Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica per il Management

# PROGETTAZIONE E VALIDAZIONE DI UN FRAMEWORK DI ALGORITMI ENSEMBLE PER LA CLASSIFICAZIONE DI OPEN DATA IOT

Relatore:
Chiar.mo Prof.
Marco Di Felice

Correlatore:
Dott.
Federico Montori

Presentata da:
Matteo De Giosa

II Sessione
Anno Accademico 2018/2019

# Abstract

La quantità di dati IoT liberamente accessibili in rete - comunemente chiamati Open Data - è potenzialmente di grande utilità per innumerevoli applicazioni pratiche. Tuttavia, tali dati sono spesso inutilizzabili o incomprensibili, al punto in cui talvolta non si riesce nemmeno a discernere la tipologia di osservazione effettuata. Per etichettare tali misurazioni è dunque necessaria l'applicazione di modelli di classificazione. Questo tuttavia non è un lavoro semplice, in quanto i dati open sono in generale molto eterogenei, per cui molti degli algoritmi comunemente usati in letteratura hanno difficoltà a classificarli correttamente. Il contributo maggiore di questa tesi è perciò la presentazione di MACE, un framework ensemble per la classificazione di Open Data IoT: dopo averne trattato progettazione ed implementazione, ne valuteremo le performance, dimostrando la sua efficacia nel risolvere quello che è, ad oggi, un problema decisamente trascurato dalla letteratura.

# Indice

**Introduzione** **7**

**1 Stato dell'arte** **9**

1.1 Machine learning . . . 9

1.1.1 Algoritmi di classificazione di base . . . 10

1.1.2 Algoritmi ensemble . . . 14

1.2 Applicazioni di Machine Learning in Sistemi IoT . . . 15

1.2.1 Algoritmi di tipo BOS . . . 17

1.2.2 Algoritmi di tipo TSC . . . 17

1.2.3 Algoritmi NLP . . . 19

**2 Analisi preliminari** **21**

2.1 Strumenti . . . 22

2.2 Dataset . . . 23

2.3 Misurazione delle performance . . . 24

2.4 Algoritmi BOS . . . 25

2.4.1 Ottimizzazione dei parametri . . . 26

2.4.2 Risultati . . . 27

2.5 Algoritmi TSC . . . 27

2.5.1 Normalizzazione . . . 28

2.5.2 1NN-ED . . . 30

2.5.3 1NN-DTW . . . 31

2.5.4 BOPF . . . 33

2.5.5 SDE . . . 34

2.5.6 Risultati . . . . . . . . . . 35
2.6 Algoritmi NLP . . . . . . . . . . 38
2.6.1 Risultati . . . . . . . . . . 39
2.7 Ensemble BOS . . . . . . . . . . 40
2.7.1 Bagging . . . . . . . . . . 40
2.7.2 Voting . . . . . . . . . . 42
2.7.3 Boosting . . . . . . . . . . 44
2.7.4 Stacking . . . . . . . . . . 44

**3 Metadata-Assisted Cascading Ensemble - MACE 49**
3.1 Descrizione del framework . . . . . . . . . . 50
3.2 Criterio di ordinamento dei classificatori . . . . . . . . . . 51
3.3 Criteri di filtraggio . . . . . . . . . . 51
3.3.1 Strategie Rank-Based . . . . . . . . . . 53
3.3.2 Strategie Distribution-Based . . . . . . . . . . 54
3.4 Implementazione . . . . . . . . . . 54
3.4.1 Modifiche agli algoritmi . . . . . . . . . . 54
3.4.2 Preprocessing del dataset . . . . . . . . . . 57
3.4.3 Classi wrapper . . . . . . . . . . 58
3.4.4 Implementazione dei criteri di filtraggio . . . . . . . . . . 59

**4 Validazione 63**
4.1 Tuning dei parametri . . . . . . . . . . 63
4.2 Risultati finali . . . . . . . . . . 64

**5 Conclusioni e sviluppi futuri 67**
5.1 Sviluppi futuri . . . . . . . . . . 68

**Ringraziamenti 69**

**Bibliografia 71**

# Elenco delle figure


1.1 Struttura di un albero decisionale. . . . . . . . . . . . . . . . . . . . . . . 10
1.2 Scatterplot di un insieme di dati dopo aver applicato SVM. La retta che li divide è l'iperpiano separatore, l'area grigia rappresenta i margini. . . . 11
1.3 Esempio di applicazione di kNN. L'oggetto in esame è il pallino centrale. Se la classe viene decisa per voto di maggioranza, per k = 3 il dato appartiene ai triangoli rossi, se k = 5 appartiene ai quadrati blu. . . . . . 13
1.4 Struttura di una serie temporale. Sull'asse x la successione delle osservazioni, sulla y il loro valore. . . . . . . . . . . . . . . . . . . . . . . . . 16

2.1 Fonte: [7] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
2.2 Accuracy a confronto tra standard e optimized classifiers su Swissex . . . 27
2.3 f1-score a confronto tra standard e optimized classifiers su Swissex . . . . 28
2.4 Accuracy a confronto tra standard e optimized classifiers su Thingspeak . 28
2.5 f1-score a confronto tra standard e optimized classifiers su Thingspeak . . 29
2.6 Accuracy a confronto tra standard e optimized classifiers su UrbanObservatory . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
2.7 f1-score a confronto tra standard e optimized classifiers su UrbanObservatory 30
2.8 Accuracy a confronto tra dati standard e normalizzati su Swissex . . . . 35
2.9 f1-score a confronto tra dati standard e normalizzati su Swissex . . . . . 35
2.10 Accuracy a confronto tra dati standard e normalizzati su Thingspeak . . 36
2.11 f1-score a confronto tra dati standard e normalizzati su Thingspeak . . . 36
2.12 Accuracy a confronto tra dati standard e normalizzati su UrbanObservatory 37
2.13 f1-score a confronto tra dati standard e normalizzati su UrbanObservatory 37

2.14 Accuracy e f1-score a confronto con diverse edit-distances su Thingspeak 39
2.15 Accuracy a confronto tra singoli classifiers ed ensemble bagging su Swissex 41
2.16 f1-score a confronto tra singoli classifiers ed ensemble bagging su Swissex 41
2.17 Accuracy a confronto tra singoli classifiers ed ensemble bagging su Thingspeak . . . 42
2.18 f1-score a confronto tra singoli classifiers ed ensemble bagging su Thingspeak 42
2.19 Accuracy a confronto tra singoli classifiers ed ensemble bagging su UrbanObservatory . . . 43
2.20 f1-score a confronto tra singoli classifiers ed ensemble bagging su UrbanObservatory . . . 43
2.21 Accuracy e f1-score di tutti i classificatori ensemble, su Swissex . . . 46
2.22 Accuracy e f1-score di tutti i classificatori ensemble, su Thingspeak . . . 47
2.23 Accuracy e f1-score di tutti i classificatori ensemble, su UrbanObservatory 47

3.1 Overview del framework . . . 50
3.2 *top-accuracy* dei principali classificatori al variare di k . . . 52

4.1 Performance di QF e SoF per diversi valori di $z$. Sono mostrate solo le combinazioni migliori. . . . 64

# Introduzione

La diffusione e la crescita esponenziale del numero di dispositivi connessi ad Internet è sicuramente un vantaggio in ambito tecnologico, ma anche un elemento abilitante di molti sistemi di utilità sociale: le applicazioni pratiche dell'IoT ci consentono di affrontare problemi - Smart Cities, Environment Monitoring, Healthcare - prima ritenuti irrisolvibili. Ma con tali sviluppi nascono anche dei problemi: una percentuale rilevante dei dati prodotti da dispositivi IoT è inutilizzabile.

Ciò avviene in particolar modo per i dati open, cioè i dati rilasciati pubblicamente da enti (ad esempio https://dati.lombardia.it) o caricati direttamente dagli utenti attraverso meccanismi di crowdsourcing (ad esempio https://thingspeak.com).

Il problema principale di questi dati è che per essere utilizzati hanno bisogno di metadati, ovvero informazioni sui dati: tipologia di osservazione, unità di misura, posizione in cui è stata effettuata la misura, etc. La maggior parte dei dati open non è fornita di metadati, li ha in forma incompleta o comunque non identificabili da una macchina [28].

Nasce dunque l'esigenza di etichettare questi dati, esigenza che si traduce in termini di machine learning nella creazione di modelli di classificazione, che attribuiscano cioè ad ogni misurazione una classe (che in genere coincide con la tipologia di osservazione). Tuttavia il problema della classificazione di Open Data non è trattato opportunamente nella letteratura scientifica, e già di per sé pone diverse problematiche, tutte causate dalla natura eterogenea dei dati: ogni dato è infatti catturato con modalità diverse i.e. diversa location, diversi intervalli di misura, diversi dispositivi di misurazione, etc.

L'obiettivo di questa tesi sarà pertanto duplice: la prima parte sarà di natura me-

ramente compilativa, in quanto prima esporremo e poi testeremo quello che è lo stato dell'arte, ovvero ciò che di meglio la letteratura ha da offrire in ambito classificazione di Open Data IoT; la seconda parte sarà invece di natura sperimentale: proporremo, discutendone estensivamente progettazione e implementazione, un nostro framework per l'etichettatura dei dati sopracitati. Infine cercheremo di dimostrare la validità del contributo offerto testando il framework su un dataset particolarmente eterogeneo e dotato di metadati incompleti.

La tesi è pertanto divisa in 5 capitoli:

- Nel Capitolo 1 (Stato dell'Arte) introdurremo e discuteremo i temi del machine learning e dell'IoT; nella prima parte introdurremo gli algoritmi di machine learning più comunemente usati, nella seconda invece quelli specifici per dati IoT - che in genere si presentano sotto forma di serie temporali -
- Nel Capitolo 2 (Analisi Preliminari) implementeremo e testeremo gli algoritmi esposti nel capitolo precedente. Questo lavoro ci servirà come punto di partenza e di riferimento per sviluppare poi il nostro framework e confrontarne i risultati
- Nel Capitolo 3 (Metadata-Assisted Cascading Ensemble) presenteremo il framework sviluppato: esporremo formalmente il suo funzionamento e tratteremo i punti critici della sua progettazione e implementazione
- Nel Capitolo 4 (Validazione) esporremo i risultati ottenuti dal framework e li discuteremo
- Nel Capitolo 5 (Conclusioni e Sviluppi Futuri) faremo un riepilogo di quanto sviluppato nel corso di questa tesi ed elencheremo le direzioni in cui il progetto si può sviluppare

# Capitolo 1

# Stato dell'arte

## 1.1 Machine learning

La grande mole di dati pubblicata in ambito IoT - e in particolare in ambito Open Data - rende necessario l'utilizzo di tecniche di Machine Learning per classificare tali dati.

Il Machine Learning - in italiano potremmo tradurlo con "apprendimento automatico" è un insieme di pratiche volte alla creazione di algoritmi e sistemi computerizzati che "imparino" dai dati in maniera autonoma [8].

Gli algoritmi di machine learning si dividono in due categorie fondamentali:

- Supervised Learning: l'algoritmo analizza i dati a partire da un set di dati detto di "training" o "addestramento" e costruisce un modello decisionale alla base di quello: il training set contiene sia le variabili di input che di output, cosicchè l'algoritmo impari ad associare le due; in seguito, il modello viene testato su un test set, e le variabili di output prodotte vengono comparate con quelle effettive.

- Unsupervised Learning: l'algoritmo non ha a disposizione un set di dati di apprendimento da cui imparare: raggruppa perciò i dati senza seguire schemi e senza avere informazioni preliminari.

Esempi di tecniche unsupervised sono il clustering e la dimensionality reduction, di quelle supervised la regressione e la classificazione: in ambito Open Data IoT ci concentreremo su quest'ultima.

Classificare un dato significa annotarlo, associargli una classe: nel nostro caso, ad esempio, la classe di un dato corrisponde alla tipologia di misura registrata (temperatura, pressione atmosferica, etc.).

L'algoritmo di classificazione prenderà in input un set di addestramento con dei dati già etichettati, e sulla base di questo costruirà un modello: quest'ultimo dovrà imparare a "generalizzare" il collegamento tra dato e classe, così da performare bene anche su dati mai visti in precedenza.

Esistono numerosi algoritmi di classificazione "di base" [15]; di seguito presenteremo quelli rilevanti ai fini di questa tesi.

### 1.1.1 Algoritmi di classificazione di base

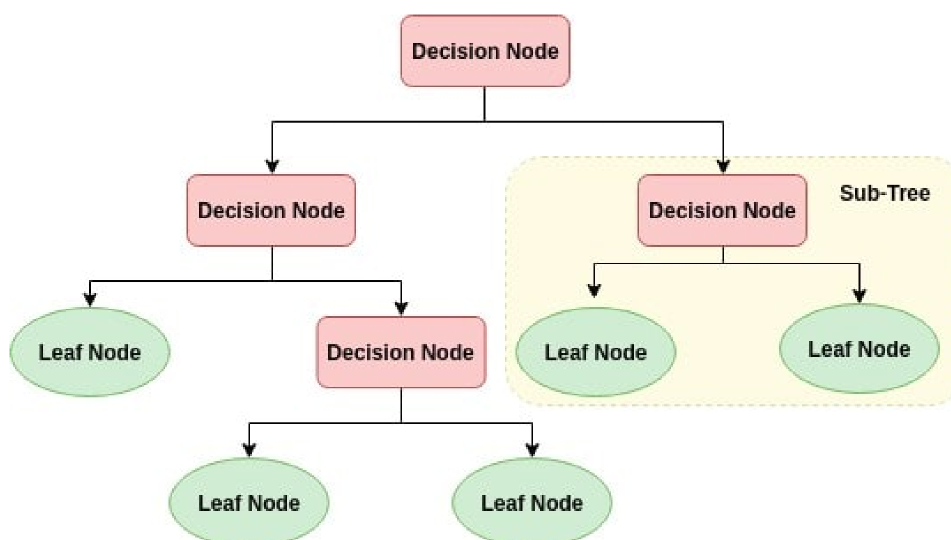



Figura 1.1: Struttura di un albero decisionale.

- Decision Tree: obiettivo dell'algoritmo è costruire un modello di predizione come quello nell'immagine 1.4. I passaggi sono i seguenti:

– Viene selezionata la feature migliore in base alla quale operare uno split del dataset (ad esempio, media $>100$)

– Rendere quella feature un Decision Node e dividere il dataset in due parti

– Ripetere il processo ricorsivamente per i nodi figli finché non ci sono più istanze nel dataset o tutte le istanze rimaste appartengono alla stessa classe

La selezione della 'regola' per lo split viene effettuata in base a una metrica che in genere misura la diminuzione dell'entropia (l'impurità) del set di dati in seguito allo split. Le metriche più popolari sono l'Information Gain, il Gain Ratio e il Gini Index.

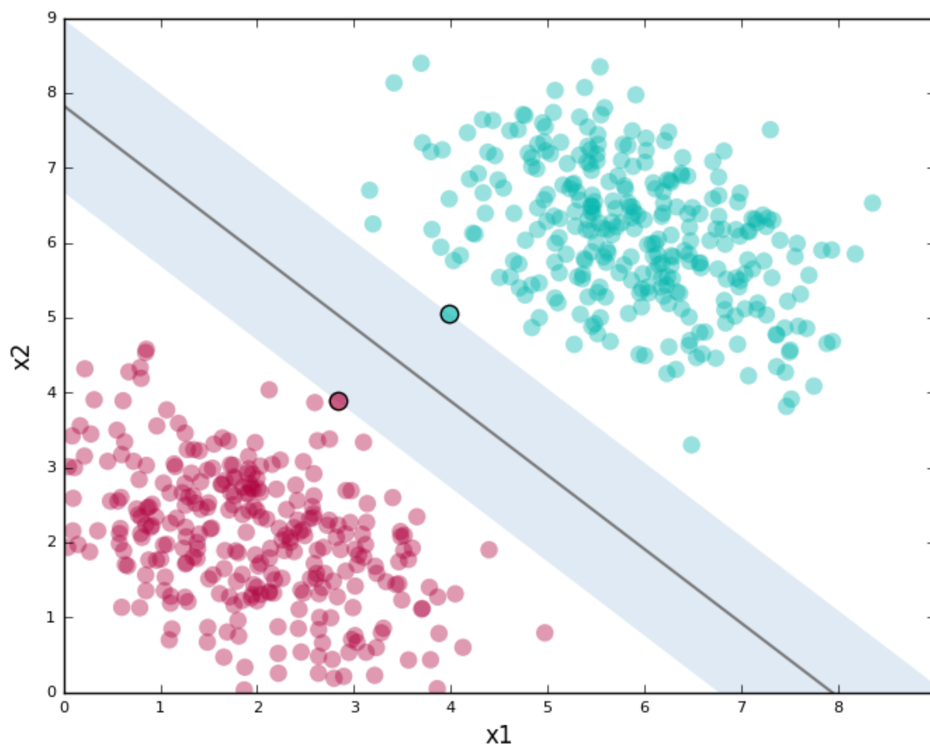



Figura 1.2: Scatterplot di un insieme di dati dopo aver applicato SVM. La retta che li divide è l'iperpiano separatore, l'area grigia rappresenta i margini.

- Support Vector Machine: algoritmo di classificazione binario, opera cioè su sole due classi alla volta. Come possiamo vedere nella figura 1.2, l'obiettivo della Support

Vector Machine è trovare l'iperpiano ottimale che separi le due classi, massimizzando il margine (la distanza dell'elemento più vicino di entrambe le classi). Un iperpiano è un sottospazio di dimensione n-1 rispetto allo spazio in cui è contenuto: quindi, se si parla di punti in uno spazio 2D, l'iperpiano sarà una retta, in uno spazio 3D sarà un piano, e così via. Ovviamente quello in figura è il caso ottimo, in cui cioè esiste un iperpiano che separi completamente i punti delle due classi; nella realtà ciò non succede, per cui vi si applicano degli aggiustamenti, sotto forma di Soft Margins o Kernel Tricks: nel primo caso si concede al modello un margine di errore (cioè uno o più punti di una classe si possono trovare nell'"area" dell'altra classe); nel secondo caso si trova un iperpiano non lineare, applicando opportune trasformazioni alle features iniziali. Per adattare le SVM a problemi di classificazione multi-classe, la libreria sklearn (quella che useremo in questa tesi) applica un approccio 'one-against-one', cioè crea un modello per ogni coppia di classi presente. Nella fase di classificazione poi si applica un semplice voto di maggioranza.

- k-Nearest-Neighbors (kNN): tale algoritmo è uno dei più semplici (e nonostante ciò, anche uno dei più usati ed efficaci) algoritmi di machine learning. L'idea dietro è quasi banale: la classe di un oggetto è la stessa dei k oggetti a lui più vicini. Nel caso di 1-nearest-neighbor (1NN), è il singolo oggetto più vicino a determinare la classe dell'oggetto in esame. Quello che però può essere più complicato è la misura della distanza tra due oggetti: la metrica standard è la distanza euclidea tra due punti, ma a seconda del problema in esame se ne possono usare diverse (vedremo più avanti la distanza con il modello Dynamic Time Window, per le serie temporali, e le edit-distances, per misurare la distanza tra parole). Nel caso di $k > 1$ è previsto un sistema di votazione. Il valore k viene spesso determinato attraverso tecniche di euristica come la cross-validation.

