Progetto di Laboratorio Computazionale: Complessi Simpliciali Filtrati

Valeria Tateo

Novembre 2024

Indice

1 Introduzione

L'obiettivo del progetto è comprendere l'impatto della scelta del complesso simpliciale filtrato come traduzione di un oggetto in esame. Nella prima parte del progetto richiamiamo dei concetti di omologia simpliciale e di omologia persistente e descriviamo diversi complessi simpliciali filtrati e l'algoritmo che permette di calcolare la persistenza omologica di un complesso simpliciale filtrato. Successivamente utilizziamo Python e la libreria GUDHI per introdurre i simplex tree e per la costruzione di complessi simpliciali partendo da point cloud, matrici delle distanze e insiemi di sottolivello di una funzione. Infine facciamo sperimentazione sui dei dati presi da SHREC2021: prese 18 configurazioni proteiche proviamo a clusterizzarle utilizzando le strategie viste nell'elaborato e calcoliamo l'accuratezza della classificazione e il tempo impiegato.

2 Omologia

Supponiamo di avere un insieme di dati X che appartengono ad uno spazio metrico, come un sottoinsieme dello spazio Euclideo dotato di una distanza. L'omologia associa uno spazio vettoriale $H_i(X)$ allo spazio X per ogni numero naturale $i \in \{0, 1, ...\}$. La dimensione di $H_0(X)$ conta il numero di componenti connesse in X, la dimensione di $H_1(X)$ conta il numero di buchi e la dimensione di $H_2(X)$ conta il numero di buchi tridimensionali. Queste strutture algebriche sono invarianti per omotopia. Sebbene esistano teorie omologiche per spazi topologici arbitrari, lo studio dell'omologia risulta spesso complesso e difficile dal punto di vista computazionale. Per questo motivo, è utile considerare strutture combinatorie dette complessi simpliciali che sono in grado di fornire una rappresentazione accurata degli spazi di interesse.

2.1 Complessi simpliciali

Definizione 1

Sia K_0 un insieme finito non vuoto di elementi, detti vertici. Un **complesso simpliciale** è una collezione K di sottoinsiemi di K_0 tale che

- per ogni $v \in K_0$, si ha $\{v\} \in K$;
- se $\tau \in K$ e $\sigma \subset \tau$ allora $\sigma \in K$.

Ogni elemento di K è detto simplesso.

Diciamo che un simplesso ha **dimensione** p se ha cardinalità $p+1$ e indichiamo con K_p l'insieme dei p-simplessi. Il k**-scheletro** di K è l'unione di tutti gli insiemi K_p per ogni $p \in \{1, ..., k\}$.

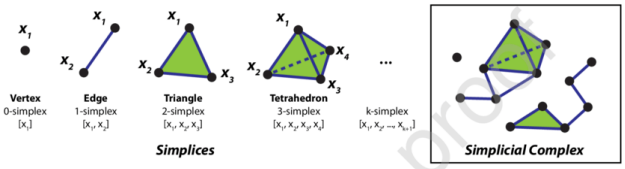


Fig 1.1: Esempio di complesso simpliciale

Definizione 2
Dati due simplessi σ e τ di un complesso simpliciale K, σ è una faccia di τ se ogni vertice di σ è anche un vertice di τ.

Diciamo che σ è una faccia di τ di codimensione k' se k' =dimτ - dimσ.

Definizione 3
Dato un complesso simpliciale K, definiamo la dimensione di K come

$$\dim(K) = \max\{\dim(\sigma) \mid \sigma \in K\}.$$

Definizione 4
Una mappa tra complessi simpliciali $f : K \to L$ è una mappa tale che $f(\sigma) \in L$ per ogni $\sigma \in K$.

Definiamo adesso l'omologia per complessi simpliciali. Sia $\mathbb{F}_2$ il campo con due elementi. Dato un complesso simpliciale K, sia $C_p(K)$ uno spazio vettoriale su $\mathbb{F}_2$ con base data dai $p-$simplessi di K. Per ogni $p \in \{1, 2, ...\}$, definiamo la seguente mappa lineare:

$$d_p : C_p(K) \to C_p(K), \qquad \sigma \mapsto \textstyle\sum_{\tau \subset \sigma,\ \tau \in K_{p-1}} \tau.$$

Per $p = 0$, d_0 è la mappa nulla. In altre parole, d_p manda i $p-$simplessi nel loro bordo, ovvero nella somma delle loro facce di codimensione 1. Osserviamo che per ogni $p \in \{0, 1, 2, ...\}$ abbiamo che $d_p \circ d_{p+1} = 0$, dunque l'immagine di d_{p+1} è contenuta nel nucleo di d_p.

Definizione 5
Per ogni $p \in \{0, 1, 2, ...\}$ definiamo il **p-esimo gruppo di omologia** di un complesso simpliciale K come il quoziente:

$$H_p(K) = \ker(d_p)/\mathrm{Im}(d_{p+1}).$$

La sua dimensione è

$$\beta_p(K) := \dim H_p(K) = \dim \ker(d_p) \text{ - } \dim \mathrm{Im}\ (d_{p+1})$$

ed è chiamato **p-esimo numero di Betti** di K.
Intuitivamente, i $p-$cicli che non sono bordi rappresentano buchi $p-$dimensionali. Dunque, il $p-$esimo numero di Betti conta il numero di $p-$buchi. Inoltre se K è un complesso simpliciale di dimensione n, allora per ogni $p > n$ si ha che $H_p(K) = 0$ e $C_p(K) = 0$. Otteniamo dunque la seguente successione di spazi vettoriali e mappe lineari:

$$0 \xrightarrow{d_{n+1}} C_n(K) \xrightarrow{d_n} ... \xrightarrow{d_1} C_0(K) \xrightarrow{d_0} 0.$$

Una delle proprietà più importanti dell'omologia simpliciale è la **funtorialità**. Una qualsiasi mappa $f : K \to K'$ tra complessi simpliciali induce la seguente mappa $\mathbb{F}_2-$lineare:

$$\tilde{f}_p : C_p(K) \to C_p(K'), \quad \textstyle\sum_{\sigma \in K_p} c_\sigma \sigma \mapsto \sum_{\sigma \in K_p\ :\ f(\sigma) \in K'_p} c_\sigma f(\sigma)$$

per ogni $p \in \{0, 1, 2, ...\}$, dove $c_\sigma \in \mathbb{F}_2$. Inoltre, $\tilde{f}_p \circ d_{p+1} = d'_{p+1} \circ \tilde{f}_{p+1}$, e la mappa $\tilde{f}_p$ induce la seguente mappa lineare tra spazi vettoriali di omologia:

$$f_p : H_p(K) \to H_p(K'), \quad [c] \mapsto [\tilde{f}_p(c)].$$

Di conseguenza, ad ogni mappa $f : K \to K'$ tra complessi simpliciali, assegniamo la mappa $f_p : H_p(K) \to H_p(K')$ per ogni $p \in \{0, 1, 2, ...\}$. Questo assegnamento ha l'importante proprietà, chiamata funtorialità, di essere compatibile con la composizione. Cioè date le mappe tra complessi simpliciali $f : K \to K'$ e $g : K' \to K''$, la mappa $(g \circ f)_p : H_p(K) \to H_p(K'')$ è equivalente a $g_p \circ f_p$.

Osservazione 1
Quando lavoriamo con complessi simpliciali, possiamo modificare un complesso simpliciale rimuovendo o aggiungendo una coppia di simplessi (σ, τ), dove τ è una faccia di σ di codimensione 1 e σ è l'unico simplesso che ha τ come faccia. Il complesso che ne risulta ha la stessa omologia del precedente. Questo procedimento è chiamato **collasso simpliciale elementare**.

