Fondamenti di Intelligenza Artificiale

Sommario

1 – Introduzione all'Intelligenza Artificiale

L'Intelligenza Artificiale (IA) è un campo della scienza che mira a creare sistemi in grado di eseguire compiti che richiedono intelligenza umana. Il suo sviluppo

ha portato innovazioni in numerosi settori, dall'automazione industriale alla personalizzazione di contenuti nei social media. Esempi comuni includono:

- *Google Maps*: Fornisce percorsi ottimizzati tenendo conto del traffico in tempo reale.
- *Sistemi di raccomandazione*: Utilizzati da piattaforme come Netflix o YouTube per suggerire contenuti personalizzati.

L'IA è sviluppata per migliorare diversi aspetti della vita e della società, risolvendo problemi complessi che sono difficili da affrontare manualmente. Le motivazioni principali per lo sviluppo di IA includono:

- *Ottimizzazione delle risorse*: Permettere una gestione più efficiente delle risorse.
- *Velocità di apprendimento*: Creare sistemi che imparano e si adattano più velocemente rispetto agli esseri umani.
- *Risoluzione di problemi complessi*: Affrontare sfide che richiederebbero troppo tempo e risorse per essere risolte dagli esseri umani.
- *Curiosità scientifica*: Spingere i confini della conoscenza su come l'intelligenza può essere simulata o replicata artificialmente.

È possibile classificare l'IA in tre gruppi:

- *Artificial Super Intelligence*, che è task-independent ed ha capacità di pensiero ben oltre quelle di un essere umano;
- *Artificial General Intelligence*, che è task-independent ed ha la capacità di imparare ed applicare capacità di problem solving con la flessibilità di un essere umano;
- *Artificial Narrow Intelligence*, che è task-specific e goal-oriented e soltanto in quello specifico compito può agire più in fretta e meglio di una mente umana.

I Large Language Model, e più in generale i Foundation Model, sono un primo tentativo di creare una AGI.

1.1 – Definizioni di Intelligenza Artificiale

L'IA può essere definita in diversi modi, a seconda di dove venga posta l'enfasi quando si definisce "intelligenza". Esistono quattro principali approcci:

- *Pensare umanamente*: L'obiettivo è simulare i processi cognitivi umani, riproducendone i meccanismi. Per comprenderli, si può scegliere fra tre diversi modelli: introspezione, che consiste nell'analisi dell'interiorità di pensieri e pulsioni, sperimentazione psicologica, che applica il metodo sperimentale di Newton per indagare i processi cognitivi, ed imaging celebrale, che prevede una mappatura di struttura, funzione e

farmacologia del sistema nervoso. Gli studi di psicologia e neuroscienze sono una parte fondamentale.
- *Pensare razionalmente*: Questo approccio si concentra sul pensiero logico e deduttivo, cercando di costruire sistemi che seguano regole formali della logica per risolvere problemi. I sillogismi aristotelici rappresentano un primo tentativo di codifica formale di un pensiero corretto.
- *Agire umanamente*: L'IA viene progettata per replicare il comportamento umano. Questo approccio è alla base del Test di Turing, che misura l'intelligenza di una macchina sulla base della sua capacità di imitare un comportamento umano in un'interazione.
- *Agire razionalmente*: L'approccio degli agenti razionali si concentra sulla creazione di agenti che agiscono in modo ottimale, basandosi sulle informazioni a loro disposizione. L'obiettivo è prendere decisioni razionali per raggiungere un obiettivo specifico, indipendentemente dal fatto che tali azioni simulino o meno il pensiero umano.

1.2 – Il Test di Turing

Il Test di Turing, proposto da Alan Turing nel 1950, è uno dei primi tentativi di definire un criterio per l'intelligenza artificiale e si basa su un esperimento chiamato gioco dell'imitazione. Durante il test, una persona C (interrogatore) interagisce con una macchina A ed un altro essere umano B tramite una tastiera o altri mezzi di comunicazione che escludono l'elemento visivo. C pone domande ad entrambi e deve determinare chi dei due sia la macchina. Se C non riesce a distinguere tra la macchina e l'umano in un numero significativo di casi, allora si dice che la macchina ha passato il Test di Turing e può essere considerata intelligente.

Il Test di Turing ha avuto un impatto profondo sulla filosofia e sullo sviluppo dell'IA e ha comportato svariate implicazioni in campo etico e filosofico sulla definizione di intelligenza. La definizione su cui ci si basa è puramente comportamentale, per cui non ci interessa né del processo con cui la macchina ha fornito le sue risposte né se ne ha davvero compreso il contenuto. Una delle conseguenze di ciò è il dibattito che ha portato a due nuove definizioni:

- *IA forte* e *IA debole*: L'IA forte sostiene che le macchine possano sviluppare una vera intelligenza, simile a quella umana, mentre l'IA debole suggerisce che le macchine possono solo imitare l'intelligenza senza comprenderla realmente.

Tra le questioni etiche sollevate troviamo invece interrogativi inerenti alla responsabilità delle proprie azioni ed al loro status nella società.

Seppur con svariate limitazioni, il test ha indubbiamente aperto la strada a questioni più profonde sulla relazione tra comportamento esterno e processi interni di pensiero.

1.3 – Dall'imitazione alla razionalità: il cambio dell'approccio

Con il tempo, la comunità scientifica si è spostata verso criteri più pratici di intelligenza, concentrandosi sul costruire agenti razionali che prendono decisioni ottimali e risolvono problemi in modo efficiente, per diverse ragioni:

- *Limitazioni nel replicare l'intelligenza umana*: gli studi su come gli esseri umani pensano ed agiscono non erano sufficienti a comprendere la complessità della mente umana e a modellarla.
- *Focalizzazione sull'efficienza e l'ottimizzazione*: più l'IA ha trovato applicazione in problemi complessi del mondo reale, più l'enfasi si è spostata verso l'efficienza del processo decisionale.
- *Risultati pratici migliori*: ottenuti con efficacia molto maggiore quando il sistema si basa sul pensiero razionale e la logica formale, piuttosto che sull'imitazione del pensiero umano.
- *Flessibilità*: in quanto gli agenti razionali non devono simulare esattamente il comportamento umano, ma possono considerare strade alternative quando prendono decisioni per giungere al risultato ottimale.

Questo cambio di direzione ha portato a significativi progressi nell'efficacia e nell'efficienza degli algoritmi di IA, consentendo loro di risolvere problemi complessi in diversi settori industriali e scientifici.

2 – Agenti Intelligenti

Un agente è un sistema che percepisce l'ambiente circostante tramite sensori e agisce su di esso tramite attuatori. Gli agenti possono essere classificati in diverse categorie a seconda delle loro capacità e del contesto in cui operano:

- *Umani:* I sensori includono occhi, orecchie, ecc.; gli attuatori includono mani, gambe, tratto vocale, ecc.
- *Robotici:* I sensori includono telecamere e telemetri; gli attuatori includono motori e bracci meccanici.
- *Software*: I sensori includono input da tastiere, contenuti di file, pacchetti di rete; gli attuatori includono la visualizzazione delle informazioni, la modifica di file e l'invio di pacchetti di rete.

2.1 – Comportamento di un Agente

Si indica con **percezione** l'insieme di input percettivi dell'agente in un dato istante, mentre con **sequenza percettiva** la storia completa di tutto ciò che l'agente ha percepito nella sua esistenza.
Il comportamento di un agente è descritto da una **funzione agente,** che mappa una sequenza di percezioni in una specifica azione. In altre parole, la funzione agente determina l'azione che l'agente esegue in base a ciò che ha percepito. Questo può essere rappresentato matematicamente come:

Funzione Agente : P -> A

dove P è l'insieme delle percezioni ed A è l'insieme delle azioni. La funzione agente è implementata da un **programma agente,** che riceve come input la percezione corrente e restituisce un'azione da eseguire.

2.2 – Agenti Razionali

Un agente razionale è un agente che, per ogni sequenza di percezioni, sceglie l'azione che massimizza il valore atteso della sua misura di prestazione, date le informazioni a sua disposizione. La razionalità di un agente dipende da:

- La misura di prestazione adottata, che dovrebbe essere progettata sulla base dell'effetto che si desidera osservare sull'ambiente.
- La conoscenza pregressa dell'agente sull'ambiente.
- Le azioni che l'agente può compiere.
- La sequenza di percezioni ricevute fino al momento attuale.

È importante notare che un agente razionale non conosce il risultato effettivo delle sue azioni ed agisce solo in base alle informazioni possedute. Un agente onnisciente, al contrario, conosce anche il loro risultato.
Per migliorare la propria razionalità, un agente può intraprendere azioni di information gathering, ovvero azioni finalizzate alla raccolta di nuove informazioni sull'ambiente circostante.

- Un esempio è quello dell'esplorazione, che porta l'agente a conoscere l'ambiente sconosciuto in cui si trova per ottenerne una migliore comprensione e prendere decisioni più informate.
- Un altro meccanismo è l'apprendimento, cioè la capacità di imparare coppie di percezioni-azioni in base a scelte fatte in precedenza. Questo consente all'agente di adattarsi più velocemente e più facilmente a nuovi ambienti o situazioni, rendendolo anche più autonomo.

2.3 – Struttura di un Agente

La struttura di un agente può essere descritta come una combinazione di:

- *Architettura*: L'hardware o la piattaforma su cui l'agente opera (ad esempio, un computer con sensori e attuatori).
- *Programma agente*: Implementa la funzione agente e decide l'azione da eseguire in base alla percezione corrente.

L'obiettivo principale della progettazione di agenti intelligenti è creare programmi che producano comportamento razionale con il minor numero di regole/codice.

2.4 – Tipi di Agenti

Esistono vari tipi di agenti intelligenti, ognuno con capacità diverse:

- *Agenti Reattivi Semplici*: che intraprendono azioni basate soltanto sulla percezione corrente, ignorando la storia percettiva. Per determinare le regole condizione-azione è però necessario un ambiente completamente osservabile, poiché nei parziali c'è rischio di entrare in cicli infiniti: una soluzione è aggiungere una certa componente casuale da invocare nel caso di dati mancanti.
- *Agenti Reattivi Basati su Modello*: dotati di un modello interno dell'ambiente che permette loro di tenere traccia dello stato del mondo e prevedere l'effetto delle proprie azioni.
- *Agenti Basati su Obiettivi*: che pianificano le proprie azioni in base anche a come le loro scelte influenzeranno il raggiungimento degli obiettivi che desiderano raggiungere; sono molto flessibili, poiché la conoscenza può essere modificata e adattata.
- *Agenti Basati sull'Utilità*: che, in aggiunta agli obiettivi, utilizzano una funzione di utilità per misurare la "desiderabilità" di uno stato, permettendo loro di scegliere tra obiettivi contrastanti in modo ottimale.
- *Agenti che Apprendono*: capaci di diventare sempre più competenti in ambienti inizialmente sconosciuti tramite l'apprendimento. Sono formati da quattro elementi: quello di apprendimento, quello esecutivo, quello critico ed infine il generatore di problemi.

2.5 – PEAS: Definizione degli Agenti

Un ambiente è un'istanza di un problema di cui gli agenti razionali rappresentano le soluzioni e possono essere descritti con la formulazione **PEAS:**

- *Performance*: La misura di prestazione per valutare l'operato dell'agente.
- *Environment*: Descrizione degli elementi che formano l'ambiente.
- *Actuators*: Gli attuatori a disposizione dell'agente per eseguire azioni.
- *Sensors*: I sensori tramite cui l'agente riceve gli input percettivi.

2.6 – Proprietà degli Ambienti

Gli ambienti possono essere caratterizzati da diverse proprietà:

- *Completamente o Parzialmente osservabile*: Se i sensori di un ambiente forniscono accesso a tutte le informazioni rilevanti sull'ambiente o meno.
- *Deterministico o Stocastico*: Se lo stato successivo dell'ambiente è determinato soltanto da quello corrente e dalle azioni dell'agente o meno.
- *Episodico o Sequenziale*: Se l'esperienza dell'agente è divisa in blocchi singoli in cui eseguire un'unica azione che dipende dall'episodio stesso o meno.
- *Statico o Dinamico*: Se l'ambiente è invariato mentre l'agente sta deliberando o se invece cambia. Se a cambiare è soltanto il punteggio delle prestazioni dell'agente allora è detto semi-dinamico.
- *Discreto o Continuo*: Se l'ambiente fornisce un numero limitato di percezioni e azioni chiaramente distinte o meno.
- *Singolo o Multi-agente*: Se l'agente opera da solo o in presenza di altri agenti. Nel secondo distinguiamo tra ambienti competitivi e cooperativi.

3 – Formulazione di Problemi

Nel contesto degli agenti risolutori di problemi, lo stato del mondo viene rappresentato in modo atomico. Ciò significa che ogni stato è considerato indivisibile e privo di struttura interna. Questo processo di astrazione è indispensabile, poiché altrimenti gli agenti intelligenti sarebbero soverchiati dalla complessità del mondo reale. Un'astrazione è valida se possiamo espandere ogni soluzione astratta ad una nel mondo più dettagliata, mentre è utile se eseguire ogni azione nella soluzione è più facile che nel problema originale.

- Esempio: un agente che cerca di viaggiare da una città all'altra considera ogni città come uno stato, senza analizzarne le caratteristiche interne.

La formulazione di un problema in IA è un passaggio cruciale per identificare le azioni che l'agente deve compiere per raggiungere un obiettivo. Un problema può essere descritto formalmente da queste componenti:

- ***Stato Iniziale*****,** cioè il punto di partenza dell'agente. Ad esempio, nel problema del viaggio in Romania, lo stato iniziale potrebbe essere “in(Arad)".
- ***Azioni*****,** cioè l'insieme di azioni applicabili in ogni stato. Ad esempio, “Go(Sibiu)", “Go(Timisoara)".
- ***Modello di transizione*****,** che definisce il modo in cui ogni azione modifica lo stato. Ad esempio, eseguire “Go(Zerind)" nello stato “in(Arad)" porterà allo stato “In(Zerind)" .

- ***Test obiettivo*****,** cioè la verifica del raggiungimento dell'obiettivo. Ad esempio, lo stato finale può essere “In(Bucarest)".
- ***Costo di cammino*****,** cioè la funzione che calcola il costo per passare da uno stato all'altro.

3.1 – Algoritmi di Ricerca

La risoluzione dei problemi tramite ricerca prevede che l'agente individui una sequenza di azioni che porti dallo stato iniziale allo stato obiettivo. Gli algoritmi di ricerca seguono tre fasi principali:

- ***Formulazione del problema*****.** Il problema deve essere descritto formalmente, identificando le componenti che ne caratterizzeranno l'ambiente di esecuzione.
- ***Ricerca di una soluzione.*** Una volta formulato il problema, l'agente deve trovare una sequenza di azioni che porti dallo stato iniziale allo stato obiettivo. Esistono diversi approcci per la ricerca, tra cui:
  - *Ricerca non informata,* in cui l'agente esplora lo spazio degli stati senza alcuna conoscenza a priori sull'obiettivo. Esempi sono la ricerca ad ampiezza e la ricerca in profondità.
  - *Ricerca informata,* in cui l'agente utilizza delle euristiche per guidare la ricerca. Un esempio è l'algoritmo A*.
- ***Esecuzione della sequenza di azioni*****.** Una volta trovata la sequenza di azioni che conduce all'obiettivo, l'agente la esegue per raggiungere lo stato desiderato. Se la soluzione trovata è subottimale, l'agente potrebbe dover riformulare il problema o rieseguire la ricerca.
  - Se durante l'esecuzione delle azioni l'agente ignora le sue percezioni si parla di *sistema a ciclo aperto,* poiché si rompe il ciclo tra lui e l'ambiente. L'approcio funziona solo in ambienti deterministici e completamente osservabili.
  - Se l'agente invece aggiorna costantemente la sua conoscenza con nuove percezioni, consentendogli di adattare la sequenza di azioni, si parla di *sistema a ciclo chiuso.* Questo è l'approccio da seguire in ambienti stocastici o parzialmente osservabili e rende l'agente più flessibile ed in grado di rispondere a degli imprevisti.

