



# DTE-2501 AI Methods and Applications

*Basic introduction to AI*

*Lecture 1/3 – Introduction*

Tatiana Kravetc  
*Førsteamanuensis*

*Office: D2240*

*Email: tatiana.kravetc@uit.no*

# Overview

I Introduction

II Basic terminology

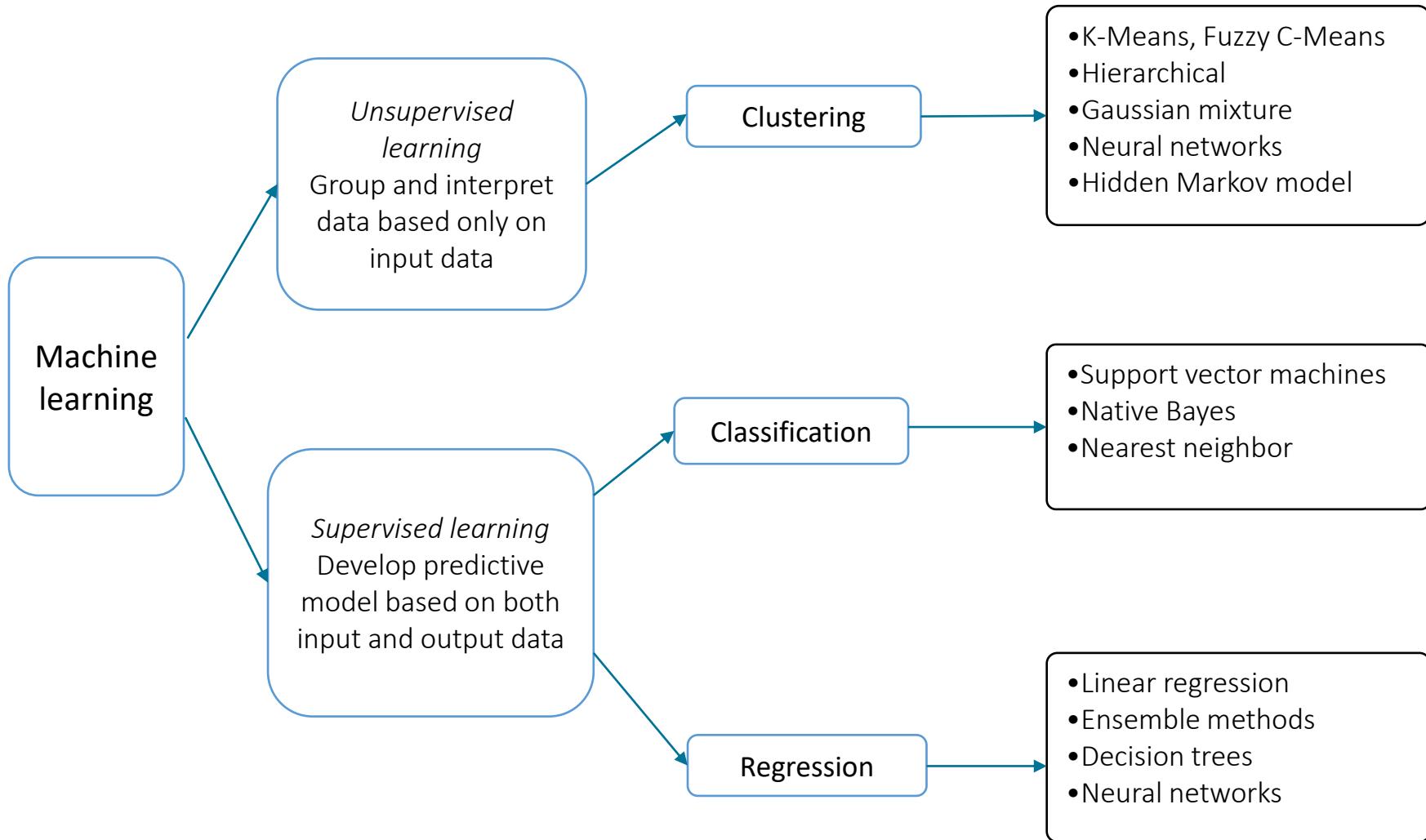
III Machine learning techniques

IV Cross-industry standard process for data mining

# I Introduction

- statistical data analysis
- artificial intelligence (1955)
- pattern recognition
- machine learning (1959)
- statistical learning
- data mining (1989)
- data science (1997)
- business analytics
- predictive analytics (2007)
- big data (2008)
- big data analytics
- Smart vehicles
- Smart buildings
- Precision medicine
- AI-enhanced education
- ...

# Diversity of machine learning approaches



## II Basic terminology

*Supervised learning* trains a model on known input and output data so that it can predict future outputs.

Let  $X$  be a set of inputs and  $Y$  be a set of outputs. The learning goal is to find an unknown *target function*  $y: X \rightarrow Y$ .

$\{x_1, \dots, x_l\} \subset X$  is a set of training samples,  $l$  is the cardinality of the data set.

$y_i = y(x_i), i = 1, \dots, l$ , known responses (outputs).

We seek for an algorithm (decision function)  $a: X \rightarrow Y$ , such that it approximates  $y$  on the entire set  $X$ .

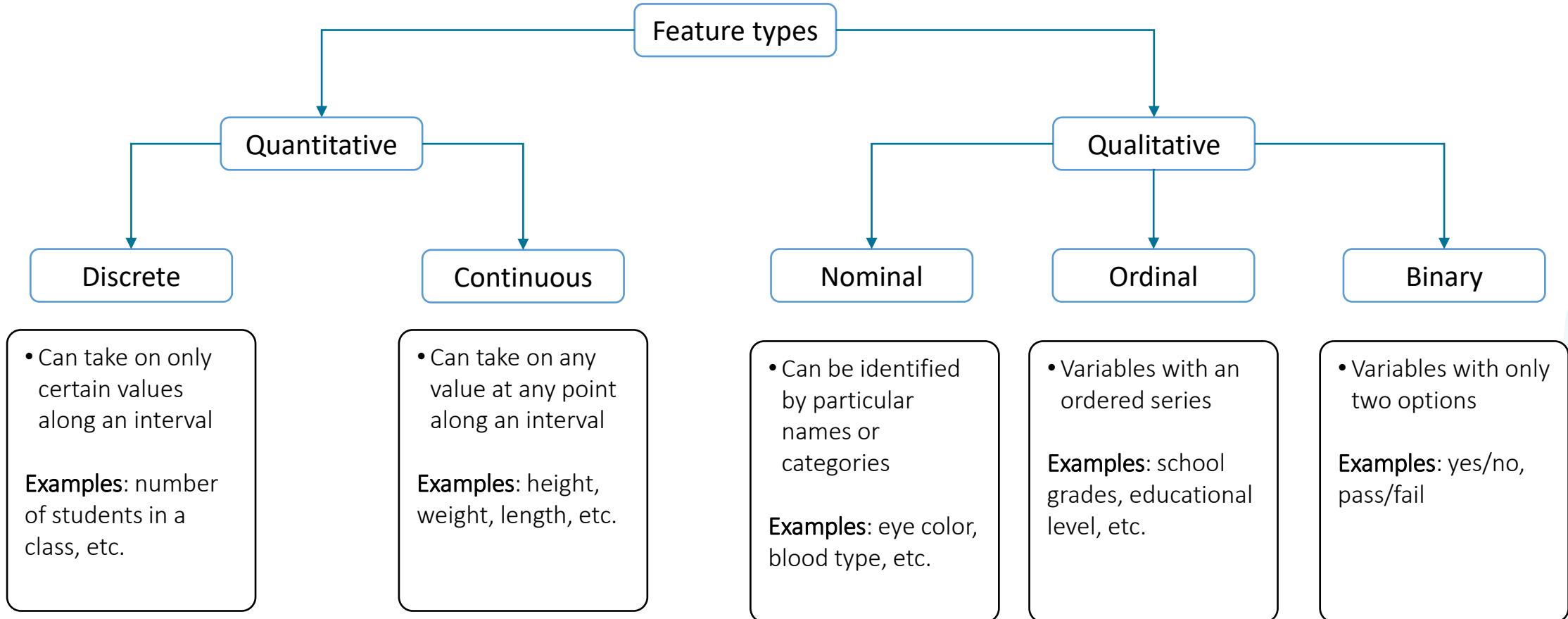
## Basic terminology

*Feature (attribute)* is a mapping  $f: X \rightarrow D_f$ , where  $D_f$  is a set of possible feature values.

Let  $f_1, \dots, f_n$  is a set of features. A vector  $(f_1, \dots, f_n)$  is called a feature description of the object  $x \in X$ . A set of all feature descriptions, written as a table of size  $l \times n$  is called a *feature data matrix*:

$$F = [f_j(x_i)]_{l \times n} = \begin{pmatrix} f_1(x_1) & \dots & f_n(x_1) \\ \vdots & \ddots & \vdots \\ f_1(x_l) & \dots & f_n(x_l) \end{pmatrix}$$

# Basic terminology



# III Machine learning techniques

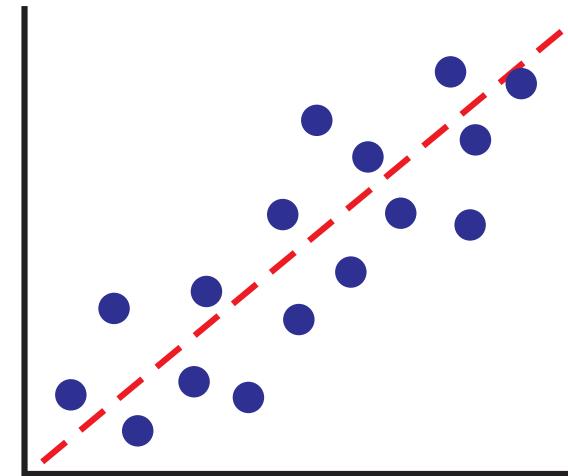
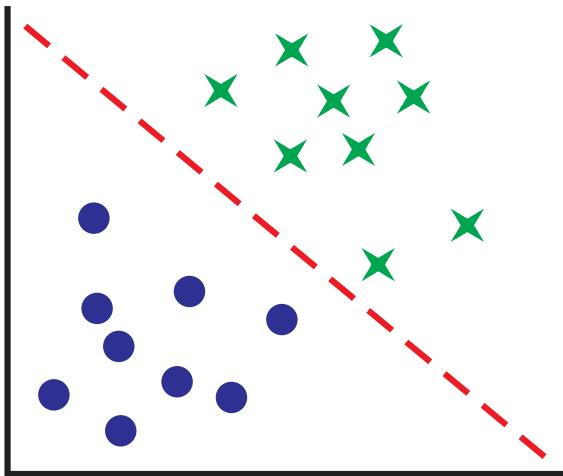
- *Classification techniques* predict discrete responses

$Y = \{1, -1\}$  is two class classification

$Y = \{1, \dots, M\}$  is M class classification

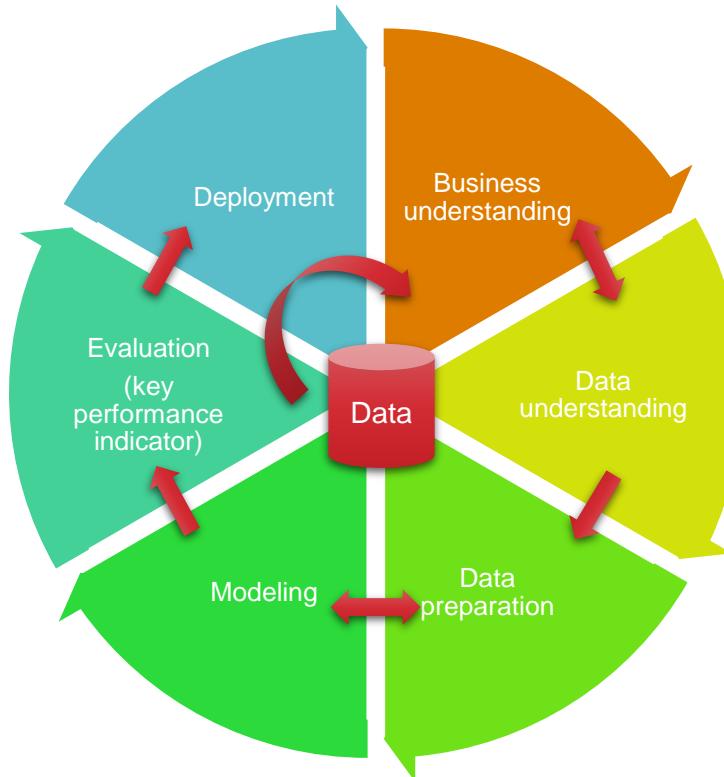
- *Regression techniques* predict continuous responses

$Y = \mathbb{R}$  or  $Y = \mathbb{R}^m$



# IV Cross Industry Standard: CRISP-DM Process for Data Mining (v.1 1999)

Open standard process model that describes common approaches used by data mining experts





# DTE-2501 AI Methods and Applications

Basic Introduction to AI

*Lecture 3/3: Dynamic Programming*

Ghada Bouzidi

PhD Candidate

*Department of Computer Science and Computational Engineering*

*Office:* C4060

*Email:* [ghada.bouzidi@uit.no](mailto:ghada.bouzidi@uit.no)

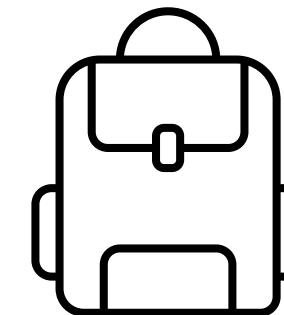
# Main Algorithmic Techniques

- Brute Force
- Divide & Conquer (sort, search ...)
- Greedy (fractional knapsack ...)
- **Dynamic Programming**

# Greedy: Fractional Knapsack

	Value (Nok)	Weight (Kg)	Unit Value (Nok/Kg)
2	250	5	50.0
4	50	25	2.0
3	500	20	25.0
1	300	5	60.0

**Problem:** Find the maximum value to fit in the knapsack without exceeding the capacity. It is allowed to take any fraction of any item.



Knapsack capacity  
**25 Kg**

**Solution pseudocode:**

1. Sort items by unit value
2. While the picked items did not exceed the knapsack capacity pick an item from the remaining items that has the highest unit value (Greedy choice)

# Dynamic Programming

- Widely used in different fields (Biology, Planning, Logistics ...)
- Solves problems that cannot be solved using Greedy and Divide & Conquer approaches
- Some Brute Force algorithms may spend years and may not even finish in our lifetime, if a Dynamic Programming solution exists then the runtime will decrease considerably.

# History of Dynamic Programming

- Developed by **Richard Bellman** in the 1950s
- "...An interesting question is, "Where did the name, dynamic programming, come from?" The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was **Secretary of Defense**, and **he actually had a pathological fear and hatred of the word "research"**...The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics...What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming". **I wanted to get across the idea that this was dynamic**...Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is **it's impossible to use the word dynamic in a pejorative sense**. Try thinking of some combination that will possibly give it a pejorative meaning. **It's impossible**. Thus, I thought dynamic programming was a good name. **It was something not even a Congressman could object to..."**

*Richard Bellman, Eye of the Hurricane: An Autobiography (1984, page 159)*

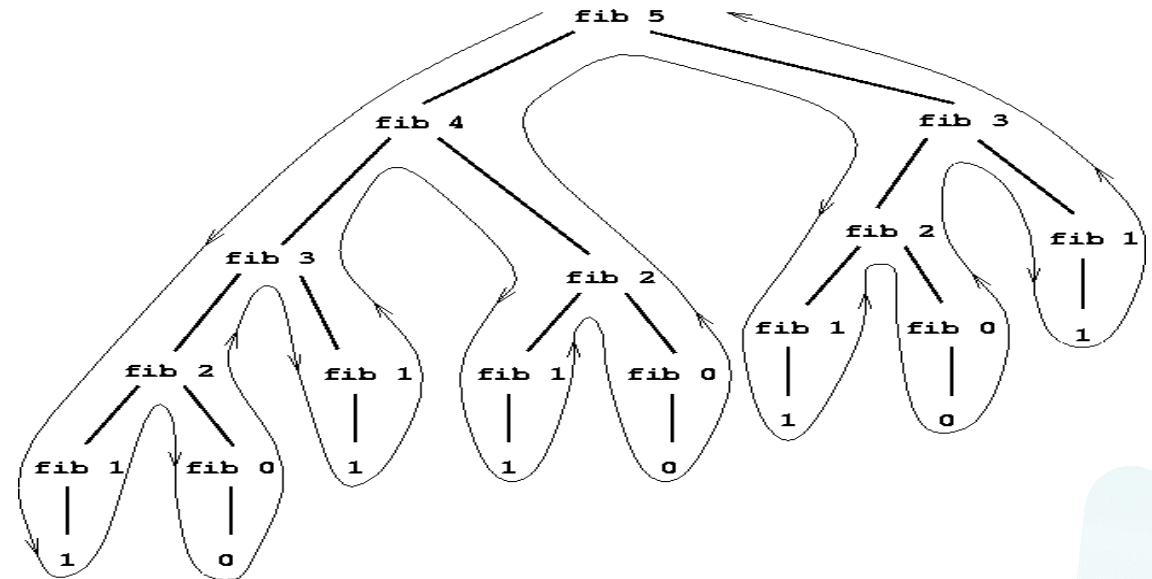
# Example 1: Fibonacci Sequence

- $\text{Fib}(0) = 0$
- $\text{Fib}(1) = 1$
- $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$  for all  $n \geq 2$

The Fibonacci sequence : 0, 1, 1, 2, 3, 5, 8, 13 ...

# Recursion

```
def Fib(n):
    if n <= 1:
        return n
    else:
        return Fib(n-1) + Fib(n-2)
```



Recursion tree of  $\text{Fib}(5)$

# Recursion + cache (Memoization)

- How many times one needs to compute  $\text{Fib}(2)$ ?
- One time is sufficient, then store its value "somewhere"
- Whenever its value is needed, one can look it up

**“Those who cannot remember the past are condemned to repeat it.”**

Philosopher George Santayana, *The Life of Reason* (1905)

# Memoized Fibonacci (Top-Down Approach)

```
def fib(n, memo):
    if n <= 1:
        return n
    elif memo[n] == -1:
        memo[n] = fib(n-1, memo) + fib(n-2, memo)
    return memo[n]
```

# Iterative Fibonacci: Tabular DP(Bottom-Up)

```
def fib(n):
    dp = [0] * (n+1)
    dp[1] = 1
    for i in range(2, n+1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]
```

dp:	0	1	1	2	3	5	8	13	...
	0	1	2	3	4	5	6	7	...

**NB:** This solution can be optimized for space since you only need the last two values at each step

# When to Use Dynamic Programming

- The problem cannot be solved through Greedy approach
- The problem can be divided into smaller subproblems (i.e., with smaller inputs)
- There exists a recurrent relation that defines how the subproblems can be combined to solve the problem
- The subproblems intersect

# Memoization vs Tabular DP

- Sometimes it is more intuitive and easier to implement a dynamic programming algorithm recursively once the recurrent relation is identified than writing a tabular DP
- It is important to keep in mind that you may run into stackoverflow problems when the recursion depth exceeds the limited size of the function call stack
- In those cases, an iterative approach such as the Tabular DP might be the solution
- The runtime analysis of a Tabular DP can be more easily calculated than that of Memoization

# Example 2: Coin Change

- Problem: Given a sum  $s$  and a set of coins with denominations  $\{1, 4, 5\}$ , write the function CoinChange that takes as input  $s$  and returns the **minimum number of coins that sum up to  $s$**
- *Can Greedy solve this problem? No*
- *If  $s = 8$  then we can first pick the coin 5 (Greedy choice, biggest coin  $\leq s$ ) then we can pick the coin 1 three times to get  $8 = 5 + 1 + 1 + 1$  output = **4 coins***  
*We know that we can have less coins that make up the sum 8 through picking the coin 4 two times ( $8 = 4 + 4$ ). Thus, the output should be **2 coins***
- *For  $s = 8$ , the minimum number of coins is 2 (4, 4)*

# Recurrent Relation for Coin Change

- If we know the minimum coins to change  $s-1$  and  $s-4$  and  $s-5$ ,  
*can we know the minimum coins to change  $s$  ?*
- If  $s = 8$  and we know how to change 7 and 4 and 3, only adding the coins of values 1, 4 and 5 respectively will give us the sum 8. However, we need to make the choice such that the result is optimal.
- Let  $\text{CoinChange}(i)$  be the minimum coins needed to change a sum  $i$  then:

$$\text{CoinChange}(i) = 1 + \min \{\text{CoinChange}(i-1), \text{CoinChange}(i-4), \text{CoinChange}(i-5)\}$$

# Memoized Coin Change

```
def CoinChange(n, memo):
    if n <= 0:
        return 0
    elif n not in memo:
        memo[n] = min(CoinChange(n-1, memo),
                      CoinChange(n-4, memo),
                      CoinChange(n-5, memo)) + 1
    return memo[n]
```

# Tabular Coin Change 1/2

For each index  $i$  in the table "coin\_change",  $\text{coin\_change}[i]$  stores the minimum number of coins to change the sum  $i$ .

0	1	2	3	1	1	2	3	2
0	1	2	3	4	5	6	7	8

How to fill this table for  $s = 8$ ?  $\text{coin\_change}[i] = \min(\text{coin\_change}[i-1], \text{coin\_change}[i-4], \text{coin\_change}[i-5]) + 1$

$\text{coin\_change}[0] = 0$

$\text{coin\_change}[1] = \min(\text{coin\_change}[1-1], \cancel{\text{coin\_change}[1-4]}, \cancel{\text{coin\_change}[1-5]}) + 1 = \min(0, \text{Inf}, \text{Inf}) + 1 = 1$

$\text{coin\_change}[2] = \min(\text{coin\_change}[2-1], \cancel{\text{coin\_change}[2-4]}, \cancel{\text{coin\_change}[2-5]}) + 1 = \min(1, \text{Inf}, \text{Inf}) + 1 = 2$

$\text{coin\_change}[3] = \min(\text{coin\_change}[3-1], \cancel{\text{coin\_change}[3-4]}, \cancel{\text{coin\_change}[3-5]}) + 1 = \min(2, \text{Inf}, \text{Inf}) + 1 = 3$

$\text{coin\_change}[4] = \min(\text{coin\_change}[4-1], \text{coin\_change}[4-4], \cancel{\text{coin\_change}[4-5]}) + 1 = \min(3, 0, \text{Inf}) + 1 = 1$

$\text{coin\_change}[5] = \min(\text{coin\_change}[5-1], \text{coin\_change}[5-4], \text{coin\_change}[5-5]) + 1 = \min(1, 1, 0) + 1 = 1$

$\text{coin\_change}[6] = \min(\text{coin\_change}[6-1], \text{coin\_change}[6-4], \text{coin\_change}[6-5]) + 1 = \min(1, 2, 1) + 1 = 2$

$\text{coin\_change}[7] = \min(\text{coin\_change}[7-1], \text{coin\_change}[7-4], \text{coin\_change}[7-5]) + 1 = \min(2, 3, 2) + 1 = 3$

$\text{coin\_change}[8] = \min(\text{coin\_change}[8-1], \text{coin\_change}[8-4], \text{coin\_change}[8-5]) + 1 = \min(3, 1, 3) + 1 = 2$

# Tabular Coin Change 2/2

- How to recover which 2 coins sum to  $s = 8$  from the dynamic programming table?

0	1	2	3	1	1	2	3	2
0	1	2	3	4	5	6	7	8

- Recall: `coin_change[8] = min(coin_change[8-1], coin_change[8-4], coin_change[8-5]) + 1`
- Work backward from index 8 to index 0 to recover the 2 coins that sum up to 8

0	2	0
1	4	5



# DTE-2501 AI Methods and Applications

*Linear methods of classification and regression*

*Lecture 1/2 – Stochastic gradient descent*

Tatiana Kravetc  
*Førsteamanuensis*

Office: D2240

Email: [tatiana.kravetc@uit.no](mailto:tatiana.kravetc@uit.no)

# Overview

- I Linear model
- II Numerical minimization
  - a) Gradient descent
  - b) Stochastic gradient
  - c) Extensions and their comparison

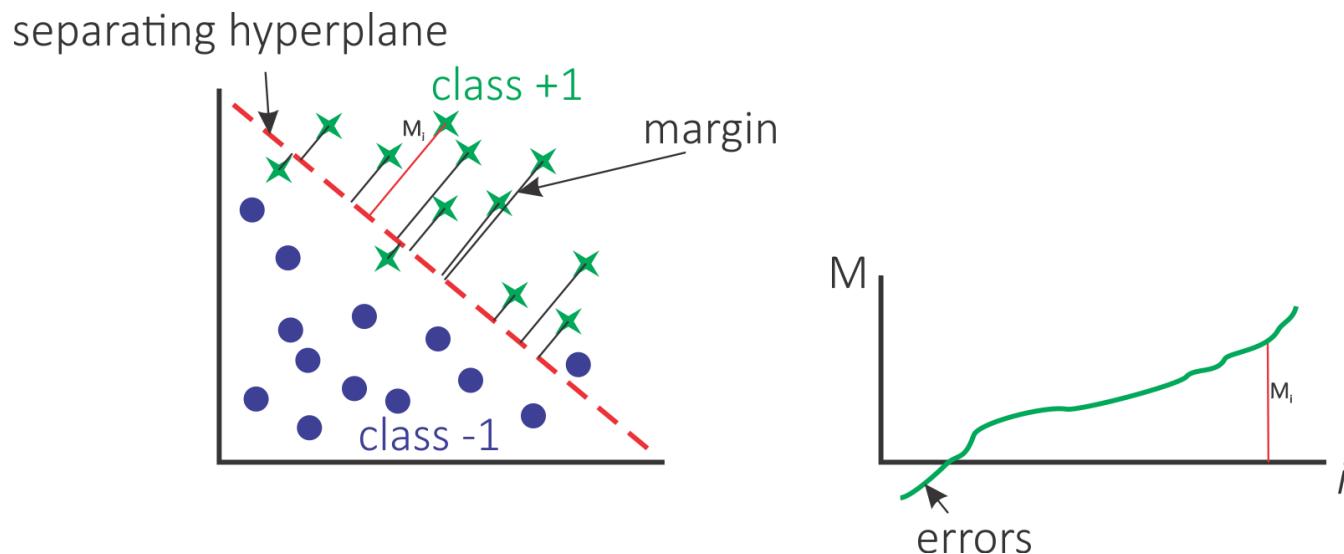
# I Linear model

Regression	Classification
<ul style="list-style-type: none"> <li>Training sample  <math>X^l = (x_i, y_i)_{i=1}^l, x_i \in \mathbb{R}^n, y_i \in \mathbb{R}</math></li> <li>Linear model. <math>\langle \cdot, \cdot \rangle</math> is a scalar product  <math>a(x, w) = \langle x, w \rangle = \sum_{j=1}^n w_j f_j(x), w \in \mathbb{R}^n</math></li> <li>Quadratic loss function  <math>\varepsilon(a, y) = (a(x, w) - y(x))^2</math></li> <li>Training phase. Learning method is a <i>least squares method</i>. <math>Q</math> is an <i>empirical risk</i>  <math display="block">Q(w) = \sum_{i=1}^l (a(x_i, w) - y_i)^2 \rightarrow \min_w</math></li> <li>Testing phase. Test sample <math>X^k = (\tilde{x}_i, \tilde{y}_i)_{i=1}^k</math>  <math display="block">\tilde{Q}(w) = \frac{1}{k} \sum_{i=1}^k (a(\tilde{x}_i, w) - \tilde{y}_i)^2</math></li> </ul>	<ul style="list-style-type: none"> <li>Training sample  <math>X^l = (x_i, y_i)_{i=1}^l, x_i \in \mathbb{R}^n, y_i \in \{-1, +1\}</math></li> <li>Linear model  <math>a(x, w) = \text{sign} \langle x, w \rangle = \text{sign} \sum_{j=1}^n w_j f_j(x)</math></li> <li>Binary loss function, or its <i>approximation</i>  <math>\varepsilon(a, y) = [a(x, w)y(x) &lt; 0] = [\langle x, w \rangle y &lt; 0] \leq \varepsilon(\langle x, w \rangle y)</math></li> <li>Training phase. Learning method is a minimization of the empirical risk  <math display="block">Q(w) = \sum_{i=1}^l [a(x_i, w)y_i &lt; 0] = \sum_{i=1}^l \varepsilon(\langle x_i, w \rangle y_i) \rightarrow \min_w</math></li> <li>Testing phase. Test sample <math>X^k = (\tilde{x}_i, \tilde{y}_i)_{i=1}^k</math>  <math display="block">\tilde{Q}(w) = \frac{1}{k} \sum_{i=1}^k [\langle \tilde{x}_i, w \rangle \tilde{y}_i &lt; 0]</math></li> </ul>

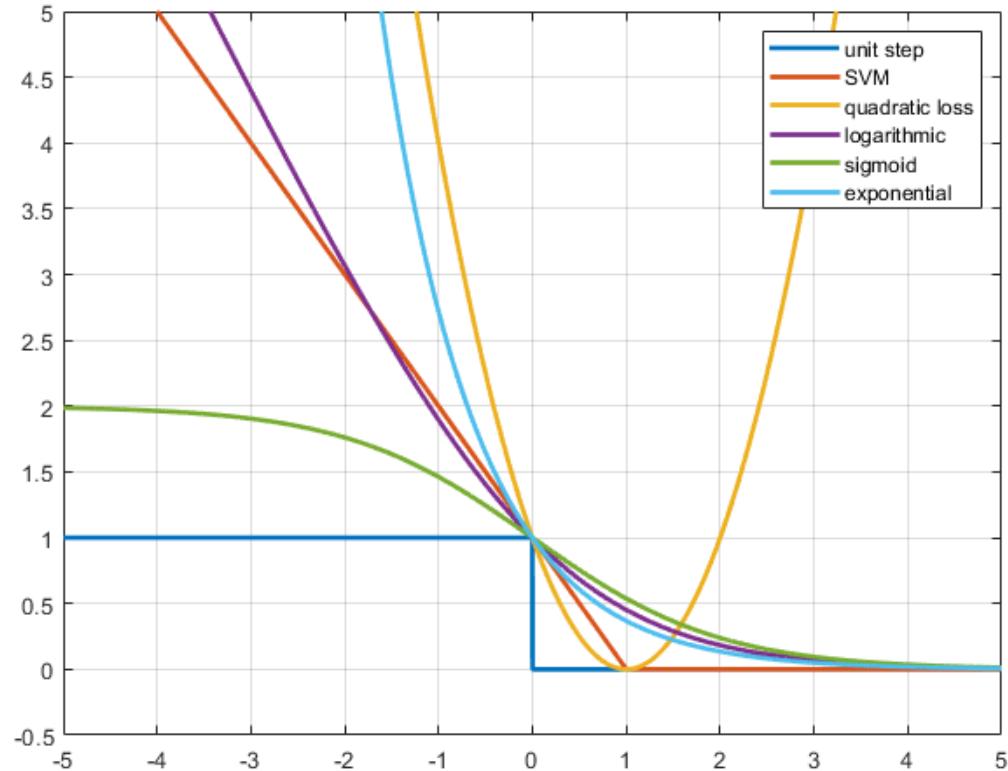
# Margin

Classifier  $a(x, w) = \text{sign}\langle x, w \rangle$

- $\langle x, w \rangle = 0$  is a separating hyperplane equation
- $M_i = \langle x_i, w \rangle y_i$  is a margin of the object  $x_i$
- $M_i < 0$  means an error of the algorithm  $a(x, w)$  on the object  $x_i$



# Continuous approximations of the threshold function



- $[M < 0]$  – the unit step (zero-one loss)
- $(1 - M)_+$  – piecewise linear (SVM)
- $(1 - M)^2$  – quadratic loss
- $\log_2(1 + e^{-M})$  – logarithmic
- $2(1 + e^M)^{-1}$  – sigmoid
- $e^{-M}$  – exponential

# II Numerical minimization

## Gradient descent

Empirical risk minimization

$$Q(w) = \sum_{i=1}^l \varepsilon_i(w) \rightarrow \min_w$$

Numerical minimization using a *gradient descent method*:

$w^{(0)}$  is an initial guess

$$w^{(t+1)} := w^{(t)} - h \cdot \nabla Q(w^{(t)}), \quad \nabla Q(w) = \left( \frac{\partial Q(w)}{\partial w_j} \right)_{j=0}^n$$

where  $h$  is a gradient step, which is also called a *learning rate*.

$$w^{(t+1)} := w^{(t)} - h \sum_{i=1}^l \nabla \varepsilon_i(w^{(t)})$$

# Stochastic gradient

**Input:** dataset  $X^l$ , learning rate  $h$ , parameter  $\lambda$

**Output:** weights  $w$

## Initialization

Set all the weights  $w_j, j = 0, \dots, n$  to small random numbers

Evaluate the objective function  $Q = \frac{1}{l} \sum_{i=1}^l \varepsilon_i(w)$

**do**

  pick randomly  $x_i$  from  $X^l$

  compute the loss function  $\varepsilon_i(w)$

  perform the gradient step  $w := w - h \nabla \varepsilon_i(w)$

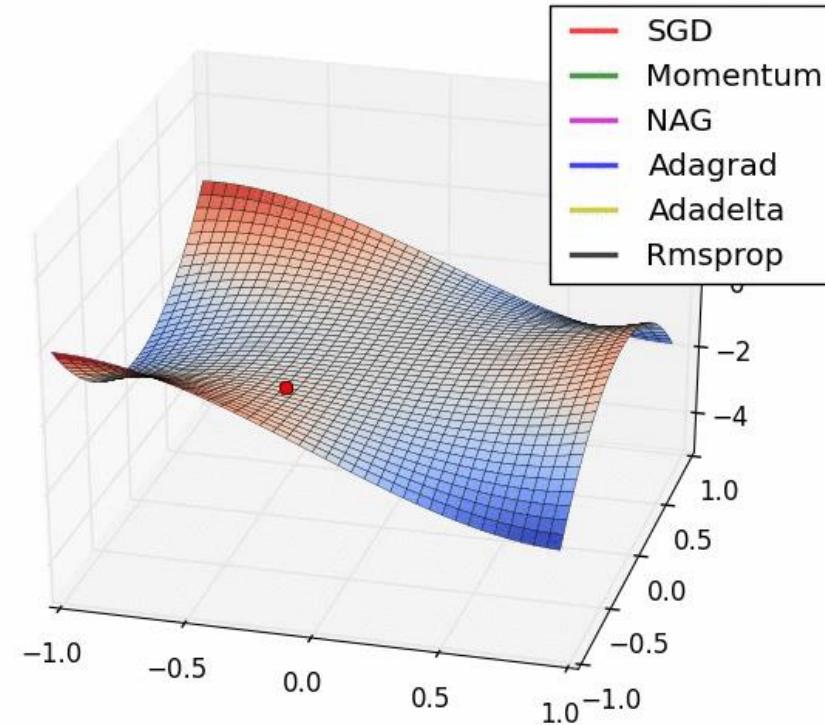
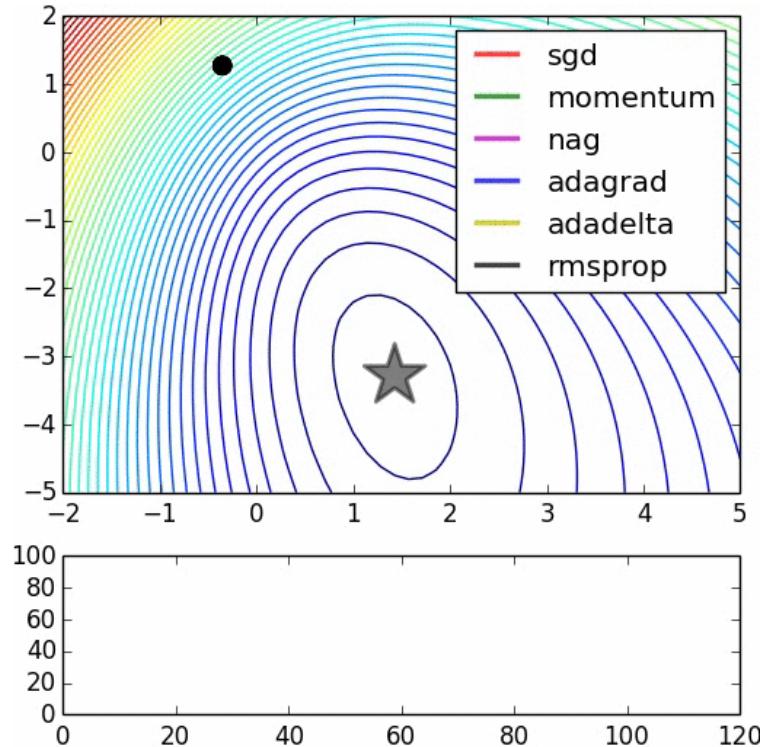
  update the objective function  $Q := \lambda \varepsilon_i + (1 - \lambda)Q$

**until**  $Q$  and/or  $w$  converges

# Extensions

- **Stochastic average gradient (SAG)**  
The algorithm records an average of its parameter vector over time
- **Momentum**  
The algorithm updates the weights as a linear combination of the gradient and the previous update
- **RMSProp (running mean square propagation)**  
The learning rate is adapted for each of the parameters
- **AdaGrad (adaptive gradient)**  
The algorithm increases the learning rate for sparser parameters and decreases for ones that are less sparse
- etc...

# Comparison of optimization algorithms



Alec Radford's animation for optimization algorithms:

<http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html>

### III Practical example

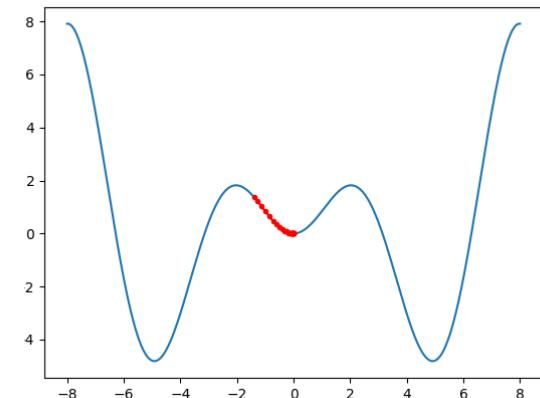
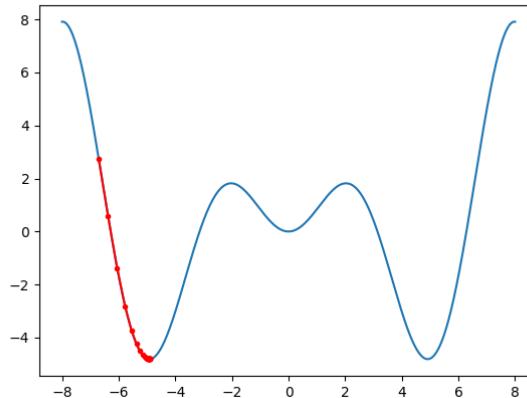
Objective function  $Q = x \sin x$  on  $[-8,8]$

Derivative of the objective function  $Q' = \sin x + x \cos x$

```
8 # objective function
9 def obj(x):
10     return x * sin(x)
11
12 # derivative of objective function
13 def derv(x):
14     return sin(x) + x * cos(x)
15
16
17 # gradient descent algorithm
18 def gradient_descent(objective, derivative, bounds, n_iter, step_size):
19     solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
20     for i in range(n_iter):
21         gradient = derivative(solution)
22         solution = solution - step_size * gradient
23         solution_eval = objective(solution)
24     return [solution, solution_eval]
25
26
27
28 bound = asarray([[-8.0, 8.0]])
29 n = 30
30 h = 0.1
31 sol, sol_eval = gradient_descent(obj, derv, bound, n, h)
```

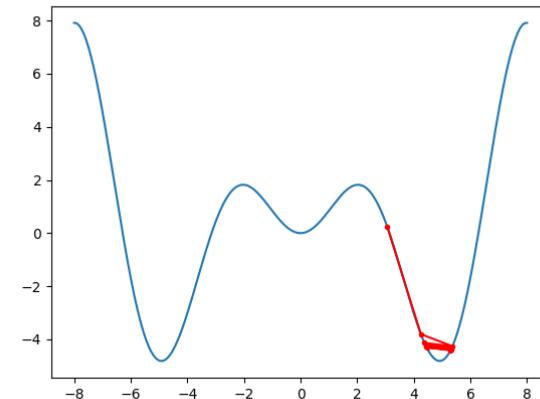
Result:

Local minima depends on starting point



Not optimal step size ( $h = 0.4$ )

} random initialization  
}  $x := x - hQ'(x)$





# DTE-2501 AI Methods and Applications

*Linear methods of classification and regression*

*Lecture 2/2 – Probabilistic learning model*

Tatiana Kravetc

Førsteamanuensis

Office: D2240

Email: [tatiana.kravetc@uit.no](mailto:tatiana.kravetc@uit.no)

# Overview

IV Probabilistic model

V Basic terminology

VI Naïve Bayes classifier

VII Example

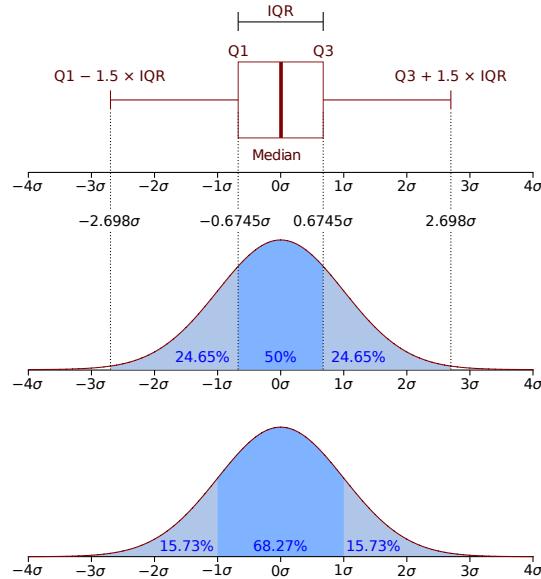
# IV Probabilistic model

$x_i \in X^l$  is the available data about a real object.

