

# 1. Introduzione

## 1.1. Prefazione

Questi appunti si rifanno alle lezioni 2023/2024 del corso Introduction to Machine Learning tenuto dalla docente Elisa Ricci, al libro 'Deep Learning' di Ian Goodfellow e Yoshua Bengio; ed infine al libro 'Hands on machine learning' di Aurélien Géron pubblicato da O'Reilly.

Gli appunti sono scritti con `typst`, senza una panoramica sui diversi argomenti, ma affrontandoli uno ad uno a seconda della necessità. All'interno di questa introduzione troverete solo i concetti basilari, utili alla comprensione dei successivi argomenti.

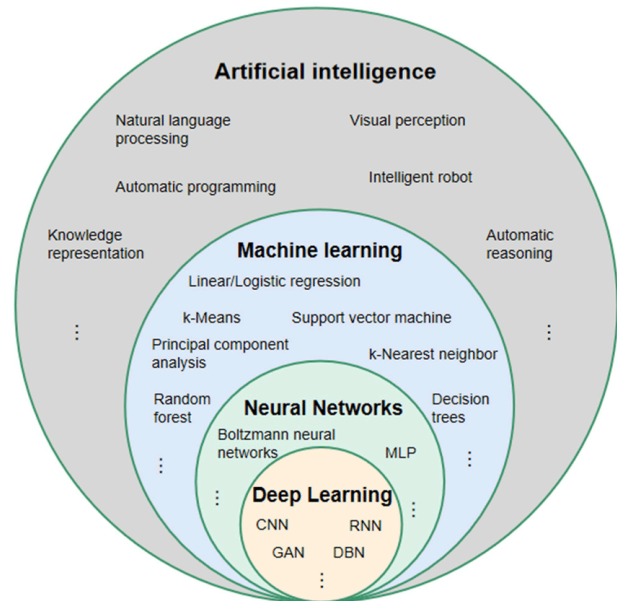


Figure 1: La relazione tra intelligenza artificiale, machine learning e deep learning.

## 1.2. Dataset

Il dataset è l'insieme dei dati disponibili per l'analisi. Su questo dataset si effettuano le operazioni di training e testing.

Il training set è il sottoinsieme del dataset utilizzato per addestrare il modello; mentre il test set è il sottoinsieme utilizzato per testare il modello. Il validation set è un sottoinsieme del training set utilizzato per regolare gli iperparametri del modello, prima della fase di testing.

Per generare questi sottoinsiemi è necessario fare due assunzioni sui dati (*i.i.d. assumption*), ovvero che siano:

- **indipendenti** (non ci sia correlazione tra i dati del training set e del test set)
- **identicamente distribuiti** (prelevati dalla stessa distribuzione di probabilità  $p_{data}$ )

## 1.3. Modello

L'obiettivo, nel Machine Learning, è che il nostro modello performi bene su dati che non ha mai visto prima; questa abilità è detta *generalizzazione*. Ogni modello ha le sue peculiarità, e la scelta del modello giusto dipende dal problema (*task*) che si vuole risolvere. I modelli possono essere divisi in categorie, anche se con eccezioni e sfumature, a seconda del tipo di apprendimento:

- **Supervised Learning**: il modello apprende da un training set etichettato precedentemente.
- **Unsupervised Learning**: il modello apprende pattern o strutture dai dati senza etichette.
- **Reinforcement Learning**: il modello apprende attraverso il feedback di un ambiente.
- **Semi-Supervised Learning**: il modello apprende sia da dati etichettati che non etichettati. Viene utilizzato in sostituzione al supervised learning nei casi in cui etichettare i dati risulti troppo costoso o, richieda troppo tempo.

## 1.4. Tasks

Le principali task per cui viene adottato il Machine Learning sono:

- **Classification**: classificare un input in una delle classi predefinite.
- **Regression**: predire un valore numerico (continuo), dato un input.
- **Transcription**: convertire un input in testo. L'input può essere un'immagine, un audio, ecc.
- **Machine Translation**: tradurre un testo in un'altra lingua.
- **Anomaly Detection**: identificare pattern anomali nei dati.

- Synthesis: generare nuovi dati che seguano la stessa distribuzione dei dati originali. (e.g. textures, speech, ecc.)
- Denoising: in questo task il modello, ha come input un dato corrotto  $\tilde{x}$  e deve predire il dato originale  $x$ ; o meglio la distribuzione di probabilità  $p(x|\tilde{x})$ .
- Density Estimation:

## 1.5. Errors

Solitamente, quando si allena un modello si effettuano delle misure sull'errore. Questo errore viene chiamato **training error**; e durante la fase di training si mira a ridurlo.

Ciò che separa il Machine Learning da un semplice problema di ottimizzazione è che oltre a minimizzare il **training error** l'obiettivo è di minimizzare anche il **generalization error** (anche noto come **test error**). Questo errore viene misurato in fase di test ed è definito, grosso modo, come: *il valore dell'errore atteso su un nuovo input*.

Proprio per questo le assunzioni fatte sui dati nella Sezione 1.2 sono necessarie.

## 2. Regressione Lineare

Come suggerisce il nome, la regressione lineare è un modello che risolve un problema di regressione, ovvero dato un vettore  $x \in \mathbb{R}^n$  in input, restituisce un valore  $y \in \mathbb{R}$  in output. L'output della regressione lineare è una funzione lineare dell'input.

Definiamo  $\hat{y}$  come il valore che il nostro modello predice, definiamo dunque l'output come:

$$\hat{y} = w^\top x$$

Dove:  $w$  è un vettore di parametri.

Questi parametri, anche chiamati pesi, determinano il comportamento del sistema; in questo specifico caso si tratta del coefficiente per cui moltiplichiamo il vettore di input  $x$ .

$$\hat{y} = w^\top x + b$$

Questa è una *affine function*, ovvero una funzione lineare con una traslazione ( $b$  è noto come *intercept term* o *bias*). Come si può notare, inoltre, l'equazione assomiglia molto a quella di una retta in due dimensioni:  $y = mx + q$ . Infatti per un grado  $n = 1$  la regressione lineare è proprio una retta.

Facciamo un breve esempio pratico: supponiamo di avere un dataset con GDP per capita e un valore di soddisfazione della vita per ogni paese del mondo e volessimo costruire un modello che preveda quest'ultimo valore<sup>1</sup>.

Prima di tutto plottiamo i dati:

---

<sup>1</sup>Questo valore viene misurato con la scala di Cantril.

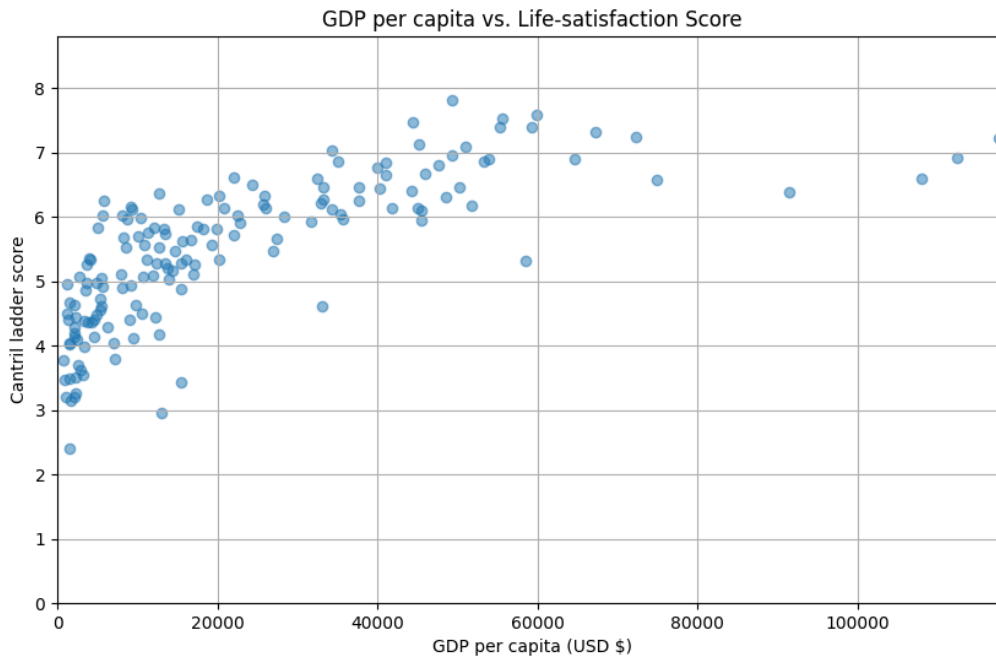


Figure 2: Plot dei dati GDP vs Life-satisfaction degli ultimi dati disponibili per ogni paese. (ex Austria)

Ora proviamo ad utilizzare la regressione lineare per prevedere il livello di soddisfazione della vita in Austria, che abbiamo escluso dal training set, dato il suo GDP per capita:

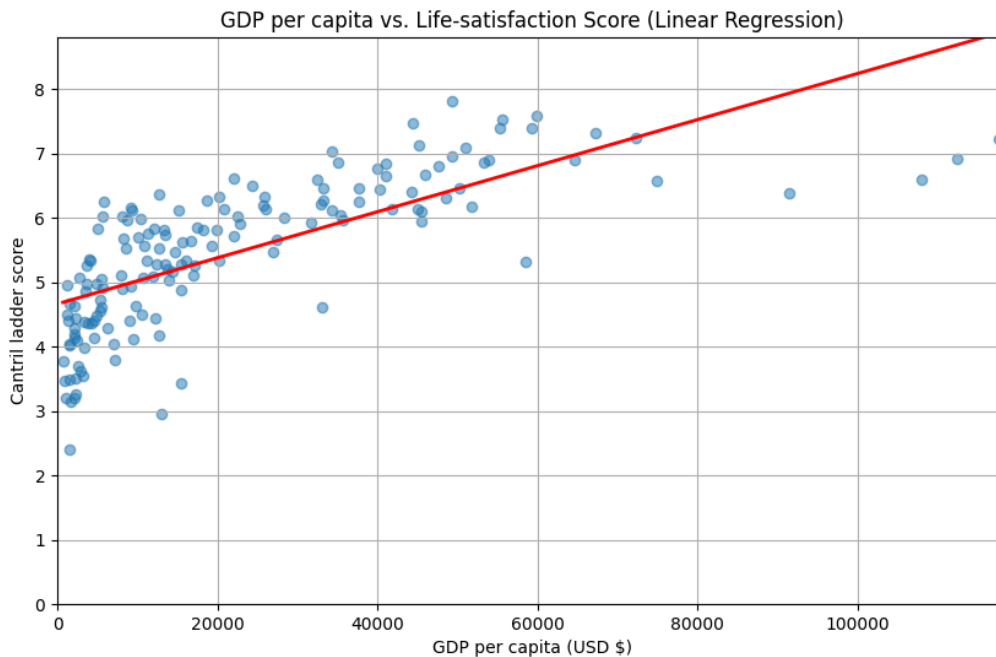


Figure 3: Plot dei dati GDP vs Life-satisfaction con la regressione lineare e grado 1

L'austria nel 2022 aveva un GDP per capita di \$55,867 e un livello di felicità di 7,09. Il modello di regressione lineare ci dice che il livello di felicità previsto è di 6,66. Forse possiamo fare di meglio.

Torniamo sulla formula della regressione lineare, possiamo generalizzarla come:

$$\hat{y} = b + w_1x_1 + w_2x_2 + \dots + w_nx_n$$

Dalla formula generalizzata capiamo che la regressione lineare può funzionare anche in più dimensioni, non solo con una variabile indipendente; ed in questo caso si dice “multivariata”. Per esempio con 2 variabili indipendenti

avremo un piano. Dunque se aggiungessimo lo Human Freedom Index come feature, avremmo un modello tridimensionale:

3D Scatter Plot of GDP per capita, Freedom Index, and Cantril ladder score

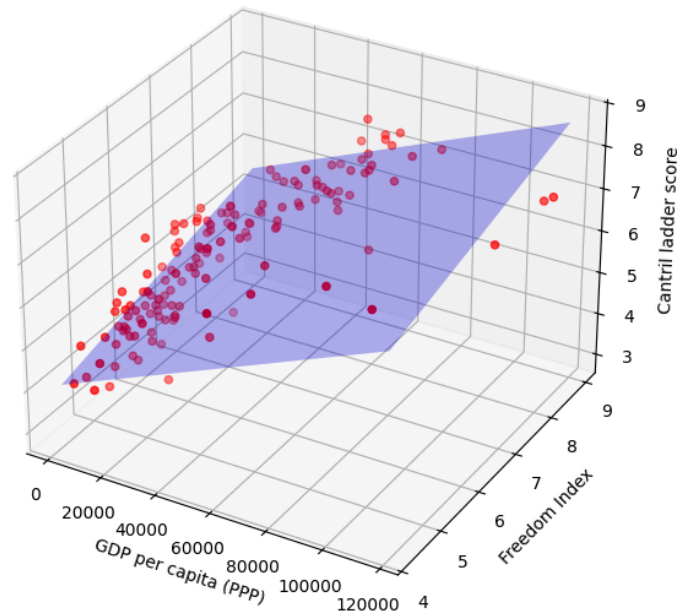


Figure 4: Plot dei dati GDP e Freedom vs Life satisfaction in 3D

In questo caso la predizione del modello per l'Austria è di 6,80, più vicina al valore reale.

### 3. Polynomial Regression

Nel caso in cui non avessimo altre features a disposizione, o la distribuzione dei dati non fosse lineare, potremmo utilizzare una regressione polinomiale. Nel caso preso in esempio, abbiamo visto come la retta non fosse in grado di generalizzare particolarmente bene i dati. Prima di procedere con la pratica vediamo la formula della regressione polinomiale, anche se è abbastanza intuitiva e non ci dovrebbe essere nulla da spiegare:

$$\hat{y} = wx + b \xrightarrow{\text{polinomiale}} \hat{y} = b + \sum_{i=1}^n w_i x^i$$

Applicando la regressione polinomiale con grado del polinomio:  $n = 2$  all'esempio visto in precedenza otteniamo:

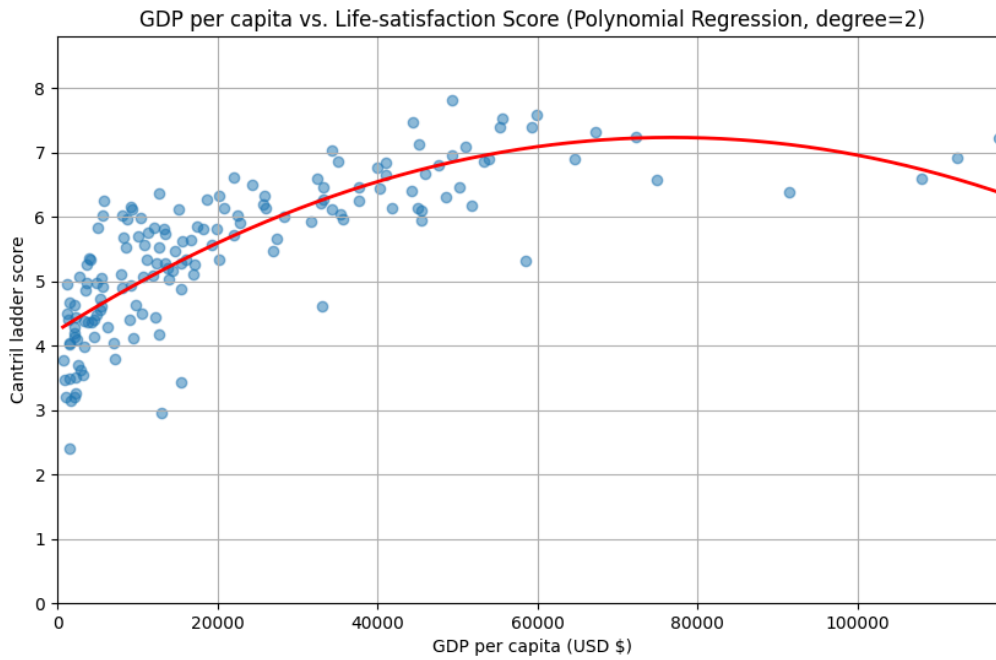


Figure 5: Plot dei dati GDP vs Life satisfaction con la regressione lineare e grado 2

Ora il modello predice un valore di 7,01 per l'Austria, più vicino al valore reale. Proviamo con gradi ancora più alti:

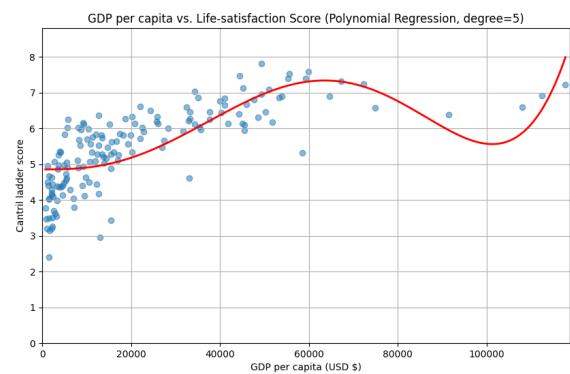
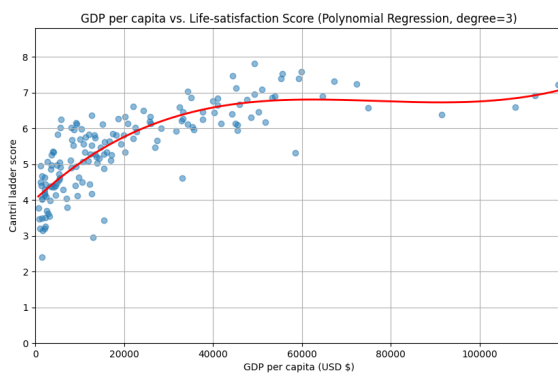


Figure 6: Plot dei dati GDP vs Life satisfaction con la regressione lineare e grado 3 e 5

Con un grado 3 il modello predice un valore di 6,79, mentre con un grado 5 il modello predice un valore di 7,21.

