Alma Mater Studiorum · Università
di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica per il Management

TECNICHE DI DATA MINING APPLICATE ALLA DECODIFICA DI DATI NEURALI

Relatore:
Chiar.mo Prof.
MARCO DI FELICE

Presentata da:
ANNA AVENA

Correlatori:
Dott. MATTEO FILIPPINI
Prof.ssa PATRIZIA FATTORI

Sessione II
Anno Accademico 2016/2017

Alla mia famiglia...

Abstract

Gli studi sulla decodifica dell'attività neuronale permettono di mappare gli impulsi elettrici della corteccia cerebrale in segnali da inviare a determinati dispositivi per poterli monitorare. È su questo tema che la ricerca scientifica si sta concentrando, al fine di aiutare le persone affette da gravi lesioni fisiche ad ottenere un maggiore grado di autonomia nelle piccole azioni di tutti i giorni.
In questo elaborato, sono stati analizzati dati derivanti da attività neuronali raccolti da esperimenti effettuati su primati non umani, eseguiti dal gruppo di ricerca della professoressa Patrizia Fattori nel Dipartimento di Farmacia e Biotecnologie dell'Università di Bologna.
Per lo svolgimento di questo esperimento, la cavia, è stata addestrata a svolgere un compito che consiste nell'afferrare gli oggetti proposti, uno alla volta, in ordine casuale. Durante l'esercizio, l'attività neuronale della cavia è stata registrata in vettori contenenti l'attività di *spiking*.
Ciò che si cerca di fare in questa tesi è ricostruire l'informazione relativa all'attività di una popolazione di neuroni, dato il suo spike vector.
Sono stati testati diversi algoritmi di classificazione e feature al fine di stabilire quale configurazione sia più affidabile per il riconoscimento dell'attività motoria svolta dalla cavia durante l'esperimento.
A tal proposito, è stato implementato un processo di data mining attraverso l'utilizzo del linguaggio python e del framework Scikit-learn che permette di

effettuare più classificazioni e stabilire quale fornisce una migliore performance.
I risultati dell'analisi dimostrano che alcune feature forniscono alti tassi di riconoscimento e che, a seconda del dominio del problema, è più indicato un determinato tipo di *preprocessing* rispetto ad un altro.

Introduzione

Nello scenario odierno l'intelligenza artificiale (AI: *artificial intelligence*) è presente in molte aree e fa da propulsore alla ricerca scientifica. In particolare, nell'ambito medico, le novità tecnologiche possono aiutare sia il medico sia il paziente per migliorare le cure e, di conseguenza, essere un valido supporto per l'intero sistema sanitario. Uno specifico impiego dell'AI riguarda la riabilitazione dei pazienti che hanno subìto gravi lesioni agli arti. La robotica ha giocato un ruolo fondamentale mettendo a disposizione prototipi di protesi robotizzate che si sono rivelate essere di grande aiuto per pazienti affetti da patologie quali lesioni spinali, esiti di ictus, tetraplegia, sindrome di Parkinson, ecc.

Ad esempio, sono state sviluppate protesi in grado di decodificare il segnale cerebrale permettendo a chi ha subìto l'amputazione di uno o più arti di replicare i movimenti di gambe e braccia in modo sempre più naturale.

A rendere possibile tutto questo sono le interfacce neurali, che mappano le scariche neuronali registrate dagli elettrodi in segnali che un computer decodifica e trasforma in movimenti.

La fase di decodifica dell'attività bioelettrica dei neuroni consiste nell'utilizzo di tecniche di *machine learning* che, sulla base delle informazioni raccolte, permettono di riconoscere pattern e costruire schemi da utilizzare per classificazioni future.

Il processo principale che si occupa di esplorare i dati e cercare relazioni sistematiche tra le variabili (o *feature*) è il *data mining*. Tale processo deve poter risolvere problemi appartenenti a diversi domini, talvolta elaborando i dati

di input per renderli più significativi prima che il modello venga addestrato, per ottenere maggiore accuratezza sulla previsione.

All'interno dell'elaborato sono stati analizzati dati derivanti da attività extracellulari di singole cellule, raccolti durante le sessioni di test condotte su due esemplari di primati non umani.
Gli esperimenti sono stati condotti dal gruppo di ricerca della professoressa Patrizia Fattori nel Dipartimento di Farmacia e Biotecnologie dell'Università di Bologna.
Per lo svolgimento di questo esperimento, le cavie, sono state addestrate ad afferrare 5 tipologie di oggetti proposti, uno alla volta, in ordine casuale.
Gli oggetti proposti sono stati scelti per indurre diverse prese: prensione con l'intera mano, prensione con tutte le dita eccetto il pollice, prensione a uncino con l'indice, prensione semplice usando il pollice contrapposto alle falangi delle altre dita e prensione avanzata con pollice e indice.
Durante l'esercizio, l'attività neuronale della cavia è stata registrata e memorizzata in vettori contenenti l'attività di *spiking*. Essendo state effettuate registrazioni sia in condizioni di luce che di buio, i dati raccolti sono stati memorizzati su diversi dataset (divisi per intervalli di interesse e condizioni visive) sui quali è stato successivamente condotto l'esperimento di classificazione.
L'obiettivo di questa tesi è la ricostruzione dell'informazione relativa all'attività di una popolazione di neuroni, dato uno *spike vector*. La fase più importante di questo processo è la costruzione del modello che verrà utilizzato per l'addestramento dell'algoritmo: esso può essere sviluppato in più modi e, a seconda di come sono implementate le *feature*, si otterranno performance diverse.
Con il lavoro svolto sono stati analizzati una molteplicità di algoritmi di data mining e *feature*, attraverso i quali è possibile stimare la classe di appartenenza di un campione e, per ciascuna classificazione, valutarne l'accuratezza e stimare la performance.

La tesi è stata strutturata in 6 capitoli.
Nello specifico, nel **capitolo 1** viene introdotto il tema del *machine learning* in ambito medico con annessi studi e ricerche correlate al problema della decodifica di dati neuronali, con un ulteriore focus sulle interfacce neurali, spiegando cosa sono e come vengono impiegate nella riabilitazione di pazienti con difficoltà motorie.
Nel **capitolo 2** viene approfondito il processo di *data mining*, con un'analisi degli algoritmi di classificazione che hanno fornito stime più accurate per questo caso di studio.
Il **capitolo 3** introduce, invece, gli strumenti utilizzati per l'implementazione degli script: il linguaggio di programmazione Python e la principale libreria orientata al data mining: Scikit-learn.
A seguire, nei **capitoli 4 e 5** si descrive più in dettaglio il caso di studio analizzato, partendo da un aspetto più teorico e successivamente dettagliando l'implementazione svolta, mostrando alcuni dei frammenti di codice relativi alle principali funzioni implementate.
Per finire, nel **capitolo 6** sono riportati i risultati ottenuti dalla fase di classificazione di alcuni campioni della popolazione.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Stato dell'arte

La perdita di un arto è un evento fisicamente e psicologicamente devastante che può trasformare le attività quotidiane in un compito molto difficile da compiere, oltre ad avere un notevole impatto anche sull'interazione sociale. È per questa ragione che negli ultimi decenni è aumentata sempre di più la necessità di sistemi di sostituzione o di riabilitazione per le persone che hanno avuto lesioni o malattie neurologiche. Numerosi sono i ricercatori che hanno compiuto studi e suggerito diversi approcci per il controllo di protesi artificiali attraverso l'analisi degli impulsi del cervello, ma spesso, quando viene generato un comando motore, questo non risulta essere altamente affidabile per l'applicazione protesica corticale controllata [1].
Sono in corso numerosi esperimenti atti a dimostrare come lo sviluppo di dispositivi artificiali basati sull'attività corticale motoria sia possibile attraverso l'utilizzo di interfacce BCI – Brain Computer Interfaces.
La finalità stessa della BCI, infatti, è quella di tradurre l'attività cerebrale in un comando per un elaboratore. Per raggiungere questo obiettivo possono essere utilizzate tecniche di *data mining* quali regressione o classificazione[1].

[1]La classificazione e la regressione sono i più comuni problemi in cui oggi è applicato il Data Mining. Essi differiscono tra loro a seconda del tipo di output che forniscono. La classificazione predice un'appartenenza ad una classe, perciò l'output predetto è categorico; la regressione predice uno specifico valore, pertanto è usata nei casi in cui il valore da predire può avere un numero illimitato (o una moltitudine) di possibili valori.

L'approccio più accreditato per identificare i modelli di attività cerebrale è basato sugli algoritmi di classificazione.

Tutto questo è stato possibile grazie all'aumento di conoscenze sul funzionamento del sistema nervoso e grazie ai numerosi progressi tecnologici avvenuti negli ultimi anni, che hanno fornito le basi per lo sviluppo di nuove interfacce elettronico-biologico in grado di intercettare, decodificare e mappare l'attività neurale.
Storicamente sono stati compiuti diversi studi il cui obiettivo è stato quello di trovare una correlazione fra attività neuronale e movimento degli arti.
Il neuroscienziato Edward V. Evarts fu il primo che, nel 1968, dimostrò l'esistenza di una relazione fra la frequenza di scarica di singoli neuroni ed i movimenti volontari che venivano effettuati. In particolare in assenza di movimento non vi è alcuna scarica e, a seconda del movimento del braccio che viene effettuato, vi sono scariche differenti. L'esperimento è stato condotto su primati non umani sottoposti a movimenti di flessione ed estensione del braccio [2].
Successivamente Donald R. Humphrey ha approfondito questi studi, sempre su primati non umani, dimostrando come fosse possibile predire con buona accuratezza un movimento, bastandosi sulla relazione esistente fra attività delle braccia e neuroni che hanno rilasciato delle scariche [3].
Apostolos P. Georgopoulos, nel 1986, analizzando popolazioni di neuroni nella corteccia motoria primaria di primati non umani, ha osservato che quando cellule individuali erano rappresentate come vettori che davano il contributo pesato lungo l'asse della loro direzione preferita, la somma risultante di tutti i vettori era in direzione congruente con la direzione del movimento [4], confermando così la possibilità di ricostruire il movimento dell'arto dall'attività neurale.

Gran parte delle protesi robotiche attualmente in studio hanno una buona correlazione fra i segnali estratti ed il movimento desiderato. Hanno un

ruolo fondamentale le tecniche innovative del data mining e di intelligenza artificiale, che possano offrire un controllo protesico naturale, sfruttando opportune sinergie muscolari. Numerosi sono gli studi innovativi il cui obiettivo è quello di analizzare in tempo reale le risposte corticali, consentendo la selezione di combinazioni complesse per un determinato compito motorio. Queste tecniche saranno applicate sia agli arti superiori che agli arti inferiori per migliorare i percorsi riabilitativi di braccia e gambe.

1.1 Brain Computer Interfaces

Le Brain Computer Interfaces (BCIs) utilizzano le informazioni neurali registrate dal cervello per il controllo dei dispositivi esterni. Lo sviluppo di sistemi BCI è principalmente focalizzato sul miglioramento dell'indipendenza funzionale per gli individui con gravi problemi motori, si vuole fare ciò attraverso la fornitura di strumenti per la comunicazione e la mobilità [5].
Lo sviluppo delle interfacce neurali non solo ha implicazioni di carattere clinico ma è anche un potente strumento nelle mani degli neuroscienziati: codifica e decodifica sono strettamente correlati, lo sviluppo di modelli di decodifica che invertono il problema da -cosa codifica un gruppo di neuroni- a -che informazione è possibile recuperare dal gruppo di neuroni oggetto di studio-, permette un punto di vista differente e complementare per confermare i modelli di codifica proposti con l'analisi classica.
Sebbene esistano molti tipi di BCIs, ci sono componenti chiave comuni a tutti: il sistema di registrazione che estrae l'attività neurale attraverso l'ausilio di sensori, il sistema di decodifica che traduce questi dati in segnali di azione, il sistema attuatore comandato dai segnali generati. Questi punti definiscono una tipica interfaccia neurale di tipo open-loop. Ciò che contraddistingue un'interfaccia open-loop da un'interfaccia closed-loop è che a quest'ultima viene aggiunto un feedback sensoriale: può essere rappresentato dalla semplice osservazione del movimento dell'attuatore (protesi meccanica, movimento di un cursore su uno schermo, etc.) accompagnato da un rinforzo

positivo, oppure da più complessi sistemi di elettrostimolazione di nervi periferici o direttamente in corteccia per emulare i meccanismi propriocettivi. La figura 1.1 rappresenta le 4 fasi di un'interfaccia closed-loop:

1. Il sistema di registrazione che estrae i segnali neurali
2. L'algoritmo di decodifica che trasla i segnali neurali in un set di comandi
3. L'attuatore che è controllato da questi segnali
4. Il feedback sensoriale

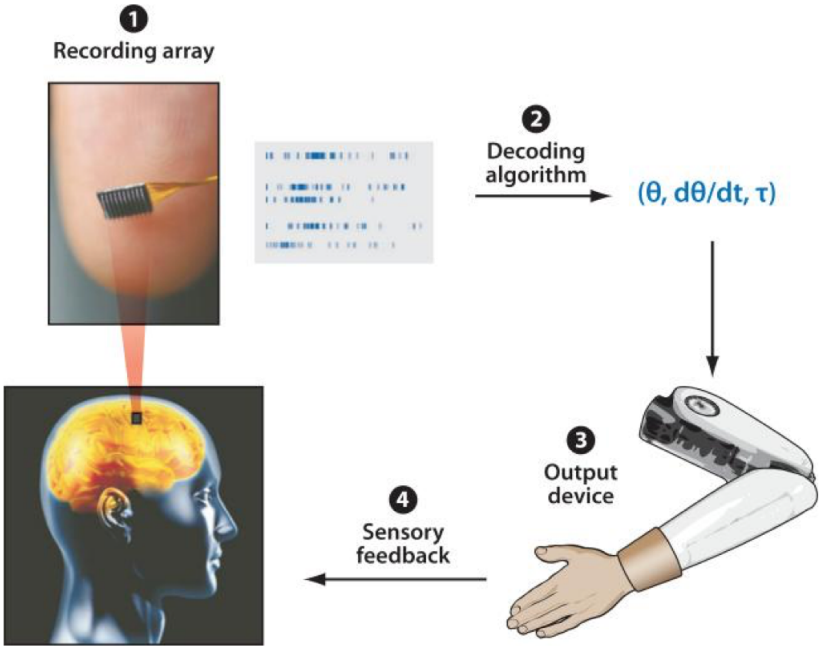


Figura 1.1: Le quattro componenti di un'interfaccia neurale closed-loop

Un sistema BCI lavora partendo dall'estrazione dei segnali emessi dal cervello e, attraverso l'applicazione di algoritmi di machine learning, cerca di effettuare una classificazione. Il miglior modo per testare la performance della classificazione è la selezione degli algoritmi di apprendimento più avanzati,

avvalendosi di tecniche di preprocessamento dei dati e tecniche di meta-classificazione. L'obiettivo di una interfaccia neurale è quello di far comunicare un individuo con il mondo esterno, avvalendosi solamente dei segnali interni emessi del cervello [6].

1.2 Intracranial recording devices

I sensori svolgono una funzione primaria poiché registrano i segnali temporali e spaziali rilasciati dalle scariche neuronali del cervello. La risoluzione temporale fa riferimento agli intervalli di tempo in cui i segnali neuronali sono rilevati e campionati. Esistono diversi strumenti, con differenti caratteristiche, che permettono di misurare le scariche neuronali: uno spettroscopio ad infrarossi[2], ad esempio, richiede 25 secondi per generare un comando binario affidabile, mentre dispositivi moderni di registrazione dei segnali cerebrali riescono ad effettuare una registrazione in tempi pari ai sub-millisecondi mantenendo la stessa precisione. Alcuni di questi strumenti sono l'elettroencefalogramma[3], la risonanza magnetica[4] e la magnetoencefalografia[5].
L'interfaccia tra il dispositivo di registrazione ed i valori neuronali rilevati è di complessa interpretazione, bisogna fare considerazioni specifiche su tutte le componenti del dispositivo che possono influire notevolmente sulle caratteristiche di registrazione: duttilità del materiale dell'elettrodo, rivestimento,

[2]Questo strumento utilizza la luce che si avvicina agli infrarossi per visualizzare in un'immagine l'ossigenazione e la perfusione del sangue. Quando un'area cerebrale si attiva per svolgere un compito, questa è più ricca di sangue che apporta i nutrienti necessari alle cellule per elaborare il segnale fisiologico.

[3]È il metodo più utilizzato ed il più facile da acquisire. Vengono utilizzati elettrodi, disposti su un elmetto in silicone o stoffa, e attraverso un cavo amplificatore di biosegnali viene registrata l'attività elettrica dell'encefalo.

[4]È stato dimostrato che una BCI può essere efficacemente implementata dalla risonanza magnetica: i cambiamenti comportamentali sono la conseguenza della regolamentazione in aree circoscritte del cervello e queste sono visibili a lungo termine.

[5]È metodo non invasivo che misura i campi magnetici causati dalla corrente elettrica generata da correnti ioniche a loro volta generate dall'attività cerebrale.

conducibilità, etc.
La risoluzione spaziale, invece, fa riferimento al volume di informazioni derivanti dal singolo neurone.
Essendo il cranio classificato come conduttore dell'attività elettrica prodotta dalla corteccia, viene stimata l'attività corticale, ed i sensori che intercettano l'attività elettrica attraverso gli elettrodi conduttivi assegnano un'impronta caratteristica ad ogni pensiero ed in questo modo ogni movimento ha una diversa mappatura.
Va specificato che i dispositivi in grado di registrare i potenziali d'azione dei neuroni danno luogo a diversi risultati, alcuni più performanti di altri, a seconda dell'area corticale che viene analizzata. È dimostrato che per i sistemi BCI ci sono aree della corteccia motoria primaria più pertinenti rispetto ad altre [5].

1.3 BCI decoding

Finora sono state descritte le BCIs come dispositivi che utilizzano informazioni neurali registrate dal cervello per il controllo volontario dei dispositivi esterni.
Si vedrà adesso qual è il processo di conversione delle informazioni neurali in segnale in uscita: questo processo prende il nome di decodifica neurale.
Iniziamo descrivendo come i moderni algoritmi di decodifica neurale riescano a decifrare i segnali di comando utilizzati per controllare dispositivi esterni.
Il processo di decodifica parte delle informazioni ottenute dai potenziali d'azione prodotti dall'attività neurale, successivamente vengono applicati algoritmi di classificazione. La scelta dell'algoritmo deve essere fatta secondo opportuni criteri: ogni algoritmo di classificazione è più o meno adatto ad una specifica interfaccia neurale [7].

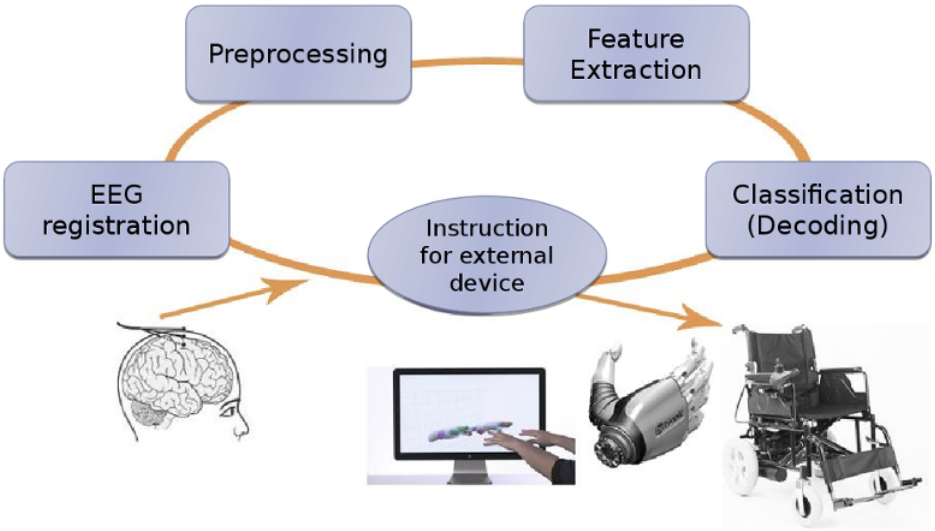


Figura 1.2: Fasi del BCI decoding

1.4 Applicazioni sperimentali

Numerosi sono stati gli esperimenti volti a decodificare ed analizzare le informazioni degli impulsi neuronali in input. La finalità di tali esperimenti è il controllo di una pluralità di tecnologie assistive come computer e dispositivi di assistenza robotica in base alla predizione del movimento.
Di seguito saranno riportati alcuni dei lavori svolti in questo ambito.
I primi esperimenti clinicamente rilevanti riportano come sia possibile controllare il movimento di un cursore su di uno schermo, ad esempio personal computer, dispositivi mobili e tablet. In particolare è stato dimostrato come le interfacce grafiche (GUI) si prestino positivamente alla rappresentazione di comandi registrati dal processore di segnale neurale di Cerebus e successivamente trasmessi e decodificati dal software Matlab. Applicando questa tecnica si è stati in grado di svolgere anche compiti molto complessi ottenendo buoni risultati in termine di performance, infatti, durante le sessioni di controllo del cervello, la cavia era in grado di completare correttamente il 98% delle prove [8].