2.2 Il complesso di Čech

Non è semplice studiare l'omologia di uno spazio arbitrario X, quindi proviamo a cercare un complesso simpliciale la cui omologia approssima l'omologia di X in modo opportuno. Uno strumento importante è il **complesso di Čech** $C(X)$. Sia $\{U_i\}$ un ricoprimento di X. I $k-$simplessi del complesso di Čech sono le intersezioni non vuote di $k+1$ insiemi del ricoprimento $\{U_i\}$.

Definizione 6
Sia $\{U_i\}_{i \in I}$ una collezione di insiemi non vuoti. Il **nervo** di $\{U_i\}_{i \in I}$ è il complesso simpliciale che ha come insieme dei vertici I e $k-$simplessi dati da $\{i_0, ..., i_k\}$ se e solo se $U_{i_0} \cap ... \cap U_{i_k} \neq \emptyset$.

Se il ricoprimento è sufficientemente 'bello', il Teorema del Nervo ci dice che il nervo del ricoprimento e lo spazio di partenza hanno la stessa omologia. Infatti:

Definizione 7
Sia K un complesso simpliciale e $\sigma = [x_1, ..., x_k] \in K$ un simplesso generato dai vertici $x_1, ..., x_k \in K$. Il **baricentro** di σ è definito come $b_\Delta(\sigma) = \frac{1}{k} \sum_{i=1}^{k} x_i$. La **suddivisione baricentrica** $Sd(K)$ è la decomposizione di $[x_1, ..., x_k]$ in n simplessi $[b_\Delta, w_0, ..., w_{n-1}]$ dove induttivamente $[w_0, ..., w_{n-1}]$ è un simplesso nella suddivisione baricentrica della faccia $[x_0, ..., x_{i-1}, x_{i+1}, ..., x_n]$.

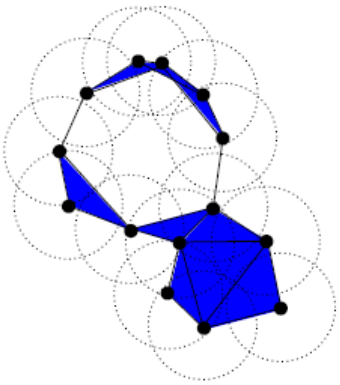
Teorema 8
Sia $X \subset \mathbb{R}^n$ uno spazio topologico e sia $U = \{U_i\}$ un ricoprimento chiuso e convesso di X. La mappa $\Gamma : X \to |Sd(N(U))|$ è un'equivalenza omotopica. In particolare X è omotopicamente equivalente a $N(U)$, dove $N(U)$ rappresenta il nervo del ricoprimento U e $Sd(N(U))$ è la suddivisione baricentrica di $N(U)$.

Una dimostrazione del Teorema del Nervo si trova nel paper di Bauer 'A Unified View on the Functorial Nerve Theorem and its Variations' [1].
Da un punto di vista computazionale, il complesso di Čech è molto costoso perché bisogna controllare le intersezioni di un numero molto grande di insiemi. Inoltre, nel caso peggiore, il compresso di

Čech ha dimensione $|\{U_i\}| - 1$. Idealmente, vorremmo costruire complessi simpliciali che approssimino l'omologia dello spazio di partenza, ma che siano semplici da calcolare e con pochi simplessi di grandi dimensioni.

3 Omologia persistente

Supponiamo di avere dei dati sperimentali sotto forma di uno spazio metrico finito S; i punti di S rappresentano delle misurazioni. Consideriamo una funzione distanza sui punti di S, che ad esempio studia la correlazione o la dissimilarità. É utile pensare l'insieme S come un campione proveniente da un qualche spazio topologico sottostante. Il nostro obiettivo è recuperare le proprietà di tale spazio sottostante in modo robusto rispetto a piccole perturbazioni nei dati di S. In senso ampio, questo è l'oggetto dell'**inferenza topologica**. Se S è un sottoinsieme dello spazio Euclideo, si può considerare un 'ispessimento' S_ϵ di S dato dall'unione di palle di raggio ϵ attorno ai suoi punti e quindi calcolare il complesso di Čech.



Si può dunque cercare di calcolare le caratteristiche qualitative dell'insieme di dati S costruendo il complesso di Čech per un valore scelto di ϵ e successivamente calcolandone l'omologia simpliciale. Il problema di questo approccio è che, a priori, non esiste una scelta chiara per il valore del parametro ϵ. L'intuizione chiave dell'omologia persistente (PH) è la seguente: per estrarre informazioni qualitative dai dati, si considerano diversi (o persino tutti) i possibili valori del parametro ϵ. L'omologia persistente cattura quindi come l'omologia dei complessi cambia al variare del valore del parametro e rileva quali caratteristiche 'persistono'.

3.1 Complessi simpliciali filtrati

Definizione 9
Sia K un complesso simpliciale finito, e sia $K_1 \subset K_2 \subset ... \subset K_l = K$ una successione finita di sottocomplessi di K annidati, detta filtrazione. Il complesso simpliciale K dotato di questa successione di sottocomplessi si dice **complesso simpliciale filtrato**.

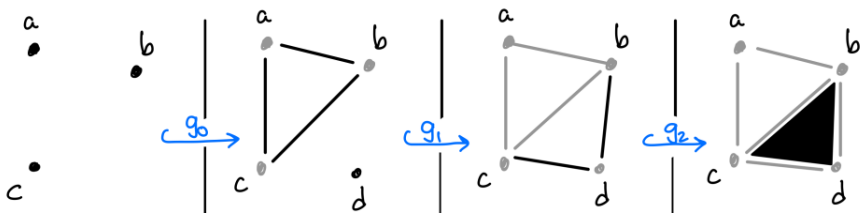


Fig 1.3: Esempio di filtrazione

Possiamo applicare l'omologia a ogni sottocomplesso. Per ogni p, la mappa di inclusione $K_i \to K_j$ induce una mappa $\mathbb{F}_2$-lineare $f_{i,j} : H_p(K_i) \to H_p(K_j)$ per ogni $i, j \in 1, ..., l$ con $i \leq j$. Usando la funtorialità, segue che $f_{k,j} \circ f_{i,k} = f_{i,j}$ per ogni $i \leq k \leq j$.

Definizione 10
Sia $K_1 \subset K_2 \subset ... \subset K_l = K$ un complesso simpliciale filtrato. La $p-$**esima omologia persistente di** K è la coppia $(\{H_p(K_i)\}_{1 \leq i \leq l}, \{f_{i,j}\}_{1 \leq i \leq j \leq l})$, dove per ogni $i, j \in \{1, ..., l\}$ con $i \leq j$ le mappe lineari $f_{i,j} : H_p(K_i) \to H_p(K_j)$ sono quelle indotte dalle mappe di inclusione $K_i \to K_j$.

La $p-$esima omologia persistente di un complesso simpliciale filtrato fornisce informazioni più raffinate rispetto alla semplice omologia dei singoli sottocomplessi. Possiamo visualizzare le informazioni fornite dagli spazi vettoriali $H_p(K_i)$ insieme alle mappe lineari $f_{i,j}$ tracciando il seguente diagramma: al passo di filtrazione i, disegniamo tanti punti quanto è la dimensione dello spazio vettoriale $H_p(K_i)$ corrispondente. Poi colleghiamo i punti nel seguente modo: tracciamo un intervallo tra il punto u al passo di filtrazione i e il punto v al passo di filtrazione $i+1$ se il generatore di u viene mandato nel generatore di v. Se invece il generatore corrispondente a un punto u al passo di filtrazione i viene mandato a 0, disegniamo un intervallo che parte da u e termina in un punto finale, che rappresenta la morte del generatore.
Un tale diagramma dipende chiaramente dalla scelta della base per gli spazi vettoriali, e una scelta non adeguata può portare a disordine. Fortunatamente, grazie al Teorema Fondamentale dell'Omologia Persistente, esiste una scelta di vettori di base per ciascuno spazio tale che si possa costruire il diagramma come una collezione ben definita e unica di intervalli semiaperti disgiunti, chiamati **barcode**.