The data can be:

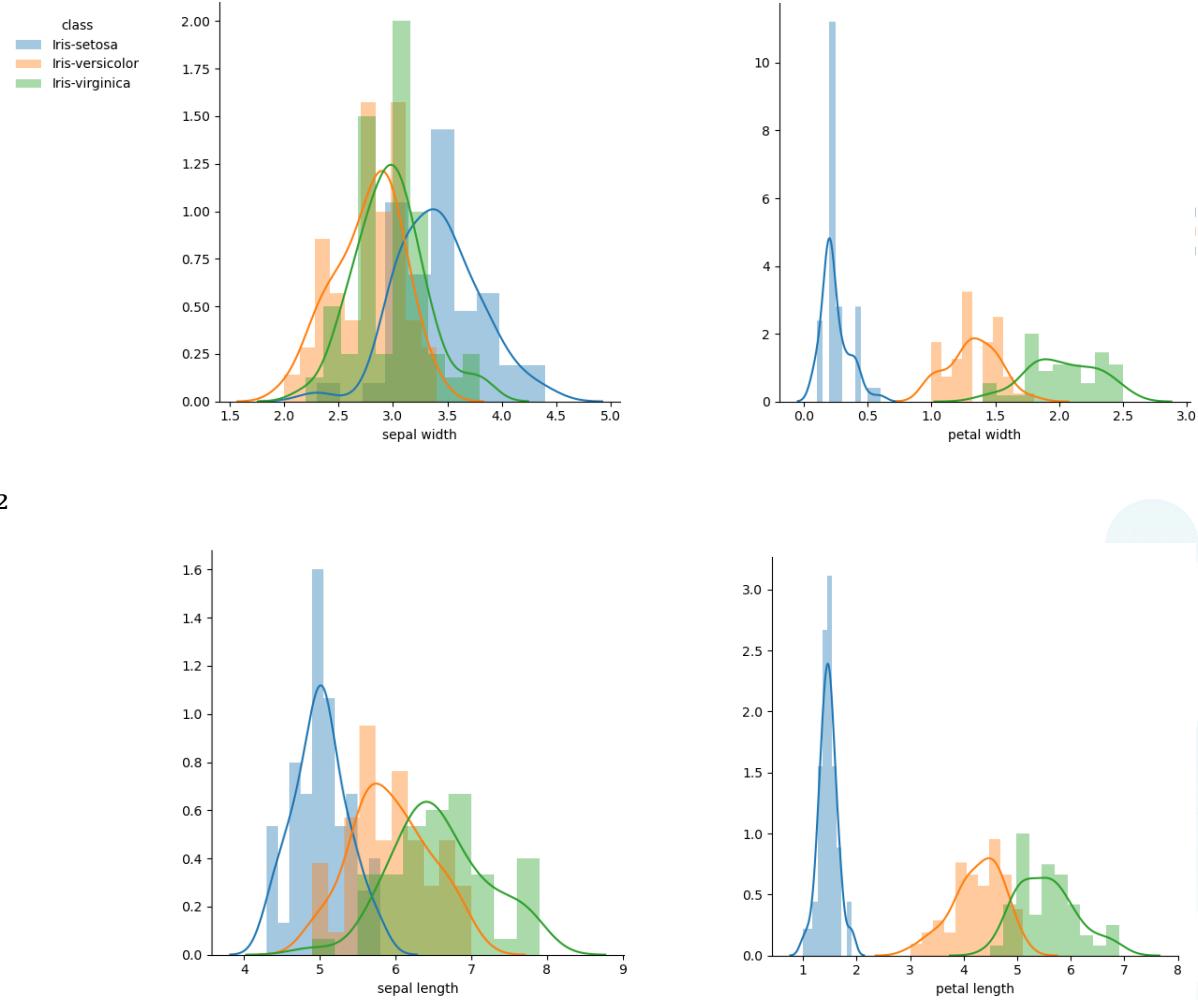
- erroneous
- incomplete

The incorrectness can be eliminated by introducing  
*a probabilistic formulation* of the problem



$$f(x; \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$$

$\mu$  is the mean  
 $\sigma^2$  is the variance  
 $\sigma$  is the standard deviation



# V Basic terminology

- Probability density of  $x = P(x)$
- Probability of a specific event  $A = P(x = A) = P(A)$
- Probability = (number of desired outcomes) / (total number of possible outcomes)
- Joint probability

$$P(A \cap B) = P(A | B) \cdot P(B)$$

If two events are *independent*, then  $P(A \cap B) = P(A) \cdot P(B)$

- Conditional probability

$$P(A | B) = P(A \cap B) / P(B)$$

# VI Naïve Bayes classifier

Given classes  $C_k$  and a sample  $x = (x_1, \dots, x_n)$  having  $n$  features to be classified

Bayes' theorem

$$P(C_k|x) = \frac{P(x|C_k)P(C_k)}{P(x)}$$

$P(C_k | x)$  is the posterior probability of class ( $C_k$ , target) given predictor ( $x$ , features)

$P(C_k)$  is the prior probability of class

$P(x | C_k)$  is the likelihood which is probability of predictor given class

$P(x)$  is the prior probability of predictor

Naïve conditional independence assumption

$$P(C_k|x_1, \dots, x_n) = P(x_1|C_k) \cdot P(x_2|C_k) \cdot \dots \cdot P(x_n|C_k) \cdot P(C_k)$$

## VII Example. Prediction problem

Historical data

Weather	Play
Sunny	No
Overcast	Yes
Rainy	Yes
Sunny	Yes
Sunny	Yes
Overcast	Yes
Rainy	No
Rainy	No
Sunny	Yes
Rainy	Yes
Sunny	No
Overcast	Yes
Overcast	Yes
Rainy	No

Frequency table

Weather ( $x$ )	Yes ( $C_0$ )	No ( $C_1$ )	$P(x)$
Sunny	3	2	=5/14 0.36
Overcast	4	0	=4/14 0.29
Rainy	2	3	=5/14 0.36
Total	9	5	
$P(C_k)$		=9/14 =5/14	
		0.64	0.36

Problem: Players will play if weather is sunny. Is this statement correct?

$$P(C_0 \mid \text{sunny}) = \frac{P(\text{sunny} \mid C_0) \cdot P(C_0)}{P(\text{sunny})} = \frac{\frac{3}{9} \cdot 0.64}{0.36} = 0.6$$

# Week4: Supervised and Unsupervised Learning

---

Shayan Dadman, PhD candidate  
UiT, Narvik



# Supervised learning

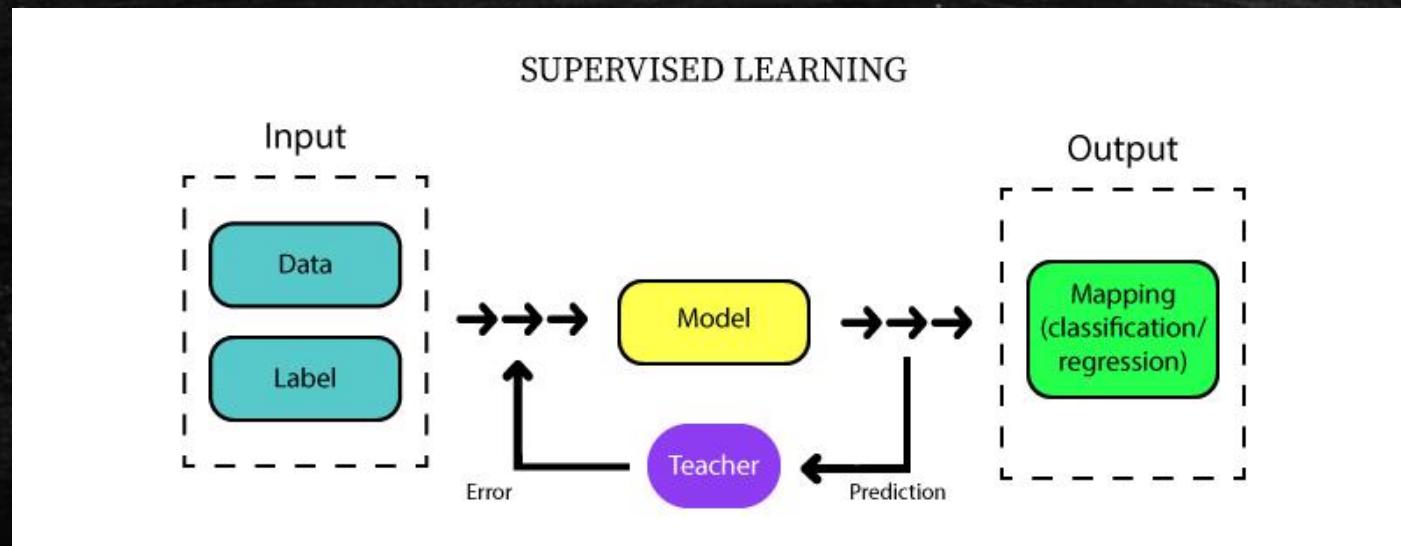
---

- Supervised learning is the standard approach in machine learning problems.
- It is called supervised learning, as a teacher supervises the learning process.
- In supervised learning, we provide input variables ( $x$ ) and output variables ( $Y$ ).
- The algorithms are used to learn the mapping function from the input to the output.



# Supervised learning

- Indeed, the goal is to obtain a mapping function that can predict output variables (Y), given the input variables (x).
- During the training process, we know the correct answers. Thus, the algorithm iteratively makes predictions on the given input variables (x) and is corrected by the teacher.



# Supervised learning

---

- Supervised learning is typically utilized in the context of classification and regression problems.
- Traditional algorithms in supervised learning are k-nearest neighbors, logistic regression, support vector machines, random forests.
- When applying supervised learning, the main concerns are the model complexity and the bias-variance tradeoff.



# Supervised learning: model complexity

---

- The model's complexity refers to the complexity of the function we are trying to learn.
- If the dataset is small or it is not uniformly spread out, then the low-complexity model is the way to go.
- This is because the high-complexity model will overfit if utilized on a small amount of data.
- Indeed, the model's learning function fits the training data very well that it cannot generalize to other data points.



# Supervised learning: bias-variance tradeoff

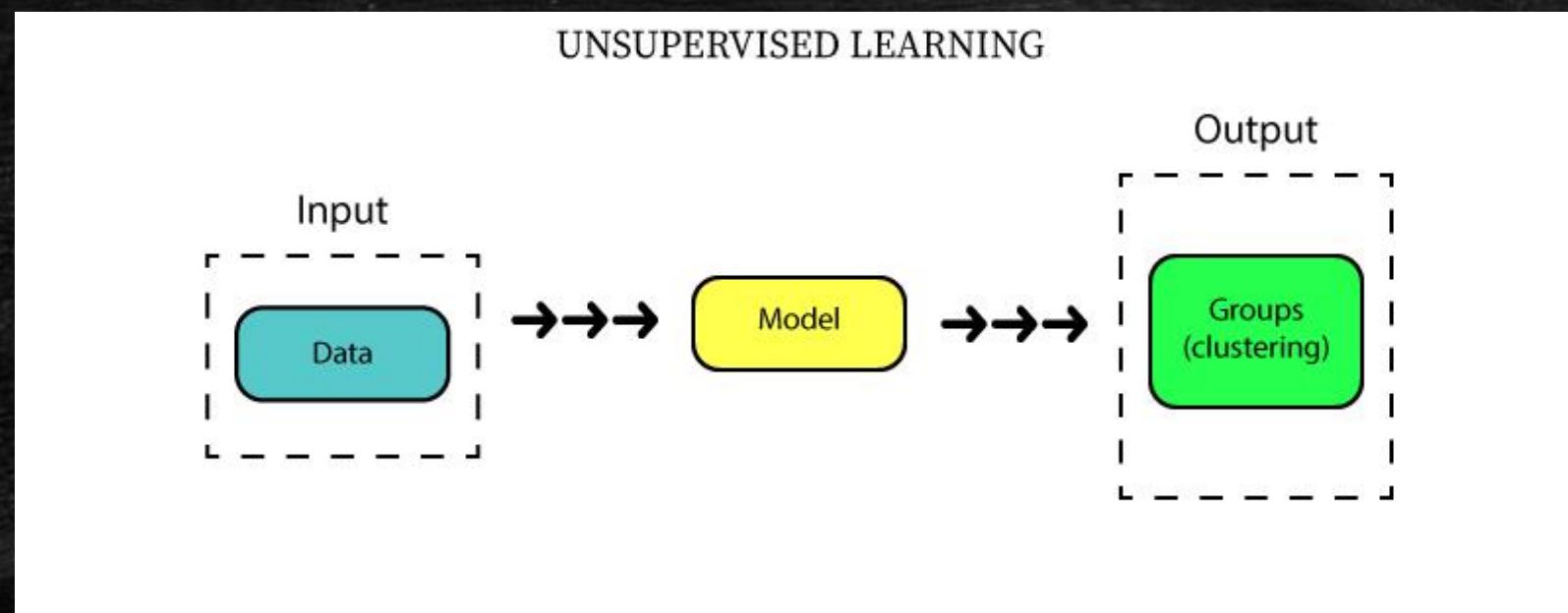
---

- The bias-variance tradeoff relates to model generalization.
- Bias is the constant error term. In other words, it is the difference between the average prediction of our model and the expected value.
- Variance indicates the distribution of model prediction for a given data point.
- The model with high bias pays little attention to the training data and oversimplifies the model. In fact, it results in a high error on training and test data.
- On the contrary, the model with high variance pays a lot of attention to training data. It performs well on training data but poorly on test data. Indeed, it lacks the ability to generalize on unseen data.



# Unsupervised learning

- Unsupervised learning is when you only have input variable (x).
- It is called unsupervised learning, because in contrary to supervised learning, there is no correct answers and a teacher.



# Unsupervised learning

---

- In unsupervised learning, the goal is to model the structure in the given input variables ( $x$ ) by extracting the features and analyzing its structure.
- In this approach, the algorithms are left to their own to find the existing structure in the given inputs.



# Unsupervised learning

---

- The typical tasks within unsupervised learning are clustering, representation learning, and density estimation.
- Common algorithms utilized for such tasks include k-mean clustering, principal component analysis, and autoencoders.
- There is no explicit way to examine model performance in unsupervised learning. It is because no labels are provided.



# Unsupervised learning

---

- Two prevalent use cases for unsupervised learning are exploratory analysis and dimensionality reduction.
- In an exploratory analysis, the unsupervised learning algorithms can be utilized to identify structure in data when it is either impossible or impractical for humans.
- In such situations, unsupervised learning can provide initial insights into the relationship between the given data points.
- In the case of dimensionality reduction, the unsupervised learning methods can be used to represent data in fewer columns or features. The principal component analysis is commonly used for this purpose.



# Questions

---

1. Email me at [Shayan.Dadman@uit.no](mailto:Shayan.Dadman@uit.no)
2. My office D3430



# Week4: K-Nearest Neighbors (KNN)

---

Shayan Dadman, PhD candidate  
UiT, Narvik



# What is KNN?

---

- KNN is a supervised machine learning algorithm that relies on labeled data.
- It can be used for both classification and regression problems.
- It is mostly considered due to its ease of interpretation and low calculation time.



# What is KNN?

---

- The KNN algorithm assumes that similar values are placed close to each other.
- $K$  in here is defined as the number of nearest neighbors.
- The KNN exhibits the idea of similarity by calculating the distance between the data points.
- Indeed, KNN considers  $K$  number of nearest neighbors to predict the class or continuous value for a new data point.



# What is KNN?

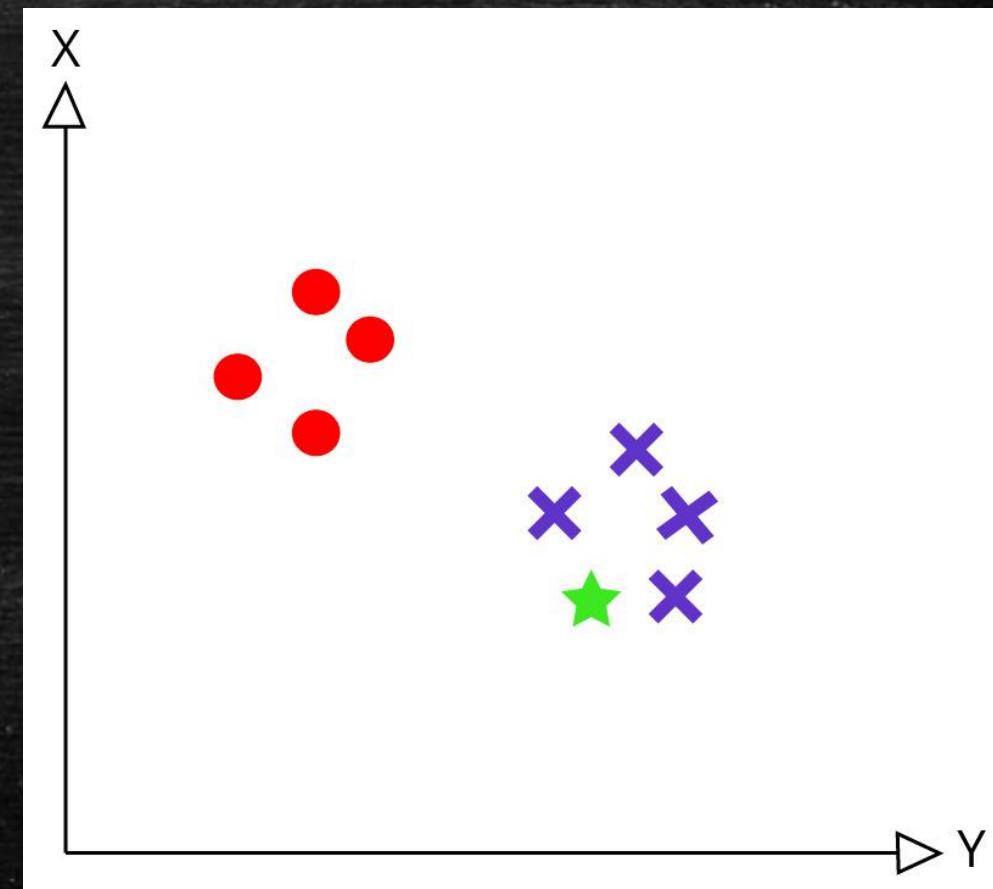
---

- The KNN algorithm has these characteristics:
  - It is instance-based learning, which uses the entire training examples to predict output for unseen data.
  - It uses a lazy learning method, in which the generalization of the training examples is postponed to a time when prediction is demanded on the new example.
  - It is non-parametric as it has no predefined form of the mapping function.



# How does the algorithm work?

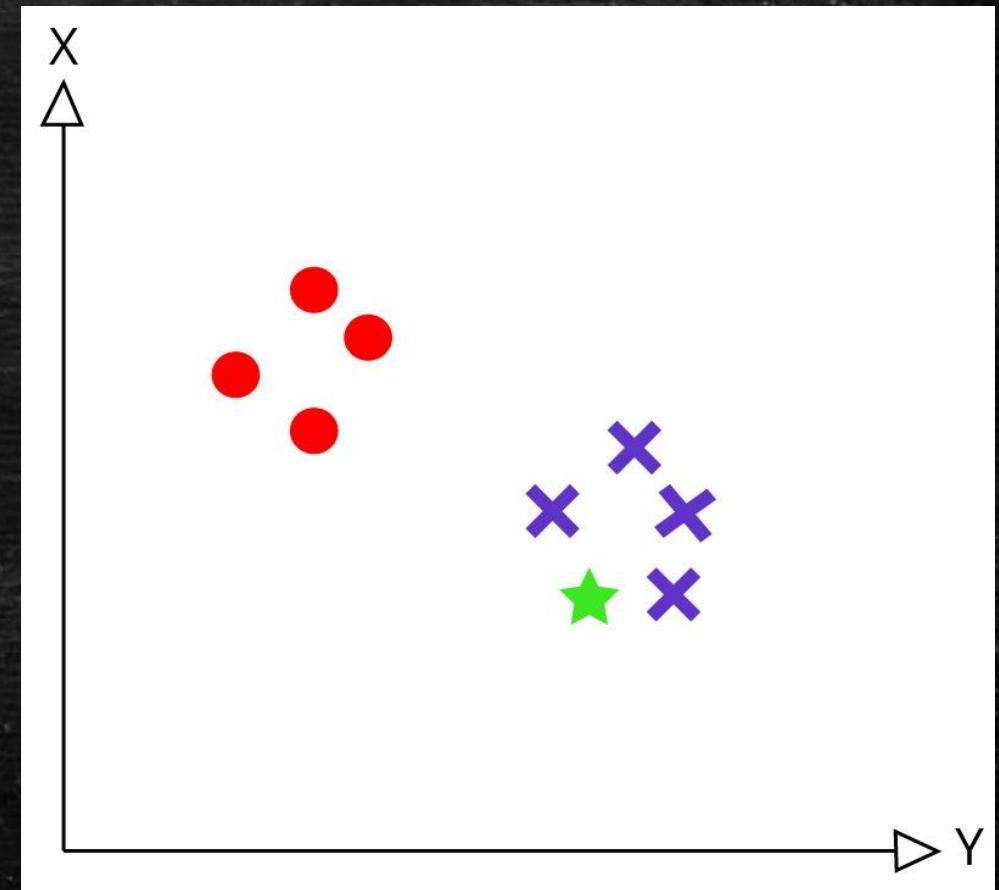
- In the following figure, we have a plot of the data points from our two dimensional feature space dataset.
- As we can see, we have a total of 8 data points, consist of 4 red and 4 purple.
- Red data points belong to **class1** and purple data points belong to **class2**.
- Green data point represents the new point, which a class is to be predicted.



# How does the algorithm work?

---

- Clearly, it belongs to **class2** (purple points).
- Why? because its nearest neighbors are those data points that have minimum distance.



# How does the algorithm work?

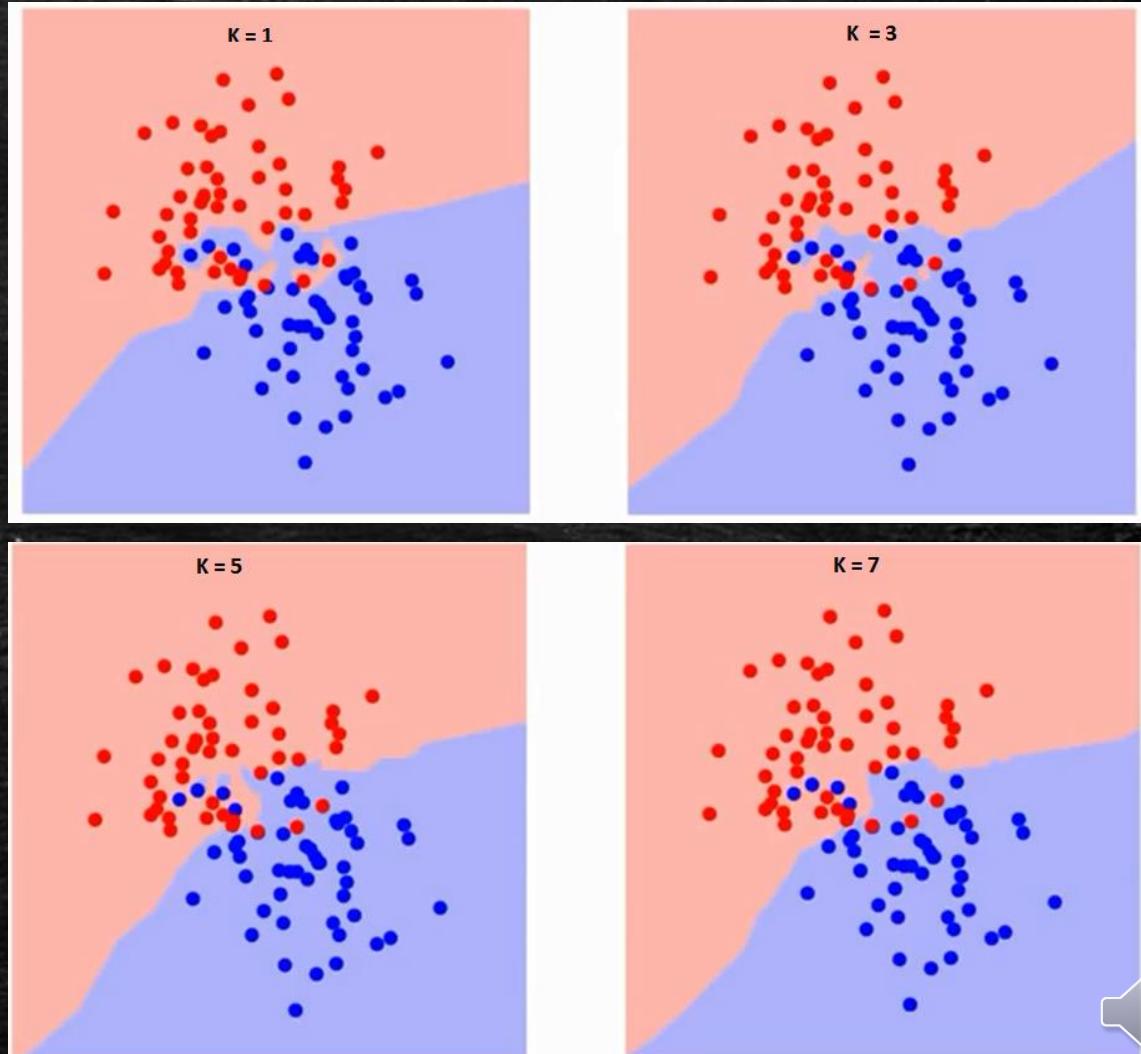
---

- They are two essential factors when using the KNN algorithm:
  - Distance metric
  - $K$  value
- Euclidean distance is the most common distance metric. Cosine, inverse document frequency, hamming distance, and manhattan distance can also be used based on the application.



# How does the algorithm work?

- The  $K$  value is the number of data points that we consider when calculating the minimum distance.
- In the following figures, you can see different  $K$  values used to separate two classes.
- As it can be seen, by increasing the  $K$  values, the boundary becomes smoother.
- In fact, increasing the  $K$  to infinity results in all blue or all red depending on the total majority.



# KNN pseudo code

---

1. Load the data
2. Initialize the value of  $K$
3. For each example in the data:
  1. Calculate the distance between test data and each row of training data.
  2. Add the distance and the index of the example to an ordered collection
4. Sort the calculated distances in ascending order based on distance values
5. Get top  $K$  entries from the sorted array
6. Get the labels of the selected  $K$  entries
7. Return the predicted class



# Questions

---

1. Email me at [Shayan.Dadman@uit.no](mailto:Shayan.Dadman@uit.no)
2. My office D3430



# Week4: K-Means

---

Shayan Dadman, PhD candidate  
UiT, Narvik



# What is K-Means?

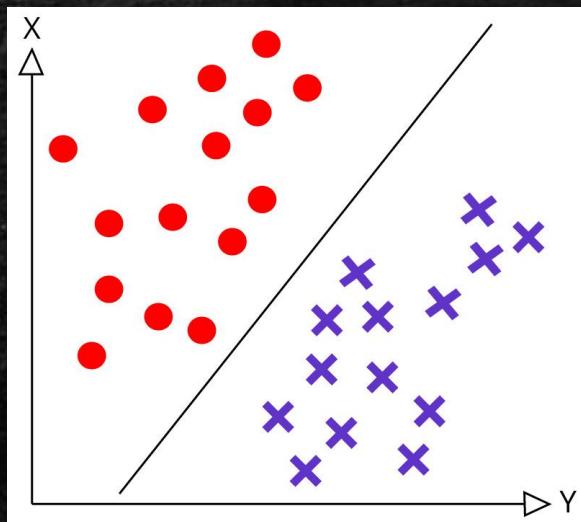
---

- K-Means is a clustering unsupervised learning algorithm.
- It has a wide range of applications such as recommendation systems, customer segmentation, document clustering, and image segmentation.
- K-Means groups the attributes in the unlabeled dataset into clusters.
  - "K" refers to the number of clusters.
  - "Means" relates to the cluster centroid, which determines by the average of the cluster content.
- The main goal of the K-Means algorithm is to minimize the sum of the distances between the data points and their corresponding clusters.

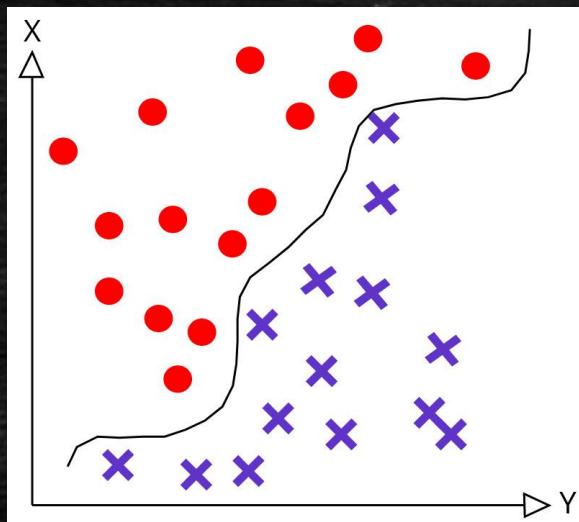


# What is clustering?

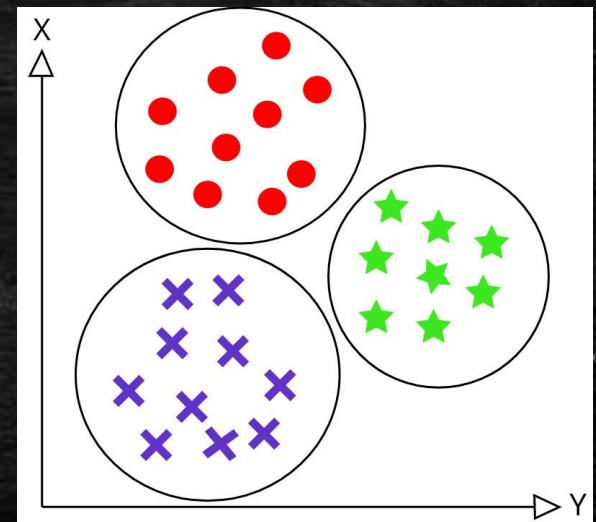
- Clustering divides the entire data into groups (clusters) based on the similarities between the data points.



Linear classification



Non-linear classification



Clustering



# What is feature vector?

---

- The feature is a list of values, for example, age, name, and height.
- The feature vector is an n-dimensional vector of features that represent a particular object or observation.
- For example, in the table below, columns are the features, and each row is a feature vector.

ID	First Name	Last Name	Email	Year of Birth
1	John	Johnson	john.johnson@university.edu	1992
2	Jack	Knife	jack.knife@university.edu	1982
3	Chris P.	Bacon	Chrispbacon@university.edu	1994
4	David	Letty	David.letty@university.edu	1976



# How does the algorithm work?

---

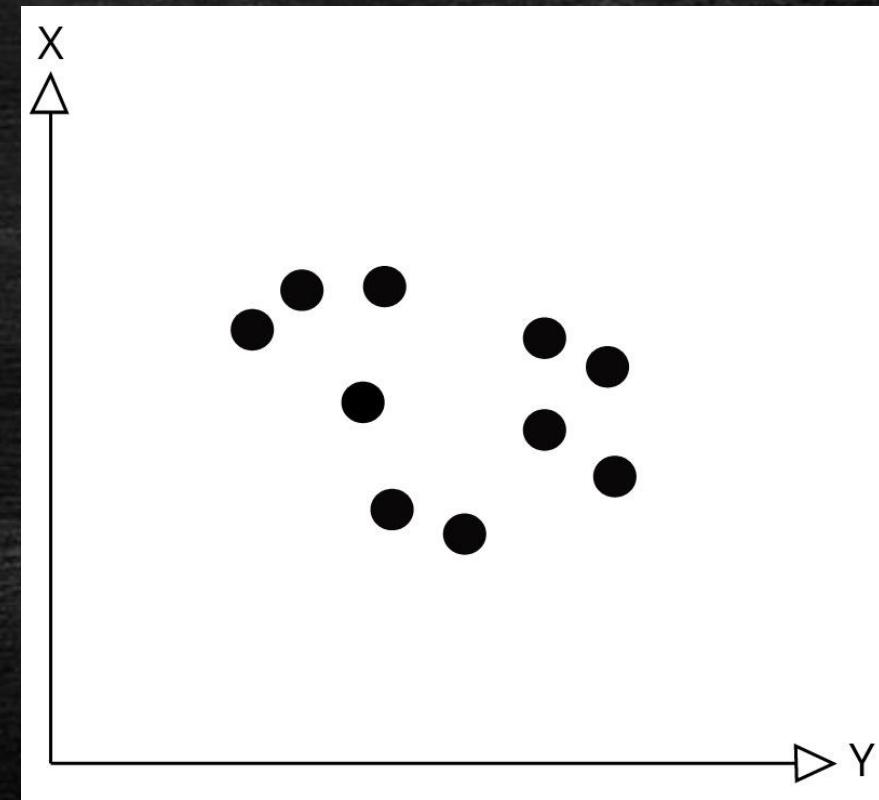
- The overall procedure is:
  - taking the unlabeled dataset
  - Dividing them into k-number of clusters
  - Iterating until cannot find the best clusters
- We choose a cluster for a given feature vector by calculating the Euclidean distance between the feature vector and the available cluster centroids.
  - By assigning a new feature to a cluster, the centroid of that cluster is also changed.
  - We update the centroids with each new entry.
- During the process, the feature vectors can move from one cluster to another.



# How does the algorithm work?

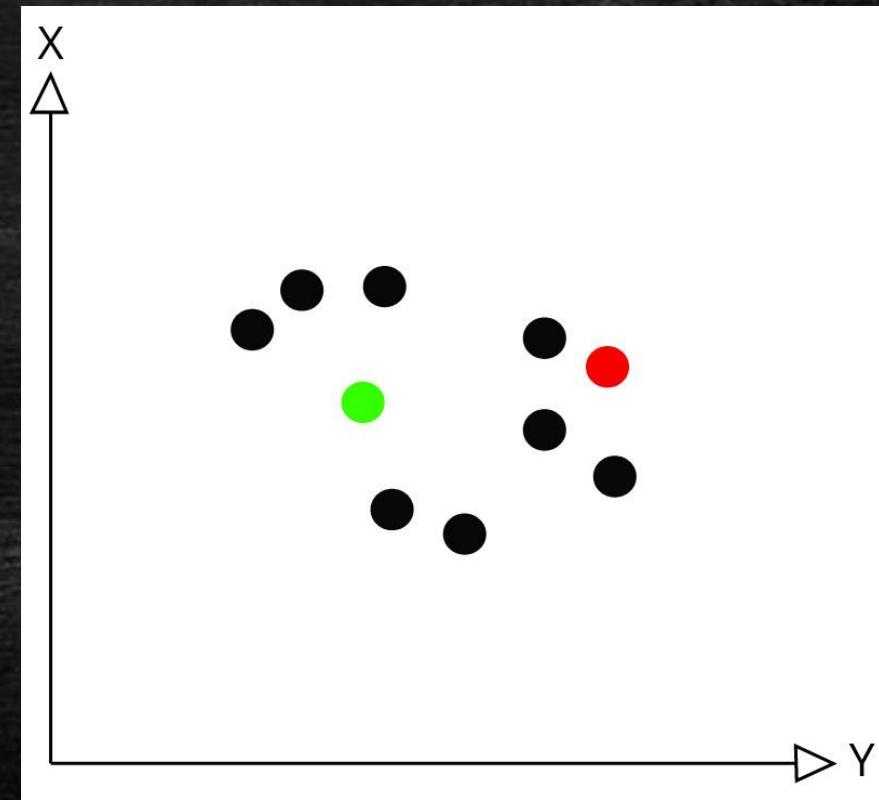
---

- Here we have a plot of our two dimensional feature space data points.
- First we pick our number of clusters. In this case we choose two clusters.



# How does the algorithm work?

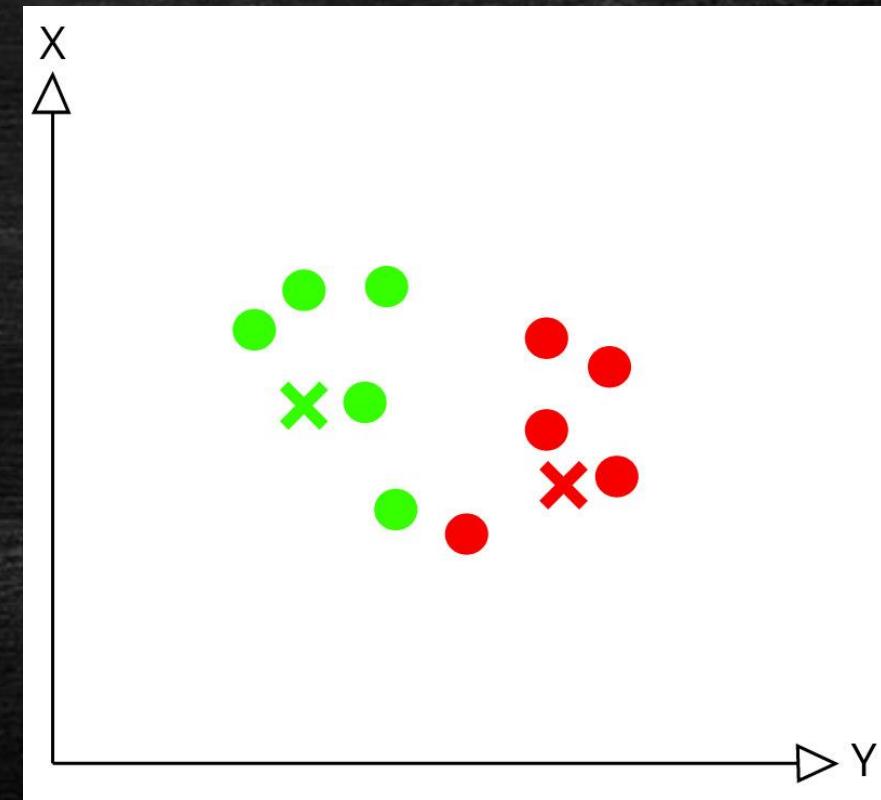
- Since our K is equal to two, we select two centroids randomly.
- Green dot is the centroid 1 and red dot is the centroid 2.
- We calculate the distance and assign the data points to the nearest cluster.



# How does the algorithm work?

---

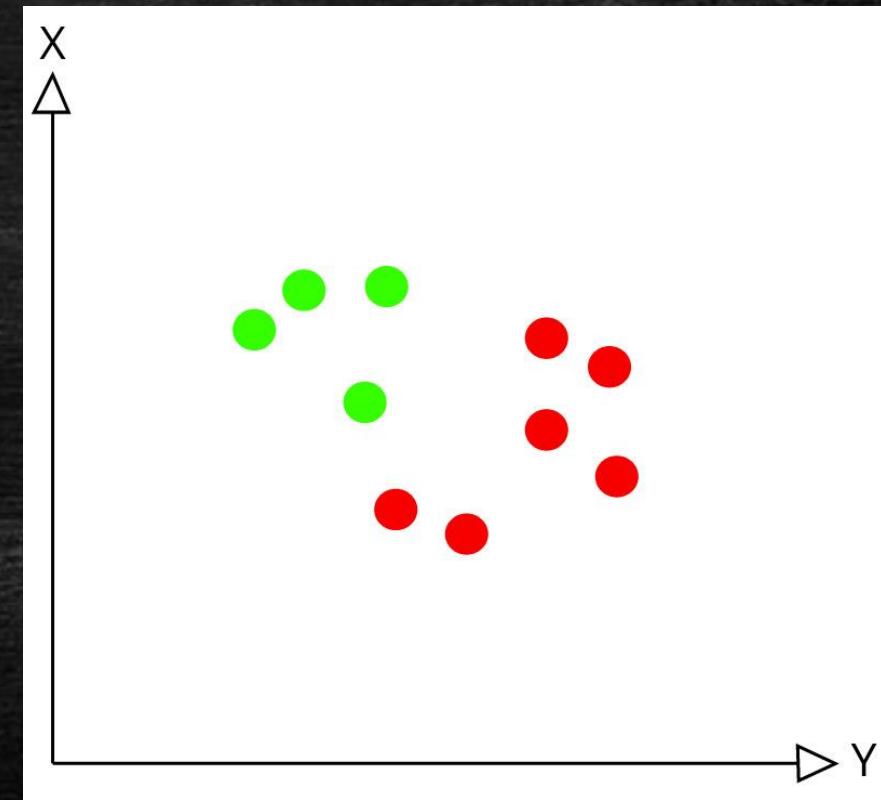
- In the next step, we recalculate the centroids of the clusters.
- The green and red crosses represent the new centroids.



# How does the algorithm work?

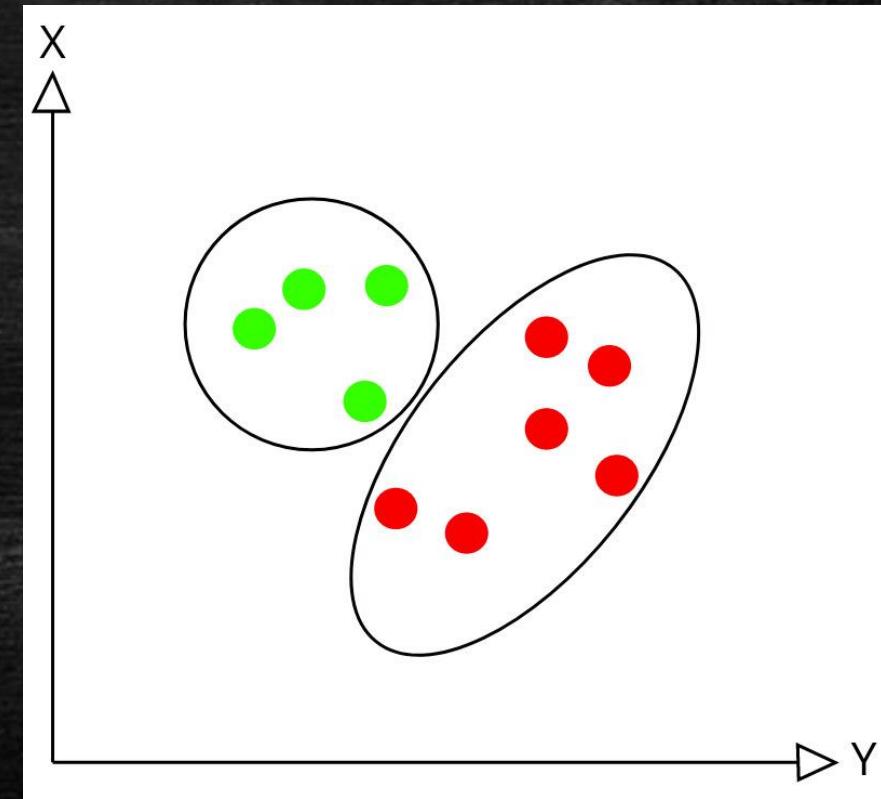
---

- Having the new centroids, we repeat the process by:
  - recalculating the distances between the data points and the centroids.
  - Updating the cluster's centroids.



# How does the algorithm work?

- We stopped since the clusters are formed. However, to stop the algorithm, we need to define criteria.
- There are three main criteria:
  - The centroids do not change after the clusters are updated
  - The data points remain in the same cluster
  - The process reached a certain number of iterations



# K-Means pseudo code

---

1. Load the data
2. Initialize the value of K to decide the number of clusters
3. Select random K points or centroids
4. Assign each data point to their closest centroid, which will form the predefined K clusters.
5. Calculate the variance and update a new centroid of each cluster.
6. Repeat the fourth step, which means reassign each data point to the new closest centroid of each cluster.
7. Stop if the criteria are met; else, go to the fifth step.



# Questions

---

1. Email me at [Shayan.Dadman@uit.no](mailto:Shayan.Dadman@uit.no)
2. My office D3430



# Swarms and evolution

The concept of Zero-Intelligence agents

Principle of swarms

Evolution and adaption

Genetics

UiT

NORGES  
ARKTISKE  
UNIVERSITET

# Swarms



Proceedings of the  
European Conference  
on Artificial Life 2015



edited by  
Paul Andrews, Leo Caves, René Doursat,  
Simon Hockinbotham, Fiona Polack,  
Susan Stepney, Tim Taylor and Jon Timmis

# What is Swarm Intelligence?

**Swarm intelligence** (SI) is a subfield of artificial intelligence. The concept originated in the field of robotics and refers to the amplified intelligence of flocks of birds, colonies of ants or swarms of bees, vs the performance of their individual members. SI is concerned with the design of multiagent systems modelled after the collective behavior of these self-organized animal populations.

The brainpower, self-organization and problem-solving abilities of such collectives in nature have always exercised fascination on researchers and scientists. More recently, they've inspired advances and applications in robotics and optimization. Now artificial swarm intelligence is brought within the reach of human groups as well.

<http://www.megamification.com/unu-the-platform-that-gamifies-human-swarm-intelligence/>

## TECH &amp; SCIENCE

# Swarm Intelligence: AI Algorithm Predicts the Future

BY ANTHONY CUTHBERTSON ON 1/25/16 AT 1:45 PM EST



A new A.I. algorithm that enables decision making through swarm intelligence could revolutionize democracy and transform healthcare.

