Intelligenza Artificiale e Machine Learning

Appunti delle Lezioni di Intelligenza Artificiale e Machine Learning
Anno Accademico: 2024/25

Giacomo Sturm

Dipartimento di Ingegneria Civile, Informatica e delle Tecnologie Aeronautiche
Università degli Studi "Roma Tre"

Sorgente del file LaTeX disponibile al seguente link:
https://github.com/00Darxk/Intelligenza-Artificiale-e-Machine-Learning

Indice

1 Introduzione: Intelligenza Artificiale e Machine Learning

L'intelligenza artificiale è l'area di studio che analizza come permettere ai computer di compiere operazioni che eseguite da un essere umano richiederebbero intelligenza. Rappresenta uno dei campi di ricerca più interessati recentemente ed ha avuto un'enorme crescita dal punto di vista economico e tecnologico.

Un modo per poter riepilogare la storia dell'intelligenza artificiale consiste nell'elencare i vincitori del Turing Award, in questo campo. Questo premio rappresenta l'analogo del premio Nobel per l'informatica.

1.1 Storia dell'IA

La prima fase di questa storia si può attribuire a personaggi come Alan Turing, che introdusse il concetto di "test di Turing" per determinare se una macchina si può considerare intelligente nel 1950. Una macchina secondo questo criterio si può considerare intelligente se un essere umano interagendo con essa, senza saperlo a priori, non sia in grado di sapere se si tratta di una macchina o di un altro essere umano.

Un altro di questi personaggi fondamentali è John McCarthy, ha coniato il termine "intelligenza artificiale", nel 1955 ed ha proposto la creazione di un gruppo di lavoro l'anno successivo al collage di Dartmouth. Al convegno di Dartmouth del '56 parteciparono tutti i personaggi che vengono chiamati i padri fondatori dell'IA. L'obiettivo del convegno era lo studio della costruzione di macchine in grado di formare astrazioni e concetti, per risolvere problemi, al tempo riservati agli umani. In questo convegno si teorizzò che queste macchine potranno svolgere funzioni umane, ritenute intelligenti.

In questo convegno Cliff Shaw, Allen Newell e Herbert Simon dimostrarono il primo programma di intelligenza artificiale, chiamato "Logic Theorist", in grado di dimostrare i teoremi dei Principia Mathematica. Proposero l'idea di "General Problem Solver" per risolvere una grande varietà di problemi simulando i processi mentali umani. Questo sistema riusciva a dimostrare teoremi, calcolare funzioni matematica e risolvere problemi logici. Da sistemi come questi si crearono grandi aspettative per l'intelligenza artificiale, in questa seconda fase tra il 1952 ed il 1969. In questo periodo Nathaniel Rochester all'IBM sviluppò tra i primi programmi di IA. Nel 1959 Herbert Gelernter scrisse il "Geometry Theorem Prover", capace di dimostrare teoremi matematici complessi. John McCarty nel 1958 all'MIT definì il linguaggio di lato livello Lisp, destinato a diventare il linguaggio di programmazione per eccellenza dell'IA per i seguenti trent'anni. Nel 1963 fondò il laboratorio dell'IA a Stanford, per costruire una versione definita dell'"Advice Taker".

Nonostante queste grandi aspettative, i sistemi di IA realizzati per problemi semplici non dimostravano un comportamento altrettanto soddisfacente per problemi di natura più complessa. Uno dei problemi per questo mancato successo dell'IA in questo periodo tra il '66 ed il '73 fu l'incapacità di comprendere l'intrattabilità di molti problemi, su cui l'IA stava cercando di operare. Nel 1973 venne stilato il rapporto Lighthill, dove venne criticata l'intelligenza artificiale per la mancata soluzione al problema dell'"esplosione combinatoria", ovvero sull'utilizzo dell'IA per problemi di natura reale.

Un altra critica giunse dal libro "Perceptrons" scritto nel 1969 da Marin Minsky e Seymour Papert che, come altri, discusse il limite fondamentale della struttura di base per generare un comportamento intelligente. In questo periodo il clima generale attorno all'IA era pessimista e libri come questo aiutarono a maturare questo clima di diffidenza nei confronti delle potenzialità dell'IA.

Negli anni successivi per risolvere il problema dell'esplosione combinatoria si realizzarono sistemi basati sulla conoscenza, riservati ad esperti umani del settore. Questi programmi erano estremamente utili su settori molto ristretti, costituiti da un motore di inferenza. Il collo di bottiglia per questi sistemi era la conoscenza, per cui esistevano intere posizioni dedicate a trovare un modo per rappresentare ed astrarre il dominio di interesse su cui opera l'IA. Queste persone venivano chiamate scienziati della conoscenza, ed erano tipicamente professionisti ed esperti di quel settore specifico.

Nel 1981 il governo giapponese avviò un progetto chiamato "Quinta Generazione" per realizzare computer intelligenti utilizzando il linguaggio Prolog. Analogamente sia gli Stati Uniti che l'Inghilterra finanziarono progetti simili.

Tuttavia questi progetti non riuscirono a mantenere le loro promesse sulle capacità dell'IA o di impatto commerciale. I sistemi erano comunque limitati da una conoscenza limitata ed una difficile capacità di apprendere dall'esperienza e verificare la correttezza. Erano sistemi poco flessibili e robusti rispetto agli obiettivi promessi da questi progetti.

Nonostante questo tra l'inizio e la fine degli anni '80 l'industria dell'IA conobbe un boom economico con centinaia di aziende costruttrici di sistemi esperti, di visione artificiale, di robot, di software ed hardware specializzati per questi scopi.

Molte di queste aziende fallirono per l'impossibilità di mantenere le loro promesse e seguì un periodo chiamato "inverno dell'IA'.

A metà degli anni '80 almeno quattro gruppi diversi reinventarono l'algoritmo di apprendimento basato sulla retropropagazione, sviluppato negli anni '60. Questi modelli furono considerati in diretta opposizione ai modelli simbolici di Newell e Simon e dell'approccio logicista di McCarthy. Geoff Hinton è una delle figure in primo piano nel risorgere delle reti neurali durante gli anni '80 e 2010, descrisse i simboli come l'"etere luminifero dell'IA".

Dai limiti dei sistemi esperti si introdusse un nuovo approccio basato sulla probabilità invece che sulla logica booleana, sul "Machine Learning" più che sulla programmazione manuale, su risultati sperimentali più che su affermazioni filosofiche. Questo permette a questi sistemi con logiche non cablate di poter apprendere dall'esperienza e migliorarsi nel tempo.

Nei primi anni del 2000 con lo sviluppo e l'espansione del "World Wide Web" ed il progresso sulla potenza di calcolo ha permesso di realizzare data set molto grandi, fenomeno indicato con il termine "big data". Questo ha portato allo sviluppo di algoritmi di machine learning progettati per trarre vantaggio da questi grandi insiemi di dati. La loro disponibilità ed il cambiamento di approccio verso il ML ha permesso all'IA di recuperare appetibilità commerciale.

Nel 2006 Geoffrey Hinton, introduce un algoritmo di apprendimento veloce per reti neurali, dando il via alla rivoluzione del "Deep Learning". Utilizzando molteplici livelli di elementi computazionali il ML diventa Deep Learning. Questo sistema ha ottenuto successi in molti domini applicativi, aumentando l'interesse verso l'IA.

Attualmente l'IA è in grado di lavorare in molti settori:

- Può guidare veicoli robotizzati come automobili o droni autonomi. Nel 2004 DARPA introduce una sfida per la guida autonoma di veicoli;
- Permette di eseguire locomozione su arti con robot umanoidi;
- Pianificazione e scheduling autonomo, per la navigazione autonoma ed il sistema GPS globale;
- Traduzione automatica;
- Riconoscimento vocale;
- Raccomandazioni su social network per gli utenti. Nel 2006 Netflix ha lanciato una borsa per aumentare la precisione del loro sistema di raccomandazione tramite ML, vinta nel 2009;
- Può battere i migliori giocatori umani in molti giochi. Nel 1997 Deep Blue batte il campione del mondo di scacchi Garry Kasparov, e nel 2011 IBM Watson batte i campioni del gioco "Jeopardy!", ora questo sistema è utilizzato in molti sistemi. Tra il 2015 ed il 2017 AlphaGo di DeepMind batte i campioni del mondo nel gioco di Go, lo stesso team ha sviluppato AlphaZero in grado di giocare sia scacchi, shogi e Go;
- Può interpretare e riconoscere immagini. Nel 2021 ImageNet comincia il suo concorso annuale per rilevare e classificare correttamente oggetti in un set di immagini ampio e ben accurato, con un incremento nella precisione dovuto ai progressi nelle "deep convolutional neural networks". Nel 2014 Facebook pubblica un lavoro sul DeepFace in grado di identificare volti con un'accuratezza del 97%;
- Permette di diagnosticare malattie, raggiungendo o superando la diagnosi di medici esperti;
- Recentemente ha accelerato la ricerca per il vaccino anti-Covid, nell'individuazione delle proteine utilizzate;
- Permette di rilevare informazioni dettagliate su eventi climatici estremi.

Nel 2014 il consumo di energia per raffreddare i Data Center è stato ridotto del 40% con un modello di ML, analogamente nel 2017 il software per analizzare le immagini di galassie sotto lenti gravitazionali è stato velocizzato di un fattore di 10^7.

Nel 2022 venne rilasciato ChatGPT, "Generative Pre-trained Transformer", il chatbot di OpenAI progettato per rendere efficace la comunicazione con un utente umano.

1.2 Il Machine Learning

L'intelligenza artificiale è una disciplina molto vasta, non copre solo tematiche sulla ricerca di una soluzione ottima. Esistono sistemi esperti per effettuare decisioni, offrire raccomandazioni, aiutare nella programmazione.

Dopo il 2006 le aziende hanno capito l'impatto che le raccomandazioni hanno nel mantenere l'attenzione del cliente e migliorare il loro servizio, quando Netflix ha annunciato un concorso di un milione di dollari per migliorare il suo programma di raccomandazioni del 10%. Da allora tutte le

principali aziende di investimenti, di viaggi, di intrattenimento, etc. Tutte le aziende che hanno di interesse l'interazione degli utenti.

Ora è veramente di interesse, infatti la massima conferenza sulla ricerca SIGIR "Special Interest Group on Information Retrieval", prima trattava principalmente problemi di ricerca, mentre recentemente la sua attenzione si è rivolta su sistemi di raccomandazione.

Un sistema di Machine Learning ML, all'interno di un dominio, dove ci sono dati validi con una loro consistenza, rappresenta un processo di apprendimento supervisionato o meno su questi dati per poter gestire e generalizzare i nuovi dati nello stesso dominio applicativo.

Uno dei grandi passi avanti nel campo delle immagini si deve al grande dataset ImageNet, grazie ad una ricercatrice e professoressa di Stanford Fei-Fei Li, che ha investito il suo tempo nel progetto. Questo insieme contiene milioni di immagini descritte e la loro classe di appartenenza dell'oggetto raffigurato, sono infatti principalmente immagini in primo piano. Tramite insiemi di dati come questi, il sistema apprende in un approccio supervisionato, poiché ad ogni immagine viene indicata la classe di appartenenza, ed in alcuni casi si indica dov'è presente l'oggetto nell'immagine. Date queste due indicazioni si possono creare sistemi di identificazione o individuazione di un'immagine. Questo approccio può realizzare sistemi di riconoscimento facciale, individuando le facce da un'immagine ed in seguito riconoscere le facce dall'immagine.

La maggior parte dei dataset saranno supervisionati, ma esistono moltissimi dataset i cui data-point non sono indicati o descritti. Questo tipo di apprendimento prevede l'assenza delle classi dell'appartenenza, questo viene utilizzato per creare sistemi che possono essere in grado di identificare il rischio di una certa malattia, tramite delle caratteristiche comportamentali o genetiche. Oppure si assegnano a degli algoritmi di clustering un insieme di utente, che sulla base delle caratteristiche di quest'ultimi individua delle classi di utenti. Nel diagramma di dispersione dove sono presenti questi data-point si possono quindi individuare cluster diverse. Si utilizza questo approccio nelle comunità social per identificare comunità distinte.

Se si trattano fenomeni fisici che non sono caratterizzabili a priori si effettuano degli studi per vedere se sono presenti correlazioni tra i data-point e quindi dei cluster, ed eventualmente individuare informazioni da questi dati.

L'apprendimento per rinforzo è molto comune, ed è un sistema dove un agente interagisce con un ambiente, senza un dataset, per permettere all'agente di operare sull'ambiente, questo fornisce delle ricompense all'agente in caso effettui un certo tipo di azioni. Queste ricompense possono essere positive o negative, e sulla base di queste, si realizza il meccanismo di apprendimento. Alcune aree di dominio che si predispongono a questo tipo di apprendimento, dove sulla base delle ricompense si possono suscitare reazioni e quindi comportamenti desiderati dall'agente, sono i videogiochi.

Grazie al DL si ha una sistema di "Deep Reinforcement Learning", dove il sistema è in grado di imparare direttamente da un'immagine, un video, uno spettrogramma, etc.

Nel 2013 la DeepMind, di proprietà di Google ha dimostrato la possibilità di un agente di apprendere tramite DRL, su numerose partite di Breakout un comportamento "super-umano", e diversi altri giochi arcade della Atari. Un altro grande dominio dove questo tipo di approccio è utile è la robotica, dimostrato dalla Boston Dynamics e dal loro robot umanoide Atlas, in grado di camminare, saltare, portare pesi, su diversi tipi di terreni autonomamente. Il loro successivo robot Spot, ha risolto alcuni dei problemi, non dovendo imparare una locomozione su due piedi molto più complessa, basandosi invece sullo scheletro di un animale quadrupede come un cane.

Il reinforcement learning venne introdotto per la prima volta una ventina di anni fa da Richard Button e Andrew Barto, degli psicologi cognitivi, senza tutta la serie di risorse di calcolo e di data per favorire questo approccio. Quindi non si è affermato per molto tempo.

Nel 1997 venne proposta l'idea della RoboCup, un campionato di calcio composto da squadre di robot autonomi, da parte di diverse università entro il 2050, in grado di confrontarsi con la squadra di calcio campione al mondo.

Il ML quindi ha l'obiettivo di generalizzare ed effettuare decisioni, il quanto più accurate, su un relativo dominio i cui dati non ha ancora visto.

L'apprendimento è una componente chiave del ragionamento, ma non è sufficiente a generare IA in grado di generare nuova conoscenza. Tecniche di generazione del testo, delle immagini e musica, etc., sono note già da anni, ma solo recentemente con ChatGPT utilizzando una serie di tecniche che consentono un addestramento molto accurato ed efficiente su di un patrimonio di dati che è sterminato. Tutte le aziende informatiche, capiscono che questo è una minaccia, poiché permetterebbe di catturare oltre alla frequenza dei termini anche il loro contesto. Dal contesto è quindi in grado di determinare la probabilità che una certa parola appaia. Infatti ChatGPT è uno strumento statistico, poiché non ha alcun tipo di ragionamento, per questo si parla anche di allucinazioni, poiché non può creare nuova conoscenza, ma offre esempi dove lo strumento non rinuncia di predire, ma produce un risultato scorretto.

Il processo di ML permetterebbe di realizzare software 2.0, tramite una rete neurale in grado di identificare le istruzioni da dover eseguire per effettuare una data azione, senza l'appoggio di un programmatore esterno. Questo solleva una serie di problematiche relativo ai pericolo del ML.

Jeremy Howard ha esposto su una "TED talk" un problema analogo alla rivoluzione industriale in termini di occupazione, dove l'80% delle persone lavora sui servizi, un dominio di interesse dove l'intelligenza artificiale è in grado di lavorare molto bene. Per cui quando questi sistemi entreranno a regime e le aziende si renderanno conto di questi strumenti così potenti a disposizione, allora sostituiranno questa occupazione umana, creando problemi occupazionali. Già nel 2015 nel concorso ImageNet, sullo stesso dataset, il tasso di errore di agenti intelligenti è sceso al disotto del tasso di errore umano nel riconoscimento delle immagini. Anche se l'essere umano è in grado di identificare in più classi distinte rispetto all'agente artificiale ed è in grado di contestualizzare queste immagine. Ma in alcune professioni quello che le macchine possono offrire, in termini economici, è più conveniente rispetto a quello che lavoratori umani possono offrire. Nel 2023 c'è stato lo sciopero più lungo nell'ambito di Hollywood, per l'introduzione dell'IA nel cinema, poiché questo avrebbe messo a rischio il loro lavoro e la loro occupazione. In seguito a questo sciopero le case cinematografiche hanno imposto dei limiti per evitare questo caso peggiore. Un altro problema che può presentarsi in circa 20 anni è la cosiddetta singolarità tecnologica, ovvero lo sviluppo ed il progresso scientifico sono più veloci della capacità di previsione o di comprensione umana.

Il ML rappresenta solo una piccola parte dell'intelligenza artificiale, ma comprende altri domini della scienza provenienti da diverse discipline come il "neurocomputing", la statistica come classificatori bayesiani, il riconoscimento di andamenti. Basti pensare a Geoffrey Hinton, Nobel per la fisica nel 2024, psicologo cognitivo informatico, chiamato il padre fondatore dell'IA, vincitore del premio Turing.

L'intelligenza artificiale si utilizza in domini dove l'approccio di forza bruta, di provare tutte le possibili combinazioni per risolvere un problema. Si usano delle euristiche che rappresentano

una conoscenza su un dato dominio e permettono di rendere più efficiente l'algoritmo, ma non garantiscono il miglioramento delle prestazioni nel caso peggiore.

Si analizzerà principalmente l'IA debole, dove non sono presenti caratteristiche di ragionamento e comprensione, ma comunque in grado di risolvere problemi complessi. Una particolare famiglia sono gli algoritmi genetici, oggetto di interesse in corsi futuri, basati sul concetto di evoluzione, a cui si ispirano per implementare metodi di ottimizzazione.

Turing è sia il padre dell'informatica moderna sia dell'intelligenza artificiale. Un grande banco di prova del Machine Learning sono i videogiochi, poiché è sufficiente fornire i pixel dell'immagine dal primo strato della rete neurale. Nel gioco degli scacchi una volta che si ha una possibile mossa, il sistema deve considerare l'albero delle possibili mosse disponibili fino ad una certa profondità, tanto maggiore quanto maggiore l'accuratezza della ricerca. Ad ogni configurazione della scacchiera si assegna un valore di "score" nell'intervallo $[-\infty, +\infty]$. Alcune delle tecniche che si effettuano sull'albero di ricerca min-max sono tecniche di potature per snellire questo albero. Il primo sistema è stato sviluppato da Arthur Samuel nel 1952, uno dei partecipanti al workshop di Dartmouth , per il gioco della dama. Questi algoritmi di reinforcement learning si è affermato successivamente quando le prestazioni dei calcolatori lo ha permesso. Non è un apprendimento supervisionato, ma l'informazione utilizzata dal sistema per addestrare l'agente è l'informazione ottenuta dalla reazione dell'ambiente in seguito alla sua azione. L'apprendimento utilizzando questa tecnica è molto più facile e molto più veloce. Giochi come gli scacchi non si prestano ad un approccio di forza bruta poiché il numero di posizioni possibili è maggiore del numero di atomi dell'universo.

Un momento importante nel 1997 fu la vittoria di Deep Blue, IA creata dall'IBM vinse a scacchi contro il campione del mondo Garry Kasparov. L'IBM inoltre ha investito molto sul sistema chiamato Watson, a dissezione delle potenze di calcolo massime dell'epoca e sconfisse i campioni di Jeopardy. Dall'impatto sulla comunità non scientifica l'impatto pubblicitario che ha avuto fu molto elevato. Watson aveva a disposizione una serie di risorse bibliografiche tutte in RAM, circa 4 TB. Watson era indicato dall'IBM nell'assistenza medica per la diagnosi, con risultati fino ad ora al di sotto delle aspettative iniziali. Tuttavia in sistemi come questi, come ChatGPT possono generare comportamenti emergenti, comportment molto diversi da quelli aspettati, di cui ancora non si conosce a fondo il motivo.

Le maggiori aziende come Google, Apple, Meta, etc. sono molto veloci ad acquistare anche a costi elevati società e startup i cui risultati sembrano promettenti, sotto forma di investimenti.

Un altro momento cruciale fu la vittoria di AlphaGO, sviluppato da Deep Mind, divisione di Google, quando ha battuto il campione coreano di Go, Lee Sedol, le cui configurazioni sono molte di più del gioco di scacchi.

In seguito a questo poiché il gioco del Go è estremamente diffuso in Cina, queste notizie vennero soppresse poiché vennero interpretato come un attacco all'orgoglio cinese.

La stessa Deep Mind ha mostrato come sia possibile realizzare agenti in grado di battere i campioni mondiali su giochi arcade della Atari.

Sul campo della robotica i risultati ottenuti sono scadenti rispetto a compiti che richiedono autonomia, o interpretazione o in ambienti sconosciuti. Anche se sono molto avanzati in termini della meccanica, elettronica e di controllo. Recentemente la Boston Dynamics ha deciso di abbandonare i suoi investimenti su di Atlas. Spot ha quindi preso il posto di Atlas, un robot con le fattezze di un cane, più semplice da realizzare rispetto ad un sistema bipede.

Dato un problema, un'istanza è un insieme di caratteristiche o "features", che possono essere di interesse "target" o meno. Queste vengono raccolte per ogni problema. E da questo insieme di caratteristiche si può generare un albero di decisione binario, dove ogni biforcazione è data da una condizione su una certa feature. Questo albero è completo se arrivando ad una foglia, questa è pura, ovvero non sono richiesti altri controlli sulle caratteristiche dell'istanza per determinare una soluzione. Per generare questo approccio si possono attuare approcci greedy che creano tutte le possibili decisioni, oppure in modo più efficiente utilizzando la teoria dell'informazione di Shannon, e da euristiche legate al problema. Nel caso dell'informazione l'entropia è il numero di bit necessario in media per memorizzare il messaggio da una sorgente. L'entropia è tanto più elevata quanto è imprevedibile il messaggio proveniente da una sorgente, tanto maggiore è il numero di bit necessari per memorizzarlo. Per cui per ogni ramo si cerca la caratteristica che fornisce il maggior guadagno di informazione, in questo modo si diminuiscono il numero di caratteristiche da dover controllare per dover raggiungere una soluzione.

Uno dei risultati importanti è Eliza, il primo chatbot della storia, un punto in avanti in IA e ML, e nella risoluzione dei problemi negli anni '70. Questi erano strumenti statistici. La differenza tra quei modelli statistici ed i modelli odierni, è la quantità di dati su cui sono stati addestrati. Uno dei primi ambiti dell'IA fu la traduzione automatica, ma nonostante tutti i sistemi proposti per questo utilizzo. La maggior parte dei finanziamenti venivano dall'impianto militare, negli Stati Uniti specialmente dalla DARPA. Negli anni '80 Marin Minsky consigliò di non utilizzare reti neurali, nonostante le loro possibilità, poiché le risorse di calcolo del tempo non lo permettevano.

Nell'idea delle CNN le prime reti neurali, vennero analizzate reti neurali animali, sull'aspetto visivo per tentare di copiare la natura ed ottenere gli stessi effetti benefici. Negli anni 2000 il classificatore Bayesiano si può dimostrare essere il migliore classificatore possibile, in condizioni di incertezza. Un altro tipo di macchina le Support Vector Machine, si utilizza teoria e principi di ottimizzazione, introdotta negli anni '90. Si basa su principi geometrici invece che probabilistici, rappresentando i vari punti, che appartengono a determinate classi, su un iperpiano, in un ambiente di apprendimento supervisionato. Queste SVM individuano l'iperspazio di separazione ottimo, per ottimizzare al meglio il processo di Machine Learning.

Oltre a caratteristiche testuali si potrebbero utilizzare feature visuali per poter identificare caratteristiche da parte di immagini nel riconoscimento. Si vuole utilizzare feature che permettono il maggiore guadagno di informazioni. Dalle feature primarie si possono realizzare feature secondarie, le SIFT, "Space Invariant Feature Transformer", delle caratteristiche derivate invariante rispetto ad una serie di trasformazioni geometriche. In modo da riconoscere la caratteristica nonostante sia stata effettuata una di queste trasformazioni.

Nel 2012 venne rivoluzionata la ImageNet challenge nell'ambito della visione artificiale e riconoscimento di immagini da parte di IA. Esistono delle piattaforme tramite cui chi ha i fondi di ricerca, dato un obiettivo, può chiedere di ottenere etichette su un set di dati. Queste piattaforme vengono chiamate turchi meccanici, poiché le operazioni vengono svolte da umani e non da macchine. Ha favorito lo sviluppo e l'introduzione del "Deep Neural Network", DNN, aumentando novelette le prestazioni su queste task. Questo fu dovuto a Geoffrey Hinton ed un suo studente Alex Krizhevsky, dall'università di Toronto, che realizzarono questa tecnologia di DNN. Vennero in seguito assunti da Google per implementare questa tecnica nei loro prodotti, chiamata AlexNet. Per cercare un'immagine si cercava dalla sua caption, mentre le immagini non etichettate erano effettivamente

invisibili a questo approccio. Con questa tecnologia invece è possibile individuare anche immagini non etichettate, cercando immagini simili da quella fornita dall'utente.

Con l'introduzione delle reti ricorrenti le loro prestazioni incrementarono notevolmente. Google utilizza infatti reti neurali ricorrenti GMLT per la traduzione automatica del linguaggio naturale. Già Turing in alcuni dei suoi studi ha mostrato come è possibile prevedere data una frase la prima lettere o parola della frase successiva. Questi sistemi di traduzione automatica tuttavia falliscono se l'utente ha un accento diverso da quello inglese, su cui sono stati principalmente addestrati. Ulteriormente l'errore aumenta notevolmente se aumenta il rapporto segnale rumore SNR, sia per un agente artificiale sia che per un utente umano.

Si sono realizzati strumenti di riconoscimento di oggetti e di immagini. YOLO, "You Only Look Once" è uno strumento che effettua un compromesso tra velocità ed attendibilità nel riconoscimento degli oggetti e della locazione. Fornisce la probabilità secondo cui l'oggetto che si sta guardando appartiene ad una certa classe e fornisce anche questo valore in percentuale. Soprattutto permette di riconoscere con certa accuratezza anche in ambienti "cluttered" con un alta complessità della scena in tempo reale. Questo è un programma open source, scaricabile gratuitamente, per cui è presente in molte applicazioni come la guida autonoma, il riconoscimento facciale, l'analisi di radiografie, etc. Un altro strumento utile per la programmazione è lo standard OpenCV, "Computer Vision", che ha standardizzato il modo in cui si gestisce la visione virtuale di un calcolatore. Realizzato e reso disponibile da un ingegnere dell'Intel Garry Bradski nei primi anni 2000, da allora si è affermato come lo standard de facto per la computer vision. Negli anni 2000 ogni volta che una certa azienda o start-up raggiungeva un certo livello tecnologico, veniva acquisita da parte delle più grandi aziende del settore ICT, "Information e Communication Technologies", e le loro tecnologie venivano implementate nei loro servizi.

Nel 1995 era nato un nuovo motore di ricerca Netscape che fu valutato oltre un miliardo di dollari, nonostante non avesse un fatturato a quel livello. Dopo l'11 Settembre gli investimenti in questi settori scesero nettamente. Gli investimenti risalirono dopo il 2012 con l'avvento del Deep Learning.

2 Risoluzione dei Problemi e Ricerca

Si definisce un agente risolutore di problemi un agente con uno specifico obiettivo da raggiungere e che deve identificare una sequenza di azioni per raggiungerlo.

Bisogna determinare l'obiettivo, un insieme degli stati del mondo dove ci si trova, che si vuole raggiungere. Inoltre bisogna formulare il problema, ovvero le azioni e gli stati considerati dall'agente.

Un agente che ha a disposizioni diverse opzioni immediate di valore sconosciuto, può decidere quale scegliere la sua azione analizzando le diverse possibili sequenze di azioni, che portano a stati di valore conosciuto, per scegliere la sequenza di costo migliore.

Questo processo di selezione viene definito ricerca, un algoritmo di ricerca quindi prende un problema e restituisce una soluzione costituita da una sequenza di azioni.

Nella formulazione si definisce uno stato iniziale e dell'obiettivo, come un insieme di stati e si definiscono le azioni come transizioni tra stati. Dopo aver trovato la sequenza di azioni corrispondente alla soluzione, la esegue.

Si possono distinguere due tipi di problemi, i problemi giocattolo o "toy problems", sono ideati come illustrazione o esercitazione dei metodi risolutivi. Rappresentano delle astrazioni, anche semplificate, dei problemi del mondo reale in generale più difficili a cui si è effettivamente interessati.

I problemi del mondo reale possono essere la configurazione VLSI, la navigazione dei robot, la sequenza di montaggio, la ricerca dell'itinerario e generali problemi di viaggio come il commesso viaggiatore.

Si possono utilizzare due tipi diversi di formulazione di un problema. Si può partire da uno stato vuoto, ed utilizzando operatori si può estendere progressivamente la descrizione dello stato. Invece nella formulazione a stato completo consiste nel partire da uno stato iniziale completo, ed ogni operatore altera questo stato per cercare una soluzione.

Lo spazio degli stati è un grafo che rappresenta tutti i possibili stati, come nodi, collegati tra archi che rappresentano le possibili azioni. Il problema consiste nel trovare un percorso in questo spazio degli stati, dallo stato iniziale ad uno dei possibili stati soluzione. L'algoritmo deve decidere ad ogni stato quale azione prendere e quindi a quale nodo del grafo spostarsi. Per definire il costo della soluzione, si considera il costo del cammino delle azioni intraprese dall'agente.

Per definire formalmente un problema, sono necessarie quattro componenti. Uno stato iniziale in cui si trova l'agente. Una descrizione delle azioni possibili, questa può utilizzare una funzione successore che dato uno stato restituisce i suoi possibili successori. Tramite la funzione successore e lo stato iniziale si può costruire lo spazio degli stati. Oppure si può utilizzare un insieme di operatori. Un test obiettivo per determinare se un particolare stato è uno stato obiettivo. Come ultimo componente è necessaria una funzione di costo del cammino per determinare il costo di una data soluzione, assegnando un valore numerico ad ogni cammino.

Il tipo di dato problema è rappresentato da questi quattro componenti, le istanze di questo tipo di dato rappresentano gli input degli algoritmi di ricerca.

