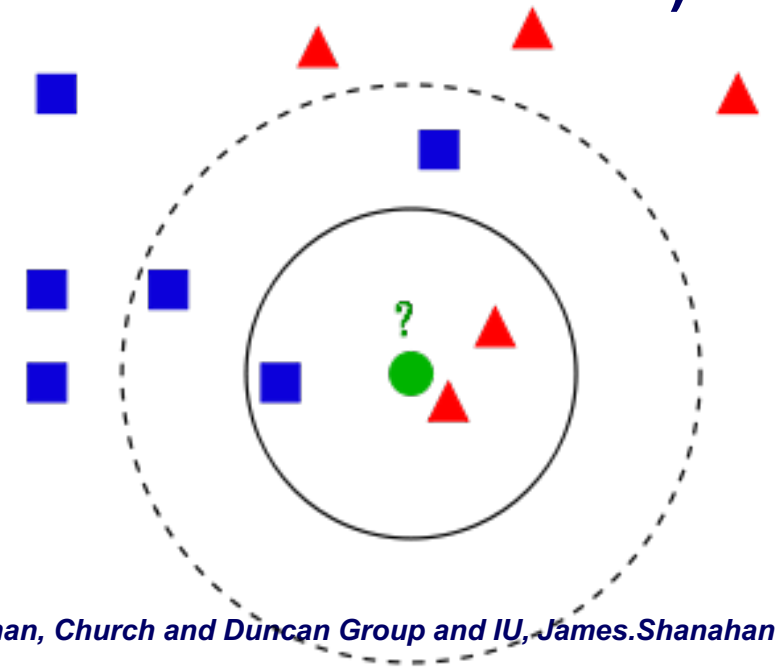
KNN Algo for Classification

- The training examples are vectors in a multidimensional feature space, each with a class label.
- The training phase of the algorithm consists only of storing the feature vectors and class labels of the training samples.
- In the classification phase, *k* is a user-defined constant, and an unlabeled vector (a query or test point) is classified by assigning the label which is most frequent among the *k* training samples nearest to that query point.



KNN Classification example

- The test sample (green circle) should be classified either to the Class 1 (blue squares) or to the Class2 (red triangles).
- If k = 3 (solid line circle) it is assigned to Class 2 because there are 2 triangles and only 1 square inside the inner circle.
- If k = 5 (dashed line circle) it is assigned to the first class (3 squares vs. 2 triangles inside the outer circle).



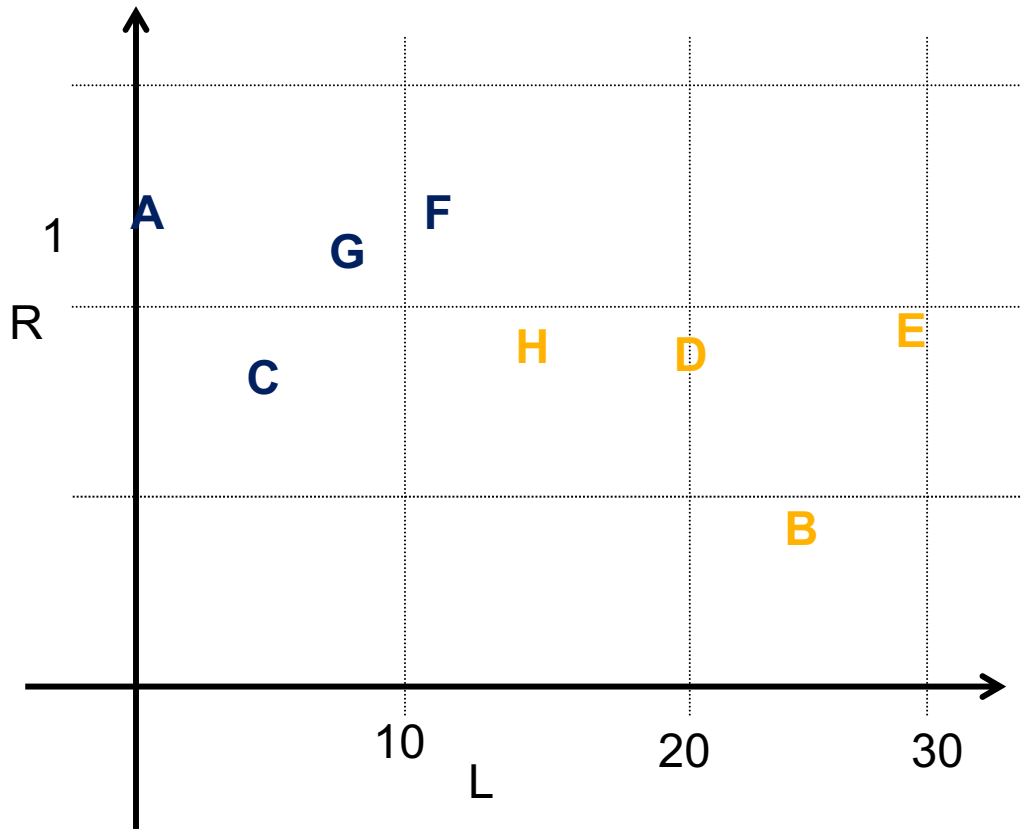
Nearest Neighbor Example

- **Credit Rating:**
  - Classifier: Good / Poor
  - Features:
    - L = # late payments/yr;
    - R = Income/Expenses

Name	L	R	G/P
A	0	1.2	G
B	25	0.4	P
C	5	0.7	G
D	20	0.8	P
E	30	0.85	P
F	11	1.2	G
G	7	1.15	G
H	15	0.8	P

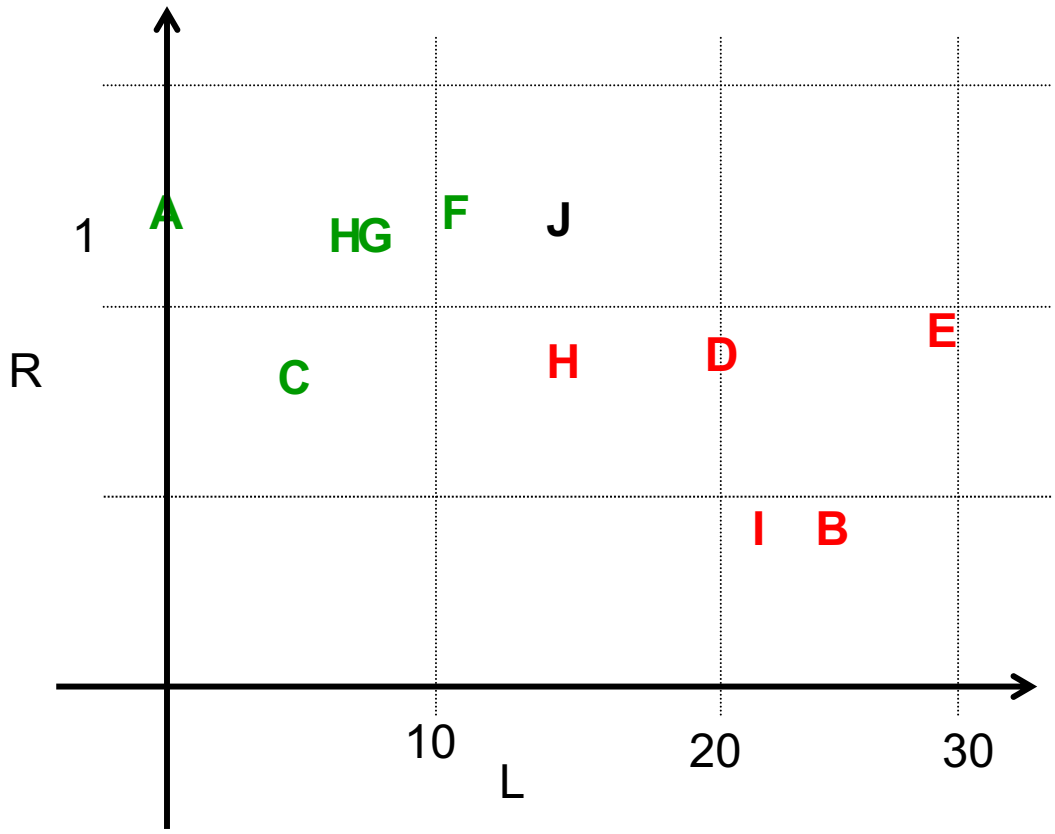
Nearest Neighbor Example

Name	L	R	G/P
A	0	1.2	G
B	25	0.4	P
C	5	0.7	G
D	20	0.8	P
E	30	0.85	P
F	11	1.2	G
G	7	1.15	G
H	15	0.8	P



Nearest Neighbor Example

Name	L	R	G/P
H	6	1.15	G
I	22	0.45	P
J	15	1.2	??



Distance Measure:

Sqrt $((L1-L2)^2 + [sqrt(10)*(R1-R2)]^2)$

- Scaled distance

Plotting the decision boundaries Voronoi Tesselation

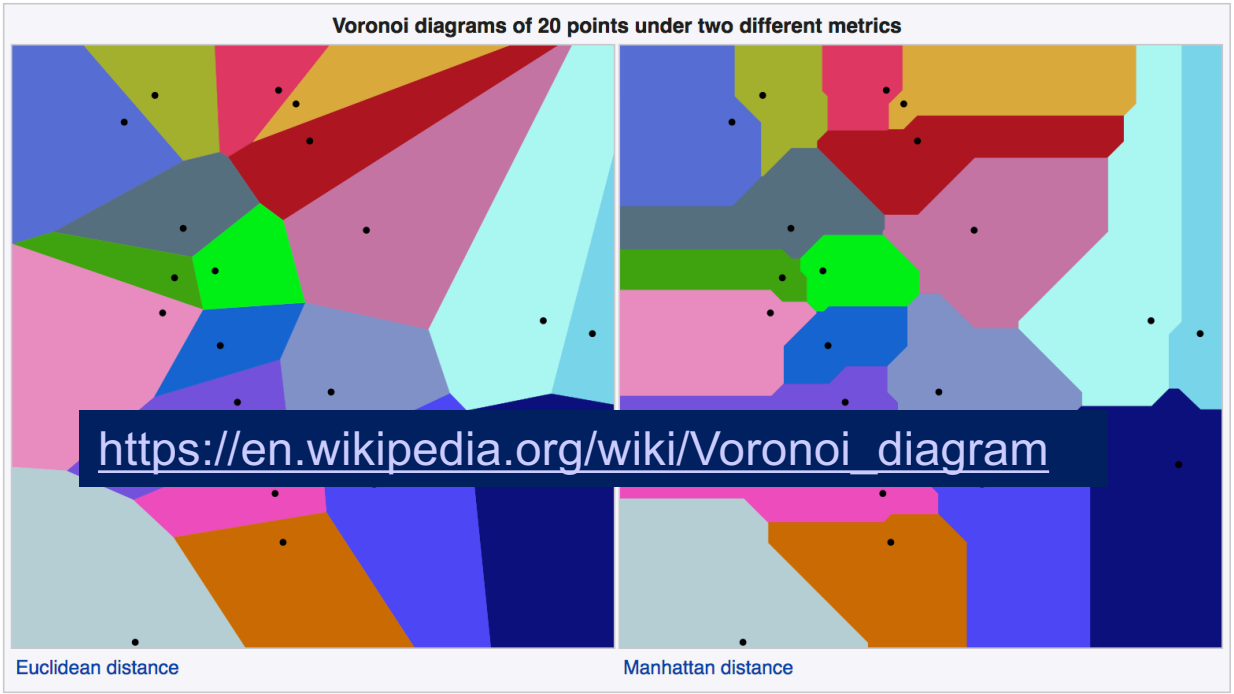
NN allows us to partition the feature space into cells consisting of all points closer to a given training point x

All points in such cells are labeled by the class of the training point. This partitioning is called a **Voronoi Tesselation**

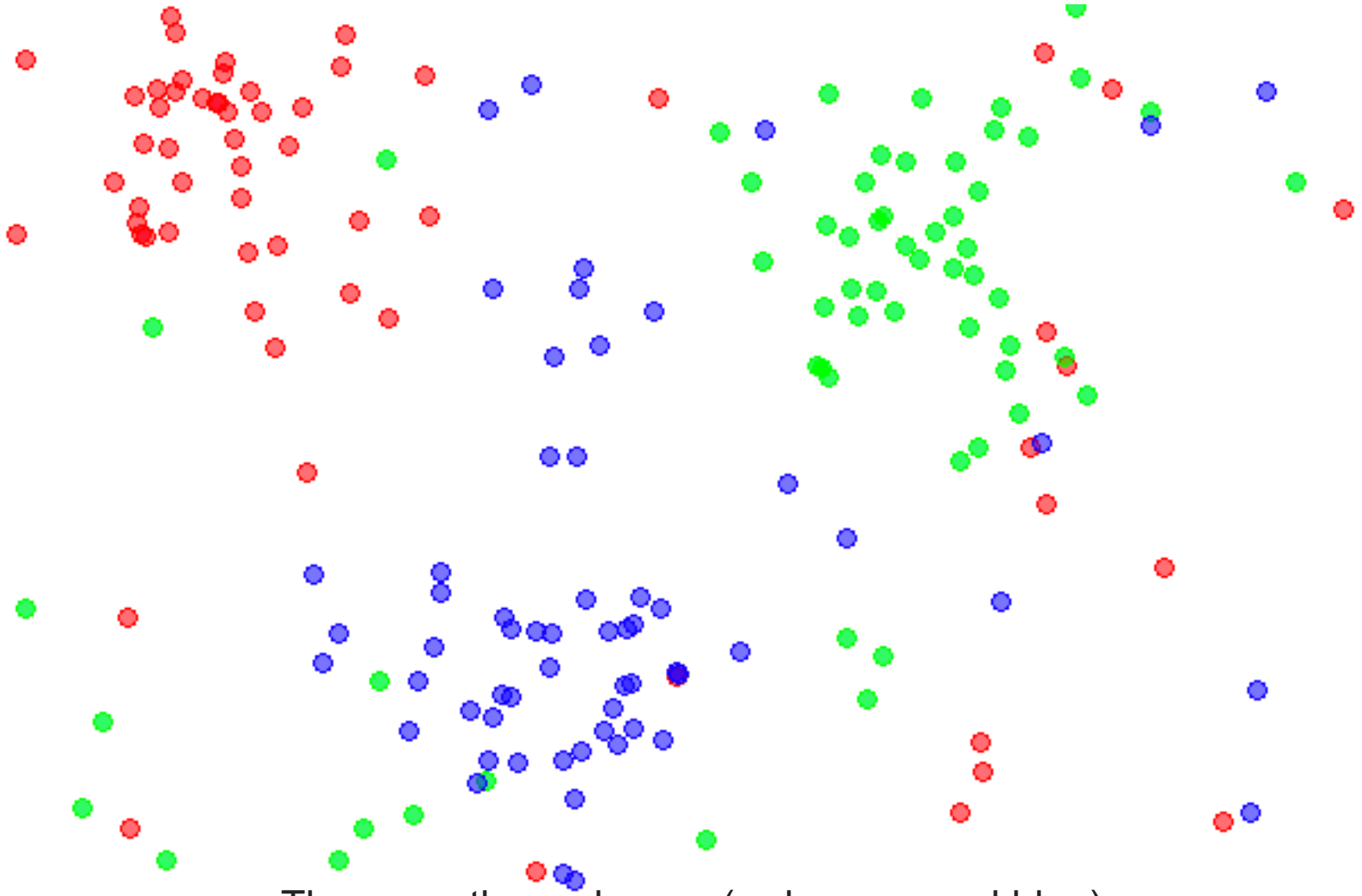
For most cities, the distance between points can be measured using the familiar Euclidean distance:

$\ell_2 = d\left[(a_1, a_2), (b_1, b_2)\right] = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2}$ or the Manhattan distance:

$d\left[(a_1, a_2), (b_1, b_2)\right] = |a_1 - b_1| + |a_2 - b_2|$. The corresponding Voronoi diagrams look different for different distance metrics.

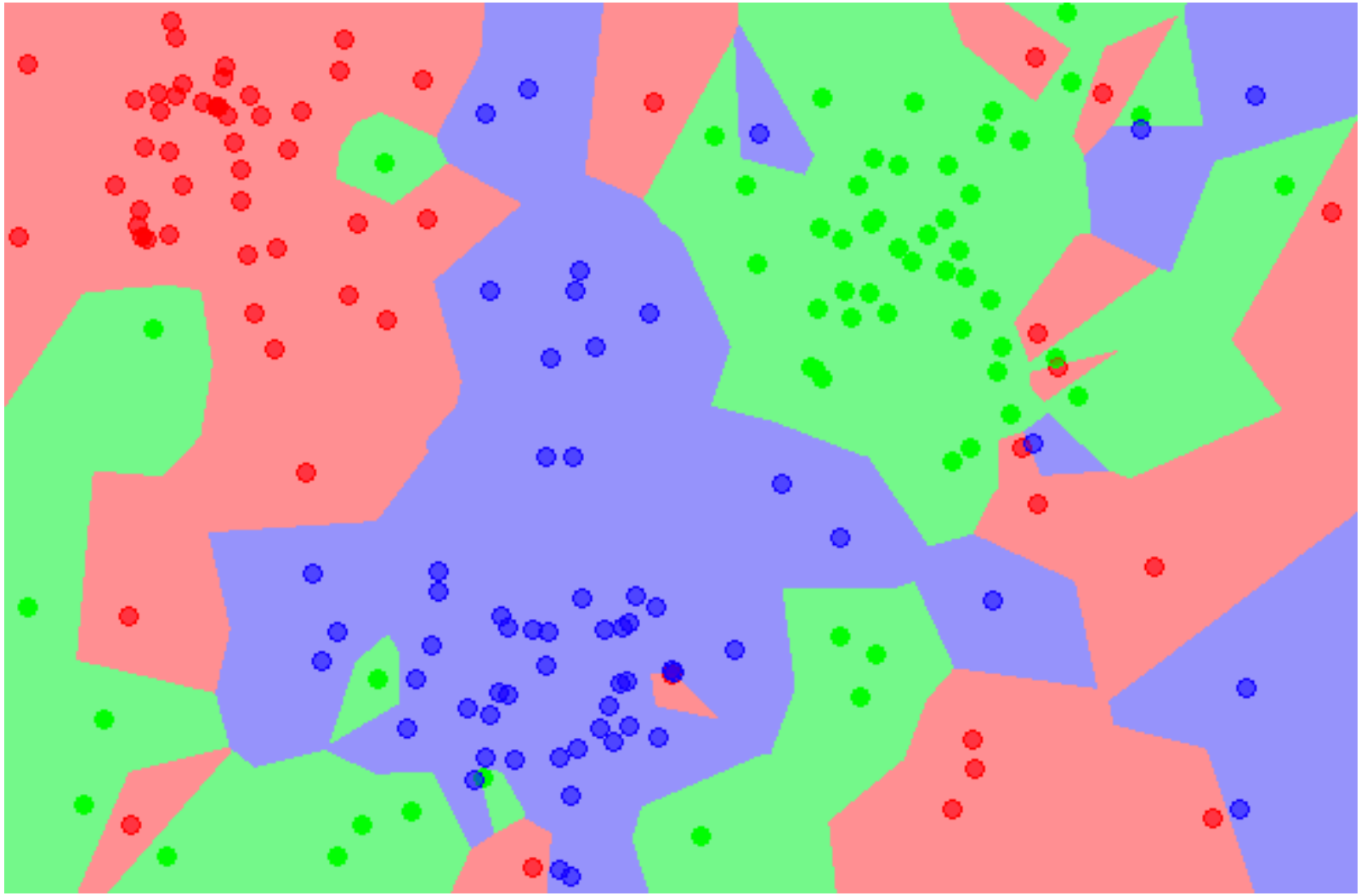


3 Class dataset

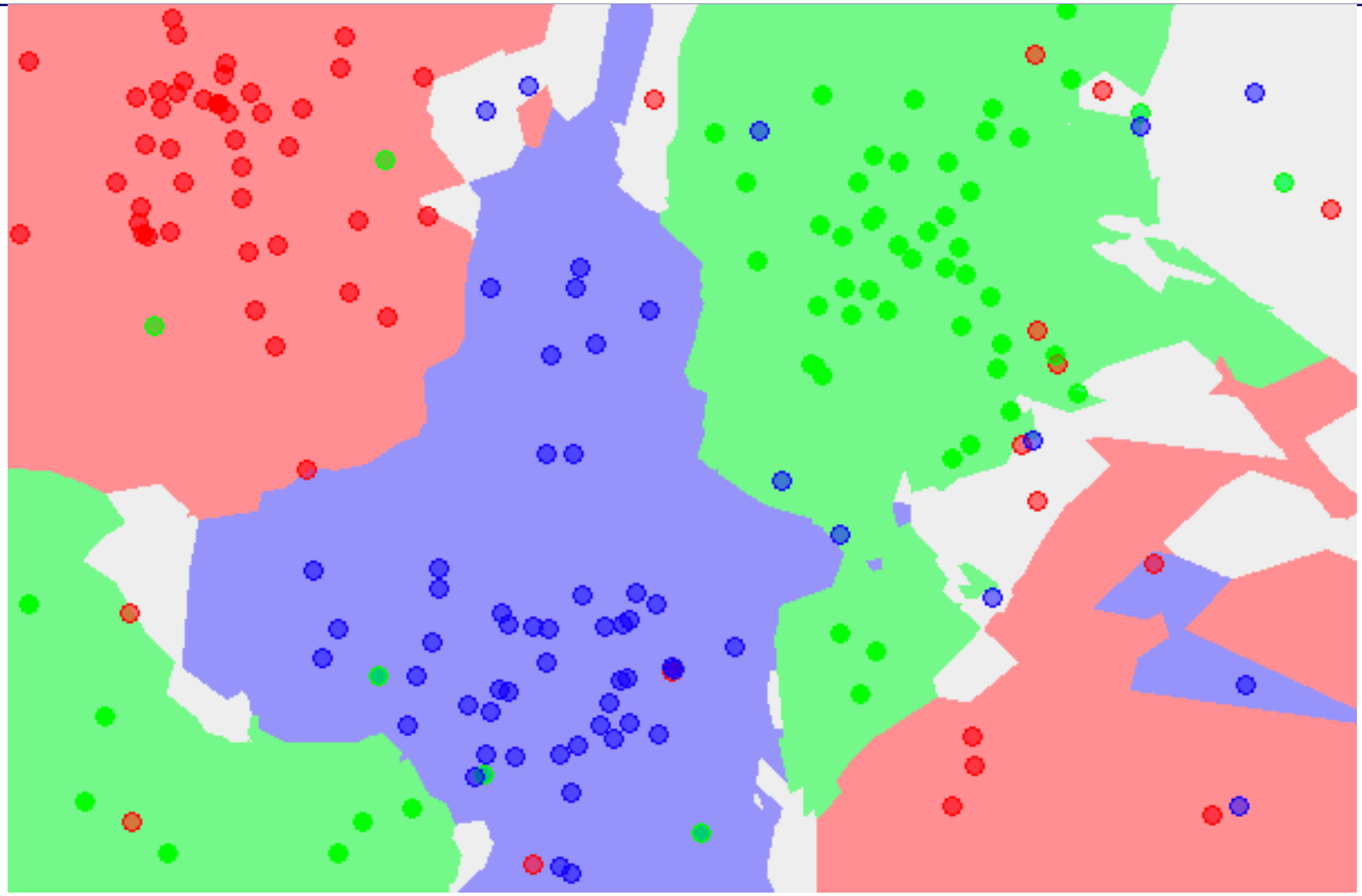


There are three classes (red, green and blue).
Initially there are 60 points in each class.

