Risposte domande di IML

Riccardo Graziani

Anno Accademico 2024/2025

Indice

1 Domande prima parte

1.1 Domanda su bias, variance e loro rapporto

(Q.) Si descrivano nel modo piú accurato possibile i concetti di bias e variance, il loro rapporto e come nella pratica possano essere affrontati e ridotti. A tal fine si riportino anche esempi concreti che aiutino a chiarire i diversi aspetti coinvolti.

(A.) Il **bias** rappresenta la differenza tra il valore **atteso** delle predizioni del modello e il valore **reale** della funzione target. Un bias elevato indica che il modello ha fatto assunzioni troppo rigide, limitando la sua capacitá di adattarsi ai dati: tale fenomeno viene definito come **underfitting**. Esistono diversi tipi di bias:

- **Induttivo**: la selezione di un algoritmo di apprendimento implica delle **assunzioni** rispetto allo spazio delle ipotesi **H**, ossia implica delle assunzioni sulla natura del target della funzione e della sua selezione. Queste assunzioni posso essere categorizzate in: **restrittive**, ossia che limitano lo spazio delle ipotesi, o di **preferenza**, ossia che impongono un ordine sullo spazio delle ipotesi.
  Un algortimo di **linear regression** assume, ad esempio, una relazione lineare tra features ed esempi. Un algoritmo **nearest neighbour**, invece, assume che gli esempi vicini condividano la stessa classe.

- **Algoritmico**: sono errori **sistematici** e **ripetuti** in un sistema che generano risultati scorretti, per esempio privilegiando un gruppo arbitrario di utenti rispetto ad un altro. Questi errori sono solitamente causati dai **dataset** forniti (che posso presentare dati **incompleti** o che favoriscono un certo tipo rispetto ad un altro).
  Un algoritmo di **facial recognition**, ad esempio, ha un accuratezza del 99% per persone occidentali, del 70% per persone caucasiche e del 60% per le persone di colore.

La **varianza** misura la **sensibilitá** del modello ai dati di training. Una varianza alta puó far si che un algoritmo modelli rumore casuale nei dati di training riducendo la capacitá di generalizzare a dati nuovi: tale fenomeno é definito **overfitting**.
Dunque in casi **underfitting** sono presenti pochi parametri nel modello e

un'elevata discrepanza nella classificazione (**high bias**): ció porta il processo di apprendimento ad essere troppo semplice. Invece in casi di **overfitting** sono presenti troppi parametri nel modello e un'elevata variabilitá della classificazione: il modello é dunque troppo complesso e sensibile ai dati di training (**high variance**).
Un modello ottimale cerca di ottenere un **low bias** e una **low variance**, ottenibile nei seguenti modi:

- **Low variance**: per ottenerela possiamo aggiungere ulteriori dati al training set, ridurre il numero di features del set o introdurre tecniche di regolarizzazione.
- **Low bias**: per ottenerlo possiamo usare un set con un numero maggiore di features, aggiungere ulteriore complessitá al modello o ridurre la regolarizzazione (se applicata).

1.2 Domanda su logistic regression, linear classification e gradient descent

(Q.) Si descriva in modo accurato il modello di logistic regression, le sue principali caratteristiche ed il contributo dei diversi elementi presenti nella funzione di costo. Si riporti inoltre una comparazione con il modello di classificazione lineare, evidenziando elementi in comune e differenze principali. Infine, si descriva chiaramente la procedura di addestramento mediante l'applicazione di gradient descent.

(A.) La **logistic regression** é un'algortimo, appartenente alla classe del **supervised learining**, che modella la probabilitá di appartenenza di un'osservazione ad una classe. L'algoritmo stima la probabilitá di occorrenza di un evento adattando i dati lineari in input a una funzione **logistica**, che in questo algoritmo é la funzione **sigmoide**:

$$h_\theta(x) = g(z) = \frac{1}{1 + e^{-z}}, \quad z = \theta^T x$$

che presenta valori compresi tra 0 e 1. Formalmente $h_\theta(x)$ é rappresentata come:

$$h_\theta(x) = P(y = 1|x; \theta); \quad h_\theta(x) = \text{probabilitá che } y = 1 \text{ su input } x$$

in cui la soglia di decisione standard é fissata a 0.5, ma puó essere modificata a seconda del problema. Con due classi, possiamo sfruttare la proprietá di **marginalizzazione**:

$$P(y = 1|x; \theta) + P(y = 0|x; \theta) = 1 \Rightarrow P(y = 0|x; \theta) = 1 - P(y = 1|x; \theta)$$

La funzione di costo per la logistic regression é la seguente (detta **cross entropy loss**):

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}log(h_\theta(x^{(i)})) + (1 - y^{(i)})log(1 - h_\theta(x^{(i)})))$$