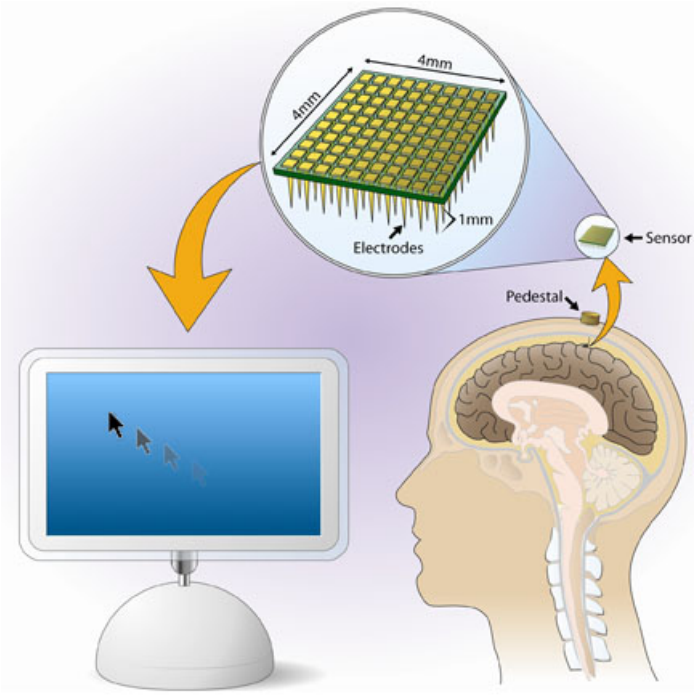


Figura 1.3: Sistema hardware-software di interfacciamento tra cervello e computer in grado di leggere le trasmissioni sinaptiche scambiate dai neuroni del cervello e trasformarle in movimenti del cursore su uno schermo di computer.

Altri esperimenti altrettanto rilevanti, in ambito di robotica controllata, sono stati compiuti recentemente e hanno dato dimostrazione di come sia possibile mappare in tempo reale l'attività nervosa relativa ai segnali di controllo dell'arto. Per riuscire a fare ciò, è particolarmente importante stabilire un flusso di informazioni tra il sistema nervoso dell'utente ed il dispositivo artificiale. A tal fine sono state utilizzate interfacce neurali invasive collegate direttamente al sistema nervoso periferico che hanno permesso di poter controllare la protesi dell'arto/mano [9].

Ulteriori studi si sono occupati di ripristinare la sensazione al tatto in una persona con amputazione della mano, consentendo così di far percepire informazioni sensoriali all'utente attraverso la mano artificiale.
La mano artificiale è stata collegata chirurgicamente all'interno dei nervi del braccio sinistro dell'utente attraverso degli elettrodi trans neuronali, si è cosi riusciti a trasmettere, in tempo reale, sensazioni tattili al cervello del paziente, permettendogli di riuscire a manipolare oggetti con la giusta forza e dandogli la capacità di percepire forma e consistenza degli oggetti stessi [10]. Questa è la prima volta in assoluto che nella neuro prostetica, ramo della neurologia che studia le interfacce cerebrali uomo-macchina, il feedback sensoriale è stato restituito e usufruito da un amputato, in tempo reale per il controllo di un arto artificiale.

Questi esperimenti mostrano che le interfacce neurali sono una buona soluzione per quanto riguarda la scrittura neurale. Tuttavia, rimane impegnativa la registrazione dell'attività bioelettrica dei neuroni, soprattutto se effettuata a lungo termine.

Capitolo 2

Tecniche del data mining

2.1 Introduzione al data mining

Il data mining è il fulcro di un processo più generale denominato *Knowledge Discovery in Databases*. Il KDD è un processo automatico di esplorazione dei dati allo scopo di identificare pattern validi, utili e non noti a priori. Questo processo è effettuato su un insieme di dati di grandi dimensioni e spesso di elevata complessità [11].

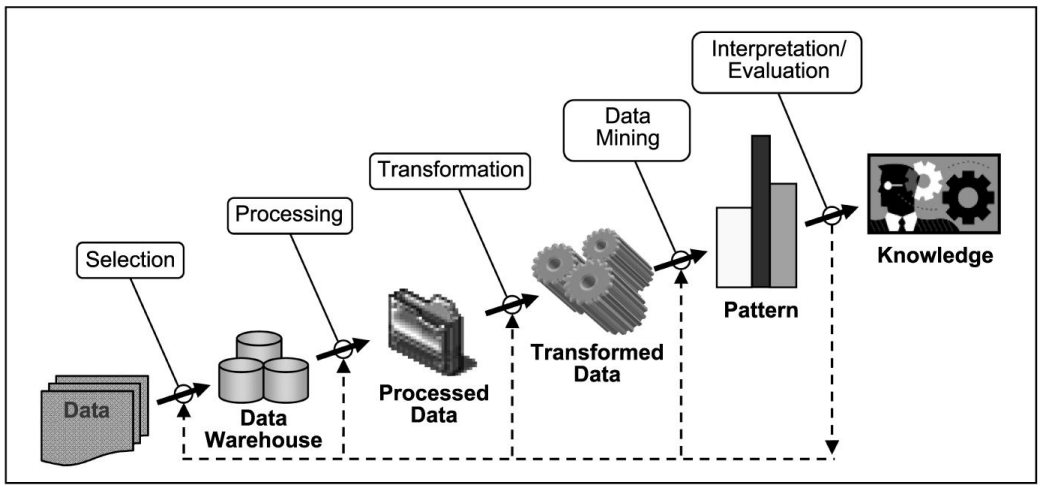


Figura 2.1: Il processo di KDD

Il data mining è l'insieme di meccanismi automatici progettati per consentire l'esplorazione di grandi moli di dati alla ricerca di tendenze consistenti e/o relazioni sistematiche tra variabili, e successivamente per validare le scoperte attraverso l'applicazione di comportamenti rilevanti su nuovi sottoinsiemi di dati.
Il data mining fa uso di una varietà di tecniche per estrarre informazioni e conoscenze che serviranno come supporto alle decisioni strategiche e/o alle previsioni: queste tecniche si dividono in:

- *verification-oriented*: il sistema si occupa di verificare le ipotesi generate da una sorgente esterna;

- *discovery-oriented*: il sistema è in grado di identificare nuovi pattern autonomamente partendo dai dati.

I metodi di learning utilizzati dal data mining possono essere di due tipi: supervisionati e non supervisionati. I primi scoprono la relazione fra attributi di input e di output dei dati e successivamente costruiscono il modello, i secondi, invece, raggruppano i campioni senza uno schema pre-specificato.
Sono elencate di seguito le principali caratteristiche:

Supervised	**Unsupervised**
Numero di classi conosciute a priori	Numero di classi non conosciute
Basato su un training set	Nessuna conoscenza preliminare
Utilizzato per classificare future osservazioni	Utilizzato per comprendere (esplorare) i dati

Tabella 2.1: Differenze fra approccio supervisionato e non supervisionato per il learning

2.2 Algoritmi di classificazione

Al fine di controllare una BCI, l'utente deve produrre modelli di attività differenti del cervello che saranno identificati dal sistema e tradotti in comandi. L'algoritmo di classificazione deve consentire di stimare automaticamente la classe di dati rappresentati da un vettore di funzione.

Le tecniche di classificazione utilizzano un training set dove tutte le istanze sono associate ad etichette di classi già note. Gli algoritmi di classificazione costruiscono un modello a partire dal training set e tale modello è utilizzato per classificare nuove istanze dette test sets.

Per la costruzione dei vari modelli, l'ideale sarebbe disporre di vari training sets tutti con uguale dimensione, diversi tra loro e soprattutto indipendenti. Tuttavia, generalmente si dispone di un unico training set, e questo raramente presenta le caratteristiche ottimali per essere trattato al meglio dagli algoritmi di data mining. È quindi necessario mettere in atto una serie di azioni volte a consentire il funzionamento degli algoritmi di interesse quali:

- *aggregazione*: combina due o più attributi in un solo attributo al fine di ridurre la cardinalità del dataset, di effettuare un cambiamento di scala e di stabilizzare i dati.

- *campionamento*: questa tecnica è particolarmente utilizzata al fine di ridurre la complessità del processo di analis, individuando un campione rappresentativo sul quale applicare le tecniche di data mining.

- *riduzione della dimensionalità*: questa tecnica cerca di ridurre la "maledizione della dimensionalità" [12] attraverso l'individuazione degli outlier e l'eliminazione di attributi non rilevanti

- *selezione degli attributi*: la selezione mira ad eliminare gli attributi ridondanti e le caratteristiche irrilevanti dell'oggetto[1]

- *creazione degli attributi*: questa tecnica mira a creare nuovi attributi che meglio catturino le informazioni rilevanti in modo più efficace rispetto agli attributi originali

- *trasformazione degli attributi*: i metodi più utilizzati sono la standardizzazione, normalizzazione, discretizzazione e binarizzazione dei dati. Sono particolarmente utili per enfatizzare alcune proprietà dei dati e ridurre range di variabilità troppo elevate

Il preprocessamento dei dati è una fase molto importante ai fini del data mining, esso infatti richiede il 70-80% del tempo e dello sforzo complessivo del processo di KDD.

È stato dimostrato che, per ciascuna BCI, esistono algoritmi di classificazione più adatti rispetto ad altri, è perciò importante confrontare gli algoritmi e valutare le loro prestazioni in base al contesto [7].
Sono riportati di seguito gli algoritmi di classificazione che hanno fornito le migliori performance per il caso di studio trattato in questa tesi.

2.2.1 K-Nearest Neighbors

I modelli di classificazione K-Nearest Neighbors (k-NN o classificatore ai primi vicini) sono costruiti per risolvere problemi del tipo "dato un insieme di punti ed un punto query nello spazio metrico multidimensionale, trovare il punto più vicino al punto query" [13]. L'applicazione di questo classificatore è molto semplice ed intuitiva per spazi a due o tre dimensioni, mentre si complica per spazi ad elevate dimensioni.

[1] Un esempio di caratteristica irrilevante ai fini del data mining è la matricola di uno studente per predire la sua media.

È importante la scelta del parametro K poiché indica il numero di elementi che devono essere presi in considerazione quando viene effettuata la scelta. Se K è troppo piccolo, la classificazione rischia di essere sensibile ai rumori, viceversa se K è troppo grande la classificazione può essere computazionalmente costosa.

Nella figura 1.3 è rappresentato un esempio di classificazione mediante k-NN. Il punto sotto osservazione è il cerchio verde. Le classi sono due:

- quella dei triangoli rossi;
- quella dei quadrati blu.

Se k = 3 (cioè vengono considerati i 3 oggetti più vicini), allora l'elemento verde viene inserito nella stessa classe dei triangoli rossi perché sono presenti 2 triangoli e 1 quadrato. Se k = 5 allora viene inserito nella stessa classe dei quadrati blu perché sono presenti 3 quadrati e 2 triangoli.

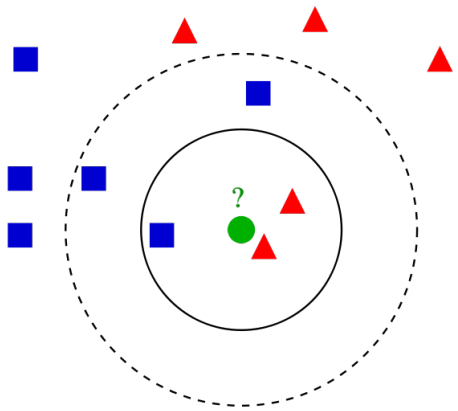


Figura 2.2: Esempio di classificazione mediante k-NN

Il calcolo della distanza fra gli oggetti viene fatto utilizzando la distanza di Manhattan per spazi unidimensionali, la distanza Euclidea per gli spazi bidimensionali e, nel caso in cui si debbano manipolare stringhe e non numeri,

si possono usare altre distanze come ad esempio la distanza di Hamming[2]. Il principale vantaggio di questo approccio basato sulla memorizzazione è il fatto che il classificatore si adatta ad ogni eventuale incremento dell'insieme dei valori utilizzati per l'addestramento, poiché per l'apprendimento non viene costruito alcun modello. Tuttavia, il metodo presenta due svantaggi: il primo è dato dal costo computazionale per la classificazione di nuovi campioni, il secondo è legato al problema dell'overfitting. Quest'ultimo è un fenomeno legato alle caratteristiche dei campioni, in particolare all'aumentare della dimensione del dataset di addestramento aumenta anche la possibilità che gli elementi più vicini risultino comunque troppo lontani, ciò determina il fatto che non si è in grado di offrire una buona stima.

2.2.2 Support Vector Machines

Le Support Vector Machines sono metodi di classificazione che generano un'approssimazione globale del modello di classificazione utilizzando i dati di addestramento.
Considerando un esempio di classificazione con sole due classi: quadrati rossi e cerchi blu, come mostrato in figura 2.3. Supponiamo di avere un insieme di addestramento di N campioni, ognuno dei quali appartenente ad una delle due classi. Quando le classi di possibile appartenenza sono due, come in questo caso, siamo di fronte ad un problema di classificazione binaria. Un approccio geometrico al problema della classificazione binaria consiste nella ricerca di una superficie che separi lo spazio di input in due porzioni distinte, dove rispettivamente giacciono gli elementi delle due classi.
Nel caso in cui i dati siano linearmente separabili (come in figura 2.3) si può effettuare una classificazione lineare in cui si assume almeno un iperpiano (ma anche più di uno) in grado di separare i campioni dell'insieme di addestramento, nell'esempio i quadrati rossi e i cerchi blu. L'obiettivo è quello di trovare l'iperpiano che separa nel modo migliore l'insieme dei campioni, cioè

[2]Nella teoria dell'informazione, la distanza di Hamming tra due stringhe di ugual lunghezza è il numero di posizioni nelle quali i simboli corrispondenti sono diversi.

l'iperpiano che rende massimo il margine che rappresenta la distanza minima fra le due classi.
La capacità di generalizzazione del classificatore cresce al crescere del margine e quindi la capacità di generalizzazione massima è data dall'iperpiano a margine massimo, nell'esempio è l'iperpiano A a soddisfare questa condizione e perciò è detto *Optimal separting hyperplane* (OSH).
Si noti che per l'addestramento sono importanti solo gli elementi più vicini all'iperpiano e questi vengono chiamati vettori di supporto.
Determinare l'iperpiano ottimo equivale a risolvere il seguente problema:

$$\max_{w \in R^n, b \in R} \quad \min_{x^i \in P \cup Q} \left\{ \frac{|w^T x^i + b|}{\|w\|} \right\}$$

Dove P e Q sono due classi diverse,
w ($w \in R^n$) è un vettore e b ($b \in R$) è uno scalare tali che

$$w^T x^i + b \quad \geq \varepsilon, \quad \forall x^i \in P$$

$$w^T x^j + b \quad \geq -\varepsilon, \quad \forall x^j \in Q$$

Graficamente:

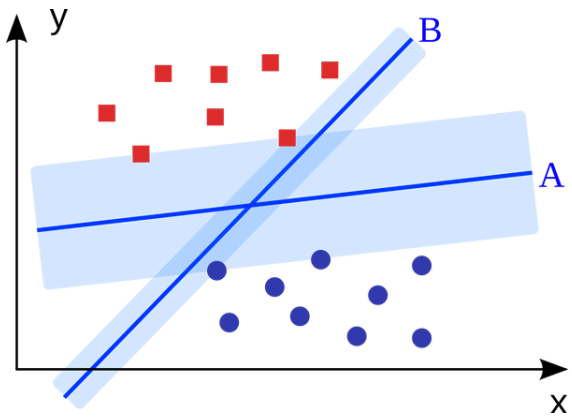


Figura 2.3: Esempio di classificazione mediante SVM

Si parla invece di classificazione non lineare se non esiste nessun iperpiano in grado di separare i campioni delle diverse classi. Una soluzione consiste nel proiettare l'insieme di addestramento in uno spazio di dimensione maggiore, in cui sia possibile effettuare una classificazione lineare e trovare l'OSH. Ciò è possibile utilizzando una funzione di *mapping* ϕ come mostrato in figura 2.4.

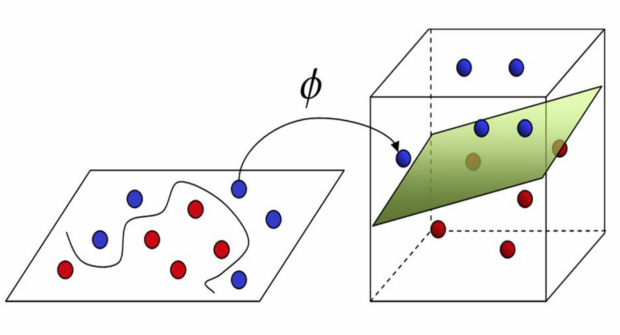


Figura 2.4: Classificazione di dati non linearmente separabili mediante l'utilizzo di una funzione di mapping ϕ in grado di proiettare l'insieme di addestramento in uno spazio di dimensioni maggiori.

2.2.3 Naive Bayes

I modelli di classificazione bayesiana si basano sul Teorema di Bayes e sono in grado di prevedere la probabilità che una determinata tupla appartenga ad una particolare classe.
Questo classificatore si basa sui modelli di probabilità che incorporano le ipotesi di forte indipendenza tra ogni coppia di funzioni. Data una classe variabile y e una funzione vettoriale $[x_1, \ldots, x_n]$ dipendente, il teorema di Bayes afferma la seguente relazione:

$$P(y|x_1, \ldots, x_n) = \frac{P(y)P(x_1, \ldots, x_n)}{P(x_1, \ldots, x_n)}$$

Utilizzando il presupposto di indipendenza "naive" si ottiene che:

$$P(x_1|y, x_1, \ldots, x_{\mathrm{i}-1}, x_{\mathrm{i}+1}, \ldots, x_n) = P(x_i, y)$$

Semplificando per ogni i:

$$P(x_1, \ldots, x_n) = \frac{P(y)\prod_{i=1}^{n} P(x_i|y)}{P(x_1, \ldots, x_n)}$$

Poiché $P(x_1, \ldots, x_n)$ è costante dato l'input, possiamo usare la seguente regola di classificazione:

$$P(x_1, \ldots, x_n)\alpha P(y)\prod_{i=1}^{n} P(x_i|y)$$

$$\Downarrow$$

$$\hat{y} = arg\ \underset{y}{max}\ P(y)\prod_{i=1}^{n} P(x_i|y)$$

Possiamo usare la stima del massimo della probabilità a posteriori (MAP) per calcolare *P(y)* e $P(x_i|y)$; $\hat{y}$ indica quindi la frequenza relativa alla classe y nel training set.

Sono possibili diverse implementazioni del classificatore di Naive Bayes, esse differiscono fra loro per le ipotesi che fanno per quanto riguarda la distribuzione di $P(x_i|y)$ [14].
Nonostante le ipotesi apparentemente troppo semplificate, i classificatori bayesiani garantiscono buone prestazioni in molte situazioni del mondo reale, ad esempio nel caso del filtraggio dello spam.
Un vantaggio del classificatore Naive Bayes è che richiede una piccola quantità di dati di training per stimare i parametri (medie e varianze delle variabili) necessari per la classificazione, questo perchè ogni distribuzione può essere stimata in modo indipendente, come se fosse una distribuzione unidimensionale.

2.2.4 Decision Tree

Gli alberi di decisione hanno l'obiettivo di creare un modello che preveda il valore di una determinata variabile attraverso l'apprendimento di semplici regole decisionali dedotte dalle caratteristiche dei dati.
Ad esempio, nell'esempio di figura 2.5, l'albero decisionale è costruito utilizzando tecniche di apprendimento a partire dall'insieme dei dati iniziati (*training set*) per i quali è nota la classe (in questo caso C1 o C2). Gli esempi usati per l'apprendimento sono descritti come vettori di coppie attributo-valore e sono utilizzati per imparare la definizione di una funzione di classificazione. Tale funzione è appresa in forma di albero in cui ogni:

Nodo interno rappresenta una variabile;

Arco verso un nodo figlio rappresenta un possibile valore per quella proprietà;

Foglia rappresenta il valore predetto per la classe a partire dai valori delle altre proprietà, che nell'albero è rappresentato dal cammino dal nodo radice al nodo foglia.