1NN classification map: *Voronoi* Tesselation



5NN classification map



White areas correspond to the unclassified regions, where 5NN voting is tied (for example, if there are two green, two red and one blue points among 5 nearest neighbors).

The 1-nearest neighbor classifier

Suppose we have pairs $(X_1,Y_1), \ldots ,(X_n,Y_n)\}$, where X_i takes values in R^d, and $Y_i \in \{C_1, \ldots,C_k)$ is the class label of X_i.

Given some norm $||.||$ on R^d and a point $x \in R^d$, let $(X_{(1)},Y_{(1)}), \ldots, (X_{(n)},Y_{(n)})$ be a reordering of the training data such that:

- $||(X_{(1)},Y_{(1)}) - x|| \leq, \ldots, \leq ||(X_{(n)},Y_{(n)}) - x||$

The most intuitive nearest neighbour type classifier is the one nearest neighbour classifier that assigns a point x to the class of its closest neighbour in the feature space, that is $C_n^{1nn}(x) = Y_{(1)}$.

As the size of training data set approaches infinity, the one nearest neighbour classifier guarantees an error rate of no worse than twice the Bayes error rate (the minimum achievable error rate given the distribution of the data).

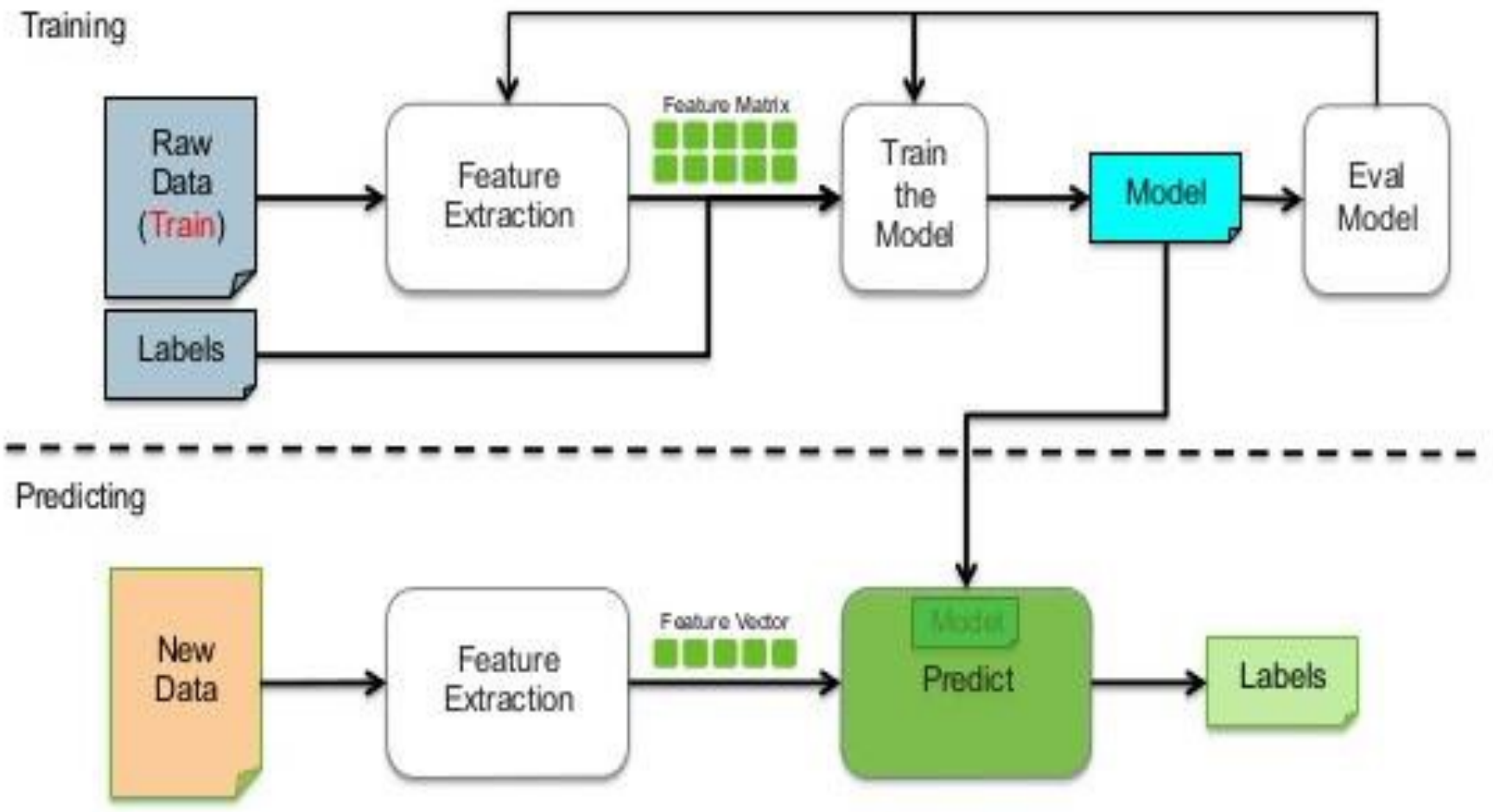
Complexity & Generalization

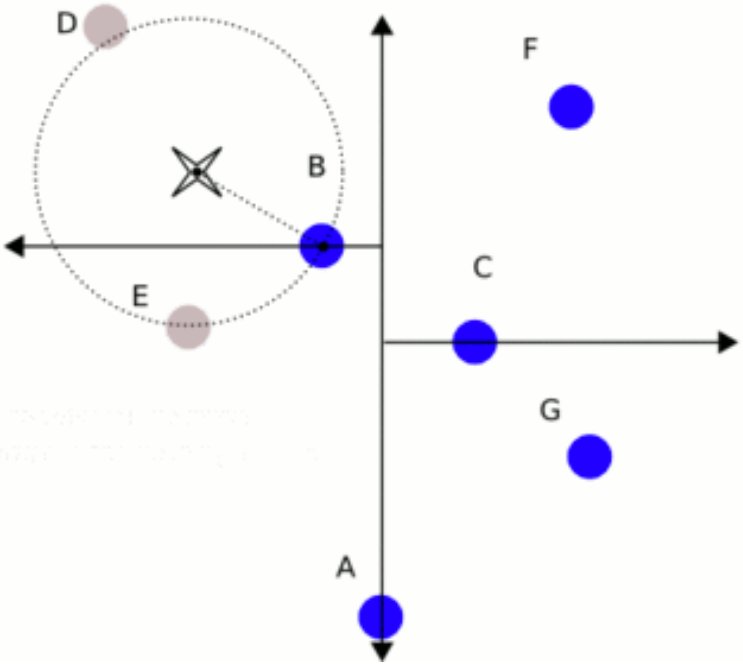
- **Goal: Predict values accurately on new inputs**
- **Problem:**
  - Train on sample data
  - Can make arbitrarily complex model to fit
  - BUT, will probably perform badly on NEW data
- **Strategy:**
  - Limit complexity of model (e.g. degree of equ'n)
  - Split training and validation sets
    - Hold out data to check for overfitting

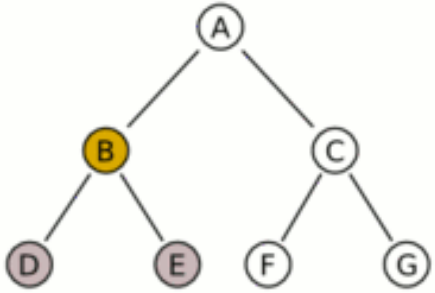
Outline

- **Introduction**
- **KNN**
  - Introduction
  - KNN
    - Regression
    - Classification
  - Efficient implementations:
    - k-d trees, parallelism
- **Image classification**
- **Hyperparameter selection crossfold validation**
- **Summary**

Machine Learning Pipeline







D & E Discarded as B
(already visited) is closer.
B is the best estimate for B's sub-branch
Proceed back to parent node

Efficient Implementations

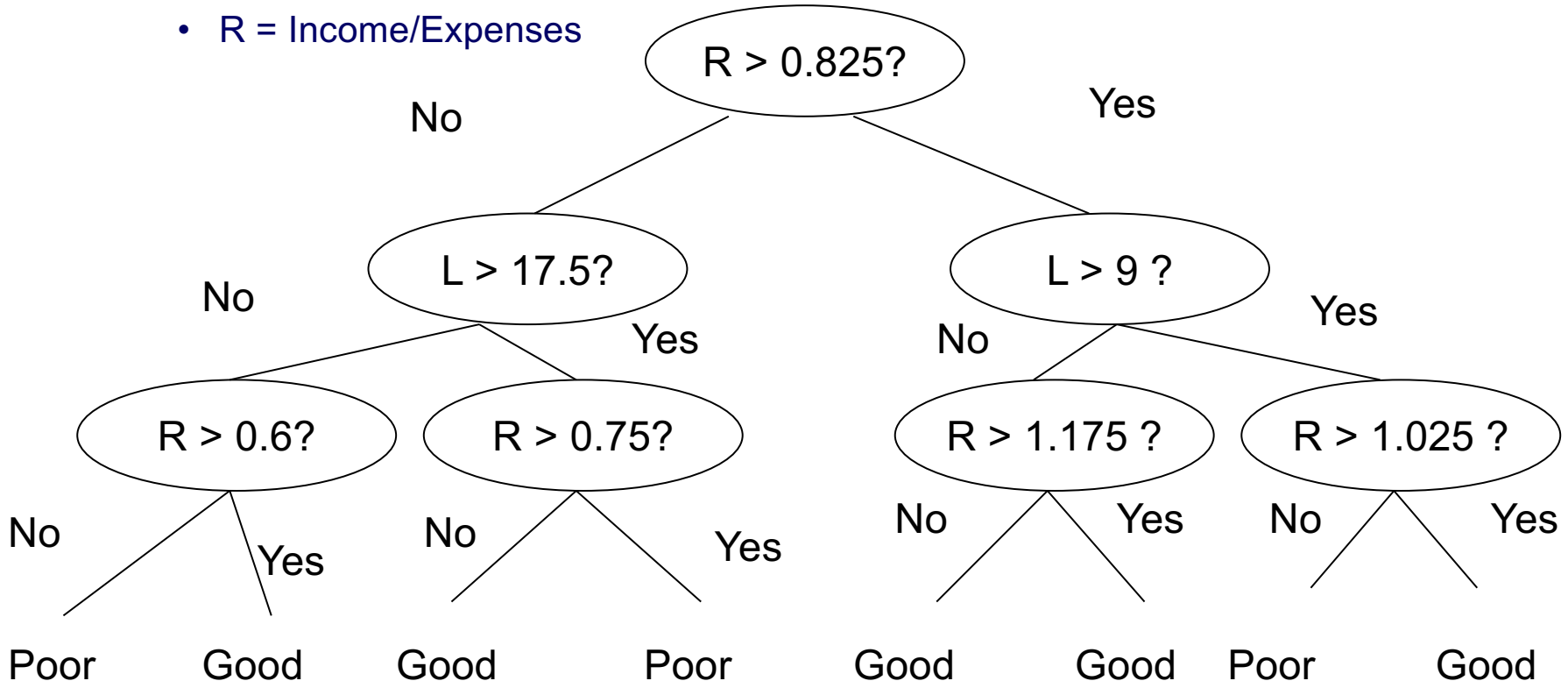
- **Classification cost:**
  - Find nearest neighbor: O(n)
    - Compute distance between unknown and all instances
    - Compare distances
  - Problematic for large data sets
- **Alternative:**
  - Use binary search to reduce to O(log n)

Efficient Implementation: K-D Trees

- **K-d Trees: where k is the depth of the tree**
  - A k-d tree, or k-dimensional tree, is a data structure used in computer science for organizing some number of points in a space with k dimensions.
  - It is a binary search tree with other constraints imposed on it.
  - K-d trees are very useful for range and nearest neighbor searches.
- **Divide instances into sets based on features**
  - Binary branching: E.g. > value
  - 2^d leaves with d split path = n
    - d= O(log n)
  - To split cases into sets,
    - If there is one element in the set, stop
    - Otherwise pick a feature to split on
      - Find average position of two middle objects on that dimension
        - » Split remaining objects based on average position
        - » Recursively split subsets

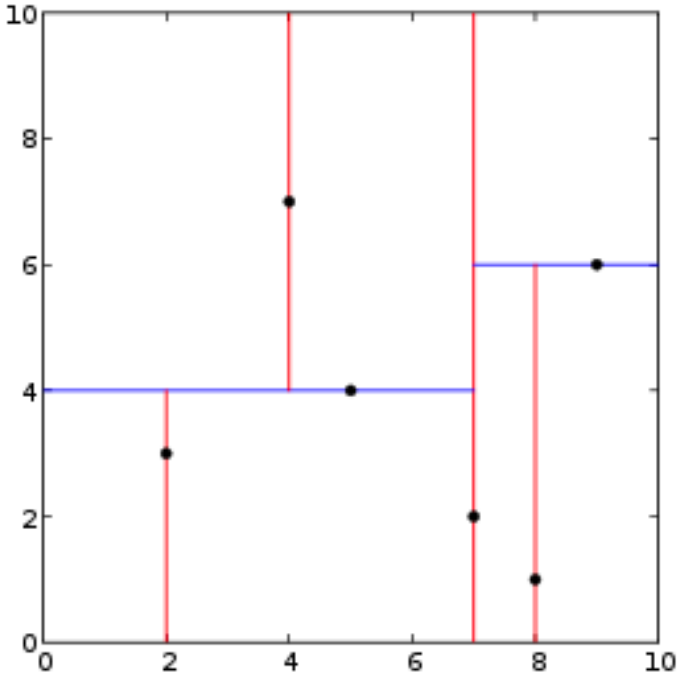
K-D Trees: Classification

- **Credit Rating:**
  - Classifier: Good / Poor
  - Features:
    - L = # late payments/yr;
    - R = Income/Expenses

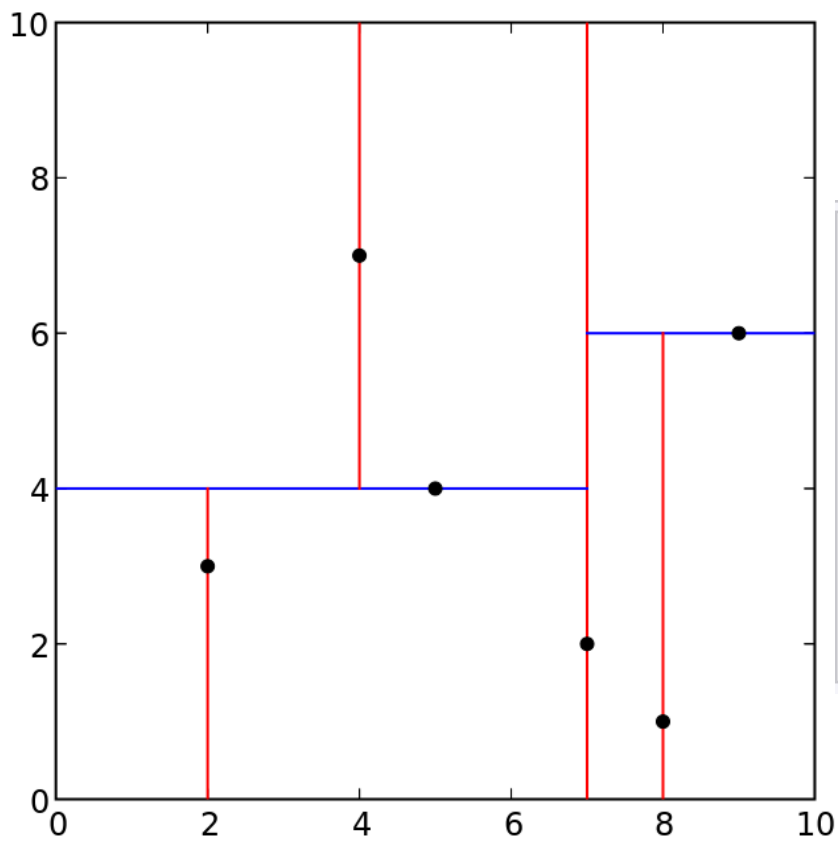


For our purposes we will generally only be dealing with point clouds in three dimensions, so all of our k-d trees will be three-dimensional. Each level of a k-d tree splits all children along a specific dimension, using a hyperplane that is perpendicular to the corresponding axis.

At the root of the tree all children will be split based on the first dimension (i.e. if the first dimension coordinate is less than the root it will be in the left-sub tree and if it is greater than the root it will obviously be in the right sub-tree). Each level down in the tree divides on the next dimension, returning to the first dimension once all others have been exhausted. They most efficient way to build a k-d tree is to use a partition method like the one Quick Sort uses to place the median point at the root and everything with a smaller one dimensional value to the left and larger to the right. You then repeat this procedure on both the left and right sub-trees until the last trees that you are to partition are only composed of one element.

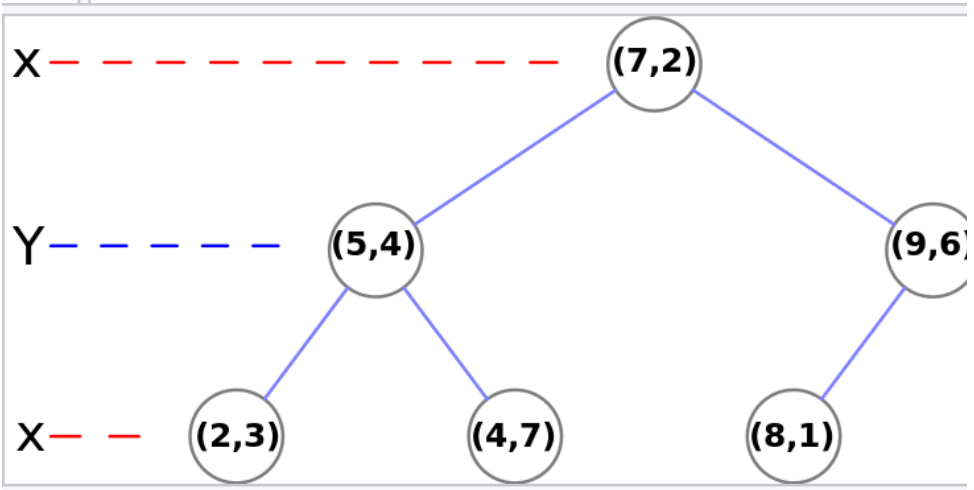


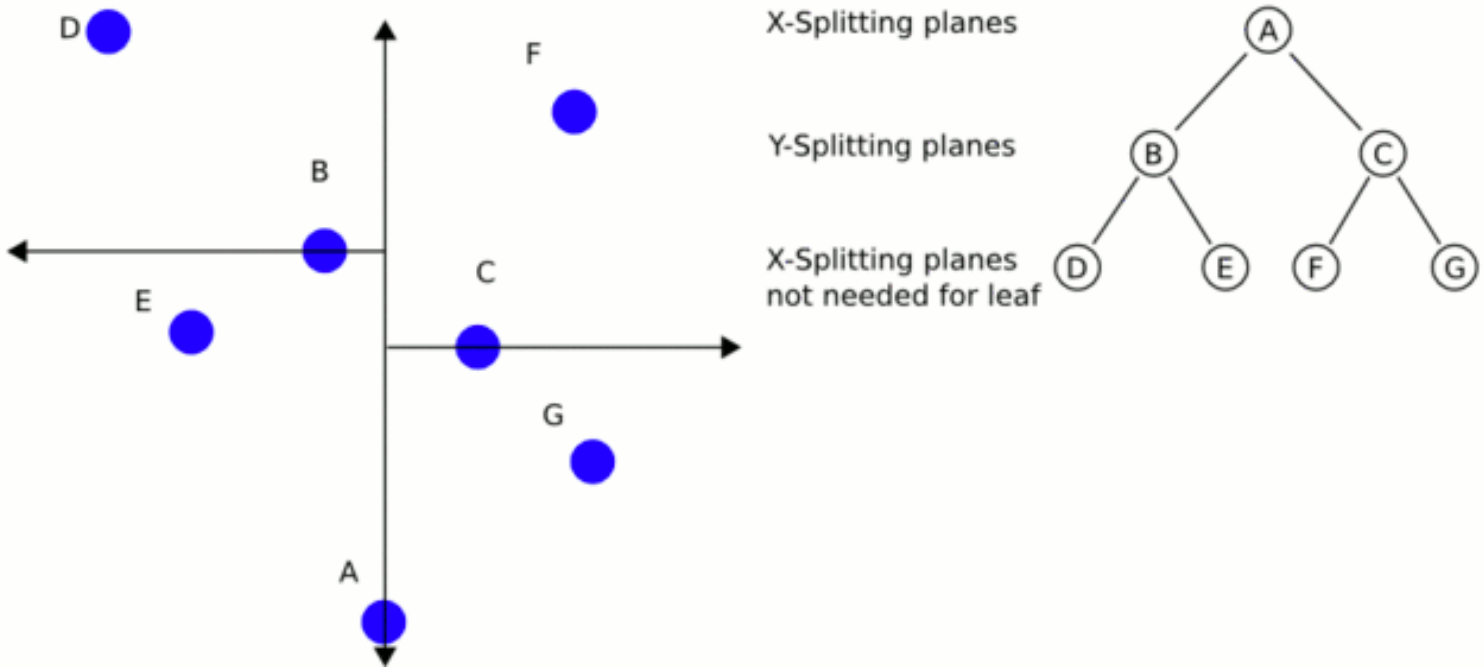
K-D Tree

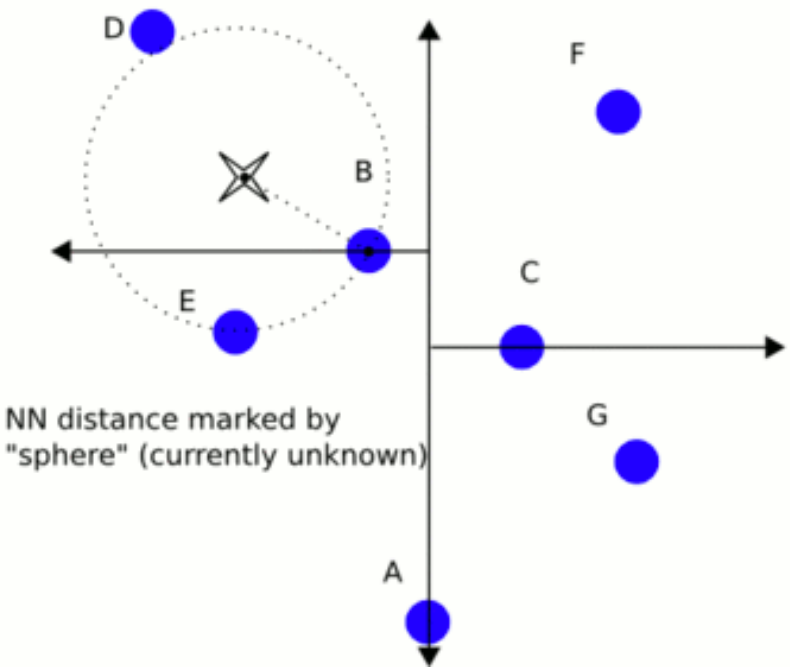


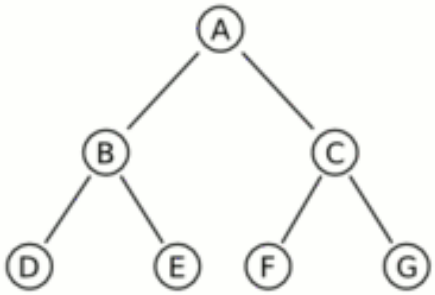
k-d tree decomposition for the point set `(2,3), (5,4), (9,6), (4,7), (8,1), (7,2)`.