[#](Questa parte copre la regolarizzazione, controllare la domanda)[#].
Possiamo applicare la tecnica della regolarizzazione alla cross entropy loss function per combattere il problema dell'overfitting:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}log(h_\theta(x^{(i)})) + (1 - y^{(i)})log(1 - h_\theta(x^{(i)}))) + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

[#](Fine parte della regolarizzazione)[#]
Le componenti principali di questa funzione apportano il seguente contributo alla sommatoria:

- $y^{(i)}log(h_\theta(x^{(i)}))$: se il vero valore di $y^{(i)}$ é 1 la **penalitá** aumenta all'aumentare della distanza tra la predizione $h_\theta(x^{(i)})$ é il valore effettivo (piú si avvicina a 0 piú il risultato del logaritmo tende a ∞).
- $(1-y^{(i)})log(1-h_\theta(x^{(i)}))$: se il vero valore di $y^{(i)}$ é 0 la **penalitá** aumenta all'aumentare della distanza tra la predizione $h_\theta(x^{(i)})$ é il valore effettivo (piú si avvicina ad 1 piú il risultato del logaritmo tende a ∞).

La cross entropy loss function presenta inoltre la proprietá di essere una funzione **convessa**. Grazie a questa proprietá possiamo applicare l'algoritmo di **gradient descent** per minimizzare il valore della loss function (e di conseguenza migliorare l'accuratezza del modello). Data la formula:

$$\theta_j := \theta_j - \eta\nabla J(\theta) = \theta_j - \frac{\eta}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

che é la stessa implementazione del gradient descent usata per la linear regression. Il parametro η é detto **learning rate** e rappresenta la lunghezza

del passo che l'algoritmo intraprende verso la discesa piú rapida. Un learning rate troppo alto rischia di non far convergere l'algoritmo, mentre un learning rate troppo basso aumenta il tempo necessario alla convergenza. Per calcolare il gradiente il costo viene calcolato su tutto il training set e dopo ogni aggiornamento il gradiente viene ricalcolato per il nuovo vettore θ_j: questo metodo é chiamato **batch gradient descent**, in quanto ogni step utilizza tutti i dati di training. Un'alternativa al BGD é lo **SGD** (Stochastic Gradient Descent), il quale effettua uno shuffle iniziale dei dati di training per poi aumentare ad ogni step la quantitá di dati utilizzata. In scenari reali é comune utilizzare una via di mezzo tra questi due metodi.
Le differenze tra linear e logistic regression sono molteplici:

- nella linear regression l'algoritmo restituisce in output un valore tramite l'interpolazione dei dati in input, mentre la logistic regression é un modello di classificazione (binaria e multiclasse);
- l'uso della tecnica del gradient descent é la stessa in entrambi i casi;
- nei problemi di classificazione, in caso di dati uniformemente distribuiti (assieme alla tecnica dei **minimi quadrati**) le due tecniche hanno un comportamento simile, se i dati non sono uniformemente distribuiti invece la linear regression crea un errore (cercando di essere troppo giusta) al contrario della sigmoide;

1.3 Domanda su principali paradigmi del machine learning

(Q.) Quali sono i principali paradigmi del machine learning? Se ne riporti una descrizione sintetica, chiarendo quali siano le principali differenze, con particolare enfasi per il caso del supervised learning. Si distinguano in particolare classificazione e regressione.

(A.) I principali paradigmi usati nel machine learning sono i seguenti:

- **Supervised learning**: l'algoritmo apprende da un insieme di dati etichettati e l'obiettivo é che l'algoritmo fornisca una risposta corretta ad ogni esempio del dataset. In questo contesto gli algoritmi di supervised learning si dividono in **classificazione** (per dati con etichette discrete) e **regressione** (per dati con etichette continue). Si presuppone che i

dati siano delle **coppie** (x, y) in cui x rappresenta il dato vero e proprio e la y rappresenta la "risposta corretta" che l'algoritmo deve fornire. Deve inoltre essere presente una funzione di apprendimento che cattura le informazioni da ogni esempio. Viene quindi costruita una funzione $h(\cdot)$ la quale, data in input una x, deve produrre il rispettivo y corretto. All'inizio deve mappare, dai dati forniti, possibilmente tutti i tipi di x con la corrispettiva y dal dataset fornito per il training. L'output ottenuto varia in base al tipo di algoritmo utilizzato (per dati discontinui si usa la classificazione, mentre per dati continui si usa la regressione). É presente un entitá **esperta** (o insegnante) che fornisce la **supervisione** (ossia i valori di $h(\cdot)$) corrispondenti alle istanze di x.

- **Unsupervised learning**: agisce in base ai dati, i quali stavolta non sono etichettati come coppie valore e risposta attesa (e dunque é assente la componente esperta di supervisione). L'algoritmo in questo paradigma cerca di individuare similitudini/regolaritá/pattern/strutture nascoste all'interno dei dati e di raccogliere tutti i dati che ricadono nella stessa categoria (**clustering**). L'obiettivo é dunque quello di trovare regolaritá su tutto il dominio di input a partire dai sample x.

