



UNIVERSITÀ DEGLI STUDI DI MESSINA

---

DIPARTIMENTO DI INGEGNERIA

Corso di Laurea Triennale in Ingegneria Elettronica e  
Informatica

MODELLI DI STIMA DELLA LUNGHEZZA DEL PASSO  
TRAMITE DATI INERZIALI

Tesi di Laurea di:  
Aliberti Andrea

Relatore:  
Chiar.mo Prof. Luca Patanè

---

ANNO ACCADEMICO 2021–2022



# Indice

<b>1 Benchmark Dataset ed elaborazione dei dati</b>	<b>5</b>
1.1 Benchmark Dataset . . . . .	6
1.2 Pre-processamento dei dati . . . . .	12
1.3 Analisi delle feature . . . . .	14
<b>2 Classificazione delle modalità di posizionamento dello smart-phone</b>	<b>21</b>
2.1 Cenni di teoria . . . . .	23
2.2 Applicazione del DecisionTree . . . . .	24
<b>3 Regressione della lunghezza del passo</b>	<b>30</b>
3.1 Cenni di teoria . . . . .	31
3.2 Applicazione del Modello Lineare . . . . .	32
3.2.1 Modello indipendente . . . . .	36
3.2.2 Modello dipendente . . . . .	44
<b>Conclusioni</b>	<b>55</b>
<b>Bibliografia</b>	<b>56</b>
<b>Appendice</b>	<b>57</b>

# Introduzione

Con l'aumento dell'importanza dell'Internet of Things (IoT) è centrale negli ultimi anni lo studio della localizzazione indoor, in particolare abbiamo attenzionato un elemento cruciale nel calcolo della distanza percorsa da un soggetto in movimento, ovvero la stima della lunghezza del passo.

La distanza percorsa in una camminata è rilevante in vari ambiti, tra cui quello medico, per valutare l'allenamento, la condizione di salute o per sistemi di navigazione; una stima efficace della distanza percorsa è cruciale. Riveste dunque particolare importanza la stima della lunghezza del singolo passo per poter risalire alla distanza percorsa.

I metodi per la stima della lunghezza del passo possono essere raggruppati in due macrocategorie: i metodi diretti basati sulla doppia integrazione dell'accelerazione frontale ed i metodi indiretti che invece sfruttano modelli o "ipotesi" per la computazione della lunghezza del passo.

Teoricamente i metodi diretti sarebbero i migliori in quanto non si affidano a nessun modello o dispositivo, tuttavia sono anche i più imprecisi in quanto l'errore dovuto al bias della doppia integrazione ed al rumore dei sensori causa un aumento dell'errore incontrollabile, oltre alla difficoltà insita nell'ottenere la sola componente antero-posteriore dell'accelerazione dai sensori. Altri metodi per la stima della lunghezza del passo si sono basati sulle caratteristiche fisiche del soggetto, sfruttando le formule del movimento del pendolo ed applicandole al corpo umano sfuggendo il suo centro di massa (CoM) [1], tuttavia questi metodi sono poco robusti e necessitano una conoscenza approfondita del soggetto. Metodi basati su ipotesi sono per esempio quelli proposti da Weinberg [2], Kim [3] e Scarlett [4] che sfruttano le misurazioni dell'accelerazione con un determinato peso per calcolare la lunghezza di ogni passo, ma tutti questi metodi presuppongono che il soggetto debba indossare un sensore esterno in una specifica posizione del corpo. Volendo quindi avere un'applicazione nella vita di tutti i giorni non è possibile sfrut-

tare questi metodi agevolmente. Così si è palesata l'importanza di spostare la complessità del sistema non sull'hardware, bensì sul software.

Alleggerendo quindi l'hardware per rendere queste applicazioni user friendly gli sforzi si sono concentrati sull'uso dello smartphone come unica unità fisica necessaria alla stima della lunghezza del passo, in particolare si sono utilizzati l'accelerometro ed il giroscopio integrati nello smartphone come unici sensori. Ad un alleggerimento dell'hardware è corrisposto un aumento di complessità del software, concentrandosi sull'uso di modelli matematici per la stima della lunghezza del passo, affidandosi ad algoritmi di Machine Learning o Deep Learning [5].

Esempi nell'uso di ML li troviamo in [6], dove viene utilizzata la Long-Short Term Memory (LSTM) e la Denoising Autoencoders (DAE) per la stima della lunghezza del passo, o in [7], che utilizza cinque modelli per la regressione concatenati ad un sesto modello per unire i cinque precedenti. Entrambi ([7] [6] propongono modelli e strategie basati su una regressione multi-modello ed un algoritmo generale piuttosto complesso.

In questa tesi abbiamo provato a sfruttare al meglio un modello molto più semplice per poterne valutare le performance e vedere se per qualche applicazione meno esigente è possibile utilizzare un modello poco complesso dal punto di vista computazionale così da poterlo implementare anche direttamente sullo smartphone.

L'obiettivo di questa tesi è dunque quello di stimare la lunghezza del passo umano durante una camminata, utilizzando solo l'accelerometro ed il giroscopio integrati nello smartphone, libero di muoversi (posizione non fissa dello smarphone) ed usando il modello più semplice possibile per la regressione dell'accelerazione, dovendo quindi ottimizzare tutto ciò che sta intorno al modello per migliorare le performance.

Il tutto è stato implementato in Python 3.10 sfruttando l'editor Jupyter Notebook.

Nel Capitolo 1 vediamo più nello specifico come sono fatti i nostri dati, come vengono gestiti e come vengono formattati per essere utilizzabili in problemi di Machine Learning, nel Capitolo 2 la descrizione del modello utilizzato per la classificazione e l'applicazione dello stesso nel nostro dataset, nel Capitolo 3 la descrizione del modello utilizzato per la regressione della lunghezza del passo e l'applicazione dello stesso nel nostro dataset, nel Capitolo 3.2.2 le conclusioni ed infine nell'Appendice l'esempio dell'algoritmo sviluppato su Jupyter Notebook, comprensivo di tutto ciò che verrà trattato

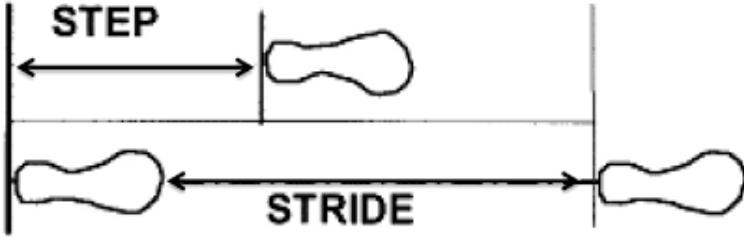
nella tesi, per la modalità "Handheld".

# Capitolo 1

## Benchmark Dataset ed elaborazione dei dati

**Background** I segnali derivanti dalla misurazione tramite i sensori dello smartphone costituiscono una serie temporale. Un serie temporale è una serie di osservazioni ad intervalli regolari che descrive la dinamica di un certo fenomeno nel tempo. Analizzando la dinamica temporale della camminata vari studi [8] hanno evidenziato la natura armonica della serie temporale derivante dalla misurazione dell’accelerazione durante la camminata, in ognuno dei 3 assi dell’accelerometro. Ciò permette, per esempio, di calcolare se si sta camminando (o se si è fermi) ed il momento in cui si compie un passo [9], anch’esso uno step fondamentale per la corretta stima della lunghezza del passo, tramite lo studio della serie temporale armonica.

Evidenziata questa natura armonica dell’andatura umana è necessario definire in maniera univoca cos’è un passo: un passo (o in inglese *stride*) è lo spazio che intercorre tra due appoggi successivi dello stesso piede Figura [1.1].



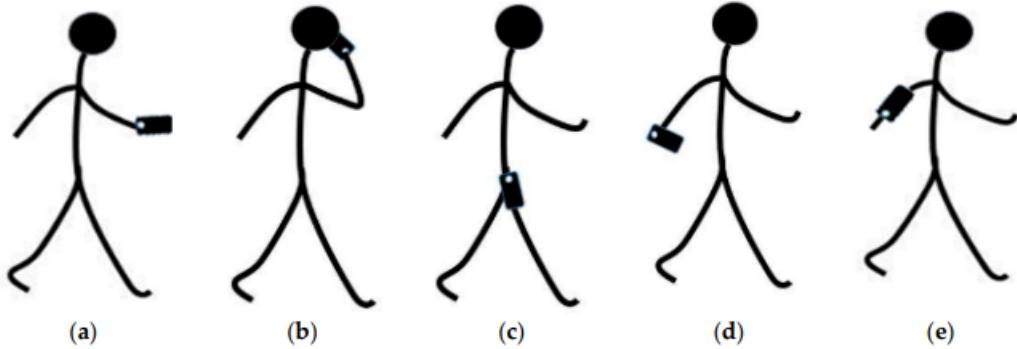
**Figura 1.1: Definizione di Stride:** "Stride" non ha una vera e propria traduzione in italiano, quindi da ora in avanti ci riferiremo allo stesso con "passo" sapendo di riferirci allo stride.

La serie temporale deve dunque essere segmentata in passi, ed essendo la natura della camminata armonica, ogni passo avrà una serie temporale molto simile al precedente.

## 1.1 Benchmark Dataset

Un problema rilevante nello studio di queste tematiche è la mancanza di dataset vari e completi. Il primo dataset valutato è stato lo Stride-Length Estimation (SLE) [6], che contiene più di 10000 passi, registrati da vari soggetti. I lati negativi di questo dataset sono la posizione fissa del telefono durante la registrazione, il telefono è tenuto in posizione "HandHeld" (Figura 1.2a), ed il fatto che contenga non solo camminata, ma anche altri movimenti (es. corsa, scale, salti) che sporcano la predizione.

Per questi motivi abbiamo deciso di utilizzare un altro dataset: il Walking Distance Estimation (WDE) [7]. Il dataset contiene più di 10.000 passi registrati da vari soggetti di età, sesso, altezza e peso variabili. Il dataset è stato acquisito con una frequenza di campionamento di 100Hz direttamente dallo smartphone che include un accelerometro (range:  $\pm 8g$ ) ed un giroscopio (range:  $\pm 2000^\circ/s$ ). In particolare questo dataset include varie modalità di trasporto dello smarphone tipiche della vita di tutti i giorni: Handheld, Arm-hand, Pocket, Calling, and Swing (Figura 1.2) e nel dataset è anche presente il *ground truth*, cioè la vera lunghezza del passo associata ad ogni misurazione.



**Figura 1.2:** Modalità di trasporto: **a.** Handheld: tenuto in mano all'altezza del petto con lo schermo rivolto verso l'alto come per una chiamata in vivavoce; **b.** Calling: tenuto all'orecchio per emulare una chiamata; **c.** Pocket: nella tasca di un pantalone; **d.** Swing: in mano mentre oscilla durante la camminata; **e.** Armhand: attaccato al braccio, poco sotto la spalla, come i supporti per la corsa.

Il *ground truth* è stata registrato tramite un modulo xIMU posizionato sul piede, tramite il quale è possibile risalire alla vera lunghezza del passo e determinare anche ogni *step event* tramite i quali il dataset è stato segmentato in passi.

Il dataset è già segmentato per ogni passo; è fornito in file di testo, dove ogni riga è uno stride in formato JSON (JavaScript Object Notation), un formato molto chiaro che trova un riscontro uno ad uno con la struttura dati del "dizionario", presente in Python, il linguaggio utilizzato per l'implementazione del lavoro di questa tesi.

Ogni passo fornito dal dataset ha una struttura rappresentata in Figura 1.3.

```

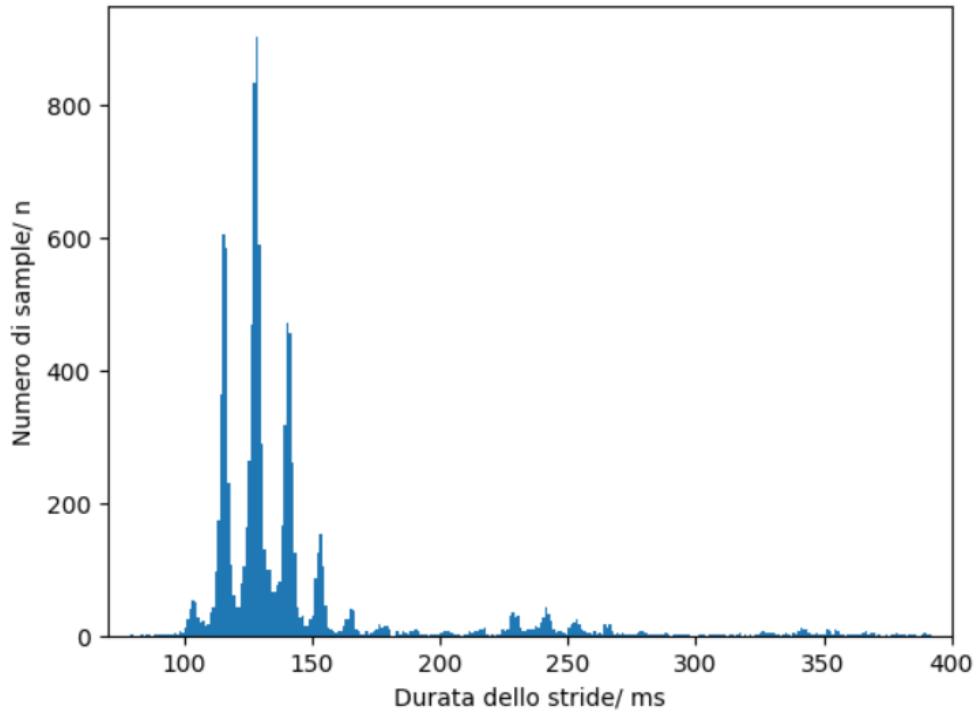
JSON
└── stride-count = float
└── stride-length = float
└── walkingdistance = float
└── mode = str
└── sensors
    └── timestamp = list
    └── acc = dict
        └── acc-x = list
        └── acc-y = list
        └── acc-z = list
    └── gyro = dict
        └── gyro-x = list
        └── gyro-y = list
        └── gyro-z = list
    └── magnetic = dict
        └── mag-x = list
        └── mag-y = list
        └── mag-z = list

```

**Figura 1.3:** Visualizzazione della struttura dati.

Il dataset è composto di dati grezzi, di conseguenza è stato necessario un lavoro di pulizia e formattazione. Il dataset è quindi stato sottoposto ad una pulizia dei dati, eliminando le misurazioni che non sono state eseguite correttamente a causa di qualche malfunzionamento dei sensori o per altre cause. Come evidenziato da vari studi [1] la durata di un passo non supera, in caso di camminata, i 3 secondi; il che vuol dire con una frequenza di campionamento di 100Hz non deve superare i 300 sample. Guardando il grafico della distribuzione della durata di ogni passo registrato (Figura 1.4) si nota che la quasi totalità dei passi contengono meno di 300 sample, ed una minuscola percentuale contiene invece 400 o più sample; questi valori contengono evidentemente degli errori e di conseguenza vengono scartati.

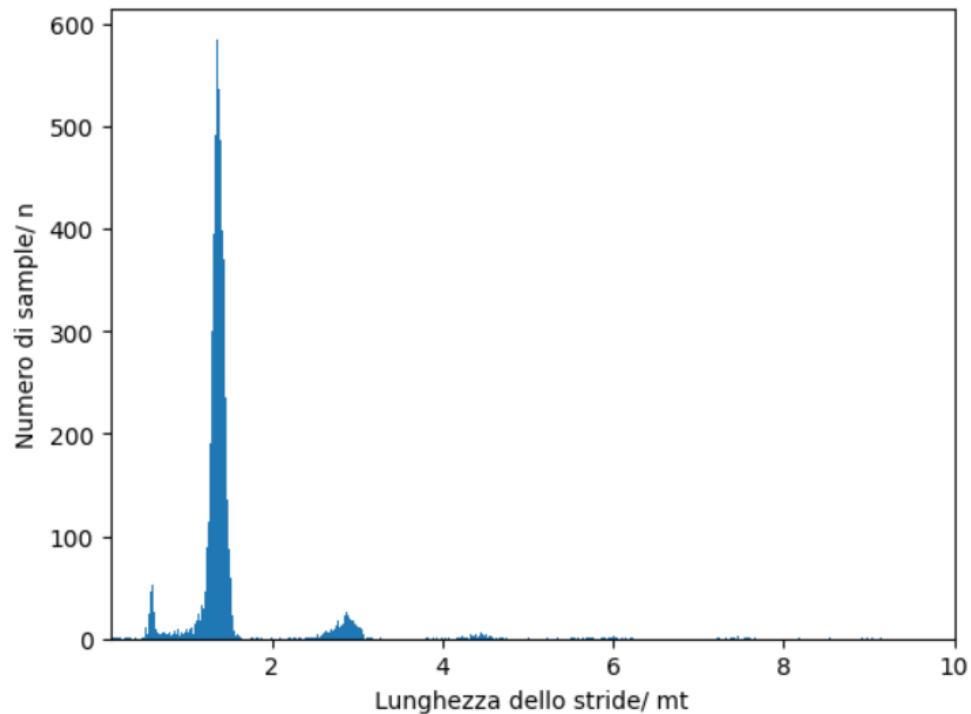
```
Registered Stride Duration: Min: 70 , Max: 4952
Media= 150.7752057148024; Deviazione standard= 106.65235046367869.
```



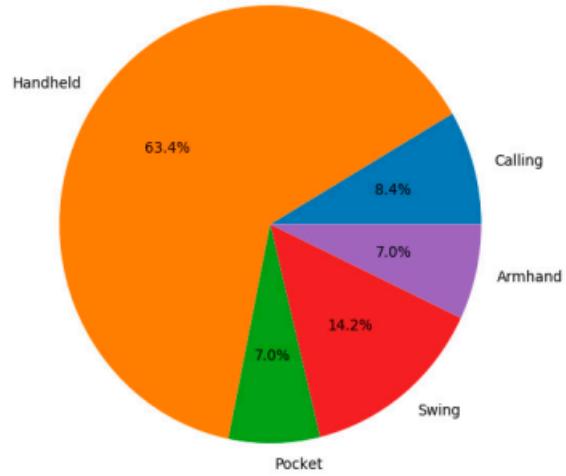
**Figura 1.4:** Ai fini di una migliore visualizzazione è stato limitato il range, ma è resa nota la presenza del massimo valore registrato. Come si vede dalla distribuzione la quasi totalità è sotto i 300ms, la presenza di questi outliers può essere dovuta al mancato rilevamento di un passo e quindi all'unione di segnali derivanti da due o più passi.

Un discorso analogo viene fatto per quanto riguarda la lunghezza del passo, che è contenuta tra 0 e 2 metri. Dalla Figura 1.5 vediamo la distribuzione della vera lunghezza del passo. Con una media di 1.57mt ed una deviazione standard di 1.30mt, la lunghezza registrata dei nostri passi rispetta le aspettative, ad eccezione di pochissimi valori di molto sopra i 2mt; questi valori sono quindi da considerarsi poco attendibili e dunque eliminati. Infine in Figura 1.6 si evidenzia la distribuzione di ognuna delle cinque modalità nell'intero dataset.

Registered Stride lenght: Min: 0.10830579656496883 , Max: 69.59508110908523  
Media= 1.5706674796266447; Deviazione standard= 1.3004713170943305.



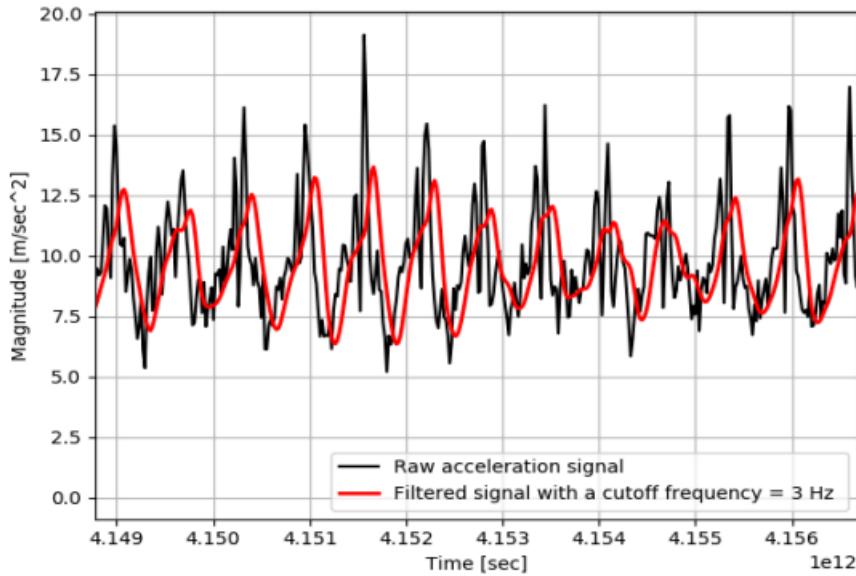
**Figura 1.5:** Ai fini di una migliore visualizzazione è stato limitato il range, ma è resa nota la presenza del massimo valore registrato. Come si vede dalla distribuzione la quasi totalità è sotto i 2mt, la presenza di questi outliers può essere dovuta al mancato rilevamento di un passo e quindi all'unione di segnali derivanti da due o più passi o alla rilevazione sbagliata di un passo col risultato di avere un passo estremamente ridotto



**Figura 1.6:** Distribuzione nel dataset delle varie modalità.

## 1.2 Pre-processamento dei dati

Avendo ora pulito il dataset dai dati fallaci, possiamo proseguire con il pre-processamento dei dati. I dati raccolti dai sensori contengono molto rumore, dovuto alle oscillazioni ad alta frequenza derivanti dall'ambiente esterno o dalla vibrazione del telefono e questo sporca l'oscillazione pulita della camminata umana. Secondo [10] per minimizzare l'apporto del rumore del sensore e la vibrazione dello smartphone e migliorare la robustezza del lavoro futuro bisogna filtrare i dati in ingresso, ed essendo la camminata umana un fenomeno armonico a bassa frequenza, abbiamo usato per la pulizia della serie temporale un Butterworth filter del primo ordine con una frequenza di taglio di 3Hz, estraendo così un segnale molto più pulito Figura 1.7.



**Figura 1.7:** Composizione dell'accelerazione lungo i tre assi, il segnale filtrato è molto più affidabile rispetto a quello grezzo.

**Formattazione dei dati** I dati sono presenti nel dataset come una lista di elementi, dove ogni elemento ha la struttura descritta in Figura 1.3, questi vengono caricati in una struttura dati DATASET, che ha la seguente struttura: (Figura 1.8)

```
DATASET = dict
    "handheld"
        └ LIST_OF = formatSLE
    "calling"
        └ LIST_OF = formatSLE
    "pocket"
        └ LIST_OF = formatSLE
    "swing"
        └ LIST_OF = formatSLE
    "armhand"
        └ LIST_OF = formatSLE
```

**Figura 1.8:** Struttura dati di DATASET, è un dizionario diviso per modalità di trasporto, come valori ha una lista di elementi con formato formatSLE Figura 1.9

Viene quindi importato in un dizionario che ha come chiave la modalità e come valore una lista contenente le registrazioni dei sensori nel seguente formato: Figura 1.9.

```

formatSLE = dict
    "target"= float
    SensorTimestamp = list
        Acc-X = list
        Acc-Y = list
        Acc-Z = list
        Gyr-X = list
        Gyr-Y = list
        Gyr-Z = list

```

**Figura 1.9:** Struttura di formatSLE, l'elemento di cui è composta la lista di ogni modalità in DATASET Figura 1.8

Viene infine equilibrato il dataset in modo tale che ogni modalità abbia lo stesso numero di stride preso in maniera eguale da ogni soggetto, in modo tale da avere un dataset equilibrato sia per quanto riguarda le modalità sia per quanto riguarda la varietà dei soggetti nelle singole modalità. Viene inoltre isolato un soggetto per ogni modalità per avere un dataset di test.

### 1.3 Analisi delle feature

Un passo può avere fino a 300 sample, per 2 sensori (accelerometro e giroscopio), per i 3 assi di ogni sensore, generando una quantità di dati importante e difficile da usare. Per questo motivo viene effettuata l'estrazione delle feature.