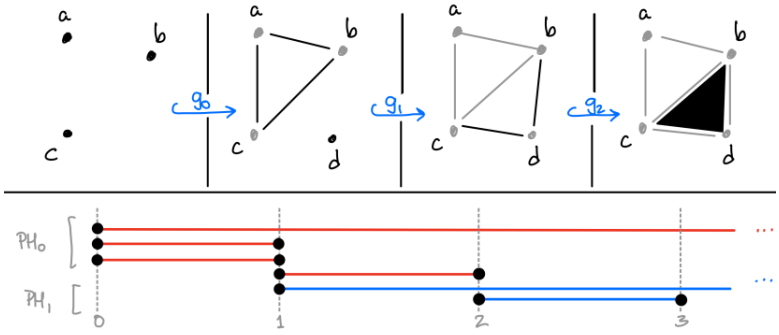


Fig 1.4: Barcode

Vediamo nello specifico l'algoritmo che ci permette di passare da complessi simpliciali filtrati a barcodes.

3.2 Dai complessi simpliciali filtrati ai barcodes

Per calcolare la persistenza omologica (PH) di un complesso simpliciale filtrato K e ottenere un barcode come quello illustrato nella Figura 1.4, è necessario associare ad esso una matrice, chiamata **matrice dei bordi** (*boundary matrix*), che memorizza informazioni sulle facce di ogni simplesso. Per farlo, bisogna imporre un ordinamento totale sui simplessi del complesso, che sia compatibile con la filtrazione, nel seguente senso:

- Una faccia di un simplesso precede il simplesso stesso;
- Un simplesso nel complesso K_i precede i simplessi in K_j per $i < j$, che non appartengono a K_i.

Sia n il numero totale di simplessi nel complesso, e sia $\sigma_1, \sigma_2, \ldots, \sigma_n$ l'insieme dei simplessi secondo questo ordinamento. Costruiamo una matrice quadrata δ di dimensione $n \times n$ memorizzando un 1 in $\delta(i, j)$ se il simplesso σ_i è una faccia di codimensione 1 del simplesso σ_j; altrimenti, inseriamo uno 0 in $\delta(i, j)$. Una volta costruita la matrice dei bordi, bisogna ridurla utilizzando l'eliminazione gaussiana.

Il cosiddetto **algoritmo standard** per il calcolo della persistenza omologica (PH) è stato introdotto per il campo $\mathbb{F}_2$ e per campi generali. Per ogni j, definiamo low(j) come il valore dell'indice più grande i tale che $\delta(i, j)$ sia diverso da 0. Se la colonna j contiene solo valori 0, allora il valore di low(j) non è definito. Si dice che la matrice dei bordi è ridotta se la mappa low è iniettiva sul suo dominio di definizione. Nell'Algoritmo 1, illustriamo l'algoritmo standard per la riduzione della matrice dei bordi. Poiché questo algoritmo opera sulle colonne della matrice da sinistra a destra, è talvolta chiamato anche 'algoritmo delle colonne'. Nel caso peggiore, la complessità dell'algoritmo standard è cubica rispetto al numero di simplessi. Una volta ridotta la

Algorithm 1

```
1: for j = 1 to n do
2:     while esiste i < j tale che low(i) = low(j) do
3:         Aggiungi la colonna i alla colonna j
4:     end while
5: end for
```

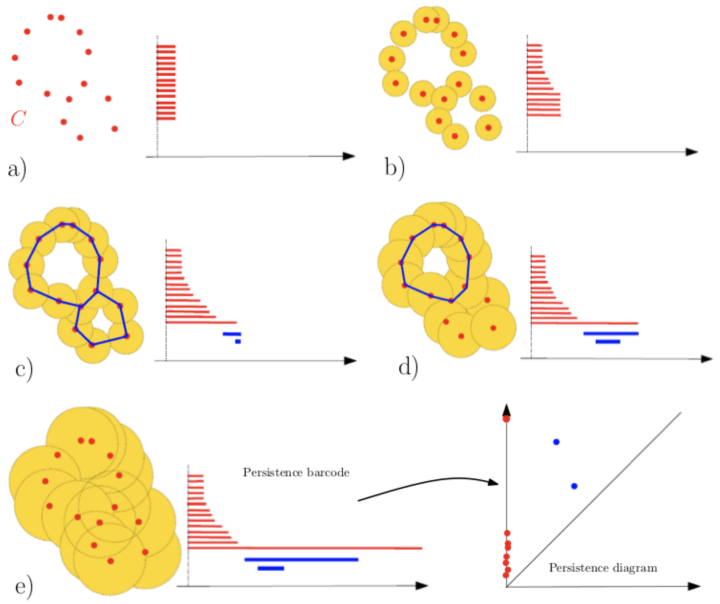
matrice dei bordi, si possono leggere gli intervalli del barcode accoppiando i simplessi nel modo seguente:

- Se $low(i) = j$ allora il simplesso σ_j è accoppiato con σ_i, e l'ingresso di σ_j nella filtrazione provoca la nascita di una caratteristica che muore con l'ingresso di σ_i.
- Se $low(i)$ non è definito, allora l'ingresso del simplesso σ_i nella filtrazione provoca la nascita di una caratteristica. Se esiste k tale che $low(k) = i$ allora σ_i è accoppiato con il simplesso σ_k, il cui ingresso nella filtrazione provoca la morte della caratteristica. Se non esiste tale k, allora σ_i non è accoppiato.

Una coppia (σ_j, σ_i) fornisce l'intervallo semi-aperto $[j, i)$ nel barcode, dove per un simplesso σ_i definiamo i come il numero più piccolo l tale che $low(l) = i$. Un simplesso σ_j non accoppiato fornisce l'intervallo infinito $[j, \infty)$.

3.3 Dai barcodes ai diagrammi di persistenza

Con il seguente esempio studiamo la costruzione dei diagrammi di persistenza, partendo da un barcode.



a) Per il raggio $r = 0$, l'unione delle sfere è ridotta all'insieme finito iniziale di punti, ciascuno dei quali corrisponde a una caratteristica 0-dimensionale, cioè una componente connessa; per ciascuna di queste caratteristiche viene creato un intervallo con nascita a $r = 0$.

b) Alcune delle sfere iniziano a sovrapporsi, causando la morte di alcune componenti connesse che si uniscono; il diagramma di persistenza tiene traccia di queste morti, segnando un punto finale negli intervalli corrispondenti man mano che scompaiono.

c) Nuove componenti si sono unite dando origine a una singola componente connessa e, quindi, tutti gli intervalli associati a una caratteristica 0-dimensionale sono stati conclusi, eccetto quello corrispondente alla componente rimanente; sono apparse due nuove caratteristiche 1-dimensionali, risultando in due nuovi intervalli (in blu) che iniziano alla scala di nascita.

d) Uno dei due cicli 1-dimensionali è stato riempito, risultando nella sua morte nella filtrazione e nella fine dell'intervallo blu corrispondente.

e) Tutte le caratteristiche 1-dimensionali sono morte, rimane solo l'intervallo lungo (e mai morente) in rosso.

Il barcode finale può essere rappresentato in modo equivalente come diagramma di persistenza, in cui ogni intervallo (a, b) è rappresentato dal punto di coordinate (a, b).

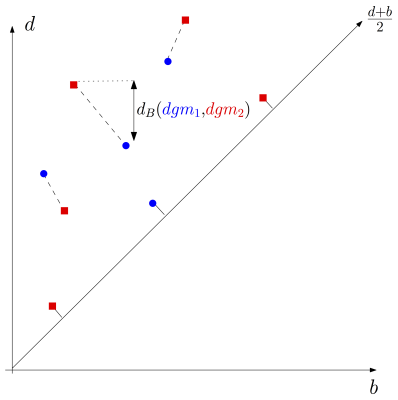
Intuitivamente, più lungo è un intervallo nel barcode o, equivalentemente, più lontano dalla diagonale è il punto corrispondente nel diagramma, più persistente, e quindi rilevante, è la caratteristica omologica corrispondente lungo la filtrazione.