- **Reinforcement learning**: un agente impara ad interagire con un ambiente con l'obiettivo di massimizzare una funzione di **ricompensa**. In questo paradigma, l'agente prende delle decisioni basate sullo stato attuale dell'ambiente, eseguendo azioni che influenzano lo stato successivo e ricevendo **ricompense** o **penalitá** in base alle proprie azioni.

Gli algoritmi di **regressione** si basano sull'interpolazione dei dati per associare tra loro due o piú caratteristiche (**feature**). Quando fornisco all'algoritmo una caratteristica in input esso mi restituisce l'altra. La regressione si puó dividere in **lineare** e **non lineare**:

- **Regressione lineare**: é un approccio lineare che modella la relazione tra una variabile **dipendente** e una o piú variabili **indipendenti**.

- **Regressione non lineare**: é un approccio che utilizza **curve** o **spazi curvi** per modellare le relazioni. Ha il pregio di avere un'accuratezza previsionale maggiore rispetto alla controparte lineare ma presenta il rischio di cadere in problemi di **overfitting**.

Gli algoritmi di **classificazione** si occupano di assegnare labels (classi) a degli esempi. Quando fornisco un esempio al classificatore, l'algoritmo mi restituisce la classe di appartenenza piú probabile. Esistono classificatori **binari** o **multiclasse**. Abbiamo inoltre classificatori **multilabel** in cui andiamo ad allenare un nostro classificatore a riconoscere un determinato tipo di classe individualmente. Anche i classificatori sono divisi in lineari e non lineari:

- **Classificatori lineari**: sono semplici e veloci ma soffrono del problema dell'**underfitting**, ossia una capacitá limitata nel rappresentare relazioni tra input e output.
- **Classificatori non lineari**: sono piú precisi dei classificatori lineari, ma snon anche piú lenti nell'elaborazione é soffrono del problema dell'**overfitting**, ossia imparare dettagli o rumore nei dati riducendo la propria capacitá di generalizzare nuovi dati.

1.4 Domanda su linear classification, logistic regression e loro differenze/similitudini

(Q.) Si descrivano nel modo piú accurato possibile i modelli di linear classification e logistic regression; si evidenzino differenze, vantaggi e svantaggi dell'uno rispetto all'altro. Si descriva infine il processo di apprendimento tramite gradient descent e le rispettive funzioni di costo, motivando in modo adeguato la particolare forma utilizzata in entrambi i casi.

(A.) Per la descrizione della logistic regression si veda la domanda 1.2.
La **linear classification** é un algoritmo di **classificazione** appartenente al paradigma del **supervised learning**. Essa si basa su una funzione lineare:

$$h_\theta(x) = \theta^T x$$

e utilizza una funzione di decisione per stabilire una **decision boundary** (come ad esempio la funzione $y = sign(h_\theta(x))$. Di fatto si confronta il valore prodotto da $h_\theta(x)$ con la decision boundary stabilita (ad esempio 0.5) e in base al confronto si assegna la classe di appartenenza. Questo metodo presenta il vantaggio di essere semplice da comprendere ed implementare e di richiedere meno tempo per l'addestramento rispetto ai modelli piú complessi. Di contro se i dati non sono separabili linearmente il modello andrá incontro

a problemi di **underfitting**. Infine un modello di linear classification non riesce a comprendere le relazioni piú complesse tra le caratteristiche dei dati. La funzione di costo per questo modello é la **MSE** (Mean Square Error):

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

che misura in maniera assoluta la distanza tra il valore previsto dal modello $h_\theta(x^{(i)})$ rispetto al valore reale atteso $y^{(i)}$. Minimizzando la funzione tramite la tecnica del gradient descent, si usa la seguente formula per aggiornare i parametri dopo ogni step:

$$\theta_j := \theta_j - \frac{\eta}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

che é la stessa formula del gradient descent per la logistic regression (svolgendo le derivate parziale si arriva alla stessa formula).

1.5 Domanda su cross-validation, empirical error e true error

(Q.) Si descriva dettagliatamente la procedura di cross-validation, motivandone scopo ed utilità, e fornendo una chiara descrizione della (corretta) procedura di addestramento di un qualunque sistema di machine learning. Si descrivano inoltre i concetti di true error e d'empirical error e se ne evidenzino le relazioni con la procedura di cross-validation.