- Regressione logistica: modello di regressione lineare usato per problemi di classificazione binaria: vogliamo sapere se l'istanza appartiene o meno ad una determinata classe. A questo scopo viene usata la funzione sigmoidea, che ha come range di valori sull'asse y [0,1]; il valore della funzione rappresenta la probabilità che l'oggetto appartenga alla classe. Nel nostro caso useremo un modello di regressione logistica

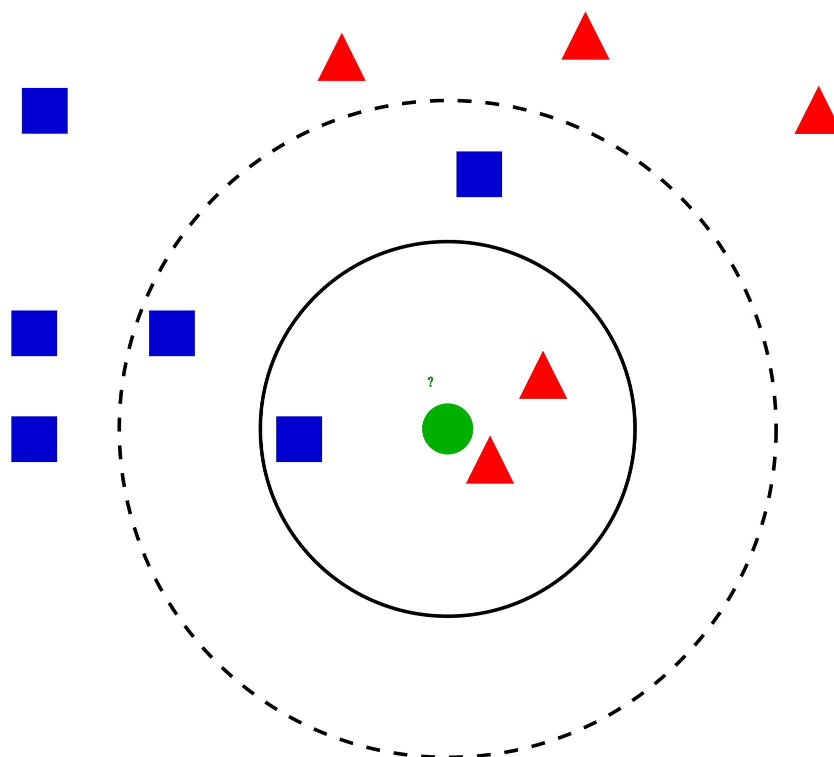



Figura 1.3: Esempio di applicazione di kNN. L'oggetto in esame è il pallino centrale. Se la classe viene decisa per voto di maggioranza, per k = 3 il dato appartiene ai triangoli rossi, se k = 5 appartiene ai quadrati blu.

multi-classe, la cui spiegazione matematica è fuori dalla portata di questa tesi.

- Ridge: particolare tipo di regressione lineare in cui vengono penalizzati i coefficienti troppo grandi. Questo è un aggiustamento molto utile nel caso in cui le variabili indipendenti (le features) siano correlate tra loro: ciò porta, nella regressione lineare classica, ad un modello "overfitted", che cioè si comporta egregiamente sul dataset di training, ma non classifica bene nuovi dati.

- Naive Bayes: prende il nome dal teorema di Bayes, ed è definito 'naive' perché parte da delle semplici ma efficaci assunzioni: che le features siano indipendenti tra loro e che siano tutte di uguale importanza. Dal teorema di Bayes,

$$P(y \mid X) = \frac{P(X \mid y)\, P(y)}{P(X)}$$

con y la classe e X la lista di features, definiamo $P(y \mid X)$ come la probabilità condizionata che l'osservazione appartenga alla classe, $P(X \mid y)$ come la probabilità che gli elementi della classe presentino le feature X, $P(y)$ come la probabilità che un elemento del dataset appartenga alla classe e $P(X)$ come la probabilità che un elemento del dataset presenti le feature X. Per un numero di feature $> 1$ la formula si trasforma in:

$$P(y \mid x_1, \ \dots \ , x_n) = \frac{P(x_1 \mid y)\, P(x_2 \mid y) \ \ \dots \ \ P(x_n \mid y)\, P(y)}{P(x_1)\, P(x_2) \ \ \dots \ \ P(x_n)}$$

Perché come abbiamo detto il modello considera le feature come indipendenti. Un classificatore Naive Bayes dunque calcola $P(y \mid X)$ per ogni classe y e assegna alla misurazione X la classe con probabilità più alta:

$$y = \arg\max_y P(y) \prod_{i=1}^{n} P(x_i \mid y)$$

### 1.1.2 Algoritmi ensemble

Gli algoritmi ensemble sono costituiti da più classificatori di base combinati tra loro, secondo la filosofia che una combinazione di classificatori fornisca migliori risultati rispetto al singolo [24]. Esistono diverse tipologie di ensemble:

- Bagging: un modello di ensemble di tipo Bagging è una combinazione di modelli 'deboli', ciascuno dei quali apprende da un sottoinsieme dei dati iniziali. La predizione finale altro non è che la media (o il voto di maggioranza) dell'output dei vari modelli Ensemble di questo tipo è per esempio Random Forest, composto da n decision trees.

- Voting: un ensemble di questo tipo è una versione 'semplificata' del Bagging, ma che ci permette però di unire i risultati di diverse categorie di classificatori (il Bagging duplica invece sempre lo stesso).

- Boosting: molto simile al Bagging, nel senso che combina i risultati di modelli singolarmente 'deboli'. Tuttavia, non lo fa alla fine, ma sequenzialmente: ogni modello viene addestrato sui risultati del modello precedente, dando più peso ogni volta alle predizioni errate. Ensemble di questo tipo sono AdaBoost e GradBoost.

- Stacking: un ensemble di questo tipo è strutturato su più livelli: l'output fornito dai classificatori di un livello viene dato in pasto ad altri classificatori, chiamati 'meta-classificatori'.

- Cascading: composto da una "cascata" di classificatori che vengono interrogati sequenzialmente quando quello precedente non fornisce una certa sicurezza (leggasi: confidenza) sui risultati ottenuti. Particolarmente usati ad esempio in ambito bancario (e.g. per assicurarsi che una transazione non sia fraudolenta). In questa tesi progetteremo, implementeremo e testeremo un algoritmo di questo tipo.

## 1.2 Applicazioni di Machine Learning in Sistemi IoT

Con il termine IoT (Internet of Things, Internet delle Cose) si indica un insieme di tecnologie che permettono di collegare ad Internet qualsiasi tipo di apparato.

Lo scopo di questo tipo di soluzioni è sostanzialmente quello di monitorare e controllare e trasferire informazioni per poi svolgere azioni conseguenti: molte di queste azioni (se escludiamo la "semplice" reportistica o l'analisi dei dati) richiedono l'utilizzo di modelli di machine learning.

IoT e machine learning sono infatti un connubio comprovato già da parecchi anni, e molte sono le applicazioni reali e i casi di studio che possiamo citare:

- Activity recognition [23]: dal classico fitness tracker che abbiamo al polso alle soluzioni più industriali, come quella proposta in [9], in cui gli autori sviluppano un modello di classificazione dell'attività dei lavoratori, per poi calcolarne la produttività

- Transport mode detection: il rilevamento automatico della modalità di trasporto attraverso algoritmi di classificazione che operano sui dati di sensoristica prodotti da un dispositivo (e.g. smartphone) [3] [16]

- Anomaly detection: rilevamento di elementi in un dataset fuori dalla norma, che destino sospetti; praticato spesso per mezzo di algoritmi unsupervised, e particolarmente utile in un contesto di manutenzione predittiva (predirre un malfunzionamento di un impianto o macchinario) [11] [17]

- Structural health monitoring: il monitoraggio della salute strutturale si riferisce al processo di implementazione di una strategia di rilevazione e caratterizzazione dei danni per le strutture ingegneristiche [14]
- Healthcare: le applicazioni in questo ambito sono vastissime, dall'assegnazione automatica di un "Surveillance Level" ai pazienti in base alle loro cartelle cliniche [22] alla predizione delle malattie [5]
- Smart Cities: anche qui, si può spaziare dalla predizione della disponibilità dei parcheggi [30] alla gestione della rete elettrica della città (Smart Grid) [27]
- Supply chain management [21]
- etc.

Gli stream di dati IoT si presentano in genere sotto forma di timeseries, o serie temporali. Le timeseries sono sequenze ordinate di valori di misurazioni, ciascuna effettuata in un dato istante di tempo. Gli intervalli tra le misurazioni sono uniformi.

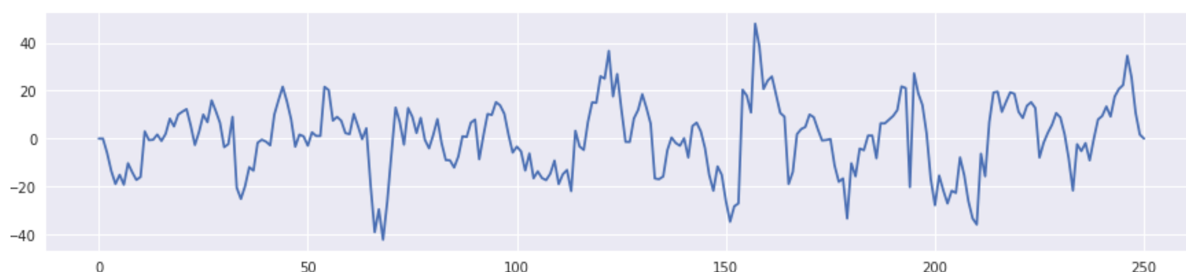



Figura 1.4: Struttura di una serie temporale. Sull'asse x la successione delle osservazioni, sulla y il loro valore.

Come già accennato nell'introduzione, tali stream di dati possono contenere o meno dei metadati, ovvero dei dati testuali che descrivano ogni stream (ad esempio il nome dato allo stream, il nome del sensore, l'unità di misura, etc).

In letteratura e tra gli esperti del settore sono diversi gli approcci proposti nella risoluzione del problema di annotazione di dati IoT: in questa tesi per comodità li abbiamo raggruppati in tre categorie, e di seguito li esamineremo, con un occhio di riguardo per

gli algoritmi specificamente pensati per serie temporali [1] (che dimostreremo non essere particolarmente perfomanti).

### 1.2.1 Algoritmi di tipo BOS

Gli algoritmi di tipo BOS o Bag of Summaries, presentati per la prima volta in [20], si basano sulla semplice ma efficace intuizione che le misurazioni provenienti da sensori ambientali siano facilmente classificabili quando si prende in considerazione statistiche come la media o la varianza: una serie temporale con media delle misurazioni di 1000 non potrà mai riferirsi alla temperatura, una con media 20 non potrà essere una misura della pressione atmosferica (in mbar).

Queste statistiche dunque, considerate il fattore discriminante da una classe all'altra, vengono calcolate e date in pasto come feature ad un qualsiasi classificatore di base (sezione 1.1.1) o ensemble (sezione 1.1.2).

I risultati ottenuti da questa categoria di algoritmi sono in generale molto buoni - a seconda del classificatore usato, ovviamente -, pur avendo una complessità computazionale molto bassa.

Questo approccio alla classificazione di timeseries verrà nei prossimi capitoli testato estensivamente, e farà poi parte dell'algoritmo progettato e implementato in questa tesi.

### 1.2.2 Algoritmi di tipo TSC

Algoritmi specificamente pensati per la classificazione di serie temporali. Per comodità li dividiamo in tre categorie:

- Algoritmi Whole-Series: Sono gli algoritmi che analizzano le serie temporali nella loro interezza, in genere sotto-tipologie di one-nearest-neighbor, come il 1NN-ED citato poco prima.

  Tale algoritmo è infatti uno dei più semplici in circolazione (nonchè non particolarmente performante), in quanto calcola la distanza - o la similarità - tra due timeseries sommando la distanza punto-punto di ciascuna serie.

Al contrario, uno degli algoritmi più performanti, e allo stesso tempo quello considerato dalla maggior parte dei ricercatori il gold standard in ambito di classificazione di timeseries, è il 1NN con misura di distanza Dynamic Time Warping (DTW) [2].

Il problema di questo approccio è tuttavia la complessità computazionale, che è più che quadratica - per la precisione $O(n^2m^2)$, con n numero di timeseries e m lunghezza di ciascuna timeseries -.

Negli anni sono state proposte diverse alternative per risolvere anche parzialmente tale problema di complessità, come il FastDTW [25].

- Algoritmi Shapelet-Based: Gli algoritmi basati sugli Shapelet, come suggerisce il nome - quasi un vezzeggiativo di 'shape', forma - puntano ad estrapolare da ogni serie temporale la sotto-sequenza, con lunghezza d « m, che meglio caratterizzi la serie; oppure, detto in altri termini, che riesca a discriminare una serie da un'altra meglio di qualsiasi altra sotto-sequenza [29]. Anche in questo caso il problema principale di questa classe di algoritmi è il costo computazionale relativo alla ricerca della migliore sotto-sequenza. In questa tesi faremo riferimento all'algoritmo presentato da Grabocka [12].

- Algoritmi Dictionary-Based: Questa famiglia di algoritmi ha in comune la filosofia di base di suddividere ciascuna timeseries in finestre temporali, estrarre pattern da ciascuna finestra ed usarli come features all'interno di un classificatore.

  Questo approccio è più veloce di quelli discussi in precedenza perchè riduce il numero di feature considerate, cioè ogni finestra temporale si trasforma in un numero di feature di ordini di grandezza inferiore rispetto alla numerosità dell misurazioni di cui è composta la finestra.

  I problemi in questo caso sono due: la scelta della suddivisione delle serie e la scelta di metriche "discriminatorie" che rappresentino il pattern.

  Famosi esempi di applicazione di questo concetto sono Bag of Pattern Features (BOPF), che trasforma ogni time-series in una sequenza di parole attraverso quello che si chiama SAX, symbolic aggregate approximation [18]; Bag of SFA-Symbols (BOSS), che codifica ciascuna sotto-sequenza attraverso la Trasformata Discreta

di Fourier (DFT) [26]; l'algoritmo presentato in [4] - che in questa tesi chiameremo Slope Distribution Encoding (SDE) -, che utilizza la pendenza di ciascuna sottosequenza come feature.

Anche questi approcci saranno estensivamente testati nel corso di questa tesi - almeno uno per categoria - reimplementando gli algoritmi quando necessario.

### 1.2.3 Algoritmi NLP

Per operare sui metadati (descrizioni, nomi, unità di misura, etc) dei datastream è necessario sviluppare degli algoritmi di Natural Language Processing (NLP). Un algoritmo di questo tipo prende i metadati come feature e li utilizza per addestrare un modello.

Una delle soluzioni presenti in letteratura, presentata per la prima volta in [19], è un k-nearest-neighbors che utilizza la edit-distance di Damerau-Levenshtein [6] per calcolare la distanza tra le parole.

Una edit-distance misura la diversità tra due parole come il minor numero di operazioni necessarie a trasformare una parola nell'altra. Ve ne sono diversi tipi: nel caso in esame testeremo la distanza di Damerau-Levenshtein, quella di Levenshtein, quella di Jaccard e quella di Jaro-Winkler.

Questo approccio alla classificazione, di per sè abbastanza performante e poco computazionalmente costoso, è particolarmente efficace quando combinato con altri approcci all'interno di un classificatore ensemble.

# Capitolo 2

# Analisi preliminari

Il lavoro svolto in questa tesi copre quella che forse è una delle aree più trascurate della classificazione di dati IoT: gli Open Data. Gli Open Data sono dati liberamente accessibili a tutti, e spesso sono collezionati e distribuiti dagli stessi utenti della rete.

A causa di questa loro natura tali dati sono fortemente eterogenei, e per tale motivo molti degli algoritmi pensati appositamente per serie temporali (anche IoT) performano male su di essi. Inoltre, anche in letteratura, sono poche le proposte di algoritmi ensemble in tale ambito, e praticamente nessuna che offra una fase di apprendimento che analizzi più aspetti del dataset in esame (quello delle statistiche, quello dei metadati, quello della successione delle misurazioni).

Ciò che presentiamo in questa tesi è dunque un algoritmo ensemble completo - analizza i dati sotto tutti gli aspetti elencati -, modulare - tanto che un'idea di sviluppo futuro è di estenderlo in framework - e altamente performante su dati open IoT.

Tuttavia, prima di fare ciò riteniamo necessario fornire al lettore una "baseline" di risultati da confrontare poi con quelli ottenuti dal framework algoritmico esposto nel prossimo capitolo.

Dedicheremo dunque questo capitolo alla discussione dell'implementazione e della validazione degli algoritmi presenti in letteratura e citati nella sezione 1.2. Le prime tre sezioni introdurranno gli strumenti, i dataset e le metriche utilizzate per valutare le performance dei modelli, le altre discuteranno ognuna una famiglia di algoritmi, nel seguente ordine:

- BOS - Bag of Summaries
- TSC - Time Series Classification
- NLP - Natural Language Processing
- Ensemble BOS - Tecniche ensemble applicate ad algoritmi Bag of Summaries

Ogni sezione avrà una struttura simile: nella prima parte spiegheremo - in molti casi dettagliatamente - la logica e il funzionamento degli algoritmi, spesso dilungandoci su temi particolarmente rilevanti per quella famiglia di classificatori (ad esempio l'ottimizzazione dei parametri per gli algoritmi BOS, o la normalizzazione per i TSC); nella seconda tratteremo la loro implementazione, mostrando il codice relativo ai punti chiave di questa; nella terza esporremo i risultati attraverso opportuni grafici e li commenteremo.

## 2.1 Strumenti

L'implementazione degli algoritmi esposti in questa tesi è stata effettuata con linguaggio Python. Python è un linguaggio di programmazione ad alto livello particolarmente adatto per analisi dati e machine learning, in larga parte grazie alle innumerevoli librerie di cui è provvisto: molti delle funzioni usate, come vedremo anche in seguito, sono importate ad esempio dalla libreria scikit-learn (https://scikit-learn.org). Ciò ci consente, tra le altre cose, di non dover implementare manualmente i modelli alla base degli algoritmi BOS, poichè li possiamo importare direttamente dalla libreria:

```
1   from sklearn import DecisionTreeClassifier
```

Listing 2.1: Esempio di import da sklearn

Anche i grafici sono stati prodotti con linguaggio Python, nello specifico con la libreria plotly (https://plot.ly/python).

Citeremo eventuali altre librerie utilizzate nelle rispettive implementazioni.

## 2.2 Dataset

Tutti i test sono stati effettuati su tre dataset, molto simili tra loro in termini di dati - tutti e tre infatti contengono misurazioni di sensoristica ambientale -, ma diversi in termini di granularità spaziale: sono infatti rispettivamente dati collezionati da una nazione, da una regione e da una città. I tre dataset si differenziano dunque per eterogeneità dei dati contenuti, e di conseguenza per difficoltà di classificazione (poi comprovata dai test).