3.4 Distanza di Bottleneck

Per sfruttare le informazioni topologiche e le caratteristiche topologiche derivate dalla omologia persistente, è necessario essere in grado di confrontare i diagrammi di persistenza.
Vediamo un diagramma di persistenza come l'unione dei suoi punti e della diagonale, dove i punti della diagonale sono contati con molteplicità infinita.
Una corrispondenza tra due diagrammi dgm_1 e dgm_2 è un sottoinsieme $M \subset dgm_1 \times dgm_2$ tale che ogni punto in $D_1 \setminus \Delta$ e $D_2 \setminus \Delta$ appare esattamente una volta in M.

Definizione 11
La distanza di Bottleneck tra due diagrammi di persistenza dgm_1 e dgm_2 è definita come:

$$d_b(dgm_1, dgm_2) = \inf_{matching\ m} \max_{(p,q)\in m} ||p-q||_\infty.$$



4 Esempi di complessi simpliciali filtrati

In questa sezione introduciamo altri complessi simpliciali filtati, come abbiamo fatto per il complesso simpliciale di Čech. Da qui in avanti denotiamo con (X, d) uno spazio metrico e supponiamo che S sia uno sottoinsieme di X. Nelle applicazioni, S è una collezione di misurazioni dotato di una distanza. Il nostro obiettivo è calcolare l'omologia persistente di una successione di spazi annidati $S_{\epsilon_1}, ..., S_{\epsilon_l}$, dove ogni spazio è un ispessimento di S in X.

4.1 Il complesso di Vietoris-Rips

Abbiamo visto che uno degli svantaggi del complesso di Čech è che occorre verificare un gran numero di intersezioni. Per aggirare questo problema, si può invece considerare il complesso di Vietoris-Rips (VR), che approssima il complesso di Čech. Per un numero reale non negativo ϵ, il complesso di Vietoris–Rips $VR_\epsilon(S)$ a scala ϵ è definito come segue:

$$VR_\epsilon(S) = \{\sigma \subset S | d(x, y) \leq 2\epsilon \text{ per ogni } x, y \in \sigma\}.$$

Il modo in cui il complesso di Vietoris–Rips approssima il complesso di Čech è il seguente: quando S è un sottoinsieme di uno spazio Euclideo, si ha

$$C_\epsilon(S) \subset VR_\epsilon(S) \subset C_{\sqrt{2}\epsilon}(S).$$

Decidere se un sottoinsieme $\sigma \subset S$ è contenuto in $VR_\epsilon(S)$ equivale a verificare se la distanza massima tra ogni coppia di vertici in σ è al massimo 2ϵ. Pertanto, la costruzione del complesso di Vietoris–Rips può essere effettuata in due fasi:

- Prima si calcola il grafo degli ϵ-vicini di S. Questo è il grafo i cui vertici sono tutti i punti di S e i cui archi sono $\{(i,j) \in S \times S | i \neq j,\ d(i,j) \leq 2\epsilon\}$.
- Successivamente, si ottiene il complesso di Vietoris–Rips calcolando il **complesso dei cliques** del grafo degli $\epsilon-$vicini. Il complesso dei cliques di un grafo è un complesso simpliciale definito nel modo seguente: un sottoinsieme σ è un $k-$simplesso se e solo se ogni coppia di vertici in σ è connessa da un arco. Una tale collezione di vertici è chiamata **clique**. Questa costruzione rende molto semplice il calcolo del complesso di Vietoris–Rips, perché per costruire il complesso dei cliques basta verificare le distanze tra le coppie di punti.

Sfortunatamente, il complesso di Vietoris–Rips ha la stessa complessità nel caso peggiore del complesso di Čech. Nel caso peggiore, può avere fino a $2^{|S|} - 1$ simplessi e dimensione $|S| - 1$.

4.2 Il complesso di Delaunay

Per evitare i problemi computazionali associati ai complessi di Čech e Vietoris–Rips, è necessario un modo per limitare il numero di simplessi di alte dimensioni. Il **complesso di Delaunay** fornisce uno strumento geometrico per raggiungere questo obiettivo, e la maggior parte dei nuovi complessi simpliciali introdotti per lo studio dei dati si basa su variazioni del complesso di Delaunay. Il complesso di Delaunay e il suo duale, il **diagramma di Voronoi**, sono oggetti centrali nello studio della geometria computazionale perché possiedono molte proprietà utili.
Nel complesso di Delaunay, generalmente si considera $X = \mathbb{R}^d$. Per ogni $s \in S$, definiamo

$$V_s = \{x \in \mathbb{R}^d \mid d(x,s) \leq d(x,s') \text{ per ogni } s' \in S\}.$$

La collezione di insiemi V_s è un ricoprimento di $\mathbb{R}^d$ chiamata **decomposizione di Voronoi** di $\mathbb{R}^d$ rispetto a S, e il **nervo** di questo ricoprimento è chiamato **complesso di Delaunay** di S, denotato da $Del(S, \mathbb{R}^d)$. In generale, il complesso di Delaunay non ha una realizzazione geometrica in $\mathbb{R}^d$. Tuttavia, se i punti di S sono 'in posizione generale', allora il complesso di Delaunay ha una realizzazione geometrica in $\mathbb{R}^d$ che fornisce una **triangolazione** dell'inviluppo convesso di S. In questo caso, il complesso di Delaunay è anche chiamato **triangolazione di Delaunay**.
La complessità del complesso di Delaunay dipende dalla dimensione d dello spazio. Per $d \leq 2$, i migliori algoritmi hanno una complessità di $O(NlogN)$, dove N è la cardinalità di S. Per $d \geq 3$, la complessità è $O(N^{\lceil d/2 \rceil})$. Quindi, la costruzione del complesso di Delaunay diventa costosa in dimensioni elevate, anche se esistono algoritmi efficienti per il calcolo del complesso di Delaunay per $d = 2$ e $d = 3$.

4.3 Il complesso Alpha

Continuiamo ad assumere che S sia un insieme finito di punti in $\mathbb{R}^d$. Usando la decomposizione di Voronoi, si può definire un complesso simpliciale simile al complesso di Čech, ma con la proprietà desiderata che (se i punti S sono in posizione generale) la sua dimensione è al massimo quella dello spazio. Sia $\epsilon > 0$, e sia $S_\epsilon = \bigcup_{s \in S} B(s, \epsilon)$. Per ogni $s \in S$, consideriamo l'intersezione $V_s \cap B(s, \epsilon)$. La collezione di questi insiemi forma un ricoprimento di S_ϵ. Il **nervo** di questo ricoprimento è chiamato **complesso Alpha (α)** di S alla scala ϵ, denotato da $A_\epsilon(S)$. Il Teorema del Nervo si applica, il che implica che $A_\epsilon(S)$ ha la stessa omologia di S_ϵ. Inoltre, $A_\infty(S)$ è il complesso di Delaunay; per $\epsilon < \infty$, il complesso Alpha è un sottocomplesso del complesso di Delaunay.