(A.) La tecnica di **cross-validation** é una tecnica **statistica** che permette di usare in modo alternato dati di training e dati di test. Rispetto alla tecnica di **hold-out**, la quale divide in modo statico i dati in parte di training e parte di test, la tecnica di cross-validation (specificamente la **k-fold cross validation**) effettua uno shuffle del set di dati, lo suddivide in k partizioni uguali e ne usa $k-1$ per il training e la restante per il test. Iterativamente si scorrono tutte le partizioni cambiando di volta in volta la partizione di test fino a quando non si é iterato su tutte le k partizioni. A questo punto viene elaborata una **media** dei vari risultati per calcolare la prestazione generale del modello. Infine viene effettuata un'ultima valutazione finale usando il set di test, il quale non é mai stato usato durante la cross-validation. Un caso speciale della k-fold cross-validation é **leave-one-out**, in cui il valore di k é

pari alla **cardinalitá** del set di training. La cross-validation vuole sostanzialmente irrobustire il training rispetto ai bias che ci possono essere sui dati disponibili e produce una stima piú realistica delle capacitá del modello. La cross-validation risulta peró molto costosa in termini computazionali quando il numero di partizioni é elevato.
Queste tecniche di validazione rispondono alla necessitá di determinare le prestazioni del modello sul nostro set di dati di addestramento: questo é definito **emprical error**. Quello che invece non siamo in grado di sapere é come si comporterá il modello in uno scenario di uso pratico (in quanto non siamo in grado di prevedere la vera **distribuzione** dei dati), ossia il **true error**. Dunque dato lo scenario in cui vogliamo modellare la seguente funzione:

$$h \sim f : X \to Y$$

il true error dell'ipotesi h rispetto all'obiettivo f e alla distribuzione dei dati D (per osservare un'istanza di x appartenente ad X) é la probabilitá che h classifichi erroneamente un'istanza estratta a caso secondo D:

$$error_D(h) \equiv Prob_{x \in D}\{f(x) \neq h(x)\}$$

Poiché abbiamo accesso solo ad un sottocampione del dominio di input, impariamo sulla base di quel campione di dati di addestramento. Non abbiamo accesso al true error ma all'empirical error. Il numero di esempi che l'ipotesi h, rispetto ad un insieme di addestramento T, classificherá in modo errato:

$$errro_T(h) \equiv \#\{(x, f(x)) \in T | f(x) \neq h(x)\}$$

Il nostro scopo é quello di minimizzare l'empirical error (**Empirical Risk Minimization**), ma poiché disponiamo solo di un sottocampione dei dati, puó accadere che minimizzando l'empirical error aumenti il true error. Questa problema si presenta sotto la forma dell'overfitting:

$$h \in H | h \text{ overfits } T \text{ if } \exists h' \in H \text{ such that}$$

$$error_T(h) < error_T(h') \text{ but } error_D(h) > error_D(h')$$

La cross-validation, in questo contesto, aiuta a stimare il true error usando dati mai visti dal modello durante il suo training.

1.6 Domanda su model selection

(Q.) Si descriva dettagliatamente la procedura di model selection (aiutandosi con un esempio concreto) e si fornisca una chiara giustificazione teorica/concettuale a tale procedura.

(A.) La model selection é il processo di scelta del modello di machine learning piú adatto al nostro problema: si cerca di bilanciare la complessitá del modello con le sue prestazioni, con l'obiettivo finale di ottenere un modello che riesca a generalizzare bene su dati mai visti, evitando i problemi di overfitting e underfitting. Prendiamo come esempio il seguente problema: vogliamo costruire un modello di machine learning che riesca a prevedere il prezzo di una casa in base alle sue caratteristiche quali metratura, numero di stanze ed etá. I passi principali per la scelta del modello ottimale sono i seguenti:

- **Definizione del problema e dei candidati**: identificare la categoria del nostro problema (regressione, classificazione, ecc.) e selezionare un gruppo di modelli adatti a tale problema (regressione lineare, regressione logistica, reti neurali, ecc.).
- **Suddividere i dati**: creare i set di dati da usare durante le fasi di **training** (training set), **validazione** (validation set), e **test** (test set).
- **Valutare le prestazioni**: utilizzando robuste tecniche di valutazione del modello come la **k-fold cross-validation**, utilizzando metriche di valutazione appropriate (accuracy, precision, recall o F1-score).
- **Scelta del migliore modello**: dai risultati di training scegliamo il modello che piú si avvicina ad un buon bilanciamento tra bias e variance, anche attraverso il confronto delle learning curves del bias e della variance (ottimizziamo il **bias-variance tradeoff**).
- **Test finale**: scelto il nostro modello lo sottoponiamo al nostro **test set** per verificarne la robustezza, con cui otteniamo una **stima imparziale** delle reali prestazioni del modello.

1.7 Domanda sulla one learning algorithm hypothesis e reti multistrato

(Q.) Cosa si intende per "*one learning algorithm hypothesis*" e come tale ipotesi si relaziona con le reti neurali artificiali? Si fornisca inoltre una descrizione esaustiva degli elementi/ingredienti principali che permettono la definizione di una rete neurale multi-strato.

