Riassunto Intelligenza Artificiale

A.A. 23-24
Falbo Andrea
Ringrazio Ruben per l'aiuto e CrystalSpider per gli appunti durante l'A.A 22-23

AI Simbolica

Semantic Web

RDF-RDFS

Differenza tra dato, informazione e conoscenza:

- I **dati** grezzi esistono ma non hanno significato. Es. 33,6
- Un **informazione** è un dato a cui è stato dato un significato in una connessione relazionale. Es. 33,6m
- Una **conoscenza** è una collezione di informazioni con l'intenzione di essere utile. Sono arricchite di semantica. Es. la balenottera è lunga max 33.6m

Il **Web Semantico** è un'estensione del nostro web dove la semantica è formale, ovvero strutturata, e ha una rappresentazione della conoscenza standardizzata.

RDF definisce la sintassi su cui si basano le ontologie del web semantico.

RDF si basa su **triple**:

- Soggetto: URI
- Predicato: URI
- Oggetto: URI/letterale

Sulla sintassi RDF si basano altri **framework** per migliorare la semantica delle triple, evitando triple incoerenti.

Un **vocabolario** è un insieme di termini utilizzato per descrivere un particolare dominio di conoscenza in modo che sia comprensibile sia agli esseri umani sia alle macchine. Questi vocabolari sono tipicamente espressi in formati come RDF e sono utilizzati per strutturare e etichettare i dati in modo che possano essere meglio interconnessi e interrogati su Internet. Ai vocabolari è associabile un prefisso, ovvero delle abbreviazioni che evitano di dover riscrivere una URI per esteso. Due esempi sono FOAF e DC.

Ciascuna tripla è classicamente espressa nella **N-Triples Notation**, dove ciascun URI è racchiuso tra parentesi angolari, i literals sono tra doppi apici e ogni tripla si conclude con il carattere punto. **Turtle** è un estensione e permette di definire delle scorciatoie

- Utilizzando @prefix si può associare un prefisso all'URI e usando @base fornisce URI da complementare a tutti gli elementi.
- Se una tripla termina con un ; indica che la prossima tripla ha come soggetto il precedente.
- Se una tripla termina con una , indica che la prossima tripla ha come soggetto e predicato il precedente.

RDFS, a differenza di RDF che si occupa solamente di definire come costruire un grafo, permette di spiegare come i nodi di un grafo sono relazionati, ovvero:

- Definire classi
- Definire il concetto di proprietà applicando restrizioni attraverso dominio e range
- Tutto è risorsa
- Possiamo definire sottoclassi e sotto proprietà

Knowledge Graph

Un'importante distinzione logica per soggetti/oggetti è quella di classi e individui. Le **classi** sono dei soggetti che rappresentano una categoria, mentre gli **individui** sono "realizzazioni" di una determinata categoria.

Infine, la conoscenza contenuta nel web semantico può essere facilmente rappresentata su un **grafo**, dove ogni soggetto/oggetto è un nodo e ogni relazione è un arco.

SPARQL

SPARQL (SPARQL Protocol And RDF Query Language) è:

1. un Declarative Query Language per grafi RDF
2. un Protocol Layer per usare SPARQL via HTTP.
3. una specifica di formato per output XML per SPARQL queries.
4. Standard W3C ispirato da SQL

SPARQL può essere utilizzato come:

1. Query: linguaggio di interrogazione dichiarativo per dati RDF
2. Algebra: mappatura da query a algebra relazionale per la manipolazione dei risultati
3. Update: linguaggio di manipolazione dichiarativo per dati RDF
4. Protocol: standard per la comunicazione tra servizi SPARQL e client

SPARQL si basa sul concetto di **serializzazione** RDF Turtle e **pattern matching** su grafi.

Le clausole principali sono:

- `PREFIX` per specificare i namespaces
- `SELECT` per specificare le variabili di output
- `FROM` per specificare il grafo su cui effettuare le query
- `WHERE` per specificare il graph pattern che deve matchare
- `FILTER` permette di aggiungere dei filtri alla ricerca, come `LANG` che permette di selezionare la lingua di output
- `MODIFIER` come `ORDER BY, LIMIT, OFFSET, GROUP BY`

SELECT:

- Uso: Viene usato per estrarre dati specifici dal dataset.
- Risultato: Restituisce una tabella di risultati, simile a SQL, con variabili e i loro valori corrispondenti.
- Esempio: Puoi chiedere i nomi e gli indirizzi email di tutte le persone in un dataset.

ASK:

- Uso: Utilizzato per porre una domanda che ha una risposta di tipo booleano
- Risultato: Restituisce true se la query corrisponde a qualche parte del dataset, altrimenti false.
- Esempio: Verificare se esiste una persona specifica nel dataset.

DESCRIBE:

- Uso: Utilizzato per ottenere una rappresentazione RDF di una risorsa specificata. Questo può includere dati sulle relazioni della risorsa con altre risorse.
- Risultato: Restituisce un insieme di triple RDF che descrivono la risorsa RDF in questione.
- Esempio: Ottenere tutte le informazioni disponibili su una particolare risorsa, come una persona o un luogo.

CONSTRUCT:

- Uso: Questa query è utilizzata per estrarre dati dal dataset e trasformarli in nuove triple RDF secondo un pattern specificato nella query.
- Risultato: Produce un nuovo set di dati RDF basato sul pattern definito nella query.
- Esempio: Creare un nuovo set di dati RDF che lega insieme specifiche risorse in modi nuovi o diversi.

OWL

Un'**ontologia** è una specifica formale esplicita di una concettualizzazione condivisa:

- formale: capibile da una macchina
- esplicito: bisogna definire i significati di tutti i concetti
- concettualizzazione: modello astratto
- condivisa: consenso sull'ontologia.

Può essere rappresentata in due aree:

- **Terminologica** (Classi): assetto profondo di quello che comprende l'area
- **Asserzionale** (Istanze): assetto profondo di quello che posso fare all'interno

Tramite RDF(S) è stato possibile creare un'ontologia basata su relazioni tipizzate e sui concetti di classi e individui. Tuttavia questo non è sufficiente per rappresentare conoscenze più complesse, come:

1. relazioni di **cardinalità**: una persona ha due genitori
2. **disgiunzioni** di insiemi: vivi e morti sottoclassi disgiunte di persone
3. **negazioni** logiche che non generano in automatico contraddizioni,
4. località di **proprietà globali**: cibo ha sottoclassi carne e verdure, ma alcuni esseri mangiano solo una dei due.
5. **combinazioni di classi** che definiscono una nuova classe
6. **proprietà** come transitività, unicità, inversi.