- Thingspeak: un dataset estratto dall'omonima piattaforma (thingspeak.com), sulla quale gli utenti possono pubblicare i propri dati di sensoristica. I dati, che si presentano come un insieme di datastream, risiedono all'interno di un channel. Ogni datastream è associato ad un insieme di metadati, inseriti in input dall'utente. Il dataset finale contiene dati provenienti dall'area geografica evidenziata nella figura 2.1, per un totale di 2121 datastream (serie temporali) appartenenti a 21 classi. Per una spiegazione più dettagliata del metodo di estrazione e processamento dei dati si rimanda a [7]

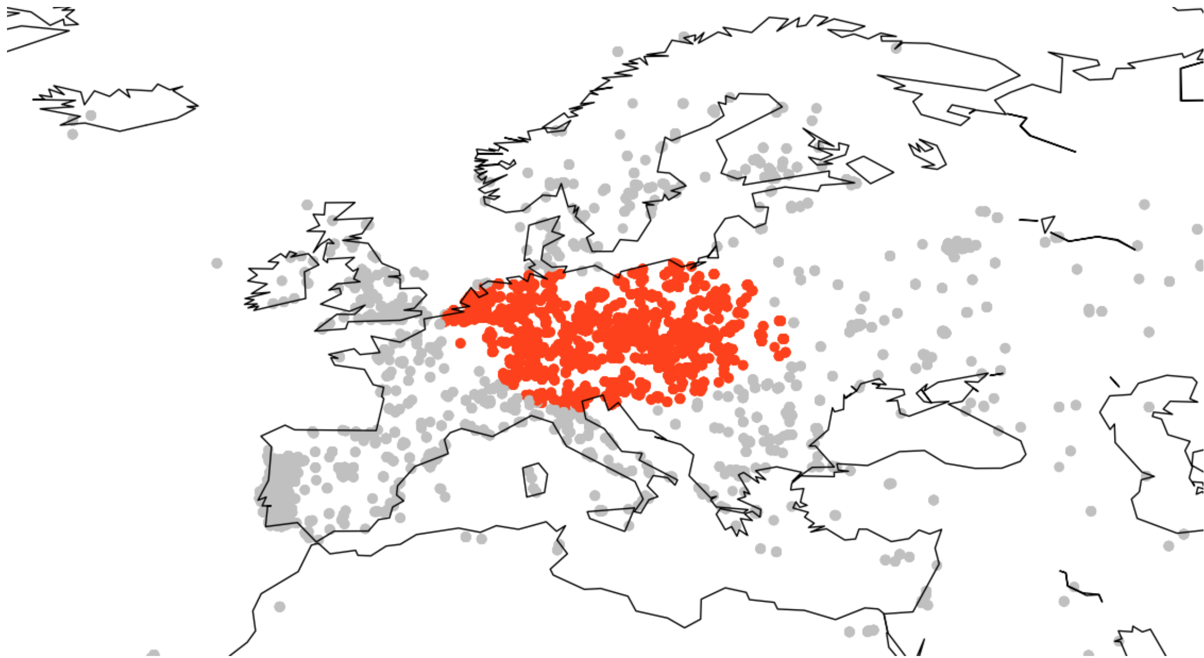


Figura 2.1: Fonte: [7]

- Swissex: Swiss Experiment (abbreviato Swissex) è una piattaforma online che permette la pubblicazione di misurazioni effettuate nell'area delle Alpi Svizzere. Il dataset estratto da questa piattaforma è notoriamente usato come banco di prova per algoritmi di classificazione [4]. Esso contiene 346 stream ciascuno con 445 misurazioni, appartenenti a 11 classi.
- UrbanObservatory: nato da un progetto dell'Università di Newcastle che consiste nello sviluppo di una rete di sensori sparsi per la città. Tale dataset conta di 1065 stream con 864 misurazioni ciascuno, appartenenti a 16 classi.

Tutti e tre i dataset si presentano nella stessa struttura: file csv senza header in cui le prime 8 colonne sono occupate da metadati, la nona da un intero che rappresenta la classe (l'etichetta) del datastream e dalla decima in poi i valori delle misurazioni.

Come possiamo vedere, tutti i dataset contano di "pochi" dati, fattore che aggiunge un grado di difficoltà al nostro lavoro di classificazione: anche per questo motivo, ad esempio, non useremo nè citeremo modelli che impiegano reti neurali, efficaci solo per quantità di dati di ordini di grandezza superiori.

## 2.3 Misurazione delle performance

Introduciamo in questa sezione quelle che sono le metriche analizzate per la valutazione della performance:

- Accuracy: l'accuratezza di un classificatore è il rapporto tra le osservazioni etichettate correttamente e le osservazioni totali,

  $$acc = \frac{TP + TN}{TP + TN + FP + FN}$$

  dove con TP e TN intendiamo le osservazioni positive (true positives) e negative (true negatives) predette correttamente, mentre con FP e FN quelle predette non correttamente (false positives e false negatives)
- Precision: la precisione di un classificatore misura il rapporto tra le osservazioni positive predette correttamente e il totale delle osservazioni positive

  $$precision = \frac{TP}{TP + FP}$$

- Recall: il richiamo misura il rapporto tra i veri positivi e tutte le osservazioni che sarebbero dovute essere positive

$$precision = \frac{TP}{TP + FN}$$

- f1-score: media armonica di precision e recall, è una buona metrica "riassuntiva" delle due sopracitate

$$f1 = \frac{2 * (recall * precision)}{recall + precision}$$

L'obiettivo della fase di testing di ogni modello è dunque quello di ottenere metriche - prendiamo in considerazione solo *accuracy* e $f1$, per comodità - più alte rispetto agli altri.

## 2.4 Algoritmi BOS

Come accennato nel paragrafo 1.2.1, la ratio dietro l'utilizzo di questa famiglia di algoritmi è che un set di feature "statistiche" è ben performante nel separare le classi in base all'information gain.

Le feature scelte nel nostro caso sono 11: media, mediana, deviazione standard, varianza, valore minimo, valore massimo, simmetria, curtosi, range dei valori, range interquartile, root mean square (radice della media dei valori elevati al quadrato).

I dataset che saranno dati in pasto agli algoritmi conterrano dunque in una colonna la classe reale delle misurazioni, nelle altre 11 colonne le varie feature statistiche.

Viene poi fatto quello che nel gergo del machine learning viene chiamato "split" del dataset in training e test set: come suggerisce il nome, il dataset viene suddiviso - randomicamente - in due parti, di norma con proporzione 70-30; il training set viene quindi usato per addestrare il modello di machine learning, mentre il test set per valutarne la performance.

```
1    X_train , X_test , y_train , y_test = train_test_split(valori_features
   , classi , test_size = 0.3 , random_state = 100)
```

Listing 2.2: Split del dataset

La funzione *train_test_split* è importata dalla libreria scikit-learn di Python. *X_train* è l'insieme di feature del training set, *X_test* quello del test, *y_train* le classi del training set, *y_test* quelle del test set.

Gli algoritmi di classificazione selezionati sono i seguenti: Decision Tree, SVM, kNN, Logistic Regression, Ridge e Naive Bayes, tutti importati da scikit-learn.

### 2.4.1 Ottimizzazione dei parametri

Il problema che si pone a questo stadio del processo è la scelta dei parametri per ogni algoritmo: per kNN infatti, per esempio, va scelto il valore di k, oppure per Decision Tree si può indicare la profondità massima dell'albero. La soluzione più banale, ma non per questo non intelligente, è di provarli tutti: sklearn infatti mette a disposizione una funzione chiamata *GridSearch*, che testa tutte le combinazioni dei parametri dati in input.

Il GridSearch opera una Cross-Validation (CV), una tecnica statistica che consiste nel dividere il training set in k parti e ad ogni passo considerare la k-esima parte come test set. La CV serve sostanzialmente per evitare problemi di overfitting, quando cioè il modello di classificazione piuttosto che imparare a "generalizzare" i pattern tra i dati impara le particolarità del training set, performando poi male su dati mai visti.

```python
# trova i parametri ottimali di un classificatore

def optimizeParams(clf, params, X_train, y_train):
    clf_gs = GridSearchCV(clf, params, cv=5)
    clf_gs.fit(X_train, y_train)
    return clf_gs.best_estimator_
```

Listing 2.3: Funzione per ottimizzare i parametri di un classificatore

Vengono dunque generati due array, uno con i riferimenti ai classificatori standard e uno con i riferimenti ai classificatori con parametri ottimizzati. Tutti i classificatori vengono poi addestrati sul training set vero e proprio (*X_train* e *y_train*), e vengono testati stimando la classe degli elementi di *X_test*. Più la lista prodotta (che chiameremo *y_pred*) si avvicinerà a *y_test* più la performance sarà buona.

### 2.4.2 Risultati

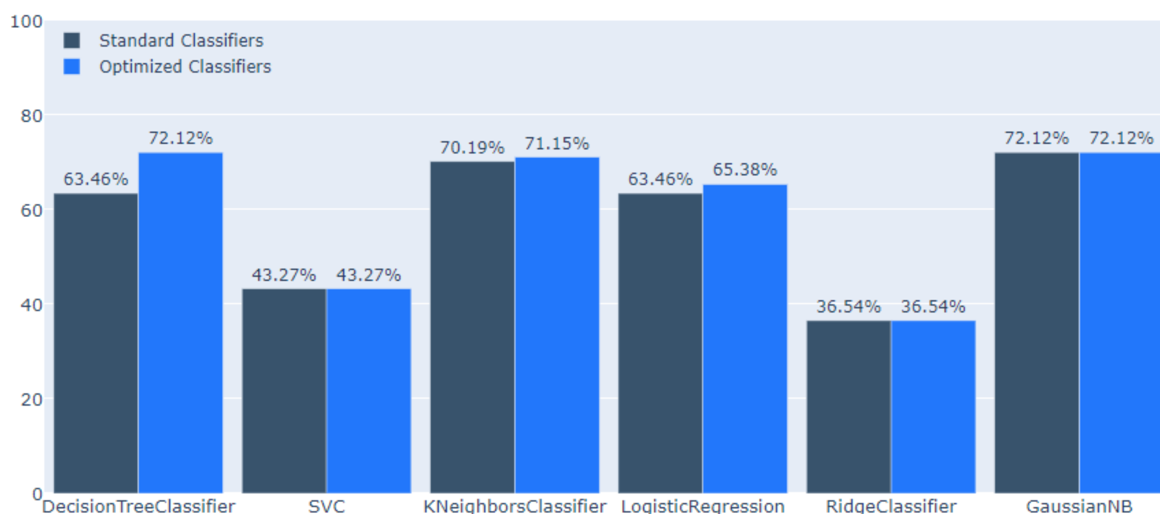



Figura 2.2: Accuracy a confronto tra standard e optimized classifiers su Swissex

Come possiamo vedere ottimizzando i parametri le performance rimangono stabili o migliorano. Pertanto d'ora in poi faremo riferimento ai classificatori ottimizzati, anche se non scritto esplicitamente. I valori raggiunti sono abbastanza dignitosi per questi classificatori relativamente semplici.

I risultati confermano tra l'altro quanto già ipotizzato in fase di descrizione dei dataset: su UrbanObservatory, il dataset più omogeneo, si ottengono risultati più alti, mentre su Thingspeak, quello più eterogeneo, i risultati sono significativamente più bassi.

## 2.5 Algoritmi TSC

Appartengono a questa categoria tutti gli algoritmi esposti nel paragrafo 1.2.2. Operiamo dunque una selezione tra quelli, selezionando un paio di esponenti per categoria (Whole-Series, Shapelet-Based e Dictionary-Based).

Testeremo dunque i seguenti: 1NN-ED, 1NN-DTW, Shapelets, BOPF, SDE.

Tali modelli di classificazione non necessitano di particolari trasformazioni preliminarie delle features, prendono perciò in input i datastream e le relative classi.

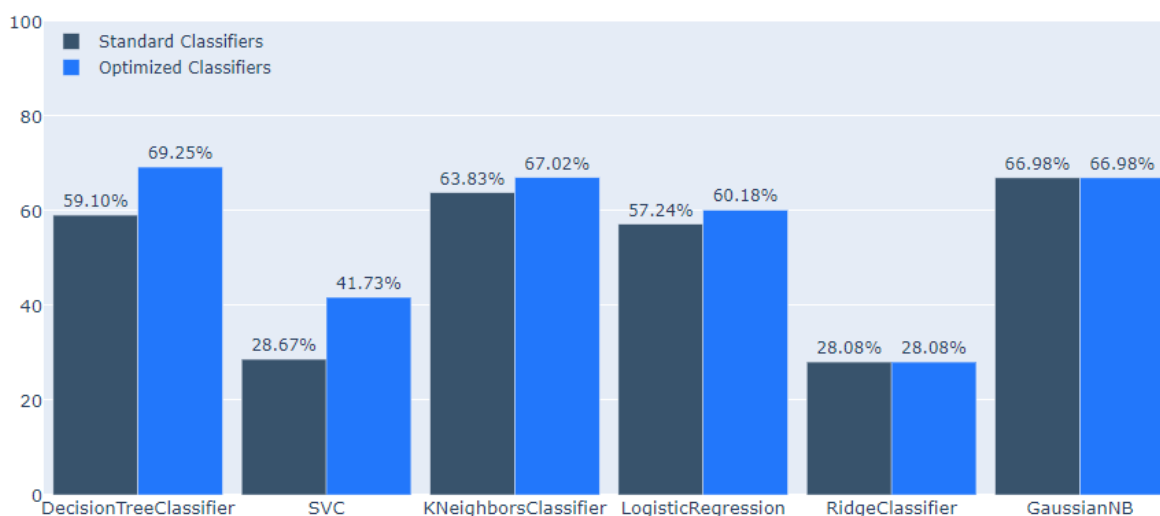



Figura 2.3: f1-score a confronto tra standard e optimized classifiers su Swissex

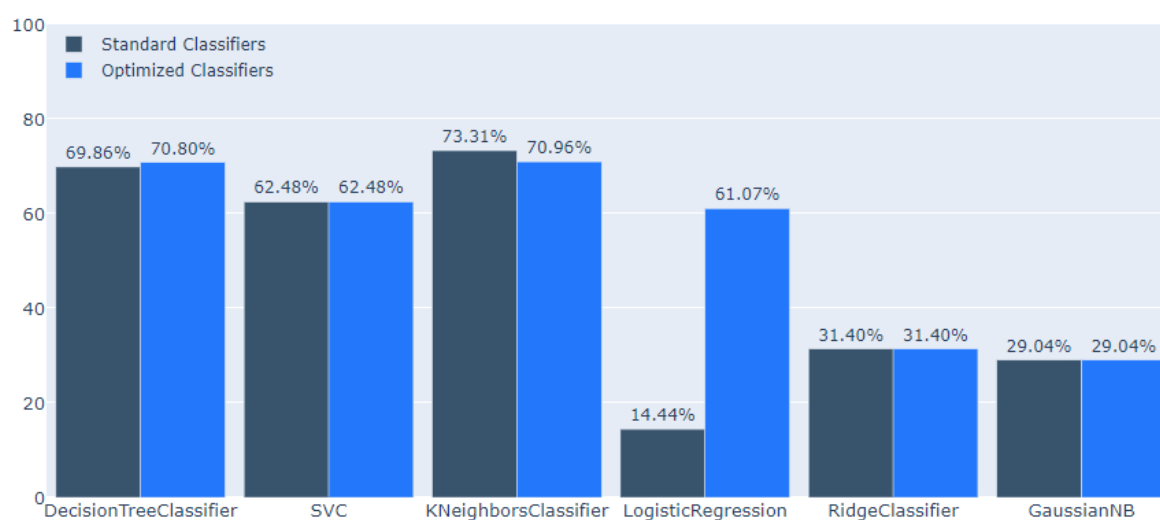



Figura 2.4: Accuracy a confronto tra standard e optimized classifiers su Thingspeak

### 2.5.1 Normalizzazione

Prima di parlare dell'implementazione dei vari algoritmi è necessario soffermarci sul tema della normalizzazione dei dati: è necessaria o meno? Normalizzare i dati significa ridimensionarli in un determinato intervallo, in modo da evitare che alcuni di questi "pesino" di più in fase di predizione. La normalizzazione più comune è la Z-Normalizzazione,

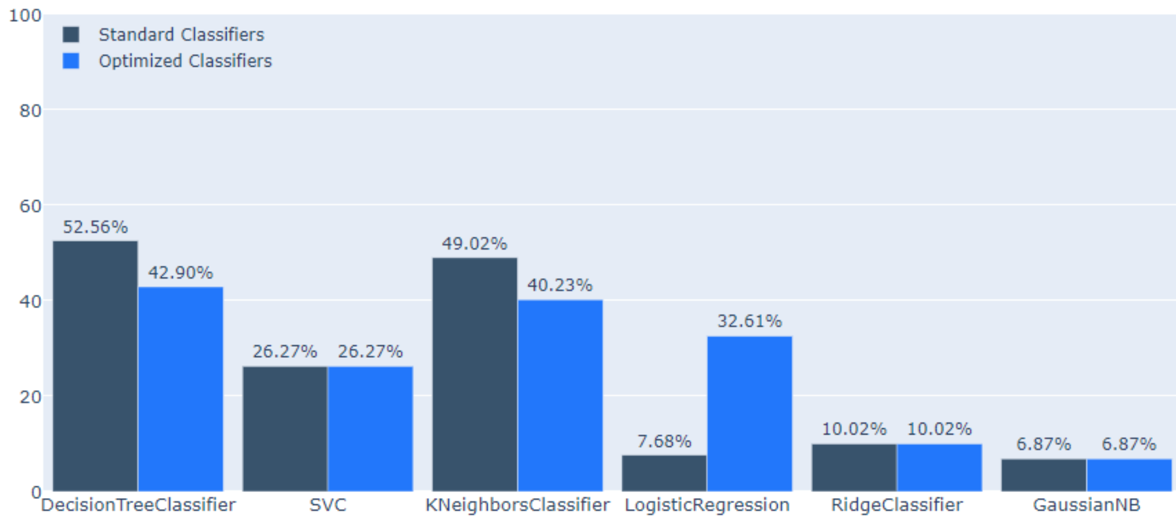



Figura 2.5: f1-score a confronto tra standard e optimized classifiers su Thingspeak

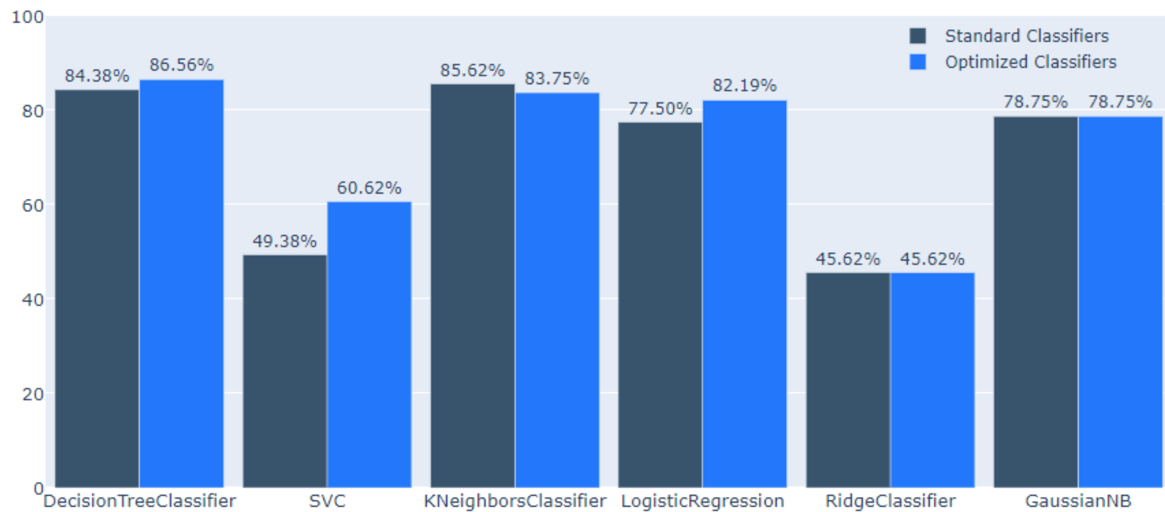



Figura 2.6: Accuracy a confronto tra standard e optimized classifiers su UrbanObservatory

ovvero il ridensionamento degli attributi in modo che abbiano media 0 e deviazione standard 1. In ambito BOS, dove gli algoritmi si basano sulla differenza delle statistiche delle varie features, normalizzare i dati è controproducente, ma con i modelli di classificazione TSC la normalizzazione potrebbe migliorare la performance anche significativa-

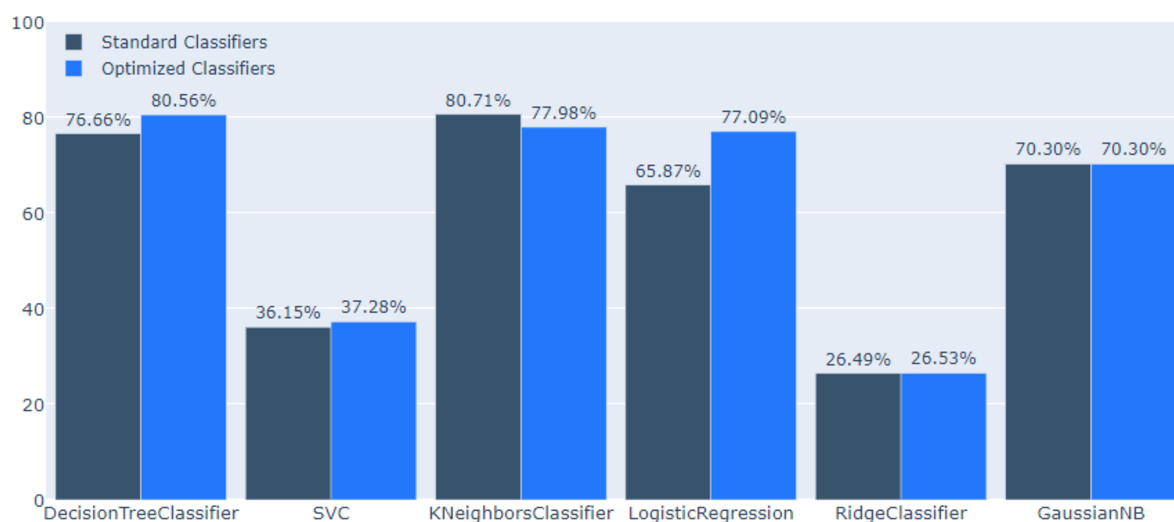



Figura 2.7: f1-score a confronto tra standard e optimized classifiers su UrbanObservatory

mente. Per implementare la normalizzazione dei dati su Python ci affideremo alle classi *RobustScaler*, *Normalizer*, *MinMaxScaler*, *MaxAbsScaler*, *StandardScaler* della libreria sklearn: tutti i risultati ottenuti sono quelli che usano la tecnica di normalizzazione più performante in quello specifico caso.