(A.) La one learning algorithm hypothesis é l'ipotesi secondo cui esiste un **unico** algoritmo di apprendimento generale che puó essere usato per imparare qualsiasi cosa, indipendentemente dal dominio, dalla modalitá o dal tipo di problema. In particolare il cervello umano esibisce una struttura relativamente **uniforme** indipendentemente dal tipo di informazione che deve processare (vista, udito, linguaggio, ecc.): ció ha portato a teorizzare che il cervello possieda una strategia di apprendimento unificata che applica a diversi contesti. Lo stesso principio viene applicato alle **reti neurali**: invece di sviluppare diversi algoritmi per diverse tipologie di attivitá di apprendimento, viene creato un unico modello semplice in grado di essere adattato al compito assegnato.
La piú piccola unitá che compone una rete neurale é il **neurone artificiale** (anche definito come **perceptron/percettrone**), una struttura che ottiene molteplici dati in input e produce, attraverso una trasformazione lineare, un output. Esso é composto da:

- **Input**: i valori dati in ingresso al percettone.
- **Pesi e bias**: i relativi pesi di ogni valore in input, assieme ad un valore di bias (usato per manipolare la funzione di attivazione).
- **Somma netta**: la somma dei valori in input moltiplicati per i rispettivi pesi.
- **Funzione di attivazione**: la funzione a cui viene dato in input il risultato della somma netta.

Esistono molte funzioni di attivazione (come la sigmoide, la tangente iperbolica, ReLU, ecc.) e si preferiscono funzioni di attivazione **non lineari** in quanto consentono ai nodi di apprendere strutture piú complesse nei dati. Dunque una rete neurale artificiale altro non é che un gruppo di neuroni diversi interconnessi tra di loro. Viene definita **architettura** di una rete

neurale la scelta strutturale sul numero di **neuroni** e **layer** e di come sono disposti. Una rete neurale artificiale consta solitamente dei seguenti elementi:

- **Strato di input**: un insieme di neuroni che raccolgono input in diversi formati.
- **Strato di output**: gli input sono sottoposti a diverse trasformazioni attraverso lo strato nascosto della rete, che si traduce in un output che verrá inviato a questo strato.
- **Strati nascosti**: una serie di strati, posti tra gli strati di input e output, che hanno il compito di eseguire i calcoli e le trasformazioni necessarie per produrre l'output (al minimo si ha 1 strato nascosto, ma il numero cresce in base alla complessitá della rete).

Dunque possiamo creare una rete neurale con N strati (**N-layer neural network**) in cui N rappresenta l'insieme dello strato di output e degli strati nascosti (non si considera nel conteggio lo strato di input). Infine, definiamo un layer come **fully connected** se ogni neurone nello strato nascosto ha un input per ciascuno degli output neuroni presenti nello strato precedente.

2 Domande seconda parte

2.1 Domanda su SVM ed estensione con kernel trick

(Q.) Spiegare in dettaglio gli elementi fondamentali di SVM, in particolare: la sua interpretazione geometrica, la funzione di costo, le differenze similitudini con altri modelli di ML. Infine, si introduca brevemente l'estensione di SVM basata sul kernel trick.

(A.) Le **SVM** (Support Vector Machines) sono dei modelli di apprendimento, appartenenti alla categoria del **supervised learning** usati principalmente per problemi di **classificazione**, ma possono essere comunque applicati per problemi di **regressione**. Le SVM sono un modello classificatore **binario** (ossia non probabilistico), ma esistono metodi per dare alle SVM un output **probabilistico**.
Dal punto di vista geometrico, una SVM cerca l'**iperpiano** che separa i punti delle due classi in modo tale da **massimizzare** la distanza, chiamata **margine**, tra l'iperpiano stesso e i punti più vicini di ciascuna classe (che vengono

detti support vectors). L'algoritmo seleziona di fatto l'iperpiano "ottimale", dove "ottimale" significa quello che lascia la più ampia fascia vuota, detta margine massimo, tra i punti delle due classi.
La funzione di loss delle SVM si basa sull'idea di avere una funzione di costo **lineare** che si comporti in modo simile alla funzione di loss della **logistic regression**: vogliamo quindi che, nel caso in cui la classificazione sia corretta, l'errore sia pari a zero; al contrario, se abbiamo classificazioni errate la funzione di loss sia lineare. Questo si traduce nella seguente funzione:

$$\min_{\theta} C \sum_{i=1}^{m} [y^{(i)} cost_1(\theta^T x^{(i)}) + (1 - y^{(i)}) cost_0(\theta^T x^{(i)})] + \frac{1}{2} \sum_{j=1}^{n} \theta_j^2$$

