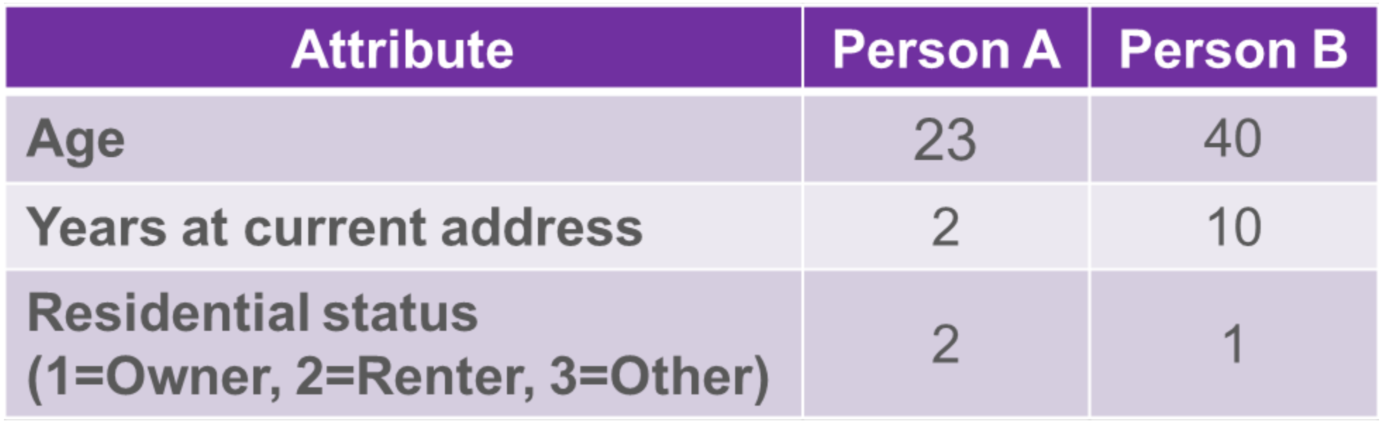
Data Science for Business
Similarity and Nearest Neighbors

Asst. Prof. Teerapong Leelanupab (Ph.D.)
Faculty of Information Technology
King Mongkut's Institute of Technology Ladkrabang (KMITL)

Week 7.1

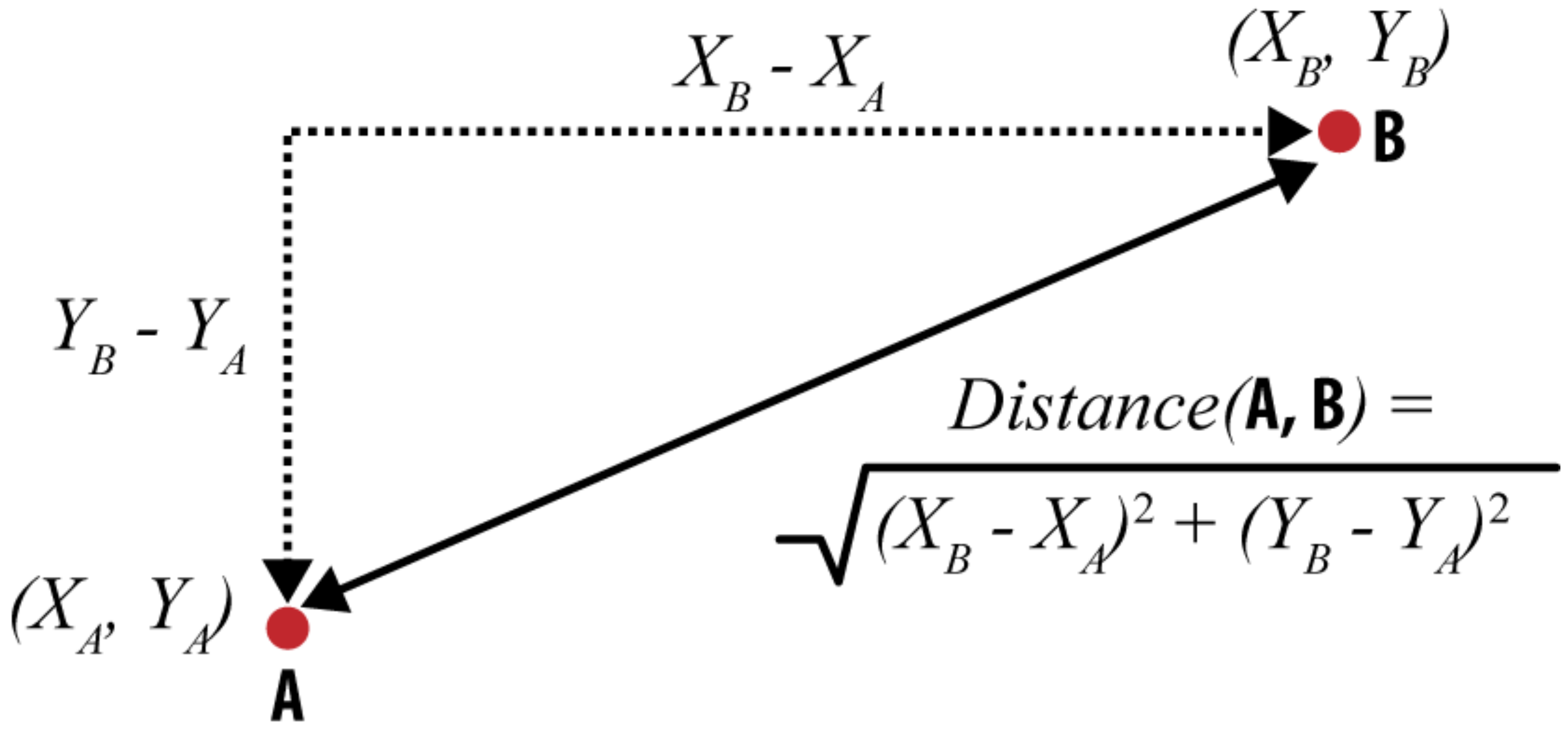
Similarity and Distance

- If two objects can be represented as feature vectors, then we can compute the distance between them



Attribute	Person A	Person B
Age	23	40
Years at current address	2	10
Residential status (1=Owner, 2=Renter, 3=Other)	2	1

Euclidean Distance



$X_B - X_A$
(X_B, Y_B)
B
$Y_B - Y_A$
(X_A, Y_A)
A
$Distance(\mathbf{A}, \mathbf{B}) =$
$\sqrt{(X_B - X_A)^2 + (Y_B - Y_A)^2}$

Euclidean Distance

- $\sqrt{\left(d_{1,A}-d_{1,B}\right)^2+\left(d_{2,A}-d_{2,B}\right)^2+\cdots+\left(d_{n,A}-d_{n,B}\right)^2}$

- $d(A,B)=\sqrt{(23-40)^2+(2-10)^2+(2-1)^2}=18.8$

Other Distance Functions

- $d_{Manhattan}(\boldsymbol{X}, \boldsymbol{Y}) = \|\boldsymbol{X} - \boldsymbol{Y}\|_1 = |x_1 - y_1| + |x_2 - y_2| + \cdots$
- $d_{Jaccard}(X, Y) = 1 - \frac{|X \cap Y|}{|X \cup Y|}$
- $d_{Cosine}(\boldsymbol{X}, \boldsymbol{Y}) = 1 - \frac{\boldsymbol{X} \cdot \boldsymbol{Y}}{\|\boldsymbol{X}\|_2 \cdot \|\boldsymbol{Y}\|_2}$

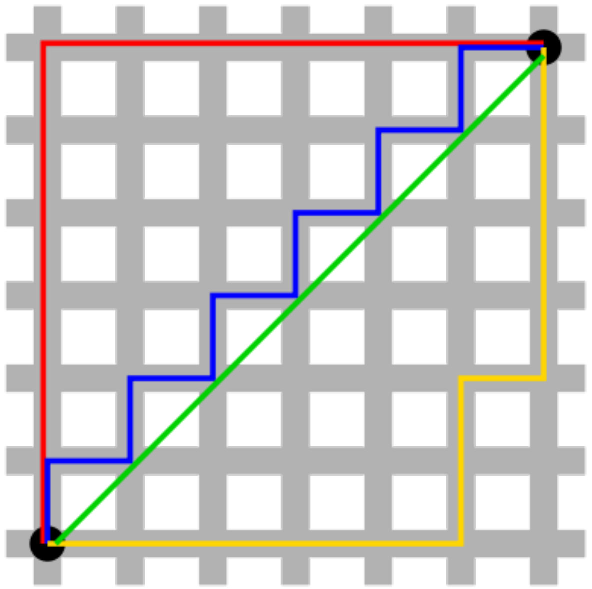
Be careful about two words between distance (*d*) and similarity (*s*)

Manhattan Distance

- Manhattan distance is also known as Manhattan length, taxicab metric, city block distance, rectilinear distance, or snake distance, with corresponding variations in the name of the geometry.
- Formally, it is L_1 distance or ℓ_1 norm

$$\mathrm{dist}(d_i, q) = \left\| d_i - q \right\|_1 = \sum_{j=1}^{t} \left| d_{i,j} - q_j \right|$$

https://en.wikipedia.org/wiki/Taxicab_geometry



Manhattan (Taxicab) geometry versus Euclidean distance: In taxicab geometry, the red, yellow, and blue paths all have the shortest length of 12. In Euclidean geometry, the green line has length 62≈8.49, and is the unique shortest path.

Jaccard’s Coefficient (Similarity)

- **Jaccard (similarity) coefficient or Jaccard Index**
  - named after Paul Jaccard
  - aka. **Tanimoto Similarity (One form of Tanimoto Similarities)**
  - Jaccard coefficient measures similarity between two finite sample sets, and is defined as the size of the intersection divided by the size of the union of the sample sets
  - The value is also in the range 0 to 1

$$\mathrm{Jaccard}(X, Y) = \frac{|X \cap Y|}{|X \cup Y|}$$

Jaccard’s Distance (Dissimmilarity)

- The value of Jaccard coefficient is in the range of [0,1].

“If *X* and *Y* are both empty, we define Jaccard(*X*,*Y*)=1.”
Clearly,

$$0 \leq \text{Jaccard}(X,Y) \leq 1$$

- **Jaccard distance,** measures dissimilarity between sample sets, is obtained by subtracting the Jaccard coefficient from 1

$$\boxed{\text{dist}(X,Y)} = \boxed{1 - \text{Jaccard}(X,Y)} = \frac{|X \cup Y| - |X \cap Y|}{|X \cup Y|}$$

Cosine Distance

- Inverted similarity measure = Cosine correlation/measure
- The numerator of the cosine measure is the *dot product* or *inner product,* i.e. **d**$_i$ • **q**.
- The denominator normalizes the score of dot product by dividing by the product of the lengths of the two vectors.