3.2 – Un esempio: Formulazione del Problema delle 8 Regine

Il problema delle 8 regine richiede di piazzare 8 regine su una scacchiera in modo che nessuna possa attaccarne un'altra. Esistono due principali formulazioni di questo problema, ognuna con implicazioni diverse in termini di complessità computazionale.

Il primo modo di definire il problema è tramite una formulazione incrementale. In particolare, il problema sarebbe descritto come segue:

- Descrizione. Si aggiungono progressivamente le regine sulla scacchiera, cominciando dallo stato vuoto. Ogni stato corrisponde a una scacchiera con un numero crescente di regine.
- Azioni: Piazzare una regina in una casella vuota.
- Complessità: Questa formulazione esplora uno spazio di ricerca molto ampio, in quanto non impone vincoli immediati sul posizionamento delle regine. La ricerca dovrà esaminare circa 1.8 × 10^14 possibili configurazioni per trovare una soluzione valida.
- Svantaggi: La mancanza di vincoli rende questa formulazione molto inefficiente per problemi con molti stati, come il problema delle 8 regine. Il numero di combinazioni da esaminare può crescere rapidamente, rendendo la ricerca complessa e lunga.

In alternativa, potremmo descrivere il problema tramite una formulazione a stato completo. Quindi:

- Descrizione: In questa formulazione, tutte le regine sono già piazzate sulla scacchiera, e il compito è spostarle in modo che nessuna minacci un'altra.
- Azioni: Spostare una regina all'interno della colonna in cui si trova, se minacciata.
- Complessità: Questa formulazione riduce drasticamente lo spazio di ricerca. Invece di esplorare tutte le possibili configurazioni, si esaminano solo configurazioni in cui ogni regina è già piazzata in una colonna separata. La complessità scende a circa 1.6 × 10^7 possibili configurazioni, rendendo la ricerca molto più gestibile.
- Bisogna notare che con l'aggiunta di vincoli è possibile ridurre ulteriormente lo spazio degli stati.

Altri esempi di problemi di ricerca sono:

- Problema del Commesso Viaggiatore: Ottimizzare un percorso che visiti una serie di città una sola volta minimizzando la distanza complessiva. Questo è un esempio di un problema di ottimizzazione NP-completo, in cui lo spazio di ricerca cresce esponenzialmente con il numero di città.
- Problemi di Navigazione Robotica: Estensione del problema di ricerca di percorsi a spazi continui o anche multidimensionali, come nel caso di robot dotati di braccia o gambe. Anche qui, la definizione corretta del problema e l'uso di astrazioni e vincoli riduce la complessità della ricerca.

4 – Algoritmi di Ricerca Non Informata

Gli algoritmi di ricerca non informata non dispongono di informazioni aggiuntive sugli stati oltre a quella fornita dalla definizione del problema. Operano cercando una sequenza di azioni che porti dallo stato iniziale all'obiettivo, costruendo un albero di ricerca.

4.1 – Ricerca di Soluzioni

Dopo lo stato iniziale, il primo passo è verificare che l'obiettivo non sia stato raggiunto, dopodiché si considerano le varie azioni, espandendo lo stato corrente e generando così nuovi dati. Si definiscono nodi foglia quei nodi che, in un certo momento della ricerca, sono privi di figli: il loro insieme forma la frontiera. È possibile incorrere in cammini ciclici, ovvero cammini in cui uno stato è ripetuto: sono casi particolari dei cammini ridondanti, cioè quei cammini in cui esiste più di un modo per passare da uno stato all'altro, e possono portare a dei fallimenti in alcuni algoritmi di ricerca. Possiamo quindi immaginare come primo miglioramento generico l'aggiunta di una struttura dati in cui mantenere l'insieme degli stati già esplorati.

4.2 – Strategie di Ricerca Non Informata

La strategia di ricerca è definita secondo una modalità di espansione dei nodi, ovvero scegliendo l'ordine in cui i nodi sono espansi. Abbiamo quattro indicatori:

- *Completezza,* cioè se l'algoritmo garantisce di trovare una soluzione, se esiste.
- *Ottimalità,* cioè se l'algoritmo garantisce di trovare la soluzione ottima.
- *Complessità temporale,* cioè di quanto tempo ha bisogno l'algoritmo per trovare una soluzione.
- *Complessità spaziale,* cioè di quanta memoria ha bisogno l'algoritmo.

Segue un'analisi delle principali strategie di ricerca.

4.2.1 – Ricerca in Ampiezza (Breadth-First Search)

La ricerca in ampiezza esplora gli stati a partire dalla radice espandendo tutti i nodi a una determinata profondità prima di passare alla successiva. Viene utilizzata una coda FIFO per la gestione della frontiera.
Pro:

- Completo: Trova sempre una soluzione se questa esiste.
- Ottimo: Trova la soluzione più breve in termini di numero di passaggi (quando il costo di cammino è uniforme).

Contro:

- Complessità spaziale: Richiede molta memoria per mantenere tutti i nodi della frontiera (O(b^d), dove b è il fattore di ramificazione e d la profondità della soluzione).
- Complessità temporale: Il tempo necessario è esponenziale rispetto alla profondità della soluzione (O(b^d)).

4.2.2 – Ricerca a Costo Uniforme (Uniform-Cost Search)

La ricerca a costo uniforme espande i nodi basandosi sul costo di cammino minimo dal nodo iniziale. Viene utilizzata una coda a priorità ordinata sul costo.
Pro:

- Completo: Trova sempre una soluzione se esiste.
- Ottimo: Trova la soluzione con il costo di cammino minimo, indipendentemente dal numero di passi.

Contro:

- Complessità temporale e spaziale: Può esplorare molti nodi, soprattutto se le differenze tra i costi dei cammini sono piccole, rendendo la ricerca lenta (O(b^(C*/ϵ)), dove C* è il costo della soluzione ottima ed ϵ è il costo minimo delle azioni).

4.2.3 – Ricerca in Profondità (Depth-First Search)

La ricerca in profondità esplora un cammino fino alla profondità massima, tornando indietro solo quando non ci sono più nodi da espandere. Utilizza una coda LIFO.
Pro:

- Complessità spaziale ridotta: Richiede meno memoria rispetto alla ricerca in ampiezza.
- Efficiente quando le soluzioni sono molto profonde.

Contro:

- Non completo: Può esplorare all'infinito in spazi di ricerca con profondità infinita o cicli.
- Non ottimo: Può trovare una soluzione non ottimale, ritornando nodi molto lontani dalla radice.

4.2.4 – Ricerca in Profondità Limitata (Depth-Limited Search)

La ricerca in profondità limitata è una variante della ricerca in profondità, in cui viene imposto un limite alla profondità massima di esplorazione. Questo evita i cicli infiniti, ma introduce la possibilità di non trovare una soluzione se questa si trova oltre la profondità limite.
Pro:

- Evita i cicli infiniti imposti dalla ricerca in profondità pura.
- Migliora l'efficienza rispetto alla ricerca in profondità nei problemi con soluzioni vicine alla radice ma con ramificazioni profonde.

Contro:

- Non completo: Se la soluzione si trova oltre il limite di profondità fissato, l'algoritmo non la troverà.
- Non ottimo: Potrebbe trovare una soluzione non ottimale.

4.2.5 – Ricerca ad Approfondimento Iterativo (Iterative Deepening Search)

La ricerca ad approfondimento iterativo combina i vantaggi della ricerca in ampiezza e della ricerca in profondità limitata. L'algoritmo esegue più iterazioni della ricerca in profondità limitata, aumentando progressivamente il limite ad ogni iterazione. Questo garantisce che l'algoritmo esplori tutti i cammini con profondità crescente, trovando così la soluzione più vicina alla radice senza esaurire la memoria.

Pro:

- Completo e ottimo come la ricerca in ampiezza.
- Complessità spaziale simile alla ricerca in profondità, poiché richiede di memorizzare solo un cammino alla volta.

Contro:

- Tempo di esecuzione: Ripete più volte la ricerca a profondità crescente, il che comporta l'espansione ripetuta degli stessi nodi a profondità diverse.

4.2.6 – Ricerca Bidirezionale (Bidirectional Search)

La ricerca bidirezionale consiste nel cercare simultaneamente da stato iniziale ed obiettivo, espandendo i due alberi di ricerca finché le frontiere non si incontrano. Questo approccio può ridurre di molto il numero di nodi da esplorare poiché espande solo metà dello spazio di ricerca da ciascuna direzione.

Pro:

- Molto efficiente: Riduce drasticamente il numero di nodi da esplorare, con complessità O(b^d/2), dove d è la profondità della soluzione.

Contro:

- Difficile da implementare: Richiede che il problema sia facilmente invertibile, cioè che sia possibile cercare “all'indietro" dallo stato obiettivo verso lo stato iniziale.

4.3 – Conclusioni

Gli algoritmi di ricerca non informata offrono diverse strategie per esplorare lo spazio degli stati. A seconda del problema, è possibile scegliere un algoritmo

adatto in base a criteri di completezza, ottimalità, complessità temporale e spaziale.

5 – Algoritmi di Ricerca Informata

Gli algoritmi di ricerca informata utilizzano conoscenze specifiche del problema, come funzioni euristiche, per migliorare l'efficienza della ricerca.
L'approccio generale è detto *best-first:* si tratta sempre di una coda a priorità, che viene ordinata in base ad una **funzione di valutazione f(n)** che comprende, tra i suoi parametri, anche una **funzione euristica h(n)** che indica il costo stimato del cammino più conveniente dallo stato n allo stato obiettivo.

5.1 – Ricerca Greedy Best-First

Espande il nodo che sembra essere il più vicino all'obiettivo, basandosi esclusivamente sulla funzione euristica h(n), senza considerare il costo accumulato del cammino. È goloso nel senso che cerca di avvicinarsi il più rapidamente possibile all'obiettivo.
Pro:

- Veloce: Spesso trova rapidamente una soluzione, specialmente quando l'euristica è ben progettata.
- Semplice da implementare rispetto a molti altri algoritmi di ricerca informata.

Contro:

- Non ottimale: Non garantisce che la soluzione trovata sia la più economica in termini di costo complessivo del cammino.
- Non completo: L'algoritmo potrebbe non trovare una soluzione se entra in un ciclo o se si trova bloccato in un vicolo cieco.

5.2 – Algoritmo A*

Combina il costo accumulato g(n) con una stima euristica h(n) del costo per raggiungere l'obiettivo, espandendo i nodi in base alla funzione f(n) = g(n) + h(n). Questo garantisce sia completezza che ottimalità, a patto che l'euristica sia ammissibile e consistente, ed anche che vengano esplorati prima i nodi che sembrano portare alla soluzione più economica.
Pro:

- Completo: Trova sempre una soluzione se una soluzione esiste.
- Ottimale: A* trova la soluzione con il costo minimo, poste le precondizioni indicate sopra.

Contro:

- Complessità spaziale: Richiede molta memoria per mantenere tutti i nodi esplorati e quelli nella frontiera.
- Complessità temporale: Può diventare inefficiente in spazi di ricerca molto grandi, per quanto sia il migliore tra gli algoritmi best-first.

5.3 – Beam Search

Limita il numero di nodi esplorati mantenendo solo k nodi più promettenti a ogni livello, con k detto ampiezza del raggio e determinato tramite euristica.
Pro:

- Minor uso della memoria: Rispetto ad A*, Beam Search mantiene in memoria solo un numero limitato di nodi a ogni livello.

Contro:

- Non ottimale: Beam Search può ignorare cammini che porterebbero a soluzioni più ottimali.
- Non completo: Potrebbe non trovare una soluzione, soprattutto se il numero di nodi mantenuti è troppo limitato.

5.4 – Iterative Deepening A* (IDA*)

Si tratta di una variante dell'algoritmo A* che utilizza l'approfondimento iterativo: usa la stessa funzione di valutazione f(n) = g(n) + h(n), ma esplora progressivamente lo spazio di ricerca con limiti crescenti di f(n), similmente a come l'approfondimento iterativo espande progressivamente la profondità.
Pro:

- Ottimale: Trova la soluzione più economica se l'euristica h(n) è ammissibile.
- Efficienza spaziale: Utilizza meno memoria rispetto ad A* poiché non mantiene tutti i nodi nella frontiera.

Contro:

- Complessità temporale: Può dover riesplorare più volte gli stessi nodi, soprattutto in spazi di ricerca con molti stati.
- Richiede una buona euristica: L'efficacia di IDA* dipende fortemente dalla qualità dell'euristica utilizzata.

5.5 – Ricerca Best-First Ricorsiva (RBFS)

Si tratta di un algoritmo che utilizza una strategia simile ad A*, ma impiega uno spazio di memoria lineare rispetto alla profondità della soluzione. Espande il nodo migliore in termini di f(n), gestendo un solo percorso alla volta e memorizzando informazioni sui nodi scartati per un eventuale ripristino.

Pro:

- Ottimale: Trova la soluzione ottimale se l'euristica h(n) è ammissibile.
- Utilizza meno memoria rispetto ad A*, con una complessità spaziale lineare.

Contro:

- Complessità temporale: Poiché può dover riesplorare più volte i nodi scartati, può risultare inefficiente in alcuni casi.
- Meno pratico rispetto ad A*: La ricorsione può comportare una complessità temporale elevata in problemi molto profondi.

5.6 – Simplified Memory Bounded A* (SMA*)

L'idea alla base di questa variante dell'algoritmo A* è quella di utilizzare meglio la memoria disponibile. Si procede proprio come A* e, quando lo spazio finisce, l'algoritmo libera il nodo peggiore e ne memorizza il valore nel suo nodo padre. In questo modo, è possibile ricordare l'informazione sul cammino migliore del sotto-albero dimenticato e lo si può rigenerare quando tutti gli altri cammini offrono soluzioni peggiori. È completo se la profondità del nodo obiettivo è inferiore alla dimensione della memoria espressa in nodi ed è anche ottimale, ma in problemi difficili potrebbe dover passare continuamente da un nodo all'altro, aumentando molto la complessità temporale.

5.7 – Conclusioni

Gli algoritmi di ricerca informata offrono strategie avanzate per esplorare lo spazio degli stati, utilizzando funzioni euristiche per guidare la ricerca verso la soluzione in modo più efficiente. A seconda del problema, è possibile scegliere l'algoritmo più adatto in base a diversi criteri come:

- *Completezza*: La capacità dell'algoritmo di trovare una soluzione se una soluzione esiste. Algoritmi come A* e IDA* garantiscono completezza, a condizione che l'euristica sia ben definita.
- *Ottimalità*: La capacità dell'algoritmo di trovare la soluzione più economica. A* è ottimale quando l'euristica è ammissibile, mentre algoritmi come Greedy Best-First e Beam Search non garantiscono l'ottimalità.
- *Complessità temporale*: Il tempo necessario per trovare una soluzione. L'uso di buone euristiche può ridurre il tempo di ricerca, come accade per A* e Greedy Best-First, ma algoritmi come IDA* possono riesplorare nodi più volte, aumentando la complessità temporale.
- *Complessità spaziale*: La quantità di memoria utilizzata dall'algoritmo. Algoritmi come A* possono richiedere molta memoria per memorizzare

tutti i nodi esplorati, mentre IDA* e RBFS riducono significativamente l'uso della memoria sacrificando in parte l'efficienza temporale.

6 – Algoritmi di Ricerca Locale

Gli algoritmi di ricerca locale sono utilizzati per risolvere problemi in cui il percorso verso la soluzione non è rilevante, ma ciò che conta è la configurazione finale. Questi algoritmi memorizzano solo lo stato corrente e tentano di migliorarlo iterativamente, rendendoli particolarmente utili per problemi di ottimizzazione. Data la loro natura usano poca memoria: hanno complessità spaziale spesso costante. Bisogna notare che, visto il loro modo di operare, non esiste un vero e proprio "test obiettivo", in quanto ogni stato è già una soluzione in sé. A guidare la ricerca di un miglioramento è la **funzione obiettivo,** che ha quindi un ruole cruciale nel funzionamento di questi algoritmi.

