Università degli Studi di Padova
Dipartimento di Scienze Statistiche
Corso di Laurea Magistrale in
Scienze Statistiche

L'analisi di Big Data nel cloud: panoramica e applicazioni

Relatore Prof. Emanuele Aliverti
Dipartimento di Scienze Statistiche

Laureando Alessio Piraccini
Matricola 2020316

Anno Accademico 2021/2022

Indice

Introduzione

Il processo di innovazione tecnologica cominciato con l'avvento dei *personal computer* ha gradualmente portato ad una rivoluzione nel modo di comunicare e lavorare delle persone. Negli ambiti produttivi, l'utilizzo di infrastrutture tecnologiche via via più sofisticate ha avuto come conseguenza l'immagazzinamento di una mole sempre più crescente di dati. In questi dati è contenuta informazione critica utile a prevedere e programmare scenari futuri: un problema rilevante che si è sviluppato parallelamente a questo processo innovativo è quindi quello di come estrarre informazione da queste grandissime moli di dati e di come utilizzarla. Mentre per le applicazioni più classiche, come la verifica di ipotesi, venivano utilizzate le tecniche della statistica inferenziale, per una serie di compiti più particolari, ad esempio la costruzione di sistemi di raccomandazione, sono state sviluppate nel tempo delle tecniche informatiche specifiche, che prendono il nome di *machine learning*.

A prescindere dal tipo di approccio utilizzato per l'analisi dei dati in questi contesti, un problema importante da tenere in considerazione è la mole stessa dei dati, potenzialmente tale da rendere impraticabile l'analisi in computer di uso comune. Per affrontare questa problematica sono state sviluppate delle infrastrutture informatiche specifiche, che mettono insieme le capacità di tante macchine contemporaneamente per aumentare il carico di lavoro possibile, chiamate *cluster* di computer. Inoltre, si è reso necessario lo sviluppo di *software* specifici per la gestione di queste infrastrutture, che le riuscissero a sfruttare al meglio nell'ambito dell'analisi dati.

In questo lavoro si vuole offrire una panoramica sugli strumenti e le tecniche utilizzate per analizzare grosse moli di dati, con particolare attenzione alle tecniche per la stima di modelli lineari e per la definizione di intervalli di previsione. Verranno anche presentate due applicazioni, rispettivamente su dati reali e simulati, per mostrare l'implementazione delle tecniche e degli algoritmi descritti in un ambiente *cloud*.

Nel primo capitolo si presentano i principali strumenti e metodi per l'analisi che vengono utilizzati quando la mole dei dati è tale da richiedere approcci e soluzioni specifiche. Si introduce il concetto di calcolo parallelo, cioè l'esecuzione contemporanea di parti di uno stesso compito su diverse unità di calcolo al fine di aumentare l'efficienza nell'esecuzione complessiva, inoltre vengono considerate le diverse architetture distribuite che permettono di sfruttare i vantaggi del calcolo parallelo, con attenzione particolare ai *cluster* di computer accessibili tramite ambienti *cloud*. Successivamente vengono introdotti MapReduce, un paradigma di programmazione che permette di scrivere programmi da svolgere in parallelo su architetture distribuite, e Spark, un sistema di gestione del flusso di lavoro in un *cluster* che rappresenta un'evoluzione di MapReduce e ne estende le potenzialità e gli ambiti applicativi.

Il capitolo 2 prende in considerazione il modello lineare, un modello statistico dalle origini storiche ma tuttora molto utilizzato nelle applicazioni di analisi. Vengono presentati e confrontati i vari algoritmi di stima a disposizione e le varianti sviluppate quando i dati sono troppo grandi per essere caricati interamente in memoria. Tra queste se ne individua una che è possibile implementare secondo una logica MapReduce, che verrà utilizzata per la stima dei modelli nelle analisi finali. Si presenta anche il modello additivo, che mediante il concetto di espansione in funzioni di base può essere visto come un'estensione del modello lineare. Per concludere, si prende in considerazione il problema dell'inferenza dell'ambito dei modelli lineari, con particolare riferimento alla costruzione di intervalli di previsione. Per affrontare questo problema, viene introdotto il paradigma della *conformal inference*, una tecnica che permette di costruire intervalli di previsione con copertura empirica garantita sulla base di qualsiasi modello e senza particolari assunzioni.

Nel terzo e ultimo capitolo vengono prese in considerazione delle applicazioni di apprendimento supervisionato su due insiemi di dati, dove si mettono in pratica alcune delle tecniche esposte precedentemente. Il primo riguarda gli annunci di affitto sulla piattaforma Airbnb, scaricati dal sito Inside Airbnb, mentre il secondo consiste in dati simulati relativi alla rilevazione di resti di mura antiche in campioni di terreno. L'analisi viene svolta in ambiente *cloud* utilizzando un *cluster* virtuale, gestito tramite PySpark, l'interfaccia per Apache Spark implementata in Python.

Capitolo 1

Strumenti e metodi per l'analisi di dati

In questo capitolo si prende in considerazione il problema dell'analisi dati, o *data mining*, in contesti moderni. Prima si presentano i problemi computazionali che emergono quando la mole di dati da analizzare è massiva, poi gli strumenti di calcolo sviluppati per contrastarli. Successivamente ci si focalizzerà su MapReduce, un paradigma di programmazione nato per sfruttare le capacità dei nuovi strumenti di calcolo, per finire valutando alcune sue estensioni, in particolare Spark, un sistema di gestione del flusso di lavoro attualmente molto diffuso nel mondo dei servizi. Per il materiale presente in questo capitolo si rimanda al capitolo 2 di Leskovec et al. (2020), che copre svariati aspetti dell'applicazione delle procedure di *data mining* a moli di dati enormi, mentre riferimenti più specifici sui vari *software* verranno forniti nelle sezioni dedicate.

1.1 Il *data mining* in tempi moderni

In seguito all'avvento dei personal computer è cominciata una nuova fase di innovazione tecnologica che ha coinvolto e modificato il modo di lavorare in un vasto numero di ambiti, dalla scienza e tecnologia fino alle pratiche aziendali e alla vita di tutti i giorni. Una conseguenza di questa evoluzione è stata la possibilità di registrare e immagazzinare una mole sempre più grande di informazione. Questo è stato reso possibile dallo sviluppo di metodi automatici per accumulare dati e dal miglioramento dei sistemi elettronici di immagazzinamento, con una conseguente riduzione dei costi.

La capacità di registrare grosse moli di dati apre una serie di opportunità legate all'estrapolazione dell'informazione contenuta in essi, da utilizzare per fini scientifici o

produttivi. Specularmente, la gestione di una tale quantità di informazione comporta diversi problemi derivanti dalle caratteristiche dei dati stessi. Questi possono essere strutturati, ad esempio in formato tabulare, ma anche molto poco strutturati, come testi scritti, immagini, audio o video. Un altro aspetto rilevante in questo contesto è la dimensione dei dati, che può essere tale da rendere impossibili le operazioni di analisi senza strumenti di calcolo specifici. Spesso dati con queste caratteristiche vengono chiamati *Big Data*, per mole e complessità.

La disciplina che si occupa dell'estrazione dell'informazione da questo tipo di insiemi di dati è detta *data mining*, e si basa su tecniche e metodologie provenienti da diversi settori scientifici quali la statistica, il *machine learning*, la gestione di *database* e in generale lo sviluppo del *software*. Un altro termine spesso utilizzato in questo ambito è *data science*, ma il concetto rimane lo stesso: utilizzare le risorse *hardware* più potenti, i sistemi di programmazione migliori, gli algoritmi più efficienti per risolvere problemi in scienza, commercio, sanità, amministrazione pubblica, scienze umane e molti altri campi di studio sulla base dell'analisi di insiemi di dati complessi.

1.1.1 Obiettivi del *data mining*

Mantenendo la massima generalità, l'obiettivo del *data mining* è quello di sviluppare algoritmi in grado di utilizzare i dati a disposizione per affrontare problemi di interesse. Questa definizione permette di inglobare in questa pratica procedure di indicizzazione di pagine Web, la costruzione di sistemi di raccomandazione in grado di suggerire possibili prodotti graditi agli utenti di un certo servizio, o ancora la costruzione di regole associative sulla base dell'osservazione di gruppi di prodotti frequenti, per citare alcuni esempi.

Una branca fondamentale delle procedure di *data mining*, sviluppata sulla base delle tecniche di modellazione proprie della statistica e sugli approcci di analisi dati sviluppati nell'ambito della tecnologia dell'informazione, è quella dell'apprendimento statistico. Al suo interno si distinguono problemi di apprendimento supervisionato, che mirano a prevedere e studiare le caratteristiche di una variabile risposta sulla base di un insieme di variabili fisse, e di apprendimento non supervisionato, dove l'obiettivo è quello di studiare le caratteristiche interne di un insieme di dati. In questo lavoro ci si concentrerà su temi relativi all'apprendimento supervisionato.

1.1.2 *Machine learning* e modellazione statistica

In molte applicazioni pratiche di *data mining*, l'obiettivo è quello di sviluppare un modello previsivo sulla base dei dati a disposizione. In generale, un modello è una rappresentazione semplificata di un fenomeno d'interesse, funzionale ad un obiettivo specifico. In alcuni contesti, è possibile ipotizzare l'esistenza di un vero modello che regola il fenomeno di interesse, ad esempio in ambiti collegati alle scienze fisiche, ma spesso i modelli utilizzati nella pratica hanno funzioni puramente operative e puntano solo ad approssimare in maniera soddisfacente il meccanismo generatore dei dati. Si rimanda al capitolo introduttivo di Azzalini & Scarpa (2012) per una trattazione più estesa sul concetto di modello e in generale sull'utilizzo di modelli matematici in ambiti applicativi.

Dal punto di vista della modellazione statistica classica la costruzione di un modello prevede la specificazione di una distribuzione per i dati osservati, indicizzata da un numero finito o infinito di parametri e identificabile nella famiglia delle distribuzioni possibili sulla base di un campione. Le tecniche sviluppate per la stima dei parametri che identificano la distribuzione specificata originano dal calcolo della probabilità e sono associate a diverse proprietà legate alla qualità dei risultati, valide sotto alcune assunzioni relative al processo generatore dei dati.

Il *machine learning* è una branca della *computer science* che si occupa dei problemi di *data mining* secondo un approccio più informatico rispetto alla modellazione statistica. In questo contesto i modelli vengono visti come oggetti da "allenare" applicando appropriati algoritmi di stima ai dati disponibili, per poter successivamente fornire previsioni in corrispondenza di nuove osservazioni. La differenza principale con la modellazione statistica classica è che in questa solitamente si cerca di specificare un modello per il fenomeno in questione che sia plausibilmente realistico, ossia in qualche modo aderente ad un ipotetico meccanismo generatore dei dati. Al contrario, nel *machine learning* i modelli hanno solo scopi predittivi e non ambiscono a spiegare correttamente il fenomeno, ma a replicare i *pattern* osservati sui dati di stima per generare delle previsioni attendibili. Si consideri come esempio l'albero di regressione, che punta a rappresentare la funzione di regressione $f(X)$ (che si può solitamente ipotizzare come continua) con una funzione costante a tratti: la rappresentazione non è realistica e non ci si può aspettare che il modello sia correttamente specificato, ma in alcuni casi le previsioni potrebbero essere più attendibili rispetto ad altri modelli più "realistici". In questo contesto raramente i modelli permettono l'interpretazione delle diverse variabili esplicative, e spesso si sente parlare approccio *black box* (cioè a scatola chiusa, non intrepretabile).

1.2 Architetture per l'analisi

Le applicazioni moderne di *data mining*, spesso chiamate analisi di *Big Data*, prevedono la gestione e l'elaborazione di grosse moli di dati in tempi relativamente brevi. In molte applicazioni, i dati sono estremamente regolari (tutti con lo stesso formato e ben immagazzinati) e c'è la possibilità di sfruttare il parallelismo per velocizzare l'analisi. Per fare questo sono necessari nuovi strumenti computazionali, sia di tipo *hardware* che di tipo *software*.

1.2.1 Approcci alla parallelizzazione

Il contesto è quello in cui si hanno a disposizione diverse unità fondamentali di calcolo (processore, memoria principale e disco fisso) e si vogliono ricercare dei metodi per velocizzare le operazioni da eseguire facendole svolgere contemporaneamente ai diversi processori. In ambito informatico questa pratica viene chiamata parallelizzazione, e si compone di tre diversi approcci: parallelismo a livello di dati, di algoritmo o di operazioni.

La parallelizzazione a livello di dati è applicabile quando le operazioni richieste possono essere divise in blocchi, per cui non è necessario che un processore utilizzi o conservi tutti i dati. Si pensi ad esempio ad un'operazione semplice come il calcolo della media degli elementi di un vettore: per la proprietà associativa è possibile dividere il vettore in blocchi, assegnare ogni blocco ad un nodo e ottenere le somme parziali, da aggregare alla fine prima di dividere per la lunghezza del vettore originale. Questo tipo di parallelismo è implementato in alcune librerie per il calcolo algebrico, ma spesso viene progettato e controllato dall'utente in maniera esplicita.

Un altro tipo di parallelismo che viene controllato in maniera esplicita è la parallelizzazione a livello di algoritmo, utilizzata per algoritmi che nascono per essere parallelizzati, come ad esempio cicli indipendenti, procedure di convalida incrociata o simili. Andando nel dettaglio, a differenza di prima ogni processore mantiene e utilizza tutti i dati per calcolare una parte specifica dell'algoritmo, e l'aggregazione finale dei risultati avviene in un secondo momento. Per la necessità di immagazzinare tutti i dati in ogni unità di calcolo, questo approccio risulta inapplicabile quando la mole di dati è eccessivamente elevata.

L'ultimo approccio alla parallelizzazione è quello a livello di operazioni. Questo termine si riferisce alla parallelizzazione di operazioni algebriche di base, come prodotti matriciali, inversioni o scomposizioni. Tale approccio lavora ad un livello più basso dei

precedenti e spesso non viene controllato dall'utente, ma implementato all'interno di librerie di base presenti di *default* nei sistemi operativi.

Nella pratica, per sfruttare le potenzialità del calcolo parallelo è stato sviluppato un nuovo *software stack*, cioè un'insieme di strumenti *hardware* e *software* disegnati per gestire la conservazione e la manipolazione dei dati tra le diverse unità fondamentali di calcolo.

1.2.2 *Cluster* di computer

Si consideri un'unità di calcolo fondamentale come descritta nella sotto-sezione precedente, cioè costituita da un singolo processore, con la sua memoria principale e il disco fisso locale. Tale unità viene detta nodo di calcolo. Nel passato le applicazioni che necessitavano del calcolo parallelo, come calcoli scientifici su larga scala, venivano effettuate su grandi computer appositamente costruiti, con molti processori e *hardware* specializzato. In tempi recenti, la prevalenza di servizi di larga scala basati sul Web ha modificato il modo in cui vengono svolte questo tipo di operazioni: sempre più di frequente si utilizzano infatti installazioni di molti (centinaia o migliaia) nodi elementari, che lavorano in maniera più o meno indipendente. In queste installazioni, i nodi di calcolo sono formati da *hardware* di uso comune, il che riduce molto i costi in confronto alle grandi macchine specificamente progettate per il calcolo parallelo che venivano utilizzate in passato. Lo sviluppo delle infrastrutture di calcolo descritte ha reso necessaria la nascita di una nuova generazione di sistemi di programmazione, che sfruttano i vantaggi della parallelizzazione e allo stesso tempo gestiscono i problemi che originano dal fatto che il sistema è formato da molte di unità di calcolo, ognuna delle quali potrebbe in ogni momento causare un errore.

Queste nuove architetture distribuite, chiamate *cluster* di computer, sono organizzate come di seguito. I nodi elementari di calcolo sono organizzati in *rack*, con un numero variabile tra 8 e 64 per ciascuno. I nodi entro un singolo *rack* sono collegati da una rete fisica, tipicamente di tipo Ethernet. In un *cluster* ci possono essere diversi *rack*, a loro volta collegati da una rete fisica. Per sistemi di questo tipo, formati da un numero elevato di componenti, gli errori nell'esecuzione saranno frequenti, in quanto ogni nodo di calcolo potrebbe arbitrariamente andare incontro a fallimenti. Nello specifico, le problematiche più diffuse sono la perdita di un singolo nodo (ad esempio si rompe il disco locale) o la perdita di un intero *rack* (perché si interrompe la rete che collega i nodi al suo interno e con il resto del sistema).

Alcune operazioni possono impiegare minuti o ore anche su *cluster* con migliaia di nodi, pertanto è fondamentale avere degli strumenti di controllo per accertarsi che

il calcolo non debba ripartire dall'inizio ogni volta che una componente del sistema riporta un errore. Questo problema viene risolto con due approcci. Il primo prevede che i documenti debbano essere salvati con ridondanza, cioè replicati tra diversi nodi. Se non si facesse così, il fallimento di un nodo comporterebbe l'irreperibilità dei *files* in esso contenuti fino alla sua sostituzione, con una perdita permanente nel caso non si fossero effettuati salvataggi secondari. Il secondo approccio riguarda l'organizzazione del calcolo, che deve essere divisa in passaggi sequenziali (chiamati compiti, o in inglese *tasks*), in modo che se un *task* non arriva a completamento, può essere fatto ripartire in maniera indipendente dagli altri.

1.2.3 *File system* distribuiti

Per sfruttare le possibilità del *cluster computing*, i documenti e *files* devono essere immagazzinati e gestiti in maniera diversa rispetto a come operano i *file systems* dei computer di uso comune. Questo tipo di sistema di gestione dei documenti viene chiamato *distributed file system* (DFS), e viene utilizzato quando i documenti sono enormi (nell'ordine dei terabyte) e vengono aggiornati raramente.

In questi sistemi, i *files* vengono divisi in blocchi (detti *chunks*), tipicamente da 64 megabytes. I blocchi vengono replicati su diversi nodi di calcolo. Inoltre, i nodi che mantengono copie di uno stesso blocco sono posizionati su *rack* diversi, in modo da non perdere tutte le copie a causa di un errore a livello di *rack*. Solitamente, sia la dimensione dei blocchi che il livello di replicazione vengono controllati dall'utente.

Per identificare i blocchi di un documento, c'è un altro piccolo *file* chiamato *master node* o *name node* del documento in questione, che mantiene gli indirizzi di tutte le copie del *file* nei vari dischi. Questo *file* viene esso stesso replicato, e un'ulteriore cartella contiene le informazioni su dove trovare le copie.

1.2.4 *Cloud computing*

L'accesso al nuovo *software stack* sviluppato per l'analisi di dati viene garantito agli utenti finali (ricercatori, aziende, pubblica amministrazione, ...) dai fornitori di servizi *cloud*. Questo è una collezione di server globali, collegati tramite Internet e utilizzati per conservare, gestire e processare dati. Il *cloud computing* utilizza l'Internet come un ambiente di calcolo di larga scala per fornire grandi risorse di calcolo ai clienti finali. Le principali caratteristiche di questa pratica che ne garantiscono la popolarità sono i bassi costi, la scalabilità, la velocità di fornitura dei servizi, la flessibilità e la possibilità di accesso globale.

Attualmente, nel contesto del *cloud computing*, ad un utente non viene richiesto di acquistare e possedere tutte le risorse richieste; egli può infatti richiedere ed utilizzare le risorse di cui ha bisogno dal *cloud* e pagare in base all'utilizzo. Questo risulta molto comodo quando si ha bisogno di certe risorse, per esempio macchine di calcolo particolarmente potenti, solo temporaneamente. L'architettura flessibile dei sistemi di questo tipo permette inoltre di caricare e accedere a grosse moli di dati da qualunque luogo e in ogni momento.

Un'altra caratteristica dei sistemi di *cloud computing* è la possibilità di virtualizzazione delle macchine: i server ad alta capacità possono essere usati da più utenti contemporaneamente. Al posto di intere macchine fisiche, all'utente vengono fornite delle macchine virtuali, cioè delle partizioni logiche di grandi macchine fisiche, che riducono i costi offrendo grandi capacità di calcolo e di memoria. Nel Web si può accedere a numerosi servizi di *cloud computing*, più o meno specializzati in base alle necessità degli utenti. Fra i principali si citano quello offerto da Google, *Google Cloud Platform* (GCP), e quello di Amazon, *Amazon Web Services* (AWS). Per le operazioni di analisi di cui ci si occupa in questo lavoro, i servizi da richiedere a questi provider sono la capacità di memoria, le risorse di calcolo e la possibilità di organizzare e gestire le macchine virtuali entro a dei *cluster*.

1.3 MapReduce

MapReduce è un paradigma di programmazione introdotto da Google per scrivere applicazioni di analisi di dati con dimensioni massive. È stato sviluppato come un generico *framework* di calcolo parallelo e distribuito, che permette l'analisi di grandi moli di dati su *cluster* di macchine in una maniera che risulta robusta rispetto a possibili fallimenti a livello di *hardware*. Per ulteriori riferimenti si rimanda a Dean & Ghemawat (2008), l'articolo che introduce questo paradigma di programmazione e l'implementazione interna di Google.

1.3.1 Funzionamento di un programma MapReduce

Uno dei vantaggi di MapReduce è che l'utente deve occuparsi solo di definire le operazioni che andranno eseguite sui dati, senza il bisogno di gestire la comunicazione tra i vari nodi del cluster o il trasferimento dei dati da e sul sistema di gestione di *files* distribuito. Nello specifico, l'utente definisce le funzioni Map e Reduce, e il sistema coordina l'esecuzione in parallelo e gestisce i possibili errori nell'esecuzione.

I *files* di input per un compito di tipo Map consistono di elementi che possono essere di ogni tipo (tuple o documenti), salvati nei blocchi di un DFS. Tecnicamente, tutti gli input dei compiti di tipo Map e gli output dei compiti di tipo Reduce hanno il formato chiave-valore, ma normalmente le chiavi degli elementi di input non sono rilevanti e vengono ignorate. Questa regola per il formato dei dati in ingresso e in uscita permette di combinare diversi processi di tipo MapReduce sequenzialmente. La funzione Map prende un elemento di input come argomento e produce zero o più coppie chiave-valore. Il tipo di chiavi e valori è arbitrario, e non viene richiesto che le chiavi siano uniche: in realtà spesso vengono prodotte diverse coppie con la stessa chiave.

Una volta che tutti i compiti di tipo Map sono stati eseguiti, le coppie chiave-valore vengono raggruppate per chiave, e i valori associati con ciascuna chiave sono aggregati in una singola lista. A questo punto viene applicata la funzione Reduce, che prende come argomenti una coppia costituita da una chiave e dalla lista di valori associati a quella chiave e produce, come la funzione Map, zero o più coppie chiave-valore. Solitamente l'operazione eseguita è una qualche aggregazione della lista di valori per ogni chiave. L'applicazione di una funzione Reduce a una singola chiave e alla lista di valori associata è detta *reducer*. Un compito Reduce riceve una o più chiavi e le loro liste di valori, quindi esegue uno o più *reducers*. I risultati di tutti i compiti di tipo Reduce vengono quindi uniti in un singolo documento. In Figura 1.1 si riporta una schematizzazione della procedura descritta.

A titolo di esempio, si consideri un programma in grado di contare il numero di occorrenze di ogni parola all'interno di un documento. In quest'applicazione la funzione Map legge il documento e lo spezza in una sequenza di parole $w_1, \ldots, w_n$. Successivamente la funzione restituisce le coppie chiave-valore $(w_1, 1) \ldots, (w_n, 1)$. Per ogni chiave, quindi per ogni parola, vengono raggruppati tutti i valori, e la funzione Reduce calcola semplicemente la somma, ottenendo il conteggio di quella parola nel documento.

In certe applicazioni, come quella appena descritta, la funzione Reduce è associativa e commutativa, cioè i valori possono essere combinati in ogni ordine, ottenendo lo stesso risultato. In questi casi, si può spostare una parte del lavoro dei *reducers* ai compiti Map. Nell'esempio del conteggio di parole, si pensi che il *file* di testo sia diviso in blocchi di righe e ogni compito di tipo Map legga un blocco. Si noti che per una parola w_i ripetuta, ci saranno a questo punto diverse copie $(w_i, 1)$. Si potrebbe quindi applicare la funzione Reduce entro ogni compito Map, prima che i risultati vengano raggruppati e aggregati per il compito Reduce. Questa funzione di aggregazione intermedia, che spesso coincide con la funzione Reduce, è detta *combiner*. Nel caso dell'esempio precedente, utilizzando questo approccio alla fine di un compito Map si avrebbero delle coppie

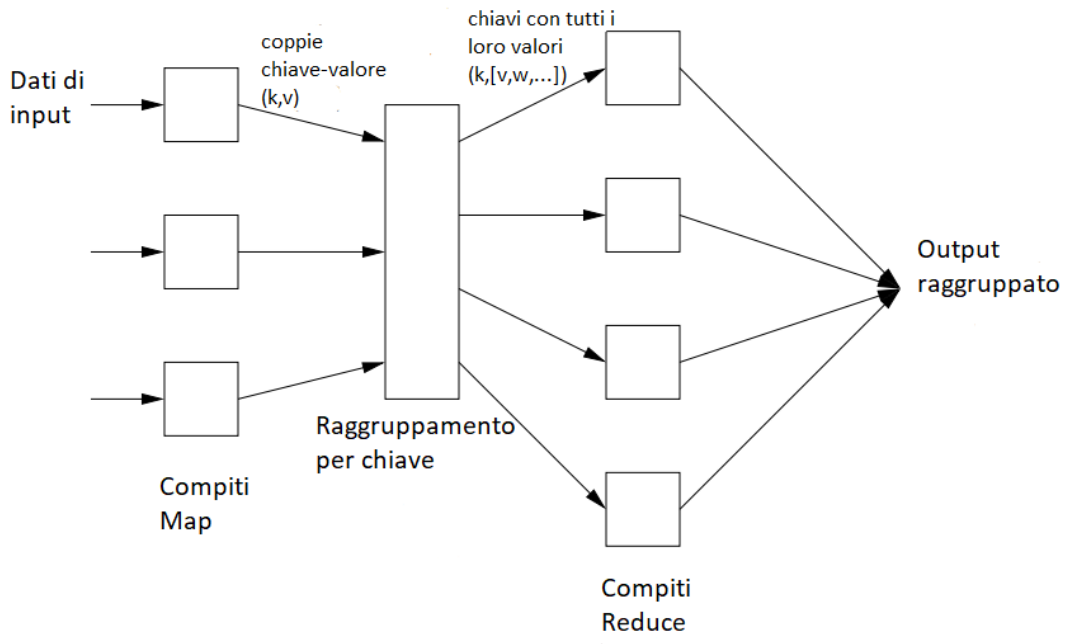

FIGURA 1.1: Rappresentazione schematica di un programma MapReduce.

chiave-valore (w, m), dove m rappresenta il conteggio di quella parola nella parte di documento associata a quello specifico compito di tipo Map. A questo punto si procede come prima, raggruppando le chiavi, aggregando i valori associati e passando i risultati ai compiti di tipo Reduce. Questa pratica a volte permette di rendere molto più efficienti gli algoritmi implementati, visto che tipicamente vengono programmati più compiti di tipo Map che di tipo Reduce.

1.3.2 Dettagli dell'esecuzione di un programma MapReduce

Si considera ora in dettaglio come viene eseguito un programma MapReduce su un *cluster* di calcolo, focalizzandosi su come interagiscono i vari processi, compiti e documenti. Sfruttando una libreria fornita da un sistema di tipo MapReduce, il programma definito dall'utente definisce tra i diversi nodi un *master* e vari *workers* (quest'operazione viene definita *fork*). Normalmente, un *worker* gestisce compiti solo di tipo Map o solo di tipo Reduce. Il *master* crea i diversi compiti Map e Reduce, in numero solitamente specificato dal programma dell'utente, e li assegna ai diversi *workers*. Risulta sensato in questo contesto creare tanti compiti Map quanti sono i blocchi del *file* di input, mentre solitamente si creano meno compiti di tipo Reduce. Il motivo è che ogni compito di tipo

Map crea un *file* intermedio per ogni compito Reduce, e se ci sono troppi compiti di questo tipo il numero di *files* intermedi potrebbe esplodere.