ผลคูณความยาวของ 2 เวกเตอร์

$$\text{sim}(d_i, q) = \boxed{\text{cosine}\,\theta_{\mathbf{d_i},\mathbf{q}}} = \frac{\mathbf{d_i} \bullet \mathbf{q}}{\|\mathbf{d_i}\|\,\|\mathbf{q}\|} = \frac{\sum_{j=1}^{t} d_{i,j} \times q_j}{\sqrt{\sum_{j=1}^{t} {d_{i,j}}^2 \times \sum_{j=1}^{t} {q_j}^2}}$$

$$\text{dist}(d_i, q) = 1 - \text{sim}(d_i, q)$$

Example: “Whiskey Analytics”

1. **Color:** *yellow, very pale, pale, pale gold, gold, old gold, full gold, amber,* etc. (14 values)
2. **Nose:** *aromatic, peaty, sweet, light, fresh, dry, grassy,* etc. (12 values)
3. **Body:** *soft, medium, full, round, smooth, light, firm, oily.* (8 values)
4. **Palate:** *full, dry, sherry, big, fruity, grassy, smoky, salty,* etc. (15 values)
5. **Finish:** *full, dry, warm, light, smooth, clean, fruity, grassy, smoky,* etc. (19 values)

Whiskey	Distance	Descriptors
Bunnahabhain	—	*gold; firm,med,light; sweet,fruit,clean; fresh,sea; full*
Glenglassaugh	0.643	gold; firm,light,smooth; sweet,grass; fresh,grass
Tullibardine	0.647	gold; firm,med,smooth; sweet,fruit,full,grass,clean; sweet; big,arome,sweet
Ardbeg	0.667	sherry; firm,med,full,light; sweet; dry,peat,sea;salt
Bruichladdich	0.667	pale; firm,light,smooth; dry,sweet,smoke,clean; light; full
Glenmorangie	0.667	p.gold; med,oily,light; sweet,grass,spice; sweet,spicy,grass,sea,fresh; full,long

Data Science for Business

Nearest Neighbour Classifiers

Asst. Prof. Teerapong Leelanupab (Ph.D.)
Faculty of Information Technology
King Mongkut's Institute of Technology Ladkrabang (KMITL)

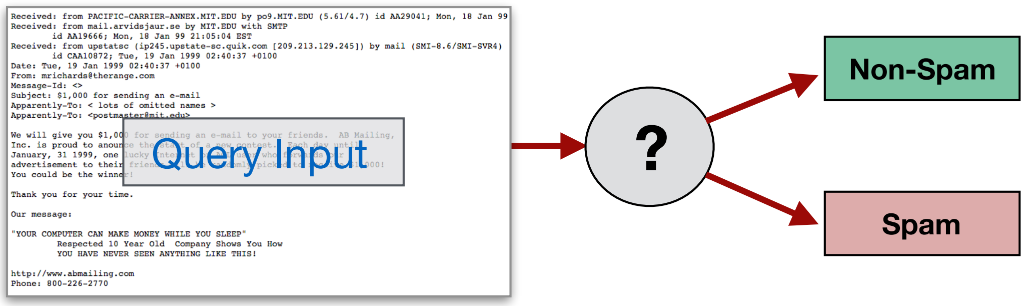
Week 7.2

Overview

- Eager v Lazy Classification Strategies
- Distance-based Models
- Feature Spaces
- Measuring Distance
- Data Normalisation
- Nearest Neighbours
- *k*-Nearest Neighbour Classifier (kNN)
- Weighted kNN
- kNN in **scikit-learn**

Reminder: Classification

- Supervised Learning: Algorithm that learns a function from manually-labelled training examples.
- Classification: Training examples, usually represented by a set of descriptive features, help decide the *class* to which a new unseen query input belongs.
- Binary Classification: Assign one of two possible target class labels to the new query input.



```
Received: from PACIFIC-CARRIER-ANNEX.MIT.EDU by po9.MIT.EDU (5.61/4.7) id AA29041; Mon, 18 Jan 99
Received: from mail.arvidsjaur.se by MIT.EDU with SMTP
        id AA19666; Mon, 18 Jan 99 21:05:04 EST
Received: from upstatsc (ip245.upstate-sc.quik.com [209.213.129.245]) by mail (SMI-8.6/SMI-SVR4)
        id CAA10872; Tue, 19 Jan 1999 02:40:37 +0100
Date: Tue, 19 Jan 1999 02:40:37 +0100
From: mrichards@therange.com
Message-Id: <>
Subject: $1,000 for sending an e-mail
Apparently-To: < lots of omitted names >
Apparently-To: <postmaster@mit.edu>

We will give you $1,000 for sending an e-mail to your friends.  AB Mailing,
Inc. is proud to anounce the start of a new contest.  Each day until
January, 31 1999, one lucky Internet or AOL user who forwards your
advertisement to their friends will be randomly picked to receive $1,000!
You could be the winner!

Thank you for your time.

Our message:

"YOUR COMPUTER CAN MAKE MONEY WHILE YOU SLEEP"
         Respected 10 Year Old  Company Shows You How
         YOU HAVE NEVER SEEN ANYTHING LIKE THIS!

http://www.abmailing.com
Phone: 800-226-2770
```

- Multiclass Classification: Assign one of *M>2* possible target class labels to the new query input.

Eager v Lazy Classifiers

decision tree is eager because you will have a "tree" after you ran model (tree = model)

- Eager Learning Classification Strategy
  - Classifier builds a full model during an initial training phase, to use later when new query examples arrive. $\rightarrow$ unseen data
  - More offline setup work, less work at run-time.
  - Generalise before seeing the query example.

- ถูก build ไว้ล่วงหน้า เวลามี unseen data มาก็โยนเข้าโมเดลได้เลย
- มีการจูนตลอด เพราะจะได้รับกับ data ใหม่ๆได้

- Lazy Learning Classification Strategy
  - Classifier keeps all the training examples for later use.
  - Little work is done offline, wait for new query examples.
  - Focus on the local space around the examples.

- Distance-based Models: Many learning algorithms are based on generalising from training data to unseen data by exploiting the distances (or similarities) between the two.

Example: Athlete Selection

- Dataset of performance ratings for 20 college athletes.
- Describe each athlete by 2 continuous features: *speed*, *agility.* Binary class label indicates whether or not they were *selected* for the college team ('Yes' or 'No').

Athlete	***Speed***	***Agility***	***Selected***
x1	2.50	6.00	No
x2	3.75	8.00	No
x3	2.25	5.50	No
x4	3.25	8.25	No
x5	2.75	7.50	No
x6	4.50	5.00	No
x7	3.50	5.25	No
x8	3.00	3.25	No
x9	4.00	4.00	No
x10	4.25	3.75	No

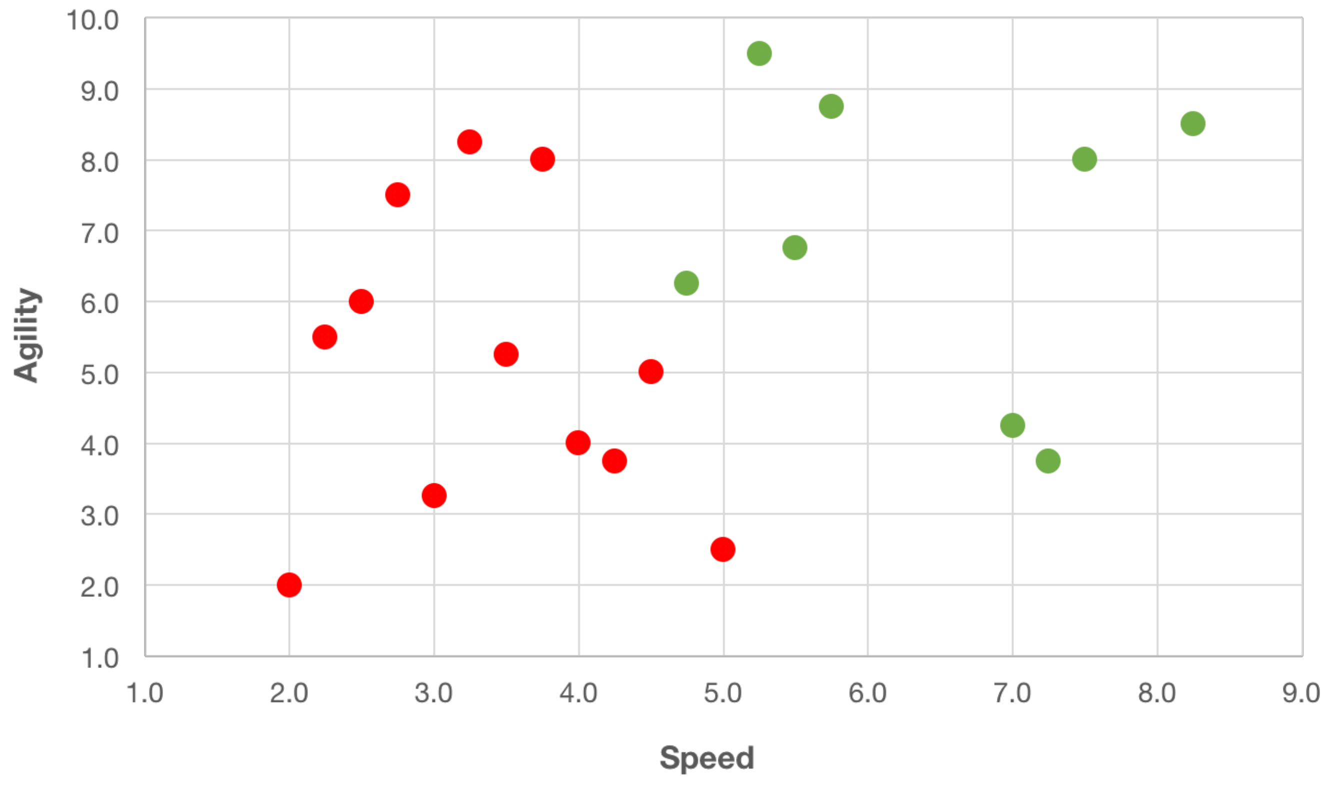
Athlete	***Speed***	***Agility***	***Selected***
x11	2.00	2.00	No
x12	5.00	2.50	No
x13	8.25	8.50	Yes
x14	5.75	8.75	Yes
x15	4.75	6.25	Yes
x16	5.50	6.75	Yes
x17	5.25	9.50	Yes
x18	7.00	4.25	Yes
x19	7.50	8.00	Yes
x20	7.25	3.75	Yes

Q. Will athlete **q** be selected?

Athlete	***Speed***	***Agility***	***Selected***
q	3.00	8.00	???

Feature Spaces

We can use the feature values to visually position the 20 athletes in a 2-dimensional coordinate space (i.e. *agility* versus *speed*):



2 features describing each example (agility & speed)

➡ 2 coordinate dimensions for measuring similarity/ distance

Features Space: A *D*-dimensional coordinate space used to represent the input examples for a given problem, with one coordinate for each descriptive feature.

Measuring Distance

- Distance function: A suitable function to measure how distant (or similar) two input examples are from one another are in some *D*-dimensional feature space.
- Local distance function: Measure the distance between two examples based on a single feature.

Athlete	*Speed*	*Agility*
x1	2.50	6.00
x2	3.75	8.00

  - e.g. what is distance between **x1** and **x2** in terms of *Speed*?
  - e.g. what is distance between **x1** and **x2** in terms of *Agility*?
- Global distance function: Measure the distance between two examples based on the combination of the local distances across all features.
  - e.g. what is distance between **x1** and **x2** based on both *Speed* and *Agility*?

Measuring Distance

- Overlap function: Simplest local distance measure. Returns 0 if the two values for a feature are equal and 1 otherwise. Generally suitable for categorical data.

Athlete	*Gender*	*Nationality*
x1	Female	Irish
x2	Male	Irish
x3	Male	Italian

For feature *Gender*

$d_g(x1,x2) = 1$
$d_g(x1,x3) = 1$
$d_g(x2,x3) = 0$

For feature *Nationality*

$d_n(x1,x2) = 0$
$d_n(x1,x3) = 1$
$d_n(x2,x3) = 1$

humming = x (intersect) y in Jaccard

- Hamming distance: Global distance function which is the sum of the overlap differences across all features - i.e. number of features on which two examples disagree.

$d(x1,x2) = 1 + 0 = 1$
$d(x1,x3) = 1 + 1 = 2$
$d(x2,x3) = 0 + 1 = 1$

Overlap distance for *Gender* +
Overlap distance for *Nationality*

Measuring Distance

- Absolute difference: For numeric data, we can calculate absolute value of the difference between values for a feature.