### 2.5.2 1NN-ED

Una soluzione semplice ed immediata al problema della classificazione delle timeseries: la distanza tra una serie e un'altra è la distanza euclidea (somma delle distanze punto-punto) tra di esse, la classe di un'osservazione nel test set è la stessa della serie più vicina all'interno del training set.

Per implementare tale algoritmo ci affidiamo al modello *KNeighborsClassifier* di sklearn, che ci permette di specificare il parametro *metric*, e accetta come valore *'euclidean'*, risparmiandoci la scrittura di una funzione apposita (che sarebbe comunque facile).

```
knn = KNeighborsClassifier(n_neighbors = 1, metric = 'euclidean')
knn.fit(X_train, y_train)
metrics_standard.append(metrics(knn, X_test, y_test))
```

```
4
5    knn.fit(X_train_norm, y_train)
6    metrics_norm.append(metrics(knn, X_test_norm, y_test))
```

Listing 2.4: 1NN-ED

### 2.5.3 1NN-DTW

Un aggiustamento al calcolo della distanza, che adesso trova l'allineamento ottimale tra le due serie prima di restituire tale metrica.

In generale, DTW è un metodo che permette di trovare una corrispondenza ottima tra due sequenze, attraverso una distorsione non lineare rispetto alla variabile indipendente (tipicamente il tempo). Alcune restrizioni per il calcolo della corrispondenza sono generalmente utilizzate: deve essere garantita la monotonicità nelle corrispondenze, ed il limite massimo di possibili corrispondenze tra elementi contigui della sequenza.

Viene generalmente usato in campi come il riconoscimento vocale (per adattarsi a diverse velocità di "parlata"), riconscimento della firma e riconoscimento delle forme.

Anche qui per l'implementazione ci affidiamo a $KNeighborsClassifier$, attribuendo al parametro *metric* una funziona da noi dichiarata:

```
1  def DTWDistance(s1, s2, w = None):
2      rows = len(s1) + 1
3      cols = len(s2) + 1
4      DTW = np.zeros((rows, cols))
5
6      if w:
7          w = max(w, abs(len(s1)-len(s2)))
8
9          for i in range(0, rows):
10             for j in range(0, cols):
11                 DTW[i, j] = float('inf')
12
13         DTW[0, 0] = 0
14
15         for i in range(1, rows):
```

```
16            for j in range(max(1, i-w), min(cols, i+w+1)):
17                DTW[i, j] = 0
18
19        distance = 0
20
21        for i in range(1, rows):
22            for j in range(max(1, i-w), min(cols, i+w+1)):
23                distance = (s1[i-1] - s2[j-1]) ** 2
24                DTW[i,j] = distance + min(DTW[i-1,j], DTW[i-1,j-1],
25                    DTW[i, j-1])
26    return DTW[len(s1), len(s2)]
```

Listing 2.5: Misura distanza DTW. Il parametro w indica la finestra di distorsione massima.

### Shapelets

Come già accennato nello scorso capitolo, utilizzeremo l'algoritmo shapelets di [12] nel corso di questo test. Tale versione dell'algoritmo si differenzia dagli altri della stessa famiglia poichè adotta un approccio nuovo, computazionalmente meno costoso (anche se comunque più che quadratico), per l'apprendimento delle shapelet ottimali.

La motivazione è abbastanza semplice: tale algoritmo viene fornito dalla libreria *tslearn* di sklearn, risparmiandoci la sua implementazione (non semplice, data la sua complessità). Pertanto, il lavoro da fare si riduce a poche righe di codice:

```
1 # per usare il dataset nella libreria tslearn dobbiamo convertirlo con
    la seguente funzione
2
3 X_train_tslearn = to_time_series_dataset(X_train)
4 X_test_tslearn = to_time_series_dataset(X_test)
5
6 X_train_tslearn_norm = to_time_series_dataset(X_train_norm)
7 X_test_tslearn_norm = to_time_series_dataset(X_test_norm)
8
9 # calcolo shapelet_sizes
10
```

```
11 shapelet_sizes = grabocka_params_to_shapelet_size_dict(n_ts=len(
       valori_ts), ts_sz=len(valori_ts[0]), n_classes=len(set(classi)), l
       =0.1, r=2)
12
13 # shapelet-based classifier
14
15 shp_clf = ShapeletModel(n_shapelets_per_size=shapelet_sizes,
       verbose_level = 0, max_iter = 5000)
16 shp_clf.fit(np.array(X_train_tslearn), np.array(y_train))
17 metrics_standard.append(metrics(shp_clf, np.array(X_test_tslearn), np.
       array(y_test)))
18
19 shp_clf.fit(np.array(X_train_tslearn_norm), np.array(y_train))
20 metrics_norm.append(metrics(shp_clf, np.array(X_test_tslearn_norm), np.
       array(y_test)))
```

Listing 2.6: Shapelets

### 2.5.4 BOPF

Anche BOPF, come gli algoritmi basati su shapelets, cerca di trovare dei ‘pattern’ all’interno delle time-series in esame, ma in modo molto diverso. BOPF trasforma ogni time-series in una sequenza di parole attraverso quello che si chiama SAX, symbolic aggregate approximation; dopodichè analizza quali di queste parole ‘discriminano’ meglio tra una classe e le altre, attraverso il calcolo dell’ANOVA f-value: le ‘best performer’ (e i loro relativi conteggi, all’interno di ogni classe) serviranno per calcolare per ciascuna classe un centroide. Nella fase di classificazione poi la serie di test verrà trasformata anch’essa in parole e verrà calcolata la sua distanza da ogni centroide, per definire la classe assegnata. Per ottenere i risultati di cui sotto, il codice di BOPF fornito in C++ dai creatori dell’algoritmo [18] è stato modificato per farlo funzionare sui dataset in esame.

### 2.5.5 SDE

SDE approssima le time-series dividendole in ‘buckets’ (sotto-sequenze) e calcolando la pendenza media di ogni bucket di punti in modo da minimizzare l’errore medio. Tali pendenze poi vengono trasformate in caratteri in base all’angolo formato, e il conteggio dei caratteri all’interno di ogni serie diventa una feature su cui poi addestrare un classificatore (nel caso testato 1NN). Sostanzialmente è un algoritmo che considera come discriminante tra una serie e le altre la velocità di variazione delle misurazioni nel tempo. Il codice fornito dall’autore [4] era in Scala, ma dati i problemi nel farlo girare e la poca dimestichezza col linguaggio è stato necessario re-implementarlo in Python.

```
1  slopes = []
2
3  # per ogni timeseries all'interno del dataset
4  for ts in X:
5      S = []
6      # per ogni misurazione all'interno della timeseries
7      for i in range(len(ts)):
8          # aggiorna i buckets
9          S = buckets(S, ts, bucket_size, i)
10     # salva array finale di buckets
11     slopes.append(S)
12
13 ds = []
14
15 # trasforma ogni array di buckets in parole
16 for S in slopes:
17     ds.append(symbolization(S))
18
19 # creiamo il dataframe e splittiamo
20 data = pd.DataFrame(ds)
21 X_train, X_test, y_train, y_test = train_test_split(data, classi,
       test_size = 0.3, random_state = 100)
22
23 knn = KNeighborsClassifier(n_neighbors = 1)
24 knn.fit(X_train, y_train)
```

```
25 y_pred = knn.predict(X_test)
```

Listing 2.7: SDE

### 2.5.6 Risultati

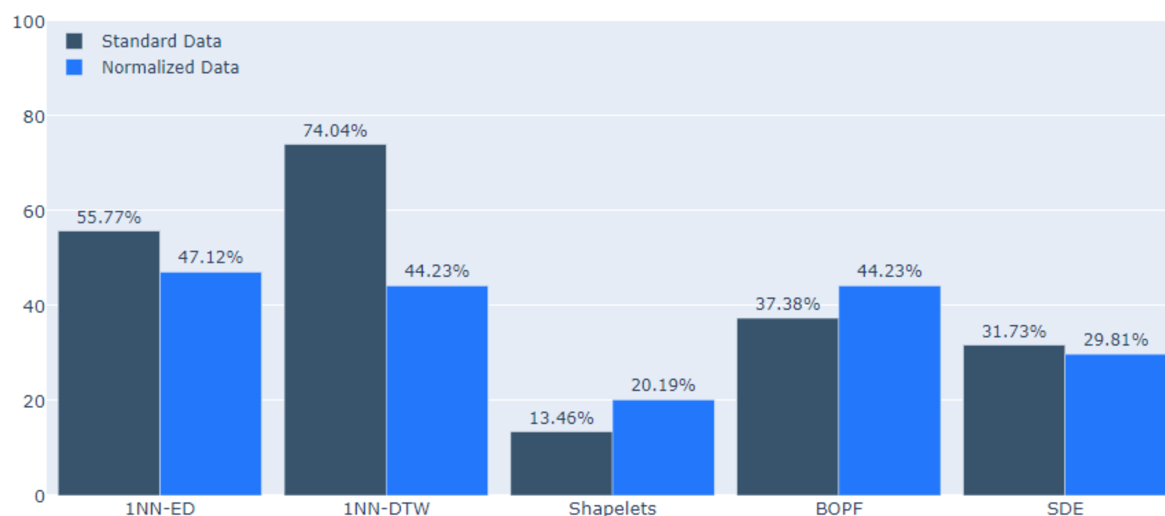



Figura 2.8: Accuracy a confronto tra dati standard e normalizzati su Swissex

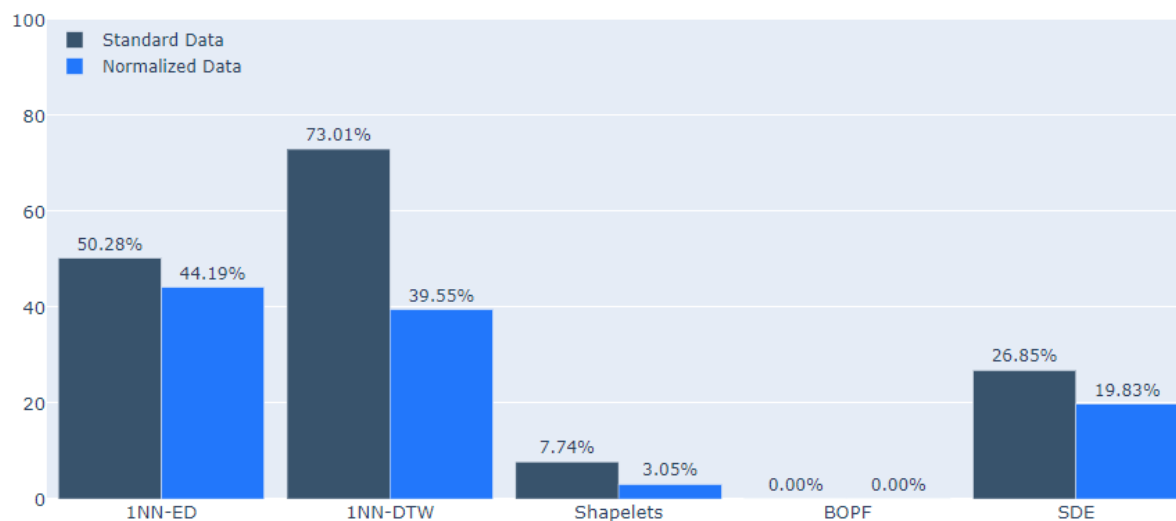



Figura 2.9: f1-score a confronto tra dati standard e normalizzati su Swissex

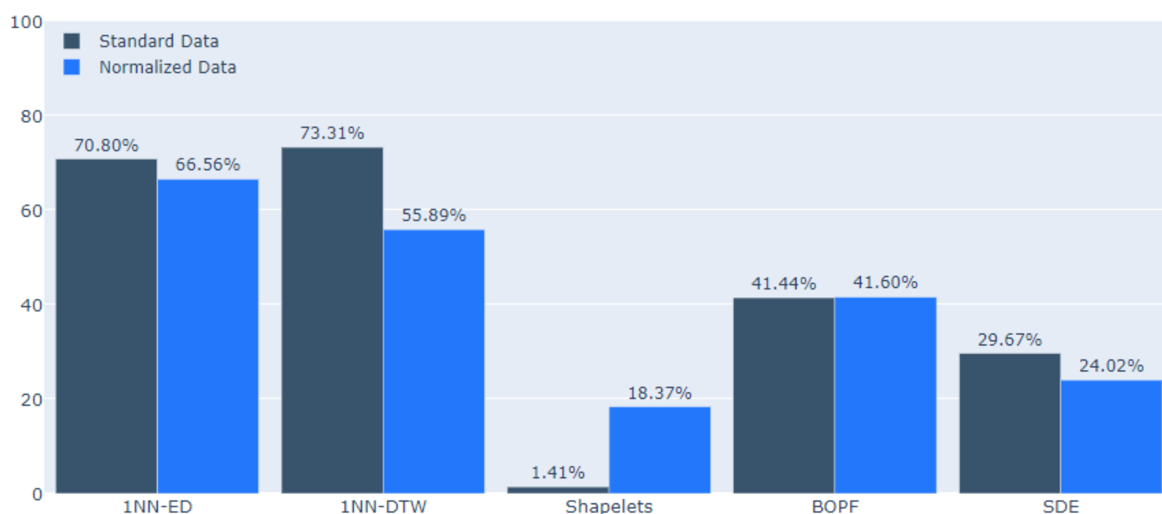



Figura 2.10: Accuracy a confronto tra dati standard e normalizzati su Thingspeak

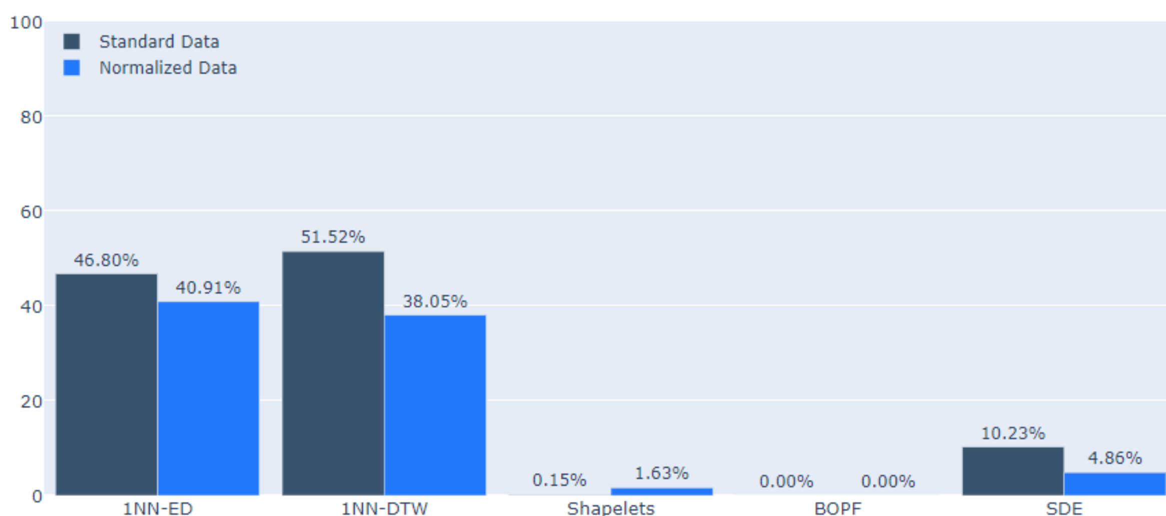



Figura 2.11: f1-score a confronto tra dati standard e normalizzati su Thingspeak

Come possiamo vedere dai grafici dei classificatori, ad eccezione di 1NN-DTW, che si riconferma il gold standard, non sono buone. Ciò probabilmente avviene a causa dell'eterogeneità dei dati, per cui gli algoritmi non riescono a trovare dei pattern distintivi all'interno delle serie temporali. Su UrbanObservatory infatti, il dataset più omogeneo, le performance migliorano di molto, toccando il 90% di accuracy.