4.4 Il complesso cubicale

Il complesso cubicale è un esempio di complesso strutturato utile in matematica computazionale e nell'analisi delle immagini.
Un intervallo elementare è un intervallo della forma $[a, b]$, oppure $[a, a]$, con $a, b \in \mathbb{R}$. Il primo è chiamato non degenere, mentre il secondo è un intervallo degenere. Il bordo di un intervallo elementare è una catena $[a] - [b]$ nel caso di intervallo elementare non degenere e $[a]$ nel caso di intervallo elementare degenere.
Un cubo elementare C è il prodotto di intervalli elementari, $C = I_1 \times I_2 \times \cdots \times I_n$.
La dimensione di immersione di un cubo è n, ossia il numero di intervalli elementari (degenere o non degenere) nel prodotto. La dimensione di un cubo C è il numero di intervalli elementari non degeneri nel prodotto.
Il bordo di un cubo C è una catena ottenuta nel modo seguente:

$$\delta C = (\delta I_1 \times ... \times I_N) + ...(I_1 \times ... \times \delta I_k \times ... \times I_N)... + (I_1 \times ... \times \delta I_n)$$

(Quando si lavora con un campo di caratteristica diversa da 2, i termini non nulli di questa somma sono associati a segni alternati ± 1).
Un complesso cubicale X è una collezione di cubi chiusa rispetto all'operazione di prendere il bordo (ossia, il bordo di ogni cubo della collezione appartiene alla collezione stessa). Un cubo C nel complesso cubicale X è massimale se non è bordo di nessun altro cubo in X. Il supporto di un cubo C è l'insieme in $\mathbb{R}^n$ occupato da C (dove n è la dimensione di immersione di C).

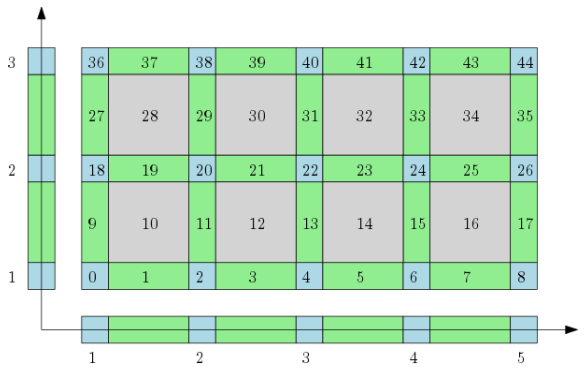


Fig 1.7: Complesso cubicale

4.5 Tecniche di riduzione

Fino ad ora, abbiamo discusso delle tecniche per costruire complessi simpliciali con un numero possibilmente 'ridotto' di simplessi. Si può anche adottare un approccio alternativo per velocizzare il calcolo della topologia dei dati (PH). Ad esempio, si può utilizzare un'euristica per ridurre le dimensioni di un complesso filtrato, mantenendo invariato il PH.
Per i complessi simpliciali, uno di questi metodi è basato sulla teoria di Morse discreta, che è stata adattata alle filtrazioni dei complessi simpliciali. L'idea di base dell'algoritmo sviluppato è che si può calcolare un accoppiamento parziale dei simplessi in un complesso simpliciale filtrato in modo tale che:

- le coppie si verificano solo tra simplessi che entrano nella filtrazione allo stesso passo;
- i simplessi non accoppiati determinano l'omologia;
- si possono rimuovere i simplessi accoppiati dal complesso filtrato senza alterare il PH.

Tali cancellazioni sono esempi dei collassi simpliciali elementari di cui abbiamo parlato precedentemente. Sfortunatamente, il problema di trovare un accoppiamento parziale ottimale è stato dimostrato essere NP-completo, e quindi ci si affida a euristiche per trovare accoppiamenti parziali per ridurre le dimensioni del complesso.
Una particolare famiglia di collassi elementari è chiamata collassi forti. I collassi forti preservano i cicli di lunghezza minima nella classe rappresentativa di un generatore di un buco; questa caratteristica rende i collassi forti utili per trovare buchi nelle reti. Una versione distribuita dell'algoritmo è stata adattata per il calcolo del PH.

5 TDA con Python e la libreria GUDHI: Simplex tree

GUDHI è una libreria che implementa in C++ e Python la costruzione di complessi simpliciali. In particolare, i complessi simpliciali filtrati sono codificati attraverso una struttura dati chiamata **simplex tree**.

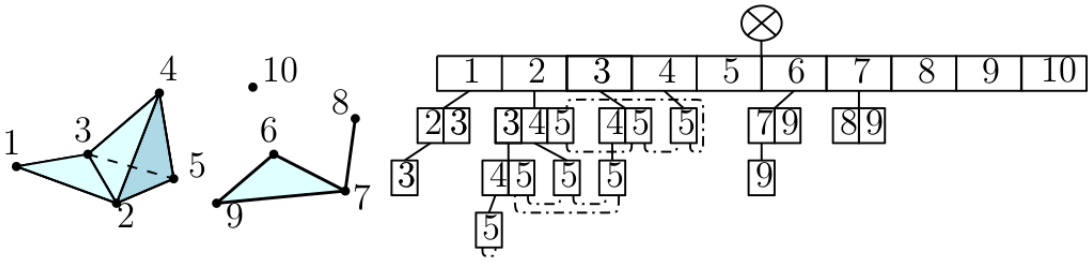


Fig 1.8: Un complesso simpliciale di 10 vertici e il suo simplex tree. Il nodo più profondo rappresenta il tetraedro del complesso. Tutte le posizioni di una data etichetta ad una data profondità sono collegate in un elenco, come illustrato nel caso dell'etichetta 5.

In questa sezione studiamo l'uso dei simplex tree per rappresentare dei complessi simpliciali a partire da punti dati.

Creiamo il nostro primo complesso simpliciale rappresentato da un `simplex tree` utilizzando dei moduli GUDHI.
Input:

```
st = gd.SimplexTree()
```

Per ora, `st` è un `simplex tree` vuoto.
Il metodo `insert()` può essere utilizzato per inserire simplessi nel `simplex tree`. In un `simplex tree`:

- i vertici sono rappresentati con interi;
- gli archi sono rappresentati con una lista di lunghezza 2 di interi (corrispondenti ai due vertici coinvolti nell'arco);
- i triangoli sono rappresentati con una lista di lunghezza 3 di interi (corrispondenti ai tre vertici coinvolti nel triangolo);
- ecc.

Ad esempio, la seguente porzione di codice inserisce tre archi nel `simplex tree`.
Input:

```
st.insert([0, 1])
st.insert([1, 2])
st.insert([3, 1])
```

Output:

```
True
```

Quando un simplesso viene inserito con successo nel `simplex tree`, il metodo `insert()` restituisce `True`. Al contrario, se il simplesso è già presente nella filtrazione, il metodo `insert()` restituisce `False`.
Con il metodo `get_filtration()`, otteniamo la lista di tutti i simplessi del `simplex tree`.
Input:

```
st_gen = st.get_filtration()
```

L'output `st_gen` è un generatore e quindi possiamo iterare sui suoi elementi. Ogni elemento della lista contiene un simplesso e il suo valore di filtrazione.
Input:

```
for splx in st_gen :
    print(splx)
```

In questo modo otteniamo una filtrazione del complesso simpliciale `st` costruito sopra.
Output:

```
([0], 0.0)
([1], 0.0)
([0, 1], 0.0)
([2], 0.0)
([1, 2], 0.0)
([3], 0.0)
([1, 3], 0.0)
```

Intuitivamente, il valore di filtrazione di un simplesso in un complesso filtrato agisce come un timestamp che corrisponde a 'quando' il simplesso appare nella filtrazione. Per impostazione predefinita, il metodo `insert()` assegna un valore di filtrazione uguale a 0. Si noti che l'inserimento di un arco comporta anche l'inserimento automatico dei suoi vertici (se non erano già presenti nel complesso) per soddisfare la proprietà di inclusione di un complesso filtrato.

5.1 Descrizione del Simplex Tree

Vediamo alcuni metodi di GUDHI che ci permettono di studiare i complessi simpliciali, in particolare lavoriamo sul complesso simpliciale costruito nella sottosezione precedente.
Il metodo `dimension()` del simplex tree permette di ottenere la dimensione di un complesso simpliciale.

Input:

```
st.dimension()
```

Output:

```
1
```

È possibile calcolare il numero di vertici in un simplex tree tramite il metodo `num_vertices()`.