Athlete	*Speed*	*Agility*
x1	2.50	6.00
x2	3.75	8.00
x3	2.25	5.50

For feature *Speed*

```
ds(x1,x2) = |2.50-3.75| = 1.25
ds(x1,x3) = |2.50-2.25| = 0.25
ds(x2,x3) = |3.75-2.25| = 1.5
```

For feature *Agility*

```
ds(x1,x2) = |6.0-8.0| = 2.0
ds(x1,x3) = |6.0-5.5| = 0.5
ds(x2,x3) = |8.0-5.5| = 2.5
```

- Again we can compute a global distance between two examples by summing the local distances over all features.

```
d(x1,x2) = 1.25 + 2.0 = 3.25
d(x1,x3) = 0.25 + 0.5 = 0.75
d(x2,x3) = 1.5 + 2.5 = 4.0
```

Absolute difference for *Speed* +
Absolute difference for *Agility*

- For *ordinal features*, calculate the absolute value of the difference between the two positions in the ordered list of possible values.

e.g. Ordinal Feature *Dosage*:
{Low,Medium,High} = {1, 2, 3}

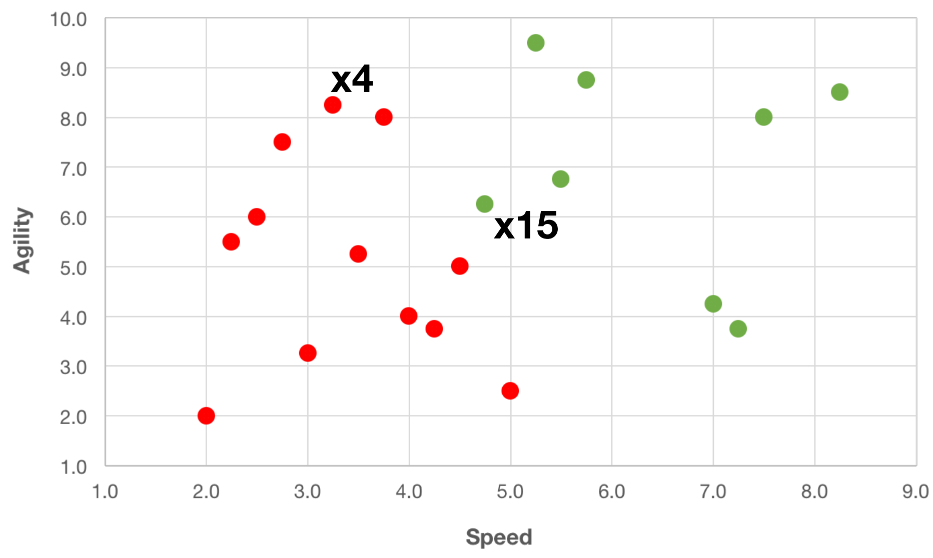
```
diff(Low,High) = |1-3| = 2
diff(Medium,Low) = |2-1| = 1
diff(High,High) = |3-3| = 0
```

Measuring Distance

- Euclidean distance: Most common measure used to quantify distance between two examples with real-valued features.
- The "straight line" distance between two points in a Euclidean coordinate space - e.g. a feature space.
- Calculated as square root of sum of squared differences for each feature *f* representing a pair of examples.

$$\mathrm{ED}(\mathbf{p},\mathbf{q}) = \sqrt{\sum_{f\in F}(q_f - p_f)^2}$$

Athlete	*Speed*	*Agility*
x4	3.25	8.25
x15	4.75	6.25



$$ED(x4,x15) = \sqrt{(3.25-4.75)^2+(8.25-6.25)^2} = \sqrt{6.25} = 2.5$$

Heterogeneous Distance Functions

- In many datasets, the features associated with examples will have different types (e.g. continuous, categorical, ordinal etc).
- We can create a global measure from different local distance functions, using an appropriate function for each feature.

Athlete	*Speed*	*Agility*	*Gender*	*Nationality*
x1	2.50	6.00	Female	Irish
x2	3.75	8.00	Male	Irish
x3	2.25	5.50	Male	Italian

Use absolute difference for continuous features *Speed* & *Agility*

Use overlap for categorical features *Gender* & *Nationality*

```
d(x1,x2) = 1.25 + 2.0 + 1 + 0 = 4.25
d(x1,x3) = 0.25 + 0.5 + 1 + 1 = 2.75
d(x2,x3) = 1.5 + 2.5 + 0 + 1 = 5.0
```

Global distance calculated as sum over individual local distances

- Often domain expertise is required to choose an appropriate distance function for a particular dataset.

good for descriptive feature compare
bad for decreasing error

Data Normalisation

- Numeric features often have different ranges, which can skew certain distance functions.
- So that all features have similar range, we apply *feature normalisation*.

Example	Age
x1	24
x2	19
x3	50
x4	40
x5	23
x6	68
x7	45
x8	33
x9	80
x10	58

- Min-max normalisation:
  Use min and max values for a given feature to rescale to the range [0,1]

$$z_i = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

- Example: Feature *Age*

$$\min(x) = 19$$

$$\max(x) = 80$$

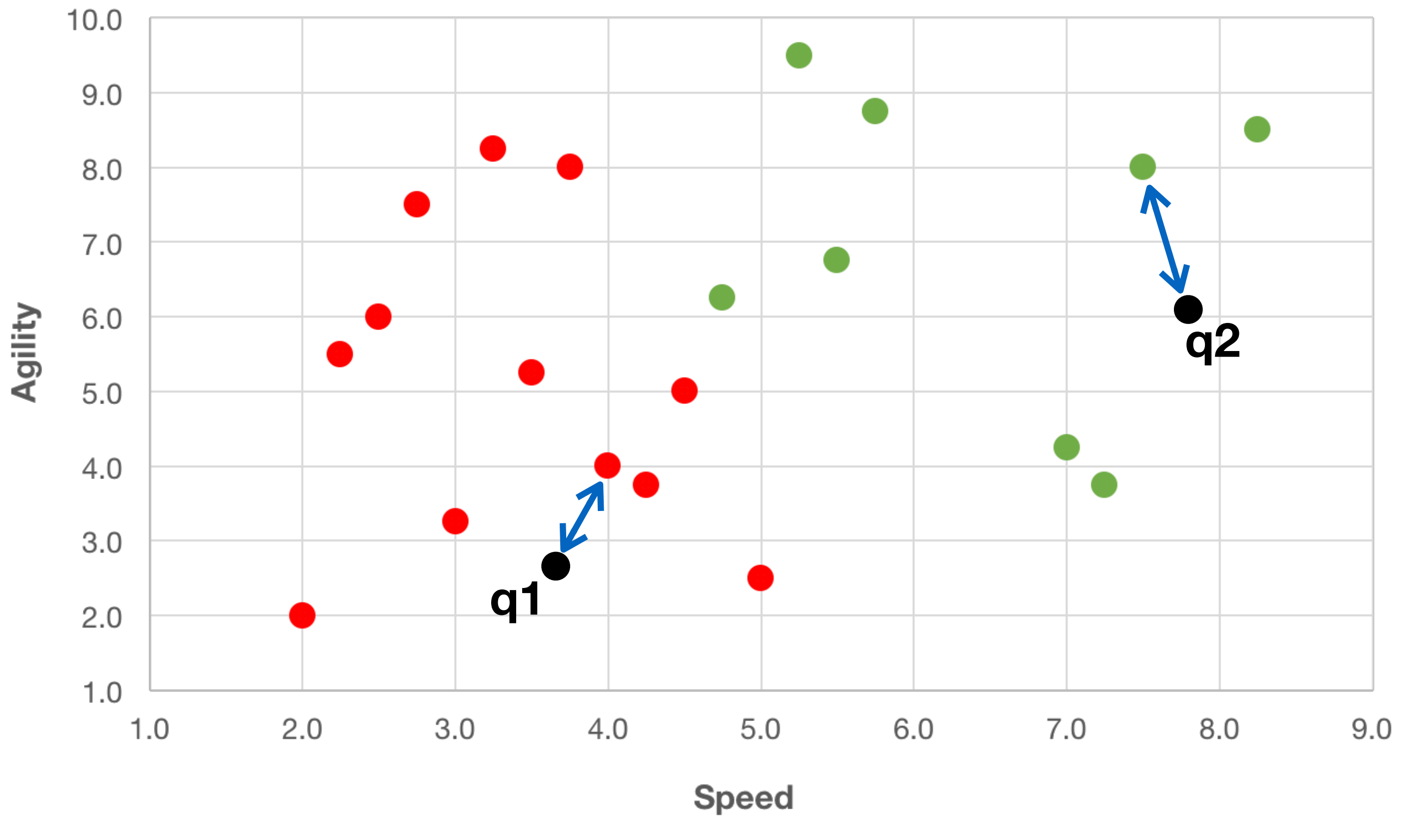
$$\max(x) - \min(x) = 61$$

Age (Non-normalised)	24	19	50	40	23	68	45	33	80	58
Age (Normalised)	0.08	0.00	0.51	0.34	0.07	0.80	0.43	0.23	1.00	0.64

Nearest Neighbour Classifier

Lazy Learning approach: Do not build a model for the data. Identify most similar previous example(s) from the training set for which a label has already been assigned, using some distance function.

Nearest neighbour rule (1NN): For a new query input **q**, find a single labelled example **x** closest to **q**, and assign **q** the same label as **x**.



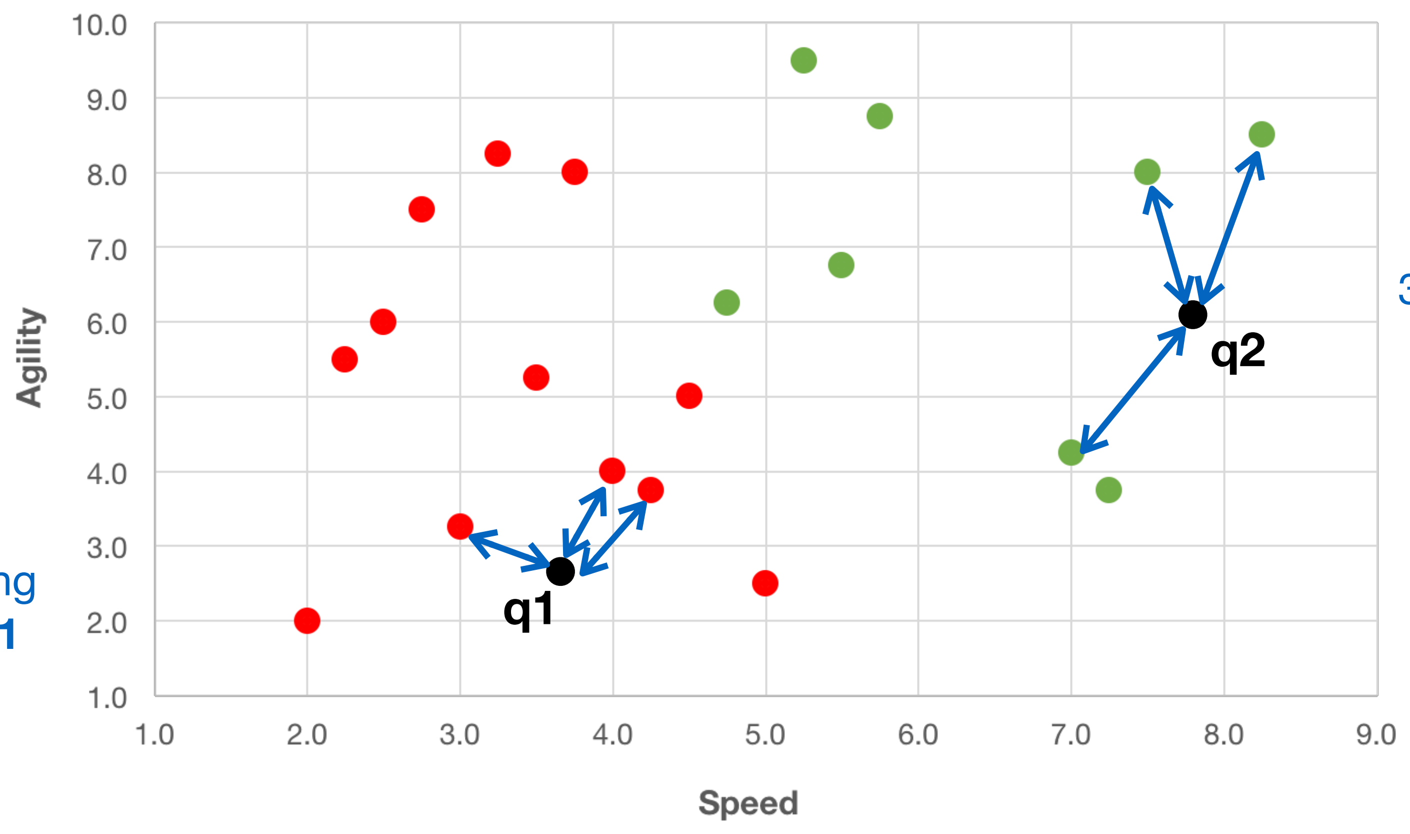
Assign **q1** to 'No' class

Assign **q2** to 'Yes' class

k-Nearest Neighbour Classifier

k-Nearest neighbours (kNN): The NN approach naturally generalises to the case where we use *k* nearest neighbours from the training set to assign a label to a new query input.