Per sopperire a queste mancanze, è stato introdotto **OWL** (Web Ontology Language) che è un sottoinsieme semantico della logica di primo ordine (**FOL**) a più o meno distanze da questo, in base alla versione.
OWL si basa sulla sintassi Turtle e gli assiomi consistono in uno dei tre seguenti blocchi:

- **Classi** che sono le classi in RDFS. Ci sono 2 classi predefinite:
    - owl:Thing, di cui ogni cosa fa parte
    - owl:Nothing, di cui nulla fa parte.
- **Individui** che sono le istanze in RDFS che possono essere definiti in 2 modi
    - attraverso l'appartenenza a una classe
    - senza appartenere ad una classe e quindi come individuo nominato
- **Proprietà** che sono le proprietà in RDFS e sono di 2 tipi
    - Datatype properties, che riguardano le misurazioni
    - Object properties, riguardano le altre caratteristiche di un oggetto

Per specificare **disuguaglianze** abbiamo:

- owl:disjointWith

Per specificare **uguaglianze** abbiamo:

- per gli individui: owl:sameAs
- per le classi: owl:equivalentClass

In OWL si possono definire degli insiemi in maniera **estensionale**. (ABox)

In OWL si possono utilizzare predicati logici basati su **predicati insiemistici** per strutturare le query:

- And →owl:intersectionOf
- Or → owl:unionOf
- Negation → owl:complementOf

Si possono applicare restrizioni per descrivere classi complesse in OWL:

- **restrizioni su valori**:
    - owl:hasValue applica un determinato valore
    - owl:allValuesFrom specifichiamo un insieme di valori che deve deve essere vero per tutti gli elementi

    - owl:someValuesFrom specifichiamo un insieme di valori che deve essere vero per almeno 1 elemento
- **restrizioni su cardinalità**:
    - owl:cardinality specifica esattamente la cardinalità
    - owl:minCardinality specifica la minima cardinalità
    - owl:maxCardinality specifica la massima cardinalità

Search and Plan

Algoritmi di Ricerca

Un **problema di ricerca** consiste in:

1. uno spazio di stati
2. una funzione successore che prende uno stato e un'azione e restituisce un nuovo stato
3. uno stato iniziale
4. uno o più stati obiettivo.

La **soluzione** di un problema di ricerca è una sequenza di azioni che, applicata allo stato attuale tramite la funzione successore, trasforma lo stato iniziale in uno stato obiettivo.

Una soluzione è **ottimale** se minimizza i costi totali delle azioni compiute.

Un **albero di ricerca** è un albero che ha come radice lo stato iniziale e ciascun nodo figlio è il risultato dell'applicazione di una diversa azione al nodo padre. Per moltissimi problemi non è possibile costruire l'albero di ricerca completo, poiché richiederebbe spazio e tempo esponenziali. In alcuni casi l'albero è persino infinito.

Un algoritmo di ricerca si definisce:

- **completo** se riesce a trovare almeno una soluzione se almeno una ne esiste
- **ottimale** se riesce a trovare almeno una soluzione ottimale.

Legenda per sotto:

- S spazio di stati
- $s \in S$ generico stato
- s_i stato iniziale
- $G \subseteq S$ insieme degli stati obiettivo
- s_g uno stato obiettivo
- A l'insieme di tutte le azioni
- A(s) insieme delle azioni che si possono compiere da s
- $a \in A(s)$ generica azione da s

- $n(s, a)$: $S \times A \to S$ la funzione successore che restituisce lo stato derivato dall'applicazione di a ad s
- $w(s, a)$: $S \times A \to \mathfrak{R}$ la funzione di costo che restituisce il costo di compiere a in s
- $T(S, A)$ l'albero di ricerca, $|T| = |S| + |A|$
- p un cammino da S_i a un S_g
- D la profondità massima per la ricerca in T.
- d la profondità della soluzione più vicina in T.

Backtracking:

- **Descrizione**: Backtracking Search visita l'intero albero di ricerca per un dato problema fino alla profondità D, partendo dallo stato iniziale ed esplorando ricorsivamente tutti gli stati attraverso iterazioni per ogni azione.
- **Spazio**: Richiede $O(D)$ spazio
- **Tempo**: La complessità temporale è $O(T^D)$.
- **Completo**: Se la soluzione si trova oltre D oppure ha stati infiniti l'algoritmo non riesce a trovare soluzione
- **Ottimale**: è ottimale.

Depth First Search (DFS):

- **Descrizione**: è come Backtracking ma si ferma appena trova il primo p.
- **Spazio**: Richiede $O(D)$ spazio
- **Tempo**: La complessità temporale è $O(T^D)$ ma può essere molto inferiore
- **Completo**: Se la soluzione si trova oltre D oppure ha stati infiniti l'algoritmo non riesce a trovare soluzione
- **Ottimale**: Non è ottimale.

Breadth First Search (BFS):

- **Descrizione**: Breadth First Search è un algoritmo di ricerca che esplora un grafo in ampiezza. Partendo da un nodo iniziale, visita tutti i nodi adiacenti prima di spostarsi verso il livello successivo.
- **Spazio**: L'utilizzo di spazio è $O(T^d)$
- **Tempo**: La complessità temporale è $O(T^d)$ meglio di DFS in quanto dipende dalla profondità della soluzione e non del problema
- **Completo**: BFS è completo, garantendo di trovare la soluzione se questa esiste.
- **Ottimale**: BFS non è ottimale.

Iterative Deepening Search (IDS):

- **Descrizione**: Iterative Deepening Search è una strategia che combina i vantaggi di BFS e DFS. Esegue DFS con profondità massima 1 fino a d
- **Spazio**: Richiede $O(D)$ spazio.
- **Tempo**: La complessità temporale è $O(T^d)$.

- **Completo**: È completo
- **Ottimale**: Non è ottimale.

Uniform Cost Search (UCS):

- **Descrizione**: Usa la funzione costo per esplorare cammini a costo minimo. Esplora la frontiera del nodo e successivamente esplora quello con cammino a costo minore indipendentemente dalla frontiera a cui appartengono, ripetendo fino a quando non esplora tutto T entro D. Mantiene quindi tutte le frontiere in memoria
- **Spazio**: Richiede $O(T^D)$ spazio.
- **Tempo**: La complessità temporale è $O(T^D)$.
- **Completo**: È completo
- **Ottimale**: È ottimale

Hill Climbing:

- **Descrizione**: Hill Climbing è un algoritmo euristico che esplora lo spazio di ricerca seguendo il gradiente locale di una funzione obiettivo. Continua finché trova nel suo spazio di ricerca una soluzione con funzione obiettivo migliore.
- **Spazio**: Richiede $O(T^D)$ spazio.
- **Tempo**: La complessità temporale è veloce se converge ad un max locale e lenta se trova stallo. Al massimo $O(T^D)$.
- **Completo**: Non è completo, poiché può rimanere bloccato in minimi locali.
- **Ottimale**: Non è ottimale, poiché potrebbe non raggiungere la soluzione globale ottimale.

Greedy Search:

- **Descrizione**: Greedy Search è un algoritmo euristico che fa scelte basate su una funzione di valutazione locale senza considerare le conseguenze a lungo termine: $h(s)$: $S \rightarrow \Re$ costo ottimale atteso del cammino da un nodo a un nodo obiettivo. Sceglie il percorso che sembra più promettente in ogni passo, ma può portare a soluzioni subottimali.
- **Spazio**: Richiede $O(D)$ spazio.
- **Tempo**: La complessità temporale dipende dall'euristica ma è al massimo $O(T^D)$
- **Completo**: È completo
- **Ottimale**: Non è ottimale, poiché non considera la globalità del problema.