Il mondo reale è estremamente complesso, e lo spazio degli stati deve essere creato mediante un processo di astrazione. In questo processo, lo stato astratto rappresenta un insieme di stati reali più complessi. Analogamente per le azioni astratte, queste rappresentano combinazioni di azioni reali. Nei problemi giocattolo non è necessario effettuare questo processo di astrazione, poiché rappresenta un problema semplice.

Per individuare queste possibili sequenze di azioni l'algoritmo si può costruire un albero di ricerca, dove il nodo radice corrisponde allo stato iniziale, ed i rami rappresentano le azioni possibili, ed i nodi figli rappresentano gli stati successori di un certo stato. Tuttavia i nodi dell'albero di ricerca e gli stati dello spazio degli stati sono differenti, poiché è possibile che più nodi condividano lo stesso stato, mentre ogni stato nello spazio è univoco. Generalmente si vuole evitare di ripetere stati all'interno di un cammino, questo rappresenta il problema degli stati ripetuti, e sarà analizzato successivamente.

Ad ogni nodo si possono inserire altre informazioni utili, oltre allo stato, un riferimento al genitore, l'operatore che ha generato lo stato, la profondità, il costo del cammino parziale fino a questo stato, etc.

```
NODO = <stato, genitore, operatore, profondità, costo parziale, ...>
```

Il processo di ricerca comporta la stessa sequenza di azioni. Deve scegliere tra le foglie dell'albero corrente un nodo da "espandere", secondo un certo criterio o strategia. In seguito bisogna determinare se questo nodo rappresenta un'obiettivo del problema, altrimenti vengono generati i suoi nodi figli, ed i corrispondenti stati successori, e tutte le componenti dei nodi figli. La collezione dei nodi in attesa di essere espansi viene chiamata in vari modi: confine, frontiera, frangia o lista aperta.

2.1 Algoritmo di Ricerca Generale: Tree Search

Un algoritmo generale di ricerca può essere chiamato `TREE-SEARCH`. Quest'algoritmo prende un problema ed una strategia come input e restituisce una soluzione oppure un fallimento. Viene realizzato semplicemente ad un ciclo che ripete le operazioni precedentemente descritte, fino a quando non identifica una soluzione o viene sollevato un problema, e quindi restituisce un fallimento. Il primo passo è la generazione dell'albero di ricerca del problema, se la frontiera è vuota, ovvero non esistono nodi candidati per l'espansione viene riportato un fallimento, poiché non è stato ancora trovato uno stato obiettivo. Se si arriva ad un nodo corrispondente ad uno stato obiettivo viene restituita rappresenta la sequenza di nodi ottenuta come soluzione.

La frontiera può contenere un nodo relativo ad uno stato obiettivo, ma l'algoritmo non termina fino a quando non viene scelto per essere espanso.

Una strategia di ricerca rappresenta un criterio per decidere quale nodo da espandere. Può essere definita come una funzione per la scelta di un elemento tra un insieme di nodi, la frontiera. Oppure si può considerare come una funzione di inserimento di un elemento in una sequenza. Se già nella fase di inserimento si analizza tramite una metrica il valore di ognuno di questi stati, allora l'elemento in prima posizione in questa struttura dati rappresenta il nodo migliore.

2.1.1 Operazioni su Frontiera, Nodi e Problemi

La frontiera viene implementata con una struttura dati chiamata coda, ma non necessariamente segue la disciplina FIFO. Su questa coda sono definite una serie di operazioni:

- `MAKE-QUEUE`: prende come input un nodo `n` e restituisce una coda `q`, realizza una coda contenente solo il nodo `n`;

- `EMPTY`: prende come input una coda `q` e restituisce un booleano, verifica se la coda è vuota;
- `REMOVE-FRONT`: prende come input una coda `q` e restituisce il primo nodo `n` della lista;
- `QUEUING-FN`: prende come input una coda `q` ed una lista di nodi `n`, e restituisce la coda con aggiunti tutti questi nodi.

L'ultima funzione si utilizza quando si producono tutti i nodi successori e si vogliono aggiungere alla lista. Questa funzione non inserisce generalmente in coda, ma dipende dalla strategia di ricerca utilizzata.

Assumendo che esistano le operazioni sul tipo di dato problema, e sul tipo di dato nodo. Le operazioni sul tipo di dato problema sono:

- `INITIAL-STATE`: prende come input un problema `p` e restituisce lo stato iniziale del problema `n`;
- `GOAL-TEST`: prende come input un problema `p` ed uno stato `n` e verifica se questo rappresenta una soluzione, restituendo un booleano;
- `OPERATORS`: prende come input un problema `p` e restituisce una lista con tutti gli operatori del problema `ops`. Ogni operatore `op` applicato ad uno stato `n` restituisce una lista di stati `ss`.

Le operazioni sul tipo di dato nodo sono:

- `MAKE-NODE`: prende come input uno stato `s` e costruisce un nodo su di esso `n`;
- `STATE`: prende come input un nodo `n` e ne restituisce lo stato contenuto `s`;
- `EXPAND`: prende come input un nodo `n` ed una lista di operatori `ops` e restituisce una lista di nodi successori `ns`.

2.1.2 Implementazione

Date queste operazioni, si può rappresentare in pseudocodice l'algoritmo di `TREE-SEARCH` in modo più semplice:

```
function TREE-SEARCH(problem) returns a solution or failure

fringe <- MAKE-QUEUE(MAKE-NODE(INITIAL-STATE(problem)))
loop do
    if EMPTY(fringe) then return failure
    node <- REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE(node)) then return SOLUTION(node)
    fringe <- QUEUING-FN(fringe, EXPAND(node, OPERATOR(problem)))
end
```

In questa variazione non viene conservato l'intero albero, ma solamente la coda con i nodi della frontiera.

2.2 Criteri di Valutazione

Per valutare questi algoritmi oltre alla complessità temporale e spaziale, si utilizzano altre due criteri, la completezza e l'ottimalità. Un'algoritmo si definisce completo, se quando esiste una soluzione è garantito sia in grado di trovarla. Un algoritmo si dice ottimo se dato un problema con diverse soluzioni, individua sempre la migliore, quella a costo minimo. La complessità dell'algoritmo dipende dal fattore di ramificazione b dello spazio degli stati e dalla profondità d della soluzione più superficiale. Il fattore di ramificazione b rappresenta il massimo numero di figli che un nodo può avere. Mentre la profondità d è la minima lunghezza di un cammino dal nodo iniziale alla radice.

2.3 Ricerca non Informata o Cieca

Questi algoritmi non sono molto efficienti in generale, ma sono utili per comprendere il comportamento gli algoritmi di ricerca informata o di euristica, che si avvalgono della conoscenza sul domino dello spazio degli stati e dalla creazione dell'albero di ricerca per scegliere il percorso più promettente. Nel caso medio quest'ultimi sono certamente più efficienti degli algoritmi trattati in questa sezione.

2.3.1 Algotimo di Ricerca in Ampiezza: Breadth First Search (BFS)

Nella ricerca in ampiezza si espande il nodo radice, e si espandono i nodi generati dalla radice, e si ripete per ogni nodo successore. Per implementare un algoritmo che utilizza una strategia di ricerca non informata in ampiezza, la "Queuing Function" inserisce i nodi appena generati in coda. Questa funzione quindi rappresenta sempre un inserimento in coda e si può chiamare "Enqueue at the End": `ENQUEUE-AT-END`.

Un algoritmo che utilizza questo tipo di strategia viene chiamato "Breadth First Search" o in ampiezza, e si può implementare in modo analogo all'algoritmo di ricerca generale trattato precedentemente:

```
function BREADTH-FIRST-SEARCH(problem) returns a solution or failure

fringe <- MAKE-QUEUE(MAKE-NODE(INITIAL-STATE(problem)))
loop do
    if EMPTY(fringe) then return failure
    node <- REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE(node)) then return SOLUTION(node)
    fringe <- ENQUEUE-AT-END(fringe, EXPAND(node, OPERATOR(problem)))
end
```

In questo approccio, tutti i nodi di profondità d vengono espansi prima dei nodi di profondità $d+1$. Rappresenta una strategia sistematica, ma permette di individuare solamente i nodi obiettivi più superficiali, non è garantito che questo rappresenta la soluzione ottima. Questo algoritmo è quindi completo, ma non è ottimo. Invece è ottimale se il costo del cammino $g(n)$ è una funzione monotona non decrescente della profondità del nodo $p(n)$:

$$p(n) < p(n) \implies g(n) \leq g(m)$$

$$p(n) = p(m) \implies g(n) = g(m)$$

Ovvero se due nodi n ed m sono a profondità diverse, dove il nodo m è a profondità maggiore, il costo del cammino dalla radice al nodo n è al massimo uguale al costo del cammino dalla radice al nodo m. Inoltre se i nodi sono alla stessa profondità, allora i costi dei loro cammini dalla radice sono uguali.

Utilizzando questo algoritmo, bisogna generare un numero di nodi, prima di trovare una soluzione, almeno pari a tutti i nodi precedenti al nodo soluzione. Nel caso peggiore questo nodo è l'ultimo nodo espanso alla profondità d, e per ogni nodo vengono generati esattamente b figli, quindi bisogna espandere al massimo un numero di nodi N pari a:

$$N = \left(\sum_{i=1}^{d+1} b^i \right) - b$$

Supponendo che ogni generazione rappresenta un'operazione semplice allora la complessità temporale di questa ricerca è $O(N) = O(b^d)$. La complessità temporale è analogamente $O(b^d)$ poiché bisogna memorizzare tutte le foglie generate.

2.3.2 Algoritmo di Ricerca Guidata dal Costo: Dijkstra

Modificando la ricerca in ampiezza espandendo il nodo della frontiera di costo più basso, si può aumentare l'efficienza dell'algoritmo precedente.

Si definisce con $g(n)$ il costo del cammino dalla radice al nodo n. Questo valore viene salvato nella struttura dati nodo, e viene scelto il nodo di costo $g(n)$ minore per essere espanso. Se il costo del cammino corrisponde alla funzione di profondità, si ha la ricerca in ampiezza. Questo algoritmo è completo e ottimale quando il costo di ogni step è sempre maggiore o uguale ad una costante positiva ε. Per cui è garantito che non attraversi costantemente lo stesso cammino, senza espandere altri nodi di profondità minore.

I costi di ogni nodo vengono salvati in un campo etichetta nella struttura dati nodo. Dalla frontiera si estrae sempre il nodo a costo minore, questa collezione viene quindi ordinata in base al costo delle etichette di ogni nodo.

2.3.3 Algoritmo di Ricerca in Profondità: Depth First Search (DFS)

La ricerca in profondità consiste nell'espansione del nodo più profondo, dopo aver espanso la radice. Per implementare questa funzione, si utilizza una queuing function che inserisce i nodi appena espansi all'inizio della lista, con una "Enqueue at the Front": `ENQUEUE-AT-FRONT`:

```
function DEPTH-FIRST-SEARCH(problem) returns a solution or failure

fringe <- MAKE-QUEUE(MAKE-NODE(INITIAL-STATE(problem)))
loop do
    if EMPTY(fringe) then return failure
    node <- REMOVE-FRONT(fringe)
```

```
    if GOAL-TEST(problem, STATE(node)) then return SOLUTION(node)
    fringe <- ENQUEUE-AT-FRONT(fringe, EXPAND(node, OPERATOR(problem)))
end
```

Questa funzione si può implementare mediante una funzione ricorsiva, dove viene passata una versione del problema, dove i nodi appena generati rappresentano i nuovi nodi radice. Quindi per ogni espansione vengono generati al massimo b sotto-problemi, risolti dallo stesso algoritmo. La lista dei nodi da visitare viene conservata implicitamente nello stack dei record di attivazione delle varie chiamate ricorsive. Quando si raggiunge un nodo foglia non obiettivo, si effettua il backtracking, ovvero si risale l'albero fino a trovare il nodo a profondità maggiore non ancora espanso su cui è possibile effettuare una scelta.

Questa ricerca non è né completa né ottimale, ha una complessità temporale di $O(b^m)$, dove m rappresenta la profondità massima dell'albero di ricerca. Se una soluzione è presente a profondità minore nel sotto-albero di destra, non verrà mai individuata se non è stato già espanso tutto il sotto-albero di sinistra, senza aver trovato una soluzione. Quindi se individua una soluzione la restituisce indipendentemente dalla sua ottimalità.

Si guadagna rispetto alla ricerca in ampiezza nella complessità spaziale. Infatti non bisogna memorizzare l'intero albero, ma solamente il cammino dalla radice alla foglia, ed i fratelli non espansi di ciascun nodo del cammino di profondità m: $O(b \cdot m)$. Se l'albero ha rami infiniti, allora la ricerca non termina.

2.3.4 Algoritmo di Ricerca in Profondità Limitata

Nella ricerca in profondità limitata si impone un limite alla profondità massima, per impedire di proseguire all'infinito su uno stesso cammino. Un nodo viene espanso solo se la lunghezza del cammino dalla radice al nodo è minore del massimo stabilito. Se non viene trovata alcuna soluzione restituisce il valore speciale taglio se alcuni nodi non sono stati espansi, altrimenti fallisce.

Si possono utilizzare conoscenze specifiche al problema per fissare questo limite. Se si lavora su di un grafo si potrebbe utilizzare il diametro del grafo come la profondità. Se non si sceglie un valore adeguato per questo limite allora l'algoritmo non funzionerà correttamente. L'algoritmo è completo, se la soluzione è ad una profondità minore della lunghezza l imposta, mentre non è ottimale. La complessità temporale e spaziale è rispettivamente $O(b^l)$ e $O(b \cdot l)$. Risolve il problema della completezza, ma non risolve l'ottimale.

2.3.5 Algoritmo di Ricerca Iterative-Deepening Search

Questo algoritmo risolve il problema dell'ottimalità sugli algoritmo di ricerca in profondità senza conoscere un limite adeguato. Evita il problema della scelta del limite provando iterativamente tutti i limiti possibili fino a quando non individua una soluzione:

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns solution or failure

for depth = 0 to ∞   do
    if DEPTH-LIMITED-SEARCH(problem, depth) succeeds
```

```
        then return its result
    end
```

Questo approccio combina i benefici di una ricerca in ampiezza con i benefici di una ricerca in profondità, poiché per ogni profondità vengono analizzati tutti i nodi.

Quindi questo algoritmo è ottimale e completo per le condizioni della ricerca in ampiezza. La complessità spaziale è $O(b \cdot d)$, quindi non è esponenziale. Mentre la complessità temporale è simile a quella a quella della ricerca in ampiezza. L'algoritmo viene richiamato ogni volta che si aumenta il limite, quindi i nodi a profondità minore vengono generati ogni volta che viene eseguito nuovamente l'algoritmo. Quindi vengono generati in totale N nodi:

$$N = \sum_{i=1}^{d} b^i \cdot (d + 1 - i)$$

I primi b nodi a profondità 1 vengono generati d volte, fino ai nodi al livello d generati una sola volta. Questo algoritmo è quindi circa l'11% meno efficiente rispetto alla ricerca in ampiezza. Ma non rappresenta un incremento considerevole rispetto alla ricerca in ampiezza, quindi è accettabile.

2.4 Problema della Ripetizione degli Stati: Graph-Search

Il problema della ripetizioni degli stati può provocare gravi complicazioni nel processo di ricerca. Questo problema sorge sopratutto quando sono possibili azioni bidirezionali ed in questo caso è possibile che gli alberi di ricerca siano infiniti. Si vuole quindi evitare quando è possibile di ripetere gli stessi stati in più nodi dell'albero di ricerca.

Questi stati ripetuti possono in certi casi rendere il problema irrisolvibile, è conveniente controllare se uno stato è replicato. Se un algoritmo arriva ad uno stesso stato attraverso due cammini differenti, allora ha individuato uno stato ripetuto e deve scartare uno di questi due cammini, per determinare quale scartare si sceglie generalmente l'ultimo cammino ottenuto. Si scarta anche se questo cammino è migliore del cammino precedente. Si utilizza questo approccio poiché negli algoritmi di ricerca euristica, sotto alcune condizioni, quando trova un percorso questo è ottimo, quindi non sorgono problemi nello scartare cammini che portano allo stesso stato.

In altre implementazioni dove non è garantito che il primo percorso trovato sia il migliore, bisogna controllare quale dei due cammini presenti il costo migliore. Per evitare la ripetizione bisogna contenere gli stati già visitati in memoria, tramite un'altra struttura dati chiamata insieme esplorato o lista chiusa, contenente ogni nodo espanso.

Si modifica l'algoritmo Tree Search nell'aggiunta alla frontiera per verificare la ripetizione degli stati:

```
function GRAPH-SEARCH(problem) returns a solution or failure

close  <- empty set
fringe <- MAKE-QUEUE(MAKE-NODE(INITIAL-STATE(problem)))
loop do
    if EMPTY(fringe) then return failure
```

```
    node <- REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE(node)) then return SOLUTION(node)
    if STATE(node) not in close
        then ADD(close, node)
        child_list <- EXPAND(node, OPERATOR(problem))
        for child_node in child_list
            if STATE(child_node) not in close then
                fringe <- QUEUING-FN(fringe, child_node)
end
```

Questo approccio si chiama Graph Search, dove prima di aggiungere un nodo alla frontiera, si controlla se il suo stato è già stato avvistato. Si suppone che il primo cammino che raggiunge nuo stato s è il più conveniente. Questo algoritmo realizza un albero direttamente sul grafo dello spazio degli stati, poiché è presente al massimo una singola copia di ogni stato. La frontiera separa nel grafo dello spazio degli stati in due regioni, una esplorata, ed una da esplorare. In questo modo ogni cammino dallo stato iniziale ad uno stato inesplorato deve passare attraverso uno stato sulla frontiera.

L'algoritmo scarta sempre il cammino appena trovato, se lo stato raggiunto è ripetuto, quindi potrebbe scartare un cammino corrispondente ad una soluzione migliore. Potrebbe quindi non essere un algoritmo ottimale.

Inoltre l'uso della lista chiusa significa che la ricerca in profondità e quella ad approfondimento iterativo non richiedono requisiti spaziali lineari.

2.5 Algoritmo di Ricerca Informata o Euristica: Best First Search

Quando si parla di ricerca euristica, l'algoritmo può sfruttare conoscenze specifiche sul problema in questione, aiutandolo nella scoperta della soluzione. Questa modifica si inserisce nella funzione di inserimento in coda QUEUING-FN. Questa conoscenza sul dominio del problema viene implementata tramite una funzione di valutazione f applicata ai nodi dell'albero di ricerca. Questa funzione stima quanto un nodo sia più o meno "promettente", ovvero stima della desiderabilità di espandere il nodo associato. Generalmente la funzione f è una funzione di stima del costo della soluzione, per si considera il nodo n appartenente alla frontiera con $f(n)$ minore.

L'algoritmo "Best First Search" ordina i nodi inseriti nella frontiera dal migliore al peggiore secondo una data funzione di valutazione f. Il nodo scelto da espandere è quello valutazione migliore, in genere con $f(n)$ minore, per cui a differenza di funzioni di valutazione f si hanno diverse versioni di questo algoritmo di ricerca.

La ricerca guidata dal costo (2.3.2) si può considerare un caso particolare della ricerca best first dove la funzione di valutazione $f(n)$ coincide con la funzione di costo del cammino $g(n)$ dallo stato iniziale al nodo n. Queste rappresenta un'informazione di euristica nulla.

```
function BEST-FIRST-SEARCH(problem, EVAL-FN) returns a solution or failure

QUEUING-FN <- function that orders nodes by EVAL-FN
fringe <- MAKE-QUEUE(MAKE-NODE(INITIAL-STATE(problem)))
```

```
loop do
    if EMPTY(fringe) then return failure
    node <- REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE(node)) then return SOLUTION(node)
    fringe <- QUEUING-FN(fringe, EXPAND(node, OPERATOR(problem)))
end
```

Dove `EVAL-FN` rappresenta la funzione di valutazione.
Per favorire la comprensione si utilizza una serie di notazioni:

- $s, s_0, s_1, \cdots, s_i$: stati del problema;
- s_0: stato iniziale;
- $k^*(s_i, s_j)$: costo del cammino minimo da s_i a s_j, se esiste;
- $g^*(s_i) = k^*(s_0, s_i)$: costo del cammino minimo dallo stato iniziale a s_i;
- $h^*(s_i)$: costo effettivo, non una stima, di un cammino da uno stato s_i ad uno stato obiettivo;
- $f^*(s_i) = g^*(s_i) + h^*(s_i)$: il costo minimo di una soluzione vincolata a passare per s_i.

Si introducono ulteriori notazioni per l'albero di ricerca:

- $n, n_1, n_2, \cdots, n_i$: nodi dell'albero di ricerca;
- $g*(n)$: costo del cammino dallo stato iniziale allo stato associato al nodo n;
- $h^*(n)$: costo effettivo di un cammino dallo stato associato al nodo n ad uno stato obiettivo;
- $g(n)$: costo del cammino dallo stato iniziale allo stato di n, $g(n) \leq g^*(n)$;
- $h(n)$: stima di $h^*(n)$.

Le funzioni senza asterisco all'apice si riferiscono a funzioni di stima, altrimenti sono funzioni di costi effettivi. La funzione h si dice funzione euristica.

2.5.1 Algoritmo di Ricerca Greedy

Questo algoritmo "goloso" utilizza la funzione h come funzione di valutazione, si espande il primo nodo che si ritiene sia vicino all'obiettivo. Se n corrisponde allo stato obiettivo, deve essere $h(n) = 0$. Minimizza il costo stimato per raggiungere l'obiettivo.

Una possibile funzione euristica è la distanza a linea d'aria " Straight Line Distance" SLD: $h_{\text{SLD}}(n)$, per problemi di viaggio. Questo algoritmo non espande nodi inutilmente, ma la soluzione trovata dall'algoritmo potrebbe non essere la soluzione ottima del problema. In generale la ricerca golosa non è ottimale. Inoltre non è neanche completa, poiché se una soluzione richiedesse allontanarsi dall'obiettivo e quindi andare verso stati con una valutazione peggiore, non sarebbe mai scoperta da questo algoritmo.

Nel caso peggiore è anche esponenziale spazialmente e temporalmente con una complessità asintotica di $O(b^m)$. Nonostante questi risultati con una buona euristica è possibile ottenere buoni risultati anche con un algoritmo greedy. Uno dei difetti è che la scelta del nuovo stato è effettuata interamente dalla stima della distanza, senza considerare il percorso parziale.

2.5.2 Algoritmo A*

L'algoritmo A* risolve i problema dell'algoritmo greedy, considerando anche il percorso parziale nella sua funzione di valutazione: $f(n) = g(n) + h(n)$, fornisce quindi una stima del costo del cammino dallo stato iniziale ad uno stato obiettivo, vincolato a passare per il nodo n.

Con questa semplice aggiunta è possibile migliorare notevolmente l'algoritmo precedente, inoltre sotto certe condizioni è possibile recuperare l'ottimalità e la completezza. Date queste due ipotesi sul grafo di partenza:

- Ogni nodo del grafo abbia un numero finito di successori;
- Tutti i costi abbiano costi maggiori di una quantità positiva δ.

Se h è un'euristica ammissibile, ovvero se per ogni nodo n del grafo vale la condizione $h(n) \leq h^*(n)$, allora l'algoritmo A* è ottimale e completo:

$$\forall n \text{ t.c. } h(n) \leq h^*(n) \implies \text{A*: completo ed ottimale} \tag{2.5.1}$$

In generale per un dato problema è possibile identificare diverse funzioni di euristica che soddisfano la condizione di limite inferiore. Nei problemi della ricerca di itinerari, e non solo, un'altra possibile alternativa è la funzione di Manhattan che considera ogni mossa, o spostamento, sicuramente ammissibile. La ricerca guidata dal costo è un approccio molto particolare della ricerca A* $\lceil f(n) = g(n) \rceil$, dove la funzione di euristica h è nulla per ogni nodo, è quindi uno stimatore super ottimistico.

Date due versioni dell'algoritmo A* con due euristiche diverse $h_1 < h_2$, per tutti i nodi non obiettivo, si dice che l'algoritmo A^*_2 è più informato dell'algoritmo A^*_1. Vale allora il teorema secondo cui al termine delle loro ricerche su un qualsiasi grafo con un percorso dal nodo iniziale n_0 al nodo obiettivo, allora ogni nodo espanso da A^*_2 sarà anche espanso da A^*_1. Quindi il primo algoritmo espande almeno tanti nodi quanto il secondo algoritmo, quindi l'algoritmo più informato A^*_2 è più efficiente di A^*_1.

In generale è più conveniente scegliere un'euristica per cui l'algoritmo è più informato. Per determinare una funzione euristica, un buon approccio consiste nel determinare la funzione di costo effettivo per un problema simile all'originale con minori restrizioni sugli operatori. Infatti il costo effettivo di una soluzione in questi "Relaxed Problems" è una buona euristica per il problema originale. Ma il calcolo della funzione euristica potrebbe avere un calcolo computazionale elevato, quindi bisogna scegliere l'euristica considerando anche la loro complessità.

L'algoritmo A* può essere eseguito sia in modalità Tree-Search che Graph-Search, ma in queste due modalità potrebbe trovare due soluzioni differenti allo stesso problema. Nella modalità Graph-Search infatti l'algoritmo potrebbe scartare nodi già visitati, ma che appartengono ad una soluzione migliore. Questa soluzione migliore può essere individuata in modalità Tree-Search poiché non

vengono scartati nodi. Per risolvere questo problema si introduce quindi la condizione di consistenza, e si dice che una certa funzione euristica h obbedisce a questa condizione se per tutte le coppie di nodi n_j, successore di n_i nel grafo di ricerca si ha:

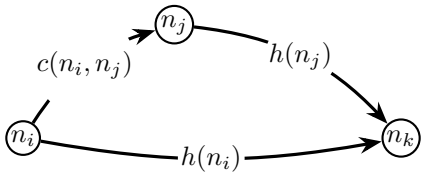
$$h(n_i) \leq c(n_i, n_j) + h(n_j) \tag{2.5.2}$$

Dove $c(n_i, n_j)$ è il costo dell'arco che congiunge i due nodi. Analogamente:

$$h(n_j) \geq h(n_i) - c(n_i, n_j)$$

Ovvero su un qualsiasi percorso, la stima del costo ottimo non può diminuire più del costo di un arco lungo quel percorso. Si può considerare anche come un tipo di diseguaglianza triangolare:

$$h(n_i) \leq c(n_i, n_j) + h_(n_j)$$



La condizione di consistenza impone che i valori della funzione di valutazione f dei nodi nell'albero di ricerca siano strettamente non-decrescenti all'allontanarsi dal nodo di partenza. Dati due nodi n_j, successore di n_i, se è soddisfatta si ha:

$$f(n_j) \geq f(n_i)$$

Dalla condizione di consistenza si aggiunge ad entrambi i membri $g(n_j) = g(n_i) + c(n_i, n_j)$:

$$\begin{gathered} h(n_j) \geq h(n_i) - c(n_i, n_j) \\ h(n_j) + g(n_j) \geq h(n_i) - c(n_i, n_j) + g(n_j) = h(n_i) - \cancel{c(n_i, n_j)} + g(n_i) + \cancel{c(n_i, n_j)} \\ f(n_j) \geq f(n_i) \end{gathered} \tag{2.5.3}$$

Per cui spesso la condizione di consistenza sulla funzione euristica h viene chiamata condizione monotona su f.

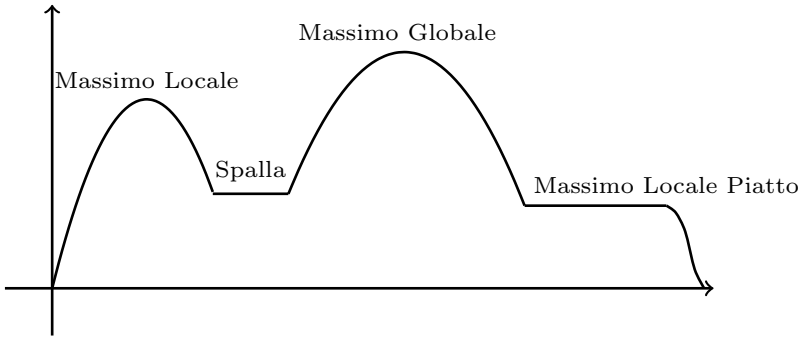
É dimostrabile che se una funzione euristica h è consistente allora è anche ammissibile.

Se la condizione di consistenza è soddisfatta su h, allora quando l'algoritmo A* espande un nodo n ha già trovato il percorso ottimo per n. In questo modo la ricerca su grafo non è differente dalla ricerca su un albero per quanto riguarda l'ottimalità della soluzione.

2.6 Algoritmo di Ricerca Locale: Hill-Climbing

Nei problemi precedenti quando l'algoritmo risolutivo raggiungeva uno stato obiettivo, il cammino verso quello stato rappresenta una soluzione del problema. Tuttavia in alcuni problemi lo stato obiettivo contiene tutte le informazioni rilevanti per la soluzione, dove il cammino è irrilevante. Come esempio si consideri il problema dell otto regine, è indifferente il cammino attraverso gli stadi intermedi, solamente lo stato finale, la disposizione delle regine nello stato finale.

Algoritmi di ricerca locale si utilizzano per risolvere questo tipi di problemi. In questi problemi è sempre presente uno spazio degli stati ed uno spazio degli stati aventi ciascuno una sua valutazione. Si può immaginare questi stadi su una superficie del territorio, uno spazio dove l'altezza di questo stato rappresenta la sua valutazione. L'algoritmo quindi itera su ognuno di questi stati per cercare quello di altezza maggiore, o minore, identificando quindi la soluzione al problema, indipendentemente dal cammino preso per raggiungerla. Questi punti di massimo rappresentano dei picchi, i cui punti adiacenti sono strettamente minori dello stato di massimo. Quindi l'algoritmo che parte da uno stato iniziale deve cercare un massimo globale in questo spazio, determinando quale sia tra i vari massimi locali ed i massimi locali piatto, e le "spalle" massimi locali "piatti", prima di un massimo globale:



Questi algoritmi chiamati anche di miglioramento iterativo, si muovono sulla superficie cercando questi picchi, senza tenere traccia del cammino effettuato, tenendo solamente traccia dello stato attuale e dei suoi vicini o successori, gli stati immediatamente adiacenti. Bisogna formulare il problema in modo che l'algoritmo non rimanga bloccato tra due massimi locali.

Questo algoritmo segue sempre le colline più ripide, si muove sempre verso l'alto nella direzione dei valori crescenti, e termina quando raggiunge uno stato per il quale si ha un picco che non ha vicino stati di valore maggiore. Tuttavia questo algoritmo può rimanere intrappolato su massimi locali.

Non viene memorizzato lo stato corrente, solamente il valore attraverso nodi che contengono lo stato ed il suo valore.

Esistono diversi tipi di algoritmi di questo genere per evitare di rimanere bloccati su picchi locali, utilizzando diverse tecniche.

- Steepest Ascent Hill-Climbing;

- First-Choice Hill-Climbing;
- Random-Restart Hill-Climbing (Iterated Hill-Climbing);
- Stochastic Hill-Climbing.

2.6.1 Algoritmo Steepest Ascent Hill-Climbing

Si considera il seguente pseudocodice dell'algoritmo Hill-Climbing di tipo "Steepest Ascent":

```
function HILL-CLIMBING(problem) returns a state local max

current <- MAKE-NODE(INITIAL-STATE(problem))
loop do
    next <- MAX(EXPAND(current))
    if VALUE(next) < VALUE(current) then return STATE(current)
    current <- next
end
```

Dallo stato corrente si ricava il suo valore, in seguito comincia un ciclo che prende in considerazione tutti i successori e si sceglie come `next` il nodo di valore più alto. Se tutti i nodi adiacenti hanno un valore minore di `next` allora questo rappresenta la soluzione dell'algoritmo e l'algoritmo termina, tuttavia questo stato potrebbe corrispondere ad un massimo locale, invece se esiste uno stato adiacente di valore maggiore, questo diventa `next` e si passa alla nuova iterazione.

2.6.2 Algoritmo Random-Restart-Hill Climbing

L'algoritmo contiene una componente di ripartenza casuale, questo infatti conduce una serie di ricerche di Hill-Climbing partendo da stati generati casualmente. Questo algoritmo da un punto di vista teorico è completo, poiché con una serie infinita di ripartenze, sicuramente l'algoritmo visita tutti gli stati del sistema, trovando sicuramente la soluzione ottima del problema.