Example: For new query inputs, calculate distance to all training examples. Find *k=3* nearest examples (i.e. with smallest distances).



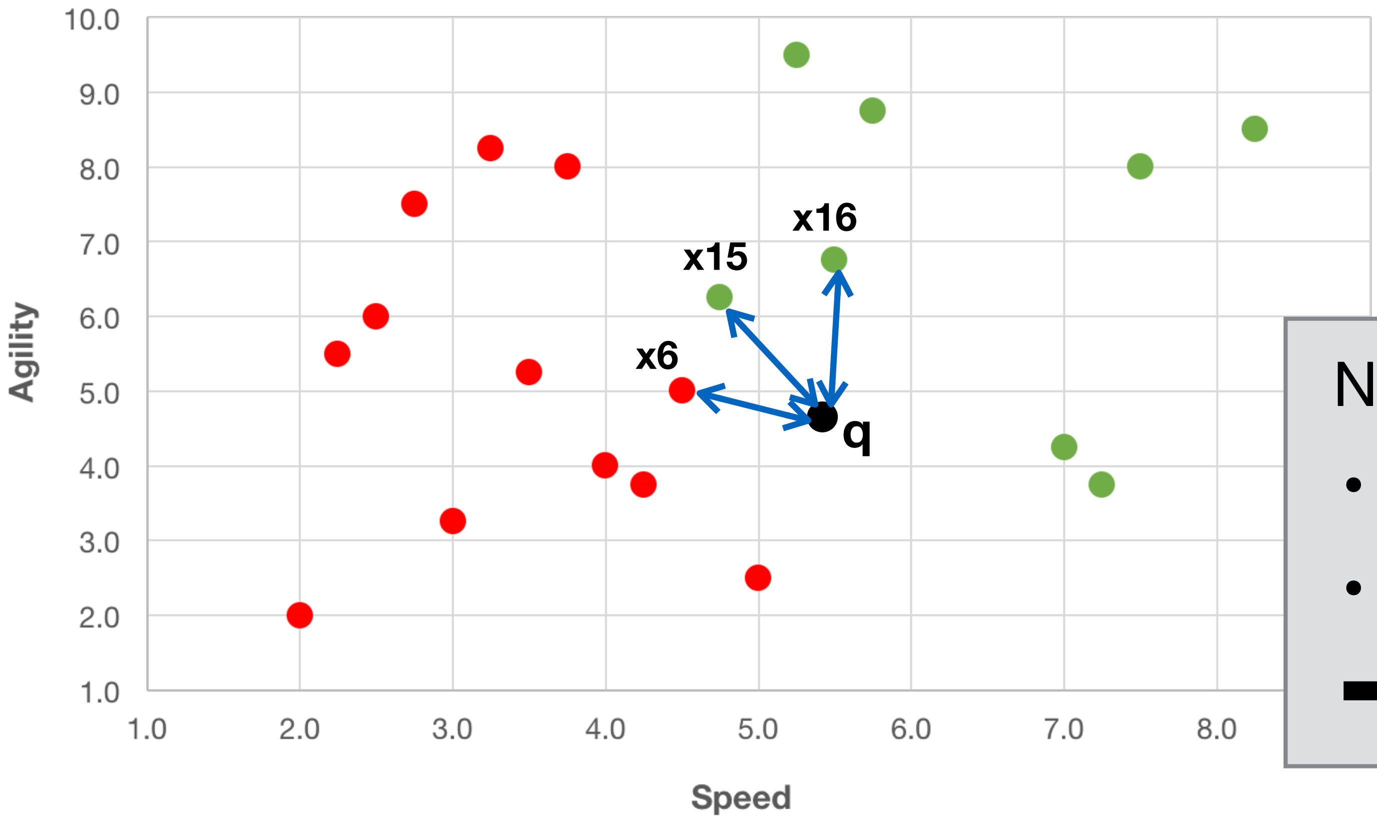
3 nearest training examples to **q2**

3 nearest training examples to **q1**

k-Nearest Neighbour Classifier

Majority voting: The decision on a label for a new query example is decided based on the “votes” of its *k* nearest neighbours. The label for the query is the majority label of its neighbours.

Example: Measure distance from **q** to all training examples. Find the *k=3* nearest examples, and use their labels as votes.



Neighbour counts

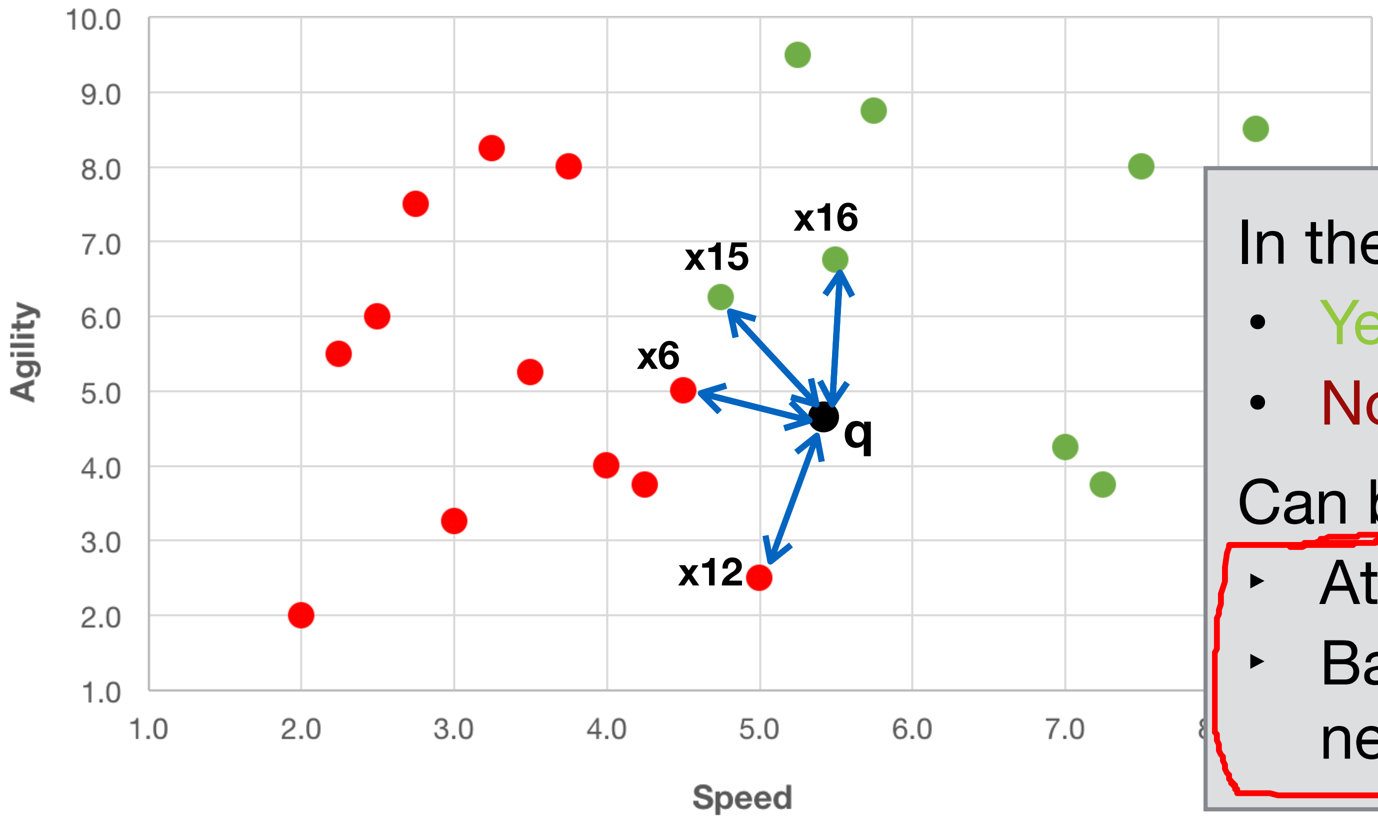
- Yes = 2 votes
- No = 1 vote

➡ Majority says Yes!

k-Nearest Neighbour Classifier

Majority voting: The decision on a label for a new query example is decided based on the “votes” of its *k* nearest neighbours. The label for the query is the majority label of its neighbours.

Example: Measure distance from **q** to all training examples. Find the *k=4* nearest examples, and use their labels as votes.



In the case that…

- Yes = 2 votes
- No = 2 votes

Can break ties…

- At random
- Based on sum of neighbour distances

Example: kNN Classification (k=3)

- Training set of 20 athletes - 8 labelled as 'Yes', 12 as 'No'.
- Each athlete described by 2 continuous features: *Speed*, *Agility*
  Euclidean distance would be an appropriate distance function.

Athlete	*Speed*	*Agility*	*Selected*
x1	2.50	6.00	No
x2	3.75	8.00	No
x3	2.25	5.50	No
x4	3.25	8.25	No
x5	2.75	7.50	No
x6	4.50	5.00	No
x7	3.50	5.25	No
x8	3.00	3.25	No
x9	4.00	4.00	No
x10	4.25	3.75	No

Athlete	*Speed*	*Agility*	*Selected*
x11	2.00	2.00	No
x12	5.00	2.50	No
x13	8.25	8.50	Yes
x14	5.75	8.75	Yes
x15	4.75	6.25	Yes
x16	5.50	6.75	Yes
x17	5.25	9.50	Yes
x18	7.00	4.25	Yes
x19	7.50	8.00	Yes
x20	7.25	3.75	Yes

Will a new input example **q** be labelled as 'Yes' or 'No'?

Athlete	*Speed*	*Agility*	*Selected*
q	5.00	8.00	???

Example: kNN Classification (k=3)

- Measure distance between **q** and all 20 training examples.

Athlete	***Speed***	***Agility***	***Selected***	***Distance***
x1	2.50	6.00	No	2.915
x2	3.75	8.00	No	1.346
x3	2.25	5.50	No	3.400
x4	3.25	8.25	No	1.904
x5	2.75	7.50	No	2.250
x6	4.50	5.00	No	2.550
x7	3.50	5.25	No	2.704
x8	3.00	3.25	No	4.697
x9	4.00	4.00	No	3.640
x10	4.25	3.75	No	3.824

Athlete	***Speed***	***Agility***	***Selected***	***Distance***
x11	2.00	2.00	No	6.265
x12	5.00	2.50	No	5.000
x13	8.25	8.50	Yes	3.400
x14	5.75	8.75	Yes	1.458
x15	4.75	6.25	Yes	1.275
x16	5.50	6.75	Yes	0.901
x17	5.25	9.50	Yes	2.016
x18	7.00	4.25	Yes	3.816
x19	7.50	8.00	Yes	2.550
x20	7.25	3.75	Yes	4.373

- Rank the training examples and identify set of 3 examples with the smallest distances.

Athlete	***Speed***	***Agility***	***Selected***	***Distance***
x16	5.50	6.75	Yes	0.901
x15	4.75	6.25	Yes	1.275
x2	3.75	8.00	No	1.346

- Yes = 2 votes
- No = 1 vote

➡ Majority says Yes,
so assign label Yes to **q**

Weighted kNN

- Weighted voting: In this approach, some training examples have a higher weight than others.
- Instead of using a binary vote of 1 for each nearest neighbour, typically closer neighbours get higher votes when deciding on the predicted label for a query example.
- Inverse distance-weighted voting: Simplest strategy is to take a neighbour's vote to be the inverse of their distance from the query (i.e. 1/Distance). We then sum over the weights for each class.

$$d(q, x16) = 0.901$$

$$\Rightarrow \text{weight}(x16) = \frac{1}{d(q, x16)} = \frac{1}{0.901} = 1.109$$

$$d(q, x2) = 1.346$$

$$\Rightarrow \text{weight}(x2) = \frac{1}{d(q, x2)} = \frac{1}{1.346} = 0.743$$

Example: Weighted kNN (k=3)

- Measure distance between **q** and all 20 training examples.