Tuttavia, anche a parità di performance con gli algoritmi BOS, questi modelli di

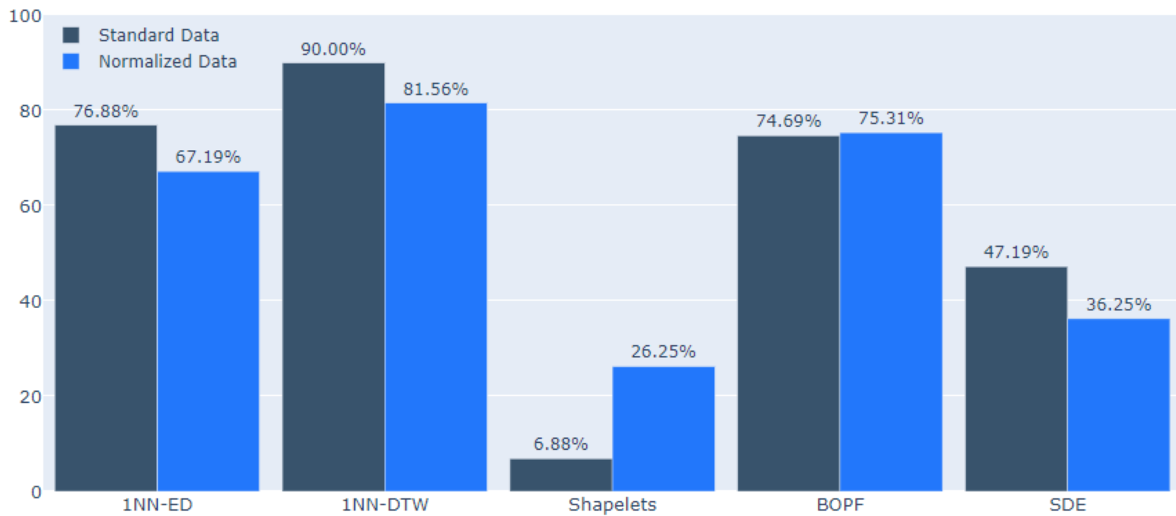



Figura 2.12: Accuracy a confronto tra dati standard e normalizzati su UrbanObservatory

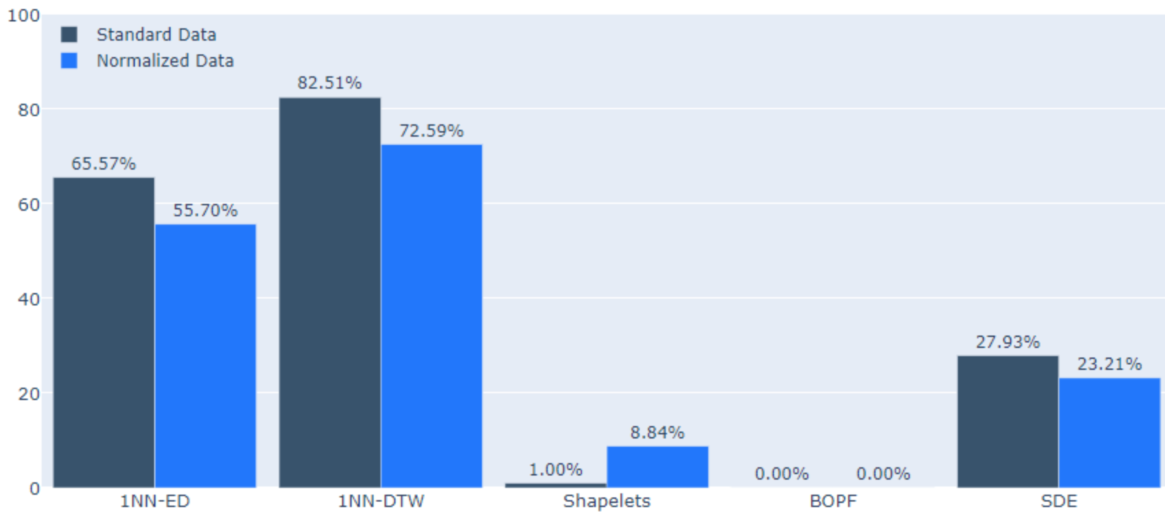



Figura 2.13: f1-score a confronto tra dati standard e normalizzati su UrbanObservatory

classificazione soffrono del problema della complessità computazionale: sono necessari 13655 secondi (quasi 4 ore) per eseguire 1NN-DTW sulla mia macchina, contro i 2 secondi di un qualsiasi modello BOS.

## 2.6 Algoritmi NLP

Come già accennato, gli algoritmi NLP operano su dati testuali, ovvero i metadati associati dagli utenti agli stream da loro pubblicati.

Nel caso dei nostri dataset, solo Thingspeak è dotato di metadati, quindi eseguiremo test solo su questo. Inoltre, ci limiteremo ad analizzare la colonna "name" degli stream, per semplicità e per comodità.

Per l'implementazione delle misure di distanza ci affidiamo alle librerie *distance*, *pyxdameraulevenshtein* e *nltk*:

```
1  from distance import jaccard
2  from pyxdameraulevenshtein import damerau_levenshtein_distance
3  from nltk.metrics.distance import edit_distance,
       jaro_winkler_similarity
4
5  # semplice 1NN classifier - inverse e' True se la distance_metric e'
       una similarity metric
6
7  def predict_labels(X_train, y_train, X_test, distance_metric, inverse =
        False):
8      y_predict = []
9
10     for i in range(len(X_test)):
11         distance = float('inf')
12         index = 0
13         for j in range(len(X_train)):
14             if (inverse):
15                 temp = 1 - distance_metric(X_test[i], X_train[j])
16             else:
17                 temp = distance_metric(X_test[i], X_train[j])
18             if temp < distance:
19                 distance = temp
20                 index = j
21         y_predict.append(y_train[index])
22     return y_predict
23
```

```
24 # damerau-levenshtein edit-distance
25 y_pred = predict_labels(X_train, y_train, X_test,
       damerau_levenshtein_distance)
26 distance_metrics.append(metrics(y_pred, y_test))
27
28 # levenshtein edit-distance
29 y_pred = predict_labels(X_train, y_train, X_test, edit_distance)
30 distance_metrics.append(metrics(y_pred, y_test))
31
32 # jaccard edit-distance
33 y_pred = predict_labels(X_train, y_train, X_test, jaccard)
34 distance_metrics.append(metrics(y_pred, y_test))
35
36 # jaro-winkler distance
37 y_pred = predict_labels(X_train, y_train, X_test,
       jaro_winkler_similarity, True)
38 distance_metrics.append(metrics(y_pred, y_test))
```

Listing 2.8: 1NN-NLP

### 2.6.1 Risultati

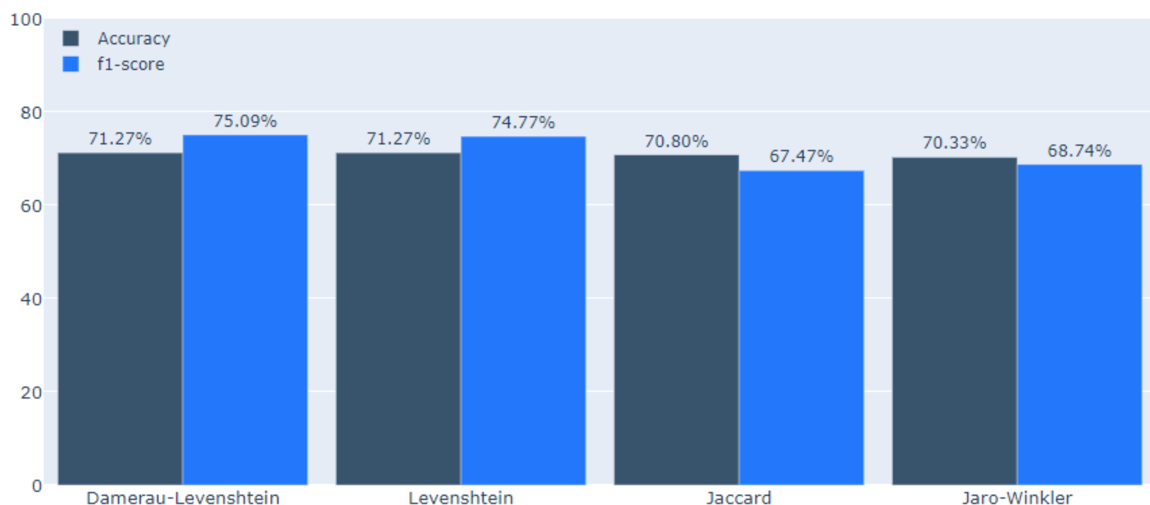



Figura 2.14: Accuracy e f1-score a confronto con diverse edit-distances su Thingspeak

I risultati sono in generale promettenti, e non si discostano molto tra una misura e l'altra: d'ora in poi prenderemo in considerazione solo la distanza di Damerau-Levenshtein.

## 2.7 Ensemble BOS

Poichè gli algoritmi di tipo BOS sono particolarmente veloci e ben performanti, nulla ci vieta di organizzarli in ensembles per tentare di migliorarne le metriche. Gli ensemble, come già accennato in 1.1.2, si basano sul principio "wisdom of the crowd" ("saggezza della folla", nozione esposta nel lontano 1903 [10]), ovvero che la "media" delle opinioni o delle stime di più individui è più accurata rispetto a quella del singolo.

### 2.7.1 Bagging

L'implementazione di un ensemble di tipo bagging non è particolarmente complessa, grazie alle apposite librerie di sklearn. Basta infatti importare la libreria *BaggingClassifier* e instanziare il modello passandoli come parametro il classificatore di base, per poi addestrarlo e testarlo come un qualasiasi altro algoritmo.

```
1  from sklearn.ensemble import BaggingClassifier
2  # creiamo un bagging ensemble per ogni tipologia di classificatore (
     prendiamo come base la variante ottimizzata)
3  
4  bagging_ensembles = []
5  metrics_bagging = []
6  
7  for clf in clfs_best:
8      ensemble = BaggingClassifier(base_estimator = clf, n_estimators =
     100, random_state = 100)
9      bagging_ensembles.append(ensemble)
10     ensemble.fit(X_train, y_train)
11     metrics_bagging.append(metrics(ensemble, X_test, y_test))
```

Listing 2.9: Bagging

I risultati migliorano di qualche punto percentuale quelli dei classificatori ottimizzati "singoli":

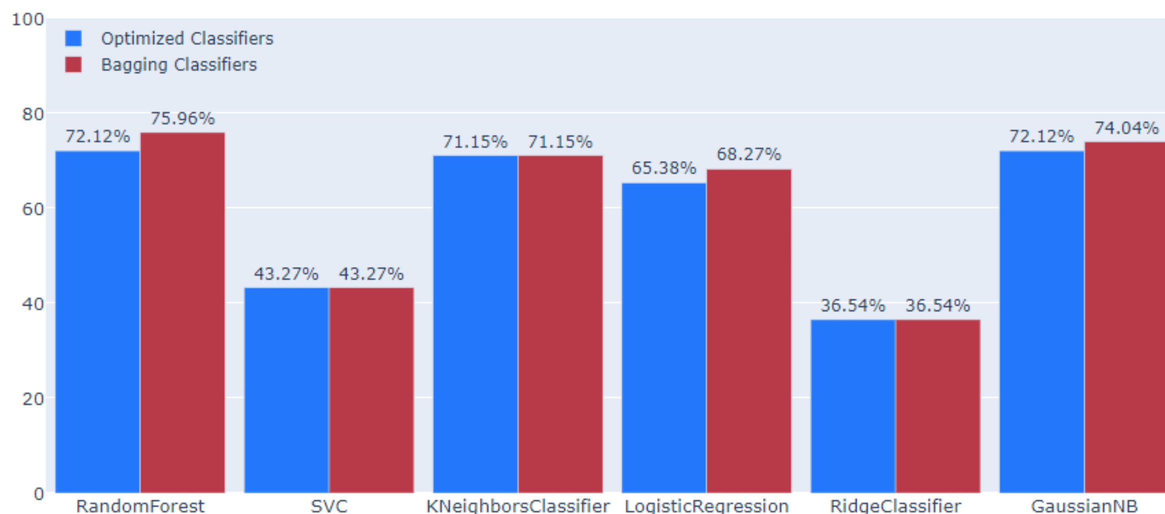



Figura 2.15: Accuracy a confronto tra singoli classifiers ed ensemble bagging su Swissex

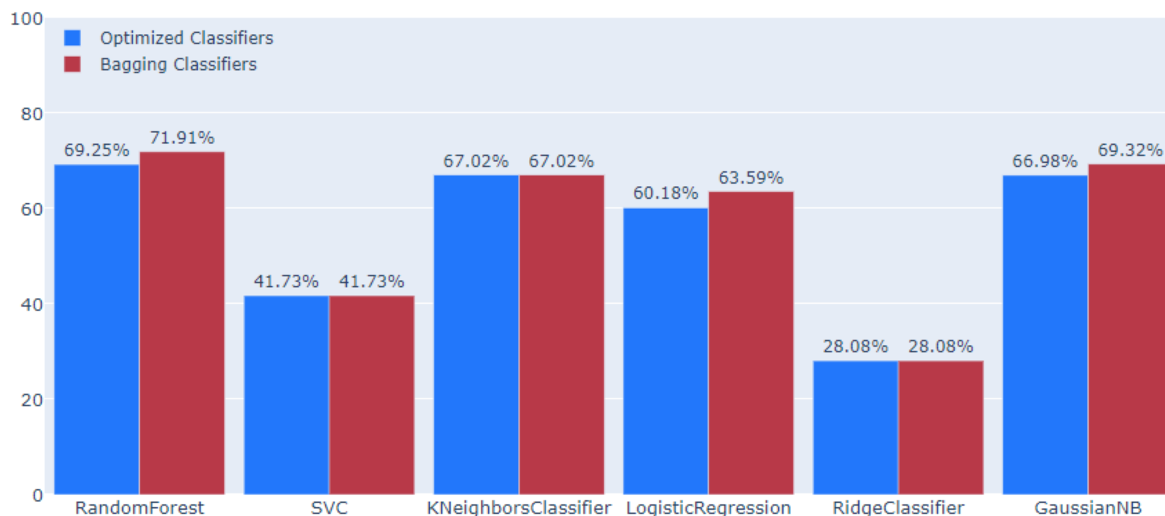



Figura 2.16: f1-score a confronto tra singoli classifiers ed ensemble bagging su Swissex

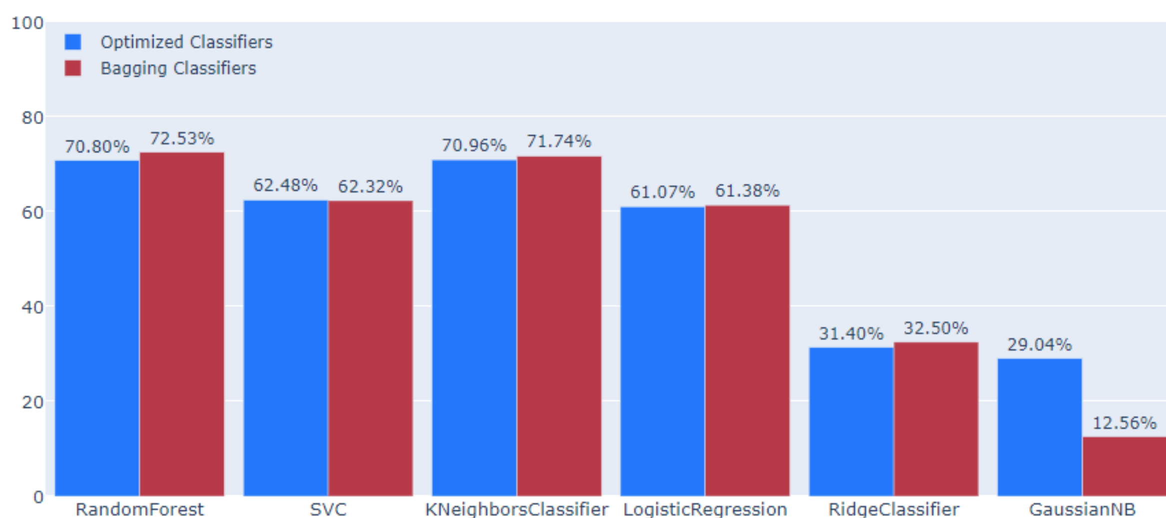



Figura 2.17: Accuracy a confronto tra singoli classifiers ed ensemble bagging su Thingspeak

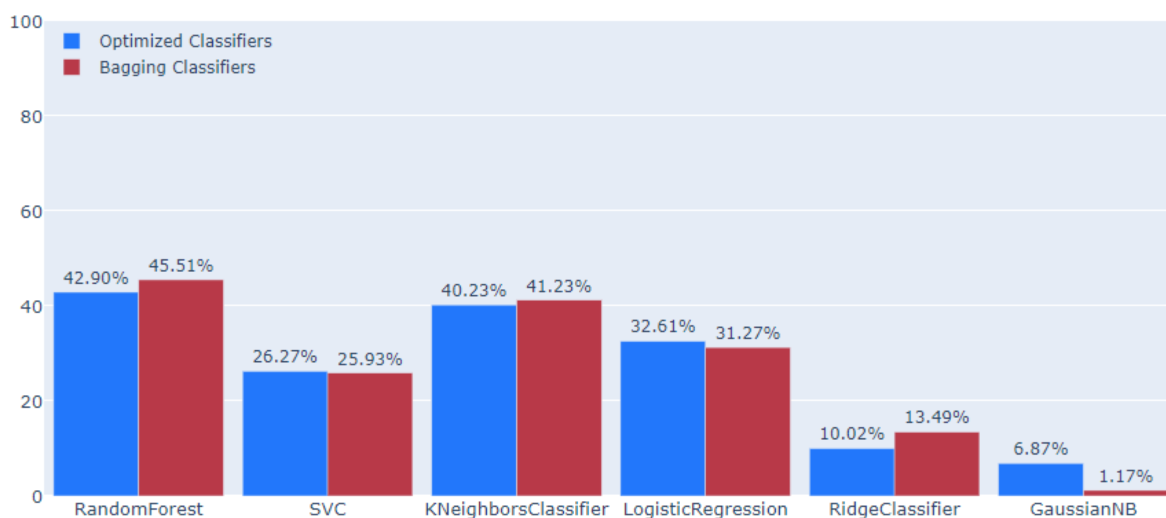



Figura 2.18: f1-score a confronto tra singoli classifiers ed ensemble bagging su Thingspeak

### 2.7.2 Voting

Anche l'implementazione di un voting non presenta difficoltà rilevanti: il procedimento è simile, cambia solo la libreria da importare, questa volta $VotingClassifier$.