Il nodo *master* tiene traccia dello stato di ogni compito di tipo Map o Reduce, cioè in attesa (in inglese *idle*), in esecuzione su uno specifico *worker*, o completato. Un nodo *worker* riporta al *master* quando finisce un compito, e subito il *master* gli assegna un nuovo compito.

Ogni compito di tipo Map lavora su uno o più blocchi del *file* di input, eseguendo il codice scritto dall'utente. Successivamente viene creato un *file* intermedio con i risultati per ogni compito di tipo Reduce, sul disco locale del nodo che esegue il compito di tipo Map. Quando il nodo *master* assegna un compito di tipo Reduce ad un nodo, questo riceve tutti i *files* necessari al calcolo, esegue il codice scritto dall'utente e scrive i risultati delle operazioni su un *file* interno al DFS su cui si appoggia il programma. Le operazioni in atto a livello di sistema al momento dell'esecuzione di un programma di tipo MapReduce vengono schematizzate in Figura 1.2.

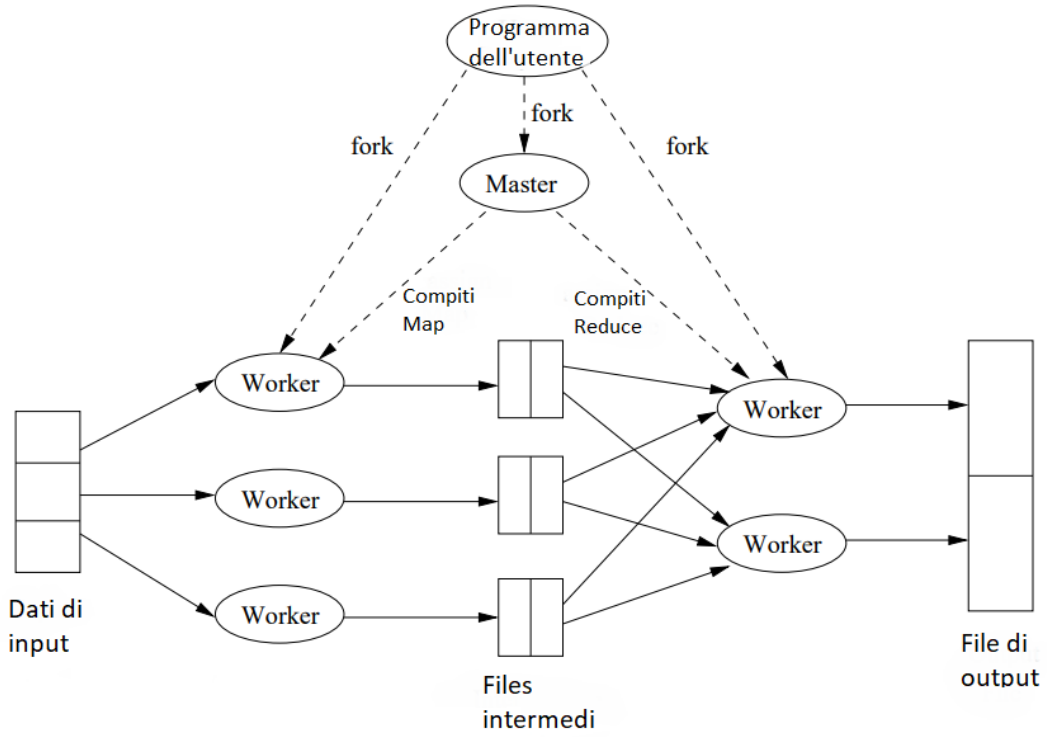

FIGURA 1.2: Esecuzione di un programma MapReduce all'interno di un *cluster*.

Relativamente ai possibili problemi che si possono incontrare durante l'esecuzione di un programma MapReduce, l'eventualità peggiore è che il nodo a cui viene assegnata la funzione di *master* si rompa. In questo caso l'intero lavoro deve ripartire dall'inizio.

Tuttavia, questo è l'unico nodo che può portare ad un fallimento dell'intero processo: altri tipi di errori vengono gestiti dal *master*, che permette a completare l'esecuzione di tutto il lavoro nonostante eventuali problemi a qualche nodo. Si supponga che un *worker* a cui sono assegnati compiti di tipo Map si rompa. Questo errore viene rilevato dal *master*, che controlla periodicamente lo stato dei vari *workers*. Tutti i compiti di tipo Map effettuati da quel nodo vanno ripetuti, anche quelli già eseguiti (il loro risultato viene salvato temporaneamente sul nodo). Il *master* cambia lo stato dei compiti Map in *idle* e li programma su un nuovo *worker* una volta disponibile, inoltre aggiorna i compiti di tipo Reduce programmati sulla nuova posizione dei loro *files* di ingresso. Gestire i possibili fallimenti di un *worker* che esegue compiti di tipo Reduce è ancora più semplice: il *master* cambia semplicemente lo stato del compito in *idle* e lo programma su un nuovo *worker*.

1.3.3 Implementazioni disponibili e principali applicazioni

Storicamente, MapReduce è stato introdotto sia come paradigma di programmazione che come implementazione vera e propria da Google. Una versione *open-source* molto popolare è Hadoop MapReduce, che può essere ottenuto insieme con HDFS (*Hadoop distributed file system*) dalla Apache Foundation. Un'altra implementazione open-source è la libreria di Python MRJob, sviluppata da YELP. Per maggiori informazioni si rimanda ai siti dedicati, rispettivamente `https://hadoop.apache.org` e `https://github.com/Yelp/mrjob`.

Come paradigma di programmazione, MapReduce non è la soluzione a tutti i problemi: non tutti gli algoritmi infatti possono beneficiare delle possibilità della parallelizzazione e non tutti gli algoritmi parallelizzabili possono essere scritti con questo stile. Il motivo originale dietro all'implementazione introdotta da Google era quello di eseguire moltiplicazioni tra matrici e vettori molto grandi, nell'ambito del calcolo di PageRank (l'algoritmo di indicizzazione di pagine Web utilizzato al tempo da Google). Operazioni di questo tipo, quindi moltiplicazioni tra matrici e vettori o tra due matrici, si prestano facilmente all'implementazione in programmi MapReduce. Un'altra importante classe di operazioni che possono sfruttare questo paradigma traendo un grande vantaggio in termini di efficienza sono le operazioni di base dell'algebra relazionale (selezioni, proiezioni, unioni, ...).

1.4 Da MapReduce a Spark

Fin dalla sua introduzione, MapReduce si è rivelato molto influente, al punto da dare origine ad un grande numero di estensioni e modifiche. Questi sistemi presentano una serie di caratteristiche che li accomunano alle implementazioni di MapReduce: sono costruiti su un *file system* distribuito, gestiscono un elevato numero di compiti che sono esecuzioni di un numero ridotto di funzioni scritte dall'utente e forniscono metodi per la gestione dei possibili errori che avvengono durante l'esecuzione di un lavoro, senza il bisogno di farlo ripartire.

Tra queste estensioni, ci si concentra sui sistemi di *workflow*, che estendono MapReduce supportando grafi aciclici di funzioni, ognuna costituita da una serie di *tasks*. Molti sistemi di questo tipo sono stati sviluppati negli anni, tra questi uno dei più popolari è certamente Spark, sviluppato da UC Berkeley. Per riferimenti specifici su Spark si rimanda a Zaharia et al. (2012) e Zaharia et al. (2016), che dove vengono trattati i principali elementi di innovazione introdotti da questo sistema.

1.4.1 Sistemi di *workflow*

I sistemi di *workflow* estendono MapReduce dal semplice flusso di lavoro a due passaggi (la funzione Map fornisce i dati di ingresso per la funzione Reduce) a una qualsiasi collezione di funzioni, con un grafo aciclico che rappresenta il flusso di lavoro tra queste funzioni. Esiste quindi un grafo di flusso i cui archi $a \to b$ rappresentano il fatto che l'ouput della funzione a corrisponde all'input della funzione b.

I dati passati da una funzione all'altra sono *files* con elementi tutti dello stesso tipo. Se una funzione ha un solo argomento di input, allora viene applicata in maniera indipendente a tutti gli elementi di input, nella stessa maniera delle funzioni Map e Reduce, e i risultati vengono scritti su un *file* intermedio che viene mandato alle funzioni successive. Se una funzione riceve input da più di un file, gli elementi di ciascun *file* possono essere combinati in vari modi, e la funzione applicata a tali combinazioni. In Figura 1.3 si può vedere un esempio di quanto detto: alcune funzioni prendono input singoli e alcuni multipli, ma il grafo risulta aciclico (parte dagli input delle funzioni f e h e finisce con l'output della funzione j).

Come le funzioni Map e Reduce, ogni funzione di un *workflow* può essere eseguita da molti *tasks*, dove ad ognuno viene assegnata una porzione del *file* di ingresso. Si pensi alla funzione Map, che dà luogo a un numero di compiti Map idealmente pari al numero di blocchi del *file* di input. Un *master controller* è responsabile della suddivisione del lavoro tra i vari *tasks* che implementano la funzione. Come per i compiti Map,

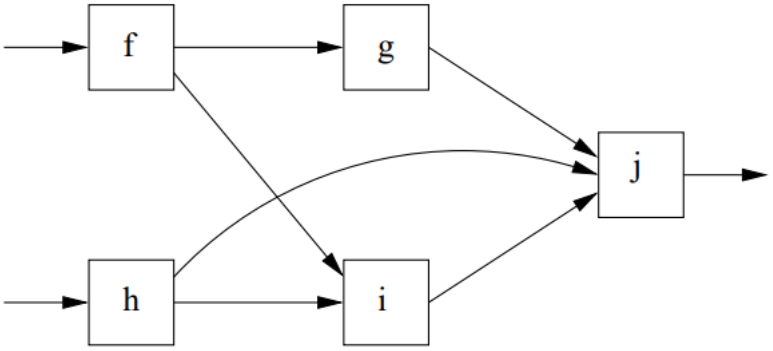

FIGURA 1.3: Esempio di *workflow* che estende il funzionamento di un programma MapReduce

ogni compito che implementa la funzione f ha un *file* di uscita con i dati destinati a ogni compito che implementa le funzioni successive ad f. Una volta che un compito è terminato, il *master controller* si occupa della migrazione di risultati dati intermedi.

C'è un'importante proprietà che le funzioni di un *workflow*, e quindi i compiti che le implementano, condividono con i compiti di un programma MapReduce: la *blocking property*, cioè il fatto di mandare il proprio output ai compiti successivi solo dopo il completamento. Come conseguenza quando un compito fallisce, non manderà risultati intermedi a nessuno dei sui successori nel grafo di flusso. Il *master controller* può quindi far ripartire il compito in questione su un altro nodo di calcolo, senza duplicazioni o corruzioni dei *file* intermedi di output.

Alcune applicazioni dei sistemi di *workflow* consistono di sequenze "a cascata" di lavori di tipo MapReduce. Anche in questo semplice caso, c'è un vantaggio importante nell'implementare tali sequenze come un singolo flusso di lavoro: la gestione e il trasferimento dei dati intermedi viene affidata al *master controller*, senza il bisogno di scrivere il *file* temporaneo, che consiste nell'output di un lavoro MapReduce, sul *file system* distribuito. Inoltre, assegnando i compiti ai nodi di calcolo che hanno già una copia dei dati di input, si evita molta della comunicazione che sarebbe necessaria se si salvassero i risultati di un primo lavoro MapReduce prima di iniziarne un secondo.

1.4.2 Spark: elementi di base

Spark è fondamentalmente un sistema di *workflow*. Tuttavia, viene definito come un'evoluzione dei primi sistemi di *workflow* sviluppati, per una serie di caratteristiche: maggior efficienza nella gestione dei fallimenti, maggior efficienza nella programmazione dell'esecuzione delle funzioni tra i nodi di calcolo e integrazione di elementi dei linguaggi di programmazione come cicli e librerie specifiche aggiuntive.

L'astrazione di dati fondamentale per Spark è chiamata RDD, *Resilient Distributed Dataset*. Un RDD è un *file* con oggetti di uno stesso tipo. Un primo esempio di RDD è il *file* con le coppie chiave-valore usato nei sistemi MapReduce. In generale, gli RDD sono i *file* che vengono passati tra le varie funzioni del *workflow system*, come visto nella sotto-sezione precedente. Queste astrazioni si dicono distribuite, in quanto un RDD è solitamente suddiviso in blocchi che vengono mantenuti su diversi nodi di calcolo, e resilienti, nel senso che ci si aspetta di riuscire a recuperarne il contenuto a dispetto della perdita di uno o più blocchi. A differenza dell'astrazione chiave-valore usata dai sistemi MapReduce, non ci sono restrizioni sul tipo di elementi che formano un RDD.

Un programma Spark consiste in una sequenza di passaggi, ciascuno dei quali applica tipicamente una qualche funzione ad un RDD per produrre un altro RDD. Operazioni di questo tipo vengono chiamate trasformazioni. Si può anche prendere i dati da un *file system* distribuito, come HDFS, trasformarli in RDD, applicare qualche funzione e restituire il risultato al programma oppure copiarlo sul *file system*. Questo tipo di operazioni prendono il nome di azioni. In seguito verranno prese in considerazione alcune operazioni di base che permettono di scrivere programmi di tipo MapReduce.

La trasformazione Map prende una funzione come parametro di ingresso, e la applica a tutti gli elementi di un RDD, producendo un altro RDD. L'operazione ricorda la funzione Map del paradigma MapReduce, ma ha alcune differenze fondamentali: in MapReduce, la funzione Map prende in input coppie di tipo chiave-valore e produce risultati dello stesso tipo. In Spark, una funzione Map può essere applicata ad oggetti di qualsiasi tipo e produce esattamente un singolo oggetto come risultato. Questo oggetto potrebbe essere una lista con elementi di tipo chiave-valore, ma per produrre oggetti multipli per ogni input bisogna utilizzare la trasformazione Flatmap. Questa è analoga a Map di MapReduce, senza il vincolo di restituire tutti oggetti del tipo chiave-valore. In Spark si trova anche un'operazione simile ad una forma limitata di Map, chiamata Filter. Invece di una funzione come parametro, questa trasformazione prende un predicato logico applicabile agli elementi del RDD di interesse. Il risultato della trasformazione consiste in un RDD con tutti gli oggetti per i quali il predicato logico è verificato.

L'operazione Reduce, in Spark, è un'azione, non una trasformazione. Questo vuol dire che, applicata ad un RDD, restituisce un valore e non un altro RDD. Questa operazione ha come argomento di ingresso una funzione che prende due elementi di un tipo e ne restituisce uno dello stesso tipo. Quando applicata ad un RDD, viene applicata ripetutamente alle coppie di elementi consecutivi, riducendole ad un singolo elemento, che viene restituito come risultato dell'operazione Reduce.

1.4.3 Spark: dettagli dell'implementazione

A livello tecnico c'è una similarità fondamentale tra Spark e altri sistemi di tipo MapReduce: la gestione degli RDD. Come visto per MapReduce infatti, Spark permette la suddivisione in blocchi degli RDD, qua chiamati *splits*. Ciascuno può essere passato a un nodo di calcolo differente, e questo permette di calcolare in parallelo una qualsiasi trasformazione per l'RDD in questione.

Oltre a questo, l'implementazione di Spark differisce per svariati motivi da Hadoop o altre implementazioni di MapReduce, tra i quali due miglioramenti molto importanti sono la *lazy evaluation* e il *lineage* degli RDD.

Un programma Spark non applica una trasformazione ad un RDD finché non viene esplicitamente richiesto, cioè finché non viene chiamata un azione sull'RDD risultante dalla trasformazione, come ad esempio salvare i risultati su disco o portarli in memoria per visualizzarli nel programma. Il vantaggio di questa strategia, chiamata *lazy evaluation*, è che si evita di costruire un gran numero di RDD. Si consideri la situazione dove si vuole scrivere un programma MapReduce per contare le occorrenze di ogni parola in un *file* di testo su Spark. Si supponga di aver recuperato da un DFS il *file* di interesse e di averlo caricato nel programma come un RDD, chiamato R_0. Si applica una funzione Map (o Flatmap) che crea le coppie $(w, 1)$ per ogni parola, ottenendo l'RDD R_1; questo vuol dire che per ogni *split* di R_0 viene create uno *split* del risultato R_1, nello stesso nodo di calcolo. In realtà, l'operazione Map non viene veramente calcolata finché non si applica un'azione a R_1, come ad esempio un'operazione Reduce che conta le occorrenze delle parole. Solo quando il programma raggiunge quest'azione Spark esegue la trasformazione Map che permette di ottenere R_1 da R_0, svolgendola in parallelo sui nodi di calcolo che contengono i blocchi di R_0. Questo vuol dire che i blocchi di R_1 esistono solo localmente nei nodi che li creano e, a meno che il programmatore non richieda esplicitamente di mantenerli, vengono eliminati una volta utilizzati localmente.

Il concetto di *lineage* è invece legato alla gestione di eventuali errori nell'esecuzione. Spark, come alternativa al salvataggio dei risultati intermedi, mantiene traccia del *lineage* (letteralmente discendenza) di ogni RDD che viene creato. Questo fornisce informazioni al programma su come ricreare un RDD o un suo blocco, se necessario. Nell'esempio precedente, il *linaege* di R_1 dice che è stato creato applicando ad R_0 una particolare trasformazione di tipo Map, a sua volta R_0 viene creato a partire da un particolare *file* nel DFS. Si pensi di aver perso uno *split* di R_1: Spark sa come ricrearlo dal corrispondente blocco di R_0. Considerando che l'errore potrebbe essere causato da una rottura del nodo, anche il blocco di R_0 di partenza potrebbe essere andato perso, tuttavia si può ricostruire questo blocco dal *file* dal *file system* distribuito, dove viene

salvato in maniera ridondante per evitare problemi. Quindi Spark trova un altro nodo di calcolo, ricostruisce lo *split* di R_0 dal *file system* e applica la trasformazione Map necessaria per recuperare finalmente lo *split* di R_1.

Capitolo 2

Modelli lineari e Big Data

In questo capitolo ci si focalizza sul modello lineare, un modello statistico risalente ai primi anni dell'800 ma ancora oggi molto utilizzato. Nello specifico si considerano svariati algoritmi disponibili per la stima e vengono presi in esame i problemi di carattere computazionale che possono emergere quando i dati da analizzare sono massivi; successivamente vengono esposti approcci alternativi alla stima in questi particolari contesti. Si considera poi il modello additivo, un modello più generale la cui stima si può ricondurre a quella di un modello lineare. Infine, si prende in considerazione il problema dell'inferenza statistica nei modelli lineari, concentrandosi sul problema dell'inferenza predittiva: dopo un'introduzione all'approccio classico basato sulla verosimiglianza, viene presentato un metodo in grado di garantire la copertura richiesta senza formulare assunzioni distributive per i dati, con qualsiasi modello utilizzato.

2.1 Apprendimento supervisionato e regressione

Si consideri il classico problema di apprendimento supervisionato: si ha una variabile dipendente Y, chiamata anche *variabile risposta* o semplicemente *output*, e un vettore di p variabili indipendenti $X = [X_1, \ldots, X_p]$, che si ritiene possano essere utili per spiegare o prevedere la variabile risposta. Le variabili indipendenti vengono anche chiamate *regressori*, *variabili esplicative*, o, per citare le terminologie inglesi utilizzate nell'ambito del *machine learning*, *features* o *inputs*. L'obiettivo dell'apprendimento supervisionato è quello di sviluppare dei metodi per prevedere la variabile risposta sulla base dei valori assunti dalle variabili esplicative. Per stimare la relazione tra risposta e predittori, si ha a disposizione un campione finito estratto da una popolazione potenzialmente infinita, spesso chiamato insieme di stima, vale a dire $(y_i, x_i) = (y_i, x_{i1}, \ldots, x_{ip})$, per $i = 1, \ldots, n$.

Le variabili esplicative possono essere di diverso tipo: variabili quantitative, cioè numeriche (e.g. la variabile *peso*, con valori osservabili nel dominio dei numeri reali positivi), o variabili qualitative che rappresentano un fattore discreto (e.g. la variabile *colore*, che assume valori *rosso*, *giallo* o *blu*). Le variabili qualitative si dicono dicotomiche se hanno solo due modalità, mentre se ne hanno più di due vengono dette politomiche. Un altro tipo di variabile è dato dalle qualitative ordinali, per le quali esiste un ordinamento tra i valori (e.g. la variabile *dimensione*, con modalità *piccolo*, *medio*, *grande*), ma non è possibile quantificare numericamente la differenza tra modalità contigue. Solitamente le variabili qualitative vengono rappresentate mediante codifiche numeriche, per facilitare il loro impiego nelle tecniche di apprendimento supervisionato. Per variabili qualitative dicotomiche, le due modalità vengono codificate come 0 e 1, ad indicare presenza o assenza di uno dei due livelli preso a riferimento. Questa è detta codifica ad angolo, mentre un'alternativa meno utilizzata è quella di codificare le due modalità come 1 e -1. Per le variabili qualitative politomiche esistono diversi approcci, ma il più semplice e utilizzato è quello dell'impiego di variabili *dummy* di supporto: per ogni modalità si definisce una variabile binaria che indica presenza o assenza di quella particolare modalità, estendendo l'approccio utilizzato per variabili dicotomiche.

La variabile risposta può a sua volta essere di diverso tipo, e in base a questo si caratterizza il problema di apprendimento supervisionato. Se la variabile risposta è quantitativa, il problema viene detto di regressione; se invece questa è una variabile qualitativa, l'obiettivo diventa quello di prevedere la modalità assunta dalla risposta e il problema viene detto di classificazione. In base al numero di modalità per la variabile risposta, i problemi di classificazione si dividono ulteriormente in problemi di classificazione binaria, se la risposta è una qualitativa dicotomica, e di classificazione multiclasse, se la risposta è una qualitativa politomica.

In un problema di regressione si vuole mettere in relazione una variabile risposta quantitativa con le variabili esplicative: più precisamente, si suppone che il valore atteso condizionato della variabile risposta $\mathbb{E}(Y|X)$ sia determinato da una funzione dei predittori, chiamata *funzione di regressione*. Questo equivale ad affermare che il valore assunto dalla variabile risposta sia univocamente determinato dai valori delle variabili esplicative, a meno di un errore casuale che viene assunto a media nulla e indicato con ϵ. Le due formulazioni equivalenti del problema di regressione sono dunque

$$\mathbb{E}(Y|X) = f(X) \qquad \text{e} \qquad Y = f(X) + \epsilon.$$

2.2 Il modello lineare

Le diverse tecniche di apprendimento supervisionato si caratterizzano in base alla generica forma che viene assunta per la funzione di regressione $f(X)$ e ai metodi usati per stimare tale funzione sulla base dei dati osservati. La formulazione più semplice per la funzione di regressione è quella di una combinazione lineare delle variabili esplicative, ossia

$$\mathbb{E}(Y|X) = f(X) = \beta_0 + \sum_{j=1}^{p} \beta_j X_j. \tag{2.1}$$

Questo è detto il *modello lineare*, un modello statistico la cui introduzione risale ai primi anni del diciannovesimo secolo, ma che rimane tuttora uno degli strumenti principali nell'ambito dell'analisi dati, sia per la semplicità della sua formulazione, che favorisce approcci interpretativi, che per la possibilità di utilizzarlo come punto di partenza per tecniche più complesse (e.g. modelli additivi, modelli per dati funzionali ...).

Il termine β_0 è detto intercetta e fornisce la possibilità di avere un valore di riferimento non nullo per la variabile risposta nel caso in cui tutte le variabili esplicative assumano il valore 0. In generale, aumenta la flessibilità del modello e per questo viene solitamente incluso nella formulazione. Includendo il valore costante 1 nel vettore X, il modello descritto in equazione (2.1) può essere riscritto in formato matriciale

$$Y = X\beta + \epsilon, \tag{2.2}$$

dove $X = [1, X_1, \ldots, X_p]$ e $\beta = [\beta_0, \beta_1, \ldots, \beta_p]$.

Sulla base del campione osservato, ossia (y_i, x_i) per $i = 1, \ldots, n$, si vogliono stimare i coefficienti β del modello. I coefficienti stimati, detti $\hat{\beta}$, possono essere utilizzati per interpretare gli effetti delle singole covariate o per prevedere il valore della risposta sulla base di un nuovo vettore di covariate. Ci sono molti metodi possibili per la stima, il più utilizzato è certamente il criterio dei minimi quadrati, dove si vogliono trovare coefficienti β tali da minimizzare la somma dei quadrati dei residui (in inglese *residual sum of squares*, RSS)

$$\begin{aligned} RSS(\beta) &= \sum_{i=1}^{n} \left(y_i - f(x_i)\right)^2 \\ &= \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j\right)^2. \end{aligned} \tag{2.3}$$

Questo tipo di criterio ha giustificazioni sia di carattere statistico che geometrico.

Intuitivamente si ha che, senza assumere valido il modello specificato in (2.2), viene cercato l'iperpiano in X per il quale risulta minima la distanza con i punti osservati Y; è un criterio che generalmente misura la qualità dell'adattamento del modello ai dati osservati.

Per minimizzare il criterio (2.3) e ottenere le stime $\hat{\beta}$ per i coefficienti, conviene lavorare con i dati in formato matriciale. Sia $\mathbf{y} = [y_1 \cdots y_n]$ il vettore $n \times 1$ contenente i valori della variabile risposta nell'insieme di stima e sia $\mathbf{X}$ la matrice $n \times (p+1)$ dove ogni riga è un vettore di variabili esplicative (con 1 in prima posizione). Il criterio ai minimi quadrati, riscritto in forma matriciale, risulta essere

$$\begin{aligned} RSS(\beta) &= ||\mathbf{y} - \mathbf{X}\beta||_2^2 \\ &= (\mathbf{y} - \mathbf{X}\beta)^T(\mathbf{y} - \mathbf{X}\beta) \end{aligned} \tag{2.4}$$

Differenziando questa funzione quadratica rispetto a β, si ottiene

$$\begin{aligned} \frac{\partial RSS}{\partial \beta} &= -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta) \\ \frac{\partial^2 RSS}{\partial \beta \partial \beta^T} &= 2\mathbf{X}^T\mathbf{X} \end{aligned}$$

Se la matrice $\mathbf{X}$ è a rango pieno, cioè non ci sono colonne ottenibili esattamente come combinazioni lineari di altre, allora la forma quadratica $\mathbf{X}^T\mathbf{X}$ è sempre definita positiva e il punto critico, se unico, sarà il punto di minimo assoluto della funzione. Si può quindi porre a zero la derivata prima

$$\mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta) = 0 \tag{2.5}$$

e ottenere le *equazioni normali*, punto di partenza per la stima dei parametri β

$$\mathbf{X}^T\mathbf{X}\beta = \mathbf{X}^T\mathbf{y}. \tag{2.6}$$

Partendo dall'equazione (2.6) appena presentata, si ottiene la soluzione esplicita

$$\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}. \tag{2.7}$$

I valori predetti per un nuovo vettore di input x^* (per comodità si assuma che il vettore contenga già il valore 1 in prima posizione), saranno $\hat{f}(x^*) = x^{*T}\hat{\beta}$, mentre le

previsioni per i valori osservati nell'insieme di stima sono

$$\hat{\mathbf{y}} = \mathbf{X}\hat{\beta} = \mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}.$$

La matrice $\mathbf{H} = \mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$ viene chiamata *hat matrix*, perché mette il "cappello" al vettore $\mathbf{y}$. A questo punto è possibile considerare una diversa interpretazione geometrica della stima ai minimi quadrati. Siano $\mathbf{x_0}, \mathbf{x_1}, \ldots, \mathbf{x_p}$ i vettori colonna che compongono la matrice $\mathbf{X}$, con $\mathbf{x_0} \equiv 1$. Questi vettori definiscono un sottospazio di $\mathbb{R}^n$, che viene detto spazio delle colonne di $\mathbf{X}$. Per minimizzare il criterio ai minimi quadrati (2.4), si cerca un vettore $\hat{\beta}$ tale che il vettore dei residui $\mathbf{y} - \hat{\mathbf{y}}$ sia ortogonale alla matrice $\mathbf{X}$. Questa condizione di ortogonalità emerge dall'equazione 2.5, e ne deriva che $\hat{\mathbf{y}}$ è la proiezione ortogonale di $\mathbf{y}$ nello spazio delle colonne di $\mathbf{X}$. Questa proiezione viene calcolata a partire da $\mathbf{y}$ grazie alla matrice $\mathbf{H}$, che viene per questo chiamata anche matrice di proiezione.