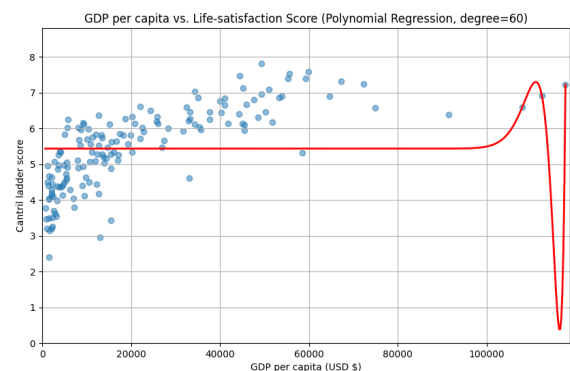
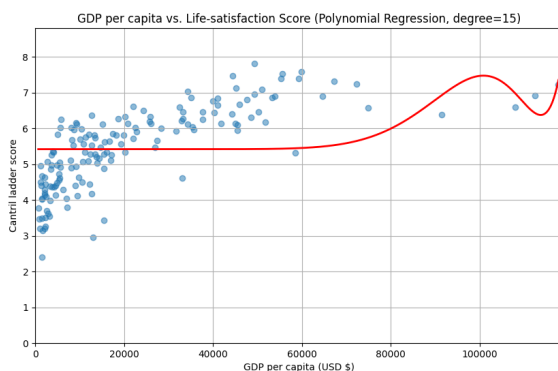


Figure 7: Plot dei dati GDP vs Life satisfaction con la regressione lineare e grado 15 e 60

Se alziamo ulteriormente il grado del polinomio, il modello non migliorerà; anzi dalla tabella di seguito e dalle immagini precedenti dovrebbe essere chiaro come avere un grado del polinomio più alto non implichi che il modello generalizzi meglio.

grado	1	2	3	5	15	20	30	40	50	60
predizione	6.66	7.01	6.79	7.21	5.43	5.43	5.43	5.43	5.43	5.43
errore	$\pm 0.44$	$\pm 0.09$	$\pm 0.31$	$\pm 0.11$	$\pm 1.67$	$\pm 1.67$	$\pm 1.67$	$\pm 1.67$	$\pm 1.67$	$\pm 1.67$

Dopo questi due capitoli, dovrebbero sorgere al lettore almeno due domande:

1. Com'è possibile che la regressione lineare performi meglio della regressione polinomiale con grado  $\geq 15$ ?

Innanzitutto è necessario precisare che come “test set” abbiamo utilizzato un solo data point. Tuttavia questo è un esempio del **Tradeoff tra Bias e Varianza**: nonostante la regressione lineare abbia un **training error** maggiore della regressione polinomiale in alcuni casi, è in grado di **generalizzare** meglio a causa della alta variabilità dei modelli con gradi più alti.

2. Come possiamo valutare il modello in modo rigoroso?

Per valutare il modello in maniera sistematica dovremmo innanzitutto dividere il dataset in training set e test set; in quanto il test set d'esempio è composto solo dall'Austria. Successivamente dobbiamo decidere come valutare l'errore del modello, una delle metriche più diffuse per la task di regressione è l'**errore quadratico medio** (MSE).

3. Come possiamo capire quale grado del polinomio è il migliore?

Il grado del polinomio è un **iperparametro** del modello; e per essere selezionato al meglio viene utilizzato un **validation set**.

Nel prossimo capitolo vedremo come valutare un modello di regressione, come selezionare il grado del polinomio.

## 4. Valutazione di un Modello di Regressione

Nei precedenti capitoli abbiamo valutato “a spanne” i modelli di regressione, per valutarli opportunamente dobbiamo avere a disposizione un dataset abbastanza grande da poterci permettere di suddividere i dati in training set e test set appunto. In generale la suddivisione si aggira attorno ad un rapporto 80/20 (80% training set, 20% test set), ma non esiste una regola fissa.

Una volta diviso il dataset, possiamo suddividere il training set in due parti; così da ottenere il validation set. Anche per questa suddivisione non esistono regole fisse.

### 4.1. Hyperparameters

Gli iperparametri sono parametri che non vengono appresi durante il training, ma che influenzano il comportamento del modello. Molti modelli di Machine Learning hanno iperparametri, per quanto riguarda la regressione lineare, di base, ha solo il grado del polinomio. Il grado del polinomio, come abbiamo visto precedentemente determina la capacità del modello. Allo stesso modo  $\lambda$  nella regolarizzazione, che vedremo successivamente è un iperparametro.

Gli iperparametri non vengono appresi durante il training, proprio perché se fosse così il modello non sarebbe in grado di generalizzare bene: ad esempio nella polinomial regression, se il grado del polinomio fosse un parametro appreso, il modello potrebbe avere un grado del polinomio molto alto, per minimizzare l'errore sul training set, ma non generalizzerebbe bene. Questo fenomeno è noto come **overfitting**; e da qui nasce la necessità di un validation set.

Ovviamente questa divisione dipende dalla quantità di dati a disposizione, nel caso del nostro esempio i dati a disposizione sono poco più di un centinaio; in questi casi è utile sfruttare la tecnica del **cross-validation**.

### 4.2. Underfitting e Overfitting

L'Underfitting si verifica quando il modello non ottiene buone prestazioni ne sul training set, ne sul test set.

L'Overfitting si verifica quando il modello ottiene buone prestazioni sul training set ma non sul test set.

**The No Free Lunch Theorem** Contrariamente a quanto si possa pensare, non esiste un modello che sia il migliore in assoluto per tutti i problemi.

### 4.3. Regulaization / Regularizzazione

La regolarizzazione è una qualsiasi modifica che apportiamo al modello per ridurre l'errore di generalizzazione (ma non il training error).

Il comportamento dell'algoritmo è influenzato infatti, non solo dalla capacità del modello (spazio delle ipotesi); ma anche dall'identità delle funzioni utilizzate. Per esempio, la regressione lineare ha uno spazio delle ipotesi composto esclusivamente da funzioni lineari e, nel caso non ci sia relazione lineare tra i dati (e.g.  $\sin(x)$ ), non sarà in grado di generalizzare bene.

Potremmo modificare il criterio di ottimizzazione per la regressione lineare includendo un termine regolarizzatore (denotato con  $\Omega(w)$ ) nella funzione di costo.

Nello specifico caso del weight decay il rego è uguale a:  $\Omega(w) = w^T w$ . Dunque il criterio sarà:

$$J(w) = \text{MSE}_{\text{train}} + \lambda w^T w$$

in questo modo minimizziamo una somma che comprende sia l'errore quadratico medio sul training set, sia il termine di regolarizzazione. In questo caso il termine  $\lambda$  è un iperparametro che regola l'importanza del termine di regolarizzazione. Con  $\lambda = 0$  il modello si comporta come una regressione lineare standard, mentre con  $\lambda > 0$  il modello tenderà a preferire pesi più piccoli, da questo il nome weight decay.

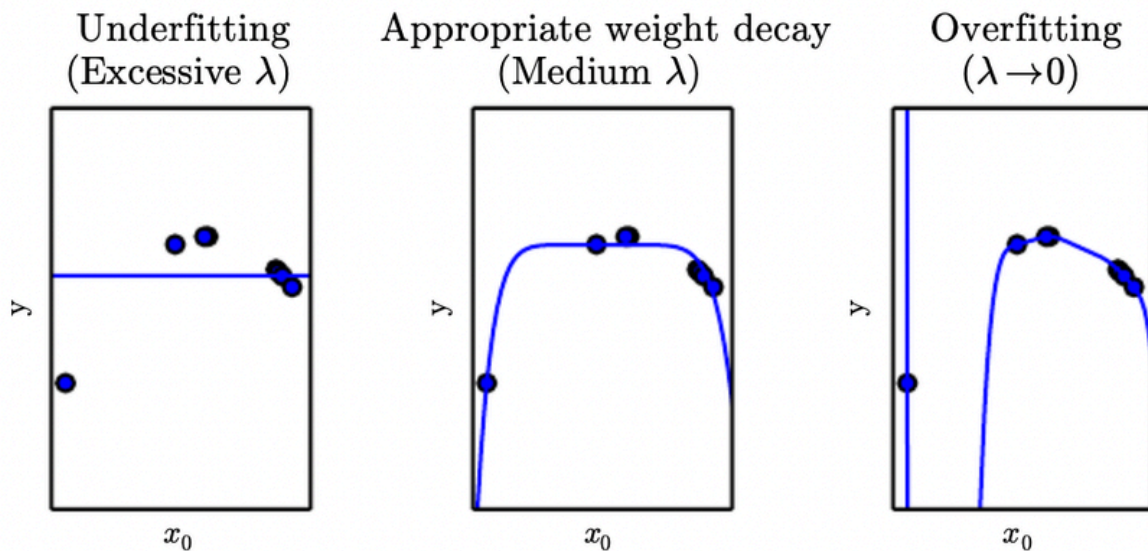


Figure 8: “Il modello utilizzato ha solo funzioni di grado 9, mentre il dataset è generato da una funzione quadratica.”