```
function RANDOM-RESTART-HILL-CLIMBING(problem) returns a state solution

t <- 0
best <- MAKE-NODE(NULL)

repeat
    local <- false
    current <- RANDOM(problem)
    repeat
        next <- MAX(EXPAND(current))
        if VALUE(next) > VALUE(current)
            then current <- next
        else local <- true
```

```
    until local
    t <- t + 1
    if VALUE(current) > VALUE(best)
        then best <- current
until t = MAX
return STATE(best)
end
```

Il ciclo interno ad ogni iterazione genera un ottimo locale, e prova ad evitare ottimi locali effettuando una nuova ricerca da un nuovo stato scelto casualmente. L'algoritmo è completo con probabilità tendente ad uno, poiché è possibile generi come stato iniziale uno stato obiettivo.

2.6.3 Algoritmo Stochastic Hill-Climbing

L'algoritmo Stochastic Hill-Climbing si ottiene modificando la procedura normale dell'algoritmo. Invece di valutare tutti i vicini dallo stato corrente, l'algoritmo sceglie casualmente uno solo dei suoi successori da valutare per determinare se si tratta il successore, ed in caso diventa il nuovo stato corrente `next`, questo viene accetta con una probabilità che dipende dalla differenza della valutazione tra i due punti: $\Delta E =$`VALUE(current)-VALUE(next)`.

```
function STOCHASTIC-HILL-CLIMBING(problem) returns a state solution

t <- 0
current <- RANDOM(problem)
best <- MAKE-NODE()

repeat
    next <- RANDOM(EXPAND(current))
    if p = 1/(1 + e^(ΔE/T))
        then current <- next
        if VALUE(current) > VALUE(best)
            then best <- current
    t <- t + 1
until t = MAX
return STATE(best)
end
```

Il nuovo stato viene scelto con una probabilità p, calcolata come:

$$p = \frac{1}{1 + e^{\Delta E/T}} \tag{2.6.1}$$

In seguito dopo una serie di iterazioni l'algoritmo restituisce uno stato ottimo. L'algoritmo ha quindi un solo ciclo, e può scegliere un nuovo punto con una probabilità p, quindi anche di valore

minore. Questa probabilità dipende da un parametro T costante durante l'esecuzione dell'algoritmo. Se vale 1, la probabilità di accettazione è sostanzialmente pari al 100%.

All'aumentare del valore i T la probabilità di accettazione tende al 50%, diventa quindi sempre meno importante la differenza della valutazione tra i due punti, effettivamente comporta una ricerca casuale, mentre al diminuire di T, la procedura rappresenta un semplice algoritmo Hill-Climbing.

In caso di stati di valore uguale, la probabilità è del 50%, se il valore dello stato `next` è minore, la probabilità diminuisce, mentre se il valore di `next` è maggiore dello stato corrente, la probabilità aumenta.

Bisogna trovare una "link function" tra l'intervallo $\Delta E/T$ e la probabilità p.

La caratteristica di poter scegliere come passo uno stato peggiore questo algoritmo potrebbe evitare massimi locali.

2.6.4 Algoritmo di Simulated Annealing

L'algoritmo di Simulated Annealing, introdotto nel 1983 da S. Kirkpatrick, C.D. Gelatt, Jr. e M.P. Vecchi nella rivista Science, è un algoritmo che migliora considerevolmente l'approccio dell'algoritmo precedente. Ha causato una vera e propria rivoluzione in termini di ottimizzazione, ebbe un enorme successo in ogni settore dell'informatica come l'algoritmo migliore per problemi di ricerca locale, anche solo nella sua versione base. Questo algoritmo è talmente importante che ogni anno vengono riuniti congressi annuali internazionali per discutere possibili ottimizzazioni.

Questo algoritmo prende il nome dall'analogia con il processo di metallurgia per temprare un materiale, questo processo infatti raggiungere uno stato di struttura cristallina ad energia minima. La differenza principale con l'algoritmo stocastico, è la possibilità di variare il valore di T diminuisce gradualmente durante l'esecuzione dell'algoritmo. Il valore di T parte da un valore elevato, per poi diminuire nel tempo, come se fosse la temperatura durante un processo di temperatura, per cui l'algoritmo si comporta in modo molto simile ad un normale hill-climber. Inoltre sceglie sempre uno stato se è migliore del punto corrente.

```
function SIMULATED-ANNEALING(problem) returns a state solution

t <- 0
current <- RANDOM(problem)
best <- MAKE-NODE(NULL)

repeat
    repeat
        next <- RANDOM(EXPAND(current))
        if VALUE(next) > VALUE(current)
            then current <- next
            if VALUE(current) > VALUE(best)
                then best <- current
            else if RANDOM[0,1) < e^{-ΔE/T}
                then current <- next
    until termination-condition
```

```
        T = g(T, t)
        t <- t + 1
    until halting-condition
    return STATE(best)
    end
```

La probabilità di accettazione è leggermente diversa rispetto all'algoritmo precedente, poiché nel simulated annealing si considera solamente un semiasse dell'ascissa, dato che in caso `next` sia migliore non viene calcolata la probabilità di accettazione. Si accetta sempre uno stato di valore migliore, mentre l'accettazione di uno stato peggiore dipende da una probabilità. Quindi il mapping della link function non viene realizzata sull'intero intervallo $(-\infty, +\infty)$, ma solo su metà dell'ascissa, su $[0, +\infty)$, per cui è sufficiente utilizzare la funzione $e^{-\Delta E/T}$ avente come dominio il semiasse $[0, +\infty)$ e come codominio $(0, 1]$.

Questo ciclo interno viene effettuato un certo numero di volte fino ad una condizione di terminazione, che verrà trattata nelle implementazioni future. Finito questo ciclo si abbassa leggermente la temperatura tramite una funzione g e si incrementa il contatore delle iterazioni `t`. Anche la condizione di terminazione dell'algoritmo dipende dallo specifico problema e verranno trattate in futuro.

Molte implementazioni dell'algoritmo seguono la stessa sequenza di passi. Si assegna la variabile `T` alla temperatura massima, e si sceglie uno stato corrente casuale al primo passo. Si determina un successore assegnandolo direttamente se è migliore oppure tramite la funzione di probabilità e si ripete per un certo numero di cicli, e come passo finale si diminuisce la temperatura e si ripete dal secondo passo se la temperatura non ha raggiunto la temperatura minima. Quando la temperatura ha raggiunto la temperatura minima, si può scegliere se terminare l'algoritmo o ripeterlo un certo numero di volte, ripartendo dal primo passo.

3 Linguaggio Python

Python è un linguaggio di programmazione vastamente utilizzato nell'area dell'intelligenza artificiale e nel machine learning. Recentemente è diventato il linguaggio di programmazione più diffuso al mondo. Python è un linguaggio general-purpose, ideato da Guido van Rossum nel 1989, a più alto livello del C, poiché gestisce automaticamente le più fondamentali operazioni. Per cui è molto semplice e spesso utilizzato a fine didattico tra i primi linguaggi di programmazione insegnati.

La versione di Python utilizzata nel corso è la versione 3, nell'ambiente Anaconda. Può essere avviato tramite un interprete.

3.1 Ambienti di Sviluppo

I principali ambienti di programmazione di Python sono Anaconda, Eclipse e Google Colab. Tutte e tre permettono di programmare in modo semplificato. Esistono due approcci alla traduzione ed esecuzione dei programmi per i linguaggi di alto livello. Possono essere trasformati in un programma in linguaggio macchina ed eseguiti, compilazione, oppure ciascuna istruzione di alto livello può essere trasformata in almeno un' istruzione di linguaggio macchina ed eseguita direttamente, interpretazione. Storicamente la modalità interpretata viene associata alla programmazione su linea di comando, e solamente su rari casi viene utilizzata, anche se con Python ha cominciato ad essere riutilizzata.

Quindi non è presente una fase intermedia di compilazione o un programma compilato da eseguire, ma il programma viene eseguito direttamente istruzione per istruzione tramite un interprete. I linguaggi compilati sono specifici per ogni piattaforma, ma estremamente ottimizzati per quella esatta piattaforma. Mentre i linguaggi interpretati prevedono la distribuzione diretta del file sorgente, uguale per tutte le piattaforme, ma deve essere disponibile il programma interprete e sono generalmente più lenti poiché l'esecuzione attende ogni nuova istruzione da eseguire. In assenza di vincoli sul tempo dell'esecuzione il ritardo introdotto dall'interpretazione è accettabile.

Linguaggi come C e C++ sono linguaggi compilati, mentre Python è un linguaggio interpretato. Java ha un approccio ibrido generando il bytecode, una versione di linguaggio macchina eseguibile dalla JVM, elaboratore virtuale simile ad un hardware tradizionale, con una conversione quasi uno ad uno delle istruzioni.

Python è un linguaggio interpretata ad alto livello simbolico. Un programma Python può essere eseguito in modalità interattiva, dove ogni istruzione viene eseguita singolarmente, ed il suo output viene visualizzato sulla console in sequenza. Questo programma può essere interrotto, si possono rieseguire certe istruzioni, etc. Si può quindi modificare il flusso e le istruzioni sulla base dell'output ottenuto. Questo tipo di codice viene utilizzato per data analytics e exploration. Altrimenti all'interprete può essere passato l'intero programma come un file di estensione `.py`. L'output viene visualizzato solamente se si invoca la funzione di stampa.

Per utilizzare la modalità interattiva serve un ambiente di interazione, l'interprete più diffuso è Cpython.

Può essere scaricato su tutte le piattaforme, aprendo una console si può entrare nella modalità interattiva con il comando `python3`. Per uscire da questa modalità è sufficiente invocare la funzione `quit()`.

Anaconda è una distribuzione di Python che semplifica la gestione ed il deployment delle librerie. Permette di creare ambiente in cui installare versioni specifiche delle librerie compatibili con certi software. Il gestore di queste librerie o package si chiama conda e si usa principalmente da linea di comando e gestisce queste librerie e le mantiene aggiornate. Può essere installato insieme a Cpython.

I principali comandi in Anaconda sono:

```
# crea un nuovo environment di versione specificata:
conda create -n nomeEnv python=versionePython
# visualizza la lista di environment creati:
conda info -envs
# attiva l'ambiente di sviluppo
source activate nomeEnv
# installa il package specificato all'interno dell'ambiente corrente
conda install nomePackage=versionePackage
```

Se non viene specificata la versione, sia per Python che per i pacchetti da installare, conda utilizzerà l'ultima versione come default.

Le principali librerie utilizzate in questo corso sono pandas, matplotlib, NumPy, SciPy, IPython e scikit-learn. Dentro Anaconda c'è un'interfaccia browser per avere un ambiente di programmazione per programmare chiamato Jupiter, dove è possibile attivare la linea di comando per programmare in modalità interattiva. Si può attivare con il comando `jupiter notebook`, che mostra a schermo l'indirizzo a cui collegarsi per accedere, generalmente `127.0.0.1`. Si analizzerà in seguito insieme all'ambiente Google Colab, anch'esso su un'interfaccia browser.

Sui notebook è possibile inserire del testo, ed accetta macro e comandi in formato Markdown, e funzioni scritte in LaTeX.

Eclipse è un ambiente di sviluppo storico, multi-linguaggio, installando dal marketplace il plugin pydev per poter programmare in Python.

Google Colab è un ambiente di sviluppo fornito gratuitamente da Google, utilizzando risorse cloud, su di un singolo notebook, direttamente collegato all'account google. Ha un'interfaccia a blocchi interattiva, da data scientist, e permette di scegliere il runtime e la GPU da utilizzare per eseguire il programma. Il codice viene salvato su un notebook, quindi su un file di estensione `.ipynb`, ed è possibile passare al Python tradizionale e viceversa. Su dispositivi mobile è possibile programmare con un'interfaccia molto simile ad un computer.

Si divide il programma in celle per dividere il codice e permettere di elaborare singoli blocchi, senza dover ricalcolare tutti i blocchi già eseguiti. Questo inoltre rende il codice più chiaro e comprensibile.

JupyterLab è un estensione di Jupyter che aggiunge funzionalità per programmare a progetto per visualizzare e gestire diverse tipologie di file.

Si possono utilizzare anche ambienti virtuali come Docker per eseguire il codice su container isolati. Inoltre esistono ambienti dedicati all'ambito scientifico con un'analisi più dettagliata dei dati prodotti come spyder, "The Scientific Python Development Environment".

3.2 Operatori

Alla creazione di una variabile non è necessario definirne il tipo, il nome identificativo è arbitrario e può contenere numeri, ma non cominciare con un numero, viene consigliato di utilizzare un carattere minuscolo come primo carattere del nome. Esistono 33 parole chiave, non utilizzabili come nomi di variabili. Si può assegnare un valore ad una variabile tramite l'operatore `=`, senza specificarne il tipo.

Esistono una serie di operatori aritmetici come `+`, `-`, `*`, \, `**`, per l'elevamento a potenza, `%` per il modulo. Una differenza tra Python 2 consiste nella gestione della divisione, infatti in Python 2 viene considerata solo la parte intera dell'operazione. Per ottenere lo stesso risultato esiste l'operatore \\, chiamato "floor division".

Gli operatori seguono un ordine di precedenza naturale, come la sintassi moderna matematica:

1. Parentesi;
2. Elevamento a potenza;
3. Moltiplicazione e divisione;
4. Addizione e sottrazione;
5. Operatori con lo stesso ordine valutati da sinistra verso destra.

In Python sono presenti tutti gli operatori booleani del C come `==` ed operatori di confronto come `<, <=, >, >=`, in aggiunta sono presenti altri operatori `is` ed `is not`. Inoltre sono presenti due versione degli operatori logici `&&` e `and`, —— e `or` e `!=` e `not`. Come in C un qualsiasi valore diverso da zero corrisponde al booleano `True`, inoltre associa numericamente questo valore ad 1, mentre `False` a 0. É possibile convertire un tipo di dato booleano ad un tipo di dato numerico come un intero `int` o un numero reale `float`.

Si possono inserire dati dall'utente tramite la funzione `input()` e la funzione `raw_input()` per lo stesso comportamento di Python 2, e si può convertire in un tipo specifico con `tipo(var)`. I commenti vengono realizzati tramite il carattere #.

3.3 Istruzioni Condizionali

In Python per identificare funzioni o istruzioni condizionali non si usano parentesi, ma si indenta di quattro posizioni. Dopo la condizione dell' istruzione condizionale vanno inserite dei due punti ::

```
if condizione:
    # corpo dell'if
else:
    # corpo dell'else
```

Si possono gestire le eccezioni con il costrutto `try` ed `except`:

```
try:
    # corpo del try
except:
    # corpo dell'except
```

3.4 Funzioni Built-In, Moduli e Definizione di Funzioni

In Python sono integrate tantissime funzioni utili, per svolgere attività comuni, utilizzabili senza doverle definire, queste sono funzioni "built-in". Per invocare funzioni presenti in un certo modulo e non built-in si utilizzata la notazione puntata `nomeModulo.nomeFunzione()`. Alcune tra le funzioni built-in più utili sono `max()` e `min()` che restituiscono il carattere più grande e più piccolo in una stringa; la funzione `len()` che restituisce la lunghezza della stringa. Per convertire variabili in certi tipi è già stata mostrata la funzione `tipo()`, dove al posto di `tipo` si inerisce il tipo specifico, si usa `str` per convertire in una stringa.

Per importare moduli contenenti altre funzioni si utilizza `importa` seguito dal nome del modulo da scaricare, che crea un object module con quel nome, si può rinominare seguendo questa istruzione con `as` seguito dall'alias del modulo. Si utilizza un alias per semplificare la notazione puntata.

Dati gli algoritmi analizzati precedentemente, si nota la necessita di introdurre generatori di numero casuali. La maggior parte di generatori casuali, sono deterministici, ovvero dato lo stesso input, generano la stessa sequenza di numeri casuali. Si utilizzano quindi numeri pseudo-casuali, generati da un calcolo deterministico, ma non è quasi possibile distinguerli da numeri generati casualmente. In Python esiste il modulo `random` contenente funzioni pertinenti alla generazione di numeri casuali. La funzione `random()` genera un numero casuale tra 0.0, compreso e 1.0, non compreso. Un altra funzione `randint()` accetta due parametri, estremi dell'intervallo, inclusi, per generare un numero intero tra loro compreso. Con la funzione `choice()` si può scegliere un elemento casualmente da una sequenza passata come argomento.

Per definire nuove funzioni si utilizza la parola chiave `def`, specificando il nome, tra parentesi tonde gli argomenti ed i due punti, indentando di quattro posizioni per scrivere il corpo della funzione:

```
def nomeFunzione(listaArgomenti):
    # corpo della funzione
    # resto del codice
```

Dopo aver passato degli argomenti ad una funzione, questi vengono assegnati a delle variabili locali. Si può utilizzare anche una variabile come argomento. Inoltre tutte le aggiunte possibili alle funzioni built-in, si possono effettuare sulle funzioni definite dall'utente.

Si dividono le funzioni in due tipi "fruitful function", funzioni produttive, e "void function", funzioni vuote, le prime restituiscono un valore, le seconde non restituiscono valore. Le prime quindi vengono usate per assegnare o inizializzare variabili. Se si tenta di assegnare il risultato di una void function ad una variabile, viene ottenuto un valore chiamato `None`. Questo valore ha un suo proprio tipo. Per definire una funzione produttiva, nel corpo si inserisce la parola chiave `return` seguita dai parametri da restituire come risultato della funzione.

3.5 Cicli

Si possono realizzare cicli tramite il costrutto `while` o `for`, seguito da una condizione booleana e dai due punti `:`:

```
while condizione:
    # corpo del ciclo
```

Si può interrompere il ciclo con `break`, e si può saltare l'iterazione corrente con `continue`. Quando bisogna iterare su una collezione, un insieme di elementi, si può realizzare un ciclo "for-each":

```
for elemento in collezione:
    # corpo del ciclo
```

Se il valore dell'elemento non viene utilizzato all'interno del ciclo, ma solo per effettuare un numero definito di cicli, è convenzione utilizzare il carattere `_` per distinguerlo:

```
for _ in collezione:
    # corpo del ciclo
```

3.6 Stringhe

Le stringhe sono sequenze di caratteri, indicizzati come fosse un array:

```
stringa[i] # (i+1)-esimo carattere
```

La funzione già discussa `len()` restituisce il numero di caratteri di una stringa, anche se può essere utilizzata per altri tipi di dati come dizionari. Poiché è strutturata come un array è possibile scandire ogni carattere della stringa individualmente con un ciclo for-each.

Talvolta è comodo accedere ad una sottostringa, o "slice", della stringa di partenza. La selezione di una sottostringa è simile alla selezione di un carattere, utilizzando due indici divisi da due punti per indicare l'inizio e la fine della sottostringa, il primo estremo è compreso, mentre il secondo no:

```
stringa[i:j] # slice contenente i caratteri da i a j-1
```

A volte si ha la necessità si realizzare una sottostringa che parte dall'inizio o la fine della stringa originaria, per effettuarlo si può omettere l'estremo corrispondente:

```
stringa[:j] # slice dall'inizio della stringa
stringa[i:] # slice fino alla fine della stringa
```

I valori di una stringa sono immutabili una volta definiti, per cui non è possibile modificarne il valore accedendo tramite indice, verrà sollevato un messaggio di errore. Si può modificare una stringa realizzando una nuova stringa, come variante, tramite l'operatore di concatenazione `+`:

```
stringa = stringa_1 + stringa_2
```

L'operatore `*` applicato su una lista, la replica un certo numero di volte specificato.

L'operatore `in` è estremamente importante, permette di individuare se una stringa è sottostringa di un'altra, restituisce un valore booleano vero o falso:

```
stringa_1 in stringa_2
```

Si possono inoltre utilizzare operatori di confronto tra stringhe all'uguaglianza con `==`, oppure con `<, >`, per confrontarle in ordine alfabetico. In Python le maiuscole vengono prima delle minuscole. Le stringhe sono degli oggetti che oltre alla sequenza di caratteri contengono oltre i dati anche i metodi disponibili per ogni istanza dell'oggetto stringa.

Con la funzione `type` si ha la possibilità di identificare il tipo dell'oggetto su cui viene operata e `dir` mostra i metodi disponibili. Per utilizzare un metodo si utilizza la notazione puntata con il nome dell'istanza dell'oggetto. Uno dei metodi sulle stringhe è `find()` che prende come argomento una sottostringa, ed un indice opzionale da cui cercare il carattere, e restituisce la prima posizione dell'occorrenza della sottostringa specificata.

Il metodo `strip()` rimuove lo spazio bianco prima e dopo la stringa. Il metodo `startswith()` restituisce un valore booleano se la stringa comincia con la sottostringa passata come argomento. Il metodo `capitalize()` consente di impostare a maiuscolo il primo carattere, per mettere tutti i caratteri in maiuscolo si utilizza il metodo `upper()`.

L'operatore "format" `%` permette di costruire stringhe formattando la stringa rispetto a dati contenuti in altre variabili. All'interno di una stringa chiamata "format string" si può inserire questo operatore seguito da una lettera per specificare il tipo di dato associato, chiamate "format sequences". Questa viene utilizzata come il primo argomento, mentre il secondo operando è la variabile da formattare, questo produce una stringa:

```
>>> x = 1
>>> 'il numero è %d' % x
'il numero è 1'
```

Se in una stringa compare una sequenza di format sequences si specifica l'operando come una sequenza, una tupla, tra parentesi tonde, separando gli elementi con virgole:

```
>>> x = 1
>>> 'il %s è %d' % ('numero', x)
'il numero è 1'
```

La tupla deve corrispondere in numero, ordine e tipo alle format sequences nella stringa. Si utilizza `%d` per numeri interi `%g` per floating point e `%s` per stringhe. Utilizzando numeri è possibile specificare il numero di cifre significative da utilizzare per rappresentare il numero, inserendo un numero subito dopo l'operatore format.

Esiste un nuovo operatore format, scrivendo tra parentesi graffe `:` seguito dal carattere corrispondente al tipo, dopo la stringa si invoca il metodo `format()` specificando nell'argomento il valore da inserire:

```
>>> 'il {:s} è {:d}' .format('numero', 1)
'il numero è 1'
```

Questa implementa tutte le caratteristiche già trattate per il format `%`.

3.7 Classi e oggetti

La fonte di ispirazione della programmazione orientata agli oggetti è dovuta a lavori realizzati dalla comunità sull'intelligenza artificiale, nell'ambito della rappresentazione della conoscenza quando Marvin Minsky propose il formalismo dei frames nel 1975. Con tutte le caratteristiche dei moderni approcci di programmazione orientata agli oggetti.

Questo paradigma di programmazione consiste nel mantenere i dati e le funzioni operabili su di essi vicine, raggruppate in entità chiamate oggetti. Una classe è una descrizione di un insieme di oggetti che hanno lo stesso comportamento, ed un oggetto è una singola istanza di questa classe, dove è definita anche un insieme di metodi da poter usare su questi oggetti.

L'insieme di tutti i metodi resi disponibili da una classe, con la descrizione del loro comportamento, si chiama interfaccia pubblica della classe.

Una classe in Python viene definita tramite la parola chiave `class` seguita dal nome della classe, utilizzando la notazione dove ogni parola del nome ha la prima lettera maiuscola, "Camel-case":

```
class NomeClasse:
    pass
```

`pass` è un placeholder ed indica dove sarà scritto codice, questa rappresenta una definizione di una classe minimale, definendo solamente il nome e nessun altro comportamento.

Si può istanziare una nuova classe dato il suo nome:

```
NomeClasse()
```

Questo istanzia un oggetto appartenente alla classe specificata, allocata in una nuova zona di memoria, una variabile assegnata a questa nuova istanza mantiene il riferimento a questa zona di memoria, quindi si ha:

```
>>> a = NomeClasse()
>>> b = NomeClasse()
>>> a == b
False
```

Poiché quest'operazione di confronto effettuata sulle variabili ha azione sugli indirizzi di memoria contenuti. Le proprietà che una classe deve avere può essere definite utilizzando un metodo particolare costruttore chiamato `.__init__()`. Questo inizializza una nuova istanza della classe ed assegna i valori al suo stato. Metodi con un doppio carattere `_` vengono chiamati "Dunder Methods", per "Data Underlined". Al costruttore vengono passati i parametri che si vuole, ma è sempre necessario passare la variabile chiamata `self` per definire nuovi attributi sull'oggetto. Poiché quando viene creata un'istanza di una classe, viene salvata automaticamente a questo parametro.