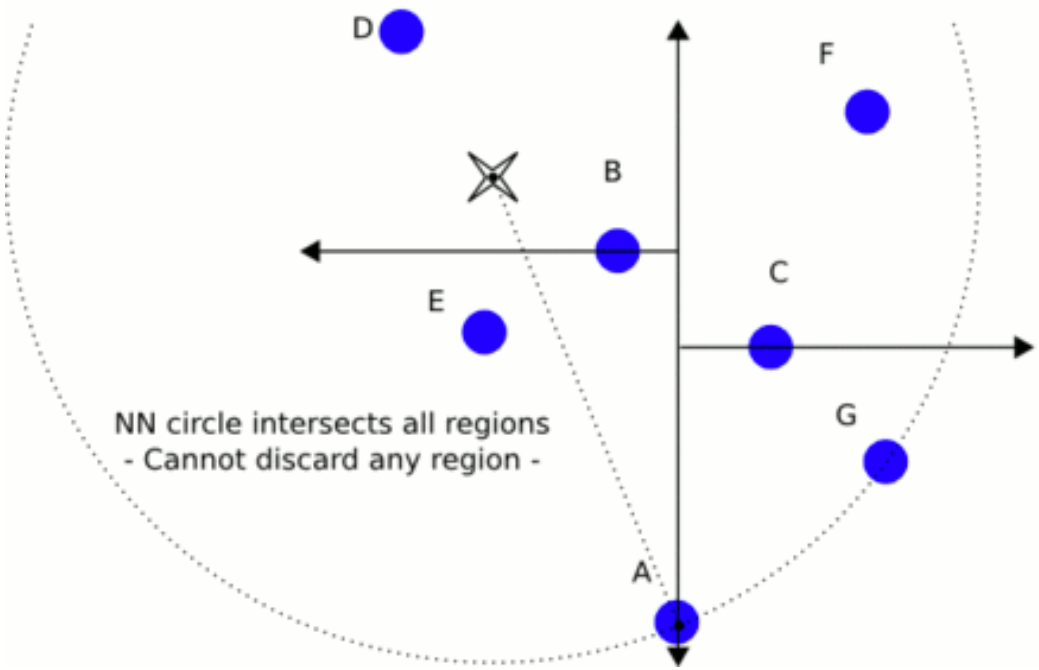
KD tree is a k-dimensional tree which partition (in the same way that decision trees do) the whole space into the regions according to some rules.

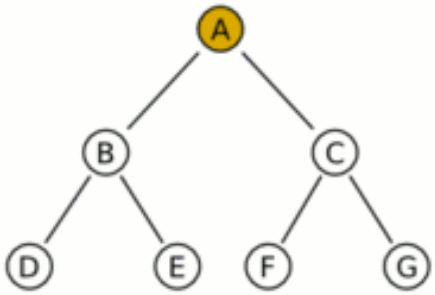



D
F
B
E
C
G
A
X-Splitting planes
Y-Splitting planes
X-Splitting planes
not needed for leaf
A
B
C
D
E
F
G

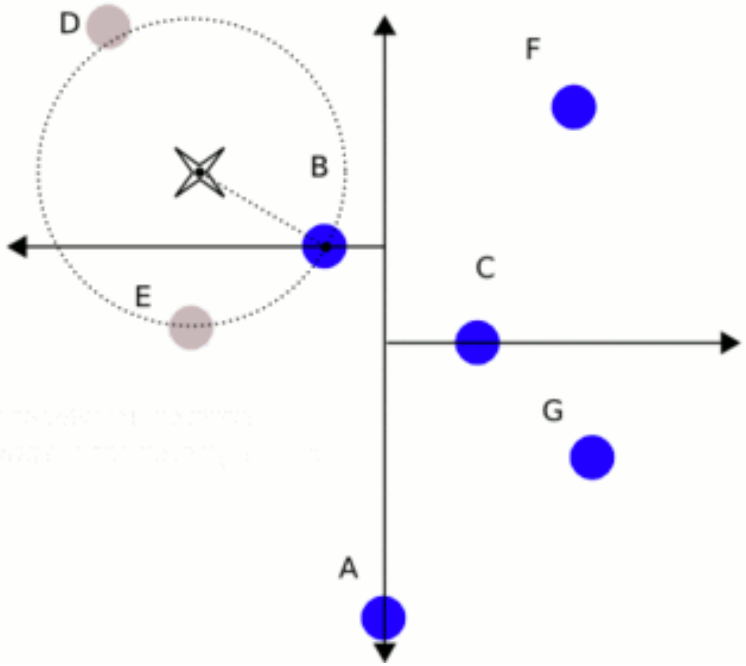

D
F
B
C
E
NN distance marked by
"sphere" (currently unknown)
G
A

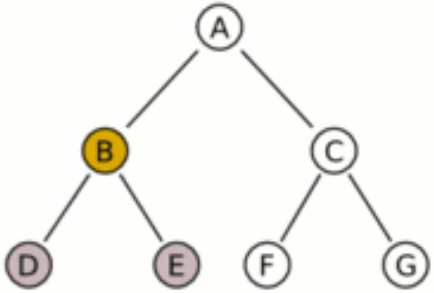

A
B
C
D
E
F
G



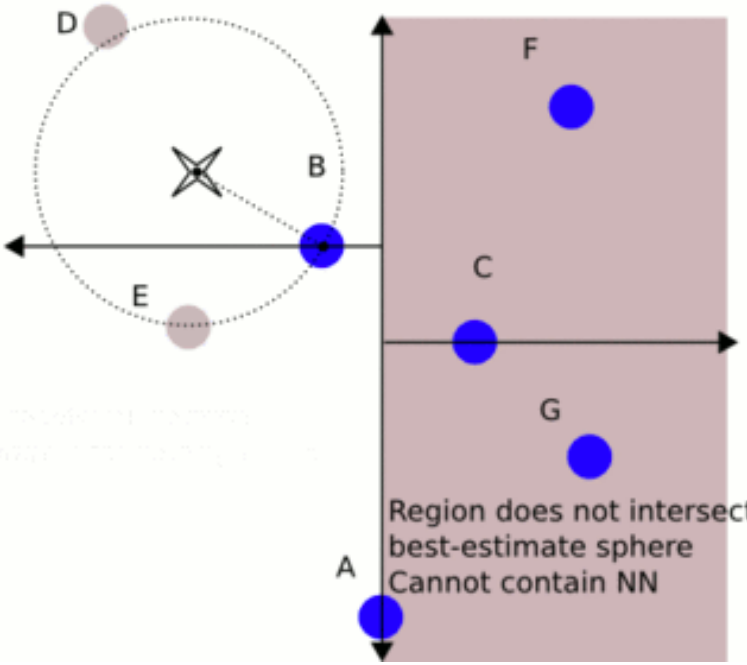


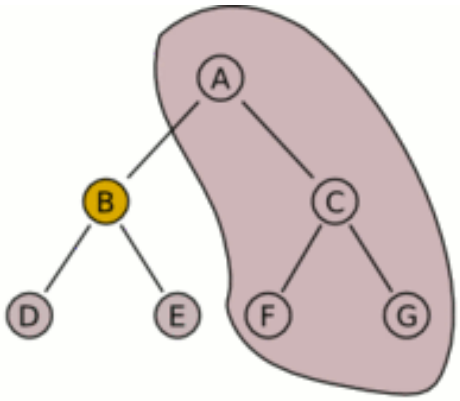
Start at A, then proceed in depth-first search (maintain a stack of parent-nodes if using a singly-linked tree). Set best estimate to A's distance Then examine left child node





D & E Discarded as B
(already visited) is closer.
B is the best estimate for B's sub-branch
Proceed back to parent node





A's children have all been searched,
B is the best estimate for entire tree

Efficient Implementation: Parallel Hardware

- **Classification cost:**
  - # distance computations
    - Const time if O(n) processors
  - Cost of finding closest
    - Compute pairwise minimum, successively
    - O(log n) time

Aside: Approximate Nearest Neighbor

Aside: Approximate Nearest Neighbor

find approximate nearest neighbors quickly

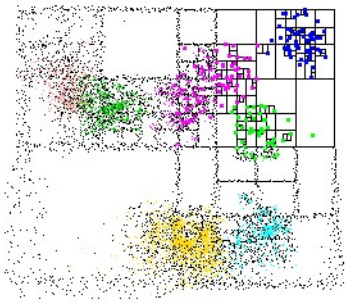
Speed up KNN

ANN: A Library for Approximate Nearest Neighbor Searching

David M. Mount and Sunil Arya

Version 1.1.2

Release Date: Jan 27, 2010



What is ANN?

ANN is a library written in C++, which supports data structures and algorithms for both exact and approximate nearest neighbor searching in arbitrarily high dimensions.

In the nearest neighbor problem a set of data points in d-dimensional space is given. These points are preprocessed into a data structure, so that given any query point q, the nearest or generally k nearest points of P to q can be reported efficiently. The distance between two points can be defined in many ways. ANN assumes that distances are measured using any class of distance functions called Minkowski metrics. These include the well known Euclidean distance, Manhattan distance, and max distance.

Based on our own experience, ANN performs quite efficiently for point sets ranging in size from thousands to hundreds of thousands, and in dimensions as high as 20. (For applications in significantly higher dimensions, the results are rather spotty, but you might try it anyway.)

The library implements a number of different data structures, based on kd-trees and box-decomposition trees, and employs a couple of different search strategies.

The library also comes with test programs for measuring the quality of performance of ANN on any particular data sets, as well as programs for visualizing the structure of the geometric data structures.

FLANN - Fast Library for Approximate Nearest Neighbors

- Home
- News
- Publications
- Download
- Changelog
- Repository

What is FLANN?

FLANN is a library for performing fast approximate nearest neighbor searches in high dimensional spaces. It contains a collection of algorithms we found to work best for nearest neighbor search and a system for automatically choosing the best algorithm and optimum parameters depending on the dataset.

FLANN is written in C++ and contains bindings for the following languages: C, MATLAB and Python.

News

- (14 December 2012) Version 1.8.0 is out bringing incremental addition/removal of points to/from indexes
- (20 December 2011) Version 1.7.0 is out bringing two new index types and several other improvements.
- You can find binary installers for FLANN on the Point Cloud Library project page. Thanks to the PCL developers!
- Mac OS X users can install flann though MacPorts (thanks to Mark Moll for maintaining the Portfile)
- New release introducing an easier way to use custom distances, kd-tree implementation optimized for low dimensionality search and experimental MPI support
- New release introducing new C++ templated API, thread-safe search, save/load of indexes and more.
- The FLANN license was changed from LGPL to BSD.

How fast is it?

In our experiments we have found FLANN to be about one order of magnitude faster on many datasets (in query time), than previously available approximate nearest neighbor search software.

Publications

More information and experimental results can be found in the following papers:

- Marius Muja and David G. Lowe: **"Scalable Nearest Neighbor Algorithms for High Dimensional Data"**. Pattern Analysis and Machine Intelligence (PAMI), Vol. 36, 2014. [PDF] [BibTeX]
- Marius Muja and David G. Lowe: **"Fast Matching of Binary Features"**. Conference on Computer and Robot Vision (CRV) 2012. [PDF] [BibTeX]
- Marius Muja and David G. Lowe, **"Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration"**, in International Conference on Computer Vision Theory and Applications (VISAPP'09), 2009 [PDF] [BibTeX]

Summary: Nearest Neighbor

- **Nearest neighbor:**
  - Training: record input vectors + output value
  - Prediction: closest training instance to new data
- **Efficient implementations**
- **Pros: fast training, very general, little bias**
- **Cons: distance metric (scaling), sensitivity to noise & extraneous features**

Outline

- **Introduction**
- **KNN**
  - Introduction
  - KNN
    - Regression
    - Classification
  - Efficient implementations:
    - k-d trees, parallelism
- **Image classification**
  - KNN for image classification
- **Hyperparameter selection via crossfold validation**
- **Summary**

Image Classification: a core task in Computer Vision

(assume given set of discrete labels)
{dog, cat, truck, plane, ...}

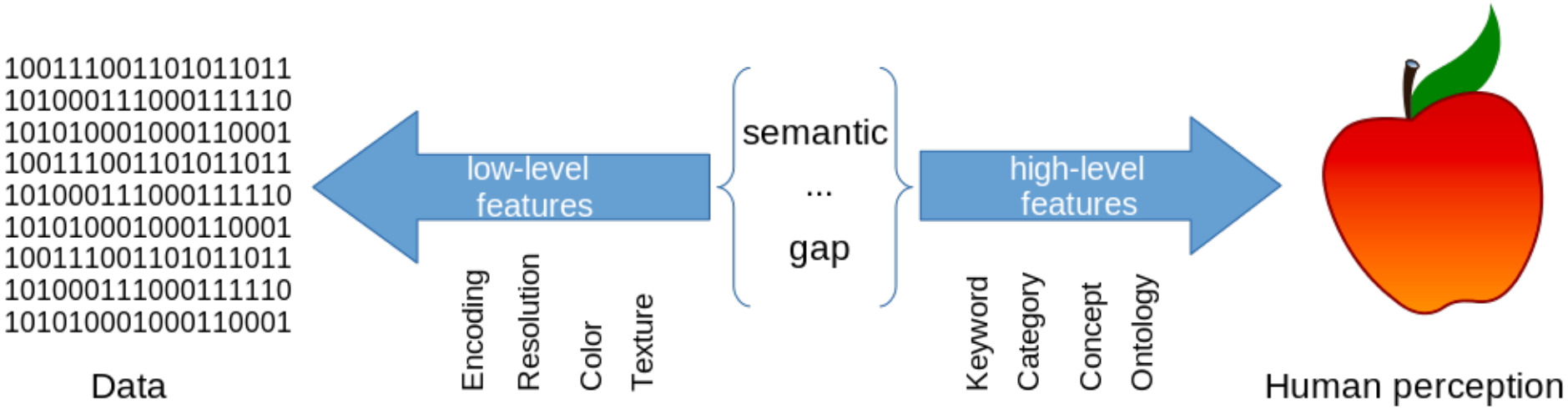
cat

If you can understand this then other tasks in CV should be easy pick up and solve

http://cs231n.stanford.edu/

Why is CV hard? Semantic gap

The **semantic gap** characterizes the difference between two descriptions of an object by different linguistic representations, for instance languages or symbols.



In many layered systems, some conflicts arise when concepts at a high level of abstraction need to be translated into lower, more concrete artifacts. This mismatch is often called **semantic gap**.

https://en.w ikipedia.org/wiki/Semantic_gap

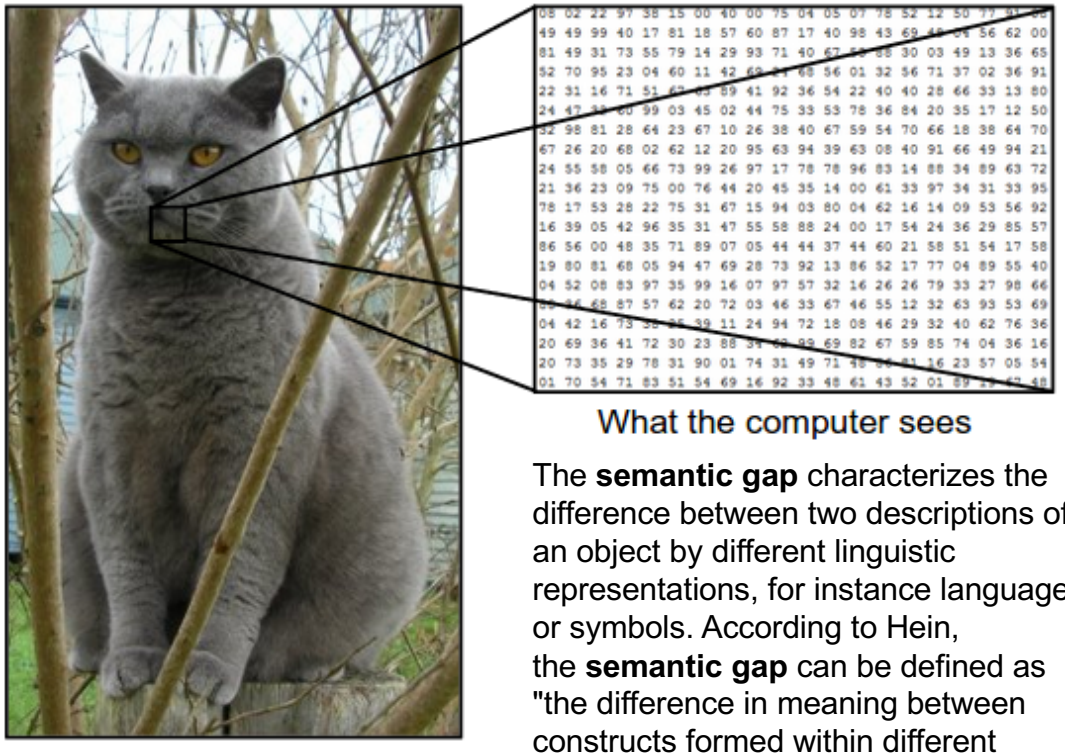
The problem: *semantic gap*

Images are represented as 3D arrays of numbers, with integers between [0, 255].

E.g.
300 x 100 x 3

Brightness in each of the 3 channels.

(3 for 3 color channels RGB)

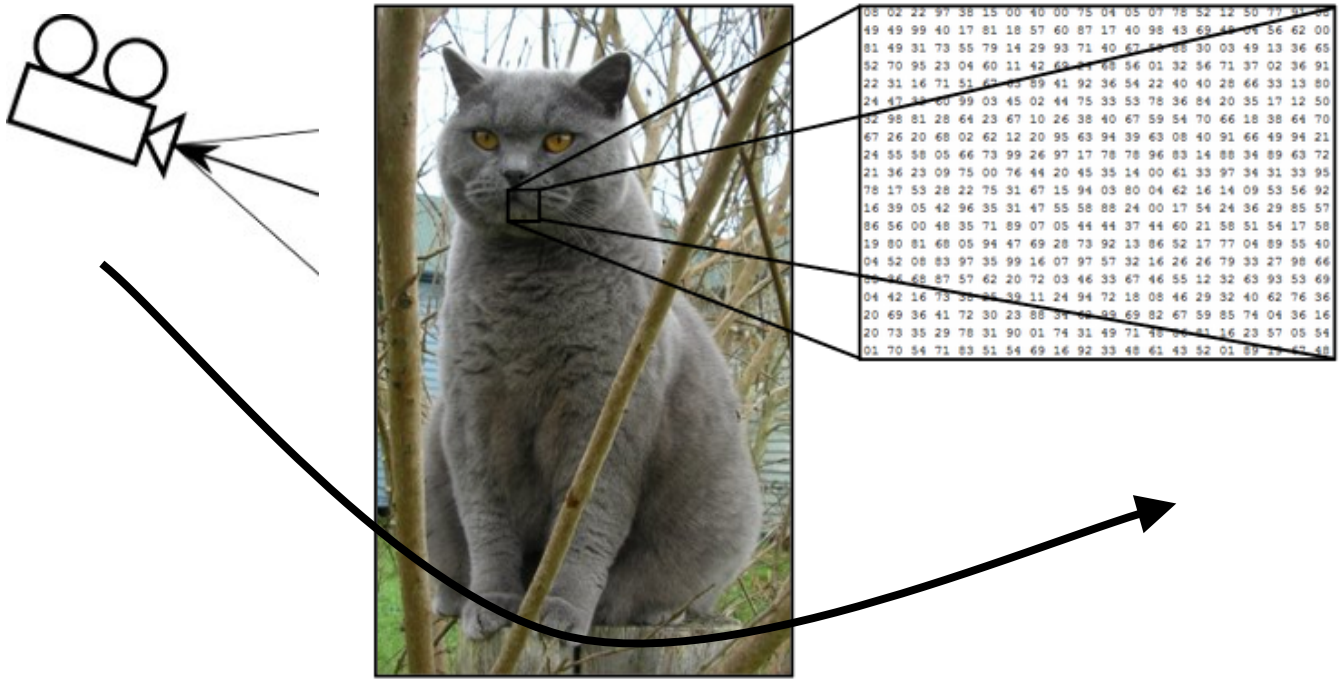


What the computer sees

The **semantic gap** characterizes the difference between two descriptions of an object by different linguistic representations, for instance languages or symbols. According to Hein, the **semantic gap** can be defined as "the difference in meaning between constructs formed within different representation systems"

http://cs231n.stanford.edu/

Challenges: Viewpoint Variation

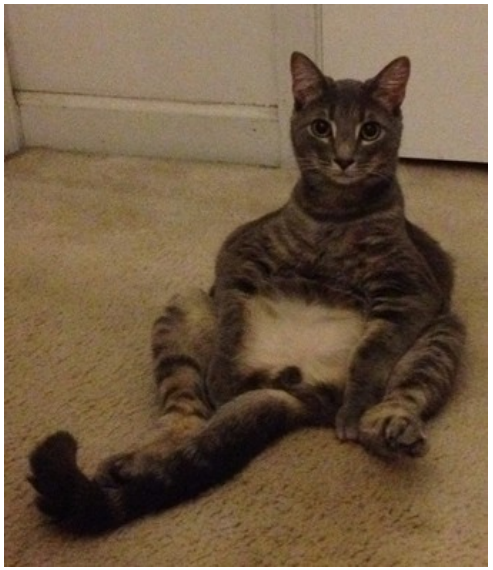


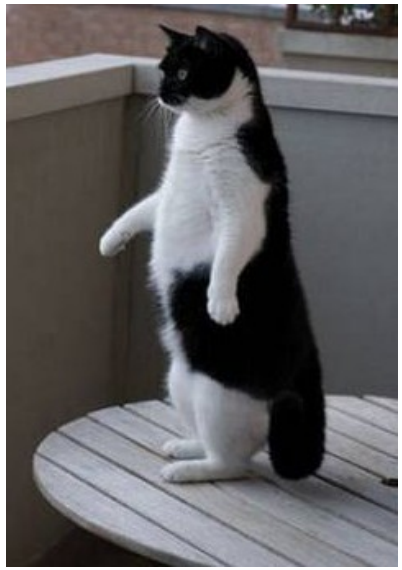
What to the brightness values in each of the 3 channels as the camera rotates?

http://cs231n.stanford.edu/

Challenges: Illumination

Challenges: Deformation



Challenges: Occlusion

This image cannot currently be displayed.