MANUEL PRESTI/SCIENCE SOURCE

SHARE



XL  
BYGG  
PROFF  
Ski Bygg = XL-BYGG  
Proffens naturlige førstevalg

## THE DEBATE

Georgia's New Voting Law Is Racist  
BY SCOTT DWORKIN

VS

Georgia's Voting Law Doesn't Go Far Enough  
BY CHARLIE KIRK

## OPINION

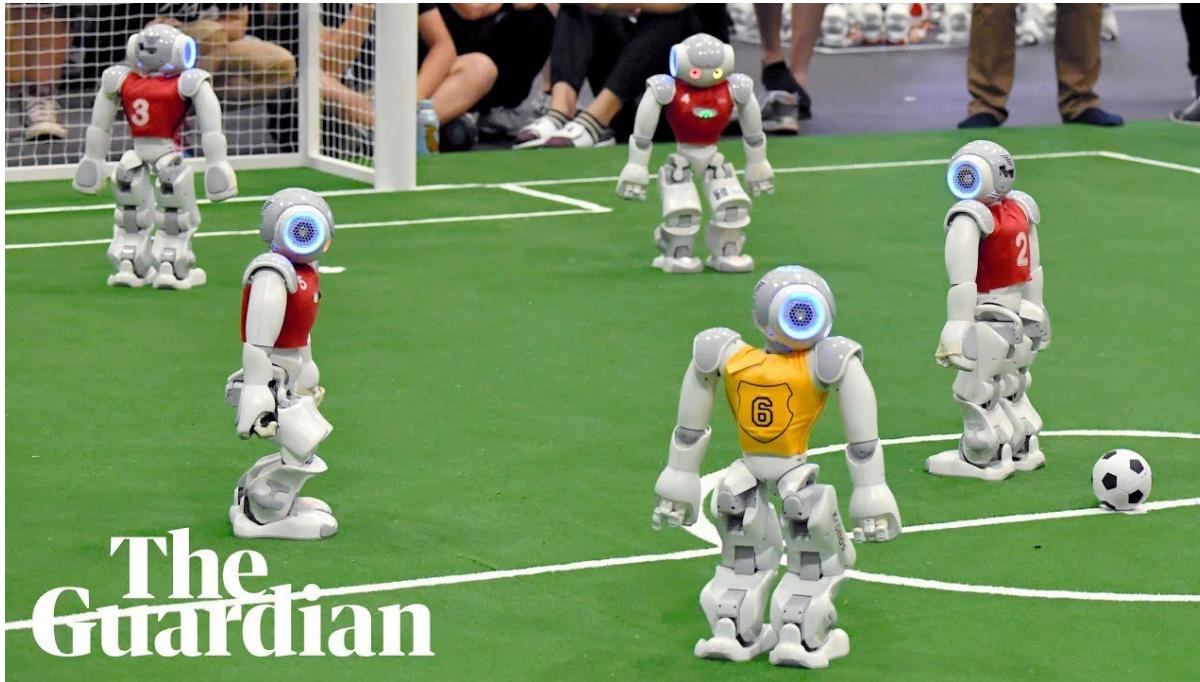
The Crisis of American Civilization  
BY NEWT GINGRICH

New Research Shows Higher Education Makes People More Anti-Semitic  
BY FREDERICK M. HESS AND HANNAH WARREN

We Need to Be Outraged About Birth Control Blood Clots Too  
BY SAM STROOZAS



# Stacks (Smarties)



# Colonies

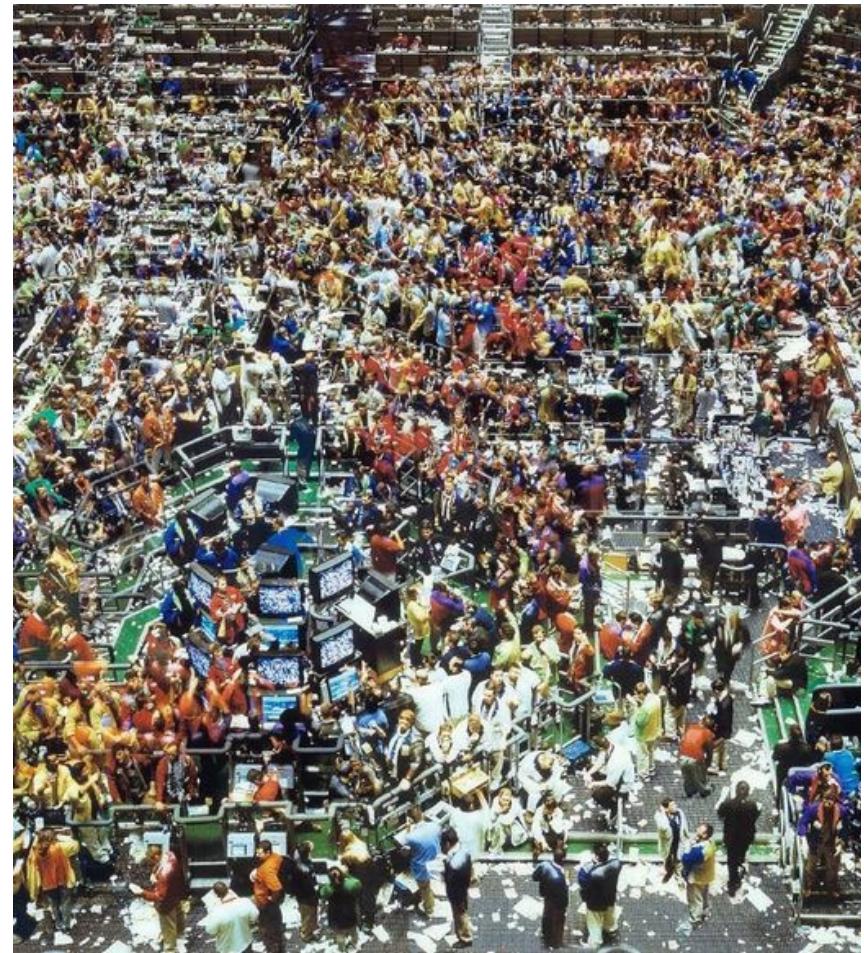
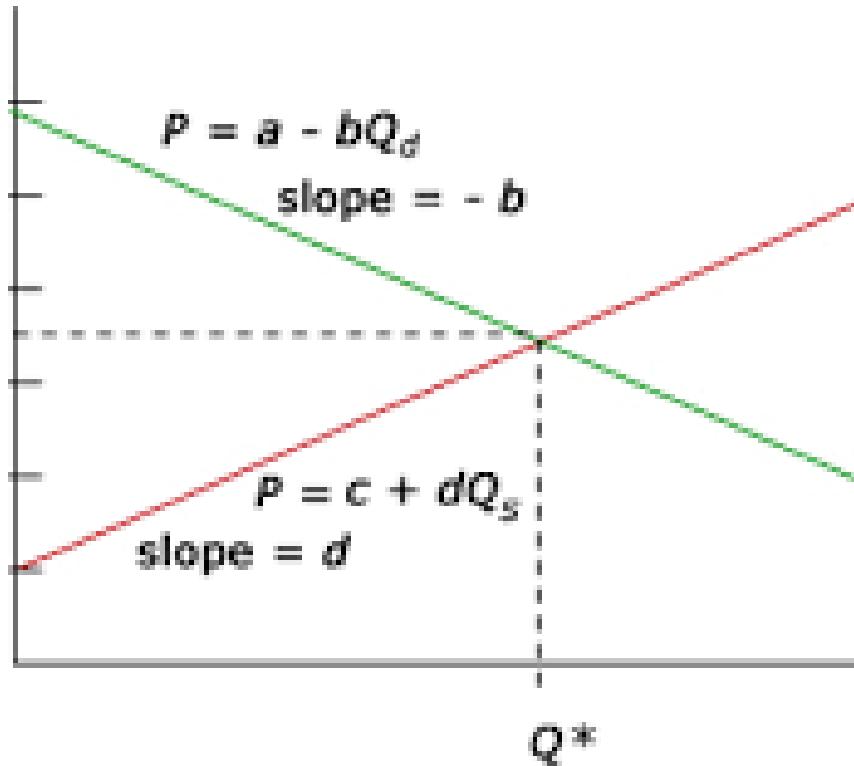
## *Ant Colony Optimization*

*Marco Dorigo and Thomas Stützle*



# Markets

## Market Equilibrium



UiT

NORGES  
ARKTISKE  
UNIVERSITET



YALE SCHOOL OF  
MANAGEMENT

Portal Alumni Give Recruiters School Leadership & Boards [Apply Now](#)

Mission Programs Faculty Research & Centers Community About [Q](#)

# The First Conference on Zero/Minimal Intelligence Agents [Virtual]

October 22 – 24, 2020



## The First Conference on Zero/Minimal Intelligence Agents

Yale School of Management and the  
Cowles Foundation for Research  
in Economics  
October 22-24, 2020

# Principal papers on swarms

## Artificial Swarm Intelligence vs Vegas Betting Markets

Louis Rosenberg  
Unanimous AI  
San Francisco, CA, USA  
Louis@Unanimous.ai

Gregg Willcox  
Unanimous AI  
San Francisco, CA, USA  
Gregg@Unanimous.ai

**Abstract**— In the natural world, Swarm Intelligence (SI) is a commonly occurring process in which biological groups amplify their collective intelligence by forming closed-loop systems. It is well known that swarms of flocks of birds, schools of fish, and human groups to form systems modeled after natural swarms, known as Artificial Swarm Intelligence (ASI), the technique has been used to amplify the effectiveness of AI systems. This study compares the predictive ability of ASI systems against large-scale betting markets when forecasting sporting events. Groups of average bettors were pitted against the collective outcome of 200 hockey games (10 games per week for 20 weeks) in the NHL. The expected win rate for Vegas betting markets across the 200 games was 85%. The probability that the system outperformed Vegas by chance was extremely low ( $p < 10^{-67}$ ). This study is a significant step forward in research combining the wisdom from two betting models—one that wagered weekly on the Vegas football and basketball teams versus on the NHL. At the end of 20 weeks, the Vegas model generated a 41% financial gain, while the ASI model generated a 170% financial gain.

**Keywords**— Swarm Intelligence, Artificial Swarm Intelligence, Collective Intelligence, Human Swarming, Artificial Intelligence.

Prior studies on Artificial Swarm Intelligence (ASI) have shown that by giving real-time “human swarms,” networked human groups can significantly amplify their accuracy in a wide

Behav Ecol Sociobiol (1999) 45: 19–31

© Springer-Verlag 1999

ORIGINAL ARTICLE

Thomas D. Seeley · Susannah C. Buhrman

**Group decision making in swarms of honey bees**

Received: 26 February 1998 / Accepted after revision: 16 May 1998

**Abstract** This study renews the analysis of honey bee swarming decision-making. We report on the results of our observations of swarms choosing future home sites but used modern videorecording and bee-labeling techniques to produce a finer-grained description of the decision-making process than was possible 40 years ago. Our results show that the process is iterative and reveal several new features of the decision-making process. Viewing the process at the group level, we found: (1) the scout bees in a swarm find potential nest sites in all directions and distances up to 1 km from the hive; (2) initially, the scouts advertise at dozen or more sites with their dances on the swarm, but eventually they just advertise just one site; (3) within about 1 h of the appearance of the first among the dancers, the swarm lifts off and flies to the chosen site; (4) there is a crescendo of dancing just before liftoff; and (5) the chosen site is not necessarily the one that is first advertised on the swarm. Viewing the process at the individual level, we found: (1) the number of workers in a dancing colony tapers off and eventually ceases, so that many dances drop out each day; (2) some scout bees switch their allegiance from one site to another; and (3) the principal

scout but also the most demanding in terms of information processing because it takes account of all of the information relevant to a decision problem. Despite being composed of small-brained bees, swarms are able to use the weighted additive strategy by distributing among many bees both the task of evaluating the alternative sites and the task of identifying the best of these sites.

**Key words** *Apis mellifera* · Communication · Dance language · Decision making · Swarming

### Introduction

One of the most spectacular examples of an animal group functioning as an adaptive unit is a swarm of honey bees choosing its future home. This phenomenon is called swarming and occurs when a colony, which has outgrown its hive and proceeds to divide itself by swarming. The mother queen and approximately half the worker bees leave the parental hive to establish a



J. R. Soc. Interface (2009) 6, 1065–1074  
doi:10.1098/rsif.2008.0511  
Published online 25 February 2009

## On optimal decision-making in brains and social insect colonies

James A. R. Marshall<sup>1,\*</sup>, Rafal Bogacz<sup>1</sup>, Anna Dornhaus<sup>2</sup>, Robert Planqué<sup>3</sup>,  
Tim Kovacs<sup>1</sup> and Nigel R. Franks<sup>4</sup>

<sup>1</sup>Department of Computer Science, University of Bristol, Woodland Road, Bristol BS8 1UB, UK

<sup>2</sup>Department of Ecology and Evolutionary Biology, University of Arizona, PO Box 210088, Tucson, AZ 85721, USA

<sup>3</sup>Department of Mathematics, VU University Amsterdam, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands

<sup>4</sup>School of Biological Sciences, University of Bristol, Woodland Road, Bristol BS8 1UG, UK

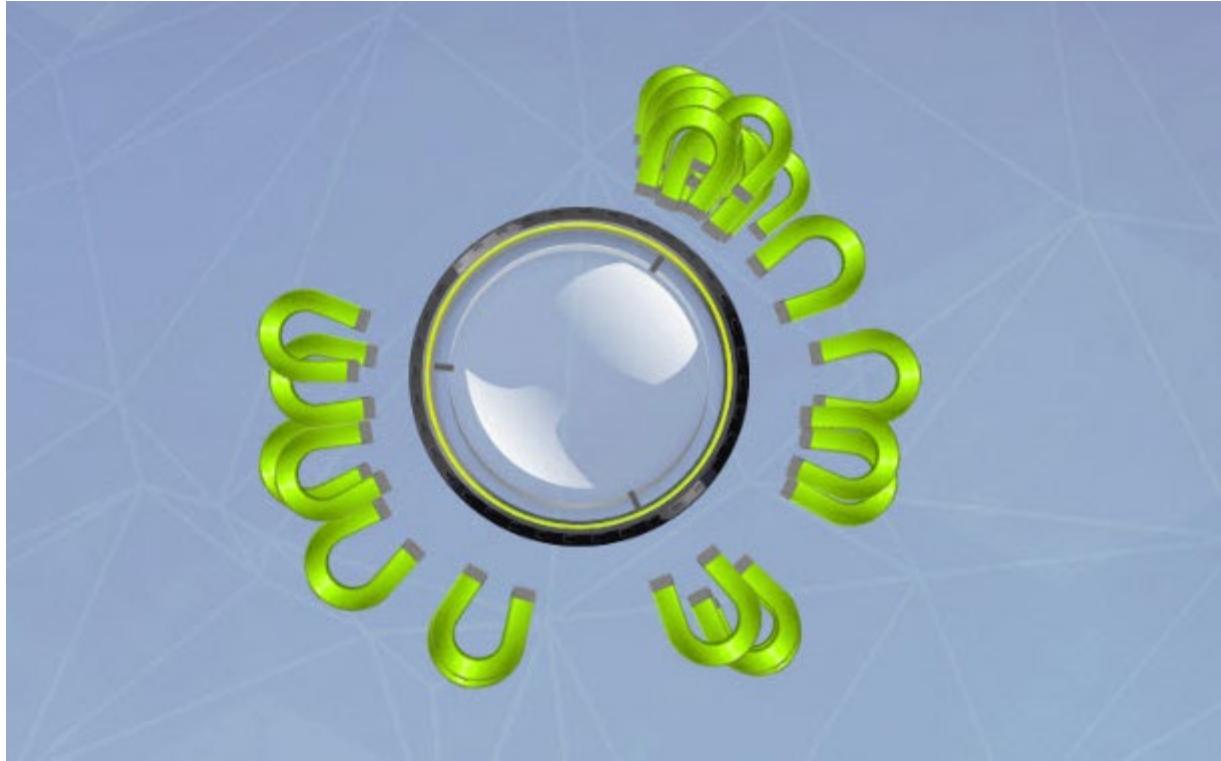
The problem of how to compromise between speed and accuracy in decision-making faces organisms at many levels of biological complexity. Striking parallels are evident between decision-making in primate brains and collective decision-making in social insect colonies: in both systems, separate populations accumulate evidence for alternative choices; when one population reaches a threshold, a decision is made for the corresponding alternative, and this threshold may be varied to compromise between the speed and the accuracy of decision-making. In primate decision-making, simple models of these processes have been shown, under certain parametrizations, to implement the statistically optimal procedure that minimizes decision time for any given error rate. In this paper, we adapt these same analysis techniques and apply them to new models of collective decision-making in social insect colonies. We show that social insect colonies may also be able to achieve statistically optimal collective decision-making in a very similar way to primate brains, via direct competition between evidence-accumulating populations. This optimality result makes testable predictions for how collective decision-making in social insects should be organized. Our approach also represents the first attempt to identify a common theoretical framework for the study of decision-making in diverse biological systems.

**Keywords:** decision-making; diffusion model; optimality; neurons; social insects; sequential probability ratio test



# UNU: The Platform that Gamifies Human Swarm Intelligence

# The Puck of Unu

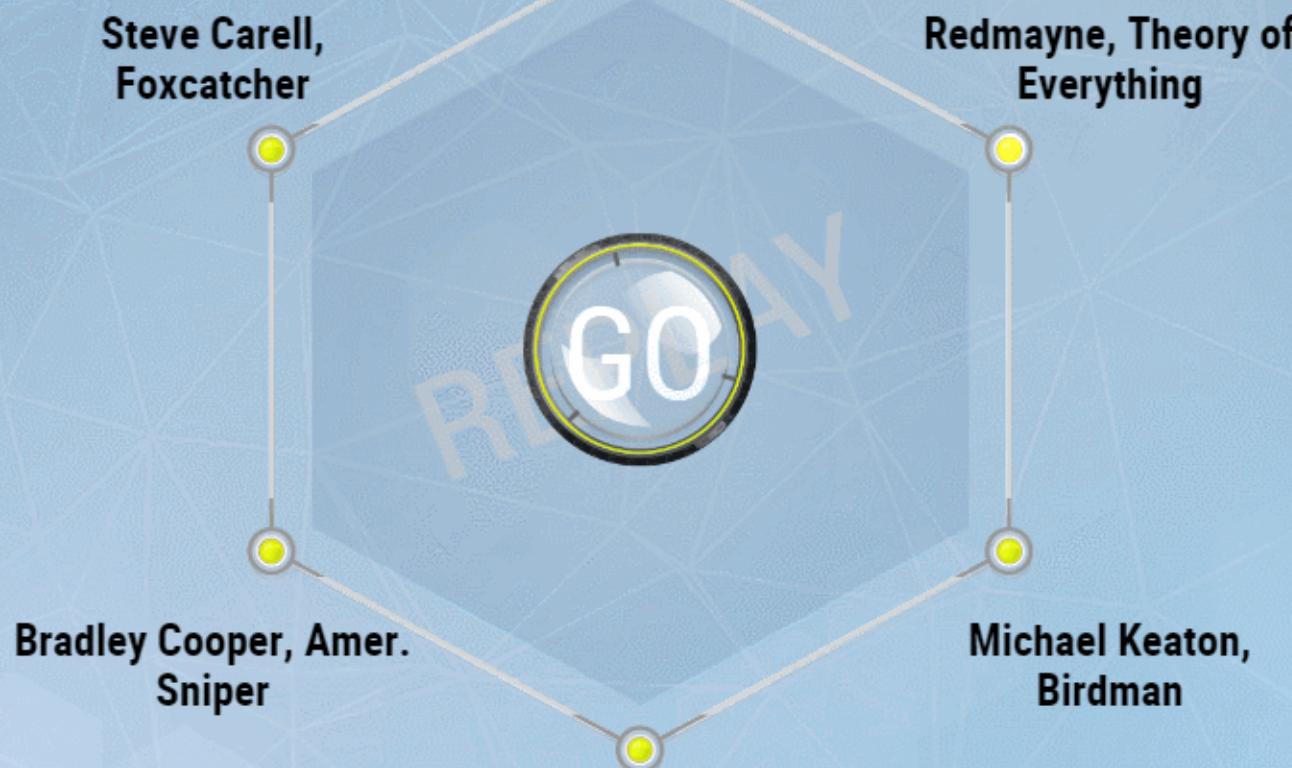


<http://www.megamification.com/unu-the-platform-that-gamifies-human-swarm-intelligence/>

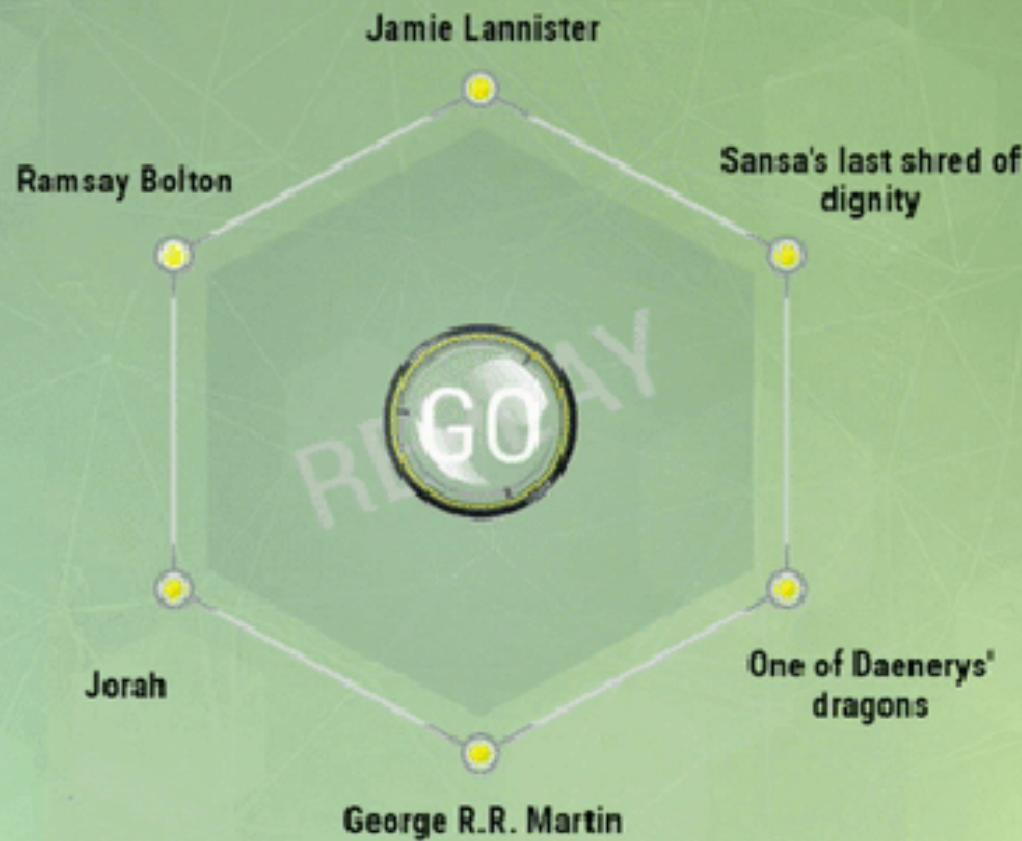


## Who will win Best Actor in a Leading Role?

Cumberbatch, Imitation Game



## Who Will Die Next?



# Marshall et al.

- In ***both brains and social insect colonies***, mutually interacting populations must reach an ***activation threshold to precipitate a decision***.
- We argue that the ***interaction patterns*** between populations are the crucial part of the decision-making process at both these levels of ***biological complexity, organismal and super-organismal***.

# Marshall et al. (2)

- Notwithstanding their impressive individual abilities (Koch 1999), ***neurons are simple*** in comparison with ***individually sophisticated social insects*** (Giurfa et al. 2001; Chittka et al. 2003; Franks et al. 2003b; Franks & Richardson 2006; Richardson et al. 2007).
- ***Simple interaction patterns*** in both these systems, however, may implement ***robust, efficient decision-making*** regardless of how sophisticated their individual components are.

# Swarm principles

- **Principle #1: Awareness**
  - Each member must be aware of its surroundings and abilities.
- **Principle #2: Autonomy**
  - Each member must operate as an autonomous master (not as a slave;) this is essential to self-coordinate allocation of labor.
- **Principle #3: Solidarity**
  - Each member must cooperate in solidarity: when a task is completed, each member should autonomously look for a new task (leveraging its current position.)
- **Principle #4: Expandability**
  - The system must permit expansion where members are dynamically aggregated.
- **Principle #5: Resiliency**
  - The system must be self-healing: when members are removed, the remaining members should undertake the unfinished tasks.

# Take aways on swarms

- Collective intelligence operating in a **synchronous** manner
  - Similar to fuzzy systems based on voting (which is asynchronous) and explicit reasoning
  - Similar to crowd sourcing which is asynchronous
  - Related to cellular automata
  - Shares some features with ensembles
- Each agent adopts a specific role to solve a problem or make a decision for the best of the hive/colony
- Even small swarms can beat “intelligent individuals”
- Social insect systems similar to neuron based systems

# Rodney Brooks: Smartness in a simple way

Artificial Intelligence 47 (1991) 139-159  
Elsevier

139

## Intelligence without representation\*

Rodney A. Brooks  
*MIT Artificial Intelligence Laboratory, 545 Technology Square, Rm. 836, Cambridge,  
MA 02139, USA*

Received September 1987

**Abstract**  
Brooks, R.A., Intelligence without representation, *Artificial Intelligence* 47 (1991) 139-159.  
Artificial intelligence research has founded on the issue of representation. When intelligence is approached in an incremental manner, with strict reliance on interfacing to the real world through perception and action, reliance on representation disappears. In this paper we outline our approach to incrementally building complete intelligent Creatures. The fundamental description of the intelligent system is not into independent information processing modules which must interface to each other through representation. Instead, the intelligent system is decomposed into independent and parallel active producers that all interface directly to the world through perception and action, rather than interface to each other particularly much. The notions of central and peripheral systems evaporate—everything is both central and peripheral. Based on these principles we have built a very successful series of mobile robots which operate without supervision as Creatures in standard office environments.

### 1. Introduction

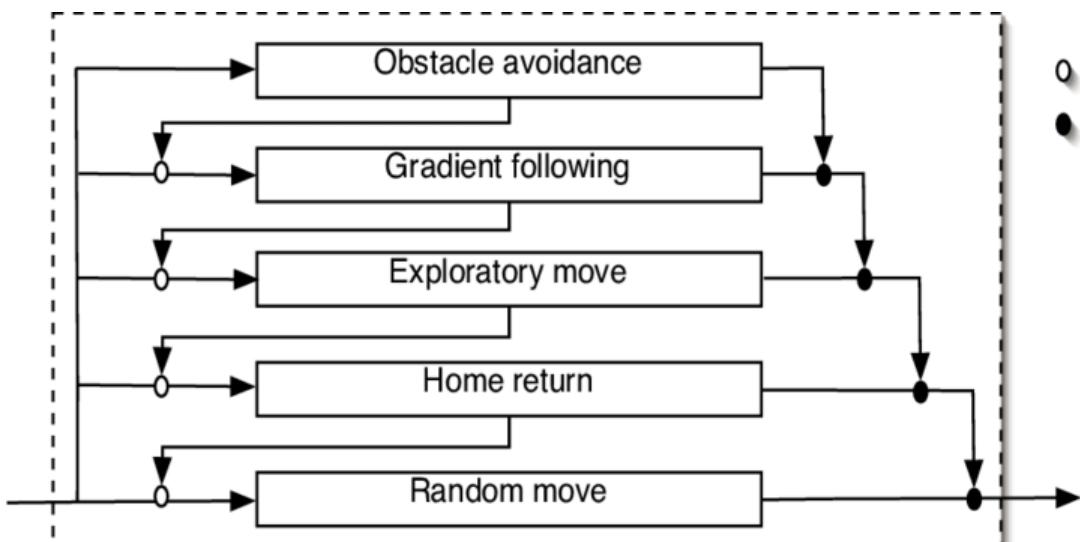
Artificial intelligence started as a field whose goal was to replicate human level intelligence in a machine. Early hopes diminished as the magnitude and difficulty of that goal was appreciated. Slow progress was made over the next 25 years in demonstrating isolated aspects of intelligence. Recent work has tended to concentrate on commercializable aspects of "intelligent assistants" for human workers.

\* This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Some of this work is performed in part by an IBM Faculty Development Award, in part by a grant from the Systems Development Foundation, in part by the University Research Initiative under Office of Naval Research contract N00014-86-K-0065 and in part by the Advanced Research Projects Agency under Office of Naval Research contract N00014-85-K-0124.

0004-3702/91/\$03.50 © 1991 — Elsevier Science Publishers B.V.

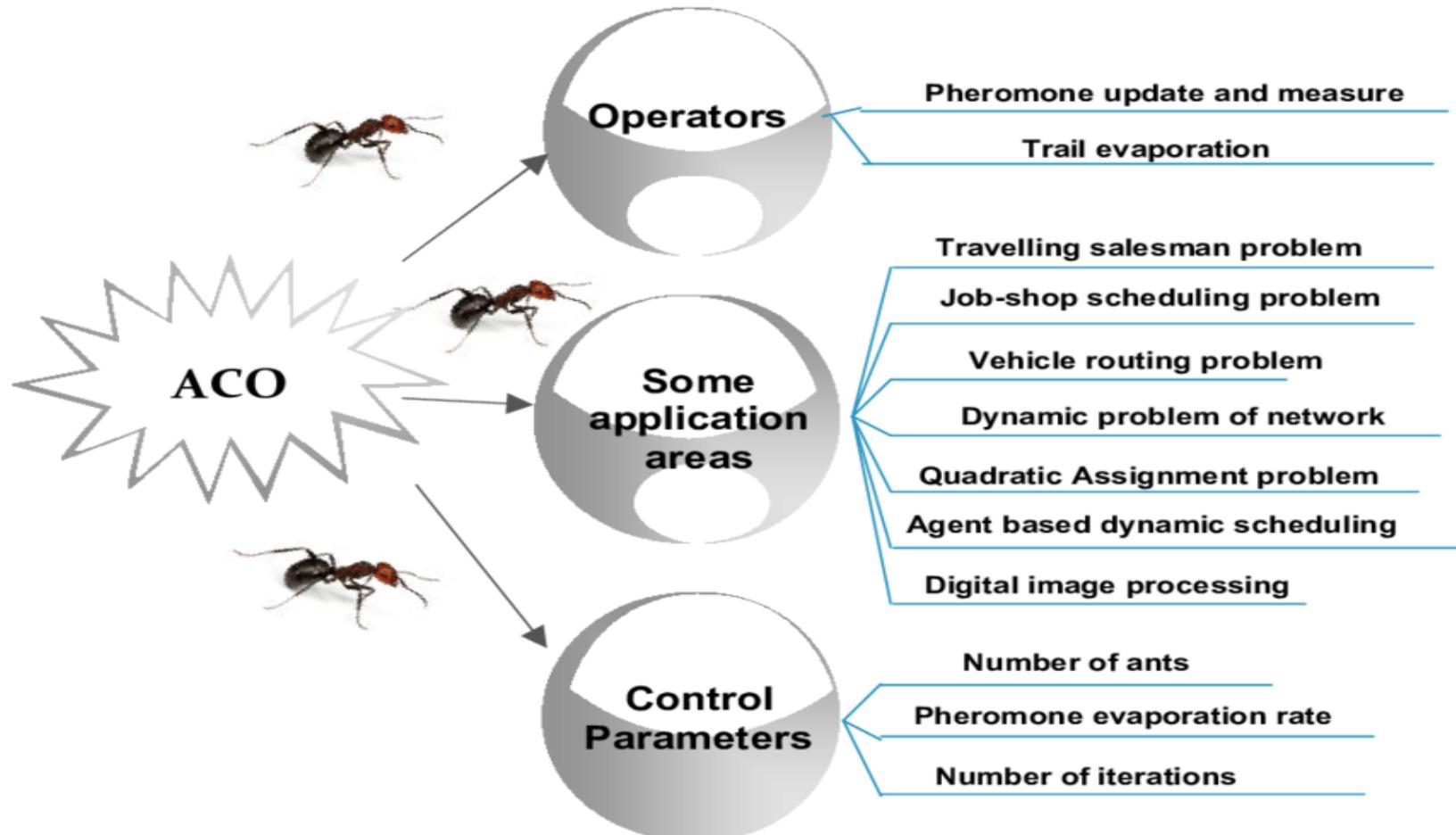


# Take aways: Rodney Brooks' robots



○ replacement  
● inhibition





# Relevant papers

## Don't Go with the Ant Flow: Ant-inspired Traffic Routing in Urban Environments

Jörg Dallmeyer  
Institute of Computer Science  
Goethe University Frankfurt  
P.O. Box 11 19 32  
60054 Frankfurt am Main,  
Germany  
dallmeyer@cs.uni-frankfurt.de

René Schumann  
Institute of Business  
Information Systems  
HES-SO  
Rue de Technopôle 3  
3960 Sierre, Switzerland  
rene.schumann@hevs.ch

Ingo J. Timm  
Computer Science  
Department of Business  
Informatics  
University of Trier  
54286 Trier  
ingo.timm@uni-trier.de

### ABSTRACT

Traffic routing is a well established optimization problem in traffic management. We consider a dynamic traffic routing problem where the load of roads is taken into account dynamically, aiming at the optimization of required travel times. We investigate ant-based algorithms that can handle dynamic routing problems, but suffer from negative emergent effects like road congestions. We propose an *inverse* ant-based routing approach to avoid these negative emergent effects. We evaluate our approach with the agent-based traffic simulation system MAINS<sup>2</sup>IM. For evaluation, we use a synthetic and two real world scenarios. Evaluation results indicate that the proposed inverse ant-based routing can lead to a reduction of travel time.

### 1. INTRODUCTION

Traffic routing is a well established research and optimization problem in traffic management [6]. Most work has been done for static problems, i.e., problems where the problem structure does not change. In static problems the routing decision boils down to find the shortest path between the start and the goal point. Once a solution has been found for all routes the optimal ones can be used whenever needed. These algorithms typically are based on shortest path algorithms, like the well known A\* algorithm.

The situation becomes more complex if we regard dynamic problems. In a dynamic problem, the problem structure changes while solving the problem. For routing decisions this

Andreas D. Lattner  
Institute of Computer Science  
Goethe University Frankfurt  
P.O. Box 11 19 32  
60054 Frankfurt am Main,  
Germany  
lattner@cs.uni-frankfurt.de

## Using Ant Colony Optimization to determine influx of EVs and charging station capacities

Kristoffer Tangrand  
Smart Technology Group, ICT Dept.  
University of Tromsø  
Narvik, Norway  
ktangrand@gmail.com

Bernt A. Bremdal  
Smart Technology Group, ICT Dept.  
University of Tromsø  
Narvik, Norway  
bernt@xalience.com

*Abstract*—This paper presents a novel method for determining peak loads and traffic patterns in the future by calculating needs based on an ACO (Ant Colony Optimization) method. The method is used to analyze traffic patterns and to determine their impact on the local grid and the design of charging stations. The research reported here also supports the design of a portfolio of Charging Stations (CS) and the general usage areas and uses this to determine the required capacity of each station. An empirical basis for the research presented has been gathered from Norway where the number of EVs are growing fast and where use of EVs for different purposes, including long-range driving, is increasing rapidly. The empirical data gathered also shows how demand for charging at different times can be determined. This lays the foundation for estimating peak loads in the local grid due to EV charging. For the individual driver the system presented can be used to find preferred routing under different circumstances such as traffic congestion.

*Index Terms*—Ant Colony Optimization, Charging Stations, *v* *tarolden*, *Vokteridet*

### II. INTRODUCTION

Understanding the traffic pattern under various conditions and to determine the potential recharging needs of plug-in vehicles (EV) in the future is essential in the Flex-CHEV project. This can help to determine where CSs should be located and the potential loads that the recharging needs could impose on the CS itself and the local grid. For an off-grid facility such loads should be absorbed by an ESS, and the magnitude of these loads would therefore be essential for its design. The approach documented here is based on Ant Colony Optimization (ACO). The aim of the paper is thus to highlight the validity of the method applied for the purpose presented above and to compare it with other, more traditional approaches, like Dynamic Programming (DP). The method developed can cater for a number of dynamic and transient aspects such as traffic congestion, seasonal changes and road

# Markets and ZI-agents

**Allocative Efficiency of Markets with Zero-Intelligence Traders: Market as a Partial Substitute for Individual Rationality**

Dhananjay K. Gode; Shyam Sunder

*The Journal of Political Economy*, Vol. 101, No. 1. (Feb., 1993), pp. 119-137.

Stable URL:  
<http://links.jstor.org/sici?&sici=0022-3808%28199302%29101%3A1%3C119%3AAFOMWZ%3E2.0.CO%3B2-3>

*The Journal of Political Economy* is currently published by The University of Chicago Press.

---

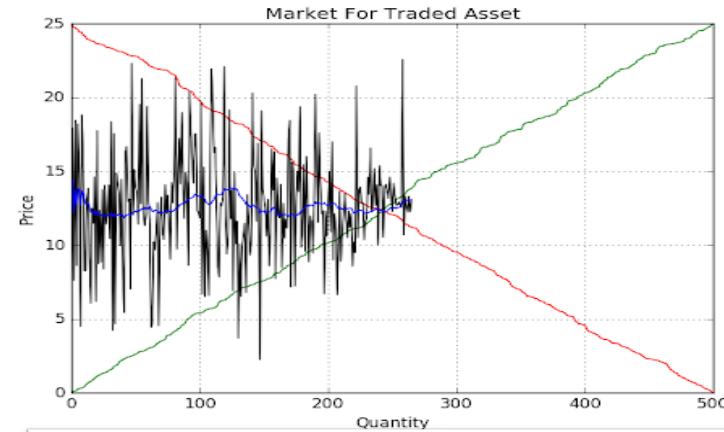
Your use of the JSTOR archive indicates your acceptance of JSTOR's Terms and Conditions of Use, available at <http://www.jstor.org/about/terms.html>. JSTOR's Terms and Conditions of Use provides, in part, that unless you have obtained prior permission, you may not download an entire issue of a journal or multiple copies of articles, and you may use content in the JSTOR archive only for your personal, non-commercial use.

Please contact the publisher regarding any further use of this work. Publisher contact information may be obtained at <http://www.jstor.org/journals/ucpress.html>.

Each copy of any part of a JSTOR transmission must contain the same copyright notice that appears on the screen or printed page of such transmission.

---

JSTOR is an independent not-for-profit organization dedicated to and preserving a digital archive of scholarly journals. For more information regarding JSTOR, please contact support@jstor.org.



## Zero is Not Enough: On The Lower Limit of Agent Intelligence for Continuous Double Auction Markets\*

Dave Cliff<sup>1</sup> and Janet Bruton<sup>2</sup>

<sup>1</sup>Artificial Intelligence Laboratory  
Massachusetts Institute of Technology  
545 Technology Square  
Cambridge MA 02139, U.S.A.  
[davec@ai.mit.edu](mailto:davec@ai.mit.edu)  
Phone: +1 617 253 6625  
Fax: +1 617 253 5060

<sup>2</sup>Hewlett-Packard Laboratories Bristol  
Filton Road  
BRISTOL BS12 6QZ, U.K.  
[jlb@hplb.hpl.hp.com](mailto:jlb@hplb.hpl.hp.com)

### Abstract

Gode and Sunder's (1993) results from using "zero-intelligence" (zi) traders, that act randomly within a structured market, appear to imply that convergence to the theoretical equilibrium price in continuous double-auction markets is determined more by market structure than by the intelligence of the traders in that market. However, it is demonstrated here that the average transaction prices of zi traders can vary significantly from the theoretical equilibrium value when the market supply and demand are asymmetric, and that the degree of difference from equilibrium is predictable from *a priori* probabilistic analysis. In this sense, it is shown here that Gode and Sunder's results are artefacts of their experimental regime. Following this, 'zero-intelligence-plus' (zip) traders are introduced: like zi traders, these simple agents make stochastic bids. Unlike zi traders, these traders are elementary enough

# Humans as “ants”

## NEWSLETTERS

Sign up to read our regular email newsletters

# NewScientist

HALF PRICE FOR 12 WEEKS

[News](#) [Podcasts](#) [Video](#) [Technology](#) [Space](#) [Physics](#) [Health](#) [More](#) [Shop](#) [Courses](#) [Events](#) [Tours](#) [Jobs](#)  [Sign In](#)  [Search](#)

## 'Zero intelligence' trading closely mimics stock market



LIFE 1 February 2005

By [Katharine Davis](#)

A model that assumes stock market traders have zero intelligence has been found to mimic the behaviour of the London Stock Exchange very closely.

However, the surprising result does not mean traders are actually just buying and selling at random, say researchers. Instead, it suggests that the movement of markets depend less on the strategic behaviour of traders and more on the structure and constraints of the trading system itself.

The research, led by J Doyne Farmer and his colleagues at the Santa Fe Institute, New Mexico, US, say the finding could be used to identify ways to lower volatility in the stock markets and reduce transaction costs, both of which would benefit small investors and perhaps bigger investors too.