```
class NomeClasse:
    def __init__(self, attributo1, ..., attributoN):
        self.attributo1 = attributo1
        # ...
        self.attributoN = attributo N
```

Ognuna di queste assegnazioni in realtà crea anche un attributo chiamato con il nome utilizzato nella notazione puntata, riferita a `self`, e contenente il valore assegnatoli dal parametro passato come input al costruttore. `self` deve essere sempre il primo parametro.

Gli attributi creati in questo modo vengono chiamati "Istance Attributes", attributi d'istanza, che sono attributi unici per ogni istanza della classe, chiamai al momento di costruzione dell'istanza. Ma possono essere definiti attributi comuni per tutte le classi, chiamati "Class Attributes", all'interno del corpo della classe, senza inserirli all'intero del costruttore:

```
class NomeClasse:
    attributo = 'valore'
    def __init__(self, attributo1, ..., attributoN):
        self.attributo1 = attributo1
        # ...
        self.attributoN = attributo N
```

Ogni istanza avrà gli stessi class attributes, contenenti gli stessi valori. Per istanziare una classe, avendo un costruttore, bisogna passare i parametri richiesti al momento di creazione:

```
>>> istanza = NomeClasse('valore1', ..., 'valoreN')
```

Si passa un parametro in meno poiché `self` viene gestito internamente, al momento della creazione dell'oggetto. Si può accedere ad un certo attributo sempre con la notazione puntata, e si può modificare. Si può modificare sia un attributo di classe o istanza.

Gli istance methods sono funzioni definite all'interno di una classe, possono essere invocati da una qualsiasi istanza della classe.

É buona norma utilizzare un metodo come `description()` per fornire una stringa contenente informazioni utili per l'istanza della classe. Ma questo non è il modo migliore per ottenere. Si utilizza il metodo `.__str__()` per restituire queste informazioni in formato di stringa:

```
class NomeClasse:
    attributo = 'valore'

    def __init__(self, attributo1, ..., attributoN):
        self.attributo1 = attributo1
        # ...
        self.attributoN = attributo N

    def __str__():
        return f'{self.attributo1}, ..., {self.attributoN}'
```

Quando viene invocata una funzione di stampa su un'istanza di questa classe, viene stampata la stringa definita da questo metodo:

```
>>> istanza = NomeClasse('valore1', ..., 'valoreN')
>>> print(istanza)
valore1, ..., valoreN
```

3.8 Collezioni

3.8.1 Liste

Una lista è una sequenza di item di qualsiasi tipo, per cui a differenza di una stringa il tipo non è omogeneo. All'interno di una lista è possible inserire una lista, creando una lista annidata. Una lista si realizza specificando gli elementi tra parentesi quadre:

```
[item_1, item_2, [item_3, item_4]]
```

Una lista che non contiene elementi si chiama lista vuota mediante `[]`, oppure con il comando `list()`. Si può assegnare ad una variabile una lista allo stesso modo di un valore:

```
>>> lista = [item_1, item_2, [item_3, item_4]]
```

Le liste al contrario delle stringhe sono modificabili, con la stessa sintassi per le stringhe, specificando tra parentesi quadre l'indice corrispondente, ed è possibile aggiornare il valore contenuto in questo indice. Se ad un certo indice è presente una lista, questo elemento viene trattato come lista e quindi si possono utilizzare due coppie di parentesi quadre per indicizzare questi elementi annidati:

```
>>> lista[2][2]
item_4
```

Si può considerare una lista come un mapping tra indici ed elementi, come nelle stringhe possono essere variabili ed espressioni, se si prova a leggere o scrivere ad un indice non esistente si solleva un `IndexError`. Mentre se si utilizza un indice negativo, si conta all'indietro partendo dalla fine della stringa. L'operatore `in` ha un funzionamento analogo a quello per le stringhe.

Un modo per scandire la lista è con un ciclo for-each, come per le stringhe, oppure iterando manualmente su ogni elemento. Questo si può effettuare tramite la funzione `range()` che permette di iterare su un insieme di valori, fornito l'ultimo come argomento:

```
for i in range(len(lista)):
    # corpo del for
```

L'operatore di concatenazione vale anche per le liste, analogamente per l'operatore `*` che replica una lista un certo numero di volte. L'operatore slice può essere utilizzato anche sulle liste per ottenere una sottolista:

```
lista[i:j] # sottolista da i a j-1
lista[:j]  # sottolista da 0 a j-1
lista[i:]  # sottolista da i a len(lista)
lista[:]   # copia della lista
```

L'operatore slice può essere utilizzato per aggiornare una sequenza di elementi in una lista. La sottolista da aggiornare deve avere lo stesso numero dei valori della lista originaria.

Come per le stringhe, le liste sono oggetti e contengono metodi built-in. Il metodo `append()`. aggiunge un nuovo elemento alla fine di una lista, il metodo `extend()` prende come argomento una

lista e la concatena alla lista su cui si è operato. Se si passa una lista al metodo `append()` si genera una lista annidata. Si può ordinare gli elementi di una lista dal minore al maggiore con `sort()` in ordine alfabetico. Molti dei metodi su liste, come questi, sono metodi void.

Per estrarre un elemento dalla lista si utilizza il metodo `pop()` che estrae dalla lista l'ultimo elemento, oppure l'elemento di indice specificato, lo restituisce rimuovendolo dalla lista:

```
>>> lista.pop(1)
item_2
>>> print(lista)
[item_1, [item_3, item_4]]
```

L'operatore `del` rimuove un elemento dalla lista:

```
>>> del lista[1]
>>> print(lista)
[item_1, [item_3, item_4]]
```

Utilizzando lo slice si può rimuovere una sottostringa allo stesso modo:

```
>>> del lista[:1]
>>> print(lista)
[[item_3, item_4]]
```

Se non si conosce l'indice dell'elemento da cancellare si può utilizzare il metodo `remove()`, ma non restituisce alcun valore:

```
>>> lista.remove(item_2)
>>> print(lista)
[item_1, [item_3, item_4]]
```

Alcuni metodi come `sum()` possono essere invocati solamente se la lista contiene solo numeri, altri metodi possono essere invocati se la lista contiene dati confrontabili tra di loro. Si può convertire una stringa in una lista tramite la funzione `list()`, suddividendo la stringa in caratteri singoli, mentre si può suddividere in parole singole, separate da spazi con la funzione `split()`. Si può specificare un delimitatore in questa funzione `split()` da utilizzare al posto del delimitatore di default spazio. La funzione inversa della `split()` è il metodo `join()`, permette di realizzare una stringa, applicato su una stringa contenente il delimitatore da utilizzare per separare gli elementi. Si può concatenare senza spazi con la stringa vuota `[]`.

In Python si possono confrontare due oggetti con l'operatore `is`, se si creano due stringhe di contenuto uguale, Python realizza un unico oggetto stringa, mentre se vengono realizzare due liste, contenenti gli stessi elementi, sono due oggetti distinti. Le due liste sono equivalenti poiché hanno lo stesso valore, ma non sono identiche, poiché non sono lo stesso oggetto. Due oggetti identici sono anche equivalenti.

Si può assegnare ad una variabile ad un'altra in modo che entrambe si riferiscano allo stesso oggetto, e quindi anche se si tratta di liste il confronto `is` sarà verificato. Un oggetto che ha più di riferimenti ha un "alias", più di un nome, e si dice un oggetto "aliased". Se gli oggetti sono mutabili

questo può essere fonte di errore. Si consiglia di evitare di utilizzare alias quando si lavora con liste ed oggetti mutabili. Si distingue tra operazioni che creano nuove liste ed operazioni che modificano lista, come l'operatore di concatenazione `+` ed il metodo `append()`, quest'ultimo modifica la lista su cui è invocato analogamente di `extend()`, mentre l'operatore di concatenazione genera una nuova lista.

C'è uno speciale costrutto per accedere a tutti gi elementi di una lista effettuando la stessa operazione su di loro e memorizzare i nuovi elementi in un'altra lista:

```
>>> lista_1 = [1, 2, 3]
>>> lista_2 = [item * 2 for item in lista_1]
>>> print(lista_2)
[2, 4, 6]
```

Quando bisogna accedere contemporaneamente a più liste si può utilizzare la sintassi che utilizza la parola chiave `zip` per trattare la sequenza di liste passate contemporaneamente:

```
for item_1, item_2, ... in zip(lista_1, lista_2, ...):
    # corpo del ciclo, per ogni iterazione si ha
    # item_1 = lista_1[i], item_2 = lista_2[i], ...
```

In questo modo si può iterare su tutte le liste insieme, senza dover specificare per ognuna il loro indice. Questa funziona associa le liste e termina alla lista più corta.

3.8.2 Dizionari e Set

Un dizionario è simile ad una lista, ma più generale, permette di accedere agli elementi contenuti tramite una chiave definita a priori. Rappresenta un mapping tra due insiemi, uno di chiavi ed uno di valori, ogni chiave individua un valore, ogni item del dizionario è quindi una coppia tra una chiave ed un valore. Può essere costruito un dizionario vuoto tramite la funzione `dict()`, e se stampato produce `{}`.

```
>>> str2num = dict()
>>> print(str2num)
{}
>>>
```

Per aggiungere un item nel dizionario, si utilizzano parentesi quadre per specificare la chiave e si assegna il suo valore contenuto, come se fosse l'indice di una lista:

```
>>> str2num['uno'] = 1
>>> print(str2num)
{'uno' : 1}
```

Questo formato di output di un dizionario è un formato che può essere utilizzato per popolare un dizionario:

```
>>> str2num = {'uno' : 1, 'due' : 2, 'tre' : 3}
```

Ma l'ordine in cui sono stati inseriti gli elementi non viene mantenuto, questo non costituisce un problema, poiché gli elementi non vengono identificati dalla loro posizione, ma attraverso una chiave. Se si prova ad accedere ad un elemento la cui chiave non esiste viene sollevato un errore. Per accedere ad un valore nel dizionario si utilizza la chiave come fosse un incide:

```
>>> str2num['uno']
1
```

Si può utilizzare la funzione `len()` per determinare il numero di elementi in un dizionario e la parola chiave `in` per determinare se una data chiave è presente in un dizionario, non controlla se è presente un valore:

```
>>> len(str2num)
3
```

Per visualizzare i valori si può utilizzare il metodo `values()` che restituisce elementi su cui si può costruire una lista, su cui è possibile verificare la presenza di certi valori tramite `in`. Questo operatore utilizza diversi algoritmi per le liste e per i dizionari, utilizza una ricerca lineare per le liste ed una tabella di hash per i dizionari.

Una tupla è una sequenza di valori, simili ad una lista, di qualsiasi tipo, indicizzati da interi. Le tuple a differenza delle liste sono immutabili, ed è possibile confrontarle tra di loro, quindi è possibile siano utilizzate per realizzare chiavi di dizionari. Si definisce una tupla come una sequenza di valori divisi da virgole, ma convenzionalmente si tende ad includere questi valori in parentesi graffe per semplificare la leggibilità:

```
>>> t={1, 2, 3}
```

Per creare una tupla con un unico valore, bisogna comunque inserire la prima virgola, altrimenti si creerebbe o una stringa o un dato di tipo uguale al tipo del primo elemento.

Si può creare una tupla vuota con la funzione `tuple()`, invece è possible inserire un argomento in questa funzione per generare una tupla dagli elementi della sequenza. Se si tratta di una stringa ogni carattere viene trattato come un elemento distinto, quindi viene scomposta. I vari operatori analizzati per le liste si possono utilizzare anche per le tuple:

```
t[i]        # valore all'indice i
t[i:j]      # valori dall'indice i a j-1
t[:j]       # valori dall'indice 0 a j-1
t[i:]       # valori dall'indice i a len(t)
t[:]        # copia della tupla
```

Se si provasse a modificare un elemento verrebbe sollevato un errore, è comunque possibile creare una copia sostituendo solamente certi valori, così come per le stringhe, effettuando una concatenazione.

Gli operatori di confronto funzionano sulle tuple e su altre sequenze su Python. Cominciano a confrontare dal primo elemento di ciascuna sequenza, se sono uguali passa all'elemento successivo fino a trovare una coppia di elementi diversi. Quindi gli elementi successivi non vengono presi in considerazione.

Si può utilizzare la funzione `sort()`, può essere utile con lo schema DSU, "Decorate, Sort, Undecorate". Secondo questo approccio si decora una sequenza costruendo un elenco di tuple con una o più chiavi di ordinamento che precedono gli elementi della sequenza, si ordina tramite questa funzione ed in seguito si rimuove questa decorazione estraendo gli elementi della sequenza.

Una caratteristica di Python consente di assegnare più di una variabile alla volta, potendo inserire una tupla sul lato sinistro dell'assegnazione, senza parentesi tonde per convenzione. Un'applicazione interessante permette di scambiare i valori tra due variabili con una singola istruzione:

```
>>> a, b = b, a
```

Il numero di valori a destra e sinistra dell'assegnazione devon corrispondere per non sollevare errori. Un altro metodo chiamato `items()` permette di restituire una lista di tuple dove ogni tupla è una coppia chiave-valore. Questa lista non è ordinata, ma essendo le tuple comparabili, si può ordinare con il metodo `sort()`.

Gli insiemi sono collezioni non ordinate di altri oggetti. L'istruzione per creare un insieme è `set()`, inserendo una sequenza di elementi. In un insieme non possono esserci elementi ripetuti quindi vengono rimossi al momento della creazione.

Si possono effettuare le classi operazioni su insiemi, con metodi di unione `union()`, intersezione `intersection()`, differenza `difference()` e differenza simmetrica `symmetric_difference()`.

3.8.3 Liste Concatenate

Il tipo di dato astratto lista concatenata è diversa dal tipo di dato predefinito lista già presente su Python. Si parlerà di liste singolarmente concatenate. Questa lista è composta da un nodo, contenente il dato ed un puntatore all'elemento successivo di questa struttura concatenata. L'ultimo nodo della lista contiene un puntatore nullo. Per accedere agli elementi si hanno due puntatori di accesso, `head` e `tail`, per accedere in testa o in coda alla lista.

É comodo avere un accesso facilitato a queste due posizioni per effettuare facilmente operazioni di aggiunta, rimozione, modifica in testa o in coda. Un nodo è una classe contenente questi due unici attributi:

```
class Node:
    data = ''
    next = None

    def __init__(self, data, next):
        self.data = data
        self.next = next
```

L'utilizzo del paradigma orientato agli oggetti non è l'unico modo per definire una lista concatenata, ma è il modo utilizzato durante questo corso.

Una lista concatenata è una classe contenente i riferimenti alla testa ed alla coda della lista, inizialmente assegnati al valore nullo, per generare una lista vuota:

```
class SinglyLinkedList:
    def __init__(self):
        self.__head = None
        self.__tail = None
```

In realtà il campo dati è un riferimento ad un altro oggetto contenente i dati, ma per semplificare si considera il campo dati direttamente contenente questi.

Per scandire la lista elemento per elemento si può utilizzare un puntatore ausiliario, inizialmente assegnato al primo elemento della lista:

```
p = self.__head
```

Per passare all'elemento successivo bisogna assegnare questo puntatore al valore del puntatore a `next` contenuto nel nodo corrente:

```
p = p.next
```

Per scandire la lista quindi si utilizza un ciclo, fino a quando il valore contenuto in questo puntato `p` non corrisponde al valore `None`:

```
class SinglyLinkedList:
    def __init__(self):
        self.__head = None
        self.__tail = None

    def operazione():
        p = self.__head
        while p != None:
            # operazioni sul nodo
            p = p.next
```

Per cominciare a scandire la lista da un punto preciso si può inserire un parametro di classe `Node` a cui assegnare `p`. Si vogliono realizzare i seguenti metodi sulle liste singolarmente concatenate:

- `append`: Inserimento in coda;
- `insert_head`: Inserimento in testa;
- `insert_position`: Inserimento in una posizione intermedia;
- `delete`: Cancellazione di un elemento;
- `is_empty`: Verifica se la lista è vuota.

```
class SinglyLinkedList:
    def __init__(self):
        self.__head = None
        self.__tail = None
```

```
def append(self, newNode)
    if self.__tail != None:
        self.__tail.next = NewNode
    if self.__head == None:
        self.__head = NewNode
    self.__tail = NewNode
    # newNode.next = None

def insert_head(self, newNode):
    if self.__head == None:
        self.__tail = self.__head
    newNode.next = self.__head
    self.__head = newNode

def insert_position(self, newNode, position):
    p = self.__head
    i = 0
    while i < position - 1 and p != None:
        p = p.next
        i += 1
    newNode.next = p.next
    p.next = newNode

def delete(self, position):
    if self.__head == None:
        return None
    if position == 0:
        self.__head = self.__head.next
        return None
    p = self.__head
    i = 0
    while i < position - 1 and p.next != None:
        p = p.next
        i += 1
    if i != position - 1:
        return None
    p.next = p.next.next

def is_empty(self):
    return self.__head == None
```

3.8.4 Liste Ordinate, Pile e Code

Una lista ordinata è una lista ai cui elementi sono ordinati rispetto ad una certa chiave. Ogni inserimento deve quindi rispettare l'ordine della lista.

Questa lista si può implementare tramite la liste concatenate:

```
class SortedLinkedList:
    def __init__(self):
        self.__head = None
        self.__tail = None
```

Si suppone di poter effettuare un inserimento in posizione intermedia. Bisogna scandire la lista fino a quando il dato da inserire non è maggiore del dato contenuto nel nuovo nodo da inserire. Si considera la chiave e la lista ordinata in modo crescente:

```
    def add(self, newNode)
        if self.__head == None or self.__head.data > newNode.data:
            self.insert_head(newNode)
        elif self.__tail.data < newNode.data:
            self.append(newNode)
        else:
            p = self.__head
            while p.next != None and p.data < newNode.data:
                p = p.next
            newNode.next = p.next
            p.next = newNode
```

Una pila è un tipo di dato astratto basato sul modello delle liste che gestisce gli elementi, implementando la disciplina LIFO, "Last-In, First-Out". Le operazioni vengono effettuate su un estremo della lista chiamato testa `top`.

La classe `Node` è sempre la stessa, viene definita con un attributo None inizializzato a `None` e può essere comodo avere a disposizione una variabile che indica il numero di elementi nella pila:

```
class Stack:
    __top = None
    __size = 0
```

Ogni inserimento in una pila è un inserimento in testa, si chiama "push":

```
    def push(self, newNode):
        newNode.next = self.__top
        self.__top = newNode
        self.__size += 1
```

Per effettuare un'estrazione si estrae un elemento dalla testa, e generalmente il dato estratto è di interesse:

```
    def pop(self):
        p = self.__top
        if p != None
            self.__top = p.next
            p.next = None
        self.__size -= 1
        return p
```

Le code sono un altro tipo di dato astratto che utilizzano il modello delle liste, ma applicano la disciplina FIFO, "First-In, First-Out":

```
class Queue:
    __head = None
    __tail = None
    __size = 0
```

L'inserimento di un elemento avviene sempre in coda, e si chiama "enqueue":

```
    def enqueue(self, newNode):
        if self.__head == None:
            self.__head = mewNode
        else:
            self.__tail.next = newNode
        self.__tail = newNode
        self.__size += 1
```

Per rimuovere il primo elemento, si utilizza la funzione "dequeue":

```
    def dequeue(self):
        p = self.__head
        if p != None
            self.__head = p.next
            p.next = None
        self.__size -= 1
        return p
```

3.9 Implementazioni

La tecnica della "memoization" permette di implementare in modo efficiente funzioni che necessitano di salvare o memorizzare gli stadi intermedi di un'operazione, come può essere una funzione che calcola il fattoriale o numeri di fibonacci.

Utilizzando una lista si possono salvare i valori intermedi calcolati, senza doverli ricalcolare in iterazioni successive dell'algoritmo.

Un'esempio di quest'implementazione è la seguente:

```
def fattoriale_ric(n):
    if n == 1:
        return [1]
    sequenza = fattoriale_ric(n-1)
    sequenza.append(sequenza[-1] * n)
    return sequenza
```

In questo modo si evitano alcune inefficienze come dover chiamare la funzione `len()` ad ogni iterazione, essendo una funzione lineare, il costo totale sarebbe quadratico. Analogamente per il calcolo della sequenza di Fibonacci fino ad un certo indice:

```
def fibonacci_ric(n):
    if n == 0:
        return [0]
    elif n == 1:
        return [0, 1]
    sequenza = fibonacci_ric(n-1)
    sequenza.append(sequenza[-1] + sequenza[-2])
    return sequenza
```

Per implementare in modo efficiente queste operazioni non bisogna ricominciare ad ogni iterazione l'operazione che si vuole ottenere.

Queste operazioni possono essere implementate utilizzando la stessa tecnica anche con un approccio iterativo.

Un modo efficiente per scambiare due porzioni di una lista, noto l'indice i in cui termina la prima sotto-lista, consiste nell'utilizzo dell'operatore slice e della concatenazione delle due sottoliste:

```
lista = lista[i:] + lista[:i]
```

3.9.1 Funzioni Anonime

In Python si possono assegnare ad una lista i nomi di alcune funzioni considerate, e si può accedere a queste tramite l'indice della lista, inserendo i parametri tra parentesi tonde come se fosse la funzione stessa:

```
lista_funzioni = [funzione1, ..., funzioneN]
lista_funzioni[i](parametri)
```

Ma questo richiede comunque di definire precedentemente delle funzioni. ALtrimenti è possibile inserire direttamente nell'assegnazione alla lista la loro definizione utilizzando la notazione lambda. Questo è un aspetto della programmazione funzionale, utilizzabile su Python tramite la parole chiave `lambda`, specificando gli argomenti della funzione, e dopo `:` si inserisce l'operazione che produce il risultato di questa funzione:

```
lambda parametri : # operazioni sui parametri per produrre un output
```

Quindi una lista contenente funzioni anonime è del tipo:

```
lista_funzioni = [lambda parametri1 : operazioni1, ..., lambda parametriN: operazioniN]
```

3.9.2 Dynamic Programming

Per determinare l'efficienza di un algoritmo, si può utilizzare il comando `%time` per ottenere informazioni sul tempo di utilizzo del processore da parte della funzione:

```
>>> %time funzione(parametri)
```

Il "memoization" è un approccio al "dynamic programming" che risolve questo problema in modo top down, scomponendo il problema in sotto-problemi, e salvando i risultati parziali in una tabella. Quando deve eseguire un nuovo sotto-problema, cerca prima se sono già presenti in questa tabella, prima di calcolare direttamente il risultato. Si può implementare il calcolo della sequenza di Fibonacci utilizzando questo approccio. Per determinare se è necessario calcolare direttamente il risultato parziale, si usa il costrutto try-except. In questo costrutto si prova ad accedere all'elemento, se questo accesso solleva un errore ovvero un fallimento, il programma calcolare direttamente questo valore parziale:

```
def fibonacci_memoization(n, m = None):
    if m == None:
        m = {}
    if n <= 1:
        return n
    try:
        return m[n]
    except:
        risultato = fibonacci_memoization(n - 1, m) + fibonacci_memoization(n - 2, m)
        m[n] = risultato
        return risultato
```

Si utilizza in questo caso si utilizza un risultato, popolando la tabella ad ogni iterazione i, inserendo il risultato parziale utilizzando come chiave questo stadio parziale i.

Questo approccio è decisamente più veloce dell'implementazione precedente `fibonacci_ric`, avendo una complessità temporale lineare $O(n)$.

Invece di utilizzare il costrutto try-except si può utilizzare un'istruzione condizionale per verificare che il valore di chiave i considerato è presente nel dizionario con l'operatore `in`:

```
m = {}
def fibonacci_memoization(n):
    if n <= 2:
        return 1
    if n not in m:
        m[n] = fibonacci_memoization(n-1) + fibonacci_memoization(n-2)
    return m[n]
```

Oltre all'approccio di memoization si può utilizzare un metodo duale chiamato "tabular", con una disciplina bottom up, partendo dai sotto-problemi più piccoli e memorizzando i loro risultati parziali su una tabella, combinando queste soluzioni per risolvere i successivi problemi, memorizzando i loro risultati sulla stessa tabella.

Si considera un'implementazione utilizzando questo approccio della sequenza di Fibonacci:

```
def fibonacci_tabular(n):
    tab = [0,1]
    for i in range(2, n+1):
        tab.append(tab[i-1] + tab[i-1])
    return tab[n]
```

Anche questo approccio ha un'implementazione in tempo lineare $O(n)$.

3.10 Algoritmi di Ricerca

3.10.1 Algoritmo Greedy: Problema di Ricerca di un Itinerario

L'algoritmo "Greedy" è un algoritmo best-first, che utilizza conoscenza euristica per determinare il miglior nodo dalla frontiera da espandere. La conoscenza si rappresenta tramite una funzione di valutazione f, per ogni funzione di valutazione si possono definire diversi algoritmi di ricerca informata. Generalmente la frontiera viene realizzata come una sequenza ordinata di nodi, sulla abase di questa funzione f.

Nell'algoritmo Greedy, la funzione di valutazione è la funzione euristica h, definita in base alla conoscenza sul dato problema per stimare la "distanza" dello stato corrente dalla soluzione.

Per problemi di viaggio e della ricerca di itinerari, la distanza rappresenta la distanza fisica dallo stato corrente alla destinazione. Una delle funzioni utilizzate in questi problemi come euristica è la distanza in linea d'aria.

Per implementare questo tipo di algoritmi di viaggio è necessario rappresentare lo spazio degli stati, generalmente ottenuto da una mappa, quindi come un grafo connesso. Si può realizzare come un dizionario, dove uno stato viene utilizzato come chiave e corrisponde ad un insieme dei suoi stati successori:

```
        # Nome Stato:
connections[ " ... " ] =
{   # Nome Primo Stato Successore:
    " . . . ",
    ... ,
    #Nome Ultimo Stato Successore:
    " . . . "
}
```

Non è l'unica possibilità per rappresentare un grafo, esistono infatti molte librerie realizzate unicamente per gestire grafi come questi, ma è utile ai fini didattici realizzare il problema per intero prima di avvalersi di librerie esterne. In questa rappresentazione del grafo non è presente la distanza tra le città, può essere inserita come un insieme di tuple contenenti lo stato successore e la distanza a quello stato. Ma per implementare l'algoritmo Greedy non è necessario sapere la distanza, poiché si utilizza solamente la funzione euristica, e non il percorso parziale, come nell'algoritmo A*. Per mettere in evidenza questo concetto viene intenzionalmente omessa in questa rappresentazione del grafo.

Anche la funzione euristica h viene implementata tramite un dizionario, contenente per ogni stato, utilizzato come chiave, il suo valore:

```
#  Nome Stato:  Valore Euristica:
h[ " ... " ] = h(s_i)
```

Per realizzare i nodi dell'albero di ricerca si crea una nuova classe, chiamata `Node`, contenente lo stato, un riferimento al suo nodo genitore ed il valore della funzione euristica per lo stato contenuto:

```
class Node:
    def __init__(self, state, parent, h):
        self.state = state
        self.parent = parent
        self.h = h
```

Ulteriormente si possono inserire la sua profondità nell'albero ed una lista dei suoi nodi figli, per rappresentare graficamente l'albero:

```
class Node:
    def __init__(self, state, parent, h):
        self.state = state
        self.parent = parent
        self.h = h
        self.children = []
        self.depth = 0
```

Per rappresentare graficamente l'albero bisogna inserire inoltre un metodo per aggiungere dei figli al nodo:

```
    def addChild(self, childNode):
        self.children.append(childNode)
        childNode.parent = self
        childNode.depth = self.depth + 1
```

Inoltre per visualizzare il percorso quando viene individuata la soluzione, si definisce un altro metodo:

```
    def printPath(self):
        if self.parent != None:
            self.parent.printPath()
        print("-> ", self.state.name)
```

Questo metodo invocato su un nodo risale i nodi dell'albero fino ad arrivare alla radice, definite con un campo parent contenente `None`, quindi comincia a stampare gli stati a partire dalla radice.

Si definisce ulteriormente la classe `State` per rappresentare gli stati del problema, con un campo nome per identificare gli stati. Questa classe ha i metodi per restituire lo stato iniziale, per restituire i successori di questo nodo, accedendo al dizionario delle connessione degli stati, e per controllare se è lo stato obiettivo:

```
class State:
    def __init__(self, name = None):
        if name == None:
            self.name = self.getInitialState()
        else:
            self.name = name

    def getInitialState(self):
                    #  Nome Stato Iniziale:
        initialState = " ... "
        return initialState

    def successorFunction(self):
        return connections[self.name]

    def checkGoalState(self)
                           # Nome Stato Obiettivo:
        return self.name ==   " ... "
```

Per implementare la frontiera si utilizza una lista ordinata con classi ed oggetti, ma esistono strutture dati più efficienti per implementare una "priority queue", come l'heap. Python contiene un'implementazione di una lista con priorità, ma analogamente allo spazio degli stati si utilizza per fini didattici un'implementazione dedicata:

```
class Item:
    value = None
    node = None
    next = None

    def __init__(self, value, node):
        self.value = value
        self.node = node
        self.next = None
```

La classe della lista ordinata è la stessa definita precedentemente nella sezione dedicata 3.8.4. Sono necessari i metodi di inserimento, mantenendo l'ordinamento, il metodo di estrazione del primo elemento `remove_front`, analogo ad una `pop` per le pile, ed un metodo per controllare se la frontiera è vuota `is_empty()`. Questi metodi sono già stati descritti nella sezione specificata 3.8.3.

Si definisce quindi l'algoritmo Greedy, inizializzando la frontiera, lo stato iniziale e l'euristica. Inserendo la radice nella frontiera:

```
def Greedy_Best_First():
    fringe = SortedLinkedList()

    initialState = State()
```

```
root = Node(initialState, None, h[initialState.name])

fringe.add(Item(root.h, root))
```

In seguito bisogna inserire il ciclo che verrà eseguito fino all'esaurimento della frontiera, controllando ad ogni iterazione che non sia vuota tramite il metodo `is_empty()`. Ad ogni iterazione viene estratto il primo nodo dalla frontiera e si controlla se questo è un nodo obiettivo. Se è un nodo obiettivo si interrompe l'esecuzione e viene stampato il percorso ottenuto. Altrimenti bisogna continuare l'esecuzione espandendo questo nodo, ed inserendo i suoi successori all'interno della frontiera:

```
while not fringe.is_empty():
    itemRemoved = fringe.remove_front()
    currentNode = itemRemoved.node

    if currentNode.state.checkGoalState():
        currentNode.printPath()
        break
    else:
        childStates = currentNode.state.successorFunction()
        for childState in childStates:
            childNode = Node(State(childState), currentNode,
                             h[State(childState).name])
            # per rappresentare graficamente l'albero:
            # currentNode.add(childNode)
            fringe.add(Item(childNode.h, childNode))
```

Iterando su tutti gli stati successori del nodo corrente bisogna creare il nodo corrispondente ed aggiungere l'elemento relativo a questo nodo nella frontiera. Quest'implementazione utilizza un approccio tree-search.

Se invece si utilizza la coda di priorità da una libreria esterna, bisogna importa, ed utilizzare i suoi metodi allo stesso modo della lista ordinata utilizzata precedentemente:

```
import queue as queue

# per la creazione:
fringe = Queue.PriorityQueue()

# per l'inserimento si utilizza una tupla, (euristica, nodo):
fringe.put((node.h, node))

# per la verifica della frontiera vuota:
fringe.empty()
```

```
# per la rimozione del primo elemento:
fringe.get()
```

3.10.2 Algoritmo A*: Problema di Ricerca di un Itinerario

L'algoritmo A* è un algoritmo Best-First, ma è molto migliore rispetto all'algoritmo Greedy poiché la sua funzione di valutazione considera anche il percorso parziale dalla radice al nodo corrente. Si può implementare nelle due versioni dell'algoritmo Best First, in modalità Tree-Search e Graph-Search rispettivamente. Nella modalità Tree-Search ogni volta che viene espanso uno stato, i nodi corrispondenti ai suoi stati successori vengono inseriti nella frontiera, mentre nella Graph-Search gli stati già visitati vengono inseriti in un insieme, per non ripetere nuovamente lo stesso stato. Questo viene effettuato senza inserire gli stati già presenti in questo insieme in frontiera.

In questa implementazione è necessario conoscere la distanza parziale, quindi lo spazio degli stati viene realizzato tramite un dizionario, dove i valori sono costituiti da coppie stato successore e distanza:

```
        #  Nome Stato i-esimo:
connections[ " ... " ] =
{   # Nome Primo Stato Successore |j|-esimo e distanza:
    [" . . . ", c_{i,j}],
    ...  ,
    # Nome Ultimo Stato Successore |k|-esimo e distanza:
    [" . . . ", c_{i,k}]
}

connections[stato] = {[successore1, distanza1], ..., [successoreN, distanzaN]}
```

La funzione euristica viene definita allo stesso modo dell'algoritmo precedente, tramite un dizionario:

```
#  Stato:    Valore Euristica:
h[ " ... " ] = h(s_i)
```

Si definisce la classe di un nodo, che ora deve contenere anche la distanza parziale:

```
class Node:
    def __init__(self, state, parent, f, partialPath):
        self.state = state
        self.parent = parent
        self.f = f
        self.partialPath = partialPath
        # self.children = []
        # self.depth = 0
```

I metodi per stampare il percorso sono gli stessi dell'algoritmo Greedy, così come la classe che definisce uno stato del problema e la lista ordinata per la frontiera.

Si implementa direttamente la versione Graph-Search dell'algoritmo. Si inizializza allo stesso modo dell'implementazione precedente, ma bisogna inizializzare un'altra collezione per rappresentare l'insieme dei nodi già visitati:

```
def A_Star():
    fringe = SortedLinkedList()
    close = []

    initialState = State()
    root = Node(initialState, None, h[initialState.name], 0)

    fringe.add(Item(root.f, root))
```

Analogamente la prima parte dell'algoritmo dentro al ciclo è uguale, bisogna estrarre un nodo dalla frontiera, controllare se corrisponde allo stato obiettivo ed in caso stampare la soluzione ottenuto e terminare l'esecuzione dell'algoritmo:

```
while not fringe.is_empty():
        itemRemoved = fringe.remove_front()
        currentNode = itemRemoved.node

        if currentNode.state.checkGoalState():
            currentNode.printPath()
            break
```

Altrimenti bisogna verificare se non è già stato espanso, in caso si aggiunge alla lista dei nodi già visitati e si espande, iterando sui suoi stati figli per creare i corrispondenti nodi ed aggiungerli alla frontiera, se non sono già stati visitati:

```
        if currentNode.state.name not in close:
            close.append(currentNode.state.name)

            childStates = currentNode.state.successorFunction()
            for (childState, distance) in childStates:
                g = childState.partialPath + distance
                f = g + h[childState]
                childNode = Node(childState, currentChild, f, g)

                if childState.name not in close:
                    fringe.add(Item(childNode.f, childNode))
```

Questo controllo non è necessario, poiché se viene estratto un nodo corrispondente ad un elemento già espanso, non viene comunque espanso.

3.10.3 Algoritmo Hill-Climbing: Problema di Ricerca di una Stringa

Si considera la versione First-Choice dell'algoritmo Hill-Climbing. L'algoritmo dato uno stato corrente sceglie un valore migliore, in base alla versione dell'algoritmo, fino a quando non trova un massimo locale. Questo algoritmo quindi passa tra stati di valore sempre crescente e termina quando raggiunge il picco. In questa versione First-Choice, l'algoritmo sceglie casualmente uno stato adiacente e verifica se questo stato è il migliore, ed in caso lo considera il nuovo stato corrente. Si vuole applicare questo problema dove da una stringa iniziale di n caratteri si vuole raggiungere una stringa obiettivo di n caratteri. Bisogna individuare gli stati e gli operatori da poter applicare ad una stringa. Uno stato è un'istanza di una stringa da n caratteri, ed un operatore può modificare un singolo di questi n caratteri sostituendolo con uno dei caratteri disponibili. In questo modo vengono creati gli stati adiacenti del problema. Per considerare tutti i successori bisognerebbe considerare per ciascuno degli n caratteri uno dei caratteri disponibili.

La funzione di valutazione in questo problema è la distanza dalla stringa obiettivo, calcolabile come la somma delle distanze dei singoli caratteri dalla stringa corrente alla stringa obiettivo.

I caratteri disponibili sono i 100 caratteri stampabili su Python, visualizzabili con `.printable`, dopo aver importato l'opportuno modulo `string`. Il numero di stati possibili è 100^n, mentre per ognuna delle n posizioni si hanno 100 posizioni diverse, quindi dato uno stato si hanno $100 \cdot n$ stati adiacenti.

Per generare uno stato iniziale, si può generare una stringa di n caratteri scelti casualmente:

```
import random
import string

def generate_random_string(length):
    return [random.choice(string.printable) for _ in range(length)]
```

Per calcolare la distanza tra due stringhe si può utilizzare la funzione built-in `ord()` che restituisce il codice Unicode del carattere inserito come parametro. Si può iterare su ognuno di questi caratteri e calcolarne la distanza dalla stringa obiettivo:

```
def evaluate(sol):
    trg = list(targetString)
    diff = 0
    for i in range(len(trg)):
        diff += abs(ord(sol[i]) - ord(trg[i]))
    return diff
```

Per scegliere il successore bisogna scegliere casualmente la posizioni da modificare ed il carattere da sostituire:

```
def tweak(sol):
    i = random.randint(0, len(sol) - 1)
    sol[i] = random.choice(string.printable)
```

L'algoritmo quindi itera continuamente fino a quando lo stato corrente non è nullo, visitando sempre e solo stati di valore minore di quello corrente. Dato che gli algoritmi Hill-Climbing risolvono

sia problemi di massimo che di minimo, essendo tra loro duali. Si può scegliere se farlo partire da una stringa casuale oppure da stringa specificata dall'utente.