L'albero di decisione viene poi consultato per stabilire a quale classe appartiene la nuova istanza che si vuole classificare.

In generale si può notare che più profondo è l'albero, più complesse sono le regole decisionali e più adatto è il modello.

La classificazione di una nuova istanza avviene seguendo l'ordine stabilito dai seguenti criteri:

- la prima regola la cui precondizione è soddisfatta dalla istanza viene usata per generare la classificazione
- se nessuna regola ha le condizioni soddisfatte, si utilizza la regola di default per classificare l'istanza (cioè si ritorna la classe più frequente nell'insieme di apprendimento)

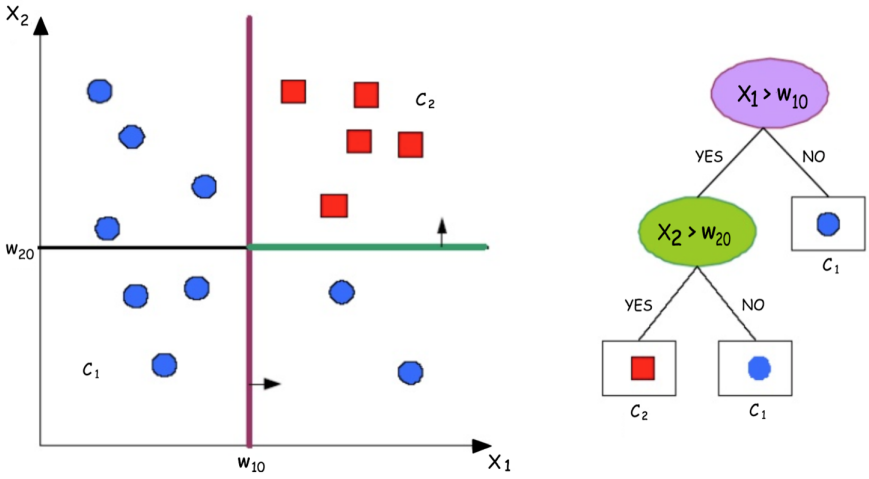


Figura 2.5: Esempio di classificazione mediante Decision Tree

L'albero di decisione è un modello di classificazione che presenta numerosi vantaggi:

- l'apprendimento è veloce, semplice da capire e da interpretare
- è possibile effettuare una classificazione anche se il training set contiene errori e/o valori mancanti
- non è necessaria la normalizzazione dei dati
- il costo computazionale richiesto per la previsione delle nuove istanze è logaritmico rispetto al numero di punti utilizzati per l'addestramento
- è in grado di gestire sia dati numerici che categoriali

Tuttavia ci sono anche alcuni svantaggi:

- può crearsi una situazione di *overfitting*, ovvero la realizzazione di un albero troppo complesso che non generalizza bene i dati in input;
- gli alberi decisionali possono essere instabili perché piccole variazioni nei dati potrebbero causare la generazione di un albero completamente diverso;

- l'apprendimento di alberi di decisione si basa su algoritmi euristici di tipo *divide et impera* in cui vengono prese decisioni localmente ottimali su ciascun nodo. Tali algoritmi non possono garantire la costruzione dell'albero decisionale globalmente ottimale;

- se ci sono delle classi che dominano su altre classi, è probabile che gli alberi di decisione creati risultino distorti. É consigliabile bilanciare il set di dati prima dell'allenamento dell'albero decisionale.

2.2.5 Neural Networks

I modelli di classificazione basati sulle reti neurali sono degli approssimatori universali, ispirati alla struttura e al funzionamento dei neuroni del cervello. I neuroni sono in grado di trasmettere un segnale elettrico verso altri neuroni, così da creare delle correnti locali la cui somma, se supera una certa soglia, si trasforma in un impulso (detto *spike*).

In una rete neurale i neuroni sono rappresentati come delle *unità* in grado di elaborare in modo molto semplice i segnali che ricevono in ingresso dagli altri neuroni. Il segnale elettrico è rappresentato mediante delle *connessioni* tra le unità, ogni connessione è pesata mediante coefficienti stimabili tramite procedura iterativa. L'effetto del peso sul segnale veicolato dalla connessione viene rappresentato *moltiplicando* il segnale stesso per il peso prima che esso raggiunga l'unità a valle.
Ad esempio se x è il segnale che viaggia lungo la connessione tra un primo neurone ed un secondo neurone, e w è il peso di questa connessione, il segnale "*pesato*" che arriva a valle sarà pari a: $w * x$.
Le unità di solito compiono due operazioni molto semplici sui segnali in arrivo: calcolano il *potenziale di attivazione* e si *attivano*.
Il calcolo del potenziale di attivazione avviene, al netto di un coefficiente

additivo definito intercetta, sommando i segnali pesati che arrivano dagli altri neuroni.

Siano x_1, x_2, x_3 i segnali che arrivano dagli altri neuroni attraverso le tre connessioni w_1, w_2, w_3, allora il potenziale d'azione P si calcola come segue:

$$P = w_1x_1 + w_2x_2 + w_3x_3$$

Le unità si attivano sulla base del potenziale d'azione. Questo significa che esse mandano un certo segnale a tutte le unità collegate con esse a valle, sulla base del potenziale di attivazione ricevuto. Il calcolo del segnale da inviare alle altre unità viene fatto mediante l'utilizzo di funzioni matematiche tipiche, chiamate *funzioni di trasferimento* che non verranno approfondite in questa tesi.
Questo classificatore è particolarmente adattivo poiché l'addestramento del modello, e quindi l'apprendimento, avviene attraverso la modifica dei pesi delle connessioni.

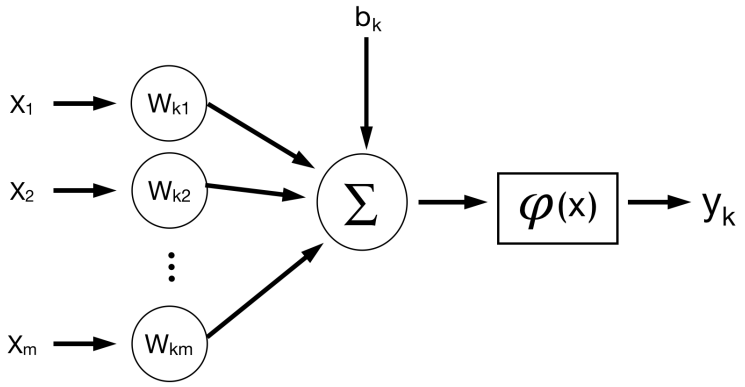


Figura 2.6: Modello non lineare di un neurone k dove x_i sono i pesi del neurone, u_k è la combinazione lineare degli input, b_k è il valore soglia del neurone, $\phi(x)$ è la funzione di attivazione e y_k è l'output generato.

2.3 Meta-Classificatori

Questo modello di classificazione trova riscontro nel comportamento umano di fronte a decisioni critiche: in tali situazioni, difficilmente ci si basa sul giudizio di un singolo individuo, piuttosto si preferisce sottoporre il problema in esame a diversi esperti che forniscono varie soluzioni. Nell'ambito del data mining, è possibile paragonare i modelli costruiti attraverso un learner di base ad un esperto a cui chiedere consiglio, mentre un metaclassificatore può essere paragonato ad un'assemblea di esperti che deve decidere la soluzione da adottare.
Questa classe di modelli è relativamente recente e attualmente non esiste uno studio specifico che possa guidare un utente nella scelta del meta-classificatore più adatto alle proprie esigenze.
In questa sezione saranno presentati alcuni dei meta-classificatori storici più noti.

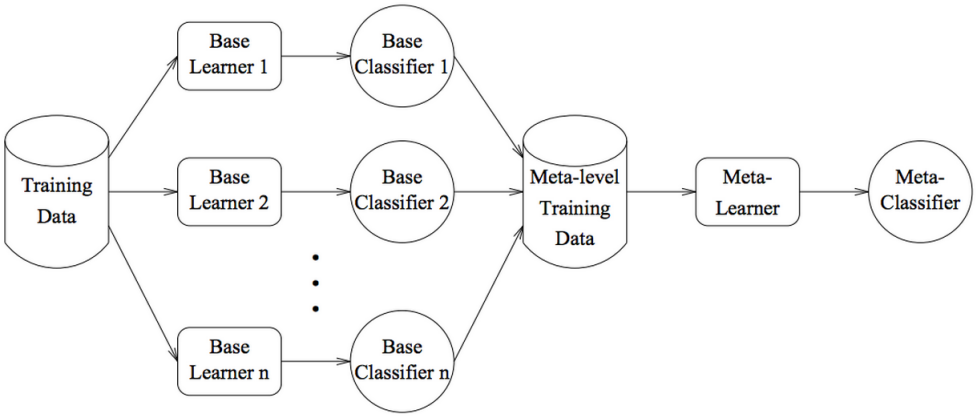


Figura 2.7: Funzionamento dei meta-classificatori per la classificazione

2.3.1 Bagging

Il bagging (o *Boostrap Aggregating*) è una tecnica proposta per la prima volta da Leo Breiman nel 1994 [15], si tratta di un meta-classificatore con

un'idea di base piuttosto semplice che consiste nel raccogliere le previsioni effettuate da diversi classificatori di base, confrontarli e ottenere un'unica previsione.
Come si evince dal nome stesso del meta-classificatore, quest'ultimo utilizza bootstrap[3] come tecnica di campionamento del training set per creare i dataset da cui effettuare l'allenamento dei classificatori di base.
Il bagging è utilizzato per migliorare la stabilità e l'accuratezza della classificazione. È da notare che questa tecnica è più performante se viene utilizzato un learner di base molto suscettibile alle variazioni del training set, mentre, se il learner ha elevata stabilità, la previsione del meta-classificatore potrebbe essere peggiore rispetto alla previsione di un singolo learner.

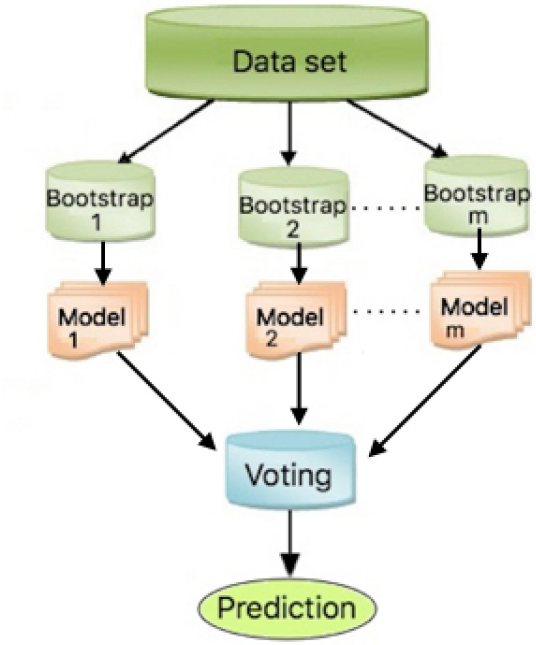


Figura 2.8: Bagging

[3]Il bootstrap è un metodo statistico di ricampionamento che consiste nell'approssimare media e varianza di una distribuzione di campionamento di uno stimatore.

2.3.2 Boosting

Il boosting è un metodo di meta-apprendimento ideato da Robert Schapire nel 1995 [16], si tratta di un meta-classificatore che utilizza il training set per addestrare i classificatori più deboli. Esso, infatti, lavora in modo iterativo. Ad ogni iterazione viene creato un nuovo modello partendo dagli errori commessi durante la classificazione delle iterazioni precedenti e, per una classificazione più efficace, vengono assegnati dei pesi ai campioni di addestramento. È proprio questa la peculiarità di questo meta-classificatore: viene assegnato un peso proporzionale all'importanza di ogni istanza.
Il boosting richiede algoritmi di induzione in grado di gestire le istanze pesate. Qualora l'algoritmo non potesse gestire tali istanze pesate, è comunque possibile ricorrere al boosting replicando le istanze proporzionalmente al peso assegnato. Per fare ciò si procede in questo modo:

1. Viene prodotto un modello con un processo che considera più rilevanti le istanze con peso maggiore.

2. Il modello prodotto viene utilizzato per classificare il training set.

3. Il peso delle istanze classificate correttamente viene ridotto, mentre quello delle istanze errate viene incrementato.

4. Si ripetono i passi da 1 a 3 finché non sarà stato prodotto un determinato numero di modelli.

5. Si assegna ad ogni modello prodotto un peso proporzionale alle sue prestazioni sul training set.

Per la classificazione di nuove istanze, si usa un sistema a votazione pesata (in genere a maggioranza) da parte di tutti i classificatori. Lo scopo di questo algoritmo è di produrre modelli diversi, per coprire un insieme più ampio di tipi di istanze. Il difetto del meta-classificatore consiste nella suscettibilità al rumore. Se ci sono dati errati nel training set, l'algoritmo di boosting tenderà a dare un peso sempre maggiore alle istanze che li contengono, portando

inevitabilmente ad un peggioramento delle prestazioni. Per ovviare a questo problema si è pensato di proporre algoritmi di boosting in grado di ridurre il peso delle istanze che vengono ripetutamente classificate erroneamente.

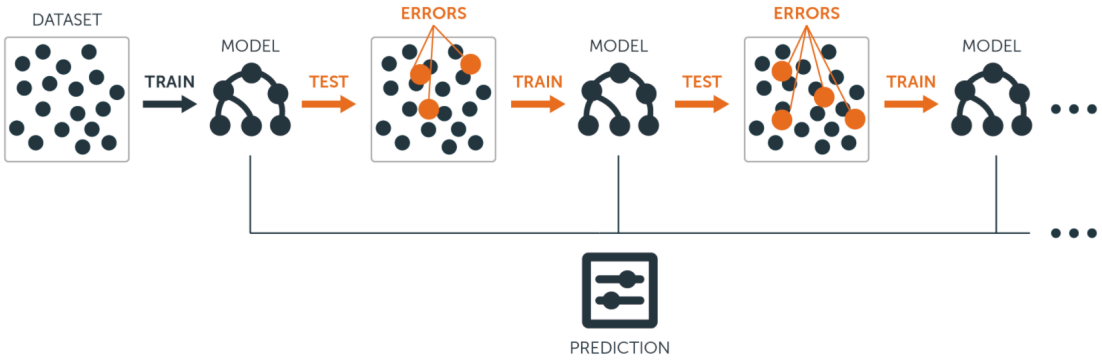


Figura 2.9: Boosting

Capitolo 3

Strumenti

In questa sezione saranno descritte le tecnologie utilizzate per l'implementazione degli script che hanno permesso di effettuare lo studio sperimentale esaminato in questa tesi.

3.1 Python

Python è un linguaggio di programmazione ad alto livello, interattivo e multi-paradigma. Ovvero, il linguaggio supporta sia la programmazione procedurale (che fa uso delle funzioni), sia la programmazione ad oggetti (includendo anche funzionalità come l'ereditarietà singola e multipla). Viene utilizzato per lo sviluppo di applicazioni distribuite, computazione numerica, system testing ed è, inoltre, uno dei linguaggi di programmazione più utilizzati per il data mining e machine learning.

Figura 3.1: Il logo di Python

Ciò che rende Python uno dei linguaggi più potenti e comodi da usare è la libreria built-in estremamente ricca, unitamente alla gestione automatica della memoria e ai robusti costrutti per la gestione delle eccezioni.
Python è nato per essere un linguaggio immediatamente intuibile, la sua sintassi è pulita e snella come i suoi costrutti, decisamente chiari e non ambigui. La particolarità di questo linguaggio di programmazione sta nella gestione dei blocchi logici, essi vengono costruiti allineando semplicemente le righe allo stesso modo, questo ne incrementa la leggibilità e l'uniformità del codice anche se vi lavorano diversi autori [17].
Python è un linguaggio *pseudocompilato*, ciò significa che un interprete si occupa di analizzare il codice sorgente e, se corretto, lo esegue. A differenza di altri linguaggi, in Python, non esiste una fase di compilazione separata e perciò non viene generato nessun file eseguibile partendo dal sorgente.
L'essere pseudointerpretato rende Python un linguaggio *portabile*. Una volta scritto un sorgente, esso può essere interpretato ed eseguito sulla maggior parte delle piattaforme attualmente utilizzate, in quanto è sufficiente la presenza della versione corretta dell'interprete.

Le strutture dati di base che Python fornisce sono riportate nella seguente tabella:

Tipo di struttura dati	Delimitato da
Tuple	( )
Liste	[ ]
Set	{ }
Dizionari	{ }
Oggetti	
Stringhe	virgolette

Tabella 3.1: Principali strutture dati in Python

Di seguito sono brevemente illustrate una per una.

Tuple

Le tuple fanno parte della categoria delle sequenze (come anche le stringhe e le liste). Le sequenze sono oggetti iterabili che rappresentano dei contenitori di lunghezza arbitraria.
Le tuple sono sequenze di oggetti eterogenei e immutabili, e sono identificate dalle parentesi tonde. Il fatto che siano immutabili significa che una volta che è stata creata una data tupla, essa non può più essere modificata, ovvero non si possono sostituire gli elementi che la compongono con altri elementi. In compenso esse sono molto efficienti per quanto riguarda il consumo di memoria e il tempo di esecuzione.
La sintassi:

$$myTuple = (\text{“}a\text{”}, 2, 3, 4, 5, \text{“}test\text{”}, 20.75)$$

Le operazioni permesse alle sequenze sono:

1. Indicizzazione e slicing[1]
2. Concatenazione e ripetizione

Liste

Le liste in Python sono elenchi di elementi di vario tipo. Sono simili alle tuple, con la differenza che sono mutabili e che è possibile aggiungere o eliminare degli elementi da una lista. Per creare una lista inseriamo i suoi elementi tra parentesi quadre, separate da una virgola:

$$myList = [\text{“}one\text{”}, 25, True]$$

Le operazioni permesse alle liste sono:

- append(): aggiunge nuovi elementi alla lista;
- clear(): rimuove tutti gli elementi di una lista;
- copy(): crea una copia della lista;

[1]Estrazione degli elementi

- extend(): permette di unire due liste;
- insert(): aggiunge un elemento in una posizione specifica della lista;
- pop(): rimuove un elemento della lista;
- remove(): rimuove un elemento da una posizione specifica della lista.

Dizionari

Un'altra struttura dati in Python sono i dizionari. Sono dei contenitori che al proprio interno contengono delle coppie *chiave - valore*. Si distinguono per l'uso delle parentesi graffe e dei due punti, e sono oggetti mutabili ma non ordinabili, quindi non è possibile estrarre degli elementi da un dizionario come abbiamo fatto ad esempio con le liste e le tuple.
La sintassi:

$$myDictionary = \{`Marketing' : 28, `Statistics' : 29, `Analysis' : 30\}$$

Le operazioni consentite sono la creazione di un dizionario vuoto, l'aggiunta e la rimozione degli elementi, l'ordinamento e la ricerca.
Una delle proprietà dei dizionari è il *nesting*, che permette di inserire un dizionario all'interno di un altro dizionario.

Set

Un'altra delle strutture di Python sono i set (o insiemi). I set sono dei contenitori di elementi non ordinati e senza duplicati. Sono un tipo di struttura immutabile e supportano le operazioni tipiche degli insiemi, quali unione, intersezione e differenza.
La sintassi:

$$myFirstSet = \{2, 5, 7, 9, 15\}$$

$$mySecondSet = \{`Zeus', `Ade', `Poseidone'\}$$

Le operazioni consentite sono la creazione di un set vuoto, l'aggiunta e la rimozione degli elementi, la ricerca e la verifica che un set sia composto da

elementi unici. Utilizzano i set si possono rimuovere i duplicati in una lista, è sufficiente convertire quest'ultima in set e quindi riconvertirla in lista.
Come precedentemente detto, i metodi che si possono applicare sono le operazioni di unione (OR), intersezione (AND), differenza (NOT) ma anche differenza simmetrica (XOR) che permette di verificare quali elementi appartengono solo ad uno degli insiemi di partenza.

Stringhe

Le stringhe sono qualitativamente diverse dagli altri tipi di dati poiché sono composte di unità più piccole: i caratteri. Una stringa, infatti, può essere vista come un'array di caratteri, dove ogni carattere è identificato da un indice che ne indica la posizione.
In Python ci sono diversi metodi e operazioni utili alla manipolazione delle stringhe, come ad esempio si può verificare la lunghezza[2] di una stringa, si possono confrontare due stringhe per vedere se sono uguali e, data una stringa, si può ottenere una sottostringa di essa.
È da notare che le stringhe sono immutabili, ciò significa che non si può sostituire alcun carattere della stringa.