Nel campo del Machine Learning esistono diverse varianti per quanto riguarda le tecniche di regolarizzazione.

Famiglia delle  $L^p$  norme; generalizzata con la formula:

$$\|x\|_p = \left( \sum_{i=1}^n |x_i|^p \right)^{\frac{1}{p}}$$

con  $n$  ad indicare le dimensioni e  $p \in [1, +\infty)$ .

- La norma 1 è banalmente la somma dei valori assoluti dei componenti.
- La norma 2 o Norma Euclidea, è la radice quadrata della somma dei quadrati dei valori:

$$\|x\|_2 = \sqrt{\sum_{i=1}^n x_i^2}$$

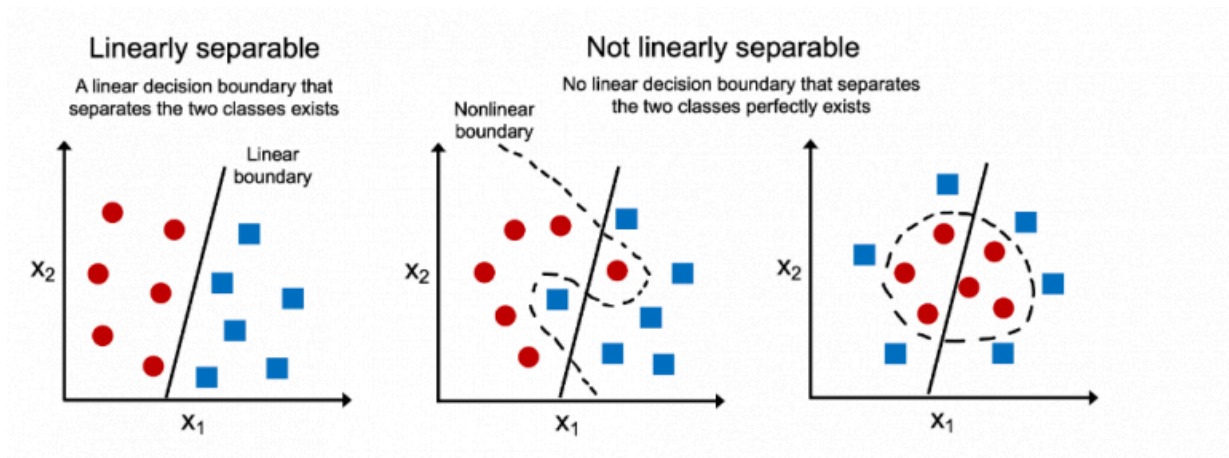
## 4.4. Dataset Augmentation

Il miglior modo per avere un modello che generalizza bene è trainarlo su più dati e, spesso, il dataset a disposizione non è abbastanza grande. Un modo per risolvere questo problema sono le tecniche di dataset augmentation. Questo approccio è molto efficace con le task di classificazione, object recognition e speech recognition. Com'è facile immaginare per quanto concerne l'object recognition, possiamo ruotare, scalare, e traslare le immagini; per lo speech recognition possiamo aggiungere rumore alle registrazioni.

L'iniezione di rumore è alla base di alcuni modelli unsupervised, come il denoising autoencoder. La noise injection può inoltre essere implementata negli hidden layer.

## 5. Linear Classifier

Il classificatore lineare, come il nome suggerisce, è un modello di classificazione che si basa su una funzione lineare per separare le classi. Questo richiede che i dati siano linearmente separabili, ovvero che esista almeno un iperpiano che separi le classi; ciò significa che questo modello è **parametrico** (*high-bias model*). Ovviamente un solo iperpiano potrà separare 2 classi; per questo motivo, in generale, un classificatore lineare è un classificatore binario.



### 5.1. Online Learning

Durante la fase di training, il classificatore lineare aggiorna i pesi in modo incrementale, utilizzando un solo esempio alla volta. Questo approccio è noto come **online learning**. L'aggiornamento dei pesi è basato sull'errore commesso dal modello rispetto all'etichetta corretta.

L'online learning rappresenta un'intera branca del Machine Learning, in cui i modelli vengono addestrati su un flusso continuo di dati. Questo approccio è particolarmente utile quando il dataset è troppo grande per essere caricato in memoria, o quando i dati arrivano in tempo reale.

L'opposto dell'online learning è il **batch learning**, in cui il modello viene addestrato su tutto il dataset in una sola volta.

### 5.2. Training Phase

Durante la fase di training, il classificatore lineare per prima cosa inizializza i pesi e il bias a valori casuali (o a 0). Successivamente, per ogni esempio del training set, calcola la predizione del modello e, se errata (non corrispondente alla stessa classe della label), aggiorna i pesi in base all'errore commesso. Lo pseudocodice dell'algoritmo 5.4.1 di training per un classificatore lineare viene riportato successivamente in questo capitolo.

### 5.3. Inference Phase

La fase di inferenza di un classificatore lineare è molto semplice: dato un input  $x$ , il modello calcola il prodotto scalare tra il vettore dei pesi  $w$  e il vettore delle features  $x$ ; a questo valore viene sommato il termine di bias  $b$ .

$$f(x, y) := \begin{cases} b + \sum_{i=1}^n w_i f_i > 0 & \text{if positive} \\ b + \sum_{i=1}^n w_i f_i < 0 & \text{if negative} \end{cases}$$



## 5.4. Perceptron

Il Perceptron è un tipo di classificatore lineare. Alla formula vista in precedenza, viene aggiunta una funzione di attivazione  $h$ . La funzione di attivazione canonica è la funzione di Heaviside (o funzione di step), che restituisce 1 se il valore è maggiore di 0, altrimenti 0:

$$h(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

Ne segue che la funzione del Perceptron è:

$$f(x) = h(wc \cdot x + b)$$

### 5.4.1. Perceptron Algorithm

L'algoritmo del Perceptron è molto semplice: inizializziamo i pesi e il bias a 0, e iteriamo su tutti gli esempi del training set. Per ogni esempio, calcoliamo la predizione del modello e aggiorniamo i pesi in base all'errore commesso. Di seguito l'algoritmo in pseudocodice; dove  $y$  è l'etichetta corretta (con valore 1 o  $-1$ ),  $f(x)$  è la predizione del modello nello step corrente, e  $x_i$  è l'elemento  $i$ -esimo del vettore delle features.

```
repeat until convergence {                                     // or for a fixed number of iterations
  for each example in training set {
    if  $y \neq f(x)$  {                                           // if the prediction is wrong
      for each weight  $w_i$  {
         $w_i = w_i + y * x_i$ 
         $b = b + y$ 
      }
    }
  }
}
```

È importante capire le limitazioni del Perceptron: questo algoritmo funziona solo se i dati sono linearmente separabili; altrimenti l'algoritmo non converge. Inoltre, se i dati non sono linearmente separabili, il Perceptron non restituisce un modello ottimale. Questa limitazione è stata la causa dell'AI

## 5.5. Multiclass Classification

Il Perceptron è un classificatore binario, ovvero può classificare un datapoint in una tra due classi. La Multiclass Classification è un'estensione della Binary Classification, in cui il modello deve classificare un datapoint in una tra più classi. Esistono diversi approcci per estendere un classificatore binario a uno multiclasse, tra cui:

### 5.5.1. OVA

L'approccio **One-Versus-All** (OVA) consiste nel creare un classificatore binario per ogni classe. Ad esempio, se abbiamo 3 classi, creiamo 3 classificatori binari; ogni classificatore sarà allenato per predire se un elemento appartiene o meno a quella classe. Durante la fase di inferenza, il modello restituisce la classe con il classificatore binario che ha restituito il valore più alto. Infatti, nella formula:  $b + \sum_{i=1}^n w_i f_i$  possiamo vedere la componente della sommatoria come *confidenza* del modello per la classe di quel classificatore. Questa componente rappresenta la distanza del punto analizzato dall'iperpiano che separa le due classi.

### 5.5.2. Pseudocodice Train OVA

```
// D_multiclass is the multiclass dataset
// K is the number of classes
// BinaryTrain is the binary classifier training function

function OneVsAll(D_multiclass, K, BinaryTrain){
  classifiers = []                                           // initialize an empty list of K classifiers
  for i = 1 to K {                                         // K is the number of classes
    D_bin = relabel(D_multiclass)                          // so class i is positive and -i is negative
    classifiers[i] = BinaryTrain(D_bin)                    // train a binary classifier on the binary dataset
  }
  return classifiers
}
```

### 5.5.3. Pseudocode Test OVA

```
// classifiers is the list of K classifiers
// x is the input to classify
// K is the number of classes

function OneVsAllPredict(classifiers, x, K){
    score[K] = 0 // initialize K-many scores to zero
    for i = 1 to K{ // for each class
        y = classifiers[i].predict(x) // predict x using i-th classifier
        score[i] = score[i] + y // add the prediction to the score
    }
    return argmax(score) // return the class with the highest score
}
```

### 5.5.4. AVA

L'approccio **All-Versus-All** (AVA) consiste nel creare un classificatore binario per ogni coppia di classi. Questo approccio è anche noto come **all pairs**. Consiste nell'allenare  $\frac{K(K-1)}{2}$  classificatori binari, dove  $K$  indica il numero di classi. Ogni classificatore binario discrimina tra due classi diverse; diversamente dall'approccio OVA (dove ogni classificatore binario discrimina tra una classe e tutte le altre). Per questo motivo, se le etichette sono equamente distribuite, l'approccio AVA sarà più veloce in fase di training: il numero di classificatori binari da allenare è maggiore, ma il numero di esempi per classificatore è minore.