Un esempio classico che illustra l'assenza di un test obiettivo negli algoritmi di ricerca locale è il problema delle N regine. In questo problema, lo stato iniziale può essere una configurazione completa di N regine su una scacchiera, ma in cui alcune regine si attaccano a vicenda. Ogni configurazione è già una soluzione “valida" in quanto contiene N regine piazzate, ma l'obiettivo è minimizzare o eliminare gli attacchi. In questo contesto, si utilizza una funzione obiettivo che misura il numero di attacchi tra le regine. L'algoritmo di ricerca locale cerca quindi di ridurre questo numero iterativamente, migliorando la configurazione corrente. La ricerca non termina quando viene trovato uno stato specifico, ma quando si raggiunge una configurazione in cui il numero di attacchi non può più essere ridotto (ad esempio, quando arriva a zero).

Andando più nel dettaglio, un algoritmo di ricerca locale cercherà di ottimizzare la soluzione corrente muovendosi verso stati “vicini", ovvero stati raggiungibili applicando una singola modifica alla soluzione corrente. Tali stati vicini formano la cosiddetta **struttura dei vicini**: per ciascuna soluzione s, viene definito un insieme di soluzioni vicine N(s), che rappresenta i possibili stati raggiungibili con una singola mossa.

Per comprendere meglio il comportamento degli algoritmi di ricerca locale, possiamo immaginare lo spazio degli stati come un panorama, dove l'asse x rappresenta gli stati raggiungibili; e l'asse y rappresenta il valore della funzione obiettivo per ciascuno stato. Alcune caratteristiche dello spazio degli stati:

- ***Massimo Locale:*** Un picco che rappresenta una soluzione sub-ottimale.
- ***Piatto (plateau):*** Una regione dello spazio con stati vicini che hanno tutti lo stesso valore.

- ***Spalla:*** Un plateau che presenta uno spigolo in salita, offrendo la possibilità di miglioramento.
- ***Massimo Globale:*** La soluzione ottimale del problema.

6.1 – Algoritmo Hill-Climbing

L'Hill-Climbing è uno degli algoritmi di ricerca locale più semplici. L'idea alla base è quella di scalare il “panorama" dello spazio degli stati fino a raggiungere un massimo o minimo globale. Viene anche detto *ricerca locale greedy.*
Pro:

- È rapido, specialmente se la funzione obiettivo è ben definita.

Contro:

- Può bloccarsi in un massimo locale, in un plateau o in una cresta.

Per migliorare l'Hill-Climbing, è possibile fare alcune modifiche:

- Hill-Climbing con **mosse laterali**: consente di fare mosse laterali su un plateau, ovvero di accettare mosse che non migliorano immediatamente la funzione obiettivo, nella speranza di trovare una soluzione migliore in seguito.
- Hill-Climbing **con prima scelta**: può generare le mosse a caso fino a trovarne una migliore dello stato corrente. Questa strategia è molto buona quando uno stato ha molti successori, poiché l’algoritmo riuscirà a navigare meglio lo spazio di ricerca.
- Hill-Climbing **stocastico**: sceglie casualmente tra le mosse migliori, evitando di selezionare sempre la mossa più ovvia e aumentando così la diversificazione.
- **Riavvio casuale**: consente alla ricerca di ripartire da un punto a caso dello spazio di ricerca.
- **Simulated Annealing**: combina gli ultimi due approcci, permettendo, con una probabilità decrescente nel tempo, l'accettazione di soluzioni peggiori. Può evitare massimi locali, ma la sua efficacia dipende fortemente dalla funzione di raffredamento e dalla temperatura iniziale.

6.2 – Ricerca Local Beam

La ricerca Local Beam tiene traccia di k stati inizialmente generati casualmente: ad ogni iterazione genera tutti i successori possibili e seleziona i k migliori tra essi. A differenza dell'Hill-Climbing con riavvio casuale, qui le ricerche non sono indipendenti, ma sono collegate tra loro. È per questo motivo che aumenta la possibilità di navigare il panorama degli stati verso stati più promettenti, ma

d'altro canto c'è il rischio di avere carenza di diversificazione, con ricerche che si concentrano molto in fretta in regioni di spazio piccole.

7 – Algoritmi Genetici

Gli algoritmi genetici (GA) sono algoritmi di ricerca e ottimizzazione ispirati ai meccanismi evolutivi osservati in natura, come la selezione naturale, l'incrocio e la mutazione. Questi algoritmi sono stati sviluppati per risolvere problemi complessi in cui i metodi tradizionali possono risultare inefficaci, specialmente in spazi di ricerca molto grandi o altamente non lineari.
L'idea alla base degli algoritmi genetici è di trattare ogni possibile soluzione ad un problema come un individuo di una popolazione. Ad ogni iterazione, chiamata generazione, vengono selezionati gli individui migliori per la riproduzione, creandone così una nuova con soluzioni potenzialmente migliori. L'obiettivo è migliorare progressivamente la qualità delle soluzioni, fino a trovare o la migliore possibile oppure una accettabile in un tempo ragionevole.
Le teoria evolutiva di Darwin è alla base di questi algoritmi, in particolare:

- gli individui più adatti hanno maggiori probabilità di sopravvivere;
- il processo di riproduzione mescola i geni dei genitori;
- i cambiamenti dovuti a mutazioni casuali possono portare a nuove parti dello spazio delle soluzioni.

Prima di proseguire, è importante rimarcare che i GA sono una meta-euristica, ossia un insieme di strategie generali per risolvere problemi di ricerca ed ottimizzazione. Si tratta di un approccio di alto livello che non richiede una conoscenza approfondita della struttura del problema specifico e si può dunque applicare ad una vasta gamma di problemi. Le meta-euristiche sono progettate per esplorare ampi spazi di ricerca che possono essere troppo vasti o complessi per metodi di ricerca esaustiva o deterministici ed introducono casualità per migliorare le loro chance di trovare soluzioni ottime. I GA, dunque, non sono problem-specific né garantiscono l'ottimalità, ma sono altamente flessibili, cioè semplici di modificare e adattare.

Questi algoritmi si basano su una **funzione obiettivo,** che assegna un punteggio (o **fitness**) a ciascuna soluzione in base a quanto è buona rispetto ai criteri definiti dal problema. Possiamo scomporli in diversi passaggi:

- **Inizializzazione**: La ricerca inizia con la creazione di una popolazione iniziale di individui, ciascuno dei quali rappresenta una soluzione potenziale al problema. Ogni individuo è generalmente rappresentato come una stringa di bit o un vettore di valori numerici (cromosomi), che codifica i parametri della soluzione. Questa popolazione iniziale può

essere generata casualmente oppure basandosi su una stima preliminare delle soluzioni buone.

- **Funzione di fitness**: La funzione di fitness valuta quanto un individuo è “adatto": più alta, migliore è la soluzione. Questa funzione guida l'evoluzione della popolazione selezionando gli individui migliori per la riproduzione, rappresentando un qualunque criterio legato al problema.
- **Selezione**: Nella fase di selezione, gli individui con una fitness maggiore hanno più probabilità di essere scelti per la riproduzione. Metodi comuni per la selezione includono:
  - ***Roulette Wheel Selection***: Gli individui vengono scelti proporzionalmente alla loro fitness relativa, dunque i migliori hanno una maggiore probabilità di essere selezionati.
  - ***Torneo***: Un sottoinsieme casuale di individui compete in un “torneo", e l'individuo con la fitness migliore viene selezionato.
  - ***Selezione Stocastica con Campionamento Universale***: Una variante della *roulette* che riduce la varianza nei risultati della selezione. In questo metodo, un numero di frecce equidistanti viene lanciato sulla “ruota" della roulette, selezionando più individui contemporaneamente. Questo garantisce una selezione più uniforme e riduce la possibilità che individui con fitness molto alta o molto bassa siano selezionati in modo sproporzionato.
  - ***Selezione per Troncamento***: Viene scelto un sottoinsieme dei migliori individui della popolazione (es. i migliori 10%) e solo questi individui vengono utilizzati per la riproduzione. Questo metodo impone una forte pressione selettiva, poiché i migliori individui dominano completamente la generazione successiva.
  - ***Selezione Basata su Ranking***: Gli individui vengono ordinati in base alla loro fitness e le probabilità di selezione sono determinate in base al rango piuttosto che al valore assoluto della fitness. Questo metodo riduce l'effetto di differenze estreme nella fitness tra gli individui, limitando il dominio dei pochi individui migliori.
- **Crossover**: Il crossover è un operatore genetico che combina il patrimonio genetico di due “genitori" per produrre “figli". Esistono diverse strategie di crossover, tra cui:
  - ***Crossover a singolo punto***: Viene scelto un punto di crossover sulla stringa dei genitori e le parti successive vengono scambiate tra i due. Questo metodo è semplice da implementare e mantiene inalterata buona parte del materiale genetico di entrambi i genitori.
  - ***Crossover a due punti***: Due punti di crossover vengono scelti e le sezioni comprese vengono scambiate. Questa tecnica fornisce una maggiore mescolanza dei geni rispetto al crossover a singolo punto e permette una più ampia esplorazione dello spazio delle soluzioni.

    - ***Crossover uniforme***: Ogni gene del figlio viene scelto casualmente tra i geni corrispondenti dei due genitori, indipendentemente dalla loro posizione nella stringa genetica. Questo metodo garantisce una mescolanza maggiore di informazioni genetiche, ma può introdurre una forte perturbazione nelle strutture già ottimizzate dei genitori.
    - ***Crossover aritmetico***: Utilizzato principalmente per problemi con rappresentazioni numeriche, il crossover aritmetico combina i valori dei genitori attraverso operazioni matematiche. Ad esempio, dati due genitori con valori p1 e p2, il figlio potrebbe avere un valore $c = \alpha p1 + (1 - \alpha)p2$, dove α è un fattore di mescolanza tra 0 e 1. Questo metodo è utile per mantenere una continuità nei parametri.
    - ***Crossover a k punti***: Si scelgono k punti sulla stringa genetica, e si alternano le sezioni di stringhe tra i genitori per formare il figlio. Maggiore è k, maggiore è la mescolanza tra i genitori, ma ciò può anche aumentare la casualità del processo.
    - ***Crossover ordinato***: Utilizzato principalmente per problemi di ottimizzazione in cui l'ordine degli elementi è importante (ad esempio, il problema del commesso viaggiatore), il crossover ordinato preserva l'ordine relativo dei geni di un genitore e riempie i restanti geni con quelli dell'altro genitore, mantenendo il loro ordine.
    - ***Crossover di ciclo***: Anch'esso utilizzato nei problemi in cui l'ordine degli elementi è importante, il crossover di ciclo costruisce il figlio identificando cicli tra i genitori. Il primo ciclo è preso da un genitore, e i cicli successivi sono presi dall'altro genitore, preservando così l'ordine.
    - ***Crossover a segmento casuale***: Viene scelto un segmento casuale della stringa dei genitori e scambiato per produrre il figlio. Questa tecnica è utile per introdurre variazioni locali limitate, mantenendo comunque una parte consistente del patrimonio genetico originale.
- **Mutazione**: La mutazione è un operatore che introduce variazioni casuali negli individui. Questo operatore è essenziale per mantenere la diversità genetica all'interno della popolazione e per evitare che l'algoritmo si blocchi in massimi locali, esplorando così nuove aree dello spazio di ricerca che il crossover da solo non coprirebbe. Le più comuni sono:
    - ***Mutazione a bit singolo***: In una rappresentazione binaria, un bit viene selezionato casualmente e il suo valore viene invertito (da 0 a 1 o da 1 a 0). Questa è la forma più semplice di mutazione ed è ampiamente utilizzata in problemi con codifica binaria.
    - ***Mutazione a più bit***: Vengono selezionati casualmente k bit da invertire all'interno della stringa genetica. Questo introduce variazioni più ampie nel cromosoma rispetto alla mutazione a singolo bit.
    - ***Mutazione gaussiana***: Utilizzata per rappresentazioni numeriche, questa tecnica altera i valori dei geni aggiungendo un valore casuale

derivato da una distribuzione gaussiana con media zero. Il risultato è una variazione continua e “leggera" dei parametri, ideale per problemi di ottimizzazione in cui la precisione dei valori numerici è cruciale.

- ***Mutazione di scambio***: In una rappresentazione permutativa (come nel problema del commesso viaggiatore), due geni vengono selezionati casualmente e scambiati di posizione. Questo tipo di mutazione è utile per problemi in cui l'ordine degli elementi è significativo, poiché altera la sequenza degli elementi senza cambiare la loro presenza.
- ***Mutazione per inversione***: In una stringa genetica, viene selezionato un segmento casuale e l'ordine dei geni in quel segmento viene invertito. Questo metodo è particolarmente utile per problemi di ottimizzazione dell'ordine, poiché altera localmente la disposizione degli elementi.
- ***Mutazione casuale completa***: In una rappresentazione numerica, un gene selezionato casualmente viene sostituito da un nuovo valore casuale all'interno del dominio permesso per quel gene. Questa tecnica può introdurre cambiamenti drastici in una soluzione, rendendola utile in situazioni in cui è necessario esplorare rapidamente una parte diversa dello spazio di ricerca.
- ***Mutazione per permutazione***: Utilizzata in rappresentazioni a permutazione (ad esempio, nel problema del commesso viaggiatore), un sottoinsieme di geni viene selezionato casualmente e il loro ordine viene permutato casualmente. Questo tipo di mutazione introduce nuove combinazioni di elementi senza alterare il contenuto della soluzione.
- ***Mutazione uniformemente distribuita***: In una rappresentazione numerica, un gene viene alterato di una quantità casuale derivata da una distribuzione uniforme, piuttosto che da una distribuzione gaussiana. Questo metodo può introdurre variazioni più ampie rispetto alla mutazione gaussiana, esplorando porzioni più distanti dello spazio di ricerca.
- ***Mutazione per inserimento***: Un gene viene selezionato casualmente, rimosso dalla sua posizione originale e reinserito in una posizione differente all'interno della stessa stringa. Questo tipo di mutazione è utile nei problemi di ottimizzazione in cui l'ordine degli elementi è importante.
- ***Mutazione a scala***: Utilizzata per la rappresentazione numerica, la mutazione a scala introduce variazioni più piccole man mano che l'algoritmo avanza verso le generazioni finali. Questo riduce la variabilità verso la fine della ricerca, permettendo una maggiore precisione quando si è vicini alla soluzione ottimale.

    - ***Mutazione adattativa***: In questo metodo, la probabilità di mutazione varia dinamicamente durante l'evoluzione in base alla diversità della popolazione o al progresso dell'algoritmo. Se la popolazione diventa troppo omogenea, la probabilità di mutazione aumenta per introdurre nuova variabilità, mentre se la popolazione è già altamente diversificata, la probabilità di mutazione viene ridotta.
- **Terminazione**: L'algoritmo genetico può essere terminato quando viene soddisfatta una condizione di arresto, come l'aver raggiunto un numero fisso di generazione o la soglia di fitness desiderata oppure quando non si ha nessun miglioramento significativo dopo un certo numero di generazioni.

7.1 – Progettazione di algoritmi genetici:

Progettare un algoritmo genetico efficace richiede una comprensione approfondita delle caratteristiche del problema e un'attenta configurazione dei vari componenti dell'algoritmo.

7.1.1 – Comprendere il problema e lo spazio delle soluzioni

Il primo passo per configurare correttamente un GA è comprendere a fondo il problema da risolvere:

- *Tipo di problema*: Identificare se il problema è continuo, discreto, combinatorio o un mix di questi. Ad esempio, problemi di ottimizzazione continua richiederanno una rappresentazione numerica, mentre problemi combinatori, come il commesso viaggiatore, potrebbero richiedere una rappresentazione basata su permutazioni.
- *Funzione obiettivo*: Valutare se la funzione obiettivo è semplice da calcolare o computazionalmente costosa. Se la valutazione della fitness è molto costosa, potrebbe essere utile ridurre la dimensione della popolazione per eseguire meno valutazioni.