I risultati raggiunti - 0.7 accuracy, 0.68 f1-score - sono sì più alti rispetto ai classifica-

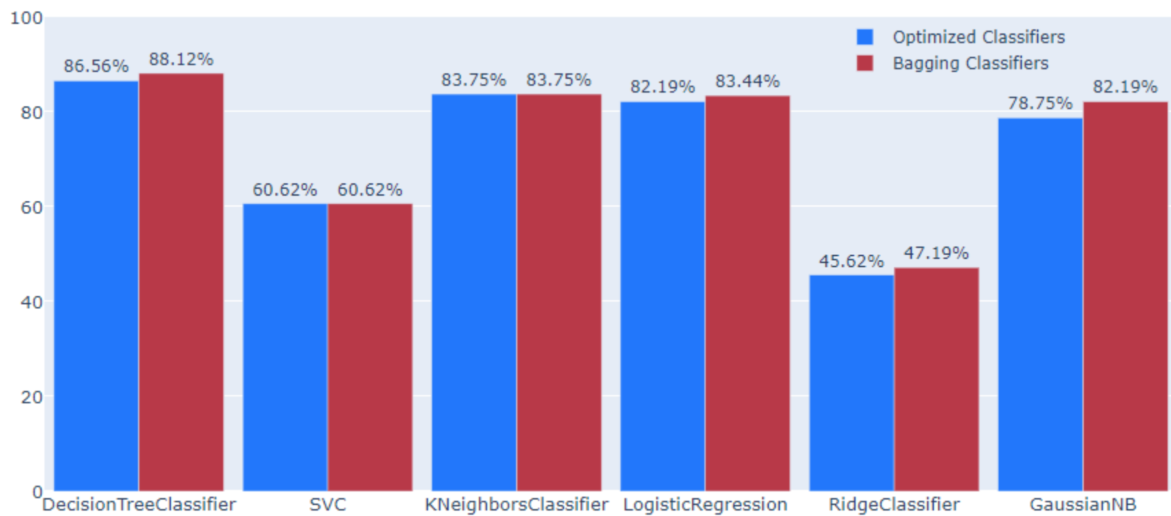



Figura 2.19: Accuracy a confronto tra singoli classifiers ed ensemble bagging su UrbanObservatory

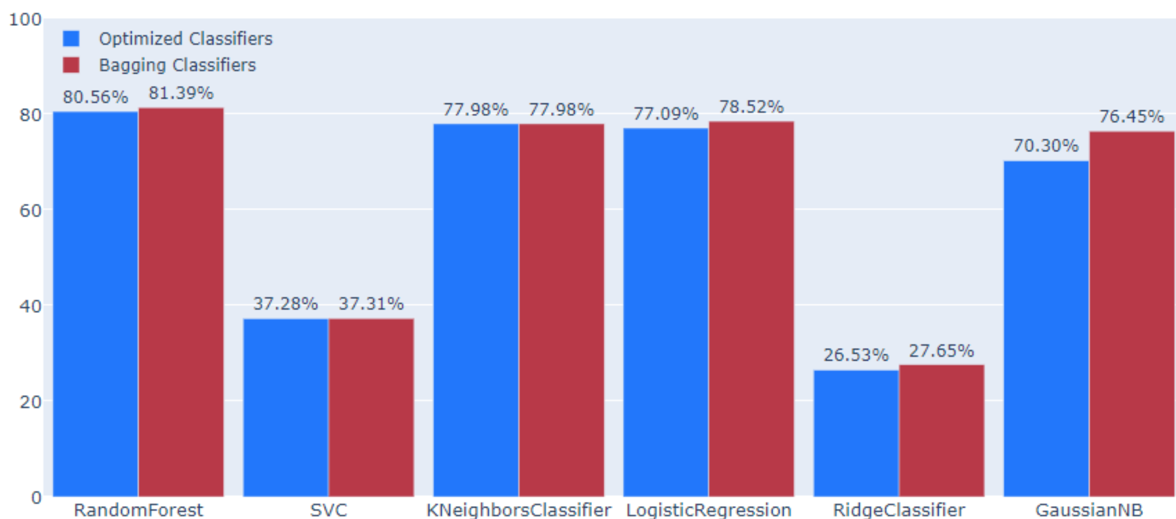



Figura 2.20: f1-score a confronto tra singoli classifiers ed ensemble bagging su UrbanObservatory

tori singoli, ma peggiori rispetto ai migliori bagging: ciò probabilmente avviene perchè il voting dà un peso uguale a tutti i classificatori al suo interno, compresi quelli poco performanti.

### 2.7.3 Boosting

In questa categoria testeremo *AdaBoostClassifier* e *GradientBoostingClassifier*, entrambi forniti come librerie da sklearn. Dato che entrambi hanno dei parametri da settare, operiamo su di essi una GridSearch.

```
from sklearn.ensemble import AdaBoostClassifier,
    GradientBoostingClassifier

ada_boost = AdaBoostClassifier()
grad_boost = GradientBoostingClassifier()
params_ada = {'n_estimators': np.arange(25, 200, 5)}
params_grad = {'n_estimators': np.arange(50, 500, 10)}

ada_best = optimizeParams(ada_boost, params_ada, X_train, y_train)
grad_best = optimizeParams(grad_boost, params_grad, X_train, y_train)

labels = ['Ada Boost', 'Grad Boost']
metrics_boosting = []
boost_array = [ada_best, grad_best]

for clf in boost_array:
    metrics_boosting.append(metrics(clf, X_test, y_test))
```

Listing 2.10: Boosting

### 2.7.4 Stacking

Per implementare uno stacking ci rivolgiamo alla libreria *mlens*, che all'interno del pacchetto Ensembles fornisce il costruttore *SuperLearner*. Questo oggetto ci consente di definire un ensemble composto da più livelli: la predizione effettuata da ciascun livello (in genere calcolata con voto di maggioranza) viene data in input al livello successivo, a quello che viene definito meta-classificatore.

Nel nostro caso, per fare un'analisi più completa possibile, cicliamo tutti i classificatori ottimizzati per trovare la combinazione ottimale su 2 livelli.

```
from mlens.ensemble import SuperLearner

names = ['DecisionTree', 'SVM', 'KNeighbors', 'Logistic', 'Ridge', '
    GaussianNB']

# funzione che restituisce tutte le possibili combinazioni di
    classifiers

def zip_stacked_classifiers(*args):
    to_zip = []
    for arg in args:
        combined_items = sum([list(map(list, combinations(arg, i))) for
    i in range(len(arg) + 1)], [])
        combined_items = filter(lambda x: len(x) > 0, combined_items)
        to_zip.append(combined_items)

    return zip(to_zip[0], to_zip[1])

stacked_clf_list = zip_stacked_classifiers(clfs_best, names)

best_combination = [0.00, "", ""]
best_ensemble = None

# troviamo il miglior ensemble

for clf in stacked_clf_list:
    for meta in clfs:
        ensemble = SuperLearner(random_state = 1000)
        ensemble.add(clf[0])
        ensemble.add_meta(meta)
        ensemble.fit(np.array(X_train), np.array(y_train))
        preds = ensemble.predict(np.array(X_test))
        accuracy = accuracy_score(preds, np.array(y_test))

        if accuracy > best_combination[0]:
            best_combination[0] = accuracy
            best_combination[1] = clf[1]
            best_combination[2] = meta.__class__.__name__
```

```
36            best_ensemble = ensemble
```

Listing 2.11: Stacking

Alla fine dell'esecuzione questo codice ci restituisce la miglior combinazione di classificatori all'interno del SuperLearner:

'Best stacking model is ['KNeighbors', 'GaussianNB'], with meta-learner SVC and accuracy of: 0.721'

Di seguito invece un grafico riassuntivo con tutti gli ensembles testati (per comodità mostriamo solo il bagging più performante, Random Forest):

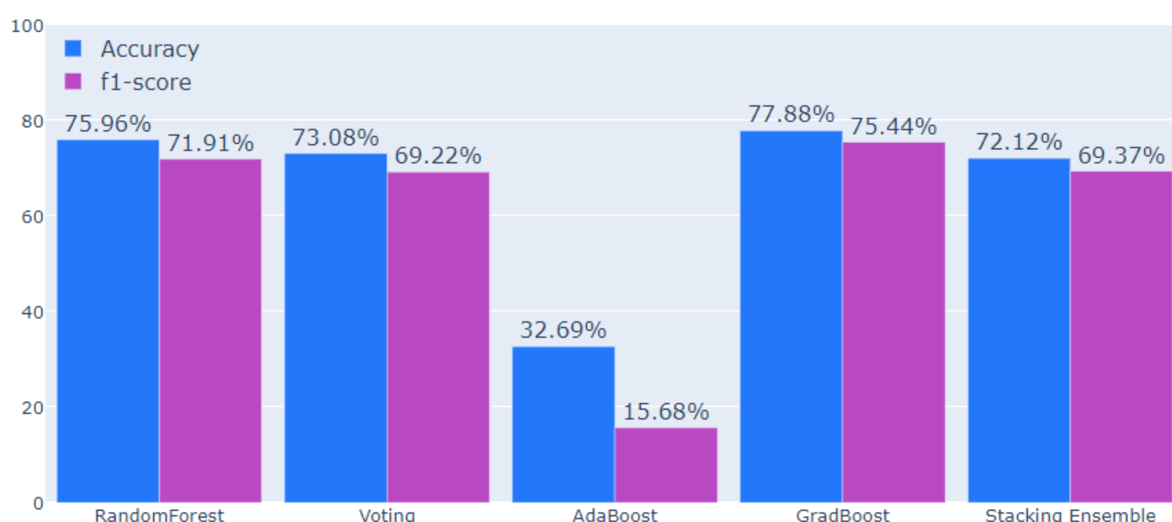



Figura 2.21: Accuracy e f1-score di tutti i classificatori ensemble, su Swissex

I numeri sono "accettabili" in ambito di classificazione di open data IoT, ma possiamo fare di più, specie sul dataset Thingspeak, che fornisce dei metadati di natura testuale (sebbene incompleti). Purtroppo però le librerie ensemble citate fino ad ora non supportano algoritmi operanti su porzioni diverse dei dati, dobbiamo perciò costruire i nostri strumenti: è anche questo lo scopo del framwork progettato ed implementato nel prossimo capitolo.

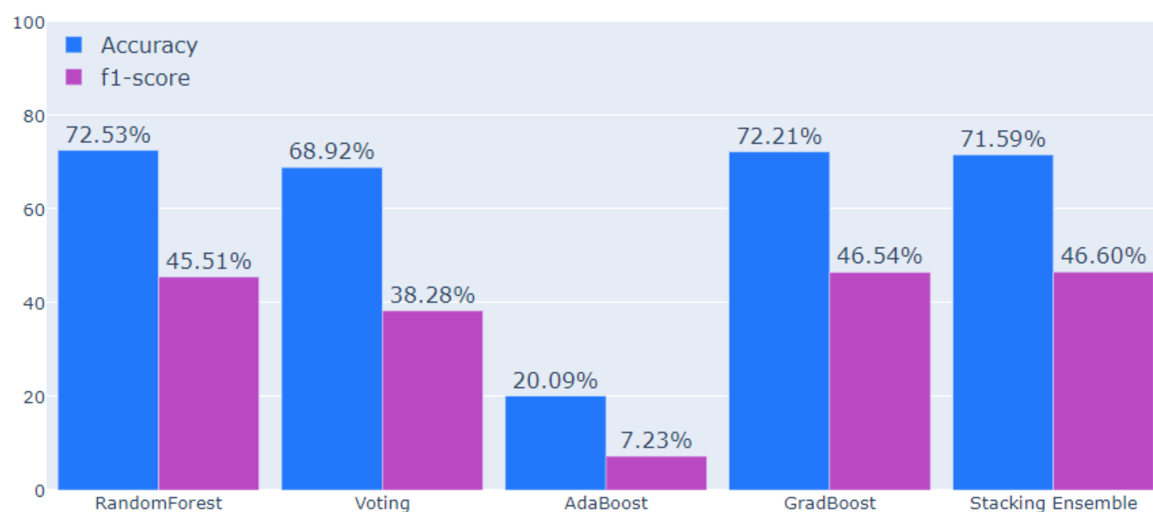



Figura 2.22: Accuracy e f1-score di tutti i classificatori ensemble, su Thingspeak

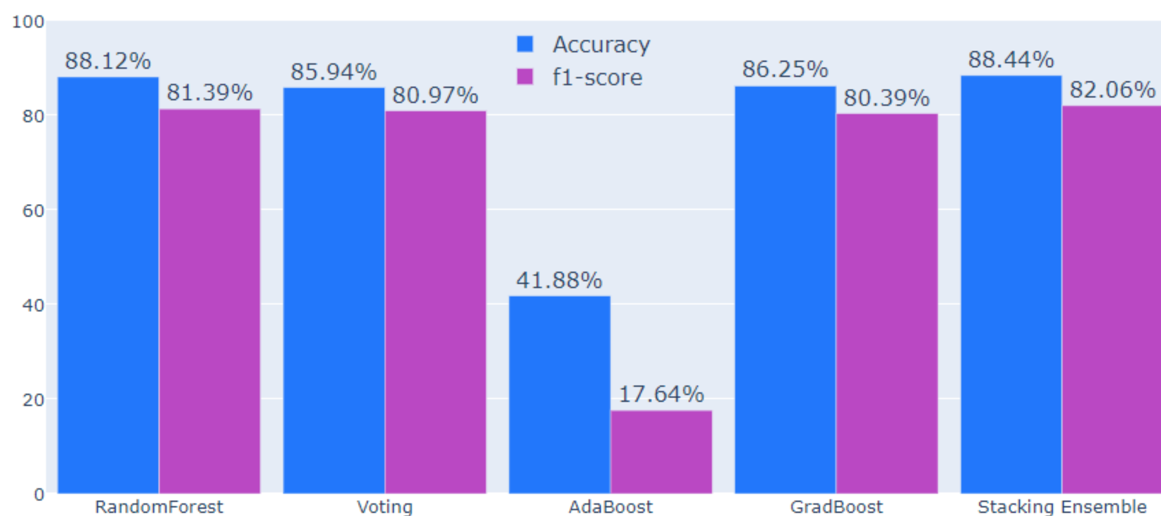



Figura 2.23: Accuracy e f1-score di tutti i classificatori ensemble, su UrbanObservatory

# Capitolo 3

# Metadata-Assisted Cascading Ensemble - MACE

Il framework presentato in questo capitolo, MACE, è un cascading ensemble che ad ogni step (cioè per ogni classificatore in sequenza) filtra tra le classi rimanenti quelle più probabili, per ogni datastream.

Al suo interno impiega, tra gli altri classificatori, un modello di tipo NLP che fa uso dei metadati per migliorare considerevolmente l'accuracy finale dell'ensemble.

Lo sviluppo di tale framework non è stato esente da problemi, alcuni di natura "scientifica", altri di natura implementativa. Di seguito un breve elenco:

- Con quale criterio scegliere l'ordine dei classificatori? Di base vorremmo che come primo step nell'ensemble ci sia un classificatore che gestisca bene il rumore dei dati

- Quale strategia di filtraggio usare? Dare in input ad ogni classificatore un numero di classi da filtrare è senz'altro una scelta viabile, ma è forse un po' troppo rigida

- Come rendere il framework "generico"? L'utilizzatore deve poter inserire al suo interno qualsiasi tipo di classificatore, in qualsiasi ordine, in qualsiasi numero

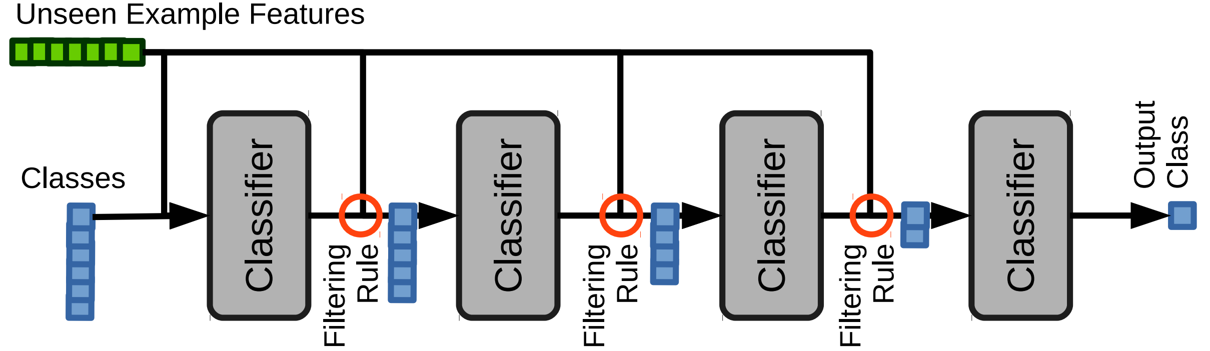



Figura 3.1: Overview del framework

## 3.1 Descrizione del framework

Dato un set di $n$ classificatori $\Gamma = \{\Gamma_1, \ldots, \Gamma_\Theta\}$ addestrati indipendentemente sullo stesso training set, con $|y_train| = c$, definiamo un classificatore con funzione filtrante $\Gamma_i$ e le sue classi di output $C_{out} = \Gamma_i(T, k, C_{in})$, dove $C_{in} \subseteq y_train$ è il set delle classi di input tra le quali il classificatore può effettuare una selezione, $T$ è l'istanza del test set da classificare e $k$ è il criterio di filtraggio.

Nella fase di classificazione $\Gamma_i$ genera una lista ordinata delle probabilità che $T$ ha di appartenere alle classi in $C_{in}$ e $C_{out} \subseteq C_{in}$ rappresenta il set delle $k$ migliori classi, o Top-$k$-classes. Il caso base è dunque l'applicazione di un singolo classificatore che genera in output la classe Top-1 $y = \Gamma_i(T, 1, y_train)$.

Dunque, applicando i classificatori $\Gamma$ in sequenza, MACE applica prima $\Gamma_1(T, k_1, y_train)$, che produce in output $k_1$ classi $\subseteq y_train$, per poi darle in input a $\Gamma_2$ e così via finchè non si arriva alla classe Top-1 restituita da $\Gamma_\Theta$. Tale processo a cascata può essere quindi così definito:

$$y = \Gamma_\theta(T, 1, \Gamma_{\theta-1}(T, k_{\theta-1}, \ \ldots \ \Gamma_1(T, k_1, y_train) \ \ldots))$$

dove $\forall\theta \in \{2, 3, \ \ldots \ , n\} : 0 < k_\theta < k_{\theta-1}$ e $k_1 < c$.

## 3.2 Criterio di ordinamento dei classificatori

La scelta dell'ordine dei classificatori è un passo fondamentale nel funzionamento del framework, in quanto incide pesantamente sulle performance finali dello stesso. Idealmente, e anche un po' ingenuamente, potremmo pensare di provare tutte le permutazioni $n!$ dei classificatori dati in input, ma questa pratica non solo non è flessibile, ma è anche potenzialmente onerosa dal punto di vista computazionale.

Invece, con un semplice test possiamo perlomeno scegliere quale modello di classificazione porre in cima alla nostra sequenza. Definiamo come *top-accuracy* di un classificatore $\Gamma_i$ una funzione $\mathcal{A}_i(k)$ che restituisce l'accuracy del classificatore sulle sue Top-$k$ classi: consideriamo dunque la predizione corretta se la classe ricade nelle Top-$k$ classi predette in ordine di probabilità di appartenenza. Quindi, per costruzione, $\mathcal{A}_i(1)$ sarà la vera probabilità di $\Gamma_i$ e $\mathcal{A}_i(c) = 100\%$.

Nel grafico in figura 3.2 possiamo vedere i risultati di questo test; non ci preoccupiamo al momento della sua implementazione, di natura banale, nè del funzionamento specifico dei classificatori su cui è stato fatto il test - anche se molti sono già riconoscibili dai capitoli precedenti -: tali temi verranno approfonditi nella sezione relativa all'implementazione. Come possiamo vedere dal grafico, dunque, è quasi evidente che il modello di tipo NLP è più performante degli altri per k $> 2$. Ciò significa concettualmente che tale algoritmo è quello che identifica per primo l'insieme delle classi all'interno del quale si può trovare la classe effettiva: è quindi un buon candidato per occupare il primo posto nei nostri test.