Per quanto riguarda la fase di inferenza, con questo approccio è possibile sia restituire la classe con il maggior numero di voti, sia restituire la classe che ha ottenuto score più alto (maggior *confidenza*). Nello pseudocodice che segue, per quanto riguarda la parte di *test*, viene restituita la classe con lo score più alto. Se compariamo questo approccio con l'approccio OVA nella fase di test, l'approccio AVA è generalmente più lento, proprio per il numero maggiore di classificatori.

A causa della sua struttura, l'approccio AVA è meno robusto rispetto all'approccio OVA, nel caso in cui una classe sia sotto-rappresentata all'interno del dataset.

### 5.5.5. Pseudocode Train AVA

```
// D_multiclass is the multiclass dataset
// K is the number of classes
// BinaryTrain is the binary classifier training function
// i and j are the classes to discriminate, combined, they represent a key

function AllVsAll(D_multiclass, K, BinaryTrain){
    classifiers = [] // initialize a list of K(k-1)/2 classifiers
    for i = 1 to K-1 {
        D_pos = D_multiclass.filter(class == i) // all instances of class i
        for j = i+1 to K {
            D_neg = D_multiclass.filter(class == j) // all instances of class j
            D_bin = D_pos + D_neg // concatenate the two datasets
            classifiers[i,j] = BinaryTrain(D_bin) // train a binary classifier
        }
    }
    return classifiers
}
```

### 5.5.6. Pseudocode Test AVA

```
// classifiers is the list of K classifiers
// x is the input to classify
// K is the number of classes

function AllVsAllPredict(classifiers, x, K){
    score[K] = 0 // initialize K-many scores to zero
    for i = 1 to K{ // for each class
        for j = i+1 to K{ // for each other class
            y = classifiers[i,j].predict(x) // predict x using i-th classifier
            score[i] = score[i] + y // add the prediction to the score
            score[j] = score[j] - y // subtract the prediction from the score
        }
    }
    return argmax(score) // return the class with the highest score
}
```

### 5.6. Micro vs Macro Average

Nel caso di classificazione multiclasse, è possibile utilizzare due diversi sistemi per valutare le performance del modello: **Micro Average** e **Macro Average**.

La **Micro Average**: non è altro che l'utilizzo della metrica prescelta su tutte le classi; senza alcuna distinzione tra loro. Se per esempio utilizziamo l'accuratezza come metrica, possiamo calcolarla come segue:

$$A_c = \frac{P_c}{P_c + P_e}$$

Dove  $P_c$  rappresenta le predizioni corrette e  $P_e$  rappresenta le predizioni errate.







La **Macro Average**: calcola l'accuratezza per ogni classe e ne fa la media. Questo significa che si calcola l'accuratezza per ogni classe e si fa la media di queste accurattezze. Questa metrica è utile quando si vuole dare lo stesso peso a tutte le classi; ovvero, anche se una classe è sottorappresentata, essa avrà lo stesso peso delle altre classi.

Per utilizzare la macro average, si calcola (con la metrica prescelta) lo score per ogni classe, che nomineremo  $Ev_i$  (dove  $i$  rappresenta la classe). Successivamente, si calcola la media di tutti gli score; un esempio con l'accuratezza:

$$A_m = \frac{\sum_{i=0}^n Ev_i}{n}$$

Dove  $A_m$  rappresenta l'accuratezza media e  $n$  rappresenta il numero di classi.

Un esempio per confrontare le due metriche: (il testo in grassetto rappresenta l'ultimo valore calcolato per quella classe),

	Label	Prediction	Micro Average	Macro Average
	circle	square	0	circle = 0/1
	square	square	1	<b>square = 1/1</b>
	circle	circle	1	<b>circle = 1/2</b>
	triangle	rectangle	0	triangle = 0/1
	triangle	triangle	1	<b>triangle = 1/2</b>
	rectangle	rectangle	1	<b>rectangle = 1/1</b>
Score			$\frac{4}{6}$	$\frac{\frac{1}{2}+1+\frac{1}{2}+1}{4} = \frac{3}{4}$

## 5.7. Confusion Matrix

La **confusion matrix** è una matrice che, a colpo d'occhio riesce a mostrare le performance di un classificatore. La confusion matrix è così costruita:

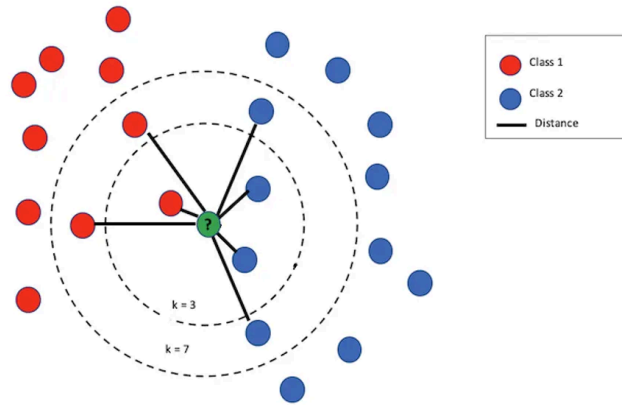
- Le righe rappresentano le classi reali (Labels)
- Le colonne rappresentano le classi predette (Predictions)

Spesso si utilizzano valori percentuali, ed in generale è utile per capire su quale classe il nostro modello performa peggio e di conseguenza concentrarci su quella (magari aggiungendo datapoints appartenenti a quella classe nel dataset). Riprendendo l'esempio precedente, possiamo costruire la confusion matrix. Sulla diagonale principale, troviamo i valori corretti, mentre gli altri valori rappresentano gli errori commessi dal modello.

Prediction Label	Circle	Square	Triangle	Rectangle
circle	1	1	0	0
square	0	1	0	0
triangle	0	0	1	1
rectangle	0	0	0	1

## 6. KNN

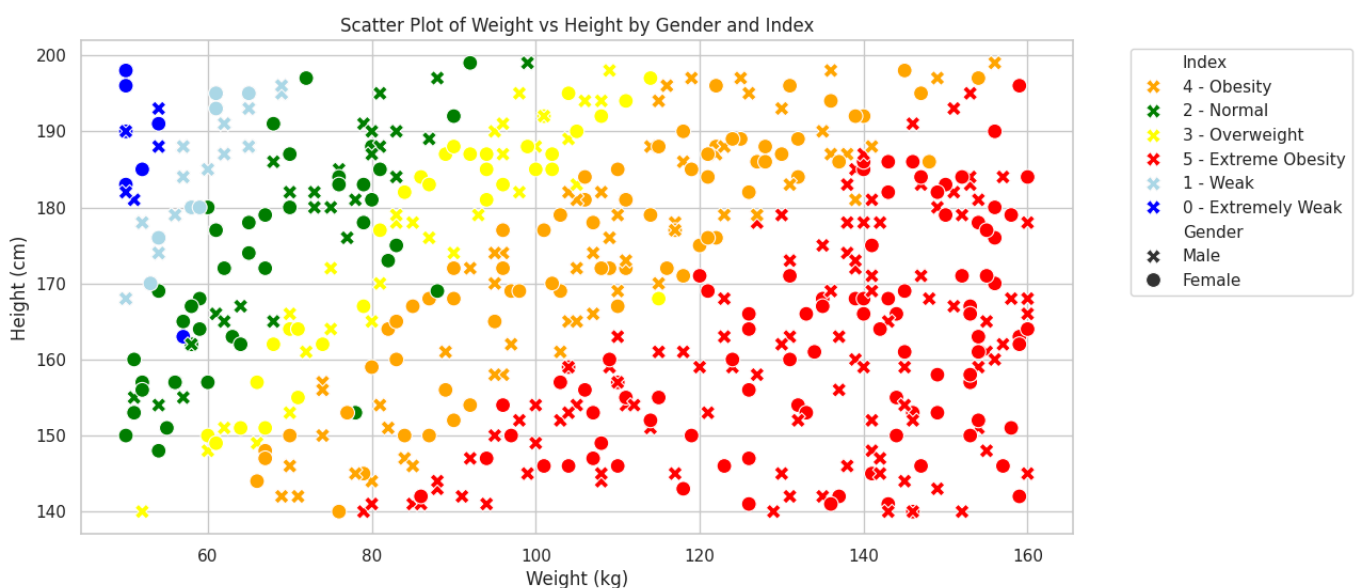
Il K-Nearest Neighbors (KNN) è un modello **supervised**, utilizzato per la classificazione, che si basa sulla distanza tra i punti del dataset. Al momento della predizione, il modello calcola la distanza tra il nuovo punto ed altri  $k$  punti del dataset, e predice la classe del nuovo punto basandosi sulla classe più frequente tra i  $k$  punti più vicini.