A spokesperson for the London Stock Exchange says: “It’s an interesting bit of work that mirrors things we’re looking at ourselves.”

marketsoI  
part of markets.com

Invest now

Smart Innovation Norway, UiT and Skagerak Energy

# The Skagerak Market in E-Regio

**Sustainable Planet**

## The E-Regio project: for a distributed local energy market

30th June 2020

**FEATURED TOPICS**

- GREEN ENERGY
- AGRICULTURE
- AQUACULTURE
- POLLUTION
- BIODIVERSITY

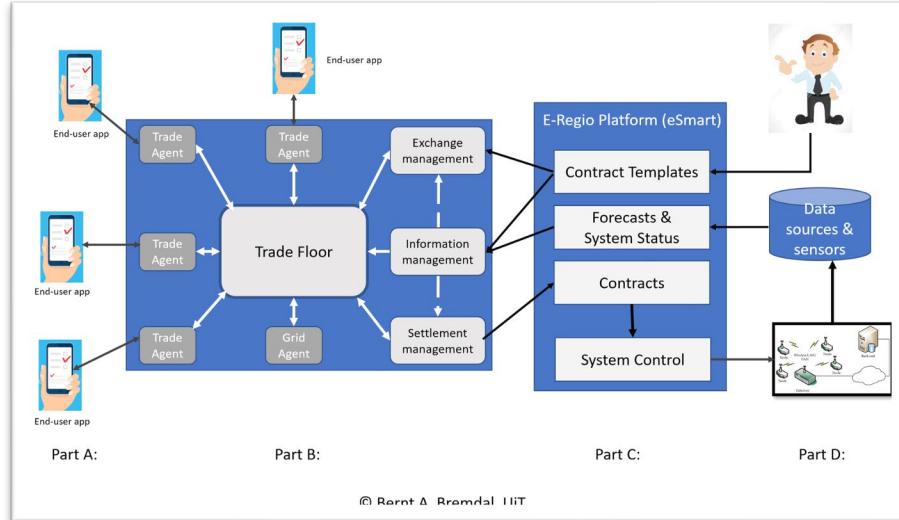
**THE E-REGIO PROJECT**

**With a focus on pure energy trade, the E-Regio project investigates different local energy market concepts for energy storage, whilst also conducting research on local flexibility trade.**

At E-Regio, we investigate different local market concepts for energy. This is primarily for the pure energy trade, but local flexibility trade is also subject to research. An overview of local energy markets and associated literature can be found in [Sumper, 2019](#). In particular, the concepts described by Breindal and Ileva (2019) have been important for the type of work that has been conducted in E-Regio.2 Multiple models are reviewed and discussed regarding

**LATEST EBOOKS**

- Innovation in batteries to support the renewable energy industry
- Cow monitoring technology: making dairy farming easier and more profitable



# E-REGIO

Spot Price (Nordpool) [NOK/kWh]

1

Energy-Tariff (NOK/kWh)

0.60

Power-Tariff (NOK/kW)

0.00

Start Trade

Start Trade

From Date

11/06/2020

June 2020

Content

Trade starts at 2020-06-12T08:05:05.124763  
 A partial settlement: CommunityManager sells 78.4707 to Consumption 1

A partial settlement: CommunityManager sells 0 to Battery

Consumption 2 bids up 0.46

PV asks lower 0.62

Consumption 2 bids up 0.58

PV asks lower 0.42

PV asks lower 0.22

A full settlement: PV sells 78.4707 to CommunityManager

A partial settlement: PV sells 0 to Consumption 1

A partial settlement: PV sells 0 to Battery

A partial settlement: PV sells 0 to Consumption 2

\*\*\* SETTLEMENTS 2020-06-12T08:05:05.12581\*\*\*  
 CommunityManager sells volume 78.4707 to Consumption 1 for price 0.6  
 CommunityManager sells volume 0 to Battery for price 0.6

PV sells volume 78.4707 to CommunityManager for price 0.33

PV sells volume 0 to Consumption 1 for price 0.6

PV sells volume 0 to Battery for price 0.6

PV sells volume 0 to Consumption 2 for price 0.6

PV sells volume 0 to Consumption 3 for price 0.6

PV sells volume 0 to Consumption 4 for price 0.6

PV sells volume 0 to Consumption 5 for price 0.6

PV sells volume 0 to Consumption 6 for price 0.6

PV sells volume 0 to Consumption 7 for price 0.6

PV sells volume 0 to Consumption 8 for price 0.6

PV sells volume 0 to Consumption 9 for price 0.6

PV sells volume 0 to Consumption 10 for price 0.6

PV sells volume 0 to Consumption 11 for price 0.6

PV sells volume 0 to Consumption 12 for price 0.6

PV sells volume 0 to Consumption 13 for price 0.6

PV sells volume 0 to Consumption 14 for price 0.6

PV sells volume 0 to Consumption 15 for price 0.6

PV sells volume 0 to Consumption 16 for price 0.6

PV sells volume 0 to Consumption 17 for price 0.6

PV sells volume 0 to Consumption 18 for price 0.6

PV sells volume 0 to Consumption 19 for price 0.6

PV sells volume 0 to Consumption 20 for price 0.6

PV sells volume 0 to Consumption 21 for price 0.6

PV sells volume 0 to Consumption 22 for price 0.6

PV sells volume 0 to Consumption 23 for price 0.6

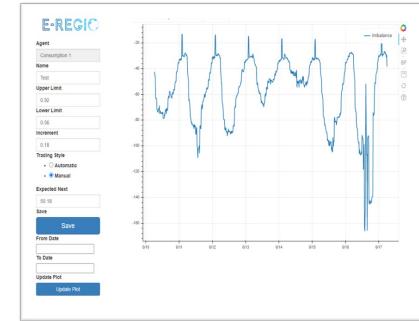
PV sells volume 0 to Consumption 24 for price 0.6

PV sells volume 0 to Consumption 25 for price 0.6

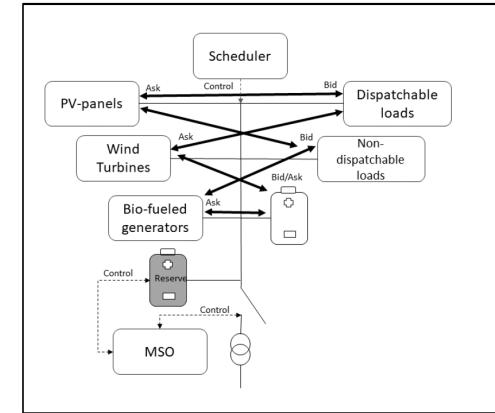
PV sells volume 0 to Consumption 26 for price 0.6

PV sells volume 0 to Consumption 27 for price 0.6

Trade floor



Agents' performances



Peer-to-peer concept w/ self-interested agents

User specifies trade limits – the agent does the rest

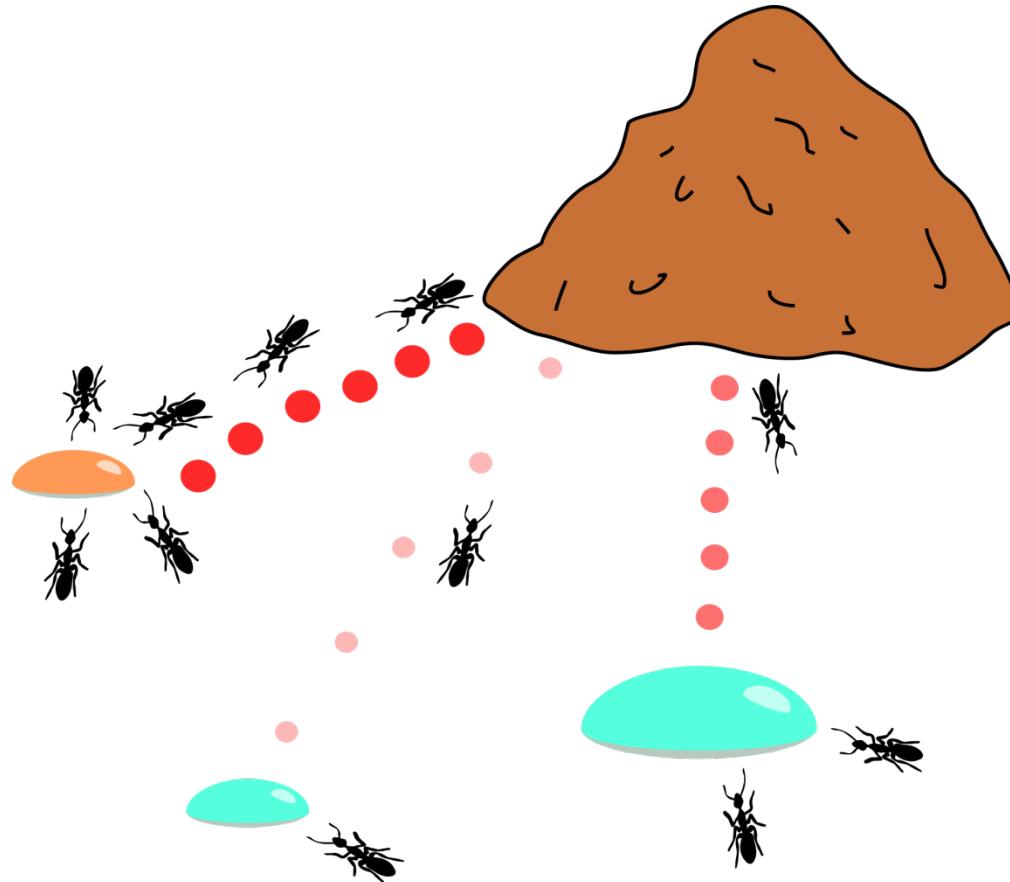
# Ants as agents

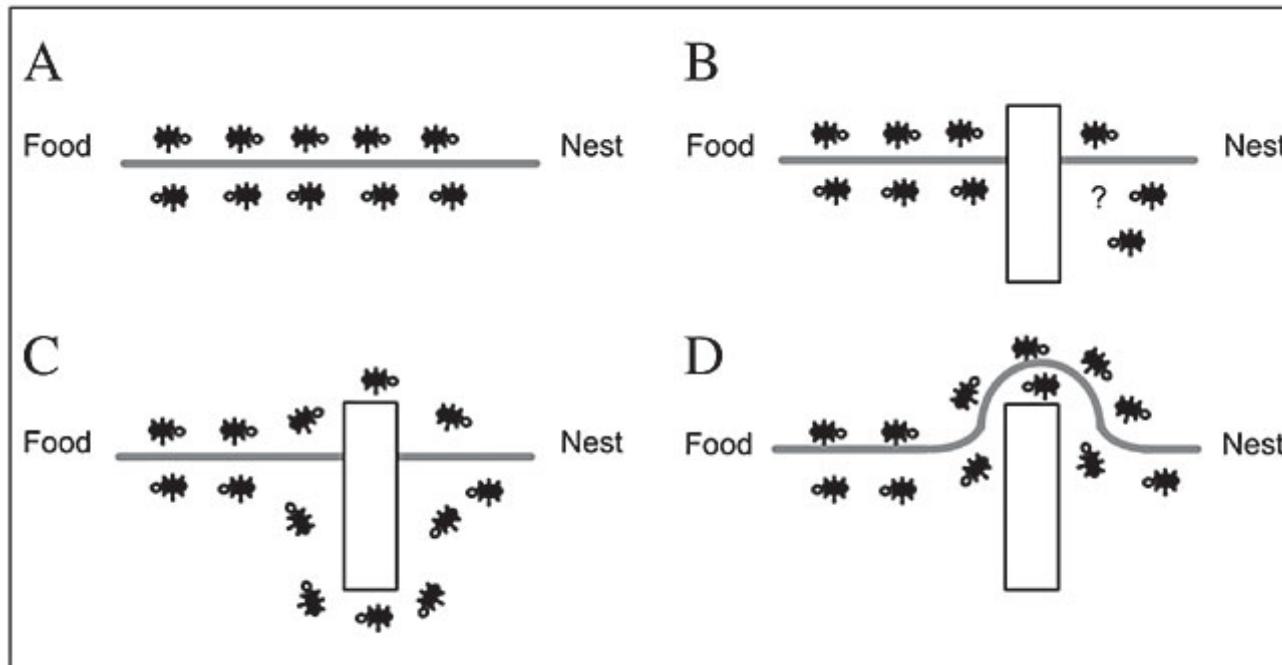
Bernt A. Bremdal

UiT 2019

On collective intelligence & memory

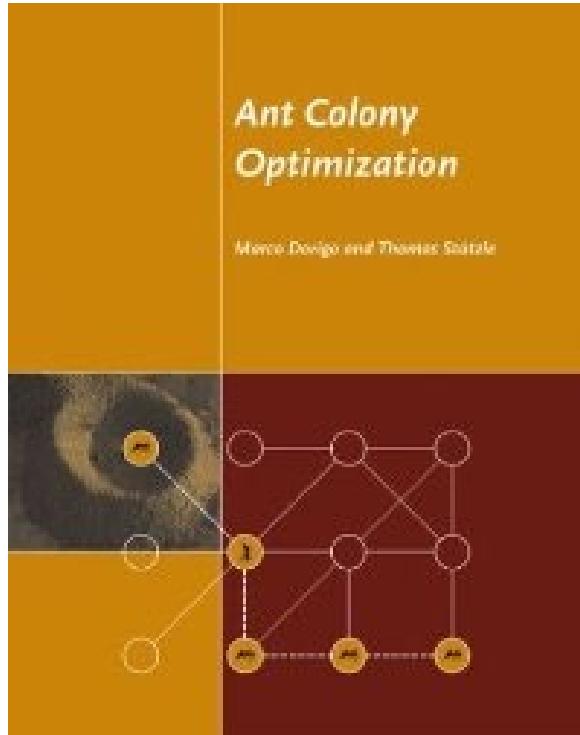
# Ant Colony Optimization





**Figure 2.** A. Ants in a pheromone trail between nest and food; B. an obstacle interrupts the trail; C. ants find two paths to go around the obstacle; D. a new pheromone trail is formed along the shorter path.

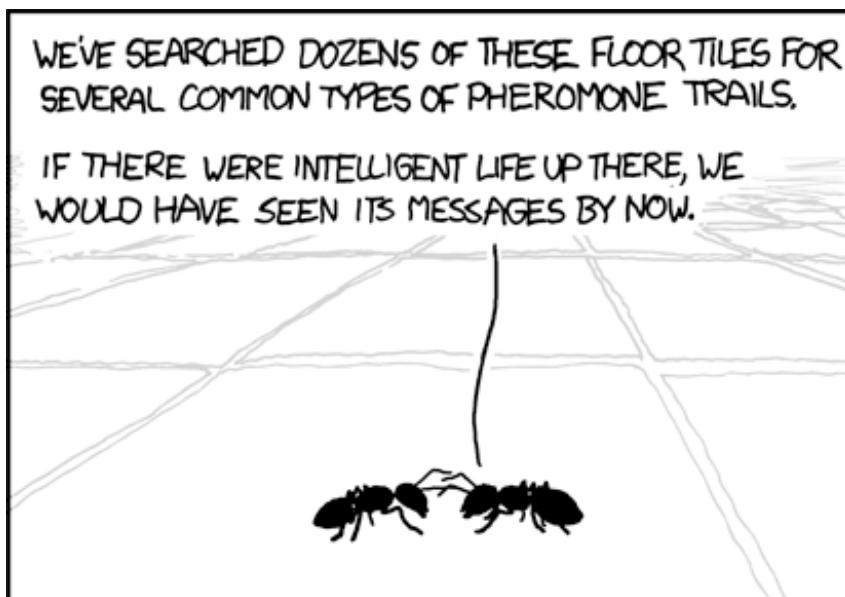
# Ant colony intelligence



Ant Colony Optimization (Ch. 12)  
Invented by Marco Dorigo in 1991

- An ant is a type of agent that solves a problem iteratively together with other ants
- Ants collaborate
- Partial solutions in the problem solving process are considered states
- Cyclic process
- Each ant moves from one state to another

# Basic concepts



THE WORLD'S FIRST ANT COLONY TO ACHIEVE SENTIENCE CALLS OFF THE SEARCH FOR US.

- Reward
- Attraction
- Profitability per path
- Pheromone and pheromone level
- Pheromone decay

# Basic behavior of ants



The ant workers are always on the road for food.



If food resources are depleted new ones must be found

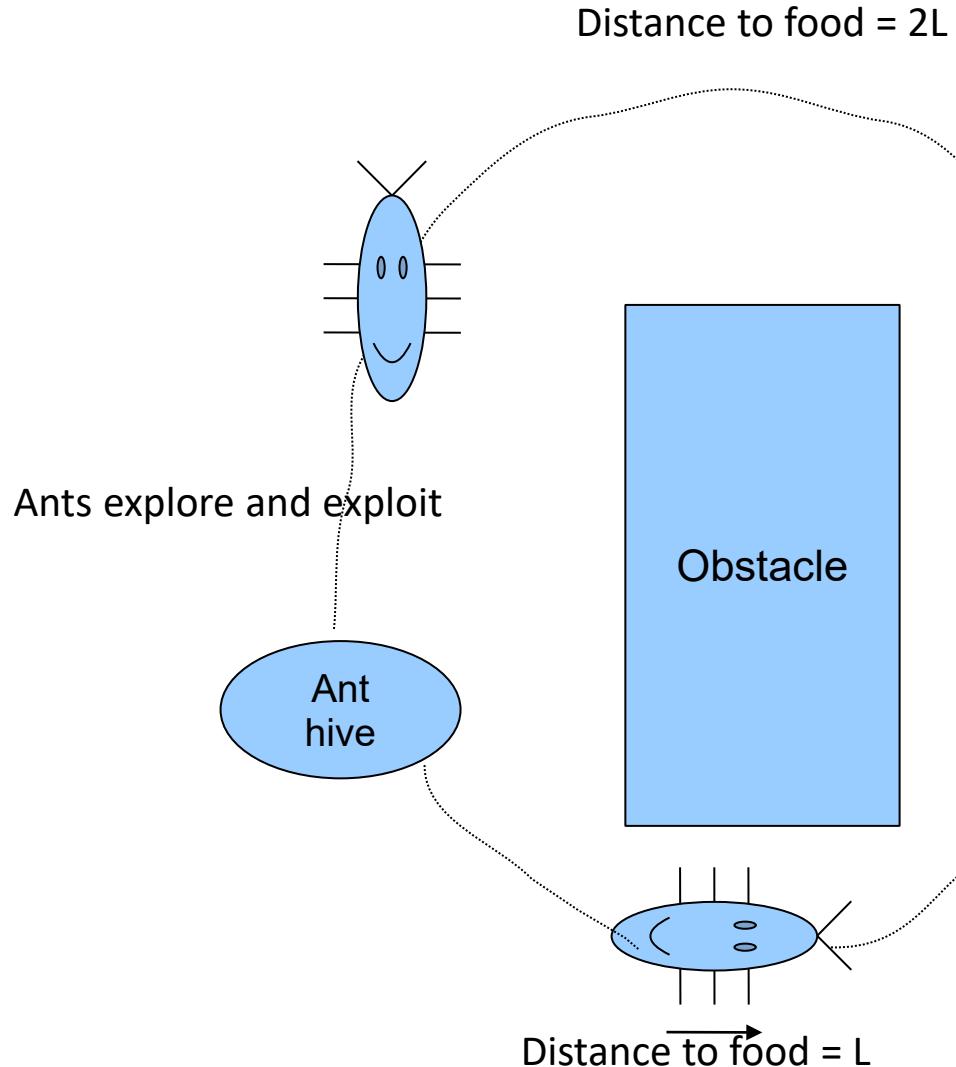


Ants shift between exploration and exploitation



If we assumed that two ants where in search for new sources of nourishment the situation that follows could occur.....

# Basic principle of work



Ants explore and exploit

Distance to food =  $2L$



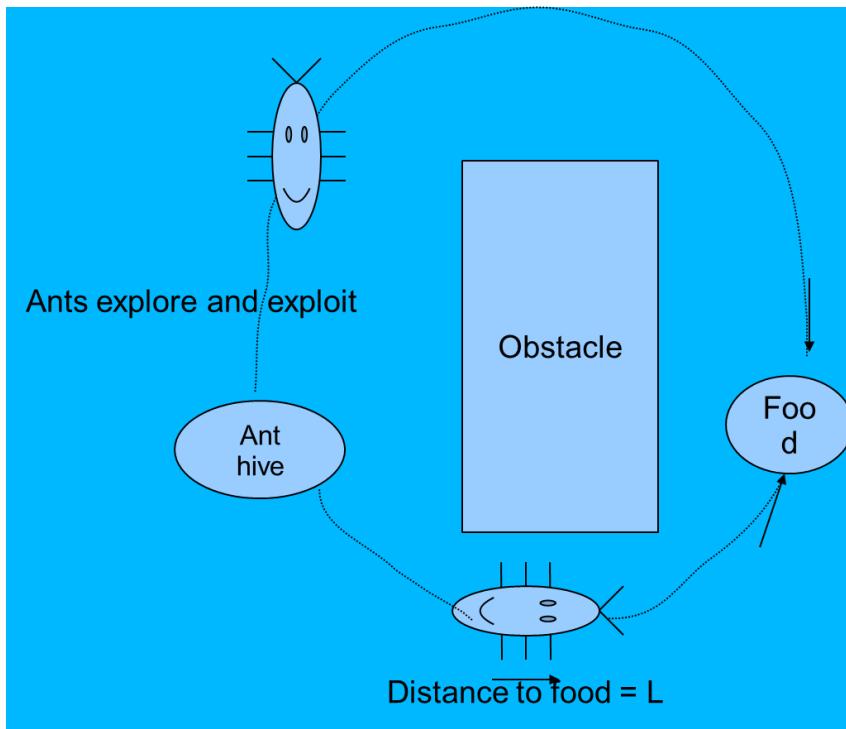
The ant that selects the shortest route will be able to move back and forth during the same time as the other spends his way to reach the food.

The pheromone level will increase more rapidly along the shorter route.

Actually twice that of the longer route

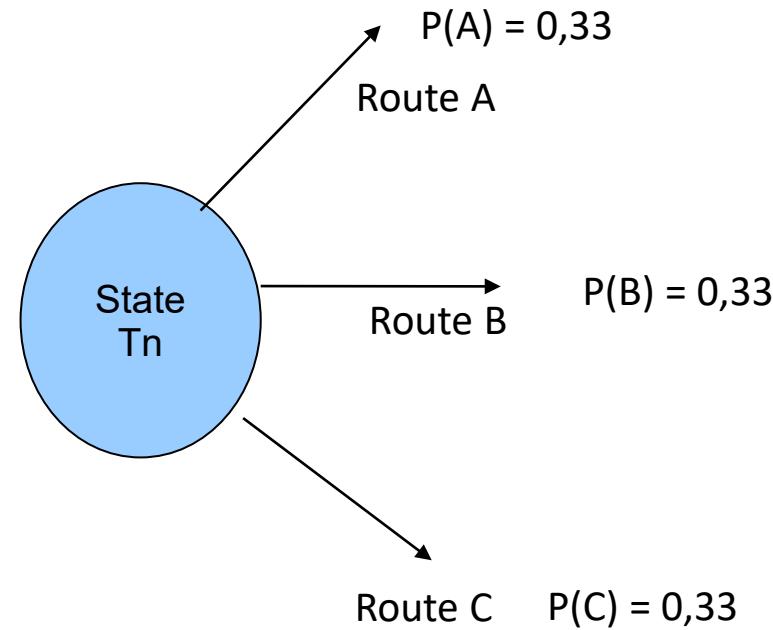
Distance  $\xrightarrow{ } L$

# We can describe this in terms of states ( $T_i$ )



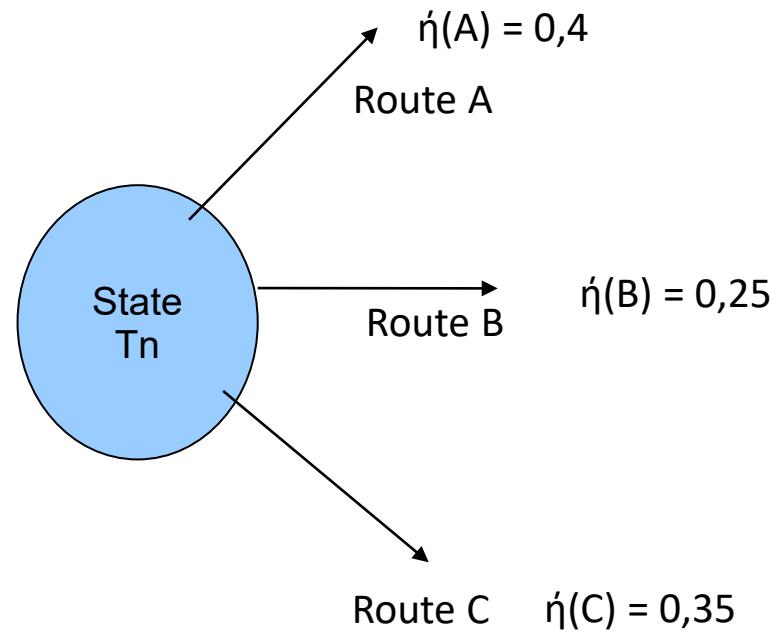
- Situation at T1: Ant 1 reaches the new source of forage or other food. It leaves a pheromone track. Ant 2 is only half way there yet.
- Situation T2: Ant 1 has returned to the hive with its load of food. Ant 2 has reached the new food source.
- Situation T3: Ant 1 is back at the food spot. The level of pheromone deposited along the track increases accordingly. If Ant 2 chooses his original track back home it has half the distance to go still.
- Situation T4: Ant 1 is once more at the hive. This time he meets up with Ant 2 who will arrive at the same time
- Etc.
- Pheromone attracts other ants. When new ones comes around which path are they likely to follow?

Assume the ants had no knowledge or information  
– a pure probabilistic model – which path will it choose?



Perfect entropy! Any direction is just as good

## A probabilistic model reflects pure exploration



$\hat{\eta}(i)$  is here a «weighted probability» reflecting the cost of a path.  
The attractiveness of a path can be established «a priori».

# How experience influences choice



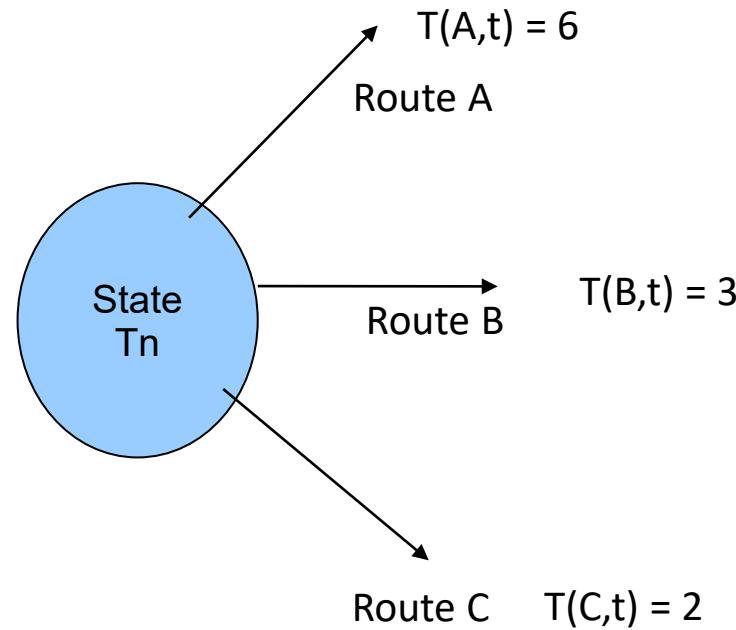
Pheromone level provides information which the ants senses



The change of pheromone level reflects the «degree of exploitation»

It makes salient the collective experience of the ants

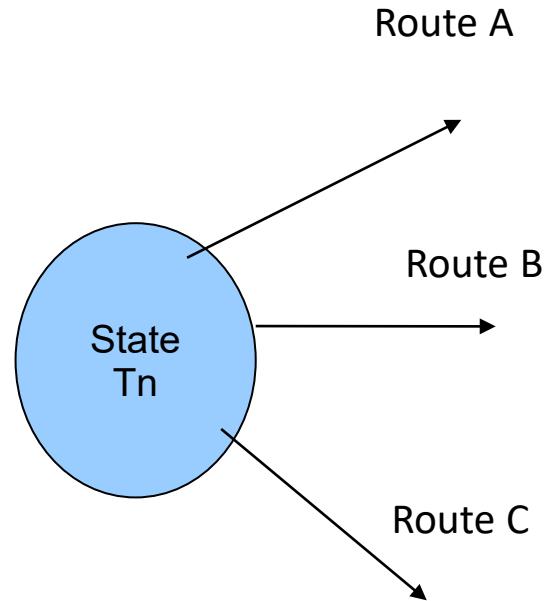
Assume that pheromone deposits can be expressed as a at a given time t by means of the level  $T(i,t)$



$T(i,t)$  reflects the collective experience at t. The relative level of any path stemming from  $T_n$  can therefore also be expressed as a probability.

$$P'(A,t) = T(A,t) / (T(A,t) + T(A,t) + T(A,t))$$

# Experience thus have an impact on choice



Which path is the ant likely to choose?

$$P(r, u) = \frac{T(r, u)^\alpha * n(r, u)^\beta}{\sum_k T(r, k)^\alpha * n(r, k)^\beta}$$

$$P(r, u) = \frac{T(r, u)^\alpha * n(r, u)^\beta}{\sum_k T(r, k)^\alpha * n(r, k)^\beta}$$

r,u is the path between node r og u in a graph or matrix  
r,k are any path from r

P(r,u) is the probability that the path r-u will be picked

T(r-u) = intensity of pheromone along the path r-u

T is a heuristic function expressed  $T = Q * 1/D$  where D is the distance between r og u.  
Q is a scalar.

Alpha,  $\alpha$  is the relative influence of pheromone, a user defined number between 0 and 1  
Beta,  $\beta$ , is the relative influence of heuristics, a number between 0 and 1.

$\eta$  is the attraction level of the path r-u

K is the number of paths that fork out from the present state or node that the ant is

$$P(r, u) = \frac{T(r, u)^\alpha * n(r, u)^\beta}{\sum_k T(r, k)^\alpha * n(r, k)^\beta}$$

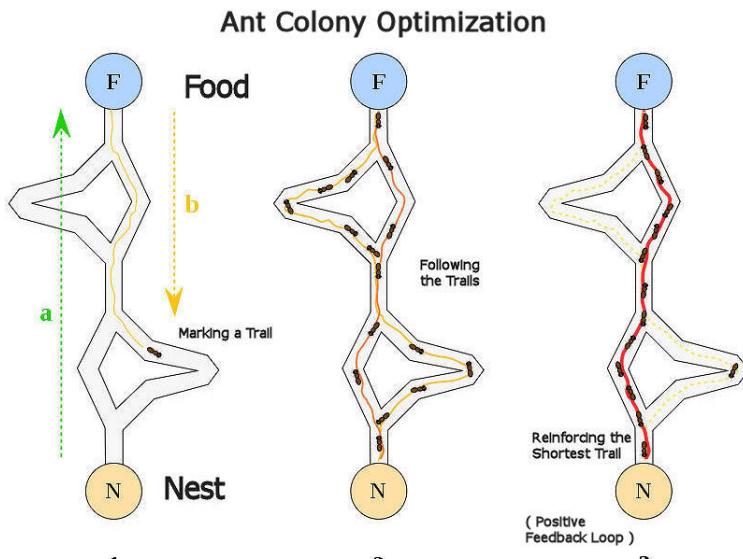
Consequently we are working with weighted probabilities.  
The likelihood of choosing the path is a weighted function of its attractiveness divided by the sum of all the paths weighted in terms of their attractiveness

The full expression is a function of the pheromone level and some heuristics related to i.e. length, cost, risk, rewards etc.  
(just like with informed search such A\*)

# Note

- Exploitation deals with the pheromone level.
- Exploration is often based on the heuristic about distance (or cost/reward like for regular search problems).
  - If we look away from the exploitation part and focus on the distance, the attractiveness of a path is related to how short the distance is.
- The probability of a shorter path being picked is the weighted probability of that choice compared to the all the others.
  - Consequently attractiveness and thus the weight that we use must be the inverse of the distance, not the distance itself. Many use the distance. This favors the longest distance. In most cases this would be wrong. If we were addressing rewards (not distance or cost) then we would not use the inverse. I might not have stressed this well enough in class.
- Attractiveness must be the inverse of the distance,  $n= 1/d$ .

# The ants will always find the path that yields the best cost-benefit ratio



[http://en.wikipedia.org/wiki/Ant\\_colony\\_optimization](http://en.wikipedia.org/wiki/Ant_colony_optimization)

- The tension between exploration and exploitation
- The use of pheromone allows the ants to communicate
- More ants on the same trail → more pheromone deposited → the more ants will be attracted
- Decay means that pheromone level decreases
  - Once popular trails that does not produce rewards will loose its p-level and be subdued
- ACO captures dynamics well

# The trip that an ant makes

- A trip that an ant makes is complete if an ant has been through the whole set of nodes in the graph
- But – only one visit per node per trip (The Hamilton Path)
- When the trip is complete the full distance can be calculated
- This also allows the total amount of pheromone to be calculated



# Learning, sharing and optimizing

$$T_{ij}(t) = T_{i,j}(t - 1) + \Delta T_{i,j}(k)^* \rho$$

$\Delta T_{i,j}$  is the pheromone deposited by one ant  $k$  on its trip

This is added to the existing level of pheromone that is already present along the paths of the trip,

$$T_{i,j}(t - 1).$$

*i-j is the path from i to j.*

*$\rho$  is a heuristic factor.*

# The amount of pheromone deposited per trip

$$\Delta T = Q * 1/D$$

D = distance

Q = is the pheromone deposition capability

# Decay reduces the pheromone level on a path

$$T_{ij}(t) = T_{i,j}(t - 1) (1 - \rho)$$

$\rho$  is the heuristic factor used for pheromone deposition

# Example 1 : Simple probability

---

The chance of taking choice 2 might be calculated like this:

$$\frac{\tau_2}{\sum \tau_{t_i}} = \frac{0.7}{0.2 + 0.7 + 0.1 + 0.6 + 0.8}$$

Choice	Variable Name	Pheromone Value	Chance of Chosing
1	$t_1$	0.2	0.083
2	$t_2$	0.7	0.291
3	$t_3$	0.1	0.041
4	$t_4$	0.6	0.25
5	$t_5$	0.8	0.333

## Example 2: Now we add more weight to the pheromone

- We can increase the differences by raising the pheromones value to a power  $\alpha$ :

$$\frac{\tau_2^\alpha}{\sum \tau_{t_i}^\alpha} = \frac{0.7^\alpha}{0.2^\alpha + 0.7^\alpha + 0.1^\alpha + 0.6^\alpha + 0.8^\alpha}$$

Choice	Variable Name	Pheromone Value	$t_i^{\alpha=2}$	Chance of Chosing
1	$t_1$	0.2	0.04	0.026
2	$t_2$	0.7	0.49	0.318
3	$t_3$	0.1	0.01	0.006
4	$t_4$	0.6	0.36	0.233
5	$t_5$	0.8	0.64	0.415

## Example 3: Combining it with heuristics

	Route alternative	Variable name	Pheromone value	Variable name	Heuristic value	$T^\alpha$	$n^\beta$	Product	Probability of choice	%
$\alpha$	1	T1	0,2	n1	0,5	0,04	0,70710678	0,02828427	0,03464737	3,4
2	2	T2	0,7	n2	0,6	0,49	0,77459667	0,37955237	0,46494011	46,4
$\beta$	3	T3	0,1	n3	0,8	0,01	0,89442719	0,00894427	0,01095646	1
0,5	4	T4	0,6	n4	0,3	0,36	0,54772256	0,19718012	0,24153965	24,1
	5	T5	0,8	n5	0,1	0,64	0,31622777	0,20238577	0,24791641	24,7
						SUM		0,8163468	1	100

$$P(r, u) = \frac{T(r, u)^\alpha * n(r, u)^\beta}{\sum_k T(r, k)^\alpha * n(r, k)^\beta}$$

*Heuristics example: admissible estimate of relative reward (or 1/cost) of path*

## Excellent for route planning and logistics

Can model human behavior and include

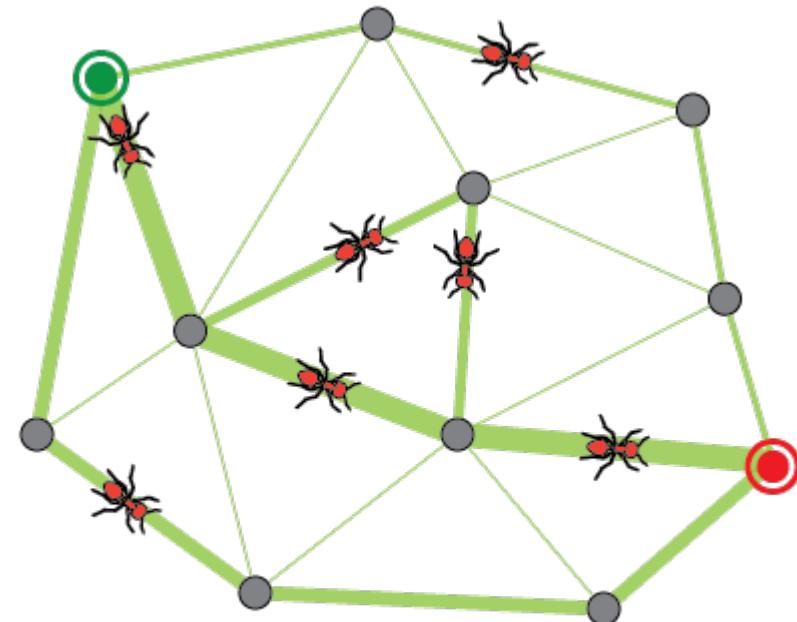
- Panic
- Rational choice
- Sentiments

Can model obstacles

- Permanent
- Temporary

Can model rewards

- Permanent
- Temporary



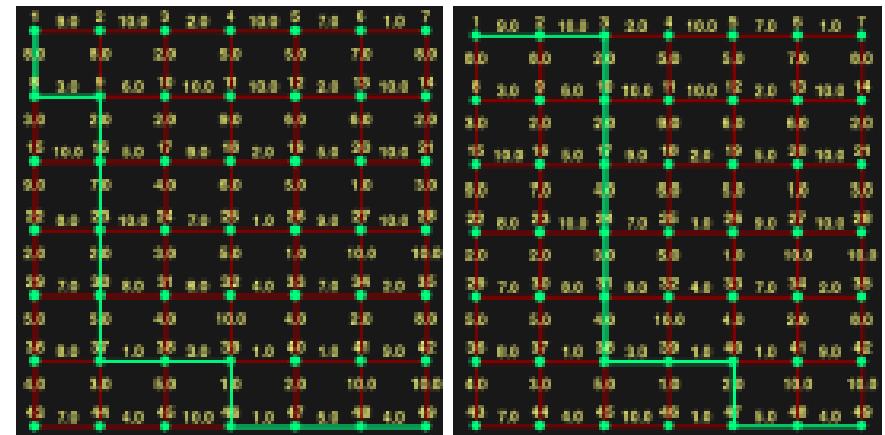
# Collective and individual planning



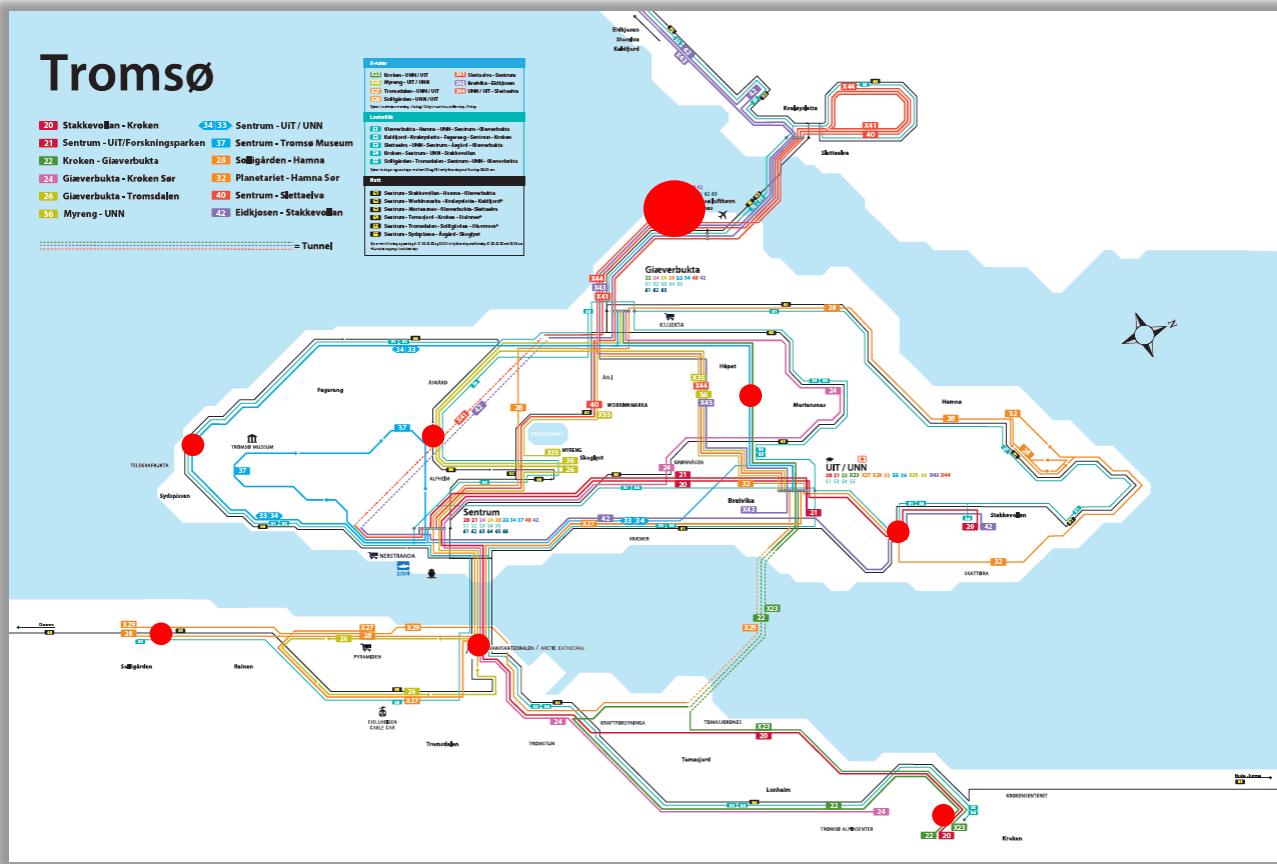
Using Ant Colony Optimization to determine influx of EVs and charging station capacities

Kristoffer Tangrand  
Smart Technology Group, ICT Dept.  
Narvik University College  
Narvik, Norway  
ktangrand@gmail.com

Bernt A. Bremdal  
Smart Technology Group, ICT Dept.  
Narvik University College  
Narvik, Norway  
NCE Smart Energy Markets  
bernt@xalience.com



(a) Optimal solution to the 7x7 graph  $d_{ij}$  shown on arcs (b) The optimal route for inhomogeneous nodes





# Genetic algorithms

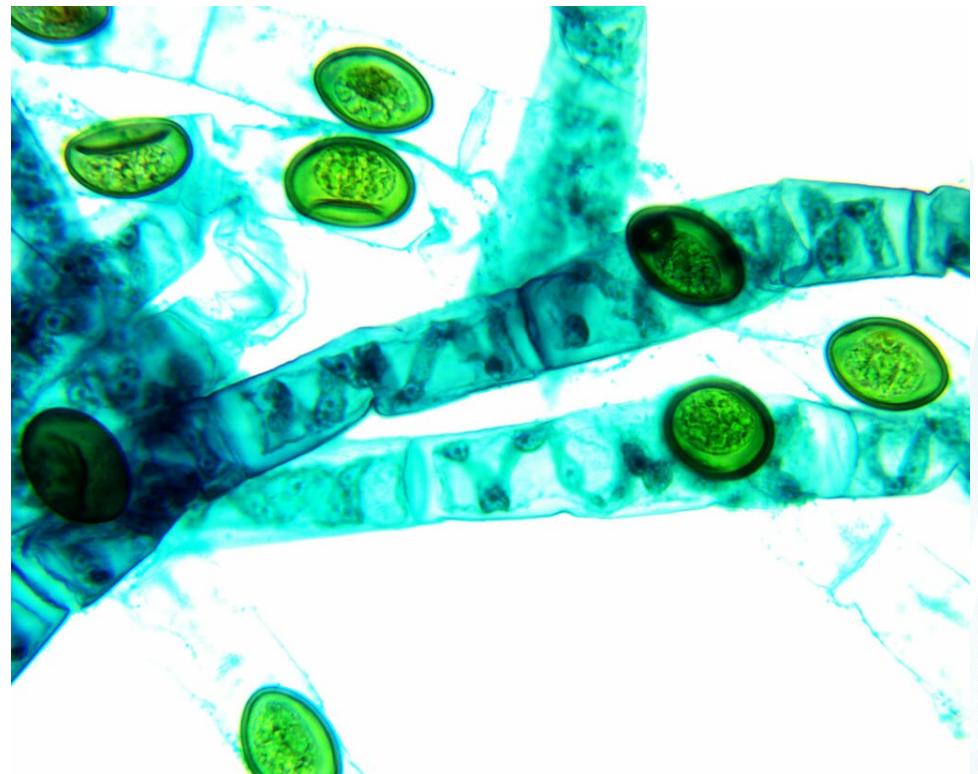
*An optimization algorithm modelled after Darwin's evolutionary theory*

*Lecture 1/3 - Introduction*

Andreas Dyrøy Jansson

# Background

- First described by John Holland (1964)
- Based on evolutionary theory
  - Key concepts include:
    - Representation
      - Chromosomes
      - Genes
    - Crossover
    - Selection
    - Fitness
    - Mutations