Oggetti

Tutte le strutture dati precedentemente viste sono oggetti standard, Python però permette anche di creare oggetti customizzati attraverso la definizione di un prototipo dell'oggetto che si vuole strutturare. Questo può essere fatto mediante la creazione di una classe e la definizione degli attributi[3] che caratterizzano ogni oggetto di quella classe.
L'oggetto, pertanto, non è altro che un'istanza di una struttura dati che è definita dalla sua classe. Un oggetto comprende le variabili d'istanza, di classe e i metodi.

[2]La lunghezza di una stringa equivale al numero di caratteri da cui è composta.
[3]Attributi inteso come variabili o metodi (funzioni).

3.2 Scikit-learn

Scikit-learn è una libreria open source di Python che fornisce molteplici strumenti per il *machine learning*, in particolare contiene numerosi algoritmi di classificazione, regressione e clustering (raggruppamento), macchine a vettori di supporto, regressione logistica, e molto altro. Dal momento in cui è stata rilasciata, nel 2007, Scikit-learn è diventata una delle librerie più utilizzate nell'ambito dell'apprendimento automatico, sia supervisionato che non, grazie alla vasta gamma di strumenti che offre, ma anche grazie alla sua API ben documentata, facile da usare e versatile.

Figura 3.2: Il logo di Scikit-learn

L'API Scikit-learn combina un'interfaccia utente funzionale ad un'implementazione ottimizzata di numerosi algoritmi di classificazione e meta-classificazione, oltre a fornire una grande varietà di funzioni di pre-elaborazione dei dati, validazione incrociata, ottimizzazione e valutazione dei modelli.
Scikit-learn è stata concepita come estensione della libreria SciPy (Scientific Python) oltre ad essere stata costruita sulla base delle librerie NumPy e matplotlib di Python.
Numpy estende Python per supportare efficientemente le operazioni su array di grandi dimensioni e matrici multidimensionali, mentre matplotlib fornisce strumenti di visualizzazione e Scipy fornisce moduli per il calcolo scientifico.

Scikit-learn è particolarmente popolare per la ricerca accademica poiché gli sviluppatori possono utilizzare il tool per sperimentare diversi algoritmi modificando solo alcune righe del codice.

In questa tesi sono stati utilizzati diversi classificatori. Di seguito, viene fornito un breve esempio del funzionamento del classificatore SVM per il calcolo della disgiunzione esclusiva (XOR).
Per prima cosa bisogna importare i moduli:

```
> from sklearn import metrics
> from sklearn import svm
```

In seguito si definisce quale classificatore si intende utilizzare:

```
> clf = SVC(kernel="linear", C=0.025)
```

Per semplicità il dataset usato per il training è stato diviso da quello che verrà utilizzato per il test:

```
> X_train = [[0,0],[0,1],[1,0]]
> y_train = [0,1,0]
> X_test = [1,1]
> y_test = [0]
```

Infine il metodo *fit* viene utilizzato per apprendere i parametri dai dati di addestramento. Gli stimatori hanno un metodo *predict* (ma possono anche avere un metodo *trasform*) per eseguire le previsioni sui nuovi campioni:

```
> clf.fit(X_train, y_train)
> preds = clf.predict(X_test)
> print("Valore predetto: ", preds)
> accuracy = metrics.accuracy_score(y_test, preds, normalize=True)
> print ("Accuratezza: ", accuracy)
```

L'output:

```
Valore predetto:  [0]
Accuratezza:  1.0
```

È da notare che l'accuratezza viene calcolata sull'intervallo [0, 1] perciò per questo esempio si è ottenuta la migliore performance possibile.

Visto che Scikit-learn mette a disposizione molti modelli diversi di classificazione, attraverso un semplice ciclo che itera su una lista di oggetti/classificatori, si può pensare di computare facilmente tutti modelli di classificazione:

```
> names = ["Nearest Neighbors", "Linear SVM", "Decision Tree",
          "Neural Net", "Naive Bayes"]

> classifiers = [
    KNeighborsClassifier(3),
    SVC(kernel="linear", C=0.025),
    DecisionTreeClassifier(max_depth=5),
    MLPClassifier(alpha=1),
    GaussianNB()]

> for (name, classifier) in zip(names, classifiers):
    # Compute predictions
    clf.fit(X_train, y_train)
    preds = clf.predict(X_test)
    accuracy = metrics.accuracy_score(y_test, preds, normalize=True)
    print(name, " =>  Accuracy: ", accuracy)
```

Il cui output:

```
Nearest Neighbors  =>  Accuracy:  1.0
Linear SVM  =>  Accuracy:  1.0
Decision Tree  =>  Accuracy:  1.0
Neural Net  =>  Accuracy:  1.0
Naive Bayes  =>  Accuracy:  1.0
```

Sono numerose le applicazioni che utilizzano la libreria Scikit-learn per la loro implementazione totale o parziale:

Capitolo 4

Metodologia

In questa tesi sono stati decodificati, attraverso diversi algoritmi di classificazione, i dati corrispondenti a scariche bioelettriche di neuroni provenienti da primati non umani. I dati sono stati raccolti dal gruppo di ricerca della professoressa Patrizia Fattori nel Dipartimento di Farmacia e Biotecnologie dell'Università di Bologna. L'obiettivo che si è cercato di raggiungere è quello di stabilire quale classificatore lavora meglio per il task rappresentato in figura 4.1 e descritto in dettaglio nella sezione successiva.

4.1 Task

Per lo svolgimento di questo task, è stata addestrata una cavia non umana il cui compito consisteva nell'afferrare gli oggetti proposti, uno alla volta, in ordine casuale. Le tipologie di oggetti proposti sono stati scelti per indurre diverse prese:

1. Palla → prensione con l'intera mano;
2. Maniglia → prensione con tutte le dita eccetto il pollice;
3. Anello → prensione a uncino con l'indice;
4. Piastra → prensione semplice usando il pollice contrapposto alle falangi delle altre dita;

5. Cilindro nella fessura → prensione avanzata con pollice e indice.

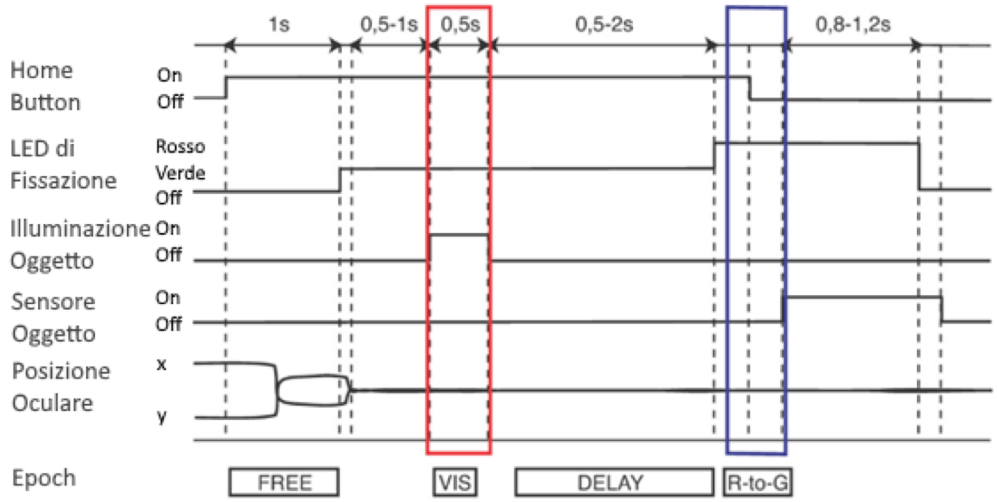


Figura 4.1: Schema del task

La cavia è posta di fronte ad un pannello rotante, ad una distanza di 22,5 cm. Su questo pannello viene proposto, per ogni *trial*, uno dei 5 oggetti precedentemente elencati. Il task inizia al buio con la pressione di un bottone, l'*home button*, posto vicino al torso della cavia. È da questo momento che parte il conteggio del cosiddetto *epoch FREE* (così chiamato in quanto non è richiesta la fissazione di alcun punto in particolare o l'esecuzione di altre operazioni). Segue l'accensione di un LED verde posto sopra l'oggetto, per una durata di 0.5-1s. Se la cavia interrompe la fissazione, il trial viene bloccato e scartato, diversamente si procede con l'illuminazione dell'oggetto per una durata di 0.5s, questo periodo, definito *epoch OBJ-VIS*, è l'unico alla luce dell'intero task e serve per segnalare alla cavia l'oggetto da afferrare nell'azione successiva. Dopo 0.5-1.5s di attesa, periodo definito *epoch DELAY*, si accende un LED rosso che sta ad indicare alla cavia che può procedere ad effettuare la presa e tirare l'oggetto (periodo definito *epoch REACH TO GRASP, R-to-G*).

La presa deve essere mantenuta fino allo spegnimento del LED rosso, segnale che indica la terminazione del trial. Ogni oggetto è stato proposto 10 volte in maniera casuale, per un totale di 50 trials per registrazione.

4.2 Decodifica dei dati neurali

La decodifica dei dati neurali, o *neural decoding*, ha come obiettivo la ricostruzione dell'informazione relativa all'attività di una popolazione di neuroni. L'approccio classico sfrutta algoritmi di apprendimento e predizione su un set di dati descrittivi per l'attività neurale. Il problema dell'apprendimento consiste nel trovare la funzione che mappi meglio input e output in modo predittivo, così che la funzione appresa possa essere usata per predire l'output di qualunque futuro input.
Un classificatore, dato un set di dati di apprendimento, ognuno marcato come appartenente ad una categoria, costruisce un modello che predice a quale categoria appartiene un nuovo esempio, come visto per l'apprendimento supervisionato nel primo capitolo.
Nel caso specifico il set di dati corrisponde alle frequenze di scarica della popolazione di neuroni analizzata (*fr_neuron*) e le categorie corrispondono agli oggetti visti precedentemente (*object*) per ogni tipo di presa effettuata (*condition*). Il set di dati neurali è così organizzato:

$Trial\_1\_Condition\_1$	fr_neuron_1	...	fr_neuron_n	$object\_1$
$Trial\_2\_Condition\_1$	fr_neuron_1	...	fr_neuron_n	$object\_1$
$Trial\_k\_Condition\_1$	...	...	...	$object\_1$
...	...	...	...	...
$Trial\_1\_Condition\_j$	...	...	...	$object\_i$
$Trial\_2\_Condition\_j$	...	...	...	$object\_i$
$Trial\_k\_Condition\_j$	...	...	...	$object\_i$

Tabella 4.1: Dataset descrittivo per l'attività di popolazione in k trials

La classificazione dovrebbe quindi generare un modello che, a partire dai dati già noti, possa essere usato per stabilire a quale oggetto corrispondono nuove rilevazioni dei potenziali d'azione.

4.3 Estrazione delle feature

La costruzione del dataset di input rappresenta per gli algoritmi di classificazione un passaggio chiave che influenza fortemente l'apprendimento e determina l'accuratezza della previsione. L'obiettivo è fornire al classificatore dei dati descrittivi per il fenomeno analizzato, evitando la ridondanza dell'informazione. Di seguito sono riportate le feature che sono state implementate in questa tesi:

1. La feature più semplice con cui è possibile descrivere l'attività di un neurone è la sua frequenza di scarica media, calcolata come *numero di spikes*/intervallo temporale preso in esame.
   Se l'informazione viene registrata a livello di popolazione, è possibile costruire un modello aggregando le frequenze di scarica estratte per ciascun trial.
   L'organizzazione dei dati avviene tramite un *feature vector* così strutturato:
   $\boldsymbol{x} = x_1, \ldots, x_n$ che corrisponde a (fire_rate_1 fire_rate_2 ... fire_rate_n) che rappresenta l'attività di una popolazione in un determinato intervallo, per un dato trial.

2. Un'altra feature di facile implementazione è rappresentata dal *numero di spikes* per intervalli.
   Dopo aver scelto un numero di intervalli in cui dividere il *feature vector*, si memorizza la frequenza di scarica di ciascun neurone nei diversi intervalli.
   Per ogni trial k, condition j, il feature vector associato è il seguente:

Neurone 1	Neurone n
nr_fr_intervallo_1	nr_fr_intervallo_1
...	...
nr_fr_intervallo_n	nr_fr_intervallo_n

Tabella 4.2: Struttura del feature vector per intervalli

3. Per i problemi di classificazione si può adottare un approccio statistico per la costruzione del *feature vector*: in questa tesi, per l'implementazione della feature, è stato utilizzato il valore minimo, la media e il massimo di ogni *spike*.
   Sono state realizzate 2 versioni di questa feature, la prima mantenendo tutti i valori (zero compresi) e la seconda riducendo orizzontalmente il set di dati, eliminando le colonne contenenti valori nulli derivanti dai *feature vectors* vuoti.

4. È possibile aggregare varie feature per ottenere un dataset il più descrittivo possibile, concatenando orizzontalmente i diversi metodi di estrazione. In questa tesi sono stati combinati i metodi descritti nel punto (1) e nel punto (3) ottenendo così un *feature vector* formato da valore minimo, media, valore massimo e dimensione di ogni *spike*.
   Anche in questo caso sono state realizzate 2 versioni di questa feature, la prima mantenendo tutti i valori (zero compresi) e la seconda riducendo orizzontalmente il set di dati, eliminando le colonne contenenti valori nulli derivanti dai *feature vectors* vuoti.

È da notare che in termini di performance, i classificatori, danno risultati peggiori quando la lunghezza dei *feature vectors* supera la dimensione delle feature (numero di istanze). Ne consegue perciò che il numero di registrazioni effettuate influenzino notevolmente il risultato della previsione del classificatore.

La seguente tabella riassume le feature implementate:

Feature	**Descrizione**
Firing rate	Frequenza di scarica media di ciascuno spike
Firing rate per intervalli	Frequenza di scarica di ciascuno spike, per intervallo
Minimo, media e massimo	Minimo, media e massimo di ciascuno spike
Minimo, media, massimo e firing rate	Minimo, media, massimo e frequenza di scarica di ciascuno spike

Tabella 4.3: Feature implementate con relativa descrizione

4.4 Preparazione dei dati

Per una classificazione più accurata si è reso necessario pre-processare i dati, in modo da eliminare:

Rumori: errori e/o outliers

Dati incompleti: attributi non valorizzati, mancanza di attributi d'interesse per gli scopi dell'analisi, presenza di sole grandezze sommarizzate

Dati inconsistenti: discrepanze nei nomi e nei codici degli attributi.

Il processo di mining necessita di dati consistenti per garantire una previsione di qualità.
La fase di preparazione dei dati richiede il $70-80\%$ del tempo e dello sforzo complessivo del processo di KDD[1].
Le fasi principali del *pre-processing* sono:

[1]Knowledge Discovery in Databases

- data cleaning: eliminazione dei valori che possono distorcere le informazioni sui dati (ad esempio gli *outliers* o i dati duplicati);

- data integration: combina i dati provenienti da diverse sorgenti in un unico ambiente coerente. Un attento processo di integrazione dei dati deve prestare particolare attenzione alle ridondanze che si creano integrando i dati derivanti da diverse fonti;

- data transformation: consiste nella *normalizzazione* dei dati.
  In questa tesi sono state trattate 4 tipologie di trasformazione dei dati:

  1. *Rescale*: sostituisce i valori dei potenziali d'azione di ciascuno spike con un valore compreso nel range fra 0 e 1.
     La formula applicata è la seguente:

     $$x' = \frac{x - min(x)}{max(x) - min(x)}$$

     dove x è il potenziale d'azione e $x'(0 \leq x' \leq 1)$ è il valore risultante dalla normalizzazione dei dati.

  2. *Standardization*: lavora sui dati relativi agli attributi, assegnando dei valori la cui media sarà 0 e la deviazione standard sarà 1. La formula applicata è la seguente:

     $$x' = \frac{x - \overline{x}}{\sigma}$$

     dove $\overline{x}$ è la media e σ è la deviazione standard.

  3. *Normalization* sostituisce ciascun dato degli spike con un valore la cui lunghezza è massimo 1. La formula applicata è la seguente:

     $$x' = \frac{x}{\|x\|}$$

     dove $\|x\|$ indica la norma.

  4. *Binarization*: sostituisce ciascun valore minore o uguale a zero con 0 e ciascun valore maggiore di zero con 1. I risultati di questo

tipo di trasformazione non sono performanti per il caso di studio esaminato, pertanto non verrà preso in considerazione questo tipo di trasformazione dei dati.

- data reduction: comporta una riduzione dimensionale del dataset in quanto terabytes di dati possono richiedere tempi di elaborazione elevati. Con questa tecnica si ha una perdita minima del contenuto informativo iniziale poiché l'obiettivo è di ridurre la rappresentazione del set di dati, tale però da produrre gli stessi (o quasi) risultati analitici;

- data discretization: è utilizzata per variabili di tipo numerico. Riduce il numero di valori per un attributo continuo dividendo il range dei valori dell'attributo in intervalli. Le etichette (*label*) degli intervalli sostituiscono i valori effettivi dell'attributo. Ad esempio, si possono sostituire i valori numerici relativi all'età con concetti più generali come 'giovane', 'media età', 'anziano'.

4.5 Training e cross-validazione

L'apprendimento è il processo cardine del data mining, esso si occupa di generalizzare e ricercare pattern partendo dai dati che descrivono l'evento oggetto di studio che si vuole analizzare. Di solito un algoritmo di apprendimento viene allenato usando un certo insieme di esempi (*feature vectors*) presi a caso dalla popolazione. Si assume che l'algoritmo di apprendimento (il *learner*) raggiungerà uno stato in cui sarà in grado di effettuare una classificazione e predire gli output per tutti gli altri esempi che ancora non ha visionato, cioè, come già detto nel secondo capitolo, si assume che il modello di apprendimento sarà in grado di generalizzare. Per fare una stima degli errori e verificare quindi l'accuratezza del classificatore, si divide l'insieme dei campioni in due parti: una di queste, detta *training set*, viene utilizzata per costruire il modello mentre l'altra, il *test set*, viene utilizzata per la successiva verifica. Le due partizioni devono essere estratte casualmente per

poter ottenere campioni il più possibile rappresentativi dell'insieme; inoltre i casi del test set devono essere indipendenti dai casi di training, ovvero l'unica relazione fra i due gruppi deve essere l'appartenenza alla stessa popolazione. Tipicamente l'insieme dei campioni viene diviso in $\frac{2}{3}$ per il training set e $\frac{1}{3}$ per il test set.

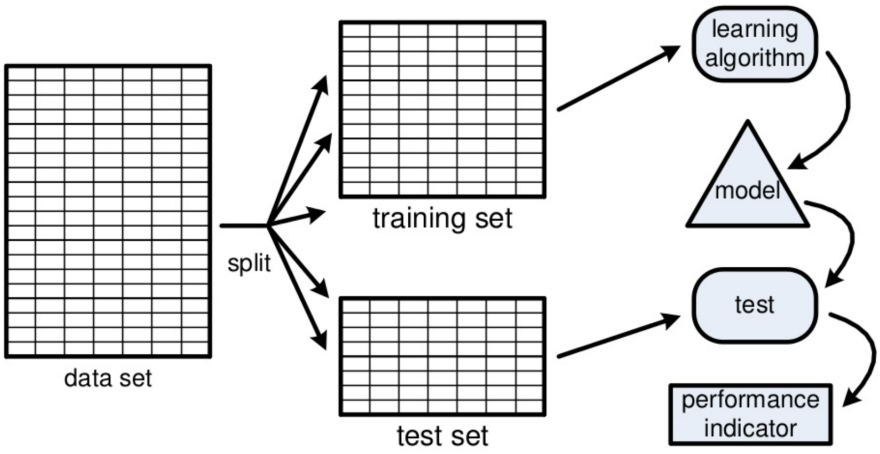


Figura 4.2: Suddivisione del dataset

Tuttavia, soprattutto nei casi in cui l'apprendimento è stato effettuato su uno scarso numero di esempi di allenamento, il modello potrebbe adattarsi a caratteristiche che sono specifiche solo del training set, ma che non hanno riscontro nel resto dei casi; perciò, in presenza di *overfitting*[2], le prestazioni (cioè la capacità di adattasi/prevedere) sui dati di allenamento aumenteranno, mentre le prestazioni sui dati non visionati saranno peggiori. Per evitare l'overfitting, è necessario attuare la tecnica della cross-validazione.