Input:

```
st.num_vertices()
```

Output:

```
4
```

Otteniamo il numero di simplessi nel simplex tree tramite il metodo `num_simplices()`.

Input:

```
st.num_simplices()
```

Output:

```
7
```

Inoltre, il k-skeletro può essere calcolato con il metodo `get_skeleton()`. Questo metodo prende come argomento la dimensione dello scheletro desiderato. Per recuperare il grafo topologico da un simplex tree, possiamo quindi chiamare:

Input:

```
print(st.get_skeleton(1))
```

Output:

```
[([0, 1], 0.0), ([0], 0.0), ([1, 2], 0.0), ([1, 3], 0.0), ([1], 0.0), ([2], 0.0), ([3], 0.0)]
```

Si può anche verificare se un simplesso è già presente nella filtrazione. Questo si ottiene con il metodo `find()`.

Input:

```
st.find([2, 4])
```

Output:

```
False
```

5.2 Valori di Filtrazione

Lavorando sempre sul complesso `st`, possiamo inserire dei simplessi con un determinato valore di filtrazione. Ad esempio, il seguente codice inserisce tre triangoli nell'albero dei simplessi a tre diversi valori di filtrazione.

Input:

```
st.insert([0, 1, 2], filtration = 0.1)
st.insert([1, 2, 3], filtration = 0.2)
st.insert([0, 1, 3], filtration = 0.4)

st_gen = st.get_filtration()
for splx in st_gen :
    print(splx)
```

Output:

```
([0], 0.0)
([1], 0.0)
([0, 1], 0.0)
([2], 0.0)
([1, 2], 0.0)
([3], 0.0)
([1, 3], 0.0)
([0, 2], 0.1)
([0, 1, 2], 0.1)
([2, 3], 0.2)
([1, 2, 3], 0.2)
([0, 3], 0.4)
([0, 1, 3], 0.4)
```

Come si può vedere, quando aggiungiamo un nuovo simplesso con un dato valore di filtrazione, tutte le sue facce che non erano già presenti nel complesso vengono aggiunte con lo stesso valore di filtrazione: qui l'arco [0, 3] non faceva parte dell'albero prima di includere il triangolo [0, 1, 3] e quindi viene inserito con il valore di filtrazione del triangolo. D'altra parte, il valore di filtrazione delle facce dei simplessi aggiunti che erano già parte dell'albero rimane invariato.
Si può modificare il valore di filtrazione di qualsiasi simplesso già incluso nell'albero utilizzando il metodo `assign_filtration()`. Ad esempio, il seguente codice modifica il valore di filtrazione del vertice [3].

Input:

```
st.assign_filtration([3], filtration = 0.8)
st_gen = st.get_filtration()

for splx in st_gen:
    print(splx)
```

Output:

```
([0], 0.0)
([1], 0.0)
([0, 1], 0.0)
([2], 0.0)
([1, 2], 0.0)
([1, 3], 0.0)
([0, 2], 0.1)
([0, 1, 2], 0.1)
([2, 3], 0.2)
([1, 2, 3], 0.2)
([0, 3], 0.4)
([0, 1, 3], 0.4)
([3], 0.8)
```

Si noti che il vertice [3] è stato spostato alla fine della filtrazione poiché ora ha il valore di filtrazione più alto. Tuttavia, questo albero dei simplessi non è più un complesso simpliciale filtrato poiché il valore di filtrazione del vertice [3] è superiore a quello dell'arco $[2, 3]$.
Possiamo risolvere il problema utilizzando il metodo `make_filtration_non_decreasing()`.

Input:

```
st.make_filtration_non_decreasing()
st_gen = st.get_filtration()

for splx in st_gen:
    print(splx)
```

Output:

```
([0], 0.0)
([1], 0.0)
([0, 1], 0.0)
([2], 0.0)
([1, 2], 0.0)
([0, 2], 0.1)
([0, 1, 2], 0.1)
([3], 0.8)
([0, 3], 0.8)
([1, 3], 0.8)
([0, 1, 3], 0.8)
([2, 3], 0.8)
([1, 2, 3], 0.8)
```

Infine, è importante menzionare il metodo `filtration()`, che restituisce il valore di filtrazione di un determinato simplesso all'interno della filtrazione.

Input:

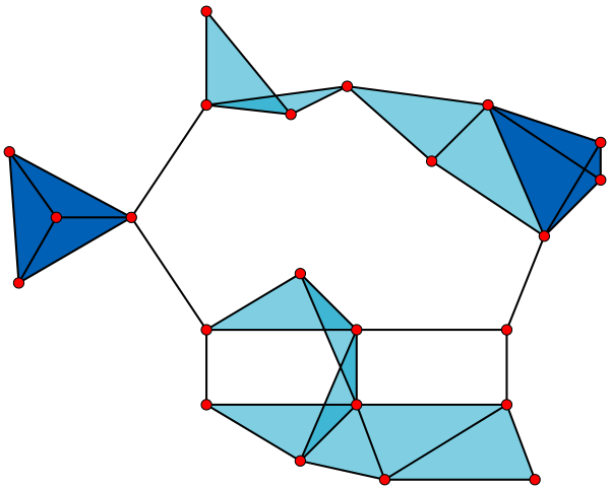
```
st.filtration([2, 3])
```

Output:

```
0.8
```

6 Costruzione di complessi simpliciali da una point cloud

In questa sezione vediamo come costruire un complesso simpliciale Vietoris-Rips e Alpha partendo da punti in $\mathbb{R}^d$.



Il dataset base del quale costruiremo sia il complesso di Vietoris Rips nella sezione 6.1 sia il complesso Alpha nella sezione 6.2 è dato dalla registrazione del cammino di 3 persone. Il cammino di 3 persone A, B e C, è stato registrato utilizzando il sensore accelerometro di uno smartphone

posto nella loro tasca, dando luogo a 3 serie temporali multivariate in $\mathbb{R}^3$: ogni serie temporale rappresenta le 3 coordinate dell'accelerazione del corrispondente camminatore in un sistema di coordinate fissato al sensore. Utilizzando una finestra mobile, ogni serie è stata suddivisa in un elenco di 100 serie temporali composte da 200 punti consecutivi, che sono memorizzate in `data_A`, `data_B` e `data_C`.

I dati vengono caricati con il modulo `pickle`, che è un modulo della libreria standard di Python. La documentazione ufficiale è a questo link: https://docs.python.org/3/library/pickle.html.

L'oggetto `data_A` è un elenco di 100 serie temporali delle accelerazioni 3D per A. Con il seguente codice, diamo un'occhiata alle dimensioni di `data_A`, la stampiamo e rappresentiamo la traiettoria delle accelerazioni per la prima serie temporale di A. Successivamente facciamo lo stesso per `data_B` e `data_C`.

```
import numpy as np
import pickle as pickle
import GUDHI as gd
from pylab import *
from mpl_toolkits.mplot3d import Axes3D

# Apri il file contenente i dati
f = open("C:/Users/valet/Desktop/datasets/data_acc", "rb")
data = pickle.load(f)
f.close()

# Estrai le serie temporali
data_A = data[0]
data_B = data[1]
data_C = data[2]
label  = data[3]
print(label)

# Controlla le dimensioni di data_A
print(np.shape(data_A))

# Seleziona un campione da data_A
data_A_sample = data_A[0]

# Crea una figura 3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Rappresenta la traiettoria delle accelerazioni
ax.scatter(data_A_sample[:, 0], data_A_sample[:, 1], data_A_sample[:, 2])
plt.show()
```

Listing 1: Serie temporali per A

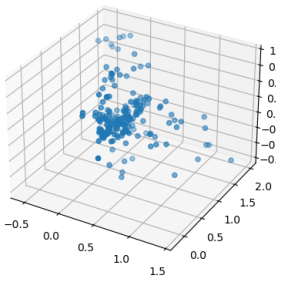


Fig 1.10: Traiettoria delle accelerazioni per la prima serie temporale di A

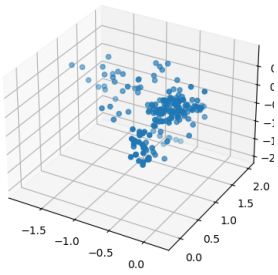


Fig 1.11: Traiettoria delle accelerazioni per la prima serie temporale di B.