Challenges: Background clutter

Challenges: Intraclass variation (lots of species)

Cats: species etc.

An image classifier

3D Array of pixels as input [[….], […], [..]]

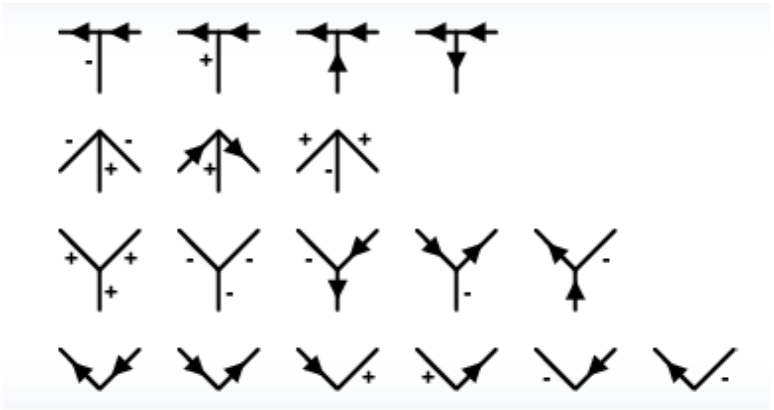
```
def predict(image):
    # ????
    return class_label
```

Unlike e.g. sorting a list of numbers,

no obvious way to hard-code the algorithm for recognizing a cat, or other classes.

Attempts have been made (explicit approaches)

???



Features, rule-based, machine learnt models

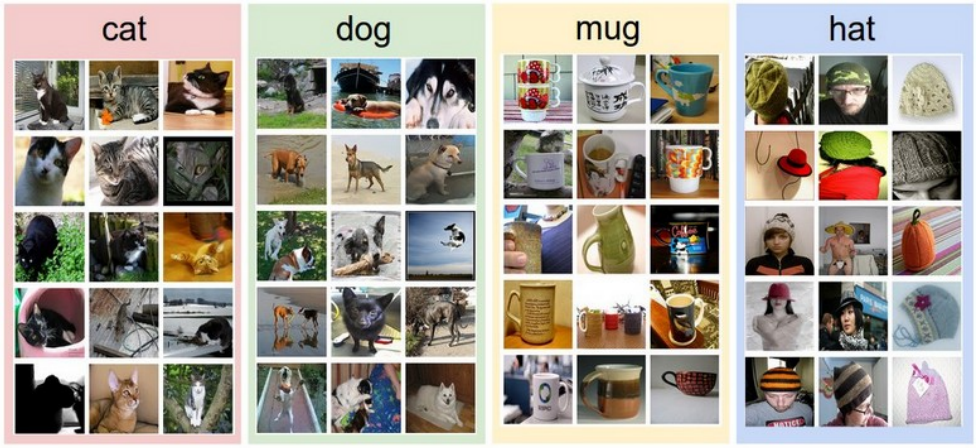
[Shanahan et al. 1994, 1995, 1997, 1998, 1999]

Data-driven approach (implicit approaches):

1. Collect a dataset of images and labels
2. Use Machine Learning to train an image classifier
3. Evaluate the classifier on a withheld set of test images

```
def train(train_images, train_labels):
  # build a model for images -> labels...
  return model

def predict(model, test_images):
  # predict test_labels using the model...
  return test_labels
```

Example training set



Outline

- **Introduction**
- **KNN**
  - Introduction
  - KNN
    - Regression
    - Classification
  - Efficient implementations:
    - k-d trees, parallelism
- **Image classification**
  - KNN for image classification in Python
  - Implementation tricks (SKLearn, pilot datasets)
  - Homegrown KNN; CIFAR-10 Kaggle Challenge
- **Hyperparameter selection via crossfold validation**
- **Summary**

First classifier: Nearest Neighbor Classifier

```
def train(train_images, train_labels):
  # build a model for images -> labels...
  return model

def predict(model, test_images):
  # predict test_labels using the model...
  return test_labels
```

Remember all training images and their labels

Predict the label of the most similar training image

How do we compare the images?

How do we compare the images? What is the **distance metric**?

L1 (aka Manhattan) distance:

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

test image					training image				pixel-wise absolute value differences				
56	32	10	18	-	10	20	24	17	= 46	12	14	1	SUM= → 456
90	23	128	133		8	10	89	100	82	13	39	33	
24	26	178	200		12	16	178	170	12	10	0	30	
2	0	255	220		4	32	233	112	2	32	22	108	

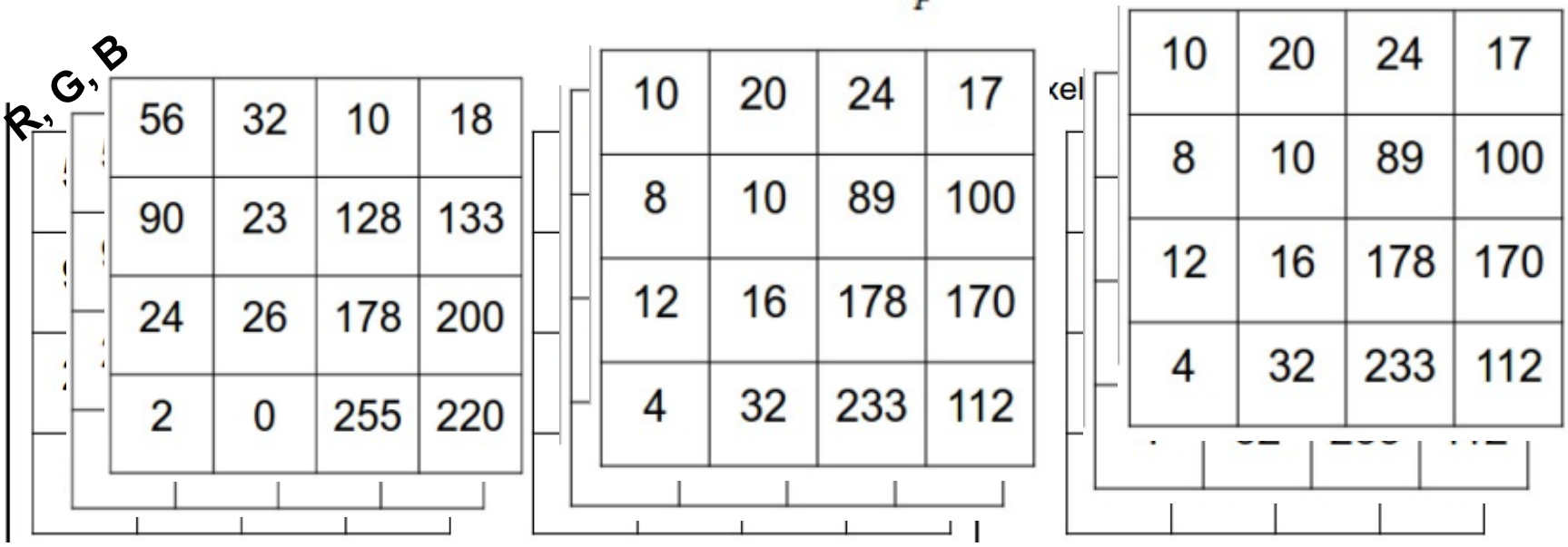
NOTE: Get ZERO if images are identical

L1 in RGB Mode

How do we compare the images? What is the **distance metric**?

L1 (aka Manhattan) distance:

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$



R G. B

SUM=456 + 202+ 34

NOTE: Get ZERO if images are identical

Np.sum

```
>>> np.sum([[0, 1], [0, 5]])          # all
6
>>> np.sum([[0, 1], [0, 5]], axis=0) # By Row
array([0, 6])
>>> np.sum([[0, 1], [0, 5]], axis=1) # By col
array([1, 5])
```

#0 total columns
[[0, 1]
[0, 5]]
>>> [0,6]
#1 Column
[[0, 1]
[0, 5]]
[1, 5]

- **axis** : *None or int or tuple of ints, optional*
- Axis or axes along which a sum is performed.
- The default, axis=None, will sum all of the elements of the input array. If axis is negative it counts from the last to the first axis.
- *New in version 1.7.0.*
  - If axis is a tuple of ints, a sum is performed on all of the axes specified in the tuple instead of a single axis or all the axes as before.

consider the numpy array `a`

```
a = np.arange(30).reshape(2, 3, 5)
print(a)

[[[ 0  1  2  3  4]
  [ 5  6  7  8  9]
  [10 11 12 13 14]]

 [[15 16 17 18 19]
  [20 21 22 23 24]
  [25 26 27 28 29]]]
```

a[0, :, :] # dim 0, pos 0

a[:, 1, :] # dim 1, pos 1

a[:, :, 3] # dim 2, pos 3

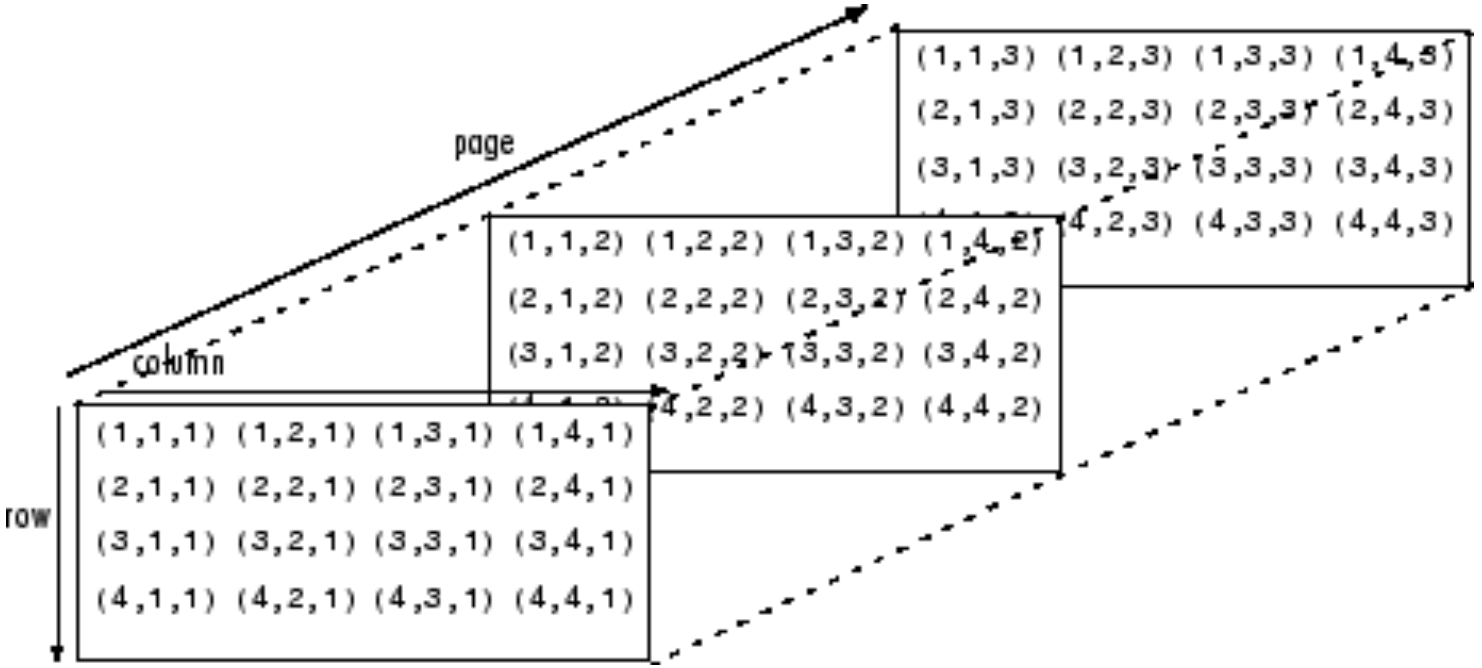
Where are the dimensions?

The dimensions and positions are highlighted by the following

```
          p  p  p  p  p
          o  o  o  o  o
          s  s  s  s  s

   dim 2  0  1  2  3  4

          |  |  |  |  |
dim 0     ↓  ↓  ↓  ↓  ↓
----> [[[ 0  1  2  3  4]   <---- dim 1, pos 0
pos 0   [ 5  6  7  8  9]   <---- dim 1, pos 1
        [10 11 12 13 14]]  <---- dim 1, pos 2
dim 0
---->  [[15 16 17 18 19]   <---- dim 1, pos 0
pos 1   [20 21 22 23 24]   <---- dim 1, pos 1
        [25 26 27 28 29]]] <---- dim 1, pos 2
          ↑  ↑  ↑  ↑  ↑
          |  |  |  |  |

                                a = np.arange(30).reshape(2, 3, 5)
   dim 2  p  p  p  p  p      print(a)
          o  o  o  o  o      [[[ 0 1 2 3 4] [ 5 6 7 8 9] [10 11 12 13 14
          s  s  s  s  s      23 24] [25 26 27 28 29]]]

          0  1  2  3  4
```

https://stackoverflow.com/questions/40857930/how-does-numpy-sum-with-axis-work

3D Matrices (aka tensor)
Page, Row, Colum

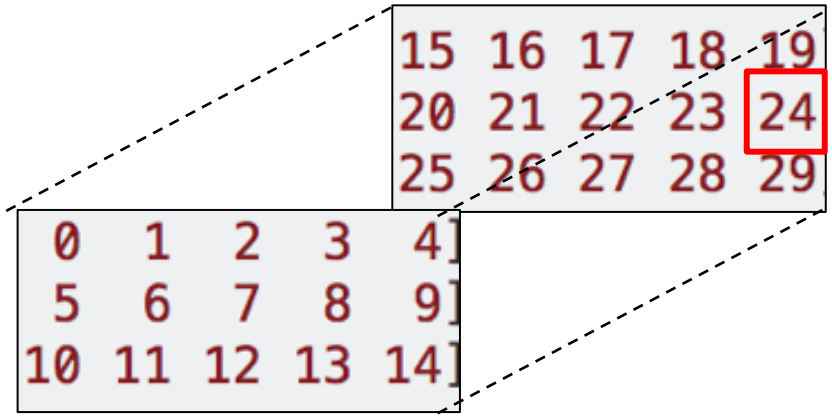


consider the numpy array `a`

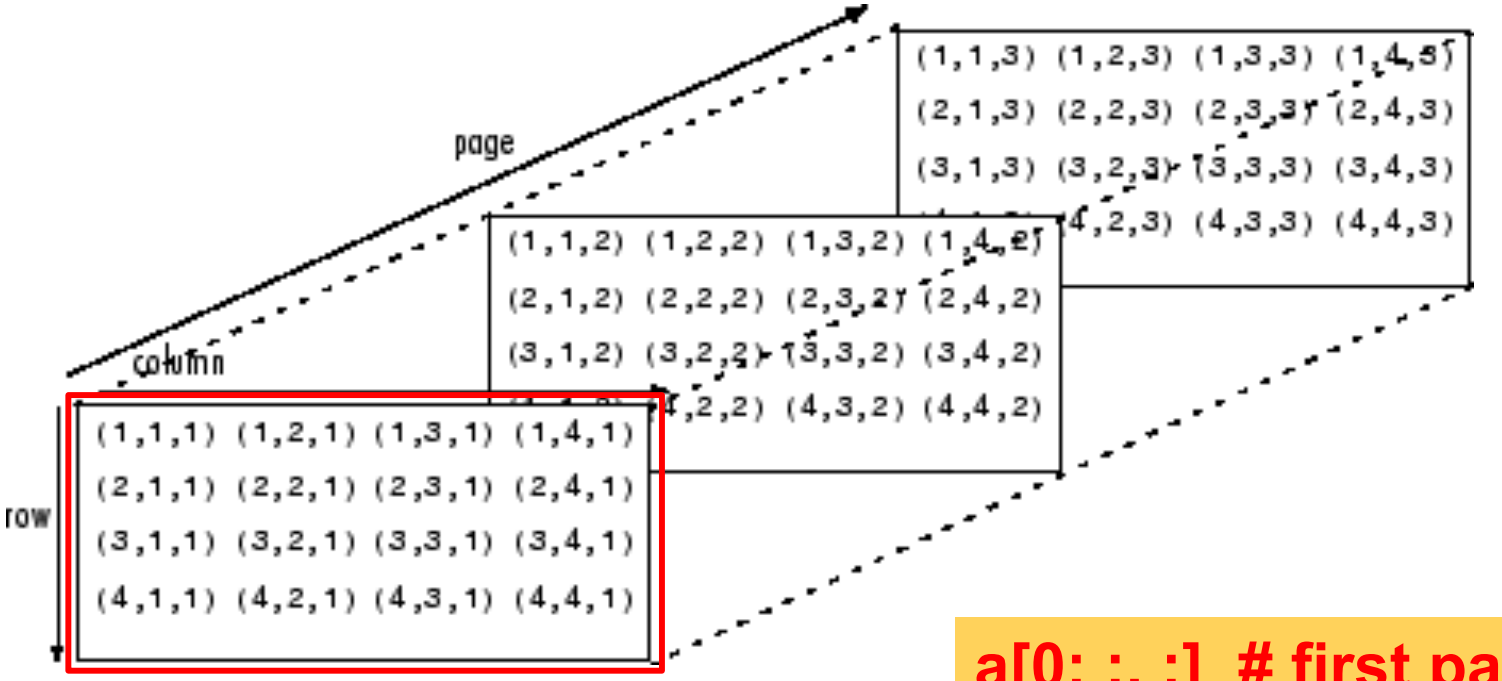
```
a = np.arange(30).reshape(2, 3, 5)
print(a)

[[[ 0  1  2  3  4]
  [ 5  6  7  8  9]
  [10 11 12 13 14]]

 [[15 16 17 18 19]
  [20 21 22 23 24]
  [25 26 27 28 29]]]
```

Zero-basis indexing in python

a[1, 1, 4] →24



3D Matrices (aka tensor)
Page, Row, Colum



a[0; :, :] # first page

consider the numpy array `a`

```
a = np.arange(30).reshape(2, 3, 5)
print(a)

[[[ 0  1  2  3  4]
  [ 5  6  7  8  9]
  [10 11 12 13 14]]

 [[15 16 17 18 19]
  [20 21 22 23 24]
  [25 26 27 28 29]]]
```

a[0; :, :] # first page

Dimension examples:

This becomes more clear with a few examples

```
a[0, :, :] # dim 0, pos 0

[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
```

```
a[:, 1, :] # dim 1, pos 1

[[ 5  6  7  8  9]
 [20 21 22 23 24]]
```

```
a[:, :, 3] # dim 2, pos 3

[[ 3  8 13]
 [18 23 28]]
```

Where are the dimensions?

The dimensions and positions are highlighted by the following

```
                p  p  p  p  p
                o  o  o
                s  s  s

    dim 2       0  1  2  3  4

                |  |  |  |  |
dim 0           ↓  ↓  ↓  ↓  ↓
----> [[[ 0  1  2  3  4]   <---- dim 1, pos 0
pos 0   [ 5  6  7  8  9]   <---- dim 1, pos 1
        [10 11 12 13 14]]  <---- dim 1, pos 2
dim 0
----->  [[15 16 17 18 19]   <---- dim 1, pos 0
pos 1    [20 21 22 23 24]   <---- dim 1, pos 1
         [25 26 27 28 29]]] <---- dim 1, pos 2
           ↑  ↑  ↑  ↑  ↑
           |  |  |  |  |

    dim 2  p  p  p  p  p
           o  o  o  o  o
           s  s  s  s  s

           0  1  2  3  4
```

ht[illegible]how-does-numpy-sum-with-axis-work

consider the numpy array `a`

```
a = np.arange(30).reshape(2, 3, 5)
print(a)

[[[ 0  1  2  3  4]
  [ 5  6  7  8  9]
  [10 11 12 13 14]]

 [[15 16 17 18 19]
  [20 21 22 23 24]
  [25 26 27 28 29]]]
```

Where are the dimensions?

The dimensions and positions are highlighted by the following

```
              p  p  p  p  p

    dim 2     0  1  2  3  4

              |  |  |  |  |
dim 0         ↓  ↓  ↓  ↓  ↓
----> [[[ 0  1  2  3  4]   <---- dim 1, pos 0
pos 0   [ 5  6  7  8  9]   <---- dim 1, pos 1
        [10 11 12 13 14]]  <---- dim 1, pos 2
dim 0
---->  [[15 16 17 18 19]   <---- dim 1, pos 0
pos 1   [20 21 22 23 24]   <---- dim 1, pos 1
        [25 26 27 28 29]]] <---- dim 1, pos 2
              ↑  ↑  ↑  ↑  ↑
              |  |  |  |  |

    dim 2     p  p  p  p  p
              o  o  o  o  o
              s  s  s  s  s

              0  1  2  3  4
```

a[:; 1, :] # first row in each matrix

Dimension examples:

This becomes more clear with a few examples

```
a[0, :, :] # dim 0, pos 0

[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
```

```
a[:, 1, :] # dim 1, pos 1

[[ 5  6  7  8  9]
 [20 21 22 23 24]]
```

```
a[:, :, 3] # dim 2, pos 3

[[ 3  8 13]
 [18 23 28]]
```

ht how-does-numpy-sum-with-axis-work

consider the numpy array `a`

```
a = np.arange(30).reshape(2, 3, 5)
print(a)

[[[ 0  1  2  3  4]
  [ 5  6  7  8  9]
  [10 11 12 13 14]]

 [[15 16 17 18 19]
  [20 21 22 23 24]
  [25 26 27 28 29]]]
```

Dimension examples:

This becomes more clear with a few examples

```
a[0, :, :] # dim 0, pos 0

[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
```

```
a[:, 1, :] # dim 1, pos 1

[[ 5  6  7  8  9]
 [20 21 22 23 24]]
```

```
a[:, :, 3] # dim 2, pos 3

[[ 3  8 13]
 [18 23 28]]
```

Where are the dimensions?