In cui il parametro C serve a bilanciare l'**ampiezza** del margine e la rispettiva **penalizzazione** degli errori di classificazione: un C grande rende il margine piú piccolo e penalizza molto gli errori, un C piccolo al contrario presenta un margine piú largo e maggiore tolleranza agli errori. Non ci resta che definire cosa sono $cost_0(\theta^T x^{(i)})$ e $cost_1(\theta^T x^{(i)})$: SVM punisce sia le predizioni **sbagliate** che quelle corrette ma **vicine** alla decision boundary. In altre parole:

- se $y^{(i)} = 1$ allora vogliamo che $\theta^T x^{(i)} \geq 1$;
- se $y^{(i)} = -1$ allora vogliamo che $\theta^T x^{(i)} \leq -1$

Grazie a queste scelte stiamo creando un'area vicino alla decision boundary che vogliamo idealmente **evitare**. Per cui la funzione di **loss** delle SVM diventa:

$$J(\theta) = \sum_{i=1}^{m} cost(h_\theta(x^{(i)}, y^{(i)}))$$

con funzione di costo, detta **hinge loss**:

$$cost(h_\theta(x^{(i)}, y^{(i)})) = \begin{cases} max(0, 1 - \theta^T x^{(i)}) & se\ y^{(i)} = 1 \\ max(0, 1 + \theta^T x^{(i)}) & se\ y^{(i)} = -1 \end{cases}$$

Dunque la **decision boundary** di SVM viene rappresentata come una retta centrale a cui aggiungiamo due **rette** parallele: lo spazio compreso tra la decision boundary e le rette parallele rappresentano le **zone** da evitare.

L'obiettivo di ottimizzazione di SVM é attuabile solo sul secondo termine della funzione:

$$\min_{\theta} C \sum_{i=1}^{m} [y^{(i)} cost_1(\theta^T x^{(i)}) + (1 - y^{(i)}) cost_0(\theta^T x^{(i)})] + \frac{1}{2} \sum_{j=1}^{n} \theta_j^2$$

in quanto il primo termine é quasi nullo. Per cui:

$$\frac{1}{2} \sum_{j=1}^{n} \theta_j^2 = \frac{1}{2} \|\theta\|^2$$

Quindi poiché $\theta^T x^{(i)} = p^{(i)} \|\theta\|$ (dove $p^{(i)}$ é la proiezione di $x^{(i)}$ su θ) SVM é per sua implementazione portato a **minimizzare** $\|\theta\|$ e a massimizzare $p^{(i)}$: dunque si ha che SVM massimizza la **proiezione** di $x^{(i)}$ sul vettore θ, massimizzando il margine.
Rispetto ad altri modelli di machine learning, come la regressione logistica o il perceptron, le SVM condividono con questi l'approccio di trovare un **confine** decisionale lineare tra le classi. Tuttavia, a differenza della regressione logistica, la SVM non restituisce **probabilità**, ma si concentra esclusivamente sulla **separazione** e sul margine massimo. Un'altra differenza importante è che i parametri della SVM dipendono solo da un **sottoinsieme** particolare dei dati di training, ovvero i support vectors, mentre altri modelli utilizzano tutti i dati per determinare il confine.
Quando i dati non sono linearmente separabili nello spazio originale delle feature, le SVM possono utilizzare una tecnica chiamata **kernel trick**. Invece di cercare una separazione lineare nello spazio originario, il kernel trick permette di proiettare implicitamente i dati in uno spazio a dimensionalità **superiore** (ossia da $R^n \to R^m, m > n$), dove spesso diventa possibile trovare un iperpiano di separazione lineare. Questo avviene attraverso l'uso di una **funzione** kernel: $K(x, x')$ che calcola il **prodotto scalare** fra le immagini dei dati nello spazio trasformato, senza dover mai calcolare esplicitamente la trasformazione (quindi risparmiando complessità computazionale). Ad esempio, il kernel polinomiale viene definito come:

$$K(x, y) = \phi(x)^T \phi(y), \quad x, y \in R^n$$

che corrisponde al prodotto scalare in R^n. Così, la SVM è in grado di trovare una frontiera di classificazione non lineare nello spazio originale, ma che corrisponde a un iperpiano lineare nello spazio delle feature 'arricchite'. Il vantaggio principale è che questa operazione avviene in modo implicito, grazie al kernel, senza bisogno di conoscere o calcolare le nuove feature esplicitamente.

2.2 Domanda su algoritmo k-NN

(Q.) Si descriva in modo accurato l'algoritmo k-NN, illustrando il ruolo dei principali iperparametri, i vantaggi e le debolezze del modello nei confronti di altri algoritmi affrontati nel corso, e si evidenzi il principale inductive bias di tale algoritmo.