7.1.2 – Dimensione della popolazione

La dimensione della popolazione è un parametro critico che influisce sull'efficienza del GA:

- *Popolazioni piccole*: Consentono un'evoluzione più rapida, ma possono portare a una perdita di diversità genetica e a una convergenza prematura verso soluzioni sub-ottimali.
- *Popolazioni grandi*: Offrono una maggiore esplorazione dello spazio di ricerca e riducono il rischio di convergenza prematura, ma richiedono più tempo e risorse computazionali.

- Regola pratica: Se non hai una conoscenza specifica del problema, inizia con una popolazione moderata (es. 50-100 individui) e regola il parametro in base ai risultati iniziali. Un buon approccio è quello di eseguire test con diverse dimensioni di popolazione e confrontare le performance.

7.1.3 – Scelta della strategia di selezione

La scelta della strategia di selezione può influenzare significativamente la velocità di convergenza e la qualità delle soluzioni finali:

- *Selezione a Roulette*: Utilizza questa tecnica se vuoi che gli individui con fitness minore abbiano comunque una probabilità di essere selezionati. È utile per mantenere la diversità genetica nelle prime fasi dell'evoluzione.
- *Selezione a Torneo*: Se vuoi aumentare la pressione selettiva usa la selezione a torneo. Se la tua popolazione sta convergendo troppo rapidamente, puoi ridurre la dimensione del torneo per mantenere una maggiore variabilità.
- *Selezione Elitaria*: Integra una piccola percentuale di selezione elitaria (ad esempio, il 5%) per assicurarti che i migliori individui non vengano persi nel processo di crossover e mutazione.

7.1.4 – Frequenza di crossover e mutazione

La scelta delle probabilità di crossover e mutazione determina l'equilibrio tra esplorazione e sfruttamento dello spazio delle soluzioni:

- *Crossover*: Generalmente, una probabilità di crossover elevata (70-90%) è consigliata per favorire la combinazione delle caratteristiche migliori degli individui.
- *Mutazione*: La mutazione ha un ruolo cruciale nell'introdurre nuova diversità genetica. Inizia con una bassa probabilità di mutazione (0.1-1%) e aumenta gradualmente se osservi che la popolazione sta convergendo troppo rapidamente su una soluzione sub-ottimale.

7.1.5 – Scelta delle tecniche di crossover e mutazione

La scelta delle tecniche di crossover e mutazione dipende dalla rappresentazione del problema:

- *Crossover*: Sperimenta con crossover a singolo punto e crossover a due punti per rappresentazioni binarie, mentre per rappresentazioni numeriche, il crossover aritmetico può aiutare a mantenere la continuità tra le soluzioni.
- *Mutazione*: Per problemi combinatori, la mutazione di scambio o la mutazione per inversione possono rivelarsi efficaci nel mantenere

soluzioni valide. In rappresentazioni numeriche, la mutazione gaussiana è spesso una scelta ragionevole, mentre la mutazione uniformemente distribuita può essere preferibile se vuoi esplorare nuove aree del problema.

7.1.6 – Parametri di terminazione

Stabilire i criteri di terminazione è fondamentale per evitare che l'algoritmo continui a evolversi inutilmente:

- *Numero di generazioni*: Se non conosci il comportamento del problema, un buon punto di partenza è fissare un numero di generazioni massimo (ad esempio, 100-500), valutando poi i risultati.
- *Convergenza*: Un altro criterio comune è arrestare l'algoritmo se non si osservano miglioramenti significativi della fitness per un numero di generazioni consecutivo (es. 20 generazioni).
- *Fitness desiderata*: Se la funzione obiettivo ha un valore noto ottimale, puoi arrestare il GA una volta che la soluzione raggiunge un fitness vicino o uguale a tale valore.

7.1.7 – Bilanciare esplorazione ed esploitazione

Il bilanciamento tra esplorazione di nuove soluzioni e sfruttamento delle soluzioni migliori è essenziale:

- *Esplorazione*: Se l'algoritmo sembra convergere troppo rapidamente verso una soluzione subottimale, aumentare la probabilità di mutazione o ridurre la pressione selettiva può aiutare a esplorare nuove aree dello spazio di ricerca.
- *Exploitation*: Se l'algoritmo non sembra convergere e continua a esplorare senza miglioramenti significativi, potrebbe essere utile aumentare la pressione selettiva (es. con un torneo più grande) o ridurre la probabilità di mutazione per focalizzarsi maggiormente sulle soluzioni più promettenti.

7.1.8 – Testare e affinare

La configurazione di un GA non è fissa e può richiedere vari tentativi per trovare il giusto equilibrio tra i diversi parametri. Alcune linee guida pratiche per ottimizzare le prestazioni del GA includono:

- *Sperimentazione iterativa*: Inizia con parametri standard e poi effettua piccoli aggiustamenti. Monitorare il comportamento del GA in termini di tempo di esecuzione e miglioramento della fitness è essenziale per identificare le configurazioni ottimali.

- *Analisi dei risultati*: Valuta la diversità della popolazione e i progressi delle soluzioni per capire se l'algoritmo è troppo orientato allo sfruttamento (e quindi rischia di perdere la diversità genetica) o se sta esplorando eccessivamente senza convergere.
- *Misurazione della convergenza*: Usa grafici per monitorare la fitness media e massima nel tempo, per identificare eventuali stagnazioni o fenomeni di convergenza prematura.

7.2 – Analisi degli Algoritmi Genetici

Vantaggi:

- *Flessibilità*: Gli algoritmi genetici possono essere applicati a una vasta gamma di problemi, inclusi quelli che non possono essere descritti da modelli matematici chiari.
- *Robustezza*: Sono in grado di gestire problemi complessi con molti parametri, fornendo soluzioni accettabili anche quando i metodi esatti falliscono.
- *Esplorazione globale*: La combinazione di selezione, crossover e mutazione permette agli algoritmi genetici di esplorare ampiamente lo spazio delle soluzioni, evitando di rimanere bloccati in soluzioni sub ottimali.

Svantaggi:

- *Convergenza lenta*: A causa della loro natura probabilistica, gli algoritmi genetici possono richiedere molte iterazioni per convergere verso una soluzione ottimale.
- *Richiedono molte valutazioni della fitness*: Ogni nuova generazione richiede una valutazione della fitness per tutti gli individui, il che può essere computazionalmente costoso, specialmente in problemi complessi.
- *Convergenza prematura*: A volte l'algoritmo può convergere troppo presto verso una soluzione sub-ottimale, specialmente se la popolazione perde diversità genetica.

7.3 – Algoritmi Genetici: Miglioramenti

Gli algoritmi genetici possono essere ulteriormente potenziati per affrontare problemi complessi o per migliorare la loro efficacia in scenari specifici.

7.3.1 – Elitismo

L'elitismo è una strategia che prevede di conservare un certo numero di individui con la fitness più elevata da una generazione alla successiva. Questo assicura che le migliori soluzioni non vadano perse durante il crossover o la mutazione.

L'elitismo può migliorare significativamente la convergenza del GA, mantenendo sempre le migliori soluzioni trovate finora.
Vantaggi:

- Garantisce che il GA non perda le soluzioni migliori già trovate.
- Aumenta la probabilità di convergenza verso una soluzione ottimale o vicina all'ottimale.

Considerazioni:

- Un numero troppo elevato di individui elitari può ridurre la diversità genetica, aumentando il rischio di convergenza prematura. Una buona pratica è utilizzare una piccola percentuale della popolazione (ad esempio, il 1-5%) come individui elitari.

7.3.2 – Algoritmi Genetici Multi-Obiettivo

Molti problemi reali richiedono l'ottimizzazione di più obiettivi contemporaneamente, che spesso possono essere in conflitto tra loro (ad esempio, massimizzare la qualità riducendo al minimo i costi). Gli algoritmi genetici multi-obiettivo (*MOGA*) sono progettati per trovare un insieme di soluzioni ottimali che rappresentino compromessi tra questi obiettivi, noti come soluzioni di Pareto ottimali.

In un contesto multi-obiettivo, una soluzione è detta **Pareto ottimale** se non esiste un'altra soluzione che migliora uno degli obiettivi senza peggiorarne almeno un altro. Questo concetto porta a un insieme di soluzioni, noto come *fronte di Pareto*, dove ciascuna soluzione rappresenta un compromesso ottimale tra gli obiettivi. Il ruolo degli algoritmi genetici multi-obiettivo è trovare e mantenere un insieme di soluzioni distribuite lungo questo fronte, permettendo al decisore di scegliere una soluzione in base alle preferenze tra i vari obiettivi.

Una soluzione A è detta dominata da un'altra soluzione B se B è migliore di A per almeno un obiettivo e non è peggiore per nessuno degli altri obiettivi. Un algoritmo genetico multi-obiettivo cerca di eliminare le soluzioni dominate, concentrandosi su quelle che non sono dominate da nessun'altra, ovvero le soluzioni Pareto ottimali.

Strategie principali:

- *NSGA-II* (Non-dominated Sorting Genetic Algorithm II): Classifica le soluzioni basandosi sulla dominanza di Pareto, mantenendo un equilibrio tra esplorazione dello spazio di ricerca e diversità delle soluzioni.
- *SPEA2* (Strength Pareto Evolutionary Algorithm 2): Un altro metodo popolare che tiene conto della dominanza di Pareto e introduce

meccanismi per mantenere la diversità nella popolazione durante la ricerca delle soluzioni ottimali.

Vantaggi:

- Permettono di ottenere un insieme di soluzioni, ognuna delle quali rappresenta un buon compromesso tra gli obiettivi.
- Mantengono una diversità elevata nelle soluzioni, utile per problemi complessi e con molteplici criteri di valutazione.

Considerazioni:

- Implementare algoritmi genetici multi-obiettivo richiede una maggiore complessità computazionale rispetto ai GA standard, a causa della necessità di valutare e mantenere un set di soluzioni non dominate.
- È importante mantenere un buon bilanciamento tra esplorazione e sfruttamento delle soluzioni, per garantire una copertura omogenea del fronte di Pareto.

7.3.3 – Strategia dell'archivio

La strategia dell'archivio è un miglioramento spesso usato negli algoritmi genetici multi-obiettivo. In questa strategia, un archivio di soluzioni viene mantenuto durante l'intero processo evolutivo per salvare le soluzioni non dominate (soluzioni di Pareto ottimali) che vengono trovate. L'archivio è generalmente limitato in dimensione, e soluzioni nuove entrano nell'archivio solo se non sono dominate dalle soluzioni esistenti o se sostituiscono soluzioni dominate già presenti nell'archivio.

Vantaggi:

- Permette di preservare un set di soluzioni non dominate, garantendo una diversità di soluzioni Pareto ottimali.
- Aiuta a conservare le migliori soluzioni trovate durante l'evoluzione, evitando che vengano perse nel processo di mutazione o crossover.

Considerazioni:

- L'archivio deve essere gestito in modo efficiente per evitare che diventi troppo grande o che vengano mantenute soluzioni ridondanti.
- Implementare una strategia di aggiornamento dell'archivio richiede un attento bilanciamento tra la conservazione delle soluzioni di qualità e l'inserimento di nuove soluzioni promettenti.

7.3.4 – Algoritmi Genetici Adattivi

Negli algoritmi genetici adattivi, i parametri chiave come le probabilità di mutazione e crossover non rimangono fissi, ma vengono modificati dinamicamente durante l'evoluzione, in base all'andamento dell'algoritmo.

Questo tipo di approccio aiuta a bilanciare meglio esplorazione ed esploitazione durante le varie fasi dell'evoluzione.
Vantaggi:

- Permettono di migliorare l'esplorazione nelle prime fasi del GA e lo sfruttamento nelle fasi finali.
- Possono ridurre il rischio di convergenza prematura e migliorare la qualità delle soluzioni finali.

Considerazioni:

- La progettazione di una strategia di adattamento efficace richiede una buona comprensione del comportamento del GA e del problema in questione.
- Parametri dinamici possono aggiungere complessità all'implementazione e richiedere maggiore tempo di sperimentazione.

7.3.5 – Island Model

Il modello ad isole divide la popolazione in sotto-popolazioni (isole), ciascuna delle quali evolve in modo indipendente per un certo numero di generazioni. Periodicamente, gli individui migliori migrano da un'isola all'altra, scambiando informazioni genetiche. Questo approccio aumenta la diversità genetica e migliora l'esplorazione.
Vantaggi:

- Migliora la diversità genetica mantenendo più popolazioni separate.
- Riduce il rischio di convergenza prematura in una singola popolazione.

Considerazioni:

- Il tasso e la frequenza della migrazione devono essere regolati con attenzione per bilanciare il mantenimento della diversità con il miglioramento della fitness.
- Implementare un modello ad isole richiede una gestione più complessa delle sotto-popolazioni.

7.3.6 – Crossover e Mutazione Non-Uniformi

Il crossover non-uniforme e la mutazione non-uniforme variano l'intensità con cui questi operatori influenzano gli individui man mano che l'algoritmo procede. Nelle prime generazioni, si favorisce una maggiore esplorazione (con mutazioni e crossover più ampi), mentre nelle fasi avanzate l'intensità diminuisce per concentrarsi su un'esplorazione più locale e su piccole modifiche per migliorare soluzioni già promettenti.
Vantaggi:

- Migliora l'equilibrio tra esplorazione iniziale e sfruttamento finale.

- Permette di adattarsi meglio alle diverse fasi dell'evoluzione, riducendo il rischio di blocchi su massimi locali.

Considerazioni:

- Richiede un controllo accurato dei parametri che regolano la riduzione dell'intensità di mutazione e crossover.
- Troppa riduzione può portare a una mancanza di esplorazione nelle ultime fasi, mentre troppa esplorazione può rallentare la convergenza.

7.3.7 – Ibridi con altre tecniche di ottimizzazione

Un approccio comune è combinare i GA con altre tecniche di ottimizzazione, creando algoritmi ibridi, anche detti **algoritmi memetici**. Ad esempio, una volta che il GA ha trovato una soluzione promettente, può essere usato un metodo di ottimizzazione locale per affinare ulteriormente quella soluzione.

Vantaggi:

- Permette di ottenere i benefici di più metodi: esplorazione globale dai GA e ottimizzazione locale da tecniche come l'Hill-Climbing.
- Migliora la qualità delle soluzioni finali e accelera la convergenza.

Considerazioni:

- Richiede una buona integrazione tra i diversi metodi per evitare conflitti o sovrapposizioni.
- La scelta di quando e come applicare l'ottimizzazione locale è fondamentale per l'efficacia dell'algoritmo ibrido.

8 – Algorimi di Ricerca con Avversari

Gli algoritmi di ricerca con avversari sono una categoria di algoritmi utilizzati per risolvere problemi in cui due o più agenti interagiscono tra loro. A differenza dei problemi di ricerca tradizionali, qui ogni agente deve prendere decisioni ottimali considerando anche le possibili mosse degli avversari.

Un esempio classico è il gioco degli scacchi, in cui ogni mossa non solo migliora la posizione di un giocatore, ma deve anche rispondere alle azioni dell'avversario. Per questo motivo, la pianificazione delle mosse richiede una strategia che massimizzi i guadagni in uno scenario collaborativo o competitivo.

8.1 – Teoria dei giochi

La **teoria dei giochi** è una branca dell'economia che considera ogni ambiente multi-agente come un gioco, sia che l'interazione sia cooperativa o competitiva. I giochi possono essere classificati in base a due dimensioni principali:

- **Condizioni di Scelta**:
  - Giochi ***con informazione perfetta***: gli stati del gioco sono completamente noti agli agenti.
  - Giochi ***con informazione imperfetta***: gli stati del gioco sono parzialmente noti agli agenti.
- **Effetti della Scelta**:
  - Giochi ***deterministici***: gli stati sono determinati solo dalle azioni degli agenti.
  - Giochi ***stocastici***: gli stati sono influenzati anche da fattori esterni.