The dimensions and positions are highlighted by the following

```
           p  p  p  p  p
           o  o  o  o  o
           s  s  s  s  s

   dim 2   0  1  2  3  4

           |  |  |  |  |
dim 0      ↓  ↓  ↓  ↓  ↓
----> [[[ 0  1  2  3  4]   <---- dim 1, pos 0
pos 0    [ 5  6  7  8  9]   <---- dim 1, pos 1
         [10 11 12 13 14]]  <---- dim 1, pos 2
dim 0
----->  [[15 16 17 18 19]   <---- dim 1, pos 0
pos 1    [20 21 22 23 24]   <---- dim 1, pos 1
         [25 26 27 28 29]]] <---- dim 1, pos 2
           ↑  ↑  ↑  ↑  ↑
           |  |  |  |  |

   dim 2   p  p  p  p  p
           o  o  o  o  o
           s  s  s  s  s

           0  1  2  3  4
```

ht[illegible]how-does-numpy-sum-with-axis-work

consider the numpy array `a`

```
a = np.arange(30).reshape(2, 3, 5)
print(a)
```

```
[[[ 0  1  2  3  4]
  [ 5  6  7  8  9]
  [10 11 12 13 14]]

 [[15 16 17 18 19]
  [20 21 22 23 24]
  [25 26 27 28 29]]]
```

Where are the dimensions?

The dimensions and positions are highlighted by the following

```
----> [[[ 0  1  2  3  4]   <---- dim 1, pos 0
pos 0   [ 5  6  7  8  9]   <---- dim 1, pos 1
        [10 11 12 13 14]]  <---- dim 1, pos 2
dim 0
---->  [[15 16 17 18 19]   <---- dim 1, pos 0
pos 1   [20 21 22 23 24]   <---- dim 1, pos 1
        [25 26 27 28 29]]] <---- dim 1, pos 2
          ↑  ↑  ↑  ↑  ↑
          |  |  |  |  |

   dim 2  p  p  p  p  p
          o  o  o  o  o
          s  s  s  s  s

          0  1  2  3  4
```

a.sum(0) #Sum by page

Within each page/image add the images (total image)

Easy way to calculate the average image

sum

explanation of `sum` and `axis`

`a.sum(0)` is the sum of all slices along `dim 0`

```
a.sum(0)

[[15 17 19 21 23]
 [25 27 29 31 33]
 [35 37 39 41 43]]
```

same as

```
a[0, :, :] + \
a[1, :, :]

[[15 17 19 21 23]
 [25 27 29 31 33]
 [35 37 39 41 43]]
```

ow-does-numpy-sum-with-axis-work

consider the numpy array `a`

```
a = np.arange(30).reshape(2, 3, 5)
print(a)

[[[ 0  1  2  3  4]
  [ 5  6  7  8  9]
  [10 11 12 13 14]]

 [[15 16 17 18 19]
  [20 21 22 23 24]
  [25 26 27 28 29]]]
```

Where are the dimensions?

The dimensions and positions are highlighted by the following

a.sum(1). #collapse each page to a row

Within each page/image sum the rows

```
    dim 2  0  1  2  3  4

           |  |  |  |  |
dim 0      ↓  ↓  ↓  ↓  ↓
----> [[[ 0  1  2  3  4]   <---- dim 1, pos 0
pos 0   [ 5  6  7  8  9]   <---- dim 1, pos 1
        [10 11 12 13 14]]  <---- dim 1, pos 2
0
>   [[15 16 17 18 19]      <---- dim 1, pos 0
1    [20 21 22 23 24]      <---- dim 1, pos 1
     [25 26 27 28 29]]]    <---- dim 1, pos 2
       ↑  ↑  ↑  ↑  ↑
       |  |  |  |  |

im 2   p  p  p  p  p
       o  o  o  o  o
       s  s  s  s  s

       0  1  2  3  4
```

es-numpy-sum-with-axis-work

Dimension examples:

This becomes more clear with a few examples

`a.sum(1)` is the sum of all slices along `dim 1`

```
a.sum(1)

[[15 18 21 24 27]
 [60 63 66 69 72]]
```

same as

```
a[:, 0, :] + \
a[:, 1, :] + \
a[:, 2, :]

[[15 18 21 24 27]
 [60 63 66 69 72]]
```

consider the numpy array `a`

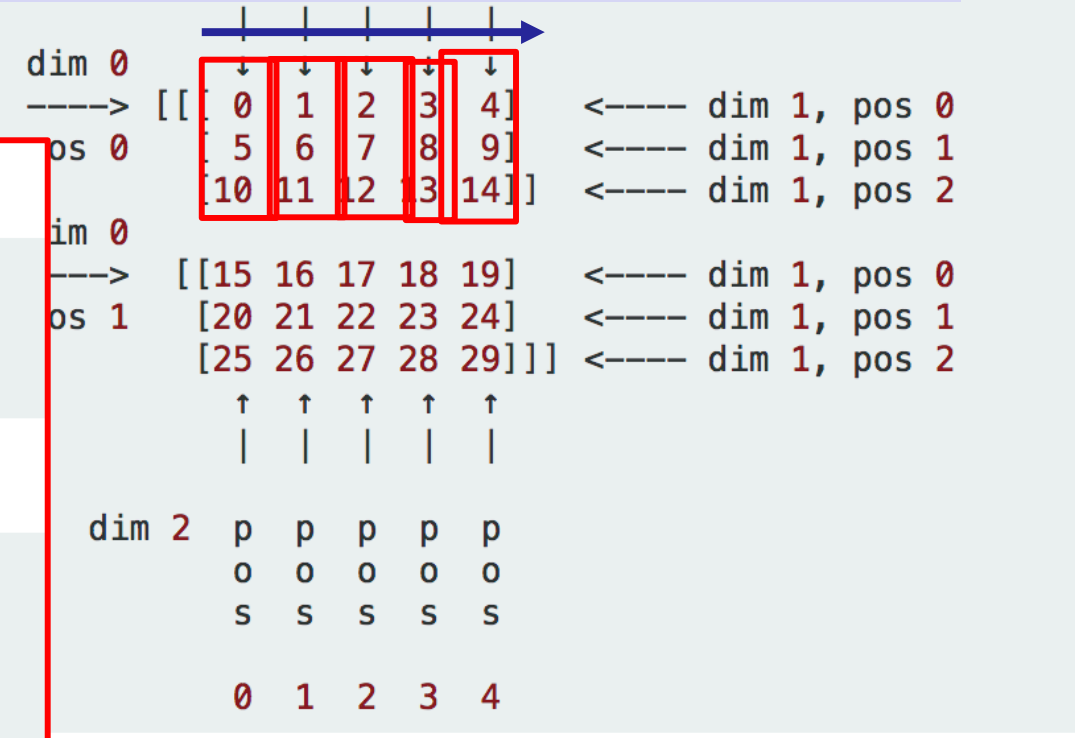
```
a = np.arange(30).reshape(2, 3, 5)
print(a)

[[[ 0  1  2  3  4]
  [ 5  6  7  8  9]
  [10 11 12 13 14]]

 [[15 16 17 18 19]
  [20 21 22 23 24]
  [25 26 27 28 29]]]
```

Where are the dimensions?

The dimensions and positions are highlighted by the following

a.sum(2) #. #collapse each page to a COLUMN vector

Within each image sum the row values [one for each row]



Dimension examples:

`a.sum(2)` is the sum of all slices along `dim 2`

```
a.sum(2)

[[ 10  35  60]
 [ 85 110 135]]
```

same as

```
a[:, :, 0] + \
a[:, :, 1] + \
a[:, :, 2] + \
a[:, :, 3] + \
a[:, :, 4]

[[ 10  35  60]
 [ 85 110 135]]
```

-does-numpy-sum-with-axis-work

consider the numpy array `a`

```
a = np.arange(30).reshape(2, 3, 5)
print(a)

[[[ 0  1  2  3  4]
  [ 5  6  7  8  9]
  [10 11 12 13 14]]

 [[15 16 17 18 19]
  [20 21 22 23 24]
  [25 26 27 28 29]]]
```

Dimension examples:

This becomes more clear with a few examples

```
a[0, :, :] # dim 0, pos 0

[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
```

default axis is `-1`
this means all axes. or sum all numbers.

```
a.sum()

435
```

```
a[:, 0, :] # dim 1, pos 0

[[ 0  1  2  3  4]
 [15 16 17 18 19]]
```

```
a[:, :, 0] # dim 2, pos 0

[[ 0  5 10]
 [15 20 25]]
```

```
a.sum(axis=0)

[[15 17 19 21 23]
 [25 27 29 31 33]
 [35 37 39 41 43]]
```

```
a.sum(axis=1)

[[ 15  18  21  24  27]
 [ 60  63  66  69  72]]
```

```
a.sum(axis=2)

[[ 10  35  60]
 [ 85 110 135]]
```

```
a.sum(axis=-1)

[[ 10  35  60]
 [ 85 110 135]]
```

```
a.sum(axis=-2)

[[ 15  18  21  24  27]
 [ 60  63  66  69  72]]
```

```
a.sum(axis=-3)

[[15 17 19 21 23]
 [25 27 29 31 33]
 [35 37 39 41 43]]
```

Where are the dimensions?

The dimensions and positions are highlighted by the following

```
           p  p  p  p  p
           o  o  o  o  o
           s  s  s  s  s

    dim 2  0  1  2  3  4

           |  |  |  |  |
dim 0      ↓  ↓  ↓  ↓  ↓
----> [[[ 0  1  2  3  4]   <---- dim 1, pos 0
pos 0    [ 5  6  7  8  9]   <---- dim 1, pos 1
         [10 11 12 13 14]]  <---- dim 1, pos 2
dim 0
----->  [[15 16 17 18 19]   <---- dim 1, pos 0
pos 1    [20 21 22 23 24]   <---- dim 1, pos 1
         [25 26 27 28 29]]] <---- dim 1, pos 2
           ↑  ↑  ↑  ↑  ↑
           |  |  |  |  |

dim 2      p  p  p  p  p
           o  o  o  o  o
           s  s  s  s  s

           0  1  2  3  4
```

ht how-does-numpy-sum-with-axis-work

Quiz 3.2 : find the average of the even odd images and the odd images

In Canvas: can move quiz 3.2 (**Quiz 3.2: How far is the origin)** to Quiz 3.3 and add a new quiz for 3.2

Quiz 3.2 : find the average of the even versus odd images

```
# Generate training data of the form 6 images with 5 by 5 pixels.
import numpy as np
np.random.seed(21)
X = np.random.rand(6, 5, 5)

print(X)
#[[[ 0 1 2 3 4]
#    [ 5 6 7 8 9]
#    [10 11 12 13 14]]
#   [[15 16 17 18 19]
#    [20 21 22 23 24]
#    [25 26 27 28 29]]]
.....
ys = np.arange(6)
```

https://canvas.instructure.com/courses/1160504/quizzes/2108197

#In python vectorized form, find the average of the even indexed images 0, 2, 4 and of the odd indexed images

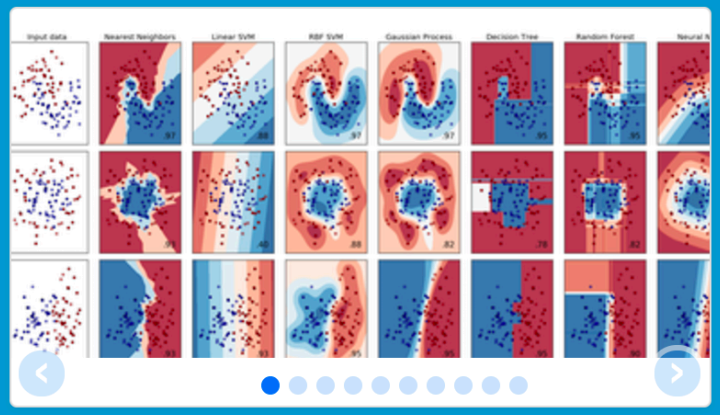
Paste in your code also into the text cell and also input the average of the even indexed images (an image of 5x5)

Outline

- **Introduction**
- **KNN**
  - Introduction
  - KNN
    - Regression
    - Classification
  - Efficient implementations:
    - k-d trees, parallelism
- **Image classification**
  - KNN for image classification
  - Implementation tricks (SKLearn, pilot datasets)
  - Homegrown KNN; CIFAR-10 Kaggle Challenge
- **Hyperparameter selection via crossfold validation**
- **Summary**

Smart coding style

- **Establish a baseline**
  - Build baseline or use a thirdparty if available
  - Benchmark your implementation against third party implementations
    - Python's SciKitLearn implementations of NN
- **Debug using smaller datasets with known results (if possible)**
  - E.g., Iris dataset
- **Benchmark on evaluation datasets (e.g., on Kaggle)**
  - CIFAR-10



scikit-learn

Machine Learning in Python

- Simple and efficient tools for data mining and data analysis
- Accessible to everybody, and reusable in various contexts
- Built on NumPy, SciPy, and matplotlib
- Open source, commercially usable - BSD license

Classification

Identifying to which category an object belongs to.

Applications: Spam detection, Image recognition.
Algorithms: SVM, nearest neighbors, random forest, ... — Examples

Regression

Predicting a continuous-valued attribute associated with an object.

Applications: Drug response, Stock prices.
Algorithms: SVR, ridge regression, Lasso, ... — Examples

Clustering

Automatic grouping of similar objects into sets.

Applications: Customer segmentation, Grouping experiment outcomes
Algorithms: k-Means, spectral clustering, mean-shift, ... — Examples

Dimensionality reduction

Reducing the number of random variables to consider.

Applications: Visualization, Increased efficiency
Algorithms: PCA, feature selection, non-negative matrix factorization. — Examples

Model selection

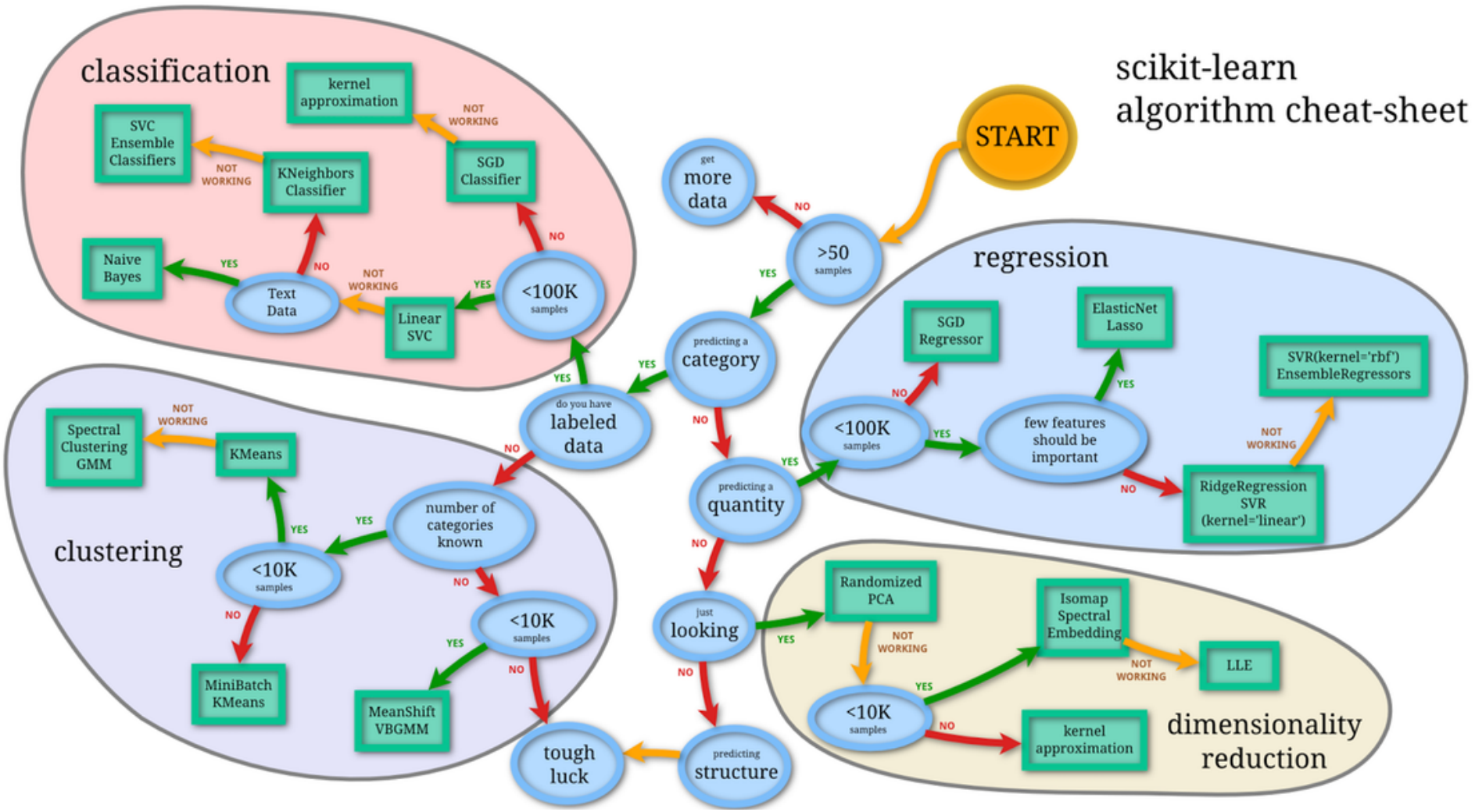
Comparing, validating and choosing parameters and models.

Goal: Improved accuracy via parameter tuning
Modules: grid search, cross validation, metrics. — Examples

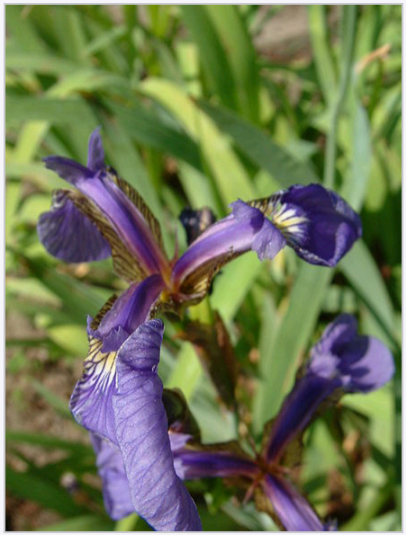
Preprocessing

Feature extraction and normalization.

Application: Transforming input data such as text for use with machine learning algorithms.
Modules: preprocessing, feature extraction. — Examples


scikit-learn
algorithm cheat-sheet
START
classification
kernel approximation
NOT WORKING
SVC Ensemble Classifiers
NOT WORKING
KNeighbors Classifier
SGD Classifier
NO
Naive Bayes
YES
NO
NOT WORKING
Text Data
NOT WORKING
YES
Linear SVC
<100K samples
YES
get more data
NO
>50 samples
YES
regression
SGD Regressor
ElasticNet Lasso
SVR(kernel='rbf') EnsembleRegressors
predicting a category
YES
NO
<100K samples
YES
few features should be important
YES
NOT WORKING
RidgeRegression SVR (kernel='linear')
NO
YES
do you have labeled data
NO
predicting a quantity
YES
NO
clustering
Spectral Clustering GMM
NOT WORKING
KMeans
YES
number of categories known
YES
NO
<10K samples
NO
<10K samples
YES
NO
MiniBatch KMeans
MeanShift VBGMM
just looking
YES
Randomized PCA
NOT WORKING
Isomap Spectral Embedding
NOT WORKING
LLE
YES
<10K samples
NO
kernel approximation
dimensionality reduction
NO
tough luck
predicting structure

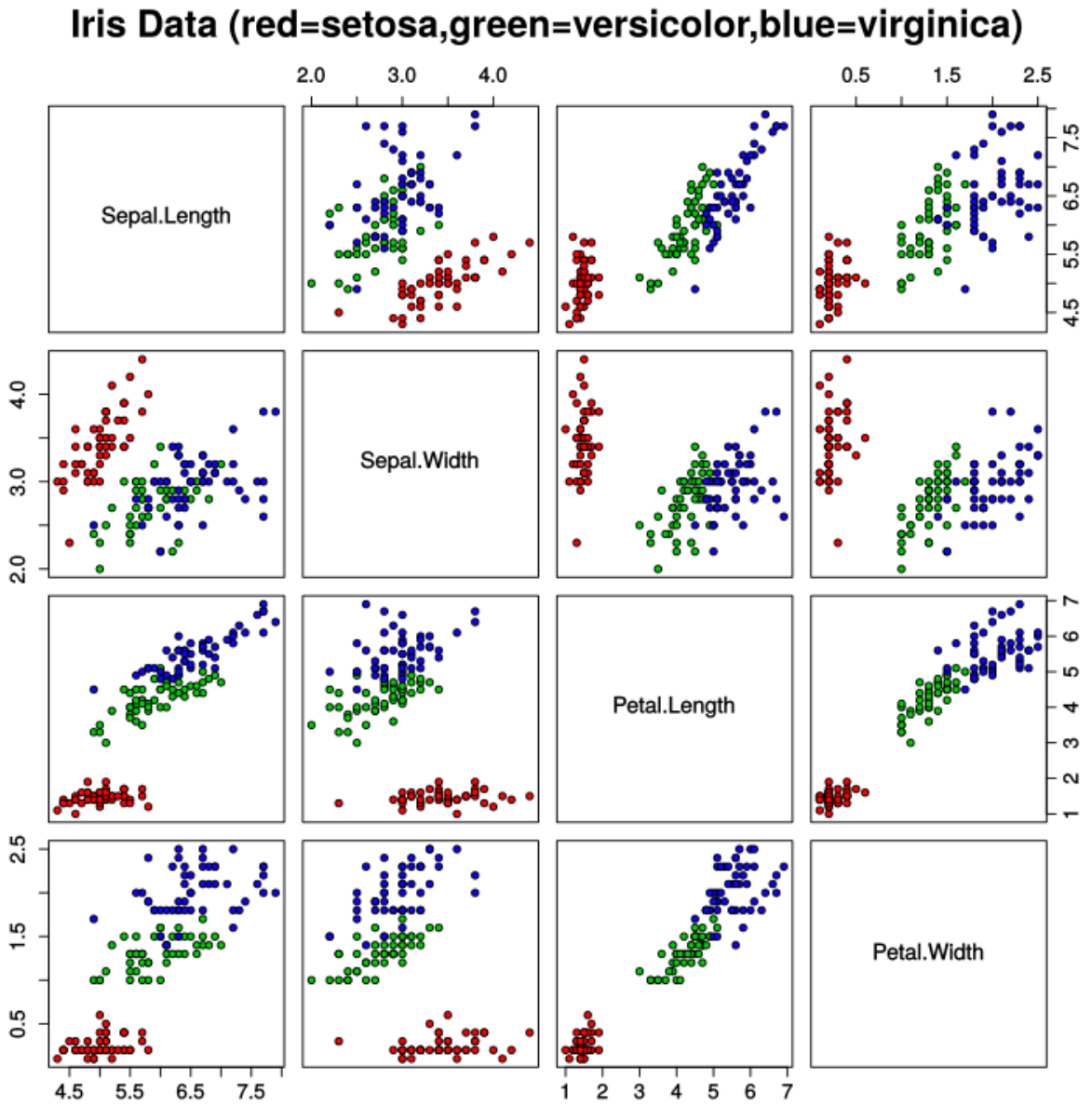
Iris Dataset: 150x4 → Class



Iris setosa

Iris versicolor

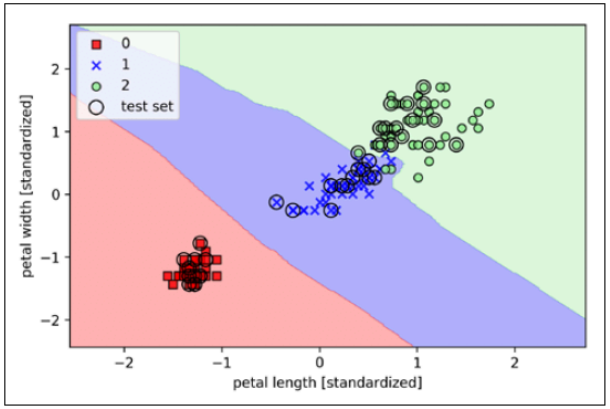
Iris virginica



Benchmark your code using KNN model in scikitlearn using a Euclidean distance metric:

By executing the following code, we will now implement a KNN model in scikit-learn using a Euclidean distance metric:

```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> knn = KNeighborsClassifier(n_neighbors=5, p=2,
...                            metric='minkowski')
>>> knn.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std, y_combined,
...                       classifier=knn, test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```



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.

Which similarity measure do use?

The choice of distance is a **hyperparameter**
common choices:

L1 (Manhattan) distance

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

L2 (Euclidean) distance

$$d_2(I_1, I_2) = \sqrt{\sum_p \left(I_1^p - I_2^p\right)^2}$$

http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

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.

Standardize data + Euclidean Distance

- **Often, a simple Euclidean distance measure is used for real-value samples, for example, the flowers in our Iris dataset, which have features measured in centimeters.**
- **However, if we are using a Euclidean distance measure, it is also important to standardize the data so that each feature contributes equally to the distance.**
  - $x_i = \frac{x_i - \mu_{xi}}{\sigma_{xi}}$
- **The Minkowski distance that we used in the previous code is just a generalization of the Euclidean and Manhattan distance, which can be written as follows:**

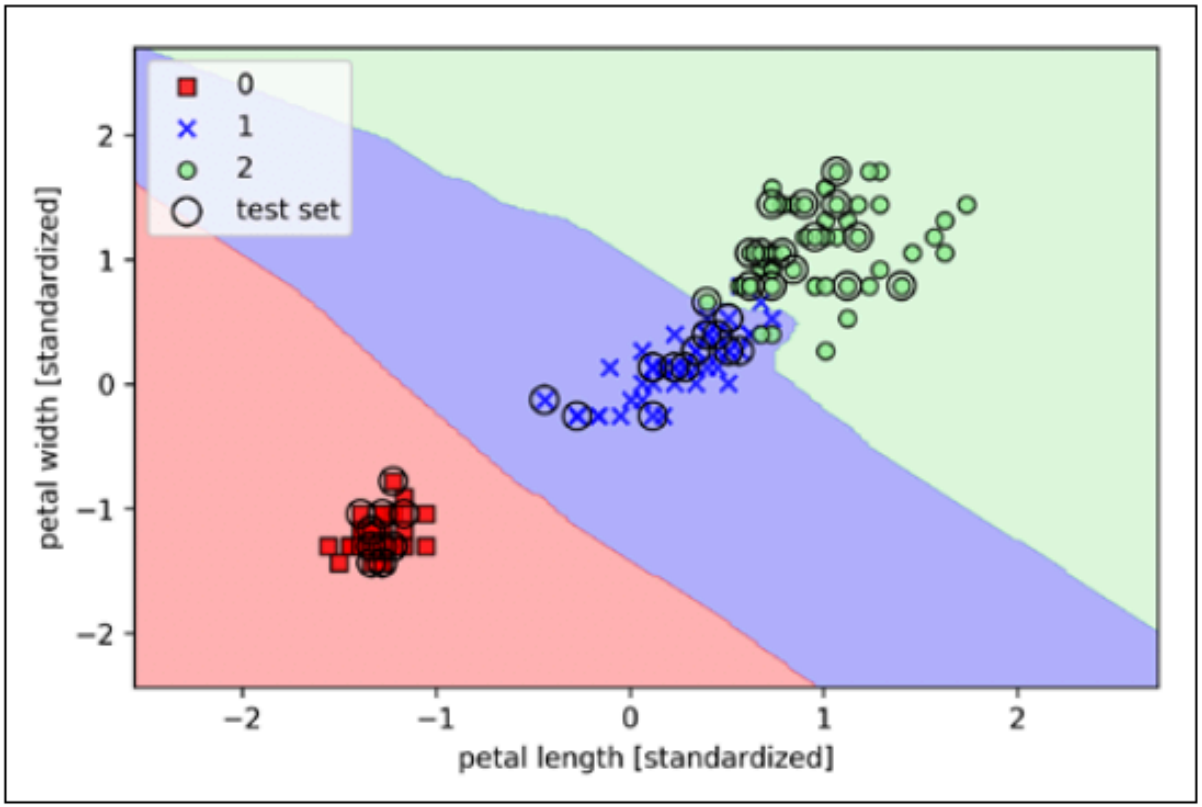
$$d\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right) = \sqrt[p]{\sum_k \left|x_k^{(i)} - x_k^{(j)}\right|^p}$$

Euclidean distance if we set the parameter p=2 or the Manhattan distance at p=1.

Standardized Iris KNN=5

Decision boundary in Petal Width and Petal Length

By specifying five neighbors in the KNN model for this dataset, we obtain a relatively smooth decision boundary, as shown in the following figure:



Outline

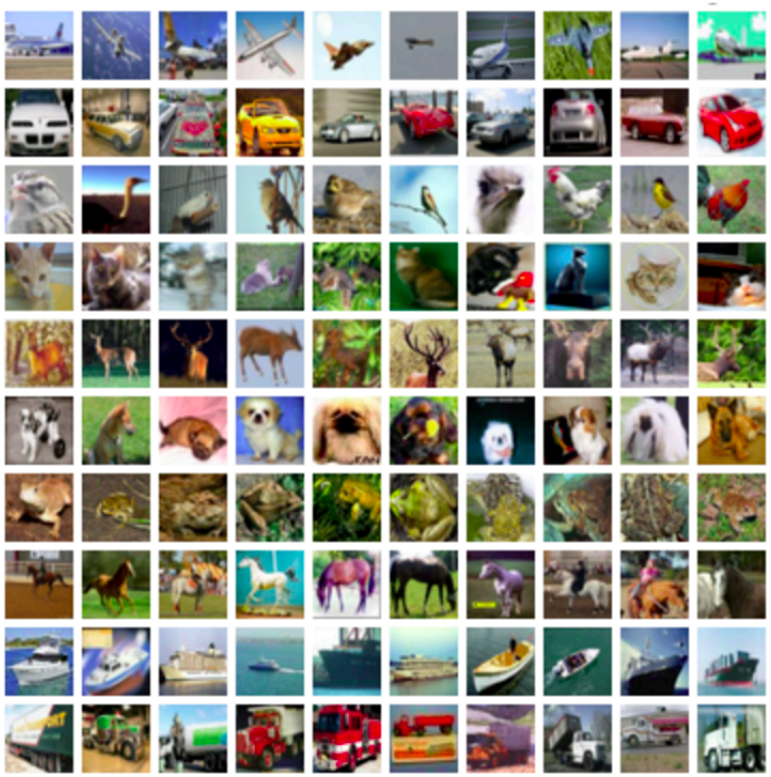
- **Introduction**
- **KNN**
  - Introduction
  - KNN
    - Regression
    - Classification
  - Efficient implementations:
    - k-d trees, parallelism
- **Image classification**
  - KNN for image classification
  - Implementation tricks (SKLearn, pilot datasets)
  - Homegrown KNN; CIFAR-10 Kaggle Challenge
- **Hyperparameter selection via crossfold validation**
- **Summary**

CIFAR-10

CIFAR-10 is an established computer-vision dataset used for object recognition. It is a subset of the 80 million tiny images dataset and consists of 60,000 32x32 color images containing one of 10 object classes, with 6000 images per class. It was collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton.

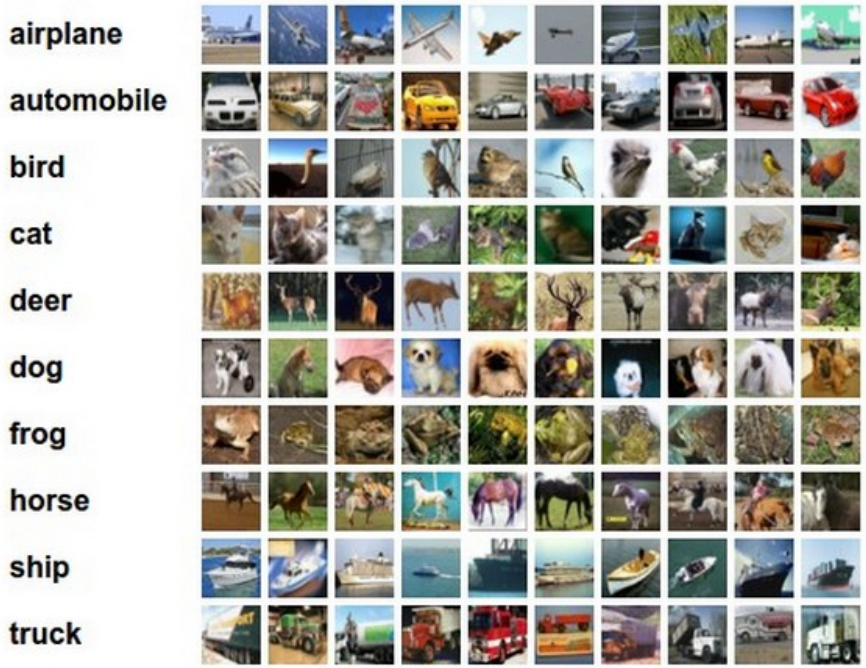
Kaggle is hosting a CIFAR-10 leaderboard for the machine learning community to use for fun and practice. You can see how your approach compares to the latest research methods on Rodrigo Benenson's classification results page.

https://www.kaggle.com/c/cifar-10/leaderboard



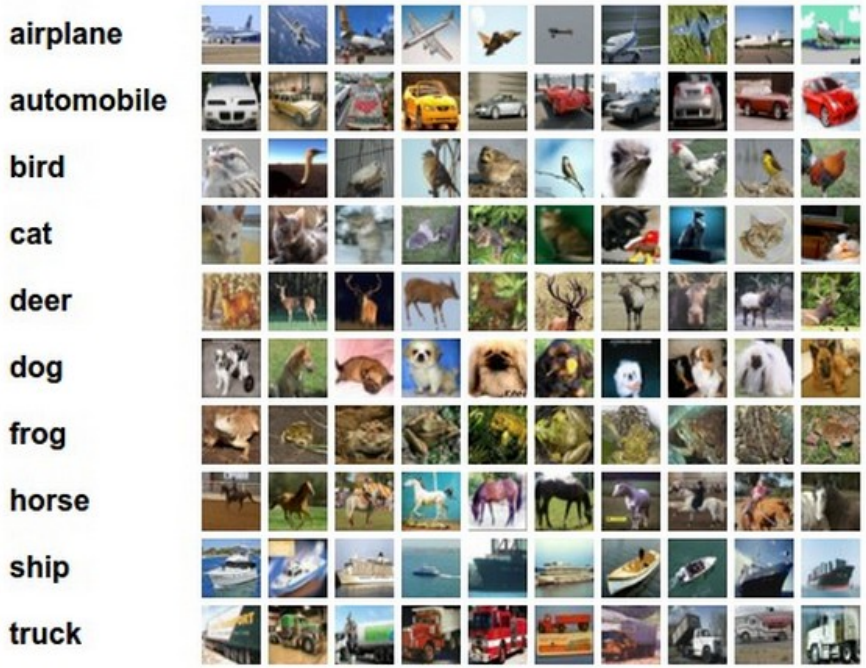
Example dataset: CIFAR-10

Example dataset: **CIFAR-10**
10 labels
50,000 training images, each image is tiny: 32x32 (thumbnail images)
10,000 test images.

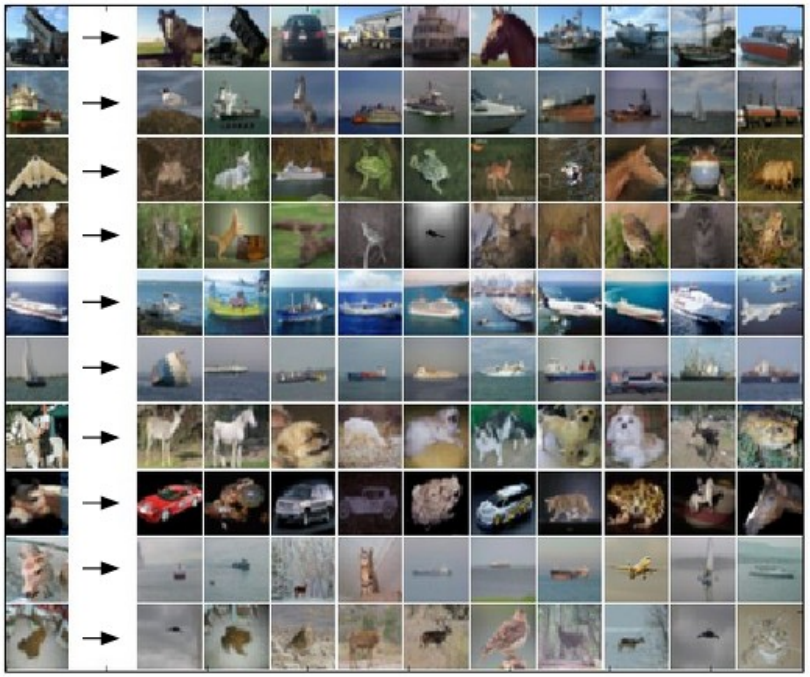


10 NNs for each image

Example dataset: **CIFAR-10**
10 labels
50,000 training images
10,000 test images.



For every test image (first column), examples of nearest neighbors in rows



CIFAR Dataset: 50,000 Train; 10,000 Test

Let's also look at how we might implement the classifier in code.

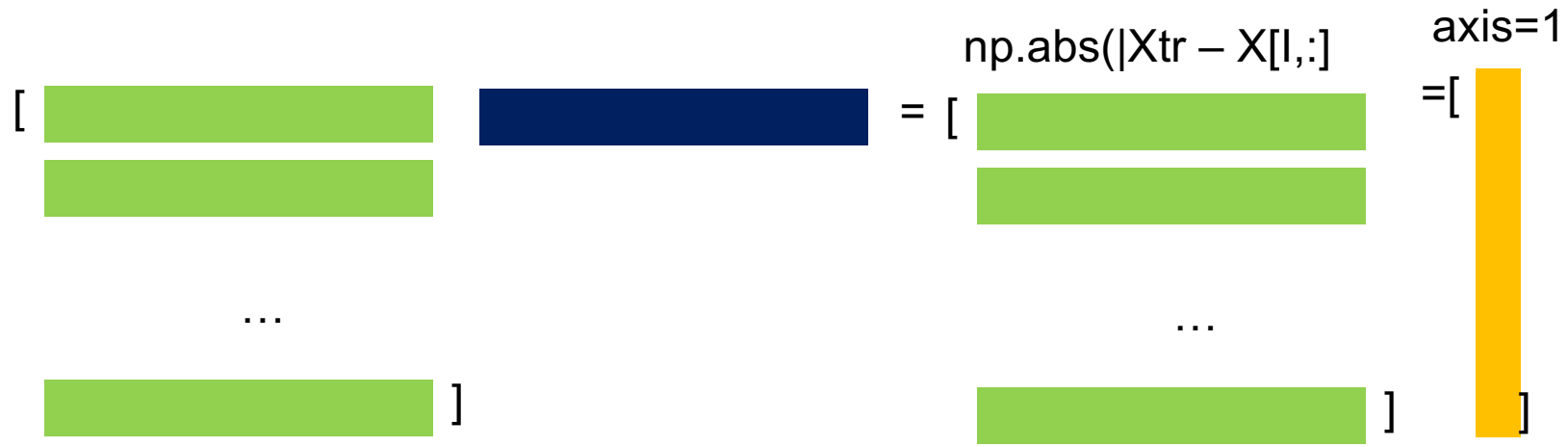
First, let's load the CIFAR-10 data into memory as 4 arrays: the training data/labels and the test data/labels. In the code below, Xtr (of size 50,000 x 32 x 32 x 3) holds all the images in the training set, and a corresponding 1-dimensional array Ytr (of length 50,000) holds the training labels (from 0 to 9)

```
Xtr, Ytr, Xte, Yte = load_CIFAR10('data/cifar10/') # a magic function we provide
# flatten out all images to be one-dimensional
Xtr_rows = Xtr.reshape(Xtr.shape[0], 32 * 32 * 3) # Xtr_rows becomes 50000 x 3072
Xte_rows = Xte.reshape(Xte.shape[0], 32 * 32 * 3) # Xte_rows becomes 10000 x 3072
```

Now that we have all images stretched out as rows, here is how we could train and evaluate a classifier:

```
nn = NearestNeighbor() # create a Nearest Neighbor classifier class
nn.train(Xtr_rows, Ytr) # train the classifier on the training images and labels
Yte_predict = nn.predict(Xte_rows) # predict labels on the test images
# and now print the classification accuracy, which is the average number
# of examples that are correctly predicted (i.e. label matches)
print 'accuracy: %f' % ( np.mean(Yte_predict == Yte) )
```

No loops: vectorized form




```
def predict(self, X):
  """ X is N x D where each row is an example we wish to predict label for """
  num_test = X.shape[0]
  # lets make sure that the output type matches the input type
  Ypred = np.zeros(num_test, dtype = self.ytr.dtype)

  # loop over all test rows
  for i in xrange(num_test):
    # find the nearest training image to the i'th test image
    # using the L1 distance (sum of absolute value differences)
    distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
    min_index = np.argmin(distances) # get the index with smallest distance
    Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

  return Ypred
```

Nearest Neighbor classifier

```python
import numpy as np

class NearestNeighbor:
  def __init__(self):
    pass

  def train(self, X, y):
    """ X is N x D where each row is an example. Y is 1-dimension of size N """
    # the nearest neighbor classifier simply remembers all the training data
    self.Xtr = X
    self.ytr = y

  def predict(self, X):
    """ X is N x D where each row is an example we wish to predict label for """
    num_test = X.shape[0]
    # lets make sure that the output type matches the input type
    Ypred = np.zeros(num_test, dtype = self.ytr.dtype)

    # loop over all test rows
    for i in xrange(num_test):
      # find the nearest training image to the i'th test image
      # using the L1 distance (sum of absolute value differences)
      distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
      min_index = np.argmin(distances) # get the index with smallest distance
      Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

    return Ypred
```

For each training example Xtr - Xi

Nearest Neighbor classifier

```
import numpy as np

class NearestNeighbor:
  def __init__(self):
    pass

  def train(self, X, y):
    """ X is N x D where each row is an example. Y is 1-dimension of size N """
    # the nearest neighbor classifier simply remembers all the training data
    self.Xtr = X
    self.ytr = y

  def predict(self, X):
    """ X is N x D where each row is an example we wish to predict label for """
    num_test = X.shape[0]
    # lets make sure that the output type matches the input type
    Ypred = np.zeros(num_test, dtype = self.ytr.dtype)

    # loop over all test rows
    for i in xrange(num_test):
      # find the nearest training image to the i'th test image
      # using the L1 distance (sum of absolute value differences)
      distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
      min_index = np.argmin(distances) # get the index with smallest distance
      Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

    return Ypred
```

X holds the images and y the labels

remember the training data

Nearest Neighbor classifier

```
import numpy as np

class NearestNeighbor:
  def __init__(self):
    pass

  def train(self, X, y):
    """ X is N x D where each row is an example. Y is 1-dimension of size N """
    # the nearest neighbor classifier simply remembers all the training data
    self.Xtr = X
    self.ytr = y

  def predict(self, X):
    """ X is N x D where each row is an example we wish to predict label for """
    num_test = X.shape[0]
    # lets make sure that the output type matches the input type
    Ypred = np.zeros(num_test, dtype = self.ytr.dtype)

    # loop over all test rows
    for i in xrange(num_test):
      # find the nearest training image to the i'th test image
      # using the L1 distance (sum of absolute value differences)
      distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
      min_index = np.argmin(distances) # get the index with smallest distance
      Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

    return Ypred
```

QUIZ: How many neighbors are we using here?

Nearest Neighbor classifier: predict

```
import numpy as np

class NearestNeighbor:
  def __init__(self):
    pass

  def train(self, X, y):
    """ X is N x D where each row is an example. Y is 1-dimension of size N """
    # the nearest neighbor classifier simply remembers all the training data
    self.Xtr = X
    self.ytr = y

  def predict(self, X):
    """ X is N x D where each row is an example we wish to predict label for """
    num_test = X.shape[0]
    # lets make sure that the output type matches the input type
    Ypred = np.zeros(num_test, dtype = self.ytr.dtype)

    # loop over all test rows
    for i in xrange(num_test):
      # find the nearest training image to the i'th test image
      # using the L1 distance (sum of absolute value differences)
      distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
      min_index = np.argmin(distances) # get the index with smallest distance
      Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

    return Ypred
```

Vectorized python code; no for loops

for every test image:
- find nearest train image with L1 distance
- predict the label of nearest training image

For each training example |Xtr – Xi|

Question

Nearest Neighbor classifier

Q: how does the classification speed depend on the size of the training data?