A* Search:

- **Descrizione**: A* Search è un algoritmo di ricerca informato che combina la completezza di BFS con l'efficienza di UCS. Utilizza in principio GS ma con 2 principali differenze:

    - il nodo esplorato non è quello con $h(s)$ minore, ma quello con $w(s,\ a)\ +\ h(s)$, ovvero tiene conto del costo del percorso finora e di una stima del costo rimanente
    - la funzione di euristica deve restituire valori che siano sempre ≤ degli effettivi costi ottimali.
- **Spazio**: Richiede $O(T^D)$ spazio.
- **Tempo**: La complessità temporale è $O(T^D)$
- **Completo**: È completo
- **Ottimale**: È ottimale con un'euristica ammissibile e consistente.

Un'euristica è **consistente** se per ogni nodo N e ciascun successore P di N, il costo stimato per raggiungere l'obiettivo G da N non è maggiore del costo del passo per arrivare a P più il costo stimato per raggiungere l'obiettivo da P:
$h(N)\ \leq\ c(N,P)\ +\ h(P)$
Un'euristica consistente è anche **ammissibile** anche un'euristica coerente, cioè che non sovrastimi mai il costo per raggiungere l'obiettivo: $h(N_i)\ \leq\ h(N_i^*)$

Classical Planning

Un **problema di planning** è simile a un problema di ricerca, ma è più fortemente connesso al concetto di stati, azioni e dimensione temporale.

La struttura di un problema di planning differisce da quella di un problema di ricerca in quanto vengono usati dei **linguaggi di pianificazione** per rappresentare lo spazio di stati e azioni in modo più ricco e generalizzabile.

Il Classical Planning si occupa di studiare quei problemi in cui lo **stato iniziale** è **completamente conosciuto** e le **azioni** sono **deterministiche**.

Poiché per ogni istante di **tempo** t l'**agente** si può trovare o meno in uno **stato** e può compiere o meno un'**azione**, il numero massimo di **ambienti** che si possono incontrare è $O(2^{NT})$.

Codificare l'interezza delle possibilità diventa velocemente non fattibile, per questo si usano i linguaggi di pianificazione per rappresentare tutte le possibilità sensate in modo sintetico.

Un problema di planning in **STRIPS** è rappresentato dalla quadrupla $P = (F, O, I, G)$:
- F è l'insieme degli **atomi**,
- O l'insieme delle **azioni**,
- $I \subseteq F$ gli atomi **iniziali**
- $G \subseteq F$ gli atomi **goal**.

L'obiettivo è, partendo dall'insieme di atomi I, compiere una sequenza di azioni $A \subseteq O$ tali da trasformare gli atomi di I in quelli di G

Formalmente determina un modello tale che:

- Le azioni $o \in O$ sono rappresentate da:
  - $p(o)$ le **precondizioni** affinché si possa compiere o
  - $d(o)$ gli atomi **rimossi** dal compimento di o
  - $a(o)$ gli atomi **aggiunti** dal compimento di o
- La **funzione successore** $n(s, o)$ restituisce $s_n = (s \setminus d(o)) \cup a(o)$
- I **costi** di ciascuna azione sono fissati a 1.
- La soluzione sia **ottimale**

In STRIPS è anche possibile utilizzare predicati, variabili e tipi per sintetizzare la scrittura di atomi e azioni.

L'atto di rimpiazzare le variabili con un valore si chiama istanziazione. Le cosiddette **grounded action** si ottengono istanziando le variabili delle azioni.

PDDL (Planning Domain Description Language) è un linguaggio di pianificazione successivo a STRIPS: ha una sintassi più rigida, ma con maggiore espressività.

In PDDL, i problemi di planning vengono specificati in 2 parti:

- **Dominio**: tipi di oggetti, predicati e azioni che possono esistere all'interno del modello
- **Istanza**: stato iniziale, obiettivo e costanti per ogni tipo

Per risolvere un problema di pianificazione classica l'idea è quella di **mappare** ciascuno stato (insiemi di atomi) in un nodo di un **grafo direzionato**, per poi sfruttare **algoritmi di ricerca** per trovare la soluzione ottimale.

L'algoritmo più comunemente utilizzato è A*. Per la funzione di euristica si costruisce il **problema rilassato** attraverso la **rimozione** delle d**(**o**)**. È poi dimostrabile che tale problema rilassato si risolve in tempo polinomiale. Risolto il rilassamento, ogni stato del problema originale ha come valore dell'euristica il valore del costo dello stato rilassato corrispondente. È da notare che l'euristica così ottenuta non è sempre ammissibile, ma offre le migliori performance

Probabilistic Planning

In generale non è vero che un agente possa avere conoscenza completa dello stato dell'ambiente.

L'idea chiave nel **Probabilistic Planning** è quella di codificare l'incertezza. Per farlo si basa sulla teoria della probabilità: l'agente ha dei **belief** che possono variare a seconda della nuova conoscenza ottenuta tramite i sensori.

Il valore di belief β in una certa proposizione p è 0 ≤ β ≤ 1. β, questa è quindi la misura di quanto l'agente sia sicuro della verità di p. Solamente se β = 0 ∨ β = 1 una nuova conoscenza non modificherà il valore di β.

Formula di Bayes: $P(x|y_1 \wedge \dots \wedge y_n) = \frac{P(y_n|x)P(x|y_1 \wedge \dots \wedge y_{n-1})}{P(y_n|y_1 \wedge \dots \wedge y_{n-1})}$

Markov Localization: Inizialmente l'agente non conosce lo stato attuale. Attraverso i sensori l'agente percepisce parte dello stato attuale, cosa che permette di utilizzare la probabilità condizionata per ricomputare le probabilità dello stato in cui si trova tramite la formula di bayes. Allo stesso modo, l'agente può anche compiere un'azione e percepire come l'ambiente è cambiato, aggiornando i suoi valori. Questo procedimento funziona sotto un assunzione.

Assunzione di Markov: lo stato attuale dipende solamente dalla conoscenza precedente e dall'azione appena compiuta.

Markov Decision Process: Un problema MDP è caratterizzato da:

- Un insieme discreto di stati S
- Un insieme discreto di azioni A
- Una funzione di transizione $T(s, a, s')$: $S \times A \times S \to [0, 1]$ che restituisce la probabilità di trovarsi nello stato s' compiendo l'azione a dallo stato s
- Una funzione di ricompensa $R(s, a, s')$: $S \times A \times S \to \mathfrak{R}$ per ciascuna transizione, che restituisce un valore di "ricompensa" per l'agente alla transizione da s a s' tramite a
- Uno stato iniziale $s_I \in S$
- Opzionalmente degli stati terminali $s_G \in S$

Le sostanziali differenze con un problema di ricerca sono che la **transizione** da uno stato all'altro è **probabilistica** e che invece di cercare di minimizzare i costi si cerca di **massimizzare le ricompense**.