```
def First_Choice_Hill_Climbing(initialString = None):
    if initialString == None:
        currentState = generate_random_string(length)
    else:
        currentState = initialString
    currentEval = evaluate(initialString)

    while True:
        if currentEval == 0:
            break

        nextState = tweak(list(currentState))
        nextEval = evaluate(nextState)

        if nextEval < currentEval:
            currentState = nextState
            currentEval  = nextEval
```

Questi algoritmi operano nel discreto, anche se molte volte si lavora su domini continui, e sono necessari altri controlli e test diversi per poter implementare algoritmi di ricerca.

3.10.4 Algoritmo Simulated Annealing: Problema delle n Regine

In questo algoritmo si parte da uno stato iniziale casuale e si sceglie casualmente un successore, come nell'algoritmo precedente, e si accettano sempre stati adiacenti migliori, mentre in base ad una probabilità se peggiori. Una variabile chiamata temperatura determina il cambiamento di questa probabilità. All'aumentare della temperatura diventa sempre meno probabile che stati adiacenti di valore peggiore vengono scelti dall'algoritmo. Questo algoritmo termina quando trova uno stato obiettivo, oppure se rispetta cerca condizioni di terminazione.

Il problema delle n regine consiste nell'individuare una disposizione su una scacchiera $n \times n$, dove n regine presenti non si attaccano fra di loro. Ovvero se si potessero muovere non potrebbero catturarsi a vicenda. Una regina negli scacchi si può muovere sia verticalmente, che orizzontalmente, che diagonalmente. In questo problema lo spazio degli stati consiste in stati contenenti n regine, una in ogni colonna ed una in ogni riga. In questo modo bisogna solamente analizzare conflitti sulle diagonali. In questo modo viene limitata notevolmente lo spazio degli stati e viene semplificata la sua rappresentazione, invece che con una matrice, con un semplice vettore. Dato uno stato per considerare i successori si considera uno scambio tra due colonne, per mantenere il primo vincolo sugli stati. Lo stato obiettivo è un qualsiasi stato dove le regine non sono in conflitto tra di loro. La funzione di valutazione sono il numero di attacchi in una data disposizione della scacchiera, che si vuole minimizzare. Una scacchiera si può rappresentare come un vettore lungo n, dove l'indice individua la colonna, ed il valore contenuto la riga corrispondente, poiché una sola regina può essere

su ogni colonna ed ogni riga. Per passare da uno stato ad uno adiacente si utilizza una funzione `tweak()`. Questa funzione scambia due colonne, scegliendo casualmente due indici ed invertendo i loro valori:

```
def tweak(sol):
    solCopy = np.copy(sol)

    x = random.randint(0, n-1)
    y = random.randint(0, n-1)
    while x == y:
        y = random.randint(0, n-1)
    solCopy[x], solCopy[y] = solCopy[y], solCopy[x]

    return solCopy
```

Per inizializzare il problema si sceglie casualmente uno stato iniziale. Quindi partendo da uno stato iniziale, come potrebbe essere una disposizione avente le regine sulla diagonale principale, ed operare un certo numero di volte la funzione `tweak()` per ottenere uno stato casuale:

```
def initialize(sol):
    for _ in range(0, initialSteps):
        sol = tweak(sol)
    return sol
```

Questo valore `initialSteps` deve essere un numero significativo per realizzare ad ogni esecuzione uno stato nuovo casuale. Convenzionalmente la funzione di valutazione nel Simulated-Annealing viene chiamata energia, poiché in metallurgia per effettuare questo processo si vuole appunto minimizzare l'energia del sistema. La funzione di valutazione deve considerare solamente gli spostamenti in diagonale delle regine sulla scacchiera. Queste si possono spostare in quattro possibili direzioni: $(1, 1)$, $(1, -1)$, $(-1, 1)$ e $(-1, -1)$. Ovvero possono incrementare o decrementare gli indici di riga o colonna di uno. Per verificare se sono presenti conflitti, data la posizione originale di una regina, si segue una delle diagonali individuata da una di queste quattro direzioni, fino a raggiungere il bordo della scacchiera. Si può implementare realizzando la scacchiera tramite una matrice contenente solo valori nulli, ed inserendo le regine con un valore specifico. In questo modo ci si può spostare sulle diagonali come su un piano cartesiano discreto, su direzioni individuate da spostamenti unitari $\mathrm{d}x$ e $\mathrm{d}y$. Iterando su ogni regina, si considera la sua posizione corrente x e y, e si incrementa progressivamente tramite questi spostamenti unitari su una delle diagonali. Ad ogni passo si controlla se la posizione è interna alla scacchiera, ed in caso si termina la ricerca per questa diagonale, oppure se la casella corrispondente contiene una regina, ed in caso si incrementa il contatore dei conflitti. Questa ricerca dei conflitti termina dopo aver percorso le diagonali di tutte le regine contenute nello stato passato come parametro, restituendo il numero di conflitti ottenuti. Questo numero sarà sempre pari, poiché per ogni coppia di regine in conflitto sono possibili due attacchi sia dalla prima che dalla seconda regina.

```
def energy(sol):
    board = [[0] * n for i in range(n)]
```

```
    for i in range(0, n):
        board[sol[i]][i] = 1

    dx = [1, 1, -1, -1]
    dy = [1, -1, 1, -1]

    conf = 0
    for i in range(0, n):
        x = sol[i]
        y = i
        for j in range(0, 4):
            tempx = x
            tempy = y
            while True:
                tempx = tempx + dx[j]
                tempy = tempy + dy[j]
                if (tempx < 0 or
                    tempx >= n or
                    tempy < 0 or
                    tempy >= n):
                    break
                if board[tempx][tempy]:
                    conf += 1
    return conf
```

In seguito bisogna definire i parametri iniziali dell'algoritmo, quali la temperatura massima `TMax` e minima `TMin`, per cui termina l'algoritmo anche se non è stata individuata una soluzione, il numero di passi per iterazione `TSteps`, prima di cambiare la temperatura. Il numero di passi per scegliere uno stato iniziale casuale, e la dimensione della scacchiera `n`. Si considera il numero di passi iniziali uguali alla dimensione della scacchiera. Si utilizza inoltre un parametro $0 < \alpha < 1$: `alpha` per diminuire la temperatura `T`.

L'algoritmo inizia scegliendo uno stato casuale, passando uno stato con le regine sulla diagonale, e si salva la valutazione di questo stato. Si inizializza lo stato migliore ed il suo valore con i valori dello stato corrente. In seguito si itera fino alla temperatura minima oppure fino ad aver trovato la soluzione, iterando per ogni temperatura diversa sugli stati adiacenti dello stato corrente. Si considerano se hanno un valore minore oppure se vengono accettati con una data probabilità. Per calcolare la probabilità si utilizza il modulo `math`, con la funzione `.exp()` per il calcolo di un esponenziale. In caso si è scelto uno stato corrente si verifica se si tratta di uno stato migliore dello stato migliore precedentemente trovato, ed in caso si aggiorna, solamente se è migliore e non con probabilità.

```
def simulated_annealing():
    currentState = initialize(range(0, n))
    currentEnergy = energy(currentState)
```

```
bestState = currentState
bestEnergy = currentEnergy

T = TMax

while T > TMin and bestEnergy != 0:
    for _ in range(0, TSteps):
        useNew = False
        nextState = tweak(currentState)
        nextEnergy = energy(nextState)

        if (nextEnergy < currentEnergy or
            random.random() > math.exp((currentEnergy-nextEnergy)/T)):
            useNew = True

        if useNew:
            currentState = nextState
            currentEnergy = nextEnergy