Si noti come i risultati presentati fino a questo punto siano stati ottenuti basandosi sull'unica assunzione che la funzione di regressione fosse lineare nei parametri. Più precisamente, l'unica assunzione utilizzata è che la funzione di regressione possa essere approssimata in maniera soddisfacente da una funzione lineare: non è stato quindi necessario fare ipotesi distributive su X e Y, né assumere di aver specificato correttamente il modello per il valore atteso condizionato.

Nella sezione 2.6 si mostrerà come utilizzando un numero maggiore di assunzioni di carattere distributivo si possano derivare ulteriori risultati e proprietà. Tali assunzioni sono necessarie per affrontare problemi di inferenza, cioè quel processo di estrarre informazioni valide a livello di popolazione a partire dai risultati ottenuti da un campione finito.

Nella sezione successiva verranno presentati alcuni algoritmi per la stima dei coefficienti del modello lineare. Ottenere la soluzione esplicita usando la formula (2.7) può risultare problematico, in quanto l'inversione della matrice $\mathbf{X}^T\mathbf{X}$ è un'operazione che può diventare computazionalmente molto onerosa, specialmente per valori elevati di n e p; per questo motivo sono state sviluppate nel tempo diverse alternative.

La letteratura statistica presenta numerosi testi che introducono la teoria dei base dei modelli lineari; per il materiale contenuto in questa sezione e nella precedente, si può fare riferimento a Hastie et al. (2009) e Azzalini & Scarpa (2012), che trattano anche altri temi di apprendimento supervisionato.

2.3 Algoritmi di stima

In questa sezione verranno esposti alcuni algoritmi disponibili per la stima dei coefficienti β di un modello lineare. Verranno anche valutati i pro e i contro di ciascun algoritmo, insieme al costo computazionale associato. Per le nozioni di calcolo matriciale e riguardanti i requisiti computazionali e di memoria degli algoritmi citati delle sottosezioni 2.3.1 e 2.3.2 a seguire, si faccia riferimento a Golub & Van Loan (2013), fonte autorevole nell'ambito.

2.3.1 Minimi quadrati tramite fattorizzazione di Cholesky

Uno dei metodi utilizzati per la risoluzione delle equazioni normali si appoggia sulla fattorizzazione di Cholesky. Questa, quando applicabile, risulta più efficiente della fattorizzazione LU, che viene solitamente utilizzata dalle librerie di calcolo algebrico per risolvere sistemi lineari generici (e quindi anche per trovare l'inversa di una qualsiasi matrice).

Sia $\mathbf{A}$ una matrice quadrata $p \times p$ simmetrica e definita positiva; allora esiste un'unica matrice $\mathbf{R}$ triangolare superiore con valori sulla diagonale positivi tale che

$$\mathbf{A} = \mathbf{R}^T\mathbf{R}. \tag{2.8}$$

La matrice $\mathbf{R}$ viene chiamata *fattore di Cholesky*.

Nel calcolo numerico, le matrici triangolari sono di particolare interesse perché permettono la risoluzione di un sistema lineare del tipo $\mathbf{L}x = b$ in p^2 operazioni elementari (dette *flops*), dove p è la dimensione del lato della matrice triangolare. Si assuma che $\mathbf{L}$ sia triangolare inferiore: per risolvere il sistema $\mathbf{L}x = b$, basta risolvere la prima equazione, sostituirla nella seconda equazione, e così via fino ad arrivare alla soluzione finale. Questo procedimento viene chiamato *forward substitution*, mentre quello analogo utilizzato per le matrici triangolari superiori è la *backward substitution*.

Nell'ambito delle equazioni normali, si ha che $\mathbf{X}^T\mathbf{X}$ è simmetrica e definita positiva, quindi valgono le seguenti uguaglianze:

$$\begin{aligned} \mathbf{X}^T\mathbf{X}\beta &= \mathbf{X}^T\mathbf{y} \\ \mathbf{R}^T\mathbf{R}\beta &= \mathbf{X}^T\mathbf{y} \\ \mathbf{R}^T\hat{\gamma} &= d, \end{aligned} \tag{2.9}$$

dove $\mathbf{R}$ è il fattore di Cholesky associato a $\mathbf{X}^T\mathbf{X}$, $\hat{\gamma} = \mathbf{R}\beta$ e $d = \mathbf{X}^T\mathbf{y}$. Con queste quantità, si può usare la *forward substitution* per risolvere $\mathbf{R}^T\hat{\gamma} = d$ e successivamente

la *backward substitution* per risolvere $\mathbf{R}\beta = \hat{\gamma}$ e trovare la stima finale $\hat{\beta}$. I passaggi presentati definiscono l'algoritmo 1, detto LS-CHOL, che viene presentato di seguito in pseudo-codice.

Algoritmo 1: LS-CHOL

Dati: $\mathbf{X}, \mathbf{y}$
Risultato: $\hat{\beta}$
1 Si ottenga $d = \mathbf{X}^T\mathbf{y}$;
2 Si ottenga il fattore di Cholesky $\mathbf{R}$ tale che $\mathbf{X}^T\mathbf{X} = \mathbf{R}^T\mathbf{R}$;
3 Si utilizzi la *forward substitution* per risolvere $\mathbf{R}^T\hat{\gamma} = d$;
4 Si utilizzi la *backward substitution* per risolvere $\mathbf{R}\beta = \hat{\gamma}$;

Per eseguire l'algoritmo LS-CHOL sono necessarie $(n + p/3)p^2$ operazioni elementari, per cui la complessità totale è $O(np^2 + p^3)$. Ci sono due principali vantaggi legati a questo algoritmo. Il primo è che è basato su algoritmi standard e affidabili, come la scomposizione di Cholesky ($p^3/3$ *flops*) e il prodotto matriciale. Il secondo vantaggio riguarda la memoria fissa utilizzata: l'algoritmo permette di comprimere la matrice $\mathbf{X}$ di dimensioni $n \times p$ nella matrice $p \times p$ triangolare superiore $\mathbf{R}$, il che comporta un notevole vantaggio in termini di memoria necessaria quando per esempio $n \gg p$. Inoltre, una matrice triangolare può essere immagazzinata in maniera efficiente mantenendo solo i $p(p+1)/2$ elementi non nulli.

Lo svantaggio principale dell'algoritmo presentato è legato a possibili instabilità numeriche nella soluzione. In un computer, sia u la metà della differenza tra 1 e il più piccolo successivo numero decimale (*floating point*). Questo valore viene definito *unit roundoff* ed è tale che, per la macchina in questione, $1 + u = 1$. In Golub & Van Loan (2013) si riporta il seguente risultato:

$$||\hat{\beta}_{LS-CHOL} - \hat{\beta}||_2 / ||\hat{\beta}||_2 \approx uk(\mathbf{X}^T\mathbf{X}) = uk(\mathbf{X})^2, \tag{2.10}$$

dove $\hat{\beta}_{LS-CHOL}$ è la soluzione delle equazioni normali ottenuta tramite l'algoritmo 1, $\hat{\beta}$ è la soluzione esplicita ottenuta tramite inversione di $\mathbf{X}^T\mathbf{X}$ e $k(\mathbf{X})$ è l'indice di condizionamento di $\mathbf{X}$ (il rapporto tra il più grande e il più piccolo autovalore di $\mathbf{X}$). Questo risultato mostra dunque come ci si debba aspettare una bassa precisione nella risoluzione delle equazioni normali per $k(\mathbf{X}) \approx 1/\sqrt{u}$. Si mostrerà come altri algoritmi risultino precisi sotto condizioni meno stringenti relative all'indice di condizionamento di $\mathbf{X}$. Un altro problema associato a questo algoritmo è che, per motivi numerici legati all'*unit roundoff* u, la costruzione della forma quadratica $\mathbf{X}^T\mathbf{X}$ può potenzialmente portare ad una perdita parziale di informazione.

2.3.2 Minimi quadrati tramite scomposizione QR

Un altro approccio frequentemente utilizzato per la risoluzione delle equazioni normali è basato sulla scomposizione QR della matrice $\mathbf{X}$.

Se $\mathbf{X}$ è una matrice reale $n \times p$, allora

$$\mathbf{X} = \mathbf{Q} \begin{bmatrix} \mathbf{R} \\ \mathbf{0}_{(n-p)\times p} \end{bmatrix} = \begin{bmatrix} \mathbf{Q}_1 & \mathbf{Q}_2 \end{bmatrix} \begin{bmatrix} \mathbf{R} \\ \mathbf{0}_{(n-p)\times p} \end{bmatrix},$$

dove $\mathbf{Q}$ è una matrice ortogonale $n \times n$ (i.e. tale che $\mathbf{Q}^T = \mathbf{Q}^{-1}$), $\mathbf{R}$ è una matrice triangolare superiore $p \times p$, $\mathbf{Q}_1$ e $\mathbf{Q}_2$ hanno dimensioni rispettivamente $n \times p$ e $n \times (n-p)$ e $\mathbf{0}_{(n-p)\times p}$ è una matrice $(n-p) \times p$ di zeri. Le matrici $\mathbf{Q}_1$ e $\mathbf{R}$ sono uniche, inoltre $\mathbf{R}$ è il fattore di Cholesky della forma quadratica $\mathbf{X}^T\mathbf{X}$.

Partendo dalla formula dei minimi quadrati, si ricavano le seguenti identità:

$$\begin{aligned} RSS(\beta) &= ||\mathbf{y} - \mathbf{X}\beta||_2^2 \\ &= ||\mathbf{Q}^T\mathbf{y} - \mathbf{Q}^T\mathbf{X}\beta||_2^2 \\ &= ||\mathbf{Q}_1^T\mathbf{y} - \mathbf{R}\beta||_2^2 + ||\mathbf{Q}_2^T\mathbf{y}||_2^2. \end{aligned} \tag{2.11}$$

Minimizzando la funzione in β si ottiene

$$\begin{aligned} 0 &= \frac{\partial}{\partial\beta} \left\{ ||\mathbf{Q}_1^T\mathbf{y} - \mathbf{R}\beta||_2^2 + ||\mathbf{Q}_2^T\mathbf{y}||_2^2 \right\} \\ 0 &= -2\mathbf{R}^T(\mathbf{Q}_1^T\mathbf{y} - \mathbf{R}\beta) \\ \mathbf{R}\beta &= \mathbf{y}^*, \end{aligned} \tag{2.12}$$

dove $\mathbf{y}^* = \mathbf{Q}_1^T\mathbf{y}$. Il sistema risultante si può risolvere facilmente usando la *backward substitution*. Si noti come il minimo del primo addendo in (2.11) sia pari a zero: ne consegue che $||\mathbf{Q}_2^T\mathbf{y}||_2^2$ è uguale alla somma dei quadrati dei residui.

La procedura viene riassunta in un algoritmo, chiamato LS-QR, che viene riportato di seguito in pseudo-codice.

Algoritmo 2: LS-QR

Dati: $\mathbf{X}, \mathbf{y}$
Risultato: $\hat{\beta}$

1 Si ottengano $\mathbf{Q}_1$, $\mathbf{Q}_2$ e $\mathbf{R}$ tali che $\mathbf{X} = \begin{bmatrix} \mathbf{Q}_1 & \mathbf{Q}_2 \end{bmatrix} \begin{bmatrix} \mathbf{R} \\ \mathbf{0}_{(n-p)\times p} \end{bmatrix}$;
2 Si ottenga $\mathbf{y}^* = \mathbf{Q}_1^T\mathbf{y}$;
3 Si utilizzi la *backward substitution* per risolvere $\mathbf{R}\beta = \mathbf{y}^*$;

L'esecuzione dell'algoritmo (2) richiede $2(n - p/3)p^2$ operazioni elementari, quindi si ha che, pur avendo lo stesso ordine di complessità $O(np^2 + p^3)$ dell'algoritmo LS-CHOL, nelle situazioni dove $n \gg p$ risulterà più lento dell'alternativa presentata in precedenza. Inoltre, nel caso fosse richiesto di salvare in memoria tutta la matrice $\mathbf{Q}$, anche l'utilizzo della memoria fissa diventerebbe un problema, poiché $\mathbf{Q}$ è di dimensioni $n \times n$. Il principale vantaggio di questo algoritmo, rispetto all'approccio tramite fattorizzazione di Cholesky, è che ha meno problemi di stabilità numerica: ci si può aspettare una prestazione non soddisfacente solo nel caso in cui $k(\mathbf{X}) \approx 1/u$ (si ricordi che l'algoritmo LS-CHOL può portare a soluzioni instabili per $k(\mathbf{X}) \approx 1/\sqrt{u}$).

Un'altra importante caratteristica di questo approccio è la possibilità di ottenere altre quantità di interesse legate al modello lineare mediante manipolazione delle matrici $\mathbf{Q}$ e $\mathbf{R}$. Oltre al risultato riportato in precedenza sulla somma dei quadrati dei residui, si ha che la varianza dello stimatore dei minimi quadrati è pari a $\sigma^2\mathbf{R}^{-1}(\mathbf{R}^T)^{-1}$, mentre la matrice di proiezione e i residui si ottengono rispettivamente come $\mathbf{Q}_1\mathbf{Q}_1^T$ e $\mathbf{Q}_2\mathbf{Q}_2^T\mathbf{y}$.

Per questi motivi l'approccio tramite scomposizione QR risulta più utilizzato rispetto a quello basato sulla fattorizzazione di Cholesky; ad esempio, la funzione `lm` del software statistico R utilizza attualmente un'implementazione di questo algoritmo per calcolare la stima di un modello lineare.

2.3.3 Metodi numerici

Per estendere il numero di algoritmi e approcci possibili per la risoluzione del problema dei minimi quadrati, si possono adattare un numero di metodi di ottimizzazione numerica. Questi metodi sono molto generici e si basano su approcci iterativi: per minimizzare una funzione generica $f(x)$, si parte da una soluzione iniziale x_0 che viene aggiornata iterativamente fino alla convergenza ad un punto di minimo. Le soluzioni numeriche sono per definizione approssimate, al contrario delle soluzioni esplicite (presentate nelle sezioni precedenti), che, quando utilizzabili sulla base delle caratteristiche del problema in questione e della macchina utilizzata per il calcolo, sono da preferire, ma per la loro generalità e formulazione sono di particolare interesse in situazioni con grosse moli di dati.

Si considerano qui i metodi di ricerca lineare. In questo tipo di algoritmi ad ogni iterazione si identifica una direzione p_k lungo la quale "muovere" la soluzione e si decide di quanto muoversi in quella direzione: l'aggiornamento ha la forma $x_{k+1} = x_k + a_k p_k$, dove lo scalare a_k è detto *step length*. Uno dei primi metodi introdotti, che tra l'altro è stato adattato e utilizzato spesso in ambito statistico (si pensi ai modelli lineari generalizzati), è il metodo di Newton.

Siano $f(x)$ una funzione di una variabile vettoriale, $g_x = g(x)$ e $H_x = H(x)$ rispettivamente il suo gradiente e la matrice hessiana calcolati nel punto x. Il problema di ottimizzazione è $\min_{x \in \mathbb{R}^p} f(x)$. Il metodo di Newton punta a trovare una sequenza di soluzioni $\{x_k\}$, a partire da una soluzione iniziale x_0, tale che la sequenza converga al minimo x_{min} richiesto, mediante una serie di approssimazioni di Taylor al secondo ordine di f intorno alle iterazioni. L'espansione al secondo ordine di f intorno a x_k è

$$f(x_k + p_k) \approx f(x_k) + g(x_k)^T p_k + \frac{1}{2} p_k^T H(x_k) p_k \tag{2.13}$$

Il passo di aggiornamento p_k viene definito in modo da minimizzare tale approssimazione quadratica, e ottenere poi $x_{k+1} = x_k + p_k$. Se la matrice hessiana è positiva definita, l'approssimazione quadratica risulta convessa in p_k e si può trovare il minimo globale uguagliando il gradiente a 0. Si ha quindi

$$0 = \frac{\partial}{\partial p_k} \left(f(x_k) + g(x_k)^T p_k + \frac{1}{2} p_k^T H(x_k) p_k \right) = g(x_k) + H(x_k) p_k, \tag{2.14}$$

e il minimo si ottiene per

$$p_k = -H(x_k)^{-1} g(x_k). \tag{2.15}$$

Il generico passo di aggiornamento secondo il metodo di Newton è quindi $x_{k+1} = x_k + p_k = x_k - H(x_k)^{-1} g(x_k)$.

La formulazione presentata per il metodo di Newton, se vista come caso particolare dei metodi di ricerca lineare introdotti in precedenza, prevede che la lunghezza del passo di aggiornamento a_k sia uguale a uno per tutte le iterazioni. In realtà, l'approccio moderno all'implementazione dei metodi di ricerca lineare prevede che tale valore sia scelto secondo qualche criterio di ottimalità. La scelta prevede un *tradeoff* tra l'identificazione di un valore tale da portare la maggior riduzione possibile di f e il non voler spendere troppe risorse per calcolarlo. Solitamente negli algoritmi di ricerca lineare viene provata una sequenza di valori candidati per a_k, accettando il primo valore per il quale certe condizioni (che garantiscono una sufficiente diminuzione della funzione obiettivo) vengono soddisfatte.

Il metodo di Newton è molto utilizzato nell'ottimizzazione numerica, e solitamente arriva a convergenza in poche iterazioni una volta che ci si trova nelle vicinanze della vera soluzione. Il problema principale di tale algoritmo è la necessità di dover calcolare ad ogni iterazione la matrice hessiana nel punto di interesse. Nel tempo, sono state sviluppate varianti di tale metodo che lavorano analogamente (i.e. minimizzando iterativamente approssimazioni quadratiche della funzione obiettivo) ma utilizzano

un'approssimazione B_k della matrice hessiana $H(x_k)$, che viene aggiornata ad ogni passo per tenere conto dell'informazione aggiuntiva sulla curvatura della funzione che viene accumulata durante l'ottimizzazione. Tutte le varianti, che differiscono nel modo di determinare l'aggiornamento di B_k, fanno parte dei metodi quasi-Newton.

Il metodo quasi-Newton più popolare è l'algoritmo BFGS, dal nome degli studiosi che l'hanno introdotto: Broyden, Fletcher, Goldfarb e Shanno. Si consideri ancora un'approssimazione quadratica per la funzione obiettivo:

$$f(x_k + p_k) \approx f(x_k) + g(x_k)^T p_k + \frac{1}{2} p_k^T B_k p_k, \tag{2.16}$$

dove la matrice B_k, simmetrica, positiva definita e di dimensioni $n \times n$, verrà aggiornata ad ogni iterazione. Come in precedenza, la direzione dell'aggiornamento p_k verrà data da $p_k = -B_k^{-1} g(x_k)$ e la lunghezza del passo di aggiornamento a_k viene scelta in modo da soddisfare alcune condizioni necessarie ad assicurare la convergenza dell'algoritmo. La singola iterazione è molto simile a quella del metodo di Newton, mentre la differenza sostanziale è l'utilizzo dell'approssimazione B_k invece della vera matrice hessiana. Una condizione logicamente sensata da imporre per la scelta dell'aggiornamento B_{k+1}, è che il gradiente dell'approssimazione quadratica aggiornata m_{k+1} dovrebbe corrispondere a quello della funzione obiettivo f, almeno per le due ultime iterazioni x_k e x_{k+1}. Tale condizione viene elicitata matematicamente e manipolata, ottenendo le seguenti identità:

$$\begin{aligned} g(x_{k+1}) - a_k B_{k+1} p_k &= g(x_k) \\ B_{k+1} a_k p_k &= g(x_{k+1}) - g(x_k) \\ B_{k+1} s_k &= y_k, \end{aligned} \tag{2.17}$$

dove $s_k = x_{k+1} - x_k = a_k p_k$ e $y_k = g(x_{k+1}) - g(x_k)$. L'ultima uguaglianza prende il nome di *equazione secante*, e insieme ad altre condizioni permette di identificare la soluzione B_{k+1}. Per determinare la soluzione, si impone il vincolo di trovare quella matrice che, tra tutte le matrici simmetriche che soddisfano l'equazione secante, sia la più vicina (secondo qualche criterio) alla matrice corrente B_k. Il problema viene formalizzato in questo modo:

$$\min_B ||B - B_k|| \qquad \text{s.t. } B = B^T, \quad B s_k = y_k. \tag{2.18}$$

Utilizzando la norma pesata di Frobenius (con matrice di pesi scelta come una qualsiasi matrice che soddisfi l'equazione secante), si ottiene una soluzione esplicita per l'aggiornamento di B:

$$B_{k+1} = (I - p_k y_k s_k^T) B_k (I - p_k s_k y_k^T) + p_k y_k y_k^T, \tag{2.19}$$

con $p_k = 1/y_k^T s_k$. Questa formula è chiamata DFP, in quanto è stata introdotta da Davison nel 1959 e succesivamente studiata e implementata da Fletcher e Powell. Partendo da questo aggiornamento, grazie alla formula di Sherman-Morrison si può determinare l'aggiornamento di $H_k = B_k^{-1}$. La formula di Sherman-Morrison permette di trovare l'aggiornamento dell'inversa di una matrice A a cui venga sommata una matrice di rango unitario ottenuta dalla moltiplicazione di due vettori colonna:

$$(A + uv^T)^{-1} = A^{-1} - \frac{A^{-1}uv^T A^{-1}}{1 + v^T A^{-1} u}. \tag{2.20}$$

Storicamente, la formula DFP è stata velocemente soppiantata dalla formula BFGS, che parte da un approccio molto simile ma lavora direttamente sull'inversa H_k. Si consideri l'equazione secante, che riscritta in termini di H_{k+1} diventa $H_{k+1} y_k = s_k$. Impostando un problema di ottimizzazione analogo a quello presentato in (2.18), si ottiene:

$$\min_H ||H - H_k|| \qquad \text{s.t. } H = H^T, \quad H y_k = s_k. \tag{2.21}$$

Come in precedenza, utilizzando la norma di Frobenius pesata, con matrice di pesi W tale da soddisfare $W s_k = y_k$, si arriva ad una soluzione esplicita per l'aggiornamento della matrice di interesse:

$$H_{k+1} = (I - p_k s_k y_k^T) H_k (I - p_k y_k s_k^T) + p_k s_k s_k^T, \tag{2.22}$$

con p_k definito come prima. Tale formula viene detta BFGS, e questo metodo viene attualmente considerato come il migliore fra i metodi di aggiornamento di tipo quasi-Newton.

Nella letteratura sono presenti numerose alternative ai metodi descritti, qua si è deciso di focalizzarsi sui metodi di ricerca lineare, e nello specifico i metodi quasi-Newton e fra questi l'algoritmo BFGS. Il motivo è che questo algoritmo, insieme a una sua estensione che verrà descritta di seguito, viene spesso utilizzato nell'ambito della stima dei modelli lineari in contesti di Big Data.

Il metodo BFGS, nonostante la maggior efficienza rispetto al metodo di Newton derivante dall'utilizzo di un'approssimazione dell'inversa della matrice hessiana, richiede comunque la scrittura in memoria di una matrice densa di dimensioni $p \times p$. Questo, anche in situazioni con p non troppo elevato, può rivelarsi un problema; nel corso del tempo sono state quindi sviluppate alternative a "memoria limitata" dei metodi quasi-Newton. L'idea di base dietro a questi approcci è quella di salvare solo alcuni vettori di

lunghezza p, che permettono di rappresentare implicitamente l'approssimazione dell'inversa della matrice hessiana, evitando quindi di dover salvare in memoria una matrice densa $p \times p$. Fra questi approcci il più famoso è il metodo L-BFGS, che, come suggerisce il nome, è basato sulla formula BFGS. L'idea principale del metodo è quella di utilizzare solo l'informazione relativa alla curvatura della funzione obiettivo contenuta nelle iterazioni più recenti per costruire l'approssimazione della matrice H_k. L'informazione proveniente dalle iterazioni più distanti da quella corrente, che verosimilmente non sarà di particolare rilievo per determinare il comportamento dell'hessiana all'iterazione corrente, viene tralasciata per occupare meno memoria. Concretamente, si salvano soltanto un certo numero m di coppie di vettori $\{s_i, y_i\}$ utilizzati nella formula (2.22). Il prodotto $H_k g(x_k)$ viene ottenuto mediante una sequenza di prodotti interni e somme vettoriali che coinvolgono il gradiente $g(x_k)$ e le coppie di vettori $\{s_i, y_i\}$. Quando un nuovo valore viene calcolato, la coppia di vettori relativa all'iterazione più lontana da quella corrente viene sostituita da quella ottenuta al passo corrente; in questo modo le coppie $\{s_i, y_i\}$ contengono solo l'informazione relativa alle m iterazioni più recenti. Solitamente si utilizza un valore modesto per m (tra 3 e 20), raggiungendo risultati soddisfacenti.

L'algoritmo L-BFGS (e varianti che consentono l'utilizzo di penalizzazioni per la funzione di perdita), sono implementati e utilizzati per la stima di modelli lineari e modelli più complessi in diversi *framework* per l'analisi di Big Data, ad esempio in Apache Spark (Zaharia et al. (2016)).

Per approfondimenti e precisazioni sugli algoritmi numerici presentati in questa sottosezione, si rimanda a Nocedal & Wright (2006), fonte autorevole nell'ambito dell'ottimizzazione numerica.

2.4 Approcci per Big Data

Gli algoritmi presentati nella sezione 2.3, specialmente LS-CHOL e LS-QR, sono adatti nelle situazioni in cui si possono caricare tutti i dati in memoria senza particolari problemi. In questo contesto per memoria si intende la RAM (*Random Access Memory*) della macchina utilizzata per il calcolo. Si ricordi che la RAM è la memoria della macchina in cui vengono immagazzinate le informazioni di cui un programma ha bisogno durante l'esecuzione. Entrambi i metodi considerati hanno una complessità computazionale pari a $O(np^2 + p^3)$, mentre la memoria richiesta è $O(np + p^2)$. Tale requisito viene difficilmente soddisfatto dalle macchine di uso comune nel caso in cui ci si trovi in una contesto di Big Data con $n \gg p$. Come si può evincere dai requisiti computazionali

e di memoria riportati, anche la situazione con $p \gg n$ porta a problemi di carattere computazionale (oltre che algebrici), tuttavia in questo contesto ci si focalizzerà nella situazione più standard di $n > p$ con n particolarmente elevato.

Ci si pone quindi nella situazione in cui i dati di input, cioè la matrice $\mathbf{X}$ e il vettore $\mathbf{y}$, siano immagazzinati in un *database* fisso e remoto, e si possono leggere una o più righe e metterle nella memoria della macchina su cui si lavora (che però non sarebbe in grado di leggere tutti i dati). Di preciso, si può leggere una riga alla volta per ottenere i valori (y_i, x_i) per $i = 1, \dots, n$ o un blocco alla volta (detto anche *chunk*) per ottenere $(\mathbf{X}_k, \mathbf{y}_k)$ per $k = 1, \dots, K$. In questo caso, si indica con $\mathbf{X}_k$ il k-esimo blocco della matrice $\mathbf{X}$ e con $\mathbf{X}_{1:k}$ la matrice formata dai primi k blocchi (si utilizza una notazione analoga per il vettore $\mathbf{y}$). Tutti i blocchi, a meno dell'ultimo, contengono un numero di righe pari a c.

L'obiettivo è quello di sviluppare metodi che permettano di aggiornare la stima una volta che si ha a disposizione una nuova riga di dati (o un nuovo blocco). Nello specifico, si vogliono aggiornare le quantità di base utilizzate per la stima. I metodi sviluppati differiscono per le quantità che puntano ad aggiornare: si può aggiornare la forma quadratica $\mathbf{X}^T\mathbf{X}$ o la sua inversa $(\mathbf{X}^T\mathbf{X})^{-1}$, oppure ancora aggiornare la scomposizione QR di $\mathbf{X}$.