Abbiamo 3 modi per rappresentare lo **spazio** del problema:

- **Grid Based Approaches**: Usata da Markov, utilizza una griglia discreta.
- **Particle Filters**: Usata dai localizzatori Montecarlo, forniscono una stima continua di dove potrebbero trovarsi i punti
- **Kalman Tracking**: fornisce l'area continua con più punti.

Reinforcement Learning: si introduce il concetto di **policy π**, ovvero una funzione che restituisce la miglior azione in base allo stato attuale. Lo scopo del Reinforcement Learning è ottenere una policy π: $S \rightarrow A$ che **massimizzi la ricompensa**.

La differenza chiave tra policy e transizioni tra stati è che una policy π è più flessibile in quanto consente di fare scelte basate sullo stato attuale **senza** essere **vincolata** a una **sequenza** specifica di azioni. Questo è particolarmente utile in ambienti in cui le transizioni tra gli stati possono essere non deterministiche o soggette a incertezza.

Lo scopo di un agente è **massimizzare** la **ricompensa cumulativa**, ovvero $max \sum_{t=0}^{T} R(s_t, a_t, s_t^{'})$ dove T il massimo di istanti temporali

Un agente che cerca di massimizzare la ricompensa cumulativa, potrebbe trovarsi in una situazione di indecisione nel caso di più sequenze di azioni la cui ricompensa totale sia uguale. Per risolvere questa indecisione si introduce il **discount factor γ** in modo da far scegliere all'agente la **massimizzazione più veloce** della ricompensa cumulativa.

La ricompensa al tempo t diventa: $R_t = \sum_{i=t}^{T} \gamma^{i-t} r_i$

Un'altro aspetto di cui tenere conto è proprio quello dell'intrinseca **incertezza delle azioni**, ovvero non è detto che una determinata azione da uno stato porti sempre allo stesso stato successivo. Per questo motivo $R_t = E[r_t] + \gamma R_{t+1}$

L'agente avrà quindi una tabella $Q^{\pi}(s_t, a_t)$ dove ogni cella della tabella rappresenta la ricompensa **attesa** per il compimento dell'azione a nello stato s al tempo t in base alla policy π. L'azione da compiere sarà quindi $Q^{\pi}(s_t, a_{max})$

Si introducono ora 3 concetti fondamentali per l'aggiornamento dei valori nella tabella Q in modo che convergano ai valori ottimali:

- **Loss function**: indica di quanto il valore sia **differente** da quello reale, si basa sul **Bellman Equation/Update**: $l = R(s, a, s^{'}) + \gamma R(s^{'}, a_{max}) - Q(s, a)$
- **Learning rate** α: 0 ≤ α ≤ 1,indica quanto velocemente aggiornare i valori nella tabella Q.
- **Exploration rate** ε: 0 ≤ ε ≤ 1, indica quanto l'agente deve dare importanza all'esplorazione, ovvero compiere un'azione casuale, rispetto al seguire Q. Si abbassa col passare delle iterazioni.

Altri concetti:

- **Bellman Update**: Il Bellman update è una relazione che collega la funzione di valore di uno stato con quella degli stati successivi. La minimizzazione di

una loss function basata su questa relazione aiuta a convergere verso la soluzione ottima.
- **Policy Extraction**: Dopo il Bellman update, la policy extraction estrae la strategia decisionale ottima dalla funzione di valore.
- **Policy Iteration**: L'algoritmo di policy iteration itera tra la valutazione e l'aggiornamento della politica basandosi sulla funzione di valore, convergendo gradualmente verso la soluzione ottima.
- **Fixed Policies**: Durante la policy iteration, le fixed policies sono le strategie valutate e adattate iterativamente in base alla valutazione della funzione di valore e agli aggiornamenti della politica.

Esistono altri algoritmi oltre al Q-Learning, che però è il più utilizzato, per applicare Reinforcement Learning:
- **SARSA λ-learning**: una variante del Q-Learning che aggiorna ad ogni iterazione la tabella Q non solo della singola cella $[s, a]$, bensì anche le ultime k celle che hanno portato alla situazione attuale. Questo è fatto per velocizzare la convergenza all'ottimo e dare importanza anche alla sequenza di azioni che ha permesso di ottenere la situazione attuale.
- **Deep Q-Learning**: applicazione di una Rete Neurale Profonda per approssimare i valori della tabella Q per tabelle che sarebbero altrimenti troppo grandi da computare esplicitamente.
- **Monte Carlo Tree Search**: approccio leggermente diverso dagli altri in quanto si basa su un albero di ricerca per trovare la ricompensa massima conosciuta, per poi, una volta raggiunto uno stato terminale, far back propagation sui nodi che hanno fatto parte del cammino.

AI Subsimbolica

Apprendimento Supervisionato

L'**apprendimento automatico** si divide in due sezioni:
- **Supervisionato**: i dati nel training set sono etichettati, ovvero già classificati. Lo scopo è quindi quello di apprendere una funzione che mappi le caratteristiche dei dati nelle loro classi. Le tecniche principali sono la Regressione e la Classificazione
- **Non Supervisionato**: i dati non sono etichettati e lo scopo sarà quindi apprendere quali e quanti classi ci siano e come assegnare l'appartenenza. Le due tecniche principali sono:
    - Clustering: algoritmo che prova a trovare delle istanze di gruppi nel dataset in base a caratteristiche dello spazio dei dati
    - Anomaly Detection: il sistema deve generalizzare il dato abituale per trovare dati anomali.

Decision Tree

Un **albero di decisione** è un albero in cui gli archi sono rappresentati da valori di features, i nodi foglia sono raggruppamenti per classi, mentre gli altri nodi sono raggruppamenti per features.

Ogni cammino dalla radice a una foglia rappresenta una **regola di decisione**

Non esiste un **unico** albero di decisione.

Trovare l'albero migliore, ovvero con meno nodi e più accurato, è un problema **NP-completo**.

La chiave per costruire un buon albero di decisione è scegliere il **giusto attributo** nel quale eseguire la **branch**, riducendo così le impurità il più possibile.

Tutti gli attuali algoritmi di costruzione di alberi di decisione sono algoritmi **euristici**.

L'**entropia** misura l'impurità di un dataset D, fornendo valori compresi tra 0 e 1, dove più l'entropia è bassa più il dataset è puro.

Dati: *C* numero di classi, *D* dimensione del dataset, $c_i \in C$ elemento i-esimo di *C*, F_i feature e *v* i valori che può assumere, avremo le seguenti formule.

Entropia totale: $e(D) = -\sum_{i=1}^{|C|} P(c_i)\, log(P(c_i))$

Entropia della **feature**: $e(F_i(D)) = \sum_{i=1}^{v} \frac{|D_i|}{|D|} \cdot e(D_i)$

L**'information gain** si ottiene selezionando una feature F_i e il dataset D:

$gain(D, F_i) = entropy(D) - entropy\, F_i(D)$

Il **gain ratio** normalizza il guadagno di informazione tenendo conto del numero di valori distinti dell'attributo.