Oltre ad essere ad informazione perfetta, i giochi che saranno privilegiati nel corso saranno anche *a somma zero,* cioè si tratterà di giochi in cui la vincita di un giocatore coincide esattamente con la perdita dell'altro.

8.2 – Definizione di un Gioco

Formalmente, un gioco può essere definito come un problema di ricerca con i seguenti componenti:

- ***s0:*** stato iniziale.
- ***Giocatore(s):*** definisce il giocatore che deve muovere nello stato s.
- ***Azioni(s):*** restituisce l'insieme delle mosse lecite nello stato s.
- ***Risultato(s, a):*** il modello di transizione che definisce il risultato della mossa a.
- ***Test Terminazione(s):*** restituisce vero se la partita è finita, falso altrimenti.
- ***Utilità(s, p):*** funzione di utilità (o di payoff) che assegna un valore numerico finale per il giocatore p nello stato terminale s.

8.3 – Strategia Ottima, Equilibrio di Nash e Ottimi Paretiani

Una **strategia ottima** porta ad un risultato che è almeno pari a quello di qualsiasi altra strategia, assumendo di giocare contro un giocatore infallibile. Nel caso di un problema di ricerca con avversari, si tratta di una strategia che massimizzi il risultato nel caso peggiore.

Vale la pena approfondire questo discorso nel contesto del celebre *dilemma del prigioniero*. Due prigionieri sono accusati di un crimine e posti in celle separate, senza possibilità di comunicare tra loro. Ogni prigioniero può scegliere se collaborare con l'altro rimanendo in silenzio, oppure tradire confessando il crimine e accusando l'altro. Le possibili pene dipendono dalle scelte di entrambi:

- Se entrambi i prigionieri restano in silenzio (collaborano), riceveranno una pena ridotta (ad esempio, 1 anno di prigione ciascuno).

- Se uno tradisce e l'altro collabora, il prigioniero che ha tradito verrà rilasciato (0 anni), mentre l'altro riceverà la pena massima (7 anni).
- Se entrambi tradiscono, riceveranno una pena moderata (ad esempio, 6 anni ciascuno).

Il dilemma sta nel fatto che, per ciascun prigioniero, tradire sembra essere la strategia ottima a breve termine, poiché offre la possibilità di evitare completamente la prigione, indipendentemente dalla scelta dell'altro. Tuttavia, se entrambi i prigionieri tradiscono, il risultato complessivo è peggiore rispetto a quello che si otterrebbe con la collaborazione reciproca.

Questo esempio illustra chiaramente come una strategia ottima in un contesto competitivo non sempre conduca al miglior risultato complessivo. Nel dilemma del prigioniero, la strategia di *equilibrio di Nash* prevede che entrambi i prigionieri tradiscano, poiché ciascuno cerca di minimizzare la propria perdita nel caso peggiore. Sono invece presenti 3 *ottimi paretiani*, cioè i due casi in cui soltanto uno dei due accusa l'altro più il caso in cui nessuno dei due parla.
Per l'equilibrio di Nash non è rilevante il fatto che se un prigioniero parla peggiora la situazione dell'altro, mentre per gli ottimi si è in una situazione in cui non è possibile migliorare la condizione di una persona senza peggiorare quella di un'altra.

8.4 – Algoritmo minimax

L'algoritmo **minimax** analizza l'intero albero di gioco, esplorando tutte le mosse e contromosse possibili fino a trovare la sequenza di azioni che massimizza il punteggio nel caso peggiore. Il valore minimax di un nodo rappresenta l'utilità di quel nodo, assumendo che entrambi i giocatori giochino in modo ottimale.
Funzionamento dell'algoritmo:

- Una funzione ausiliaria si occupa di iniziare la ricerca, che sarà effettuata chiamando in alternanza le funzioni ValoreMax(s) e ValoreMin(s).
- Il primo turno è sempre del giocatore Max e si prosegue in modo ricorsivo fino a giungere ad una foglia dell'albero di gioco.
- Se ad arrivarci è MIN, allora sarà scelto come valore del nodo padre il valore di utilità più basso presente tra le figlie; se invece è MAX, il più grande.
- Ci si alterna in questo modo risalendo fino alla radice.
- MAX a questo punto sceglie il valore più alto trovato, poiché sono state calcolate tutte le sequenze di mosse possibili.

Da un punto di vista computazionale, l'algoritmo minimax ha le seguenti proprietà:

- *Completezza*: L'algoritmo è completo se l'albero di gioco è finito.
- *Ottimalità*: L'algoritmo è ottimo se l'avversario gioca in modo ottimo.
- *Complessità temporale*: La complessità temporale è O(b^m), dove b è il fattore di ramificazione ed m è la profondità massima dell'albero.
- *Complessità spaziale*: Richiede O(bm) di memoria, dovendo mantenere un solo cammino dalla radice alla foglia assieme ai nodi fratelli non espansi per ciascun nodo sul cammino.

8.5 – Potatura Alfa-Beta

Le prestazioni computazionali dell'algoritmo minimax non consentono la risoluzione di problemi reali in quanto spesso si ha a che fare con alberi di gioco molto vasti. La **potatura alfa-beta** è una tecnica che permette di ottimizzare l'algoritmo minimax riducendo il numero di nodi da esplorare e senza influenzare la decisione finale. Il nome deriva dai due parametri aggiuntivi che si usano: alfa indica il valore della scelta migliore per MAX trovata finora, beta invece la migliore per MIN. Consente di esplorare soltanto O(b^(m/2)) nodi.

In generale, per spiegarne il funzionamento, possiamo considerare un certo nodo n da qualche parte nell'albero di gioco che possiamo raggiungere. Se esiste una scelta migliore m al livello del nodo padre, o anche di uno qualunque dei nodi precedenti, allora n non sarà mai raggiunto in tutta la partita.
La potatura funziona grazie al confronto continuo tra i valori di alfa e beta, aggiornati durante l'esplorazione dei rami dell'albero di gioco. Ogni volta che si scopre che il valore di un ramo non migliorerà oltre i limiti di alfa o beta, quel ramo può essere potato, poiché non influenzerà il risultato finale. Questo consente di ridurre significativamente il numero di nodi da esplorare.

8.6 – Ordinamento Dinamico delle Mosse

L'ordinamento dinamico delle mosse è una tecnica che migliora ulteriormente l'efficienza della potatura alfa-beta. L'idea di base è ordinare le mosse più promettenti all'inizio del processo di esplorazione, in modo da effettuare più potature precoci ed espolorare meno nodi. È una tecnica efficace perché:

- Aumenta la probabilità di aggiornare i valori di alfa e beta rapidamente.
- Migliora la velocità complessiva dell'algoritmo, soprattutto in giochi complessi come gli scacchi, dove il numero di possibili mosse è elevato.
- Riduce il tempo di ricerca necessario per identificare la mossa ottima, specialmente nei livelli superiori dell'albero di gioco, dove le decisioni hanno maggiore impatto sul risultato finale.

Una strategia comune per implementare l'ordinamento dinamico delle mosse consiste nell'utilizzare una funzione di valutazione euristica che approssima la qualità di una mossa prima che l'intero albero venga esplorato. Esempi:

- Usare la ***ricerca ad approfondimento iterativo:*** cerchiamo in uno strato in profondità e registriamo il miglior cammino di mosse, poi cerchiamo uno strato ancora più in profondità, ma usando il cammino trovato per ordinare le mosse.
- Usare una ***tabella delle trasposizioni:*** si tratta di una HashTable in cui memorizziamo la valutazione di una configurazione la prima volta che la incontriamo.

8.7 – Decisioni Perfette in Tempo Reale

Il concetto di decisioni perfette in tempo reale si riferisce alla capacità di un algoritmo di prendere la miglior decisione possibile entro un limite di tempo prestabilito. In giochi complessi, è spesso impraticabile esplorare l'intero albero di gioco a causa dei limiti di tempo imposti.
Esistono diverse tecniche per garantire che un algoritmo di ricerca con avversari possa prendere decisioni in tempo reale:

- **Euristiche veloci**, con cui si stima il valore di stati non terminali in modo rapido, evitando l'esplorazione completa dei rami dell'albero.
- **Timeout,** per cui l'algoritmo, allo scadere del tempo disponibile, restituisce la mossa che ha esplorato più a fondo fino a quel momento.

Possiamo pensare di sostituire la funzione di utilità dell'algoritmo minimax con una funzione di valutazione EVAL che restituisce una stima del guadagno atteso in una determinata posizione. In generale, andrebbe applicata soltanto a *posizioni quiescenti,* cioè posizioni in cui è improbabile il verificarsi di grandi variazioni di valore nelle mosse immediatamente successive.

8.8 – Ulteriori Miglioramenti della Potatura Alfa-Beta

Sezione assente sulle slide: ripete l'approfondimento iterativo e la tabella di trasposizione e suggerisce come altri miglioramenti le seguenti tecniche:

- Una tecnica chiamata ***finestra aspettativa*** può essere applicata per migliorare ulteriormente la potatura alfa-beta. Invece di esplorare l'intero intervallo di valori di alfa e beta, l'algoritmo parte con una finestra ristretta intorno al valore atteso della mossa migliore. Se il valore della mossa rientra nella finestra, non è necessario esplorare ulteriori rami. Se invece il valore esce fuori dalla finestra, la finestra viene estesa e l'esplorazione continua.

- La ***potatura scout*** o ***negascout*** è una variante della potatura alfa-beta che cerca di ridurre ulteriormente il numero di nodi esplorati attraverso una tecnica più aggressiva. Negascout esplora prima un ramo del sottoproblema con una finestra ristretta, e solo se trova una mossa migliore, esplora gli altri rami. Questo riduce la quantità di rami da esplorare rispetto alla potatura alfa-beta standard.
- La ***potatura multi-cut*** è una tecnica che cerca di interrompere l'esplorazione di sottorami all'interno di un livello di profondità se si scopre che più di un certo numero di rami stanno portando a valori non promettenti. Questa tecnica consente di eliminare grandi sezioni dell'albero che non influenzeranno il risultato finale.

9 – Teoria dell'Apprendimento

Gli algoritmi di apprendimento operano tramite interazione con l'ambiente ed accumulo di esperienza, consentendo agli agenti di diventare sempre più competenti, cioè di migliorare le loro prestazioni per un compito/task.
Il *machine learning* permette ai modelli di imparare dai dati e sulla base di questi fare previsioni. Possiamo classificare le tipologie di apprendimento così:

- **Apprendimento supervisionato,** in cui l'agente apprende da un insieme di dati etichettati, cioè di esempi composti da input + risposta corretta, detta variabile dipendente o output. È utile quando l'obiettivo è costruire un modello capace di predire valori futuri sulla base dei dati osservati. Come algoritmi comuni abbiamo: alberi di decisione, regressione lineare e logistica, reti neurali, support vector machines (SVM) e K-nearest neighbours (KNN).
- **Apprendimento non supervisionato,** in cui l'algoritmo non riceve alcuna etichetta di riferimento e deve scoprire da solo pattern o strutture nascoste. Come algoritmi comuni abbiamo: K-means clustering, principal component analysis (PCA), clustering gerarchico, FP-Growth e t-SNE.
- **Apprendimento semi-supervisionato,** in cui l'algoritmo riceve un dataset misto che presenta soltanto un'etichettatura parziale. È utile quando l'architettura dei dati è costosa o difficile da ottenere. Come algoritmi comuni abbiamo: modelli basati su grafi, approcci di clustering assistiti, metodi di auto-apprendimento e tecniche di propagazione delle etichette.
- **Apprendimento per rinforzo,** in cui l'agente riceve un feedback positivo o negativo a seconda del suo comportamento all'interno dell'ambiente ed ha l'obiettivo di massimizzare una ricompensa cumulativa. Come algoritmi comuni troviamo: Q-learning, SARSA, DDPG e PPO.

Una classificazione alternativa per i problemi di machine learning può essere fatta in base agli output che si desidera ottenere:

- **regressione,** quando l'output è un valore continuo;
- **classificazione,** quando l'output è discreto;
- **clustering,** quando l'obiettivo è suddividere i dati in gruppi;
- **riduzione della dimensionalità,** quando bisogna eliminare le caratteristiche ridondanti e ridurre quelle significative in insiemi grandi di dati;
- **mining delle associazioni,** quando vanno identificati pattern comuni in insiemi di transazioni.

9.1 – Scelta dell'algoritmo e Teoria dell'Errore

La scelta dell'algoritmo, in generale, dipende dalla natura del problema e dalle caratteristiche del dataset: la scelta ottimale, in molti casi, si basa su valutazioni empiriche, fatte testando vari modelli. Ad essere scelto è quello con il minor errore, cioè quello che si avvicina maggiormente ai valori reali.
Suddividiamo gli errori in due categorie principali:

- **Errori irriducibili,** cioè intrinseci al processo stesso di raccolta dati ed ai fattori di rumore presenti nelle misurazioni. Si tratta di errori inevitabili dovuti alle componenti casuali o ai fattori esterni che influenzano i risultati.
- **Errori riducibili,** che possono essere ridotti tramite miglioramenti del modello. Sono determinati da bias e varianza, tra cui bisogna individuare il giusto compromesso con una certa attenzione:
  - Il ***bias,*** o errore sistematico, misura quanto le previsioni del modello si discostino sistematicamente dai valori reali. Avere un alto bias porta al fenomeno dell'*underfitting:* significa tendere a fare previsioni troppo semplici, adattandosi male sia ai dati di training che a quelli di test.
  - La ***varianza*** misura la sensibilità del modello alle variazioni nei dati di training. Una varianza troppo alta porta all'*overfitting*, cioè all'adattarsi così tanto ai dati di training da catturarne anche il rumore, portando l'algoritmo a diventare troppo complesso ed incapace di generalizzare.

In alcuni casi è possibile risolvere, o quantomeno mitigare, i rischi di underfitting e overfitting, lavorando sulla configurazione degli algoritmi di machine learning. Alcune operazioni tipiche riguardano:

- **selezione delle caratteristiche rilevanti,** tramite cui l'algoritmo si focalizza sui soli dati che rappresentano la variabile dipendente;
- **convalida incrociata,** o cross-validation, in cui si generano una serie di dataset per consentire all'algoritmo di perfezionare l'apprendimento;

- **regolazione dei parametri,** in cui, quando possibile, consentiamo all'algoritmo di studiare meglio i dati di input;
- **aumento della dimensione dei dati di input,** cioè ingrandire l'insieme delle osservazioni da cui l'algoritmo può apprendere.