            if currentEnergy < bestEnergy:
                bestState = currentState
                bestEnergy = currentEnergy
        T = T * alpha
return bestState
```

4 Fondamenti di Machine Learning

In programmi convenzionali, è possibile determinare l'output, dati un insieme di input ed il suo algoritmo, in modo deterministico. Questo però non è possibile per programmi che utilizzano machine learning. Si utilizzano dei modelli che vengono addestrati ad eseguire un'operazione, con certe prestazioni. Ma il loro comportamento non è noto a priori, ma dipende dai dati di input. L'output per una stessa task cambia a seconda dei dati in input. In generale il ricercatore di ML prende il nome di "Data Scientist", poiché i dati, e non gli algoritmi hanno ruolo centrale in questo settore.

4.1 Pattern

Si utilizza spesso il termine di "Pattern" per riferirsi ai dati, oltre a rappresentare un forma, esempio o modello, si utilizzerà questo termine per riferirsi ad un generico data point, un elemento all'interno di una task. Uno dei padri del pattern recognition Satoshi Watanabe, un settore con ampia intersezione ed influenza con il ML, definisce il pattern come l'opposto del caos. Sopratutto a livello della biometria il ML e la computer vision sono costantemente presenti, nelle loro analisi su dati sia comportamentali o fisiologico. Tramite questi dati è possibile identificare o riconoscere un individuo.

Si analizzeranno solamente pattern di tipo numerico, quindi misurabili. Sono caratteristiche che hanno un valore numerico e sono soggette ad un ordinamento. Tipicamente sono valori continui, ma possono anche essere discreti. Si rappresentano come vettori di features, vettori numerici contenenti le singole caratteristiche misurabili. Ognuna delle features del vettore assume un valore numerico misurabile, ed insieme definiscono un pattern.

Dei dati altrettanto importanti sono i dati categorici, caratteristiche qualitative o la presenza assenza di una caratteristica, quindi molto spesso sono features binarie. In alcuni casi sono soggetti ad ordinamento, alcune caratteristiche per essere analizzate devono essere categorizzabili. Normalmente vengono gestiti da sistemi a regole e alberi di classificazione.

I dati più complicati sono delle sequenze, spaziali o temporali, come suoni, frasi, immagini, video, etc. Spesso la lunghezza delle sequenze sono variabili, e la loro posizione interna e le relazioni con gli elementi adiacenti, i predecessori ed i successori, sono importanti. Bisogna definire una struttura di similarità per poter confrontare tra di loro due sequenze. Tecniche utilizzate sono il "Dynamic Time Warping" (DTW), "Hidden Markov Models" (HMM), "Recurrent Neural Networks" (RNN), "Long Short-Term Memory" (LSTM).

Altri tipi di dati possono essere strutturati, in strutture complesse come alberi o grafi. Per esempio in bioinformatica, per l'analisi delle strutture genetiche, oppure nel riconoscimento e processamento del linguaggio naturale, dove l'output desiderato è l'insieme dei parse tree plausibili. Si utilizzano tecnica come le "Structured SVMs", "Bayesian Networks" oppure HMM.

Osservando un determinato fenomeno si può realizzare un "heatmap", caratterizzata da toni freddi o caldi, dove più sono caldi più la feature è alta, e più sono freddi più è bassa. Non rappresenta necessariamente una mappa di calore, ma deve la sua struttura e rappresentazione convenzionale a questo tipo di mappe. Questo riguarda tantissimi fenomeni dove è possibile rappresentare su un piano le features analizzate.

Avendo pochi dati in un dataset è possibile generarne di nuovi da dati preesistenti con una tecnica chiamata "Data Augmentation". Oppure si possono utilizzare reti neurali già addestrate, con una tecnica chiamata "Transfer Learning". Esistono dei metodi che permettono di aggiungere insieme alla risposta del sistema una condizione, o pattern, individuato tra i dati in input che l'ha portato a scegliere quella risposta. In questo modo il sistema è in grado di fornire una spiegazione per la sua risposta.

4.2 Applicazioni

4.2.1 Classificazione

Le applicazione classiche del ML sono primo tra tutti la classificazione. Ovvero dato un dataset, un piano la classificazione deve assegnare ad ognuno di questi pattern o data point una classe di appartenenza. Bisogna apprendere dai dati e generare una funzione che permette di effettuare un mapping tra lo spazio dei dati e lo spazio delle classi. Nel caso di sole due classi si usa il termine "Binary Classification", con più classi invece "Multi-Class Classification". Anche se spesso si utilizza il termine riconoscimento.

Una classe è quindi un insieme di pattern aventi delle caratteristiche simili o comuni tra di loro. Il concetto di classe è semantico e dipende strettamente dall'applicazione.

Problemi di classificazione classici sono:

- Spam Detection: strumenti, operativi nei client di posta, per identificare se la posta è indesiderata o illegittima. Il pattern sono le singole mail e le classi sono mail legittime o spam;
- Credit Card Fraud Detection: i pattern sono le singole transazioni e le classi sono transazioni fraudolente o operazioni legittime;
- Face Recognition: presuppone l'identificazione facciale, su cui effettuare il riconoscimento. Classifica le faccia in uno dei soggetti appartenenti ad un dataset;
- Pedestrian Classification;
- Medical Diagnosis;
- Stock Trading.

4.2.2 Regressione

L'altra grande applicazione del ML è la regressione, sia la regressione che la classificazione sono predizioni. Ma nella regressione si attribuisce un valore continuo ad un pattern. Corrisponde ad apprendere una funzione approssimante delle coppie di input-output. Si parla di funzioni di grado di regolarità, con certe condizioni analitiche che devono rispettare, quali la continuità o la derivabilità.

Problemi tipici della regressione sono:

- Stima dei Prezzi, in certi tipi di mercati;

- Stima del Rischio, per compagnie assicurative;
- Predizione Energia Prodotta;
- Modelli Sanitari di Predizione dei Costi;
- Object Detection: la stima delle posizioni e le dimensioni della bounding box di oggetti in un immagine o video. Mentre il problema di classificazione associato è l'associazione di una classe all'oggetto dentro questa bounding box.

4.2.3 Clustering

Un'altra applicazione comune è il clustering, il classico problema dell'apprendimento non supervisionato. Sulla base della similarità o dissimilarità su di un dataset non etichettato, dopo aver definito queste misure e metriche per individuare similarità, è possibile raggruppare i pattern in cluster, di cui nemmeno il numero è noto a priori. Questi cluster possono essere poi utilizzati come delle classi.

Un classificatore di questo tipo si chiama k-NN, e sceglie nel piano gli i primi identificatori più vicini, e sulla base di un voto di maggioranza, sulla base di quanti di questi campioni sono assegnati a ciascuna delle classi. Questi data point possono essere utilizzati a loro volta come classi per realizzare un modello di apprendimento supervisionato.

Problemi di clustering sono:

- Marketing: definizione di gruppi di utenti in base ai consumi.
- Genetica: raggruppamento sulla base delle analogie del DNA;
- Bioinformatica: partizionamento dei gruppi di geni con simili caratteristiche, per individuare sequenze genetiche nuove rispetto a quelle note;
- Social Network Analysis: si possono analizzare gli utenti sulla base delle loro interazioni, individuando gruppi di utenti con interessi o interazioni simili;
- Visione: segmentazione non supervisionata, non trattata in questo corso.

Il flusso virtuale di un immagine nella computer visione effettua il processo analogo compiuto dal cervello umano, cercando i bordi nell'immagine. Questi bordi possono essere raggruppati per creare delle regioni. Quest'operazione si chiama segmentazione, una volta che si hanno questi segmenti si possono classificare. Con delle informazioni a contorno si potrebbe definire il grado di accuratezza di questa classificazione, ma in un ambiente non supervisionato ciò non è possibile. Si utilizza spesso nell'ambito delle immagini satellitari per individuare immagini, in seguito classificate successivamente.

4.2.4 Riduzione di Dimensionaltà

Un'altra applicazione molto usata nell'ambito del ML, meno diffuse rispetto ai primi è la riduzione di dimensionalità. Il data point potrebbe avere un certo numero di features molto elevato, nell'ambito dei "Big Data", rappresentabili in spazi con un numero altrettanto elevato di dimensioni. Si può dimostrare che questo numero di dimensioni non aiutano l'analisi, ma introducono rumore. Quindi è conveniente individuare metodi per ridurre la dimensione dei dati, effettuando un mapping da $\mathbb{R}^d$ a $\mathbb{R}^k$, dove $k < d$.

Questa operazione deve minimizzare la perdita delle informazioni, riducendo la dimensionalità dei dati. Una di queste tecniche è la PCA, risale agli anni '30, da Fisher, così come la LDA.

Queste tecniche combinano tra di loro dimensioni generando nuove features e diminuendo le features totali. Esistono delle tecniche per rendere rappresentabili su schermo i dataset, per visualizzarli in un ambiente 2D o 3D, partendo da $d > 3$. Scartando informazioni ridondanti o inutili.

La PCA è una tecnica non supervisionata, non tiene conto della differenza tra le classi, e nella riduzione di dimensionalità preserva il più possibile l'informazione dei pattern. Mentre nella LDA, "Linear Discriminant Analysis", tiene conto delle classi e preserva al massimo le informazioni che permettono di discriminare tra pattern. Una tecnica chiamata t-SNE, "Stochastic Neural Embedding", basata su reti neurali profonde, proposta da Geoffrey Hinton ed un suo studente. Che funziona particolarmente bene quando si vuole ridurre a due o tre dimensioni i dati.

4.2.5 Representation Learning

Ci sono delle features grezze che possono diminuire l'efficacia dei modelli per rappresentare i modelli. Esistono delle tecniche per creare feature a partire dai dati grezzi per rendere più efficace l'identificazione dei pattern, quest'applicazione prende il nome di "Feature Engineering".

4.2.6 Problemi di Computer Vision

I problemi di computer vision si dividono in due problemi, dove è presente una singola classe, quindi si hanno problemi di classificazione e locazione, mentre se sono presenti più classi in una stessa immagine si hanno problemi di object detection o istance segmentation, localizzando diversi oggetti attribuite a diverse classi.

4.3 Apprendimento

Su questi dati per effettuare le operazioni descritte un modello di ML deve essere in grado di apprendere. Si individuano due tipi di apprendimento, un approccio supervisionato, dove ogni datapoint è associato ad una classe. Si dice che il training set è etichettato, spesso in applicazioni di riconoscimento di immagini, approccio molto tipico nella classificazione, regressione ed in alcune tecniche di riduzione di dimensionalità.

L'approccio duale consiste nell'apprendimento non supervisionato, come nella social network analysis, dove sulla base delle caratteristiche degli utenti si individuano dei cluster. Questo approccio viene usato principalmente nei problemi di clustering, dove il training set non è etichettato, e

nella maggior parte delle tecniche di riduzione di dimensionalità. Molto spesso questi algoritmi di clustering devono anche individuare il numero di classi in un dataset.

Un altro algoritmo che ha ricominciato ad essere utilizzato, grazie all'aumento delle prestazioni delle risorse di calcolo, è l'approccio semi-supervisionato, dove il training set è parzialmente etichettato. La distribuzione dei pattern non etichettati può aiutare ad ottimizzare la regola di classificazione. Molto spesso viene utilizzato su applicazioni geometriche, si individua una distanza di decisione, tra questi datapoint etichettati. In seguito sui datapoint non etichettati si analizza la superficie di separazione ed il suo cambiamento nell'avvicinamento ad uno di questi datapoint.

Un ulteriore tecnica di addestramento si chiama batch, ed effettua l'addestramento una singola volta, dopo il quale l'algoritmo si trova in modalità di lavoro, e non è in grado di apprendere ulteriormente. Un modello duale è incrementale dove sono possibili diverse sessioni di addestramento, in sequenze di batch. Il rischio di questo approccio è che successive sessioni di addestramento il sistema dimentichi quello che ha appreso in precedenza. Attualmente con le risorse a disposizione non sono presenti questi rischi.

L'apprendimento di tipo naturale, è molto simile all'apprendimento umano, implica una quantità limitata di istruzioni dirette, in un approccio supervisionato, seguito da esperienze non supervisionate, in modo coesistente.

Un altro approccio molto utilizzato è il Reinforcement Learning, dove l'apprendimento si basa sulla reazione dell'ambiente, in seguito ad un'azione dell'agente. L'ambiente può fornire delle ricompense positive o negative, ottenute dall'agente, che utilizza per scegliere l'azione successiva. Molto spesso la ricompensa può essere ritardata rispetto ad una certa azione. L'obiettivo è di apprendere la sequenza di azioni migliori in un dato stato per ottenere il massimo dei risultati.

I domini più utilizzati sono quelli della robotica e dei videogiochi, dove è difficile formalizzare la realtà di interesse e descrivere in maniera completa questa realtà.

Il Deep Reinforcement Learning, è un reinforcement learning dove il dato in input all'agente non è formalizzato ma è direttamente un dato grezzo. Nel 2013 questo approccio da parte di ricercatori del DeepMind ha permesso ad agenti di imparare a giocare a videogiochi Atari. Questo loro approccio di "Deep Q-Learning" che si basa sull'equazione di Bellman. Hanno ottenuto ottimi risultati da parte degli agenti che in certi casi ha superato l'agente umano.

4.3.1 Parametri e Funzione Obiettivo

Un modello di ML è definito da un insieme di parametri Θ, come i pesi delle connessioni di una rete neurale. Questo modello è costituito da una serie di equazioni non particolarmente complicate, con tantissimi parametri, anche nell'ordine di milioni per reti profonde, che possono essere i pesi delle connessioni, stabilite durante l'addestramento.

In una rete questi pesi vengono acquisiti con un apprendimento supervisionato, deve quindi esistere una funzione che dipende dai dati di apprendimento `train` e questi insiemi di parametri Θ. Questa funzione viene chiamata funzione obiettivo:

$$f(\texttt{train}, \Theta) \tag{4.3.1}$$

L'apprendimento consiste nel modificare questi parametri, cercando una loro soluzione ottima Θ^*. Si vuole trovare l'ottimalità della soluzione, da massimizzare, oppure minimizzando l'errore o

la perdita, in base a come viene definita questa funzione f:

$$\Theta^* = \text{argmax}_\Theta f(\texttt{train}, \Theta)$$
$$\Theta^* = \text{argmin}_\Theta f(\texttt{train}, \Theta)$$

Questo si può effettuare esplicitamente con metodi matematici, a partire dalla sua definizione, con tecniche di "Gradient-Descent", calcolando le derivate parziali rispetto ai vari parametri e ponendo questo gradiente uguale a zero, risolvendo per i parametri. Altrimenti si può calcolare implicitamente utilizzando delle euristiche che modificano questi parametri, in modo coerente con la funzione f. Uno di questi approccio è il clustering con algoritmo k-means.

Questi parametri sono legati ad un approccio geometrico e forniscono informazioni sulla regolarità della superficie di separazione. Trovando una superficie di separazione ottima, si è in grado di discriminare al meglio le classi.

4.3.2 Iperparametri

Nelle reti neurali a monte di questi parametri devono essere definiti altri parametri chiamati iperparametri H, e devono essere scelta a priori, prima ancora di realizzare il modello. Questi iperparametri possono esser il numero di neuroni in una rete neurale, il valore k, nell'algoritmo k-NN, un polinomio per la regressione lineare o loss function da minimizzare. Questi vengono scelti quando viene realizzato il modello. Dopo aver scelto questi parametri si avvia il training vero e proprio. Si segue un approccio esaustivo per sceglierli, secondo l'idea di analizzare tutte le possibili combinazioni tra vari iperparametri in base alle loro prestazioni.

Si considera un intervallo all'interno del quale possono essere presenti questi iperparametri, e se sono presenti molti iperparametri e questo intervallo scelto per ognuno di essi è anche molto esteso il problema di individuare una loro combinazione può crescere in maniera esponenziale, rendendo il problema ingestibile sotto un punto di vista pratico.

Si utilizzano quindi tecniche come la ricerca casuale, o altre tecniche di ottimizzazione probabilistiche come l'ottimizzazione Bayesiana, che utilizza il teorema di Bayes, oppure tecniche di "Evolutionary Computing", basate sull'evoluzione. Quest'ultime tecniche si ispirano al modello di evoluzione e di selezione naturale, dove la popolazione migliore di iperparametri è quella in grado di adattarsi al meglio all'ambiente.

Algoritmi genetici erano molto diffusi una ventina di anni fa, anche se recentemente stanno venendo utilizzati più spesso.

Bisogna valutare le prestazioni di questi iperparametri, in caso di classificazione si potrebbe lavorare direttamente con la funzione da ottimizzare. Ma generalmente si vuole lavorare sulla semantica del problema, quindi l'accuratezza su certi insiemi di dati. Si rappresenta l'accuratezza in termini percentuali.

In problemi di classificazione con cardinalità delle classi molto sbilanciate, si potrebbe raggiungere una percentuale di accuratezza vicina al 100% senza aver mai attribuito dei datapoint alle classi più rare. Questo è un classificatore che funziona solamente su questo particolare dominio, ma su un qualsiasi altro dominio dove non è presente questa distribuzione di classi l'agente avrà un'accuratezza molto minore. Esistono diverse tecniche per gestire questo fenomeno, nei problemi di regressione si utilizza una metrica chiamata "Root Mean Squared Error", RMSE, molto utilizzata

nei sistemi di raccomandazione. Per ogni oggetto prova a predire la valutazione di un utente su un oggetto o un servizio che ancora non ha acquistato. Questa metrica valuta la distanza tra questo valore predetto ed il valore reale. Si potrebbe considerare anche la media tra i vari valori. Ma il valore predetto potrebbe essere positivo o negativo, quindi si considera il quadrato per compensarlo.

$$\text{RMSE} = \sqrt{\frac{1}{2}\sum_{i=1}^{N}(\texttt{pred}_i - \texttt{true}_i)^2} \tag{4.3.2}$$

4.3.3 Matrice di Confusione

Un'altra modalità per valutare le prestazioni di un classificatore è la matrice di confusione, una heatmap realizzata operando su un dataset con un certo numero n di classi. La matrice $n \times n$, presente sulle righe i valori reali e sulle colonne i valori predetti. Un classificatore sempre accurato avrebbe valori non nulli presenti solamente sulle diagonali, quindi avrebbe un'esatta corrispondenza tra i valori predetti e quelli attuali. Nella realtà su ogni colonna è presente un valore percentuale che indica la probabilità secondo cui quell'elemento viene attribuito ad una determinata classe.

4.3.4 Problemi Closed e Open Set

Un'altra differenza da analizzare è se i problemi sono closed o open set. Nei sistemi closed set, l'insieme delle classi è predefinito ed immutabile, quindi ogni pattern appartiene sicuramente ad una di queste classi note. In molti casi reali invece è possibile che questi pattern possono appartenere ad una delle classi note o nessuna di queste. Per risolvere questo problema è possibile aggiungere classi fittizie al problema, e si aggiungono training set con esempi negativi, ovvero appartenenti a classi fittizie. Altrimenti è possibile aggiungere valori soglia, e si assegna un pattern alla classe più probabile, solo quando la probabilità è maggiore di questo valore di confidenza. Se la soglia è molto alta si avranno tanti falsi positivi, analogamente se fosse bassa ci sarebbero tanti falsi negativi. Un sistema di questo tipo deve poter confrontare la probabilità p che un pattern appartenga alla classe rispetto al valore di soglia t:

$$p > t$$

4.3.5 Sistemi con Soglia

Sulla base di questo valore di soglia si definisce un classificatore binario per decidere se assegnare al pattern corrente un valore positivo o negativo. Sono possibili quattro situazioni nel tentativo di classificare un pattern:

- Vero Positivo (TP);
- Vero Negativo (TN);
- Falso Positivo (FP), detto anche errore di tipo 1, o False;
- Falso Negativo (FN), detto anche errore di tipo 2, o Miss.

In base all'applicazione, è necessario impostare la soglia, valutando se sono preferibili falsi positivi o falsi negativi. Nel settore dell'assistenza sanitaria, si vuole minimizzare i falsi negativi, per una data patologia, poiché si vuole minimizzare il numero di paziente che non vengono curati. Mentre i falsi positivi non pongono problemi, poiché non presentano la patologia in questione, ed alle analisi successive risulteranno negativi. Mentre i falsi negativi non verranno sottoposti a queste ulteriori analisi. Ad oggi modelli di ML sono molto comuni in questo settore esattamente per questo scopo.

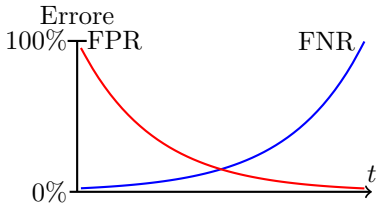
Si possono calcolare diversi valori percentuali sulla probabilità di:

- Veri positivi (TPR): $\frac{\text{TP}}{\text{P}}$;
- Veri negativi (TNR): $\frac{\text{TN}}{\text{N}}$;
- Falsi positivi (FPR): $\frac{\text{FP}}{\text{P}}$;
- Falsi negativi (FNR): $\frac{\text{FN}}{\text{N}}$;

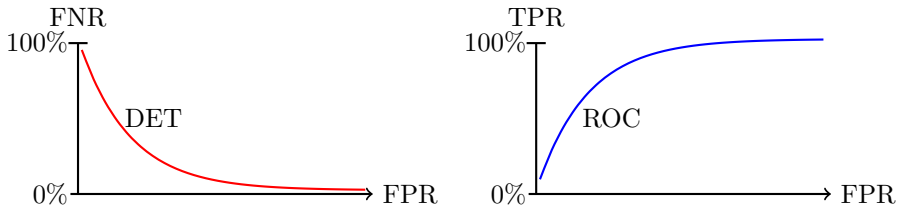
L'accuratezza si calcola come la percentuale delle scelte positivi effettuate:

$$\text{Acc.} = \frac{\text{TP} + \text{TN}}{\text{P} + \text{N}} \tag{4.3.3}$$

Nei sistemi con soglia si può misurare l'andamento dei valori di errore FNR e FPR rispetto al valore della soglia. Si nota come il all'aumentare della soglia diminuiscono i falsi positivi ed aumentano i falsi negativi, al contrario per la diminuzione della soglia.



Queste due curve possono essere espresse tramite le curve DET, "Detection Error Tradeoff", definita in maniera implicita del valore di soglia, dove su un asse sono presenti i falsi negativi e sull'altro asse i falsi positivi. Ogni valore della curva rappresenta quindi un valore di soglia. Una curva più utilizzata è la ROC, "Receiver Operating Characteristic", proposta nella seconda guerra mondiale dagli ingegneri elettrotecnici che lavorano ai radar, per riconoscere i nemici nelle battaglie aeree. Il punto ideal della curva ROC è un TPR pari ad uno ed un FPR pari a zero.



Sulla base della curva ROC per valutare se un classificatore è meglio di un altro si determina il suo integrale sotto la curva (AUC). Questo è uno scalare nell'intervallo $[0, 1]$. Questo determina la prestazione media del classificatore, può essere calcolato come un integrale numerico. Un classificatore casuale ha una percentuale di falsi positivi pari al 50%, come la percentuale di falsi negativi, quindi la curva ROC è una bisettrice del piano, con un AUC pari a 0.5.

4.4 Regressione

I modelli a regressione sono molto utilizzati per prevedere variabili su una scala continua, utili per risolvere problemi in diversi ambiti per analizzare variabili, predire andamenti, etc. Con questi metodi bisogna addestrare un agente a prevedere il valore una variabile target, a differenza della classificazione dove viene predetta una certa categoria.

4.4.1 Simple Regression

Il modello a regressione lineare semplice, chiamata anche univariata. Consiste nel provare le relazioni esistenti tra una variabile descrittiva x e la variabile target y.

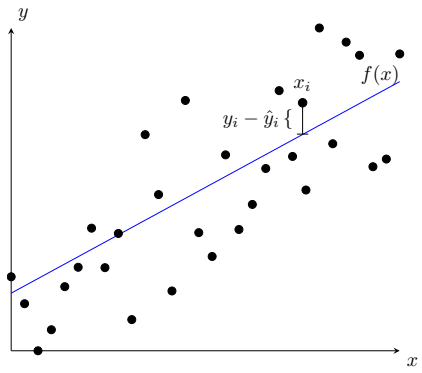
Per prevedere un certo valore bisogna effettuare prima delle misurazioni o osservazioni sul dataset su cui si lavora. Queste osservazioni verranno utilizzate per addestrare il sistema, e dovrà essere in grado di prevedere il valore della variabile y, per una variabile descrittiva non presente nel dataset di addestramento. Si tratta quindi di un modello di apprendimento supervisionato. Ogni punto può essere rappresentato su di un piano cartesiano.

Il primo passo necessario è definire l'approssimazione di questa funzione incognita chiamata ipotesi, definendo lo spazio delle ipotesi H, definendo un insieme di polinomi fino ad un grado massimo k. In questo spazio si vuole definire la funzione $f(x)$, di ipotesi che approssima al meglio le osservazioni. Queste seguono una funzione sconosciuta al modello.

Non è facile stabilire data un'ipotesi f se questa si tratta di una buona approssimazione, della funzione sconosciuta. Una buona ipotesi verrà generalizzata bene, prevedendo correttamente esempi che non ha ancora incontrato.

Nel modello Simple Linear Regression Model, si vuole utilizzare un modello lineare per la funzione f, dove per ogni valore dell'ascissa, si ha un valore vero y_i, ed un valore previsto dalla funzione $\hat{y}_i = f(x_i)$. Si individua questa retta con due pesi w_0, l'intercetta e w_1 la pendenza della retta:

$$\hat{y}_i = f(x_i) = w_0 + w_1 x_i$$



L'errore di questo valore previsto si definisce per ogni punto x_i come ε_i:

$$y_i - \hat{y}_i = \varepsilon_i \tag{4.4.1}$$
$$y_i = w_0 + w_1 x_i + \varepsilon_i$$

Una volta definita la forma della retta bisogna definire questi due pesi per approssimare la funzione incognita, secondo un certo criterio ancora da stabilire.

Un criterio abbastanza ovvio è quello di individuare due pesi che minimizzano gli errori che si trovano sui vari esempi presenti. Si considera la differenza tra il valore attuale ed il valore previsto, ma se si considerasse solamente la differenza, differenze positive ammortizzerebbero le differenze negative, e quindi non si avrebbe una buona rappresentazione dell'errore. Quindi si considera una funzione RSS, "Residual Sum of Squares", largamente utilizzata, simile alla RMSE, vista in precedenza per l'accuratezza di un classificatore. Quest'approccio in generale si chiama metodo dei minimi quadrati. Su n misurazioni effettuate quindi, si definisce la RSS come:

$$\text{RSS}(w_0, w_1) = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{N} (y_i - (w_0 + w_1 x_i))^2 \tag{4.4.2}$$

Si vuole cercare i valori dei pesi $\hat{w}_0$ e $\hat{w}_1$ che minimizzano la funzione RSS.

Ad ogni iterazione il valore previsto dalla funzione viene confrontato con il valore iniziale, comparando i valori misurati si calcola il nuovo vettore dei pesi $\hat{\mathbf{w}}$, per determinare una nuova funzione lineare f.

Rappresenta la ricerca di un minimo nello spazio dei pesi. Si può utilizzare il concetto di gradiente della funzione RSS, che si può dimostrare essere convessa, quindi esiste un minimo globale associato ad un gradiente nullo.

Data una funzione $g(\mathbf{w}) : \mathbb{R}^2 \to \mathbb{R}$, con $\mathbf{w} = \begin{pmatrix} w_0 & w_1 \end{pmatrix}$, il suo gradiente viene definito come:

$$\nabla g(\mathbf{w}) = \frac{\partial g(\mathbf{w})}{\partial w_0}\mathbf{i} + \frac{\partial g(\mathbf{w})}{\partial w_1}\mathbf{j}$$

Se ci si sposta nella direzione del gradiente ci si sposta nella direzione di massima pendenza. Quindi per minimizzare la funzione di costo bisogna spostarsi nella direzione opposta rispetto al gradiente.

Si calcola quindi il gradiente della funzione RSS:

$$\nabla \text{RSS}(\mathbf{w}) = \begin{bmatrix} \frac{\partial \text{RSS}(\mathbf{w})}{\partial w_0} \\ \frac{\partial \text{RSS}(\mathbf{w})}{\partial w_1} \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial w_0} \left(\sum_{i=1}^{N} (y_i - (w_0 + w_1 x_i))^2 \right) \\ \frac{\partial}{\partial w_1} \left(\sum_{i=1}^{N} (y_i - (w_0 + w_1 x_i))^2 \right) \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{N} \frac{\partial}{\partial w_0} (y_i - (w_0 + w_1 x_i))^2 \\ \sum_{i=1}^{N} \frac{\partial}{\partial w_1} (y_i - (w_0 + w_1 x_i))^2 \end{bmatrix}$$

$$\nabla \text{RSS}(\mathbf{w}) = \begin{bmatrix} -2 \sum_{i=1}^{N} (y_i - (w_0 + w_1 x_i)) \\ -2 \sum_{i=1}^{N} x_i (y_i - (w_0 + w_1 x_i)) \end{bmatrix} = \begin{bmatrix} -2 \sum_{i=1}^{N} (y_i - \hat{y}_i(\mathbf{w})) \\ -2 \sum_{i=1}^{N} x_i (y_i - \hat{y}_i(\mathbf{w})) \end{bmatrix} \tag{4.4.3}$$

A questo punto bisogna risolvere il problema determinano il punto dello spazio degli stati dove il gradiente è nullo. Si può risolvere in due modi, in forma chiusa, uguagliando a zero il gradiente e risolvendo esplicitamente il sistema risultante. Questo approccio è semplice uno spazio bidimensionale, ma all'aumentare delle dimensioni diventa sempre più costoso in termini di risorse di calcolo, quindi non è sempre conveniente. Un altro metodo utilizza l'algoritmo di discesa del gradiente, "Gradient Descent", essenzialmente un Hill-Climbing, sui minimi, analogo all'algoritmo sui massimi già trattato precedentemente 2.6. Questo algoritmo sceglie ad ogni iterazione uno stato adiacente di gradiente minore e lo selezione come nuovo corrente, fino ad un criterio di convergenza, poiché lavorando su valori continui, è molto difficile arrivare al punto di minimo assoluto. Si termina l'algoritmo quindi ad una certa soglia vicina al punto di minimo assoluto, e lo "step size", la distanza attraversata ad ogni passaggio dell'algoritmo.

Si comincia con la forma chiusa:

$$\nabla \text{RSS}(\hat{\mathbf{w}}) = \begin{bmatrix} -2 \sum_{i=1}^{N} (y_i - (\hat{w}_0 + \hat{w}_1 x_i)) \\ -2 \sum_{i=1}^{N} x_i (y_i - (\hat{w}_0 + \hat{w}_1 x_i)) \end{bmatrix} = \mathbf{0}$$

$$\begin{cases} -2 \sum_{i=1}^{N} (y_i - (\hat{w}_0 + \hat{w}_1 x_i)) = \sum_{i=1}^{N} y_i - \sum_{i=1}^{N} \hat{w}_0 - \hat{w}_1 \sum_{i=1}^{N} x_i = 0 \\ -2 \sum_{i=1}^{N} x_i (y_i - (\hat{w}_0 + \hat{w}_1 x_i)) = \sum_{i=1}^{n} x_i y_i - \hat{w}_0 \sum_{i=1}^{n} x_i - \hat{w}_1 \sum_{i=1}^{n} x_i^2 = 0 \end{cases}$$

$N \cdot \hat{w}_0$

$$\begin{cases} \hat{w}_0 = \dfrac{\sum_{i=1}^{N} y_i}{N} - \hat{w}_1 \dfrac{\sum_{i=1}^{N} x_i}{N} \\ \hat{w}_1 = \dfrac{\sum_{i=1}^{n} x_i y_i - \hat{w}_0 \sum_{i=1}^{n} x_i}{\sum_{i=1}^{n} x_i^2} = \dfrac{\displaystyle\sum_{i=1}^{n} x_i y_i - \dfrac{\sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{N}}{\displaystyle\sum_{i=1}^{n} x_i^2 - \dfrac{\left(\sum_{i=1}^{n} x_i\right)^2}{N}} \end{cases}$$

Con l'algoritmo di gradient descent bisogna aggiornare ad ogni iterazione i pesi per spostarci nella direzione opposta al gradiente. Si considera uno step size $\alpha > 0$:

$$\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \alpha \cdot \nabla \mathrm{RSS}\left(\mathbf{w}^{(t)}\right) = \begin{bmatrix} w_0^{(t)} - \alpha \cdot \dfrac{\partial \mathrm{RSS}\left(\mathbf{w}^{(t)}\right)}{\partial w_0} \\ w_1^{(t)} - \alpha \cdot \dfrac{\partial \mathrm{RSS}\left(\mathbf{w}^{(t)}\right)}{\partial w_1} \end{bmatrix}$$

$$\begin{bmatrix} w_0^{(t+1)} \\ w_1^{(t+1)} \end{bmatrix} \leftarrow \begin{bmatrix} w_0^{(t)} + 2\alpha \cdot \displaystyle\sum_{i=1}^{N} \left(y_i - \hat{y}_i\left(\mathbf{w}^{(t)}\right)\right) \\ w_1^{(t)} + 2\alpha \cdot \displaystyle\sum_{i=1}^{N} x_i \left(y_i - \hat{y}_i\left(\mathbf{w}^{(t)}\right)\right) \end{bmatrix} \tag{4.4.4}$$