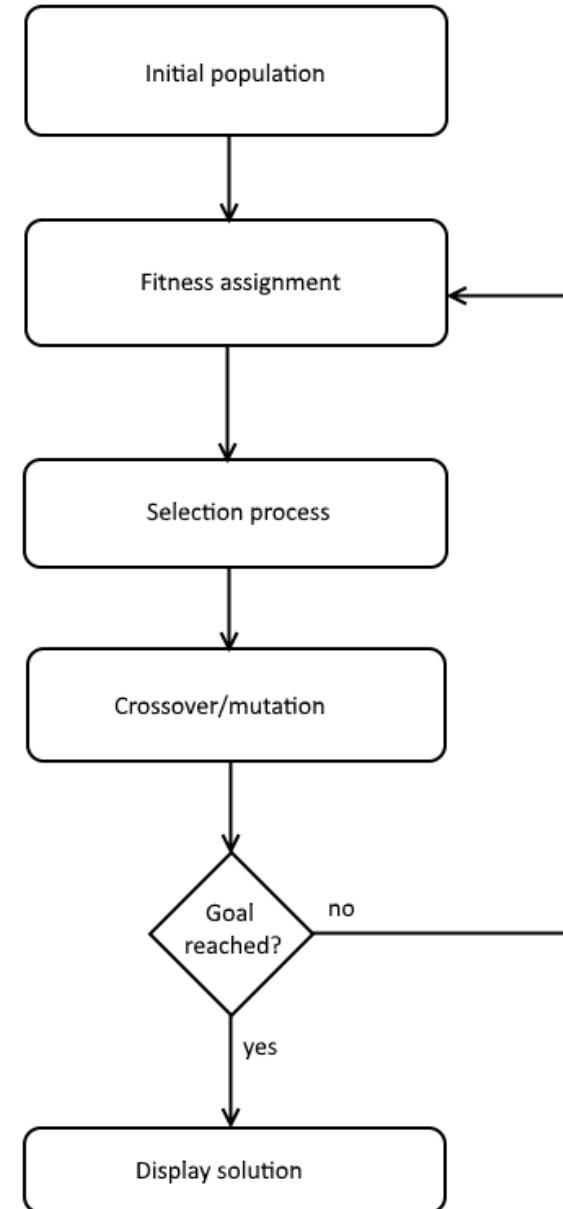
# Application areas for genetic algorithms

- Multi-dimensional search
  - Multiple parameters
- Optimization problems
  - Design of mechanical parts
  - Visual design
- Pattern recognition
- Text analysis



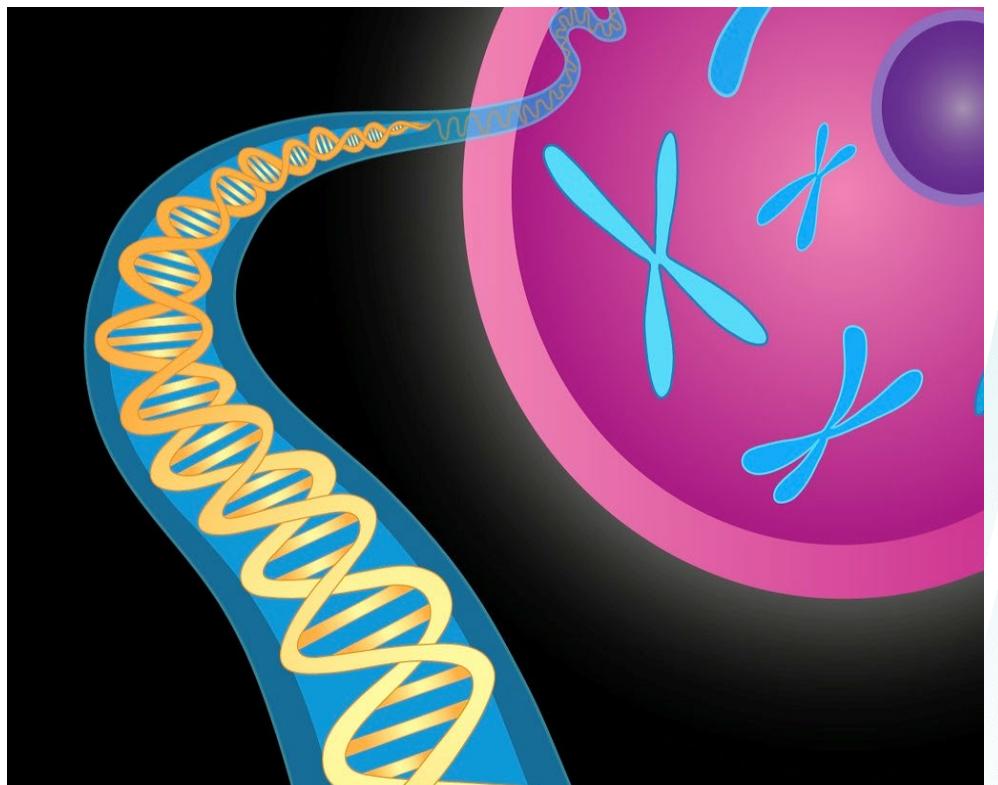
# Algorithm overview

- Generate a random initial population
- Assign a fitness score
- Select the fittest individual(s)
- Recombine features
  - Crossover
  - Mutation
- Repeat until termination criteria is met



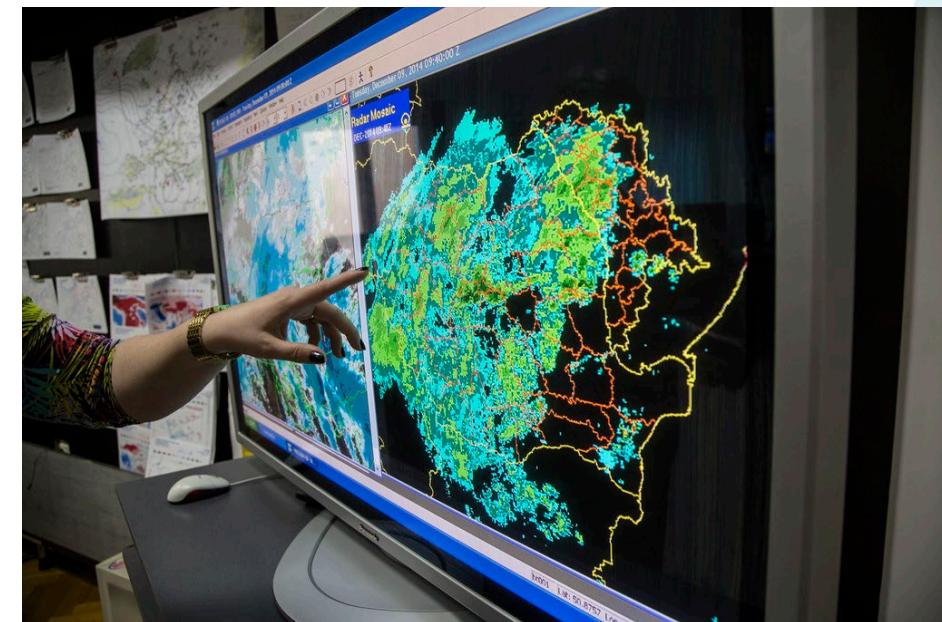
# Problem representation

- Chromosomes
  - A set of genes (features)
- High-level representation
  - Feature values are used directly
- Low-level representation
  - Values are mapped to a simpler form
    - Binary strings



# Evaluation methods

- Depends on the problem one is trying to solve
- Criteria may be static or change over time
- Determines fitness of individuals
  - How “close” are they to the optimum?
  - May be based on cost, speed, efficiency etc.



# Coming up

- Initial population definition
  - Encoding
  - Generation
- Selection processes
- Crossover
- Mutation

# Image credits

- "Flickr is a search engine" by kevin dooley is licensed under CC BY 2.0
- "Sample Analysis at Mars (SAM) Media Day" by NASA Goddard Photo and Video is licensed under CC BY 2.0
- "Chromosomes and DNA double helix" by National Institutes of Health (NIH) is licensed under CC BY-NC 2.0
- "Monitoring and Evaluation System" by World Bank Photo Collection is licensed under CC BY-NC-ND 2.0



# Genetic algorithms

*An optimization algorithm modelled after Darwin's evolutionary theory*

*Lecture 2/3 – Selection, crossover and mutation*

Andreas Dyrøy Jansson

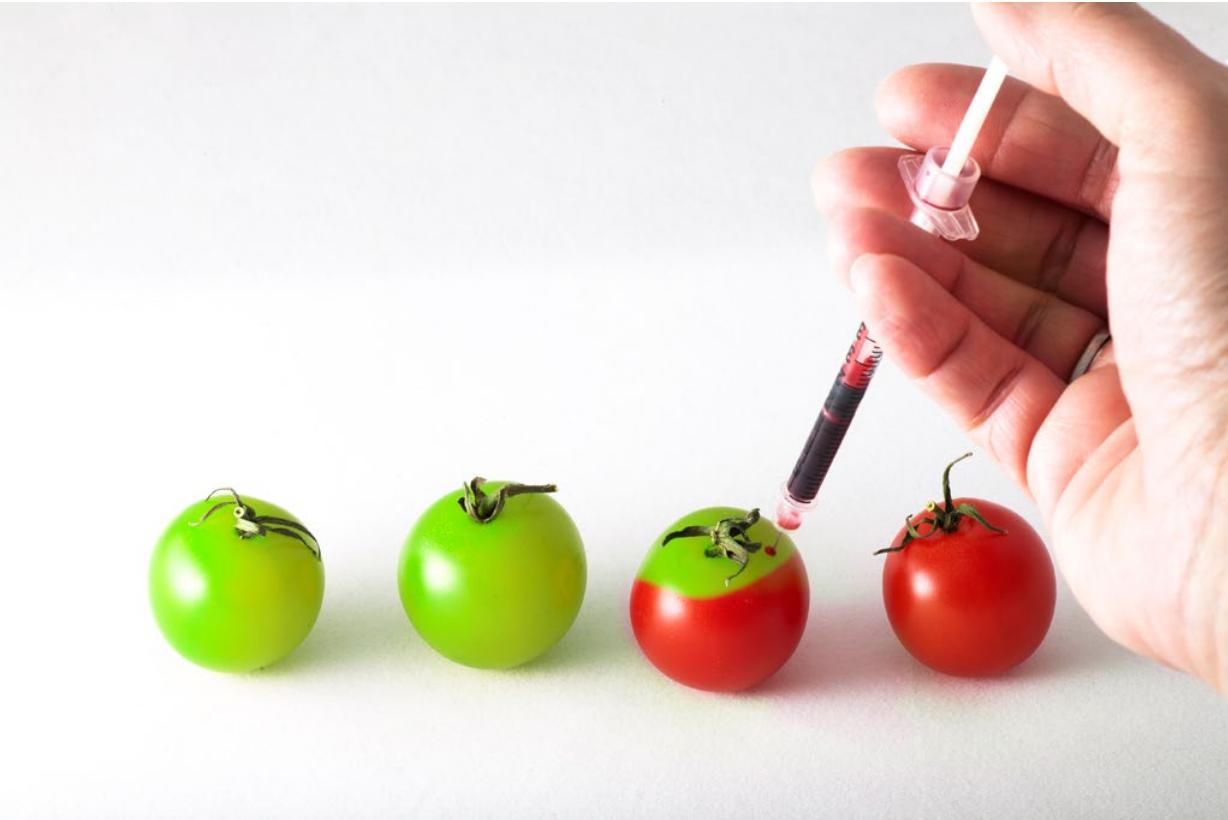
# Initial population definition

- All features generated randomly
- Random range should cover all solutions
- How many individuals?
  - The need for genetic diversity
  - Parallelization



# Operators

- Selection
- Crossover
- Mutation
- Inversion



# Selection

- How individuals are selected for crossover
- Based on fitness score
- Most common:
  - Elitist selection
    - Only the top % fittest (elite) is considered
  - Roulette wheel selection
    - All individuals are assigned a random chance based on their fitness



# Crossover

- How features from each “parent” combine to form offspring
- Single-point
  - Genes are “cut” and “pasted” at a certain point
- Multi-point
  - Multiple genes from each parent are combined in the offspring
- Arithmetic
  - Gene values are “mixed” to make a new value

# Mutation

- Genetic diversity
- Adapt to evolutionary pressure
  - Changing conditions
- May produce new, beneficial features
- Avoid getting “stuck” in sub-optimal solutions
- Representation dependent
  - Bit flipping for binary strings

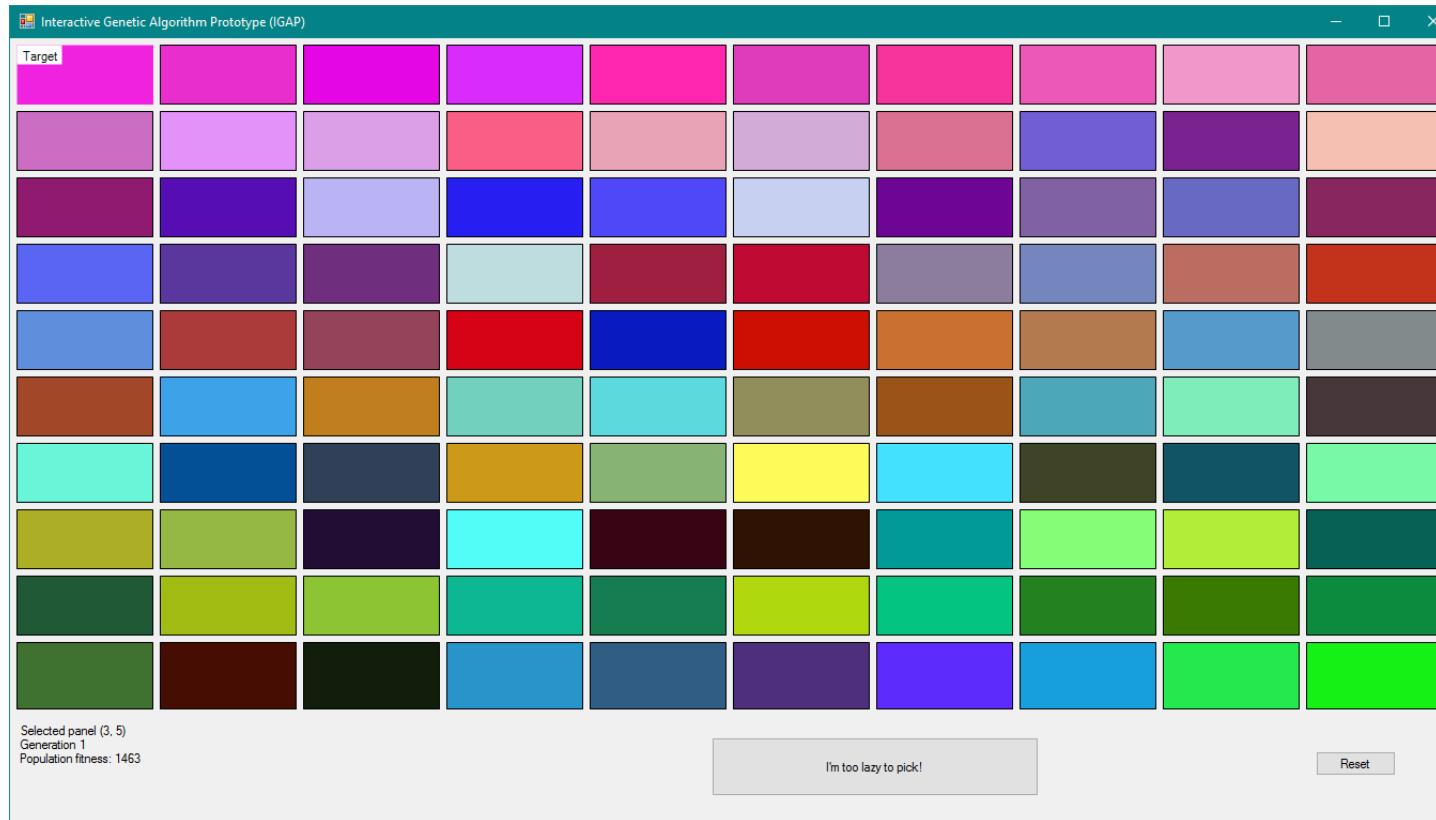


# Evolutionary pressure

- How many offspring are created in every generation, and
- How the new individuals are introduced to the population
- Option 1 – the population increases by the number of new individuals
- Option 2 – The least fit are “killed off” and replaced by the new individuals
  - The number of new individuals determine how many of the old are removed. This relationship may be tweaked for different problems

# Coming up

- A practical example: colors



# Image credits

- "Korea and a World Population of 7 Billion" by United Nations Photo is licensed under CC BY-NC-ND 2.0
- "Genetic Engineering" by stumayhew is licensed under CC BY-NC-ND 2.0
- "It's A Gamble" by MarkyBon is licensed under CC BY-NC-SA 2.0
- "is this cherry a (GMO) Genetically modified organism?" by Kalexanderson is licensed under CC BY-NC-ND 2.0



## Genetic algorithms

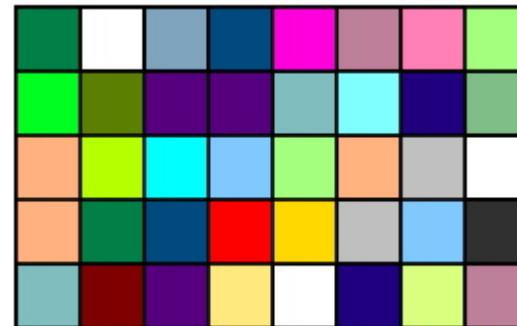
*An optimization algorithm modelled after Darwin's evolutionary theory*

*Lecture 3/3 – A practical example*

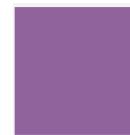
Andreas Dyrøy Jansson

# Practical example: RGB colors

- Problem: find the target color
- Three features/genes: R, G and B
  - Values may range from 0 – 255
- Initial population:



- Target color:



# Representation

- Consider the following color (123, 63, 139):
- High level representation:
  - R=123
  - G=63
  - B=139
- Low level representation
  - Binary string (01111011 00111111 10001011)



# How to determine fitness

- Again, this depends on the task we are trying to solve
  - Target color
  - Compare each feature with the target
  - Fitness  $f$  is based on the difference from the target
    - Smaller difference means a fitter individual
    - Values are assigned as follows: 0 – no similarity, 100 – perfect match
- Other problems may optimize the speed, cost or similar of some system

# Initial population fitness

- Fitness is assigned as values from 0-100 for every individual of the initial population:

- 0 = low fitness
- 100 = high fitness

0	30	20	25	70	60	70	15
25	0	80	80	20	40	45	15
50	25	40	35	15	50	20	30
50	0	25	65	10	20	40	10
35	50	80	10	30	45	25	70

# Selection

- Based on fitness
- In our example, how different is each color from the target
- Elitist selection
  - We select from the top 50% of the population:
  - Only the fittest individuals are used to create the next generation
  - Can potentially find the solution faster, as the worst individuals are removed from the gene pool

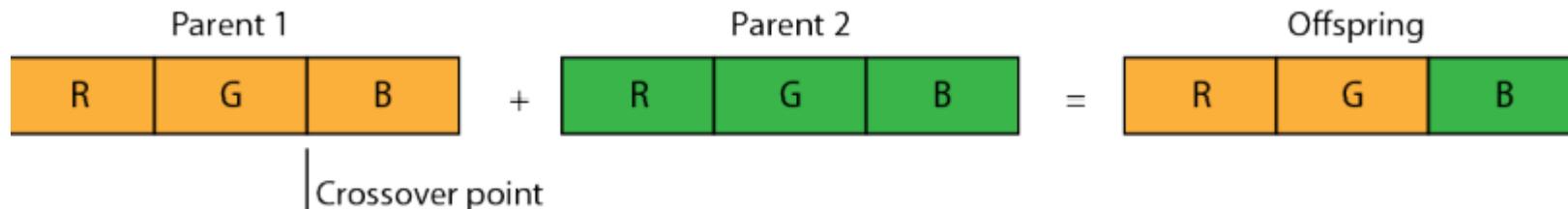


# Selection – roulette wheel

- As before, all individuals are assigned a fitness score from 0-100. Only this time, every individual may be selected for crossover.
- Fit individuals have a higher chance of being selected
- Less fit individuals may still be selected, and their features carried over to the next generation, albeit with a lower chance
  - This may actually be beneficial in the long run to ensure genetic diversity

# Crossover – single point

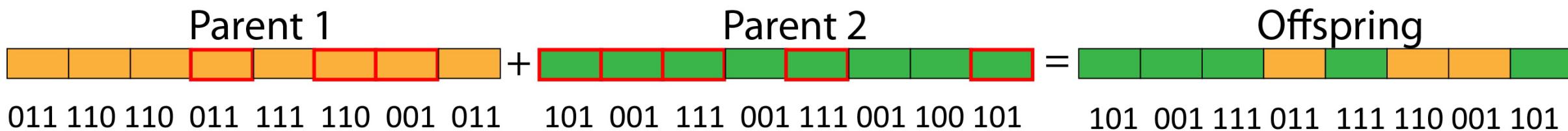
- How features from each “parent” combine
- Two parents are selected using either elitist or roulette wheel selection
- Consider an RGB color as example:



- A crossover point is selected, and determines what genes are copied over from each parent

# Crossover – multi-point

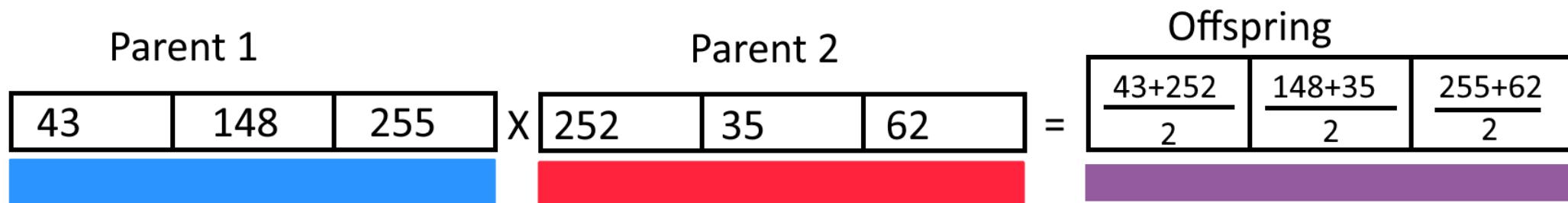
- As before, two parents are selected and their features combined. However, this time, multiple genes are copied randomly from each parent:



- In this example, RGB values are encoded as binary strings and split into 8 chunks.
- Note: Binary representation is not necessary for using multi-point crossover, but makes it easier to show the method in this context.

# Crossover - arithmetic

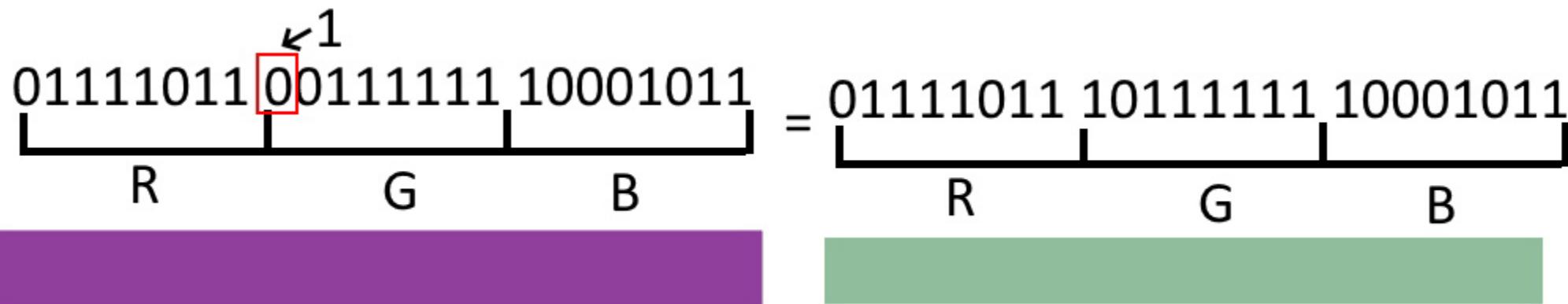
- Instead of just copying features directly from each parent, some arithmetic function may be used. Consider again the RGB color example:



- In this case, we take the average of each feature value to produce the offspring. Using the average is just an example, there is no limit on what function may be used
  - On binary strings, one may for example use bitwise operations

# Mutations

- Mutations are a way to ensure genetic diversity
  - If the population becomes too uniform, it may have a hard time adapting to unexpected changes in its fitness function
- Additionally, the algorithm may converge into a sub-optimal solution before it has tried every combination possible
- Mutations may be implemented in different ways, the most popular being “bit-flip” for binary strings:



# Demonstration





# Natural Language Processing

## Week 5: Introduction to course

Shayan Dadman - Phd Candidate, UiT Narvik  
Room: D3430  
Email: Shayan.dadman@uit.no

Andreas Dyrøy Jansson - Phd Candidate,  
UiT Narvik  
Room: C3190  
Email: Andreas.d.jansson@uit.no

# Objectives of the course

- The main objective of the course is to get familiarized with Natural Language Processing (NLP) and why it is important
- Particularly, you will learn:
  - What is natural language
  - What are the NLP tasks and challenges
  - What are the NLP tools and approaches
  - What are the applications of NLP



# Topics in week5

- Introduction to Natural Language Processing
- NLP tasks and challenges
  - Speech recognition
  - Sentiment analysis
  - Natural language generation
  - etc.
- NLP tools and approaches
  - Natural Language Toolkit (NLTK)
  - Statistical NLP
- NLP applications
- Text preprocessing in NLP

# References

- Foundations of Statistical Natural Language Processing, Christopher D. Manning and Hinrich Schütze
- Mathematical Linguistics (Advanced Information and Knowledge Processing) 1st ed. 2008 Edition, Andras Kornai
- Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit, by Steven Bird, Ewan Klein and Edward Loper (<https://www.nltk.org/book/>)



**UiT** The Arctic University of Norway

## Genetic algorithms

*An optimization algorithm modelled after Darwin's evolutionary theory*

*Genetic Algorithm Code example*

Andreas Dyrøy Jansson  
C3190

Hello there

## What we will cover

- In this lecture, we will take a look at the code for a simple GA
  - Making a class representing a chromosome with a set of features
  - Implement a simple mutation function with constraints
  - Implement arithmetic crossover
  - Implement a mapping function from lower-level to higher-level representation
  - Implement a fitness function based on a given problem
  - Assemble each part into a working algorithm

In this lecture, we will cover the implementation of a complete genetic algorithm. First, we will make a class to represent the chromosome and its features.

Next, we will make a mutation function and show how we can deal with constraints.

We will implement arithmetic crossover, which is the simplest crossover method

We will implement a mapping function to map from one representation level to another

We will implement a fitness function based on a given

problem

And finally, we will assemble all these parts into a complete, working genetic algorithm

## The problem

- Consider the following problem:
  - $a ? b = 42$
  - Where  $a$  and  $b$  are two unknown numbers, and “?” is some operator
- We have the following constraints:
  - $a$  **must** be between -1000 and 1000
  - $b$  **must** be between -200 and 200
  - The operator ? can be either +, -, \*, or /
- In the next few slides, we will solve this problem using a genetic algorithm

For this example, let's consider the following problem: A with B equals 42. A and B are two unknown numbers, and the question mark is a mathematical operator, like addition, subtraction, division and multiplication.

We also have a set of constraints: The value of A **MUST** be larger than -1000 and smaller than 1000, while B must be between -200 and 200. The operator can be one of four possible options: plus, minus, star, or slash.

In the next few slides, we will solve this problem by implementing a genetic algorithm

## The chromosome class

- We start by creating a chromosome class to hold our features
  - GAs are generally easier to implement in an object-oriented approach
  - An alternative is to use dictionaries and multidimensional vectors
- We analyze the problem, and find that we have three features:  $a$ ,  $b$ , and the operator ?
  - We see that the chromosome should represent an **equation**
- We will refer to them in the code as *feature\_1*, *feature\_2*, and *feature\_3*
- We set the fitness to infinity
  - Huh?

```
8  class Chromosome:  
9      def __init__(self, feature_1, feature_2, feature_3):  
10         self.feature_1 = feature_1 # [-1000, 1000]  
11         self.feature_2 = feature_2 # [-200, 200]  
12         self.feature_3 = feature_3 # [+,-,*,/]   
13         self.fitness = float('inf')
```

We start by creating a class representing a chromosome.

In general, it's much easier to implement genetic algorithms through an object-oriented approach. Of course, it's possible to just use lists, dictionaries and multidimensional vectors to implement the GA, but object orientation makes it easier to see the connection with the flowchart we looked at in the previous lectures.

First, we must analyze and understand the problem we are given. We find that we have three features, A, B, and the operator. We conclude that the chromosome we are making should represent an equation.

Going forward, we will refer to these features as feature 1, feature 2, and feature 3.

The chromosome also holds its own fitness value. For the moment, we set this to infinity. This may seem counterintuitive, but if you think about it, we are trying to minimize the difference between the expression in the chromosome, and the given answer. Since we don't have values for A and B, we assume the worst, and set the initial fitness to a very large number.

## Representation

- When implementing a genetic algorithm to solve a problem, we must decide how we want to represent the features
- Numerical values can be used directly
  - We call this high-level representation
  - Can be mutated by adding or subtracting a random value
- In our problem, we also have the operator *feature\_3*
  - We could save it as a string with values "+", "-", "\*", or "/"
- But - how do we "mutate" a string?
  - We can map the string to a numerical representation

Another important factor we must consider is how we want to represent the values in the chromosome. For example, in this case, we can represent feature 1 and feature 2 as numerical values without any mapping. We call this high-level representation. We can mutate the values by adding or subtracting a random number.

We also have feature 3, which is the operator. We could try to just save it in the chromosome as a string, but this raises the question: how can we easily mutate a string? The easiest solution is to map the string to a numerical representation.

## Mapping features

- We create a helper function which takes an integer 0 – 3 and returns the corresponding operator

```
48     def map_to_operator(self, val):  
49         if val == 0:  
50             return "+"  
51         if val == 1:  
52             return "-"  
53         if val == 2:  
54             return "*"  
55         if val == 3:  
56             return "/"
```

Here is an example of how this can be done. We say that 0 represents plus, one is minus, 2 is multiply and 3 is divide.

## Fitness and crossover

- We create simple getters and setters for the fitness
- We implement **arithmetic** crossover
  - For the numerical values `feature_1` and `feature_2`, we can simply take the **average**
  - We can't take the "average" of two operators. We decide to copy `feature_3` randomly from one of the parents (25)
  - The method returns a new chromosome as a mix of both "parents" (23)

```
15 |     def set_fitness(self, new_fitness):  
16 |         self.fitness = new_fitness  
17 |  
18 |     def get_fitness(self):  
19 |         return self.fitness
```

```
21 |     def crossover(self, other):  
22 |         d = bool(rnd.randint(0, 1))  
23 |         return Chromosome((self.feature_1 + other.feature_1) / 2, \  
24 |             (self.feature_2 + other.feature_2) / 2, \  
25 |             self.feature_3 if d else other.feature_3)
```

Next we create simple getters and setters for the fitness. We then implement crossover, and for this example, we use the simplest arithmetic crossover. Intuitively, you can imagine that we are simply taking the average of the two parents to create the offspring. This is easy enough when we are working with numerical values. However, things become slightly trickier when we are dealing with discrete values like the operator in feature 3. It doesn't really make sense to take the average of two operators like + and – directly. This can be solved in many ways, for example, we can take inspiration from multi-point

crossover, and copy the feature directly from one of the parents.

First we generate a random True or False value, and on line 25 in the code, we select the operator from either self or other and copy it to the offspring using the constructor.

Finally, the crossover function returns a new offspring, which is a mix of both parents “self” and “other”.

# Mutation

- To avoid getting stuck in local minima, we use mutation to perturb the population
- We try a mutation rate of 5%
  - Mutation rate is a hyperparameter of GAs
- Since we are using high-level representation on *feature\_1* and *feature\_2*, we offset them with random values (30), (36)
  - Note that the random range corresponds to the legal values of each feature
- Since we mapped the operator to a numerical value, it is easy to mutate in the same way as *feature\_1* and *feature\_2* (42)
- We must also check that we do not exceed the constraints of each feature (31-34), (37-40), (43-46)

```
1 import random as rnd
2
3 mutation_rate = 0.05
```

```
27 def mutate(self):
28     chance = rnd.random()
29     if chance < mutation_rate:
30         self.feature_1 += rnd.randint(-100, 100)
31         if self.feature_1 > 1000:
32             self.feature_1 = 1000
33         elif self.feature_1 < -1000:
34             self.feature_1 = -1000
35
36         self.feature_2 += rnd.randint(-30, 30)
37         if self.feature_2 > 200:
38             self.feature_2 = 200
39         elif self.feature_2 < -200:
40             self.feature_2 = -200
41
42         self.feature_3 += rnd.randint(-2, 2)
43         if self.feature_3 > 3:
44             self.feature_3 = 3
45         elif self.feature_3 < 0:
46             self.feature_3 = 0
```

We also need to implement the mutation operator. This is a function to help the population to avoid getting stuck in a local minimum. Mutations introduce some randomness into the population, and if we are lucky, we get a beneficial mutation that helps certain individuals solve the problem better, resulting in increased fitness.

Usually, the mutation rate is quite low, for example from 2 to 10 percent. We try a mutation rate of 5 percent, but we can tweak this later if we see that it is too high or too low. We say that mutation rate is a hyperparameter of genetic algorithms.

Since we are using high-level representation in feature 1 and feature 2, we can mutate them by adding a random number which can be positive or negative. Note that the mutation range should be adjusted to fit with the legal values for each feature. Feature 1 has a much larger range, -1000 to 1000, so we can use a larger range. Feature 2 can only be between -200 and 200, so we use a smaller range.

Also, since we mapped the operator to numerical values, we can do the same for feature 3. Since feature 3 can only be one of four values, we use a very small range.

Finally, we must check that the mutated values do not violate our constraints. If they do, we simply clamp them to their respective legal values.

# Evaluation

- In order to compute the fitness of each chromosome, we need a way to test it against our problem
  - $feature\_1 \text{ "feature\_3" feature\_2} = 42$
- We create the function “eval()” which computes the expression represented by the chromosome
- We want to minimize the difference between the expression and the answer
  - Low or zero difference thus equals high fitness
- Finally, we make a function to print out the chromosome in a readable format

```
58 |     def eval(self):  
59 |         if self.map_to_operator(self.feature_3) == "+":  
60 |             return self.feature_1 + self.feature_2  
61 |         if self.map_to_operator(self.feature_3) == "-":  
62 |             return self.feature_1 - self.feature_2  
63 |         if self.map_to_operator(self.feature_3) == "*":  
64 |             return self.feature_1 * self.feature_2  
65 |         if self.map_to_operator(self.feature_3) == "/" and self.feature_2 != 0:  
66 |             return self.feature_1 / self.feature_2  
67 |         else:  
68 |             return float('inf')
```

```
70 |     def __str__(self):  
71 |         return str(self.feature_1) + " " + self.map_to_operator(self.feature_3) \  
72 |             + " " + str(self.feature_2) + " = " + str(self.eval())
```

Finally, we must create a way to evaluate and test the chromosome against the problem. In this case, we are creating an expression, so we evaluate each chromosome by computing its expression, and comparing the result to the answer. As we said in the start, we want to minimize the difference between the output of each chromosome, and the actual answer.

To help us debug the code, we also create a str-function to print out the features in a more readable format.

## Create the initial population

- Create an empty list (74)
- Define the population size (75)
- Populate the list with chromosomes (77-79)
  - Initial values are random within the legal ranges

```
74 | population = []
75 | pop_size = 100
76 |
77 | for i in range(pop_size):
78 |     population.append(Chromosome(rnd.randint(-1000, 1000), \
79 |                                         rnd.randint(-200, 200), rnd.randint(0, 3)))
```

Now we can begin to implement the actual genetic algorithm. We create an empty list to hold our population, and we define a population size. We initially set it to 100, but we can change this later. This is another hyperparameter.

We then populate the list with randomly generated chromosomes.

# Selection implementation

5 answer = 42

- We need a way to select the fittest individuals for crossover
- We implement **elitist** selection
- First, assign fitness to each chromosome (86-88)
  - Fitness is the absolute value of the difference. In this particular case, low “self.fitness” actually means high fitness
  - We could *invert* self.fitness by setting it to  $1/difference$ , but this would cause problem when the difference is exactly 0, which is actually what we want
    - In this case, it's easier to just keep it as is
- Sort the population based on fitness (90)
- Select a number of the top fittest pairs and perform crossover (92-95)
  - We choose 20% (92)
- Mutate the new offspring (94)
- We also replace the least fit individuals (95)

```
86     for p in population:  
87         f = abs(p.eval() - answer)  
88         p.set_fitness(f)  
89  
90     population.sort(key=lambda x: x.get_fitness())  
91  
92     for j in range(1, 21):  
93         new_offspring = population[j - 1].crossover(population[j + 1])  
94         new_offspring.mutate()  
95         population[len(population) - j] = new_offspring
```

Now we need a way to select the most fit individuals for crossover. We can use elitist selection, which is easy to implement. The selection algorithm works as follows: first, we compute the fitness of every single chromosome in the population. Since we are minimizing the difference, we define the fitness as the absolute difference between the output of each chromosome, and the answer. So in this case, a low “self.fitness” value is actually the same as high fitness. Alternatively, we could invert this value by taking one over difference, which would produce a large number when the

difference is small, and a small number when the difference is big. However, we want the difference to be exactly zero, and thus would not work as we would get a division by zero. So in this case, it is easier to just keep it as is. Remember that, depending on the type of problem we are solving using GA, how we interpret the fitness value is implementation dependent.

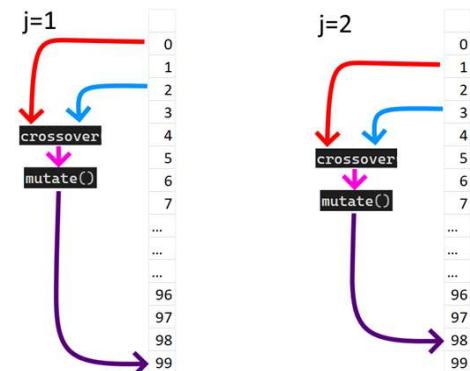
Next, we sort the population based on fitness, or difference in this case.

As we know, elitist selection works by selecting a number of individuals from the most fit percentage for crossover. In this case, we try to pick from the top 20%. This is also a hyperparameter that we can change later. We get a new offspring, apply mutation, and replace one of the least fit individuals.

## Elitist selection breakdown

- We know that the most fit individuals are near the start of the list after sorting
- The for-loop works in the following way:
  - On the first iteration,  $j$  is 1
  - This means that we select individual  $j-1 = 0$  and  $j + 1 = 2$
  - We replace the  $\text{len}()-j$ th individual
- The loop runs for 20 iterations
  - We replace the 20 least fit chromosomes

```
for j in range(1, 21):
    new_offspring = population[j - 1].crossover(population[j + 1])
    new_offspring.mutate()
    population[len(population) - j] = new_offspring
```



Let's take a closer look at how the selection algorithm is implemented. We know that the most fit individuals are near the start of the list after we sorted it. We use a for-loop, and pick pairs of individuals in the following way: first, the value of  $j$  is 1. We take individual  $j$  minus 1 and  $j$  plus one from the list, and perform crossover. So for the first iteration, we pick number zero and two, next we pick number one and three, and so on, until we have 20 new chromosomes. These 20 new chromosomes replace the 20 least fit ones.

## Assembling the algorithm

- We set population size to 100 (74)
  - Another hyperparameter
- We create the initial population (76-78)
- We run the GA until the average fitness (difference from the answer) is less than a certain value (82)
- For each generation:
  - Compute the fitness of each chromosome (86-89)
  - Compute the average fitness (91)
  - Sort the population based on fitness (93)
  - Perform selection (98-101)

```
73 population = []
74 pop_size = 100
75
76 for i in range(pop_size):
77     population.append(Chromosome(rnd.randint(-1000, 1000), \
78                                rnd.randint(-200, 200), rnd.randint(0, 3)))
79
80 average_population_fitness = float('inf')
81
82 while average_population_fitness > 0.1:
83
84     pop_sum = 0
85
86     for p in population:
87         f = abs(p.eval() - answer)
88         p.set_fitness(f)
89         pop_sum += f
90
91     average_population_fitness = pop_sum / len(population)
92
93     population.sort(key=lambda x: x.get_fitness())
94
95     print("Average population fitness: " + str(average_population_fitness) \
96          + " best individual: " + str(population[0]))
97
98     for j in range(1, 21):
99         new_offspring = population[j - 1].crossover(population[j + 1])
100        new_offspring.mutate()
101        population[len(population) - j] = new_offspring
```

Now we have all the parts we need, and we put them together to make the full genetic algorithm.

First, we set the population size to 100, and create the initial population. We then use a while-loop to run the algorithm until the average fitness of the entire population is less than some small value. This is also a hyperparameter we can change. We could also make the while-loop so that it runs for a certain number of iterations before it stops. In GAs, the number of iterations of the while-loop is the same as the number of generations of chromosomes.

So, to summarize: for each generation, compute the fitness of every chromosome, compute the average fitness, sort the population so that the most fit individuals are grouped together, perform the selection and crossover, replace the least fit, and repeat.



## Natural Language Processing

### Week 5: Natural Language Processing

Shayan Dadman - Phd Candidate, UiT Narvik  
Room: D3430  
Email: Shayan.dadman@uit.no

Andreas Dyrøy Jansson - Phd Candidate,  
UiT Narvik  
Room: C3190  
Email: Andreas.d.jansson@uit.no

*Department of Computer Science and Computational Engineering*



# What is natural language?

- It is the way humans communicate with each other using text and speech.
- Indeed, we are surrounded by text:
  - Signs
  - Messages
  - Emails
  - Manuals
  - Web pages
  - Etc.



# Natural Language Processing

- It is a branch of computer science, artificial intelligence (AI)
- It is concerned with machine ability to process and understand text and spoken words
- It combines computational linguistics and statistical modeling.



# Natural Language Processing

- Linguistics
  - It is a scientific study of language
  - Includes grammar, semantics, and phonetics
- Computational linguistics
  - It is a modern study of language using computer science
  - Big data and fast computers
- Statistical natural language processing
  - Similar to computational linguistics, but more engineer-based
  - It more focused on developing tools for machine translation, question answering, etc.



# What are the challenges?

- Natural language processing is still an unsolved task.
- It is hard because:
  - Text has no specific structure:
    - words with different meaning according to the context.
      - I *ran* to the store because I *ran* out of milk
    - Synonyms
    - Irony and sarcasm
    - Domain-specific language
  - It is messy
    - Errors in text and speech



# NLP Tasks

- Speech recognition:
  - known as speech-to-text, is the task of converting voice data into text data, e.g., voice commands.
- Sentiment analysis:
  - To extracts subjective qualities such as attitudes, emotions, sarcasm, etc.
- Natural language generation:
  - To produce natural language. For example, weather report, chatbots, image captions, etc.



# NLP Tools

- Natural Language Toolkit (NLTK)
  - It is an open-source python library to attack NLP tasks.
  - It contains libraries such as, text preprocessing, tokenization, parsing, classification, etc.
  - It also includes sample datasets
- Statistical NLP
  - It uses the machine learning and deep learning algorithms
  - Using these methods, the NLP systems can learn from huge amount of data.
  - Methods such as:
    - Convolutional neural networks (CNN)
    - Recurrent neural networks (RNN)



# NLP Applications

- Here are some use cases of NLP in real world:
  - Spam detection:
    - Initially, this is a text classification task to scan emails for related language. For instance, overuse of financial terms, bad grammar, threatening language, etc.
  - Machine translation:
    - Best example is Google Translate. It is not about the replacing the words but to capture accurately the meaning and tone of the input language and maintain same characteristics in the output language.
  - Chatbots:
    - It is a combination of speech recognition and natural language generation. Examples are Apple's Siri and Amazon's Alexa.



# NLP Applications

- Social media sentiment analysis:
  - Here, sentiment analysis can be used to analyze language of posts, responses, reviews. Furthermore, the companies can use this information for product design, advertisements, etc.
- Text summarization:
  - To summarise the huge volume of text to provide useful context for research databases, busy readers, etc.