2.4.1 Aggiornamento di $\mathbf{X}^T\mathbf{X}$

Per una matrice $\mathbf{X}$, si consideri la quantità $\mathbf{X}^T\mathbf{X}$. Dalle basi di algebra lineare, è noto il risultato $\mathbf{X}^T\mathbf{X} = \sum_{i=1}^{n} x_i x_i^T$, dove x_i è il vettore colonna contenente la riga i-esima della matrice in questione. Da questo risultato si evince come per costruire la matrice $\mathbf{X}^T\mathbf{X}$ non sia necessario leggere tutta la matrice $\mathbf{X}$ (di dimensioni $n \times p$, si ricordi che siamo nella situazione con n molto elevato), ma basti leggere le righe una per volta e utilizzare le n componenti $x_i x_i^T$ per aggiornare la matrice di interesse, di dimensione $p \times p$. Similmente si ha che $\mathbf{X}^T\mathbf{y} = \sum_{i=1}^{n} x_i y_i$, e si può quindi facilmente aggiornare la quantità di interesse quando viene letta una nuova riga. Volendo procedere leggendo le quantità $\mathbf{X}$ e $\mathbf{y}$ a blocchi, si parta da $\mathbf{C}_k = \mathbf{X}_{1:k}^T\mathbf{X}_{1:k}$ e $\mathbf{d}_k = \mathbf{X}_{1:k}^T\mathbf{y}_{1:k}$. L'aggiornamento delle due quantità è di facile derivazione:

$$\mathbf{C}_{k+1} = \mathbf{C}_k + \mathbf{X}_{k+1}^T\mathbf{X}_{k+1}, \qquad \mathbf{d}_{k+1} = \mathbf{d}_k + \mathbf{X}_{k+1}^T\mathbf{y}_{k+1}$$

Ci si focalizza ora sul caso in cui si vogliano aggiornare le quantità di interesse leggendo i dati una riga per volta. Il motivo per questa scelta è quello di mantenere la coerenza con l'implementazione di questo algoritmo che verrà utilizzata nell'applicazione

con dati reali esposta nel capitolo 3. Una volta costruite le quantità di interesse $\mathbf{X}^T\mathbf{y}$ e $\mathbf{X}^T\mathbf{X}$, per risolvere le equazioni normali si utilizza la fattorizzazione di Cholesky. I passaggi vengono sintetizzati nell'algoritmo 3, chiamato LS-CHOL-ROWWISE, che viene riportato di seguito.

Algoritmo 3: LS-CHOL-ROWWISE

Dati: $\mathbf{X}$ e $\mathbf{y}$, salvati su un *database remoto*
Risultato: $\hat{\beta}$

$\mathbf{C} \leftarrow \mathbf{0}_{p\times p}, \quad \mathbf{d} \leftarrow \mathbf{0}_{p\times 1}$;
per $i = 1, \ldots, n$ **fai**
  $x, y \leftarrow x_i, y_i$;
  $\mathbf{C} \leftarrow \mathbf{C} + xx^T$;
  $\mathbf{d} \leftarrow \mathbf{d} + xy$
fine
Si utilizzi ora l'algoritmo LS-CHOL, con $\mathbf{X}^T\mathbf{X} = \mathbf{C}$ e $\mathbf{X}^T\mathbf{y} = \mathbf{d}$;

L'algoritmo LS-CHOL-ROWWISE richiede uno spazio in memoria RAM nell'ordine di $O(p^2)$, e ha gli stessi vantaggi e svantaggi della versione che legge tutti i dati allo stesso tempo. Si noti come questo tipo di approccio risulti particolarmente adatto nella situazione considerata, ossia di *Big Data* con $n \gg p$ (a volte chiamati anche *long data*): permette infatti di neutralizzare il problema della numerosità, leggendo i dati uno per volta, e di lavorare solo con matrici $p \times p$ facilmente gestibili in memoria.

2.4.2 Aggiornamento di $(\mathbf{X}^T\mathbf{X})^{-1}$

Un approccio alternativo rispetto a quello esposto precedentemente prevede di aggiornare direttamente l'inversa $(\mathbf{X}^T\mathbf{X})^{-1}$. Questo permette di ottenere la soluzione finale come risultato di un semplice prodotto matriciale. Per aggiornare l'inversa di una matrice, ci si basa sull'identità di Sherman-Morrison-Woodbury, della quale è già stata considerata una variante nella sezione 2.3.

Per matrici A, U, B, V con dimensioni compatibili, vale la seguente identità:

$$(A + UBV)^{-1} = A^{-1} + A^{-1}U(B^{-1} + VA^{-1}U)^{-1}VA^{-1} \tag{2.23}$$

Sfruttando questo risultato, si può trovare facilmente la formula per aggiornare l'inversa della matrice $\mathbf{C}_k$.

$$\begin{aligned} \mathbf{C}_{k+1}^{-1} &= (\mathbf{C}_k + \mathbf{X}_{k+1}^T\mathbf{X}_{k+1})^{-1} \\ &= (\mathbf{C}_k + \mathbf{X}_{k+1}^T\mathbf{I}_c\mathbf{X}_{k+1})^{-1} \\ &= \mathbf{C}_k^{-1} + \mathbf{C}_k^{-1}\mathbf{X}_{k+1}^T(\mathbf{I}_c + \mathbf{X}_{k+1}\mathbf{C}_k^{-1}\mathbf{X}_{k+1}^T)\mathbf{X}_{k+1}\mathbf{C}_k^{-1} \end{aligned} \tag{2.24}$$

Nel caso in cui si leggano i dati una riga alla volta, si assuma di aver letto le prime m righe della matrice $\mathbf{X}$, e di aver ottenuto la matrice $\mathbf{C}_m = \mathbf{X}_{1:m}^T \mathbf{X}_{1:m}$. Utilizzando la formula 2.20, si ottiene

$$\mathbf{C}_{m+1}^{-1} = \mathbf{C}_m^{-1} - \frac{\mathbf{C}_m^{-1} x_{m+1} x_{m+1}^T \mathbf{C}_m^{-1}}{1 + x_{m+1}^T \mathbf{C}_m^{-1} x_{m+1}}. \tag{2.25}$$

Utilizzando questi approcci, una volta ottenute le quantità di interesse la soluzione al problema dei minimi quadrati si ottiene mediante un semplice prodotto matriciale. L'algoritmo 4, detto LS-SM-ROWWISE, riporta la procedura in pseudo-codice per il caso in cui si leggano i dati una riga alla volta.

Algoritmo 4: LS-SM-ROWWISE

Dati: $\mathbf{X}$ e $\mathbf{y}$, salvati su un *database remoto*
Risultato: $\hat{\beta}$

$\mathbf{C} \leftarrow \mathbf{0}_{p\times p}, \quad \mathbf{d} \leftarrow \mathbf{0}_{p\times 1}$;
per $i = 1, \ldots, m$ **fai**
  $x, y \leftarrow x_i, y_i$;
  $\mathbf{C} \leftarrow \mathbf{C} + xx^T$;
  $\mathbf{d} \leftarrow \mathbf{d} + xy$;
fine
$\mathbf{C}* \leftarrow \mathbf{C}^{-1}$;
per $i = m+1, \ldots, n$ **fai**
  $x, y \leftarrow x_i, y_i$;
  $\mathbf{C}^* = \mathbf{C}^* - \mathbf{C}^* x x^T \mathbf{C}^* / (1 + x^T \mathbf{C}^* x)$;
  $\mathbf{d} \leftarrow \mathbf{d} + xy$;
fine
$\hat{\beta} = \mathbf{C}^* \mathbf{d}$;

Questo algoritmo ha il vantaggio di ottenere la soluzione finale come semplice prodotto matriciale, e come l'algoritmo 3 ha requisiti di memoria pari a $O(p^2)$, tuttavia risulta più suscettibile ad instabilità di tipo numerico. Nella procedura riportata, si leggono le prime m righe della matrice $\mathbf{X}$ (con $m \geq p$ per assicurarsi che $\mathbf{X}^T\mathbf{X}$ sia invertibile) per ottenere la matrice $\mathbf{C}^*$, che viene poi aggiornata sequenzialmente leggendo le righe da $m+1$ a n. C'è anche la possibilità utilizzare un'approssimazione iniziale per $\mathbf{C}^*$, ad esempio con una matrice diagonale, e di cominciare la procedura sequenziale di aggiornamento leggendo le righe dalla prima; tale approssimazione risulta non rilevante nel caso in cui il numero di righe sia particolarmente elevato.

Vale la pena di riportare che una variante di questo algoritmo, che lavora con le stesse quantità, permette di definire una stima ricorsiva della soluzione ai minimi quadrati, senza il bisogno di aspettare di aver costruito le quantità di interesse finali prima di

ottenere una soluzione. Questo approccio si rivela particolarmente utile nel caso di un flusso di dati continuo (*data stream*) e per approfondimenti si rimanda a Azzalini & Scarpa (2012).

2.4.3 Aggiornamento della scomposizione QR

L'ultimo metodo di aggiornamento delle quantità necessarie per ottenere la stima ai minimi quadrati considerato è quello che prevede l'aggiornamento della scomposizione QR della matrice $\mathbf{X}$, all'aggiunta di una nuova riga o blocco.

Si pensi di aver letto un numero k di blocchi della matrice $\mathbf{X}$, e di aver ottenuto la corrispondente scomposizione QR, ossia

$$\mathbf{X}_{1:k} = \begin{bmatrix} \mathbf{Q}_{1,1:k} & \mathbf{Q}_{2,1:k} \end{bmatrix} \begin{bmatrix} \mathbf{R}_{1:k} \\ \mathbf{0}_{(ck-p)\times p} \end{bmatrix}$$

Quando viene letto un nuovo blocco di dati, si ha $\mathbf{X}_{1:(k+1)} = \begin{bmatrix} \mathbf{X}_{1:k} \\ \mathbf{X}_{k+1} \end{bmatrix}$ e vale la seguente identità:

$$\begin{bmatrix} \mathbf{X}_{1:k} \\ \mathbf{X}_{k+1} \end{bmatrix} = \overbrace{\begin{bmatrix} \mathbf{Q}_{1,1:k} & \mathbf{0}_{ck\times c} & \mathbf{Q}_{2,1:k} \\ \mathbf{0}_{c\times c} & \mathbf{I}_c & \mathbf{0}_{c\times ck-p} \end{bmatrix}}^{augmented\ \mathbf{Q}} \underbrace{\begin{bmatrix} \mathbf{R}_{1:k} \\ \mathbf{X}_{k+1} \\ \mathbf{0}_{(ck-p)\times p} \end{bmatrix}}_{augmented\ \mathbf{R}} \tag{2.26}$$

Per aggiornare la scomposizione QR, si prenda in considerazione la matrice $\mathbf{R}$ aumentata dell'equazione precedente. L'obiettivo è quello di trovare delle matrici di rotazione (cioè ortogonali) in grado di eliminare gli elementi di $\mathbf{X}_{k+1}$ all'interno della matrice $\mathbf{R}$ aumentata, in modo da renderla nuovamente triangolare superiore. Ipotizzando di avere queste matrici di rotazione, prendendo le rotazioni opposte e post-moltiplicandole alla matrice $\mathbf{Q}$ aumentata l'identità (2.26) continua a valere, la matrice $\mathbf{Q}$ rimane ortogonale (lo sono le matrici di rotazione) e la matrice $\mathbf{R}$ viene resa nuovamente triangolare, portando ad un aggiornamento della scomposizione QR.

Le matrici di rotazione che vengono utilizzate per eliminare selettivamente alcuni elementi di una data matrice sono le cosiddette matrici di Givens. Una matrice di Givens

è ottenuta come una modifica di rango due della matrice identità, con la seguente forma:

$$\mathbf{G}_{ij} = \mathbf{G}(i,j,\theta) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix}.$$

Qui, si ha che $\mathbf{G}_{[i,i]} = \mathbf{G}_{[j,j]} = c = \cos\theta$ e $\mathbf{G}_{[i,j]} = -\mathbf{G}_{[j,i]} = s = \sin\theta$ per un certo θ. Tale matrice è ortogonale, e per un vettore (o matrice) compatibile $\mathbf{u}$ si ha che $\mathbf{G}(i,j,\theta)^T\mathbf{u}$ corrisponde ad una rotazione in senso antiorario di θ radianti di $\mathbf{u}$, nel piano definito dalle coordinate (i,j). A titolo di esempio si consideri il vettore $\mathbf{u} = [1\ 1\ 3]^T$, del quale si vuole eliminare l'elemento in posizione (1,2). Si ha allora

$$\mathbf{G}(1,2,\theta)^T \begin{bmatrix} 1 \\ 1 \\ 3 \end{bmatrix} = \begin{bmatrix} c_\theta & s_\theta & 0 \\ -s_\theta & c_\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 3 \end{bmatrix} = \begin{bmatrix} c_\theta + s_\theta \\ c_\theta - s_\theta \\ 3 \end{bmatrix}$$

Poiché dalle proprietà trigonometriche vale che $s_\theta^2 + c_\theta^2 = 1$, si ha che $c_\theta - s_\theta = 0$ per $c_\theta = s_\theta = 1/\sqrt{2}$. Questo esempio mostra come sia possibile eliminare facilmente un elemento di interesse da una matrice mediante le rotazioni di Givens. Per trovare i valori di c ed s tali da raggiungere il risultato richiesto, sono disponibili diversi algoritmi. In Golub & Van Loan (2013) ne viene riportato uno estremamente parsimonioso, che evita calcoli trigonometrici e richiede 5 *flops* e il calcolo di una radice quadrata.

Partendo dall'equazione (2.26) vista in precedenza, si possono usare cp rotazioni di Givens per rimuovere i cp elementi del blocco $\mathbf{X}_{k+1}$ nella matrice $\mathbf{R}$ aumentata. Per aggiornare $\mathbf{Q}$, basta post-moltiplicare la matrice $\mathbf{Q}$ aumentata per le rotazioni inverse.

$$\begin{bmatrix} \mathbf{X}_{1:k} \\ \mathbf{X}_{k+1} \end{bmatrix} = \overbrace{\begin{bmatrix} \mathbf{Q}_{1,1:k} & \mathbf{0}_{ck\times c} & \mathbf{Q}_{2,1:k} \\ \mathbf{0}_{c\times c} & \mathbf{I}_c & \mathbf{0}_{c\times ck-p} \end{bmatrix} \mathbf{G}_1 \dots \mathbf{G}_{cp}}^{updated\ \mathbf{Q}} \underbrace{\mathbf{G}_{cp}^T \dots \mathbf{G}_1^T \begin{bmatrix} \mathbf{R}_{1:k} \\ \mathbf{X}_{k+1} \\ \mathbf{0}_{(ck-p)\times p} \end{bmatrix}}_{updated\ \mathbf{R}} \tag{2.27}$$

Sulla base delle caratteristiche delle rotazioni di Givens, a questo punto la matrice $\mathbf{R}$ aggiornata sarà triangolare superiore, mentre la matrice $\mathbf{Q}$ aggiornata sarà ancora

ortogonale. Poiché la scomposizione QR di una matrice è unica, allora le due matrici aggiornate formano una scomposizione QR valida per la matrice $\mathbf{X}_{1:(k+1)}$. Una volta aggiornate le quantità di interesse, per ottenere la soluzione $\hat{\beta}$ basta eseguire i passaggi 2 e 3 dell'algoritmo 2.

In merito al materiale contenuto in questa sezione, per approfondimenti sui metodi di aggiornamento della regressione si vedano Chambers (1971) e Miller (1992), mentre per le nozioni di calcolo matriciale (scomposizioni, algoritmi, requisiti computazionali e di memoria) si rimanda nuovamente a Golub & Van Loan (2013).

2.4.4 Considerazioni

Gli ultimi due approcci presentati, cioè l'aggiornamento dell'inversa di $\mathbf{X}^T\mathbf{X}$ e l'aggiornamento della scomposizione QR, sono procedure di tipo sequenziale. Entrambi i metodi infatti consistono nell'aggiornamento di certe quantità, che viene calcolato sulla base della nuova riga (o blocco) e sul valore della stessa quantità al passaggio precedente.

L'aggiornamento della matrice $\mathbf{X}^T\mathbf{X}$ mediante somma delle quantità $x_i x_i^T$ per $i = 1, \ldots, n$ è anch'esso una procedura sequenziale, per come è stato posto. In realtà, le componenti $x_i x_i^T$ possono essere calcolate indipendentemente dal risultato delle operazioni precedenti: questo significa che questa procedura, a differenza delle altre due, è parallelizzabile. Le operazioni che vanno a costruire le componenti $x_i x_i^T$ possono infatti essere svolte da diverse macchine contemporaneamente, e i risultati aggregati soltanto in un secondo momento.

Questa caratteristica rende l'algoritmo adatto all'implementazione in software che sfruttano le possibilità del calcolo parallelo utilizzando paradigmi di programmazione specifici, ad esempio MapReduce, come si potrà vedere nelle applicazioni del capitolo 3.

2.5 Il modello additivo

In questa sezione ci si focalizza su una delle estensioni del modello lineare: il modello additivo, introdotto in Hastie & Tibshirani (1986). Il motivo della scelta è che la stima di questo tipo di modelli, più flessibili dei modelli lineari, si può ricondurre, in alcuni casi particolari, alla soluzione di un classico problema ai minimi quadrati. Questo implica la possibilità di estendere tutti gli approcci alla stima considerati finora ad un'altra intera classe di modelli.

Nelle applicazioni pratiche, i modelli lineari spesso non ottengono risultati soddisfacenti. I motivi possono essere molteplici, ma se ne può ipotizzare uno di maggior rilievo:

molto spesso gli effetti delle covariate sulla risposta non sono lineari. Il modello additivo viene introdotto per modellare effetti non lineari, e assume la seguente formulazione per la funzione di regressione:

$$\mathbb{E}(Y|X) = \alpha + \sum_{j=1}^{p} f_j(X_j), \tag{2.28}$$

dove le f_j sono funzioni continue. La differenza principale rispetto al modello lineare risiede proprio in queste funzioni dei predittori. Se si ipotizza di avere una funzione lineare, si ha $f_j(X_j) = \beta_j X_j$. Questo permette di vedere il modello lineare come un caso particolare del modello additivo e evidenzia la flessibilità che quest'ultimo introduce: le funzioni lisce stimate possono essere funzioni di qualsiasi tipo, ad esempio non monotòne o non lineari.

Ci sono diverse procedure per la stima di questo tipo di modelli, che si caratterizzano in base alla forma che viene assunta per le funzioni dei predittori. Nel caso particolare in cui si utilizzino delle espansioni in funzioni di base come funzioni lisce, ci si può ricondurre alla stima di un semplice modello lineare. Un'espansione in funzioni di base per una generica funzione f punta ad approssimare quest'ultima mediante una combinazione lineare di funzioni più semplici. Si ha quindi una forma del tipo $f(X) = \sum_{m=1}^{M} \beta_m h_m(X)$, dove h_m sono le funzioni elementari che costituiscono l'espansione in funzioni di base della funzione di partenza.

Come funzioni elementari si possono prendere trasformazioni non lineari (logaritmi, radici quadrate) o polinomiali, ma spesso l'obiettivo è quello di ottenere rappresentazioni flessibili per $f(X)$ e vengono quindi utilizzate le cosiddette *splines*, cioè polinomi a tratti con alcuni vincoli di continuità. Una rappresentazione polinomiale a tratti per $f(X)$ viene ottenuta suddividendo il dominio di X in intervalli contigui e rappresentando f come un polinomio entro ogni intervallo. Aggiungendo vincoli su continuità e derivate ai nodi (i punti in cui viene diviso lo spazio di X), si ottiene una rappresentazione continua e liscia per f. Le *splines* cubiche sono una particolare famiglia di funzioni polinomiali a tratti, che utilizza polinomi di terzo grado con vincoli di continuità fino alla seconda derivata ad ogni nodo. Per questo tipo di *splines* esiste una facile rappresentazione compatta:

$$\begin{aligned} h_j(X) &= X^{j-1}, \quad j = 1, \ldots, 4 \\ h_{4+l}(X) &= (X - k_l)_+^3, \quad l = 1, \ldots, K, \end{aligned} \tag{2.29}$$

dove $k_1, \ldots, k_K$ sono i K nodi definiti e l'operatore $(\cdot)_+$ restituisce la parte positiva del suo operando. Per approfondimenti sul tema delle *splines* e delle espansioni in funzioni di base, si rimanda a Wang (2011).

Utilizzando *splines* cubiche come funzioni lisce, il modello additivo diventa

$$\begin{aligned}\mathbb{E}(Y|X) &= \alpha + \sum_{j=1}^{p} f_j(X_j) \\ &= \alpha + \sum_{j=1}^{p} \left(\sum_{m=1}^{M_j} \beta_{jm} h_{jm}(X_j) \right),\end{aligned} \tag{2.30}$$

dove le funzioni h_{jm} sono definite per ogni variabile come nella formula (2.29). Questa formulazione evidenzia come si possa ricondurre il modello additivo ad un modello lineare in uno spazio aumentato: si ha quindi che, una volta costruita la nuova matrice del modello, la stima dei coefficienti si può ottenere risolvendo il problema ai minimi quadrati mediante uno degli algoritmi presentati nelle sezioni precedenti.

Lavorando con le *splines* cubiche, la rugosità delle funzioni stimate viene controllata dal numero di funzioni di base (cioè il numero di nodi più tre) che si utilizza per la rappresentazione: con molte funzioni di base si ottengono funzioni frastagliate, mentre usandone poche le funzioni risultanti saranno più lisce. In questo contesto, la rugosità di una funzione stimata viene regolata in maniera discreta, aggiungendo o togliendo basi. Un approccio alternativo è quello di utilizzare un numero di funzioni base fisso e controllare la forma della funzione stimata aggiungendo una penalizzazione sulla rugosità nella funzione obiettivo da minimizzare. Considerando il modello con una sola variabile esplicativa, per la quale è già stata ottenuta un'espansione tramite *splines* cubiche ($f(\mathbf{x}) = \mathbf{X}\beta$, dove $\mathbf{x}$ è il vettore $n \times 1$ contenente le osservazioni campionarie della variabile esplicativa), il criterio dei minimi quadrati penalizzati risulta essere

$$||\mathbf{y} - \mathbf{X}\beta||_2^2 + \lambda \int [f''(t)]^2 dt, \tag{2.31}$$

dove l'integrale del quadrato della derivata seconda penalizza modelli che rappresentano $f(x)$ come una funzione troppo frastagliata. Il *trade-off* tra aderenza ai dati e penalizzazione viene controllato dal parametro di lisciamento λ, che assume valori nei reali positivi. Per $\lambda \to \infty$ si vincola f ad essere una funzione lineare, mentre per valori prossimi a 0 si ritorna alla stima non penalizzata. Poiché la funzione f è lineare nei parametri, la penalizzazione può essere sempre riscritta mediante una forma quadratica in β:

$$\int [f''(t)]^2 dt = \beta^T \mathbf{S} \beta, \tag{2.32}$$

dove la matrice $\mathbf{S}$ è nota e dipende dai nodi scelti e dalle funzioni di base utilizzate. In Gu & Gu (2013) viene riportata una formulazione alternativa per le *splines* cubiche

naturali rispetto a quella presentata nelle equazioni (2.29), valida quando il campo di variazione della variabile esplicativa è $(0, 1)$, che permette anche di definire facilmente gli elementi della matrice $\mathbf{S}$. Il problema di ottimizzazione risulta essere

$$||\mathbf{y} - \mathbf{X}\beta||_2^2 + \lambda\beta^T\mathbf{S}\beta \tag{2.33}$$

e minimizzando rispetto a β si ottiene la soluzione esplicita

$$\hat{\beta} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{S})^{-1}\mathbf{X}^T\mathbf{y}. \tag{2.34}$$

Analogamente a quanto succede per i modelli lineari, dove si evita la soluzione esplicita del sistema dei minimi quadrati e si favorisce l'approccio basato sulla scomposizione QR per la sua stabilità numerica, per la stima dei minimi quadrati penalizzati spesso nei pacchetti dedicati, come ad esempio pacchetto `mgcv` (Wood (2011)) di R, ci si riconduce alla stima di un modello non penalizzato. Per fare questo si sfrutta l'identità

$$\left\| \begin{bmatrix} \mathbf{y} \\ \mathbf{0} \end{bmatrix} - \begin{bmatrix} \mathbf{X} \\ \sqrt{\lambda}\mathbf{B} \end{bmatrix} \beta \right\|_2^2 = ||\mathbf{y} - \mathbf{X}\beta||_2^2 + \lambda\beta^T\mathbf{S}\beta, \tag{2.35}$$

dove $\mathbf{B}$ è una radice quadrata della matrice $\mathbf{S}$ tale che $\mathbf{B}^T\mathbf{B} = \mathbf{S}$. Dall'uguaglianza si nota come la somma dei quadrati penalizzati corrisponda ad un criterio ai minimi quadrati classico dove la matrice del modello è stata aumentata aggiungendo una radice quadrata della matrice di penalizzazione, mentre il vettore della risposta è stato aumentato con q zeri, dove q è la dimensione di $\mathbf{S}$. Utilizzando questa rappresentazione si può ottenere la soluzione penalizzata utilizzando la scomposizione QR della matrice $\mathbf{X}$ aumentata.

Questo approccio risulta direttamente adattabile ad una situazione di *Big Data*, mediante l'aggiornamento della scomposizione QR visto nella sottosezione 2.4.3. Alternativamente, si può scegliere di utilizzare il metodo per aggiornare le quantità $\mathbf{X}^T\mathbf{X}$ e $\mathbf{X}^T\mathbf{y}$ impiegato nell'algoritmo 3. Si ricorda che questo approccio può tornare utile nel caso in cui si voglia implementare la stima con un paradigma di programmazione come MapReduce, come visto in sezione 2.4.4. Una volta ottenute le quantità di interesse, per ottenere la stima non si può sfruttare la scomposizione di Cholesky come per i modelli lineari perché la matrice $\mathbf{X}^T\mathbf{X} + \lambda\mathbf{S}$ non sempre risulta simmetrica; bisogna quindi utilizzare l'approccio generico (e meno efficiente) alla soluzione dei sistemi lineari fornito dalla scomposizione LU.

Con questa formulazione, il problema di controllare la rugosità della funzione risultante diventa il problema di scegliere il parametro di lisciamento λ: questo può essere selezionato mediante convalida incrociata o qualsiasi altro metodo di valutazione oggettiva dell'errore di previsione (e.g. *bootstrap*, suddivisione in stima e verifica ...).

Nel caso di modello additivo con più di una covariata, ad ogni funzione $f_j(X_j)$ corrisponde una matrice di penalità S_j, che penalizza solo gli elementi corrispondenti a f_j nel vettore dei coefficienti. Una volta ottenuta la matrice del modello, risultante dall'espansione in basi per ogni variabile esplicativa, si ottiene un'unica penalità calcolando $\mathbf{S} = \sum_{j=1}^{p} \lambda_j \mathbf{S}_j$ e ci riconduce alla forma vista in equazione (2.33), dalla quale è possibile ottenere la stima del modello additivo con uno dei due metodi proposti.

Si è presentato il modello additivo in maniera estremamente sintetica: l'obiettivo è stato infatti quello di mettere in risalto come la stima di tali modelli possa essere ricondotta in alcuni casi alla stima dei modelli lineari, per evidenziare la possibilità di utilizzare alcuni degli algoritmi presentati per stimare modelli più complessi rispetto al modello lineare. Per approfondimenti su questo tipo di modelli, tra cui ulteriori estensioni e algoritmi di stima generici, si rimanda a Wood (2006), che offre una trattazione completa sul tema.

2.6 Problemi inferenziali

L'inferenza statistica è quel processo induttivo che punta a formulare affermazioni valide a livello di popolazione, partendo dall'informazione contenuta nel campione osservato.

Nella modellazione statistica le procedure inferenziali sono volte alla costruzione di intervalli di confidenza e test di ipotesi per il vero valore dei parametri stimati (e.g. i coefficienti di regressione β di un modello lineare), al confronto tra modelli (annidati e non) e alla costruzione di intervalli di previsione, cioè la definizione di una regione di valori plausibili in corrispondenza della previsione per una nuova unità. Nel caso di modelli parametrici, cioè quando la distribuzione del fenomeno di interesse viene completamente identificata da un numero finito di parametri, questi problemi si possono affrontare sfruttando la teoria della verosimiglianza, un approccio che gode di diverse proprietà di ottimalità. Questo metodo è basato sulla funzione di verosimiglianza, che sintetizza l'informazione contenuta nel campione osservato e permette di individuare i valori più verosimili per i parametri di interesse. In casi dove non sia possibile definire approcci di verosimiglianza, perché non sono verificate le assunzioni necessarie o per le caratteristiche del modello utilizzato, è necessario lo sviluppo di altri metodi.