## 3.3 Criteri di filtraggio

Come descritto in sezione 3.1, per configurare $n$ classificatori dovremmo dare in input i valori $k_1, \ldots, k_{n-1}$ in modo che ottengano performance massime. Il problema di trovare i valori di $k$ ottimali è un problema matematico difficilmente risolvibile, e non possiamo lasciare all'utilizzatore finale del framework l'onore di scegliere tali valori: un valore non corretto di $k$ potrebbe tralasciare troppe classi, se sottodimensionato, o introdurre troppo rumore, se sovradimensionato. L'approccio più banale, anche qui, sarebbe di provare tutti i valori di $k \in [c]$ e scegliere, ad ogni step $u$, il valore $k_u^*$ che restituisce la

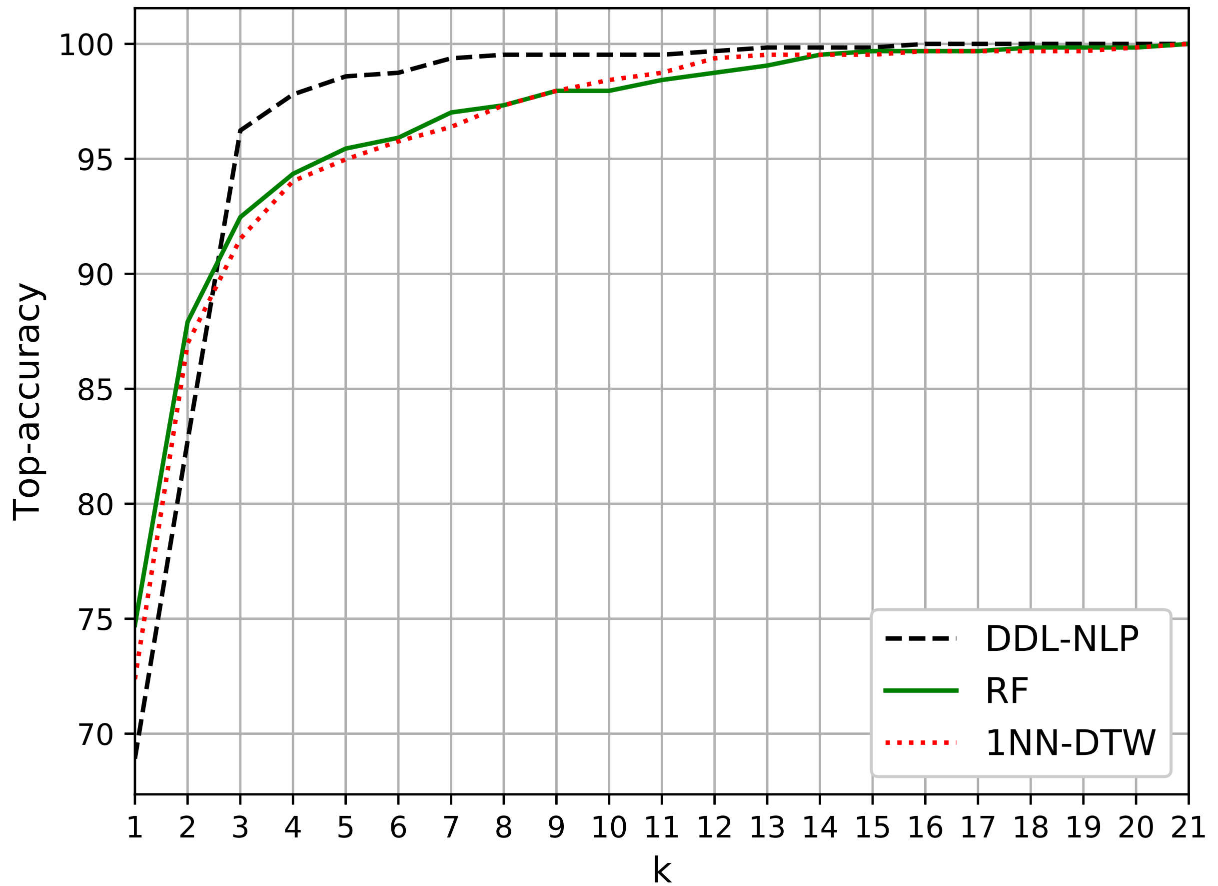



Figura 3.2: *top-accuracy* dei principali classificatori al variare di k

maggiore accuracy in una $c$-fold cross-validation sul training set. Tuttavia anche in questo caso il calcolo sarebbe proibitorio per una quantità di classificatori abbastanza grande, in quanto richiederebbe $c^n$ step. Potremmo tentare di operare una ricerca logaritmica, provando i valori di $k$ da 1 a $c$ $\{2^0, 2^1, 2^2, ..., c\}$, ma la situazione migliora di poco, in quanto il numero di step computazionali sarebbe $(\log c)^n$. Per ovviare a questo problema proponiamo la definizione di una funzione $f$ che genera le serie $k_1, ..., k_{n-1}$ in modo che $\forall u : k_u = f(z, C_{in,u})$ dove $C_{in,u}$ è il set delle classi di input tra ordinare e $z$ è un parametro di filtraggio uguale per tutti i classificatori. In questo modo, in quanto il parametro è condiviso, la ricerca del valore di $z$ ottimale è indipendente dal numero di classificatori, e si riduce a $O(1)$. L'algoritmo 1 descrive il framework introdotto in sezione 3.1 con l'aggiunta delle modifiche appena discusse. In particolare il framework si ferma quando

$f(z, C_{in,u}) = 1$ o $\Gamma_u$ è l'ultimo classificatore (righe 5-8).

**Algorithm 1** Pseudocodice di MACE

**Require:** Training set $X_train$, istanza di test $T \in X_test$ , set di $n$ classificatori $\Gamma$

**Ensure:** output $|C_{out}| = 1$

1: estrazione delle classi $L$ da $y_train$ e delle feature $BOS$ per $X_train$, $T$
2: ordinamento dei classificatori $\Gamma$ su $X_train$
3: scelta di una strategia di filtraggio $f$ con parametro $z$
4: $C_{out} = \Gamma_1(T, f(z,\text{L}),\text{L})$
5: **for** $\Gamma_u \in \Gamma \setminus \Gamma_1$ **do**
6: $\quad C_{out,u} = \Gamma_u(T, f(z, C_{out}), C_{out})$
7: $\quad C_{out} = C_{out,u}$
8: **end for**
9: **return** $C_{out}$

Nella prima versione del framework, quella descritta in questa tesi, sono incluse due strategie di filtraggio che si basano sulla definizione di $f(z, C_{in,u})$ sopracitata: una *rank-based* e una *distribution-based.*

### 3.3.1 Strategie Rank-Based

Questa categoria di strategie mira a estrarre direttamente le classi più probabili dalla lista delle classi di input. La specifica di un valore di $k$ per ogni classificatore è un'applicazione di tale strategia, ma come abbiamo detto è poco flessibile, e pone il problema della scelta di multipli valori di $k$. Invece proponiamo un'altra strategia, chiamata Quantile Filtering (QF), che definisce $\forall u : k_u = f(z, C_{in,u}) = \lceil z \cdot |C_{in,u}| \rceil$. La funzione $f$ definisce le top classi da estrarre come una percentuale della dimensione del set. Per esempio, se $z = 0.25$, il quarto quartile di $C_{in}$ viene passato allo step successivo. Questa strategia generalizza bene l'approccio Top-$k$, ma è forse troppo generica, perchè si comporta allo stesso modo su tutti i classificatori: il classificatore in prima posizione restituirà $k_1 = z \cdot c$ classi, il secondo $k_2 = z \cdot k_1$, e così via. Manca dunque un'analisi della distribuzione delle probabilità prodotte da ogni classificatore.

### 3.3.2 Strategie Distribution-Based

Queste strategie tengono conto della distribuzione delle probabilità per selezionare la porzione di classi da passare allo step successivo. Definiamo all'interno di questa categoria la strategia Survival of the Fittest (SoF), che restituisce le classi tali che $P(l_i) \geq \mu + z\sigma$, dove $P(l_i)$ è la probabilità attribuita dal classificatore alla classe $l_i$, $\mu$ è la media delle probabilità e $\sigma$ è la deviazione standard. $z$ è il parametro che gestisce la generosità dell'intervallo i.e. uno $z$ piccolo (anche negativo) tenderà a comprendere più classi nella selezione, uno $z$ più grande terrà per buone solo classi che si distanziano molto, in positivo, dalla media delle probabilità. Quindi, $\forall u : f(z, C_{in,u}) = |\{l_i \mid P(l_i) \geq \mu + z\sigma\}|$ dove $\mu$ e $\sigma$ sono calcolati dalle probabilità delle classi in output prodotte per $C_{in,u}$. In questo modo il numero di classi prodotte in output da ogni classificatore può variare, anche in base all'istanza del test set presa in considerazione.

## 3.4 Implementazione

Come già accennato nell'introduzione di questo capitolo, l'implementazione del framework MACE pone dei problemi rilevanti, la cui soluzione è senz'altro degna di essere esplorata. Il linguaggio scelto per l'implementazione è Python, lo stesso usato nell'implementazione delle analisi esposte nel capitolo precedente. Python si presta bene a lavori di machine learning grazie alle innumerevoli librerie disponibili (abbiamo già citato per esempio la più importante, scikit-learn) e alla sua natura di alto livello.

### 3.4.1 Modifiche agli algoritmi

La struttura base a cui ci affidiamo per gli algoritmi è quella dei modelli forniti da scikit-learn: devono tutti implementare un metodo $fit()$, un metodo *predict*(), e un metodo *predict_proba*(); in particolare l'ultimo di questi ci servirà particolarmente in quanto la logica fondante del framework si basa sul predire le probabilità di appartenenza alle classi per ogni istanza del test set.

Non sono dunque necessarie modifiche per gli algoritmi forniti dalla libreria (tutti i BOS e i TSC basati su kNN), ma a tutti gli altri manca il metodo *predict_proba*().

In particolare, non ci preoccupiamo tanto degli altri algoritmi TSC, poco performanti e computazionalmente costosi, ma di NLP: la nostra implementazione va rivista.

Per fare ciò prendiamo alcune delle caratteristiche dell'algoritmo introdotto in [19]: in fase di training dunque costruiamo un "dizionario" che, per ogni classe distinta $L_j \in y_train$, colleziona tutti i metadati appartenenti a quella classe - nel nostro caso, solo i nomi degli stream -; questa forma è chiamata Bag-of-Words ($BOW_j$). In fase di testing, poi, per ogni classe $L_j$ viene calcolata la minima edit-distance tra tutti i metadati collezionati per quella classe $d_j = min\{ed(w, s) \,|\, s \in BOW_j\}$, con $w$ i metadati dell'istanza di test. La classe predetta è quella con distanza minima, ma possiamo anche calcolare le probabilità di appartenere alle varie classi invertendo e normalizzando l'edit-distance per ogni classe. Chiameremo questo algoritmo DDL-NLP.

```
1  class NLP_Classifier(BaseEstimator):
2
3      MAX_DIST = 1000
4      dictionary = {}
5      probs = []
6
7      def __init__(self):
8          pass
9
10     def fit(self, X_train, y_train):
11
12         self.classes = set(y_train)
13         # istanziamo il dizionario
14         for c in self.classes:
15             self.dictionary[c] = []
16
17         X_train = X_train['metadata']
18         # popoliamo il dizionario
19         for i, word in enumerate(X_train):
20             c1 = y_train[i]
21             self.dictionary[c1].append(word)
22
23     def predict_proba(self, X_test):
```

```
24
25         self.probs = []
26         for i, word in enumerate(X_test['metadata']):
27 
28             edit_distances = []
29             for c1 in self.classes:
30                 # calcoliamo le distanze per ogni parola nel dizionario
31                 dam = [damerau_levenshtein_distance(word, field)
32                         for field in self.dictionary[c1]]
33                 if len(dam) > 0:
34                     # per evitare di dividere per 0 dopo
35                     edit_distances.append((min(dam)+1))
36                 else:
37                     edit_distances.append(MAX_DIST)
38 
39             # invertiamo le distanze
40             edit_distances = np.true_divide(1, edit_distances)
41             # normalizziamo
42             proba = np.true_divide(edit_distances, sum(edit_distances))
43             self.probs.append(proba)
44 
45         return self.probs
46 
47     def predict(self, X_test):
48 
49         if (len(self.probs) == 0):
50             self.probs = self.predict_proba(X_test)
51         y_pred = []
52         for series in self.probs:
53             y_pred.append(np.argmax(series))
54 
55         return y_pred
```

Listing 3.1: Algoritmo NLP rivisitato

### 3.4.2 Preprocessing del dataset

Per garantire la flessibilità del framework non possiamo dare per scontato che i modelli di classificazione sappiano quali colonne del dataset prendere in input: dobbiamo in qualche modo indicargliele. Riorganizziamo dunque il dataset, o meglio la $X$, sotto forma di dizionario: gli algoritmi di tipo BOS accederanno alla chiave $'statistics'$, quelli di tipo TSC accederanno alla chiave $'timeseries'$, quelli di tipo NLP alla chiave $'metadata'$.

```
y = []
X = {'metadata': [], 'statistics' : [], 'timeseries' : []}

path = ""

# calcoliamo le features di ogni timeseries
with open(path + 'ThingspeakEU.meta.csv', 'r', encoding='utf-8') as
    dati:
    for row in dati:
        riga = row.strip().split(',')

        classe = int(riga[8])
        y.append(classe)

        valori = np.array(riga[9:]).astype(np.float)
        X['timeseries'].append(valori)

        # metadati
        stream_name = riga[1]
        X['metadata'].append(stream_name)

        # statistiche
        valori = np.array(riga[9:]).astype(np.float)
        media = np.mean(valori)
        mediana = np.median(valori)
        maxim = np.max(valori)
        minim = np.min(valori)
        std_dev = np.std(valori)
        rms = np.sqrt(np.mean(np.square(valori)))
```

```
        quantile = np.quantile(valori, 0.4)
        i_q_r = iqr(valori)
        simmetria = skew(valori)
        curtosi = kurtosis(valori)
        rang = maxim - minim

        features = [rang, maxim, std_dev, rms, media, minim, quantile,
            mediana, curtosi, simmetria, i_q_r]
        X['statistics'].append(features)

X = pd.DataFrame(X)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =
    0.3, random_state = 100)
```

Listing 3.2: Lettura del dataset

Trasformiamo poi il dizionario in un dataframe Pandas (un'altra comodissima libreria di Python) per renderlo compatibile con la funzione *train_test_split* di sklearn.

### 3.4.3 Classi wrapper

Dobbiamo ora fare in modo che i classificatori dati in input al framework sappiano quale parte del dataset analizzare: tutti gli algoritmi BOS si devono comportare allo stesso modo, accedendo alla chiave *'statistics'* del dataset, e così via. In questo caso utilizziamo un pattern di programmazione chiamato Decorator [13]: questo pattern permette di aggiungere funzionalità ad un oggetto esistente (i.e. i vari modelli di classificazione, nel nostro caso) "avvolgendolo" in una classe decoratrice. Questa classe, che chiameremo *wrapper*, conserverà come variabile d'istanza un riferimento all'oggetto originale, e disporrà degli stessi metodi, richiamando all'interno di essi prima le funzionalità aggiuntive, e poi i metodi originali degli oggetti wrappati. A seguire un esempio:

```
class BOS_Classifier(BaseEstimator):

    def __init__(self, classifier):
        self.classifier = classifier

```

```
6     def fit(self, X_train, y_train):
7
8         X_train = list(X_train['statistics'])
9         self.classifier.fit(X_train, y_train)
10
11    def predict_proba(self, X_test):
12
13        X_test = list(X_test['statistics'])
14        return self.classifier.predict_proba(X_test)
15
16    def predict(self, X_test):
17
18        X_test = list(X_test['statistics'])
19        return self.classifier.predict(X_test)
```

Listing 3.3: BOS Wrapper

### 3.4.4 Implementazione dei criteri di filtraggio

Non rimane ora null'altro se non l'implementazione vera e propria dell'algoritmo. Decidiamo di istanziare il framework con tre parametri:

- *criterion*, il criterio di filtraggio, una stringa
- *parameters*, un dizionario contenente i valori di $k$ o $z$
- un array *layers* contenente per ogni classificatore un dizionario con chiavi *'type'* e *'name'*: all'interno di *'name'* la stringa che istanzia il classificatore

Il costruttore del nostro oggetto *EnsembleFramework* sarà dunque così implementato:

```
1 def __init__(self, criterion = 'topk', layers = [{'type' : 'NLP'}, {'
    type' : 'BOS', 'name' : 'DecisionTreeClassifier()'}], params = {'k'
    : [4, 1]}):
2         self.criterion = criterion
3         self.layers = layers
```

```
4        self.params = params
```

Listing 3.4: Costruttore

I valori attribuiti nel costruttore sono quelli di default. In quanto l'algoritmo NLP è per ora uno solo omettiamo di esplicitare la chiave $'name'$.

Definiamo ora un metodo $run()$ che prenda in input $X_train$, $y_train$ e $X_test$ e si occupi della fase di addestramento e predizione.

La fase di addestramento è quasi banale: dobbiamo istanziare i nostri classificatori, all'interno delle nostre classi wrapper, e dobbiamo addestrarli indipendentemente sul dataset. Prepariamo poi una variabile *classes* contenente le classi da dare in input al primo classificatore (cioè tutte).

```
1  def run(self, X_train, y_train, X_test):
2          classes = [list(set(y_train))] * len(X_test)
3          self.classifiers = []
4          for classifier in self.layers:
5              if (classifier['type'] == 'BOS'):
6                  clf = BOS_Classifier(eval(classifier['name']))
7              elif (classifier['type'] == 'NLP'):
8                  clf = NLP_Classifier()
9              elif (classifier['type'] == 'TSC'):
10                 clf = TSC_Classifier(eval(classifier['name']))
11             clf.fit(X_train, y_train)
12             self.classifiers.append(clf)
```

Listing 3.5: Fase di addestramento

Nella fase di predizione/classificazione poi implementiamo i vari criteri di classificazione, partendo dal più semplice, Top-$k$:

```
1  if (self.criterion == 'topk'):
2              for clf, k in zip(self.classifiers, self.params['k']):
3                  probs = clf.predict_proba(X_test)
4                  for i, series_probs in enumerate(probs):
5                      classes[i] = nlargest(k, classes[i],
```

```
6                                key = lambda x : series_probs[x])
```

Listing 3.6: Strategia di filtraggio Top-$k$

Come possiamo vedere si tratta di applicare un semplice ciclo for per iterare sui classificatori e sui valori di $k$. Ordiniamo e filtriamo le classi poi in base al loro valore all'interno di *series_probs* grazie al metodo *nlargest* della libreria *heapq*.

Un metodo di filtraggio leggermente più complesso è Quantile Filtering:

```
1 elif (self.criterion == 'qf'):
2             for j, clf in enumerate(self.classifiers):
3                 probs = clf.predict_proba(X_test)
4                 for i, series_probs in enumerate(probs):
5                     k = int(len(classes[i])*self.params['z'])
6                     if (j == (len(self.classifiers)-1) or k == 0):
7                         k = 1
8                         classes[i] = nlargest(k, classes[i],
9                             key = lambda x : series_probs[x])
```

Listing 3.7: Strategia di filtraggio QF

In questo caso deriviamo il valore di $k$ per ogni classificatore a partire da $z$ e, ad ogni step di filtraggio, controlliamo che non ci troviamo all'ultimo classificatore della sequenza o che $k$ sia stato approssimato a 0: in tal caso poniamo $k = 1$ e usciamo dall'iterazione.

Veniamo poi all'ultimo criterio di filtraggio, Survival of the Fittest:

```
1 elif (self.criterion == 'sof'):
2             for j, clf in enumerate(self.classifiers):
3                 probs = clf.predict_proba(X_test)
4                 for i, series_probs in enumerate(probs):
5                     classes[i] = self.survival(series_probs, classes[i
    ])
6                     if (j == (len(self.classifiers)-1)):
7                         classes[i] = nlargest(1, classes[i], key =
    lambda x : series_probs[x])
```

Listing 3.8: Strategia di filtraggio SoF

La struttura generale è la stessa, ma questa volta richiamiamo la funzione *survival* da noi definita per filtrare le classi:

```
def survival(self, probs, classes):
        survived = []
        survived_probs = []

        for c in classes:
            survived_probs.append(probs[c])

        media = np.mean(survived_probs)
        std_dev = np.std(survived_probs)

        for c in classes:
            if (probs[c] >= (media + self.params['z']*std_dev)):
                survived.append(c)

        if (len(survived) == 0):
            survived = nlargest(1, classes, key = lambda x : probs[x])

        return survived
```

Listing 3.9: funzione *survival*

In questa funzione calcoliamo la media $\mu$ e deviazione standard $\sigma$ delle classi in input e restituiamo solo quelle con $P(l_i) \geq \mu + z\sigma$.