Athlete	Speed	Agility	Selected	Distance
x1	2.50	6.00	No	2.915
x2	3.75	8.00	No	1.346
x3	2.25	5.50	No	3.400
x4	3.25	8.25	No	1.904
x5	2.75	7.50	No	2.250
x6	4.50	5.00	No	2.550
x7	3.50	5.25	No	2.704
x8	3.00	3.25	No	4.697
x9	4.00	4.00	No	3.640
x10	4.25	3.75	No	3.824

Athlete	Speed	Agility	Selected	Distance
x11	2.00	2.00	No	6.265
x12	5.00	2.50	No	5.000
x13	8.25	8.50	Yes	3.400
x14	5.75	8.75	Yes	1.458
x15	4.75	6.25	Yes	1.275
x16	5.50	6.75	Yes	0.901
x17	5.25	9.50	Yes	2.016
x18	7.00	4.25	Yes	3.816
x19	7.50	8.00	Yes	2.550
x20	7.25	3.75	Yes	4.373

- Rank the training examples and identify set of 3 examples with the smallest distances. Assign weights based on 1/Distance, and sum weights for each class.

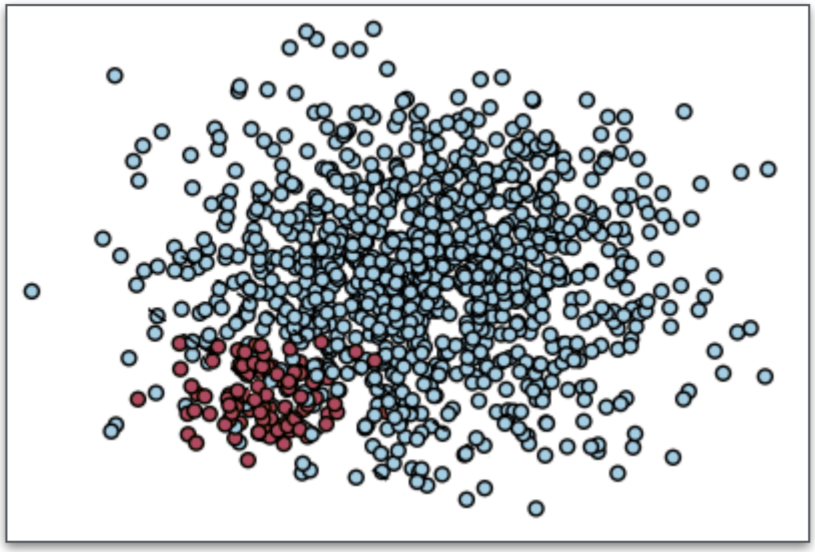
Athlete	Speed	Agility	Selected	Distance	Weight
x16	5.50	6.75	Yes	0.901	1.109
x15	4.75	6.25	Yes	1.275	0.784
x2	3.75	8.00	No	1.346	0.743

- Weights for Yes = 1.109 + 0.784 = 1.893
- Weights for No = 0.743

➡ Majority says Yes

Noisy Data

- A simple 1-NN classifier is easy to implement.
- But it will be susceptible to “noise” in the data.
  - ➡ A misclassification will occur every time a single noisy example is retrieved.
- Using a larger neighbourhood size (e.g. $k > 2$) can sometimes make the classifier more robust and overcome this problem.
- But when k is large ($k \rightarrow N$) and classes are *unbalanced*, we always predict the majority class.

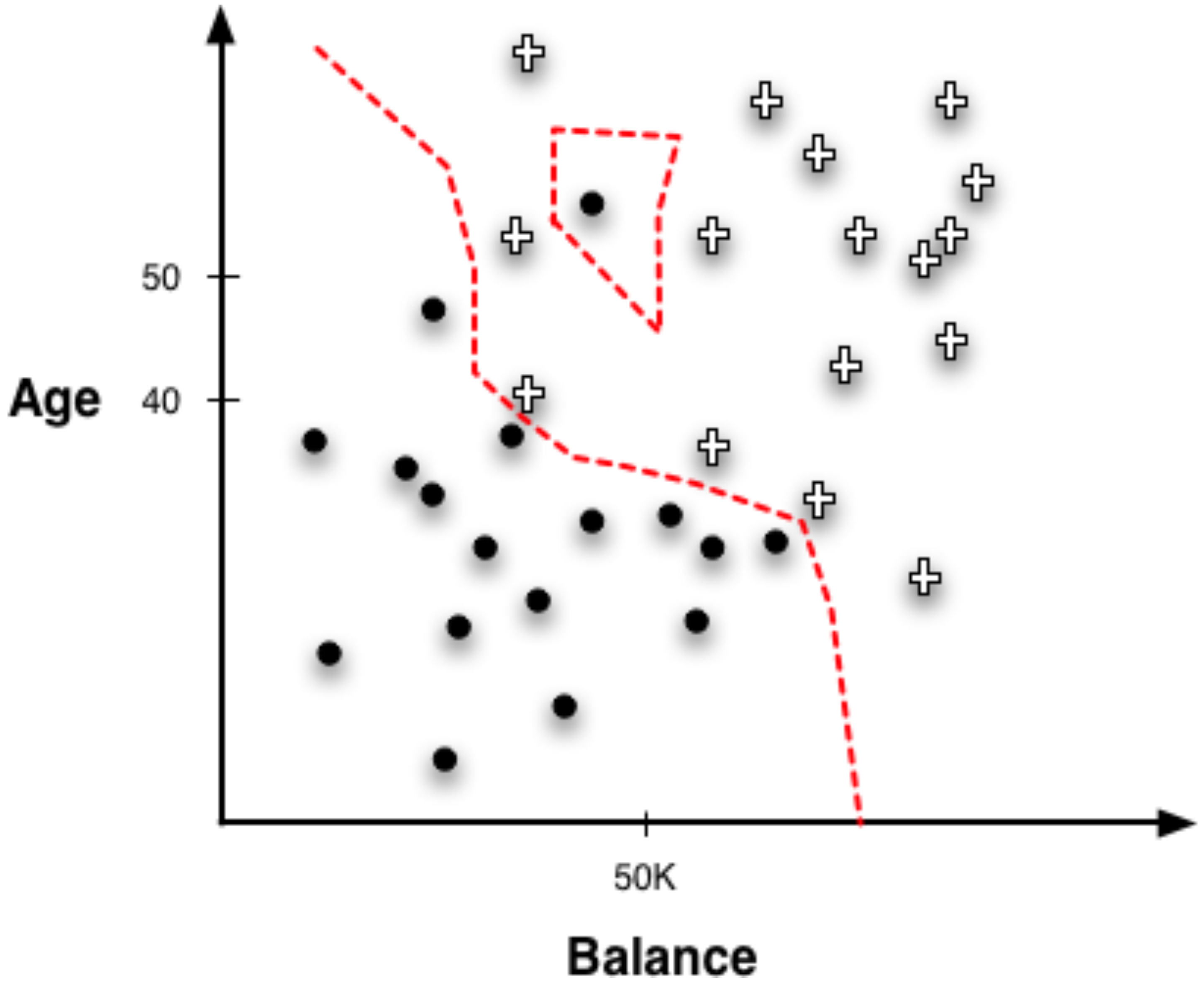


How Many Neighbors and How Much Influence?

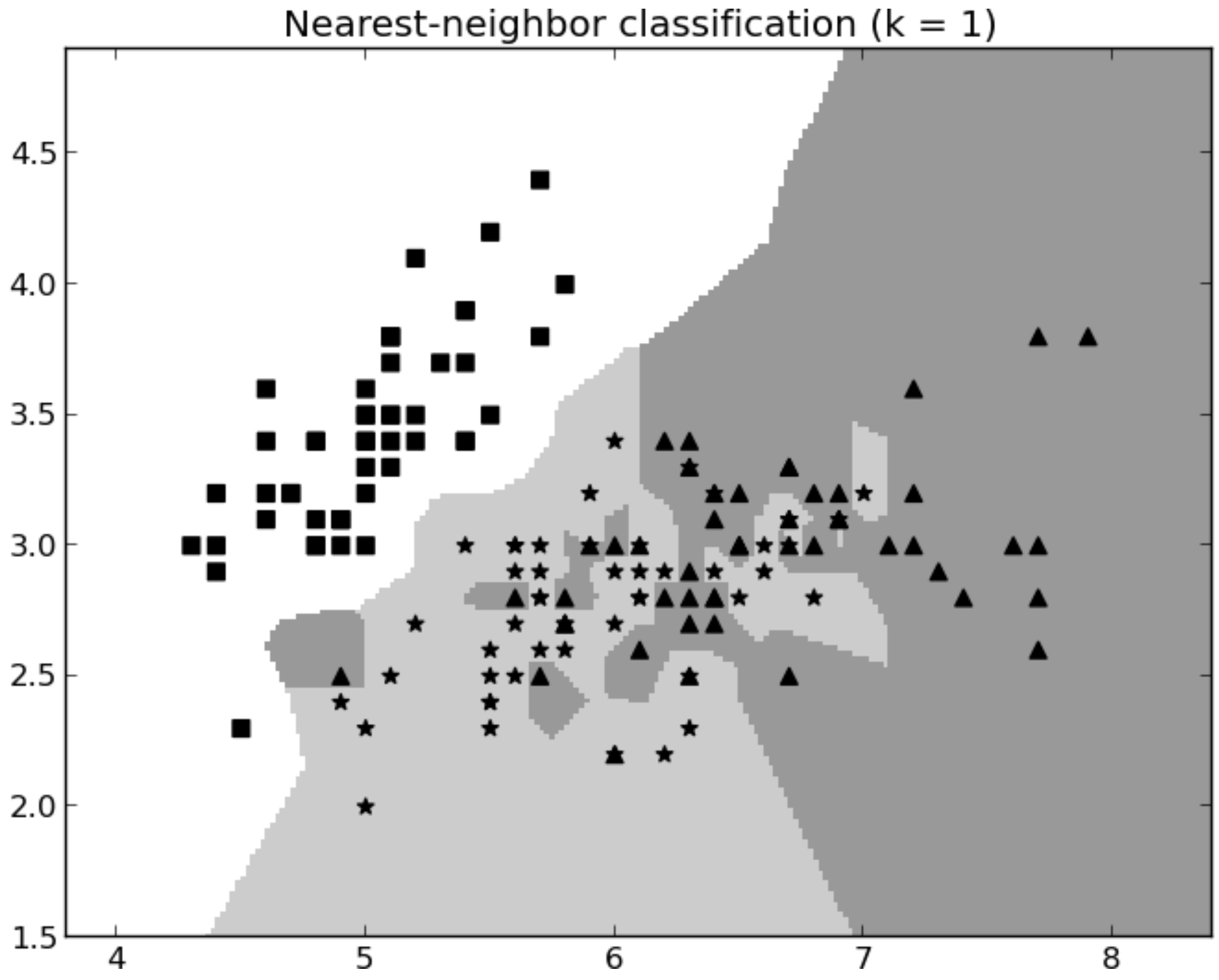
- **$\boldsymbol{k}$ Nearest Neighbors**