(A.) L'algoritmo **k-Nearest Neighbors** (k-NN) é un algoritmo di apprendimento, appartenente alla categoria del supervised learning, usato per risolvere problemi di classificazione e regressione. Questo algoritmo non include una fase di **apprendimento** (detto **lazy learning**) ma trova le **similitudini** tra le feature, inoltre il k-NN appartiene alla classe degli algoritmi **non parametrici**, ossia algoritmi che non richiedono un alto numero di parametri e in cui il lavoro viene svolto a tempo di **predizione**.
L'algoritmo k-NN si basa sull'algoritmo **Nearest Neighbor** (NN): si assume che dato un esempio di cui predire la classe di appartenenza e la sua rappresentazione nello spazio, esso apparterrá alla **stessa classe** degli esempi vicini ad esso. Il problema principale dell'algoritmo NN é che non riesce a gestire in modo efficiente i dati **anomali**: prendiamo per esempio un NN binario, e uno spazio nell'iperpiano in cui le due classi sono marcatamente distinte. Un dato anomalo é rappresentato da un esempio che appartiene alla classe X, ma la cui rappresentazione sull'iperpiano é molto vicina ad esempi di classe Y. Ció porterá il modello a collocare erroneamente nuovi esempi nell'iperpiano.
L'algoritmo k-NN migliora su questo aspetto, in quanto esso decide la classe di appartenenza basandosi sui k esempi piú vicini nello spazio, riducendo la sua sensibilitá ai dati anomali. Al diminuire di k il modello diventa sempre piú sensibile alle anomalie, ma un k troppo elevato rischia di considerare esempi troppo lontani, con il rischio di modellare erroneamente la classe di appartenenza. Una buona regola per il calcolo di k é la **radice quadrata** del numero di esempi, ma per avere un valore **ottimale** é utile effettuare una cross validation.
Un altro iperparametro importante del k-NN é il **modo** in cui viene calcolata la distanza tra i campioni del dataset: in generale viene utilizzata la distanza **Euclidea**, come espresso nella seguente formula:

$$d(x_j^{(a)}, x_j^{(b)}) = \sqrt{\sum_{j=1}^{n} (x_j^{(a)} - x_j^{(b)})^2}$$

ma si possono usare altre distanze come la **Manhattan** a seconda dei dati. Nell'algoritmo k-NN si assume che, nella maggior parte dei casi, i casi vicini tra loro appartengano alla stessa classe: questo rappresenta un caso di **inductive bias**. I principali vantaggi legati all'uso dell'algoritmo k-NN sono:

- la sua **semplicitá** di implementazione;
- la capacitá di **adattarsi** a nuovi dati;
- il numero limitato di **iperparametri**;

Di seguito sono invece elencati i principali problemi e limitazioni dell'algoritmo k-NN:

- la fase di predizione é molto costosa, poiché richiede il calcolo della distanza da **tutti** i campioni;
- all'aumentare del numero di feature il **concetto** di distanza perde di significato, rendendo piú complicato trovare dei vicini significativi (definita come **Curse of dimensionality**);
- l'algoritmo richiede molta memoria poiché deve mantenere **tutto** il dataset;
- alcune feature sono piú importanti di altre poiché hanno **range** maggiore;

Un aspetto importante dell'algoritmo k-NN è come si colloca rispetto ad altri modelli di machine learning comunemente incontrati. Rispetto ai modelli parametrici, come la regressione logistica o le Support Vector Machine (SVM), il k-NN si distingue perché è un metodo non parametrico. Questo significa che non costruisce una **formula esplicita** che relazioni le feature all'output, ma si basa direttamente sui dati memorizzati: la previsione viene effettuata calcolando le distanze tra il nuovo esempio e tutti i dati del training set. Di conseguenza, mentre nei modelli parametrici la predizione è rapidissima, nel k-NN la fase di predizione può risultare **molto lenta**, specie se il dataset è ampio. Se confrontiamo k-NN con gli alberi di decisione, emerge che gli alberi offrono una maggiore interpretabilità delle decisioni: producono regole chiare e leggibili, mentre k-NN **non fornisce** spiegazioni esplicite sulle sue previsioni. Inoltre, gli alberi riescono a gestire facilmente sia variabili numeriche sia categoriche e non necessitano necessariamente di normalizzazione

delle feature, contrariamente a quanto avviene per k-NN, dove tutte le feature devono essere portate sulla **stessa scala** per evitare che alcune dominino il calcolo delle distanze.

2.3 Domanda su alberi di decisione e random forest

(Q.) Si descrivano nel modo più accurato possibile gli alberi di decisione, i loro vantaggi e svantaggi rispetto ad altri modelli (in particolare rispetto ad algoritmi/modelli parametrici), e si evidenzi il principale inductive bias di tale algoritmo. Infine, si illustri brevemente l'estensione di tale modello attraverso random forest.