Ogni passo ha associate 6 serie temporali (2 sensori a 3 assi), ogni serie temporale può essere descritta utilizzando un buon set di feature che viene descritto nel paragrafo successivo.

**Estrazione delle Feature** Per evidenziare variazioni sia temporali che frequenziali sono state selezionate varie feature:

- **Feature statistiche:** Media del segnale, deviazione standard, skewness, kurtosis, interquartile range per una descrizione statistica del segnale.

- **Feature nel dominio del tempo:** Magnitude area, zero crossing per una destrizione nel dominio del tempo.
- **Feature nel dominio della frequenza:** Frequenza ed ampiezza della prima e della seconda dominante della FFT del segnale e autocorrelazione tra l'accelerometro ed il giroscopio, per avere una visione delle frequenze dominanti.
- **Feature di alto livello:** Queste feature provengono da lavori precedentemente citati e come suggerito da [7], sono feature che stimano la lunghezza del passo a partire da delle ipotesi sul range dell'ampiezza della serie temporale ognuna con un determinato peso che viene calcolato dal modello e possono fornire informazioni importanti.

Le feature in questione sono:

Scarlett [4] che calcola la lunghezza del passo secondo la formula:

$$L_S = \frac{\sum_{i=1}^N |a_i| - a_{min}}{a_{max} - a_{min}} \quad (1.1)$$

Kim [3] che calcola la lunghezza del passo secondo la formula:

$$L_K = \sqrt[3]{\frac{\sum_{i=1}^N |a_i|}{N}} \quad (1.2)$$

Weinberg [2] che calcola la lunghezza del passo secondo la formula:

$$L_W = \sqrt[4]{a_{max} - a_{min}} \quad (1.3)$$

Dove  $a_{max}$  ed  $a_{min}$  sono il valore massimo e minimo dell'accelerazione durante lo stride,  $a_i$  rappresenta l'accelerazione al sample  $i$ -esimo ed  $N$  il numero totale di sample.

Avremo così ottenuto un vettore contenente in tutto 92 feature che descrive in maniera sintetica e quanto più fedele possibile il passo.

La nostra struttura dati finale che verrà quindi usata nei modelli ha la struttura di una lista dove ogni elemento rappresenta un passo e sarà un vettore contenente 92 feature che descrivono tutte le serie temporali misurate dai sensori durante quel passo Figura 1.10 .

```

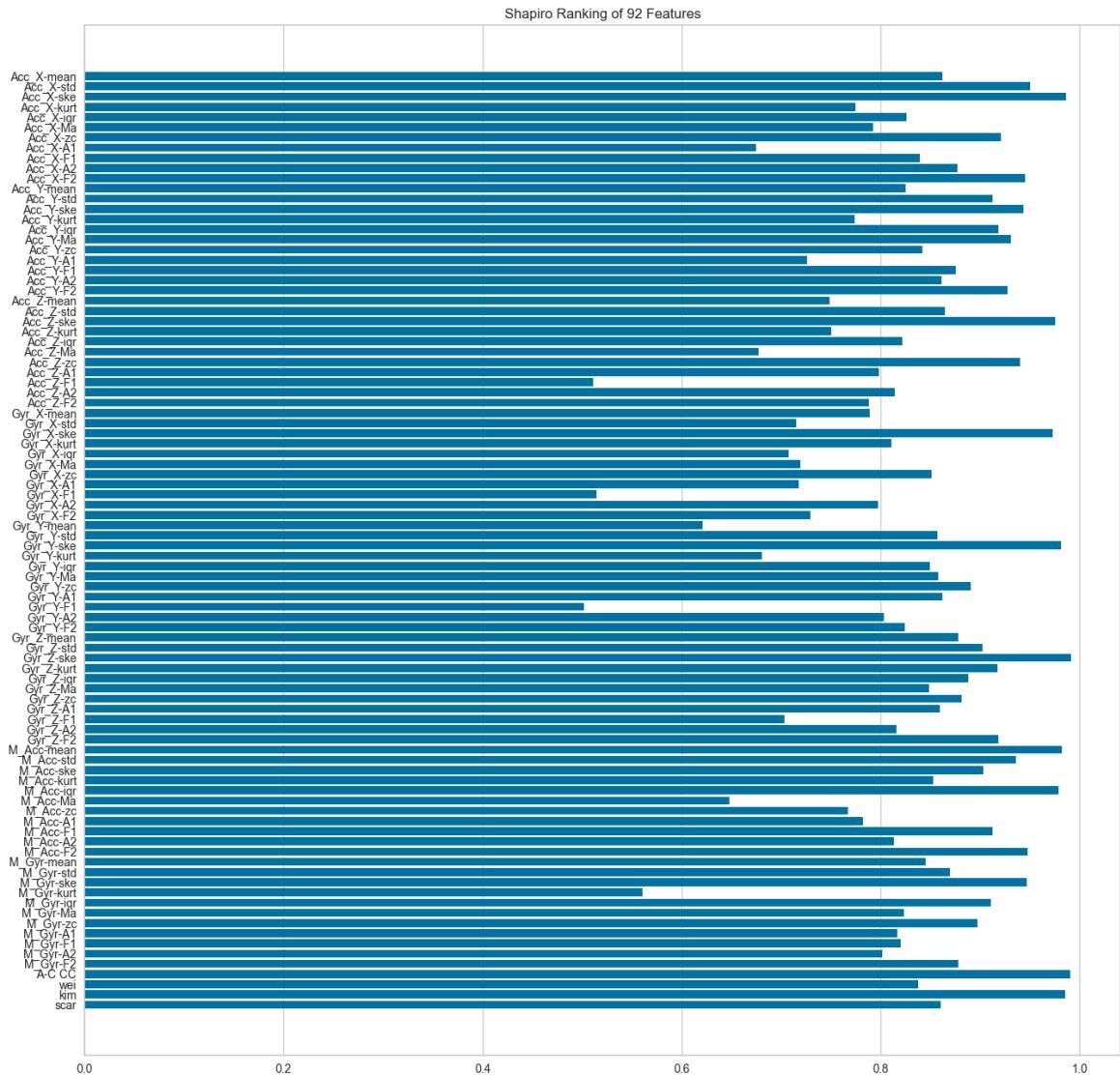
Final-Dataset = dict
    "handheld" = dict
        "target" = list of float 500
        "feature" = list of feature vector500
            -92 float
    "calling" = dict
        "target" = list of float 500
        "feature" = list of feature vector500
            -92 float
    "pocket" = dict
        "target" = list of float 500
        "feature" = list of feature vector500
            -92 float
    "swing" = dict
        "target" = list of float 500
        "feature" = list of feature vector500
            -92 float
    "armhand" = dict
        "target" = list of float 500
        "feature" = list of feature vector500
            -92 float

```

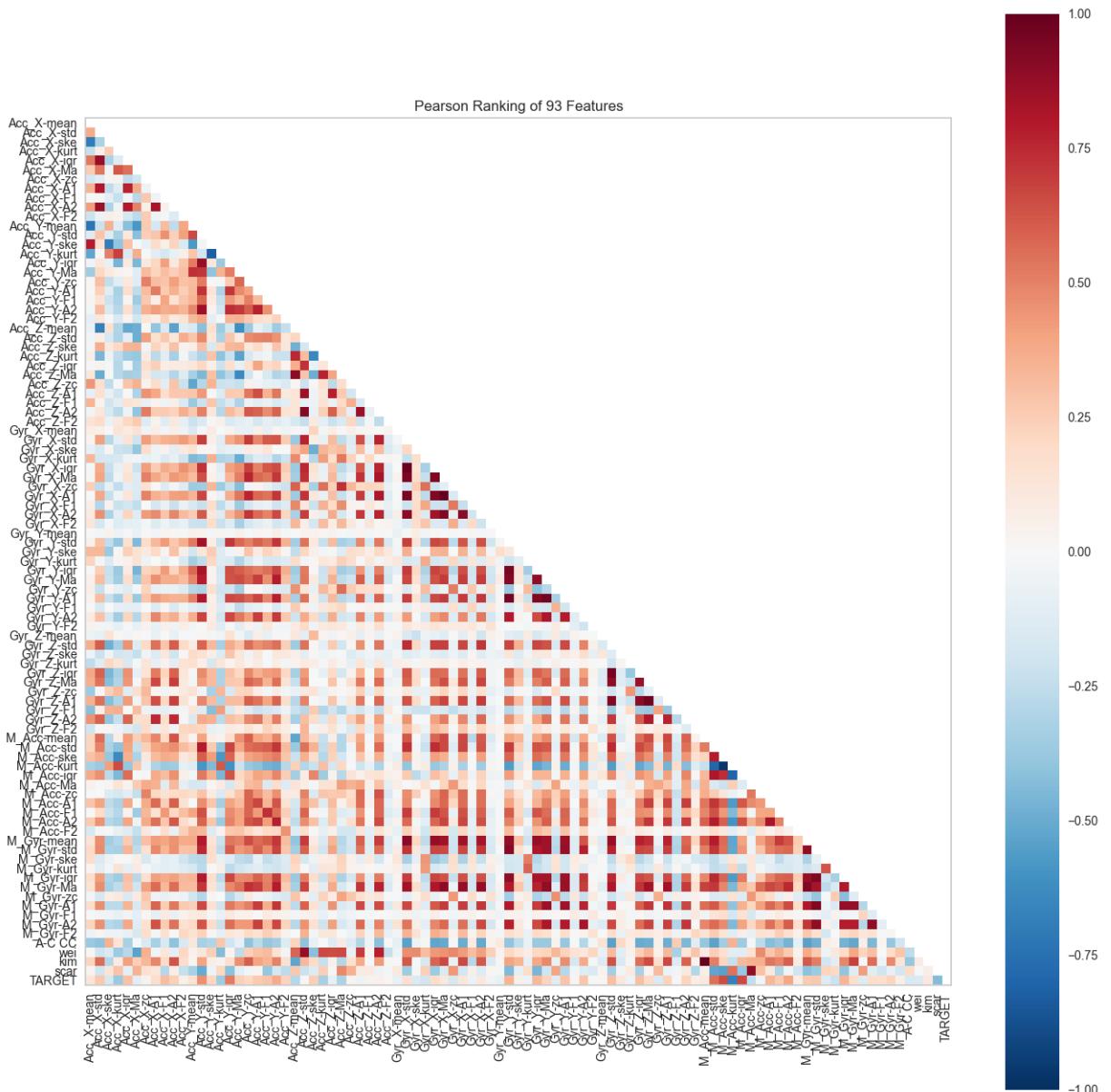
**Figura 1.10:** Struttura del dataset finale, equilibrato. Diviso per modalità, ogni modalità contiene due liste di un numero fissato di elementi (500), una lista ha il label ”target” ed è una lista di float, l’altra ha il label ”feature” ed è una lista di vettori di feature, dove ogni vettore di feature ha 92 elementi, che sono dei float, e sono il risultato della nostra feature extraction di ogni passo.

**Analisi delle feature** Ottenuto il nostro set di feature è importante verificare quanto queste sono informative per il nostro problema, è importante valutarne la qualità. Questo è stato fatto utilizzando la libreria yellowbrick [11], ed in particolare le funzioni **rank1d** e **rank2d**.

- **rank1d** utilizza l'algoritmo di Shapiro implementato da `scipy.stats` [12] che assegna uno score alla feature in base alla probabilità che questa abbia un'istanza nulla. In Figura 1.11 lo score di ogni feature dal quale notiamo come tutte le feature siano informative; non abbiamo aggiunto feature poco informative ovvero non abbiamo aggiunto feature che in media hanno valore nullo molto frequente.
- **rank2d** valuta lo score per ogni coppia di feature, lo scoring usato è l'algoritmo di Pearson, che verifica quanto le due feature sono correlate tra loro calcolato come la covarianza delle due feature diviso il prodotto della loro deviazione standard. In Figura 1.12 la heatmap indica l'entità della correlazione tra le singole feature.



**Figura 1.11:** Sull'asse delle y il nome delle feature e nell'asse delle x il valore del ranking 1d.



**Figura 1.12:** In questa figura è stato aggiunto il TARGET, ovvero la lunghezza dello stride per valutare quanto ogni feature è correlata al valore che vogliamo ottenere.

Si nota la presenza di molte coppie di feature correlate tra loro come le coppie che riguardano l' interquartile range e la Magnitude Area del vettore Girosc-

pio (M\_Gyr\_iqr, M\_Gyr\_Ma) e quindi potrebbero fornire poca informazione aggiuntiva se prese entrambe, mentre altre coppie sono molto poco correlate, vedi per esempio le coppie con la frequenza della seconda dominante del Giroscopio nell'asse Y (Gyr\_Y\_F2). Per questo motivo affronteremo il tema della feature selection.

Per quanto riguarda invece la correlazione tra le feature ed il target vediamo che spiccano per esempio la deviazione standard, la skewness, la kurtosis del vettore Accelerazione (M\_Acc\_std, M\_Acc\_ske, M\_Acc\_kurt).

Terminata l'analisi delle feature, proseguiamo applicando le tecniche di Machine Learning presentate nei capitoli successivi tra le quali sarà presente anche una sezione dedicata alla selezione delle feature.

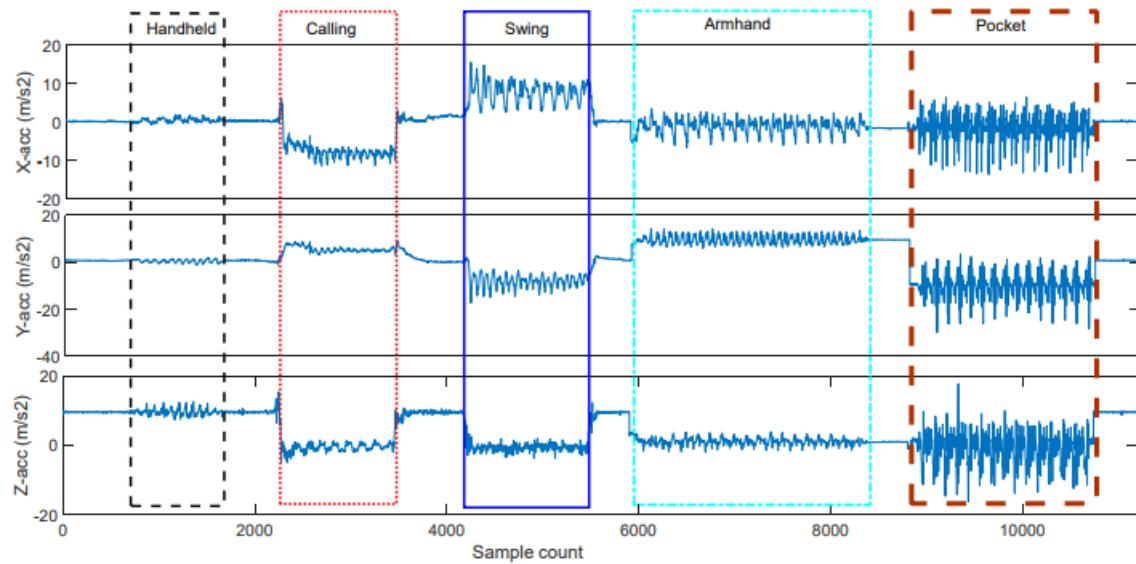
## **Capitolo 2**

# **Classificazione delle modalità di posizionamento dello smartphone**

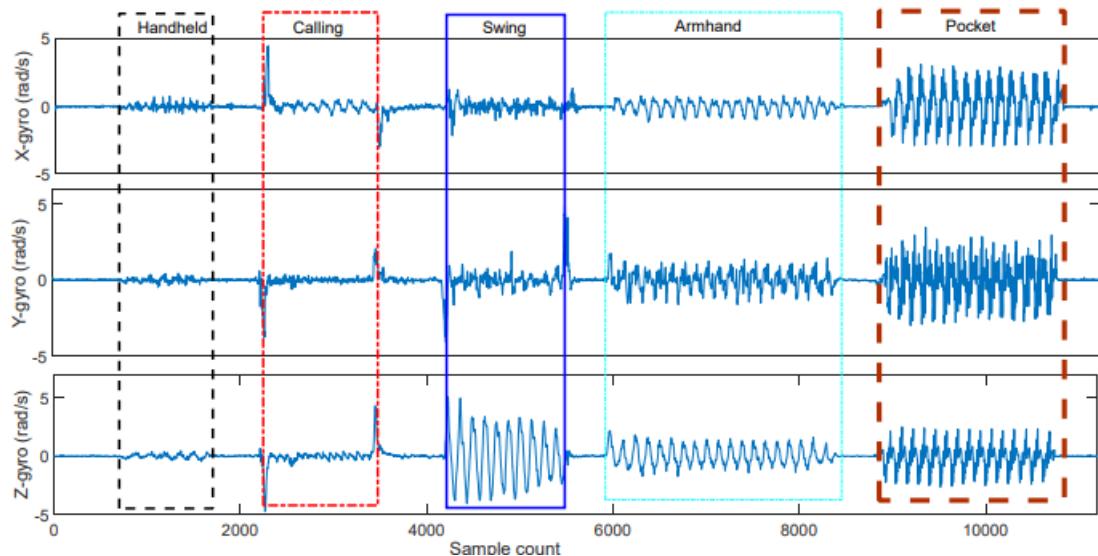
Dopo aver pre-processato i dati, estratto le feature e valutato la loro bontà, possiamo procedere con la classificazione della modalità di trasporto del telefono.

Il nostro dataset contiene varie modalità di trasporto Figura (1.2) che generano serie temporali profondamente diverse come evidenziato dalla Figura 2.2.

Usando quindi il set di feature estratto abbiamo utilizzato un semplice classificatore per classificare le cinque modalità.



**Figura 2.1:** Differenza nelle misurazioni dell' accelerometro



**Figura 2.2:** Differenza nelle misurazioni del giroscopio

## 2.1 Cenni di teoria

**Classificazione** La Classificazione è un insieme di problemi di Machine Learning in cui è necessario catalogare un set di dati o di osservazioni in categorie prestabilite. Il nostro è un chiaro esempio di classificazione, in quanto abbiamo cinque modalità ben definite che generano serie temporali diverse, e da queste serie temporali si deve dedurre a che modalità di trasporto si riferiscono.

In generale un problema di classificazione è un problema in cui si deve trovare una funzione ( $f$ ) in grado di mappare una variabile di input continua ( $X$ ) in una variabile di output di un insieme discreto ( $y$ ).

Per poter utilizzare un algoritmo di ML è necessario disporre di un dataset di train con cui allenare il modello, ed un dataset di test, dello stesso formato del train, con cui testare la bontà del nostro modello. La classificazione appartiene alla categoria del "supervised learning", ovvero l'insieme degli algoritmi in cui è necessario fornire in input anche i dati di target relativi al dataset di train in modo tale da poter allenare il modello fornendo già il risultato che dovrebbe ottenere.

Esistono due filosofie per allenare il modello negli algoritmi di classificazione [13]:

- **Lazy:** Nel momento in cui arriva un dato di test si cerca nel dataset di train i dati correlati maggiormente ed in base a quelli si fa la classificazione. Tempi di train brevi, ma tempi di predict lunghi.
- **Eager:** Un modello più generale possibile viene allenato ed è in grado di ridurre la classificazione ad una sola ipotesi. Ne deriva che ha tempi di train più lunghi e tempi di predict più brevi.

L'algoritmo di classificazione scelto è uno dei più noti di tipo Eager ovvero il DecisionTree, in quanto per la nostra applicazione sono molto più importanti i tempi di test rispetto quelli di train.

**DecisionTree** Il DecisionTree (DT) è un algoritmo che attua la classificazione mediante un albero. L'albero è costruito dall'alto verso il basso, ad ogni iterazione divide il ramo secondo una regola, durante il train tiene traccia di tutte le regole che portano ad ogni classe ed in questo modo arrivando alla foglia dell'albero classifica il dato in maniera univoca (Figura 2.3).



**Figura 2.3:** Struttura tipo di un DecisionTree

E' importante nel DT limitare il fenomeno dell'overfitting, ovvero il modello non deve essere troppo specifico per i dati di input, in quanto le foglie potrebbero classificare in base ad outliers o rumore dei dati di partenza.

## 2.2 Applicazione del DecisionTree

I dati in ingresso al DecisionTree sono ottenuti come specificato nel Capitolo 1, vengono selezionati 500 passi per ogni modalità presi da  $n - 1$  soggetti disponibili e 50 passi per ogni modalità dal soggetto rimasto. La nostra variabile di input è l'insieme di tutti i vettori di feature, ed il target è il label che corrisponde alla modalità del rispettivo vettore di feature.

Uno step fondamentale nell'utilizzo di algoritmi di ML, nel nostro caso DT

è la regolazione dei parametri. Facendo riferimento al classificatore fornito dalla libreria sklearn [14] DecisionTreeClassifiers i parametri fondamentali sono ”splitter”, ”min\_samples\_split”, ”min\_sample\_leaf”, ”max\_depth”.

- ”**splitter**” definisce la regola con la quale generare le ramificazioni dell’albero; ho selezionato ”best” in quanto il nostro modello ha molte feature ed è conveniente valutare di volta in volta le più informative.
- ”**min\_samples\_split**” definisce qual è il numero minimo di sample che deve avere un nodo per poter essere diviso, il minimo è 2 ed il massimo valore consigliato è 40.
- ”**min\_sample\_leaf**” definisce qual è il numero minimo di sample che deve avere un nodo per essere considerato una foglia, il range di valori consigliato è tra 1 e 20.
- ”**max\_depth**” è la massima profondità a cui deve scendere l’albero, ho lasciato che fosse il modello a deciderlo in base ai valori di min\_sample\_leaf e min\_sample\_split.

I parametri necessitano di una regolazione, pur conoscendo un range in cui è valido il parametro bisogna trovare il miglior parametro per la nostra applicazione. E’ un problema di selezione del modello, in cui, definito un algoritmo, dobbiamo trovare il modello che produce lo score migliore tramite la regolazione dei suoi parametri.

La ricerca e selezione del miglior modello è stata eseguita tramite la funzione **GridSearchCV**[14] e consiste in:

- Un modello da ottimizzare;
- Un range di parametri da testare;
- Uno schema di *cross-validation*;
- Una funzione per definire lo score sul quale effettuare la selezione.

Abbiamo definito quindi il modello, il DT; i parametri da testare ed il loro range, min\_sample\_leaf e min\_sample\_split.

La funzione scelta per lo scoring della classificazione è l’*Accuracy*, la misura che identifica la percentuale di dati classificata correttamente, in riferimento

alla Figura 2.4 la formula per l'Accuracy è:

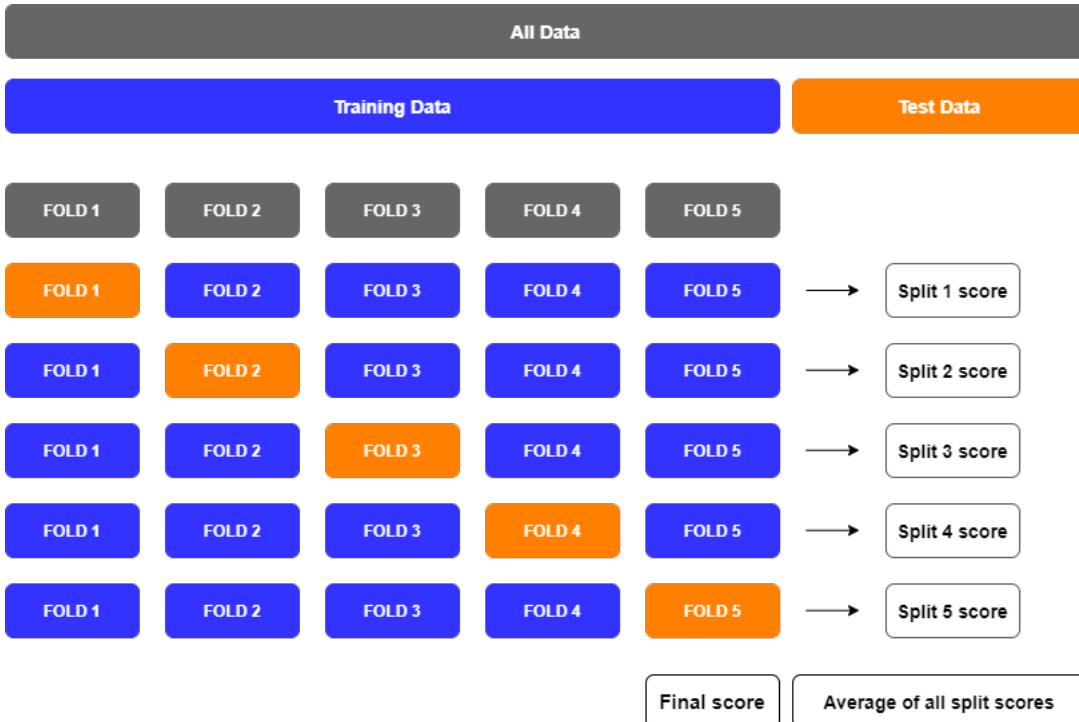
$$Acc = \frac{TruePositives + TrueNegatives}{TruePositives + FalsePositives + TrueNegatives + FalseNegatives} \quad (2.1)$$

		Predicted (Clusters)	
		Positive	Negative
Actual (Classes)	Positive	TP	FN
	Negative	FP	TN

**Figura 2.4:** Tabella tipica per la valutazione di un classificatore.

**Cross-validation** La **Cross-validation** è uno schema con cui si divide il dataset a disposizione in modo tale da evitare l'overfitting ed evitare errori metodologici. E' infatti un errore metodologico usare dati del train per testare il modello; il modello deve essere valutato sulla base di dati che non ha mai visto e su quelli valutare le performance del modello. La cross-validation è una metodologia per dividere i dati di input in dati di train e dati di test in modo tale che questi siano sempre vari e mai sovrapposti e con questi si possa valutare il modello.