- $k = ?$
- $k = 1\ ?$
- $k = n\ ?$

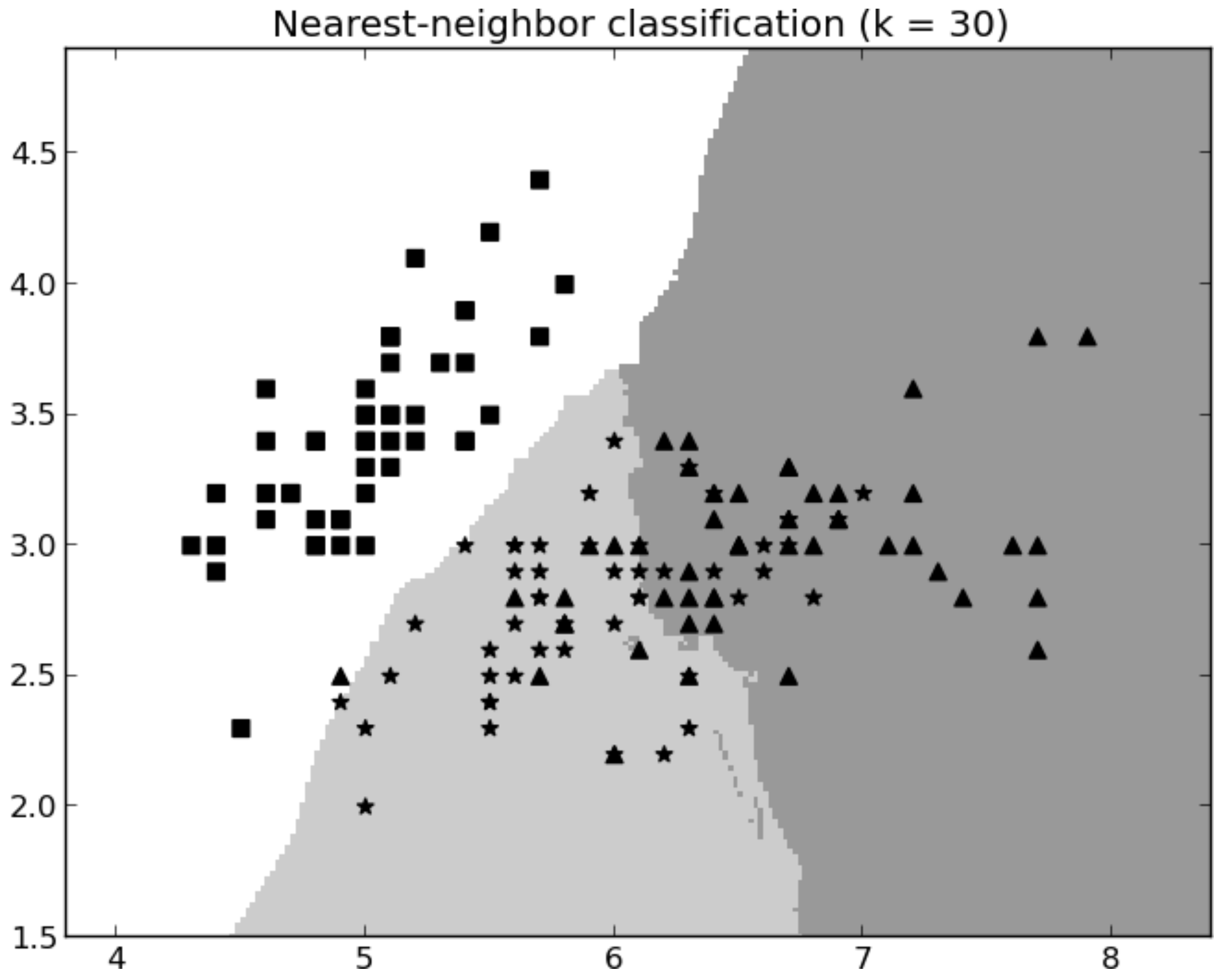
Geometric Interpretation, Over-fitting, and Complexity



1-Nearest Neighbor



30-Nearest Neighbors



Issues with Nearest-Neighbor Models (1)

- Dimensionality and domain knowledge
  - Numeric attributes may have vastly different ranges, and unless they are scaled appropriately the effect of one attribute with a wide range can swamp the effect of another with a much smaller range.
  - But apart from this, there is a problem with having too many attributes, or many that are irrelevant to the similarity judgment.

Solutions (แก้ noise)

- Alleviate these by *feature selection*
- Assign more weights to key features in similarity computation got some correlation
- *Pre-normalize* training features

Issues with Nearest-Neighbor Models (2)

Easy model build
But much computational cost
when predict

- Computational efficiency
  - kNN is actually very fast.
  - However, yhe main computational cost of a nearest neighbor method is borne by the prediction/classification step, when the database must be queried to find nearest neighbors of a new instance. This can be very expensive, and the classification expense should be a consideration.

kNN in scikit-learn

```
>>> X = [[0], [1], [2], [3]]
>>> y = [0, 0, 1, 1]
>>> from sklearn.neighbors import KNeighborsClassifier
>>> neigh = KNeighborsClassifier(n_neighbors=3)
>>> neigh.fit(X, y)
KNeighborsClassifier(...)
>>> print(neigh.predict([[1.1]]))
[0]
>>> print(neigh.predict_proba([[0.9]]))
[[0.66666667 0.33333333]]
```

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

Data Science for Business
Numerical Computing

Asst. Prof. Teerapong Leelanupab (Ph.D.)
Faculty of Information Technology
King Mongkut's Institute of Technology Ladkrabang (KMITL)

Week 7.3

Overview

- NumPy Array Basics
  - 1-dimensional Arrays
  - Multidimensional Arrays
- Array Creation and Reshaping
- Array Operations
- Basic Statistics on Arrays
- Storing NumPy Data
- Using Matplotlib with NumPy
- Pandas v NumPy

Introduction to NumPy

- Standard Python containers are convenient but not designed for large scale data analysis.
- NumPy is the standard Python package for scientific computing:
    - Provides support for multidimensional arrays (i.e. matrices).
    - Implemented closer to hardware for efficiency.
    - Designed for scientific computation, useful for linear algebra and data analysis.
- NumPy can "turn Python into the equivalent of a free and more powerful version of Matlab".

http://www.numpy.org

- NumPy included in the Anaconda distribution. General convention to import numpy is using:

```
import numpy as np
```

NumPy Arrays

- The fundamental NumPy data structure is an array: a memory-efficient container that provides fast numerical operations.
- Unlike standard Python lists, a NumPy array can only contain a single type of value (e.g. only floats; only integers etc).
- The simplest type of array is 1-dimensional - i.e. a vector.
- We can manually create an array from an existing Python list:

```
a = np.array([1,2,3,4])
a
```

```
array([1, 2, 3, 4])
```

```
a.dtype
```

```
dtype('int64')
```

```
a.shape
```

```
(4,)
```

A 1-dimensional array of 4 ints

```
b = np.array([0.1,1.45,0.04])
b
```

```
array([ 0.1 ,  1.45,  0.04])
```

```
b.dtype
```

```
dtype('float64')
```

```
b.shape
```

```
(3,)
```

A 1-dimensional array of 3 floats

Basic Numerical Operations

- We can apply standard numerical operations to arrays using scalars (numbers). The operations are applied element-wise - i.e. applied separately to every element (entry) in the array.
- The operations are much faster than if run in pure Python on a standard list structure.

```
c = np.array([2,4,6,8])
c
```

```
array([2, 4, 6, 8])
```

```
c + 1
```

```
array([3, 5, 7, 9])
```

```
c - 2
```

```
array([0, 2, 4, 6])
```

```
c * 10
```

```
array([20, 40, 60, 80])
```

```
c / 10
```

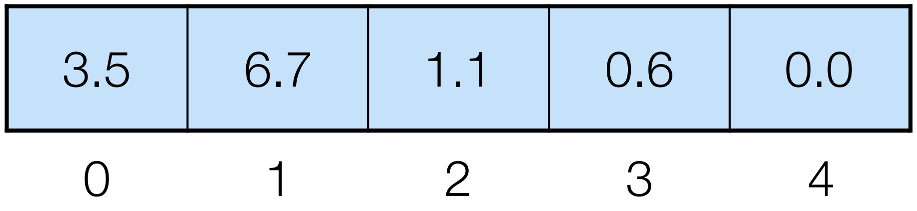
```
array([ 0.2,  0.4,  0.6,  0.8])
```

These numerical operations create a new array of the same size as the original.

Accessing Values in 1D Arrays

- We can access entries in a 1-dimensional array using the same position-based notation as standard Python lists.

```
d = np.array([3.5, 6.7, 1.1, 0.6, 0.0])
```



Access individual entries in the array

```
d[1]
```
```
6.7
```

```
d[4]
```
```
0.0
```

```
d[-1]
```
```
0.0
```

Apply slicing to an array using the using the `[i:j]` notation

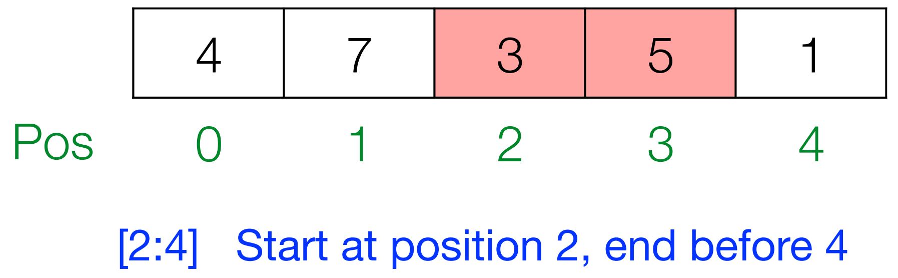
```
d[:2]
```
```
array([ 3.5,  6.7])
```

```
d[0:2]
```
```
array([ 3.5,  6.7])
```

```
d[2:]
```
```
array([ 1.1,  0.6,  0. ])
```

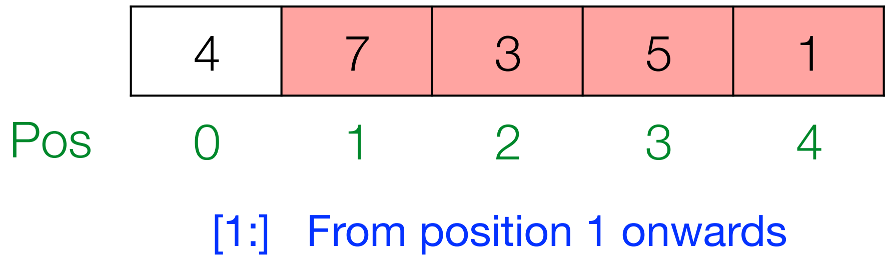
Accessing Values in 1D Arrays

- Important: Slicing creates a "view" on the original array, not a copy.



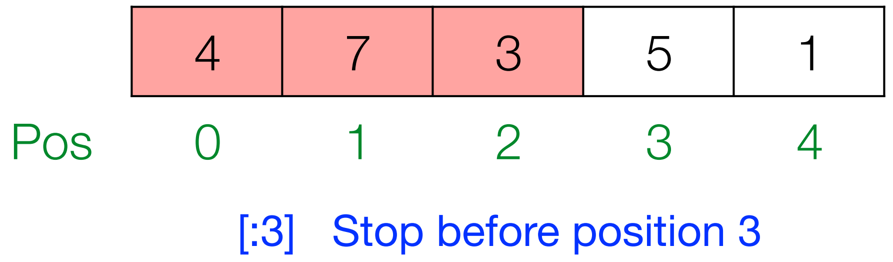
```
a = np.array([4,7,3,5,1])
print( a[2:4] )
```

```
[3 5]
```



```
print( a[1:] )
```

```
[7 3 5 1]
```



```
print( a[:3] )
```

```
[4 7 3]
```

Multidimensional Arrays

- An array can have > 1 dimension. A 2-dimensional array can be viewed as a matrix, with rows and columns. It has these properties:
  - Rank of array: Number of dimensions it has.
  - Shape of array: A tuple of integers giving the length of the array in each dimension.
  - Size of array: Total number of entries it contains.

0.4	2.3	4.5
1.5	0.1	1.3
3.2	0.4	3.2
2.7	2.3	6.3
0.1	0.1	0.9

Example:

Rank = 2 **(x.dnim)**
i.e. 2 dimensions: rows, columns

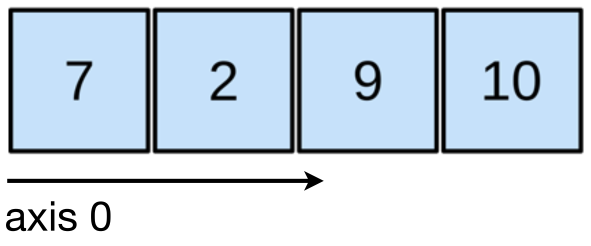
Shape = 5x3 **(x.shape)**
i.e. 5 rows x 3 columns

Size = 15 **(x.size)**
i.e. 5 x 3 = 15 total elements

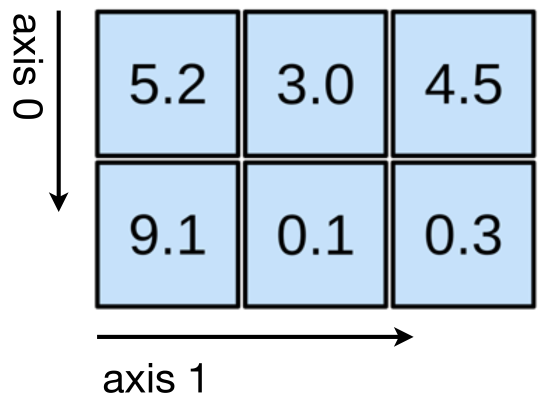
Multidimensional Arrays

- As well as creating 1-dimensional and 2-dimensional arrays, we can also create arrays with > 2 dimensions.
- Axes are defined for arrays with more than one dimension - e.g. a 2-dimensional array has two corresponding axes.