Per una presentazione completa dell'inferenza basata sul concetto della verosimiglianza, si rimanda a Pace & Salvan (1996).

2.6.1 Inferenza nei modelli lineari

I risultati e le quantità presentate nella sezione 2.2 sono stati definiti senza il bisogno di particolari assunzioni distributive. Infatti, spesso è possibile pensare di poter approssimare la vera funzione di regressione $f(X)$ con una funzione lineare, mentre i minimi quadrati costituiscono un criterio generalmente valido per quantificare l'aderenza di un qualsiasi modello ai dati osservati. I risultati concernenti la stima ai minimi quadrati, come il fatto che la soluzione del problema identifichi la proiezione ortogonale di $\mathbf{y}$ nello spazio delle colonne di $\mathbf{X}$ e tutti gli approcci e algoritmi sviluppati per la stima, rimangono validi sotto la minima assunzione di poter approssimare la funzione di regressione con una funzione lineare.

Utilizzando delle assunzioni più specifiche, è possibile ottenere risultati aggiuntivi che portano allo sviluppo di tecniche inferenziali. Si ipotizzi di aver specificato correttamente il modello per il valore atteso condizionato di Y: $\mathbb{E}(Y|X) = f(X) = X\beta$. Aggiungendo le assunzioni campionarie di errori a media nulla, omoschedastici e incorrelati, si vanno a definire le ipotesi del secondo ordine. Il nome deriva dal fatto che queste ipotesi riguardano i primi due momenti del processo di errore (e quindi della risposta). Formalmente, si possono riassumere nel seguente modo:

$$y_i = x_i\beta + \epsilon_i, \quad \mathbb{E}(\epsilon_i) = 0, \quad Var(\epsilon) = \sigma^2, \quad i = 1, \ldots, n \tag{2.36}$$

Alternativamente, le assunzioni del secondo ordine possono essere scritte in termini della variabile risposta, cioè

$$\mathbb{E}(y_i) = x_i\beta, \quad Var(y_i) = \sigma^2, \quad i = 1, \ldots, n \tag{2.37}$$

Partendo da queste ipotesi, si possono calcolare i momenti dello stimatore $\hat{\beta}$ ottenuto come soluzione del problema dei minimi quadrati. Partendo da $\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$, per il valore atteso si ha $\mathbb{E}(\hat{\beta}) = \beta$, quindi lo stimatore ai minimi quadrati è uno stimatore non distorto del parametro β. Per individuare la matrice di varianza e covarianza del vettore stimato, si sfruttano i risultati sui momenti di combinazioni lineari di variabili casuali per ottenere

$$Var(\hat{\beta}) = \sigma^2(\mathbf{X}^T\mathbf{X})^{-1}. \tag{2.38}$$

La varianza del termine di errore σ^2 può essere stimata con la varianza campionaria $\hat{\sigma}^2 = \frac{1}{n}(\mathbf{y} - \mathbf{X}\hat{\beta})^T(\mathbf{y} - \mathbf{X}\hat{\beta})$. Questo stimatore risulta distorto, per questo gli viene spesso preferita la variante non distorta:

$$s^2 = \frac{1}{n-p}(\mathbf{y} - \mathbf{X}\hat{\beta})^T(\mathbf{y} - \mathbf{X}\hat{\beta}). \tag{2.39}$$

Qui p è il numero di variabili esplicative del modello, intercetta compresa. Questo stimatore, a differenza di quello basato sul metodo dei momenti, risulta non distorto, ossia $\mathbb{E}(s^2) = \sigma^2$. Basandosi sulle assunzioni del secondo ordine, valgono anche tre ulteriori risultati, di carattere più tecnico ma non per questo meno importanti. Il primo è il teorema di Gauss-Markov, che afferma come lo stimatore ai minimi quadrati sia il più efficiente (i.e. a varianza minima) tra tutti gli stimatori lineari non distorti. Gli altri due sono risultati asintotici, validi per $n \to \infty$ sotto ulteriori assunzioni sugli errori, che garantiscono la consistenza dello stimatore $\hat{\beta}$ e la validità dell'approssimazione normale $\hat{\beta} \dot{\sim} \mathcal{N}_p(\beta, \sigma^2(\mathbf{X}^T\mathbf{X})^{-1})$. Quest'ultimo risultato è particolarmente utile per sviluppare procedure inferenziali quando n è elevato, senza il bisogno di ulteriori assunzioni distributive.

L'assunzione distributiva più utilizzata storicamente è l'assunzione di gaussianità per l'errore ϵ e di conseguenza per la risposta Y. Formalizzando tale assunzione, in aggiunta alle ipotesi del secondo ordine, si ottiene

$$y_i = x_i\beta + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2), \quad i = 1, \ldots, n \tag{2.40}$$

o in formato matriciale

$$\mathbf{Y} \sim \mathcal{N}_n(\mathbf{X}\beta, \sigma^2\mathbf{I}_n), \tag{2.41}$$

dove $\mathcal{N}_n(\mu, \Sigma)$ indica una variabile casuale variabile normale multivariata con vettore medio μ e matrice di varianza e covarianza Σ.

Utilizzando i risultati noti sulle combinazioni lineari di variabili casuali normali, basandosi sull'assunzione di normalità è facile mostrare che

$$\hat{\beta} \sim \mathcal{N}_p(\beta, \sigma^2(\mathbf{X}^T\mathbf{X})^{-1}), \tag{2.42}$$

e

$$(n-p)s^2 \sim \sigma^2\mathcal{X}^2_{n-p}. \tag{2.43}$$

Nelle formule precedenti, $\mathcal{N}_p$ indica una variabile casuale normale multivariata con p componenti e $\mathcal{X}^2_{n-p}$ indica una distribuzione chi-quadro con $n-p$ gradi di libertà. Si noti

come nella formula (2.42) la distribuzione normale per lo stimatore ai minimi quadrati valga in maniera esatta, a differenza dell'approssimazione normale per $n \to \infty$, che vale anche senza l'assunzione di gaussianità per la risposta, ma solo in maniera approssimata. In aggiunta a quanto riportato, vale un risultato che afferma come gli stimatori $\hat{\beta}$ e s^2 siano indipendenti.

Sulla base di questi risultati distributivi, è possibile sviluppare approcci inferenziali per i risultati della stima di un modello lineare. Un problema tipico affrontato nell'inferenza è quello di identificare delle procedure per verificare ipotesi sui parametri di un modello. Per verificare l'ipotesi che un coefficiente $\beta_j = 0$, una situazione standard nel caso dei modelli lineari in quanto permette di verificare la rilevanza di una variabile nella previsione della risposta, si forma il coefficiente standardizzato o *Z-score*

$$z_j = \frac{\hat{\beta}_j}{s\sqrt{v_j}}, \tag{2.44}$$

dove v_j è il j-esimo elemento della diagonale principale di $(\mathbf{X}^T\mathbf{X})^{-1}$. Sotto l'ipotesi nulla che $\beta_j = 0$, z_j si distribuisce come una t_{n-p} (una distribuzione t di Student con $n-p$ gradi di libertà), e si ha che un valore (assoluto) elevato indica evidenza empirica contro l'ipotesi nulla e porta al suo rifiuto. Se nella formula (2.44) si utilizzasse la vera deviazione standard σ invece che la sua stima s, la distribuzione di riferimento sarebbe una normale standard. Comunque, la differenza nel comportamento delle code tra una distribuzione normale e una distribuzione t_{n-p} diventa poco rilevante al crescere della numerosità campionaria (per $n \geq 50$), pertanto di frequente nella pratica si utilizzano i quantili della distribuzione normale per identificare valori anomali di z_j.

Spesso si desidera verificare la significatività di un gruppo intero di coefficienti. Per esempio, per testare se una variabile qualitativa con k livelli può essere esclusa dal modello, bisogna verificare se tutti i coefficienti associati ai livelli di quella variabile possono essere simultaneamente messi a zero. Questo equivale a confrontare due modelli annidati: quello completo con $p_1 = p$ parametri e quello ridotto con p_0 parametri, in cui i $p_1 - p_0$ coefficienti associati alla verifica di ipotesi in questione vengono fissati a zero. Per raggiungere questo obiettivo, si utilizza la statistica F,

$$F = \frac{(RSS_1 - RSS_0)/(p_1 - p_0)}{RSS_1/(n - p_1)}, \tag{2.45}$$

dove RSS_1 è la somma dei quadrati dei residui per il modello completo e RSS_0 la stessa quantità per il modello ridotto. La statistica F quantifica la variazione nella somma dei quadrati residui per ogni parametro aggiuntivo nel modello più grande, scalata per

una stima della varianza σ^2. Sotto le assunzioni di normalità e l'ipotesi nulla che il modello ridotto sia quello corretto, la statistica F segue una distribuzione $\mathcal{F}_{p_1-p_0,n-p_1}$ (distribuzione di Fisher-Snedecor con $p_1 - p_0$ e $n - p_1$ gradi di libertà). Si può mostrare come la statistica z_j calcolata in (2.44) sia equivalente alla statistica F calcolata per rimuovere il singolo coefficiente β_j dal modello. Per grandi valori di n, i quantili di una $\mathcal{F}_{p_1-p_0,n-p_1}$ tendono a quelli di una $\mathcal{X}^2_{p_1-p_0}$, che può essere usata come approssimazione. Per testare ipotesi più complesse, del tipo $\beta_1 + 4\beta_2 = 3$, un approccio possibile è quello di scrivere i vincoli da testare come un sistema lineare del tipo $\mathbf{C}\beta = \mathbf{u}$. A questo punto si possono utilizzare i risultati sulla distribuzione normale per calcolare la distribuzione di $\mathbf{C}\beta$, dalla quale poi si può trovare la quantità pivotale necessaria per svolgere le procedure di verifica di ipotesi.

Un altro problema inferenziale di rilievo è quello di costruire intervalli di confidenza con livello di copertura fissato per i coefficienti β_j. Precisamente, la costruzione di intervalli di confidenza è strettamente collegata alle procedure di verifica di ipotesi: un intervallo di confidenza per un parametro con livello di copertura fissato corrisponde alla regione di accettazione per un test di verifica d'ipotesi bilaterale con lo stesso livello su quel parametro. Si parta dalla normalità del vettore β, cioè $\hat{\beta} \sim \mathcal{N}_p(\beta, \sigma^2(\mathbf{X}^T\mathbf{X})^{-1})$. Marginalmente, questo implica che

$$\hat{\beta}_j \sim \mathcal{N}(\beta_j, \sigma^2 v_j), \tag{2.46}$$

con v_j definito come prima. Vale allora che

$$1 - \alpha = \mathbb{P}\left\{ \frac{|\hat{\beta}_j - \beta_j|}{s\sqrt{v_j}} < t_{n-p,1-\alpha/2} \right\} \tag{2.47}$$

dove $t_{n-p,1-\alpha/2}$ è il quantile di livello $1-\alpha/2$ di una distribuzione t di Student con $n-p$ gradi di libertà. L'uguaglianza (2.47) mostra come per un intervallo di confidenza con livello di copertura pari a $1 - \alpha$ si possa ottenere come

$$IC_{1-\alpha}(\beta_j) = \left[\hat{\beta}_j - t_{n-p,1-\alpha/2} s\sqrt{v_j}, \hat{\beta}_j + t_{n-p,1-\alpha/2} s\sqrt{v_j}\right]. \tag{2.48}$$

Anche in questo caso viene spesso utilizzata l'approssimazione normale per una t di Student con gradi di libertà elevati. Da questo risultato origina anche la procedura semplificata di riportare $\hat{\beta} \pm 2 \cdot se(\hat{\beta})$, che è un intervallo di confidenza con copertura approssimativamente pari al 95% ($z_{0.975} = 1.96 \approx 2$). Anche nel caso in cui non fosse valida l'assunzione di normalità, l'intervallo risulta approssimativamente corretto, con livello di copertura nominale valido per $n \to \infty$. In maniera simile, sfruttando i risultati

noti per la distribuzione normale multivariata, si può ottenere una regione di confidenza per tutti gli elementi del vettore β simultaneamente. La regione viene ottenuta come

$$IC_{1-\alpha}(\beta) = \left\{\beta|(\hat{\beta}-\beta)^T\mathbf{X}^T\mathbf{X}(\hat{\beta}-\beta) \leq s^2\mathcal{X}_{p;1-\alpha}\right\}, \tag{2.49}$$

dove $\mathcal{X}_{p;1-\alpha}$ è il quantile di livello $1-\alpha$ di una distribuzione chi-quadro con p gradi di libertà.

L'ultimo problema inferenziale considerato è quello della costruzione di intervalli di previsione. Successivamente alla stima del modello basata sul campione osservato, si supponga di voler ottenere la previsione per una nuova unità, con il suo set di predittori. Sia x^* il vettore contenente le variabili esplicative (per comodità si assuma che il vettore contenga già 1 in prima posizione), vale quindi $y^* = x^{*T}\beta + \epsilon^*$. La stima puntuale sarà il valore atteso condizionato al valore assunto da x^*, quindi si ha $\hat{y}^* = x^{*T}\hat{\beta}$. Per calcolare la varianza del predittore lineare fornito, si ha $Var(\hat{y}^*) = Var(x^{*T}\hat{\beta}) = \sigma^2 x^{*T}(\mathbf{X}^T\mathbf{X})x^*$.

Nel momento in cui si desidera costruire un intervallo per la previsione di y^*, bisogna tenere in considerazione due fonti di variabilità: quella derivante dalla stima del predittore lineare sulla base del campione precedentemente osservato e quella intrinseca di una nuova osservazione che segue il processo generatore dei dati assunto per il fenomeno. Si ha quindi

$$\begin{aligned} Var(y^*) &= Var(x^{*T}\hat{\beta} + \epsilon^*) \\ &= Var(x^{*T}\hat{\beta}) + Var(\epsilon^*) \\ &= \sigma^2\sqrt{1 + x^{*T}(\mathbf{X}^T\mathbf{X})x^*}. \end{aligned} \tag{2.50}$$

Sfruttando la normalità della nuova osservazione, si ha

$$y^* \sim \mathcal{N}(x^{*T}\hat{\beta}, \sigma^2\sqrt{1 + x^{*T}(\mathbf{X}^T\mathbf{X})x^*}), \tag{2.51}$$

e si può utilizzare questo risultato per costruire intervalli di previsione. Si ipotizzi che la dimensione campionaria dell'insieme di stima sia abbastanza elevata, in modo da poter sfruttare l'approssimazione normale per la t di Student. Un intervallo di previsione con livello di copertura nominale approssimato pari a $1-\alpha$ è dato da

$$IC_{1-\alpha}(y^*) \approx \left\{x^{*T}\hat{\beta} - z_{1-\alpha/2}\sqrt{Var(y^*)}, x^{*T}\hat{\beta} + z_{1-\alpha/2}\sqrt{Var(y^*)}\right\}. \tag{2.52}$$

Le procedure inferenziali presentate possono essere viste come metodologie basate sulla verosimiglianza: la stima ai minimi quadrati per un modello lineare è equivalente

alla massimizzazione della verosimiglianza sotto l'assunzione di normalità. Di conseguenza, le procedure considerate godono di numerose proprietà: la potenza dei test per la verifica d'ipotesi, la copertura nominale degli intervalli di confidenza e di previsione, la consistenza delle stime. Tuttavia, tutti questi risultati sono basati sulle assunzioni del secondo ordine o di normalità. Quest'ultima è difficilmente verificabile in situazioni pratiche, ma in realtà anche le assunzioni del secondo ordine sono in qualche modo stringenti: senza considerare eventuale correlazione o eteroschedasticità dei residui, nella pratica anche la sola assunzione di linearità può risultare forzata. Come detto all'inizio del capitolo, spesso si può solo decidere di voler approssimare la vera funzione di regressione $f(X)$ con una funzione lineare; questo è molto diverso dall'affermare di aver specificato correttamente il modello per il valore atteso condizionato. Se questa assunzione non vale, non è possibile neanche sfruttare la normalità asintotica delle stime per costruire procedure inferenziali approssimate. Per questi motivi, nel caso dove non si possano verificare le assunzioni distributive, è necessario l'utilizzo di approcci alternativi all'inferenza, si pensi ad esempio al *bootstrap* (si veda Efron (1979)).

Per il materiale relativo all'inferenza nell'ambito dei modelli lineari si rimanda a Hastie et al. (2009) e Azzalini & Scarpa (2012), che approfondiscono anche i collegamenti tra l'approccio di massima verosimiglianza e i risultati della modellazione lineare. Alcuni riferimenti si possono trovare anche nei capitoli dedicati di Salvan et al. (2020). Un'altra fonte autorevole nell'ambito è Weisberg (2005), al quale si rimanda per approfondimenti riguardanti gli intervalli di previsione e gli aspetti più applicati dell'analisi tramite modelli lineari.

2.6.2 Conformal inference per inferenza predittiva

Nel tempo ci sono stati numerosi sforzi volti allo sviluppo di procedure inferenziali basate su meno assunzioni possibili, in concomitanza allo sviluppo di modelli semi-parametrici e non parametrici (*suppor vector machines*, foreste casuali, *k-nearest neighbors* ...). Di seguito ci si focalizzerà su un approccio sviluppato per affrontare il problema dell'inferenza predittiva, quindi della costruzione di intervalli di previsione.

Questa metodologia prende il nome di *conformal inference* e viene sviluppata in Lei et al. (2018) sulla base delle tecniche introdotte in Vovk et al. (2005). Per il restante della sottosezione, si faccia riferimento al primo dei testi citati.

Si utilizza in questo contesto una formalizzazione leggermente diversa del problema di regressione, utile per le manipolazioni probabilistiche necessarie per calcolare le proprietà delle tecniche proposte. Si considerino dati di regressione i.i.d (indipendenti ed

identicamente distribuiti) del tipo $Z_1, \ldots, Z_n \sim P$, dove ogni $Z_i = (X_i, Y_i)$ è una variabile casuale definita in $\mathbb{R}^p \times \mathbb{R}$, composta dalla variabile risposta Y_i e da un vettore p-dimensionale di covariate $X_i = [X_{i1}, \ldots, X_{ip}]$. Per comodità si assuma di includere un vettore di 1 nella matrice del modello X. Sia $\mu(x) = \mathbb{E}(Y|X = x)$ la funzione di regressione. L'obiettivo è quello di costruire delle bande di previsione per una nuova risposta Y_{n+1} avendo osservato il vettore X_{n+1}, senza assunzioni su μ o P. Dato un livello di copertura nominale $1 - \alpha$, si vuole costruire una banda $C \subseteq \mathbb{R}^p \times \mathbb{R}$ basata su $Z_1, \ldots, Z_n$ tale che

$$\mathbb{P}\left(Y_{n+1} \in C(X_{n+1})\right) \geq 1 - \alpha, \tag{2.53}$$

dove la probabilità è calcolata sulle $n+1$ istanze $Z_1, \ldots, Z_n, Z_{n+1}$ e per un punto $x \in \mathbb{R}^p$ si scrive $C(x) = \{y \in \mathbb{R} : (x, y) \in C\}$.

L'idea di base dietro a questo approccio è collegata a un semplice risultato riguardante i quantili empirici. Siano $U_1, \ldots, U_n$ delle estrazioni i.i.d. da una variabile casuale scalare. Per un livello di copertura fissato $1 - \alpha$, considerando un'ulteriore estrazione U_{n+1}, si ha

$$\mathbb{P}\left(U_{n+1} \leq \hat{q}_{1-\alpha}\right) \geq 1 - \alpha, \tag{2.54}$$

con

$$\hat{q}_{1-\alpha} = \begin{cases} U_{(\lceil (n+1)(1-\alpha) \rceil)} & \text{se } \lceil (n+1)(1-\alpha) \rceil \leq n \\ \infty & \text{altrimenti,} \end{cases} \tag{2.55}$$

dove $\hat{q}_{1-\alpha}$ è il quantile empirico di livello $1-\alpha$ basato su $U_1, \ldots, U_n$ e $U_{(1)}, \ldots, U_{(n)}$ definiscono le statistiche d'ordine di $U_1, \ldots, U_n$. La proprietà di copertura per campioni finiti in (2.54) si verifica facilmente: per scambiabilità, il rango di U_{n+1} tra $U_1, \ldots, U_n, U_{n+1}$ è uniformemente distribuito nell'insieme $\{1, \ldots, n+1\}$. Nel problema di regressione definito, dove si osservano le estrazioni i.i.d $Z_i = (X_i, Y_i) \in \mathbb{R}^p \times \mathbb{R} \sim P$, $i = 1, \ldots, n$, si può utilizzare questo risultato per costruire un intervallo di previsione *naive* per Y_{n+1} in corrispondenza del valore X_{n+1} osservato, considerando la coppia (X_{n+1}, Y_{n+1}) come un'estrazione casuale da P. Seguendo l'idea precedente, si può formare un intervallo di previsione con copertura $1 - \alpha$ definito da

$$C_{naive}(X_{n+1}) = \left[\hat{\mu}(X_{n+1}) - \hat{F}_n^{-1}(1-\alpha), X_{n+1} + \hat{F}_n^{-1}(1-\alpha)\right], \tag{2.56}$$

dove $\hat{\mu}$ è un qualsiasi stimatore della funzione di regressione, $\hat{F}_n$ è la funzione di ripartizione empirica dei residui osservati in valore assoluto ($|Y_i - \hat{\mu}(X)_i|$, $i = 1, \ldots, n$) e $\hat{F}_n^{-1}(1-\alpha)$ è il quantile empirico di livello $1 - \alpha$ di $\hat{F}_n$. Questo intervallo risulta molto semplice da costruire, ma ha il difetto di essere valido solo sotto alcune assunzioni di

regolarità per P e per lo stimatore $\hat{\mu}$: non garantisce quindi la copertura nominale per campioni finiti senza utilizzare assunzioni su P e μ. Inoltre, generalmente offre povera copertura empirica, in quanto spesso la distribuzione dei residui nell'insieme utilizzato per la stima risulta distorta.

Il metodo *conformal prediction* supera i difetti degli intervalli naive, e fornisce intervalli che garantiscono il livello di copertura nominale non asintotica desiderato senza il bisogno di assunzioni su P e μ. Si consideri la seguente strategia: per ogni valore $y \in \mathbb{R}$, si costruisca uno stimatore della funzione di regressione "aumentato" $\hat{\mu}_y$, stimato sulla base del campione aumentato $Z_1, \ldots, Z_n, (X_{n+1}, y)$. Ora si definiscano

$$\begin{aligned} R_{y,i} &= |Y_i - \hat{\mu}_y(X_i)|, \text{ per } i = 1, \ldots, n \\ R_{y,n+1} &= |y - \hat{\mu}_y(X_{n+1})|, \end{aligned} \tag{2.57}$$

e si trovi il rango di $R_{y,n+1}$ fra i rimanenti residui, calcolando

$$\pi(y) = \frac{1}{n+1} \sum_{i=1}^{n+1} \mathbb{1}\{R_{y,i} \leq R_{y,n+1}\}, \tag{2.58}$$

cioè la proporzione di punti nel campione aumentato il cui corrispondente residuo risulta minore dell'ultimo, $R_{y,n+1}$. Per la scambiabilità dei dati osservati, si ha che la statistica $\pi(Y_{n+1})$ (si noti come ora venga calcolata nel punto Y_{n+1}) è uniformemente distribuita su $\{1/(n+1), 2/(n+1), \ldots, 1\}$, il che implica

$$\mathbb{P}\left((n+1)\pi(Y_{n+1}) \leq \lceil (1-\alpha)(n+1) \rceil\right) \geq 1 - \alpha. \tag{2.59}$$

La proprietà precedente si può interpretare vedendo $1 - \pi(Y_{n+1})$ come un p-value conservativo per testare l'ipotesi nulla $H_0 : Y_{n+1} = y$. Invertendo un simile test su tutti i possibili valori di $y \in \mathbb{R}$, la proprietà 2.59 porta alla costruzione dell'intervallo *conformal prediction* di livello $1 - \alpha$ per X_{n+1}, dato da

$$C_{conf}(x) = \{y \in \mathbb{R} : (n+1)\pi(y) \leq \lceil (1-\alpha)(n+1) \rceil\}. \tag{2.60}$$

Nella pratica si restringe la ricerca in (2.60) ad una griglia discreta di valori di prova per y. I passaggi vengono sintetizzati nell'algoritmo 5 riportato di seguito, dove si ottengono bande di previsione per nuovo insieme di covariate $\mathcal{X}_{new}$.

Per costruzione, l'intervallo C_{conf} fornisce la copertura desiderata per campioni finiti. Inoltre è anche accurato, cioè non risulta troppo esteso intorno alla previsione (si ricordi che un intervallo del tipo $[-\infty, \infty]$ offre una copertura perfetta ma è inutile). Queste

Algoritmo 5: Conformal Prediction

Dati: dati (x_i, y_i) per $i = 1, \ldots, n$, livello di copertura desiderato $1 - \alpha$, algoritmo di regressione $\mathcal{A}$, punti $\mathcal{X}_{new}$ ai quali costruire intervalli di previsione, valori $\mathcal{Y}_{trial} = \{y_1, y_2, \ldots\}$ da utilizzare come valori di prova

Risultato: Intervalli di previsione per ogni elemento di $\mathcal{X}_{new}$

per $x \in \mathcal{X}_{new}$ **fai**
  per $y \in \mathcal{Y}_{trial}$ **fai**
    $\hat{\mu}_y = \mathcal{A}\left(\{(x_1, y_1), \ldots, (x_n, y_n), (x, y)\}\right)$;
    $R_{y,i} = |y_i - \hat{\mu}_y(x_i)|$, per $i = 1, \ldots, n$;
    $R_{y,n+1} = |y - \hat{\mu}_y(x)|$;
    $\pi(y) = (1 + \sum_{i=1}^{n} \mathbb{1}\{R_{y,i} \leq R_{y,n+1}\})/(n+1)$;
  fine
  $C_{conf}(x) = \{y \in \mathcal{Y}_{trial} : (n+1)\pi(y) \leq \lceil (1-\alpha)(n+1) \rceil\}$;
fine
Restituire $C_{conf}(x)$, per ogni $x \in \mathcal{X}_{new}$;

due proprietà sono sintetizzate nella disuguaglianza

$$1 - \alpha \leq \mathbb{P}\left(Y_{n+1} \in C_{conf}(X_{n+1})\right) \leq 1 - \alpha + \frac{1}{n+1}, \tag{2.61}$$

che risulta valida sotto l'unica assunzione aggiuntiva che i residui in valore assoluto abbiano distribuzione congiunta continua, un'assunzione debole che serve solo per evitare *ties* nell'ordinamento, e garantisce la convergenza del livello di copertura nominale per campioni finiti a $1 - \alpha$ (Teorema 1, Lei et al. (2018)).

Il metodo *conformal prediction* appena esposto risulta essere computazionalmente intensivo. Per ogni X_{n+1} a disposizione, per costruire una banda di previsione per Y_{n+1} è necessario definire una griglia di valori per y, stimare nuovamente il modello sui dati aumentati per ogni valore della griglia, calcolare e riordinare i residui. Nel caso della modellazione lineare, con un appropriato utilizzo della formula di Sherman-Morrison 2.20 si può evitare il problema di ricalcolare la stima, ma per modelli semi-parametrici o non parametrici l'applicazione della procedura può risultare impossibile con le capacità computazionali a disposizione.

Per risolvere questo problema viene sviluppato un approccio, chiamato *split conformal prediction*, che mantiene la generalità del precedente con un costo computazionale molto ridotto. Il metodo *split conformal* separa le fasi di adattamento del modello e ordinamento dei residui suddividendo a l'insieme di dati a disposizione: il suo costo computazionale è dominato dalla fase di stima, che viene eseguita una volta sola. La procedura viene riportata in Lei et al. (2018) e descritta di seguito nell'algoritmo 6.