Alla fine del metodo *run* la variabile *classes* sarà la nostra *y_pred*: possiamo darla in input al seguente metodo per calcolare l'accuracy del framework:

```
def accuracy(self, classes, y_test):
        y_pred = []
        for pred in classes:
            y_pred.append(*pred)

        return accuracy_score(y_pred, y_test)*100
```

Listing 3.10: Calcolo accuracy

# Capitolo 4

# Validazione

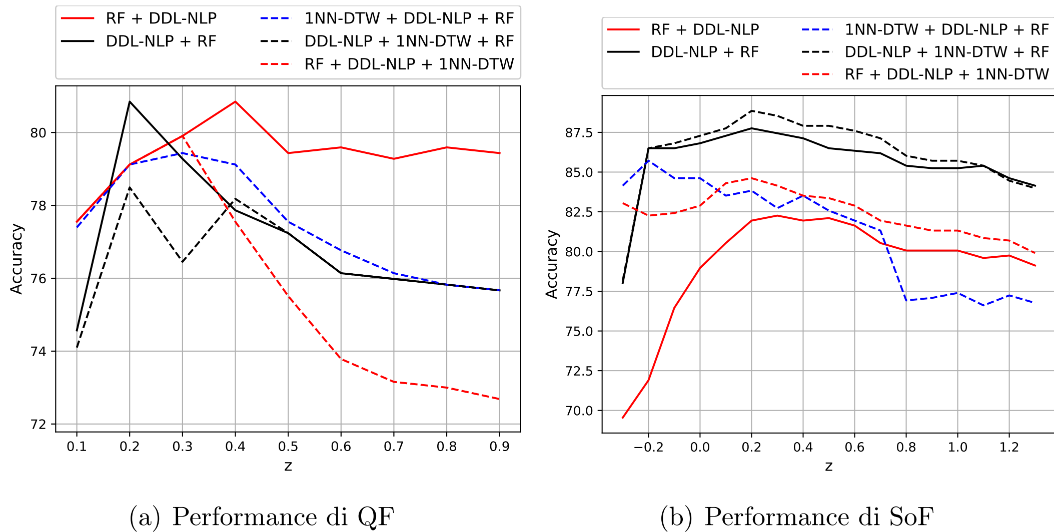
In questo capitolo valuteremo le performance del framework MACE proposto in questa tesi. Testeremo al suo interno prima due (BOS e NLP) e poi tre algoritmi di classificazione (BOS, TSC e NLP), testando tutti gli ordinamenti possibili - anche se probabilmente, come abbiamo scoperto in sezione 3.2, il framework performerà meglio con NLP al primo posto -. Per comodità testeremo solo i migliori algoritmi per ogni categoria: RandomForest per BOS, 1NN-DTW per TSC, DDL-NLP per NLP.

Tutti i test sono svolti sul dataset Thingspeak, l'unico dotato di metadati.

## 4.1 Tuning dei parametri

Come spiegato in sezione 3.3, per operare il framework andrebbero in teoria trovati i valori di $k$ ottimali, compito ad elevato costo computazionale. Con le due strategie di filtraggio proposte, tuttavia, i valori di $k$ sono generati a partire da una funzione generatrice $f(z, C_{in,u})$, per cui trovare il valore ottimale di $z$ non è particolarmente oneroso.

In figura 4.1(a) viene mostrato il comportamento di QF: possiamo vedere che due classificatori sono in generale meglio di tre, probabilmente a causa della rigidità del metodo di filtraggio - per ogni classificatore viene presa una percentuale fissa delle migliori classi -. Forse dunque all'aumentare dei classificatori risulta più difficile trovare quella percentuale fissa, motivo per il quale le performance di tre classificatori hanno un andamento quasi randomico. Possiamo poi verificare che le combinazioni che iniziano con un




(a) Performance di QF (b) Performance di SoF

Figura 4.1: Performance di QF e SoF per diversi valori di $z$. Sono mostrate solo le combinazioni migliori.

NLP hanno performance massime per valori di $z$ (quindi $k$) bassi, a conferma di quando mostrato empiricamente in sezione 3.2.

In figura 4.1(b) invece possiamo notare risultati più regolari: le combinazioni con NLP al primo posto hanno accuracy più alta, e hanno un massimo localizzato per $z = 0.2$, con la combinazione a tre leggermente più performante di quella a due - anche se non signficativamente, e ad un costo, quello del 1NN-DTW, non giustificabile -. Ciò conferma che la strategia di filtraggio SoF è molto più flessibile ed adattabile al numero di classificatori, in quanto il numero di classi selezionate ad ogni step dipende dalla specifica distribuzione di probabilità prodotta per quell'elemento del test set.

## 4.2 Risultati finali

Come già spiegato nei capitoli precedenti, non ci sono - perlomeno, a nostra conoscenza - framework o librerie ensemble che permettano di concatenare algoritmi che operano su diverse porzioni (leggasi domini) del dataset, motivo per il quale sarà difficile trovare dei rivali con cui confrontare le performance (non è possibile ad esempio confrontare l'accuracy con l'ensemble stacking della libreria *mlens*).

| | $\Theta = 2$ | $\Theta = 3$ |
|---|---|---|
| Soft Voting | 83.8% | 83.5% |
| MACE con QF | 80.9% | 79.4% |
| MACE con SoF | 87.8% | 88.9% |

Tabella 4.1: Accuracy di strategie ensemble a due e tre classificatori

Possiamo però implementare una nostra versione del *voting ensemble* introdotto nel paragrafo 2.7.2; anzichè un voto di maggioranza (hard voting), però, implementiamo un soft voting: calcoliamo la media delle probabilità prodotte da ciascun algoritmo per ogni classe, e la media più alta sarà la classe predetta.

Confrontiamo i migliori risultati prodotti da Soft Voting, QF e SoF nella tabella 4.1: notiamo subito che QF produce risultati più bassi di Soft Voting, probabilmente a causa dei suoi problemi di rigidità citati pocanzi. Allo stesso tempo però SoF migliora di cinque punti percentuali il migliore risultato di Soft Voting, a riprova della solidità del criterio di filtraggio frutto del lavoro di questa tesi.

Notiamo poi che l'accuracy di Soft Voting decresce con l'aggiunta di un terzo classificatore, mentre quella di SoF migliora leggermente: evidentemente Survival of the Fittest gestisce meglio il rumore introdotto dalla concatenazione di un nuovo classificatore.

In generale comunque i risultati raggiunti sono degni di rispetto: siamo partiti da un 70% ottenuto con un semplice Decision Tree e siamo arrivati a sfiorare il 90%, un numero decisamente poco comune da vedere su un dataset così eterogeneo e così poco numeroso come Thingspeak.

# Capitolo 5

# Conclusioni e sviluppi futuri

Questa tesi è nata da un duplice scopo: analizzare estensivamente lo stato dell'arte in ambito classificazione open data IoT e fornire un contributo sostanziale e positivo in tale ambito.

Abbiamo adempiuto al primo scopo nei capitoli 1 e 2, nei quali abbiamo prima presentato e discusso, e poi implementato e validato una grossa fetta degli algoritmi presentati in letteratura inerenti il problema. Abbiamo volutamente escluso algoritmi di deep learning perchè non adatti alla numerosità dei dataset in esame.

Abbiamo discusso ampiamente dei risultati ottenuti affinchè servissero da baseline con cui confrontare le nostre idee.

Nei capitoli 3 e 4 abbiamo poi presentato il nostro contributo al settore: un framework ensemble, MACE, che concatena a cascata algoritmi che operano su tre domini diversi: BOS, TSC e NLP. Abbiamo discusso ampiamente dei problemi sorti sia in fase di progettazione che in fase di implementazione, poichè di grande spunto per gli sviluppi futuri.

Abbiamo introdotto due criteri di filtraggio all'interno del framework, che in fase di testing si sono rivelati di successo (in particolare SoF): abbiamo migliorato la miglior accuratezza di cinque punti percentuali rispetto ad altri algoritmi multi-dominio (Soft Voting) e di più di dieci punti percentuali rispetto ai migliori algoritmi a singolo dominio anche ensemble (Random Forest BOS, 1NN-DTW). La miglior combinazione di parametri ha sfiorato il 90% di accuracy, valore mai raggiunto in letteratura su dataset così

eterogenei.

## 5.1 Sviluppi futuri

Il punto in cui siamo giunti non è un punto di arrivo, ma di partenza. Fermo restando che un articolo scientifico sul framework proposto in questa tesi è comunque in cantiere, sono tre le direzioni che questo lavoro può intraprendere - e non sono mutualmente esclusive, anzi, ogni percorso gioverebbe da eventuali buoni risultati riportati negli altri -:

- Lato usabilità: si potrebbe pensare di trasformare questo framework in una libreria Python liberamente accessibile; per fare ciò bisogna lavorare molto sul codice: bisogna esporre agli utenti i giusti metodi, magari semplificando il processo di inserimento degli input. Bisogna poi decidere se mantenere fissa la struttura del dataset presentata in questa tesi, e in caso fornire dei metodi generici per la conversione.
- Lato validazione: sono decisamente necessari più test. Bisogna ancora capire se il framework è robusto per qualsiasi numero di classificatori i.e. se aggiungendo un classificatore poco performante in coda alla sequenza questo viene ignorato dal framework. Necessario poi verificare se l'accuratezza del framework aumenta all'aumentare dei classificatori, o c'è un punto di massimo - che io immagino essere due, non ritenendo significativo l'aumento dell'1% ottenuto con tre -. Implementare un modello stacking multi-dominio per confrontare i risultati.
- Lato "scientifico": trovare un modo per ordinare i classificatori in modo automatico. Sviluppare SoF in qualcosa di più complesso (la direzione è giusta). Progettare ed implementare un filtraggio "infinito": finchè non si raggiunge una percentuale di probabilità di una classe, altri classificatori vengono accodati alla sequenza. Progettare e implementare il *feature tuning*: dare la possibilità ad ogni classificatore di aggiungere, rimuovere o modificare le feature su cui addestrare il successivo.

# Ringraziamenti

Chi mi conosce sa che non sono una persona particolarmente espansiva, e che scrivere queste parole per me è quasi più difficile che scrivere venti tesi diverse, ma nonostante questo ho deciso di cogliere quest'occasione per spendere due parole per le persone che per me contano di più. Spero solo che non me le facciate leggere dal vivo perchè è una roba imbarazzantissima.

Ringrazio il Prof. Marco Di Felice per avermi dato l'opportunità di sviluppare questo argomento di tesi, nonchè di conoscere il Dott. Federico Montori, correlatore di questa tesi; ringrazio in particolar modo quest'ultimo, che mi ha seguito lungo tutto questo percorso: sarebbe un po' cliché dire che mi ha trasmesso l'amore per la ricerca, ma sicuramente ora è una strada che sto valutando seriamente per il mio futuro, e questo lo devo a lui.

Un ringraziamento poi a tutti quelli che mi hanno accompagnato in questi tre anni di avventura bolognese, compresi coloro che ho conosciuto solo negli ultimi mesi: non faccio una lista, dato che sarebbe troppo lunga, ma se siete stati ad una delle nostre* feste... siete chiaramente inclusi. Quelle feste sono il ricordo più bello che ho di questi anni, ed è solo grazie a voi.

*a proposito di nostre - ed avete capito tutti dove voglio arrivare - : un ringraziamento al mio ex coinquilino, che mi ha accompagnato in questi tre anni di avventura bolognese. E in cinque di superiori. E in tanti degli anni a venire, finchè perlomeno gli infortuni non avranno la meglio sul suo corpo. Ci sono tante cose di cui ringraziarlo, molte delle quali gli dirò di persona, altrettante che rimarranno nel non detto, ma voglio che almeno questo rimanga scritto nero su bianco: grazie Andrea.

Un ringraziamento al mio attuale coinquilino, Gianluca, che mi ha spinto ad intraprendere questo nuovo percorso che mai da solo avrei scoperto, e forse nemmeno mai intrapreso. Mi hai condannato a guadagnare milioni ed essere super produttivo, prometto di renderti la vita un inferno.

Un ringraziamento a tutti quelli che hanno fatto chilometri su chilometri per venire qui a vedermi, nonostante i miei inviti improvvisati all'ultimo momento e i prezzi dei biglietti alle stelle: non mi dimenticherò mai di questo (sì, è una minaccia).

Un ringraziamento anche a chi si trova altrove e non ce l'ha fatta: in questi anni ho scoperto che non c'è sensazione più bella che viaggiare sapendo che arrivato a destinazione ci sia qualcuno ad accoglierti - che tu sia a Lecce, Torino, Milano, Roma o a Brescia, grazie.

Un ringraziamento all'Eurogym e a tutti i suoi membri, il mio primo approccio ad uno sport è stato grazie a voi.

Ultimo ringraziamento, ma non per importanza, quello alla mia famiglia. Anche loro, forse come me, non sono bravi a supportare con le parole - ed io di questo spesso mi lamento -, ma quello che non ho mai detto loro è che compensano ampiamente con i fatti. Se sono così e sono qui è grazie a loro.

# Bibliografia


[1] Anthony J. Bagnall et al. «The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances». In: *Data Min. Knowl. Discov.* 31.3 (2017), pp. 606–660.

[2] Gustavo E. A. P. A. Batista, Xiaoyue Wang e Eamonn J. Keogh. «A Complexity-Invariant Distance Measure for Time Series». In: *Proceedings of the Eleventh SIAM International Conference on Data Mining, SDM 2011, April 28-30, 2011, Mesa, Arizona, USA*. 2011, pp. 699–710.

[3] Luca Bedogni, Marco Di Felice e Luciano Bononi. «By train or by car? Detecting the user's motion type through smartphone sensors data». In: *2012 IFIP Wireless Days*. IEEE. 2012, pp. 1–6.

[4] Jean-Paul Calbimonte et al. «Deriving Semantic Sensor Metadata from Raw Measurements». In: *Proceedings of the 5th International Workshop on Semantic Sensor Networks, SSN12, Boston, Massachusetts, USA, November 12, 2012*. A cura di Cory A. Henson, Kerry Taylor e Óscar Corcho. Vol. 904. CEUR Workshop Proceedings. CEUR-WS.org, 2012, pp. 33–48.

[5] Min Chen et al. «Disease prediction by machine learning over big data from healthcare communities». In: *Ieee Access* 5 (2017), pp. 8869–8879.

[6] Fred Damerau. «A technique for computer detection and correction of spelling errors». In: *Commun. ACM* 7.3 (1964), pp. 171–176.

[7] Marco Di Felice, Dott Federico Montori e Mattia Maniezzo. «Realizzazione e validazione sperimentale di un dataset open per l'Internet of Things». In: ().

[8] Pinar Donmez. «*Introduction to Machine Learning*, 2nd ed., by Ethem Alpaydın. Cambridge, MA: The MIT Press 2010. ISBN: 978-0-262-01243-0. 54/39.95 + 584*pages*». In: *Natural Language Engineering* 19.2 (2013), pp. 285–288.

[9] Abdur Rahim Mohammad Forkan et al. «An Industrial IoT Solution for Evaluating Workers' Performance Via Activity Recognition». In: *39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, TX, USA, July 7-10, 2019*. 2019, pp. 1393–1403.

[10] Francis Galton. «Vox populi (the wisdom of crowds)». In: *Nature* 75.7 (1907), pp. 450–451.

[11] Mari Cruz Garcia, Miguel A Sanz-Bobi e Javier Del Pico. «SIMAP: Intelligent System for Predictive Maintenance: Application to the health condition monitoring of a windturbine gearbox». In: *Computers in Industry* 57.6 (2006), pp. 552–568.

[12] Josif Grabocka et al. «Learning time-series shapelets». In: *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*. A cura di Sofus A. Macskassy et al. ACM, 2014, pp. 392–401.

[13] Ralph Johnson e John Vlissides. «Design patterns». In: *Elements of Reusable Object-Oriented Software Addison-Wesley, Reading* (1995).

[14] Qingkai Kong et al. «Structural health monitoring of buildings using smartphone sensors». In: *Seismological Research Letters* 89.2A (2018), pp. 594–602.

[15] Sotiris B. Kotsiantis. «Supervised Machine Learning: A Review of Classification Techniques». In: *Informatica (Slovenia)* 31.3 (2007), pp. 249–268.

[16] Zahra Ansari Lari e Amir Golroo. «Automated transportation mode detection using smart phone applications via machine learning: Case study mega city of Tehran». In: *Proceedings of the Transportation Research Board 94th Annual Meeting, Washington, DC, USA*. 2015, pp. 11–15.

[17] Hongfei Li et al. «Improving rail network velocity: A machine learning approach to predictive maintenance». In: *Transportation Research Part C: Emerging Technologies* 45 (2014), pp. 17–26.

[18] Xiaosheng Li e Jessica Lin. «Linear Time Complexity Time Series Classification with Bag-of-Pattern-Features». In: *2017 IEEE International Conference on Data Mining, ICDM 2017, New Orleans, LA, USA, November 18-21, 2017*. A cura di Vijay Raghavan et al. IEEE Computer Society, 2017, pp. 277–286.

[19] Federico Montori, Luca Bedogni e Luciano Bononi. «A Collaborative Internet of Things Architecture for Smart Cities and Environmental Monitoring». In: *IEEE Internet of Things Journal* 5.2 (2018), pp. 592–605.

[20] Federico Montori et al. «Classification and Annotation of Open Internet of Things Datastreams». In: *Web Information Systems Engineering - WISE 2018 - 19th International Conference, Dubai, United Arab Emirates, November 12-15, 2018, Proceedings, Part II*. A cura di Hakim Hacid et al. Vol. 11234. Lecture Notes in Computer Science. Springer, 2018, pp. 209–224.

[21] Selwyn Piramuthu. «Knowledge-based framework for automated dynamic supply chain configuration». In: *European Journal of Operational Research* 165.1 (2005), pp. 219–230.

[22] Juliana T Pollettini et al. «Using machine learning classifiers to assist healthcare-related decisions: classification of electronic patient records». In: *Journal of medical systems* 36.6 (2012), pp. 3861–3874.

[23] Nishkam Ravi et al. «Activity recognition from accelerometer data». In: *Aaai*. Vol. 5. 2005. 2005, pp. 1541–1546.

[24] Lior Rokach. «Ensemble-based classifiers». In: *Artif. Intell. Rev.* 33.1-2 (2010), pp. 1–39.

[25] Stan Salvador e Philip Chan. «Toward accurate dynamic time warping in linear time and space». In: *Intell. Data Anal.* 11.5 (2007), pp. 561–580.

[26] Patrick Schäfer. «The BOSS is concerned with time series classification in the presence of noise». In: *Data Min. Knowl. Discov.* 29.6 (2015), pp. 1505–1530.

[27] Navin Sharma et al. «Predicting solar generation from weather forecasts using machine learning». In: *2011 IEEE international conference on smart grid communications (SmartGridComm)*. IEEE. 2011, pp. 528–533.

[28] Eugene Siow et al. «TritanDB: time-series rapid Internet of Things analytics». In: *arXiv preprint arXiv:1801.07947* (2018).

[29] Lexiang Ye e Eamonn J. Keogh. «Time series shapelets: a new primitive for data mining». In: *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Paris, France, June 28 - July 1, 2009*. A cura di John F. Elder IV et al. ACM, 2009, pp. 947–956.

[30] Yanxu Zheng, Sutharshan Rajasegarar e Christopher Leckie. «Parking availability prediction for sensor-enabled car parks in smart cities». In: *2015 IEEE Tenth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*. IEEE. 2015, pp. 1–6.