(A.) Gli **alberi di decisione** (o decision trees) sono modelli di apprendimento supervisionato utilizzati sia per classificazione che per regressione. Rappresentano una **sequenza** di decisioni gerarchiche, in cui ciascun nodo interno verifica una **condizione**, ciascun ramo rappresenta l'**esito** della condizione, e ciascun nodo foglia corrisponde a una **predizione finale** (che puó rappresentare la classe o il valore target).
Le variabili in un albero decisionale possono essere:

- **Discrete**: quindi con valore numerici interi e per i quali parliamo di **classificazione**.
- **Continue**: quindi con valori numerici reali e per i quali parliamo di **regressione**.

Esistono diversi algoritmi per costruire un albero: uno degli algoritmi storici è l'ID3 , **Iterative Dichotomiser 3**. L'albero, in questo algoritmo, viene costruito in maniera **top-down** usando la politica nota come **divide et impera** utilizzando il concetto di Entropia, definito in statistica come:

$$H(S) = -\sum_{c \in C} p(c) log_2 p(c)$$

in cui S é il nostro dataset, C é un sottinsieme di classi in S e $p(c)$ é la proporzione di elementi in c rispetto ad S. Il valore dell'entropia ci permette di scoprire diverse informazioni sul set di dati:

- **Entropia alta**: significa che la variabile ha **distribuzione uniforme** rispetto alle classi e i loro valori.

- **Entropia bassa**: significa che i dati sono principalmente distribuiti su **poche** classi.

Un altro concetto importante é quello di **Information Gain**, ossia la **differenza** di entropia nel prima e dopo aver **separato** un set S rispetto ad un attributo a. In altre parole, misura la riduzione dell'incertezza dopo che il set S é stato separato. Viene definito secondo la formula:

$$IG(S, a) = H(S) - \sum_{t \in T} p(t)H(t) = H(S) - H(S|a)$$

Dove abbiamo che T é il risultato del partizionamento di S secondo a, $p(t)$ é la proporzione di elementi in t rispetto ad S e $H(t)$ é l'entropia del subset T. Viene comunemente utilizzato nella costruzione di alberi decisionali da un set di dati di addestramento, valutando l'information gain per **ciascuna** variabile e selezionando quella che **massimizza** l'information gain, che a sua volta riduce al **minimo** l'entropia e divide al meglio il set di dati in gruppi per una classificazione efficace.
Il bias induttivo degli alberi di decisione consiste nell'assunzione che sia possibile **suddividere** lo spazio delle feature attraverso una serie **gerarchica** di split logici su singole variabili, ottenendo così predizioni accurate. Questo approccio funziona bene in molte situazioni pratiche, ma può essere limitato in presenza di relazioni **complesse** tra le feature che non siano ben **separabili** con condizioni univariate.
Tra i **vantaggi** principali degli alberi di decisione vi è sicuramente la **facilità** di interpretazione: l'intero processo decisionale può essere seguito passo passo, rendendoli molto trasparenti anche per utenti non esperti. Inoltre, non richiedono pre-processing particolarmente complessi, come la normalizzazione dei dati, e sono in grado di trattare sia variabili numeriche che categoriche. Sono anche modelli in grado di catturare **relazioni** non lineari tra le variabili, e in parte selezionano automaticamente le feature più rilevanti.
Tuttavia, presentano anche diversi **svantaggi**. Il principale è l'elevata **varianza**, ovvero la tendenza a **overfittare** i dati di training: un albero troppo **profondo** si adatta al rumore e generalizza male. Inoltre, sono modelli **instabili**, in quanto piccole variazioni nel dataset possono generare alberi completamente diversi. Rispetto ai modelli parametrici, come la regressione logistica o lineare, gli alberi di decisione non fanno **ipotesi** forti sulla forma della relazione tra le feature e il target e non producono una regola **compatta** in termini di una funzione parametrica da ottimizzare. Questo li rende

più **flessibili** nel catturare pattern complessi o relazioni non lineari, ma anche più suscettibili a **fluttuazioni** nei dati e meno regolarizzabili rispetto ai modelli parametrici, che generalmente sono più stabili e hanno una maggiore capacità di generalizzare, soprattutto quando il dataset non è particolarmente ampio.
Per ovviare ai limiti degli alberi di decisione, in particolare all'elevata varianza, è stata introdotta l'estensione nota come **Random Forest**. Una Random Forest consiste in un insieme di alberi decisionali, ciascuno addestrato su un diverso **sottoinsieme** casuale dei dati disponibili e, ad ogni split, su un sottoinsieme **casuale** delle feature. Questo duplice processo di campionamento rende ogni albero parzialmente **indipendente** dagli altri; in fase di predizione, i risultati dei singoli alberi vengono aggregati (ad esempio tramite voto di maggioranza per la classificazione o media per la regressione). Random Forest permette così di ridurre drasticamente la varianza complessiva del modello, aumentando la robustezza e la capacità di generalizzazione rispetto al singolo albero.