In maniera a simile a quanto visto nella disuguaglianza (2.61) per quanto riguarda le

Algoritmo 6: Split Conformal Prediction

Dati: dati (x_i, y_i) per $i = 1, \dots, n$, livello di copertura desiderato $1 - \alpha$, algoritmo di regressione $\mathcal{A}$

Risultato: Intervalli di previsione per $x \in \mathbb{R}^p$

1 Si divida casualmente $\{1, \dots, n\}$ nei due sottoinsiemi di uguale dimensione $\mathcal{I}_1$, $\mathcal{I}_2$;
2 $\hat{\mu} = \mathcal{A}(\{(x_i, y_i) : i \in \mathcal{I}_1\})$;
3 $R_i = |y_i - \hat{\mu}(x_i)|$, $i \in \mathcal{I}_2$;
4 Sia d il k-esimo valore più piccolo di $\{R_i : i \in \mathcal{I}_2\}$, con $k = \lceil (n/2 + 1)(1 - \alpha) \rceil$;
5 Restituire $C_{split}(x) = [\hat{\mu}(x) - d, \hat{\mu}(x) + d]$, per ogni $x \in \mathbb{X}^p$;

proprietà di copertura empirica dell'approccio *conformal*, per il metodo *split conformal* vale che

$$1 - \alpha \leq \mathbb{P}\left(Y_{n+1} \in C_{split}(X_{n+1})\right) \leq 1 - \alpha + \frac{2}{n+2}. \tag{2.62}$$

Quando si ha a disposizione un insieme di dati (x_i, y_i), $i = 1 \dots, n$ e un metodo per ottenere bande di previsione, può essere di interesse calcolare le bande per tutti i punti x_i dell'insieme di stima. I metodi *conformal* e *split conformal* visti in precedenza sono formulati per fornire una banda di previsione valida per un punto X_{n+1} dalla stessa distribuzione di (X_i, Y_i) ma non ancora osservato. Una soluzione semplice sarebbe quella di trattare ogni X_i come una nuova variabile e usare i restanti $n - 1$ punti per una delle due procedura definite, ma questo approccio moltiplica per n la complessità computazionale della procedura scelta, inoltre risulta problematico calcolare le proprietà di copertura. Per superare questi limiti, nell'articolo di riferimento viene sviluppata una variante del metodo *split conformal*, dove la fase di ordinamento è svolta con un approccio *leave-one-out*. Questa variante prende il nome di *rank-one-out split conformal*, e viene presentata nell'algoritmo 7.

Una variante ancora più semplice della procedura *rank-one-out* approssima gli intervalli $C_{roo}(x_i)$ con

$$\tilde{C}_{roo}(x_i) = [\hat{\mu}_k(x_i) - \tilde{d}_k, \hat{\mu}_k(x_i) + \tilde{d}_k], \tag{2.63}$$

dove $\tilde{d}_k$ è l'm-esimo valore più piccolo nell'insieme $\{R_j : j \notin \mathcal{I}_k\}$, per $m = \lceil (1-\alpha)n/2 \rceil$. In questo modo, bisogna calcolare e ordinare i quantili solo una volta per ogni *split*, ma l'intervallo corrispondente risulterà più ampio della controparte C_{roo}.

In conclusione, gli ultimi due metodi presentati permettono di ottenere bande di previsione con livello di copertura desiderato non asintotico in corrispondenza di tutti i punti x_i di un *dataset* osservato, senza nessuna assunzione sulla natura dei dati o della funzione di regressione da estimare, e senza alcuna restrizione rispetto all'algoritmo di stima impiegato.

Algoritmo 7: Rank-One-Out Split Conformal

Dati: dati (x_i, y_i) per $i = 1, \ldots, n$, livello di copertura desiderato $1 - \alpha$, algoritmo di regressione $\mathcal{A}$

Risultato: Intervalli di previsione per ogni x_i, $i = 1, \ldots, n$

Si divida casualmente $\{1, \ldots, n\}$ nei due sottoinsiemi di uguale dimensione $\mathcal{I}_1$, $\mathcal{I}_2$;
per $k \in \{1, 2\}$ **fai**
  $\hat{\mu}_k = \mathcal{A}(\{(x_i, y_i) : i \in \mathcal{I}_k\})$;
  per $i \notin \mathcal{I}_k$ **fai**
    $R_i = |y_i - \hat{\mu}_k(x_i)|$;
  fine
  per $i \notin \mathcal{I}_k$ **fai**
    Sia d_i il m-esimo valore più piccolo di $\{R_j : j \notin \mathcal{I}_k, j \neq i\}$, con $m = \lceil n/2(1-\alpha) \rceil$;
    $C_{roo}(x_i) = [\hat{\mu}_k(x_i) - d_i, \hat{\mu}_k(x_i) + d_i]$;
  fine
fine
Restituire $C_{roo}(x_i)$, per $i = 1, \ldots, n$;

2.7 Approcci alternativi

In questa sezione si presentano brevemente due ulteriori metodologie legate alle tematiche considerate nel capitolo. Queste vengono riportate per completezza e pertinenza agli argomenti trattati, tuttavia non verranno utilizzate nelle analisi pratiche riportate nel capitolo 3. Il primo degli approcci esposti è un algoritmo di stima distribuita per il modello lineare, sviluppato secondo un approccio MapReduce e basato sulla scomposizione QR. Il secondo è un approccio generico per applicare procedure inferenziali basate sulla verosimiglianza in maniera efficiente in contesti di architetture distribuite, che prende il nome di *Communication-Efficient Surrogate Likelihood*.

2.7.1 Modello lineare distribuito con MapReduce

Il metodo in analisi, introdotto in Adjout & Boufares (2014), riguarda un'implementazione in parallelo della regressione lineare, che utilizzando un approccio MapReduce su *framework* distribuiti permette di stimare i coefficienti del modello a partire da grandi masse di dati. Il calcolo viene basato sulla scomposizione QR e i minimi quadrati ordinari.

Il risultato di partenza è quello riportato in (2.12): utilizzando la scomposizione QR della matrice del modello $\mathbf{X}$ $n \times p$, si riesce a riscrivere il sistema di equazioni normali come $\mathbf{R}\beta = \mathbf{Q}^T\mathbf{y}$, dove $\mathbf{y}$ è il vettore contenente le osservazioni della variabile risposta.

La procedura è costituita da due parti. La prima parte prende in ingresso la matrice del modello $\mathbf{X}$ e la scompone in k blocchi, ottenendo $\mathbf{X}_k$ $(m \times p)$ per $k = 1, \ldots, K$. Successivamente la funzione Map calcola la scomposizione QR locale per ogni $\mathbf{X}_k$, ottenendo le matrici $\mathbf{Q}_{1,k}$ $(m \times p)$ e $\mathbf{R}_k$ $(p \times p)$ per $i = 1 \ldots, K$, che vengono passate alla funzione Reduce. A questo punto le coppie chiave-valore $(k, \mathbf{Q}_{1,k})$ vengono scritte nel *file* di uscita del primo passaggio. Le matrici $\mathbf{R}_k$ vengono combinate per ottenere $\mathbf{R}_{temp}$ $(pK \times p)$. A questa viene applicata una seconda scomposizione QR, ottenendo $\mathbf{Q}_2\mathbf{R}_{final}$. La matrice $\mathbf{R}_{final}$ viene salvata con chiave R, mentre la matrice $\mathbf{Q}_2$ viene suddivisa nuovamente in blocchi, emessi come $(k, \mathbf{Q}_{2,k})$. La prima parte di algoritmo emette quindi come risultato un *file* con le coppie chiave-valore $(k, \mathbf{Q}_{1,k})$, $(k, \mathbf{Q}_{2,k})$ per $k = 1, \ldots, K$ e $(R, \mathbf{R}_{final})$.

La seconda parte della procedura prende in ingresso i risultati della prima parte e il vettore $\mathbf{y}$ contenente i valori della risposta. La funzione Map scompone il vettore nei blocchi $\mathbf{y}_k$ e li manda alla funzione Reduce con chiave k, insieme alle matrici $\mathbf{Q}_{1,k}$ e $\mathbf{Q}_{2,k}$, per $k = 1, \ldots k$. La matrice $\mathbf{R}_{final}$ viene salvata, in quanto sarà utilizzata per la stima finale dei coefficienti. La funzione Reduce prende in ingresso le coppie $(k, \{\mathbf{Q}_{1,k}, \mathbf{Q}_{2,k}, \mathbf{y}_k\})$ e calcola le quantità $\mathbf{V}_k = (\mathbf{Q}_{1,k}\mathbf{Q}_{2,k})^T\mathbf{y}_k$. Avendo ottenuto queste quantità di interesse, si possono ottenere le stime del modello lineare risolvendo il sistema $\mathbf{R}_{final}\beta = \sum_k \mathbf{V}_k$ mediante la *backward substitution*.

Per approfondimenti sull'implementazione della procedura descritta si rimanda all'articolo citato, dove si può trovare anche un'analisi dei tempi di esecuzione al variare delle dimensioni della matrice del modello e del tipo di architettura distribuita utilizzata.

2.7.2 *Communication-efficient surrogate likelihood*

Questa metodologia, introdotta in Jordan et al. (2019), definisce un approccio efficiente per lo sviluppo di procedure basate sulla verosimiglianza in contesti dove i dati siano distribuiti su diverse macchine. L'approccio si basa sulla costruzione di un surrogato della funzione di verosimiglianza, che permette di raggiunge una precisione statistica soddisfacente e un costo di comunicazione ridotto (si vuole evitare quanto possibile il passaggio di dati tra macchine).

Siano $\mathbf{Z}_1^N = \{\mathbf{Z}_{ij}, i = 1, \ldots, n, j = 1, \ldots, k, N = nk\}$ osservazioni indipendenti ed identicamente distribuite con distribuzione marginale $\mathbb{P}_{\theta^*}$, dove $\{\mathbb{P}_\theta : \theta \in \Theta\}$ è la famiglia di modelli parametrizzati da $\theta \in \Theta \subset \mathbb{R}^d$, Θ è lo spazio parametrico e θ è il vero parametro della distribuzione generatrice dei dati. Si supponga che i dati siano immagazzinati in maniera distribuita: ogni macchina, per $j = 1, \ldots, k$, contiene la partizione $\mathbf{Z}_k$. L'obiettivo è quello di condurre procedure di inferenza statistica sul

parametro θ, tenendo in considerazione il costo di comunicazione tra le macchine. Sia $\mathcal{L} : \Theta \times \mathcal{Z} \to \mathbb{R}$ una funzione di perdita due volte differenziabile, minimizzata a livello di popolazione dal vero parametro del modello θ, cioè: $\theta = \text{argmin}_\theta \mathbb{E}_{\theta^*}[\mathcal{L}(\theta; \mathbf{Z})]$. Un esempio classico di funzione di perdita è la log-verosimiglianza negativa. Si definiscono la funzione di perdita locale e globale:

$$\mathcal{L}_j(\theta) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(\theta, z_{ij}), \quad j = 1, \ldots, k,$$
$$\mathcal{L}_N(\theta) = \frac{1}{N} \sum_{j=1}^{k} \sum_{i=1}^{n} \mathcal{L}(\theta, z_{ij}) = \frac{1}{k} \sum_{j=1}^{k} \mathcal{L}_j(\theta).$$

Qui $\mathcal{L}_j(\theta)$ è la funzione di perdita valutata in θ utilizzando solo i dati immagazzinati nella macchina j.

Per costruire la *communication-efficient surrogate likelihood* (CLS), si parte dall'espansione in serie di Taylor per $\mathcal{L}_N$:

$$\mathcal{L}_N(\theta) = \mathcal{L}_N(\bar{\theta}) + \langle \nabla \mathcal{L}_N(\bar{\theta}), \theta - \bar{\theta} \rangle + \sum_{j=2}^{\infty} \frac{1}{j!} \nabla^j \mathcal{L}_N(\bar{\theta})(\theta - \bar{\theta})^{\otimes j}.$$

Nella formula precedente $\bar{\theta}$ è uno stimatore iniziale per θ, ad esempio $\text{argmin}_\theta \mathcal{L}_1(\theta)$, la stima ottenuta utilizzando solo i dati della prima macchina. Poiché i dati sono divisi tra tutte le macchine, la valutazione delle derivate comporta la necessità di metterle in comunicazione tra loro. Questo ragionamento porta alla costruzione di un'approssimazione a $\mathcal{L}_N(\theta)$ che utilizza derivate locali:

$$\tilde{\mathcal{L}}(\theta) = \mathcal{L}_N(\bar{\theta}) + \langle \nabla \mathcal{L}_N(\bar{\theta}), \theta - \bar{\theta} \rangle + \sum_{j=2}^{\infty} \frac{1}{j!} \nabla^j \mathcal{L}_1(\bar{\theta})(\theta - \bar{\theta})^{\otimes j}.$$

Si noti come l'unica differenza con $\mathcal{L}_N(\theta)$ è che la somma infinita delle derivate viene calcolata in locale, sulla base dei dati presenti nella prima macchina. Utilizzando l'espansione di Taylor di $\mathcal{L}_1(\theta)$ intorno a $\bar{\theta}$ per sostituire la somma infinita, si ottiene

$$\tilde{\mathcal{L}}(\theta) = \mathcal{L}_N(\bar{\theta}) + \langle \nabla \mathcal{L}_N(\bar{\theta}), \theta - \bar{\theta} \rangle + \mathcal{L}_1(\theta) - \mathcal{L}_1(\bar{\theta}) - \langle \nabla \mathcal{L}_1(\bar{\theta}), \theta - \bar{\theta} \rangle$$

A questo punto omettendo le costanti additive si ottiene la forma finale della CLS:

$$\tilde{\mathcal{L}}(\theta) = \mathcal{L}_1(\theta) - \langle \theta, \nabla \mathcal{L}_1(\bar{\theta}) - \nabla \mathcal{L}_N(\bar{\theta}) \rangle \tag{2.64}$$

Dopo averla costruita, si può usare $\tilde{\mathcal{L}}$ per sostituire la funzione di perdita globale in

varie procedure statistiche. Nello specifico, può essere utilizzata come sostituto della verosimiglianza in procedure frequentiste, ottenendo lo stimatore $\tilde{\theta} = \text{argmin}_\theta \tilde{\mathcal{L}}(\theta)$. In situazioni ad elevata dimensionalità, per ottenere stime attendibili spesso si impone al parametro di interesse una struttura a bassa dimensionalità (come la sparsità del vettore di parametri θ^*), utilizzando funzioni di perdita regolarizzate. Ad esempio, utilizzando una regolarizzazione in norma ℓ_1 si ottiene $\tilde{\theta} = \text{argmin}_\theta\{\tilde{\mathcal{L}}_N(\theta) + \lambda||\theta||_1\}$. La verosimiglianza surrogata può essere utilizzata anche con un approccio bayesiano: in questo caso il parametro θ è una variabile casuale con distribuzione a priori $\pi(\theta)$, e per la posteriori si ottiene $\tilde{\pi}_N(\theta|\mathbf{Z}_1^N) \propto \exp\{-N\tilde{\mathcal{L}}(\theta)\}\pi(\theta)$.

Nell'articolo di riferimento sono riportati risultati teorici e di convergenza a supporto della metodologia, che quantificano la distanza da $\mathcal{L}$ e l'importanza della stima iniziale $\bar{\theta}$. Viene anche riportato un algoritmo iterativo per la stima distribuita, che viene ulteriormente studiato e sviluppato in Fan et al. (2021).

Capitolo 3

Applicazioni con dati reali e simulati

In questo capitolo conclusivo si presentano diverse analisi, dove si utilizzano le infrastrutture di *cloud computing* e le tecniche coperte nei capitoli precedenti per affrontare dei problemi di regressione. Inizialmente si definiscono degli scenari di simulazione per verificare il corretto funzionamento delle tecniche implementate e il guadagno in efficienza associato allo sviluppo in ambiente *cloud*, successivamente ci si focalizza su due insiemi di dati specifici per sviluppare delle analisi più complete. Il primo dei due insiemi di dati considerati riguarda gli annunci di affitto sulla piattaforma Airbnb, mentre il secondo consiste in dati simulati relativi alla rilevazione di resti di mura antiche in campioni di terreno. Le analisi vengono svolte sfruttando le risorse di *cloud computing* messe a disposizione dalla piattaforma AWS: per svolgere le operazioni di analisi si utilizza quindi un *cluster* virtuale delocalizzato. Il *software* utilizzato per sfruttare le capacità del *cluster* è PySpark, l'interfaccia per Apache Spark scritta in Python; i modelli impiegati sono il modello lineare e il modello additivo, stimati utilizzando un'implementazione in stile MapReduce. In aggiunta alla stima puntuale, si costruiscono intervalli di previsione per tutte le unità a disposizione utilizzando l'approccio *conformal inference*.

All'inizio del capitolo vengono presentate le risorse necessarie nell'ambito dell'esecuzione dell'analisi in un ambiente *cloud* e la configurazione del *cluster* utilizzato. Successivamente si entra nel dettaglio dell'implementazione in PySpark dei modelli e delle procedure di inferenza. A questo punto si presentano dei risultati relativi a dati simulati per verificare il comportamento degli stimatori e fare un confronto tra esecuzione in locale e in ambiente *cloud*. Per concludere si presentano le due analisi: per ogni *dataset* viene esposto il problema e discussa l'impostazione generale dell'analisi, successivamente si confrontano i risultati delle stime dei due modelli, sulla base dell'errore di previsione e degli intervalli predittivi ottenuti.

3.1 Strutture di calcolo utilizzate

L'analisi viene effettuata sfruttando il *cloud computing*: come *provider* si sceglie Amazon, che mette a disposizione servizi *cloud* tramite la piattaforma AWS. I tipi di servizio richiesto sono tre: la capacità di memoria fissa per immagazzinare i dati necessari all'analisi, delle macchine virtuali da utilizzare come nodi e la possibilità di gestirle entro un *cluster*.

Per la capacità di memoria fissa si utilizza Amazon *Simple Storage Service* (anche Amazon S3), un servizio di conservazione di oggetti che salva i dati come oggetti dentro a dei *buckets*. Qui come oggetto si intende un *file* e qualsiasi meta-dato che lo descriva, mentre un *bucket* è un contenitore per gli oggetti. Per salvare i propri dati su Amazon S3 bisogna creare un *bucket* specificando un nome e una regione, dove verrà fisicamente riservato dello spazio su disco fisso. Successivamente bisogna caricare i dati come un oggetto, anch'esso con un nome unico, nel *bucket* creato. La combinazione di nome del *bucket* e nome dell'oggetto (in alcuni casi anche la versione dell'oggetto, in quanto volendo si possono mantenere diverse versioni dello stesso oggetto) identificano i dati in maniera univoca.

I nodi di calcolo che andranno a comporre il *cluster* su cui lavorare vengono garantiti dal servizio Amazon *Elastic Compute Cloud* (anche Amazon EC2), che fornisce agli utenti accesso agli strumenti di calcolo fondamentali nell'ambito dei servizi *cloud* di Amazon. Oltre alla fornitura degli ambienti elementari di calcolo virtuale, detti istanze, il servizio si occupa di una serie di funzionalità accessorie, come l'installazione del sistema operativo desiderato e di *software* aggiuntivi sulle istanze create o la possibilità di accedere a diverse configurazioni di CPU, memoria volatile e fissa.

Per creare e gestire un *cluster* formato da istanze Amazon EC2, si utilizza Amazon *Elastic MapReduce* (anche Amazon EMR), una piattaforma di gestione di *cluster* che semplifica l'utilizzo di *framework* per *Big Data* (come Apache Hadoop o Apache Spark) su AWS per processare e analizzare grandi moli di dati. Utilizzando queste infrastrutture è anche possibile trasformare e muovere i dati dentro e fuori dai servizi di *storage* su AWS, come i *bucket* di Amazon S3. La componente centrale di Amazon EMR è il *cluster*, cioè una collezione di istanze di tipo Amazon EC2. Ogni istanza nel *cluster* viene chiamata nodo, con ogni nodo associato ad un ruolo specifico. Amazon EMR installa anche le diverse componenti *software* su ciascuno dei nodi, per permettere la gestione di sistemi come Hadoop o Spark. I nodi di un *cluster* creato con Amazon EMR possono essere di diversi tipi. Il nodo *master* gestisce il *cluster* utilizzando componenti di tipo *software* che permettono di coordinare la distribuzione dei dati e dei compiti fra

gli altri nodi per l'analisi; inoltre è responsabile del monitoraggio dello stato dei vari compiti e della salute generale del *cluster*. Gli altri nodi vengono detti di tipo *core*, e utilizzano le loro componenti *software* per eseguire i diversi compiti e salvare i dati nel *file system* distribuito associato al cluster.

L'architettura di un *cluster* EMR si compone di diversi strati, detti anche *layers*, ognuno dei quali fornisce certe possibilità e funzionalità al *cluster*. Il primo *layer* è rappresentato dal *file system* utilizzato nel *cluster*: solitamente si utilizzano soluzioni distribuite come HDFS o EMRFS (EMR *file system*, un'estensione di Hadoop che permette di accedere direttamente ai *bucket* Amazon S3 come fossero parte di un *file system* distribuito), ma si possono anche utilizzare i *file systems* locali dei diversi nodi. Il secondo strato è quello di gestione delle risorse, responsabile per la gestione delle risorse del *cluster* e per la programmazione dei diversi compiti necessari all'esecuzione dell'applicazione. Di *default* Amazon EMR usa YARN (*Yet Another Resource Negotiator*), una componente introdotta in Apache Hadoop 2.0 per gestire le risorse di un *cluster* fra sistemi di analisi su larga scala multipli. L'ultimo strato è il sistema di analisi dati, il motore utilizzato per processare ed analizzare i dati. La scelta del *framework* determina i linguaggi e le interfacce disponibili a livello di applicazione. Le principali alternative disponibili sono Hadoop MapReduce e Apache Spark.

Amazon EMR mette a disposizione diverse possibilità per mandare in esecuzione dei compiti ad un *cluster*. Per l'applicazione in questione ci si appoggia su EMR Studio, un ambiente di sviluppo integrato (IDE) che semplifica lo sviluppo e la visualizzazione di applicazioni di *data science*, fornendo agli utenti dei Jupyter *notebook* che vengono eseguiti sui *cluster* Amazon EMR.

Il materiale presentato in questa sezione sui servizi disponibili su AWS è tratto dalla documentazione ufficiale della piattaforma, consultabile all'indirizzo `https://docs.aws.amazon.com/index`, alla quale si rimanda per ulteriori approfondimenti.

Per l'esecuzione delle applicazioni su dati reali su cui ci si focalizza in questo lavoro, come prima cosa vengono creati dei *bucket* Amazon S3, dove si caricano i dati di interesse. Successivamente tramite Amazon EMR viene creato un *cluster* con tre istanze Amazon EC2 come nodi. I nodi sono istanze di tipo `m5.xlarge`: M5 sono le istanze EC2 di quinta generazione basate su processori Intel® Xeon® Platinum 8175M, tra queste le istanze `m5.xlarge` hanno quattro processori virtuali e 16 GiB (*gibibyte*, ossia 2^{30} *bytes*) di memoria RAM. Su tutte le istanze è installato un sistema operativo Amazon Linux, versione 2.0.20220426.0. Il *cluster* utilizza HDFS come *distributed file system* (versione 2.10.1) e Apache Spark (di specifico PySpark, l'interfaccia per Apache Spark in Python, versione 2.4.8) come motore di analisi. Prima di cominciare con le

analisi, vengono installate su tutti i nodi le librerie necessarie: `scipy` (Virtanen et al. (2020)) e `numpy` (Harris et al. (2020)) per il calcolo algebrico, `pandas` (McKinney et al. (2010)) per definire le funzioni vettorizzate che operano sulle colonne di un *DataFrame* PySpark.

Utilizzando EMR Studio, per ogni insieme di dati viene creato un Jupyter *notebook*, che viene eseguito sul *cluster*, e lo si utilizza per svolgere le operazioni di analisi. Alla fine delle operazioni di analisi i risultati vengono salvati ciascuno nel rispettivo *bucket* S3 precedentemente creato. Terminata questa operazione si scaricano i risultati, successivamente si può terminare l'utilizzo dei servizi di AWS.

3.2 Algoritmi utilizzati

In questa sezione si esaminano nel dettaglio le diverse tecniche utilizzate per l'analisi, da un punto di vista algoritmico e di implementazione.

I modelli scelti per l'analisi sono il modello lineare e il modello additivo. Le funzioni non parametriche lisce del modello additivo vengono rappresentate mediante un'espansione in basi, in modo da utilizzare lo stesso approccio risolutivo al problema dei minimi quadrati per entrambi i modelli.

La stima del modello lineare si basa su una variante dell'algoritmo 3 (LS-CHOL-ROWWISE), implementata secondo lo stile MapReduce. Si ipotizzi di aver già ottenuto la matrice del modello $\mathbf{X}$ (con la colonna costante per l'intercetta e la codifica tramite variabili *dummy* per le variabili quantitative) e di aver salvato i dati in un DataFrame di PySpark (un'estensione del classico RDD di Spark). Per ogni riga i, si ha quindi una colonna con il vettore x_i e una colonna con il corrispondente valore di y_i (la risposta). A questo punto si scrive la funzione Map, che prende in input una riga, separa x_i e y_i e restituisce la coppia $(x_i x_i^T, x_i y_i)$. Questa funzione Map non ha una chiave, una possibilità introdotta da Spark, a indicare che la funzione Reduce verrà applicata a tutte le righe (come se avessero tutte la stessa chiave). La funzione Reduce calcola semplicemente la somma degli elementi di ogni coppia, per ottenere $\mathbf{X}^T\mathbf{X}$ e $\mathbf{X}^T\mathbf{y}$. Una volta ottenute le quantità di interesse, la stima dei coefficienti viene ottenuta in due passaggi utilizzando la fattorizzazione di Cholesky, come negli algoritmi 1 e 3. Le previsioni del modello vengono ottenute tramite una seconda funzione Map, che per ogni riga calcola $\hat{y}_i = x_i^T \hat{\beta}$. Le funzioni per la stima del modello lineare e la previsione, scritte in PySpark, sono riportate in Appendice A.1.

Per la stima del modello additivo si ottiene una rappresentazione in basi mediante *splines* cubiche di regressione e successivamente si procede come per il modello lineare.

Come nodi si prendono dieci valori equispaziati nell'intervallo aperto $(0, 1)$. Il parametro di lisciamento λ, che controlla la rugosità delle funzioni stimate, viene selezionato con metodo euristico, effettuando alcune prove su campioni ridotti dell'insieme di dati di riferimento. Nello specifico, in questo caso la funzione Map per la stima ottiene per ogni riga l'espansione in basi per il vettore di covariate x_i e successivamente calcola la coppia $(x_i x_i^T, x_i y_i)$, mentre la funzione Reduce è identica a quella descritta in precedenza e permette di ottenere $\mathbf{X}^T\mathbf{X}$ e $\mathbf{X}^T\mathbf{y}$. La matrice di penalità $\mathbf{S}$ viene definita sulla base dei nodi e del valore di λ selezionato. L'espansione in basi per le covariate e la matrice di penalità $\mathbf{S}$ vengono calcolate utilizzando l'approccio introdotto in Gu & Gu (2013). Dopo aver ottenuto queste quantità, il sistema lineare $(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{S})\beta = \mathbf{X}^T\mathbf{y}$ viene risolto mediante scomposizione LU, per le ragioni delineate in sezione 2.5. La funzione Map utilizzata per la previsione, similmente a quella impiegata per la stima, per ogni riga ottiene prima l'espansione in basi e successivamente calcola $\hat{y}_i = x_i^T\hat{\beta}$. Le funzioni implementate in PySpark per la stima del modello additivo, la creazione della matrice di penalità e la previsione sono riportate in Appendice A.2.