10 – Ingegneria del Machine Learning

Anche i modelli di machine learning richiedono metodi di ingegneria rigorosi per evitare errori e limitazioni, che possono anche avere conseguenze gravi.
Il modello **CRISP-DM** (Cross-Industry Standard Process for Data Mining) rappresenta il ciclo di vita dei progetti di machine learning: è simile al modello a cascata con feedback continuo ed è non sequenziale, cioè le diverse fasi possono essere eseguite un numero illimitato di volte. Nello specifico, abbiamo:

- **Business Understanding,** in cui si definisce con esattezza ciò che il progetto dovrebbe raggiungere dal punto di vista aziendale e tecnico. È prevista la definizione dei *business success criteria,* cioè i criteri con cui accertarci che il sistema sia in linea con gli obiettivi di business, e si valutano costi, risorse, rischi ed eventuali piani di contingenza. Si definiscono poi gli obiettivi tecnici da raggiungere e si scelgono tecnologie e tool necessari. L'output è un piano di progetto completo che ne spiega l'esecuzione da un punto di vista di gestione tecnica.
- **Data Understanding,** in cui il team esplora i dati disponibili per valutare se siano adeguati per il progetto e per ottenere insight preliminari. I dati raccolti vengono inseriti in un tool di analisi, in modo da esaminarli e documentarli rigorosamente. Si procede poi identificando le correlazioni tra i dati ed infine si ricercano anomalie e valori mancanti o duplicati. L'output è il documento di analisi dei dati.
- **Data Preparation,** in cui ci si concentra sulla trasformazione e sulla pulizia dei dati per assicurarne l'idoneità. Viene effettuato *feature engineering*, cioè si selezionano le caratteristiche del problema che hanno la maggiore potenza predittiva. Si effettua poi la pulizia dei dati sulla base dei problemi di qualità riscontrati ed infine si normalizzano le variabili, cioè si formattano affinché possano essere prese in input da un modello di machine learning. L'output è un dataset pronto.
- **Modeling,** in cui il team seleziona ed addestra i modelli scegliendo gli algoritmi e i parametri più appropriati. Dopo aver selezionato la tecnica o l'algoritmo, lo si addestra con il dataset ottenuto alla fase precedente. L'output è un modello addestrato e configurato sui dati a disposizione.
- **Evaluation,** in cui il modello viene rigorosamente testato e valutato per assicurarsi che soddisfi i requisiti del progetto. Si misurano le sue prestazioni su nuovi dati per verificare se sappia generalizzare

correttamente o soffra di underfitting o overfitting e ci si assicura che sia in linea con obiettivi e requisiti. Si ottiene un rapporto di valutazione del modello, che ne rivelano attendibilità e conformità.
- **Deployment,** in cui il modello viene integrato nel sistema di produzione, dove può essere effettivamente utilizzato. Ci si assicura che possa reggere ai carichi di lavoro previsti e si implementa un sistema di monitoraggio per tracciare le prestazioni ed individuare eventuali problemi o rallentamenti. Si documenta infine il funzionamento del modello e si forma il personale. L'output è un report finale di progetto.

10.1 – Dinamiche socio-tecniche nei team di Machine Learning

Il modello **CRISP-DM** è considerato allo stesso tempo sia tradizionale che agile, in quanto le diverse fasi possono essere ripetute più e più volte, garantendo un livello variabile di flessibilità. Il modello però non considera esplicitamente gli elementi *socio-tecnici*, poiché non chiarisce chi è responsabile di cosa. Con **socio-technical congruence** ci si riferisce proprio all'allineamento tra le strutture sociali, come team e ruoli, e le componenti tecniche, come strumenti e workflow, ed è una componente chiave se si vogliono evitare conflitti ed inefficienze.

TDSP (Team Data Science Process) è una metodologia sviluppata per combinare l'approccio CRISP-DM con i principi agili di SCRUM. Le fasi di sviluppo qui sono:
- ***Business Understanding:*** in cui si definiscono gli obiettivi di business.
- ***Data Acquisition & Understanding:*** che combina le fasi di *data understanding* e *data preparation* del CRISP-DM.
- ***Modeling:*** che unisce *modeling* ed *evaluation* del CRISP-DM.
- ***Deployment:*** che ha a che fare con le scelte ingegneristiche per rendere il modello usabile.

Come in SCRUM, vengono usati sprint per suddividere il progetto in cicli di sviluppo brevi e iterativi, con produzione di deliverable parziali, prototipi di modelli o analisi di dati preliminari. Ogni sprint inizia con la fase di *sprint planning*, viene monitorato tramite *daily scrum*, termina con una *sprint review* ed infine viene chiuso con una *sprint retrospective*.

Sono definiti sei ruoli espliciti:
- Project Manager, responsabile dell'intero progetto, di cui gestisce le risorse e monitora l'avanzamento.

- Project Lead, che coordina gli sprint e facilita la comunicazione e la collaborazione, eliminando eventuali ostacoli.
- Data Engineer, responsabile della data acquisition e quindi della raccolta, della trasformazione e della gestione dei dati.
- Data Scientist, responsabile della data understanding e quindi dell'analisi e della modellazione dei dati.
- Application Developer, responsabile dell'implementazione dell'applicazione e della sua integrazione nel sistema di produzione.
- Solution Architect, responsabile dell'architettura di sistema, che definisce, e che si assicura che il modello sia ben integrato con le altre componenti.

I rischi legati a potenziali misunderstanding dei requisiti sono minimizzati e viene esplicitato il legame tra ingegneria del machine learning ed ingegneria del software. La necessità di definire degli sprint può risultare però limitante, soprattuto quando le attività di preparazione dati e modellazione richiedono tempi lunghi e incerti.

https://learn.microsoft.com/it-it/azure/architecture/data-science-process/overview

11 – Qualità dei dati e Feature Engineering

La qualità e la preparazione dei dati influiscono direttamente sulle prestazioni del modello, determinandone l'accuratezza e la capacità di generalizzare. Con ***qualità*** si intende la loro misura di accuratezza, completezza e consistenza.
L'**ingegneria dei dati** è l'insieme delle tecniche e degli algoritmi che consentono l'estrazione, l'analisi e la preparazione di dati che saranno utilizzabili per altre tecniche o algoritmi di data analytics**.**
La **data governance** è il processo tramite cui si gestiscono la disponibilità, l'usabilità, l'integrità e la sicurezza dei dati. Si tratta di un insieme di standard interni ad un organizzazione o di politiche e pratiche che ne regolano l'utilizzo.
Uno dei problemi di qualità principali è il **data leakage**, che si verifica quando sono incluse, in fase di addestramento, delle informazioni che poi non si avranno a disposizione: il risultato è che la prestazione del modello sarà sovrastimata ed esso non lavorerà accuratamente durante l'uso reale.

11.1 – Tipologie di Dati

Un dato è un qualsiasi elemento di cui si dispone per formulare un giudizio o risolvere un problema; ne esistono diversi tipi:

- **Dati strutturati**, organizzati in formati tabulari che consentono di sapere con esattezza il significato di ogni elemento. Sono facili da analizzare e processare, ma possono risultare limitanti per informazioni complesse.
- **Dati non strutturati,** che sono privi di una struttura predefinita e vanno quindi estratti con strumenti ad-hoc. Un esempio sono i dati testuali, per i quali è stata definita una branca specifica dell'IA, che prende il nome di *Natural Language Processing.*
- **Dati semi-strutturati,** in cui è fissato il formato ma non la struttura, come in *JSON* o *XML*. Sono più flessibili dei dati strutturati e posssono rappresentare relazioni più complesse, ma necessitano di parsing e conversioni. Un esempio sono dati tabulari con informazioni mancanti o espresse in formato non strutturato.

11.2 – Tecniche di Data Preparation

La preparazione dei dati è fondamentale per garantire che il modello possa apprendere correttamente. Si suddivide in 4 fasi:

- **Data cleaning,** cioè la fase di pulizia dei dati. Ricorre ad un insieme di tecniche di *data imputation* per stimare il valore dei dati mancanti oppure, quando possibile, si usano metodi più banali, come lo scarto delle righe e delle colonne che presentano dati mancanti. Tra le tecniche di imputazione abbiamo quella *statistica*, che per dati numerici ricorre a media e mediana, quella *deduttiva*, che definisce regole basate sulla deduzione logica, e quella tramite *most frequent imputation*, che inserisce il valore più frequente contenuto nella colonna.
- **Feature scaling,** cioè la fase in cui si normalizza o "si scala" l'insieme dei valori di una caratteristica, al fine di non "far confondere" l'algoritmo riguardo la sua importanza. Troviamo due tecniche comuni:
  - *min-max normalization*, che porta i valori entro un intervallo predefinito (ad esempio [0,1]) ed è utile per modelli basati su distanza:
    - $x' = a + \frac{(x - min(x)) \cdot (b - a)}{max(x) - min(x)}$
    - con a e b rispettivamente minimo e massimo;
  - *z-score normalization*, che centra i dati intorno alla media e li scala sulla base della deviazione standard ed è ideale per dati che seguono una distribuzione normale:
    - $x' = \frac{(x - \bar{x})}{\sigma}$
    - con x valore originale, $\bar{x}$ media della distribuzione e σ deviazione standard;

- **Feature engineering,** che consiste nel creare e selezionare le caratteristiche più rilevanti. Si compone di *feature extraction*, che riduce la dimensionalità del dataset per semplificarlo senza perdere informazioni importanti (PCA ed LDA), *feature construction*, che crea nuove variabili basandosi su dati esistenti, e *feature selection*, che elimina le variabili meno rilevanti per ridurre il rumore (RFE).
- **Data balancing**, cioè un insieme di tecniche per convertire un dataset da sbilanciato a bilanciato, in modo da evitare che il modello favorisca la classe maggioritaria. Un metodo è l'*undersampling*, in cui si eliminano un certo numero di righe dalla classe maggioritaria se i dati presenti sono abbondanti, l'altro è l'*oversampling*, che invece aggiunge casualmente righe nel dataset della classe di minoranza tramite duplicazione o generazione sintetica (SMOTE). Si possono migliorare le prestazioni, ma si rischia di generare problemi, per cui si usano versioni modificate e migliorate. (vedi clustering)

12 – Classificazione e Classificatori

La **classificazione** è un task fondamentale nell'apprendimento supervisionato che consiste nel predire il valore di una variabile categorica, detta variabile dipendente o target, sulla base di un training set, ovvero un insieme di osservazioni in cui il target era noto.

Un modello di classificazione usa un algoritmo di apprendimento detto **classificatore** per categorizzare le nuove osservazioni. Ne esistono diverse famiglie, ognuna con caratteristiche specifiche:
[**NB:** elenco non presente sulle slide ma solo sugli appunti]

- **Classificatori probabilistici,** basati sui concetti di probabilità e distribuzione. Richiedono stime accurate, sono efficienti su dataset di grandi dimensioni e con molte classi e tendono ad assumere ipotesi forti.
- **Classificatori basati su entropia,** che usano misure di impurezza come l'entropia per suddividere i dati in insiemi sempre più puri. Sono intuitivi e facila da interpretare, ma sono suscettibili all'overfitting.
- **Metodi ensemble,** ad esempio il *Majority Voting* o la *Random Forest,* che combinano le predizioni di più classificatori. Possono gestire dataset complessi e sbilanciati, ma sono computazionalmente intensivi.
- **Classificatori lineari,** che si basano sull'ipotesi che le classi siano separabili facilmente con un confine reale. Sono semplici ed efficienti.
- **Classificatori non lineari,** che sono progettati per gestire confini decisionali complessi. Possono modellare relazioni complesse, ma sono computazionalmente costosi.

- **Classificatori basati su regole,** che si basano su regole if-then, pertanto facili da interpretare e adatti dove è necessaria la spiegabilità, ma inefficienti su dataset grandi e con potenziali problemi di generalizzazione.
- **Problemi basati su prossimità,** che usano la distanza tra i punti del dataset per determinare la classi di un'istanza. Adatti a problemi con distribuzioni locali complesse, ma computazionalmente costosi.

12.1 – Classificatore Naive Bayes

L'algoritmo considera le caratteristiche della nuova istanza da classificare e calcola la probabilità che questa faccia parte di una classe tramite l'applicazione del teorema di Bayes. È chiamato naive poiché assume che le feature o caratteristiche siano indipendenti tra loro, ipotesi spesso non realistica. Segue tre step:

- **Calcolo della probabilità della classe,** cioè, semplicemente, il rapporto tra le frequenze delle istanze di una classe ed il loro numero totale.
- **Calcolo della probabilità condizionata,** cioè l'applicazione del teorema.
- **Decisione,** cioè la classificazione effettiva nella classe che ha ottenuto il valore di probabilità più elevato.

12.2 – Alberi decisionali

L'algoritmo mira a creare un albero in cui i nodi rappresentano una caratteristica o un sottoinsieme di esse e gli archi rappresentano delle decisioni. Sono semplici da leggere, in quanto ogni percorso dell'albero corrisponde ad una regola di decisione inferita dai dati di training. Opera in 3 fasi:

- **Selezione della radice,** in cui si sceglie la miglior caratteristica secondo una certa metrica.
- **Divisione del dataset** in sottoinsiemi.
- **Ripetizione,** per cui si ripetono i passi precedenti per ciascun sottoinsieme, fino al raggiungimento di un criterio di arresto.

Per la scelta della radice possiamo usare l'**information gain**, una metrica che misura il grado di purezza di un attributo, cioè quanto sarà in grado di dividere adeguatamente il dataset. La formula dell'entropia nella teoria dell'informazione:

- $H(D) = -\sum_{c} p(c) * \log_2(p(c))$
- dove p(c) è la proporzione della classe c nel dataset D.

Nei decision tree l'entropia è usata come base per l'analisi dell'information gain, per cui vogliamo minimizzarla ad ogni iterazione. Per ogni attributo calcoliamo:

- $Gain(D, A) = H(D) - \sum_{v \in values(A)} (\frac{|D_v|}{|D|}) * H(D_v)$
- dove D è l'entropia del dataset, |D| è il numero di elementi del dataset, D_v è il sottoinsieme di D per cui l'attributo A ha valore v e $|D_v|$ è il numero di elementi di D_v.

Un'altra metrica è il *Gini Index.*

Gli alberi decisionali sono semplici ed efficaci, ma presentano due limitazioni: la prima è l'overfitting, a cui si va incontro quando diventano troppo grandi, e la seconda è l'instabilità. Per cercare di mantenere un buon equilibrio tra accuratezza e generalizzazione si possono usare diversi iper-parametri:

- **Massima profondità dell'albero,** che limita il numero di livelli.
- **Numero minimo di campioni per divisione,** che specifica un numero minimo di esempi richiesti in un nodo per consentire una divisione.
- **Numero minimo di campioni per foglia,** che specifica il numero minimo di esempi che devono essere presenti in un nodo foglia.

12.3 – Validazione dei classificatori

La validazione è un passaggio cruciale per garantire che un classificatore non solo funzioni bene sui dati di addestramento, ma generalizzi correttamente su dati nuovi. Esistono diversi metodi:

- **Training/Test split,** in cui si suddivide il dataset di partenza in due parti di grandezza variabile: una per la fase di training e l'altra per il testing. Andiamo in contro a due problemi principali: l'uso inefficiente dei dati, particolarmente rilevante se si ha un dataset piccolo, e la dipendenza dalla partizione, in quanto le prestazioni dipendono dalla scelta casuale dei set.
- **Convalida incrociata,** o k-fold cross validation, in cui i dati di partenza vengono inizialmente mescolati e divisi in k gruppi. A questo punto, per ciascuna iterazione, uno sarà scelto come test set ed i restanti come training set. I risultati sono infine mediati. Ha costo computazionale alto.
- **Stratified cross-validation,** variante della convalida incrociata classica che punta a mitigare gli effetti di suddivisioni irrealistiche. In questo caso i sottoinsiemi hanno un numero simile di istanze delle diverse classi.
- **Validazione temporale,** una strategia specifica per quei dataset in cui sono presenti relazioni temporali tra i dati. La suddivisione è temporale ed il modello viene addestrato su una porzione iniziale e validato sulle successive. Anche qui non si usano tutti i dati a disposizione per l'addestramento e soffre nel caso di cicli stagionali o tendenze complesse.

12.4 – Metriche di validazione

A prescindere dalla procedura scelta, è necessario usare gli strumenti adatti per valutare la bontà delle predizioni: per farlo usiamo una *matrice di confusione,* o tabella di errata classificazione.