Ad ogni passo dell'algoritmo bisogna effettuare questi due passaggi. Si definisce un criterio di convergenza, definendo una soglia $\varepsilon > 0$ arbitrariamente piccola. L'algoritmo quindi termina quando la norma del gradiente raggiunge o supera questa soglia:

$$||\nabla \mathrm{RSS}(\mathbf{w}^{(t)})||_2 \leq \varepsilon \tag{4.4.5}$$

Si definisce l'algoritmo quindi come:

$\mathbf{w}^{(1)}$ <- 0
t <- 1
while $||\nabla \mathrm{RSS}(\mathbf{w}^{(t)})||_2 > \varepsilon$
    $w_0^{(t+1)} \leftarrow w_0^{(t)} + 2\alpha \cdot \displaystyle\sum_{i=1}^{N} \left(y_i - \hat{y}_i\left(\mathbf{w}^{(t)}\right)\right)$
    $w_1^{(t+1)} \leftarrow w_1^{(t)} + 2\alpha \cdot \displaystyle\sum_{i=1}^{N} x_i \left(y_i - \hat{y}_i\left(\mathbf{w}^{(t)}\right)\right)$
    t <- t + 1

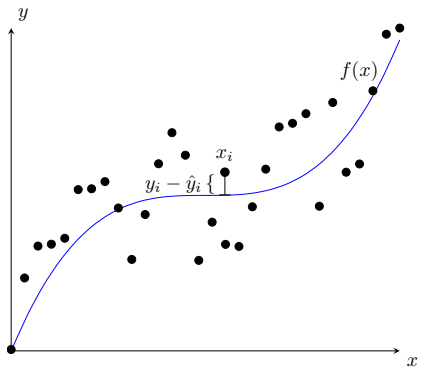
Nella prima iterazione il vettore $\mathbf{w}^{(1)}$ può essere anche inizializzato in modo casuale.

4.4.2 Multiple Regression

Fino ad ora si è ipotizzato un andamento lineare per la funzione f, ma molto spesso questa funzione non rappresenta al meglio le misurazioni ottenute. Si potrebbe quindi pensare di avere una funzione

quadratica o polinomiale:

$$f(x) = w_0 + w_1 x + \cdots w_p x^p$$



$$y_i = \hat{y}_i + \varepsilon_i = w_0 + w_1 x + \cdots w_p x^p + \varepsilon_i \tag{4.4.6}$$

Non necessariamente una funzione che passa su tutti i punti è la migliore, il fenomeno dell'"overfitting", infatti è da evitare. Se un sistema si adatta troppo alle osservazioni si rischia di non generalizzare bene il sistema.

Avendo una funzione polinomiale si parla di Polynomial Regression. In genere ogni potenza della x viene chiamata ciascuna caratteristica o feature:

$$\texttt{feature } i+1 = x^i$$

Si può considerare un caso più generale dove si ha al posto di queste feature una funzione ϕ_j, dipendente dall'input x_i:

$$y_i = w_0\phi_0(x_i) + w_1\phi_1(x_i) + \cdots w_D\phi_D(x_i) + \varepsilon_i$$

$$y_i = \sum_{j=0}^{D} w_j\phi_j(x_i) + \varepsilon_i \tag{4.4.7}$$

Se ci fossero più input in input ci dovrebbe essere non uno scalare x_i, ma un vettore di feature $\mathbf{x}_i$:

$$y_i = \sum_{j=0}^{D} w_j\phi_j(\mathbf{x}_i) + \varepsilon_i \tag{4.4.8}$$

Tutte le funzioni ϕ_j prendono quindi come input un vettore $\mathbf{x}_i$.

Per addestrare il sistema, analogamente alla regressione lineare, si può utilizzare la funzione RSS:

$$\text{RSS}(\mathbf{w}) = \sum_{i=1}^{N}(y_i - \hat{y}_i)^2 = \sum_{i=1}^{N}\left(y_i - \sum_{j=0}^{D} w_j\phi_j(\mathbf{x}_i)\right)^2$$

Si può dimostrare che questa funzione di costo RSS è convessa.

Molte volte è comodo rappresentare questa funzione, relativa all'i-esimo valore di y in notazione matriciale:

$$y_i = \sum_{j=0}^{D} w_j \phi_j(\mathbf{x}_i) + \varepsilon_i$$

$$y_i = \begin{bmatrix} w_0 & \cdots & w_D \end{bmatrix} \cdot \begin{bmatrix} \phi_0(\mathbf{x}_i) \\ \vdots \\ \phi_D(\mathbf{x}_i) \end{bmatrix} + \varepsilon_i$$

Quindi dati questi tre vettori:

$$\mathbf{x}_i = \begin{bmatrix} x_{i,1} \\ \vdots \\ x_{i,d} \end{bmatrix} \quad \mathbf{w}_i = \begin{bmatrix} w_0 \\ \vdots \\ w_D \end{bmatrix} \quad \phi(\mathbf{x}_i) = \begin{bmatrix} \phi_0(\mathbf{x}_i) \\ \vdots \\ \phi_D(\mathbf{x}_i) \end{bmatrix}$$

Si può definire il valore i-esimo di y come:

$$y_i = \mathbf{w}^T \cdot \boldsymbol{\phi}(\mathbf{x}_i) + \varepsilon_i = \phi(\mathbf{x_i})^T \cdot \mathbf{w} + \varepsilon_i$$

Si possono quindi rappresentare tutte le osservazioni $\mathbf{y}$ in modo compatto come:

$$\mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix} = \begin{bmatrix} \phi_0(\mathbf{x}_1) & \cdots & \phi_D(\mathbf{x}_1) \\ \vdots & \ddots & \vdots \\ \phi_0(\mathbf{x}_N) & \cdots & \phi_D(\mathbf{x}_N) \end{bmatrix} \cdots \begin{bmatrix} w_0 \\ \vdots \\ w_D \end{bmatrix} + \begin{bmatrix} \varepsilon_1 \\ \vdots \\ \varepsilon N \end{bmatrix}$$

$$\mathbf{y} = \boldsymbol{\Phi} \cdot \mathbf{w} + \boldsymbol{\varepsilon} \tag{4.4.9}$$

Le righe di $\boldsymbol{\Phi}$ corrisponde ad un'osservazione, mentre le colonne corrispondono alle specifiche features. Definiti questi vettori si può esprimere allora la funzione di valutazione RSS come:

$$\text{RSS}(\mathbf{w}) = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{N} \varepsilon_i^2 = \boldsymbol{\varepsilon}^T \cdot \boldsymbol{\varepsilon} \tag{4.4.10}$$

Si può ricavare il vettore $\boldsymbol{\varepsilon}$, dalla forma compatta delle osservazioni $\mathbf{y}$:

$$\mathbf{y} = \boldsymbol{\Phi} \cdot \mathbf{w} + \boldsymbol{\varepsilon} \rightarrow \boldsymbol{\varepsilon} = \mathbf{y} - \boldsymbol{\Phi} \cdot \mathbf{w}$$

$$\text{RSS}(\mathbf{w}) = (\mathbf{y} - \boldsymbol{\Phi} \cdot \mathbf{w})^T \cdot (\mathbf{y} - \boldsymbol{\Phi} \cdot \mathbf{w}) \tag{4.4.11}$$

Si è ottenuta la funzione RSS in forma matriciale. Si può quindi rappresentare il gradiente mediante regole del calcolo differenziale matriciale come:

$$\nabla \text{RSS}(\mathbf{w}) = \nabla \left[(\mathbf{y} - \boldsymbol{\Phi} \cdot \mathbf{w})^T \cdot (\mathbf{y} - \boldsymbol{\Phi} \cdot \mathbf{w}) \right] = -2\boldsymbol{\Phi}^T (\mathbf{y} - \boldsymbol{\Phi}\mathbf{w}) \tag{4.4.12}$$

Per minimizzare la funzione di costo si può utilizzare la forma chiusa oppure il gradient descent, definendo lo step size ed il criterio di convergenza. In forma chiusa si può esprimere facilmente, data la rappresentazione matriciale:

$$\begin{aligned}
\nabla \text{RSS}(\hat{\mathbf{w}}) &= -2\boldsymbol{\Phi}^T(\mathbf{y} - \boldsymbol{\Phi}\hat{\mathbf{w}}) = \mathbf{0} \\
-2\boldsymbol{\Phi}^T\mathbf{y} + 2\boldsymbol{\Phi}^T\boldsymbol{\Phi}\hat{\mathbf{w}} &= \mathbf{0} \\
\cancel{2}\boldsymbol{\Phi}^T\boldsymbol{\Phi}\hat{\mathbf{w}} &= \cancel{2}\boldsymbol{\Phi}^T\mathbf{y} \\
\cancel{(\boldsymbol{\Phi}^T\boldsymbol{\Phi})^{-1}}\cancel{(\boldsymbol{\Phi}^T\boldsymbol{\Phi})}\hat{\mathbf{w}} &= (\boldsymbol{\Phi}^T\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^T\mathbf{y} \qquad (\mathbf{I}) \\
\hat{\mathbf{w}} &= (\boldsymbol{\Phi}^T\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^T\mathbf{y}
\end{aligned} \tag{4.4.13}$$

Per quanto riguarda l'algoritmo gradient descent, è analogo alla regressione lineare, definendo allo stesso modo un passo α:

$$\begin{bmatrix} w_0^{(t+1)} \\ \vdots \\ w_D^{(t+1)} \end{bmatrix} \leftarrow \begin{bmatrix} w_0^{(t)} - \alpha \cdot \dfrac{\partial \text{RSS}\left(\mathbf{w}^{(t)}\right)}{\partial w_0} \\ \vdots \\ w_D^{(t)} - \alpha \cdot \dfrac{\partial \text{RSS}\left(\mathbf{w}^{(t)}\right)}{\partial w_D} \end{bmatrix}$$

I singoli pesi vengono calcolati analogamente alla regressione lineare, ciò che cambia sono le derivate parziali. Rispetto ad un generico peso j-esimo la derivata parziale è:

$$\begin{aligned}
\frac{\partial \text{RSS}(\mathbf{w}^{(t)})}{\partial w_j} = \sum_{i=1}^{N} 2\left(y_i - \hat{y}_i(\mathbf{w}^{(t)})\right) \cdot -\frac{\partial \hat{y}_i\left(\mathbf{w}^{(t)}\right)}{\partial w_j} &= 2\sum_{i=1}^{N}\left(y_i - \hat{y}_i(\mathbf{w}^{(t)})\right) \cdot -\phi_j(\mathbf{x}_i) \\
\frac{\partial \text{RSS}(\mathbf{w}^{(t)})}{\partial w_j} &= -2\sum_{i=1}^{N}\phi_j(\mathbf{x}_i)\left(y_i - \hat{y}_i(\mathbf{w}^{(t)})\right)
\end{aligned} \tag{4.4.14}$$

Il criterio di convergenza è analogo, poiché si ha una stessa funzione convessa, si decide un valore di soglia $\varepsilon > 0$ arbitrario tale che la norma della funzione RSS è sempre maggiore di questo valore per continuare le iterazioni:

$$||\nabla \text{RSS}(\mathbf{w}^{(t)})||_2 \leq \varepsilon \tag{4.4.15}$$

Si definisce l'algoritmo quindi come:

$\mathbf{w}^{(1)}$ <- 0
t <- 1
while $||\nabla \text{RSS}(\mathbf{w}^{(t)})||_2 > \varepsilon$
    for $j = 0, \cdots, D$
        $w_j^{(t+1)} \leftarrow w_j^{(t)} + 2\alpha \cdot \sum_{i=1}^{N}\phi_j(\mathbf{x}_i)\left(y_i - \hat{y}_i(\mathbf{w}^{(t)})\right)$
    t <- t + 1

4.4.3 Workflow

Le due task da affrontare per attuare una regressione sono la scelta del modello e la valutazione del modello. Nella selezione del modello bisogna scegliere dei parametri di tuning λ per controllare il modello. Dopo averlo scelto bisogna effettuare una valutazione del "Generalization Error" o "True Error". Quindi bisogna avere un ulteriore insieme di dati di osservazione, non appartenenti al training set, su cui puoi essere valutato questo modello. L'errore dei dati di test dovrebbe avvicinarci il più possibile all'errore vero, non noto. Se fosse possibile determinare l'errore vero, allora non servirebbe creare un modello per predire il valore y.

Si potrebbe operare con un approccio intuitivo, ma sbagliato, dove per ogni modello di complessità λ si stimano dei pesi $\hat{\mathbf{w}}_\lambda$, valutando le prestazioni sui dati di test, e scegliendo i parametri λ^* che ottengono il più basso test error. Per la valutazione del modello si considera il test error calcolato sul $\hat{\mathbf{w}}_{\lambda^*}$. Questo approccio ha un difetto intrinseco, poiché determina i valori dei parametri λ^* delle ipotesi non solo sui valori di training ma anche con i valori del test set, quindi non sarà possibile giudicare in maniera adeguata il modello. Questo fenomeno nell'area dell'IA si chiama "Peaking", dove l'ipotesi viene selezionata in base alle sue prestazioni sull'insieme di test.

L'informazione di test avrebbe dovuto rimanere confinata nel test, ma si è infiltrata nell'algoritmo di apprendimento. Una soluzione consiste nell'utilizzare tre diversi set, se ciò è permesso dal numero di osservazioni, creando un ulteriore validation set. Se si risparmiano osservazioni sul training set, allora non si crea un modello adeguato, se si risparmia sul training set allora non viene valutato adeguatamente il modello. Nella fase di addestramento invece di addestrare il sistema sul test set, si valuta sul validation set. Quindi si determina l'ipotesi migliore su questo set e si valuta sul test set. Non è presente una regola generale per dividere i dati, in generale il training set comporta la maggior percentuale, mentre il training ed il set test hanno all'incirca la stessa percentuale delle osservazioni totali.

4.5 Nozioni di Matematica

4.5.1 Funzioni Convesse

Un insieme C in uno spazio vettoriale $V^k(O)$, si dice convesso se per ogni due punti $\mathbf{v}, \mathbf{w} \in C$, il segmento che unisce i due punti appartiene anch'esso all'insieme C:

$$\forall \mathbf{v},\, \mathbf{w} \in C, ,\, \forall \lambda \in [0,1] \text{ t.c. } \lambda\mathbf{v} + (1-\lambda)\mathbf{w} \in C \implies C: \text{ Convesso} \tag{4.5.1}$$

Ovvero tutti i punti appartenenti a questo segmento appartengono all'insieme C.

Nel caso bidimensionale si può verificare graficamente questa condizione, considerando un insieme C convesso, una retta: Si può esprimere l'espressione come:

$$\lambda\mathbf{v} + (1-\lambda)\mathbf{w} = \mathbf{w} + \lambda(\mathbf{v} - \mathbf{w})$$

Per ogni valore di $\lambda \in [0,1]$, quindi si ha che questo punto appartiene all'insieme C, poiché si ottiene traslando il vettore $\mathbf{w}$, certamente appartenente all'insieme, di un fattore ottenuto dalla differenza con $\mathbf{v}$, quindi anche questo dovrà appartenere all'insieme. Si può osservare come questo rappresenti una traslazione di un punto già presente nell'insieme lungo la retta C.

Una funzione $g : C \to \mathbb{R}$ si dice convessa, se per ogni coppia di punti appartenenti all'insieme C, $\mathbf{v}, \mathbf{w} \in C$, la funzione g nell'intervallo tra questi due punti è sempre al di sotto del segmento generato collegando i punti $g(\mathbf{v})$ e $g(\mathbf{w})$ sulla funzione:

$$\forall \mathbf{v},\ \mathbf{w} \in C,\ \lambda \in [0,1] \text{ t.c. } g(\lambda\mathbf{v} - (1-\lambda)\mathbf{w}) \leq \lambda g(\mathbf{v}) + (1-\lambda)g(\mathbf{w}) \implies g : \text{Convessa} \tag{4.5.2}$$

Si può verificare graficamente data una funzione $g : \mathbb{R} \to \mathbb{R}$, con $C \equiv \mathbb{R}$, quindi si considera ogni coppia di scalari $v, w \in \mathbb{R}$, e si suppone che l'insieme $\mathbb{R}$ sia convesso.

Per una funzione convessa $g(\mathbf{w})$ differenziabile, per ogni punto dell'insieme di definizione $\mathbb{D}$, il piano tangente a questo punto della funzione giace sempre al di sotto della funzione:

$$\forall \mathbf{v}, \mathbf{w} \in \mathbb{D} \implies (\mathbf{w}) \geq g(\mathbf{w}) + \nabla g(\mathbf{v})^T(\mathbf{w} - \mathbf{v})$$

4.5.2 Derivate Parziali

Si considera una funzione g definita su un campo A, di due variabili w_0 e w_1. Si considera un punto $\overline{P}$ appartenente ad A, di coordinate $(\overline{w}_0, \overline{w}_1)$. Esiste allora un interno circolare di centro $\overline{P}$ e opportuno raggio σ, contenuto in A:

$$0 < |\Delta w_0| < \sigma$$

Si ha $(\overline{w}_0 + \Delta w_0, \overline{w}_1) \in A$, e si può calcolare il suo rapporto incrementale parziale rispetto alla variabile w_0:

$$\frac{g(\overline{w}_0 + \Delta w_0, \overline{w}_1) - g(\overline{w}_0, \overline{w}_1)}{\Delta w_0}$$

Il limite di questo rapporto incrementale si definisce derivata parziale rispetto a w_0, se esiste la funzione g si dice parzialmente derivabile rispetto ad w_0 nel punto di $\overline{P}$:

$$\lim_{\Delta w_0 \to 0} \frac{g(\overline{w}_0 + \Delta w_0, \overline{w}_1) - g(\overline{w}_0, \overline{w}_1)}{\Delta w_0}$$

Se questa funzione g è parzialmente derivabile non solo in questo punto, ma su ogni punto nel campo A dov'è definita, ed assume per ogni punto un valore definito e determinato, allora si definisce la derivata parziale di g rispetto a w_0, sul campo A di due variabili w_0 e w_1:

$$\frac{\partial g}{\partial w_0} \tag{4.5.3}$$

Analogamente si può definire il limite del rapporto incrementale rispetto a w_1, sul punto $\overline{P}$, se questo rapporto incrementale ha un limite ben definito su ogni punto del campo A, allora si può definire analogamente la derivata parziale di g rispetto a w_1, sul campo A di due variabili w_0 e w_1:

$$\lim_{\Delta w_1 \to 0} \frac{g(\overline{w}_1, \overline{w}_1 + \Delta w_1) - g(\overline{w}_0, \overline{w}_1)}{\Delta w_1}$$

$$\frac{\partial g}{\partial w_1}$$

Per funzioni a più di una variabile però la continuità in un punto non dipende dalla derivabilità della funzione. La funzione g può quindi essere derivabile rispetto ad entrambe le sue variabili nel punto $\overline{P}$, ma essere comunque discontinua nel stesso punto.

Queste derivate parziali si possono definire analogamente per funzioni di più variabili $g(\mathbf{w})$.

4.5.3 Gradiente di una Funzione

Il gradiente si può considerare la generalizzazione del concetto di derivata, per variabili a più variabili. Si considera una funzione $g(\mathbf{w})$ di n variabili, allora si definisce il gradiente il vettore le cui componenti sono le derivate parziali della funzione:

$$\nabla g(\mathbf{w}) = \begin{bmatrix} \dfrac{\partial g(\mathbf{w})}{\partial w_0} \\ \vdots \\ \dfrac{\partial g(\mathbf{w})}{\partial w_n} \end{bmatrix} \tag{4.5.4}$$

Si considera un caso bidimensionale con $g(\mathbf{w}) = g(w_0, w_1)$. Si considera quindi una retta n sul piano w_0, w_1, la cui intersezione con il gradiente genera due angoli i cui coseni sono rispettivamente α con l'asse w_0 e β con l'asse w_1.

Si considera un punto P di coordinate iniziali (w_0, w_1), se ci si sposta di una distanza ρ su questa retta n, si raggiungerà un punto Q di coordinate $(w_0 + \alpha\rho, w_1 + \beta\rho)$. Si considera il rapporto incrementale dello spostamento effettuato su questa retta n partendo dal punto P. Il limite di questo rapporto incrementale, se determinato e finito viene chiamato derivata rispetto ad una direzione n della funzione g.

Si può dimostrare che questa derivata direzionale gode delle seguenti proprietà:

$$\frac{\partial g(\mathbf{w})}{\partial n} = \alpha \cdot \frac{\partial g(\mathbf{w})}{\partial w_0} + \beta \cdot \frac{\partial g(\mathbf{w})}{\partial w_1} \tag{4.5.5}$$

Definito $\hat{vectn}$ il versore della direzione n, allora la derivata direzionale si può esprimere come il prodotto scalare tra questo versore ed il gradiente della funzione g:

$$\nabla g(w_0, w_1) \cdot \hat{\mathbf{n}} = \begin{bmatrix} \dfrac{\partial g(\mathbf{w})}{\partial w_0} \\ \dfrac{\partial g(\mathbf{w})}{\partial w_1} \end{bmatrix} \cdot \begin{bmatrix} \alpha & \beta \end{bmatrix} = \alpha \cdot \frac{\partial g(\mathbf{w})}{\partial w_0} + \beta \cdot \frac{\partial g(\mathbf{w})}{\partial w_1}$$

La derivata direzionale di una funzione g su una retta n rappresenta una proiezione del gradiente ∇g sulla retta n. Ed è massima quando la direzione del gradiente coincide con la direzione della retta n, e minima quando le due direzioni sono tra di loro ortogonali.

4.5.4 Algoritmo di Gradient Descent

La proprietà del gradiente di fornire la direzione di pendenza più ripida permette di realizzare due algoritmi di ricerca duali su spazi continui, di salita l'algoritmo di Hill-Climbing, per la ricerca di massimi, ed algoritmi di Gradient Descent, per la ricerca di minimi.

Questi algoritmi partono da uno stato iniziale $\mathbf{w}^{(t)}$ e si spostano nella stessa o opposta direzione del gradiente, di un passo α, per passare ad un nuovo stato $w^{(t+1)}$, cercando di arrivare ad un punto di massimo o di minimo, oppure un punto molto vicino.

$\mathbf{w}^{(1)}$ <- $\mathbf{0}$
t <- 1
while $||\nabla g(\mathbf{w}^{(t)})||_2 > \varepsilon$
    $\mathbf{w}^{(t+1)} \rightarrow \mathbf{w}^{(t)} \pm \alpha \nabla g(\mathbf{w}^{(t)})$
    t <- t + 1

Lo stato iniziale può essere scelto casualmente oppure assegnato al vettore nullo. Si definisce una soglia, sopra la quale l'algoritmo continua la sua discesa, o salita. Questo algoritmo funziona molto bene su funzioni convesse, invece su funzioni non convesse, dove si trovano dei punti di minimo locali, allora riscontra gli stessi problemi discussi per l'algoritmo di Hill-Climbing, precedentemente.

4.5.5 Calcolo di Probabilità

Variabili aleatorie sono quantità di interesse determinate dal risultato di un esperimento casuale, ovvero non determinabile a priori con certezza. Per cui è possibile assegnare una probabilità ai suoi possibili valori.

Il valore atteso è uno dei concetti più importanti del campo della statistica. Data una variabile aleatoria discreta X, che può assumere i valori $x_1, \cdots, x_N$, si definisce il valore atteso come la media pesata di questi valori, dove i pesi sono la probabilità di ognuno di questi valori:

$$E[X] \triangleq \sum_{i=1}^{N} (x_i P(X = x_i)) \tag{4.5.6}$$

Si chiama anche media di X, oppure "Expectation". Anche se viene chiamato valore atteso di X, non rappresenta uno dei possibili valori della variabile aleatoria X, ma rappresenta il limite a cui tende il valore della variabile su un numero infinito di misurazioni.

Si definisce l'"Indicator Function" di un evento A come:

$$I[A] \triangleq \begin{cases} 1 & \text{Se } A \text{ si verifica} \\ 0 & \text{Se } A \text{ non si verifica} \end{cases}$$

Il suo valore atteso è quindi:

$$E[I] = 1 \cdot P(I = 1) + \cancelto{0}{0 \cdot P(I = 0)} = P(A)$$

Il valore atteso di un Indicator Function è la probabilità stessa dell'evento A corrispondente.

Per la funzione valore atteso valgono le seguenti proprietà:

$$E[a+b] = E[a] + E[b]$$
$$E[k \cdot a] = k \cdot E[a]$$
$$E[a \cdot b] = E[a] \cdot E[b]$$

Dove k è costante ed a e b sono due eventi indipendenti, nell'ultima proprietà.
Si definisce la varianza di una funzione aleatoria X come:

$$\sigma_x^2 = \mathrm{Var}(X) \triangleq E[(X - \mu_x)^2] \tag{4.5.7}$$

Si può esprimere alternativamente come:

$$\mathrm{Var}(X) \triangleq E[(X-\mu_x)^2] = E[X^2 - 2\mu_x X + \mu_x^2] = E[X^2] - 2\mu \cdot \cancelto{\mu_x}{E[X]} + \mu_x^2 \cdot \cancel{E[1]}$$
$$\mathrm{Var}(X) \triangleq E[X^2] - 2\mu_x^2 + \mu_x^2 = E[X^2] - \mu_x^2 = E[X^2] - E[X]^2 \tag{4.5.8}$$

4.6 Valutatone delle Prestazioni

La valutazione delle prestazioni di un sistema è estremamente cruciale, ed i criteri di valutazione permettono di scegliere modelli migliori, per cui è necessario definire delle metriche per poter valutare questi diversi modelli, in questo caso di regressione, ma si utilizzano anche per altri tipi di sistemi. Si definisce quindi una funzione di "Loss", che definisce quanto si perde utilizzano previsione del modello $f_{\hat{\mathbf{w}}}(\mathbf{x})$, dopo essere stato addestrato, invece che il valore reale $\hat{y}$:

$$L[\hat{y}, f_{\hat{\mathbf{w}}}(\mathbf{x})] \tag{4.6.1}$$

Questa funzione L rappresenta il costo che si ha con la funzione f, con il vettore dei pesi $\hat{\mathbf{w}}$, su di un input $\mathbf{x}$.

Può essere definita come errore assoluto:

$$L[\hat{y}, f_{\hat{\mathbf{w}}}(\mathbf{x})] = |\hat{y} - f_{\hat{\mathbf{w}}}(\mathbf{x})|$$

Oppure come errore quadratico:

$$L[\hat{y}, f_{\hat{\mathbf{w}}}(\mathbf{x})] = (\hat{y} - f_{\hat{\mathbf{w}}}(\mathbf{x}))^2$$

4.6.1 Training, Generalization e Test Error

Ai fini della valutazione del "loss", bisogna definire il Training Error, l'errore di un sistema addestrato minimizzando l'errore sul training set. Il Generalization Error o True Error, è l'errore sul valore effettivo, ma non è possibile calcolarlo, poiché richiederebbe di conoscere a priori tutti i possibili valori e associazioni del problema. Il Test Error si utilizza per approssimare il True Error, valutando il modello sul test set.

Si considerano gli andamenti di questi errori, ed il problema dell'"Over-Fitting", non solo su modelli di ML, ma anche in generale.

Dato un insieme di dati non bisogna utilizzarle tutte per allenare il modello, bisogna dividere quest'insieme in un insieme di training ed uno di testing. Si sceglie quindi un modello e si calcolano i pesi $\mathbf{w}$ da minimizzare con la funzione RSS. Una volta addestrato il modello si può calcolare il training error, definendo la funzione di Loss. Definita questa funzione si può calcolare il training errore come l'errore medio definito sugli N punti dell'insieme di training:

$$\text{Training Error} = \frac{1}{N} \cdot \sum_{i=1}^{N} L[y_i, f_{\hat{\mathbf{w}}}(\mathbf{x}_i)] \tag{4.6.2}$$

Si considera come cambia questo errore rispetto alla complessità del modello, partendo da un modello costante, ed aumentando il grado della funzione $f(x)$.

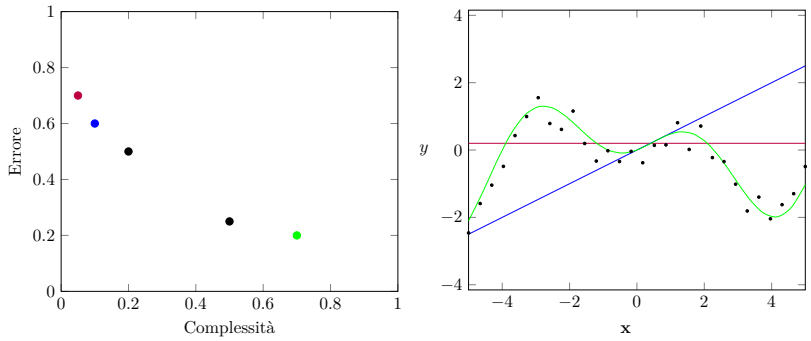


Fig. 1: Training Error Rispetto a Complessità del Modello

Intuitivamente passano da un modello costante ad un modello lineare l'errore diminuisce, questo continua fino ad un modello complesso, che potrebbe passare su tutti i datapoint. Ma il Training Errore non è una buona metrica, poiché è troppo ottimistico. Non rappresenta una buona misura delle prestazioni di previsione del sistema, infatti un modello polinomiale potrebbe essere molto buono solamente con i dati del training set, mentre risulta poco accurato su altri insiemi di dati.

Per valutare questa caratteristica si considerano tutte le possibili coppie $(\mathbf{x}, y)$ per un certo problema. E l'errore di generalizzazione si calcola come la media della funzione di Loss su tutti questi punti. Ma questo è irrealizzabile, quindi bisogna approssimare questo errore effettivo, avendo un numero limitato di osservazioni.

Bisognerebbe cercare di pesare tutte le coppie $(\mathbf{x}, y)$ in base alla loro probabilità di essere presenti nella zona di interesse, quindi prendendo in considerazione la distribuzione dell'insieme rispetto a $\mathbf{x}$ ed anche la distribuzione del valore y in a parità di $\mathbf{x}$.

Da un punto di vista formale si può definire questo valore avvalendosi del valore atteso:

$$\text{Generalization Error} = E_{\mathbf{x},y}[L(\hat{y}, f_{\hat{\mathbf{w}}}(\mathbf{x}))] \tag{4.6.3}$$

Per valutare i modelli in base a questo errore si considera una rappresentazione del piano $\mathbf{x}, y$ che mostri le diverse gradazioni dei dei vari punti corrispondenti alla distribuzione di probabilità nel data set.

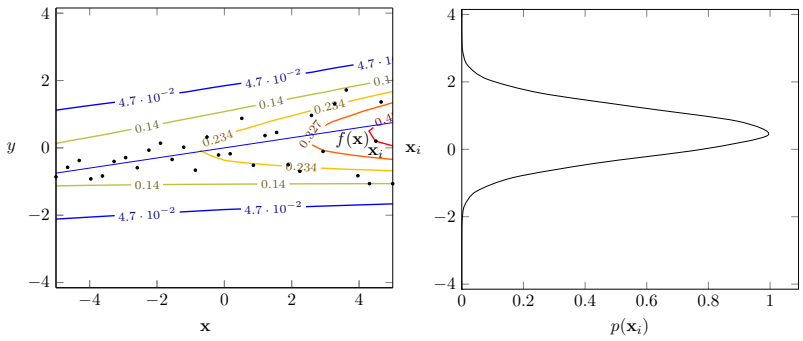


Fig. 2: Data Set con Distribuzione di Probabilità

Bisogna determinare come varia questo errore rispetto alla funzione f e come si adatta alle osservazioni del training set, ed in che modo possa predire i valori di dati non presenti nel training set, pesati rispetti alle loro probabilità.

Analogamente al training error, partendo da un modello costante, aumentando la complessità l'errore diminuisce, ma questo non è un andamento costante, poiché si potrebbe verificare il fenomeno dell'over-fitting, dove il modello segue esattamente solo i dati presenti nel training set, e non è in grado di prevedere altri dati al di fuori di questo insieme. Superata una certa complessità quindi l'errore ricomincia a salire.

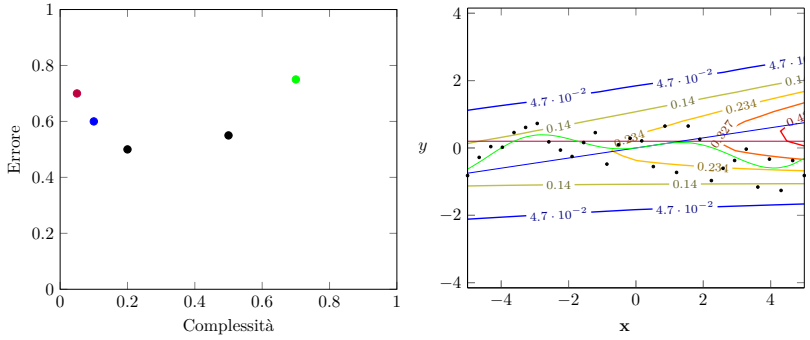


Fig. 3: Generalization Error Rispetto a Complessità del Modello

Non è possibile calcolare il Generalization Error, poiché dovrebbe essere nota la vera distribuzione di probabilità di tutte le coppie $(\mathbf{x}, y)$.

L'ultimo tipo di errore è il Test Error, calcolato come la media della somma del Loss, nei punti nel Test Set:

$$\text{Test Error} = \frac{1}{N_{\text{test}}} \sum_{i \in 1}^{N_{\text{test}}} L[y_i, f_{\hat{\mathbf{w}}}(\mathbf{x}_i)] \tag{4.6.4}$$

Il Test Set deve essere scelto in modo tale da approssimare al meglio il Generalization Error.

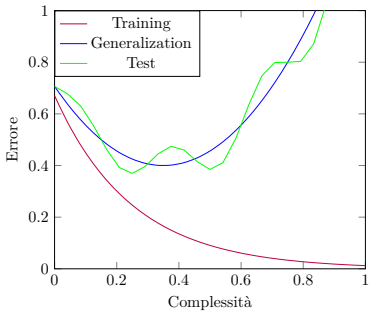


Fig. 4: Confronto Training, Generalization e Test Error

Il fenomeno del sovra-adattamento si può definire in modo formale dati questi errori. Dato un modello con parametri $\hat{\mathbf{w}}$, si ha over-fitting se esiste un modello con parametri stimati $\mathbf{w}'$, tale che il training error del primo modello è minore, mentre il true error del primo modello è maggiore di quello relativo al secondo modello.

4.6.2 Classificatori Binari

Per valutare le prestazioni si possono utilizzare metriche generiche, oppure metriche più consone al task del modello. Nel ML si possono utilizzare metriche molto usate nell'ambito dell'"Information Retrieval", ovvero dei motori di ricerca, come Precision e Recall negli anni '60. Molto spesso utilizzate quando si ha una classificazione binaria, per i motori di ricerca i risultati possono essere determinanti o non determinanti ad una certa query.

La precision e la recall sono delle metriche ancora migliori di altre metriche come il RSS per valutare le prestazioni di un classificatore binario.

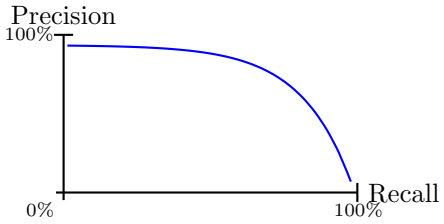
La precision si può definire come la percentuale di risultati rilevanti che sono recuperati:

$$\mathrm{p} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}} \tag{4.6.5}$$

La recall invece si definisce come la percentuale dei risultati recuperati che sono rivelanti:

$$\mathrm{r} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} = \mathrm{TPR} \tag{4.6.6}$$

Il seguente grafico rappresenta l'andamento della precision e della recall per una data soglia, dove ogni punto della curva rappresenta un valore di soglia, di coordinate date dai rispettivi valori di precision e recall:



In un sistema di ML generalmente queste due metriche sono tra di loro inversamente proporzionali. Più che la precision e la recall bisogna considerare un parametro che tiene conto di entrambi chiamato F_1-Score, definito come la media armonica di precision e recall, è quindi il reciproco della media dei reciproci delle due metriche:

$$F_1 = \frac{2}{\frac{1}{\mathrm{p}} + \frac{1}{\mathrm{r}}} = 2 \cdot \frac{\mathrm{p} \cdot \mathrm{r}}{\mathrm{p} + \mathrm{r}} \tag{4.6.7}$$

Per valori di soglia elevati la possibilità di individuare elementi rilevanti diminuisce, mentre se si abbassa la soglia molti più elementi vengono individuati come rilevanti, quindi diminuisce la precisione del sistema.

All'aumentare delle prove, la precision diminuisce mentre la recall aumenta.

La "Specificity" indica quando è accurato il sistema, indica la percentuale di non positivi dichiarati positivi:

$$\mathrm{s} = \frac{\mathrm{TN}}{\mathrm{TN} + \mathrm{FP}} \tag{4.6.8}$$

Mentre la "Sensitivity" coincide alla recall. Molto spesso queste metriche dipendono dall'applicazione.

In modelli di object detection bisogna individuare le istanze di un elemento ed in seguito viene classificato. Per cui le valutazioni devono essere divise in queste due fasi, prima si controlla che individua correttamente un oggetto tramite la sua bounding box, ed in seguito determinare che la classificazione sia corrette. Se fosse un modello di face detection allora sarebbe un classificatore binario.

Si possono definire diverse metriche per questi modelli, per i diversi tipi di errore, come classe errata o bounding box non trovata, o sbagliata. La prima fase coincide con un modello di regressione e può essere analizzata con gli strumenti descritti nelle sezioni precedenti,

- AP, "Average Precision": definita come l'AUC sotto la curva Recall-Precision per oggetti della singola classe:

$$\mathrm{AP} = \int_0^1 \mathrm{p(r)dr} \tag{4.6.9}$$

- mAP, mean AP: definita come la media delle AP su tutte le classi:

$$\mathrm{mAP} = \frac{1}{N} \sum_{i=1}^{N} \mathrm{AP}_i \tag{4.6.10}$$

- IoU, "Intersection over Union": rappresenta la percentuale della bounding box che si trova all'interno della bounding box definita nell'insieme.

In generale se questo valore di IoU è maggiore di 0.5 si considera la classe giusta, altrimenti si considera errata. Questo è un valore che viene determinato alla creazione del modello per poter calcolare TP e FP, con una certa confidenza.

4.6.3 Selezione Iperparametri

In generale sul training set vengono determinati i parametri Θ, possono essere i pesi delle connessioni di una rete neurale, mentre il validation set si utilizza per definire gli iperparametri H, ma questi devono essere scelti a priori prima di effettuare l'addestramento, per poi cercare i loro valori ottimali.

Per evitare il sovra-adattamento quando il dataset lo consente si effettua la procedura chiamata "k-Fold Cross-Validation", ovvero si divide il set di training in k, in genere 5 o 10, fold, ovvero piccoli dataset mescolati. Si effettua k volte l'addestramento, dove $k-1$ vengono scelti come set di addestramento e l'ultimo come insieme di testing. In questo modo le prestazioni del modello sono la media delle prestazioni su questi k insiemi e si possono valutare più combinazioni di iperparametri H, individuando la combinazione con l'accuratezza migliore. Dopo aver scelto questi iperparametri ottimali si riaddestra il modello sull'intero training set ed a questo punto si possono valutare le prestazioni con il test set.

Se i dataset sono di piccoli dimensione, si può arrivare al caso estremo del "Leave One Out", dove tutti i fold hanno dimensione pari ad uno. Si utilizza come dataset di validation quindi un singolo data point.

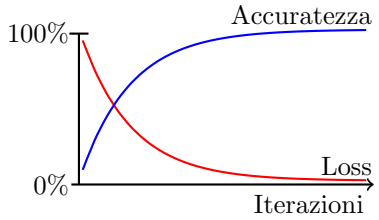