# Natural Language Processing

## Week 5: Text Preprocessing

Shayan Dadman - Phd Candidate, UiT Narvik  
Room: D3430  
Email: Shayan.dadman@uit.no

Andreas Dyrøy Jansson - Phd Candidate,  
UiT Narvik  
Room: C3190  
Email: Andreas.d.jansson@uit.no

*Department of Computer Science and Computational Engineering*



# Text Preprocessing

- Data preprocessing is as essential as model building.
- It is even more important for text data due to unstructured nature of it.
- Some common steps include:
  - Lower casing
  - Punctuations Removal
  - Stopwords Removal
  - Frequent words Removal
  - Rare words Removal
  - Emoji Removals
  - Emoticons Removal
  - Conversion of emoticons to words
  - Conversion of emojis to words
  - URLs Removal



# Text Preprocessing

- Lower Casing:
  - It is the most common text preprocessing step.
  - The idea is to convert the input text into the same casing format.
  - This helps the model to treat all texts in the same way.
  - Nevertheless, it may not be helpful for tasks like:
    - Part of Speech tagging
    - Sentiment analysis
- Punctuations Removal:
  - This is part of the text normalization process.
  - For instance, it helps the model to treat *Hi* and *Hi!* the same.
  - Here are the punctuation symbols in python:
    - !"#\$%&\'()\*+,.-./:;<=>?@[\\]^\_{}~`
  - We can add or remove based on the task



# Text Preprocessing

- Stopwords Removal:
  - Stopwords are commonly occurring words like 'the', 'a', etc.
  - They can be removed, as they don't provide valuable information for analysis.
  - However, they are required for Part of Speech tagging task.
- Frequent Words Removal
  - It is the removing process of words that appeared frequently in text
  - It is common in domain specific corpus
- Rare Words Removal
  - Same as previous step, but for the rare words in the corpus



# Text Preprocessing

- Emojis Removal:
  - Depends on the task, for textual analysis it is recommended to remove the emojis.
- Emoticons Removal:
  - Emoticons are different from emojis.
  - Emoticon represent a facial expression by putting characters together.
  - However, emoji is an actual image.
  - Again, based on the task, we may remove the emoticons.
- Conversion of Emoticons/Emojis to Words:
  - In case of sentiment analysis, emoticons/emojis provide valuable information.
  - In this case, instead of removing them we can convert them to words.
- URLs Removal
  - Simply removing the URLs from text for further analysis.



# Text Preprocessing

- You can find the jupyter notebook of the text preprocessing steps in canvas.
  - We used Twitter dataset for this demonstration.





# DTE-2501

Natural Language Processing

*Levenshtein Distance & Term-Frequency Inverse Document Frequency*

Andreas Dyrøy Jansson – Phd Candidate,  
UiT Narvik

Room: C3190

Email: [Andreas.d.jansson@uit.no](mailto:Andreas.d.jansson@uit.no)

Shayan Dadman – Phd Candidate, UiT Narvik

Room: D3430

Email: [Shayan.dadman@uit.no](mailto:Shayan.dadman@uit.no)

# More practical applications of NLP

Levenshtein distance for spell checking

Inverse document frequency for keyword extraction

# Levenshtein distance I

- A way to measure the «edit distance» between two strings
  - I.e., how many changes do we have to make to go from one string to the other
  - Insertions, deletions and substitutions

$$\text{lev}(a, b) = \begin{cases} |a| & \text{if } |b| = 0, \\ |b| & \text{if } |a| = 0, \\ \text{lev}(\text{tail}(a), \text{tail}(b)) & \text{if } a[0] = b[0] \\ 1 + \min \begin{cases} \text{lev}(\text{tail}(a), b) \\ \text{lev}(a, \text{tail}(b)) \\ \text{lev}(\text{tail}(a), \text{tail}(b)) \end{cases} & \text{otherwise,} \end{cases}$$

# Levenshtein distance II

- Example: change *survey* to *surgery*
  1. *survey* -> *surgey* (substitution of v for g)
  2. *surgey* -> *surgery* (insertion of r)
  - Levenshtein distance = 2
- Example 2: change *robot* to *rabbit*
  1. *robot* -> *rabot* (substitution of o for a)
  2. *rabot* -> *rabbot* (insertion of b)
  3. *rabbot* -> *rabbit* (substitution of o for i)
  - Levenshtein distance = 3

# Levenshtein distance III

- Can be computed using (tabular) dynamic programming
    - Wagner–Fischer algorithm
  - Given two strings,  $a$  and  $b$ :
    - Create a matrix of size  $\text{len}(a) + 1$  by  $\text{len}(b) + 1$
    - Initialize the matrix with worst-case values
      - I.e., how many operations must we do to create the strings from nothing?
      - “” to “” requires 0 changes, “” to “s” = 1, “” to “su” = 2, “” to “sur” = 3
    - We are interested in the number of changes to go from  $a$  to  $b$ 
      - “sur” to “sur” requires 0 changes, “surg” to “surv” is 1 change etc.

		s	u	r	g	e	r	y
	0	1	2	3	4	5	6	7
s	1							
u	2							
r	3							
v	4							
e	5							
y	6							

# Wagner–Fischer algorithm

- Given  $a = \text{"survey"}$ ,  $b = \text{"surgery"}$
- Initialize  $d[\text{len}(a) + 1][\text{len}(b) + 1]$
- Set row 0 and column 0 to worst-case
- FOR every character  $c$  in  $b$  ( $j$ ):
  - FOR every character  $k$  in  $a$  ( $i$ ):
    - IF  $c = k$ :
      - cost is 0
    - ELSE:
      - cost is 1
    - Set  $d[i][j] = \text{MIN}(d[i-1][j]+1, d[i][j-1]+1, d[i-1][j-1]+\text{cost})$
- The answer is 2 and is found in  $d[\text{len}(a)][\text{len}(b)]$

	s	u	r	g	e	r	y
s	1						
u	2						
r	3						
v	4						
e	5						
y	6						



[0, 1, 2, 3, 4, 5, 6, 7]
[1, 0, 1, 2, 3, 4, 5, 6]
[2, 1, 0, 1, 2, 3, 4, 5]
[3, 2, 1, 0, 1, 2, 3, 4]
[4, 3, 2, 1, 1, 2, 3, 4]
[5, 4, 3, 2, 2, 1, 2, 3]
[6, 5, 4, 3, 3, 2, 2, 2]

# Tabular approach

		s	u	r	g	e	r	y
	0	1	2	3	4	5	6	7
s	1	0	1	2	3	4	5	6
u	2	1	0	1	2	3	4	5
r	3	2	1	0	1	2	3	4
v	4	3	2	1	1	2	3	4
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	2	2	2

"s" to "" is 1 change  
"s" to "s" is 0 changes  
"s" to "su" is 1 change  
"s" to "sur" is 2 changes  
"s" to "surg" is 3 changes  
"s" to "surge" is 4 changes  
"s" to "surger" is 5 changes  
"s" to "surgery" is 6 changes

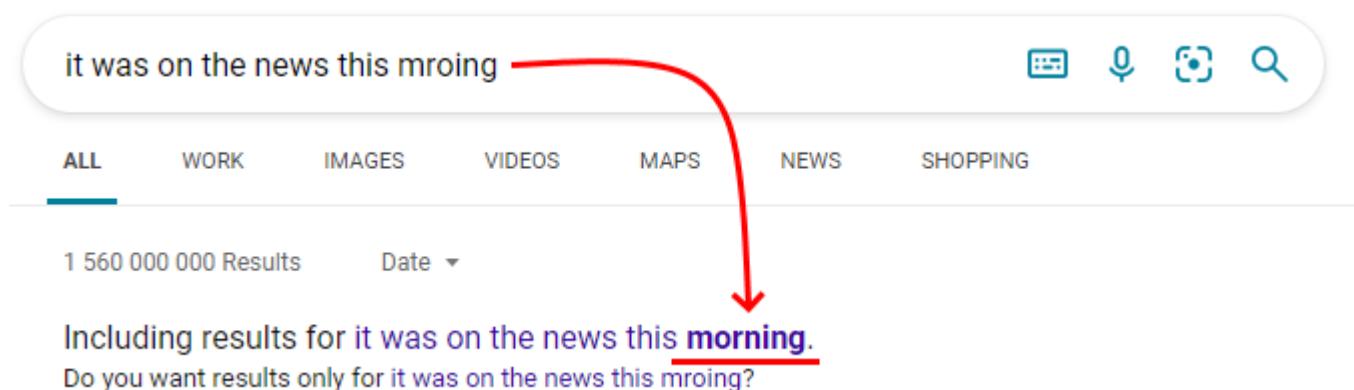
"survey" to "" is 6 changes  
"survey" to "s" is 5 changes  
"survey" to "su" is 4 changes  
"survey" to "sur" is 3 changes  
"survey" to "surg" is 3 changes  
"survey" to "surge" is 2 changes  
"survey" to "surger" is 2 changes  
"survey" to "surgery" is 2 changes

# Spell checking

- Levenshtein distance can be used for simple spell checking
  - Compare input with known words
  - Suggest word(s) with shortest edit distance
- Example:
  - Dictionary: {halloween, halo, hello, help, home}, {work, world, worth}
  - Input: hlllo wrld
  - Compare input to each known word
    - hlllo [> halloween = 5, > halo = 2, > **hello = 1**, > help = 3, > home = 4]
    - wrld [>work = 3, > **world = 2**, > worth = 4]
      - We thus conclude that «hello world» is correct and suggest it to the user

# Other practical applications...

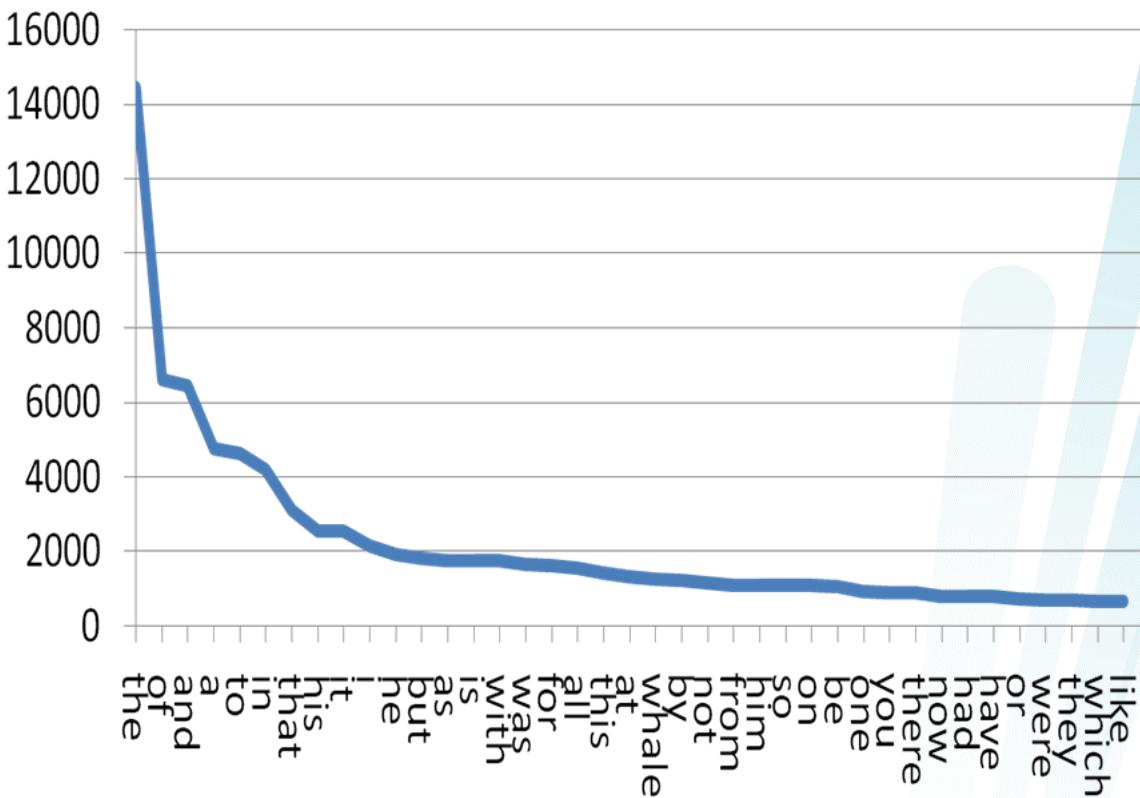
- Levenshtein distance can also be used for clustering words based on similarity, like any other distance metric
  - E.g., Euclidean distance etc. in K-NN
- Comparing two languages and their intelligibility (linguistics)
  - $\text{Lev}(\text{"arbeidslass"}, \text{"Arbeitsplatz"}) = 3$
  - $\text{Lev}(\text{"arbeidslass"}, \text{"lieu de travail"}) = 13$
- Search engines



```
2  def levenshtein_distance(s, t):
3      m = len(s)
4      n = len(t)
5
6      d = [0] * (m + 1)
7      for z in range(m + 1):
8          d[z] = [0] * (n + 1)
9
10     for i in range(1, m + 1):
11         d[i][0] = i
12
13     for j in range(1, n + 1):
14         d[0][j] = j
15
16     for j in range(1, n + 1):
17         for i in range(1, m + 1):
18             cost = 1
19             if s[i - 1] == t[j - 1]:
20                 cost = 0
21             d[i][j] = min([d[i - 1][j] + 1, d[i][j - 1] + 1, d[i - 1][j - 1] + cost])
22
23     return d[m][n]
```

# Keyword extraction and information gain

- Inverse document frequency
  - How often does a word appear in a set of texts?
  - May be used for preprocessing to remove «filler words» automatically
  - Based on Zipf's law
    - This means that the least used words are the most important, or informative
    - Words like «the, of, and, a, to» and so on add little or no value when looking for meaning in a text
  - Similarly, if a word is used multiple times in different, unrelated texts, we assume that this word is less important



# Term frequency–inverse document frequency (TF-IDF) I

- Statistic for determining the importance of a word in a document
- Based on the number of occurrences per document, compared to all the words in the entire collection of documents (corpus)
- Recall Zipf's law – the less frequent a word is, the more important it is
- Term frequency TF is the relative frequency of a word in a single document

$$tf(t, d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}} \rightarrow tf(t, d) = \frac{\text{number of times word } t \text{ appears in } d}{\text{total number of words in } d}$$

# Term frequency–inverse document frequency (TF-IDF) II

- Inverse document frequency IDF is a measure of how much information the word provides
  - If a term  $t$  appears in multiple, or even all, documents, we can assume that this word is less important for the meaning of a single document  $d$
  - Again, based on Zipf's theory

$$\text{idf}(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|} \quad \longrightarrow \quad \text{idf}(t, D) = \log \left( \frac{\text{total number of documents}}{\text{number of documents containing } t} \right)$$

- Combined with TF to form TF-IDF:

$$\text{tfidf}(t, d, D) = \text{tf}(t, d) \cdot \text{idf}(t, D)$$

# An example:

- Corpus – a set of documents containing some text:

This is a long text about banana. This is very nice because you can read it. Here comes the part about the yellow banana. This is a new line. Here comes a new line. And a new line. More new lines all the way to the end. This is the last line.

This is a long text about lime. This is very nice because you can read it. Now comes the line about the green lime. This is a new line. Here comes a new line. And a new line. More new lines all the way to the end. This is the last line.

This is a long text about python. This is very nice because you can read it. Now, here is a new line about python and C++. This is a new line. Here comes a new line. And a new line. More new lines all the way to the end. This is the last line.

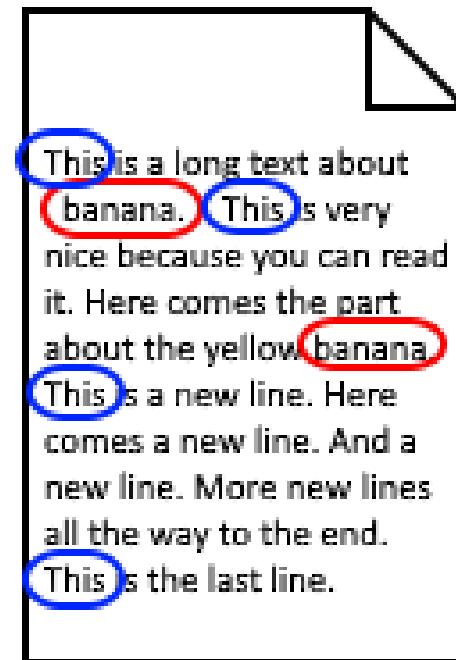
This is a long text about some stuff. This is not the final document, but we are getting close now. This document contains the word document multiple times. I wonder if this is important in some way. Anyway, this is the end. This is the last line.

# The basic concept

- We want to find the words that appear often in ONE document – these are probably important to the meaning of THAT document
  - This is the TF-part
- BUT – we want to «filter out» the words that appear often in ALL the documents
  - These do not add any value, and are most likely filler words
  - This is the IDF-part

# Term Frequency

- Count how often every word appears, in each document
- Considering the first document:
  - «banana» only appears 2 times. This means that the Term Frequency is  $TF = 2/52 = 0.038$
  - «this» appears 4 times,  $TF = 4/52 = 0.077$ 
    - Is it more important to the meaning though?



# Inverse Document Frequency

- This is the LOG-10 of the number of documents in our corpus (4) divided by the number of documents containing the term (word)
- «banana» only appears in the first document
  - $\text{IDF} = \log(4/1) = 0.6$
  - $\text{TF-IDF} = \text{TF} * \text{IDF} = 0.038 * 0.6 = 0.023$
- «this» appears in all 4 documents
  - $\text{IDF} = \log(4/4) = \log(1) = 0$
  - $\text{TF-IDF} = \text{TF} * \text{IDF} = 0.077 * 0 = 0$
  - We can conclude that «this» is a *stop word*

# OO-Implementation

- A Term-class
  - Holds the word itself, number of occurrences in its document, and the TF-IDF value
  - Comparator functions for sorting
  - A function to compute the TF-IDF

```
5  class Term:  
6      def __init__(self, word):  
7          self.word = word  
8          self.count = 1  
9          self.tfidf = 0  
10  
11     def increase_count(self):  
12         self.count += 1  
13  
14     def __lt__(self, other):  
15         return self.tfidf < other.tfidf  
16  
17     def __gt__(self, other):  
18         return self.tfidf > other.tfidf  
19  
20     def __eq__(self, other):  
21         return self.tfidf == other.tfidf  
22  
23     def get_word(self):  
24         return self.word  
25  
26     def compute_tfidf(self, word_count, N, num_occ):  
27         self.tfidf = (self.count / word_count) \  
28             * (math.log(N/num_occ))
```

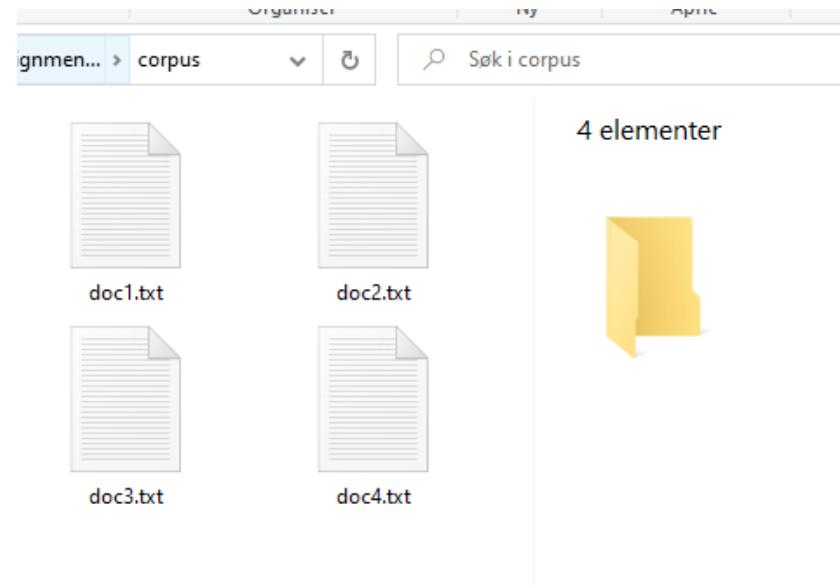
# OO-Implementation

- A Document-class
  - Reads the file, and puts each term in a dictionary
    - \*Here is a good place to do preprocessing, like remove punctuations etc.
  - Count how many times each word appears in the text
  - Functions for checking if a word is present in the document, sorting, and retrieving the most important words according to TF-IDF

```
30  class Document:  
31      def __init__(self, name, file_path):  
32          self.name = name  
33          self.all_terms = {}  
34          self.word_count = 0  
35  
36          with open(file_path) as file:  
37              content = file.read()  
38              terms = content.split()  
39              self.word_count = len(terms)  
40              for t in terms:  
41                  term = t.lower()  
42                  if term in self.all_terms:  
43                      self.all_terms[term].increase_count()  
44                  else:  
45                      self.all_terms[term] = Term(term)  
46  
47      def contains_term(self, term):  
48          return term in self.all_terms  
49  
50      def sort_terms(self):  
51          templist = sorted(self.all_terms.items(), key=lambda x:x[1], reverse = True)  
52          self.all_terms = dict(templist)  
53  
54      def get_top_words(self, n):  
55          top_words = []  
56          for t in list(self.all_terms)[0:n]:  
57              top_words.append(t)  
58          return top_words
```

# OO-Implementation

1. Read the files using the Document-class
  1.  $N =$  the total number of documents
2. Loop over all documents
  1. Loop over all terms
    1. Count how many of the documents the current term appears in
    2. Compute the TF-IDF of every term
3. Sort the terms of each document based on its TF-IDF value
4. Print out the “most important” words in each document



```
61  docs = []
62  for i in range(1, 5):
63      docs.append(Document("doc" + str(i), "\\\corpus\\\doc" + str(i) + ".txt"))
64
65  N = len(docs)
66
67  for doc in docs:
68      for term in doc.all_terms.items():
69          num_occ = 0
70          for doc1 in docs:
71              if doc1.contains_term(term[1].get_word()):
72                  num_occ += 1
73              term[1].compute_tfidf(doc.word_count, N, num_occ)
74
75
76
77  for i in range(len(docs)):
78      docs[i].sort_terms()
79      print("top words in doc" + str(i) + ": ", docs[i].get_top_words(3))
```

# Output

```
C:\Users\Andreas\miniconda3\python.exe
top words in doc0:  ['banana', 'yellow', 'new']
top words in doc1:  ['lime', 'green', 'new']
top words in doc2:  ['python', 'new', 'c++']
top words in doc3:  ['document', 'some', 'stuff']
Press any key to continue . . .
```

# Further reading

- The String-to-String Correction Problem (Levenshtein)  
<https://dl.acm.org/doi/pdf/10.1145/321796.321811>



# DTE-2501 AI Methods and Applications

*Boolean logic vs Fuzzy logic*

*Lecture 1/2 – Probability theory*

Tatiana Kravetc  
*Førsteamanuensis*

*Office: D2240*

*Email: tatiana.kravetc@uit.no*

# Overview

I Repetitive operations and sample spaces

II Events

III Probability

IV Conditional probability

V Example

# I Repetitive operations and sample spaces

*Repetitive operation* is the notion of being able to repeat an operation over and over again «under essentially the same conditions».

- a) Tossing a coin
- b) Throwing a die
- c) Shuffling a pack of playing cards and then cutting the pack
- d) Taking two screws «at random» from a box of 100 screws



*Sample space* is the set or collection of all possible outcomes of a repetitive operation.

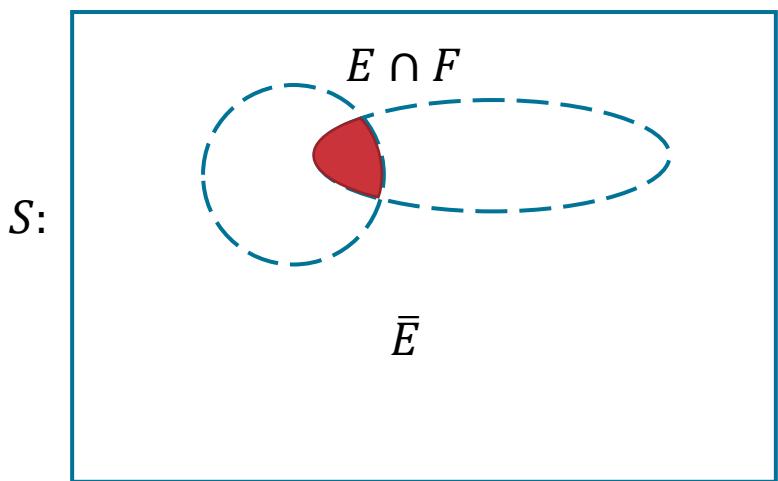
- a) Coin:  $S = \{H, T\}$
- b) Die:  $S = \{1, 2, 3, 4, 5, 6\}$
- c) Playing cards: 52 outcomes
- d) Screws: 4950 possible outcomes

## II Events

$$S = \{e_1, e_2, \dots, e_m\}$$

$$E \subset S$$

*Event* is some subset  $E$  of the sample space  $S$  of particular interest.



# III Probability

$$S = \{e_1, e_2, \dots, e_m\}$$

Example:  $S = \{e_1 = H, e_2 = T\}$

Experiment: toss a coin 200 times



Heads: 108

$$f_1 = \frac{108}{200}$$

→ repeat this experiment infinitely many times

Tails: 92

$$f_2 = \frac{92}{200}$$

→ idealized values for the  $f$ 's

In general, the quantities  $p(e_1), \dots, p(e_m)$  are called *probabilities* of occurrence of  $e_1, \dots, e_m$  respectively and

$$p(e_1) + p(e_2) + \dots + p(e_m) = 1$$

Suppose  $E$  is any event in  $S$ ,  $E = \{e_{i_1}, \dots, e_{i_r}\}$ . The probability of the occurrence of  $E$  is denoted by  $P(E)$ , and is defined as follows:

$$P(E) = p(e_{i_1}) + \dots + p(e_{i_r})$$

#### The axioms of probability

1. If  $E$  is any event in  $S$ , then  $P(E) \geq 0$
2.  $P(S) = 1$
3. If  $E$  and  $F$  are two disjoint events, then  $P(E \cup F) = P(E) + P(F)$

**Rule of complementation**

If  $E$  is an event in a sample space  $S$ , then

$$P(\bar{E}) = 1 - P(E)$$

**General rule of complementation**

If  $E_1, \dots, E_k$  are events in a sample space  $S$ , then

$$P\left(\bigcap_{i=1}^k \bar{E}_i\right) = 1 - P\left(\bigcup_{i=1}^k E_i\right)$$

**Rule of addition of probabilities for mutually exclusive events**

If  $E_1, \dots, E_k$  are *disjoint* events in a sample space  $S$ , then

$$P(E_1 \cup \dots \cup E_k) = P(E_1) + \dots + P(E_k)$$

**Rule for addition of probabilities for two arbitrary events**

If  $E_1$  and  $E_2$  are *any* two events in a sample space  $S$ , then

$$P(E_1 \cup E_2) = P(E_1) + P(E_2) - P(E_1 \cap E_2)$$

More generally, for  $n$  events  $E_1, \dots, E_n$ , we have

$$P(E_1 \cup \dots \cup E_n) = \sum_{i=1}^n P(E_i) - \sum_{j>i=1}^n P(E_i \cap E_j) + \sum_{k>j>i=1}^n P(E_i \cap E_j \cap E_k) + \dots + (-1)^{n-1} P(E_1 \cap \dots \cap E_n)$$

# IV Conditional probability

The probability that an event  $F$  occurs if it is known that an event  $E$  has occurred is called the *conditional probability of  $F$  given  $E$* , and it is denoted by  $P(F|E)$

$$P(F|E) = \frac{P(E \cap F)}{P(E)},$$

where  $P(E) \neq 0$ .

## Rule of multiplication of probabilities

If  $E$  and  $F$  are events in a sample space  $S$ , such that  $P(E) \neq 0$ , then

$$P(E \cap F) = P(E)P(F|E)$$

## Rule of multiplication of probabilities of two independent events

$E$  and  $F$  are independent events if and only if

$$P(E \cap F) = P(E)P(F)$$

# V Example

A box of 1000 screws contains:

- 50 screws with type  $A$  defects;
- 32 screws with type  $B$  defects;
- 18 screws with type  $C$  defects;
- 7 screws with type  $A$  and type  $B$  defects;
- 5 screws with type  $A$  and type  $C$  defects;
- 4 screws with type  $B$  and type  $C$  defects;
- 2 screws with all three types of defects.



Event	Number of elements
$E_A$	50
$E_B$	32
$E_C$	18
$E_A \cap E_B$	7
$E_A \cap E_C$	5
$E_B \cap E_C$	4
$E_A \cap E_B \cap E_C$	2

- a) The probability that the screw will have a type  $A$  or type  $B$  defect, or both, is

$$P(E_A \cup E_B) = P(E_A) + P(E_B) - P(E_A \cap E_B)$$

- b) The probability that the screw will have at least one of three types of defect is

$$P(E_A \cup E_B \cup E_C) = P(E_A) + P(E_B) + P(E_C) - P(E_A \cap E_B) - P(E_A \cap E_C) - P(E_B \cap E_C) + P(E_A \cap E_B \cap E_C)$$

- c) The probability that the screw will be free of these defects is

$$P(\bar{E}_A \cap \bar{E}_B \cap \bar{E}_C) = 1 - P(E_A \cup E_B \cup E_C)$$

- d) Let the screw drawn from the box has a type  $A$  defect. The probability that it also have a type  $B$  defect is

$$P(E_B | E_A) = \frac{P(E_A \cap E_B)}{P(E_A)}$$



**UiT** The Arctic University of Norway

## DTE-2501 AI Methods and Applications

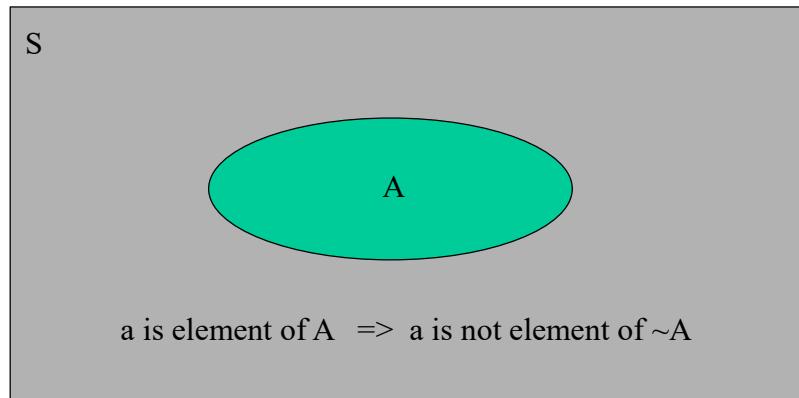
*Lecture 2/2 – Boolean logic vs Fuzzy logic*

Andreas Dyroy Jansson  
*PhD Candidate*  
C3190  
[andreas.d.jansson@uit.no](mailto:andreas.d.jansson@uit.no)

This is lecture 2 of 2 in a series on probabilistics theory and fuzzy logic

## Classic Boolean Logic

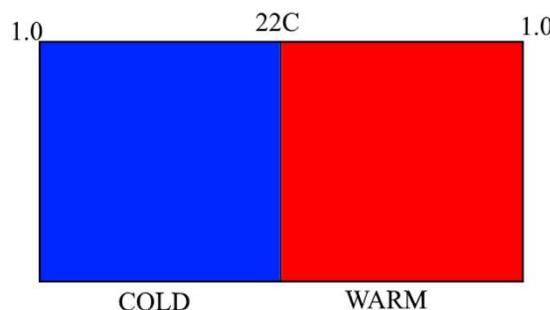
- You are either in or you are out
  - True or false
  - Yes or no



Let's start by taking a look at classic Boolean logic. In Boolean logic, you are working with values that are either true or false, yes or no. You are either part of a set, or not part of a set. Simple as that. This is fine for many applications, but let us consider a simple example.

## A practical example: temperature

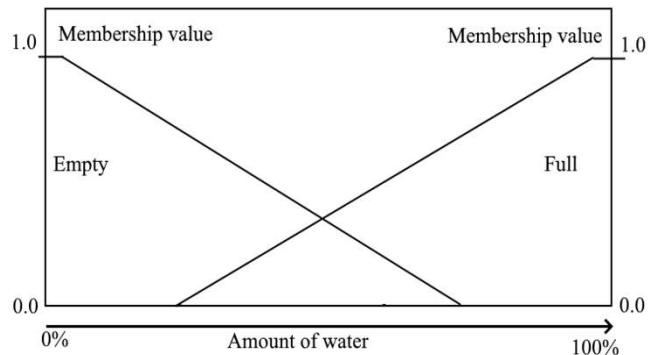
- Strict boolean logic is not always the best way to model a problem
- Example: temperature
  - Everything below 22 C is COLD and everything above is WARM
  - In other words, IF temp > 22 THEN WARM, ELSE COLD



Take temperature. Let us say that we are programming a heater to respond to a certain temperature. If we are working with simple true and false values, we have to determine a set point to work around. If we assume that normal indoor temperature is 22 centigrades, we can program our heater to turn on if it gets below this temperature. In other words, we claim that everything below 22 is cold, and everything above is warm. Then, our heater will do nothing as long as it is warm, and will let the temperature drop until it gets below our set point. Then, it will turn on at full blast, wasting energy.

## Is a glass half empty or half full?

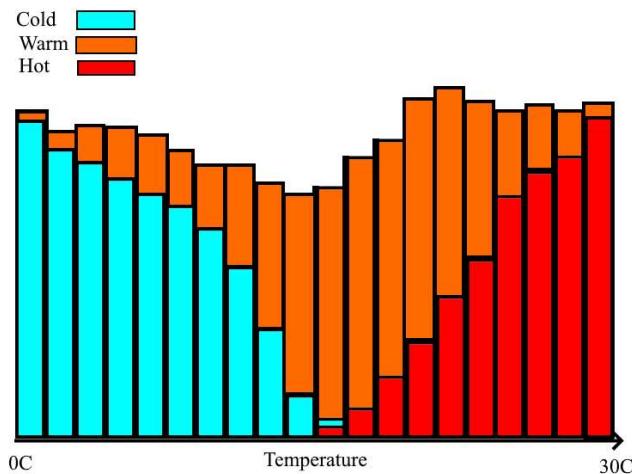
- A possible answer to the age-old debate:
  - Fuzzy sets and fuzzy logic
    - Zadeh, 1965
  - Membership values
  - “Possibility”-theory
  - The glass is part full
    - But also part empty
  - Truth value
    - How “true” is our claim?



Let's consider another example. Is a glass half full or half empty? A possible answer to this can be given using fuzzy logic. The term was introduced by Azerbaijani scientist Lofti Zadeh in 1965, but has roots back to the 1920s. Fuzzy logic is based on the concept of membership values and possibility theory. In other words, we can say that the glass is part full, while still being part empty. Membership values, or truth values, gives us a number between zero and one which says how true a claim is.

## Back to the temperature example

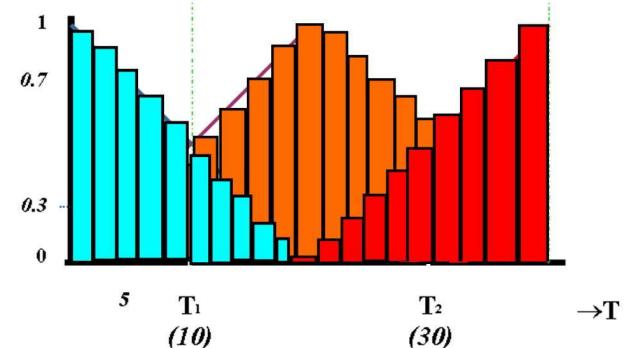
- What is «cold»?
- What is «warm»?
  - Who are we asking?
  - In what context?
- If we ask a group of people, we may get something like this:
- This is our fuzzy set
  - There is no «hard» edge between cold, warm and hot



Let us go back to the temperature example. How can we determine what is cold and warm? It will depend on who we are asking, and in what context. Let's say we ask a group of people to label temperatures from 0 to 30 as cold, warm and hot. If we were to visualize the data, we might get something like this figure. This is our fuzzy set, as there are several overlaps between classes, and there is no hard edge between cold, warm and hot.

## Modelling this as a fuzzy system

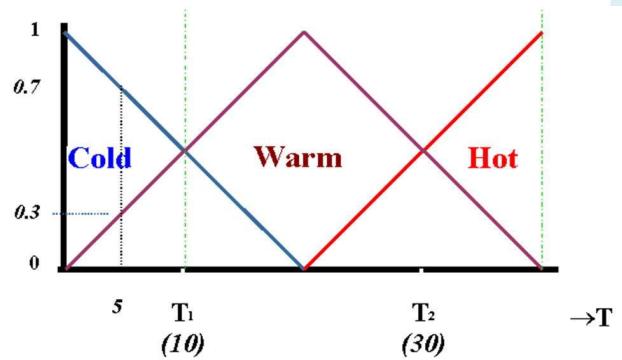
- Three states: Cold, Warm and Hot
- We assign membership values based on our findings
  - We calculate the probability for each state based on our findings – this becomes the truth values for each temperature
- This is called «fuzzification»



We can use our data to model a fuzzy input system. We are working with three states: cold, warm, and hot. We assign the membership values to each class based on our findings, as probabilities of a temperature belonging to a certain class. This also becomes the truth value for each class at a given temperature. This process is called fuzzification. We then get the following fuzzy input system.

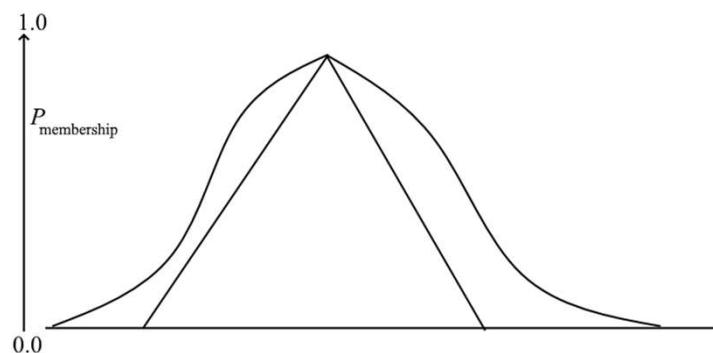
## Modelling this as a fuzzy system

- Three states: Cold, Warm and Hot
- We assign membership values based on our findings
  - We calculate the probability for each state based on our findings – this becomes the truth values for each temperature
- This is called «fuzzification»



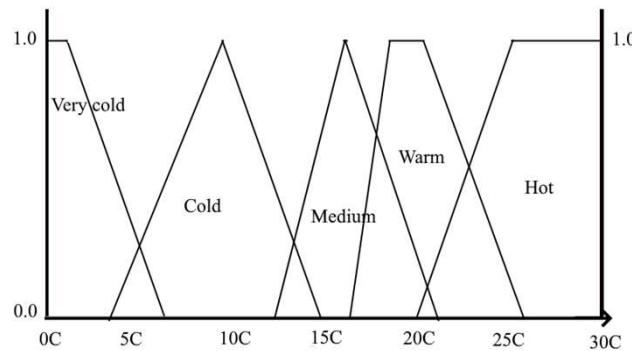
## Different shapes express differences in fuzzy set membership

- In essence, we are interested in the probability of an input belonging to one or more of our classes



Depending on what method we use to calculate the probability of truth value for each class, our function may make take on different shapes. In essence, we are only interested in the probability that an input belongs to one or more of our classes.

We can add more classes as desired:



We can even add more classes if we want. For temperature, we can enhance our input set with classes like very cold, medium, and so on.

## The construction procedure for a fuzzy inference system

1. Express the states to be monitored in real world terms
  1. i.e., VERY COLD, HOT, etc.
2. Fuzzificate by establishing the fuzzy sets for these input states applying a suitable set of functions
3. Define the reaction modes in real world terms
  1. i.e., EMPTY, FULL, HIGH, LOW etc.
4. Fuzzificate similarly – using suitable functions

In summary, we have the following steps for constructing a fuzzy inference system:

First, start, by expressing the states to be monitored in real world terms. So far we have looked at classes like Very cold, hot, et cetera. Then, we fuzzificate by establishing the fuzzy sets for these input states applying a suitable set of functions. We then define the reaction modes in real world terms, like empty, full, high, low, and so on.

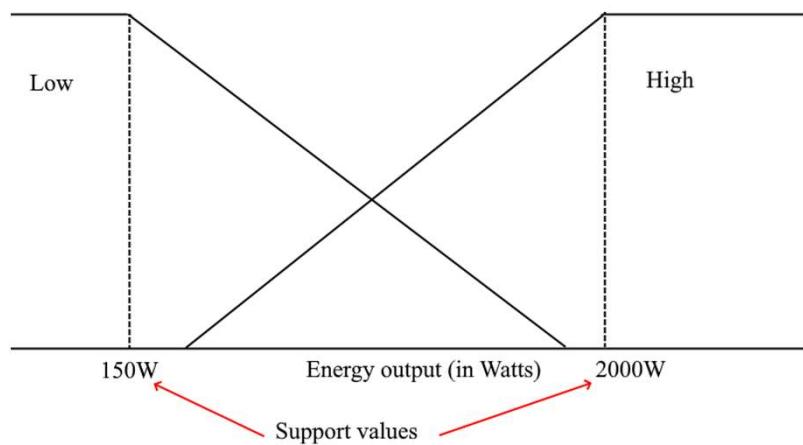
Finally, we use the same fuzzification process on the outputs using suitable functions.

## Using fuzzy logic for control

- Example: heater output
  - Base logic:
    - IF temperature IS COLD THEN HIGH OUTPUT
    - IF temperature IS WARM THEN LOW OUTPUT
  - Again: what is «cold» and what is «warm»?
    - How to deal with uncertainty?

Alright, but why do we do this? Let's look at a practical use case: heater output. How can we program a heater to output the appropriate wattage for any given situation? If we were to simply use a rule like the first we looked at, we would get a rather poor system.

## Heater control based on fuzzy logic



Let's fuzzificate the heater's output. The heater has two main modes: high output and low output.

## Support values

- Each output mode is associated with a support value
  - A parameter for one type of output
- We usually select from the middle of the output set – where the associated truth value is high
- Used for defuzzification

The heater in our example can output between 150 watts and 2000 watts. These will be our support values, associated with each of the output modes. We usually select from the middle of the output set, in other words, where the associated truth value is high. Support values are used in the defuzzification step.

## Defuzzification

- We must defuzzicate in order to reach a conclusion
- **Regular Mamdani:** center of gravity of resulting output set

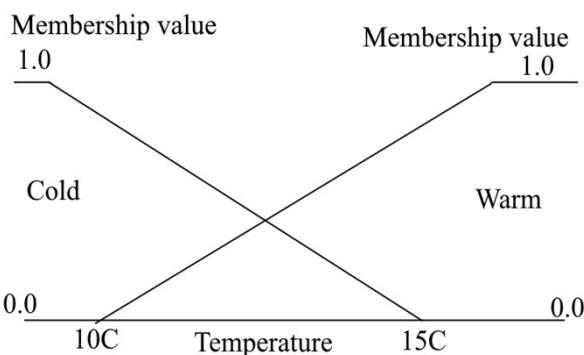
$$X = \frac{\sum x * \mu_x \Delta x}{\sum \mu_x * \Delta x}$$

- **Sugeno** using simple singletons (support values for output set)
  - Non scaled: SUM for all i (input truth value<sub>i</sub> \* support value<sub>i</sub>)
  - Scaled: NonScaled output / SUM for all i(input truth value<sub>i</sub>)

In order to actually reach a conclusion we must defuzzicate. Defuzzification methods include the regular Mamdani, which calculates the center of gravity of our resulting output set, and Sugeno, which use the support values for the output set, called singletons.