Per trovare il valore migliore per la quale dividere il dataset possiamo:

- Utilizzare l'information gain o gain ratio
- Ordinare tutti i valori di un attributo continuo in ordine crescente
- Provare tutte le soglie possibili e trovare quella che massimizza il guadagno

Overfitting:

- Buona precisione sui dati di addestramento ma scarsa sui dati di test
- Sintomi: albero troppo profondo o troppe ramificazioni, alcuni valori potrebbero essere presi come anomalie dovute a rumori o valori anomali

Due approcci per evitare l'overfitting:

- **Pre-potatura**: interrompere anticipatamente la costruzione degli alberi
    - Difficile decidere quando interrompere perché non sappiamo cosa potrebbe succedere se continuiamo a far crescere l'albero.
- **Post-potatura**: rimuovere rami o sottoalberi dall'albero completamente cresciuto:
    - Questo metodo è quello comunemente usato.
    - C4.5 (algoritmo per alberi decisionali) utilizza un metodo statistico per stimare gli errori su ciascun nodo per la potatura.
    - È possibile utilizzare un set di convalida anche per la potatura.

Altri problemi nell'apprendimento degli alberi decisionali

- Passaggio dall'albero alle regole alla potatura
- Gestione dei valori mancanti
- Gestire distribuzioni asimmetriche
- Gestire attributi e classi con differenti costi.
- Costruzione di attributi

Classificatori

Per valutare un metodo di classificazione abbiamo i seguenti valori:

- **accuratezza**: numero di classificazioni corrette sul numero totale dei casi
- **efficienza**: tempo per costruire ed usare il modello
- **robustezza**: gestione di rumori e mancanza di valori
- **scalabilità**: efficienza a nuovi dati
- **interoperabilità**: possibilità di capire e fornire informazioni sul modello
- **compattezza**: grandezza dell'albero o dell'insieme di regole

Holdout Set è il metodo utilizzato quando il dataset D è grande: i dati disponibili D sono divisi in due subset:

1. Training Set D_{train}
2. Test Set D_{test}

Si fa apprendimento esclusivamente sul training set e poi si applica il classificatore addestrato al test set. Poiché il test set sarà comunque parte del dataset etichettato, è possibile confrontare le classi assegnate dal classificatore con quelle reali

N-fold Cross-Validation:

- I dati disponibili sono partizionati in n sottoinsiemi disgiunti di uguali dimensioni.
- Si utilizza ciascun sottoinsieme come set di test e si combinano i restanti n-1 sottoinsiemi come set di training per l'apprendimento.
- La procedura viene eseguita n volte, fornendo n precisioni.

- L'accuratezza finale stimata dell'apprendimento è la media delle n precisioni.
- Sono comunemente usate le forme da 5 fold e 10 fold
- Metodo utilizzato quando i dati disponibili non sono molti.

Leave-One-Out Cross Validation: Utilizzato quando il dataset è estremamente piccolo: ogni fold ha un singolo esempio di test e i restanti vengono usati nel training.

Validation Set: i dati disponibili sono suddivisi in tre sottoinsiemi:
1. Training set
2. **Validation set**
3. Test set

Si scelgono k combinazioni di iperparametri e si esegue holdout set su training set e validation set. Per ogni esecuzione si valutano le metriche, per poi scegliere la combinazione di iperparametri che offre le metriche migliori. Infine si ripete un'ultima volta la normale procedura di Holdout Set riaddestrando il classificatore sul training set e valutandolo sul test set.

N-Fold Cross-Validation con Validation Set: si divide il dataset in 2 sottoinsiemi disgiunti, ovvero il training set e il test set. Successivamente si itera k volte, con k numero di combinazioni di iperparametri da studiare, la normale N-Fold Cross Validation, ma solo sul training set. Terminate tutte le iterazioni, si sceglie la combinazione di iperparametri che offre le metriche migliori, per poi eseguire un semplice Holdout Set riaddestrando il classificatore sul training set e valutandolo sul test set.

L'**accuratezza** (errore = 1 - accuratezza) è un unica misura e risulta scomoda nelle classificazioni che coinvolgono dati asimmetrici o sbilanciati. L'elevata accuratezza non rileva intrusioni.

La classe di interesse è comunemente chiamata classe positiva, e il resto classi negative. Applicando un concetto di Vero/Falso otteniamo la **matrice di confusione**:

	Classificato Positivo	Classificato Negativo
Attuale Positivo	TP	FN
Attuale Negativo	FP	TN

La **precisione p** è il numero di esempi positivi diviso per il numero totale di esempi classificati come positivi.

Il **recall r** è il numero di esempi positivi correttamente classificati divisi per il numero totale di esempi positivi nel set di test.

F1-score combina la precisione e il recall in unica misura: è la media armonica dei due. Il valore tende al numero più piccolo dei due.

$$p = \frac{TP}{TP+FP} \qquad r = \frac{TP}{TP+FN} \qquad F = \frac{2pr}{p+r}$$

La curva **ROC** (Receive Operating Characteristic) è un plot del TPR e FPR

$$TPR = \frac{TP}{TP+FN} \qquad FPR = \frac{FP}{FP+TN} \qquad TNR = \frac{TN}{TN+FP}$$

Sensitività = TPR
Specificità = TNR
FPR = 1 - Specificità

L'area sotto alla curva ROC è chiamata **AUC**. Se l'AUC per C_i è maggiore di quella di C_j allora il primo è meglio del secondo.

- Se un classificatore è perfetto, il suo valore AUC è 1
- Se un classificatore fa tutte le ipotesi in modo casuale, la sua AUC è 0,5

kNN e Ensemble

Il metodo **k Nearest Neighbor** (kNN) non crea il modello dai dati di training.

Per **classificare** una nuova 'istanza di test d, kNN definisce l'intorno P come i k valori più vicini a tale d. La stima che il nuovo individuo faccia parte di un insieme c_j è $Pr(c_j \mid d) = \frac{n}{k}$, dove n è il numero di istanze di c_j nell'intorno P.

Non è necessario tempo di training, mentre il tempo per classificare è lineare nella dimensione del set di addestramento per ogni caso di test:

- **Pro**
  - kNN può gestire problemi complessi e arbitrari nei confini decisionali.
  - Nonostante la sua semplicità l'accuratezza della classificazione di kNN può essere abbastanza forte.
- **Cons**
  - kNN è lento nella classificazione
  - kNN non produce un modello comprensibile

```
Unset
kNN(D,d,k)
Compute the distanche between d and every example in D
choose the k examples in D that are nearest to d, denote the set by P
Assign d the class that is the most frequent class in P
```

L'**ensemble** è una tecnica che coinvolge la combinazione di più modelli di apprendimento automatico per migliorare le prestazioni complessive del sistema:

- **Bagging**, se dice una cosa è quasi sempre vera, ma perde alcuni casi
    - Alta precisione
    - Basso recall
    - Training:
        - Dato un dataset D con m esempi, crea un sample $S[i]$ estraendo e sostituendo valori da D. (posso avere elementi ripetuti)
        - Eseguo k volte la procedura ottenendo $S[1]$, ..., $S[k]$ ed effettuo la classificazione su ciascuno di essi con lo stesso algoritmo
    - Test:
        - Le previsioni di tutti i sample vengono poi combinate, di solito attraverso la media o il voto a maggioranza, per ottenere la predizione finale del classificatore
    - Funziona quando l'algoritmo di training non è molto stabile, ovvero con piccoli cambiamenti in input causiamo grandi cambiamenti in output, ma potremmo essere sensibili ai rumori
- **Boosting**, non perde casi ma c'è il rischio di creare falsi positivi
    - Bassa precisione
    - Alto recall
    - Training:
        - Inizialmente viene addestrato un modello debole (quasi random) sul set di addestramento originale
        - Ad ogni esempio viene assegnato un peso che se il modello precedente ha classificato male, vale di più.
        - I prossimi modelli apprendono dando più importanza a ciò che è stato sbagliato dai classificatori precedenti
    - Test per Boosting
        - Le previsioni di tutti i modelli deboli vengono combinate in modo pesato per ottenere una previsione finale. I modelli che hanno prestazioni migliori ricevono più peso nella combinazione finale.
    - Anche lui richiede che il training non sia stabile e sembra suscettibile ai rumori.

Un esempio comune di classificatore bagging è il **Random Forest**, che utilizza una collezione di alberi decisionali addestrati su set di dati diversi per ottenere una previsione più accurata e stabile.

Reti Neurali

Il **Percettrone** consiste di connessioni in input, operazioni interne che vengono applicate agli input, e connessioni in output.

Ciascuna connessione in input ha un peso. La somma dei pesi è l'operazione che viene applicata e il risultato prende il nome di **attivazione**. Infine l'attivazione viene passata a tutte le connessioni in output.

Si ha quindi che in **input** il percettrone prende una serie di valori e moltiplica ciascuno per il **peso** ad esso associato. Successivamente si fa una sommatoria di tutti i valori ottenuti e il risultato viene passato in output: $O = \sum_{i=1}^{I} f_i(x) w_i$

Cosa fare se il dataset non permette una **separazione lineare**? L'idea è quella di mappare le feature in uno spazio dimensionale maggiore, in modo da creare una separazione che sia lineare. La scelta della mappatura diventa il nuovo problema. Possiamo far **apprendere** alla macchina qual è la **mappatura** di volta in volta migliore?

Il **Deep Learning** utilizza più percettroni per le feature in input e usa le loro attivazioni come input ad altri percettroni.

- Più profondo → migliore precisione → maggiori costi

Le **Deep Neural Networks** hanno una particolarità rispetto alle reti composte da soli percettroni: sono in grado di imparare anche separazioni non lineari. Questo perché, dopo aver fatto le sommatorie degli input per i pesi, prima di mandare in output, si utilizza una funzione di attivazione non lineare.

Una comune funzione di attivazione è la **ReLU**: $max(0, v)$ dove v è il risultato della sommatoria.
Un'altra spesso usata è la **Sigmoide**: $\frac{1}{1+e^{-v}}$.
Il **softmax** è una funzione di attivazione dove si converte un vettore di valori reali in una distribuzione di probabilità normalizzata.

$$Softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

L’unica parte di una rete che può variare nel corso del tempo è l’insieme dei **pesi** e saranno quindi ciò che varia durante l’apprendimento in modo da permettere la convergenza.

Per permettere ad una rete di apprendere, è necessario definire una misura della qualità delle sue predizioni. Entra in gioco quindi la **Loss Function**. Ovviamente non ne esiste una sola e, come per la funzione di attivazione, la più adatta dipende dall’applicazione della rete.

Una volta stabilita la Loss Function più adatta, è quindi necessario scegliere come **aggiornare** i pesi in modo da riflettere il “punteggio” ottenuto dalla Loss Function con l’obiettivo di migliorarlo. In genere la Loss Function comprende in qualche modo una differenza tra l’output della rete e l’effettivo target e, quindi, bisogna cercare di **minimizzare** il suo valore.

Le DNN funzionano attraverso due processi principali:

- **Forward Propagation**: In questa fase, l'input viene passato attraverso la rete. Durante questo processo, ogni neurone in un dato strato della rete esegue fondamentalmente due operazioni principali:
    - **Sommatoria Ponderata**: Ogni input al neurone viene moltiplicato per un peso corrispondente. Questi prodotti vengono poi sommati insieme, spesso con l'aggiunta di un termine noto come bias.
    - **Funzione di Attivazione**: Dopo la sommatoria ponderata, il risultato viene passato attraverso una funzione di attivazione non lineare.
- **Backpropagation**: Dopo che l'output è stato generato, la rete calcola l'errore, che è la differenza tra l'output previsto e quello effettivo. Durante la backpropagation, questo errore viene propagato all'indietro nella rete, partendo dall'ultimo strato fino al primo. Durante questo processo, i pesi e i bias della rete vengono aggiustati in modo da minimizzare l'errore, attraverso il Gradient Descent

Gradient Descent (GD): Si tratta del metodo più basilare di ottimizzazione. Aggiorna tutti i pesi nella stessa direzione basandosi sul calcolo del gradiente dell'intero dataset di training. I principali problemi del GD sono:

- **Lento e inefficace** su **dataset** molto **grandi**, poiché richiede di calcolare il gradiente sull'intero dataset prima di fare un singolo aggiornamento.
- **Progressione oscillante**: quando la funzione ha una discesa ripida in una direzione, mentre diminuisce poco in un’altra, la progressione del GD oscilla facendo fatica a raggiungere il minimo.
- **Stallo su minimi locali**: facilmente si blocca nei punti di minimo locale della funzione, poiché attorno ad essi la direzione di discesa punta sempre verso il minimo locale stesso.

Stochastic Gradient Descent (SGD): Una variante di GD che aggiorna i pesi utilizzando il gradiente calcolato su un singolo esempio di training o su un piccolo

batch. Questo rende l'aggiornamento più **veloce** e può aiutare a **evitare minimi locali**, ma può introdurre una certa varianza nell'andamento dell'ottimizzazione.

SGD + **Momentum**: Questo metodo aggiunge un termine di momentum agli aggiornamenti. Aiuta ad **accelerare** SGD nelle **direzioni rilevanti** e **ammorbidisce le oscillazioni**, conducendo a convergenze più veloci. Memorizza le modifiche dei pesi precedenti e le utilizza per influenzare gli aggiornamenti correnti, aumentando così la stabilità.

AdaGrad: Adatta il **learning rate** (ampiezza dell'aggiornamento dei pesi) per ogni parametro individualmente, basandosi sulla **frequenza** con cui un parametro viene aggiornato durante l'allenamento. È efficace per dati sparsi ma può avere problemi nelle fasi avanzate dell'allenamento a causa di una riduzione continua del learning rate.

RMS Prop: Modifica AdaGrad per affrontare il suo problema di diminuzione aggressiva del learning rate. Utilizza una **media mobile del quadrato dei gradienti** per normalizzare l'aggiornamento di ogni peso. Questo permette un adattamento più efficace del learning rate durante l'allenamento.