Per effettuare questa procedura bisogna suddividere il dataset, e bisogna mantenere le corrispondenze corrette con le etichette, nonostante gli elementi vengano rimescolati casualmente. Si utilizza la libreria Python Scikit Learn, `sklearn` per effettuare questo processo, tramite le funzioni `shuffling`, `train_test_split` e `cross_val_score`, che valuta da un unico training set k il sistema, restituendo un array con i valori di prestazioni su ognuno dei k addestramenti. In questa libreria sono presenti metodi di classificazione, regressione e clustering. Questa libreria offre la "Support Vector Machine", SVM, che permette di valutare ogni iperparametri singolarmente per poter determinare il loro valore ottimale, utilizzando diversi kernel, sempre offerti dalla libreria.

Gli iperparametri vanno scelti a monte del modello e la ricerca dei valori ottimali può essere laboriosa. Quindi si vuole automatizzare la loro ricerca, il metodo "Grid Search" consiste nell'utilizzo di questa SVM, definendo per ogni iperparametro un insieme di valori da provare. Si può quindi effettuare un'analisi sulle combinazioni di tutti i questi possibili valori per tutti gli iperparametri, operazione molto costosa, poiché presenta dei raffinamenti incrementali delle prestazioni. Il kernel rappresenta il modello della SVM utilizzata, può essere lineare o polinomiale.

Un'altra maniera di operare consiste nella "Random Search" selezionando casualmente le combinazioni di valori degli iperparametri. Altre tecniche sono la Bayesian Optimization, selezionando ulteriori combinazioni da verificare dal punto di vista probabilistico, e la "Evolutionary Optimization", che si basa sul concetto di evoluzione, dove la popolazione più forte che sopravvive consiste nella combinazione migliore di iperparametri. Una sottoclasse consiste negli algoritmi genetici, che verranno trattati approfonditamente nel corso di Intelligenza Artificiale.

4.6.4 Controllare l'Overfitting

L'obiettivo che si vuole ottenere è la convergenza, ovvero che l'accuratezza aumenti e la funzione di loss diminuisca, sui valori del training set. Si vuole ottenere la convergenza sull'insieme di test per avere un modello adeguato.

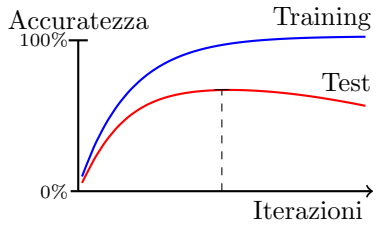


Si possono verificare vari casi, la loss potrebbe decresce con andamento oscillatorio, quindi il metodo di ottimizzazione potrebbe non essere efficace, oppure gli iperparametri potrebbero non essere ottimali, oppure il learning rate è inadeguato, questo rappresenta uno dei parametri più complessi da assegnare, quindi è necessaria una ricerca migliore.

La funzione di loss potrebbe decrescere, ma l'accuratezza potrebbe non aumentare, quindi è stata scelta una funzione di loss errata.

Inoltre è possibile che l'accuratezza non si avvicini al 100% sui dati di train, questo può essere poiché i gradi di libertà del modello sono troppo pochi, quindi si dovrebbero aumentare progressivamente per ottenere le prestazioni richieste. I gradi di libertà sono le variabili da specificare affinché sia individuato in maniera univoca il modello utilizzato. Per le reti dipendono dal numero dei neuroni, numero connessioni, etc. Di solito il numero dei gradi di libertà dipende da un iperparametro.

Si vorrebbe generalizzare il sistema, ovvero avere un'accuratezza il massimo possibile anche sul valid test, dopo aver addestrato il sistema sul training test. Quindi si cerca la capacità di trasferire l'elevata accuratezza dall'insieme di training ad un altro insieme, su cui il sistema ancora non è stato addestrato. Alcune volte si indica valid invece di test set, quindi si suppone che il valid test sia rappresentativo di test. Il problema che si riscontra da questa analisi è quello del sovra-adattamento. In caso il sistema abbia elevati gradi di libertà, è possibile abbia una accuratezza elevata sul training set, ma il modello non riesce a trasferire l'accuratezza anche su altri insiemi di test.



Si vuole quindi arrestare l'addestramento quando si raggiunge il picco nell'accuratezza sul test set, e questo rappresenta il modello ideale per il dato problema, con certi iperparametri scelti. Per cui monitorando l'andamento dell'accuratezza si può decidere quando arrestare l'addestramento, se questo viene effettuato iterativamente.

La soglia di Occam, è un principio di economia o parsimonia, ed afferma che se sono possibili più soluzioni, di privilegiare quella più semplice. Si parte quindi da un certo numero di gradi di libertà, analizzando diversi parametri, oltre l'accuratezza, come la regolarità della soluzione appresa. La regolarità può essere controllata introducendo un ulteriore iperparametro, chiamato fattore di regolarizzante alla funzione di loss che penalizza soluzioni irregolari.

In generale si consiglia di utilizzare dati il più possibile rappresentativi di tutte le possibili situazioni, e di un numero adeguato. Altrimenti si può effettuare del cloud sourcing per etichettare i dati. Nel caso dove è difficile avere pattern rappresentativi del problema si possono trovare delle aree del piano delle soluzioni non necessariamente corrette, e quindi in caso rimuoverle.

Inoltre è consigliabile automatizzare fin dall'inizio le procedure di valutazione delle prestazioni. Diversi modelli vanno testati utilizzando gli stessi protocolli. Bisogna effettuare test statistici per escludere a priori l'ipotesi nulla, ovvero che le differenza tra vari algoritmi siano dovute al caso, soprattutto in ambienti dove diversi modelli sono molti simili tra di loro.

4.7 Classificatore di Bayes

Il classificatore di Bayes si basa sull'omologo teorema, ed è un modello basato sulla probabilità. È tra i modelli di ML più utilizzati, utilizzato nell'analisi e classificazione dell'attitudine dei messaggi di utenti su social network o per identificare messaggi di spam, etc.

Si analizzeranno due tipi di approcci ed i criteri necessari per scegliere quale utilizzare:

- Approccio Parametrico: distribuzione Multi-Normale;
- Approccio Non Parametrico: Parzen Window.

Si ha un agente in un ambiente, con un comportamento non noto a priori, ma probabilistico. Se l'ambiente non è deterministico, quindi in caso di incertezza la teoria della probabilità è il modo migliore in cui si può analizzare il problema.

Si definisce lo spazio di probabilità una terna $(\Omega, \mathscr{A}, P)$, dove Ω è lo spazio degli insiemi, rappresentanti tutti i risultati possibili, $\mathscr{A}$ è la σ-algebra, ovvero l'insieme di eventi per i quali si può calcolare la probabilità P, compresa tra zero ed uno.

Una σ-algebra è una famiglia di insiemi che rispetta tre condizioni:

$$\emptyset \in \mathscr{A}$$

$$A \in \mathscr{A} \implies \overline{A} \in \mathscr{A}$$

$$A_k \in \mathscr{A}, \text{ per } k = 1, \cdots, N \implies \bigcup_{k=1}^{N} A_k \in \mathscr{A}$$

La probabilità di un evento $A \in \mathscr{A}$ si effettua dal rapporto di tutti i casi favorevoli in Ω e tutti i possibili risultati in Ω:

$$P(A) \triangleq \frac{\dim \Omega_A}{\dim \Omega}$$

Una probabilità deve rispettare gli assiomi di Kolmogorov:

- La misura di ogni evento è compresa fra zero ed uno:

$$\forall A \in \mathscr{A},\ P(A) \in [0, 1]$$

- La misura dell'intero insieme di eventi è pari ad uno:

$$P(\Omega) = 1$$

- La probabilità dell'unione di eventi disgiunti è pari alla somma delle probabilità dei singoli eventi. Due eventi di dicono disgiunti quando la loro intersezione è nulla:

$$\forall A, B \in \mathscr{A} \text{ t.c. } A \cap B = \emptyset \implies P(A \cup B) = P(A) + P(B)$$

Due eventi disgiunti quindi non hanno alcuna correlazione statistica, quindi due feature disgiunte tra di loro non avranno in alcun modo impatto l'una sull'altra.

Dato uno spazio di probabilità $(\Omega, \mathscr{A}, P)$due eventi $A, B \in \mathscr{A}$ si definiscono disgiunti se vale la seguente proprietà sulle loro probabilità:

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

Quest'assioma è alla base di alcuni classificatori, ma rappresenta un'idealizzazione poiché due eventi non saranno mai disgiunti, ma si può dimostrare che rappresenta un buon modello.

Un modello probabilistico è uno spazio di possibili esiti, descrizioni complete di stati, mutualmente esclusivi insieme alla misura della probabilità di ciascun elemento

Il teorema di Bayes su cui si basa il classificatore di Bayes, si basa sulla probabilità condizionata. Dati due eventi $A, B \in \mathscr{A}$, la probabilità condizionata di A consiste nella probabilità che avvenga A, noto l'avvenimento dell'evento B:

$$P(A|B)$$

Rappresenta una correzione delle aspettative dell'evento A, in seguito all'osservazione dell'evento B. Questo evento condizionante deve avere una probabilità non nulla di verificarsi, altrimenti non avrebbe senso la sua probabilità.

Si può dimostrare che in condizioni probabilistiche, con distribuzioni di probabilità note, l'approccio probabilistico è la soluzione migliore. Si suppone di avere uno spazio $\mathbf{V}$ d-dimensionale di pattern e un insieme di s classi $W = \{w_1 \cdots, s\}$ disgiunte costituite da elementi di $\mathbf{V}$.

Una densità di probabilità diventa una probabilità se viene moltiplicata per un intervallo, arbitrariamente piccolo. Per ogni datapoint $\mathbf{x}$ in $\mathbf{V}$, per ogni classe $w_j \in \mathbf{W}$, si indica con $p(\mathbf{x}|w_j)$ la densità di probabilità condizionale di $\mathbf{x}$, data la sua classe di appartenenza w_j. Si indica la probabilità a priori $P(w_j)$, indipendente dalle osservazioni, per ogni classe w_j dell'insieme W, probabilità

che il prossimo pattern da classificare sia di classe w_j. Per ogni elemento $\mathbf{x} \in \mathbf{V}$ si indica la densità di probabilità assoluta $p(\mathbf{x})$ di $\mathbf{x}$, la somma delle densità di probabilità che il prossimo pattern da classificare sia $\mathbf{x}$:

$$\forall \mathbf{x} \in \mathbf{V},\ p(\mathbf{x}) \triangleq \sum_{i=1}^{s} p(\mathbf{x}|w_j) \cdot P(w_j),\ \text{con}\ \sum_{i=1}^{s} P(w_j) = 1 \tag{4.7.1}$$

Per ogni classe $w_j \in W$ e per ogni pattern $\mathbf{x} \in \mathbf{V}$ si definisce la probabilità a posteriori $P(w_j|\mathbf{x})$ di w_j dato $\mathbf{x}$, la probabilità che avendo osservato il pattern $\mathbf{x}$, la sua classe di appartenenza sia w_j. Questa rappresenta l'obiettivo della classificazione, individuare per ogni classe la probabilità a posteriori che un pattern appartiene a questa classe. Si può calcolare mediante il teorema di Bayes come:

$$P(w_j|\mathbf{x}) = \frac{p(\mathbf{x}|w_j) \cdot P(w_j)}{p(\mathbf{x})} \tag{4.7.2}$$

Dato un pattern $\mathbf{x}$ da classificare in una delle s classi w_j, e note le probabilità a priori $P(w_j)$ di ogni classe e la densità di probabilità condizionali $p(\mathbf{x}|w_j)$, la regola di classificazione di Bayes assegna a $\mathbf{x}$ la classe b per cui è massima la probabilità a posteriori:

$$b = \operatorname{argmax} \{P(w_j|\mathbf{x})\} \tag{4.7.3}$$

Quindi massimizzare la probabilità a posteriori corrisponde a massimizzare la probabilità condizionata. Si può dimostrare che la regola di Bayes è quella ottima, poiché minimizza l'errore di classificazione $P(\text{errore})$. Si considera un caso di un classificatore binario, ed uno spazio monodimensionale $\mathbf{V}$. Data una densità di probabilità, il suo integrale corrisponde alla probabilità di quell'evento.

La probabilità di errore corrisponde si ottiene dall'integrale sullo spazio degli eventi $\mathbb{R}_i$ della classe w_i, dell'evento $x \in V$ di classe w_j e probabilità condizionata $p(x|w_j) \cdot P(w_j)$. Questo si effettua per entrambi le classi e si somma la probabilità di errore in entrambi i casi:

$$P(\text{errore}) = \int_{\mathbb{R}_1} p(x|w_2)P(w_2)\mathrm{d}x + \int_{\mathbb{R}_2} p(x|w_1)P(w_1)\mathrm{d}x$$

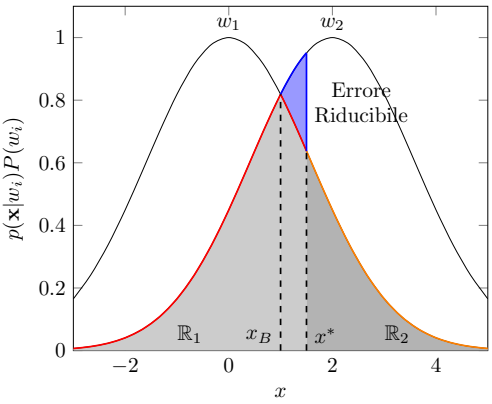


Fig. 5: Errore di Classificazione Binaria

Si può identificare un valore di separazione x^*, per determinare l'errore come l'area sotto le due curve. Il valore Bayesiano x_t corrisponde all'intersezione tra le due curve di probabilità, e si può dimostrare che se questo valore corrisponde al valore di separazione, allora l'errore riducibile è pari a zero.

Mentre per le probabilità a priori $P(w_i)$ delle varie classi il calcolo è semplice, per calcolare la densità probabilità condizionale $p(\mathbf{x}|w_i)$ il procedimento è più complesso. Dato un pattern $\mathbf{x} \in \mathbf{V}$, si può considerare un cerchio di raggio r, dove r rappresenta un iperparametro, si può stimare la densità di probabilità contando le occorrenze delle varie classi $w_j \in W$ che cadono nell'intorno centrato in $\mathbf{x}$. La probabilità assoluta $p(\mathbf{x})$ quindi si calcola date queste due approssimazioni moltiplicando la probabilità a priori delle probabilità delle classi per la densità di probabilità condizionale delle varie classi, da cui si può ottenere grazie al teorema di Bayes le probabilità a posteriori $P(w_i|\mathbf{x})$. L'approccio Bayesiano assegna quindi un pattern alla probabilità a posteriori maggiore calcolata in questo modo. Questo non fornisce solamente l'appartenenza, ma fornisce anche una probabilità di appartenenza ad una classe, e questo comporta vantaggi notevoli nell'ambito della classificazione.

Ciò che interessa è calcolare il numeratore del teorema di Bayes, calcolabile in maniera approssimativa nel moto descritto precedentemente. Il problema è il calcolo della densità probabilità condizionata. Esistono due approcci per calcolarla, l'approccio parametrico e l'approccio non parametrico, poiché la conoscenza delle densità di probabilità è possibile solo in teoria.

Nell'approccio parametrico si ipotizza che la densità di probabilità sia di un certo tipo, in generale si assume sia una gaussiana, ma bisogna stimare i parametri, i gradi di libertà, di questa gaussiana, ovvero valore medio μ e la varianza σ^2, nel caso monodimensionale. Nel caso di una distribuzione multinormale, il vettore medio e la matrice di covarianza, generalizzazione dei parametri precedenti. Questi parametri vengono stimati sulla base del training set, ipotizzando una distribuzione campionaria. Questi parametri permettono di effettuare certe valutazioni sul comportamento delle features.

L'altro approccio si chiama non parametrico e non effettua alcuna ipotesi sulla distribuzione di probabilità, ma si calcola questa densità utilizzando il metodo di Bernoulli oppure della Parzen Window. L'approccio non parametrico si basa sull'apprendere la distribuzione direttamente dal

training set, ed è un'operazione complicata applicabile in pochi casi. Può essere che in alcuni casi la determinazione della distribuzione campionaria sia più complessa del problema stessa.

4.7.1 Approccio Parametrico

L'approccio parametrico si utilizza quando sono note delle conoscenze dal problema su quale potrebbe essere la densità di probabilità e quindi si ha una ragionevole certezza che la forma ipotizzata sia adeguata al problema. L'approccio parametrico per definizione ha un numero di gradi di libertà basso, per cui diminuisce il rischio di overfitting, quando il training set è piccolo.

La densità di probabilità della distribuzione normale, monodimensionale, è:

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right]$$

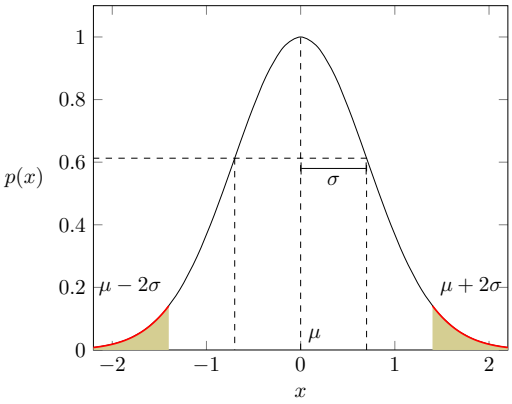


Fig. 6: Gaussiana

Bisogna considerare che la notazione per la densità di probabilità della distribuzione normale e della probabilità è la medesima, e quindi questo potrebbe causare confusione, poiché rappresentano due grandezze diverse.

Dove μ è il valore medio, la deviazione standard σ, la distanza tra il valore medio ed il punto di flesso della gaussiana, ed il suo quadrato σ^2 chiamato varianza. La gaussiana è tale che più ci si allontana dal valore meglio più diminuisce, quindi generalmente si considera solo l'intervallo compreso fino a 2σ, poiché quando cadono al di fuori di questo intervallo il contributo della gaussiana vale meno del 5%, quindi si suppone sia nullo.

Uno stimatore possibile per determinare i parametri della distribuzione è il metodo della Massima Verosimiglianza, "Maximum Likelihood". Consente di massimizzare la realizzazione campionaria, in base ai valori assunti da parametri statistici stimati.

Il valor medio si determina come la media tra i campioni nel training set:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$$

Per stimare la varianza si calcola la media delle deviazioni standard del campione ed il valore medio:

$$\sigma = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2$$

In questo modo si può definire la gaussiana dati questi parametri, che rappresenta la densità di probabilità. Per definire la probabilità bisogna moltiplicare questa densità per l'entità di un intervallo.

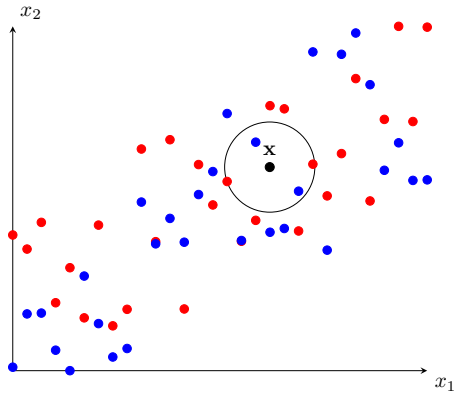


Fig. 7: Metodo della Massima Verosimiglianza

Tuttavia non si avranno solo due classi o una features, quindi servono dei modi per poter rappresentare queste informazioni. Si utilizza una notazione matriciale, dove ogni pattern $\mathbf{x}_i$ i-esimo rappresenta un vettore con j componenti x_i^j. La densità di probabilità nella distribuzione multinormale è definita come:

$$p(\mathbf{x}) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu})} \tag{4.7.4}$$

Dove $\Sigma = [\sigma_{ij}]$ rappresenta la matrice di covarianza $d \times d$, e generalizza il concetto di varianza. Questa matrice è sempre simmetrica, ovvero trasposta di sé stessa, e definita positiva, ovvero che gli autovalori sono strettamente positivi. E $\boldsymbol{\mu} = [\mu_1, \cdots, \mu_d]$ rappresenta il valore medio, dove d è la dimensione di $\mathbf{x}$, ovvero il numero di features. In generale la dimensione d della gaussiana dipende dal numero delle features, ed il numero delle gaussiane dal numero di classi.

Si definisce la matrice trasposta data una matrice $\Sigma = [\sigma_{ij}]$ come $\Sigma^T = [\sigma_{ji}]$. Si definisce la matrice inversa Σ^{-1} la matrice tale che moltiplicata per la matrice originaria produce la matrice

identità $\Sigma \cdot \Sigma^{-1} = I$. Si definisce la matrice trasposta coniugata come la matrice trasposta dove ogni valore viene sostituito al suo complesso coniugato.

Data una matrice Σ quadrata di ordine n, in un campo $\mathbb{K}$ che può essere $\mathbb{R}$ o $\mathbb{C}$, si dice uno scalare $\lambda_i \in \mathbb{K}$ un autovalore della matrice quadrata se esiste un vettore colonna non nullo $\mathbf{v} \in \mathbb{K}^n$ tale che:

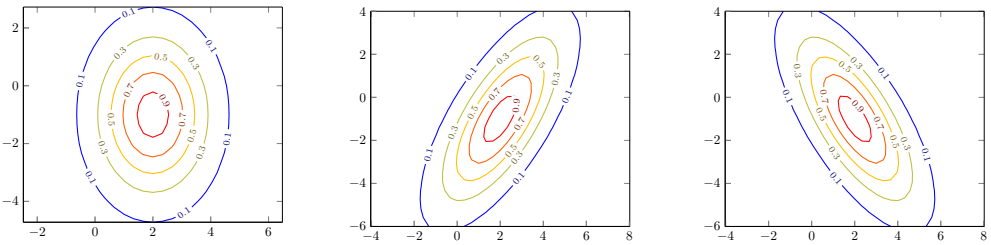
$$\Sigma \cdot \mathbf{v} = \lambda_i \cdot \mathbf{v}$$

Questo vettore $\mathbf{v}$ viene chiamato autovettore relativo all'autovalore λ_i. Dato questo autovettore anche $\alpha\mathbf{v}$ rappresenta un autovettore associato all'autovalore λ_i.

Si assume che tutti i vettori interessati siano vettori colonna, e per passare da vettore colonna a riga si utilizza la trasposta. Se la matrice Σ è simmetrica, il numero di parametri che la definisce è $d \cdot (d+1)/2$, questi rappresentano i gradi di libertà della matrice.

Gli elementi diagonali σ_{ii} rappresentano le varianze dei rispettivi x^i, σ_i^2, mentre gli elementi non diagonali σ_{ij} rappresentano le covarianze tra x^i e x^j. Se sono nulli queste due feature sono statisticamente indipendenti, se è positivo sono correlati positivamente, altrimenti sono correlati negativamente.

Per $d = 2$, la forma della distribuzione è quella di un ellisse, dove il vettore medio $\boldsymbol{\mu} = \begin{bmatrix}\mu_1 & \mu_2\end{bmatrix}$ individua il centro dell'ellisse, mentre le varianze σ_{11} e σ_{22} rappresentano l'allungamento dell'ellisse sull'asse principale, questo è legato allo spazio di variazione delle features. I due valori $\sigma_{12} = \sigma_{21}$ determinano la rotazione delle ellissi sugli assi principali. Se questi valori fuori dalla diagonale principale sono nulli, allora le ellissi sono parallele agli assi principali. Se è maggiore di zero l'ellisse è ruotata in senso antiorario, se sono negativi, saranno ruotate in senso orario. Gli assi dell'ellisse sono paralleli agli autovettori di Σ, i valori di σ_i e σ_j rappresentano gli autovalori di questa matrice.



(a) $\sigma_{ij} = \sigma_{ji} = 0$ (b) $\sigma_{ij} = \sigma_{ji} > 0$ (c) $\sigma_{ij} = \sigma_{ji} < 0$

Fig. 8: Distribuzione Rispetto a Variazione di $\sigma_{ij} = \sigma_{ji}$

Quando le feature sono statisticamente indipendenti si utilizzano i classificatori di Naïve Bayes, e si possono dimostrare lavorare bene su molti dataset. Ma questa caratteristica è molto rara per due features. In realtà si può applicare anche su molti casi dove l'assunzione di indipendenza statistica non viene verificata. La selezione delle feature va effettuata in modo da minimizzare la correlazione tra le feature, selezionando quelle più indipendenti. Features strettamente correlate tra di loro rappresentano soltanto rumore.

Nella gaussiana bidimensionale l'esponente viene definito come distanza di Mahalanobis r:

$$r^2 = (\mathbf{x} - \boldsymbol{\mu})^T \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu}) \tag{4.7.5}$$

Spesso utilizzata invece della distanza euclidea, tiene conto degli spazi di variazione e di come sono collegati tra di loro.

Si possono individuare curve di distanze di Mahalanobis costanti, utilizzate per determinare la distanza tra due pattern e la relazione della distanza con lo spazio di variazione della feature e le loro correlazioni. Queste ellissi sono i luoghi geometrici dei punti a densità costante:

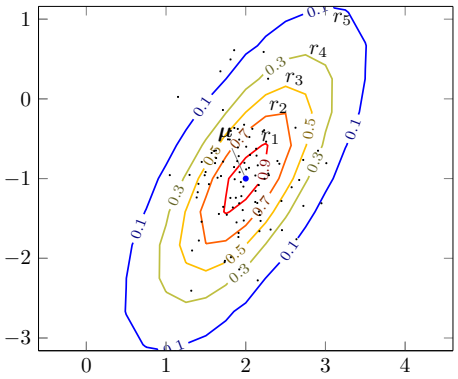


Fig. 9: Densità di Probabilità con Curve di Mahalanobis

Per eseguire il calcolo della funzione distribuzione multinormale, si effettua un processo analogo a quello monodimensionale, generalizzando al caso dove i pattern hanno più di una feature. Si può applicare quindi il metodo della massima verosimiglianza allo stesso modo, effettuando le operazioni per ciascuna componente del vettore medio e della matrice di covarianza:

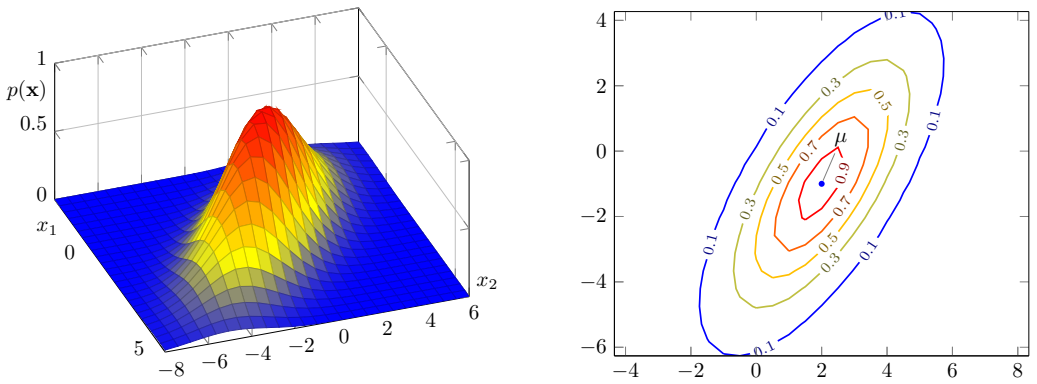
$$\boldsymbol{\mu} = \begin{bmatrix} \mu_1 \\ \vdots \\ \mu_d \end{bmatrix} = \begin{bmatrix} \frac{1}{N} \sum_{j=1}^{N} x_j^1 \\ \vdots \\ \frac{1}{N} \sum_{j=1}^{N} x_j^d \end{bmatrix} = \sum_{i=1}^{N} \mathbf{x}_i \tag{4.7.6}$$

La matrice di covarianza si calcola per ogni σ_{ij}:

$$\sigma_{ij} = \frac{1}{N} \sum_{k=1}^{N} (x_k^i - \mu_i) \cdot (x_k^j - \mu_j)$$

Si può rappresentare quindi in forma vettoriale come:

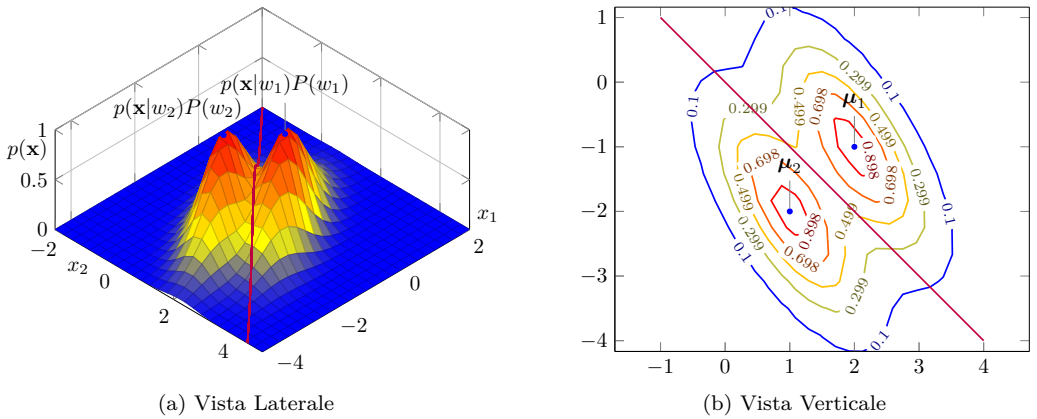
$$\Sigma = \frac{1}{N} \sum_{i=1}^{N} (\mathbf{x}_i - \boldsymbol{\mu}) \cdot (\mathbf{x}_j - \boldsymbol{\mu})^T \tag{4.7.7}$$



(a) Vista Laterale (b) Vista Verticale

Fig. 10: Distribuzione Normale Bivariata

Si rappresentano in uno spazio tridimensionale come una gaussiana bidimensionale, dove le dimensioni del piano danno le due feature, ed il valore della densità di probabilità rappresenta l'altezza di questa superficie. Analogamente dal valore di separazione, per distinguere due classi tra di loro, date le distribuzioni multinormali di queste classi, si utilizza la superficie di separazione .



(a) Vista Laterale (b) Vista Verticale

Fig. 11: Superficie Decisionale fra Due Distribuzioni Normali Bivariate

In questo caso di distribuzione bivariata normale la linea rosso identifica l'iperpiano decisionale tra due classi w_1 e w_2.

Se il punto cade su una di queste superfici decisionali, allora il suo valore è ambiguo, queste rappresentano zone intermedie tra le distribuzioni di probabilità. Nel caso di due classi, le curve possono assumere due forme, in genere due iperboli, se le matrici di covarianza sono tra di loro

uguali, la superficie è un iperpiano, ovvero ha una dimensione inferiore di uno rispetto alla superficie su cui è definita. Altrimenti se le due matrici di covarianza sono diverse tra di loro la superficie decisionale è un iper-quadratica.

Nel momento in cui bisogna solamente assegnare una classe, non è necessario conoscere l'intervallo di confidenza, e quindi è sufficiente calcolare il numeratore del teorema di Bayes, poiché il denominatore è sempre uguale alla probabilità assoluta. La classe di un certo pattern $\vec{x}$ è quindi dato dal prodotto massimo tra la probabilità condizionale rispetto ad una classe w_j e la probabilità a priori della suddetta classe:

$$b = \text{argmax}\left\{p(\mathbf{x}|w_j) \cdot P(w_j)\right\} \tag{4.7.8}$$

Se le matrici di covarianza sono tutte uguali, non è necessario calcolare la probabilità assoluta per ogni classe, dato che il denominatore è uguale, bisogna solo calcolare l'esponente dell'esponenziale. Solamente il prodotto tra $(\mathbf{x}-\boldsymbol{\mu})^T \cdot (\mathbf{x}-\boldsymbol{\mu})$, poiché Σ è uguale, e si assegna la classe per cui questo valore è minimo. La distanza di Mahalanobis in questi casi diventa semplicemente una distanza euclidea.

Il classificatore di Bayes può essere utilizzato anche in problemi Open Set, poiché fornisce una probabilità che può essere confrontata con un valore di soglia. In questo modo dato l'iperparametro valore di soglia si possono classificare anche pattern di cui non si ha una certezza, ed in caso assegnarli alla classe fittizia. Questo metodo funziona anche con multi-classificatore, più classificatori di Bayes come il modello "Random Forest" con più alberi di decisione, e si confrontano in base all'intervallo di confidenza.

4.7.2 Approccio Non Parametrico

Quando si hanno pochi campioni conviene scegliere l'approccio parametrico, dove si ipotizza di conoscere la distribuzione dei campioni, ma se i campioni sono molti allora si può stimare questa funzione direttamente. Con pochi campioni si preferisce una soluzione più semplice per diminuire l'overfitting, poiché una gaussiana ha due gradi di libertà il valore medio o l'eventuale vettore medio, e la varianza o matrice di covarianza, infatti il modello potrebbe essere troppo complesso sui dati di training.

L'approccio parametrico può portare a dei problemi, poiché in base alla situazione, quest'ipotesi forte potrebbe non coincidere alla realtà. Ovvero la distribuzione dei campioni non rappresenta una gaussiana. Si possono effettuare diversi test formali ed empirici per determinare se veramente si tratta di una gaussiana per determinare se l'approccio sia corretto. In caso è corretto si calcolano i parametri statistici tramite il metodo della verosimiglianza.

Se i campioni invece sono in numero maggiore da poter stimare direttamente la distribuzione si utilizza questo approccio non parametrico. Questo rappresenta un problema statisticamente più complesso della classificazione. Molto spesso infatti è conveniente scegliere un altro tipo di classificatore invece di quello Bayesiano. Quanti più campioni si hanno più la stima è affidabile. Si conviene applicarlo quando si hanno tanti campioni e poche dimensioni al massimo $d = 3$ o $d = 4$. Poiché all'aumento delle dimensioni, il volume dello spazio aumenta considerevolmente da diminuire notevolmente la precisione del modello. I campioni diventano sparsi all'aumentare delle dimensioni dell'ipercubo dello spazio dei campioni.

Per stimare la distribuzione di probabilità si considera la distribuzione binomiale, utilizzata per descrivere un processo di Bernoulli. Avviene in caso di prove ripetute e dove il problema ha solamente due possibilità, di successo o fallimento. Queste prove devono essere indipendenti l'una dall'altra. Se su n prove, la probabilità che l'evento abbia successo k volte è data dalla seguente formula:

$$P(k \text{ su } n) = \binom{n}{k} p^k \cdot (1-p)^{n-k}$$

Formalizzando il problema in questo modo si può determinare la probabilità che un campione $\mathbf{x}$ cada in una regione $\mathbb{R}$ oppure no.

$$P_1 = \int_{\mathbb{R}} p(\overline{\mathbf{x}}) \mathrm{d}\overline{\mathbf{x}}$$

Si può quindi estendere la probabilità che tra n pattern indipendenti k cadano all'interno di $\mathbb{R}$:

$$P(k \text{ su } n) = \binom{n}{k} P_1^k \cdot (1-P_1)^{n-k}$$

Se si assume un volume abbastanza piccolo della regione $\mathbb{R}$, e la probabilità sia abbastanza costante in esso si può approssimare la probabilità in:

$$\begin{aligned} P_1 &= \int_{\mathbb{R}} p(\overline{\mathbf{x}}) \mathrm{d}\overline{\mathbf{x}} \approx p(\mathbf{x}) \cdot V \\ p(\mathbf{x}) &= \frac{P_1}{V} = \frac{k}{n \cdot V} \end{aligned} \tag{4.7.9}$$

Questo volume nel caso più semplice è un ipercubo d-dimensionale definito dalla funzione φ:

$$\varphi(\mathbf{u}) = \begin{cases} 1 & |u_j| \leq \dfrac{1}{2},\ j = 1, \cdots, d \\ 0 & \text{altrimenti} \end{cases} \tag{4.7.10}$$

Dato un ipercubo centrato nel campione che si vuole classificare, dato il lato h_n dell'ipercubo anch'esso un iperparametro. Il volume è quindi $V_n = h_n^d$. Il numero di pattern del training set che cadono all'interno dell'ipercubo è data da:

$$k_n = \sum_{i=1}^{n} \varphi\left(\frac{\mathbf{x}_i - \mathbf{x}}{h_n}\right)$$

Sostituendo k_n nella formula ottenuta precedente dal processo di Bernoulli:

$$p(\mathbf{x}) = \frac{1}{n \cdot V} \sum_{i=1}^{n} \varphi\left(\frac{\mathbf{x}_i - \mathbf{x}}{h_n}\right) \tag{4.7.11}$$

L'ampiezza è un iperparametro, e più la finestra è piccola, più la stima è statisticamente instabile e rumorosa, ovvero la differenza tra due classi è molto piccola. Se la finestra è grande invece la stima è più stabile, ma sfocata.

Si può dimostrare che per ottenere convergenza la dimensione della finestra deve essere calcolata tenendo conto del numero di campioni del training set:

$$V_n = \frac{V_1}{\sqrt{n}}$$

Dove V_1 è un iperparametro. Emanuel Parzen ha introdotto negli anni '60 la teoria sull'utilizzo di Soft Kernel, fornendo una rigorosa dimostrazione matematica. Nell'approccio precedente, se il pattern cade sulla frontiera oppure all'interno del volume, la probabilità non cambia, e le superfici di separazione sono nette e definite. Invece con un Soft Kernel, un pattern contribuisce alla stima della densità in base alla sua distanza dal pattern di cui si vuole stimare. Le superfici decisionali sono quindi molto più regolari.

Le "Kernel Function" devono essere funzioni densità sempre positive, e con integrale su tutto lo spazio unitario. Si utilizza quindi anche in questo caso una funzione gaussiana multinormale, con vettore medio nullo, e matrice di covarianza pari alla matrice identità:

$$\varphi(\mathbf{u}) = \frac{1}{(2\pi)^{d/2}} e^{-\frac{\mathbf{u}^T\mathbf{u}}{2}} \tag{4.7.12}$$

Si prende quindi come iperparametro la deviazione standard della gaussiana per definire la dimensione di questa finestra, invece del lato dell'ipercubo. Questa rappresenta una funzione Soft Kernel, dove il valore dell'iperparametro h non è legato alla lunghezza del lato ma all'ampiezza della funzione scelta per sostituirlo.

L'approccio non parametrico con la Parzen Window produce con una funzione Kernel normale e con un iperparametro basso, oppure con una funzione Kernel ipercubo ed iperparametro elevato, una probabilità molto simile. Le superfici di separazione sono molto più nette con un ipercubo, mentre con una funzione Soft Kernel la frontiera di queste superfici di separazione è sfumata e rappresenta un calo della probabilità all'avvicinarsi ad esse.