TP = veri positivi, FP = falsi positivi, TN = veri negativi, FN = falsi negativi

$$Precision = \frac{TP}{(TP + FP)} \qquad Recall = \frac{TP}{(TP + FN)} \qquad Specificity = \frac{TP}{(TN + FP)}$$

$$Accuracy = \frac{(TP + TN)}{(TP + TN + FP + FN)} \qquad MCC = \frac{TP \bullet TN - FP \bullet FN)}{\sqrt{(TP + FP) \bullet (TP + FN) \bullet (TN + FP) \bullet (TN + FN)}}$$

[Specificity forse è sbagliata ed ha TN al numeratore?]
Con **precision** si indica il numero di predizioni corrette per la classe "true" rispetto a tutte le predizioni positive fatte dal classificatore. Non considera i falsi negativi, ma pone l'attenzione sui falsi positivi.
Con **recall** si indica il numero di predizioni corrette per la classe "true" rispetto a tutte le istanze positive di quella classe. È possibile fare in modo che sia molto alto classificando tutto come positivo, ma si comprometterebbe la precisione.
Con **accuracy** si indica l'accuratezza del modello, cioè la proporzione di predizioni corrette sul totale. Non è indicativa in dataset sbilanciati.
[**NB:** Le slide non forniscono le spiegazioni per le altre formule, mentre gli appunti hanno questa in più]
Con **F1-Score** = 2 * [(Precision * Recall) / (Precision + recall)], detta anche F-Measure, si combinano precision e recall in una media armonica, bilanciando i due aspetti. Diventa difficile da interpretare in contesti con classi sbilanciate o dove una singola metrica è più importante delle altre.

13 – Regressione e Regressori

I problemi di regressione sono istanze di problemi di apprendimento supervisionato in cui l'obiettico è predire il valore di una variabile numerica tramite l'uso di un training set, ovvero un insieme di osservazioni in cui la variabile dipendente è nota. La regressione produce output continui, poiché qui i modelli cercano di approssimare una funzione f(x) per rappresentare al meglio la relazione tra input e output. Anche qui, troviamo diverse famiglie sia a seconda delle assunzioni fatte sui dati, sia a seconda del numero di variabili indipendenti (predittori) di cui disponiamo:
[**NB:** elenco non presente sulle slide]

- **Regressori lineari**, che assumono una relazione lineare tra le variabili indipendenti e la variabile dipendente, ideali per problemi in cui le relazioni sono facilmente implementabili. Efficienti computazionalmente e con buone prestazioni, ma sensibili ad outlier.
- **Regressioni basate su alberi,** che usano alberi decisionali per suddividere iterativamente i dati in sottoinsiemi omogenei, adatti a dati categoriali. Senza regolazioni sono suscettibili all'overfitting.
- **Metodi ensemble,** che combinano più modelli per migliorare robustezza e capacità predittiva. Computazionalmente intensivi e con risultati difficili da interpretare, ma gestiscono bene dati rumorosi e complessi. Alcuni esempi sono il *Random Forest Regressor*, il *GBM* e l'*AdaBoost Regressor*.
- **Regressori non lineari,** come *SVR* e *Reti Neurali*, progettati appositamente per modellare relazioni complesse. Richiedono un tuning accurato degli iper-parametri e sono complessi computazionalmente.
- **Regressori probabilistici,** che si basano su modelli probabilistici per stimare la distribuzione dell'output, come il *Gaussian Process Regressor.* Sono adatti per dati rumorosi con alta incertezza, fornendo intervalli di confidenza per le predizioni, ma non sono scalabili a dataset grandi.

13.1 – Regressori lineari: semplici e multipli

Ciò che vogliamo fare con la regressione lineare semplice è trovare una funzione che si adatti ai dati di training e che sarà efficace nel predire il valore della variabile dipendente y su nuovi dati:

- $\widetilde{y} = \beta_0 + \beta_1 \cdot x_1$
- dove $\widetilde{y}$ è il valore predetto dal modello, β_0 l'intercetta e β_1 l'inclinazione.

L'altra relazione che ci interessa è:

- $y = \widetilde{y} + \varepsilon$
- dove y è il valore reale ed epsilon indica il residuo, cioè la differenza tra valore reale e valore predetto.

Per capire quale retta corrisponda alla miglior approssimazione si possono usare diverse metriche di valutazione:

Mean Absolute Error (MAE)

$$\frac{\sum_{i=1}^{n} |\tilde{y} - y|}{n}$$

La metrica MAE indica la differenza media osservata tra i valori predetti e i valori reali del test set.

Mean Squared Error (MSE)

$$\frac{\sum_{i=1}^{n} (\tilde{y} - y)^2}{n}$$

La metrica MSE indica l'errore quadratico medio commesso sui dati presenti nel test set.

Root Mean Squared Error (RMSE)

$$\sqrt{\frac{\sum_{i=1}^{n} (\tilde{y} - y)^2}{n}}$$

La metrica RMSE indica la radice quadrata dell'errore quadratico medio commesso sui dati presenti nel test set.

MAE non penalizza gli errori grandi, mentre **MSE** è sensibile agli outlier.
A queste 3 va aggiunto anche il **coefficiente di determinazione,** che misura la proporzione della variabilità spiegata dal modello:

- $R^2 = 1 - \frac{\sum_{i=1}^{n} (\tilde{y}_i - y_i)^2}{\sum_{i=1}^{n} (y_i - \bar{y}_i)^2}$

un valore vicino ad 1 indica un modello ben adattato.

La somma dei residui viene chiamata *funzione di perdita,* o *loss function,* e molte delle tecniche di regressione mirano a minimizzarla. Il metodo più conosciuto è quello dei **minimi quadrati:**

1. Per ogni punto noto *(x, y)* dell'insieme *N* di punti, calcolare x^2 e xy.
2. Sommare tutti gli x, y, x^2, e xy. Questo ci porta ad avere: $\sum x$, $\sum y$, $\sum x^2$, $\sum xy$;
3. Calcolare l'inclinazione della retta, tramite la formula:

$$\beta_1 = \frac{N \bullet \sum xy - \sum x \bullet \sum y}{N \bullet \sum (x^2) - (\sum x)^2}$$

4. Calcolare l'intercetta della retta, tramite la formula: $\beta_0 = \frac{\sum y - \beta_1 \bullet \sum x}{N}$
5. Mettere tutto insieme nella versione: $\tilde{y} = \beta_0 + \beta_1 \bullet x_1$

L'uso della regressione lineare assume:

- **Linearità dei dati,** cioè che la relazione tramite X ed Y sia lineare, ovvero rappresentabile tramite una funzione lineare.
- **Normalità dei residui,** cioè che gli errori residui siano normalmente distribuiti.
- **Omoschedasticità,** cioè che gli errori residui abbiano una varianza costante.
- **Indipendenza degli errori,** cioè che gli errori residui siano indipendenti per ogni valore di X: un test statistico utile è il *Durbin-Watson,* il cui risultato, in condizioni di errori indipendenti, è vicino a 2.

Il linguaggio di riferimento quando si parla di statistica è R, poiché implementa gia molte delle funzioni e dei test a cui si è fatto riferimento.

Quando la variabile indipendente è più di una si parla di **regressione lineare multipla:** valgono gli stessi vincoli della singola, per rappresentare le variabili si usa una forma matriciale e l'obiettivo diventa quello di trovare un piano che meglio interpoli i dati di training.

13.2 – Alberi decisionali regressivi

L'idea alla base degli alberi regressivi è di individuare regioni dello spazio in cui i valori interni assumono valori simili, per poi calcolarne un valore rappresentativo ed infine assegnare i nuovi dati alla regione più plausibile.

- Come primo passo, l'algoritmo posiziona come radice dell'albero la miglior caratteristica del training set, cioè quella caratteristica, chiamata *variabile di split,* tale che suddivida il set in modo che la somiglianza dei valori target sia massimizzata. Per fare ciò, viene calcolata, per ogni caratteristica, la **varianza** dei valori della variabile dipendente in questo modo: [sopra slide, sotto appunti]
  - $Var(D) = \frac{\sum (X - \mu)^2}{N}$
  - $Var(D) = \frac{1}{N} \cdot \sum_{i=1}^{n} (y_i - \bar{y})^2$
  - per cui la varianza totale è data dalla somma delle varianze calcolate per ogni valore della variabile indipendente.
- Come secondo passo si calcola il miglior punto di split, cioè quello che massimizza la riduzione della varianza.
- Come terzo, si divide il dataset in base al punto calcolato.

I tre passi sono ripetuti su ogni sottoinsieme fino al raggiungimento di un criterio di fermata, come profondità massima o un certo numero di campioni in una foglia.

Il valore di output di una foglia è spesso la media dei valori target delle istanze di dati che raggiungono quella foglia, ma è possibile usare anche indicatori diversi, ad esempio la mediana nel caso bisogni prestare attenzione agli outlier.

Gli alberi decisionali regressivi sono semplici ed efficaci, ma presentano alcune limitazioni: la prima è l'overfitting, a cui si va incontro quando diventano troppo grandi, e la seconda è l'instabilità e la terza è la difficoltà nell'interpolazione, cioè non generalizzano bene fuori dall'intervallo dei dati osservati. Come per gli alberi decisionali per la classificazione, anche qui possiamo usare diversi iper-parametri per mantenerli efficaci:

- **Massima profondità dell'albero,** che limita il numero di livelli.
- **Numero minimo di campioni per split,** che specifica un numero minimo di campioni necessari per dividere un nodo.
- **Numero minimo di campioni per foglia,** che specifica il numero minimo di esempi che devono essere presenti in un nodo foglia.

13.3 – Bonus

[**NB:** questa sezione tratta gli argomenti a partire dalla slide 34, che sono indicati sulle slide come non facenti parte dell'esame.]

Il machine learning non si limita soltanto a classificazioni e regressioni ed è bene sapere anche cosa c'è al di là dei modelli di base, in quanto i tradizionali non possono risolvere tutto.
L'**apprendimento per rinforzo** è un task in cui l'obiettivo è imparare i comportamenti corretti partendo da una conoscenza pregressa che va migliorata iterativamente tramite interazioni con l'ambiente e l'ottimizzazione di una *funzione reward*. In altri termini, bisogna massimizzare una funzione obiettivo, che qui è una rappresentazione numerica delle azioni dell'agente. Possiamo associare ad ogni azione una probabilità che indichi quanto questa possa portare ad un miglioramento della ricompensa e da questa definiamo una *politica di azione*, cioè una strategia di aggiornamento di probabilità ed azioni.

Partendo da un *neurone artificiale* possiamo costruire delle **reti neurali artificiali,** che non presentano alcuna informazione su ciò che bisogna predire. Sostituendo un nucleo semplice con una serie di layer nascosti stratificati, ciascuno specializzato in qualcosa, si può andare sempre più in profondità, ottenendo un **deep learner**.

Il **quantum computing** ha due idee alla sua base: *superposition*, cioè i q-bit possono assumere stati diversi allo stesso momento, ed *entanglement*, cioè i q-

bit possono essere fortemente connessi anche in assenza di interazioni fisiche dirette. Ad oggi siamo in attesa della cosiddetta "quantum supremacy", ovvero il momento in cui un programma interamente su quantum computing risolverà problemi che nessun programma tradizionale può risolvere in un ragionevole ammontare di tempo.
Il *quantum-enhanced machine learning* è la branca che si occupa di come ingegnerizzare architetture quantistiche che possano supportare l'addestramento dei modelli, che è la parte più costosa. Il quantum machine learning è anche associato all'uso di algoritmi per analizzare e predire dati quantistici e viene inoltre usato per problemi di ricerca (quantum annealing). La vera sfida consiste nel combinare componenti tradizioni e quantistiche, o addirittura usare solo quest'ultime, per effettuare l'analisi dei dati.
[Rimando al documento di lettura aggiuntiva "How Far Are We?"]

14 – Problemi di Clustering

Il **clustering** è un'attività di apprendimento non supervisionato in cui l'obiettivo è raggruppare degli oggetti in cluster tali che al loro interno gli oggetti abbiano alta omogeneità ed al loro esterno abbiano invece alta eterogeneità. Anche qui si effettua una classificazione dei dati, ma piuttosto che assegnare un'etichetta si creano classi di oggetti simili.
Alla base di questi algoritmi è il concetto di *similarità,* da cui dipende fortemente la qualità del clustering. Le misure di similarità sono definite tramite metriche, ovvero quantità calcolabili di distanza, a loro volta matematicamente definite come:

- Dato un insieme S di campioni, una distanza d è **metrica** se valgono le seguenti proprietà:
  - *Identità:* per ogni x appartenente ad S, d(x,x) = 0;
  - *Positività:* per ogni x diverso da y e appartenente ad S, d(x,y) > 0;
  - *Simmetria:* per ogni x,y appartenenti ad S, d(x,y) = d(y,x);
  - *Disuguaglianza triangolare:* per ogni x,y,z appartenente ad S, d(x,z) <= d(x,y) + d(y,z).

Se soltanto le prime tre proprietà sono rispettate si parla di **semi-metrica,** mentre se sono rispettate 1-3-4 allora si parla di **pseudo-metriche.**

Tra le metriche più comuni troviamo:

- la *distanza euclidea*, ideale per dati numerici continui;
- la *distanza di Manhattan,* che considera il valore assoluto delle coordinate dei punti ed è adatta per dati con coordinate discrete o categoriali;
- la *distanza di Mahalonobis,* che tiene conto delle correlazioni tra variabili;
- la *distanza Coseno,* che misura l'angolo tra due vettori nel loro spazio;

- la *distanza di Jaccard,* che misura la similarità tra insiemi come rapporto tra intersezione ed unione degli insiemi campionari ed è utile per dati binari ed insiemi;
- la *distanza di Canberra,* utile per dati normalizzati di cui enfatizza le differenze relative;
- la *distanza di Levenshtein,* che misura il numero minimo di modifiche elementari che consentono di trasformare una stringa X in un'altra Y.

14.1 – Tipologie di Algoritmi di Clustering

Una prima suddivisione degli algoritimi di clustering che possiamo fare è per tipologia:

- **Esclusivi,** se ogni pattern appartiene ad un solo cluster, o **non esclusivi,** se ciascun pattern può essere assegnato a più di un cluster.
- **Gerarchico,** se mira a costruire gerarchie di cluster, dette sequenze di partizione, o **partizionale,** se effettua solo una partizione del pattern.

Un altra suddivisione è invece quella per categorie:

- **Agglomerativo,** se unisce iterativamente dei cluster atomici per ottenerne di più grandi, o **divisivo,** se invece punta a suddividere un cluster in blocchi più piccoli.
- **Seriale,** se elabora un pattern alla volta, o **simultaneo,** se li elabora assieme.
- **Graph-theoretic,** se elabora i pattern in base alla loro collegabilità, o **algebrici,** se li elabora sulla base di criteri di errore.

La maggior parte degli algoritmi si basa su partizionamento iterativo ad errore quadratico e clustering gerarchico agglomerativo.

14.2 – K-means e Clustering Gerarchico

Il **k-means** è uno degli algoritmi più popolari per dati numerici ed ha lo scopo di suddividere il dataset in k cluster per minimizzare la varianza interna.
Funzionamento:

- vengono selezionati k centroidi in maniera casuale, cioè vengono scelti k pattern come rappresentanti;
- viene generato un partizionamento assegnando ogni campione al centroide più vicino;
- viene calcolato il nuovo centroide di ogni cluster, considerando la media dei valori;

- ripete il secondo step fin quando i cluster non restano uguali all'iterazione seguente.

È *iterativo* per il modo di costruire i cluster, *partizionale,* in quanto fornisce un'unica partizione degli elementi, e *ad errore quadratico,* poiché minimizza l'errore rispetto ai centroidi.
Per stabilire il valore da assegnare a k è possibile usare algoritmi di ricerca locale in cui la funzione obiettivo da ottimizzare è una metrica di valutazione della bontà dei cluster.

La normalizzazione dei dati e la rimozione degli outlier sono necessari per fornire al k-means dati distribuiti in modo più uniforme, ma è possibile agire anche a posteriori, rimuovendo cluster piccoli ed unendo quelli vicini.

Il **clustering gerarchico** è un metodo che costruisce una struttura ad albero detta *dendrogramma* per rappresentare una gerarchia di cluster: ogni nodo rappresenterà un cluster, mentre i livelli successivi indicano unioni/divisioni.
Ne troviamo due tipologie, a seconda di come avvenga la costruzione:

- ***divisivo,*** o top-down, se partiamo dalla similarità più bassa per arrivare agli elementi atomici;
- ***agglomerativo,*** o bottom-up, se partiamo dalla similarità più alta per arrivare ad un unico cluster.