Adam: Combina le idee di momentum e RMS Prop. Mantiene una **media mobile** sia dei **gradienti** passati (simile a momentum) sia del **quadrato dei gradienti** (simile a RMS Prop). Questa combinazione rende Adam molto efficace in molte applicazioni di reti neurali.

Inizializzare una rete è molto importante, dato che il valore iniziale dei pesi e degli iperparametri influenza moltissimo la velocità di apprendimento e la convergenza o meno della rete.

Una prima scelta per inizializzare i **pesi iniziali** potrebbe essere quella di porli tutti a **0**. Questa scelta non è consigliata dato che se tutti i pesi hanno lo stesso valore iniziale, tutte le unità all'interno dello stesso strato produrranno lo stesso output durante la fase di forward propagation. Di conseguenza, durante la fase di backpropagation, tutti i pesi nello stesso strato riceveranno lo stesso segnale di errore e si aggiornano in modo identico. Ciò può portare a **problemi di simmetria** e alla **mancanza di diversità nel modello**, **limitando** la **capacità di apprendimento**.

Un'altra strategia comune è inizializzare i **pesi** con valori **randomici** presi da una distribuzione ampia, come una distribuzione normale con media zero e varianza non piccola. Questa inizializzazione consente ai pesi di avere una maggiore varietà iniziale, che può essere utile quando la rete neurale ha una maggiore complessità e profondità. Tuttavia, **valori troppo grandi** possono portare a **rallentare l'apprendimento** o causare **problemi di stabilità** durante la fase di aggiornamento dei pesi.

Gli **iperparametri** ricoprono un ruolo ancor più fondamentale e a seconda del modello di rete, della Loss Function e dell'ottimizzazione scelta, possono variare in numero.

Esistono quindi delle linee guida e delle procedure per aiutare a trovare gli iperparametri migliori:

1. Verificare la **loss** iniziale
2. Addestrare il modello su un **piccolo campione** del dataset
3. Trova **learning rate** che riduce la perdita
4. Fare **cross validation** con diversi iperparametri
5. Fare **training** e ripetere passo 4 con i migliori valori
6. Osservare le **curve di perdita e precisione**
7. Ripetere dal passo 5

Un modo efficace per scegliere le combinazioni di iperparametri è quello di scegliere dei limiti per i valori di ciascun iperparametro ad occhio. Questi valori saranno raffinati durante le varie iterazioni. Si scelgono poi dei valori campione e si esegue la cross validation, ottenendo dei nuovi valori migliori che saranno i nuovi limiti.

La regolarizzazione è una tecnica molto diffusa per **evitare l'overfitting**. Per contrastarlo, ci sono diverse strategie ampiamente utilizzate:

- **Modifica della Loss Function**: viene aggiunto un nuovo termine alla Loss Function. Questo termine extra penalizza i modelli che hanno pesi troppo grandi, limitando così l'eccessiva specializzazione sui dati di addestramento. Ci sono due tipi principali:
    - **L1**: Aggiunge un termine proporzionale alla somma assoluta dei pesi del modello. Questo può portare ad alcuni pesi che si azzerano, favorendo la sparsità.
    - **L2**: Aggiunge un termine proporzionale alla somma dei quadrati dei pesi. Questo riduce il valore di tutti i pesi, ma non li annulla, prevenendo l'overfitting mantenendo la complessità del modello più controllata.
- **Early stopping**: fermare l'addestramento del modello quando le prestazioni sul set di validazione smettono di migliorare. Monitorando le metriche di valutazione sul set di validazione durante l'addestramento, è possibile individuare il punto in cui il modello inizia ad adattarsi eccessivamente ai dati di addestramento.
- **Dropout**: una tecnica di regolarizzazione che mira a rendere il modello più robusto e meno dipendente da singoli neuroni specifici. Durante l'addestramento, il dropout disattiva casualmente alcuni neuroni e imposta i loro output a zero. Ciò costringe il modello a imparare caratteristiche utili da diverse combinazioni di neuroni e previene l'eccessiva specializzazione.

La Convoluzione è un'operazione matematica che consiste nel prendere una funzione e applicarla iterativamente a **porzioni** di un'altra funzione.

Il concetto fondamentale è che, data un'immagine, si decide una **finestra** con associata dei **pesi** e la si fa **scorrere** sull'immagine, in modo da ottenere un risultato.

Le classiche DNN prendono una sequenza lineare di dati in input, mentre le immagini sono composte da dati che sono relazionati tra loro anche nello **spazio** in cui si trovano. Per ovviare a questo problema sono state create le prime **Reti Neurali Convolutive** (CNN).

Le CNN, o meglio gli strati convolutivi, fanno uso della procedura convolutiva.

1. L'immagine viene descritta in **3 dimensioni**, ovvero larghezza, altezza e profondità. La **profondità** è data dai **canali**, per esempio un'immagine in bianco e nero avrà 1 canale i cui valori per ogni pixel vanno da 0 a 255. Un'immagine a colori avrà 3 canali ciascuno che va da 0 a 255.
2. Scelta una finestra, spesso chiamata anche **kernel** la si scorre per tutta l'immagine. Questa è la parte convolutiva, infatti ad ogni applicazione della finestra sull'input **si moltiplicano i valori dell'input** inquadrati dalla finestra con i **pesi della finestra stessa**, si **sommano** tra di loro e si **salva** il risultato in un unico nodo nello strato successivo.
3. Questo significa che si avranno tot **filtri**, ovvero set di pesi per finestre, e per ciascuno si farà convoluzione sull'input come descritto sopra. Questo permette anche di astrarre l'input dall'immagine, ovvero è possibile applicare strati convoluzionali all'output di altri strati convoluzionali, dato che l'output sarà anch'esso descritto in 3 dimensioni (larghezza, altezza, numero filtri).

È da notare che la procedura avviene su un input con **padding**, ovvero si aggiunge una cornice ai dati in modo da poter applicare la convoluzione un numero maggiore di volte e permettendo di dare più importanza anche ai dati verso i margini dell'input.

Gli strati convoluzionali più vicini all'input (all'immagine) imparano a riconoscere feature di **basso livello**, mentre gli strati convoluzionali più vicini all'output (il risultato della rete) imparano a riconoscere feature di **alto livello**. Questo fenomeno è dato dal fatto che i pesi del kernel usato sono in percentuali e fatti in modo che la loro somma dia sempre 100%.

Spesso tra uno strato convoluzionale e l'altro viene aggiunto uno strato di **Max Pooling**, ovvero uno strato che si occupa di ridurre la dimensionalità dei dati facendo una semplice operazione di scelta del massimo in una certa finestra.

Poiché per addestrare una CNN con risultati soddisfacenti è richiesta un'enorme quantità di dati, una pratica molto diffusa è quella del **trasferimento dell'apprendimento**. Questa pratica consiste nell'**estrarre** le attivazioni (embeddings) da uno o più strati di una CNN già addestrata e usarli come input della rete che si vuole addestrare.