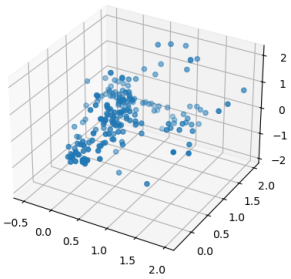


Fig 1.12: Traiettoria delle accelerazioni per la prima serie temporale di C.

6.1 Complesso di Vietoris-Rips

Per calcolare in modo efficiente un complesso di Vietoris-Rips, si può iniziare costruendo un grafo topologico.

> **Definizione 12**
> Un grafo topologico è un grafo tale che ha:
>
> - tanti vertici quanti sono i punti distinti del dataset;
> - come archi solo coppie di punti la cui distanza è minore o uguale a ϵ.

In altre parole, corrisponde all'1-scheletro del cloud di punti con lunghezza degli archi limitata superiormente. In GUDHI, questo viene effettuato tramite la funzione `RipsComplex()`.

Input:

```
skeleton = gd.RipsComplex(points = data_A_sample, max_edge_length = 0.2)
```

Il parametro `max_edge_length` è il diametro massimo: solo archi di lunghezza minore o uguale a questo valore sono inclusi nell'1-scheletro. Dallo 1-scheletro troncato è quindi possibile aggiungere simplessi di dimensione superiore al complesso simpliciale, a condizione che tutte le loro facce siano già nel complesso. Il loro valore di filtrazione è quindi definito come il massimo valore di filtrazione delle sue facce, che corrisponde al loro diametro. Questo processo genera necessariamente l'ϵ-complesso di Vietoris-Rips, poiché i valori di filtrazione non possono mai superare ϵ in questo modo. In pratica, è opportuno definire una dimensione massima dei simplessi da aggiungere al complesso di Vietoris-Rips per motivi computazionali. In GUDHI, questo si ottiene tramite il metodo `create_simplex_tree()` della classe `RipsComplex`, che accetta l'argomento `max_dimension` per limitare l'albero:

```
Rips_simplex_tree_sample = skeleton.create_simplex_tree(max_dimension = 3)
```

Il parametro `max_dimension` è la dimensione massima dei simplessi inclusi nella filtrazione. Il risultato è un simplex tree, di dimensione 3 in questo esempio.

Input:

```
Rips_simplex_tree_sample.dimension()
```

Output:

```
3
```

Possiamo utilizzare i metodi dell'oggetto simplex tree per descrivere la filtrazione Vietoris-Rips. Ad esempio, possiamo verificare che i 200 punti di `data_A_sample` siano tutti vertici della filtrazione Vietoris-Rips.

Input:

```
Rips_simplex_tree_sample.num_vertices()
```

Output:

```
200
```

Il numero di simplessi nel Vietoris-Rips complex è:

Input:

```
Rips_simplex_tree_sample.num_simplices()
```

Output:

```
27851
```

Notiamo che questo è effettivamente il numero di simplessi nell'ultimo Vietoris-Rips complex della filtrazione, ossia con parametro `max_edge_length=0.2`. Osserviamo che il numero di simplessi in un Vietoris-Rips complex aumenta molto rapidamente con il numero di punti e la dimensione.
Ora calcoliamo l'elenco dei simplessi nel Vietoris-Rips complex con la funzione `get_filtration()`.

Input:

```
rips_generator = Rips_simplex_tree_sample.get_filtration()
```

Per stampare i primi 300 elementi nella lista.

Input:

```
rips_list = list(rips_generator)
for splx in rips_list[0:300]:
    print(splx)
```

La funzione di filtrazione è il diametro del simplesso, che è naturalmente pari a zero per i vertici. Il primo edge nella filtrazione è $[6, 34]$, questi due punti sono i due più vicini in `data_A_sample`, a una distanza $\delta_{6,34} = 0.0100$ l'uno dall'altro. Il primo triangolo è $[4, 53, 191]$, con un valore di filtrazione pari a $\delta_{4,53,191} = 0.0327$.

6.2 Complesso Alpha

I complessi Alpha contengono meno simplessi rispetto ai complessi di Vietoris-Rips, e quindi possono essere considerati un'opzione migliore dal punto di vista computazionale. Essi sono sottocomplessi del complesso di Delaunay e, in quanto tali, sono complessi simpliciali geometrici.
La funzione `AlphaComplex()` calcola direttamente il simplex tree che rappresenta il complesso Alpha:

```
alpha_complex = gd.AlphaComplex(points = data_A_sample)
st_alpha = alpha_complex.create_simplex_tree(max_alpha_square = 0.2^2)
```

Il parametro `max_alpha_square` definisce il valore massimo del quadrato della costante alpha, che controlla la dimensione massima dei simplessi inclusi nel complesso Alpha. Un valore più piccolo di `max_alpha_square` limita la dimensione massima dei simplessi, restringendo il complesso Alpha ai simplessi più piccoli. Nell'esempio riportato, viene utilizzato un valore di $0.2^2 = 0.04$, che impone un limite alla dimensione dei simplessi all'interno del complesso.
Il parametro `default_filtration_value` si riferisce al valore di filtrazione associato ai simplessi, che viene assegnato automaticamente quando si costruisce un complesso simpliciale. Questo valore determina l'ordine in cui i simplessi appaiono durante la costruzione del complesso.
Il set di punti `data_A_sample` appartiene al complesso Alpha, come mostrato dal seguente comando.
Input:

```
st_alpha.dimension()
```

Output:

```
3
```

Come per il complesso di Vietoris-Rips, i 200 punti di `data_A_sample` sono tutti vertici del complesso Alpha.
Input:

```
st_alpha.num_vertices()
```

Output:

```
200.
```

Si noti che il numero di simplessi nel complesso Alpha è inferiore rispetto a quello del complesso di Vietoris-Rips.
Input:

```
st_alpha.num_simplices()
```

Output:

```
3334
```

7 Visualizzare complessi simpliciali

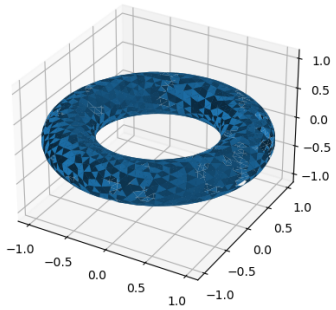
7.1 Complesso Alpha

In questa sezione costruiamo un complesso simpliciale da una nuvola di dati campionati in maniera casuale da un toro bidimensionale. Il seguente codice, che utilizza la libreria Matplotlib, ci permette di rappresentare la triangolazione. La documentazione ufficiale è al link: https://matplotlib.org/.

```python
import numpy as np
import GUDHI as gd

# Lettura del file OFF del toro 3D
torus = gd.read_points_from_off_file(off_file='datasets/tore3D_1307.off')

# Creazione dell'Alpha Complex
ac = gd.AlphaComplex(points=torus)
st = ac.create_simplex_tree()

# Estrazione dei punti dall'Alpha Complex
points = np.array([ac.get_point(i) for i in range(st.num_vertices())])

# Impostiamo alpha a 0.005 di default e selezioniamo solo i triangoli
triangles = np.array([s[0] for s in st.get_skeleton(2) if len(s[0]) == 3 and s[1]
    <= 0.005])

# Import delle librerie per la visualizzazione 3D
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib.widgets import Slider

# Creazione della figura 3D
fig = plt.figure()
ax = fig.add_subplot(projection='3d')

# Visualizzazione della superficie con i triangoli
l = ax.plot_trisurf(points[:, 0], points[:, 1], points[:, 2], triangles=triangles)

# Impostazione dei limiti degli assi
ax.set_xlim(-1.1, 1.1)
ax.set_ylim(-1.1, 1.1)
ax.set_zlim(-1.1, 1.1)

# Mostra il grafico
plt.show()
```

Listing 2: Codice Python per il rendering del 2-Toro 3D

Output:



8 Costruzione di complessi simpliciali da matrici delle distanze

In questa sezione studiamo come costruire complessi di Vietoris-Rips (8.1) e complessi Alpha (8.2) a partire da una matrice di distanze a coppie. In particolare, prima di poter costruire un complesso Alpha, è necessario applicare una trasformazione di Multi-Dimensional Scaling (MDS) sulla matrice (la documentazione si trova a questo link https://scikit-learn.org/stable/modules/manifold.html#multidimensional-scaling).