Il KNN ha alcuni aspetti caratteristici:<sup>2</sup>

- È un modello **non parametrico**, cioè non necessita di assunzioni sulla distribuzione dei dati. La complessità del modello cresce con la dimensione del dataset.
- È un **lazy learner**. Il termine *lazy learning* si riferisce all'approccio dell'apprendimento in cui il modello non fa praticamente nulla durante la fase di addestramento. In altre parole, l'algoritmo rimanda il processo di apprendimento fino a quando non è necessario fare una previsione.
- È un modello **instance-based**. L'*instance-based learning* è un tipo di apprendimento dove il modello memorizza semplicemente i dati di addestramento e non costruisce un modello esplicito per fare previsioni. Quando si richiede una previsione, il modello confronta il nuovo esempio con gli esempi memorizzati per fare una stima.
- **K** è appunto l'hyperparametro che determina il numero di data-points "vicini" da tenere in considerazione durante la predizione. Ne segue che se  $K = n$ , dove  $n$  è la cardinalità del mio dataset, allora verrà sempre predetta la classe con più elementi (underfitting). Se al contrario, viene impostato  $k = 1$  si ottiene un modello che sicuramente overfitta. Un valore di  $K$  dispari aiuta ad evitare situazioni di ambiguità. Come per tutti gli altri hyperparametri, si utilizza il validation set per determinare il  $K$  migliore.

Per l'esempio pratico di questo modello non useremo il classico dataset *Iris*, ma questo dataset generato artificialmente, contenente informazioni riguardo il sesso, l'altezza, il peso e l'indice corporeo di 500 persone:<sup>3</sup>



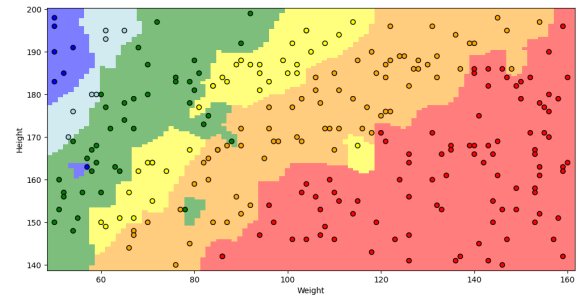
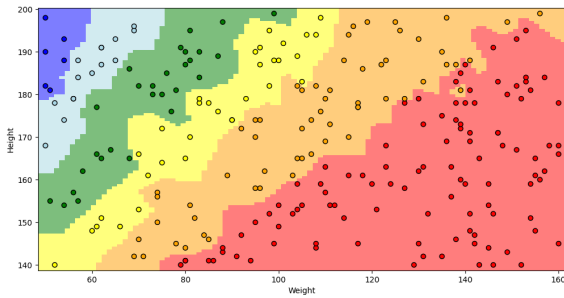
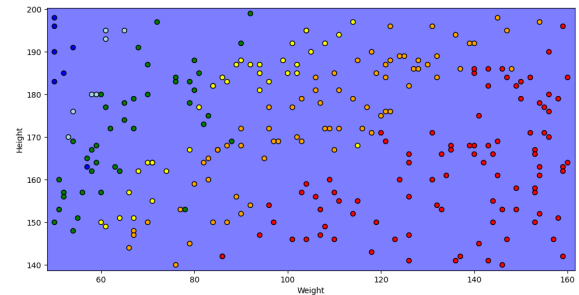
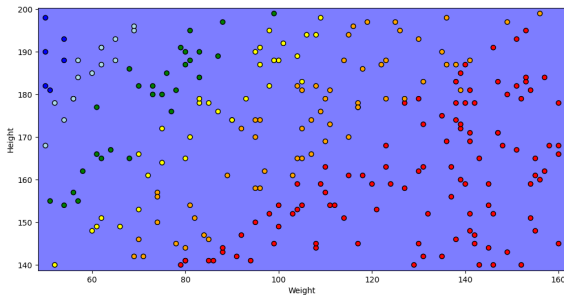
Per riprendere ciò che abbiamo detto sopra, ecco qui un esempio di overfitting e underfitting con KNN:

<sup>2</sup>Se la differenza tra *lazy* e *instance-based* non è chiara ora lo sarà nel capitolo dell'SVM.

<sup>3</sup>Gli autori degli appunti non credono che possano esistere persone alte 200cm e pesanti 45kg; perlomeno non vive.

Male

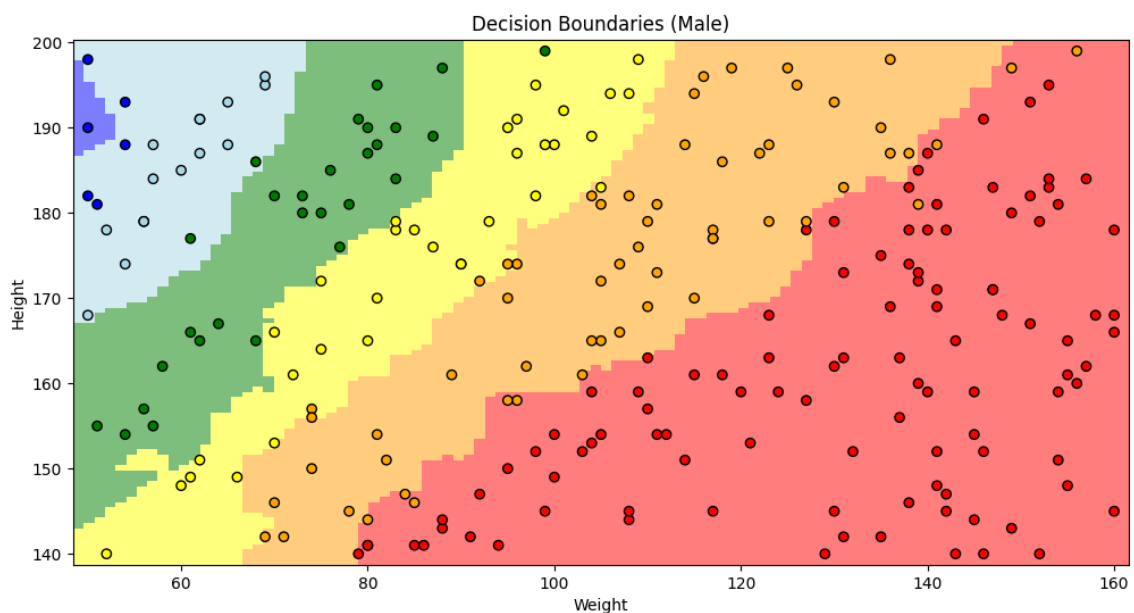
Female

Overfitting:  $K = 1$ Underfitting:  $K = 245$ 

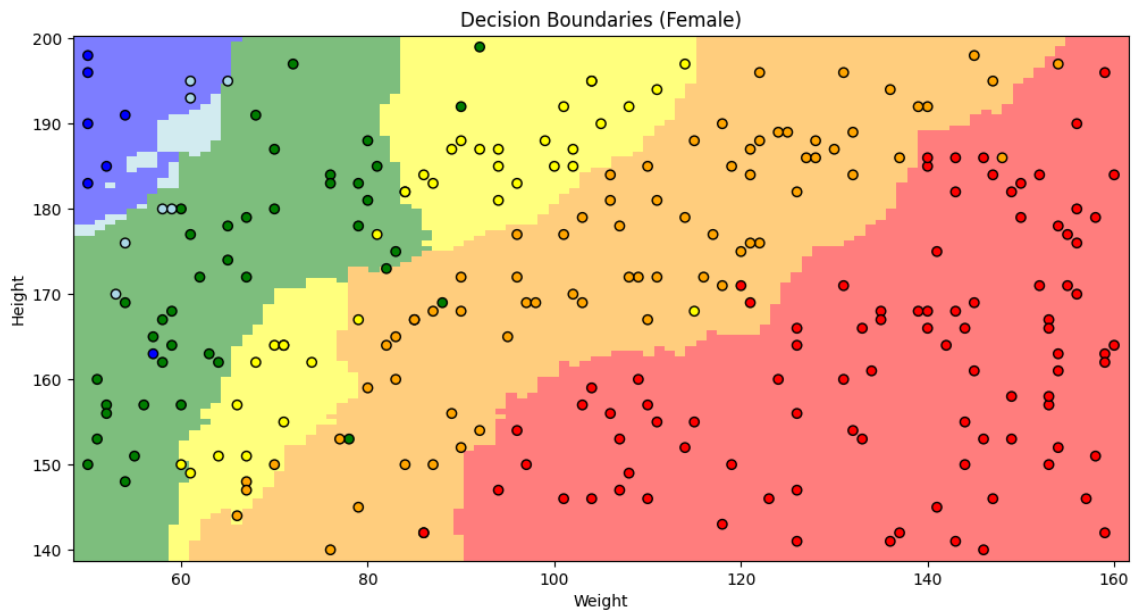
Il fatto che overfitti si nota dalla presenza di molteplici “isole” all’interno delle diverse categorie (idealmente vorremmo che fosse a fasce); mentre l’underfitting si nota dalla presenza di una sola categoria o dall’assenza di alcune. Da questo esempio possiamo inoltre dedurre che il KNN può funzionare senza modifiche anche per la classificazione multi-classe.