```
import numpy as np

class NearestNeighbor:
  def __init__(self):
    pass

  def train(self, X, y):
    """ X is N x D where each row is an example. Y is 1-dimension of size N """
    # the nearest neighbor classifier simply remembers all the training data
    self.Xtr = X
    self.ytr = y

  def predict(self, X):
    """ X is N x D where each row is an example we wish to predict label for """
    num_test = X.shape[0]
    # lets make sure that the output type matches the input type
    Ypred = np.zeros(num_test, dtype = self.ytr.dtype)

    # loop over all test rows
    for i in xrange(num_test):
      # find the nearest training image to the i'th test image
      # using the L1 distance (sum of absolute value differences)
      distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
      min_index = np.argmin(distances) # get the index with smallest distance
      Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

    return Ypred
```

Vectorized python code

NN's predictive performance is terrible (slow)

```
import numpy as np

class NearestNeighbor:
  def __init__(self):
    pass

  def train(self, X, y):
    """ X is N x D where eac
    # the nearest neighbor c
    self.Xtr = X
    self.ytr = y

  def predict(self, X):
    """ X is N x D where eac
    num_test = X.shape[0]
    # lets make sure that th
    Ypred = np.zeros(num_tes

    # loop over all test rows
    for i in xrange(num_test):
      # find the nearest training image to the i'th test image
      # using the L1 distance (sum of absolute value differences)
      distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
      min_index = np.argmin(distances) # get the index with smallest distance
      Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

    return Ypred
```

Q: how does the classification speed depend on the size of the training data? **linearly :(**

This is **backwards**:

- test time performance is usually much more important in practice.
- CNNs flip this: expensive training, cheap test evaluation
- Constant compute for classification for CNNs

CIFAR Dataset: 50,000 Train; 10,000 Test

Let's also look at how we might implement the classifier in code.

First, let's load the CIFAR-10 data into memory as 4 arrays: the training data/labels and the test data/labels. In the code below, Xtr (of size 50,000 x 32 x 32 x 3) holds all the images in the training set, and a corresponding 1-dimensional array Ytr (of length 50,000) holds the training labels (from 0 to 9)

```
Xtr, Ytr, Xte, Yte = load_CIFAR10('data/cifar10/') # a magic function we provide
# flatten out all images to be one-dimensional
Xtr_rows = Xtr.reshape(Xtr.shape[0], 32 * 32 * 3) # Xtr_rows becomes 50000 x 3072
Xte_rows = Xte.reshape(Xte.shape[0], 32 * 32 * 3) # Xte_rows becomes 10000 x 3072
```

Now that we have all images stretched out as rows, here is how we could train and evaluate a classifier:

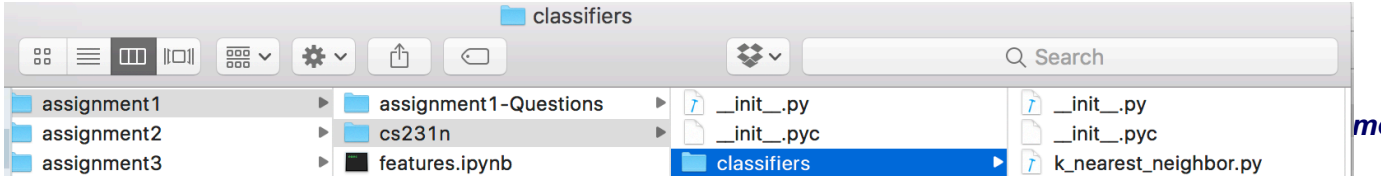
```
nn = NearestNeighbor() # create a Nearest Neighbor classifier class
nn.train(Xtr_rows, Ytr) # train the classifier on the training images and labels
Yte_predict = nn.predict(Xte_rows) # predict labels on the test images
# and now print the classification accuracy, which is the average number
# of examples that are correctly predicted (i.e. label matches)
print 'accuracy: %f' % ( np.mean(Yte_predict == Yte) )
```

Notebook

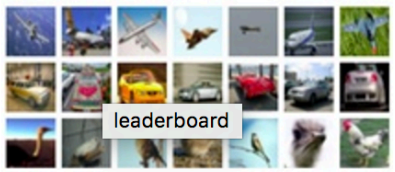
- **Random 10%**
- **KNN can yield accuarcy of 27% only using KNN**
- **Neural networks (CNNs) → 95%**

> If you ran this code, you would see that this classifier only achieves **38.6%** on CIFAR-10. That's more impressive than guessing at random (which would give 10% accuracy since there are 10 classes), but nowhere near human performance (which is estimated at about 94%) or near state-of-the-art Convolutional Neural Networks that achieve about 95%, matching human accuracy (see the leaderboard of a recent Kaggle competition on CIFAR-10).

- **Do a submission on Kaggle**
  - (see the leaderboard of a recent Kaggle competition on CIFAR-10).



CIFAR-10 on Kaggle: make a submission



CIFAR-10 - Object Recognition in Images

Identify the subject of 60,000 labeled images

231 teams · 3 years ago

Overview Data Discussion Leaderboard More Submit Predictions

Make a Submission

Make a submission for BigDataAnalytics

You have 10 submissions remaining today. This resets a day from now (00: 00 UTC).

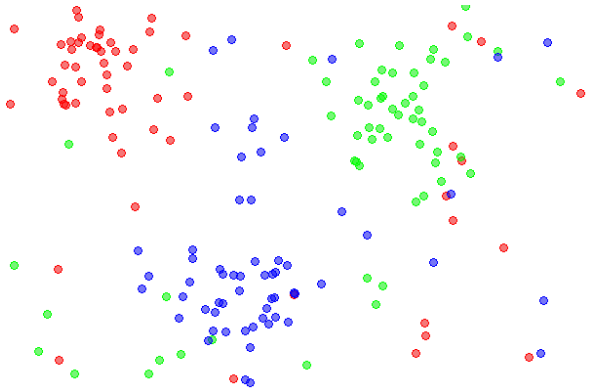
Upload Submission File

k-Nearest Neighbor

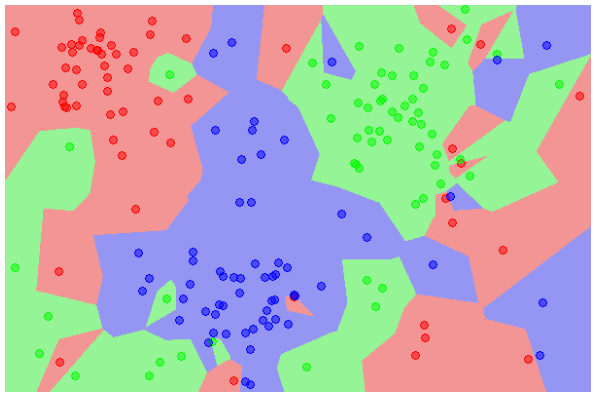
find the k nearest images, have them vote on the label

With larger K Smooth out

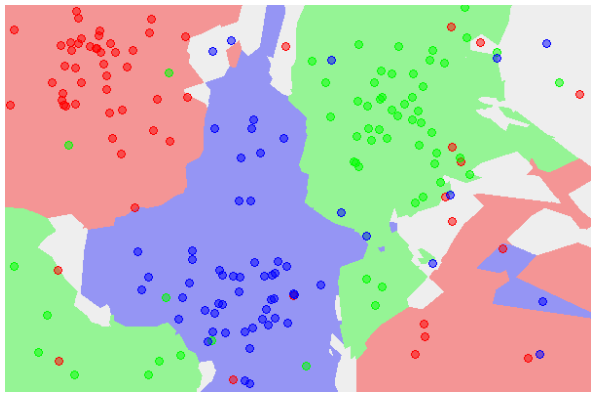
the data



NN classifier



5-NN classifier

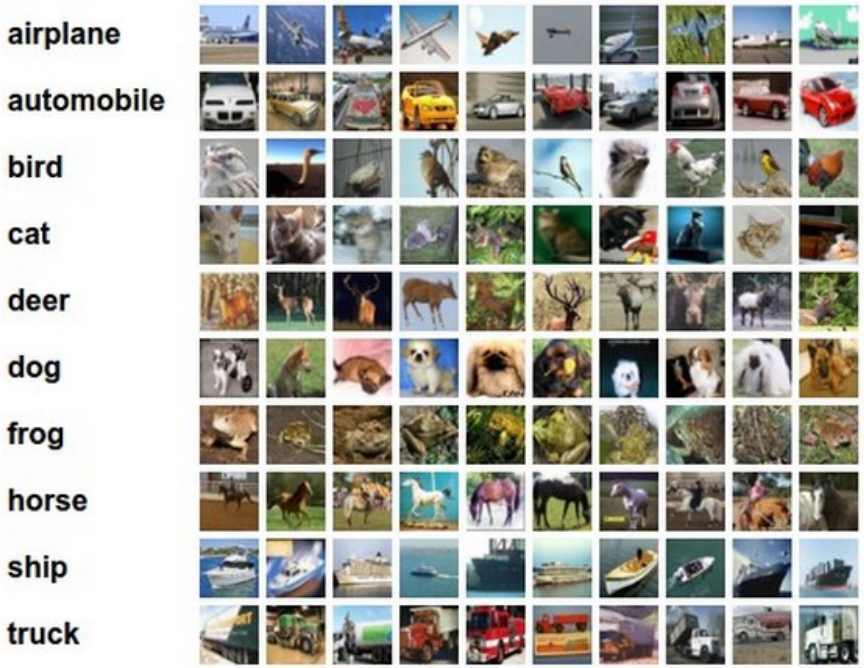


http://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm

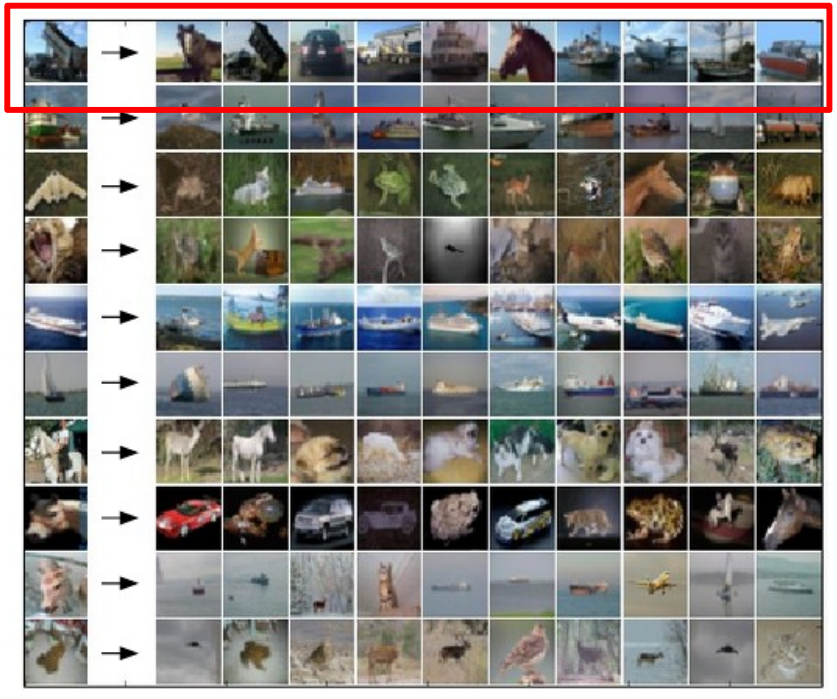
KNN; Majority vote of the neighbors

k-Nearest Neighbor

Example dataset: **CIFAR-10**
10 labels
50,000 training images
10,000 test images.



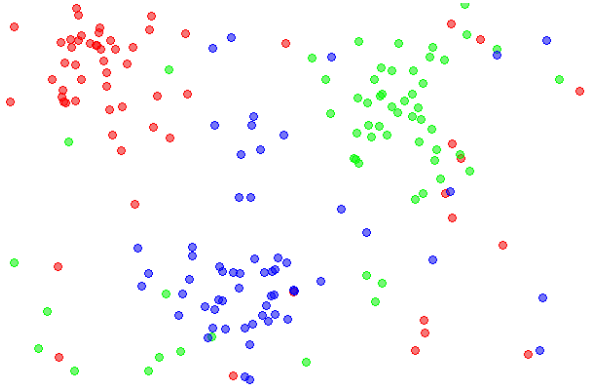
For every test image (first column), examples of nearest neighbors in rows



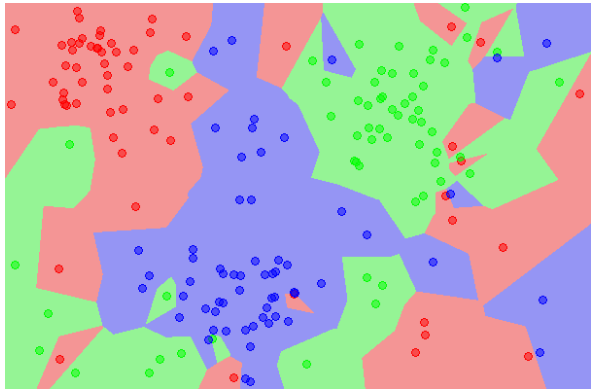
Majority vote over 10 NN to classify the example

k-Nearest Neighbor: Quiz

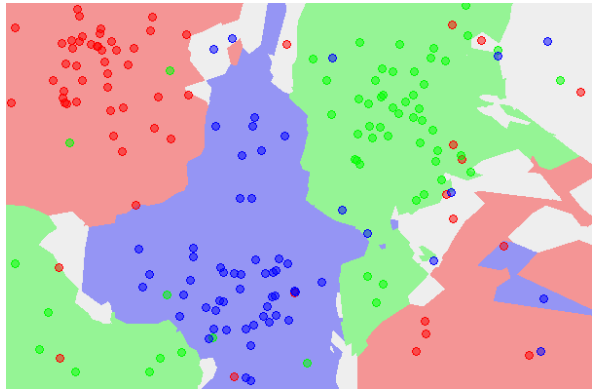
the data



NN classifier



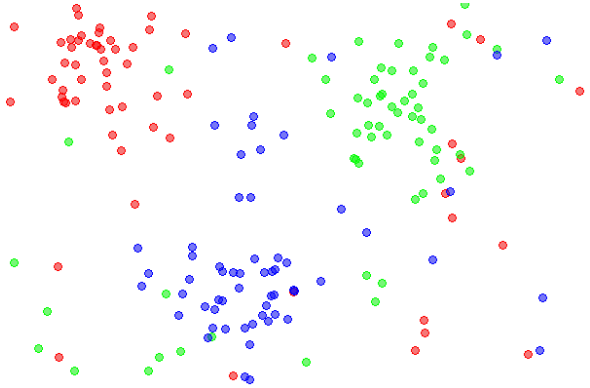
5-NN classifier



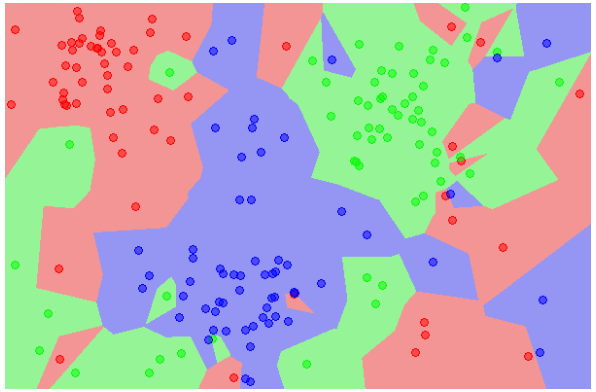
Q: what is the accuracy of the nearest neighbor classifier on the training data, when using the Euclidean distance?
Manhattan Distance (trick question)

Question:

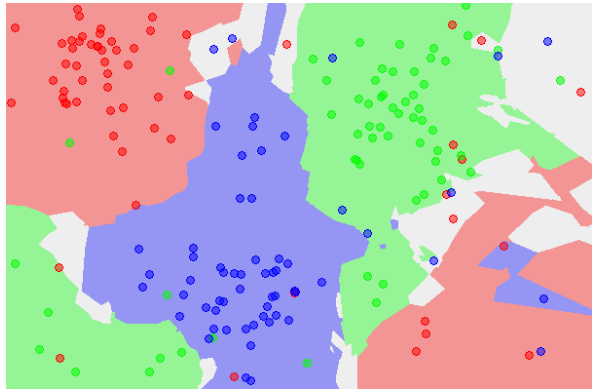
the data



NN classifier

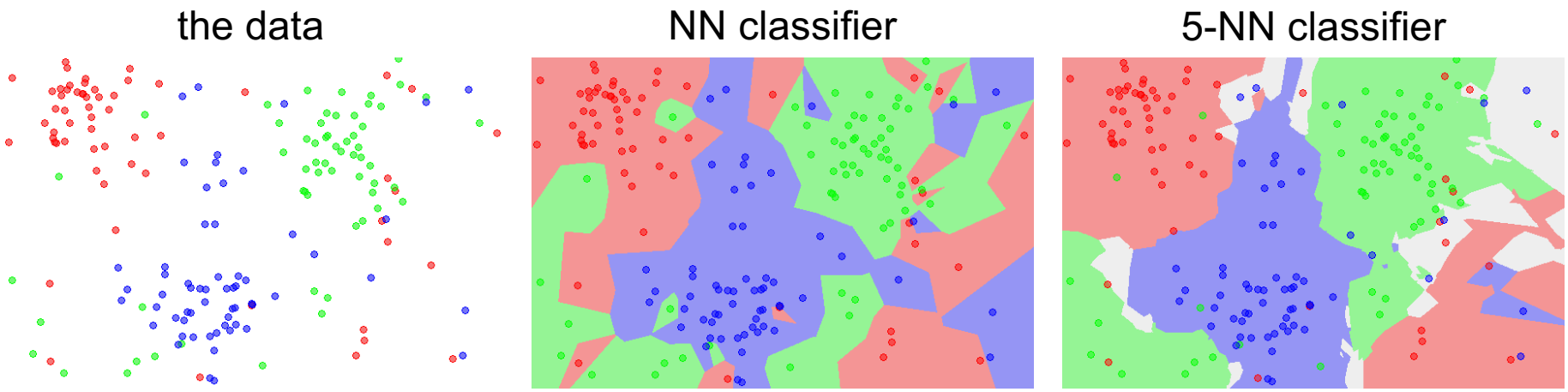


5-NN classifier



Q2: what is the accuracy of the **k-**nearest neighbor classifier on the training data?

Question:



Q2: what is the accuracy of the **k-**nearest neighbor classifier on the training data?

Close to 100% but not always. Why?

Outline

- **Introduction**
- **KNN**
  - Introduction
  - KNN
    - Regression
    - Classification
  - Efficient implementations:
    - k-d trees, parallelism
- **Image classification**
  - KNN for image classification
  - Implementation tricks (SKLearn, pilot datasets)
  - Homegrown KNN; CIFAR-10 Kaggle Challenge
- **Hyperparameter selection via crossfold validation**
- **Summary**

KNN hyperparameters

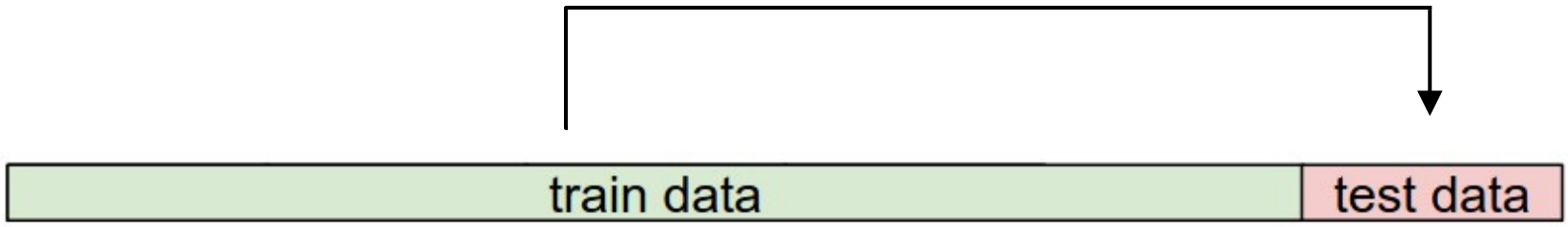
What is the best **distance** to use?
What is the best value of **k** to use?
Which implementation should I use?

- Memory usage; case per second

i.e. how do we set the **hyperparameters**?

Hyperparameters tuning

Try out what hyperparameters work best on test set.



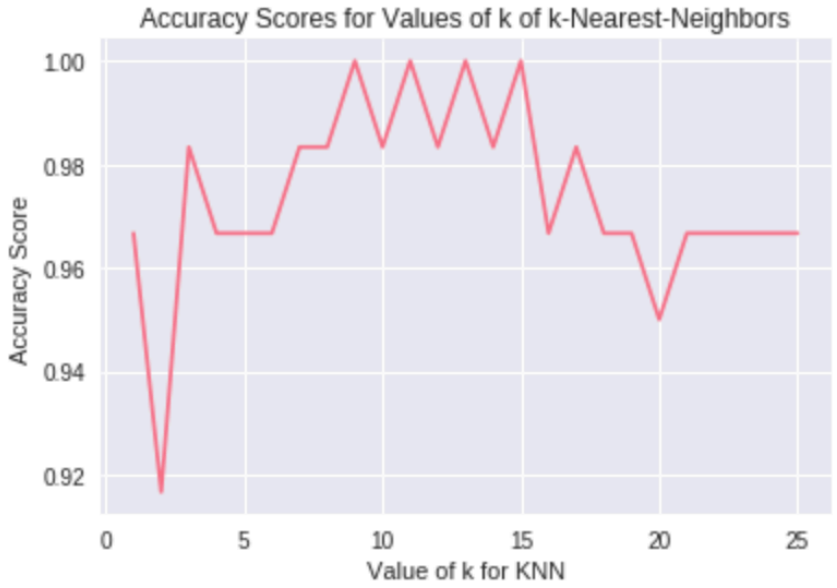
Split data into training and testing

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=5)
print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
```

```
(90, 4)
(90,)
(60, 4)
(60,)
```

```
# experimenting with different n values
k_range = list(range(1,26))
scores = []
for k in k_range:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    y_pred = knn.predict(X_test)
    scores.append(metrics.accuracy_score(y_test, y_pred))

plt.plot(k_range, scores)
plt.xlabel('Value of k for KNN')
plt.ylabel('Accuracy Score')
plt.title('Accuracy Scores for Values of k of k-Nearest-Neighbors')
plt.show()
```

K=12



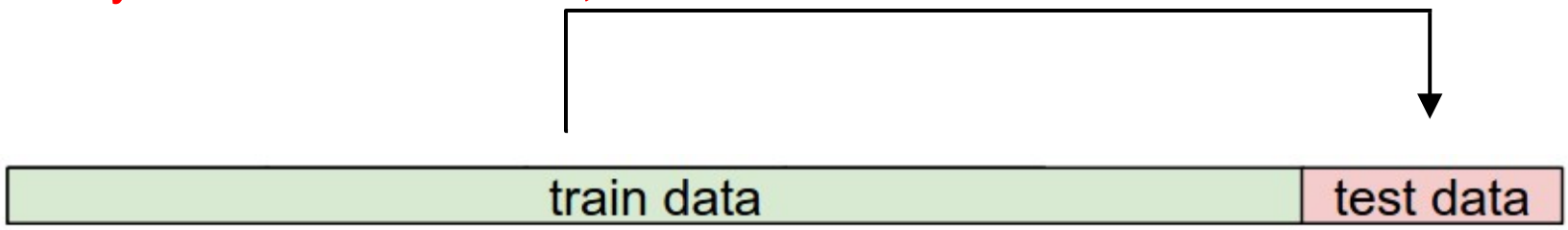
K=12 is best using a train and test split. Better to use a train-validation-test split

Using TEST is a not good!