[2]Si parla di *overfitting* quando un modello statistico si adatta ai dati osservati (il campione) usando un numero eccessivo di parametri.

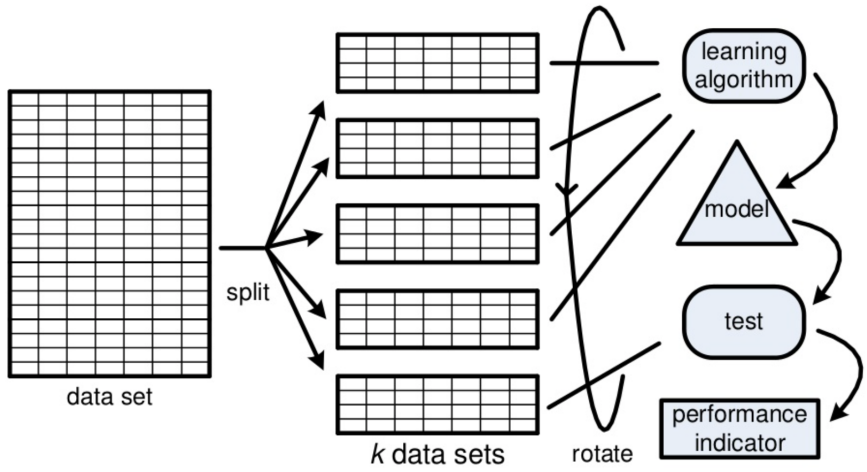


Figura 4.3: Tecnica della cross-validazione

Con la tecnica della cross-validazione (o validazione incrociata) si divide l'insieme dei campioni in K partizioni delle stesse dimensioni, indipendenti tra loro . Ad ogni ripetizione della procedura una partizione viene utilizzata per il test e le rimanenti per il training, con cui si costruisce il modello. Si ottengono quindi N stime del tasso di errore dei campioni di test: il valore medio di questi è chiamato *tasso* di errore *cross-validato* e fornisce una stima del tasso di errore reale molto attendibile.

Nei casi in cui c'è un ridotto numero di osservazioni (pochi *trials*), ossia la scarsa quantità dei dati da partizionare renda critica la dimensione del set di apprendimento, la cross-validazione permette di massimizzare il processo di creazione dei dataset. Con l'approccio "Leave-One-Out" *n-1* trials vengono usati per l'apprendimento, ne segue che la predizione viene fatta sull'unico trial escluso. Il processo viene iterato n volte, cambiando ogni volta il trial escluso così da saggiare tutti i trials.

Capitolo 5

Implementazione

In questa sezione verrà esaminato più da vicino il caso di studio oggetto di questa tesi.

5.1 Dataset

I dataset sui quali è stato condotto l'esperimento di classificazione sono stati ottenuti dalle registrazioni dell'attività extracellulare di singole cellule, durante le sessioni di test condotte su due esemplari maschi di macaca fascicularis. Le registrazioni sono state effettuate sia in condizioni di luce che di buio, di seguito sono riportati i datasets con relativi intervalli di interesse (attività dei neuroni allineata allo stimolo) e condizioni visive (luce o buio):

Dataset	Light	Dark
KeyUp	keyupLIGHT[-2.5s; +1s]	keyupDARK[-2.5s; +1s]
LuceOn	luceOnLIGHT[-1s; +2.5s]	luceOnDARK[-1s; +2.5s]

Tabella 5.1: Datasets

Ciascun dataset conta 395 record, ognuno dei quali descrive una registrazione. Di seguito è raffigurato un prototipo di dataset:

[elettrodo, unit, id, neurone 1, oggetto 1]	...	...	[T1 [...], ..., T10 [...]]
...	...	...	...
[elettrodo, unit, id, neurone 79, oggetto 1]	...	...	[T1 [...], ..., T10 [...]]
...	...	...	...
...	...	...	...
[elettrodo, unit, id, neurone 1, oggetto 5]	...	...	[T1 [...], ..., T10 [...]]
...	...	...	...
[elettrodo, unit, id, neurone 79, oggetto 5]	...	...	[T1 [...], ..., T10 [...]]

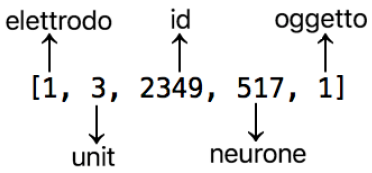
Il set di dati che verrà utilizzato per la classificazione è stato così costruito:

Trial	neurone 1	...	neurone 79	oggetto
T1 [...]				1
...				1
T10 [...]				1
...				...
...				...
T1 [...]				5
...				5
T10 [...]				5

Il codice per ottenere le informazioni relative alla registrazione è:

```
infoTrial = file[0][0]
print(infoTrial)
```

Output:



Ogni registrazione è stata ripetuta 10 volte, per ottenere i 10 array (*trial*) contenenti i potenziali d'azione in millisecondi basti eseguire il codice:

```
for i in range(0,10):
    print("TRIAL ", i+1)
    spike = file[0][3][i]
    print(spike)
```

Output:

```
TRIAL  1
[5.56, 31.7, 33.34, 58.14, 68.77, 96.34, 130.73, 131.86, 144.46, 154.44, 185.19, 207.64, 270.26, 296.13,
321.39, 324.25, 351.9, 380.65, 391.47, 414.32, 416.1, 430.64, 459.79, 484.67, 503.09, 508.89, 525.2,
553.95, 573.29, 579.15, 617.27, 621.86, 627.59, 650.28, 651.03, 671.82, 678.13, 703.46, 735.28, 741.54,
765.41, 815.11, 823.51, 838.82, 887.17, 907.38, 911.44, 982.2, 1072.16, 1102.94, 1155.82, 1214.67,
1230.59, 1259.28, 1271.19, 1327.66, 1407.61, 1428.98, 1503.32, 1512.13, 1563.74, 1599.28, 1604.9, 1621.19,
1650.7, 1660.07, 1690.63, 1693.78, 1719.12, 1738.36, 1756.18, 1757.82, 1821.73, 1878.8, 1882.74, 1954.92,
1989.31, 1993.57, 1996.92, 2034.05, 2037.11, 2061.44, 2062.5, 2067.71, 2129.33, 2306.24, 2318.43, 2341.78,
2356.71, 2362.22, 2405.18, 2407.78, 2412.61, 2423.74, 2433.87, 2451.48, 2462.82, 2465.76, 2471.9, 2475.0,
2497.45, 2508.03, 2510.47, 2511.64, 2520.53, 2537.15, 2547.04, 2566.53, 2613.81, 2641.91, 2647.75,
2693.66, 2729.19, 2746.42, 2793.75, 2806.01, 2822.19, 2860.66, 2881.98, 2887.33, 2956.74, 2982.61,
3021.11, 3054.85, 3069.24, 3089.92, 3165.33, 3167.83, 3190.58, 3207.25, 3279.83, 3303.32, 3309.61,
3335.18, 3339.46, 3384.99, 3421.0, 3466.57, 3485.63, 3489.6]
TRIAL  2
[17.61, 40.61, 81.82, 99.48, 127.96, 203.6, 265.78, 279.76, 327.43, 338.9, 392.18, 394.39, 429.98, 458.47,
558.97, 563.76, 616.61, 640.19, 673.35, 748.11, 829.25, 1125.55, 1136.11, 1247.88, 1266.99, 1297.84,
1340.59, 1349.27, 1389.53, 1407.03, 1420.95, 1457.59, 1466.86, 1481.75, 1491.37, 1498.58, 1501.55, 1513.4,
1544.01, 1547.99, 1566.99, 1584.42, 1606.34, 1612.8, 1641.15, 1647.45, 1652.67, 1666.84, 1685.28, 1699.96,
1712.15, 1718.14, 1719.7, 1749.55, 1750.54, 1772.02, 1795.84, 1817.66, 1833.61, 1839.13, 1864.41, 1870.41,
1873.57, 1894.08, 1909.93, 1952.34, 1958.77, 1966.86, 1998.14, 2000.55, 2033.11, 2045.06, 2046.25,
2068.02, 2091.36, 2113.58, 2140.53, 2147.72, 2163.82, 2180.63, 2188.69, 2200.91, 2208.42, 2231.02,
2259.88, 2276.19, 2297.58, 2309.11, 2310.52, 2338.59, 2353.32, 2357.67, 2366.02, 2384.5, 2388.35, 2414.68,
2430.46, 2475.4, 2497.9, 2500.63, 2517.18, 2546.48, 2563.02, 2582.33, 2605.56, 2638.5, 2660.27, 2679.58,
2713.7, 2723.76, 2774.3, 2782.5, 2808.19, 2828.64, 2836.4, 2881.16, 2912.73, 2976.54, 3055.46, 3237.67,
3248.9, 3342.9, 3375.66, 3432.2, 3455.22, 3480.95]
```

...

```
TRIAL  10
[8.18, 56.3, 61.82, 82.04, 82.83, 98.82, 108.94, 149.35, 177.36, 227.56, 229.57, 296.22, 318.7, 338.19,
360.16, 388.05, 391.72, 399.82, 423.39, 424.62, 431.01, 443.99, 475.82, 481.74, 483.71, 497.94, 523.81,
536.2, 544.31, 561.49, 587.67, 600.83, 624.28, 685.19, 782.89, 832.41, 899.91, 901.05, 937.52, 951.05,
994.16, 997.51, 1035.39, 1100.18, 1113.86, 1130.07, 1167.8, 1190.24, 1260.25, 1267.29, 1279.75, 1306.65,
1321.82, 1326.97, 1359.3, 1375.48, 1382.38, 1409.39, 1423.04, 1428.85, 1444.52, 1479.36, 1489.31, 1516.71,
1531.04, 1554.3, 1588.38, 1608.95, 1619.55, 1642.04, 1669.13, 1692.28, 1727.08, 1728.34, 1738.33, 1755.25,
1762.92, 1789.68, 1809.01, 1821.0, 1853.05, 1888.44, 1902.47, 1908.03, 1932.65, 1945.48, 1961.5, 1968.61,
2014.22, 2025.68, 2028.23, 2072.84, 2085.28, 2105.09, 2107.4, 2141.64, 2173.3, 2177.57, 2197.15, 2199.72,
2244.55, 2257.44, 2288.58, 2314.03, 2360.73, 2388.43, 2414.5, 2434.53, 2450.16, 2478.47, 2492.54, 2502.85,
2512.2, 2537.45, 2556.73, 2572.08, 2586.86, 2605.86, 2631.73, 2651.97, 2685.18, 2709.56, 2731.51, 2787.92,
2793.81, 2810.67, 2842.14, 2879.22, 2920.17, 2978.01, 3031.26, 3045.82, 3085.87, 3118.85, 3157.28,
3193.31, 3248.5, 3281.83, 3290.96, 3319.91, 3362.24, 3480.37]
```

Per realizzare il dataset che sarà usato dal classificatore per l'allenamento e la costruzione del modello, sono stati rappresentati sulle righe i 10 trial (per ognuno dei 5 oggetti) e sulle colonne le feature relative alle scariche di ogni neurone, come mostrato in figura 4.1.
La funzione che permette di implementare il dataset nel modo descritto sopra è la seguente:

```
def setVal(inizio, fine, obj, index):
   for i in range(inizio,fine): ############### oggetto
    for j in range(10):
        vals[j].append(len(file[i,3][j]))

   for k in range(10):
       values=[]
       values=np.array(vals[k]).reshape(1,-1) #riduco dimensione
       array[index,0:79]=values
       array[index,79]=obj
       objs.append(obj)
       index=index+1
   reset(vals)


def reset(vals):
    for v in range(len(vals)):
        vals[v]=[]
```

SetVal prende in input 4 valori: il riferimento alla prima e all'ultima rilevazione relativa ad un certo oggetto, l'oggetto che si sta analizzando e infine index che è usato come variabile di appoggio. *Vals[j]* è un array che memorizza il numero di potenziali d'azione di ognuno dei 10 trial, *values* permette di ridurre di una dimensione l'array. Questo passaggio è fondamentale in quanto i classificatori accettano solo dataframe bidimensionali, per questo motivo non è stato possibile costruire dataset che modellino matrici multidimensionali.

Dopo l'invocazione della funzione *setVal*, il set di dati che viene creato sarà composto da 50 istanze (righe, 10 trial * 5 oggetti) e 79 colonne corrispondenti al numero di impulsi rilasciati da ciascun neurone analizzato, più un'ultima colonna corrispondente all'oggetto considerato.

Output:

```
ISTANZA  1
 [ 140.   13.  120.   91.   37.   99.   80.   31.   40.   28.  133.   22.
  143.   44.   55.   76.   31.   46.  120.  142.   70.   73.   15.  145.
   30.  173.  134.  460.  295.   92.   54.   38.   11.   43.   24.  151.
   12.   24.  100.   15.   24.   43.    4.   29.   28.   19.   10.   97.
   90.   33.  202.   72.   17.   17.   30.   32.  492.    9.   49.   81.
    1.  134.  105.   39.   66.   62.   12.   22.    6.    7.   54.   38.
   19.   28.   28.    7.   12.   82.   10.    1.]
ISTANZA  2
 [ 126.   43.  149.   78.   33.   99.   62.   30.   14.   64.  127.   36.
   80.   39.   55.   79.   21.   56.   85.  109.   65.   60.   26.  250.
   19.  163.   74.  319.  336.  104.   90.   41.   19.   21.   60.  149.
   13.   25.   56.   10.   15.   38.   11.   26.   32.   20.   22.   12.
   95.   38.  154.   69.   30.   15.   40.   20.  340.    4.   76.   51.
   16.  119.   48.   24.   80.   55.    9.   12.    0.    3.   37.   31.
   36.   28.   28.    3.    5.   93.   41.    1.]
```

...

```
ISTANZA  50
 [ 159.   60.  165.   54.   23.  344.   59.   40.    9.   51.   47.   62.
   49.   32.   58.  106.   15.   62.   97.   26.   78.   19.   27.  317.
   37.  182.   98.  331.   93.  123.   79.   55.  101.   31.   79.  124.
    8.   28.   31.   24.   21.   41.   10.   61.   39.   30.   12.   45.
   56.   11.   91.   82.   13.    0.   68.   20.  315.   16.   58.   41.
   11.  115.   73.   34.   63.   84.   17.    0.   11.   17.   62.   29.
   27.    8.    8.    2.   19.   50.   60.    5.]
```

5.2 Preprocessing

Per la preparazione dei dati si è deciso di implementare diversi criteri di normalizzazione dei dati, in modo da verificare quale di questi lavora meglio con i valori corrispondenti alle scariche neuronali del dataset oggetto di studio. Si prenda in esempio questo *spike vector*:

```
[83.09, 149.98, 187.24, 251.84, 256.03, 264.62, 329.85, 461.67, 498.89,
557.48, 601.74, 654.72, 774.78, 790.18, 845.71, 872.85, 890.95, 969.81,
1026.4, 1053.9, 1135.89, 1248.67, 1274.83, 1374.59, 1386.86, 1442.24,
1499.42, 1517.0, 1758.92, 1956.07, 1979.49, 2054.92, 2146.9, 2178.84,
2206.91, 2270.2, 2294.01, 2307.63, 2313.58, 2343.8, 2373.31, 2415.29,
2435.31, 2459.26, 2541.99, 2554.86, 2588.95, 2627.24, 2677.85, 2707.31,
2784.62, 2824.04, 2862.26, 2977.18, 3044.8, 3114.6, 3171.37, 3316.69,
3419.37, 3458.25]
```

Attraverso l'invocazione della funzione "rescale_data" è possibile applicare la normalizzazione di tipo *rescale* ai dati dell'array:

```
def rescale_data(classificatore):
    print("####### Rescaled data #######")
    scaler = MinMaxScaler(feature_range=(0, 1))
    rescaledX = scaler.fit_transform(array)
    np.set_printoptions(precision=3)
    colNames=[]
    for i in range(len(array[0])):
        colNames.append(i+1)
    df=pd.DataFrame(rescaledX, columns=colNames)
    df = df.drop(80,1)
    df[80] = pd.Categorical(objs)
    ss = ShuffleSplit(n_splits=5, test_size=0.3, random_state=0)

    for train_index, test_index in ss.split(df):
        process(train_index, test_index, classificatore, df)

    plot_matrix()
```

Se applicata al precedene spike vector si ottiene:

```
####### Rescaled data #######
[ 0.    0.02  0.03  0.05  0.05  0.05  0.07  0.11  0.12  0.14  0.15  0.17
  0.2   0.21  0.23  0.23  0.24  0.26  0.28  0.29  0.31  0.35  0.35  0.38
  0.39  0.4   0.42  0.42  0.5   0.55  0.56  0.58  0.61  0.62  0.63  0.65
  0.66  0.66  0.66  0.67  0.68  0.69  0.7   0.7   0.73  0.73  0.74  0.75
  0.77  0.78  0.8   0.81  0.82  0.86  0.88  0.9   0.92  0.96  0.99  1.  ]
```

Con questo tipo di trasformazione dei dati viene sostituito ciascun dato del dataset con un valore compreso nel range fra 0 e 1.

Attraverso, invece, l'invocazione della funzione "std_scaler_data" è possibile applicare la normalizzazione di tipo *standardized* ai dati dell'array:

```
def std_scaler_data(classificatore):
    print("####### Standardized data #######")
    scaler = StandardScaler().fit(array)
    rescaledX = scaler.transform(array)
    np.set_printoptions(precision=3)
    colNames=[]
    for i in range(len(array[0])):
        colNames.append(i+1)
    df = pd.DataFrame(rescaledX, columns=colNames)
    df = df.drop(80,1)
    df[80] = pd.Categorical(objs)
    ss = ShuffleSplit(n_splits=5, test_size=0.3, random_state=0)

    for train_index, test_index in ss.split(df):
        process(train_index, test_index, classificatore, df)

    plot_matrix()
```

Se applicata al precedene spike vector si ottiene:

```
####### Standardized data #######
[-1.69 -1.62 -1.58 -1.52 -1.51 -1.5  -1.44 -1.3  -1.26 -1.2  -1.16 -1.11
 -0.98 -0.97 -0.91 -0.88 -0.87 -0.79 -0.73 -0.7  -0.62 -0.5  -0.48 -0.37
 -0.36 -0.31 -0.25 -0.23  0.02  0.22  0.24  0.32  0.41  0.44  0.47  0.54
  0.56  0.57  0.58  0.61  0.64  0.68  0.7   0.73  0.81  0.83  0.86  0.9
  0.95  0.98  1.06  1.1   1.14  1.25  1.32  1.39  1.45  1.6   1.7   1.74]
```

Con questo tipo di trasformazione dei dati viene sostituito ciascun dato del dataset con dei valori la cui media sarà 0 e la deviazione standard sarà 1.

Infine, attraverso l'invocazione della funzione "normalized_data" è possibile applicare la normalizzazione di tipo *normalized* ai dati dell'array:

```
def normalized_data(classificatore):
    print("####### Normalized data #######")
    scaler = Normalizer().fit(array)
    normalizedX = scaler.transform(array)
    np.set_printoptions(precision=3)
    colNames=[]
    for i in range(len(array[0])):
        colNames.append(i+1)
    df = pd.DataFrame(normalizedX, columns=colNames)
    df = df.drop(80,1)
    df[80] = pd.Categorical(objs)
    ss = ShuffleSplit(n_splits=5, test_size=0.3, random_state=0)

    for train_index, test_index in ss.split(df):
        process(train_index, test_index, classificatore, df)

    plot_matrix()
```

Se applicata al precedene spike vector si ottiene:

```
####### Normalized data #######
[[ 0.01  0.01  0.01  0.02  0.02  0.02  0.02  0.03  0.03  0.04  0.04  0.04
   0.05  0.05  0.05  0.06  0.06  0.06  0.07  0.07  0.07  0.08  0.08  0.09
   0.09  0.09  0.1   0.1   0.11  0.13  0.13  0.13  0.14  0.14  0.14  0.15
   0.15  0.15  0.15  0.15  0.15  0.16  0.16  0.16  0.16  0.16  0.17  0.17
   0.17  0.17  0.18  0.18  0.18  0.19  0.2   0.2   0.2   0.21  0.22  0.22]]
```

Con questo tipo di trasformazione dei dati viene sostituito ciascun dato del dataset con un valore la cui lunghezza è massimo 1.

Le funzioni "*rescale_data*", "*std_scaler_data*" e "*normalized_data*" implementano anche la partizione del dataset in training e validation set, come spiegato nel paragrafo 5.3.