Come sempre per questi esempi il dataset è troppo piccolo ed il tempo è poco, per non complicarci troppo le cose evitiamo di utilizzare un validation set, impostiamo  $K = 11$  e vediamo come si comporta il modello.<sup>4</sup>

Una buona regola empirica se non sai quali valori prendere in considerazione per la scelta di  $K$  è  $K = \sqrt{n}$   
— Il web



<sup>4</sup>Sfortunatamente con questo dataset, utilizzando valori maggiori, si perdono categorie



## 6.1. Standardization and Scaling

Poiché KNN si basa sulla distanza tra i punti, se una feature ha un range molto più ampio rispetto ad un'altra, la distanza sarà dominata dalla feature con il range più ampio. Per evitare questo problema, è necessario standardizzare o scalare i dati. Gli strumenti presentati durante il corso sono la standardizzazione (o Z-score normalization) e il Min-Max scaling.

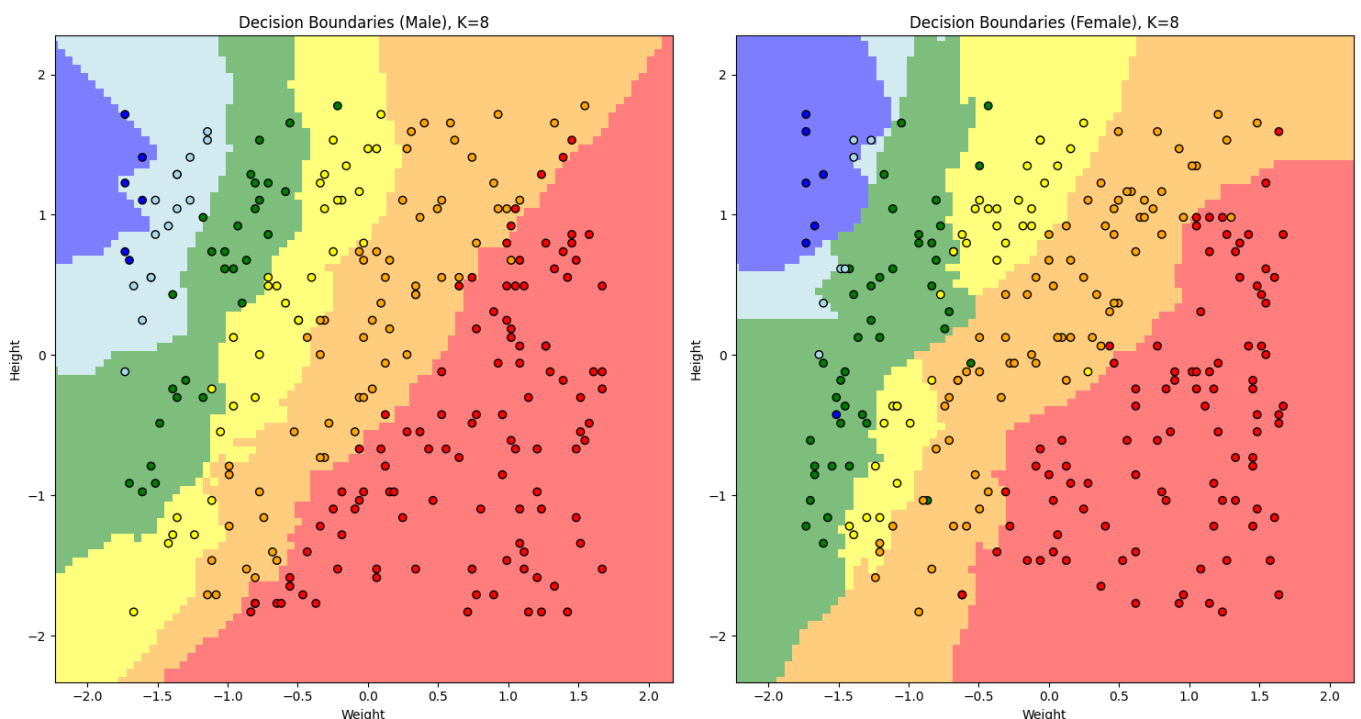
### 6.1.1. Z-score normalization

Questa procedura scala i dati in modo che abbiano una media di 0 e una deviazione standard di 1. La formula è la seguente:

$$z = \frac{x - \mu}{\sigma}$$

Dove  $x$  è il valore della feature,  $\mu$  è la media della feature e  $\sigma$  è la deviazione standard della feature.

Tornando all'esempio precedente, possiamo vedere come la standardizzazione influenzi la densità del dataset:

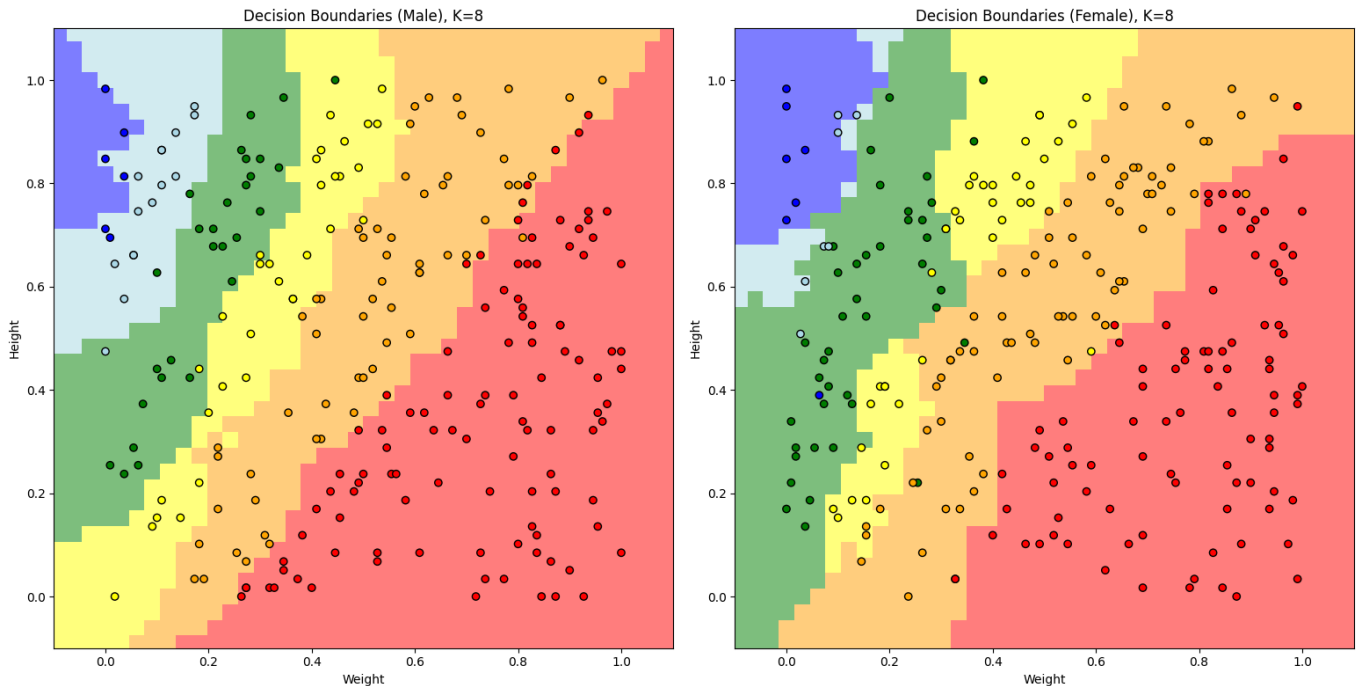


### 6.1.2. Min-Max scaling

Questa procedura scala i dati in modo che siano compresi tra 0 e 1. La formula è la seguente:

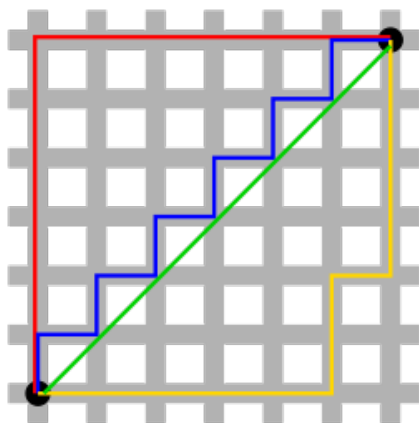
$$x_{\text{scaled}} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

Di seguito possiamo vedere come il Min-Max scaling influenzi la densità del dataset:



### 6.2. Distance Metrics

Contrariamente a quanto si possa pensare esistono diverse metriche che possono essere utilizzate per determinare la distanza da i data-points (indichiamo con  $x$  e  $y$  due punti nello spazio  $n$ -dimensionale e con  $x_1, x_2, \dots, x_n$  e  $y_1, y_2, \dots, y_n$  le loro coordinate):



La lunghezza della linea blu, rossa e gialla è la medesima e rappresenta la Manhattan distance tra i due punti (12). La lunghezza della linea verde rappresenta la Euclidean distance tra i due punti ( $6\sqrt{2}$ )