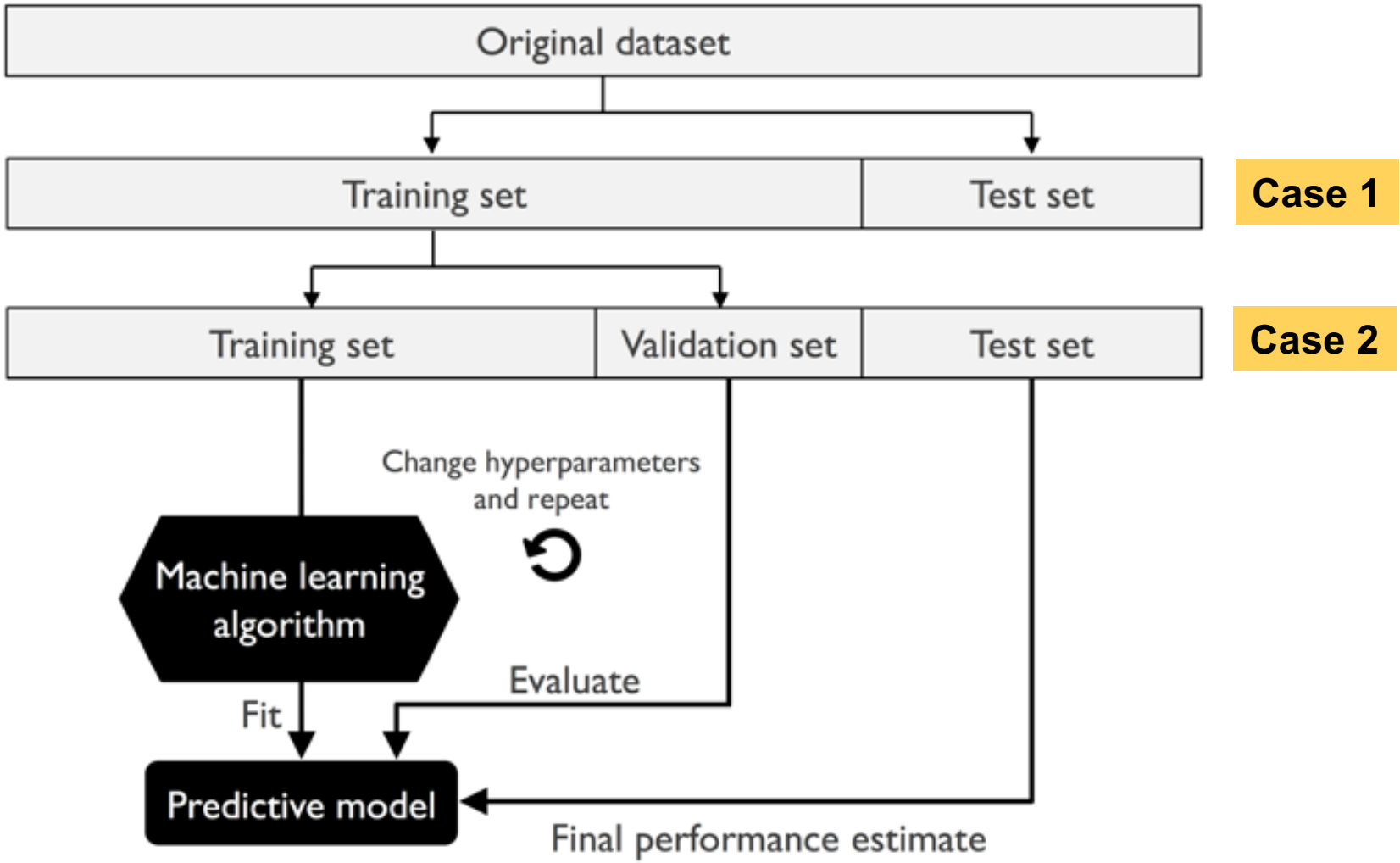
Trying out what hyperparameters work best on test set:

Very bad idea. The test set is a proxy for the generalization performance!

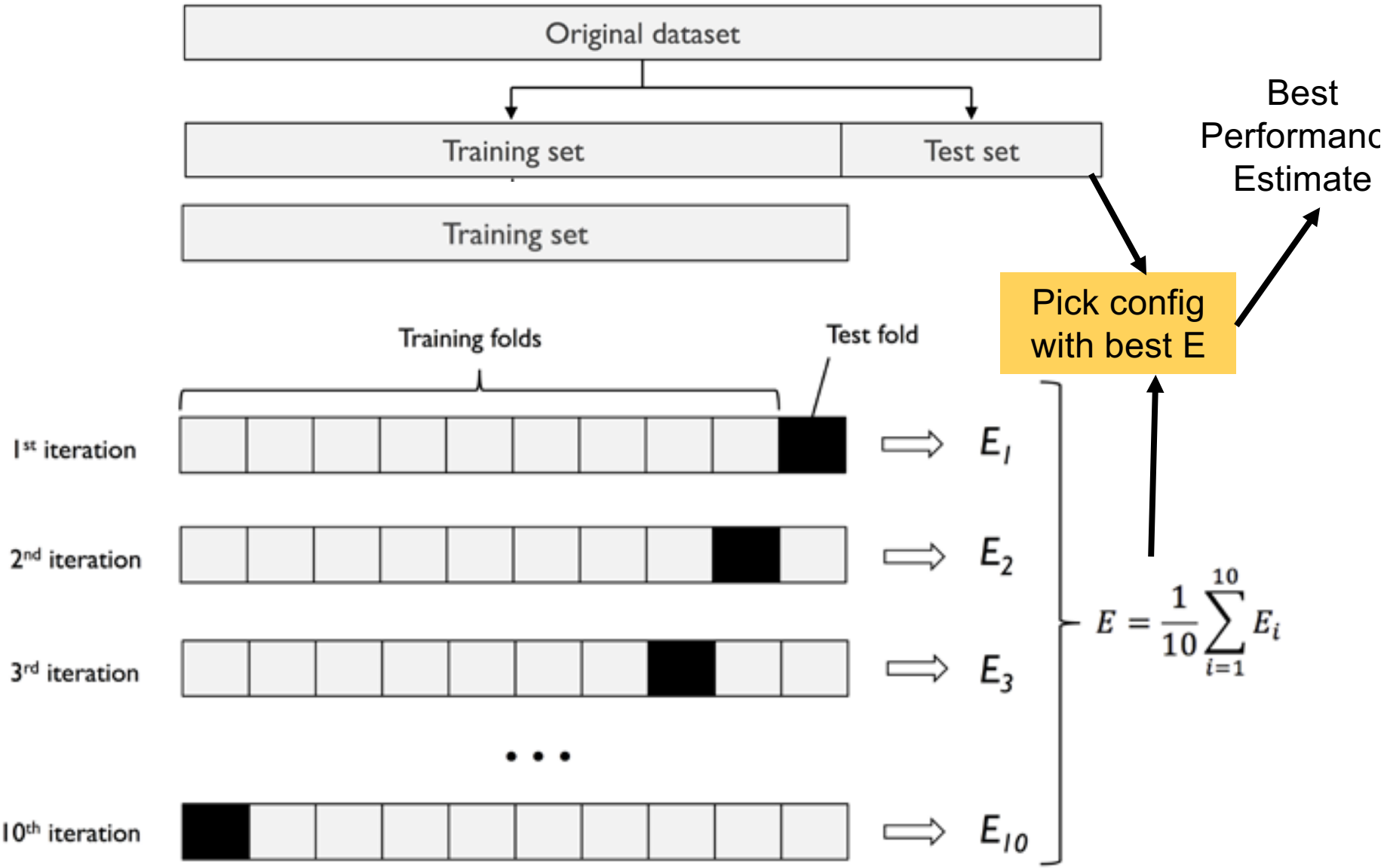
Use only **VERY SPARINGLY,** at the end.



Train-Validation-Test Split



Crossfold validation



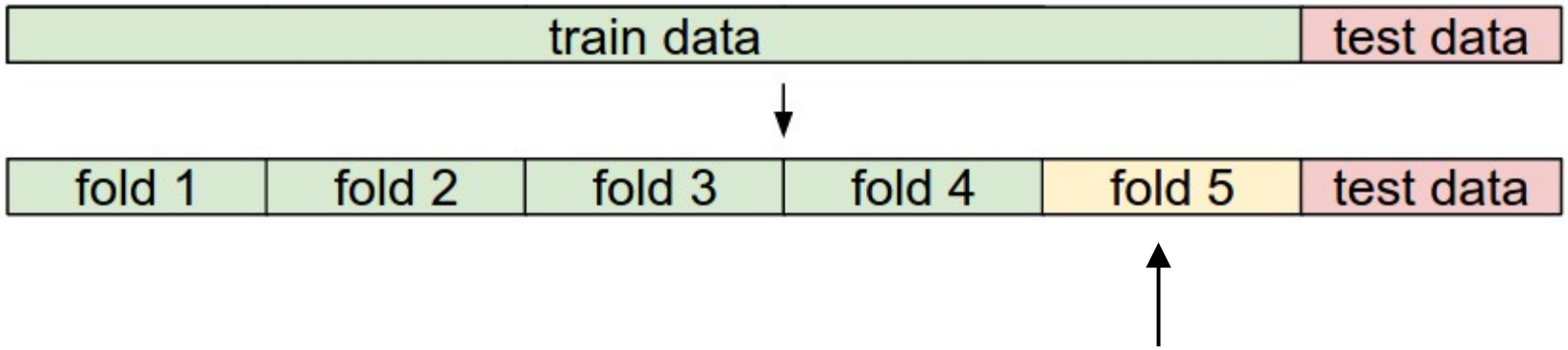
Hyperparameter tuning

What is the best **distance** to use?
What is the best value of **k** to use?

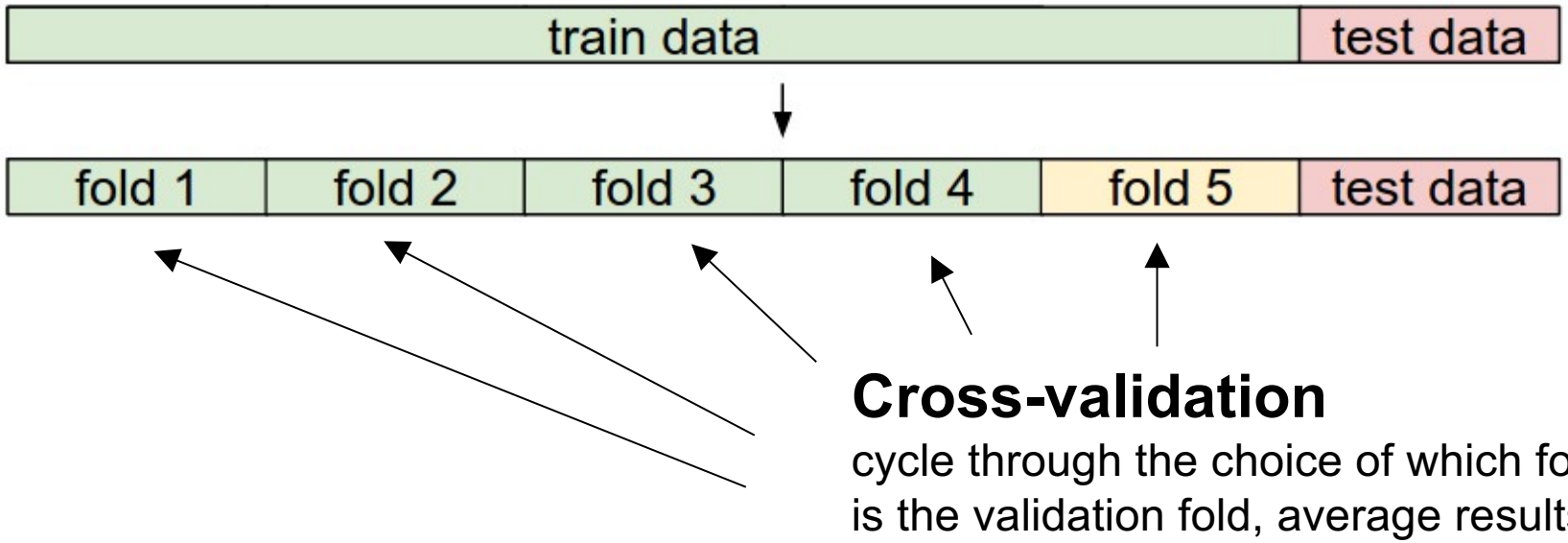
i.e. how do we set the **hyperparameters**?

Very problem-dependent.
Must try them all out and see what works best.

Cross fold validation



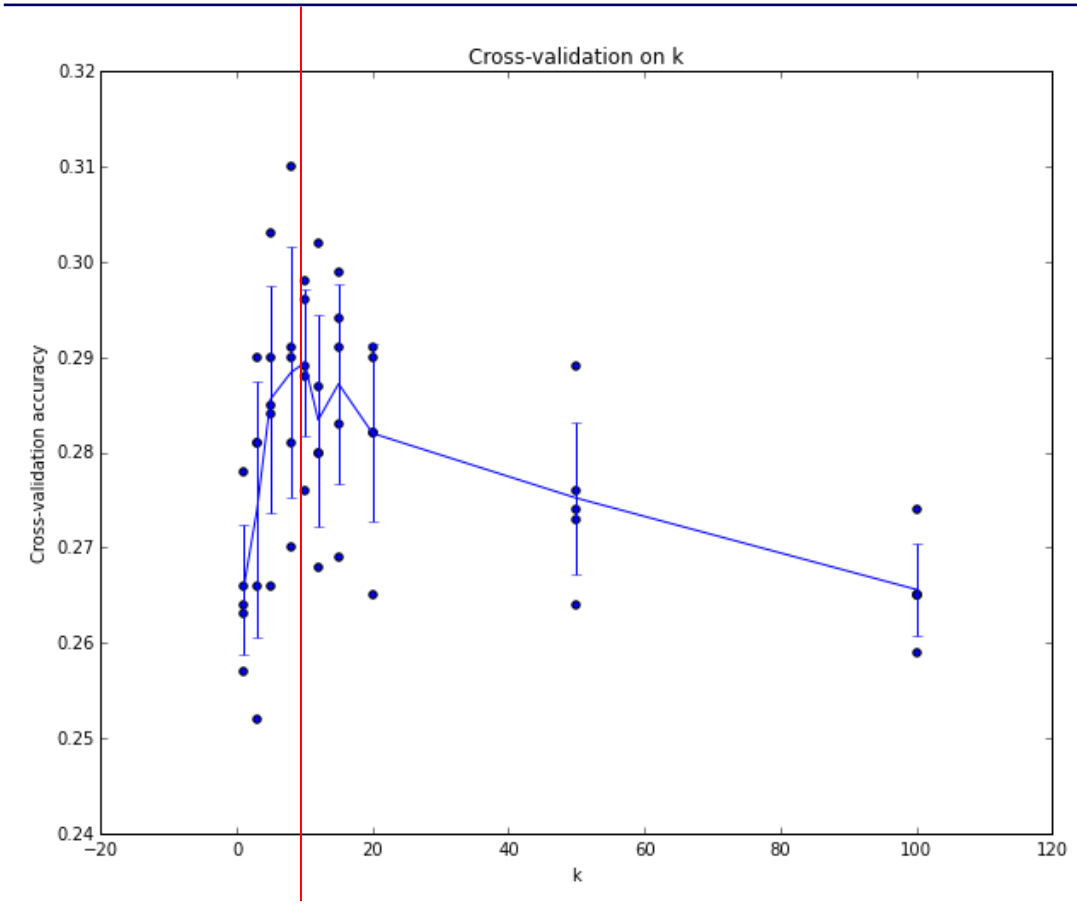
CV on training only!



Cross-validation

cycle through the choice of which fold is the validation fold, average results.

Choosing K in KNN using 5-fold CV



Example of
5-fold cross-validation
for the value of **k.**

Each point: single
outcome.

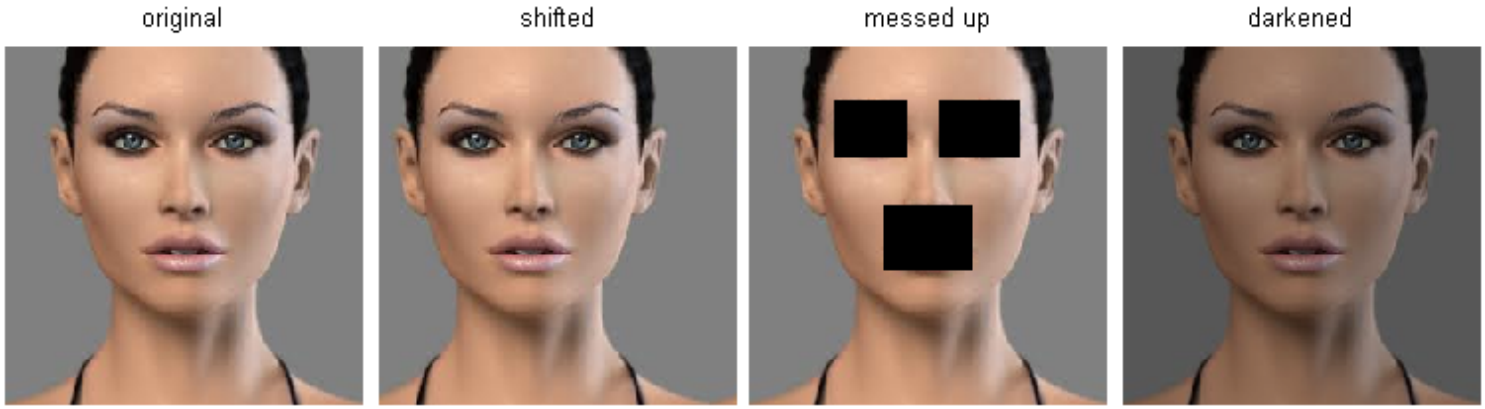
The line goes
through the mean, bars
indicated standard
deviation

(Seems that k ~= 7 works best for this data)

KNN has some problems

KNN is versatile and be applied to many problems
k-Nearest Neighbor on images **never used.**

- terrible performance at test time
- distance metrics on level of whole images can be very unintuitive



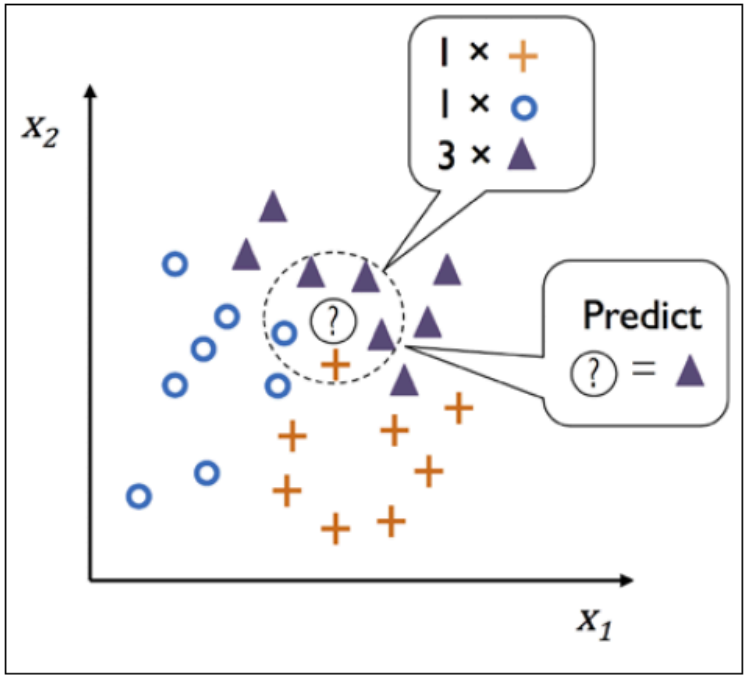
(all 3 images have same L2 distance to the one on the left)

Outline

- **Introduction**
- **KNN**
  - Introduction
  - KNN
    - Regression
    - Classification
  - Efficient implementations:
    - k-d trees, parallelism
- **Image classification**
  - KNN for image classification in Python
  - Implementation tricks (SKLearn, pilot datasets)
  - Homegrown KNN; CIFAR-10 Kaggle Challenge
- **Hyperparameter selection via crossfold validation**
- **Summary**

K nearest Neighbors for classification

- **KNN Summary**
  - 1. Choose the number of **k** and a **distance metric**.
  - 2. Find the k-nearest neighbors of the sample that we want to classify.
  - 3. Assign the class label by majority vote.
  - With **K=5**, and using **Euclidean Distance metric** A new data point (?) is assigned the triangle class label based on majority voting among its five nearest neighbors.



KNN Summary

- **KNN belongs to a subcategory of nonparametric models that is described as instance-based learning.**
- **Models based on instance-based learning are characterized by memorizing the training dataset,**
- **Zero cost during the learning process UNLESS**
  - Memorize the data
  - Build an index using KD-Trees
    - [similar cost as building a decision tree]
    - Deployment goes n to n(LOG(n))

Summary: KNN Classification on images

- **We introduced the k-Nearest Neighbor Classifier, which predicts the labels based on nearest images in the training set**
- **Image Classification: accuracy of predictions:**
  - We are given a Training Set of labeled images, asked to predict labels on Test Set. Common to report the Accuracy of predictions (fraction of correctly predicted images)
- **Use Crossfold validation to selection hyperparameters**
  - We saw that the choice of distance and the value of k are hyperparameters that are tuned using a validation set, or through cross-validation if the size of the data is small.
- **Report performance on test set for best hyperparameters**
  - Once the best set of hyperparameters is chosen, the classifier is evaluated once on the test set, and reported as the performance of kNN on that data.

End of lecture