La scelta che viene presa ad ogni passaggio consiste nel misurare la distanza tra cluster, che può essere calcolata in diversi modi:

- **Distanza minima,** o clustering del *nearest neighbor,* cioè la distanza tra due cluster è la distanza minima tra i punti dei due cluster.
- **Distanza massima,** o clustering del *farthest neighbor,* cioè la distanza tra due cluster è la distanza massima tra i punti dei due cluster.
- **Distanza media,** o clustering dell'*average linkage,* cioè la distanza è la media di tutte le distanze tra i punti dei due clustering.
- **Distanza centroidale,** o clustering del *centroid linkage,* cioè la distanza è calcolata tra i centroidi dei due cluster.

Anche il clustering gerarchio soffre di sensibilità agli outlier.

Il **clustering basato su densità,** di cui il più noto è DBSCAN, raggruppa i pattern considerando la loro densità nella distribuzione e pertanto risolve il problema delle forme che è presente nel k-means. In generale, ricorrono a due parametri: *minPts,* che stabilisce il numero minimo di punti per considerare un intorno di un punto denso, e *epsilon*, che definisce un intorno cicolare.

14.3 – Valutazione dei Risultati

Trattandosi di algoritmi non supervisionati, non abbiamo etichette a priori con cui eseguire confronti per capire la bontà del risultato ottenuto. Tra le tante metriche di stima presenti, le più usate sono:

- **Punto di gomito,** o Elbow Point, che viene usato per determinare il numero migliore di cluster da generare per gli algoritmi partizionali. Si tratta di un metodo empirico basato sull'osservazione che aumentando il numero di cluster la somma degli errori quadratici diminuisce con tasso decrescente.
- **Coefficiente di forma,** o Silhouette coefficent, che misura quanto sono simili gli elementi che compongono un singolo cluster. Si basa su due parametri: quanto bene i dati sono ammassati nel cluster di riferimento e quanto è distante ciascun campione da qualsiasi altro cluster.
  - $s(i)=\frac{b(i)-a(i)}{max(b(i),a(i))}$
  - con a(i) distanza media dell'i-esimo punto rispetto ai punti interni al cluster, b(i) quella rispetto ai punti appartenenti al cluster più vicino ed s(i) che rappresenta il coefficiente di silhouette.
- **MoJo distance,** che calcola il numero minimo di operazioni di spostamento e raggruppamento necessarie per passare dal clustering calcolato al clustering ideale, motivo per cui necessita di etichette iniziali.
- **Calinski-Harabasz Index,** che valuta la bontà del clustering in base alla varianza intra ed inter-cluster.

15 – Data Mining

Il data mining è il processo di estrazione di informazioni utili da un dataset, che porta alla scoperta di pattern e relazioni. I suoi scopi includono, ma non si limitano a: predizioni, classificazioni e clustering. Abbiamo diverse tecniche:

- *Data preprocessing,* che riguarda la pulizia, la trasformazione e l'organizzazione preliminare dei dati.
- *Pattern recognition,* che riguarda l'identificazione di pattern di relazioni.
- *Model building,* che riguarda la creazione effettiva dei modelli ed il loro successivo addestramento.

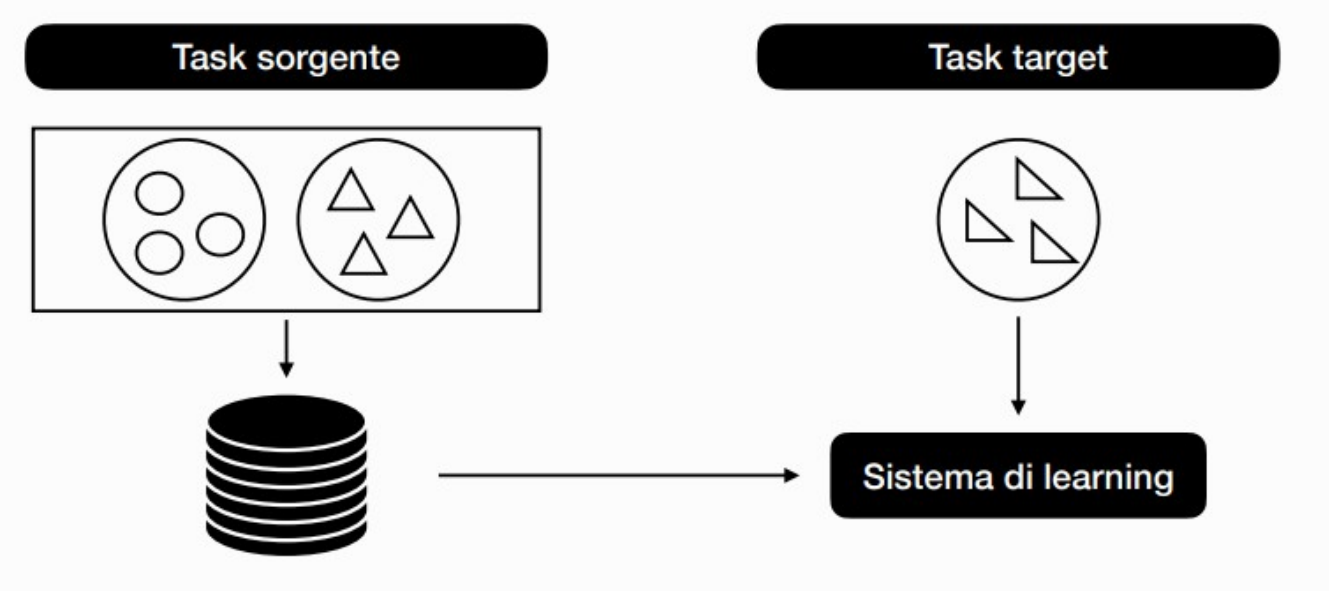
Tra i vari esempi che possiamo fare abbiamo i sistemi di recommendation di Amazon e Netflix, il motore di ricerca di Google, l'ottimizzatore della supply chain di Walmart, la guida autonoma di Tesla, l'aggiustamento dinamico dei prezzi di Airbnb e la pubblicità mirata di Faceook.

È facile notare come si tratti di applicazioni diffusissime il cui scopo varia dal migliorare l'esperienza di un utente all'ottimizzare delle operazioni.
Tecniche di machine learning, come il *collaborative filtering,* la *regression analysis* e le *reti neurali,* sono comunemente usate in queste applicazioni.

16 – LLM

Un **Large Language Model** è un modello di IA basato su reti neurali, addestrato su grandi quantità di testo per comprendere e generare linguaggio naturale ed utile per compiti come traduzioni, risposte e programmazione.

Possiamo collegarne la nascita ad un problema dei normali modelli di machine learning, cioè la necessità di effettuare l'addestramento da zero nel caso si voglia usare un modello finito in un ambiente diverso.
Come primo passo, è stato preso in considerazione il concetto di **riuso** dall'ingegneria del software, cioè, in questo caso, la pratica di usare un modello già addestrato come base per risolvere un altro problema. Schematicamente:

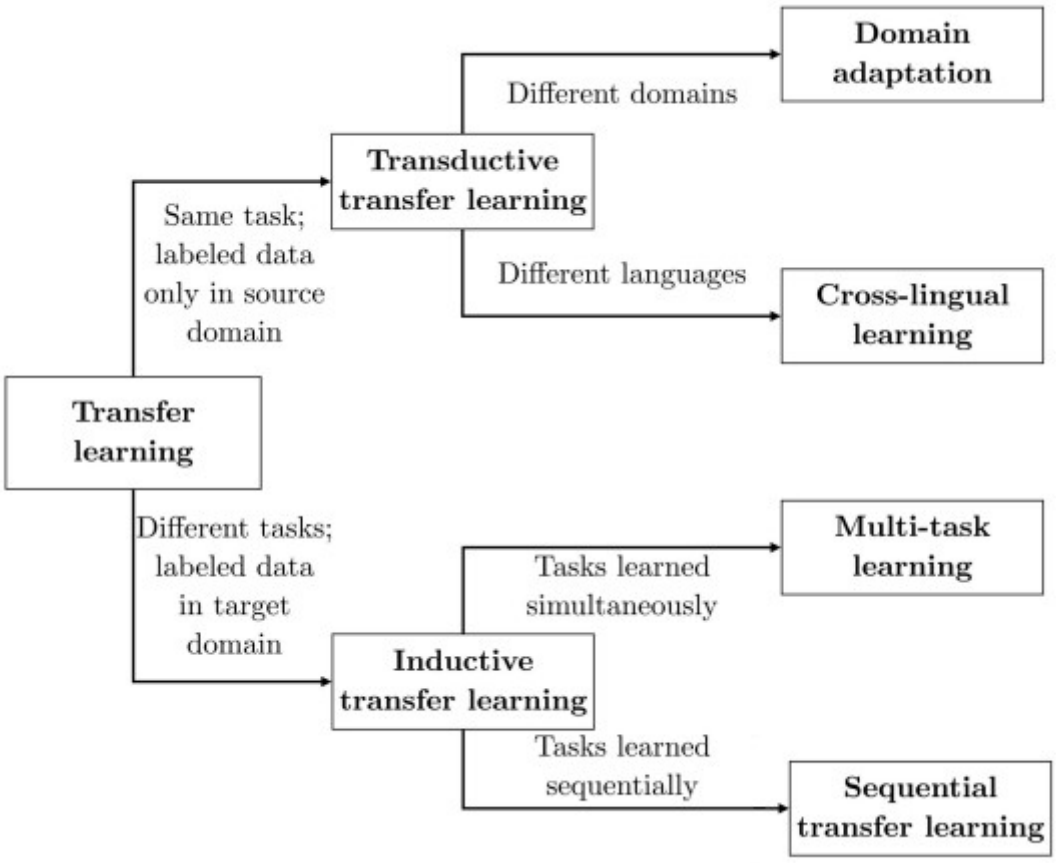


indichiamo con **Ds** [è un pedice] il dominio della sorgente dei dati, cioè l'insieme dei dati per costruire la conoscenza di base per il task sorgente **Ts,** e con **Dt** il dominio dei dati target, cioè quelli che il modello utilizza per adattarsi al task target **Tt** a partire dal task sorgente.

Possiamo definire il **transfer learning** come una tecnica di machine learning in cui un modello pre-addestrato sul dominio Ds per un task Ts viene riutilizzato,

adattato o perfezionato per risolvere un nuovo task Tt nel dominio Dt. Vale la seguente proprietà: *Ds != Dt OR Ts != Tt*

Ciò implica che possiamo riusare un modello addestrato nello stesso dominio variando il task richiesto o variare il dominio per addestrare un modello su uno stesso task. Da questo deriva una classificazione delle tipologie di transfer:



Nel **sequential transfer learning** il modello addestrato su un task sorgente viene successivamente adattato ad uno o più task target in modo sequenziale:

- Il modello viene addestrato su un dataset sorgente generico e di grandi dimensioni.
- Vengono congelati i primi layer e si procede ad addestrare il modello in modo sequenziale su dataset intermedi o target, che possono essere più specifici.
- Nei passaggi successivi, i dati del dominio di target sono usati per perfezionare ulteriormente il modello e soltanto una parte degli strati viene aggiornata.

Le reti neurali a cui si sta facendo riferimento sono architetture particolari dette **transformer,** che riescono a modellare relazioni complesse tra gli elementi di

una sequenza indipendentemente dallo loro posizione relativa. Possono analizzare ogni elemento della sequenza rispetto a tutti gli altri contemporaneamente, tramite il meccanismo di *self-attention.* Possiamo individuare due moduli chiave:

- **Encoder,** che prende in input una sequenza e la trasforma in una rappresentazione comprensibile per il modello. Lo possiamo immaginare come un traduttore che riassume il contenuto di una frase in un linguaggio comprensibile per il computer.
- **Decoder,** che genera una sequenza di output basata sulla rappresentazione creata dall'encoder e sugli elementi precedentemente generati. Può essere visto come uno scrittore creativo che prende il riassunto dell'encoder e lo trasforma in un testo leggibile, parola dopo parola.

Sono nari diversi modelli su questa base:

- **BERT** (Bidirectional Encoder Representations form Transformers), un modello pre-addestrato, progettato solo come encoder, su due task specifici: *Masked Language Modeling* e *Next Sencence Prediction*. Da questo sono derivati altri modelli, come *CodeBERT*, che è addestrato su dati di codice sorgente e documentazione, e *RoBERTa,* che invece ha un tipo di task di addestramento diverso (nello specifico, non ha il compito next sentence prediction).
- **T5** (Text-to-Text Transfer Transformer), un modello pre-addestrato, progettato sia come decoder che come encoder, che si occupa di una vasta gamma di compiti linguistici trattandoli come problemi di conversione da testo a testo.
- **GPT** (Generative Pre-trained Transformer), un modello pre-addestrato, progettato solo come decoder, che genera testo in modo sequenziale basandosi su ciò che è stato prodotto fino a quel momento.

Gli **LLM** sono uno dei tanti modelli detti "fondazionali", cioè pre-addestrati su vasti dati non supervisionati e capaci di generalizzare a molteplici task tramite fine-tuning o prompting. Gli elementi chiave che ne hanno determinato la nascita sono il transfer learning e le architetture capaci di gestire grandi moli di dati.

Eccellono nei task di elaborazione del linguaggio naturale (*NLP*), tra cui:

- *Sentiment Analysis,* che consiste nell'identificare il sentimento di un testo (positivo, negativo, neutrale).
- *Language Detection,* cioè determinare il linguaggio di un testo dato un input.
- *Part-of-Speech Tagging,* cioè identificare parti del discorso in un testo.
- *Question-Answering,* cioè fornire risposte basandosi sulle informazioni disponibili in un testo o altra fonte.

- *Summarization,* cioè produrre riassunti di testi.
- *Code Generation,* cioè generare snipper di codice da una descrizione in linguaggio naturale della funzionalità desiderata.
    - I primi 3 rientrano nel Language Understanding, gli altri nel Generation.

Alcuni problemi a cui si va incontro tipicamente quando si interagisce con un LLM sono:

- *Ambiguità,* in quanto i prompt devono essere chiari e specifici.
- *Sensibilità al contesto,* in quanto i prompt richiedono una comprensione del contesto che il modello potrebbe non avere o non saper interpretare.
- *Limitazioni del modello,* in quanto le risposte sono limitate alla conoscenza disponibile nel set di addestramento.
- *Conoscenza dinamica,* in quanto anche i prompt ben progettati possono fornire risposte solo entro i limiti delle capacità del modello.
- *Complessità,* in quanto i prompt necessitano di bilanciare il grado di dettaglio e la complessità.

Esistono però altri problemi riguardanti gli LLM, che vanno al di là della semplice propensione agli errori:

- *Flakiness,* che si riferisce all'incoerenza o variabilità delle risposte generate dal medesimo prompt.
- *Allucinazioni,* che si riferiscono alle volte in cui il modello presenta come plausibili delle informazioni false o non supportate dai dati disponibili.

Da ciò la necessità di definire strategie ben disciplinate per interrogare gli LLM, che ha portato alla nascita del **prompt engineering.** Si tratta della tecnica per progettare prompt ottimizzati per guidare un LLM verso risposte accurate e pertinenti e consiste nel formulare input chiari, specifici e contestuali. Esempi:

- *Few-shot learning,* in cui il prompt include alcuni esempi da cui il modello può imparare.
- *Chain-of-thoughts,* che consente al modello di risolvere un problema tramite una serie di step intermedi che ricalcano il ragionamento umano.
- *Emotion prompting,* in cui il prompt incorpora degli stimoli emotivi.
- *Secure prompting,* in cui vengono inseriti, a livello di prompt engineering, dei design pattern; queste istruzioni specifiche, se introdotte, rendono le risposte intrinsecamente più robuste e sicure.
- *Multi-agent architectures,* in cui i task sono divisi tra più agenti in base ai loro ruoli specifici.

[Da slide 64 in poi inizia dello yapping sulla sua visione del futuro che non vorrei riportare]