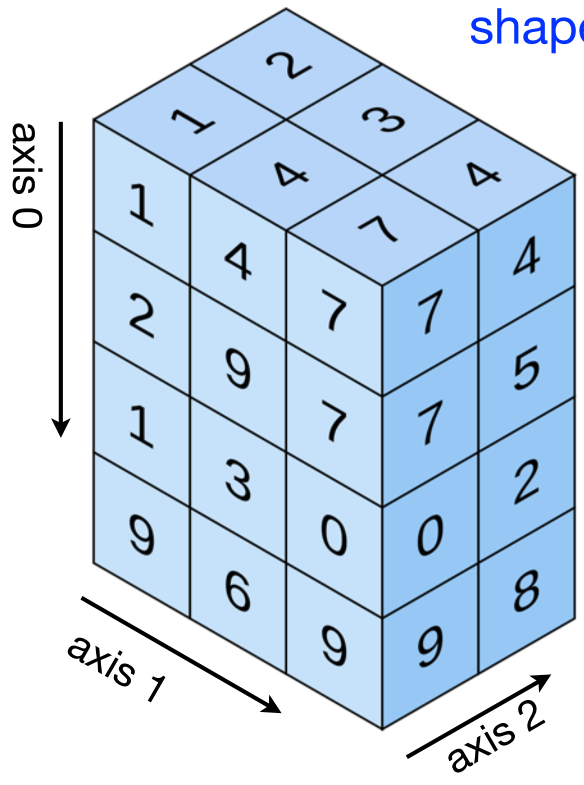
1D array: shape = (4,)



2D array: shape = (2, 3)



3D array:
shape = (4, 3, 2)



(Dashnow et al, 2017)

Multidimensional Arrays

- We can create 2D arrays from a list of Python lists.
- Note: Make sure to include the outer [ ] brackets!

```
d = np.array([[0,4,3], [9,8,6]])
print(d)
```

```
[[0 4 3]
 [9 8 6]]
```

Pass in a list containing 2 nested lists

Create a 2D array, with 2 rows and 3 columns. Total of 2x3 = 6 values

```
m = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
print(m)
```

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

Pass in a list containing 3 nested lists

Create a 2D array, with 3 rows and 4 columns. Total of 3x4 = 12 values

```
m.ndim
```

```
2
```

```
m.shape
```

```
(3, 4)
```

```
m.size
```

```
12
```

We can can check the rank, shape, and size of the new array.

Array Creation Functions

- Rather than using Python lists, a variety of functions are available for conveniently creating and populating arrays.

- Use the `zeros()` function to create an array full of zeros with required shape

```
np.zeros(4)
```

```
array([ 0.,   0.,   0.,   0.])
```

```
np.zeros((2,3))
```

```
array([[ 0.,   0.,   0.],
       [ 0.,   0.,   0.]])
```

Default type is float. For multi-dimensional arrays, specify shape as a tuple.

- Use the `ones()` function to create an array full of ones with required shape

```
np.ones((2,4))
```

```
array([[ 1.,   1.,   1.,   1.],
       [ 1.,   1.,   1.,   1.]])
```

```
np.ones((2,4),dtype=int)
```

```
array([[ 1,  1,  1,  1],
       [ 1,  1,  1,  1]])
```

Use the `dtype` parameter to tell NumPy we want an array of ints, not floats.

Array Creation Functions

- We can create an array corresponding to a sequence using the `arange()` function.

Start at 2, end before 7

```
np.arange(2,7)
```

```
array([2, 3, 4, 5, 6])
```

- We can also specify a step size for the values. The default step is 1.

```
np.arange(2,7,2)
```

```
array([2, 4, 6])
```

- The range and step sizes do not have to be integers. We can also specify floats:

```
x = np.arange(0.5, 9.4, 1.3)
print(x)
```

```
[0.5  1.8  3.1  4.4  5.7  7.   8.3]
```

Start at 0.5, increment in steps of 1.3, end before 9.4

- The `linspace()` function creates an array with a specified number of evenly-spaced samples in a given range:

```
y = np.linspace(1, 10, 4)
print(y)
```

```
[  1.   4.   7.  10.]
```

Divides up the range [1,10] into 4 evenly-spaced values, including the endpoints.

พยายามแบ่งให้เท่าๆกัน

Array Shape Manipulation

- The previous functions all created 1D arrays. What if we want to create multidimensional arrays?
- We can change array shape. The original values are copied to a new array with the specified shape.

```
x = np.arange(2,8)
print(x)
```

```
[2 3 4 5 6 7]
```

Original 1D array with 6 values

```
m1 = x.reshape(3,2)
print(m1)
```

```
[[2 3]
 [4 5]
 [6 7]]
```

New 2D array with 3 rows and 2 columns, same values.

```
m2 = x.reshape(2,3)
print(m2)
```

```
[[2 3 4]
 [5 6 7]]
```

New 2D array with 2 rows and 3 columns, same values.

- The <u>size</u> of the reshaped array has to be same as the original. e.g. we cannot reshape a 1D array with 6 values into 2D arrays of size 2x2, 4x2, etc.

Accessing Multidimensional Arrays

- To access a value in a 1D array, specify the position `[i]` counting from 0, just like a Python list.
- We can also use this notation to change the values in an existing array.

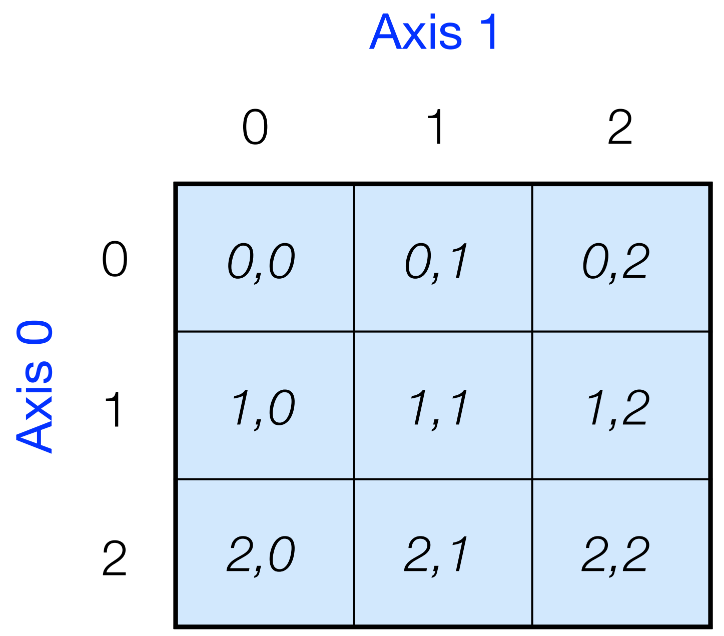
```
a = np.array([8,6,12])
a[1]
```

```
6
```

```
a[2] = 25
print(a)
```

```
[ 8  6 25]
```

- When working with arrays with more than one dimension, use the notation `[i,j]`, where the position in each dimension is separated by commas.
- Axis 0 refers to the rows, Axis 1 refers to the columns.

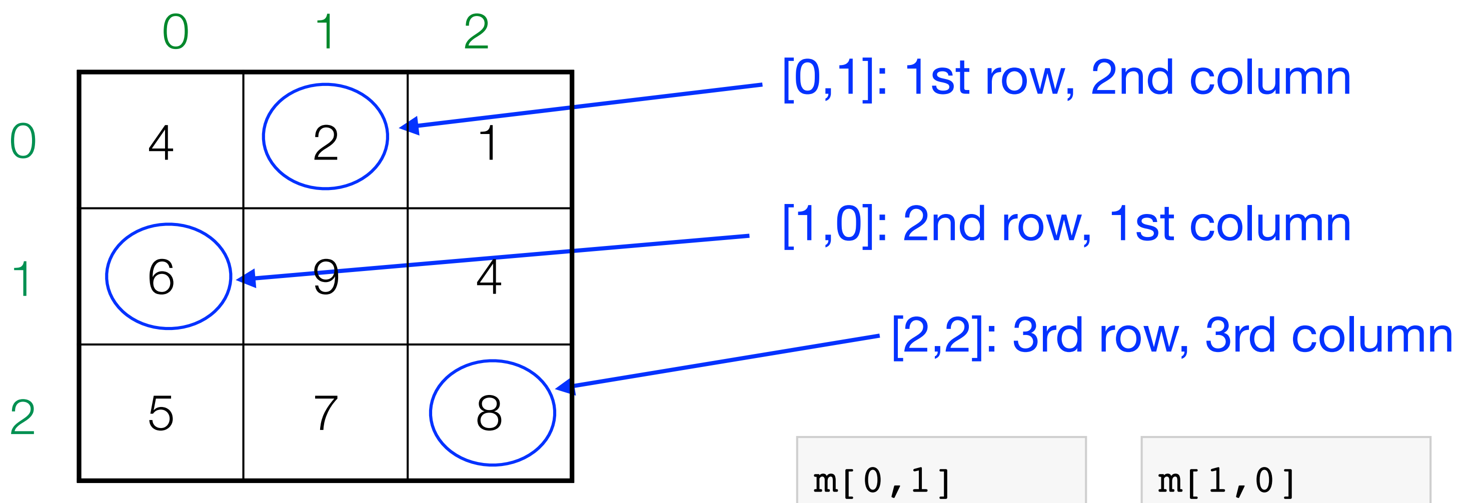


Accessing Multidimensional Arrays

- When working with arrays with more than 1 dimension, use the notation `[i,j]`, where the position in each dimension is separated by commas.

```
np.array([[4,2,1],[6,9,4],[5,7,8]])
```

Create 3x3 array



0
1
2
0
1
2
4
2
1
6
9
4
5
7
8
[0,1]: 1st row, 2nd column
[1,0]: 2nd row, 1st column
[2,2]: 3rd row, 3rd column

```
m[0,1]
```
```
2
```

```
m[1,0]
```
```
6
```

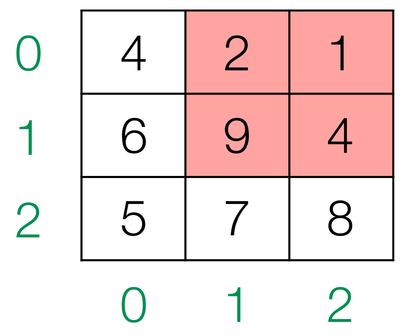
```
m[2,2]
```
```
8
```

- Same notation applies to higher dimensional arrays (e.g. 3D, etc).

Slicing Multidimensional Arrays

- For multidimensional arrays, we specify the slices for each dimension, separated by commas - e.g. for 2D `[i:j,p:q]`

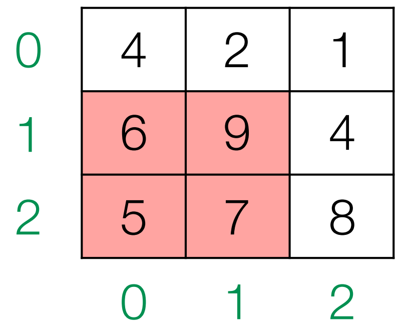
```
d = np.array([[4,2,1],[6,9,4],[5,7,8]])
```



```
d[0:2,1:3]
```

```
array([[2, 1],
       [9, 4]])
```

Rows: Start at 0, end before 2
Columns: Start at 1, end before 3



```
d[1:3,0:2]
```

```
array([[6, 9],
       [5, 7]])
```

Rows: Start at 1, end before 3
Columns: Start at 0, end before 2

```
d[1,:]
```

```
array([6, 9, 4])
```

Full row at position 1

```
d[:,2]
```

```
array([1, 4, 8])
```

Full column at position 2

Iterating Over Arrays

- Generally, we want to avoid iterating over the individual elements of arrays as it is extremely slow.
- NumPy arrays are designed to be used for vectorised operations i.e. applying one operation to every value in an array at once.
- Sometimes it might be unavoidable. We can use a `for` loop...

```
M = np.array([[1,2], [3,4]])
for row in M:
    print("row", row)
    for x in row:
        print(x)
```

```
row [1 2]
1
2
row [3 4]
3
4
```

```
v = np.array([1,2,3,4])
for x in v:
    print( x * 10 )
```

```
10
20
30
40
```

- But, better to apply an operation to all values in an array whenever possible, unless running time does not matter.