Nello schema scelto, la **K-Fold cross-validation** questo viene fatto prendendo i dati di Training e dividendoli in un numero prestabilito, K, di sezioni dette *Fold*. Vengono eseguite un numero di iterazioni, K, pari al numero di Fold ed ad ogni iterazione, detta *Split*, vengono usati  $K - 1$  Fold per allenare il modello ed il rimanente Fold per testare il modello. In questo modo, facendo la media dello score di ogni iterazione è possibile valutare il modello in maniera più completa ed oggettiva Figura 2.5.



**Figura 2.5:** Il dataset iniziale viene suddiviso in training e test, a sua volta il dataset di training viene diviso in fold e viene applicata la cross-validation. Ad ogni split viene associato il suo score, la media degli score è lo score della cross validation. Per la verifica finale il modello deve essere testato sul dataset di test.

**GridSearchCV** Avendo definito ciò che necessita per funzionare esplicitiamo il comportamento del GridSearchCV.

L'obiettivo è ottenere il miglior modello, quello con lo scoring maggiore, scegliendo da un set di parametri disponibili i valori dei parametri che generano il modello migliore.

La funzione quindi esegue per ogni possibile combinazione di parametri (*nel nostro caso min\_sample\_leaf ha range [1..30] e min\_sample\_split [2..40], per un totale di 780 combinazioni*) una KFold cross-validation (nel nostro caso con  $K = 10$ ), ed infine valuta i risultati di tutte le KFold cross-validation, restituendo un modello regolato con i parametri che hanno avuto il miglior riscontro Figura 2.6.

```

Fitting 10 folds for each of 780 candidates, totalling 7800 fits
[CV] END .....min_samples_leaf=1, min_samples_split=2; total time= 0.0s
[CV] END .....min_samples_leaf=1, min_samples_split=3; total time= 0.0s
[CV] END .....min_samples_leaf=1, min_samples_split=3; total time= 0.0s
[CV] END .....min_samples_leaf=1, min_samples_split=3; total time= 0.0s

```

**Figura 2.6:** Sezione dell'output del GridSearchCV durante la computazione. Dovendo effettuare la cross-validation per ogni combinazione di parametri la funzione ha effettuato 7800 fit e predict. Notare quindi che per la prima coppia di parametri (`min_sample_leaf=1` e `min_sample_split=2`) effettua 10 fit, e poi passa alla seconda coppia di parametri (`min_sample_leaf=1` e `min_sample_split=3`).

**Risultati** Con le metodologie descritte è stato possibile ottenere partendo dai dati di training il miglior set di parametri per il nostro modello Figura 2.7.

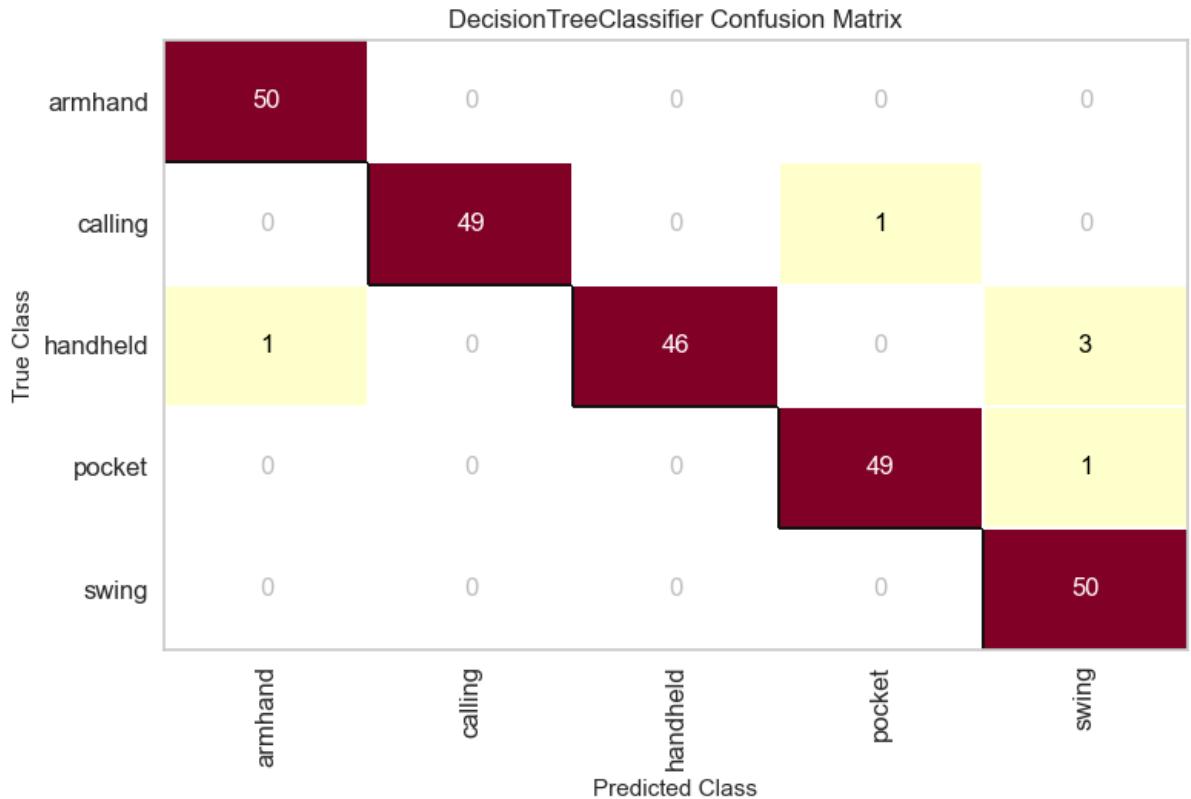
```

{'min_samples_leaf': 1, 'min_samples_split': 37}
Accuracy:0.9963999999999998

```

**Figura 2.7:** Da notare che il risultato della cross validation è comunque un risultato di train, di conseguenza deve essere valutato successivamente nel dataset di test.

A questo punto bisogna usare il dataset di test e valutare le performance del nostro modello selezionato. Il dataset di test contiene 50 passi prelevati da soggetti non presenti nel train. Per la classificazione abbiamo ottenuto ottimi risultati Figura 2.8 classificando in maniera corretta più del 98% dei dati di test; Dalla Figura 2.9 è possibile vedere lo score anche in base ad altri metodi di scoring: precision, recall, F-1.



**Figura 2.8:** Confusion Matrix relativa alla classificazione.

	precision	recall	f1-score	support
armhand	0.98	1.00	0.99	50
calling	1.00	0.98	0.99	50
handheld	1.00	0.92	0.96	50
pocket	0.98	0.98	0.98	50
swing	0.93	1.00	0.96	50
accuracy			0.98	250
macro avg	0.98	0.98	0.98	250
weighted avg	0.98	0.98	0.98	250

**Figura 2.9:** Classification report, in cui è possibile valutare le performance anche in base ad altri criteri

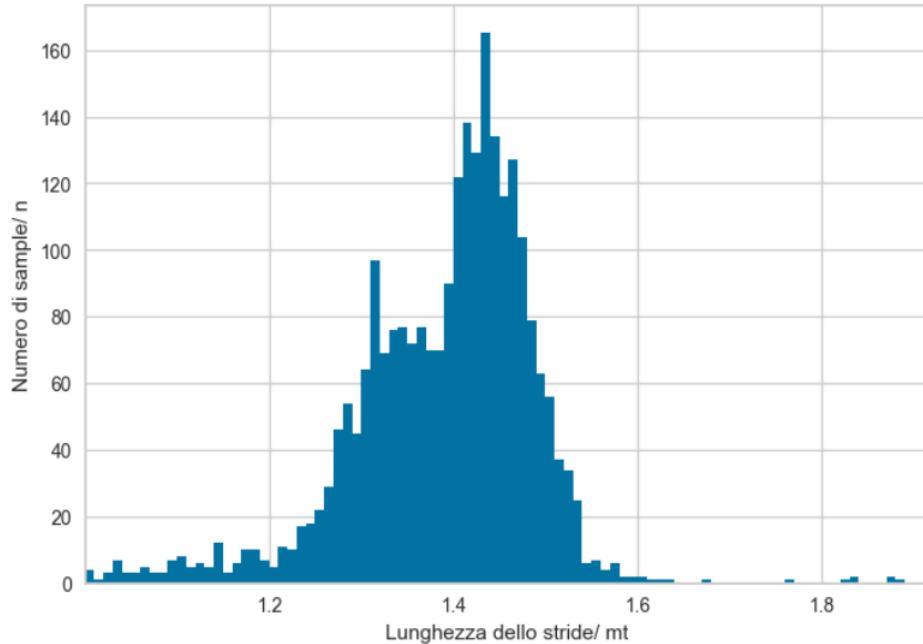
## Capitolo 3

# Regressione della lunghezza del passo

Utilizzando gli stessi dati di input, la lista di vettori di feature calcolata nel Capitolo 1 possiamo concentrare la nostra attenzione sul problema della regressione della lunghezza del passo. Il nostro obiettivo è dunque quello di, a partire da un vettore di feature che definisce un passo, ottenerne la lunghezza.

### Distribuzione del target senza distinzione di modalità

Registered Stride lenght: Min: 1.000034177353732 , Max: 1.917364683119593  
 Media= 1.3906684693896598; Deviazione standard= 0.09820942152593022.



**Figura 3.1:** Istogramma che rappresenta la distribuzione della lunghezza del passo registrata, il nostro target per la regressione. Rispetto alla Figura 1.5 i dati sono puliti, con una media di 1.39m ed una deviazione standard di 0.09m.

## 3.1 Cenni di teoria

**Regressione** La Regressione è il processo della stima della relazione tra un set di variabili indipendenti ed un set di variabili dipendenti. La regressione è dunque usata per predire, a partire da una serie di dati o osservazioni indipendenti un fenomeno che li accomuna e predirre il risultato del determinato fenomeno nel caso di dati mai visti, ma della stessa classe. In breve si tratta di ottimizzare, rispetto ad una funzione di errore, una funzione ( $f$ ) su dei dati ( $X$ ) in modo tale da poter predirre i risultati, o target ( $y$ ).

Dal punto di vista matematico si tratta quindi di stimare una funzione  $f_{\beta}(\cdot)$  (dipendente da  $\beta$ ) a partire da un set di dati iniziale (dati,  $x_i$  e target,

$y_i$ ) in modo tale da **minimizzare** la funzione di costo (*Loss Function*,  $l$ ) calcolata tra  $f_\beta(x)$  e  $y$ :

$$\min \sum_i l(f_\beta(x_i), y_i) \quad (3.1)$$

E' importante notare come il target della regressione, al contrario del target della classificazione, è un target *continuo* ovvero che può assumere qualsiasi valore; la lunghezza del passo infatti non può essere solo 1m o 2m, ma anche tutti gli infiniti numeri contenuti nel range.

Per questo lavoro di tesi abbiamo deciso di colmare un vuoto presente in letteratura sull'argomento, ed abbiamo investigato le performance del più semplice regressore: il **Rettangolare Lineare**.

**Linear Regressor** Nella regressione lineare l'obiettivo è trovare una retta tramite la minimizzazione della somma dell'errore quadratico medio di ogni dato con la retta stessa. E' il più semplice regressore, per questo è anche il più veloce e meno dispendioso dal punto di vista dell'hardware. Nella libreria usata in questo lavoro di tesi [14] viene implementato tramite la funzione LinearRegressor che è in grado di trovare la retta che approssima i dati anche nel caso di più dati in input (ricordiamo che nel nostro caso sono 92 feature per ogni passo), ed assegna ad ogni feature un peso ( $\beta$ ) tramite il quale calcola il valore predetto. L'equazione matematica che descrive un regressore lineare è la seguente:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_n x_n \quad (3.2)$$

dove le  $\beta_i$  sono i pesi riferiti alle  $n$  feature  $x_i$  accessibili tramite l'attributo .COEF\_ e  $\beta_0$  è il bias accessibile tramite l'attributo .INTERCEPT\_.

## 3.2 Applicazione del Modello Lineare

Usando i dati pre-processati e fromattati come visto nel Capitolo 1, i nostri dati in ingresso al modello saranno divisi: dati di train, due vettori di 500 elementi per ogni modalità uno per le feature (quindi un vettore bidimensionale di 2500 per 92 elementi) ed uno per i target (quindi un vettore unidimensionale di 2500 elementi) collezionati da vari soggetti; dati di test, due vettori analoghi ma di 50 elementi per modalità utilizzati per il test e

presi da un soggetto non considerato nel dataset di train. Il target in questo caso è la **lunghezza del passo**.

Il regressore lineare essendo un modello semplice non ha nessun parametro da regolare, tuttavia in alcuni casi potrebbe giovare della presenza di un *minore* numero di feature. A questo scopo abbiamo utilizzato una funzione di feature selection per selezionare solo le migliori feature. La funzione in questione è SelectKBest [14].

**SelectKBest** SelectKBest è una funzione che seleziona le migliori K feature, secondo uno *scoring method*, e scarta le altre. Il metodo di scoring utilizzato per la classificazione è lo stesso utilizzato nell'analisi delle feature, rank2d, ovvero viene calcolato lo score tramite l'algoritmo di Pearson tra ogni feature ed il target (Figura 1.12). Avendo aggiunto l'algoritmo di SelectKBest adesso abbiamo un parametro da regolare, K, ovvero il numero di top feature da selezionare, ed a questo scopo utilizzeremo il GridSearchCV GridSearchCV. La lista di parametri scelta per K è tutto lo spazio delle feature, quindi il modello verrà valutato usando da 1 a 92 feature.

**StandardScaler** A differenza del DecisionTree dove ogni feature viene valutata singolarmente per scegliere la regola, per il modello lineare è importante avere una visione di insieme delle feature. Per far sì che tutte le feature siano valutate correttamente queste devono essere scalate per far sì che la distribuzione delle feature risulti indicativamente una Gaussiana, con media zero e varianza unitaria.

Questo è fatto per far sì che fenomeni diversi abbiano importanza diversa, spieghiamo con un esempio: Immaginiamo di avere due feature una di 1000mm ed una in 1km, il modello vedrà solo una feature di 1000 ed una di 1 ed ipotizzerà che essendo 1000 molto maggiore di 1 la prima feature sia più importante; ma se scalassimo entrambe le feature a metri allora la prima sarebbe 1 e la seconda sarebbe 1000 invertendo le priorità.

E' importante quindi fornire al modello una misura di ciò che significano i numeri, e questo viene fatto standardizzando tutte le feature ed in particolare lo StandardScaler rimuove la media e scala la feature a varianza unitaria secondo questa formula:

$$z = \frac{(x - u)}{s} \quad (3.3)$$

Dove  $u$  è la media ed  $s$  è a deviazione standard della distribuzione della feature. E' inoltre importante, come per ogni modello, scalare le feature in base al dataset di train, ed usare lo stesso StandardScaler sul test, per far sì che le feature vengano scalate solo in base al dataset di train.

**Pipeline** A questo punto dobbiamo concatenare tre modelli per ottenere il nostro regressore: StandardScaler, SelectKBest, LinearRegressor. Per fare ciò usiamo la funzione Pipeline [14] che crea un unico oggetto concatenando una serie di modelli. Questo oggetto dovrà essere quindi ottimizzato tramite il GridSearchCV.

**Scoring method** Lo scoring method utilizzato per la valutazione del modello lineare è l' $R^2$  score, coefficiente di determinazione. Il range dello score è tra 0 ed 1, valori negativi significano che il modello sta approssimando i dati peggio di come li avrebbe approssimati una linea orizzontale; il valore 1 è o score migliore in che approssima perfettamente i dati, mentre lo 0 significa che approssima i dati come la media dei dati. La formula è la seguente:

- Data la media dei valori osservati:

$$\bar{y} = \frac{\sum_{i=1}^n y_i}{n} \quad (3.4)$$

- La somma dei quadrati dei residui:

$$SS_{res} = \sum_i (y_i - f_i)^2 \quad (3.5)$$

Dove  $f_i$  sono i valori predetti;

- La somma totale dei quadrati:

$$SS_{tot} = \sum_i (y_i - \bar{y})^2 \quad (3.6)$$

- il coefficiente di determinazione,  $R^2$ :

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}} \quad (3.7)$$

**Risultati** A questo punto il nostro modello può essere ottimizzato con metodologie analoghe a quanto visto in Capitolo 2 usando KFold cross-validation nella GridSearchCV.

Per valutare le performance del modello nello specifico della predizione, e quindi valutare l'errore in percentuale sul passo abbiamo usato l'errore assoluto percentuale (APE), calcolato tramite la seguente formula:

$$E_i = \frac{|P_i - T_i|}{T_i} \times 100\% \quad (3.8)$$

Dove  $E_i$ ,  $P_i$  e  $T_i$  sono rispettivamente l'errore in percentuale, il valore predetto ed il valore reale della lunghezza dell' $i$ -esimo passo.

Dalla Formula 3.8 calcoliamo la distribuzione dell'errore ed il **mean absolute percentage error (MAPE)** che è la media degli errori, la formula è la seguente:

$$MAPE = \frac{\sum_{i=1}^n E_i}{n} \quad (3.9)$$

Dove  $E_i$  è l'errore assoluto percentuale al passo  $i$ -esimo ed  $n$  è il numero totale di passi

Inoltre calcoleremo il bias per vedere se il modello in media sovrastima o sottostima il valore reale, calcolato tramite la seguente formula:

$$B = \frac{\sum_{i=1}^n T_i - P_i}{n} \quad (3.10)$$

Dove  $P_i$  e  $T_i$  sono rispettivamente l'errore in percentuale, il valore predetto ed il valore reale della lunghezza dell' $i$ -esimo passo ed  $n$  è il numero totale di passi.

Per quanto riguarda i dati a cui applicare le metodologie utilizzate abbiamo separato due casi: nel primo caso verrà creato un modello indipendente e verrà valutato l' $R^2$  score ed il MAPE per ogni modalità; nel secondo caso, essendo che come abbiamo visto nel Capitolo 2 è possibili classificare facilmente le cinque modalità di trasporto, verrà creato un modello ad hoc per ogni modalità e confrontati questi modelli con i risultati in termini di  $R^2$  e MAPE con il modello indipendente dalle modalità.

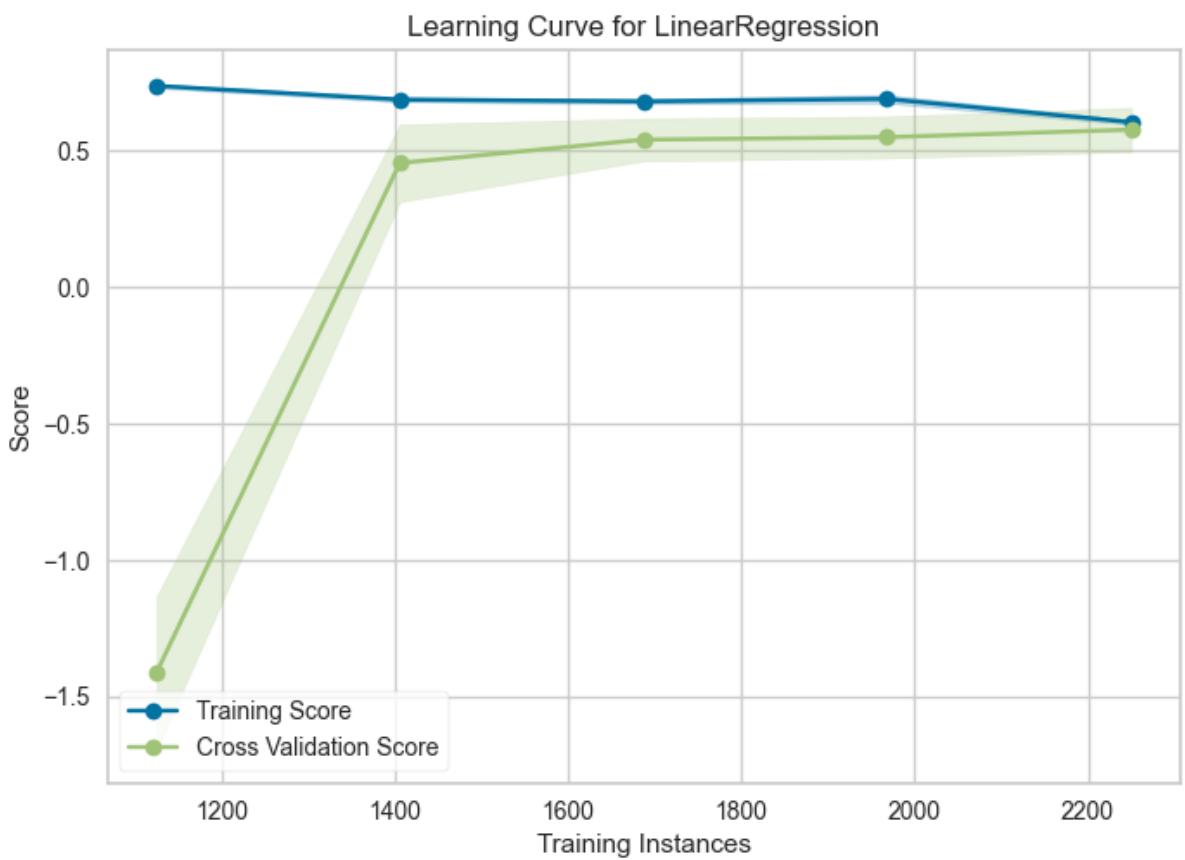
La valutazione dei modelli si comporrà delle seguenti fasi:

- **LearningCurve**, permette di vedere l'evouzione del modello al variare del numero di dati di train in ingresso ed è quindi possibile a partire dal grafico valutare se il modello necessita di più dati, se il modello sta overfittando i dati e se l'errore è dovuto più ad un bias od alla varianza.
- **Feature Importance**, visualizzare il miglior modello selezionato dal GridSearchCV quali feature ha considerato ed il peso  $\beta$  assegnato ad ogni feature.
- **Mean Test e Train score della GridSearchCV**, per valutare di quanto il miglior modello è migliore degli altri modelli.
- **Akaike Information Criterion**, Akaike Information Criterion(AIC) è un indice che serve a valutare quanto il modello sia informativo al variare del numero di feature.
- **Regression Visualizers sul TEST**, visualizzare tramite la libreria yellowbrick [11] le performance dei nostri modelli.
- **Valutazioni sul MAPE**, verranno valutati i risultati ottenuti dal MAPE.