## Defuzzification example

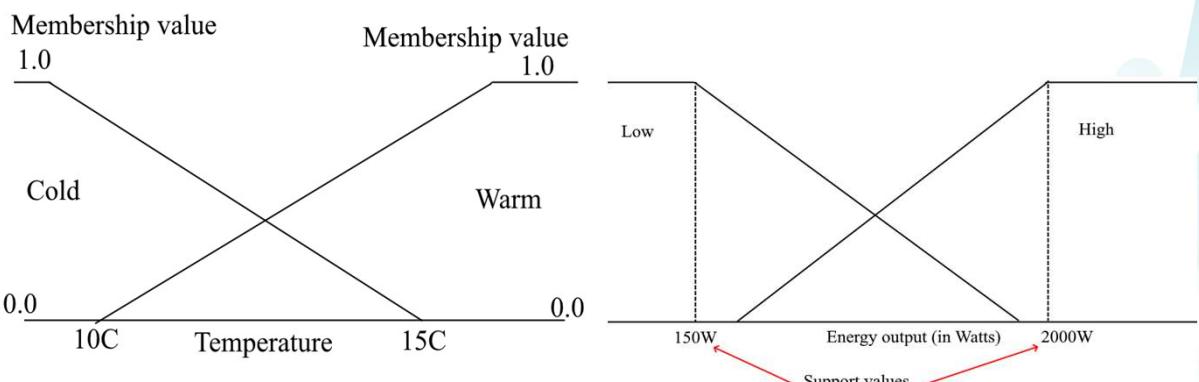
- Given the following inputs and outputs:



Let's go back to our heater control example. On the input side, we have the following. Similarly, for the heater output, we have this.

## Defuzzification example

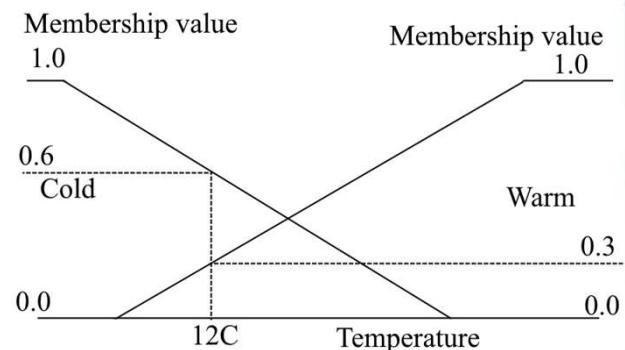
- Given the following inputs and outputs:



Let's go back to our heater control example. On the input side, we have the following. Similarly, for the heater output, we have this.

## Defuzzification example

- Remember:
  - IF temperature IS COLD THEN HIGH OUTPUT
  - IF temperature IS WARM THEN LOW OUTPUT
- Input: 12C
  - This means that it is COLD with a truth value of 0.6
  - But it is also slightly WARM, with a truth value of 0.3



The heater has the following outputs. IF temperature is COLD then output is HIGH. And if temperature is WARM then output is LOW. Let's say that we measure a temperature of 12 degrees. Looking at our input, this means that it is COLD with a truth value of 0.6. But it is also a little bit WARM, with a truth value of 0.3

## Defuzzification example

- From this, we can infer that:
  - Heater must provide HIGH output with a truth value of  $2000 * 0.6 = 1200W$
  - Heater must provide LOW output with a truth value of  $150 * 0.3 = 45W$
- This is harmonized as part of the defuzzification process:
  - $(1200 + 45) / (0.3 + 0.6) = 1383$  Watt
- We therefore conclude that the heater should output 1383 Watt of heat when the temperature is 12C

From this, we can infer that the heater must provide HIGH output with a truth value of 2000 times 0.6, which is 1200 watts. In addition the heater must also provide low output with a truth value of 150 times 0.3 equals 45 watts.

This is harmonized as part of the defuzzification process: 1200 plus 45 divided by 0.3 plus 0.6 equals 1383 watts.

We can therefore conclude that the heater should output 1383 watt of heat when the temperature is 12 degrees.

## Real-world applications of Fuzzy control

- Nissan: fuzzy automatic transmissions
- Nissan: fuzzy anti-skid braking system
- Subaru/Honda: continuously variable (fuzzy) transmission
- Mitsubishi: fuzzy air conditioner (FC 110) Toshiba: Camcorder
- Electrolux: Fuzzy home appliances
- NASA: Fuzzy control in space station docking
- Sendai, Japan: Subway train control
- And many, many more.....

Fuzzy controllers are used in many different applications in the real world, some examples include automatic transmissions for cars, air conditioners and heaters, home appliances, subway train control, and many more.

# When no knowledge of the world, what do you do

Model free method:

You venture the world, you live to learn, you crash and burn in order to learn to live.

You experience and for every step you make you improve =  
Temporal Difference Learning

# Model free approach: Q-learning

- When we address Q-learning we have a model-free method.
- If that is the case we can still train a system, but then we score an action rather than a state.
  - In fact, using the analogy of A\* again we can imagine the  $h(s)$  of  $V(s) = g(s) + h(s)$  is incrementally learned rather than “guessed”.
  - Q-learning is also a TD method.
- The basic steps of the elementary Q-learning method are:
- An agent is in state  $s$
- For all states  $s \in S$ :
  - For all actions  $a_i \in A_s$ :
    - We sample an action  $a_i$
    - We observe the reward and the next state for that action ( $a_i$ )
    - We take the action with the highest Q and move to the next state,  $s'$

# Q-learning

- learns Q-values – no explicit model for state transitions
- Standard Q-learning algo:
  - $$Q(s,a) \leftarrow Q(s,a) + \alpha( R(s) + \gamma \max' Q(s',a') - Q(s,a))$$
  - This incremental update happens every time an action  $a$  is executed in state  $s$  leading to the new state  $s'$ .

# A not-so formal algo description

The Q-Learning algorithm goes as follows:

- 1. Set the gamma parameter, and environment rewards in matrix R.
- 2. Initialize matrix Q to zero.
- 3. For each episode:
  - Select a random initial state (!!!!!!!!)
  - Do While the goal state hasn't been reached.
    - Select one among all possible actions for the current state.
    - Using this possible action, consider going to the next state.
    - Get maximum Q value for this next state based on all possible actions.
    - Compute:
      - $$Q(\text{state}, \text{action}) = \alpha (R(\text{state}, \text{action}) + \gamma * \text{Max}[Q(\text{next state}, \text{all actions})] - Q(\text{s.a}))$$
    - Set the next state as the current state.
  - End Do
- End For

# The formal Q-method

---

*Q*-learning: Learn function  $Q : \mathcal{X} \times \mathcal{A} \rightarrow \mathbb{R}$

---

**Require:**

Sates  $\mathcal{X} = \{1, \dots, n_x\}$

Actions  $\mathcal{A} = \{1, \dots, n_a\}, \quad A : \mathcal{X} \Rightarrow \mathcal{A}$

Reward function  $R : \mathcal{X} \times \mathcal{A} \rightarrow \mathbb{R}$

Black-box (probabilistic) transition function  $T : \mathcal{X} \times \mathcal{A} \rightarrow \mathcal{X}$

Learning rate  $\alpha \in [0, 1]$ , typically  $\alpha = 0.1$

Discounting factor  $\gamma \in [0, 1]$

**procedure** QLEARNING( $\mathcal{X}, A, R, T, \alpha, \gamma$ )

    Initialize  $Q : \mathcal{X} \times \mathcal{A} \rightarrow \mathbb{R}$  arbitrarily

**while**  $Q$  is not converged **do**

        Start in state  $s \in \mathcal{X}$

**while**  $s$  is not terminal **do**

            Calculate  $\pi$  according to  $Q$  and exploration strategy (e.g.  $\pi(x) \leftarrow \arg \max_a Q(x, a)$ )

$a \leftarrow \pi(s)$

$r \leftarrow R(s, a)$

                ▷ Receive the reward

$s' \leftarrow T(s, a)$

                ▷ Receive the new state

$Q(s', a) \leftarrow (1 - \alpha) \cdot Q(s, a) + \alpha \cdot (r + \gamma \cdot \max_{a'} Q(s', a'))$

**return**  $\overset{s}{\overleftarrow{Q}}$

# Q-learning-agent (1)

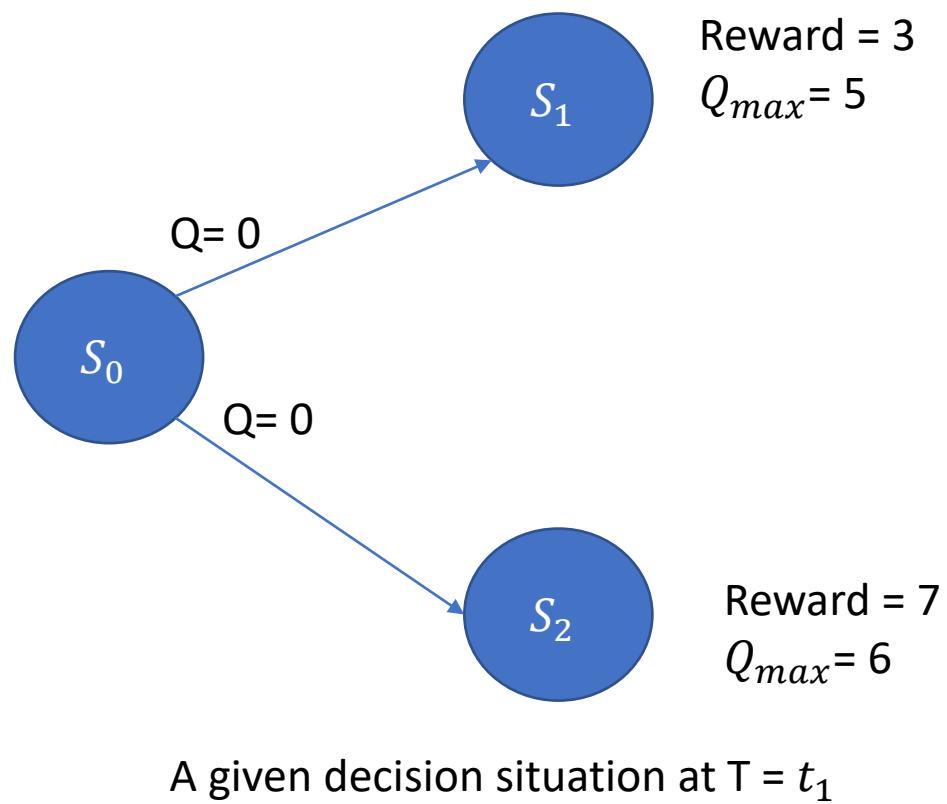
- Function Q-learning      Return *action a*
  - Input: A *percept* indicating the current state ‘*s*’ and *reward signal r*’ =  $R(s)$
  - Persistent values:
    - *Q*, a table of action values indexed by state and action, initialized to zero
    - *Ns,a*, a table of frequencies for state-action pairs, initialized to zero (if all moves are equal can be omitted)
    - *s,a,r* the previous state, action and reward, initially null
      - (continued..... Next)

## Q-learning-agent (2)

- IF state  $s$  is terminal? Then  $Q[s, \text{None}] \leftarrow r'$
- IF  $s$  is *not null* THEN
  - Increment  $N_{s,a}$
  - $Q(s,a) \leftarrow Q(s,a) + \alpha(N_{s,a}[s,a])(r + \gamma \max_{a'} Q[s',a'] - Q[s,a])$
  - $s,a,r \leftarrow s', \arg\max_{a'}, f(Q[s',a'], N_{s,a}[s,a], r')$
- Return  $a$

# Basic principles

$$Q(s,a) \leftarrow Q(s,a) + \alpha( R(s) + \gamma \max' Q(s',a') - Q(s,a))$$



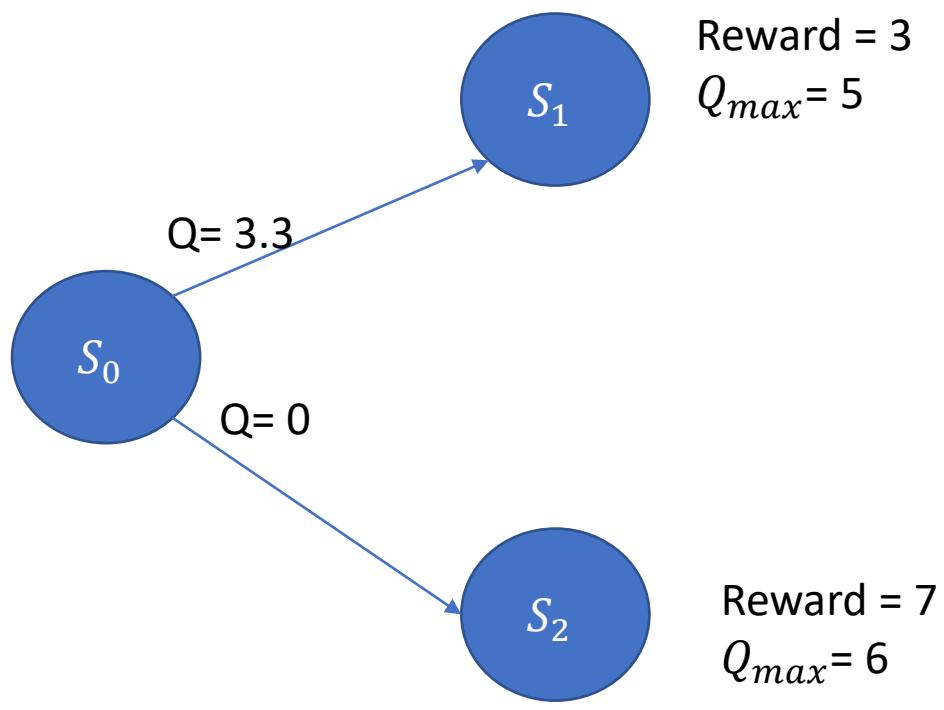
Learning rate  $\alpha = 0,6$   
Discount factor  $\gamma = 0,5$

An agent can move to two other states from  $S_0$ . Which one?

The roulette makes its choice according to the probability:

$$P(S_j, a_{j,i}) = \frac{a_{j,i}}{\sum_{i=1}^A a_{j,i}}$$

# Basic principles



Learning rate  $\alpha = 0,6$   
Discount factor  $\gamma = 0,5$

$$Q'(S_0 a_{0,1}) = Q(S_0 a_{0,1}) + \alpha(R(S_1) + \gamma * Q_{max,S1} - Q(S_0 a_{0,1}))$$
$$Q'(S_0 a_{0,1}) = 0 + 0,6(3 + (0,5 * 5)) - 0 = 3,3$$

An agent can move to two other states from  $S_0$ . Which one?

The roulette makes its choice according to the probability:

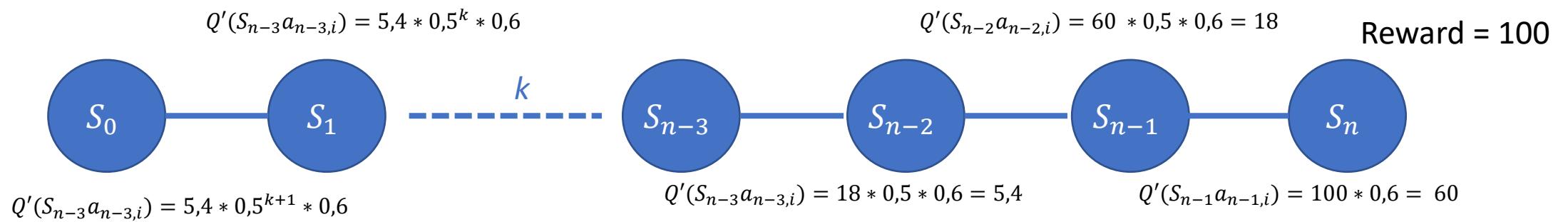
$$P(S_j, a_{j,i}) = \frac{a_{j,i}}{\sum_{i=1}^A a_{j,i}}$$

# Q-value propagation

Recall:

$$G_t = R_{k+1} + \gamma R_{k+2} + \gamma^2 R_{k+3} + \dots = \sum_{k=0} \gamma^k R_{k+1} \quad \gamma \in [0, 1]$$

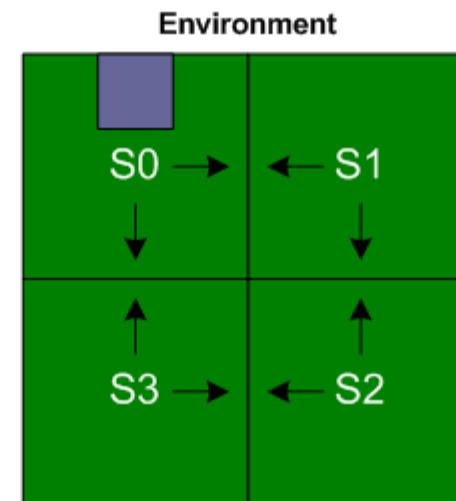
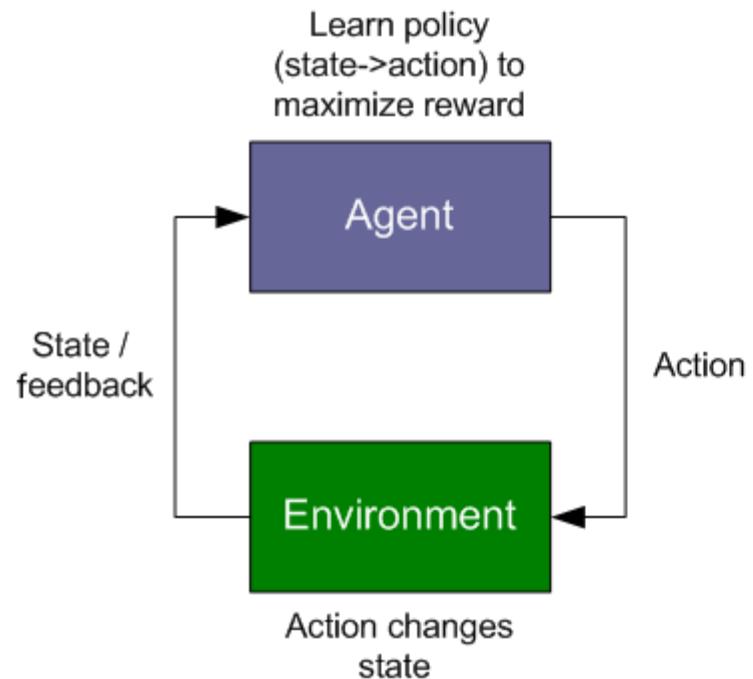
Works for Q-learning too:



Learning rate  $\alpha = 0,6$

Discount factor  $\gamma = 0,5$

# A tabular approach



State	Action	Q-Value
$S_0$	East	$Q_{S1}$
$S_0$	South	$Q_{S3}$
$S_1$	West	$Q_{S0}$
$S_1$	South	$Q_{S2}$
$S_2$	West	$Q_{S3}$
$S_2$	North	$Q_{S1}$
$S_3$	East	$Q_{S2}$
$S_3$	North	$Q_{S0}$

Basic Q-learning requires a table to maintain Q-values (and Rewards)

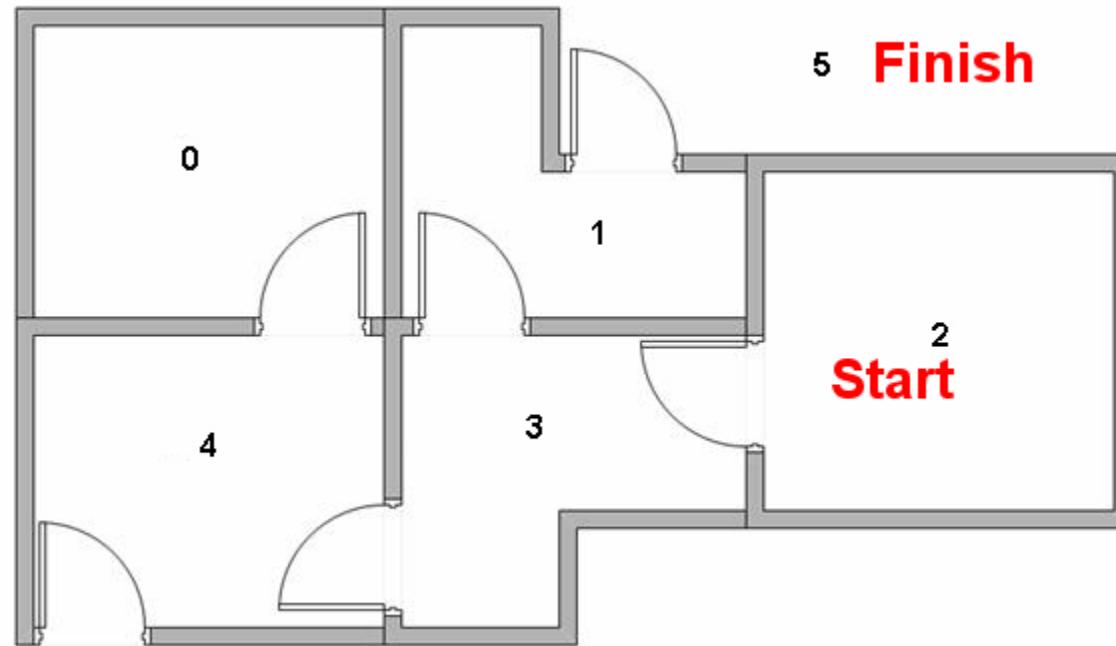
$Q(s, a)$		Action (a)			
State (s)		0	1	2	...
	0	0.21772	0.09723	0.03119	0.04508
	1	0.00168	0.01841	0.09021	0.03007
	2	0.09021	0.03901	0.08233	0.06260
	3	0.07745	0.06769	0.09672	0.09747
	4	0.01272	0.05147	0.09482	0.08170
	...	0.02254	0.02488	0.08215	0.03041

Works only with discrete states  
Requires computational power and memory for larger tasks

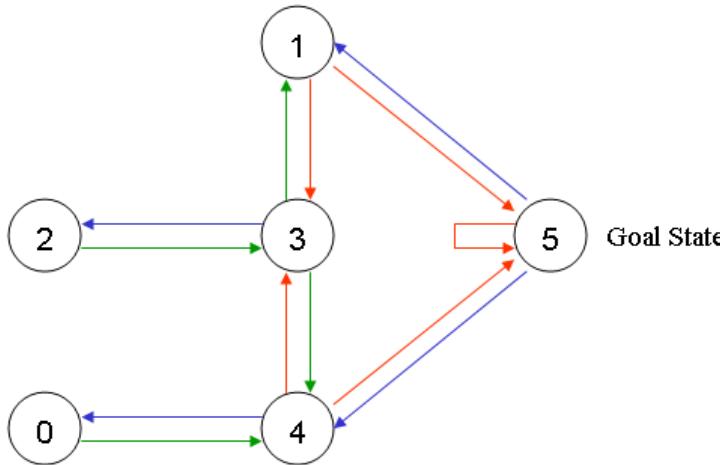
# Example

Suppose we have 5 rooms in a building connected by doors as shown in the figure next. We'll number each room 0 through 4. The outside of the building can be thought of as one big room (5). Notice that doors 1 and 4 lead into the building from room 5 (outside).

(sourced from <http://mnemstudio.org/>)

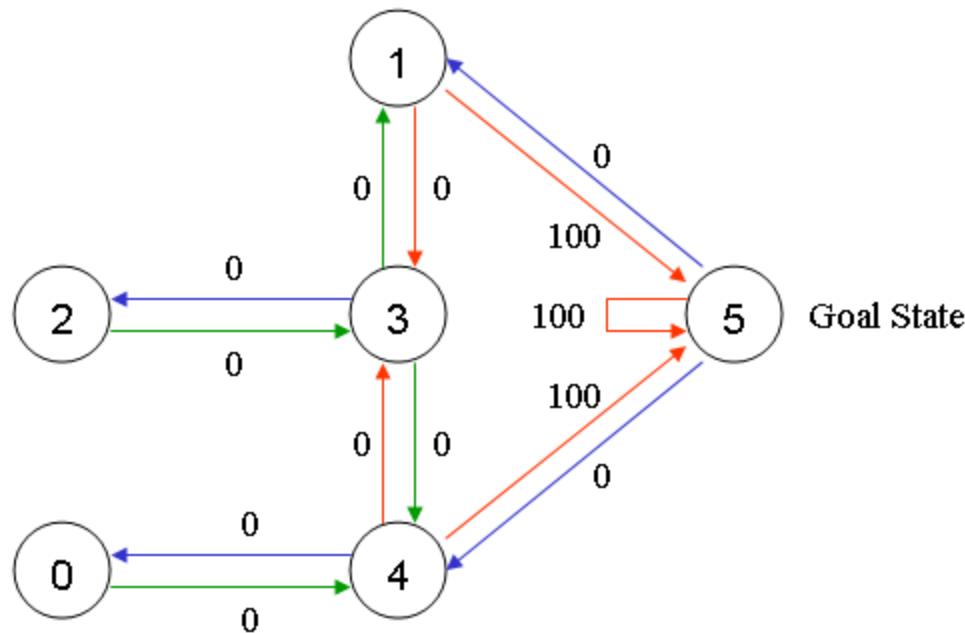


This can be modelled in the usual way – as a graph



For this example, we'd like to put an agent in any room, and from that room, go outside the building (this will be our target room). In other words, the goal room is number 5. To set this room as a goal, we'll associate a reward value to each door (i.e. link between nodes). The doors that lead immediately to the goal have an instant reward of 100. Other doors not directly connected to the target room have zero reward. Because doors are two-way ( 0 leads to 4, and 4 leads back to 0 ), two arrows are assigned to each room. Each arrow contains an instant reward value, as shown below:

# Reward assigned to goal state



Room 5 loops back to itself with a reward of 100, and all other direct connections to the goal room carry a reward of 100.

# Pursue the highest reward

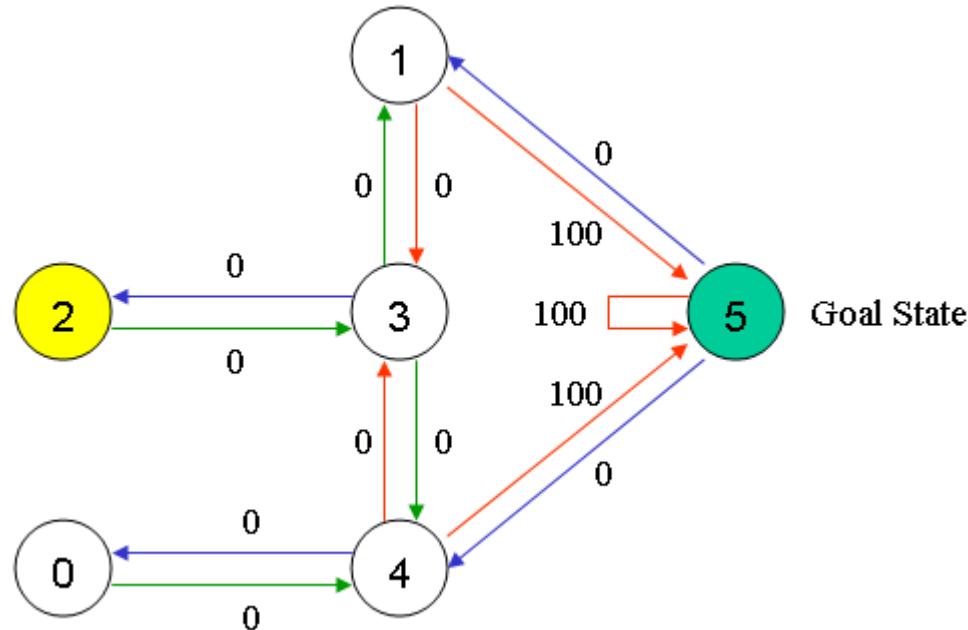
In Q-learning, the goal is to reach the state with the highest reward, so that if the agent arrives at the goal, it will remain there forever. This type of goal is called an "absorbing goal".

Imagine our agent as a dumb virtual robot that can learn through experience. The agent can pass from one room to another but has no knowledge of the environment, and doesn't know which sequence of doors lead to the outside.

Suppose we want to model some kind of simple evacuation of an agent from any room in the building. Now suppose we have an agent in Room 2 and we want the agent to learn to reach outside the house (5).

Room 2 is start state

Room 5 (outdoor) is goal state



Suppose the agent is in state 2. From state 2, it can go to state 3 because state 2 is connected to 3. From state 2, however, the agent cannot directly go to state 1 because there is no direct door connecting room 1 and 2 (thus, no arrows). From state 3, it can go either to state 1 or 4 or back to 2 (look at all the arrows about state 3). If the agent is in state 4, then the three possible actions are to go to state 0, 5 or 3. If the agent is in state 1, it can go either to state 5 or 3. From state 0, it can only go back to state 4.

# The Q update

*The agent starts out knowing nothing, the matrix Q is initialized to zero.*

*In this example, for the simplicity of explanation, we assume the number of states is known (to be six).*

*If we didn't know how many states were involved, the matrix Q could start out with only one element. It is a simple task to add more columns and rows in matrix Q if a new state is found.*

# Reward matrix R

$$R = \begin{matrix} & \text{Action} \\ \text{State} & \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{matrix}$$

Hitting the wall is penalized  
So is cycling (beyond the goal state)

# The Q update

*The agent starts out knowing nothing, the matrix Q is initialized to zero.*

*In this example, for the simplicity of explanation, we assume the number of states is known (to be six).*

*If we didn't know how many states were involved, the matrix Q could start out with only one element. It is a simple task to add more columns and rows in matrix Q if a new state is found.*

The Q matrix is updated incrementally

- Now we'll add a similar matrix, "Q", to the brain of our agent, representing the memory of what the agent has learned through experience.
  - The rows of matrix Q represent the current state of the agent, and the columns represent the possible actions leading to the next state (the links between the nodes)

$$Q = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

# The transition rule

The transition rule of Q learning is a very simple formula:

$$Q(\text{state}, \text{action}) = R(\text{state}, \text{action}) + \text{Gamma} * \text{Max}[Q(\text{next state}, \text{all actions})]$$

*According to this formula, a value assigned to a specific element of matrix Q, is equal to the sum of the corresponding value in matrix R and the learning parameter Gamma, multiplied by the maximum value of Q for all possible actions in the next state. (assuming the learning coefficient  $\alpha = 1$ )*

# Updating the Q matrix

Initially

$$Q = \begin{matrix} & \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \left[ \begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{matrix} \right] \end{matrix}$$

		Action					
		0	1	2	3	4	5
State	0	-1	-1	-1	-1	0	-1
	1	-1	-1	-1	0	-1	100
2	-1	-1	-1	0	-1	-1	
3	-1	0	0	-1	0	-1	
4	0	-1	-1	0	-1	100	
5	-1	0	-1	-1	0	100	

$$R = \begin{matrix} & \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \left[ \begin{matrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{matrix} \right] \end{matrix}$$

Assume agent is in state 5. Look at the sixth row of the reward matrix R (i.e. state 5). It has 3 possible actions: go to state 1, 4 or 5.

$$\text{Q(state, action)} = \text{R(state, action)} + \text{Gamma} * \text{Max}[\text{Q(next state, all actions)}]$$

$$Q(1, 5) = R(1, 5) + 0.8 * \text{Max}[Q(5, 1), Q(5, 4), Q(5, 5)] = 100 + 0.8 * 0 = 100$$

The road onwards is analyzed, values gathered from the present Q matrix

The memory of the reward is injected into the brain of the agent =  
The updated Q-matrix

New value= 100  
inserted  
here

$$Q = \begin{matrix} & \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \left[ \begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{matrix} \right] \end{matrix}$$

For the next episode, we start with a randomly chosen initial state. This time, we have state 3 as our initial state.

Look at the fourth row of matrix R; it has 3 possible actions: go to state 1, 2 or 4. By random selection, we select to go to state 1 as our action.

Now we imagine that we are in state 1. Look at the second row of reward matrix R (i.e. state 1). It has 2 possible actions: go to state 3 or state 5. Then, we compute the Q value:

$$Q(\text{state, action}) = R(\text{state, action}) + \text{Gamma} * \text{Max}[Q(\text{next state, all actions})]$$

$$Q(3, 1) = R(3, 1) + 0.8 * \text{Max}[Q(1, 2), Q(1, 5)] = 0 + 0.8 * \text{Max}(0, 100) = 80$$

We use the updated matrix Q from the last episode.  $Q(1, 3) = 0$  and  $Q(1, 5) = 100$ . The result of the computation is  $Q(3, 1) = 80$  because the reward is zero. The matrix Q becomes:

$$Q = \begin{matrix} & \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \left[ \begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 80 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{matrix} \right] \end{matrix}$$

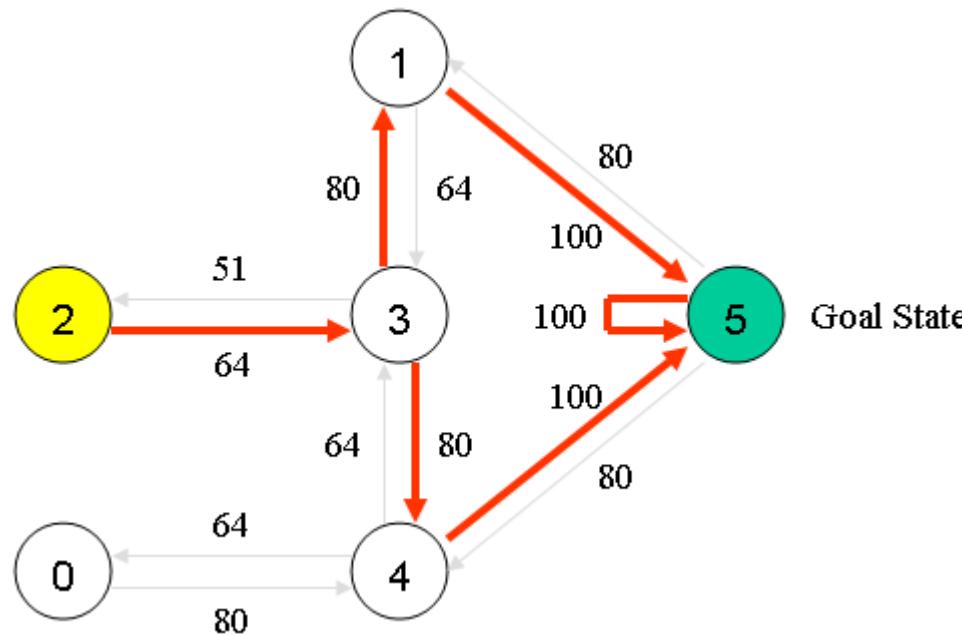
# After several episodes

$$Q = \begin{matrix} & \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \left[ \begin{matrix} 0 & 0 & 0 & 0 & 400 & 0 \\ 0 & 0 & 0 & 320 & 0 & 500 \\ 0 & 0 & 0 & 320 & 0 & 0 \\ 0 & 400 & 256 & 0 & 400 & 0 \\ 320 & 0 & 0 & 320 & 0 & 500 \\ 0 & 400 & 0 & 0 & 400 & 500 \end{matrix} \right] \end{matrix}$$

# After normalization

$$Q = \begin{matrix} & \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \left[ \begin{matrix} 0 & 0 & 0 & 0 & 80 & 0 \\ 0 & 0 & 0 & 64 & 0 & 100 \\ 0 & 0 & 0 & 64 & 0 & 0 \\ 0 & 80 & 51 & 0 & 80 & 0 \\ 64 & 0 & 0 & 64 & 0 & 100 \\ 0 & 80 & 0 & 0 & 80 & 100 \end{matrix} \right] \end{matrix}$$

After several episodes when the system converges and exploitation rather than exploration dominates (steady state)



For example, from initial State 2, the agent can use the matrix Q as a guide:

From State 2 the maximum Q values suggests the action to go to state 3.

From State 3 the maximum Q values suggest two alternatives: go to state 1 or 4. Suppose we arbitrarily choose to go to 1.

From State 1 the maximum Q values suggests the action to go to state 5.

Thus the sequence is 2 - 3 - 1 - 5.

# Q-learning basics

- $Q_{k+1}(s, a) = E[R_{t+1} + \gamma \max_{a'} Q_k(S_{t+1}, a') | S_t = s, A_t = a]$ 
  - This suggests that we build state values incrementally.
  - For the action  $a$  in state  $s$  the value assigned at iteration  $k+1$  is the expected value of  $R_{t+1}$  at the state where  $a$  takes the agent plus the discounted value of the best action,  $a'$  afterwards. Q-learning propagates rewards from states that lie ahead backwards along the string of selected actions.



## DTE-2501 AI Methods and Applications

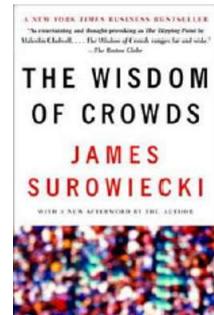
*Lecture 1/2 – Collaborative filtering*

Andreas Dyroy Jansson  
*PhD Candidate*  
C3190  
[andreas.d.jansson@uit.no](mailto:andreas.d.jansson@uit.no)

This is lecture 1 of 2 in a series on collaborative filtering and ensemble methods. We will start by examining the concept of crowd intelligence.

# The Wisdom of Crowds

- Under some conditions, group IQ > individual IQ
- Examples:
  - Guess the weight of a bull: 542,9 kg (543,4 kg)
  - Guess the number of sweets in a jar: 871 (850)
  - Find the way out of a maze:
    - First individual attempt: 34,3 turns on average
    - Second individual attempt: 12,8 turns on average
    - Majority turn decision: 9 turns on average
    - -> We see that group decision is better than best individual



Under some conditions, we observe that the collective intelligence of a group is higher than that of the best individual. Here are some examples from the book “The wisdom of crowds”:

A group of people were asked to guess the weight of a bull at a country fair. The average guess was only half a kilo off. Similarly, when guessing the number of sweets in a jar, the average guess was much closer than any individual guess. When trying to find their way out of a maze, even after two tries, the best individual still had to take more turns than a collective group. What all of

these examples show us is that in these cases, the group decision is better than the best individual.

## Some other examples

- Online user reviews and product recommendations
- Google search - based on collective linking preferences (pageRank)
- eBay edge: collective reputation building
- Netflix, Spotify, Facebook etc.
- Voting
  - How a user's preference is gauged

Some other examples include things like online user reviews and product recommendations. We are more likely to trust a product's review if multiple people vouch for it. It is also less likely to be a scam, than if only one person is shilling for it. Similarly, Google search is more likely to place websites higher up on their results based on how many other webpages are linking to it. In short, pages are ranked based on collective linking preferences. There are many other determining factors when it comes to Google search, but that is a discussion for another day. We also see this on sites like eBay: sellers gain

reputation and trust from their buyers, which again makes them less likely to be shills or scams. When you are recommended a movie or show from Netflix, songs on Spotify and posts on Facebook, the underlying algorithms are based on the activity of a subgroup with similar preferences.

Going forward, we will use the term “Voting”. Votes in this context is a measure of whether a user liked something or not. This can be measured in different ways depending on the context. On Netflix, Facebook and Spotify you got “like” and “dislike” buttons. This is quite intuitive, but there are also less obvious ways to measure user preference. You could for instance look at the time a user spends on a certain activity, and make assumptions. YouTube does this quite effectively, and will recommend videos based on your watch history, even if you didn’t interact with the video through likes or dislikes.

## Memory-based algorithms for Collaborative Filtering (Breese et al, UAI98):

- $v_{i,j}$  = vote of user  $i$  on item  $j$
- $I_i$  = items for which user  $i$  has voted
- Mean vote for  $i$  is

$$\bar{v}_i = \frac{1}{|I_i|} \sum_{j \in I_i} v_{i,j}$$

- Predicted vote for “active user”  $a$  is weighted sum

$$p_{a,j} = \bar{v}_a + \kappa \sum_{i=1}^n w(a, i) (v_{i,j} - \bar{v}_i)$$

normalizer                    weights of  $n$  similar users

Now we will take a look at some algorithms for Collaborative Filtering. The basic formula is based on a sum of weighted votes. Remember that a vote can be more than a direct interaction. We compute the mean vote for the user  $i$  for every item  $j$  as such. In short, we are creating a profile for our user. We are then able to use this profile to predict their vote, or interest or whatever, based on the votes of similar users. The other users’ votes are weighted based on similarity. We may also introduce a normalization factor to account for any deviations and outliers.

## How to compute weights

- K-nearest neighbor

$$w(a, i) = \begin{cases} 1 & \text{if } i \in \text{neighbors}(a) \\ 0 & \text{else} \end{cases}$$

- Pearson correlation coefficient (Resnick '94, GroupLens):

$$w(a, i) = \frac{\sum_j (v_{a,j} - \bar{v}_a)(v_{i,j} - \bar{v}_i)}{\sqrt{\sum_j (v_{a,j} - \bar{v}_a)^2} \sqrt{\sum_j (v_{i,j} - \bar{v}_i)^2}}$$

- Cosine distance

$$w(a, i) = \sum_j \frac{v_{a,j}}{\sqrt{\sum_{k \in I_a} v_{a,k}^2}} \frac{v_{i,j}}{\sqrt{\sum_{k \in I_i} v_{i,k}^2}}$$

The weighted votes of similar users can be computed in many ways. One way you should be familiar with by now is the K nearest neighbor. This is quite simple in that either you are part of a neighborhood, or not. The weights thus become zero and one. We also have more advanced methods like the Pearson correlation coefficient and Cosine distance.

## How to compute weights 2

- Cosine with “inverse user frequency”  $f_i = \log(n/n_j)$ , where  $n$  is number of users,  $n_j$  is number of users voting for item  $j$

$$w(a, i) = \frac{\sum_j f_j \sum_j f_j v_{a,j} v_{i,j} - (\sum_j f_j v_{a,j})(\sum_j f_j v_{i,j})}{\sqrt{UV}}$$

where

$$U = \sum_j f_j (\sum_j f_j v_{a,j}^2 - (\sum_j f_j v_{a,j})^2)$$

$$V = \sum_i f_i (\sum_i f_i v_{i,j}^2 - (\sum_i f_i v_{i,j})^2)$$

We also have the concept of inverse user frequency, which basically means that universally liked items are less useful than less common items when finding similar users. If a lot of users like a lot of the same thing, it becomes harder to find the common factor for the subgroups. We can account for this by using the log function; the frequency for item  $j$  thus becomes  $\log$  number of users divided by the number of users voting for item  $j$ . This can also be combined with the previous methods, like the Cosine distance as seen here.

## Evaluation

- split users into train/test sets
- for each user  $a$  in the test set:
  - split  $a$ 's votes into observed ( $I$ ) and to-predict ( $P$ )
  - measure average absolute **deviation** between predicted and actual votes in  $P$
  - predict votes in  $P$ , and form a **ranked list**
  - assume (a) utility of  $k$ -th item in list is  $\max(v_{a,k}, d, 0)$ , where  $d$  is a “default vote” (b) probability of reaching rank  $k$  drops exponentially in  $k$ . Score a list by its expected utility  $R_a$
  - average  $R_a$  over all test users

Let's close off by looking at evaluation. Start by splitting users into training and test sets. Then we loop over every user  $a$  in the test set. We split the active users votes into two groups: observed and to-predict. We measure the average absolute deviation between the predicted votes and actual votes. We predict the votes and form a ranked list. We assume A, that the utility of the  $k$ -th item in the list is the maximum of the difference between the user's vote and a default vote, and B, that the probability of reaching rank  $k$  drops exponentially in  $k$ . We score the list by its expected utility  $R$  of  $a$ .

Finally, we average R of all test users.



**UiT** The Arctic University of Norway

## DTE-2501 AI Methods and Applications

*Using the Pearson Correlation for Collaborative Filtering*

Andreas Dyrøy Jansson  
PhD Candidate  
C3190  
[andreas.d.jansson@uit.no](mailto:andreas.d.jansson@uit.no)

In this lecture, we will go through an example where we calculate the Pearson correlation between two users and use this to make a simple recommendation engine.