Apprendimento Non Supervisionato

Clustering Analysis

Il **Cluster Analysis** è un tipo di apprendimento non supervisionato dove si hanno dati non etichettati e lo scopo è apprendere quali e quanti classi ci siano e come assegnare l'appartenenza per ciascun elemento.

Ogni elemento è descritto da delle features, tuttavia la classe di appartenenza è **ignota**. Si cerca quindi di creare dei raggruppamenti tra gli oggetti tali che oggetti di uno stesso gruppo siano simili tra loro, mentre oggetti di gruppi distinti siano diversi tra loro.

Per ogni algoritmo di clustering sarà quindi fondamentale definire una metrica di distanza tra elementi del dataset, cosicché si possa cercare di minimizzare le distanze intra-cluster e massimizzare le distanze inter-cluster.

Differenze Clustering e Classificazione:

- Per la classificazione:
    - Le classi devono essere conosciute
    - Un numero relativamente grande di esempi etichettati deve essere disponibile
    - Vogliamo occuparci di nuovi elementi per i quali vogliamo sapere a quale classe esso appartiene
    - Possiamo evitare alcuni dati etichettati per valutare la prestazione del classificatore
- Per il clustering:
    - **Non** necessariamente **sappiamo** quanti **classi** sono presenti nel dataset
    - Potremmo non saper dare **l'interpretazione** di un cluster
    - Valutare un cluster è più complicato

Clustering è più **esplorativa** del dato. Il clustering potrebbe essere usato nelle prime fasi di un problema, quando i dati sono disponibili ma non etichettati, per poi utilizzare Classificazione.

Inoltre esistono 2 tipi di clustering:

- **Partitional** Clustering: i cluster sono disgiunti

- **Hierarchical** Clustering: i cluster sono annidati tra loro, formando un albero gerarchico

K-Means

K-Means è un clustering partizionale. Ho dei punti in uno spazio R-dimensionale e i dati devono essere numerici. Determina la partizione dei dati in k cluster: ogni classe ha un centro chiamato **centroide**, che **spesso non fa parte del dataset**. k è specificato dall'utente.

L'algoritmo di k-means è banale, dato k:

1. Peschiamo k **seeds** (dati random) che saranno i miei centroidi iniziali
2. **Assegno** ogni punto del dataset al centroide più vicino
3. **Ricomputo** i centroidi in base ai punti di appartenenza al cluster
4. Se il **criterio di convergenza** (arresto) non è stato raggiunto, si parte da 2

Criterio di convergenza:

- non c'è riassegnamento di punti a nuovi cluster
- non c'è cambiamento nei centroidi
- metrica che calcola la compattezza del cluster: somma degli errori quadrati. (**SSE**)

Pros:

- Semplice da comprendere ed implementare
- Efficienti con complessità **lineare** in tempo O(tkn) in quanto:
    - n sono il numero di punti
    - k è il numero di cluster (solitamente piccolo)
    - t è il numero di iterazioni (solitamente piccolo)

Osservazione: termina in un ottimo locale sse SSE è utilizzato.

Cons:

- L'algoritmo è applicabile solo se la media è definita. Per i dati categoriali il centroide è rappresentato dai valori più frequenti
- L'utente deve specificare k.
- L'algoritmo è sensibile agli **outliers**, punti molto lontani dagli altri.
- L'algoritmo è sensibile ai seed iniziali.
    - Ci sono alcuni metodi per scegliere bene i seed
- L'algoritmo k-means non è adatto a scoprire cluster che non sono iper ellissoidi

Possiamo alleviare gli effetti degli outliers:

- Rimuovere durante il processo di clustering i dati che sono molto distanti dai centroidi rispetto ad altri punti dati.

- Eseguire un campionamento casuale. Così facendo nel campionamento scegliamo solo un piccolo sottoinsieme di dati in modo tale che la possibilità di selezionare un valore anomalo è molto piccola.

La qualità di un clustering è molto difficile da valutare perché non conosciamo i cluster corretti. Ci sono alcuni metodi:
- Studiare i centroidi e i suoi spread
- Regole da un albero decisionale
- Costruire una matrice di confusione sulla quale computare misurazioni e confrontarle con dati etichettati di una classificazione.

```
Unset
k-means(k, D)
Choose k datya points as the initial centroids
do
       for each data point x in D
              compute the distance from x to each centroid
              assign x to the closest centroid
       recompute centroids using the current cluster membership
while the stopping criterion is met
```

DBScan

L'idea di fondo di questo approccio è quello che i cluster siano regioni **dense** di elementi delimitate da zone poco dense o vuote. Questa idea permette di identificare cluster di qualsiasi forma.

L'algoritmo più diffuso è **DBSCAN**. Si definiscono:
- **ε-vicinato** di un elemento x come tutti gli elementi entro un raggio ε da x, x incluso
- **Alta Densità**: un ε-vicinato che contiene almeno **MinPts** elementi.

I valori di ε e MinPts vengono scelti arbitrariamente, sono iperparametri dell'algoritmo.

Si definiscono inoltre:
- **Core**: un elemento si definisce core point se il suo ε-vicinato è ad alta densità
- **Border**: un elemento di definisce border point se il suo ε-vicinato non è ad alta densità, ma si trova nell'ε-vicinato di un core point.
- **Outlier**: un elemento si definisce outlier/noise point se non è né core né border.

L'ultima definizione è infine quella di **Density Reachability**: un elemento q si definisce raggiungibile per densità da p se p è un punto core e q si trova nel suo ε-vicinato. Ne consegue che questa relazione non è necessariamente simmetrica. Inoltre un elemento può essere raggiungibile per densità **direttamente**, ovvero come descritto sopra, oppure **indirettamente**, ovvero esiste un cammino di elementi raggiungibili per densità che porta da p a q.

DBScan è **sensibile** ai parametri: Al variare di ε, i cluster possono cambiare in quanto la dimensione della sfera di ricerca influisce su quali punti sono considerati vicini e quindi su quali vengono inclusi nei cluster. Se ε è troppo piccolo, molti punti potrebbero essere considerati come rumore, e i cluster saranno più numerosi e più piccoli. Se ε è troppo grande, punti distanti potrebbero essere collegati nello stesso cluster, portando a cluster più grandi e meno distinti.

```
Unset
DBSCAN(D, eps, MinPts)
C = 0
for each unvisited point P in dataset D
       mark P as visited
       NeighborPts = regionQuery(P, eps)
       if sizeof(NeighborPts) < MinPts
              mark P as NOISE
       else
              C = next cluster
              expandCluster(P, NeighborPts, C, eps, MinPts)

expandCluster(P, NeighborPts, C, eps, MinPts)
add P to cluster C
for each point P' in NeighborPts
       if P' is not visited
              mark P' as visited
              NeighborPts' = regionQuery(P', eps)
              if sizeof(NeighborPts') >= MinPts
                     NeighborPts = NeighborPts joined with NeighborPts'
       if P' is not yet member of any cluster
                     add P' to cluster C

regionQuery(P, eps)
return all point within P's eps-neighborhood including P
```