### 3.2.1 Modello indipendente

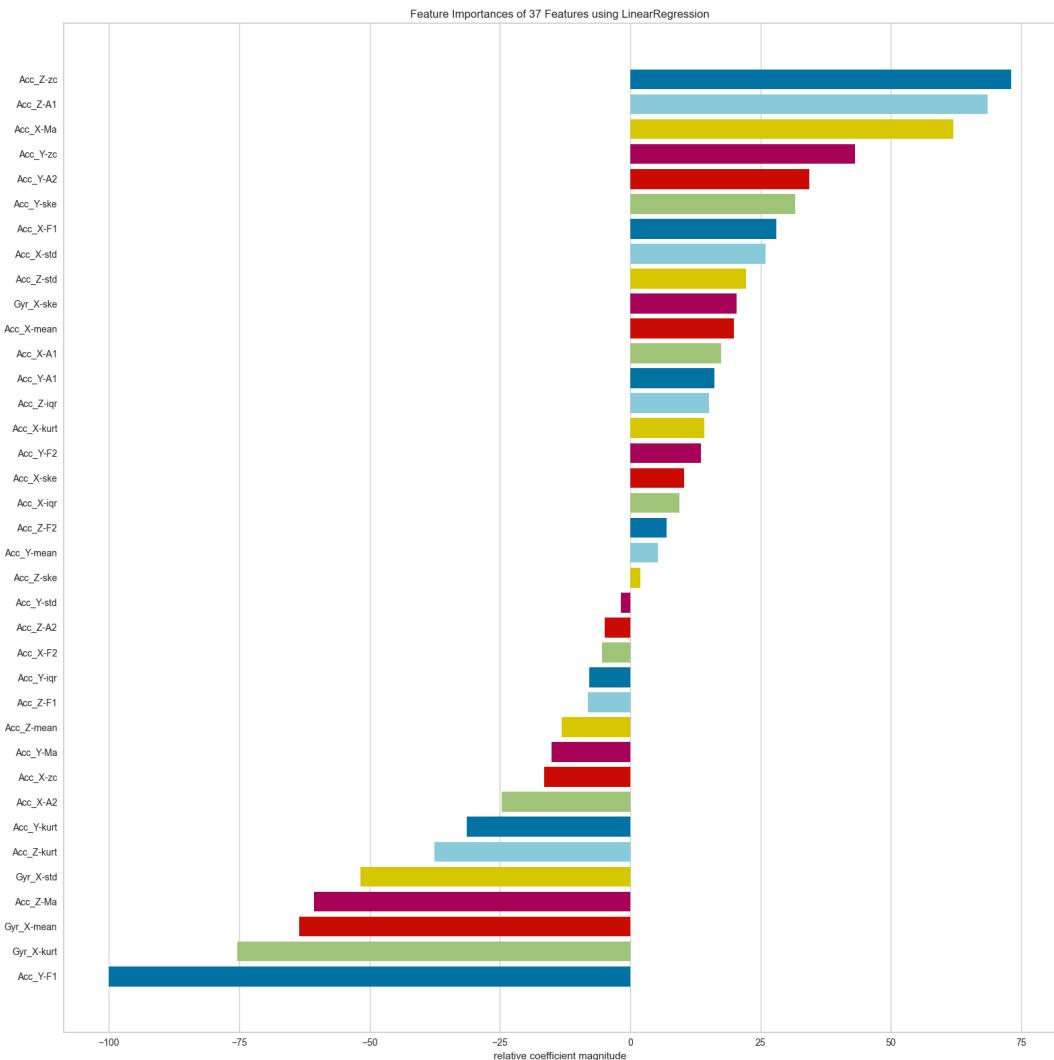
E' stato dunque usato l'intero dataset per il train ed il test, senza distinzione di modalità.

**Learning Curve** Dalla Figura 3.2 vediamo che il modello di sicuro necessita di un buon numero di dati, ma essendo che le curve del train e della cross validation convergono aumentare il numero di dati potrebbe portare ad un overfitting. Inoltre la varianza dell'errore è molto limitata, tuttavia convergono ad un valore non troppo buono di  $R^2$  pari a 0.57.



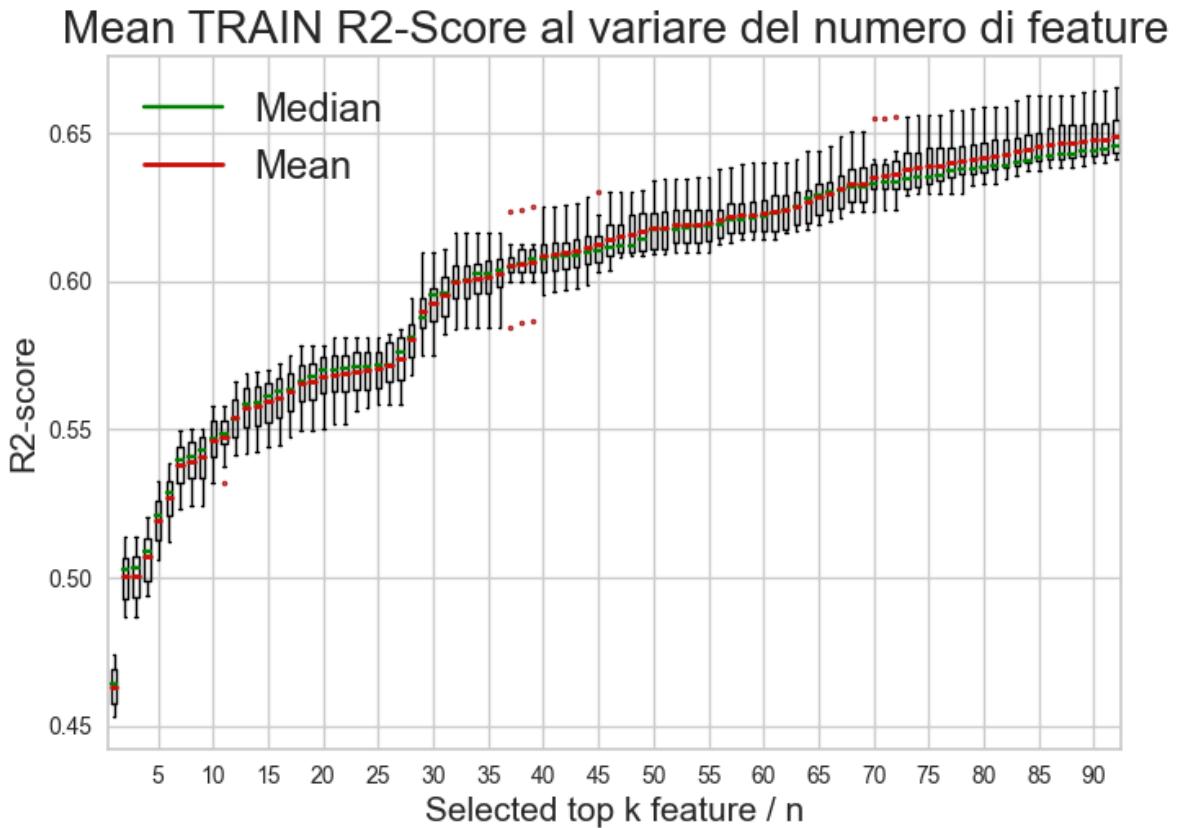
**Figura 3.2:** Learning Curve

**Feature Importance** La feature importance è un grafico che rappresenta il peso di ogni feature nel modello, quindi il valore numerico di ogni  $\beta_i$  della Formula 3.2 per ogni feature (Figura 3.3). L'algoritmo di ranking scelto è l'algoritmo di Pearson e viene computato tra il target ed ogni feature. Il numero di feature selezionato dal GridSearchCV, che rappresenta il miglior modello, è 37.

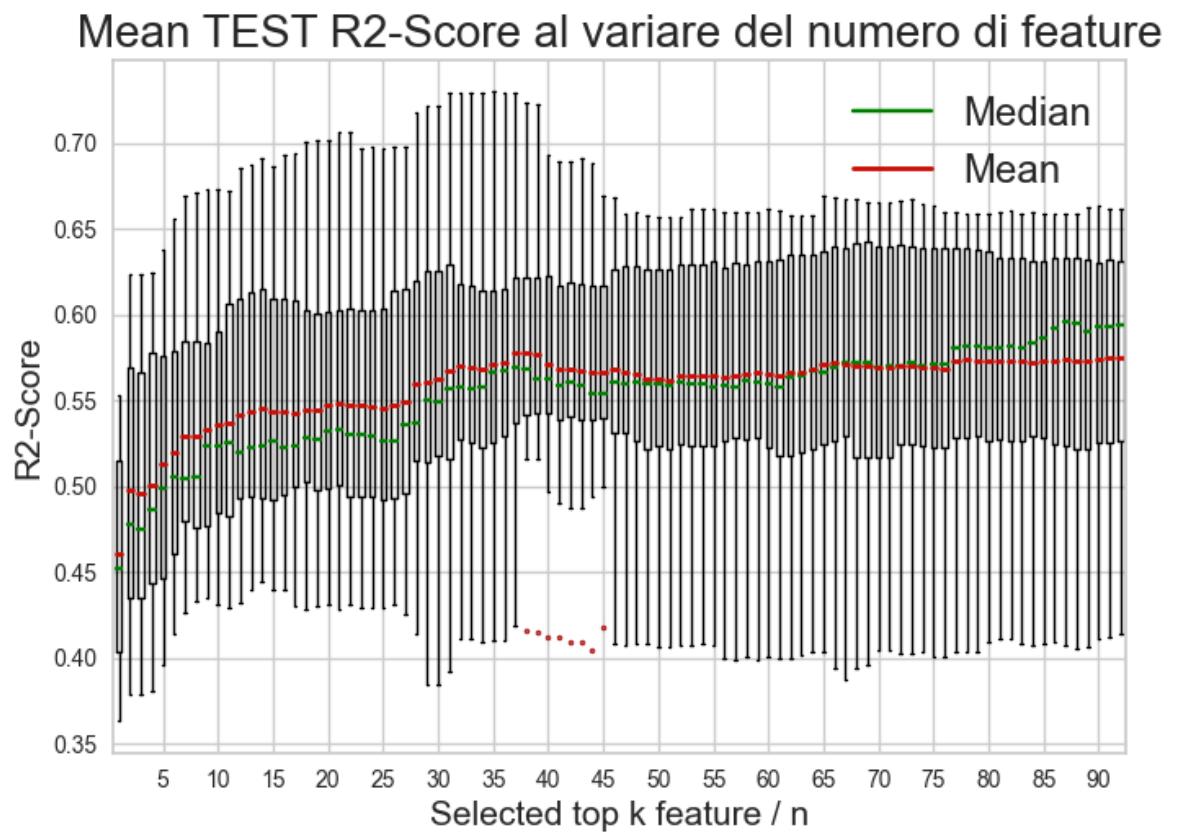


**Figura 3.3:** Feature Importance

**Mean Test e Train score della GridSearchCV** Rappresentiamo i risultati in termini di Mean  $R^2$  score sia per quanto riguarda i dati di train che i dati di test durante la cross validaiton. Dalla Figura 3.5 si può notare come i risultati di train migliorino sempre, avendo anche una limitata varianza, mentre i risultati di test hanno una varianza maggiore, come è normale che sia, ed il valore migliore è ovviamente in corrispondenza di 37 feature, ma tutti i valori oltre le 30 feature hanno uno score paragonabile.



**Figura 3.4:** TRAIN score della GridSearchCV al variare del numero di feature

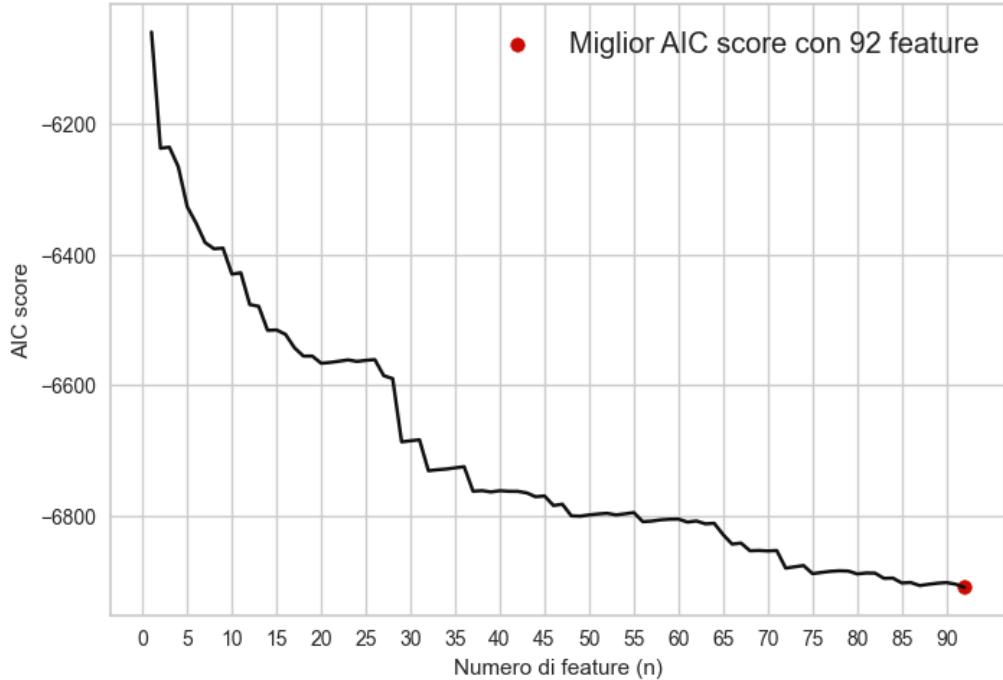


**Figura 3.5:** TEST score della GridSearchCV al variare del numero di feature

**Akaike Information Criterion** L'Akaike Information Criterion (AIC) è un indice per il confronto tra modelli che misura quanto è utile complicare un modello, un trade-off tra complessità (in termini di tempo impiegato) ed informazione (in termini di quantità di informazioni presenti). L'indice è stato valutato per ogni modello da 1 a 92 feature tramite la funzione ms.OLS [15] ed il risultato è in Figura 3.6. L' AIC restituisce uno score in base al numero di feature del modello ed alla maggiore similitudine stimata del modello, ovvero in che misura riesca a riprodurre i dati secondo. E' calcolato secondo la formula:

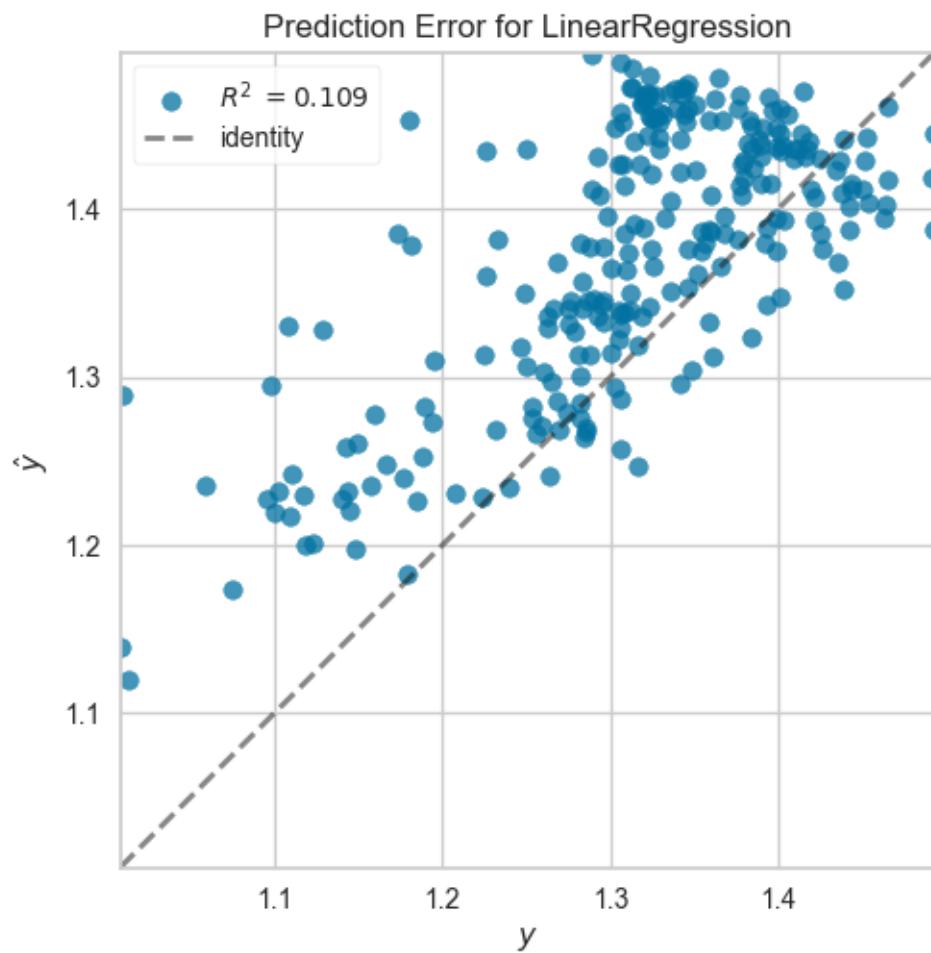
$$AIC = 2k - \ln(L) \quad (3.11)$$

Dove  $k$  è il numero di parametri del nostro modello ed  $L$  è la *likelihood*, la somiglianza del modello con i dati. Il miglior modello secondo questo parametro sarebbe il modello con 90 feature (Figura 3.6), tuttavia tramite il GridSearchCV abbiamo ottenuto come miglior modello in termini di  $R^2$  score il modello con 37 feature. Ci riserviamo un maggiore studio di questo indice per lavori futuri.

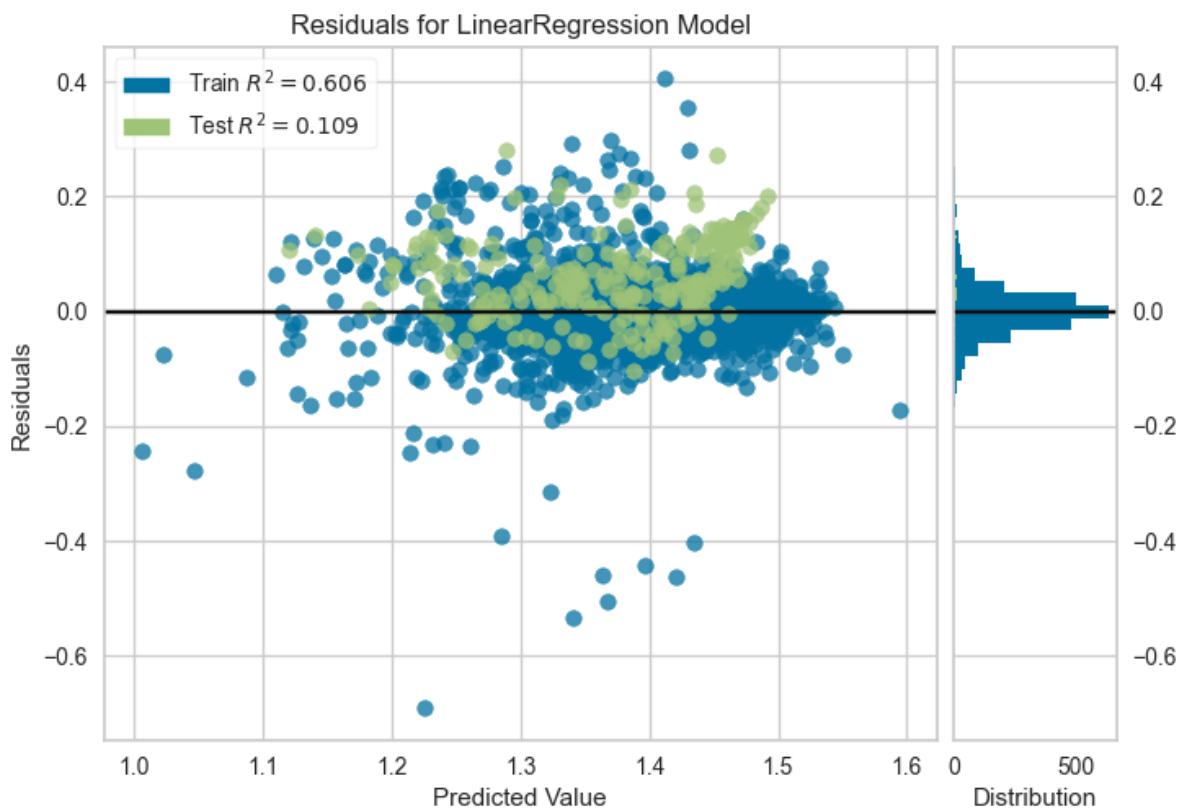


**Figura 3.6:** Akaike Information Criterion

**Regression Visualizers sul TEST** Nella Figura 3.7 è possibile vedere come il modello fitta i dati di TEST,  $R^2$  score è molto basso. Nella Figura 3.8 invece si vede come l'errore sia non disposto in una gaussiana.

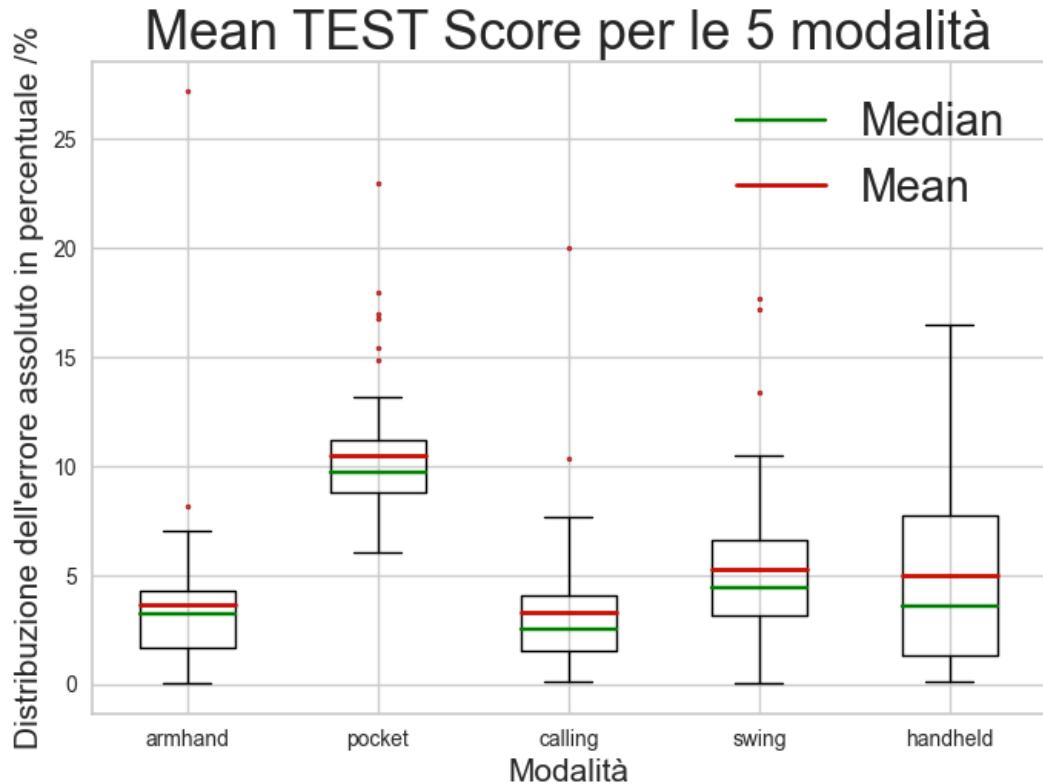


**Figura 3.7:** Nell'asse delle y il valore predetto, nell'asse delle x il valore effettivo



**Figura 3.8:** Nell'asse delle y il residuo (valore predetto- valore reale) , nell'asse delle x il valore predetto. Nell'istogramma a destra la distribuzione dell'errore

**Valutazioni sul MAPE e sul fit delle varie modalità** Il MAPE calcolato sul dataset di Test è di 5.6%, ma guardando la distribuzione dell'errore vediamo dalla Figura 3.9 che la varianza per ogni singola modalità è molto marcata.