- **Manhattan distance:** la distanza di Manhattan è la somma delle differenze assolute tra le coordinate dei punti. La formula è la seguente:

$$D(x, y) = \sum_{i=1}^n |x_i - y_i|$$

- **Euclidean distance:** la distanza euclidea è la radice quadrata della somma dei quadrati delle differenze tra le coordinate dei punti. La formula è la seguente:

$$D(x, y) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

- **Minkowski distance:** una generalizzazione della distanza euclidea e della distanza di Manhattan. La formula è la seguente:

$$D(x, y) = \left( \sum_{i=1}^n |x_i - y_i|^p \right)^{\frac{1}{p}} \quad \text{con } p \geq 1$$

- **Cosine distance:** la cosine distance è derivata dalla *cosine similarity*, che misura quanto due vettori sono orientati nella stessa direzione. La formula per la cosine similarity tra due vettori  $A$  e  $B$  è:



$$\text{Cosine Similarity}(\mathbf{A}, \mathbf{B}) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|}$$

dove  $\mathbf{A} \cdot \mathbf{B}$  è il prodotto scalare, mentre  $\|\mathbf{A}\| \|\mathbf{B}\|$  sono le norme dei vettori. La cosine similarity varia tra  $-1$  e  $1$ , dove  $1$  indica che i vettori puntano esattamente nella stessa direzione,  $0$  indica che sono ortogonali (non correlati), e  $-1$  indica che puntano in direzioni opposte.

La cosine distance è semplicemente definita come:

$$\text{Cosine Distance}(\mathbf{A}, \mathbf{B}) = 1 - \text{Cosine Similarity}(\mathbf{A}, \mathbf{B})$$

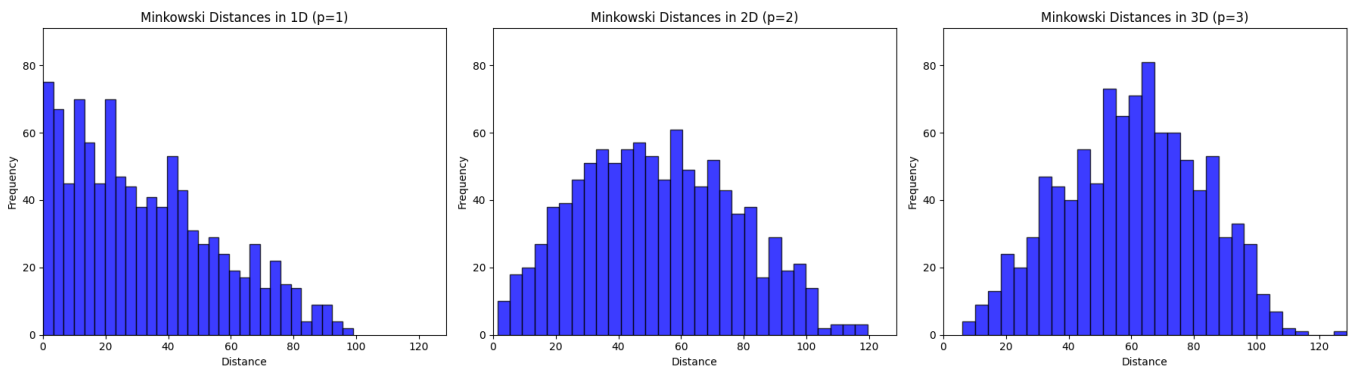
Dunque, la cosine distance varia tra  $0$  e  $2$ . Un valore di  $0$  indica che i vettori sono identici in termini di direzione, mentre un valore di  $1$  indica che sono ortogonali.

La cosine distance è particolarmente utile in applicazioni come l'analisi di testi e il riconoscimento di immagini, dove i dati possono essere vettori di caratteristiche normalizzati. Per esempio nell'analisi dei testi ogni elemento del vettore può rappresentare il numero di occorrenze di una certa parola.

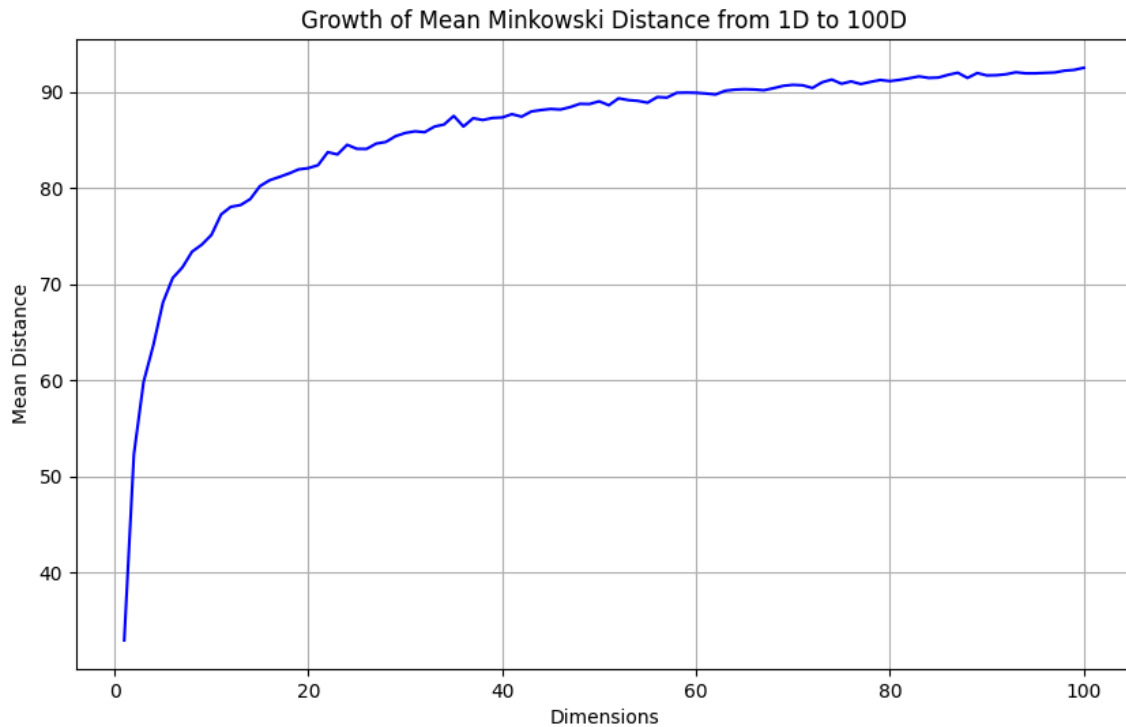
### 6.3. Curse of Dimensionality

Il problema principale dell'utilizzo delle distanze è che queste crescono esponenzialmente al crescere delle dimensioni; questo problema è noto come **Curse of Dimensionality**. Con KNN è necessario avere un dataset denso, i data-point dello stesso gruppo devono essere vicini **in ogni dimensione**; a differenza di altri algoritmi. Inoltre, per ottenere una buona performance, il numero di dati deve crescere esponenzialmente con il numero delle dimensioni.

Possiamo osservare le distribuzioni delle distanze tra due punti in 1,2,3 dimensioni:



Per evidenziare l'andamento esponenziale trovate qui un grafico che indica la distanza media di due punti con coordinate  $[0; 100]$ , in un sample composto da 1000 elementi, nelle dimensioni da 1D a 100D.



## 6.4. Computational Cost

Il fatto che KNN sia un *lazy learner* non implica che non ci sia un costo computazionale; anzi il costo computazionale pesa solo sulla fase di predizione (*inference phase*). L'algoritmo è lineare, il calcolo della distanza per tutti i  $k$  punti del dataset:  $O(kN)$ . Alcune strutture dati possono essere utilizzate per ridurre il costo computazionale, come il KD-Tree o il Ball-Tree, che permettono di ridurre il costo computazionale a  $O(\log(N))$ . Di seguito presentiamo una carrellata delle strutture dati più comuni:

### 6.4.1. KD-Tree

La struttura richiede un pre-processing dei dati e si presenta come un albero binario. Ogni nodo dell'albero rappresenta un iperpiano che divide lo spazio in due parti, o più semplicemente impone un vincolo tra i due nodi figli successivi. I nodi foglia contengono i punti del dataset. La " $k$ " nel nome si riferisce appunto al numero di features presenti. La ricerca di un punto nel KD-Tree è simile alla ricerca di un punto in un albero binario di ricerca. La ricerca di un punto in un KD-Tree ha un costo computazionale di  $O(\log(N))$ .

Sempre utilizzando l'esempio pratico, osserviamo la struttura del KD-Tree dopo l'applicazione del Min-Max scaling:

### 6.4.2. Ball-Tree

### 6.4.3. Cover Tree

### 6.4.4. R-Tree

### 6.4.5. VP-Tree

### 6.4.6. Approximate Nearest Neighbor Search (ANN) Techniques

### 6.4.7. Locality Sensitive Hashing (LSH)

### 6.4.8. Annoy

### 6.4.9. Hierarchical Navigable Small World (HNSW) Graphs

### 6.4.10. Lattice-based Methods

#### **6.4.11. FLANN (Fast Library for Approximate Nearest Neighbors)**