5.3 Training e validation set

La fase di preprocessing procede con lo split del dataset originale in due dataset: uno di training per la stima del modello e l'altro di validation per la verifica della bontà della classificazione ottenuta. Anche la variabile target viene divisa allo stesso modo. La suddivisione dei record nei due dataset è casuale e in questo caso prevede una divisione tra training e validation set usando il 70% dei record originali per il primo (35 istanze) ed il 30% per il secondo (15 istanze).
Per questo caso di studio, inoltre, è stata implementata la *validazione incrociata K-fold* perché il numero di istanze è basso e si rischierebbe di ottenere il sovradattamento del modello rispetto ai dati.
Nelle funzioni "*rescale_ data*", "*std_ scaler_ data*" e "*normalized_ data*" viste nel paragrafo 5.2 è stata impostata la *cross-validazione* con $K = 5$, la seguente funzione si occupa di costruire il set di dati utilizzati per l'addestramento e la validazione del classificatore utilizzando gli indici relativi alle istanze da classificare che gli vengono passati come parametro di input:

```
def process(train, test, clf, df):

    X_train = np.zeros([35, len(df.columns)-1])
    X_test = np.zeros([15, len(df.columns)-1])
    y_train = np.zeros([35])
    y_test = np.zeros([15])
    for i in range(len(train)):
        X_train[i,:] = df.loc[train[i], 0:len(df.columns)-1]
        y_train[i] = df.loc[train[i], len(df.columns)]
    for i in range(len(test)):
        X_test[i,:] = df.loc[test[i], 0:len(df.columns)-1]
        y_test[i] = df.loc[test[i], len(df.columns)]

    preprocess(X_train, X_test, y_train, y_test, clf)
```

Train e *test* contengono rispettivamente gli indici delle istanze da utilizzare per il training e per il test, clf è il classificatore con cui poi verrà effettuata la classificazione e df è il dataframe oggetto di studio.
La funzione "*process*" richiama "*preprocess*" passando come parametri di in-

put il dataset con cui effettuare il train, quello per il test, le relative etichette (target) ed il classificatore da utilizzare.
La funzione "*preprocess*" è stata così implementata:

```
def preprocess(X_train, X_test, y_train, y_test, classificatore):

    clf = classificatore
    clf.fit(X_train, y_train)
    preds = clf.predict(X_test)
    print("\nreal  -> ",np.array(y_test))
    print("preds -> ",np.array(preds))

    # Compute confusion matrix
    cm = metrics.confusion_matrix(y_test,preds)
    cf_matrix.append(cm)
    np.set_printoptions(precision=2)
    cm_normalized = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
    print('\nNormalized confusion matrix: ')
    print(cm_normalized)

    ac=metrics.accuracy_score(np.array(y_test),np.array(preds), normalize=False)
    accuracy_score = (ac*100)/15
    print("Accuratezza ",accuracy_score)
    accuracy.append(ac)
```

Questa è la funzione che effettua la classificazione e ritorna il valore relativo all'accuratezza dell'algoritmo:

```
real  ->  [ 1.  4.  5.  1.  5.  1.  3.  3.  2.  5.  3.  5.  5.  2.  3.]
preds ->  [ 1.  4.  5.  1.  5.  1.  3.  3.  2.  5.  3.  5.  5.  2.  3.]

Normalized confusion matrix:
[[ 1.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.]
 [ 0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  1.]]
Accuratezza  100.0 %
```

I singoli risultati saranno analizzati più in dettaglio nel prossimo capitolo.

Capitolo 6

Risultati

In questo capitolo verranno analizzati i risultati ottenuti dalla classificazione effettuata sul dataset "*keyupLIGHT*" di cui a tabella 5.1. Sono state valutate le prestazioni in termini di accuratezza relative agli algoritmi di decodifica, al preprocessamento dei dati e alle features implementate.

6.1 Confronto algoritmi

Per la classificazione dei *test* dataset sono stati confrontati 10 fra i diversi classificatori che Scikit-learn mette a disposizione. Questo si può fare con un semplice ciclo iterando su una lista di oggetti/classificatori. Come si evince dai risultati delle accuratezze, alcuni algoritmi sono più adatti di altri per il set di dati che si sta analizzando.
Si noti che non sempre un algoritmo che ha una forte complessità computazionale lavora meglio di un algoritmo più *semplice*; in aggiunta, è da notare anche che ci sono algoritmi più performanti su un numero non elevato di istanze. Per scegliere quale algoritmo utilizzare per la classificazione, oltre al numero di istanze, è utile considerare anche altri parametri quali l'indipendenza fra gli attributi, la possibilità di classificare il problema linearmente, il numero di casistiche per il training e la probabilità di overfitting.

In questa tesi si è cercato di confrontare le performance dei seguenti algoritmi:

Nearest Neighbors
Linear SVM
Radial Basis Function SVM
Gaussian Process
Decision Tree
Random Forest
Neural Networks
AdaBoost
Naive Bayes
Quadratic Discriminant Analysis

Tabella 6.1: Classificatori utilizzati per la costruzione dei modelli, in verde quelli che forniscono una migliore performance per il set di dati considerato.

Per definire la performance di ogni singolo algoritmo è stato implementato uno script che permette di confrontare l'accuratezza relativa ai diversi classificatori:

```
names = ["Nearest Neighbors", "Linear SVM", "RBF SVM", "Gaussian Process",
         "Decision Tree", "Random Forest", "Neural Net", "AdaBoost",
         "Naive Bayes", "QDA"]

classifiers = [
    KNeighborsClassifier(3),
    SVC(kernel="linear", C=0.025),
    SVC(gamma=2, C=1),
    GaussianProcessClassifier(1.0 * RBF(1.0), warm_start=True),
    DecisionTreeClassifier(max_depth=5),
    RandomForestClassifier(max_depth=5, n_estimators=10, max_features=1),
    MLPClassifier(alpha=1),
    AdaBoostClassifier(),
    GaussianNB(),
    QuadraticDiscriminantAnalysis()]
```

La funzione "*classify*" prende come parametri in input i dataset relativi al training e al test con le relative etichette e permette di effettuare la classificazione di tutti i modelli:

```
def classify(X_train, X_test, y_train, y_test):
    for name, clf in zip(names, classifiers):
        print("\nclassificatore:",name)
        clf.fit(X_train, y_train)
        preds = clf.predict(X_test)
        score = clf.score(X_test, y_test)
        print("score",score)
```

Un esempio di output è il seguente:

```
classificatore: Nearest Neighbors
score 1.0

classificatore: Linear SVM
score 0.266666666667

classificatore: RBF SVM
score 0.266666666667

classificatore: Gaussian Process
score 1.0

classificatore: Decision Tree
score 1.0

classificatore: Random Forest
score 0.666666666667

classificatore: Neural Net
score 0.133333333333

classificatore: AdaBoost
score 0.466666666667

classificatore: Naive Bayes
score 1.0

classificatore: QDA
score 0.466666666667
```

Dove *score* indica l'accuratezza calcolata sull'intervallo [0, 1].
Di seguito sono rappresentati tutti i risultati relativi alla performance dei classificatori utilizzati.

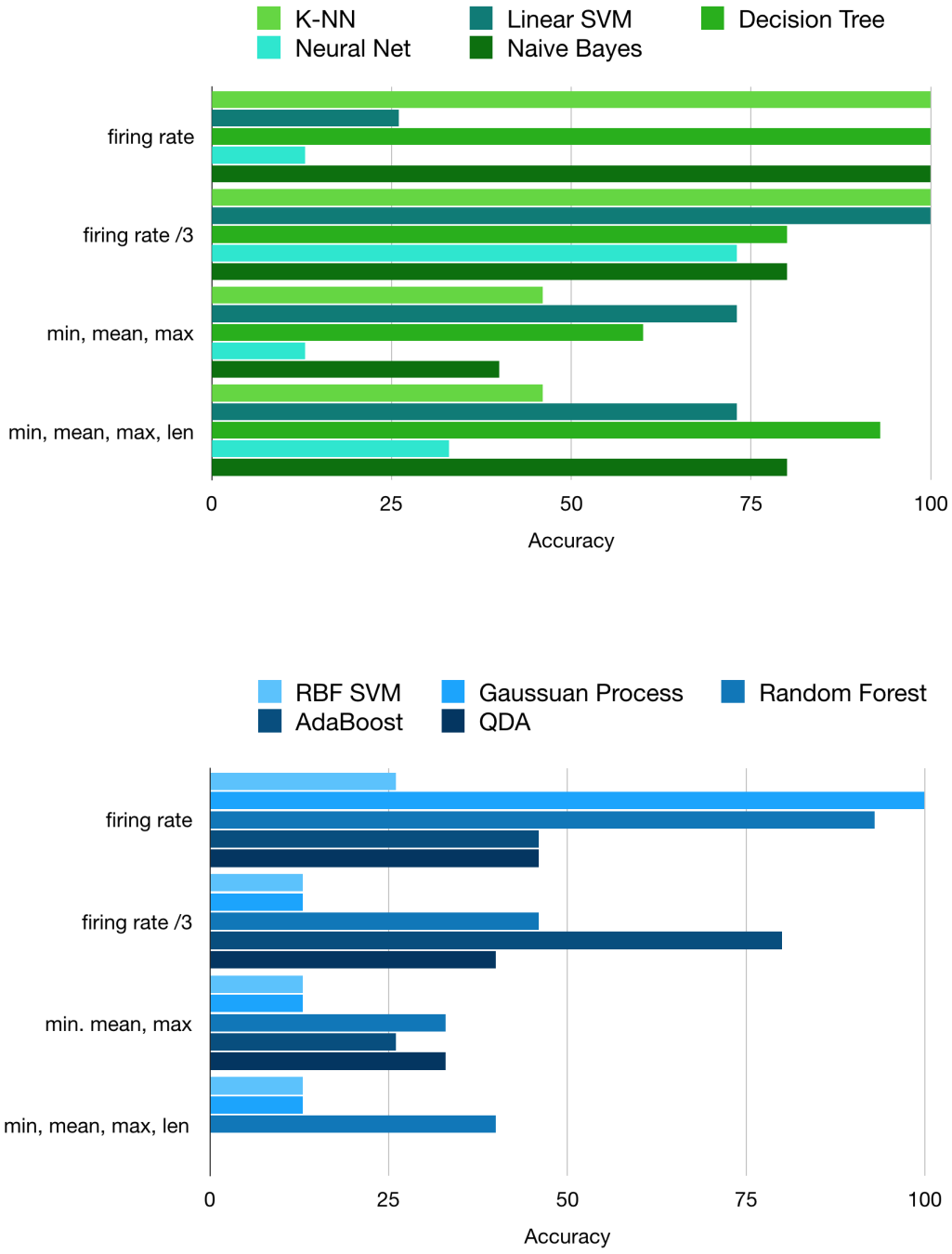


Figura 6.1: Confronto della performance dei vari classificatori utilizzati. Sulle ascisse l'accuratezza e sulle ordinate le feature analizzate.

6.2 Confronto feature

L'implementazione delle feature e la relativa performance che si ottiene una volta fatta la classificazione è l'aspetto più interessante di questa tesi in quanto, ovviamente, parametri diversi utilizzati nelle chiamate ai metodi possono portare a classificazioni più o meno precise.
In questa sezione verranno analizzate le singole feature e la performance di ciascuna di esse verrà rappresentata mediante la cosiddetta *matrice di confusione* che si ottiene incrociando i veri valori della variabile target con quelli stimati dal modello di classificazione.

La classificazione di tutte le feature è stata fatta utilizzando il learner Nearest Neighbors con $K=3$ ed è stata, inoltre, applicata la cross-validazione con 5 folds.

Firing rate

Per la realizzazione di questa feature è stata considerata la frequenza di scarica media di ciascun neurone, come descritto nel paragrafo 4.3 al punto 1.
La funzione "*firing_rate*" implementa tale feature nel seguente modo:

```
def firing_rate(inizio, fine, obj, index):
   for i in range(inizio,fine): #oggetto
    for j in range(10):
        vals[j].append(len(file[i,3][j])/3500)
```

In termini di complessità implementativa questo algoritmo è il più semplice da realizzare, è da notare tuttavia che la performance è molto buona e in generale si ottiene un'accuracy[1] nell'ordine del 95%.

[1]Percentuale di previsioni indovinate dal modello di classificazione.

Performance della classificazione:

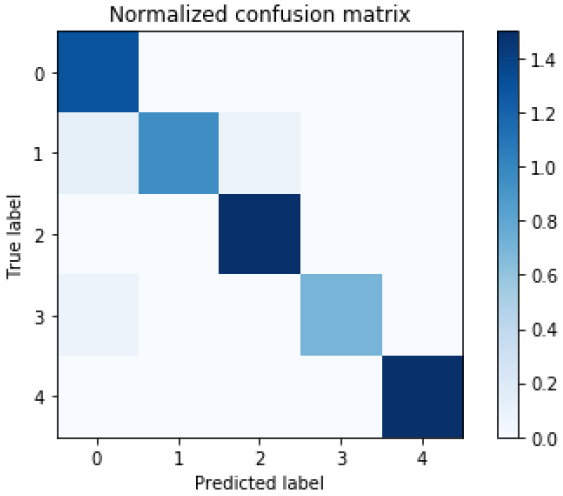


Figura 6.2: Per il firing rate l'accuracy score è pari al 94,67%

Firing rate per intervalli

Per l'implementazione di questa feature è stato suddiviso il timespan considerato in 3, 5 e 7 intervalli, come descritto nel paragrafo 4.3 al punto 2. L'idea che sta alla base di questa feature consiste nell'analizzare gli intervalli in cui il neurone ha rilasciato le scariche elettriche con maggior frequenza, ipotizzando che questa informazione potrebbe migliorare il modello costruito dal learner. La funzione "*firing_rate_intervals*" implementa tale feature nel seguente modo:

```
def fire_rate_intervals(inizio, fine, obj, index):
   for i in range(inizio,fine): ################ oggetto
    for j in range(10):
        vals[j].append(setValues(file[i,3][j]))
```

Questa funzione invoca "*setValues*" che permette di suddividere lo *spike vector* in 3, 5 o 7 intervalli.

Di seguito è mostrato il codice relativo al primo caso.

```
def setValues(array):
    #intervalli considerati
    a=1150
    b=2300
    val=[]

    values1=[]
    values2=[]
    values3=[]
    for i in range(len(array)):
        if array[i]>=0 and array[i]<=a:
            values1.append(array[i])
        if array[i]>=a and array[i]<=b:
            values2.append(array[i])
        if array[i]>=b and array[i]<=3500:
            values3.append(array[i])

    l1 = seq(values1)
    val.append(l1)
    l2 = seq(values2)
    val.append(l2)
    l3 = seq(values3)
    val.append(l3)

    return val
```

La funzione "*seq*" permette di indicare la frequenza di scarica dei neuroni nell'intervallo considerato:

```
#funzione che calcola la lunghezza di un array
def seq(sequence):
    if sequence == []:
        length=0
    else:
        length=len(sequence)
    return length
```

Per le varianti relative a 5 e 7 intervalli il codice è lo stesso, vengono solo modificati i timestamp della funzione "*setValues*".
I risultati delle rispettive varianti sono di seguito rappresentati attraverso la matrice di confusione.

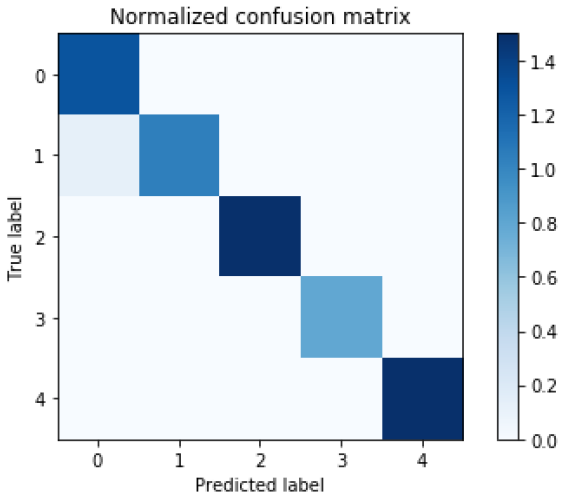


Figura 6.3: Per il firing rate dei 3 intervalli l'accuracy score è pari al 96,67%

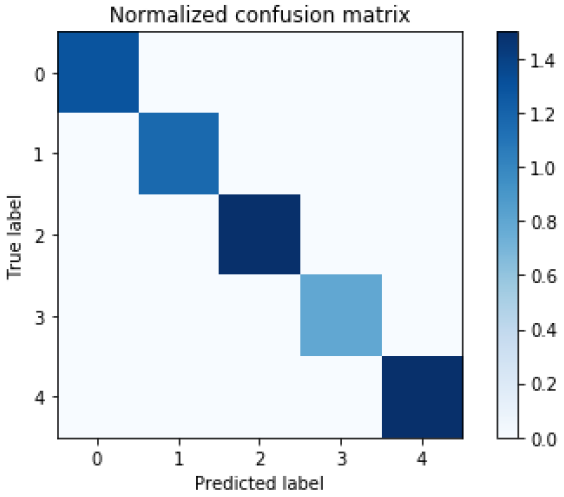


Figura 6.4: Per il firing rate dei 5 intervalli l'accuracy score è pari al 97,07%

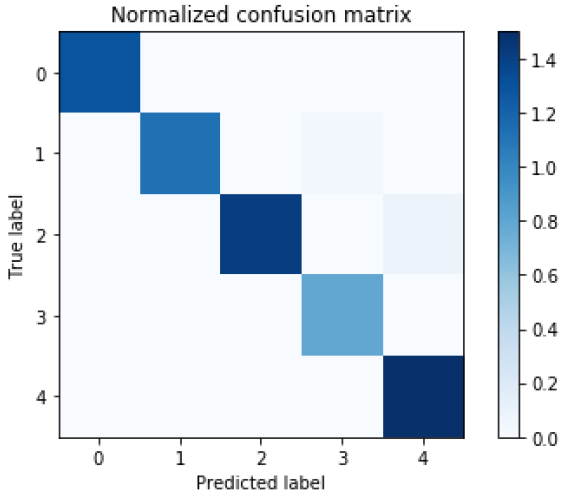


Figura 6.5: Per il firing rate dei 7 intervalli l'accuracy score è pari al 97,11%

Minimo, media e massimo

Per la realizzazione di questa feature è stato considerato il valore minimo, medio e massimo di ogni *spike vector*, come descritto nel paragrafo 4.3 al punto 3. La funzione che implementa ciò è la seguente:

```
#funzione che calcola minimo, media e massimo di ogni trial
def setValues(array):
    #intervallo considerato
    minimo=0
    massimo=3500
    val=[]
    values=[]
    for i in range(len(array)):
        if array[i]>=minimo and array[i]<=massimo:
            values.append(array[i])
    minVal = my_min(values)
    val.append(minVal)
    media=round(np.mean(values))
    if math.isnan(media):
        media=0
    val.append(media)
    maxVal=my_max(values)
    val.append(maxVal)
    return val
```

Rispetto alle precedenti featues, la performance di quest'ultima implementazione è peggiore:

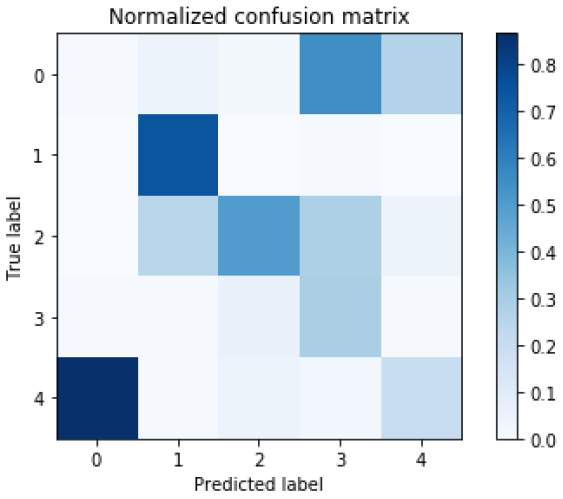


Figura 6.6: Per il valore minimo, la media ed il massimo l'accuracy score è pari al 89,33%.

Per ottenere prestazioni più performanti si è provato ad eliminare i neuroni che hanno valori relativi al minimo, media e massimo uguali a *zero*. Tuttavia, il risultato derivante dalla classificazione del dataset *filtrato* non è stato migliore: l'accuracy score è pari all' 83,50%.
È da tenere in considerazione che questi risultati si possono attribuire unicamente al learner *Nearest Neighbors*, altri classificatori, con lo stesso dataset, darebbero luogo a risultati differenti (migliori o peggiori).