**Figura 3.9:** Distribuzione dell'errore in percentuale per ogni modalità, con in blu il relativo  $R^2$  score.

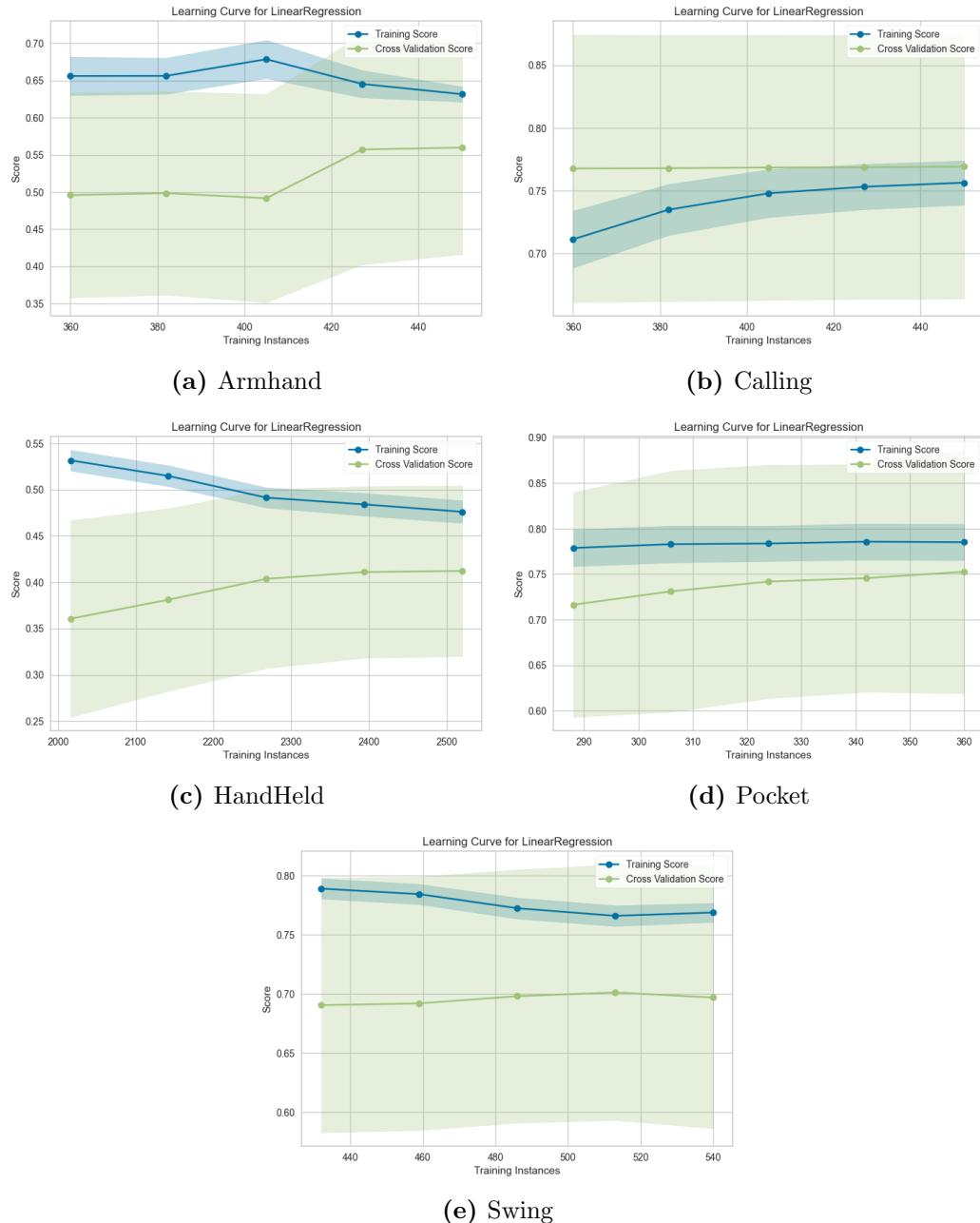
**Conclusioni** Come ci si aspettava un modello lineare non può stimare in maniera affidabile la lunghezza del passo nel caso di varie classi.

### 3.2.2 Modello dipendente

Viste le performance ottenute col modello indipendente abbiamo applicato le stesse identiche metodologie isolando i dati di ogni singola modalità, creando

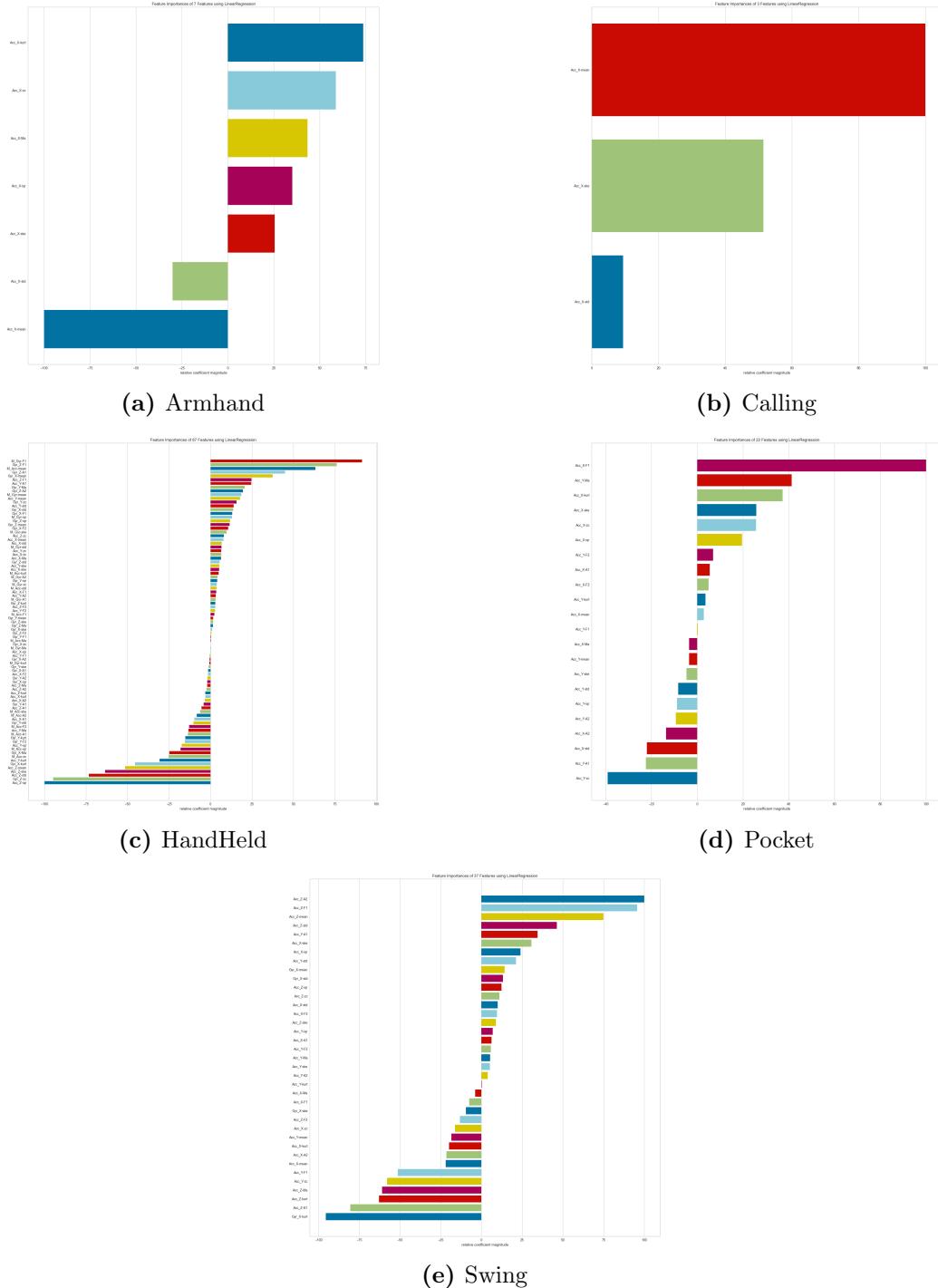
quindi cinque modelli differenti per vedere se prendendo una sola modalità di trasporto le performance migliorassero.

**Learning Curve** Dalle learning curve di ogni modello è chiaro, dalla Figura 3.10 che chi per una troppa varianza, chi per una poca convergenza, gioverebbe dall'uso di dati di train più numerosi e vari in quanto a soggetti.



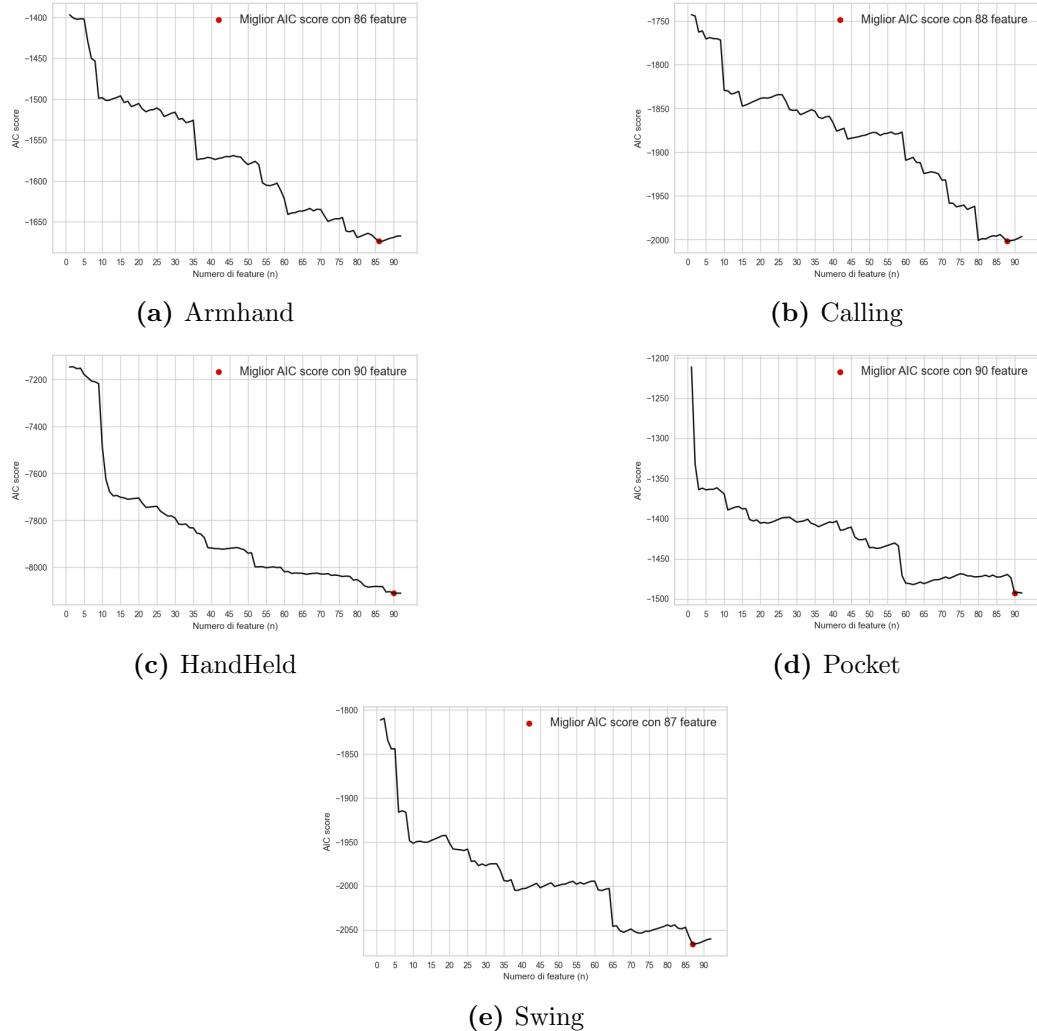
**Figura 3.10:** Tutte le Learning Curve dei cinque modelli.

**Feature Importance** Come si vede dalla Figura 3.11 essendo che ogni modalità genera una serie temporale diversa, come visto in Figura 2.2, sono anche diversi i numeri di feature che vengono scelti. Per questo motivo il lavoro di feature extraction, e la completezza di feature estratte (Capitolo 1) sono molto importanti.



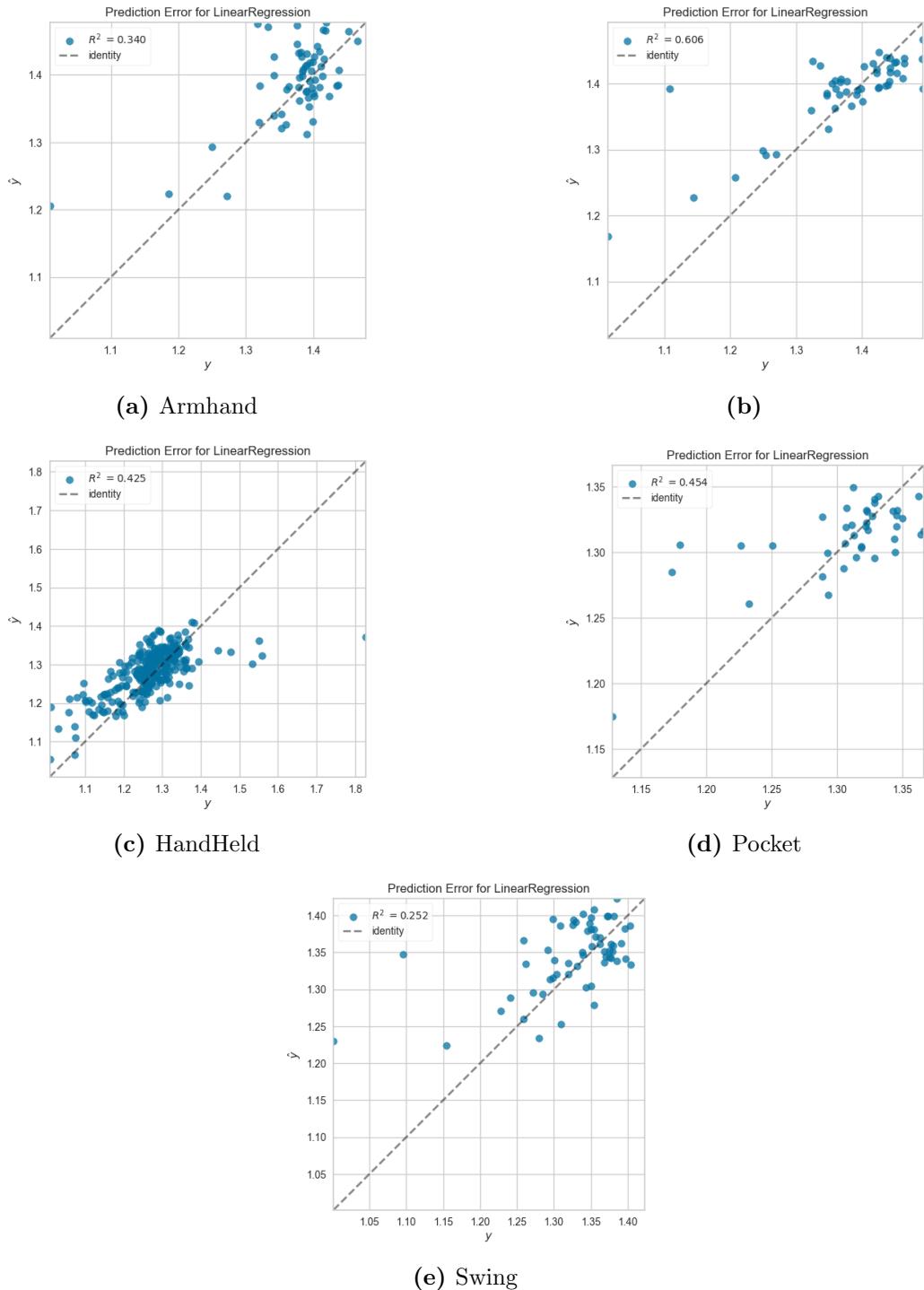
**Figura 3.11:** Tutte le Feature Importance dei cinque modelli.

**Akaike Information Criterion** Come si vede dalla Figura 3.12, l'AIC score migliore viene ottenuto usando la quasi totalità delle feature, nonostante il miglior modello stimato dal GridSearchCV abbia un differente numero di feature. Anche in questo caso lasciamo lo studio specifico di questo indice ai lavori futuri.

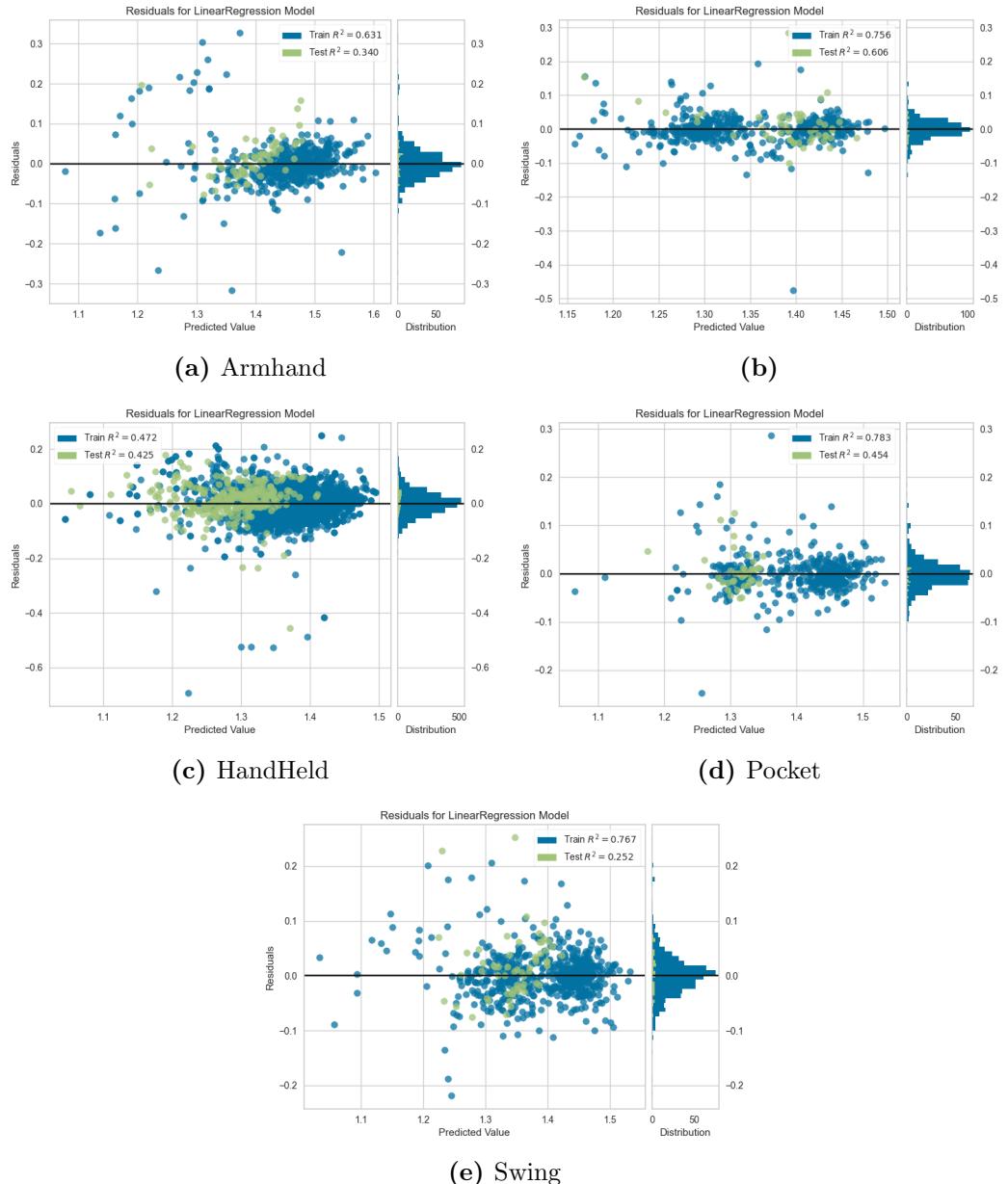


**Figura 3.12:** Tutte gli AIC score dei cinque modelli.  
Anche per i singoli modelli

**Regression Visualizers sul TEST** Visualizziamo ora il parametro fondamentale per la regressione linear, l' $R^2$  score, e come vediamo dalle Figure 3.13 e 3.14 l' $R^2$  score è di: 0.631 nel train e 0.340 nel test per Armhand; 0.756 nel train e 0.606 nel test per Calling; 0.472 nel train e 0.425 nel test per HandHeld; 0.783 nel train e 0.454 nel test per Pocket; 0.762 nel train e 0.252 nel test per Swing. Tutti valori, **maggiori**, anche di molto, rispetto a quelli ottenuti con un singolo modello.



**Figura 3.13:** Tutte le Prediction Error visualizer dei cinque modelli.



**Figura 3.14:** Tutte le Residual Plot visualizer dei cinque modelli.

**Risultati** In riferimento ai valori ottenuti con il modello indipendente dalla modalità (Figura 3.9), possiamo notare un miglioramento generale rispetto ai valori ottenuti con i modelli personalizzati (Figura 3.15).

I risultati sono raccolti nella Tabella 3, dove il MAPE è calcolato con la Formula (3.9), la distribuzione è stata calcolata tramite gli errori ottenuti con la Formula (3.8) ed il bias tramite la Formula (3.10).

**Tabella 3** : Risultati dei modelli nel test.

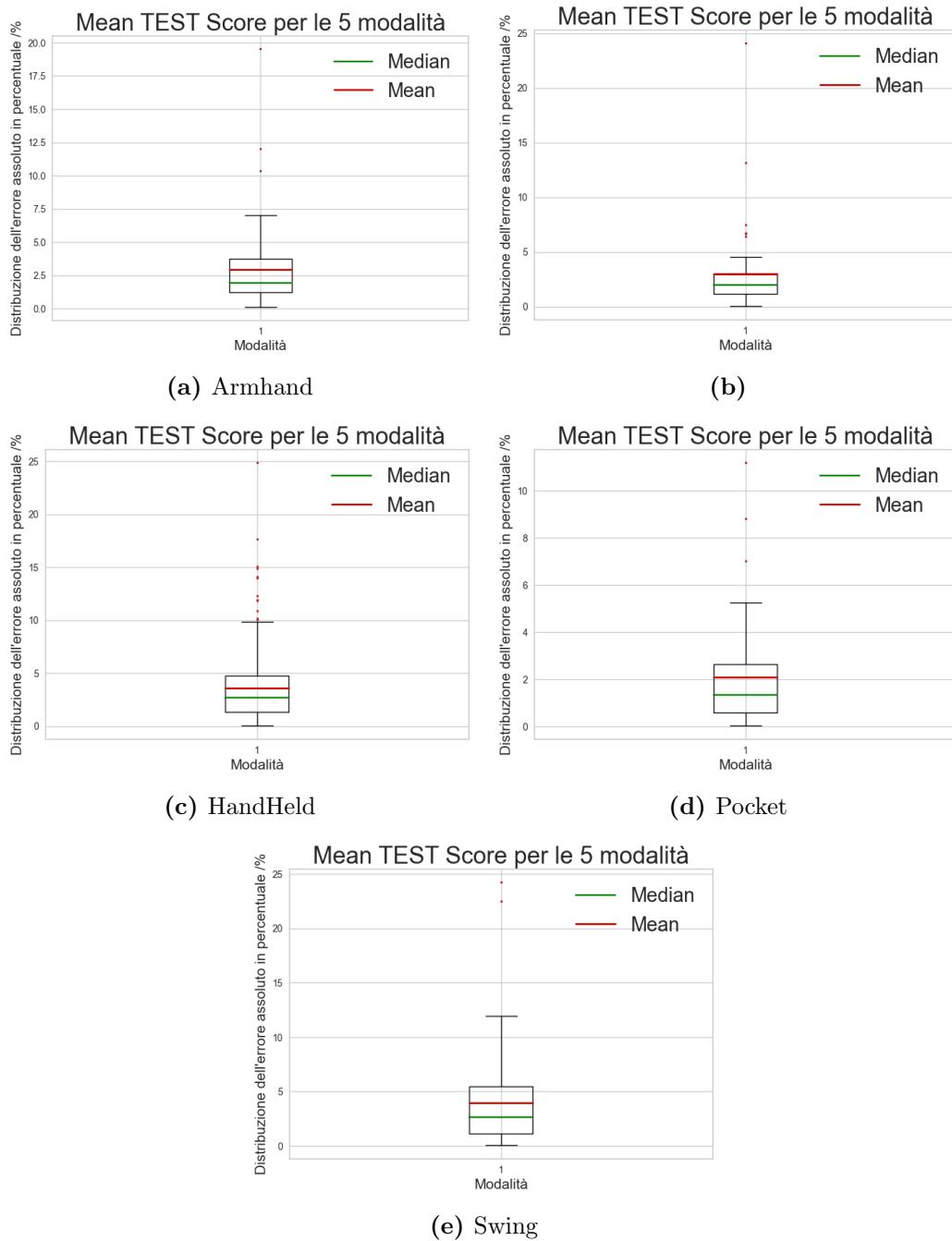
Mode	Multi-mode				Mode Indipendent			
	MAPE	st. dev.	$R^2$	bias	MAPE	st. dev.	$R^2$	bias
Armhand	2.96%	3.10%	0.33	-0.02	3.64%	3.89%	-0.105	-0.05
Calling	2.97%	3.79%	0.60	-0.01	3.31%	3.14%	0.64	0.01
HandHeld	3.58%	3.28%	0.42	-0.02	4.99%	4.19%	0.26	-0.05
Pocket	2.11%	2.37%	0.45	-0.01	10.52%	3.06%	-8.31	-0.14
Swing	3.96%	4.43%	0.25	-0.04	5.28%	3.68%	-1.8	-0.06

In Tabella 3.1 vengono confrontati i risultati ottenuti dal nostro metodo rispetto al metodo utilizzato da [7]. Quest'ultimo utilizza per la regressione vari modelli: in una prima fase utilizza XGBRegressor, DecisionTreeRegressor, AdaBoostRegressor e LightGBM come modelli singoli, infine utilizza un Support Vector Regression (SVR) con kernel='rbf' per ottenere dai cinque modelli precedenti una predizione unica.