Numerical Operations

- We can run batch operations on multidimensional arrays without for loops. These operations create a new copy of the original array.

```
d = np.array([[1,4,2], [9,8,2]])
d
```

```
array([[1, 4, 2],
       [9, 8, 2]])
```

```
d * 2
```

```
array([[ 2,  8,  4],
       [18, 16,  4]])
```

```
1.0/d
```

```
array([[1.,    0.25,   0.5        ],
       [0.11111111, 0.125, 0.5 ]])
```

```
d * d
```

```
array([[ 1, 16,  4],
       [81, 64,  4]])
```

Note that * between arrays multiplies corresponding elements together, does not perform matrix multiplication.

คูณกันตามตำแหน่งที่ตรงกัน

- We can also apply functions to all elements in an array.

```
np.log(d)
```

```
array([[ 0.        , 1.38629436, 0.69314718],
       [ 2.19722458, 2.07944154, 0.69314718]])
```

Function np.log() is applied to every element in the array.

Comparison Operations

- We can use standard boolean expressions in batch to all elements in an array. The result is a new boolean array of the same shape.

```
d = np.array([[5,2], [1,3]])
d
```

```
array([[5, 2],
       [1, 3]])
```

```
d > 2
```

```
array([[ True, False],
       [False, True]],dtype=bool)
```

Is each element > 2?

```
d == 1
```

```
array([[False, False],
       [ True, False]], dtype=bool)
```

Which elements are equal to 1?

```
d != 1
```

```
array([[True, True],
       [False, True]], dtype=bool)
```

Which elements are not equal to 1?

- We can also use this approach to change certain values in arrays.

```
d[d < 5] = 0
d
```

```
array([[5, 0],
       [0, 0]])
```

Modify the original array to change all elements < 5 to 0

Basic Statistics

- NumPy arrays also have basic descriptive statistics functions.

```
a = np.array([0.1,0,1.4,0.04])
a.sum()
```

```
1.54
```

```
a.min()
```

```
0.0
```

```
a.max()
```

```
1.4
```

```
a.mean()
```

```
0.38500000000000001
```

```
a.std()
```

```
0.58709028266528129
```

Can compute the mean (average), standard deviation (std), and min/max

- For multidimensional arrays, the above can also take an optional `axis` parameter. If this is specified, calculations are only performed along that axis (dimension) and the result is a new array.

```
d = np.array([[5,4,0],[0,1,2]])
d.mean()
```

```
2.0
```

Mean based on all elements in the array

```
d.mean(axis=0)
```

```
array([ 2.5, 2.5, 1. ])
```

Average over rows, to get mean for each of the 3 columns

```
d.mean(axis=1)
```

```
array([ 3.,  1.])
```

Average over columns, to get Mean for each of the 2 rows

Storing NumPy Data

- The `np.savetxt()` function can be used to save CSV formatted version of NumPy arrays.

```
x = np.arange(1,4,0.5)
m1 = x.reshape(3,2)
print(m1)
```

```
[[ 1.   1.5]
 [ 2.   2.5]
 [ 3.   3.5]]
```

```
np.savetxt("out.txt",m1)
```

File: out.txt

```
1.00000000000000e+00 1.50000000000000e+00
2.00000000000000e+00 2.50000000000000e+00
3.00000000000000e+00 3.50000000000000e+00
```

Output file shows full precision for float values

We can also specify extra parameters specifying a format string to control the output and a separator for values.

```
np.savetxt("res.csv",m1,"%.1f",",")
```

File: out2.txt

```
1.0,1.5
2.0,2.5
3.0,3.5
```

บอก delimiter ใน
csv(comma seperate)

Output file shows values are comma-separated, and only written to 1 decimal place.

Storing NumPy Data

- The `np.loadtxt()` function can be used to read CSV data from a file and create a multidimensional NumPy array from the data.
- Each line is a row, and should contain the same number of values. By default values are separated by spaces.

File: numbers.txt

```
0.8 0.8 0.9
0.7 0.1 0.2
0.6 0.4 0.1
1.0 0.8 0.2
0.9 0.1 0.4
```

```
a = np.loadtxt("numbers.txt")
a
```

```
array([[ 0.8,   0.8,   0.9],
       [ 0.7,   0.1,   0.2],
       [ 0.6,   0.4,   0.1],
       [ 1. ,   0.8,   0.2],
       [ 0.9,   0.1,   0.4]])
```

- We can also load files with other separators, by specifying the `delimiter` parameter.

File: scores.csv

```
0.74,0.63,0.58,0.89
0.91,0.89,0.78,0.99
0.43,0.35,0.34,0.45
0.56,0.61,0.66,0.58
0.50,0.49,0.76,0.72
0.88,0.75,0.61,0.78
```

```
a = np.loadtxt("scores.csv",delimiter=",")
a
```

```
array([[ 0.74,  0.63,  0.58,  0.89],
       [ 0.91,  0.89,  0.78,  0.99],
       [ 0.43,  0.35,  0.34,  0.45],
       [ 0.56,  0.61,  0.66,  0.58],
       [ 0.5 ,  0.49,  0.76,  0.72],
       [ 0.88,  0.75,  0.61,  0.78]])
```

Using Matplotlib with NumPy

- Matplotlib can be used in conjunction with NumPy arrays to visualise numeric data, in the same way we saw with Python lists.
- For example, a scatter plot of one 1D array against another:

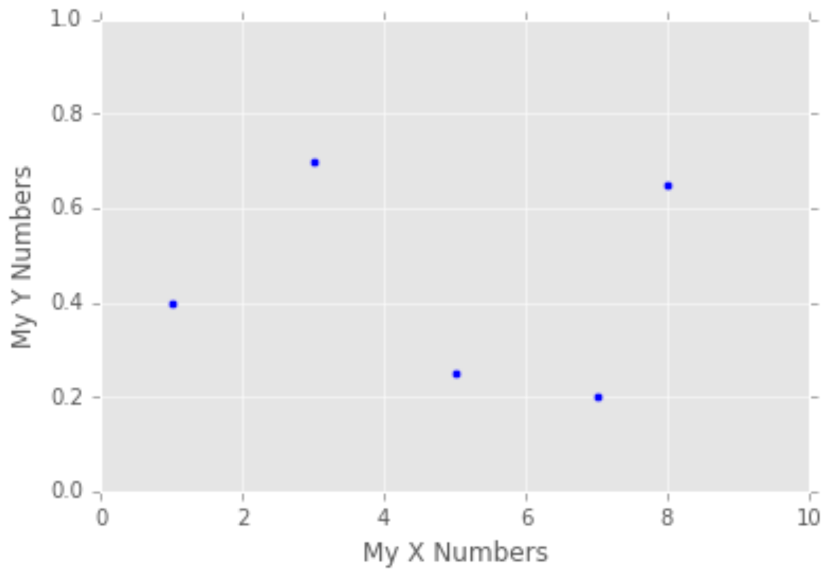
Create the X and Y values

```
xvalues = np.array([1, 5, 8, 3, 7])
yvalues = np.array([0.4, 0.25, 0.65, 0.7, 0.2])
```

Create a scatter plot of X versus Y values

```
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
```

```
plt.figure(figsize=(8,5))
plt.scatter(xvalues,yvalues)
plt.axis([0,10,0,1])
plt.xlabel("My X Numbers")
plt.ylabel("My Y Numbers")
```



Using Matplotlib with NumPy

- For 2D NumPy arrays, a common type of visualisation is a colour plot, which can be produced using `plt.pcolor()`.

Create a 5x4 2D array of random numbers

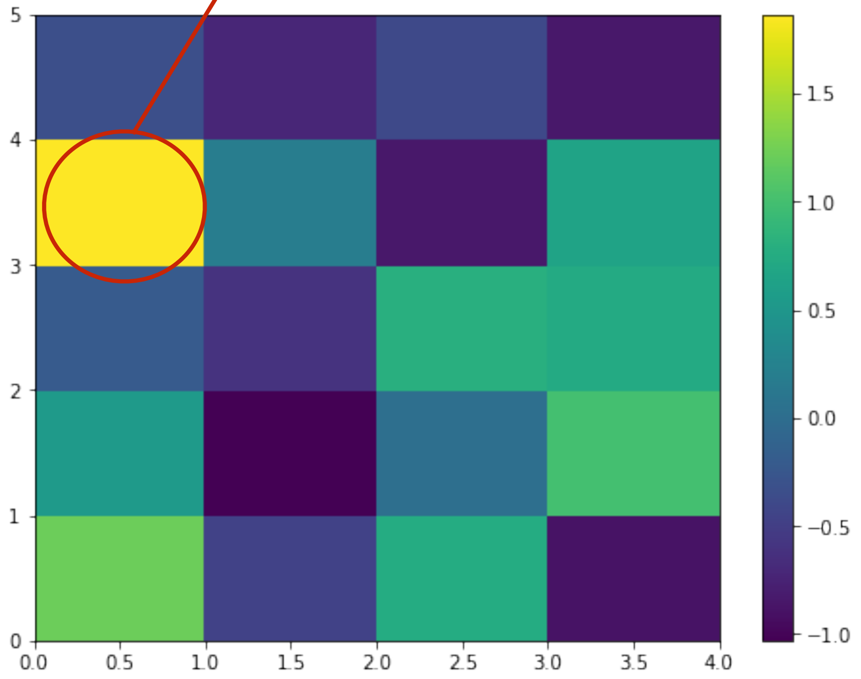
```
v = np.random.randn(20)
a = v.reshape(5,4)
```

Create the figure

```
plt.figure(figsize=(8,6))
plt.pcolor(a)
plt.colorbar()
```

Each entry in the coloured matrix corresponds to an entry in the original 2D arra.y

```
array([[ 1.2, -0.5,  0.7, -0.9],
       [ 0.5, -1. ,  0. ,  1. ],
       [-0.2, -0.6,  0.8,  0.7],
       [ 1.9,  0.2, -0.8,  0.6],
       [-0.3, -0.7, -0.4, -0.8]])
```



Pandas and NumPy

- NumPy is primarily useful for working with arrays. Highly optimised for efficient operations on numeric arrays.
- Pandas provides higher level data manipulation tools built on top of NumPy arrays, along with more semantics (e.g. indexes).
- Some operations are not as efficient, but Pandas provides additional functionality - e.g dictionary-style access via row or column index to tabular data.
- Since Pandas is built on top of NumPy, we can easily convert values between a NumPy array and a Pandas Series or Data Frame.

Create a 1D array, then construct a Series from it.

```
import numpy as np
import pandas as pd
a = np.array([0.1,0,1.4,0.04])
s = pd.Series(a)
print(s)
```

```
0     0.10
1     0.00
2     1.40
3     0.04
dtype: float64
```

Pandas and NumPy

- Since NumPy arrays do not have row or column indices, we may need to specify these if we convert an array to a Data Frame.

Create a 4x3 2D array of random numbers

```
v = np.random.randn(12)
m = v.reshape(4,3)
```

```
[[ 0.78016711 -1.53057067  0.67719719]
 [ 0.78171014  1.29939974 -0.47013332]
 [-1.28672265 -0.55481209 -0.9546214 ]
 [ 1.33540396  1.81220164
-2.05715548]]
```

Create a corresponding number of row and column index labels

```
col_index = ["A","B","C"]
row_index = ["r1","r2","r3","r4"]
```

Now use this to create a Data Frame

```
df = pd.DataFrame(m,
columns=col_index,
index=row_index )
```

	A	B	C
r1	0.780167	-1.530571	0.677197
r2	0.781710	1.299400	-0.470133
r3	-1.286723	-0.554812	-0.954621
r4	1.335404	1.812202	-2.057155