# The basics of the Pearson correlation

- Pearson correlation coefficient
  - Tells us something about the relationship between two datasets

$$w(a, i) = \frac{\sum_j (v_{a,j} - \bar{v}_a)(v_{i,j} - \bar{v}_i)}{\sqrt{\sum_j (v_{a,j} - \bar{v}_a)^2 \sum_j (v_{i,j} - \bar{v}_i)^2}}$$

- Gives a number between 1 and -1
  - 1 means we have a strong correlation
  - 0 means no correlation
  - -1 means a strong inverse correlation
- Takes the variations ( $v_{i,j}$  - average  $v$ ) as "argument"

Pearson correlation is a number between -1 and 1 which tells us if there is a correlation between two sets of data. 1 means that there is a strong correlation, 0 means no correlation and -1 is a strong inverse correlation. This means that we can use this to see if there is some sort of relationship between the users, which in turn can be used as a basis for a recommendation system.

## An example: product reviews

Index (j)	Product	User 1	User 2	User 3	User 4	User 5	User 6	User 7
0	Crazy Cola	1		3	4	1	4	5
1	Zombo Soda	1	3	4	5	3	3	5
2	Ol Geezer's				2	3	3	
3	Dr. Hiccups			4	5	2	4	3
4	Fantom	2		5	3	2	5	4
5	Sprizz	1	5	5	2	3	2	
6	Chimpus			1	2	4	3	
7	TopChug				3	1	3	

- These are the voting vectors (profiles) of each user

Let's say that we collect some product review data for a set of users. Each user has voted on the products and given them a score from 1 to 5, where 1 is bad and 5 is good. We call this the voting vectors, or profiles, of each user. Notice that not all users have voted on every product.

# Recommendation system

- Consider that we get a new user
  - We will refer to them as *a* (active user)
- We observe their votes as such:
- Problem: How can we predict their opinion on "Dr. Hiccups"?

Product	Active user
Crazy Cola	3
Zombo Soda	2
Ol Geezer's	3
Dr. Hiccups	
Fantom	4
Sprizz	1
Chimpus	
TopChug	5

Let's look at a new user with their own voting profile. Let's imagine that we are doing market research for a company. How can we predict what this user's opinion will be on the product Dr. Hiccups? Recall the lecture about the wisdom of crowds. If we can somehow find a way to group users together based on their profiles, we can use collaborative filtering to predict a new user's vote based on the history of similar users. Going forward, we will refer to this new user as active user, or simply *a*.

## Memory-based algorithms for Collaborative Filtering (Breese et al, UAI98):

- Predicted vote for “active user”  $a$  is weighted sum

$$p_{a,j} = \bar{v}_a + \kappa \sum_{i=1}^n w(a, i) (v_{i,j} - \bar{v}_i)$$

normalizer              Pearson correlation coefficient of  $a$  and  $i$

- $v_{i,j}$  = vote of user  $i$  on item  $j$  (product)
- $I_i$  = items for which user  $i$  has voted
- Mean vote for  $i$  is

$$\bar{v}_i = \frac{1}{|I_i|} \sum_{j \in I_i} v_{i,j}$$

We can predict the vote for a product for the active user by using a weighted sum. What we have here is that the predicted vote for product  $j$  by the active user  $a$  is equal to the average vote of the  $a$  plus a normalizer multiplied by the sum of all correlations between the active user and the other users multiplied by the variation for product  $j$  by each user.

This might seem a bit overwhelming, but we will step through each part of this formula in the next few slides, along with some code examples.

## Step 1 – find the mean votes of each user

- Variation:  $(v_{i,j} - \bar{v}_i)$
- We start by finding the mean vote for each person using the formula:  
$$\bar{v}_i = \frac{1}{|I_i|} \sum_{j \in I_i} v_{i,j}$$
- The formula decoded (User 1):
  - $(1 + 1 + 2 + 1) / 4 = 1,25$

The first step is to find the average vote of each user. Let's look at user 1 for example. Their votes are 1, 1, 2, and 1. We simply sum them together and divide by 4 to get the mean vote. If we do the same for the rest of our users, we get the following table.

## Step 1 – find the mean votes of each user

- Variation:  $(v_{i,j} - \bar{v}_i)$
- We start by finding the mean vote for each person using the formula:  
$$\bar{v}_i = \frac{1}{|I_i|} \sum_{j \in I_i} v_{i,j}$$
- The formula decoded (User 1):
  - $(1 + 1 + 2 + 1) / 4 = 1,25$

User 1	User 2	User 3	User 4	User 5	User 6	User 7	Active user
1,25	4	3,666667	3,25	2,375	3,375	4,25	3

## Step 2 – find the variation for each vote

- The variation is the difference between each vote and the user's average

$$(v_{i,j} - \bar{v}_i)$$

- We take the product vote and subtract the mean vote
  - Example - User 1:

Next, we find the variance using this formula. This is simply the vote for product j minus the average vote for the user. If we look at user 1 again, computing the variance gives us the following table.

## Step 2 – find the variation for each vote

- The variation is the difference between each vote and the user's average

$$(v_{i,j} - \bar{v}_i)$$

- We take the product vote and subtract the mean vote
  - Example - User 1:

Product	User 1
Crazy Cola	$1 - 1,25 = -0,25$
Zombo Soda	$1 - 1,25 = -0,25$
Ol Geezer's	-
Dr. Hiccups	-
Fantom	$2 - 1,25 = 0,75$
Sprizz	$1 - 1,25 = -0,25$
Chimpus	-
TopChug	-

## Step 3 – Compute the correlation matrix

- Using `corrcoef` from Numpy
- Example: find the correlation matrix for two users
- Ignore missing values

Now we move on to the Pearson correlation. Since we are using Python, we can use a function from Numpy to compute the correlation matrix between our users. For this first example, we will simply look at how to compute the correlation between two of the users. In order to use the Numpy function, each variance vector must be the same size. We see that we have some missing values. We ignore these products, and end up with the following table:

## Step 3 – Compute the correlation matrix

- Using `corrcoef` from Numpy
- Example: find the correlation matrix for two users
- Ignore missing values

Product	User 1	Active user
Crazy Cola	-0,25	0
Zombo Soda	-0,25	-1
Fantom	0,75	1
Sprizz	-0,25	-2

Now we move on to the Pearson correlation. Since we are using Python, we can use a function from Numpy to compute the correlation matrix between our users. For this first example, we will simply look at how to compute the correlation between two of the users. In order to use the Numpy function, each variance vector must be the same size. We see that we have some missing values. We ignore these products, and end up with the following table:

## Step 3 – Compute the correlation matrix

- Using Python (Numpy):
  - Put the variances for all users into arrays:

```
import numpy as np
###
user_1_variance = np.array([-0.25, -0.25, 0.75, -0.25])
active_user_variance = np.array([0, -1, 1, -2])

corr_matrix = np.corrcoef(user_1_variance, active_user_variance)
```

Product	User 1	Active user
Crazy Cola	-0,25	0
Zombo Soda	-0,25	-1
Fantom	0,75	1
Sprizz	-0,25	-2

We put the values from the table into two Numpy-arrays like so. We call the function, and get the correlation matrix.

## Step 3 – Compute the correlation matrix

- Using Python (Numpy):

- Put the variances for all users into arrays:

```
import numpy as np
##
user_1_variance = np.array([-0.25, -0.25, 0.75, -0.25])
active_user_variance = np.array([0, -1, 1, -2])

corr_matrix = np.corrcoef(user_1_variance, active_user_variance)
```

- This gives us the correlation matrix:

- About the correlation matrix

- Symmetric
    - Diagonal 1's is the self-correlation
    - The other values is the actual correlation value with the other user
      - In this case, the value is 0.77, which indicates a strong correlation between these users
    - We keep the value from [0][1]

Product	User 1	Active user
Crazy Cola	-0,25	0
Zombo Soda	-0,25	-1
Fantom	0,75	1
Sprizz	-0,25	-2

```
[[1.          0.77459667]
 [0.77459667 1.        ]]
```

Press any key to continue

On the diagonal, we have User 1 correlated with User 1 and Active user correlated with Active user. We see that these values are 1, which is expected, since this is the self-correlation. This is not interesting to us however, so we keep the values in  $[0][1] = 0.77459667$ , which is the actual correlation between these two users. This is close to 1, which tells us that there is a somewhat strong correlation between these two users.

## Step 4.5 – set up the vectors for all users

- We must compute the correlation  $r$  between Active User  $a$  and all the other users

```
# pad the vectors with a token so they are of equal length
user_1_variance = [-0.25, -0.25, 'x', 'x', 0.75, -0.25, 'x', 'x']
# other user vectors go here...

active_user_variance = [0, -1, 0, 'x', 1, -2, 'x', 2]

# put all the users' vectors in an array so we can loop over them
all_user_variances = [user_1_variance, user_2_variance, user_3_variance,\nuser_4_variance, user_5_variance, user_6_variance, user_7_variance]
```

We must do this for every user 1 to 7 and the active user. To make things easier, we will use arrays and loops. Since most of the users' vectors are incomplete, we will pad them with a placeholder value  $x$  so that they all are the same length. When we compute the correlation in the next step, we check if this value is present in the vector. If it is, we simply skip that product. We put the vectors of user 1 to 7 in an array so we can loop over them in the next step.

## Step 5 - Set up the for-loops to compute the correlations

```
#we also need to save the results for later. we keep them in this array
correlations = []

# set up vectors of equal size. we ignore the products that have not been voted on, i.e where the variance is 'x'
for other_user_variance in all_user_variances:
    # we create two temporary arrays to keep our variances
    temp_active = []
    temp_other = []
    # we then loop over all the values in both users' vectors
    for i in range(len(active_user_variance)):
        # we can only use the values for products that the user has voted on. We therefore ignore the x's
        if active_user_variance[i] != 'x' and other_user_variance[i] != 'x':
            # we put the values that are not 'x' in our temp arrays
            temp_active.append(active_user_variance[i])
            temp_other.append(other_user_variance[i])

    # now that we have set up vectors of equal length, we can compute the correlation using numpy
    correlation_matrix = np.corrcoef(np.array(temp_active), np.array(temp_other))
    correlation = correlation_matrix[0][1]
    # keep the result in an array
    correlations.append(correlation)
```

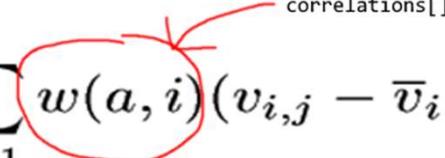
In the first for-loop, we iterate over all the users' variance arrays. We let the code do the work of creating the actual arrays we use for computing the correlations. Using two temporary arrays, we only keep the variances for the products that are not x for both of the users. When we have done this for all variances of one user and the active user, we pass these arrays to the Pearson correlation function of Numpy, like we saw in the first code example. The only difference this time is that the arrays are created dynamically, and their lengths change based

on the variances we ignore for each user.

Remember that we are not interested in the self-correlation, so we only keep the value from [0][1] in the matrix. We store this value in an array called correlations.

## Step 6 – Compute the predicted vote

- Recall the formula:
  - $p_{a,j}$  is the predicted vote

$$p_{a,j} = \bar{v}_a + \kappa \sum_{i=1}^n w(a, i)(v_{i,j} - \bar{v}_i)$$


- Simplified:
  - $p_{a,j} = \text{average\_vote\_for\_a} + \kappa * (\text{sum\_correlation\_variance})$
  - We set  $K = 1$

After we have computed all the correlations, we can take a step back and take another look at the formula for predicting the vote. As before, the predicted vote for product j by the active user is equal to the average vote of the active user + kappa multiplied by the sum of all correlations between active user and the other users multiplied by the user's variance for product j. We'll set kappa to 1.

## Sum correlation \* variance

- Finally, we can compute the predicted vote for the product:
  - Remember, we skip the products that users have not voted on (x)

```
kappa = 1
average_vote_for_a = 3
sum_correlation_variance = 0
j_index = 3 # the index of the product we want to predict the vote for (Dr.Hiccups)
for i in range(len(all_user_variances)):
    if all_user_variances[i][j_index] != 'x':
        sum_correlation_variance += all_user_variances[i][j_index] * correlations[i]

p_a_j = average_vote_for_a + kappa * (sum_correlation_variance)
```

- We find that predicted vote for j=3 is 4.6
  - Conclusion: Active user is likely to give this product a high score

This is easier to show in code. If we write the formula like so, we can compute the sum using a for-loop. Kappa is 1, the average vote for active user we found earlier, and the product we want to predict the vote for has index 3 in the list. We then simply multiply and add together all the correlations and variances that are not x, and put it into the formula like so. The predicted vote for Active User thus becomes their average vote plus 1 multiplied by the sum of correlation variances. When running this calculation on the data presented in the start, we get 4.6 as result for the product Dr.Hiccups. We can

therefore conclude that the user is likely to give this product a high score, probably a 4 or a 5.



**UiT** The Arctic University of Norway

## DTE-2501 AI Methods and Applications

*Lecture 2/2 – Ensemble methods*

Andreas Dyroy Jansson  
*PhD Candidate*  
C3190  
[andreas.d.jansson@uit.no](mailto:andreas.d.jansson@uit.no)

This is lecture 2 of 2 in a series on collaborative filtering and ensemble methods

## What is an ensemble

- A suite of simple methods
  - “Rules of thumb”
  - Easier than finding a single advanced algorithm
  - The power of many
  - Remember, collective group intelligence is normally higher than the best individual
- Machine learning algorithms that construct a set of classifiers
  - Classification/prediction based on voting

A suite of simple methods makes up an ensemble.

Basically, finding simple methods or rules of thumb can be a lot easier than finding a single, advanced algorithm for accurate predictions and classifications. Remember the previous lecture, where we looked at the wisdom of crowds, and the power of many. We saw that in many cases, the combined intelligence of a group is higher than the best individual. The same principle applies here, as ensembles are machine learning algorithms that construct a set of classifiers, and their individual predictions are combined through a voting system.

## Multiple methods

- Bayesian averaging
- Bagging
- Random forest
- Boosting
  - Gradient boosting
  - AdaBoost

There are many ways to create an ensemble, and we will take a look at some of these in the next slides. Some common methods include Bayesian averaging, bagging, random forest, and boosting.

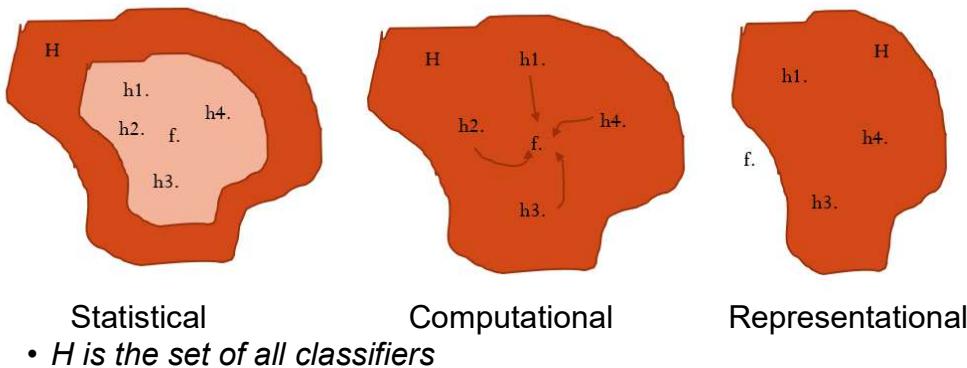
## Repetition: classifiers and learners

- Perceptrons
- K-classifiers
- Decision trees
- Regressions
- Simple Bayesian methods
- ++

Now for a quick repetition on classifiers and learners. You remember from previous that we have classes like perceptrons, K-classifiers, decision trees, regressions and simple Bayesian, probability based methods. These are what we call individual classifiers.

## Strengths of an ensemble

- Why an ensemble may work better than a single classifier



Just as with people, combining individual classifiers into an ensemble can be beneficial in three ways: Statistical, computational, and representational. Statistically speaking, some classifiers are more likely to give better predictions from the start by random chance, and on average, the combined output will be closer to the actual value. Similarly, dividing the load on multiple simple classifiers can speed up execution time, compared to one big, heavy algorithm. It is also easier to represent complex problems if we split them up into smaller parts for each classifier in our ensemble.

## The origin of ensemble methods

- Roots in crowdsourcing
  - Committees, majority vote and unweighted averages
- Simple ensembles
  - Simple average
  - Simple combination of learners/classifiers
- Example: Committee approach:

$$\hat{y} = \frac{1}{N_h} \sum_{h_i} \sigma_i * h_i(x), \sigma_i \in [0,1]$$

The concept of ensemble methods has its origin in crowdsourcing. For example, in the real world, we have committees, majority voting and unweighted averages. Like the example from the previous lecture, when guessing an unknown number, we assume that every individual has the same probability of guessing the correct answer.

It is possible to simply take the average prediction of every classifier, or do some other basic combination. For example, using a committee approach, we simply sum up every prediction and take its average. As we have

seen, this usually leads to higher accuracy, but we can do even better. With classifiers in ensembles, we usually have some idea of the performance of each model.

More about this later.

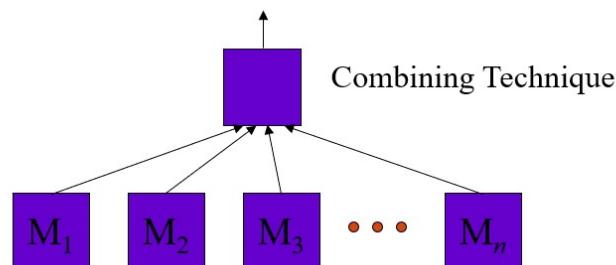
## Ensembles using simple statistics

- Multiple diverse models trained on the same problem
  - Outputs are combined to form the final output
  - Overfit can be average out
  - Diverse models can be accurate even if the individuals are weak generalizers

Another benefit of splitting the dataset and combining classifier output is that we can have multiple diverse models trained on the same problem. Their outputs are combined to come up with a final output. This is good because the specific overfit of each learning model can be averaged out. Also, if models are diverse (meaning that we assume uncorrelated errors) then even if the individual models are weak generalizers, the ensemble can be very accurate

## Combining outputs

- Many different Ensemble approaches:
  - Stacking, Gating/Mixture of Experts, Bagging, Boosting, Wagging, Mimicking, Heuristic Weighted Voting, Combinations



As we saw earlier, we have many different ensemble approaches, or ways of combining outputs.

## Bootstrap Aggregation (Bagging)

- A way to improve machine learning by splitting the data set
  - improves overall accuracy by decreasing variance
- Often used with the same learning algorithm
  - Best for diverse hypotheses based on initial conditions
- A set of  $m$  learners
  - Same initial parameters
  - Each training set chosen uniformly at random (2/3)
- Does not overfit (compared to boosting)
  - May be more conservative on accuracy improvements

Let's start by taking a look at Bootstrap Aggregation, or Bagging. Basically what we do, is split the dataset in order to improve the accuracy of a machine learning technique. This is achieved due to decreased variance. This approach is often used with the same learning algorithm and thus best for those which tend to give more diverse hypotheses based on initial conditions. This is done by inducing a set of  $M$  learners starting with the same initial parameters with each training set chosen uniformly at random with replacement from the original data set. These training sets might be 2/3rds of the

dataset, as we still need to save some separate data for testing. We also give all  $M$  hypotheses an equal vote for classifying novel instances. As we have previously discussed, this results in consistent empirical improvement.

One major advantage of bagging compared to boosting is that it does not overfit. However, bagging may also be more conservative overall on accuracy improvements.

It is also possible to use other schemes to improve diversity between learners, like different initial parameters, sampling approaches etc. We could even use different learning algorithms for each classifier. In short – the more diversity the better. Most often bagging is simply implemented with the same learning algorithm and just different training sets.

## More advantages of bagging

- Bias
  - Error due to model choice
  - We only see «part of the world»
- Variance
  - Randomness due to data size
- Bias and Variance are expectations:
  - Averaging helps reduce outliers

Further advantages of bagging, and ensemble methods in general, are that we reduce bias. Bias is usually caused due to model choice, and hidden data. In short, we only see part of the whole image. An ensemble of classifiers trained on a diverse dataset are therefore less likely to have bias. Similarly, if each classifier has a high variance (unstable) the aggregated classifier has a smaller variance than each single classifier. The bagging classifier is like an approximation of the true average computed by replacing the probability distribution with bootstrap approximation.

## Boosting

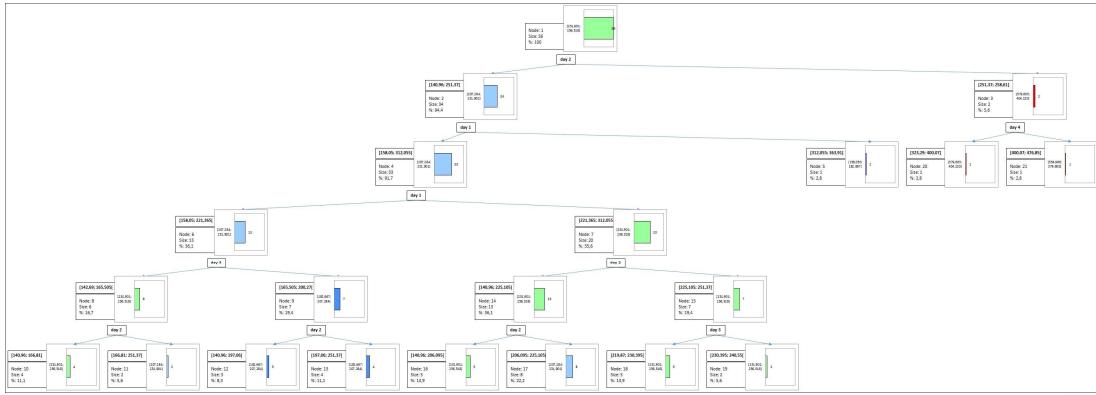
- All models have a weighted vote
  - Vote is scaled based on the model's accuracy
- More «aggressive» than bagging
  - Better than bagging on average
  - May overfit and do worse
    - Could theoretically converge to the training set
  - May be worse than non-ensemble in rare cases
- Many variations
  - Gradient boost, boosted regression tree, AdaBoost etc.

Finally, we will take a look at boosting. As with bagging, all models have a vote, but we don't simply take the average. As I mentioned earlier, we do have some idea of the performance of each individual classifier in our ensemble. We can use this in order to scale each model's vote based on its accuracy on the training set it was trained on. This means that boosting is a more aggressive approach than bagging, and may also be prone to overfitting and worse performance in some cases. In theory, using boosting may even make the ensemble converge to the training set. This may even

lead to worse performance than non-ensemble methods in some rare cases. However, on average, boosting is still better than bagging for most problems.

There are many variations, some examples include gradient boost, boosted regression trees, and AdaBoost.

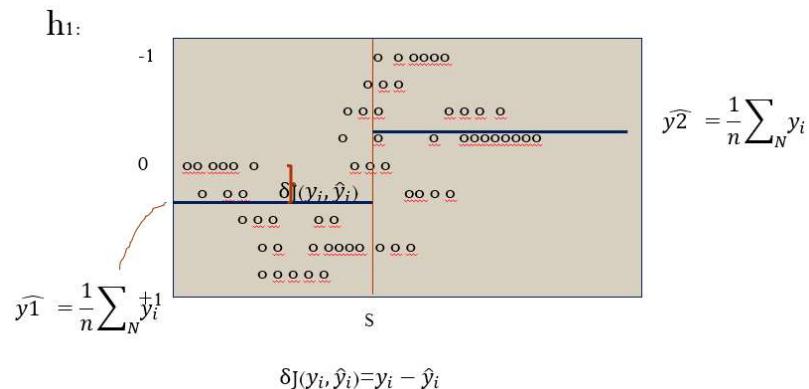
# Regression tree



To close off, we will take a quick look at a boosted regression tree. In order to create an accurate prediction, a single tree can become quite unmanageable, like this example for predicting electricity prices:

## Example: Boosted regression tree

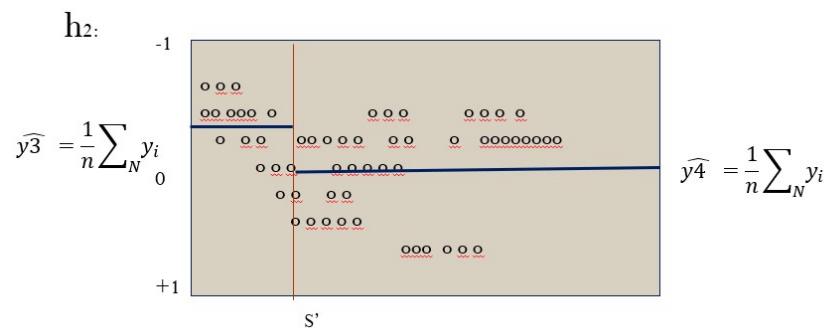
- The first simple learner  $h_1$ :



To simply show the concept, let's say we have some points we want to classify. We create a simple learner called  $h_1$  with two classes:  $y_1$  and  $y_2$  divided by the point  $S$ .

## Example: Boosted regression tree

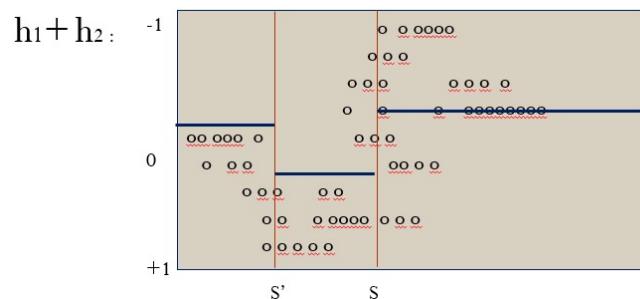
- The second simple learner  $h_2$ :



We do the same with another tree  $h_2$ , but we choose a different split point  $S'$  as so:

## Simple learners combined

- We see that combining the trees gives a more accurate classifier



Finally, we combine the outputs from the two trees, and we get the following: Each tree is still a simple yes/no, but with their outputs combined we get a more accurate approximation of our points.



**UiT** The Arctic University of Norway

## DTE-2501 AI Methods and Applications

*Setting up a simple ensemble using Bootstrap Aggregation*

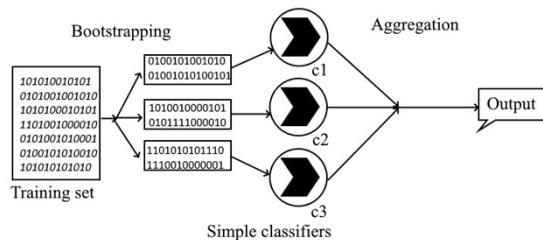
Andreas Dyroy Jansson  
PhD Candidate  
C3190  
[andreas.d.jansson@uit.no](mailto:andreas.d.jansson@uit.no)

In this lecture we will take a closer look on how to create an ensemble using bagging and simple learners.

# Bagging and ensembles revisited

- Remember:

- An ensemble is a combination of simple methods
- Output of each model is combined to form the final output
- Bootstrap aggregation (bagging)
  - Split the training data into smaller sets (Bootstrapping)
  - Train each classifier on one of the sets
  - Combine the output of each classifier (Aggregation)

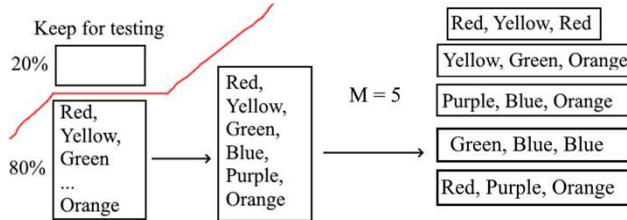


Remember from Lecture 2 – an ensemble is a combination of simple classifiers, where the outputs of each classifier are combined in some way to form the actual output. The process we will look at in this lecture is bagging, or bootstrap aggregation.

The bagging process can be summarized as follows: Start by splitting the training set into smaller sets, based on random samples. Then we train each classifier on one of the smaller datasets. When we test the ensemble with a new datapoint, we combine, or aggregate, the output of each model to form the actual output.

# Bootstrapping

- Split the data into a training set and test set
  - 70-90% for training, the rest for testing and validation
- Split the remaining training set into smaller «bootstrapped» sets
  - We pick a random number of samples from the training set, with replacement
    - This implies that a sample may be picked multiple times for each bootstrapped set



First, we start by splitting our data into a training set and a testing set. A good rule of thumb is to use between 70 and 90 % as the training set, and the rest for testing.

After we have split our data, we do the bootstrap aggregation, or bagging. As mentioned in Lecture 2, this is done by inducing a set of  $M$  learners starting with the same initial parameters with each training set chosen uniformly at random with replacement from the original training set.

What this means is that we have some data, and we decide a number  $M$  of learners to create. What this

means is that we have some data, and we decide a number  $M$  of learners to create. Let's say that  $M$  is 5.. We then generate 5 unique datasets by selecting random samples from the training set. This number should be smaller than the total size of the data, say 60%. The term “with replacement” means that we do not “remove” the randomly selected sample from the possible candidates. This means that the same sample may be chosen multiple times, or not at all for some of the bootstrapped sets.

## Bootstrapping pseudocode

- `all_data = read_file("datafile.csv")`
- **FOR EACH** sample in `all_data`:
  - IF `random(0, 1] < 0.2` //we split the data 20/80 between test and train
    - THEN put sample into the test data array
  - **ELSE**
    - put sample into the training data array
- create a set of  $m$  classifiers/models
- **FOR EACH** classifier  $c_i$ 
  - `bootstrap_data_ci = pick  $n'$  random samples from training data`
  - ( $n'$  is constant and the same for all bootstrapped sets)

The bootstrapping process may be summarized in the following psuedocode:

Read all the data into an array

Split the data into training set and testing set. In this example, we simply generate a random number between 0 and 1. Given that the distribution is uniform, we will end up with 20% of the data in a test set, and 80% in the training set.

Create a set of  $m$  simple classifiers

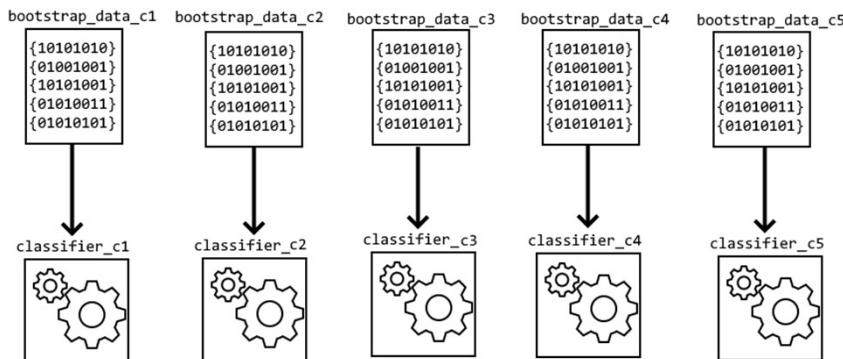
For each classifier, pick  $n'$  random samples from the

training set.

Each classifier needs to have their own bootstrapped training set.

## Training the ensemble

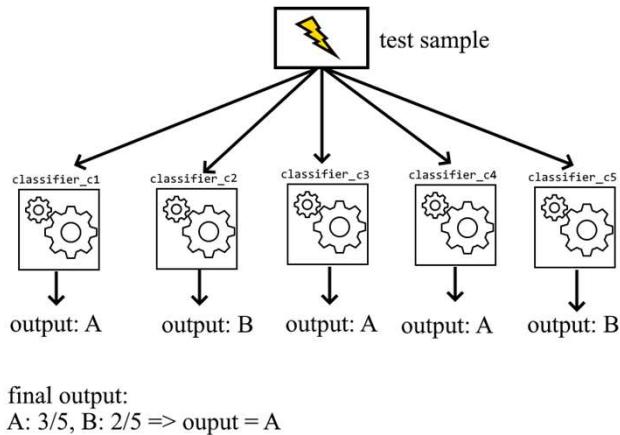
- We now have a set of  $m$  classifiers
- Each classifier is trained on its own bootstrapped training set



The next step is to train each simple classifier on their own bootstrapped dataset. The actual implementation and methods used for the classifier is not important in this example, but this can be something like a K-NN, Naïve Bayes or other.

## Combining the outputs - aggregation

- We must combine the outputs of all the classifiers to get a prediction
  - Average value
  - Majority vote



The aggregation part of bagging comes into play when we combine, or aggregate, the outputs of each trained classifier in order to make a prediction. The simplest way is to use the average value for regressions, or the majority vote for classification problems. We test all 5 classifiers on the same sample. Let's say that 3 of our fictitious classifiers predict the class A, and the other predict class B. If we use the majority vote principle, the final output of the ensemble will thus be A.



**UiT** The Arctic University of Norway

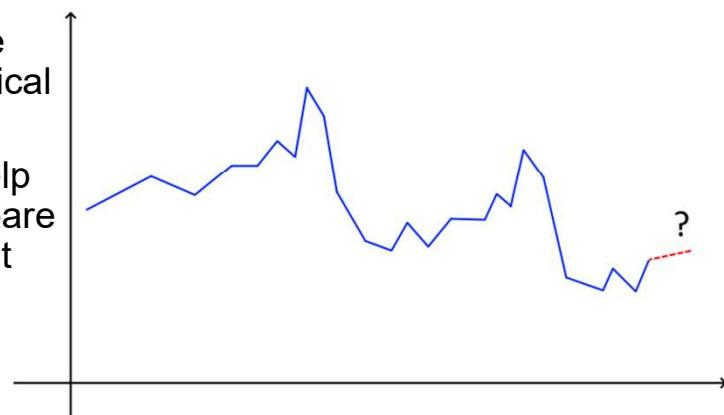
## DTE-2501 AI Methods and Applications

*Creating a dataset for Time Series Prediction using Sliding Window*

Andreas Dyroy Jansson  
*PhD Candidate*  
C3190  
[andreas.d.jansson@uit.no](mailto:andreas.d.jansson@uit.no)

## Time series prediction – the short version

- We want to predict the future based on historical data
- AI (regression) can help us – but we must prepare the data so that we get an “input->output” dataset
  - One approach is the “sliding window”

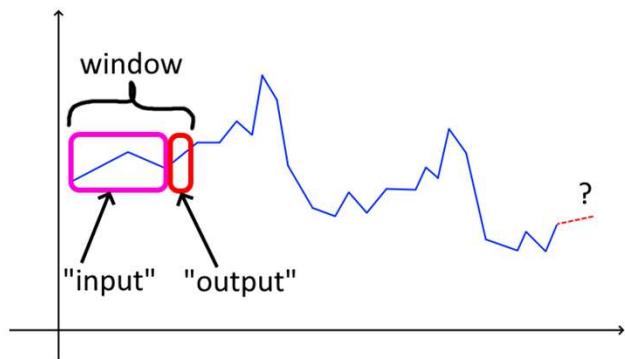


Time series prediction is simply that we want to predict some future state based on historical records. We can do this using regressions, which you may remember from mathematics. You know that we mainly have two types of models in AI and machine learning: classification and regression. For classification, the model is trained on some data, and it predicts a discrete value, corresponding to a class. Regression models are basically the same, but instead they output continuous values. In order to train any machine learning model, we must feed it data on the correct format, for supervised learning models, this means a set of input values  $X$ , and the

expected output Y associated with each X value. For time series, which is basically a series of observations, we can use a technique called sliding window, which we will take a closer look at now.

## The sliding window concept

- What the name implies – we “slide” a “window” across our data
- A range of data points becomes the input, X, and the data point immediately after becomes the expected output, Y
- Our supervised model can then be trained on the X-Y dataset



The name sliding window comes from the fact that we basically create a window in time, and slide this window across our time series data set. The window has a certain length in time, and this range of data points becomes the input. The trick here is to make the next data point immediately after the points captured by the window, the expected output Y. We can then train our model on the X-Y pairs of data.

## Sliding window algorithm

*Given a series of observations S on the form  $[x_1, x_2, x_3, x_4, \dots, x_n]$*

*$X, Y = []$*

*window size W is some small number  $< n$*

*FOR  $i < n - W$  DO*

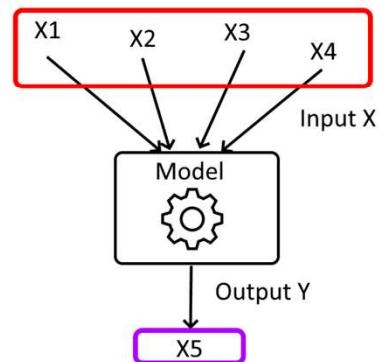
*$X.append(S[i:i + w])$*

*$Y.append(S[i + W])$*

The basic algorithm for a sliding window parser looks like this. Let's say that we have a series of observations like  $x_1, x_2, x_3, x_4, \dots, x_n$ . We start by initializing two empty lists  $X$  and  $Y$ , which will hold our transformed values. We pick the size of the window  $W$ , which is a small number. Then, for  $i$  less than  $n$  minus  $W$ , we simply append the  $i$  to  $i + w$  datapoints to the list  $X$ , and the  $i + w$ th data point to the list  $Y$ . We keep going until we reach the end, and the very last data point becomes the last expected output.

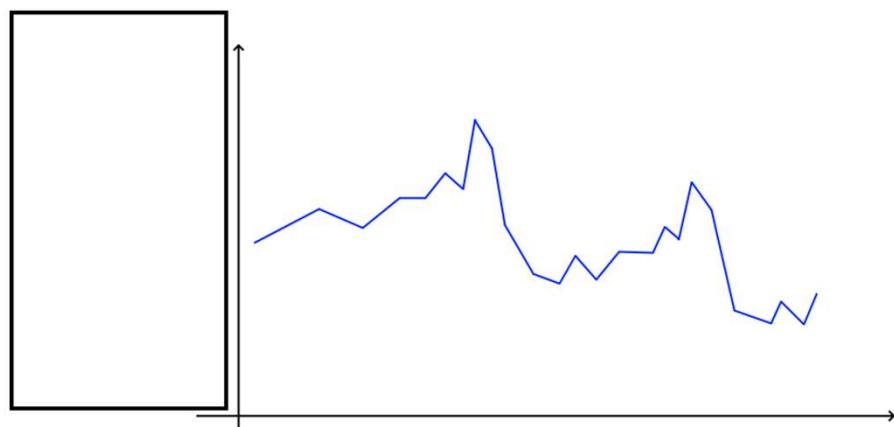
## The basics

- For example, let's choose a window size of 4
- Then, the first 4 data points become the input X, and the 5<sup>th</sup> is the expected output Y
- We move the “window” one step ahead, and repeat
  - X= X2, X3, X4, X5
  - Y = X6
  - Etc.



To show the basic principle in practice, let's say that we choose a window size of 4. Then, the first 4 data points in the time series become the input X, and the 5<sup>th</sup> data point is the expected output, based on the 4 preceding points. We move the window one step ahead, and repeat the procedure, so that the next pair of input and output values becomes X equals x2, x3, x4, x5, and x6 as Y.

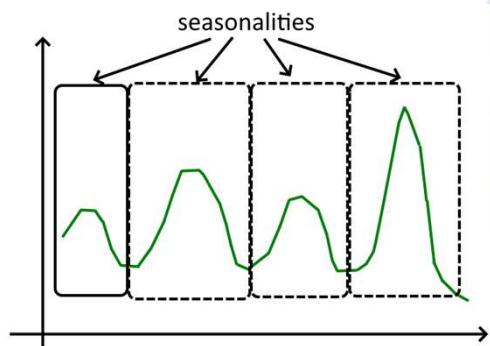
In practice:



This animation shows how the window moves across the time series, and puts each x and y value in our training set.

## How to choose window size?

- Plot the data using your favorite data plotter
- If you notice seasonality of a certain length, this is a good place to start
- A smaller window is better at generalizing, but may miss out on long-term trends
- A larger window can capture trends better, but may cause the model to overfit



Our next question is how do we determine the window size? One simple way is to examine the data visually. Plot the data using your favorite plotter, and look for any repeating patterns or seasonality. If you notice seasonality of a certain length, this is a good initial value for  $W$  to start with. Fine tuning can be done by testing the model on different window lengths until you achieve satisfactory performance. In general, we can say that a smaller window is better at generalizing, but may fail to capture longer trends in the data. Likewise, a larger window is more likely to capture long-term trends, but is also be more likely to overfit.

- Using our data set, we can train a simple learner
  - For example, a regression tree:



```

from sklearn.tree import DecisionTreeRegressor # pip install scikit-learn

# window size is a small number, depending on the length
# of our time series, and if there are any seasonalities
dataset = [2.1, 1.3, 1.2, 1.7, 0.9, 0.8, 1.8]

def make_dataset(window_size = 4):
    x, y = [], []
    # read and format the data using the sliding window algorithm...
    # the result should be:
    # x = [[2.1, 1.3, 1.2, 1.7], [1.3, 1.2, 1.7, 0.9], [1.2, 1.7, 0.9, 0.8]]
    # y = [0.9, 0.8, 1.8]
    return x, y

X, Y = make_dataset() # make the dataset using sliding window
model = DecisionTreeRegressor(random_state = 0) # create the model
model.fit(X, Y) # train the model on the input X and the expected output Y
prediction = model.predict([[1.7, 0.9, 0.8, 1.8]]) # predict the next value based on a given input

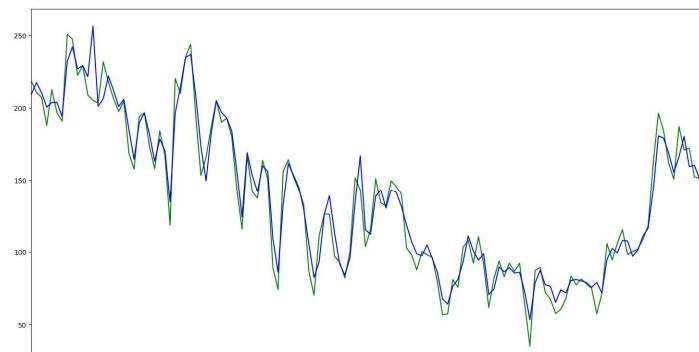
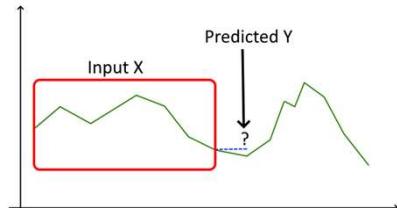
```

When we have parsed the data file and created a test set, we can train the model. For example, we can train a simple regression tree, like so. Let's say that we have the following data: 2.1, 1.3, 1.2, 1.7, 0.9, 0.8, and 1.8. We see from the plot that there may be seasonality of length 4, so we pick 4 as the window size. We then slide the window across the data, and end up with these pairs of X and Y data. We use the regression tree from scikit, and we set the random state to 0. This is just a seed for the internal random generator, which means that we should get the same result every time we train the model.

We train the model using the fit() function, and pass in the data we just formatted. Then, we can pass in the last four data points, and predict the future value.

## Plot the data

- To test our model, we can plot its predictions alongside the original data
- In this example, the original data is in green, and the prediction is blue



To show the accuracy of our model visually, we can plot the predicted values alongside the original data. The first data points are the input, and the model tries to predict the next value based on these points. Here we have a much longer time series, and the original data is shown in green. The blue line shows each predicted output of a regression model, based on the previous data points, repeated over the whole original data set.

## Plotting the predictions...

```
import matplotlib.pyplot as plt
predictions = []
for i in range(len(original_data) - window_size):
    # predict the next value of original_data using the same sliding window approach
    predictions.append(model.predict([original_data[i]]))

plt.plot(data[window_size:], color="green") # skip the first data points in the original data
plt.plot(predictions, color= "blue")
plt.show()
```

- Note: When we plot the predictions alongside the original data, we must offset the original data by “window\_size”!
  - Remember, we need at least “window\_size” number of data points in order to predict the next value

To plot the predictions in Python, we can use matplotlib. Here is an example of how this can be done. Here we use the same sliding window approach on our test set, and make a prediction. We save each prediction in a list, and plot it alongside the original data.

Remember that in order for both graphs to line up, we must offset the original data by the same length as the window size. This is because we needed window size number of data points to predict the next value, so the first value in our predicted time series may be the 4<sup>th</sup> or 7<sup>th</sup> element in the original data set.