**Tabella 3.1:** Confronto tra il metodo proposto ed il metodo di riferimento.

	Metodo proposto	Metodo di confronto
MAPE	3.20%	3.04%
st.dev	3.39%	1.85%

Notiamo che principalmente l'errore è dovuto alla deviazione standard parametru molto importante per applicazioni in cui è necessaria una grande precisione. Da questo studio risulta chiaro come sia molto importante la divisione delle varie modalità di trasporto.



**Figura 3.15:** Tutte le Distribuzioni dell'errore in percentuale sul passo dei cinque modelli.

## Conclusioni

In questo lavoro di tesi abbiamo provato a fornire un’alternativa a modelli molto complessi, sofisticati e onerosi dal punto di vista dell’hardware sostituendoli con un semplice modello lineare, ottimizzando tutto ciò che stava attorno al modello.

I risultati presentati nella Tabella 3 evidenziano come la suddivisione nelle varie modalità abbia apportato notevoli miglioramenti.

Nonostante un errore medio molto basso il punto debole di questo modello è la varianza dell’errore che è molto marcata. Tuttavia, seppur non in contesti medici o dove è necessaria una precisione estrema, il seguente modello può rivelarsi migliore rispetto a modelli più precisi trattati nel Capitolo grazie alla sua notevole portabilità potrebbe rivelarsi utili in contesti “amatoriali”.

L’argomento è ancora molto vasto, miglioramenti futuri di questa tesi possono riguardare sicuramente l’aumento della robustezza del modello, utilizzando un modello capace di gestire meglio i valori anomali che in una applicazione reale possono verificarsi, l’aumento dei campioni e dei dati per una migliore valutazione delle singole modalità e la calibrazione del modello sul singolo soggetto. Un suggerimento potrebbe essere aggiungere feature che descrivano il soggetto per esempio: Altezza/Lunghezza della gamba, Peso del soggetto ed Età del soggetto.

Tuttavia per questi ultimi è necessario creare un nuovo dataset, completo di tutte queste informazioni e molto vario in quanto a soggetti esaminati, che in letteratura non è presente.

# Bibliografia

- [1] Rafael C Gonzalez et al. «Modified pendulum model for mean step length estimation». In.
- [2] Harvey Weinberg. «Using the ADXL202 in pedometer and personal navigation applications». In: () .
- [3] Jeong Won Kim et al. «A step, stride and heading determination for the pedestrian navigation system». In: (). URL: [https://file.scirp.org/pdf/nav20040100034\\_69728139.pdf](https://file.scirp.org/pdf/nav20040100034_69728139.pdf).
- [4] Masahiro Hayashitani et al. «10ns High-speed PLZT optical content distribution system having slot-switch and GMPLS controller». In: () .
- [5] Vinay Kukreja, Deepak Kumar e Amandeep Kaur. «Deep learning in Human Gait Recognition: An Overview». In: URL: <https://www.mdpi.com/1424-8220/19/4/840>.
- [6] Qu Wang et al. «Pedestrian Stride-Length Estimation Based on LSTM and Denoising Autoencoders». In: (). URL: <https://www.mdpi.com/1424-8220/19/4/840>.
- [7] Qu Wang et al. «Pedestrian Walking Distance Estimation Based on Smartphone Mode Recognition». In: (). URL: <https://www.mdpi.com/2072-4292/11/9/1140>.
- [8] W. Zijlstra e A.L. Hof. «Displacement of the pelvis during human walking: experimental data and model predictions». In: (). URL: <https://www.sciencedirect.com/science/article/pii/S0966636297000210>.
- [9] Anshul Rai et al. «Zee: Zero-Effort Crowdsourcing for Indoor Localization». In: URL: <https://doi.org/10.1145/2348543.2348580>.
- [10] GA Cavagna e P Franzetti. «The determinants of the step frequency in walking in humans.» In: () .

- [11] Yellowbrick. *Machine Learning Visualization*. URL: <https://www.scikit-yb.org/en/latest/>.
- [12] Scipy. *Fundamental algorithms for scientific computing in Python*. URL: <https://scipy.org/>.
- [13] *Collection of independent publication*. URL: <https://towardsdatascience.com/about>.
- [14] Statsmodel. *statistical models, hypothesis tests, and data exploration*. URL: <https://www.statsmodels.org/stable/index.html>.
- [15] Scikit-learn. *Scikit-learn: Machine Learning in Python*. URL: <https://scikit-learn.org/stable/>.

# Appendice

Esempio di algoritmo con modalità "handheld"

```
In [1]: #WDE dataset
WDE_path="C:/Users/aliba/OneDrive/Desktop/UNIVERSITA/TESI/DATASET/WalkingDistanceEstimation-master/dataset/"
classi=['armhand', 'pocket', 'calling', 'swing', 'handheld']
n_elem=500
n_left=50

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.pyplot as plt
import pandas as pd

#model
from sklearn.model_selection import KFold
from sklearn.model_selection import train_test_split, GridSearchCV

#regression
from sklearn.model_selection import KFold
from sklearn import linear_model
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.pipeline import Pipeline
from sklearn.metrics import make_scorer, r2_score

#visualization
from yellowbrick.regressor import PredictionError
from yellowbrick.regressor import ResidualsPlot
from yellowbrick.features import rank1d, rank2d
from yellowbrick.model_selection import FeatureImportances
from yellowbrick.model_selection import LearningCurve

from ipynb.fs.full.functioncollection import trueeqWDE,error_rate, importWDE, filtWDE, f_ext_WDE,makeeqWDE,full
```

## Import all WDE

```
In [2]: DATASET, stop_list =importWDE()
```

```

PDR_Raw_2019-03-20-09-10-12 {'armhand': 0, 'pocket': 0, 'calling': 0, 'swing': 0, 'handheld': 288}
Outliers eliminati      {'armhand': 0, 'pocket': 0, 'calling': 0, 'swing': 0, 'handheld': 12}

PDR_Raw_2019-03-20-09-21-02 {'armhand': 0, 'pocket': 0, 'calling': 284, 'swing': 0, 'handheld': 0}
Outliers eliminati      {'armhand': 0, 'pocket': 0, 'calling': 15, 'swing': 0, 'handheld': 0}

PDR_Raw_2019-03-20-09-29-55 {'armhand': 0, 'pocket': 0, 'calling': 34, 'swing': 0, 'handheld': 45}
Outliers eliminati      {'armhand': 0, 'pocket': 0, 'calling': 3, 'swing': 0, 'handheld': 1}

PDR_Raw_2019-03-21-08-32-39 {'armhand': 196, 'pocket': 0, 'calling': 0, 'swing': 0, 'handheld': 0}
Outliers eliminati      {'armhand': 26, 'pocket': 0, 'calling': 0, 'swing': 0, 'handheld': 0}

PDR_Raw_2019-03-21-09-07-51 {'armhand': 527, 'pocket': 0, 'calling': 0, 'swing': 0, 'handheld': 0}
Outliers eliminati      {'armhand': 203, 'pocket': 0, 'calling': 0, 'swing': 0, 'handheld': 0}

PDR_Raw_2019-03-21-11-57-56 {'armhand': 0, 'pocket': 0, 'calling': 0, 'swing': 197, 'handheld': 0}
Outliers eliminati      {'armhand': 0, 'pocket': 0, 'calling': 0, 'swing': 151, 'handheld': 0}

PDR_Raw_2019-03-24-11-12-21 {'armhand': 0, 'pocket': 142, 'calling': 0, 'swing': 139, 'handheld': 0}
Outliers eliminati      {'armhand': 0, 'pocket': 8, 'calling': 0, 'swing': 10, 'handheld': 0}

PDR_Raw_2019-03-28-11-50-11 {'armhand': 0, 'pocket': 0, 'calling': 275, 'swing': 429, 'handheld': 425}
Outliers eliminati      {'armhand': 0, 'pocket': 0, 'calling': 16, 'swing': 42, 'handheld': 121}

PDR_Raw_2019-03-29-07-37-22 {'armhand': 0, 'pocket': 0, 'calling': 171, 'swing': 0, 'handheld': 0}
Outliers eliminati      {'armhand': 0, 'pocket': 0, 'calling': 64, 'swing': 0, 'handheld': 0}

PDR_Raw_2019-03-29-08-30-54 {'armhand': 0, 'pocket': 157, 'calling': 0, 'swing': 0, 'handheld': 0}
Outliers eliminati      {'armhand': 0, 'pocket': 116, 'calling': 0, 'swing': 0, 'handheld': 0}

PDR_Raw_2019-03-30-11-29-16 {'armhand': 0, 'pocket': 0, 'calling': 0, 'swing': 0, 'handheld': 897}
Outliers eliminati      {'armhand': 0, 'pocket': 0, 'calling': 0, 'swing': 0, 'handheld': 167}

PDR_Raw_2019-03-31-01-23-59 {'armhand': 0, 'pocket': 0, 'calling': 0, 'swing': 0, 'handheld': 1726}
Outliers eliminati      {'armhand': 0, 'pocket': 0, 'calling': 0, 'swing': 0, 'handheld': 202}

PDR_Raw_2019-03-31-10-04-54 {'armhand': 0, 'pocket': 0, 'calling': 0, 'swing': 385, 'handheld': 0}
Outliers eliminati      {'armhand': 0, 'pocket': 0, 'calling': 0, 'swing': 156, 'handheld': 0}

PDR_Raw_2019-03-31-10-33-25 {'armhand': 0, 'pocket': 386, 'calling': 0, 'swing': 3, 'handheld': 0}
Outliers eliminati      {'armhand': 0, 'pocket': 45, 'calling': 36, 'swing': 2, 'handheld': 0}

PDR_Raw_2019-03-31-12-03-05 {'armhand': 0, 'pocket': 0, 'calling': 0, 'swing': 232, 'handheld': 0}
Outliers eliminati      {'armhand': 0, 'pocket': 0, 'calling': 0, 'swing': 21, 'handheld': 0}

PDR_Raw_2019-03-31-12-29-51 {'armhand': 0, 'pocket': 0, 'calling': 0, 'swing': 0, 'handheld': 568}
Outliers eliminati      {'armhand': 0, 'pocket': 0, 'calling': 0, 'swing': 0, 'handheld': 205}

PDR_Raw_2019-04-01-10-45-07 {'armhand': 0, 'pocket': 0, 'calling': 0, 'swing': 0, 'handheld': 1214}
Outliers eliminati      {'armhand': 0, 'pocket': 0, 'calling': 0, 'swing': 0, 'handheld': 29}

PDR_Raw_2019-04-02-08-44-50 {'armhand': 0, 'pocket': 0, 'calling': 0, 'swing': 0, 'handheld': 664}
Outliers eliminati      {'armhand': 0, 'pocket': 0, 'calling': 0, 'swing': 0, 'handheld': 60}

```

In totale=> armhand:723, pocket:685, calling:764, swing:1385, handheld:5827, -->9384 stride

```

In [3]: print(stop_list)

{'armhand': [0, 196, 723], 'pocket': [0, 142, 299, 685], 'calling': [0, 284, 318, 593, 764], 'swing': [0, 197, 336, 765, 1150, 1153, 1385], 'handheld': [0, 288, 333, 758, 1655, 3381, 3949, 5163, 5827]}

```

```

In [4]: #select only one mode
mode="handheld"
for c in classi:
    if c!=mode:
        del DATASET[c]
        del stop_list[c]

```

"DATASET" è la variabile in cui importiamo il nostro dataset WDE, è un dizionario che ha come chiavi le cinque modalità, come valori ha una lista. La lista è una lista di stride, dove ogni stride è un dizionario che rappresenta le misurazioni dello stride

```

In [5]: print(type(DATASET))
for c,v in DATASET.items():
    print(c,type(v),len(v),type(v[0]),v[0].keys())

<class 'dict'>
handheld <class 'list'> 5827 <class 'dict'> dict_keys(['target', 'Acc_X', 'Acc_Y', 'Acc_Z', 'Gyr_X', 'Gyr_Y', 'Gyr_Z', 'SensorTimestamp'])

```

Applichiamo il butterworth filter di primo ordine con cutoff frequency di 3Hz ad ogni stride

```

In [6]: filtWDE(DATASET);

```

```
Filtering:##  
Done!
```

"Feature\_DS" è un dizionario con chiavi le classi, e valori dizionari con chiavi 'feature' che contiene la list di feature dello stride e 'target' che contiene il float della lunghezza dello stride.

```
In [7]: Feature_DS=f_ext_WDE(DATASET)  
Extracting handheld#####
```

```
In [8]: for k,v in Feature_DS.items():  
    print("\n",k,type(v),v.keys(),len(v['feature']),end=" ")  
    if k==mode:  
        print(len(v['feature'][0]),len(v['target']))
```

```
armhand <class 'dict'> dict_keys(['feature', 'target']) 0  
pocket <class 'dict'> dict_keys(['feature', 'target']) 0  
calling <class 'dict'> dict_keys(['feature', 'target']) 0  
swing <class 'dict'> dict_keys(['feature', 'target']) 0  
handheld <class 'dict'> dict_keys(['feature', 'target']) 5827 92 5827
```

- **DS\_train** è il dataset equilibrato di train prendendo i primi 500 elementi per ogni modalità
- **DS\_test** è il dataset di test equilibrato prendendo gli *ultimi* 50 elementi per ogni modalità

```
In [9]: #nt=int(len(Feature_DS[mode]['target'])/10)  
DS_train,DS_test = trueeqWDE(Feature_DS,stop_list,n_train=2800,n_test=280)
```

```
In [10]: for k,v in DS_train.items():  
    print("\n",k,type(v),v.keys(),len(v['feature']),end=" ")  
    if k==mode:  
        print(len(v['feature'][0]),len(v['target']))
```

```
armhand <class 'dict'> dict_keys(['feature', 'target']) 0  
pocket <class 'dict'> dict_keys(['feature', 'target']) 0  
calling <class 'dict'> dict_keys(['feature', 'target']) 0  
swing <class 'dict'> dict_keys(['feature', 'target']) 0  
handheld <class 'dict'> dict_keys(['feature', 'target']) 2800 92 2800
```

```
In [11]: for k,v in DS_test.items():  
    print("\n",k,type(v),v.keys(),len(v['feature']),end=" ")  
    if k==mode:  
        print(len(v['feature'][0]),len(v['target']))
```

```
armhand <class 'dict'> dict_keys(['feature', 'target']) 0  
pocket <class 'dict'> dict_keys(['feature', 'target']) 0  
calling <class 'dict'> dict_keys(['feature', 'target']) 0  
swing <class 'dict'> dict_keys(['feature', 'target']) 0  
handheld <class 'dict'> dict_keys(['feature', 'target']) 280 92 280
```

- **regr\_dataset\_train** è il dataset formattato per la regressione per il train, non tiene in considerazione le modalità ed unisce tutte le feature in una lista e così tutti i target. È un dizionario con chiavi 'feature' e 'target'
- **regr\_dataset\_test** è lo stesso, ma con i dati di test

```
In [12]: regr_dataset_train, regr_dataset_test = full_regr_dataset(DS_train,DS_test)
```

```
In [13]: for k,v in regr_dataset_train.items():  
    print(k,type(v),len(v),type(v[0]))
```

```
feature <class 'list'> 2800 <class 'list'>  
target <class 'list'> 2800 <class 'float'>
```

```
In [14]: for k,v in regr_dataset_test.items():  
    print(k,type(v),len(v),type(v[0]))
```

```
feature <class 'list'> 280 <class 'list'>  
target <class 'list'> 280 <class 'float'>
```

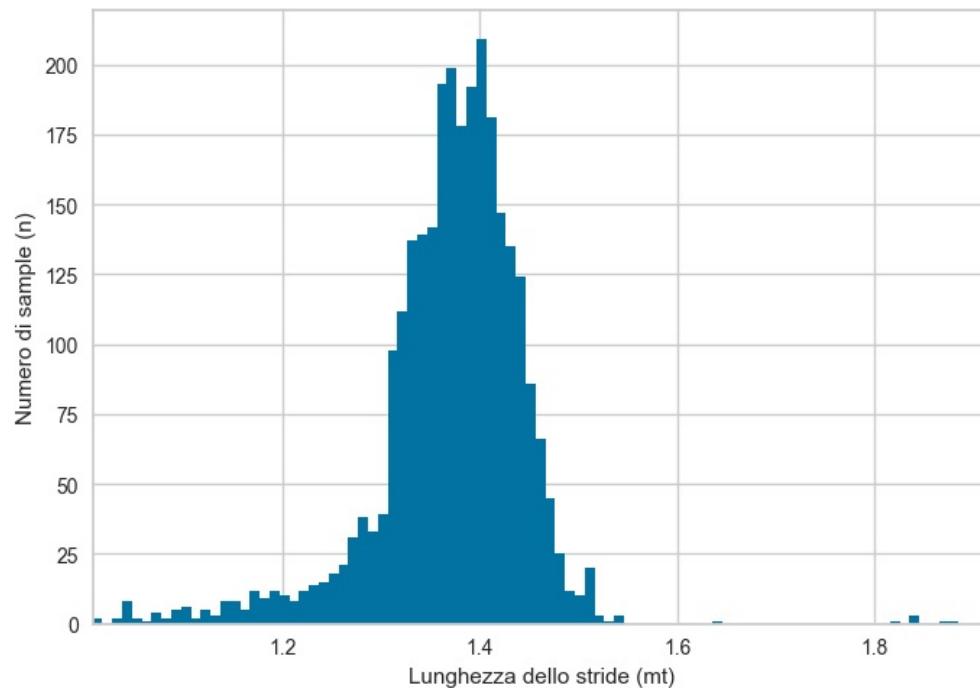
Isoliamo i singoli **F\_x,F\_y** rispettivamente feature e target del train e **F\_x\_test,F\_y\_test** rispettivamente feature e target di test

```
In [15]: F_x=np.array(regr_dataset_train["feature"])  
F_y=np.array(regr_dataset_train["target"])  
F_x_test=np.array(regr_dataset_test["feature"])  
F_y_test=np.array(regr_dataset_test["target"])
```

Vediamo come sono distribuite le lunghezze registrate degli stride

```
In [16]: #istogramma dei target F_y  
ist_stride_lenght(F_y)
```

Registered Stride lenght: Min: 1.006223362540999 , Max: 1.917364683119593  
Media= 1.3709574082485223; Deviazione standard= 0.0762269550613965.



Out[16]: 0

### -----Feature Analysis-----

Salviamo la lista dei nomi delle feature.

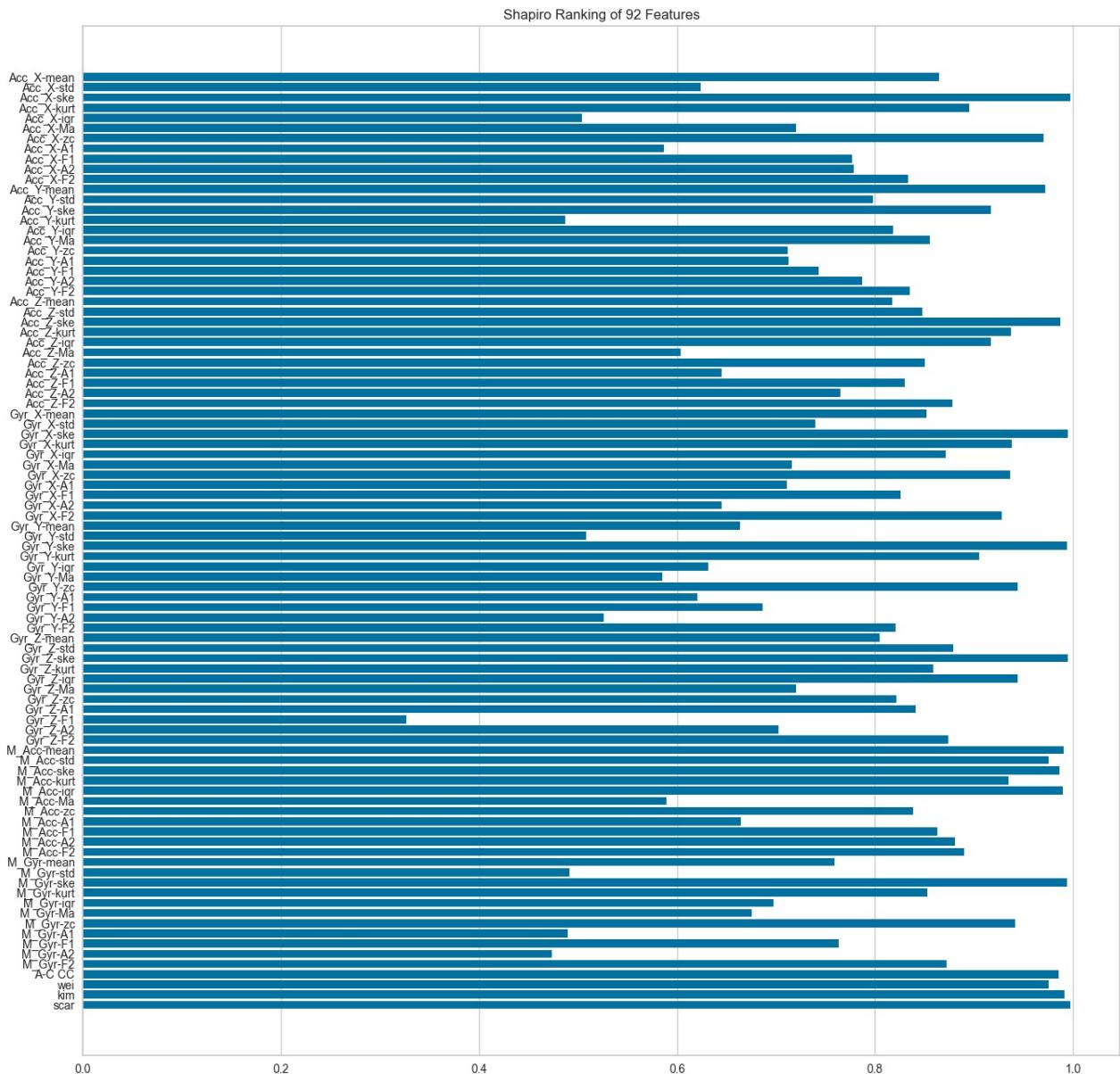
```
In [17]: feature_name=['Acc_X-mean', 'Acc_X-std', 'Acc_X-ske', 'Acc_X-kurt', 'Acc_X-iqr', 'Acc_X-Ma', 'Acc_X-zc', 'Acc_X  
print(f"Abbiamo: {len(feature_name)} feature.")
```

Abbiamo: 92 feature.