Sulla base dei risultati dei due modelli, si costruiscono anche degli intervalli di previsione per tutte le unità a disposizione. All'inizio dell'analisi, l'insieme di dati viene diviso in due parti uguali. Successivamente si stimano i coefficienti del modello (lineare o additivo) sulla base dei dati contenuti nel primo *split*, e si ottengono le previsioni per le unità del secondo *split*. Questa procedura viene ripetuta in maniera speculare partendo dal secondo *split*, arrivando ad avere per tutti i dati di ognuna delle due partizioni una previsione basata sul modello relativo all'altra partizione. Partendo da queste previsioni è possibile applicare la procedura *rank-one-out split conformal* e ottenere gli intervalli di previsione in corrispondenza di tutte le unità dell'insieme di partenza. L'implementazione dell'algoritmo 7 è strutturata come di seguito. Vengono definite due `pandas_udf`, funzioni vettorizzate che operano sulle colonne di un DataFrame PySpark: la prima calcola l'errore di previsione in valore assoluto sulla base delle colonne contenenti il vero valore della risposta e le previsioni, la seconda parte dal vettore degli errori assoluti ordinati (per l'ordinamento ci si appoggia al metodo interno di PySpark, basato su una variante dell'algoritmo BucketSort) e restituisce per ogni unità il quantile empirico *rank-one-out* corrispondente. Avendo definito queste due funzioni, la funzione che esegue l'algoritmo *rank-one-out* split conformal viene definita come una trasformazione che parte da un DataFrame contenente valori reali e previsioni, calcola l'errore assoluto, individua il quantile empirico di riferimento e infine definisce i due estremi dell'intervallo di previsione. Le funzioni PySpark descritte vengono riportate in Appendice A.3.

3.3 Prime simulazioni

In questa sezione si riportano i risultati di tre semplici scenari di simulazione: i primi due sono sviluppati per verificare il corretto funzionamento degli stimatori implementati, il terzo punta a fare un confronto tra i tempi di esecuzione dell'algoritmo di stima del modello lineare in locale e in ambiente *cloud*.

Si definisce ora il primo scenario di simulazione. Si simula dal modello lineare $y_i = \beta_0 + \sum_{j=1}^{p} x_{ij}\beta_j + \epsilon_i$, $i = 1, \ldots, n$, con $n = 50\,000$ e $p = 6$. La matrice del modello viene creata con tutte le entrate estratte da una distribuzione uniforme tra 0 e 1 e fissata. I veri coefficienti del modello vengono estratti da una distribuzione normale a media nulla e con deviazione standard pari a 5. L'errore casuale ϵ segue una distribuzione normale a media nulla, con varianza impostata in modo tale da avere un *sound-to-noise ratio* pari a 5. Al pari di come si lavorerà con i dati reali, si divide l'insieme di stima in due partizioni uguali, ottenendo le stime per i coefficienti del modello per entrambe le partizioni. A questo punto si ottengono le previsioni in corrispondenza di tutti i dati a disposizione (per ottenere le previsioni per le unità della prima partizione si utilizza il modello stimato sulla seconda partizione e viceversa), sulla base delle quali applicare le procedure di *split conformal inference*.

Dopo aver stimato il modello nelle due partizioni, inizialmente si verifica che i coefficienti stimati siano in linea con quelli reali. Per fare questo si ottiene la media delle stime nelle due partizioni, a seguire si calcola la media delle differenze al quadrato tra valore vero e stimato per ogni coefficiente, ottenendo un valore pari a 0.00013. Successivamente, si valutano la bontà delle previsioni e degli intervalli costruiti calcolando la radice dell'errore quadratico medio (RMSE, *root mean squared error*), la copertura empirica e l'ampiezza media degli intervalli di previsione. Le metriche di interesse vengono riportate in Tabella 3.3.

Modello	RMSE	Copertura	Ampiezza
Lineare	4.088984	0.950009	8.869898

Tabella 3.1: Primo studio di simulazione, metriche di interesse.

Si noti come, in virtù dell'elevata numerosità, la copertura empirica abbia raggiunto la convergenza al valore nominale (si ricordino le identità (2.61) e (2.62) associate alle procedure *conformal*). Si evidenzia anche come per costruzione l'ampiezza degli intervalli costruiti sia collegata all'errore di stima: essendo definiti sulla base di quantili empirici degli errori di previsione, si ha che ad una stima migliore (e quindi ad un RMSE più basso) corrispondono intervalli più stretti.

Per il modello additivo si definisce uno scenario di simulazione simile, dove però questa volta il vero modello generatore dei dati è additivo e non lineare. Il modello generatore è $y_i = \sum_{j=1}^{p} f_j(x_{ij}) + \epsilon_i$, $i = 1, \ldots, n$, con $n = 50\,000$ e $p = 4$. Anche in questo caso il termine di errore ϵ_i viene estratto da una distribuzione normale a media nulla e con varianza tale da avere un *sound-to-noise ratio* pari a 5. Le funzioni lisce utilizzate sono le seguenti:

$$
\begin{aligned}
f_1(x) &= -2 \exp\left(-\frac{(x-0.5)^2}{0.2}\right), \\
f_2(x) &= -10(x-0.5)^3, \\
f_3(x) &= -2x^3 + 4x^5, \\
f_4(x) &= \sin(10x).
\end{aligned}
$$

Le funzioni definite, dopo aver normalizzato i valori nell'intervallo $(0, 1)$, vengono visualizzate in Figura 3.2.

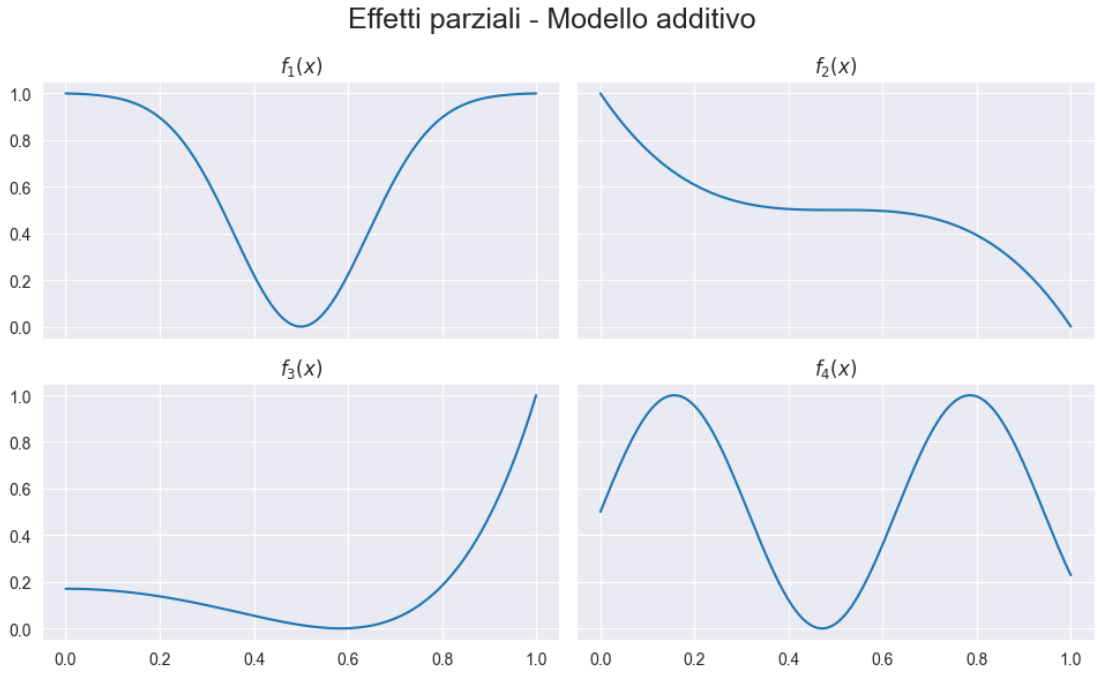

FIGURA 3.1: Secondo studio di simulazione, effetti parziali nel vero modello generatore dei dati.

Il modello additivo in questo caso utilizza *splines* cubiche di regressione penalizzate. In linea con quanto fatto precedentemente, per valutare l'aderenza degli effetti stimati a quelli reali si prende la media dei coefficienti stimati sulle due partizioni e si confrontano le stime lisce risultanti con gli effetti reali. I risultati sono riportati i figura 3.2, dove

si può notare come le forme funzionali stimate siano molto simili a quelle del modello generatore dei dati.

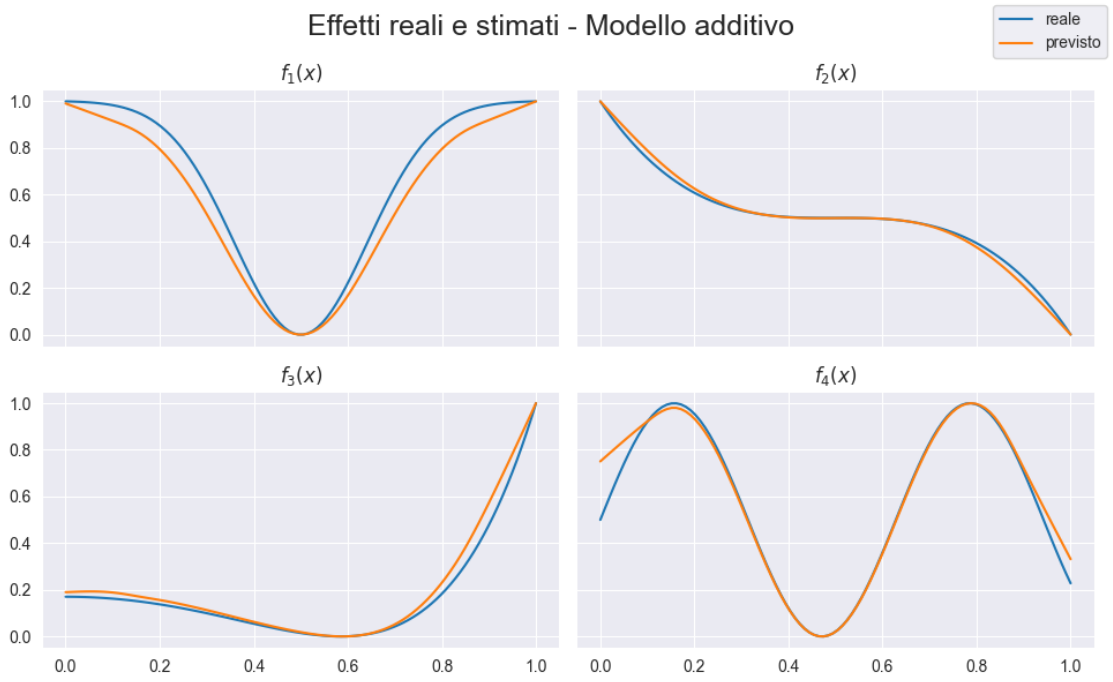

FIGURA 3.2: Secondo studio di simulazione, effetti parziali reali e stimati.

In Tabella 3.3 di seguito si possono vedere le metriche di valutazione delle previsioni ottenute sulle due partizioni. Anche in questo caso la copertura empirica degli intervalli di previsione costruiti converge a quella nominale.

Modello	RMSE	Copertura	Ampiezza
Additivo	1.643057	0.950009	3.568326

TABELLA 3.2: Secondo studio di simulazione, metriche di interesse.

Dopo aver verificato il corretto funzionamento delle procedure di stima e di inferenza scelte, si imposta un ultimo scenario di simulazione per confrontare i tempi di esecuzione necessari per la stima di un modello lineare in locale e in ambiente *cloud*. Lo scenario di simulazione è simile al primo introdotto in questa sezione: il modello generatore è $y_i = \beta_0 + \sum_{j=1}^{p} x_{ij}\beta_j + \epsilon_i$, $i = 1, \ldots, n$, $p = 6$. I coefficienti vengono estratti da una distribuzione normale a media nulla e deviazione standard pari a 5, mentre l'errore casuale ϵ_i segue una distribuzione normale con varianza tale da avere un *sound-to-noise ratio* pari a 5. Nelle simulazioni si fa variare il numero di unità n da 100 000 a 20 000 000. Come descritto in sezione 3.1, il *cluster* virtuale utilizzato si compone di tre nodi (di cui uno svolge il ruolo del *master*). I nodi sono istanze di tipo `m5.xlarge`: sono basate su processori Intel® Xeon® Platinum 8175M, hanno quattro processori virtuali e 16

GiB di memoria RAM. La macchina locale utilizzata per il confronto ha un processore Intel® Core™ i5-8250U CPU, otto processori virtuali e 8 Gb di memoria RAM.

In Tabella 3.3 sono riportati i tempi di esecuzione per la stima del modello lineare in locale e nel *cloud*.

Sistema	10^5	10^6	$5 \cdot 10^6$	$10 \cdot 10^6$	$20 \cdot 10^6$
Locale	11s	56s	4m 16s	8m 13s	14m 38s
Cloud	6s	30s	2m 47s	4m 53s	9m 12s

TABELLA 3.3: Terzo studio di simulazione, confronto tra tempi di esecuzione per la stima del modello lineare con approccio MapReduce.

Dai risultati si evidenzia come effettuando l'analisi in un ambiente *cloud* si ottenga un guadagno in termini di efficienza che è approssimativamente lineare con il numero di nodi *workers* utilizzati: in questo caso con due *workers* i tempi di esecuzione in *cloud* sono poco più della metà dei corrispettivi tempi in locale. Si suppone che il tempo aggiuntivo sia da attribuire al carico del lavoro del *master* che mette in comunicazione i vari nodi. Si noti che, come era logico attendersi visto la struttura MapReduce dell'algoritmo di stima utilizzato, il tempo di esecuzione aumenta linearmente con il numero di righe. Si ha quindi che, utilizzando un *cluster* semplice con due soli nodi, rispetto ad un approccio in locale si riesce a processare il doppio dei dati nello stesso tempo, o si dimezza il tempo di esecuzione per la stessa quantità di dati. Questo vantaggio acquisisce particolare importanza nel momento in cui si desideri di scalare la propria architettura virtuale *orizzontalmente*: utilizzando il doppio dei nodi rispetto a quelli utilizzati, il tempo di esecuzione dimezzerebbe ancora, anche se aumenterebbe il carico di lavoro del *master* per mettere in comunicazione tutte le componenti. Inoltre, nel caso in cui il numero di righe fosse stato tale da rendere impossibile il caricamento in memoria locale dei dati, l'approccio in ambiente *cloud* sarebbe stato l'unica soluzione per l'analisi.

Per avere un riferimento sul quale valutare generalmente le prestazioni dell'approccio di stima utilizzato, si riportano in Tabella 3.3 i tempi di esecuzione in locale e nel *cloud* della funzione `LinearRegression` della libreria di *machine learning* di PySpark, nello stesso schema di simulazione.

Sistema	10^5	10^6	$5 \cdot 10^6$	$10 \cdot 10^6$	$20 \cdot 10^6$
Locale	6s	21s	1m 30s	2m 48s	6m 50s
Cloud	9s	11s	52s	1m 37s	3m 01s

TABELLA 3.4: Terzo studio di simulazione, confronto tra tempi di esecuzione per la stima del modello lineare con la funzione interna di PySpark.

Utilizzando questa funzione la stima del modello avviene mediante un algoritmo di discesa del gradiente stocastica, ottimizzato per l'esecuzione distribuita. Si noti come i

tempi di esecuzione in locale siano minori di quelli registrati nel *cloud* con l'approccio MapReduce.

3.4 Analisi dei dati di Airbnb

Airbnb è un portale online statunitense che si occupa di intermediazione tra persone in cerca di un alloggio o una camera per periodi brevi e persone (generalmente privati) che dispongono di spazi da affittare. Il sito (`https://www.airbnb.com`) è stato lanciato nel 2007 da Brian Chesky, Joe Gebbia e Nathan Blecharczyk e conta oggi più di sei milioni di annunci (detti *listings*) da parte di più di quattro milioni di *host*, in oltre 200 paesi in tutto il mondo.

La compagnia dichiara di promuovere un'economia di condivisione, ma nel tempo ha ricevuto diverse critiche per l'impatto sul mercato immobiliare residenziale: l'accusa è che la maggior parte degli annunci riguardino intere case in zone residenziali, che vengono affittate a prezzi inflazionati per tutto l'anno andando ad influenzare negativamente il mercato immobiliare limitrofo. In questo contesto Murray Cox, un attivista di origini australiane, lancia nel 2016 il sito Inside Airbnb (`http://insideairbnb.com`), una piattaforma che riporta e visualizza i dati relativi a tutti i *listings* del sito, con l'obiettivo di aiutare le città e le comunità locali a comprendere e contrastare il fenomeno della diffusione di Airbnb nei quartieri residenziali. Il sito scarica e rende disponibili i dati prendendoli direttamente dalla piattaforma Airbnb.

Per l'applicazione con dati reali considerata in questo lavoro, si scaricano tutti i dati relativi al mese di giugno disponibili sul sito Inside Airbnb. Nel sito si trovano dati relativi sia a singole città turistiche che a interi stati, per un totale di 115 insiemi di dati distinti. I dati vengono scaricati automaticamente mediante una procedura di *web scraping*, realizzata per mezzo della libreria `BeautifulSoup` (`https://www.crummy.com/software/BeautifulSoup/bs4/doc/`) di Python. Per ogni annuncio in una città o paese, si hanno a disposizione una serie di variabili che caratterizzano la proprietà in questione insieme al prezzo giornaliero nella valuta locale. L'obiettivo dell'applicazione con dati reali presa in considerazione è quello di unire tutti i dati in un grande *dataset*, dove utilizzare un sottoinsieme delle variabili disponibili per prevedere il prezzo della proprietà.

Per unire tutti i dati, bisogna tenere in considerazione che i prezzi sono riportati nella valuta corrente in ognuno dei *dataset* di partenza. Per ovviare a questo problema si decide di costruire un punteggio per ogni *listing*, prendendo il logaritmo del prezzo e standardizzandolo. In questa operazione si utilizza la mediana come misura di centralità

per far sì che la procedura risulti meno influenzata da potenziali *outliers*. Operando in questo modo i punteggi ottenuti risultano comparabili tra diverse città e paesi e distribuiti approssimativamente nel dominio di una distribuzione normale standard; inoltre, vengono assorbite caratteristiche specifiche dei mercati immobiliari locali, mantenendo solo l'informazione rispetto al valore relativo della proprietà rispetto alle altre nella zona.

L'insieme di dati ottenuto unendo tutti i *datasets* a disposizione in seguito a questa fase di elaborazione preliminare risulta composto da 1 270 990 rilevazioni di 8 variabili, con una dimensione totale pari a 213 Mb. Va fatto notare a questo punto che le applicazioni moderne di analisi dati, dove si sfruttano le possibilità del *cloud computing* grazie a programmi come Spark, sono realisticamente pensate per moli di dati molto maggiori, spesso anche nell'ordine dei *terabytes*. Un insieme di dati come quello preso in considerazione potrebbe essere analizzato senza particolari problemi anche con normale personal computer; l'implementazione mediante strutture per il calcolo parallelo e l'analisi dati di larga scala viene qui presentata solo a scopo esemplificativo.

Le variabili a disposizione per l'analisi sono elecante di seguito:

- *room type*, il tipo di stanza, variabile qualitativa con quattro livelli (stanza condivisa, stanza privata, intera casa/appartamento, altro);
- *minimum nights*, permanenza minima, il numero minimo di notti di permanenza per effettuare la prenotazione;
- *number of reviews*, il numero totale di recensioni per il *listing*;
- *reviews per month*, il numero medio mensile di recensioni per il *listing*;
- *calculated host listings count*, il numero di annunci di affitto emessi da quel determinato *host* in quella regione o città;
- *availability 365*, il numero di giorni non prenotati per la proprietà, guardando un anno in avanti dal momento della rilevazione;
- *number of reviews ltm*, il numero totale annuale di recensioni per il *listing*;
- *price score*, il punteggio associato al prezzo e calcolato come descritto in precedenza.

L'analisi impostata ha due obiettivi: quello di prevedere il punteggio associato al prezzo sulla base delle restanti variabili e quello di fornire intervalli di previsione con copertura finita desiderata per tutti i dati a disposizione. Per le previsioni puntuali

vengono adattati un modello lineare e un modello additivo, mentre gli intervalli di previsione per tutti i dati vengono calcolati secondo il metodo *rank-one-out split conformal* sulla base delle previsioni di entrambi i modelli.

In Figura 3.3 sono riportati i coefficienti del modello lineare. Per la visualizzazione,

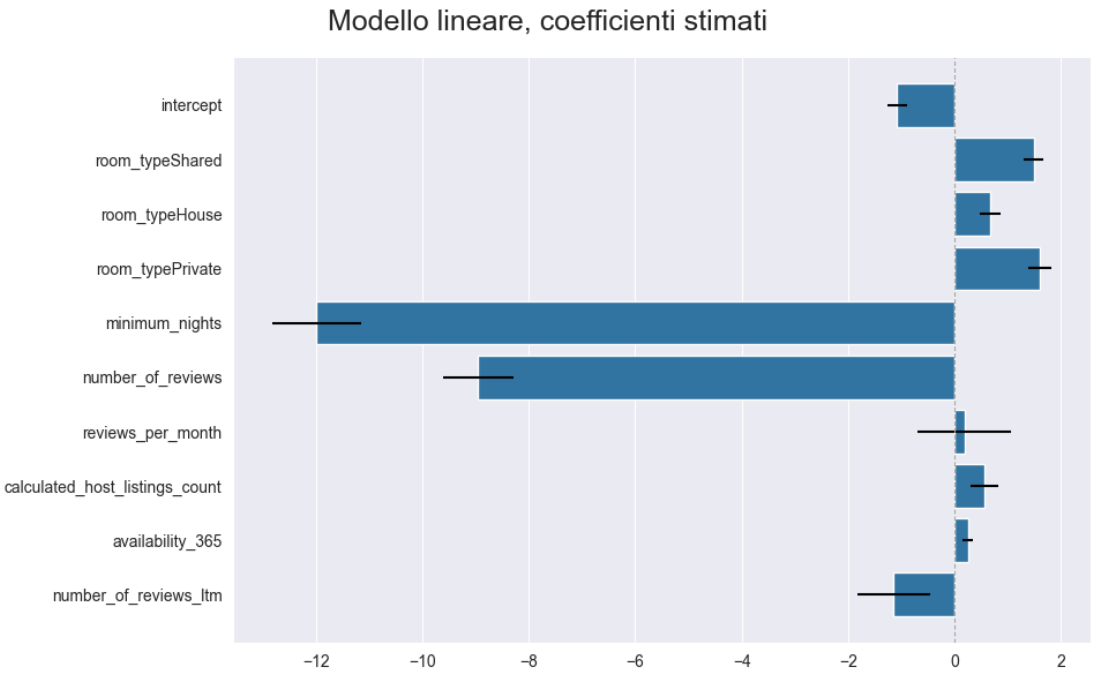

FIGURA 3.3: Dati di Airbnb, coefficienti stimati.

si prende la media dei valori sulle due partizioni per aggregare le stime ottenute. Analizzando le stime, i coefficienti più importanti sono quelli relativi a *minimum nights* e *number of reviews*, negativi e con valore assoluto molto maggiore rispetto agli altri. Si potrebbe ipotizzare che gli annunci con associato un numero minimo di notti di pernottamento elevato siano relativi a soluzioni di affitto di media o lunga durata, con prezzi minori rispetto a quelli del mercato turistico. Rispetto al valore assunto dal coefficiente associato al numero di recensioni, va fatto notare che nell'insieme di dati non si fanno distinzioni fra buone e cattive recensioni, mentre idealmente questa è un'informazione da tenere in conto nel modello; non è facile in questo caso fare ipotesi sulla natura del legame tra il numero di recensioni e il punteggio associato al prezzo. Guardando all'effetto del tipo di abitazione, in maniera sorprendente risulta che gli affitti di camere (private o in stanza condivisa) siano associati ad effetti positivi maggiori a quelli associati ad interi appartamenti, a parità di altre condizioni. Si ricorda che questo non vuole per forza dire che a livello marginale il modello preveda prezzi minori per affitti di stanze rispetto ad interi appartamenti, poiché questo dipende anche dalla configurazione delle altre variabili esplicative a disposizione. Si valuta che questo fenomeno

potrebbe essere causato dall'operazione di aggregazione di tutti i piccoli insiemi di dati di partenza sugli annunci relativi ai singoli paesi o città, che potrebbe aver mascherato alcune caratteristiche dei dati.

Gli effetti parziali del modello additivo stimato sono visualizzati in Figura 3.4. Nella figura sono anche visualizzati gli effetti parziali associati alle stime ottenute con il modello lineare con le relative bande di variabilità. I risultati ottenuti circa gli effetti

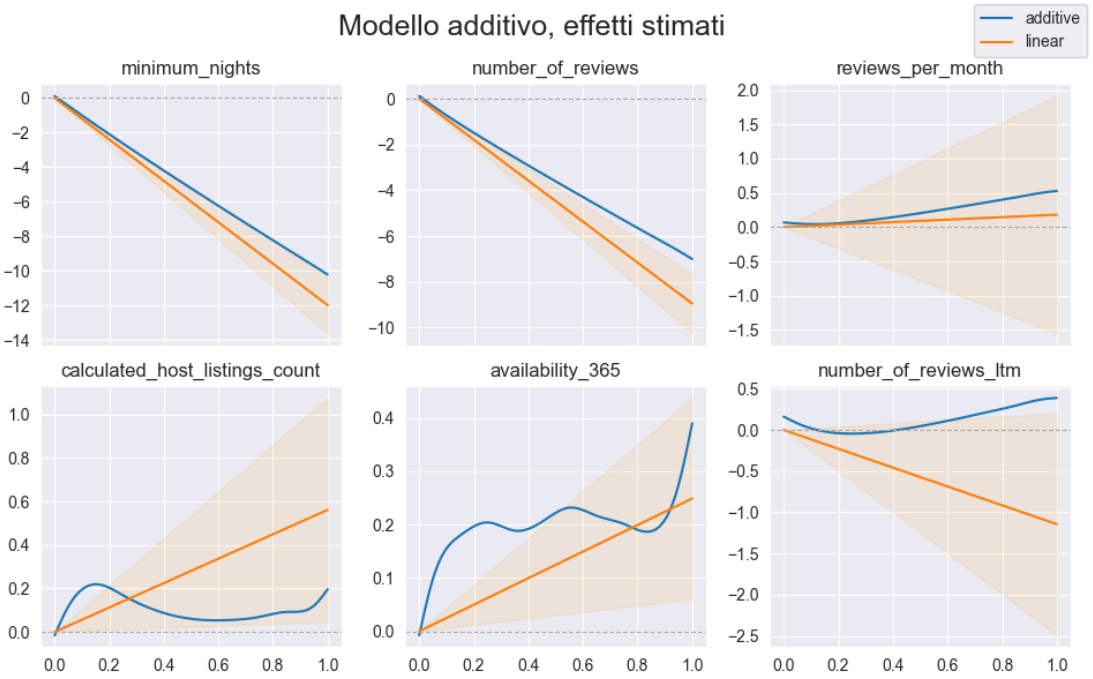

FIGURA 3.4: Dati di Airbnb, effetti parziali stimati.

parziali stimati sono tendenzialmente in linea con le indicazioni del modello lineare. In aggiunta a quanto emerso, si può notare come per le variabili *calculated host listings count* e *availability 365* gli effetti stimati siano positivi ma non lineari. Riguardo al primo dei due coefficienti, sembra che un *host* risulti più attraente se ha più di un annuncio, ma l'effetto si capovolge nel momento in cui ha troppi annunci a suo nome. Può darsi che i clienti di questa piattaforma preferiscono che l'*host* sia professionale, quindi è meglio se non ha un solo annuncio, ma dia sempre l'impressione di essere un privato che affitta i propri spazi piuttosto che un imprenditore del settore immobiliare.

I due modelli vengono confrontati sulla base dell'errore quadratico medio delle previsioni, la copertura empirica degli intervalli predittivi e la loro ampiezza media. Le metriche per i due modelli vengono riportare in Tabella 2.

Si può vedere come in questo caso la flessibilità introdotta dal modello additivo non porti ad una prestazione migliore a livello predittivo, ne deriva che le varie indicazioni aggiuntive emerse sono da prendere in considerazione con prudenza. Riguardo agli

Modello	RMSE	Copertura	Ampiezza
Lineare	2.663427	0.950019	5.117859
Additivo	3.187767	0.950011	6.210262

Tabella 3.5: Dati di Airbnb, metriche di errore.

intervalli di previsione *split conformal*, come per i risultati sulle simulazioni riportati in precedenza si ha che ad una prestazione migliore corrisponde generalmente un intervallo di previsione più stretto, mentre in virtù dell'elevata numerosità la copertura empirica di entrambi gli intervalli converge per costruzione al valore nominale.