È stato fatto un ulteriore tentativo provando a lavorare su diversi intervalli del timespan. Sono state compiute diverse classificazioni incrementando ogni volta l'intervallo considerato di *500 secondi*. Ciò che è risultato da questa variante si può vedere nel grafico raffigurato di seguiito.

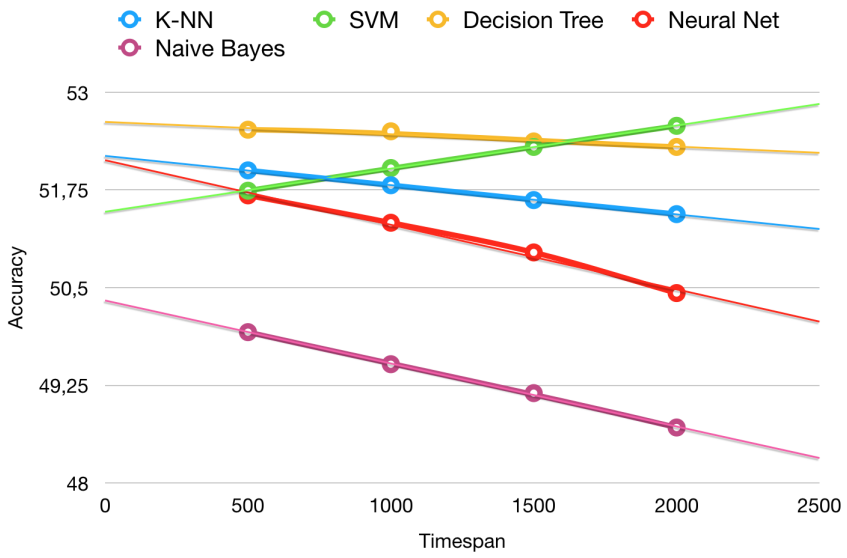


Figura 6.7: Trend dei diversi classificatori considerando i vari intervalli

Minimo, media, massimo e frequenza

Questa feature è frutto della concatenazione fra il *firing rate* ed i valori di *minimo, media e massimo*, come descritto nel paragrafo 4.3 al punto 4.
La sua implementazione è stata pensata per generare un dataset più descrittivo, in modo da avere una performance migliore rispetto la sola classificazione delle due features (*firing rate* e *minimo, media, massimo*) separate.
Sebbene il dataset così costruito possa sembrare l'opzione migliore, nella realtà i classificatori non forniscono risultati più performanti, anzi, il modello di classificazione risulta di molto peggiore rispetto alle altre features.
Come è stato fatto per la feature precedente, anche in questo caso si è pensato di eliminare i neuroni che hanno valori relativi al minimo, media, massimo e frequenza uguali a *zero* e, inoltre, è stato fatto anche qui un ulteriore tentativo provando a lavorare su diversi intervalli del timespan.
L'accuratezza, in entrambi i casi, è peggiorata.
Di seguito sono riportati i risultati ottenuti dalla classificazione:

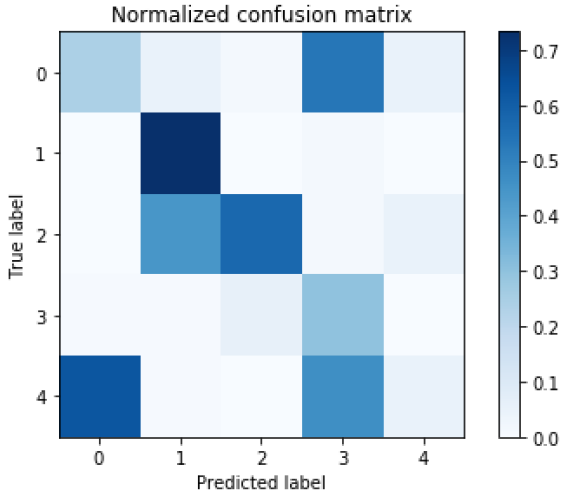


Figura 6.8: Per il valore minimo, la media, il massimo e la frequenza, l'accuracy score è pari al 48,21%.

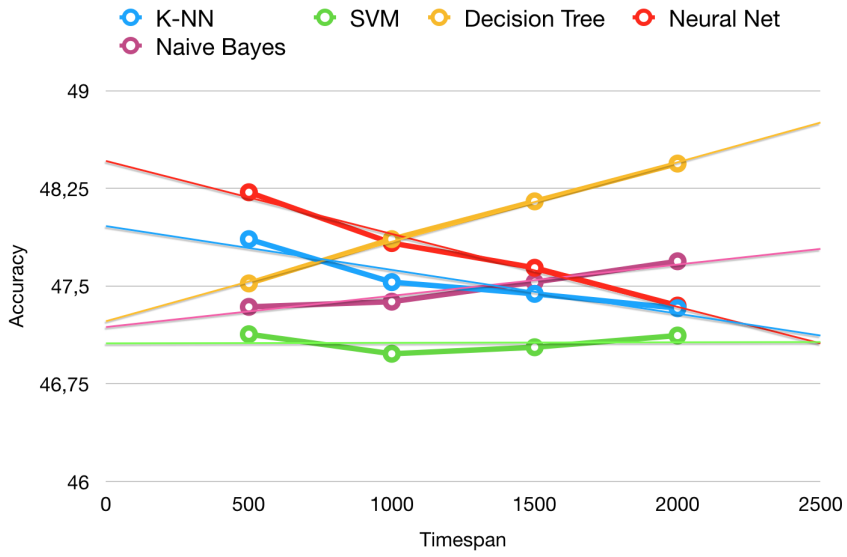
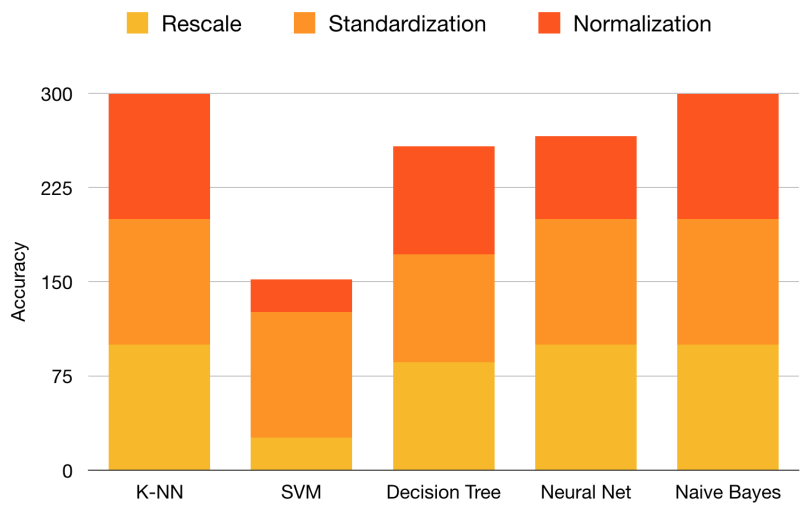


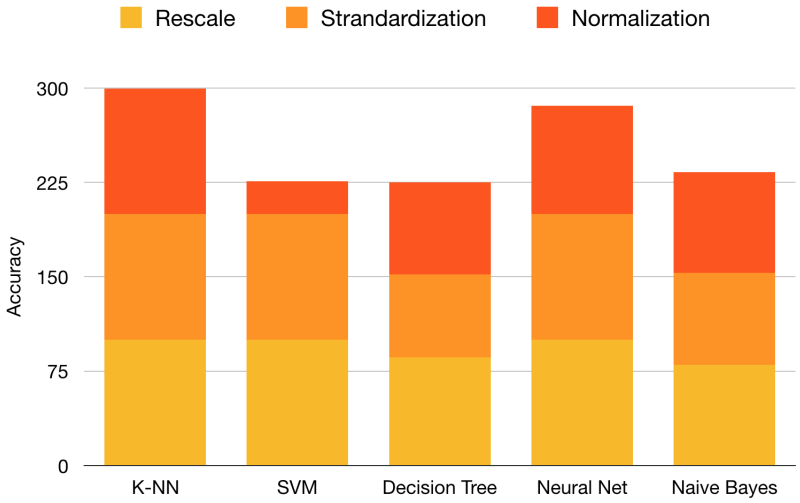
Figura 6.9: Trend dei diversi classificatori considerando i vari intervalli

6.3 Confronto preprocessi

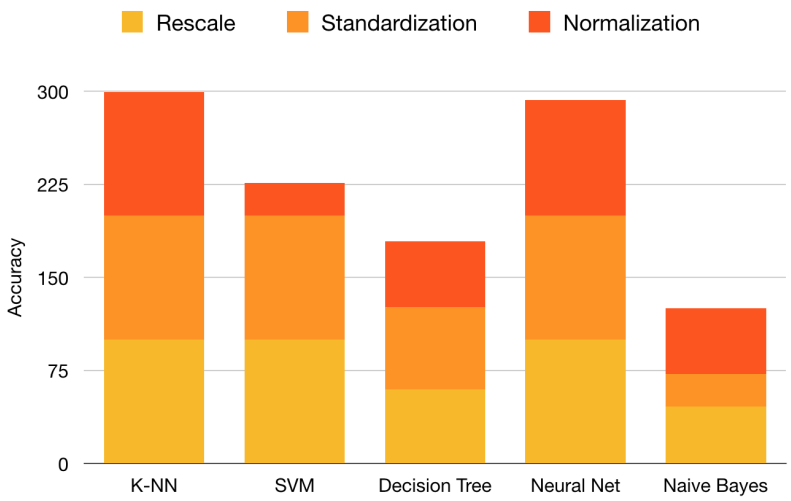
In questa sezione verranno analizzati i risultati forniti dai classificatori considerati più performanti per questo caso di studio, applicando la normalizzazione dei dati di tipo "*rescale*", "*standardize*" e "*normalize*".
Per maggiore chiarezza sono state divise le diverse features.
Per il *firing rate* si ottiene:



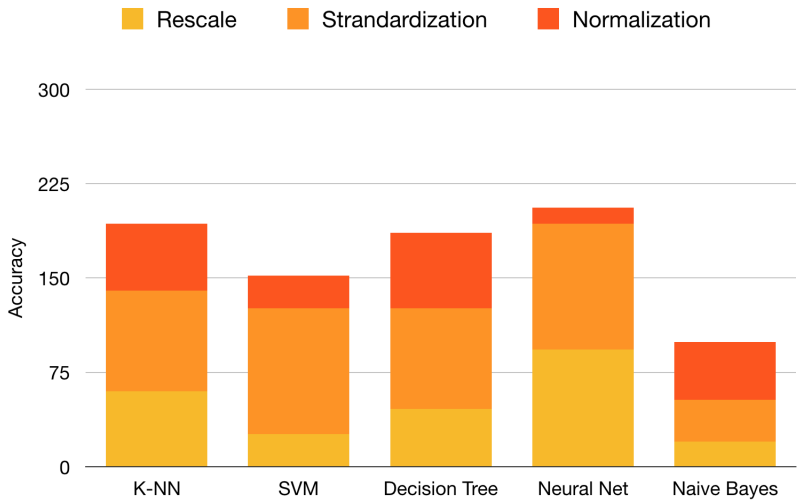
Per il *firing rate* analizzato negli intervalli [0,1150], [1150, 2300] e [2300, 3500] invece:



Per il *firing rate* analizzato negli intervalli [0, 500], [500, 1000], [1000, 1500], [1500, 2000], [2000, 2500], [2500, 3000] e [3000, 3500]:



Per i valori relativi al *minimo, media e massimo*:



Per la combinazione di *minimo, media, massimo e frequenza*:

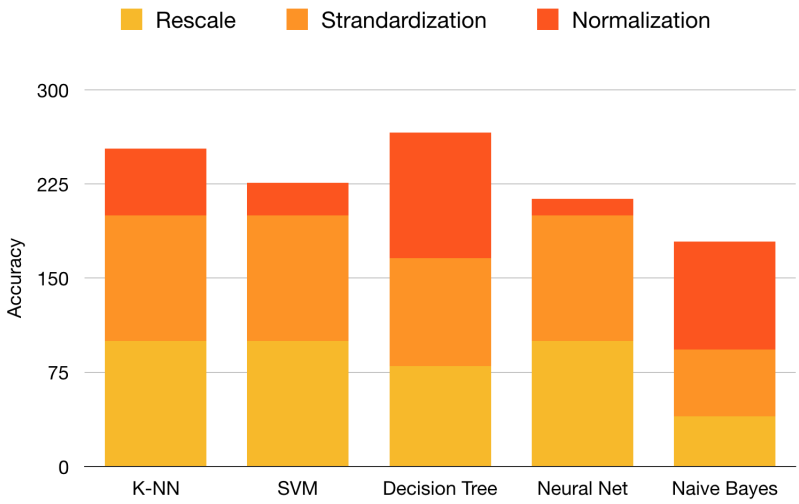


Figura 6.10: Confronto della performance dei diversi preprocessi

Come si evince dai grafici, i risultati ottenuti dalla classificazione sono più efficienti se i dati vengono normalizzati, in particolare si nota che si ottiene un'accuratezza migliore se ai dati viene applicato un *preprocessing* di tipo *rescale* o standardization.

Conclusioni

L'unione di innovazioni tecnologiche e ricerche in ambito medico hanno permesso, fra le altre cose, di costruire numerosi sistemi in grado di assistere le persone che stanno seguendo un percorso riabilitativo, mettendo a loro disposizione arti artificiali in grado di estrarre l'informazione sull'intenzione di movimento e successivamente di riprodurlo attraverso l'ausilio di protesi.

Questo lavoro di tesi si concentra sull'aspetto relativo alla decodifica dell'attività neuronale per il riconoscimento del movimento che è stato effettuato durante le sessioni sperimentali svolte su primati non umani.
L'analisi di decodifica presentata dimostra come sia possibile recuperare efficientemente l'informazione sul tipo di presa che si vuole effettuare, attraverso l'impiego di un algoritmo di classificazione e, eventualmente, un *preprocessing* dei dati. Quest'ultima fase è particolarmente importante poiché definisce la costruzione del modello sul quale verrà poi addestrato il classificatore.
È stato inoltre dimostrato come diverse configurazioni per la costruzione del modello possano influire sulla bontà della previsione. Si è quindi cercato di costruire un modello che fosse il più significativo possibile per l'analisi dei dati del caso di studio esaminato.
I risultati mostrano tassi di riconoscimento più alti quando la classificazione viene eseguita in base alla frequenza media degli *spike* piuttosto che a valori come minimo, media e massimo dell'attività di *spiking*.
Provando inoltre ad aggregare diverse feature fra loro, la performance risulta peggiore rispetto alla classificazione delle feature separate. Sebbene in questo

modo il dataset può sembrare più descrittivo, questa ipotesi viene smentita poiché gli algoritmi di classificazione scalano male quando il dataset ha un numero di attributi (colonne) maggiore del numero delle istanze (righe).
Si può notare anche che alcuni algoritmi sono più efficienti se i dati vengono normalizzati, in particolare si nota che le *reti neurali* generalmente raggiungono un'accuratezza migliore se ai dati viene applicato un *preprocessing* di tipo *rescale* o *standardization*.

Riguardo al caso di studio trattato in questa tesi, si sono riscontrate alcune limitazioni poiché, durante la messa in atto degli esperimenti, è stato prodotto un ridotto numero di campioni.
Un'estensione naturale sarebbe quella di ripetere l'analisi dei dati, a fronte di un arricchimento significativo del campione sperimentale.
Inoltre, sarebbe utile ampliare la gamma di *feature* sviluppate per vedere come queste rispondono al problema del riconoscimento della presa: un esempio potrebbe essere dato dal calcolo della differenza fra il valore *t* e *t-1* di ogni *spike vector*.

Appendice A

Statistiche

In questa sezione sono riportati i risultati relativi alla performance di ciascuna feature implementata. Le seguenti classificazioni sono state effettuate sul dataset '*keyupLIGHT*' allineato a [-2.5; +1] secondi:

Firing rate:

	no preprocess	rescale	std	normalize
Nearest Neighbors	100%	100%	100%	100%
Linear SVM	27%	27%	100%	27%
Radial Basis Function SVM	27%	13%	27%	100%
Gaussian Process	100%	100%	13%	100%
Decision Tree	100%	87%	80%	73%
Random Forest	87%	67%	60%	87%
Neural Net	13%	100%	100%	67%
AdaBoost	47%	40%	40%	47%
Naive Bayes	100%	100%	100%	100%
Quadratic Discriminant Analysis	47%	13%	20%	40%

Firing rate in 3 intervalli:

	no preprocess	rescale	std	normalize
Nearest Neighbors	100%	100%	100%	100%
Linear SVM	100%	100%	100%	27%
Radial Basis Function SVM	13%	13%	27%	100%
Gaussian Process	13%	100%	13%	100%
Decision Tree	87%	80%	73%	80%
Random Forest	47%	87%	40%	47%
Neural Net	93%	100%	100%	93%
AdaBoost	80%	67%	80%	67%
Naive Bayes	80%	80%	73%	80%
Quadratic Discriminant Analysis	40%	13%	53%	27%

Firing rate in 5 intervalli:

	no preprocess	rescale	std	normalize
Nearest Neighbors	100%	100%	100%	100%
Linear SVM	100%	100%	100%	27%
Radial Basis Function SVM	13%	13%	13%	100%
Gaussian Process	13%	100%	13%	100%
Decision Tree	73%	93%	80%	60%
Random Forest	60%	60%	33%	53%
Neural Net	67%	100%	100%	100%
AdaBoost	73%	73%	80%	87%
Naive Bayes	80%	80%	47%	80%
Quadratic Discriminant Analysis	53%	20%	13%	13%

Firing rate in 7 intervalli:

	no preprocess	rescale	std	normalize
Nearest Neighbors	100%	100%	100%	100%
Linear SVM	100%	100%	100%	27%
Radial Basis Function SVM	13%	13%	13%	100%
Gaussian Process	13%	33%	13%	100%
Decision Tree	60%	73%	67%	60%
Random Forest	40%	47%	67%	40%
Neural Net	93%	100%	100%	93%
AdaBoost	60%	73%	67%	73%
Naive Bayes	53%	47%	27%	53%
Quadratic Discriminant Analysis	0%	47%	47%	40%

Minimo, media, massimo:

	no preprocess	rescale	std	normalize
Nearest Neighbors	47%	60%	87%	53%
Linear SVM	73%	27%	100%	27%
Radial Basis Function SVM	13%	13%	27%	27%
Gaussian Process	13%	47%	13%	87%
Decision Tree	67%	47%	73%	47%
Random Forest	33%	27%	27%	20%
Neural Net	20%	93%	100%	20%
AdaBoost	27%	33%	20%	27%
Naive Bayes	47%	27%	40%	53%
Quadratic Discriminant Analysis	33%	27%	20%	13%

Minimo, media, massimo con riduzione orizzontale:

	no preprocess	rescale	std	normalize
Nearest Neighbors	27%	73%	53%	33%
Linear SVM	67%	13%	80%	7%
Radial Basis Function SVM	7%	7%	7%	7%
Gaussian Process	7%	40%	7%	47%
Decision Tree	67%	53%	47%	47%
Random Forest	13%	20%	20%	20%
Neural Net	0%	87%	73%	7%
AdaBoost	53%	53%	27%	53%
Naive Bayes	60%	27%	40%	60%
Quadratic Discriminant Analysis	20%	33%	13%	27%

Minimo, media, massimo, frequenza:

	no preprocess	rescale	std	normalize
Nearest Neighbors	47%	100%	100%	53%
Linear SVM	73%	100%	100%	27%
Radial Basis Function SVM	13%	13%	13%	27%
Gaussian Process	13%	40%	13%	67%
Decision Tree	100%	67%	87%	87%
Random Forest	47%	47%	33%	53%
Neural Net	33%	100%	100%	13%
AdaBoost	47%	40%	40%	47%
Naive Bayes	80%	40%	53%	87%
Quadratic Discriminant Analysis	40%	0%	20%	27%

Minimo, media, massimo, frequenza con riduzione orizzontale:

	no preprocess	rescale	std	normalize
Nearest Neighbors	27%	93%	87%	33%
Linear SVM	73%	53%	93%	7%
Radial Basis Function SVM	7%	7%	7%	7%
Gaussian Process	7%	40%	7%	47%
Decision Tree	60%	100%	93%	100%
Random Forest	33%	47%	47%	47%
Neural Net	20%	100%	100%	7%
AdaBoost	73%	60%	60%	73%
Naive Bayes	60%	27%	47%	67%
Quadratic Discriminant Analysis	13%	33%	13%	33%

Di seguito sono riportati i risultati relativi alle varie classificazioni effettuate sul database '*keyupDARK*' allineato a [-2.5; +1] secondi:

Firing rate:

	no preprocess	rescale	std	normalize
Nearest Neighbors	73%	100%	100%	53%
Linear SVM	27%	27%	100%	20%
Radial Basis Function SVM	27%	13%	27%	27%
Gaussian Process	80%	100%	13%	93%
Decision Tree	33%	53%	60%	53%
Random Forest	33%	47%	40%	60%
Neural Net	13%	100%	100%	27%
AdaBoost	13%	33%	33%	47%
Naive Bayes	80%	80%	87%	87%
Quadratic Discriminant Analysis	6%	6%	13%	47%

Firing rate in 3 intervalli:

	no preprocess	rescale	std	normalize
Nearest Neighbors	67%	100%	93%	53%
Linear SVM	73%	67%	100%	27%
Radial Basis Function SVM	13%	13%	27%	60%
Gaussian Process	13%	NaN	13%	80%
Decision Tree	67%	47%	73%	53%
Random Forest	53%	27%	47%	47%
Neural Net	53%	100%	100%	27%
AdaBoost	40%	33%	40%	40%
Naive Bayes	73%	67%	13%	80%
Quadratic Discriminant Analysis	40%	20%	27%	6%

Firing rate in 5 intervalli:

	no preprocess	rescale	std	normalize
Nearest Neighbors	67%	100%	100%	73%
Linear SVM	87%	100%	100%	27%
Radial Basis Function SVM	13%	13%	13%	73%
Gaussian Process	13%	100%	13%	80%
Decision Tree	80%	60%	73%	53%
Random Forest	60%	13%	20%	40%
Neural Net	67%	100%	100%	53%
AdaBoost	40%	40%	40%	40%
Naive Bayes	47%	53%	47%	47%
Quadratic Discriminant Analysis	20%	13%	20%	20%

Firing rate in 7 intervalli:

	no preprocess	rescale	std	normalize
Nearest Neighbors	53%	100%	100%	67%
Linear SVM	80%	100%	100%	27%
Radial Basis Function SVM	13%	13%	13%	67%
Gaussian Process	13%	13%	13%	67%
Decision Tree	67%	47%	53%	67%
Random Forest	40%	53%	53%	47%
Neural Net	20%	100%	100%	67%
AdaBoost	53%	53%	53%	33%
Naive Bayes	27%	47%	27%	27%
Quadratic Discriminant Analysis	20%	0%	20%	13%

Minimo, media, massimo:

	no preprocess	rescale	std	normalize
Nearest Neighbors	53%	73%	87%	53%
Linear SVM	73%	27%	80%	27%
Radial Basis Function SVM	13%	13%	20%	27%
Gaussian Process	13%	40%	13%	53%
Decision Tree	53%	53%	47%	47%
Random Forest	53%	33%	13%	27%
Neural Net	33%	87%	93%	13%
AdaBoost	40%	33%	33%	40%
Naive Bayes	40%	20%	33%	33%
Quadratic Discriminant Analysis	6%	13%	13%	26%

Minimo, media, massimo con riduzione orizzontale:

	no preprocess	rescale	std	normalize
Nearest Neighbors	47%	40%	40%	53%
Linear SVM	80%	6%	53%	6%
Radial Basis Function SVM	6%	6%	6%	6%
Gaussian Process	6%	40%	6%	47%
Decision Tree	47%	53%	33%	53%
Random Forest	40%	13%	27%	27%
Neural Net	6%	60%	73%	6%
AdaBoost	20%	20%	20%	13%
Naive Bayes	6%	27%	20%	13%
Quadratic Discriminant Analysis	27%	20%	6%	6%

Minimo, media, massimo, frequenza:

	no preprocess	rescale	std	normalize
Nearest Neighbors	53%	93%	100%	53%
Linear SVM	73%	73%	100%	27%
Radial Basis Function SVM	13%	13%	13%	27%
Gaussian Process	13%	73%	13%	53%
Decision Tree	73%	67%	67%	73%
Random Forest	53%	27%	27%	33%
Neural Net	20%	100%	100%	13%
AdaBoost	40%	33%	33%	40%
Naive Bayes	53%	33%	53%	47%
Quadratic Discriminant Analysis	20%	33%	13%	27%

Minimo, media, massimo, frequenza con riduzione orizzontale:

	no preprocess	rescale	std	normalize
Nearest Neighbors	47%	67%	73%	53%
Linear SVM	80%	33%	93%	6%
Radial Basis Function SVM	6%	6%	6%	6%
Gaussian Process	6%	13%	6%	60%
Decision Tree	67%	73%	73%	73%
Random Forest	27%	40%	33%	47%
Neural Net	6%	93%	87%	6%
AdaBoost	53%	53%	53%	53%
Naive Bayes	6%	33%	26%	20%
Quadratic Discriminant Analysis	20%	20%	20%	33%

Appendice B

Codice

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import metrics
from sklearn.model_selection import train_test_split
from sklearn.model_selection import ShuffleSplit
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.gaussian_process import
   GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import
   RandomForestClassifier, AdaBoostClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.discriminant_analysis import
```

```
    QuadraticDiscriminantAnalysis
from sklearn.preprocessing import MinMaxScaler,
    StandardScaler, Normalizer, Binarizer
from sklearn.model_selection import train_test_split

#importo i datset
#keyup light [-2.5; +1]
file = np.load("./dataset/keyupLIGHT.npy")
#keyup dark [-2.5; +1]
#file = np.load('./dataset/keyupDARK.npy')
#luceOn light [-1; +2.5]
#file = np.load('./dataset/luceOnLIGHT.npy')
#luceOn dark [-1; +2.5]
#file = np.load('./dataset/luceOnDARK.npy')

cf_matrix = []
accuracy = []

def setVal(inizio, fine, obj, index):
    for i in range(inizio,fine):
     for j in range(10):
         vals[j].append(len(file[i,3][j])/3500)
    for k in range(10):
        values=[]
        values=np.array(vals[k]).reshape(1,-1)
        array[index,0:79]=values
        array[index,79]=obj
        objs.append(obj)
        index=index+1
    reset(vals)

```

```
48 def reset(vals):
49     for v in range(len(vals)):
50         vals[v]=[]
51 
52 def plot_confusion_matrix(cm, title='Confusion
   matrix', cmap=plt.cm.Blues):
53         plt.imshow(cm, interpolation='nearest',
   cmap=cmap)
54         plt.title(title)
55         plt.colorbar()
56         plt.tight_layout()
57         plt.ylabel('True label')
58         plt.xlabel('Predicted label')
59 
60 def preprocess(X_train, X_test, y_train, y_test,
   classificatore):
61     clf = classificatore
62     clf.fit(X_train, y_train)
63     preds = clf.predict(X_test)
64     print("\nreal  -> ",np.array(y_test))
65     print("preds -> ",np.array(preds))
66 
67     # Compute confusion matrix
68     cm = metrics.confusion_matrix(y_test,preds)
69     cf_matrix.append(cm)
70     np.set_printoptions(precision=2)
71     cm_normalized = cm.astype('float') /
   cm.sum(axis=1)[:, np.newaxis]
72     print('\nNormalized confusion matrix: ')
73     print(cm_normalized)
74 
```

```
75      ac=metrics.accuracy_score(np.array(y_test),
    np.array(preds), normalize=False)
76      accuracy_score = (ac*100)/15
77      print("Accuratezza ",accuracy_score,"%")
78      accuracy.append(ac)
79
80  def process(train, test, clf, df):
81      X_train = np.zeros([35, len(df.columns)-1])
82      X_test = np.zeros([15, len(df.columns)-1])
83      y_train = np.zeros([35])
84      y_test = np.zeros([15])
85      for i in range(len(train)):
86          X_train[i,:] = df.loc[train[i],
    0:len(df.columns)-1]
87          y_train[i] = df.loc[train[i],
    len(df.columns)]
88      for i in range(len(test)):
89          X_test[i,:] = df.loc[test[i],
    0:len(df.columns)-1]
90          y_test[i] = df.loc[test[i], len(df.columns)]
91
92      preprocess(X_train, X_test, y_train, y_test,
    clf)
93
94  def plot_matrix():
95      elem = 0
96      mat = np.zeros([5,5])
97      for i in range(len(cf_matrix)):
98          if len(cf_matrix[i]) == 5:
99              elem = elem + 1
100             mat = mat + cf_matrix[i]
```

```
101             mat = mat/elem
102     plot_confusion_matrix(mat, title='Normalized
    confusion matrix')
103     print("\n\nAccuratezza: %1.2f"
    %((np.mean(accuracy)*100)/15),"%")
104
105 #Data preprocessing: Rescale Data
106 #After rescaling you can see that all of the values
    are in the range between 0 and 1.
107 def rescale_data(classificatore):
108     print("\nData preprocessing: Rescale Data")
109     scaler = MinMaxScaler(feature_range=(0, 1))
110     rescaledX = scaler.fit_transform(array)
111     np.set_printoptions(precision=3)
112     colNames=[]
113     for i in range(len(array[0])):
114         colNames.append(i+1)
115     df=pd.DataFrame(rescaledX, columns=colNames)
116     df = df.drop(80,1)
117     df[80] = pd.Categorical(objs)
118     ss = ShuffleSplit(n_splits=5, test_size=0.3,
    random_state=0)
119
120     for train_index, test_index in ss.split(df):
121         process(train_index, test_index,
    classificatore, df)
122
123     plot_matrix()
124
125 #Data preprocessing: Standardize Data
126 #After S.D. the values for each attribute have a
```

```
    mean value of 0 and a standard deviation of 1.
def std_scaler_data(classificatore):
    print("\nData preprocessing: Standardize Data")
    scaler = StandardScaler().fit(array)
    rescaledX = scaler.transform(array)
    np.set_printoptions(precision=3)
    colNames=[]
    for i in range(len(array[0])):
        colNames.append(i+1)
    df = pd.DataFrame(rescaledX, columns=colNames)
    df = df.drop(80,1)
    df[80] = pd.Categorical(objs)
    ss = ShuffleSplit(n_splits=5, test_size=0.3,
    random_state=0)
    for train_index, test_index in ss.split(df):
        process(train_index, test_index,
    classificatore, df)
    plot_matrix()

#Data preprocessing: Normalize Data
#After normalizing, the rows are normalized to
    length 1.
def normalized_data(classificatore):
    print("\nData preprocessing: Normalize Data")
    scaler = Normalizer().fit(array)
    normalizedX = scaler.transform(array)
    np.set_printoptions(precision=3)
    colNames=[]
    for i in range(len(array[0])):
        colNames.append(i+1)
    df = pd.DataFrame(normalizedX, columns=colNames)
```

```
154     df = df.drop(80,1)
155     df[80] = pd.Categorical(objs)
156     ss = ShuffleSplit(n_splits=5, test_size=0.3,
    random_state=0)
157     for train_index, test_index in ss.split(df):
158         process(train_index, test_index,
    classificatore, df)
159     plot_matrix()
160
161 #Data preprocessing: Binarize Data
162 #After binarizing you can see that all values equal
    or less than 0 are marked 0
163 #and all of those above 0 are marked 1.
164 def bin_data(classificatore):
165     print("\nData preprocessing: Binarize Data")
166     binarizer = Binarizer(threshold=0.0).fit(array)
167     binaryX = binarizer.transform(array)
168     np.set_printoptions(precision=3)
169     colNames=[]
170     for i in range(len(array[0])):
171         colNames.append(i+1)
172     df = pd.DataFrame(binaryX, columns=colNames)
173     df = df.drop(80,1)
174     df[80] = pd.Categorical(objs)
175     ss = ShuffleSplit(n_splits=5, test_size=0.3,
    random_state=0)
176     for train_index, test_index in ss.split(df):
177         process(train_index, test_index,
    classificatore, df)
178     plot_matrix()
179
```

```
180 #No preprocessing
181 def default(classificatore):
182     colNames=[]
183     for i in range(len(array[0])):
184         colNames.append(i+1)
185     df = pd.DataFrame(array, columns=colNames)
186     ss = ShuffleSplit(n_splits=5, test_size=0.3,
    random_state=0)
187     for train_index, test_index in ss.split(df):
188         process(train_index, test_index,
    classificatore, df)
189     plot_matrix()
190
191 """ COMPARE ALL PREPROCESSINGS """
192 def compare(classificatore):
193     print("\nComparing results with and without
    preprocessing data")
194     X = pd.DataFrame(array)
195     X = X.drop(79,1)
196     y = objs
197     X_train, X_test, y_train, y_test =
    train_test_split(X, y, test_size=0.3,
    random_state=42)
198
199     df = pd.DataFrame(X)
200     df['obj']=pd.Categorical(y)
201
202     print('\nNumber of observations in the training
    data:', len(X_train))
203     print('Number of observations in the test
    data:',len(X_test))
```

```
204
205     print("real   -> ",np.array(y_test))
206
207     print("\nNot prepocessed data")
208     clf = classificatore
209     clf.fit(X_train, y_train)
210     preds = clf.predict(X_test)
211     print("preds -> ",np.array(preds))
212     ac=metrics.accuracy_score(y_test,preds,
    normalize=False)
213     print(ac, " su ", len(y_test))
214
215     print("\nRescaled data")
216     scaler = MinMaxScaler(feature_range=(0, 1))
217     rescaledX_train = scaler.fit_transform(X_train)
218     rescaledX_test = scaler.fit_transform(X_test)
219     clf = classificatore
220     clf.fit(rescaledX_train, y_train)
221     preds = clf.predict(rescaledX_test)
222     print("preds -> ",np.array(preds))
223     ac=metrics.accuracy_score(y_test,preds,
    normalize=False)
224     print(ac, " su ", len(y_test))
225
226     print("\nStandardized data")
227     std = StandardScaler().fit(X_train)
228     stdX_train = std.transform(X_train)
229     std = StandardScaler().fit(X_test)
230     stdX_test = std.transform(X_test)
231     clf = classificatore
232     clf.fit(stdX_train, y_train)
```

```
233     preds = clf.predict(stdX_test)
234     print("preds -> ",np.array(preds))
235     ac=metrics.accuracy_score(y_test,preds,
    normalize=False)
236     print(ac, " su ", len(y_test))
237 
238     print("\nNormalized data")
239     norm = Normalizer().fit(X_train)
240     normalizedX_train = norm.transform(X_train)
241     norm = Normalizer().fit(X_test)
242     normalizedX_test = norm.transform(X_test)
243     clf = classificatore
244     clf.fit(normalizedX_train, y_train)
245     preds = clf.predict(normalizedX_test)
246     print("preds -> ",np.array(preds))
247     ac=metrics.accuracy_score(y_test,preds,
    normalize=False)
248     print(ac, " su ", len(y_test))
249 
250 #MAIN
251 array=np.zeros([50,80])
252 objs=[]
253 
254 for i in range(0,10):
255     vals.append([])
256 
257 setVal(0,79,1,0)
258 setVal(79,158,2,10)
259 setVal(158,237,3,20)
260 setVal(237,316,4,30)
261 setVal(316,395,5,40)
```

```
""" Classificators """
clf=KNeighborsClassifier(3)
#clf=SVC(kernel="linear", C=0.025)
#clf=SVC(gamma=2, C=1)
#clf=GaussianProcessClassifier(1.0 * RBF(1.0),
   warm_start=True)
#clf=DecisionTreeClassifier(max_depth=5)
#clf=RandomForestClassifier(max_depth=5,
   n_estimators=10, max_features=1)
#clf=MLPClassifier(alpha=1)
#clf=AdaBoostClassifier()
#clf=GaussianNB()
#clf=QuadraticDiscriminantAnalysis()

""" Data processings """
# Compared data
#compare(clf)

# Not processed data
default(clf)

# Preprocessed data
#rescale_data(clf)
#std_scaler_data(clf)
#normalized_data(clf)
#bin_data(clf)
```

Bibliografia

[1] Velliste M., Perel S., Spalding M. C., Whitford A. S., & Schwartz A. B. "Cortical control of a prosthetic arm for self-feedin". Nature, 2008.

[2] Edward V. Evarts
"Relation of pyramidal tract activity to force exerted during voluntary movement". Journal of Neurophysiology, 1968.

[3] Donald R. Humphrey, E. M. Schmidt, W. D. Thompson.
"Predicting measures of motor performance from multiple cortical spike trains". Science, 1970.

[4] A. P. Georgopoulos, A. B. Schwartz and R. E. Kettner.
"Neuronal population coding of movement direction". Science, 1986.

[5] David M. Brandman, Sydney S. Cash, and Leigh R. Hochberg.
"Review: Human intracortical recording and neural decoding for brain computer interfaces". IEEE Transactions on Neural Systems and Rehabilitation Engineering, 2017.

[6] Paul S. Hammon and Virginia R. de Sa.
"Preprocessing and meta-classification for brain-computer interfaces". IEEE Transactions on Biomedical Engineering, 2007.

[7] F. Lotte, M. Congedo, A. Lécuyer, F. Lamarche and B. Arnaldi.
"A review of classification algorithms for EEG-based brain-computer interfaces". Journal of Neural Engineering, 2007.

[8] Boris Revechkis, Tyson NS Aflalo, Spencer Kellis, Nader Pouratian and Richard A Andersen.
"Parietal Neural Prosthetic Control of a Computer Cursor in a Graphical-User-Interface Task". J Neural Eng,2014.

[9] Silvestro Micera, Stanisa Raspopovic, Francesco Petrini, Jacopo Carpaneto, Calogero Oddo, Jordi Badia, Thomas Stieglitz, Xavier Navarro, Paolo M. Rossini, Giuseppe Granata.
"On the Use of Intraneural Transversal Electrodes to Develop Bidirectional Bionic Limbs". Converging Clinical and Engineering Research on Neurorehabilitation, 2016.

[10] Stanisa Raspopovic, Marco Capogrosso, Francesco Maria Petrini, Marco Bonizzato2, Jacopo Rigosa, Giovanni Di Pino, Jacopo Carpaneto, Marco Controzzi, Tim Boretius, Eduardo Fernandez, Giuseppe Granata, Calogero Maria Oddo, Luca Citi, Anna Lisa Ciancio, Christian Cipriani, Maria Chiara Carrozza, Winnie Jensen, Eugenio Guglielmelli, Thomas Stieglitz, Paolo Maria Rossini and Silvestro Micera.
"Restoring Natural Sensory Feedback in Real-Time Bidirectional Hand Prostheses". Science Translational Medicine, 2014.

[11] Oded Z. Maimon, Lior Rokach.
"Data Mining and Knowledge Discovery Handbook". Springer Science & Business Media, 2010.

[12] A. Azzalini, B. Scarpa.
"Analisi dei dati e data mining". Springer Science & Business Media, 2009.

[13] Jussi Tohka.
"Introduction to Pattern Recognition: k-Nearest neighbors classification". Institute of Signal Processing, Tampere University of Technology.
[http://www.cs.tut.fi/sgn/m2obsi/courses/IPR/Lectures/IPR_Lecture_8.pdf]

[14] Scikit-Learn – Machine Learning in Python.
[http://scikit-learn.org/stable/modules/naive_bayes.html]

[15] Leo Breiman.
"Bagging predictors". Machine learning, 1996.

[16] Robert E. Schapire.
"A Brief introduction to Boosting". Machine learning, 1999.

[17] [https://www.python.it/about/]

Ringraziamenti

Desidero innanzitutto ringraziare il Prof. Marco Di Felice, relatore di questa tesi, per il tempo che mi ha dedicato, la cortesia e la pazienza che ha avuto nei miei confronti.
Un ringraziamento particolare va al mio correlatore Matteo Filippini per la disponibilità, l'aiuto e i consigli che mi ha dato durante la realizzazione di questa tesi. Grazie Matteo.
Ringrazio tantissimo anche la Professoressa Patrizia Fattori per avermi permesso di svolgere il tirocinio all'interno del Dipartimento di Farmacia e Biotecnologie dell'Università di Bologna.

Un sentito ringraziamento va a tutti i miei amici e, soprattutto, alle mie migliori amiche, Michaela e Fabiola, che mi sono sempre state vicine e mi hanno sostenuta e compresa in tutti questi anni. Vi voglio bene ragazze.

È doveroso un ringraziamento ai miei colleghi di corso, che mi hanno sopportata e supportata in questi 3 anni, rendendoli più leggeri e riempiendoli di risate. Grazie ragazzi, senza di voi non sarebbe stata la stessa cosa.
Ringrazio anche i nuovi colleghi che, pur conoscendomi da poco, sono stati gentili e affettuosi con me fin da subito.

Il ringraziamento più grande va ai miei genitori, che sono stati e saranno sempre il mio punto di riferimento, la mia forza ed il mio coraggio.