**rank1d** e **rank2d** offrono la possibilità di visualizzare in modo semplice ed intuitivo il ranking delle feature. In particolare:

- **rank1d** utilizza shapiro da scipy.stats per calcolare lo score di ogni feature e poi plotta lo score in un barchart di matplotlib.
- **rank2d** valuta lo score per ogni coppia di feature, le funzioni di scoring supportate sono <"pearson", "covariance", "spearman", "kendalltau"> che misurano quanto le feature sono legate tra di loro; noi utilizziamo "pearson".

```
In [18]: #rank1d  
_, axes = plt.subplots(ncols=1, figsize=(16,16))  
rank1d(F_x,F_y,features=feature_name,ax=axes , show=False)  
plt.show()
```



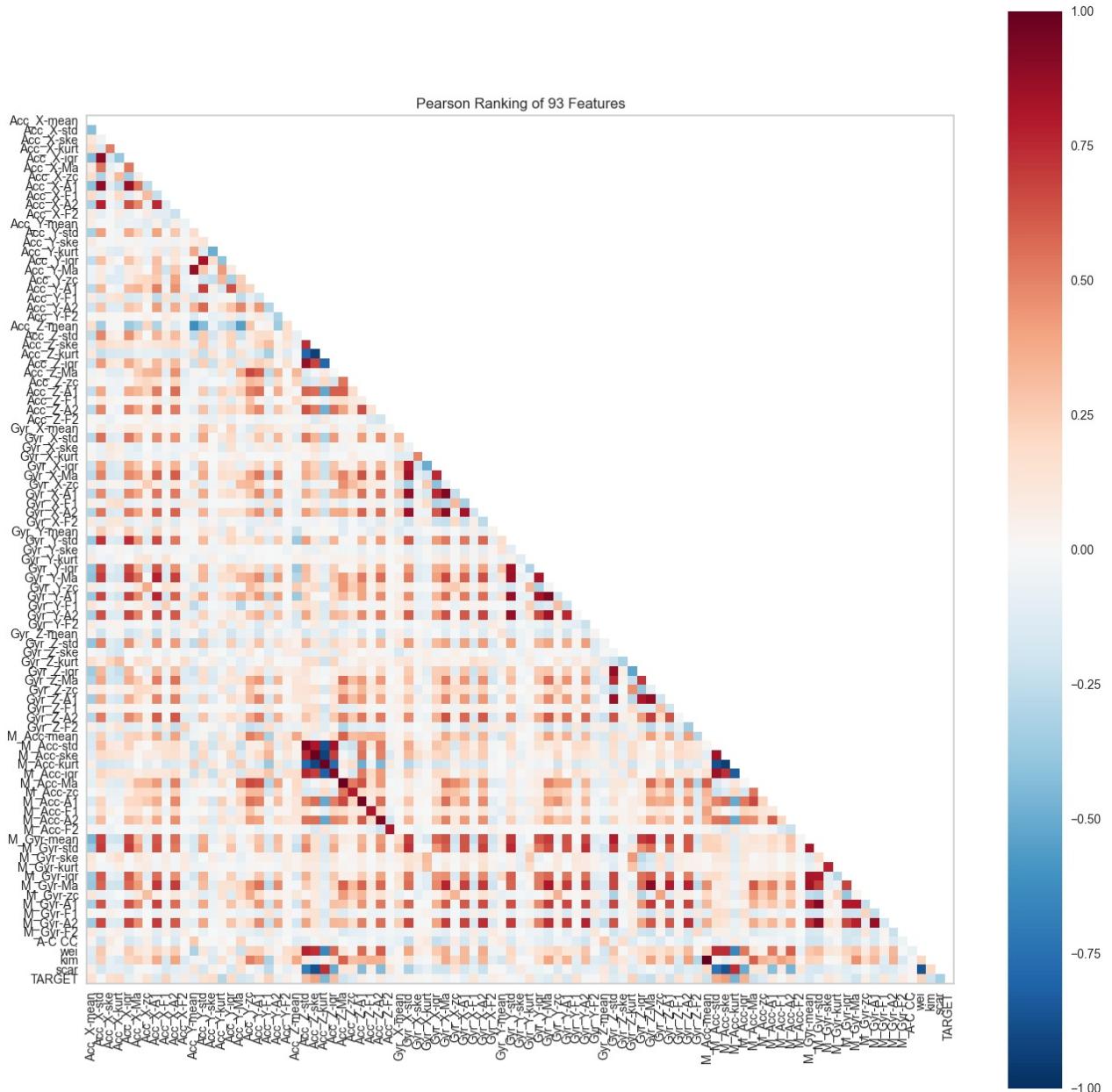
In [19]: `####rank2d`

```

##Per visualizzare cosa sceglierà f_regression
#devo fare in modo che tra le feature ci sia anche il target #poi vedere solo la linea riferita al target e ved
HP_x=np.hstack((F_x,np.reshape(F_y,(-1,1))))
HP_featurename=feature_name+["TARGET"]
_, axes = plt.subplots(ncols=1, figsize=(16,16))
rank2d(HP_x,features=HP_featurename,ax=axes , show=False)
plt.show()

#Si vede sia ogni coppia di feature come è relazionata, sia il pearson score con il target di ogni feature!

```



## Linear Regression

Descrizione:

- **input** : F\_x ed F\_y che sono rispettivamente il vettore di vettori di feature ed il vettore di target.
- **perf** è un oggetto scorer costruito dalla funzione che noi desideriamo usare comemetro di valutazione per le performance, nel nostro caso `error_rate`.
- **kf** è l'oggetto che crea gli indici per dividere i nostri input in k (10) fold da usare per la cross validation.
- **pipe** è l'oggetto che rappresenta la pipeline degli estimatori, ogni step fa il fit con il predict del precedente; in ordine l'input viene ridotto selezionando le feature (SelectKbest con `f_regression`)> le feature vengono scalate (StandardScaler)> il modello lineare viene fittato (LinearRegression) `f_regression` usa come metodo di ranking il pearson tra la singola feature ed il target.
- **est** è l'istanza GridSearchCV che permette di eseguire la cross validation tramite gli indici di kf (facendo quindi in tutto k (10) train usando ogni volta un fold diverso per il test) e di volta in volta testa una nuova combinazione di parametri presenti nel dizionario param (nel nostro caso solo quante feature selezionare, da 1 a 92). Per ogni parametro esegue una 10-fold cross-validation e calcola la media dello score calcolato con perf. Infine restituisce come `.best_params_` i parametri con cui si ha avuto la migliore media, con `.best_estimator_` restituisce un estimatore fittato su tutti i dati di train con il `best_param_`.

Salvo quindi il miglior estimatore in best che rappresenta quindi l'estimatore che ha il minor `error_rate` medio nei k fold con i migliori parametri, fittato su tutto il set di dati di train senza la divisione in fold.

In [20]: #F\_x feature #F\_y target

```
#Creiamo l'oggetto per valutare le performance tramite la nostra funzione
perf=make_scorer(error_rate, greater_is_better=False)
```

```

#Creiamo oggetto cross-validator tramite KFold
kf = KFold(n_splits=10, shuffle=True, random_state=0)#con lo shuffle= False assicuriamo che i blocchi siano comunque diversi
#####GRIDSEARCH_CV#####
#select = SelectKBest(score_func=f_regression)
#scaler=StandardScaler()
#regr = linear_model.LinearRegression()

#pipeline di estimatori
pipe = Pipeline([('select',SelectKBest(score_func=f_regression)),('scaler', StandardScaler()), ('regr', linear_model.LinearRegression())])
print("La pipeline è composta da: ", pipe)

#Parametri da testare:
param=[{'select_k':[x+1 for x in range(F_x.shape[1])]}]

#Creiamo l'oggetto per la ricerca dei parametri
est=GridSearchCV(pipe,param_grid= param, cv=kf,verbose=1,scoring="r2", return_train_score= True) #Fittiamo sul dataset
est.fit(F_x,F_y)

#Ricaviamo il miglior set di parametri
best=est.best_estimator_
print(f"\n\nIl miglior estimatore si ha col parametro {est.best_params_} ottenuto all'indice: {est.best_index_}")

La pipeline è composta da: Pipeline(steps=[('select',
    SelectKBest(score_func=<function f_regression at 0x000001CA5BCAA8C0>),
    ('scaler', StandardScaler()), ('regr', LinearRegression()))])
Fitting 10 folds for each of 92 candidates, totalling 920 fits

```

Il miglior estimatore si ha col parametro {'select\_k': 87} ottenuto all'indice: 86 .

L'R2-score medio è di 0.414 !

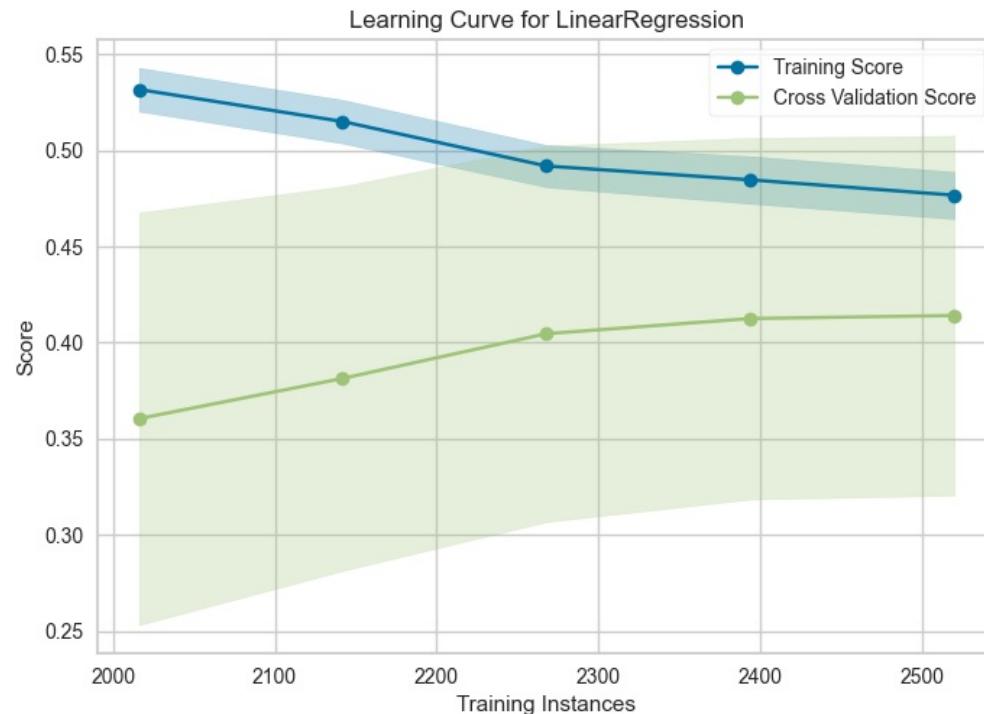
## -----MODEL EVALUETION-----

**LearningCurve** permette di vedere l'evoluzione delle performance del modello facendo la media dello score nella cross validation (asse y) in base al numero di train samples (asse x). Permette di valutare:

- Se il modello necessita di maggiori dati di training (se converge vuol dire che aumentare il numero di sample migliora le performance, se converge in un valore troppo alto vuol dire che necessita più sample in training)
- Se l'errore dipende di più dalla media o dalla varianza

```
In [21]: viz = LearningCurve(best, scoring="r2",cv=kf,train_sizes=np.linspace(0.8,1.0,5))

viz.fit(F_x,F_y) # Fit the data to the visualizer
viz.show()
```



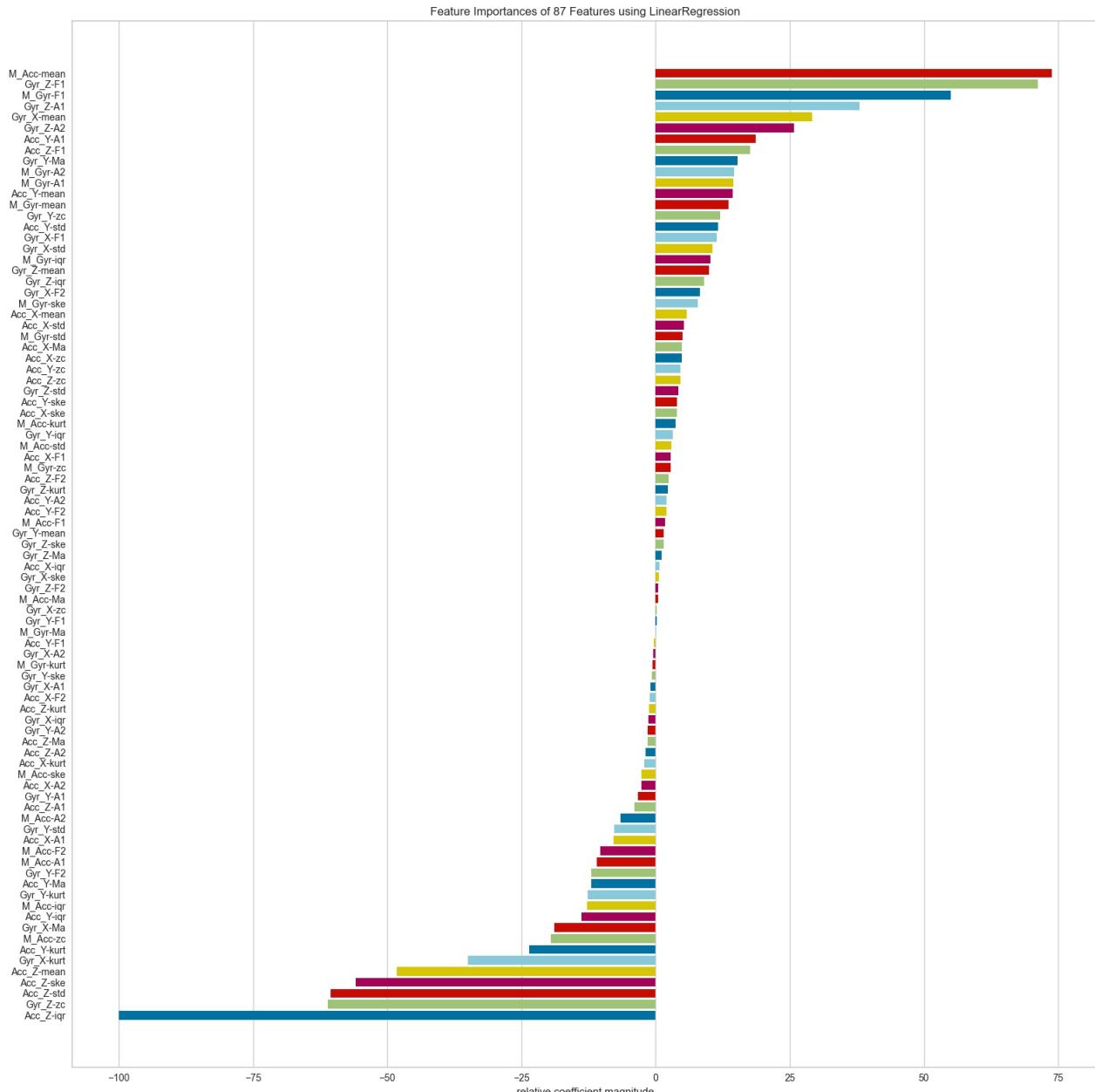
```
Out[21]: <AxesSubplot:title={'center':'Learning Curve for LinearRegression'}, xlabel='Training Instances', ylabel='Score'>
```

## -----FEATURE IMPORTANCE-----

**FeatureImportances** permette di plottare le feature del best estimator ordinate in ordine decrescente secondo la loro importanza!

```
In [22]: #per la feature importance dobbiamo scomporre la pipeline, ci serve solo l'ultimo step, il regressore nominato
##Attenzione FeatureImportance.fit trasforma il regressore, quindi creiamone uno nuovo
kk=est.best_params_['select_k']
skb=SelectKBest(score_func=f_regression, k=kk)
temp_X=skb.fit_transform(F_x,F_y)
stdsc=StandardScaler()
temp_X=stdsc.fit_transform(temp_X)
linmod=linear_model.LinearRegression()

_, axes = plt.subplots(ncols=1, figsize=(16,16))
viz = FeatureImportances(linmod,ax=axes,labels=feature_name)
viz.fit(temp_X,F_y)
viz.show()
```



```
Out[22]: <AxesSubplot:title={'center':'Feature Importances of 87 Features using LinearRegression'}, xlabel='relative coefficient magnitude'>
```

Visualizziamo tutti i risultati del GridSearchCV

Qui possiamo vedere ogni split con ogni parametro che score ha ottenuto, sia per il train (9 fold) che per il test (1 fold). Es: split3 è la cross validation usando come train fold (0..2,4..10) e come test fold 3 .

Le righe corrispondono alle performance dell'estimatore con un determinato parametro (92 righe come il numero di parametri testati); le colonne rappresentano gli il determinato score per ogni parametro.

```
In [23]: risultati=pd.DataFrame(est.cv_results_)
display(risultati)
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_select_k	params	split0_test_score	split1_test_score	split2_te
0	0.005501	0.007046	0.001002	0.003007	1	{'select_k': 1}	0.257723	0.190931	0
1	0.005331	0.007209	0.001563	0.004688	2	{'select_k': 2}	0.258982	0.190739	0
2	0.005899	0.005440	0.002336	0.004955	3	{'select_k': 3}	0.268374	0.199494	0
3	0.009481	0.007747	0.000000	0.000000	4	{'select_k': 4}	0.268176	0.199472	0
4	0.006901	0.007366	0.000000	0.000000	5	{'select_k': 5}	0.260035	0.218646	0
...	...	...	...	...	...	...	...	...	...
87	0.045075	0.003646	0.001600	0.003199	88	{'select_k': 88}	0.516631	0.383841	0
88	0.045390	0.005570	0.001600	0.003200	89	{'select_k': 89}	0.524551	0.372994	0
89	0.047329	0.006790	0.000000	0.000000	90	{'select_k': 90}	0.523961	0.367028	0
90	0.046316	0.006488	0.001600	0.003200	91	{'select_k': 91}	0.528656	0.361556	0
91	0.047981	0.003769	0.001352	0.004055	92	{'select_k': 92}	0.529608	0.362194	0

92 rows × 31 columns

Selezioniamo solo i dati relativi all'errore negli split di train e test Per poter costruire i seguenti boxplot.

```
In [24]: #table di tutte le colonne
column_names=list(risultati)
#table di ciò che non ci interessa nei nuovi dataframe
column_names=[x for x in column_names if x[0:5]!='split"]
#creiamo un nuovo dataframe eliminando queste colonne
data=risultati.drop(column_names,axis=1)

#creiamo due nuovi dataframe con solo split test e split train
column_names=list(data)
data_train=data.drop(column_names[:int(len(column_names)/2)],axis = 1)
data_test=data.drop(column_names[int(len(column_names)/2):],axis = 1)
display(data_train.iloc[[0,1, -1]])
display(data_test.iloc[[0,1, -1]])
```

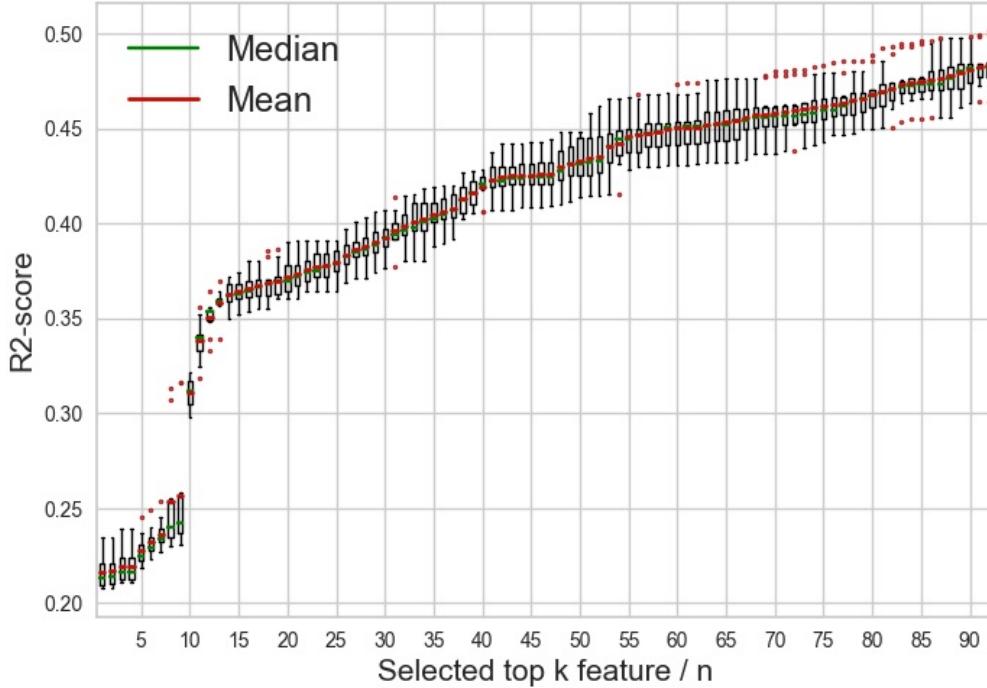
	split0_train_score	split1_train_score	split2_train_score	split3_train_score	split4_train_score	split5_train_score	split6_train_score	split7_t
0	0.212310	0.219016	0.234517	0.208945	0.225696	0.207641	0.213849	
1	0.212442	0.219297	0.234740	0.209534	0.225910	0.208058	0.214026	
91	0.472678	0.484746	0.501256	0.499460	0.480951	0.484756	0.476737	
	split0_test_score	split1_test_score	split2_test_score	split3_test_score	split4_test_score	split5_test_score	split6_test_score	split7_test_s
0	0.257723	0.190931	0.075287	0.268805	0.102954	0.266742	0.238441	0.245
1	0.258982	0.190739	0.075642	0.265832	0.103453	0.265397	0.239143	0.246
91	0.529608	0.362194	0.269951	0.293415	0.380435	0.389637	0.420415	0.447

Plottiamo le performance al variare del parametro k Nell'asse x il numero di feature selezionate, nell'asse y il relativo error\_rate per il nostro dataset.