Come commento finale, si fa notare che la strada intrapresa per l'analisi di questi dati non può definirsi ottimale, ma anzi è stata vincolata dalla volontà di utilizzare i dati in questo particolare contesto applicativo, ossia quello di un'analisi di "larga scala". L'approccio ideale probabilmente sarebbe stato quello di analizzare ogni insieme di dati distintamente, per la possibilità di tenere separati paesi e città e di indagare le varie caratteristiche spaziali dei luoghi considerati (quartieri, conformazione geografica...), oppure una modellazione di tipo gerarchico, che avrebbe consentito di fare *pooling* dell'informazione mantenendo comunque le differenze tra gli insiemi di dati. Questo avrebbe anche permesso di evitare possibili problemi di mascheramento introdotti dall'operazione di standardizzazione e aggregazione.

3.5 Analisi dei dati simulati sulle rilevazioni archeologiche

I dati utilizzati in questa seconda applicazione sono quelli forniti per la MODE Data Challenge (Strong et al. (2022)), parte del *Second MODE Workshop on Differentiable Programming for Experiment Design*, tenutosi a Creta dal 12 al 16 Settembre 2022. I dati sono stati messi a disposizione dal professor Bruno Scarpa, del Dipartimento di Scienze Statistiche dell'Università degli Studi di Padova. Il contesto applicativo è quello della rilevazione di reperti archeologici, nello specifico mura antiche, in campioni di sottosuolo: viene scannerizzato un sito archeologico utilizzando la tomografia a muoni, con l'obiettivo di mappare le posizioni delle mura antiche. I muoni sono particelle che vengono naturalmente prodotte quando i raggi cosmici incontrano l'atmosfera terrestre. Queste particelle sono in grado di attraversare la materia; quando questo avviene, si registrano delle variazioni nelle loro proprietà cinematiche. La tomografia a muoni permette di produrre immagini tridimensionali di un sito di interesse, analizzando queste

variazioni nella cinematica dei muoni. La quantità di variazione dipende dalla lunghezza della radiazione del materiale attraversato dal muone, chiamata X0 e misurata in metri. A materiali densi corrispondono generalmente bassi valori di X0, a loro volta rappresentanti grandi variazioni nella cinematica del muone.

Il *dataset* simulato consiste in immagini 3D di volumi di interesse (VoI, *Volumes of Interest*) generati casualmente, divisi in voxels. I VoI sono generati per rappresentare una serie di piccole porzioni di muratura sepolte e circondate dal suolo. Per ogni voxel si ha una stima di X0, ottenuta utilizzando un algoritmo standard del settore chiamato PoCA (*Point of Closest Approach*). La previsione per X0 è un reale positivo espresso in metri. La PoCA produce immagini sfocate e imprecise quindi, alla luce del rischio di danneggiare i reperti antichi durante gli scavi del sito, si richiede di sviluppare un algoritmo dedicato che per ogni voxel riesca a classificare se è un pezzo di muro o semplice suolo.

Per questa applicazione si considera un campione di 10 000 volumi estratto dai dati dell'insieme di stima fornito per la competizione. Ciascun VoI è un cubo con dieci voxel per lato; ad ogni voxel è associata la classificazione in suolo o muro e il valore di radiazione X0 stimato. L'approccio utilizzato è quello di ricondurre i dati in un formato tabulare: un volume viene quindi "srotolato" per ottenere una riga per ogni voxel, con le variabili y, la risposta, x, la radiazione stimata, e le coordinate del voxel nel volume (tre interi compresi fra 1 e 10), ottenendo un insieme di dati con 10 milioni di righe. Per affrontare questo problema di classificazione con un approccio tabulare, l'idea è quella di prevedere la classe per un voxel sulla base della radiazione del voxel stesso e di quelli circostanti: si creano quindi sei nuove variabili, che per ogni voxel riportano il valore della radiazione dei voxel con una faccia in comune (ossia per ogni voxel si considerano spostamenti di un voxel nelle tre coordinate). Successivamente si rimuovono le righe con valori mancanti, corrispondenti ai bordi del volume di interesse, ottenendo un insieme con 5 120 000 osservazioni di otto variabili (la risposta e il valore della radiazione stimata nei sette voxel associati).

Per affrontare questo problema di classificazione binaria, si decide di utilizzare dei metodi di regressione sulla risposta codificata come numerica. Questo approccio, nonostante abbia il difetto principale di poter fornire previsioni fuori dall'intervallo $(0, 1)$, viene spesso impiegato nei problemi di classificazione binaria come alternativa ai metodi specifici per la classificazione. Uno dei vantaggi principali è quello di semplificare la stima dei modelli e il carico computazionale: ad esempio il costo della stima ai minimi quadrati per il modello lineare è una frazione del costo di un algoritmo come quello dei minimi quadrati pesati iterativi (IRLS, *iterativeley reweighted least squares*)

impiegato per la stima del modello logistico. Lavorando in questo modo le previsioni ottenute possono essere interpretate come probabilità grezze, tuttavia non è possibile applicare le tecniche della *conformal inference* per fornire intervalli di previsione sulle probabilità stimate: supponendo di classificare come eventi le previsioni superiori a 0.5, basterebbe un singolo errore per ottenere un quantile empirico pari a 1, che a sua volta determinerebbe intervalli sempre fuori da $(0, 1)$.

Nell'insieme di dati ottenuto si registra una frequenza relativa di muri sul totale di voxels pari a 0.08439. Per contrastare questo sbilanciamento, per ognuno dei due *splits* si ottiene un insieme ridotto con una proporzione uguale tra mura e suolo, sul quale ottenere le stime dei modelli. Si effettuano diverse prove in locale ed emerge che, utilizzando l'intercetta nella modellazione, i valori previsti per entrambi i modelli sono sempre fuori dall'intervallo $(0, 1)$. Al contrario, senza l'utilizzo dell'intercetta si ottengono previsioni all'interno dell'intervallo desiderato, ma le stime sono instabili e la classificazione ottenuta è pessima. Si sceglie quindi di utilizzare la prima strada e di trovare un modo per classificare sulla base delle previsioni fuori scala che emergono. Per fare questo, si prende come soglia di classificazione per un vettore di previsioni il suo quantile empirico di livello $1 - s$, dove s è la proporzione osservata di eventi. In questo modo, si forza il modello a classificare una proporzione esattamente pari ad s di voxel come mura, a prescindere dal fatto che le previsioni non fossero dentro l'intervallo $(0, 1)$.

In Figura 3.5 sono riportati i coefficienti del modello lineare. Si nota come tutti i coefficienti associati ai valori della radiazione X0 nei dintorni del voxel di interesse siano significativi e negativi. Questo è in linea con quanto ci si poteva aspettare visto il problema: si ricorda che a materiali densi sono associati bassi valori di lunghezza della radiazione, quindi ha senso che valori bassi di X0 aumentino la fiducia del modello nel classificare un voxel come muro. È interessante notare come, mentre per le coppie relative agli spostamenti in orizzontale dal voxel di riferimento gli effetti sono pressapoco identici, i coefficienti associati al valore della radiazione nei voxel esattamente sopra e sotto quello di interesse riportano una grande differenza. Cercando di interpretare logicamente questa indicazione, si potrebbe dire che nel momento in cui il modello vede che il voxel immediatamente sopra a quello di interesse è un muro, è più portato a classificare come muro anche il voxel di interesse, per il semplice fatto che non si aspetta di trovare mura che non poggiano alla base del volume.

Gli effetti parziali del modello additivo stimato sono visualizzati in Figura 3.6, insieme agli effetti associati alla stima del modello lineare. In questo caso la modellazione additiva non permette di comprendere meglio il fenomeno: la flessibilità aggiuntiva a

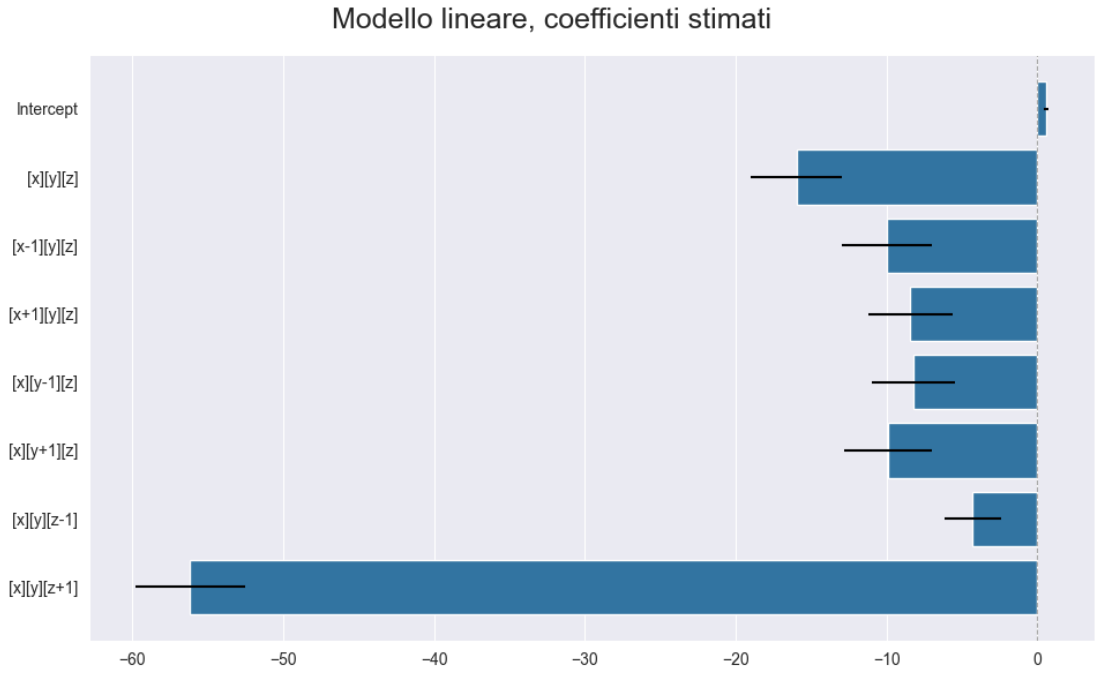

FIGURA 3.5: Dati di Airbnb, coefficienti stimati.

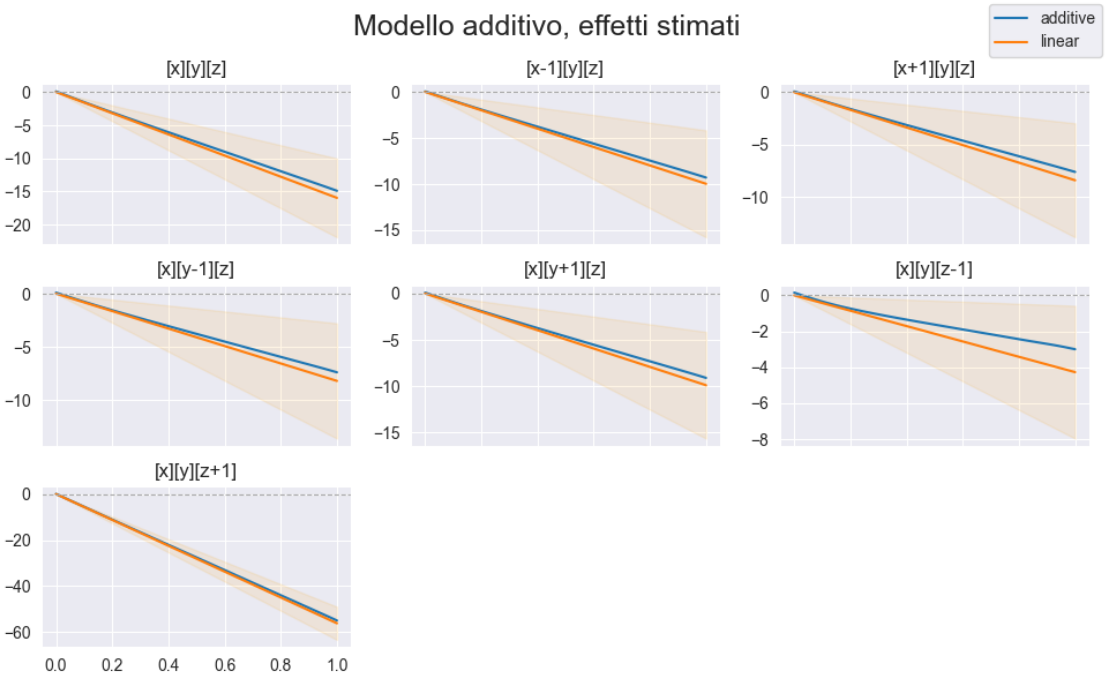

FIGURA 3.6: Dati di Airbnb, effetti parziali stimati.

disposizione del modello per cogliere il segnale nei dati non viene utilizzata e le stime risultano praticamente identiche a quelle del modello lineare, senza offrire indicazioni aggiuntive di interesse.

In Tabella 3.5 si riportano diverse metriche di interesse per valutare le prestazioni

dei due modelli.

Modello	Errore	Precisione	Sensibilità	F1 Score	AUC	AUC-PR
Lineare	0.090985	0.460958	0.460987	0.460973	0.665080	0.263174
Additivo	0.090644	0.462981	0.463010	0.462995	0.744651	0.315771

Tabella 3.6: Dati simulati sulle rilevazioni archeologiche, metriche di interesse.

Nonostante gli accorgimenti utilizzati per classificare come mura una proporzione di previsioni pari alla quella osservata, non si riesce ad ottenere un errore di classificazione inferiore alla proporzione osservata: se qualcuno classificasse tutti i voxels come suolo, farebbe comunque meglio del modello in termini di errore di classificazione. Nonostante questo, guardando ai valori di precisione e sensibilità, emerge che circa metà delle previsioni di mura del modello sono esatte, e parallelamente il modello riesce a identificare circa la metà dei veri voxel corrispondenti a mura: anche se non raggiunge ottime prestazioni, il modello riesce comunque a cogliere parte del segnale dei dati. Guardando ai valori di AUC e AUC-PR (aree sotto le curve ROC e Precision-Recall), si evidenzia una prestazione globalmente superiore a livello predittivo per il modello additivo.

I risultati solo parzialmente soddisfacenti per l'analisi di questo insieme di dati potrebbero essere da attribuire sia al tipo di modelli scelti che all'approccio generale adottato per l'analisi. Per quanto riguarda il tipo di modelli scelti, chiaramente sarebbe stato opportuno, specialmente alla luce del forte sbilanciamento dell'insieme di dati, utilizzare degli approcci di classificazione. In secondo luogo, si sarebbero potute provare delle strade di modellazione che enfatizzassero l'interazione tra le variabili esplicative, ad esempio strutture ad albero. Guardando all'impostazione generale dell'analisi, si ricordi che l'unità statistica di partenza è un volume tridimensionale. Anche se con l'approccio tabulare utilizzato si cerca di mantenere la nozione sulla località utilizzando i valori dei voxel circostanti per la previsione, è chiaro che la ristrutturazione dei dati oltre ad aver semplificato il problema ha portato ad una perdita di informazione. In un contesto reale sarebbe stato interessante provare degli approcci specifici per dati tridimensionali rappresentanti volumi: nell'ambito del *deep learning* la strada sarebbe stata quella di utilizzare convoluzioni tridimensionali per mantenere l'informazione sulla località, volendo rimanere in ambiti più parametrici si sarebbero potute usare *splines* prodotto tensoriale a tre dimensioni o approcci per dati funzionali.

Capitolo 4

Commenti finali

In questo lavoro ci si è focalizzati sul tema dell'analisi di Big Data. Nei settori produttivi, lo sviluppo tecnologico promosso dall'avvento dei personal computer ha portato all'accumulo di moli di dati sempre più grandi da analizzare. In questi contesti, al contrario rispetto a quello che accadeva in passato, le procedure di analisi devono tenere conto delle dimensioni dei dati e delle capacità computazionali a disposizione dell'analista. Per questi motivi, nel tempo si è reso necessario lo sviluppo di un nuovo *stack* tecnologico, in grado di affrontare i problemi legati all'analisi di questo tipo di insiemi di dati; nello specifico, sono state sviluppate infrastrutture *hardware* per aumentare la potenza di calcolo a disposizione e nuovi *software* che le potessero gestire.

Nel primo capitolo vengono inizialmente esposti i problemi legati all'analisi di Big Data, per passare ad una trattazione del concetto di calcolo parallelo e di architettura distribuita. Queste tematiche permettono di comprendere l'approccio moderno ai grossi carichi di lavoro computazionali, che prevede la parallelizzazione delle operazioni di analisi e lo svolgimento delle stesse in grandi *cluster* di computer, principalmente sviluppati e gestiti in ambienti *cloud*. Nella parte conclusiva del capitolo si prendono in considerazione due *software* sviluppati per gestire questi sistemi di calcolo, ossia MapReduce e Spark. Quest'ultimo viene utilizzato per le applicazioni riportate nel terzo capitolo.

Nel secondo capitolo si è esposta la teoria dei modelli lineari, concentrandosi in particolare sui diversi approcci di stima possibili in situazioni di Big Data, che solitamente prevedono aggiornamenti sequenziali delle quantità di interesse. Tra questi, se ne individua uno che è possibile implementare in parallelo secondo una logica MapReduce. Successivamente vengono presentati i modelli additivi, evidenziando come la stima di questi modelli possa essere ricondotta a quella di un modello lineare. Infine ci si concentra su uno dei principali problemi dell'inferenza statistica, ossia la costruzione di

intervalli di previsione in corrispondenza di nuove osservazioni. Considerata l'inapplicabilità delle procedure basate sulla verosimiglianza nel caso in cui non sia possibile verificare le assunzioni del modello, si espone il paradigma della *conformal inference*, un approccio che permette di costruire intervalli di previsione con copertura empirica garantita senza bisogno di assunzioni distributive o sul modello.

Nel terzo capitolo si sono messe in pratica le tecniche di stima e inferenza presentate nel capitolo 2, basandosi sulle infrastrutture di calcolo distribuito introdotte nel capitolo 1. Si è mostrato come richiedere ad un *provider* di servizi *cloud*, in questo caso Amazon, i servizi necessari per gestire ed utilizzare un *cluster* virtuale delocalizzato.

Utilizzando PySpark si sono potute sfruttare le capacità del *cluster* creato per l'analisi. Nello specifico, si è scritto un algoritmo con paradigma MapReduce per la stima di un modello lineare, sfruttando le funzioni interne del sistema. L'implementazione sviluppata risulta meno efficiente a livello di prestazioni rispetto a quella interna di PySpark, basata su un algoritmo di discesa del gradiente stocastica. Nonostante questo, la procedura proposta permette di ottenere una soluzione esatta, a differenza dell'implementazione interna, e affronta in maniera efficace il problema di partenza. In una situazione di *Big Data*, con $n \gg p$ e tale da non riuscire a caricare i dati in memoria, l'approccio sviluppato permette di neutralizzare il problema della numerosità, suddividendo il carico di lavoro tra i vari *workers* all'interno del *cluster*. In aggiunta a quanto detto, la metodologia è stata adattata alla stima del modello additivo, estendendo le possibilità applicative dell'approccio proposto.

Si sono confrontati i tempi di esecuzione dell'algoritmo in locale e nel *cloud*, notando il vantaggio in termini di efficienza di questo approccio anche nella situazione in cui i dati si possano caricare in memoria. Chiaramente, la vera potenzialità del *cloud computing* sta nella possibilità di analizzare in tempi brevi dati che non si sarebbero potuti processare altrimenti e nella flessibilità della definizione dell'architettura distribuita, che può essere scalata orizzontalmente o verticalmente in base alle necessità dell'analisi. Nello specifico, l'approccio utilizzato per la stima comporta una riduzione dei tempi di esecuzione pressapoco lineare con il numero di nodi *workers* utilizzati, prestandosi quindi all'impiego in architetture estese orizzontalmente, cioè con molti nodi.

Un commento a parte va fatto per l'approccio *conformal inference* per la definizione di intervalli di previsione. L'utilizzo pratico di questa metodologia, nella sua variante *rank-one-out* split conformal, risulta computazionalmente efficiente e permette di ottenere intervalli di previsione per tutte le unità a disposizione. La caratteristica di rilievo

di questi intervalli è il comportamento della copertura empirica: è garantita a prescindere dal modello e dalle sue prestazioni (gli intervalli saranno validi anche per modelli con pessime previsioni, anche se molto estesi), e converge dall'alto alla copertura nominale desiderata già per valori modesti di numerosità campionaria. L'implementazione adottata per questa metodologia si basa solamente su funzioni e metodi interni di PySpark, pertanto risulta perfettamente integrata nel *framework* di calcolo parallelo.

Appendice A

A.1 Modello lineare

```
# setup
import numpy as np
from scipy.linalg import solve_triangular

# functions for liner model estimation and prediction

def map_lm(row):
    """calculate x*t(x) and x*y for each row"""

    x = [1]
    x.extend(row['X'].toArray())
    x = np.array(x)
    y = row["y"]
    out = [np.outer(x, x), x*y]

    return out

def get_prediction_lm(row, beta):
    """calculate predictions for a row from estimated linear model"""

    x = row["X"].toArray()
    ypred = beta[1] + np.dot(x, beta[1:])

    return (row + (float(ypred),))

def cholesky_normal_equations(xtx, xty):
    """calculate the solution to the normal equations
    using Cholesky scomposition"""

    L = np.linalg.cholesky(xtx)
    z = solve_triangular(L, xty, lower = True)
```

```
    beta_hat = solve_triangular(L, z, lower = True, trans = "T")

    return beta_hat
```

A.2 Modello additivo

```
# setup
import numpy as np

# functions for additive model estimation and prediction
knots = np.linspace(0.05, 0.95, 10)

def ncs_elem(x, xk):
    """given a feature and a knot, calulates basis functtion"""

    out = (((xk-1/2)**2-1/12)*((x-1/2)**2)-1/12)/4 -
    ((np.abs(x-xk)-1/2)**4-1/2*(np.abs(x-xk)-1/2)**2 + 7/240)/24

    return out

def splS(knots):
    """define S penalty matrix for a single term"""

    q = len(knots)+2
    S = np.zeros([q,q])
    k = np.array(knots)
    S[2:, 2:] = ncs_elem(k[:,None], k)

    return S

def get_bigS(knots, lam = 0.01):
    """obtain global penalty matrix"""

    Slist = []
    p = len(num)
    for i in range(p):
        q = len(knots)+1
        S = np.zeros([p*q, p*q])
        S2 = np.zeros([len(S)+4, len(S)+4])
        S[i*q:(i+1)*q, i*q:(i+1)*q] = splS(knots)[1:,1:]
        S2[4:,4:] = S
        Slist.append(S2)
```

```
    bigS = sum(lam*elem for elem in Slist)

    return bigS

def map_gam(row):
    """
    for each row, compute natural cubic splines
    basis expansion for x then calculate x*t(x) and x*y
    """

    nx = [1]
    x = row["X"].toArray()
    y = row["y"]
    for j, xij in enumerate(x):
        if j in range(3):
            nx.append(xij)
        else:
            nx.append(xij)
            nx.extend([ncs_elem(xij, knot) for knot in knots])
    x = np.array(nx)
    out = [np.outer(x, x), x*y]

    return out

def get_prediction_gam(row, beta):
    """calculate predictions for estimated additive model"""

    nx = []
    x = row["X"].toArray()
    for j, xij in enumerate(x):
        if j in range(3):
            nx.append(xij)
        else:
            nx.append(xij)
            nx.extend([ncs_elem(xij, knot) for knot in knots])
    x = np.array(nx)
    ypred = beta[1] + np.dot(x, beta[1:])

    return (row + (float(ypred),))
```

A.3 Conformal inference

```
# setup
import pandas as pd
from pyspark.sql.functions import pandas_udf

# functions for conformal prediction

@pandas_udf("double")
def get_abs_error(yreal: pd.Series, ypred: pd.Series) -> pd.Series:
    """calculate absolute prediction error"""

    return np.abs(yreal-ypred)

@pandas_udf("double")
def find_quantile(err: pd.Series) -> pd.Series:
    """
    given ordered absolute errors,
    calculate ROO empirical quantile
    """

    level = 0.95
    m = int(np.ceil(len(err)*level))
    out = [err.drop([i]).values[m-1] for i in range(len(err))]

    return pd.Series(out)

def ROOsplitconformal(df):
    """
    given a Spark dataframe with columns y and yhat,
    calculate and sort absolute errors,
    find ROO empirical quantiles,
    calculate lower and upper bounds for
    ROO split conformal prediction intervals
    """

    out = df \
            .withColumn("abs_error",
            get_abs_error(col("y"), col("yhat"))).sort(col("abs_error"
                                    )) \
            .withColumn("ROO_eq", find_quantile(col("abs_error"))) \
            .withColumn("lower", col("yhat")-col("ROO_eq")) \
            .withColumn("upper", col("yhat")+col("ROO_eq"))

    return out
```

Bibliografia

Adjout, M. R. & Boufares, F. (2014). A massively parallel processing for the multiple linear regression. In *2014 Tenth International Conference on Signal-Image Technology and Internet-Based Systems*. IEEE.

Azzalini, A. & Scarpa, B. (2012). *Data analysis and data mining: An introduction*. OUP USA.

Chambers, J. M. (1971). Regression updating. *Journal of the American Statistical Association* **66**, 744–748.

Dean, J. & Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM* **51**, 107–113.

Efron, B. (1979). Bootstrap methods: another look at the jacknife. *Ann. Stat.* **7**, 1–26.

Fan, J., Guo, Y. & Wang, K. (2021). Communication-efficient accurate statistical estimation. *Journal of the American Statistical Association* .

Golub, G. H. & Van Loan, C. F. (2013). *Matrix computations*. JHU press.

Gu, C. & Gu, C. (2013). *Smoothing spline ANOVA models*, vol. 297. Springer.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., Fernández del Río, J., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C. & Oliphant, T. E. (2020). Array programming with NumPy. *Nature* **585**, 357–362.

Hastie, T. & Tibshirani, R. (1986). Generalized Additive Models. *Statistical Science* **1**, 297 – 310.

Hastie, T., Tibshirani, R., Friedman, J. H. & Friedman, J. H. (2009). *The elements of statistical learning: data mining, inference, and prediction*, vol. 2. Springer.

Jordan, M. I., Lee, J. D. & Yang, Y. (2019). Communication-efficient distributed statistical inference. *Journal of the American Statistical Association* **114**, 668–681.

Lei, J., G'Sell, M., Rinaldo, A., Tibshirani, R. J. & Wasserman, L. (2018). Distribution-free predictive inference for regression. *Journal of the American Statistical Association* **113**, 1094–1111.

Leskovec, J., Rajaraman, A. & Ullman, J. D. (2020). *Mining of massive data sets*. Cambridge University Press.

McKinney, W. et al. (2010). Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, vol. 445. Austin, TX.

Miller, A. J. (1992). Algorithm as 274: Least squares routines to supplement those of gentleman. *Applied Statistics* , 458–478.

Nocedal, J. & Wright, S. J. (2006). *Numerical Optimization*. New York, NY, USA: Springer, 2nd ed.

Pace, L. & Salvan, A. (1996). *Introduzione alla statistica: Inferenza, verosimiglianza, modelli*. Cedam.

Salvan, A., Sartori, N. & Pace, L. (2020). *Modelli Lineari Generalizzati*, vol. 124. Springer Nature.

Strong, G. C., Dorigo, T., Giammanco, A., Vischia, P., Kieseler, J., Lagrange, M., Nardi, F., Zaraket, H., Lamparth, M., Fanzango, F., Savchenko, O. & Bordignon, A. (2022). Dataset for the challenge at the 2nd MODE workshop on differentiable programming 2022.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P. & SciPy 1.0

Contributors (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* **17**, 261–272.

Vovk, V., Gammerman, A. & Shafer, G. (2005). *Algorithmic learning in a random world*. Springer Science & Business Media.

Wang, Y. (2011). *Smoothing splines: methods and applications*. CRC press.

Weisberg, S. (2005). *Applied linear regression*, vol. 528. John Wiley & Sons.

Wood, S. N. (2006). *Generalized additive models: an introduction with R*. Chapman and Hall/CRC.

Wood, S. N. (2011). Fast stable restricted maximum likelihood and marginal likelihood estimation of semiparametric generalized linear models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* **73**, 3–36.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S. & Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*.

Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J. et al. (2016). Apache spark: a unified engine for big data processing. *Communications of the ACM* **59**, 56–65.