```
In [25]: #plot train
boxdata=data_train.values
flierprops = dict(marker='.', markeredgecolor='firebrick', markersize=3, linestyle='none')
medianprops = dict(linestyle='-', linewidth=1.75, color='green')
meanprops = dict(linestyle='--', linewidth=2, color='r')

bp=plt.boxplot(boxdata.tolist(), flierprops=flierprops, medianprops=medianprops, showmeans= True, meanline = True)
plt.xticks([x for x in range(1,len(risultati["mean_train_score"])+1) if x%5==0],[str(x) for x in param[0]["sele
plt.xlabel("Selected top K feature / n", fontsize=15)
plt.ylabel("R2-score", fontsize=15)
plt.grid(True)
plt.legend([bp['medians'][0], bp['means'][0]], ['Median', 'Mean'], fontsize='x-large')
plt.title("Mean TRAIN R2-Score al variare del numero di feature", fontsize=20)
plt.show()
```

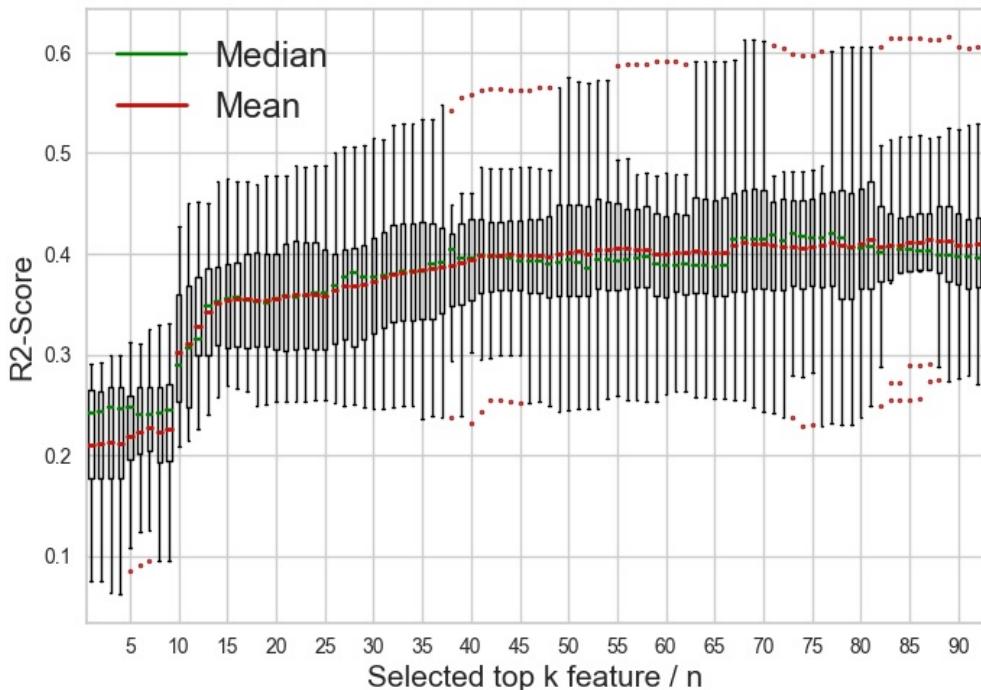
## Mean TRAIN R2-Score al variare del numero di feature



```
In [26]: #plot test
boxdata=data_test.values
flierprops = dict(marker='.',markeredgecolor='firebrick', markersize=3, linestyle='none')
medianprops = dict(linestyle='--', linewidth=1.75, color='green')
meanprops = dict(linestyle='--', linewidth=2,color='r')

bp=plt.boxplot(boxdata.tolist(), flierprops=flierprops, medianprops=medianprops, showmeans= True, meanline = True)
plt.xticks([x for x in range(1,len(resultati["mean_test_score"])+1) if x%5==0],[str(x) for x in param[0]["selezione"]])
plt.xlabel("Selected top k feature / n",fontsize=15)
plt.ylabel("R2-Score",fontsize=15)
plt.grid(True)
plt.legend([bp['medians'][0], bp['means'][0]], ['Median', 'Mean'], fontsize='x-large')
plt.title("Mean TEST R2-Score al variare del numero di feature",fontsize=20)
plt.show()
```

## Mean TEST R2-Score al variare del numero di feature



Alla luce dei seguenti dati potrebbe essere conveniente non utilizzare il miglior estimatore, in quel caso basterebbe creare una pipeline allo stesso modo, utilizzando come parametro il parametro desiderato, senza l'utilizzo di GridSearchCV

Es (k=20):

```
best = Pipeline([('select',SelectKBest(score_func=f_regression,k=20)),('scaler', StandardScaler()), ('regr', linear_model.LinearRegression())])
```

Calcoliamo Akaike Information Criterion (**AIC**) per il modello con feature da 1-92 e vediamo il contenuto informativo

```
In [27]: #import statsmodels.api as sm
from statsmodels.regression.linear_model import OLS
from statsmodels.tools import add_constant

AICs=[]
print("Fitting model:",end=" ")
for sk in [x+1 for x in range(92)]:

    print(f"{sk}.",end="")
    SKB=SelectKBest(score_func=f_regression, k=sk)
    t_X=SKB.fit_transform(F_x,F_y)
    SSC=StandardScaler()
    t_X=SSC.fit_transform(t_X)

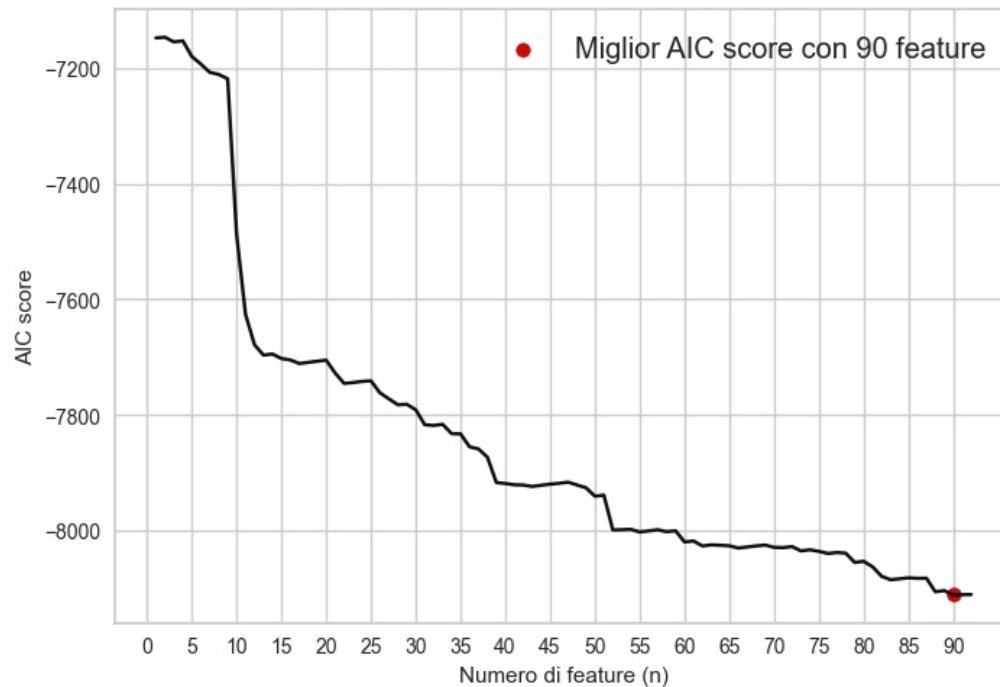
    #fit regression model
    m_a = OLS(F_y, add_constant(t_X)).fit()

    AICs.append(m_a.aic)

Fitting model: 1.2.3.4.5.6.7.8.9.10.11.12.13.14.15.16.17.18.19.20.21.22.23.24.25.26.27.28.29.30.31.32.33.34.35.
36.37.38.39.40.41.42.43.44.45.46.47.48.49.50.51.52.53.54.55.56.57.58.59.60.61.62.63.64.65.66.67.68.69.70.71.72.
73.74.75.76.77.78.79.80.81.82.83.84.85.86.87.88.89.90.91.92.
```

\*Plottiamo\*

```
In [38]: plt.plot([x+1 for x in range(92)],AICs,color="k")
plt.xticks([x for x in range(92) if x%5==0])
plt.scatter(AICs.index(min(AICs))+1,min(AICs),color="r",label=f"Miglior AIC score con {AICs.index(min(AICs))+1}")
plt.xlabel("Numero di feature (n)")
plt.ylabel("AIC score")
plt.grid(True)
plt.legend(fontsize="large")
plt.show()
print(np.array(AICs))
```



```

[-7147.57626034 -7146.4099979 -7154.56152995 -7152.56547477
-7179.6503326 -7192.68402838 -7207.06832261 -7210.74612499
-7218.21804849 -7487.42382512 -7626.05006166 -7678.4934891
-7696.47544157 -7694.57533087 -7702.32213144 -7704.77382838
-7710.94807752 -7708.95881522 -7706.97025966 -7705.12131411
-7727.38664056 -7745.35991524 -7743.89249466 -7741.95098484
-7740.92507056 -7761.62127321 -7772.14854516 -7782.27513755
-7781.50653641 -7790.98645069 -7816.80422753 -7818.07951266
-7816.14758378 -7832.31130055 -7832.71414404 -7855.03476271
-7858.75283916 -7873.12198801 -7916.86508746 -7918.57698713
-7920.75356255 -7921.19638076 -7923.67444322 -7921.6763277
-7919.74627347 -7918.21657254 -7916.23459079 -7921.13331517
-7925.95982562 -7940.47140903 -7938.65415845 -7998.9103631
-7998.51598951 -7997.85028369 -8002.41028601 -8000.5556268
-7998.5794279 -8001.87563788 -8000.35487056 -8019.67310222
-8017.95749769 -8026.62324354 -8024.75411949 -8025.44115936
-8026.36166932 -8030.20230076 -8028.36689356 -8026.57048029
-8025.04130572 -8029.040766 -8029.5438327 -8027.67903491
-8035.15714788 -8033.25645498 -8035.94935821 -8039.65539277
-8037.99327994 -8039.11154547 -8055.14275105 -8053.19540567
-8062.78063934 -8079.11071201 -8085.02557166 -8083.48130597
-8081.79931335 -8082.69291317 -8082.49168771 -8105.90788209
-8103.94387328 -8111.37146148 -8110.87800569 -8110.73725155]

```

Essendo il modello molto leggero il tempo di fit è sempre uguale, quindi è ovvio che avere più feature sia più informativo e quindi un AIC score migliore

```
+++++
```

## Visualize regression with yellowbrick

**y** = true value

**$\hat{y}$**  = predicted value

**Residuals** =  $y_{\text{pred}} - y_{\text{true}}$

Purtroppo non è possibile cambiare lo scoring method della figura, quindi nella legenda comparirà  $R^2$  invece del nostro error rate, che viene quindi calcolato separatamente.

---

----- USE BEST PREDICTOR ON TRAIN DATA -----

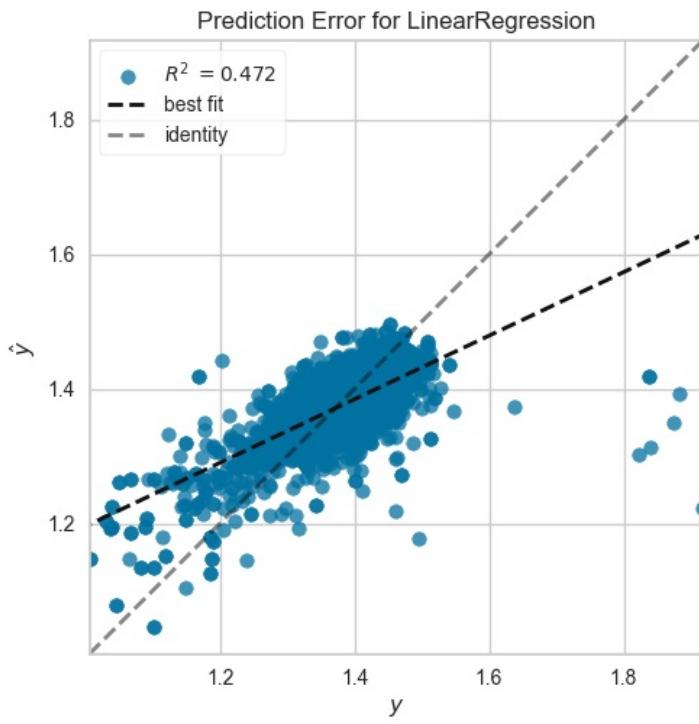
CALCOLIAMO l'error\_rate medio nel train.

```
In [29]: y_pred=best.predict(F_x)
print(f"\nErrore medio del dataset: {error_rate(F_y,y_pred):.3f} % !")
```

Errore medio del dataset: 2.755 % !

```
In [30]: #F_x feature 92 #F_y target
visualizer= PredictionError(best,bestfit=True)

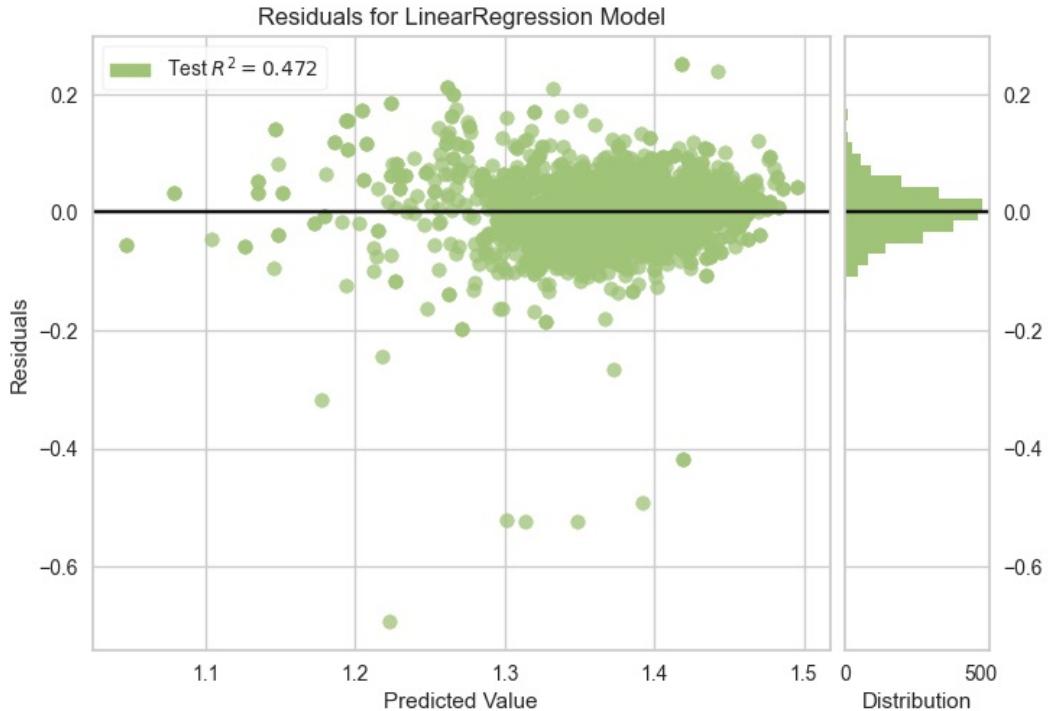
#visualizer.fit(F_x, F_y) # Fit the training data in the visualizer # best is already fitted on F_x F_y!
visualizer.score(F_x, F_y)#train # Evaluate the model on data (train or test)
visualizer.show() # Finalize and render the figure ;
```



```
Out[30]: <AxesSubplot:title={'center':'Prediction Error for LinearRegression'}, xlabel='$y$', ylabel='$\hat{y}$'>
```

```
In [31]: visualizer=ResidualsPlot(best)
```

```
#visualizer.fit(F_x, F_y)# Fit the training data in the visualizer # best is already fitted on F_x F_y!
visualizer.score(F_x, F_y)#train # Evaluate the model on data (train or test)
visualizer.show() # Finalize and render the figure ;
#facendo il fit questo grafico restituisce anche il train, ma nel nostro caso train e test sono gli stessi, #st.
```



```
Out[31]: <AxesSubplot:title={'center':'Residuals for LinearRegression Model'}, xlabel='Predicted Value', ylabel='Residuals'%gt;
```

Dall'istogramma laterale vediamo anche la distribuzione dell'errore in metri!

----- USE BEST PREDICTOR ON TEST DATA -----

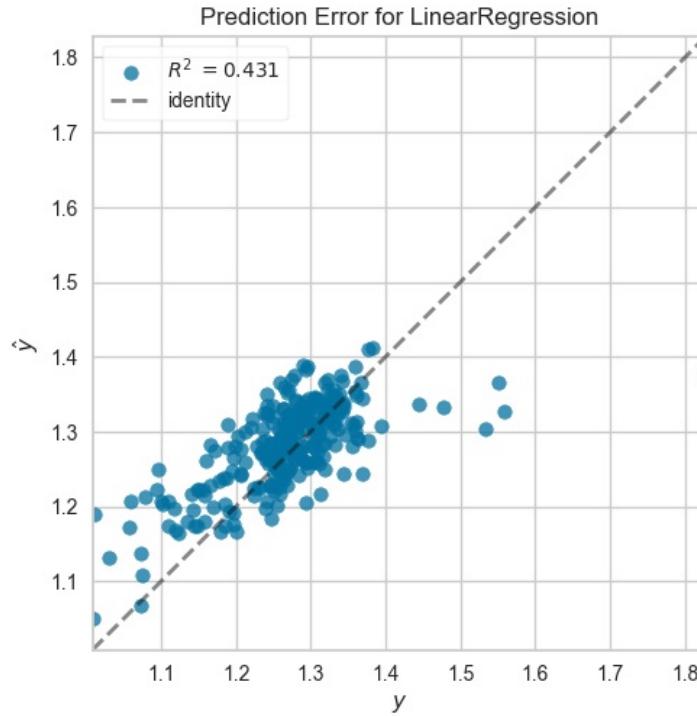
```
In [32]: #predict with best predictor
b_pred=best.predict(F_x_test)
```

```
#evaluate error_rate
print(f"Error rate: {error_rate(F_y_test,b_pred):.3f}%")
```

Error rate: 3.580%

In [33]: `visualizer = PredictionError(best, bestfit=False)`

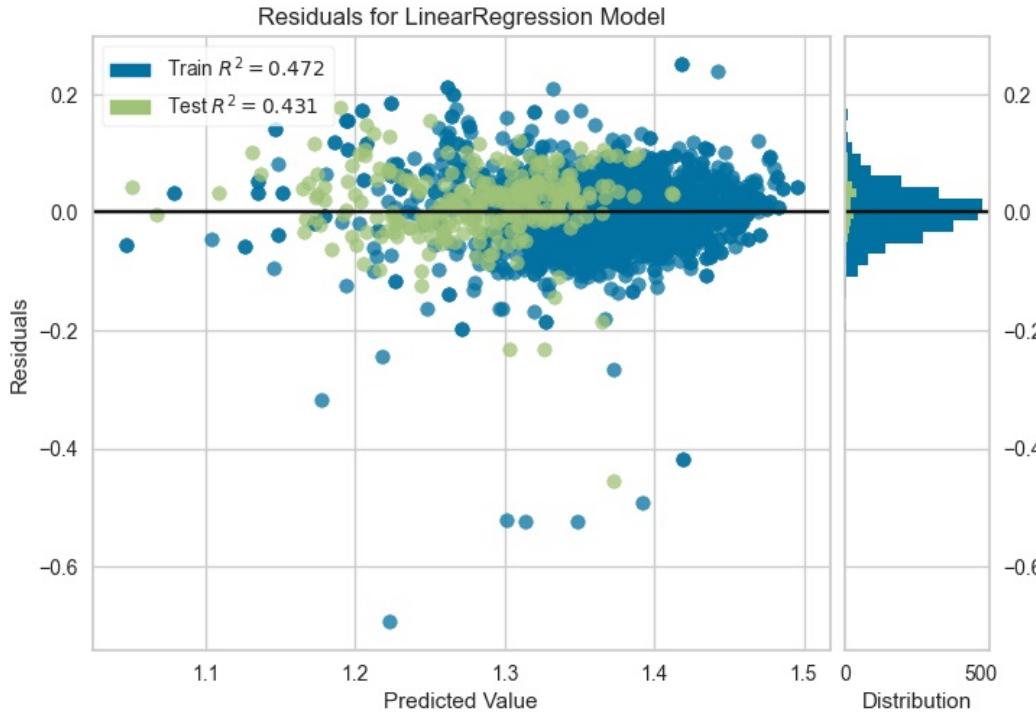
```
#visualizer.fit(F_x, F_y) # Fit the training data in the visualizer # best is already fitted on F_x F_y!
visualizer.score(F_x_test, F_y_test)#test # Evaluate the model on data (train or test)
visualizer.show() # Finalize and render the figure
```



Out[33]: <AxesSubplot:title={'center':'Prediction Error for LinearRegression'}, xlabel='\$y\$', ylabel='\$\hat{y}\$'>

In [34]: `visualizer = ResidualsPlot(best)`

```
visualizer.fit(F_x, F_y) # Fit the training data in the visualizer # best is already fitted on F_x F_y!
visualizer.score(F_x_test, F_y_test)#test # Evaluate the model on data (train or test)
visualizer.show() # Finalize and render the figure
```



Out[34]: <AxesSubplot:title={'center':'Residuals for LinearRegression Model'}, xlabel='Predicted Value', ylabel='Residua ls'>

Dall'istogramma laterale vediamo anche la distribuzione dell'errore in metri!

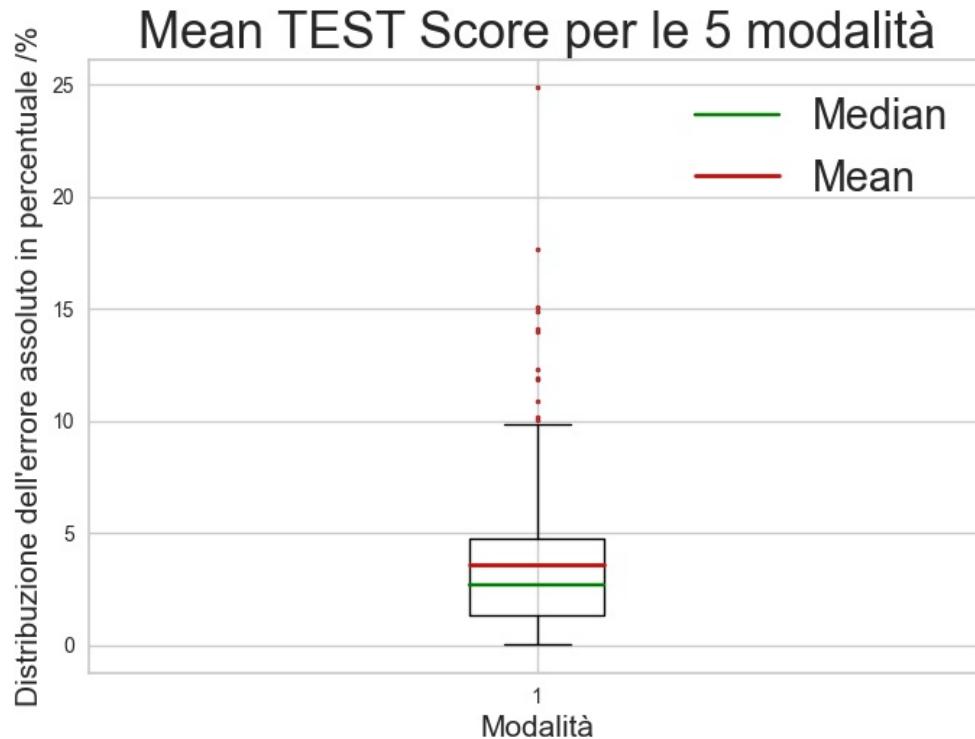
Vediamo l'R2 score e plottiamolo sopra il boxplot che rappresenta la distribuzione dell'

## ERRORE IN METRI

```
In [35]: #predict
b_pred=best.predict(F_x_test)
mt_err_dist=(abs(b_pred-F_y_test)/F_y_test)*100
r2_scores=r2_score(F_y_test,b_pred)

flierprops = dict(marker='.',markeredgecolor='firebrick', markersize=3, linestyle='none')
medianprops = dict(linestyle='--', linewidth=1.75, color='green')
meanprops = dict(linestyle='--', linewidth=2,color='r')

bp1=plt.boxplot(mt_err_dist.tolist(), flierprops=flierprops, medianprops=medianprops, showmeans= True, meanline
# plt.text(1,np.mean(mt_err_dist)+np.std(mt_err_dist),int(r2_scores*100)/100, horizontalalignment="center",color=
plt.xlabel("Modalità",fontsize=15)
plt.ylabel("Distribuzione dell'errore assoluto in percentuale %",fontsize=15)
plt.grid(True)
plt.legend([bp1['medians'][0], bp1['means'][0]], ['Median', 'Mean'], fontsize='xx-large')
plt.title("Mean TEST Score per le 5 modalità",fontsize=25)
plt.show()
```



```
In [36]: print(f"sum(F_y_test-b_pred)/len(F_y_test) :.2f")
&-0.02
```

```
In [37]: print(f"np.mean(mt_err_dist):.2f}\n&{np.std(mt_err_dist):.2f}\n")
```

3.58%&3.28%

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

## *Ringraziamenti*

*Ringrazio i professori che mi hanno seguito nel mio corso di studi, ed in particolare i professori che mi hanno aiutato in questo lavoro di tesi il Chiar.ssmo Luca Patanè ed il Dott. Cristiano De Marchis.*

*Ringrazio la mia famiglia che mi ha supportato durante i tre anni di studio.*

*Ringrazio tutti gli amici e colleghi che in questi tre anni mi sono stati vicino e mi hanno aiutato, in particolare ad Alessandro Ficarra, Alessandro T., Marco, Gabriele, chiunque altro si sia interfacciato con me durante questi tre anni all'università ed un ringraziamento speciale alla mia ragazza, Rita, per avermi supportato.*