8.1 Complesso VR definito da una matrice delle distanze

L' α-complesso di Vietoris-Rips di uno spazio metrico X è un complesso simpliciale astratto formato da tutti i simplessi di ogni sottoinsieme finito di X che hanno diametro al più α.
I dati che analizzeremo in questa sezione rappresentano configurazioni di legame proteico. Questo esempio è preso da un articolo di Kovacev-Nikolic [7]. L'articolo confronta le forme chiusa e aperta della proteina legante il maltosio (MBP), una macromolecola costituita da una catena di 370 amminoacidi. L'analisi non si basa su distanze geometriche, ma su una metrica di distanze dinamiche definita da

$$d_{ij} = 1 - |C_{ij}|$$

dove C è la matrice di correlazione tra i gli amminoacide in posizione i e j della sequenza. Il seguente codice carica le matrici di correlazione con pandas e stampa le prime righe della matrice delle distanze.

```
import numpy as np
import pandas as pd
import pickle as pickle
import GUDHI as gd
from pylab import *

path_file = "./datasets/Corr_ProteinBinding/"
files_list = [
    '1anf.corr_1.txt',
    '1ez9.corr_1.txt',
    '1fqa.corr_2.txt',
    '1fqb.corr_3.txt',
    '1fqc.corr_2.txt',
    '1fqd.corr_3.txt',
    '1jw4.corr_4.txt',
    '1jw5.corr_5.txt',
    '1lls.corr_6.txt',
    '1mpd.corr_4.txt',
    '1omp.corr_7.txt',
    '3hpi.corr_5.txt',
    '3mbp.corr_6.txt',
    '4mbp.corr_7.txt'
]
```

```python

# Carica le matrici di correlazione
corr_list = [
    pd.read_csv(
        path_file + u,
        header=None,
        sep='\s+'  # Usa sep='\s+' per eliminare l'avviso
    ) for u in files_list
]

# Calcola le matrici di distanza
dist_list = [1 - np.abs(c) for c in corr_list]

# Seleziona la prima matrice di distanza e stampa le prime righe
D = dist_list[0]
print("Prime righe della prima matrice di distanza:")
print(D.head())

# Stampa le prime righe di tutte le matrici
for i, matrix in enumerate(dist_list):
    print(f"\nPrime righe della matrice di distanza {i+1}:")
    print(matrix.head())
```

Listing 3: Codice Python per la stampa delle prime righe delle matrici di distanza

Output:

```
Prime righe della matrice di distanza 1:
        0         1         2         3         4    ...       365       366       367       368       369
0  0.000000  0.076200  0.171364  0.378207  0.461747  ...  0.686334  0.640850  0.617944  0.695108  0.748451
1  0.076200  0.000000  0.122763  0.233837  0.350744  ...  0.808770  0.754748  0.730646  0.804961  0.848953
2  0.171364  0.122763  0.000000  0.084642  0.131528  ...  0.740322  0.667525  0.653546  0.742430  0.766030
3  0.378207  0.233837  0.084642  0.000000  0.045478  ...  0.856491  0.797437  0.781044  0.842591  0.858435
4  0.461747  0.350744  0.131528  0.045478  0.000000  ...  0.874522  0.804538  0.779865  0.841695  0.849836

[5 rows x 370 columns]

Prime righe della matrice di distanza 2:
        0         1         2         3         4    ...       365       366       367       368       369
0  0.000000  0.435620  0.549594  0.663020  0.780320  ...  0.825498  0.797601  0.787814  0.823452  0.838433
1  0.435620  0.000000  0.061687  0.139978  0.238547  ...  0.685082  0.616944  0.578383  0.646728  0.673970
2  0.549594  0.061687  0.000000  0.038316  0.100271  ...  0.804434  0.740816  0.696357  0.747616  0.771968
3  0.663020  0.139978  0.038316  0.000000  0.034367  ...  0.839132  0.790826  0.736601  0.766731  0.800310
4  0.780320  0.238547  0.100271  0.034367  0.000000  ...  0.907866  0.855508  0.794774  0.815674  0.840803

[5 rows x 370 columns]
```

Fig 1.14 Matrice delle distanze (prime righe)

La funzione `RipsComplex()` genera l'1-scheletro di Vietoris-Rips a partire dalla nuvola di punti.

Input:

```
skeleton_protein = gd.RipsComplex(
    distance_matrix = D.values,
    max_edge_length = 0.8
)
```

Il parametro `max_edge_length` rappresenta il diametro massimo: solo gli spigoli con lunghezza inferiore a questo valore sono inclusi nell'1-scheletro del grafo. Successivamente, creiamo il complesso simpliciale di Vietoris-Rips a partire da questo grafo a 1-scheletro. Questo è un complesso filtrato di Vietoris-Rips, dove la funzione di filtrazione è esattamente il diametro dei simplessi. Usiamo la

funzione `create_simplex_tree()`.

Input:

```
Rips_simplex_tree_protein = skeleton_protein.create_simplex_tree(max_dimension = 2)
```

Il parametro `max_dimension` rappresenta la dimensione massima dei simplessi inclusi nella filtrazione. L'oggetto restituito dalla funzione è un simplex tree, con dimensione 2 in questo esempio.

Input:

```
Rips_simplex_tree_protein.dimension()
```

Output:

```
2
```

Possiamo utilizzare le funzionalità dell'oggetto `simplex tree` per descrivere la filtrazione di Vietoris-Rips. Ad esempio, possiamo verificare che i 370 punti della prima matrice di distanza siano tutti vertici della filtrazione di Vietoris-Rips.

Input:

```
Rips_simplex_tree_protein.num_vertices()
```

Output:

```
370
```

Il numero di simplessi in un complesso di Vietoris-Rips cresce molto velocemente con il numero di punti e la dimensione. In questo caso, ci sono oltre un milione di simplessi nel complesso di Vietoris-Rips.

Input:

```
Rips_simplex_tree_protein.num_simplices()
```

Output:

```
1626660
```

Questo numero rappresenta in effetti il numero di simplessi nell'ultimo complesso di Vietoris-Rips della filtrazione, ovvero con parametro `max_edge_length=0.8`.
Calcoliamo l'elenco dei simplessi nel complesso di Vietoris-Rips con la funzione `get_filtration()`.

Input:

```
rips_filtration = Rips_simplex_tree_protein.get_filtration()
rips_list = list(rips_filtration)
len(rips_list)
```

Output:

```
1626660
```

Input:

```
for splx in rips_list[0:400]:
    print(splx)
```

8.2 Complesso Alpha definito da matrici delle distanze

Come abbiamo visto nella sezione 4.3, la filtrazione del complesso Alpha di un insieme di punti in uno spazio Euclideo è un complesso simpliciale filtrato costruito dalle celle finite di una Triangolazione di Delaunay. Nel nostro caso, i dati non appartengono a uno spazio Euclideo e non siamo in grado di calcolare direttamente una Triangolazione di Delaunay nello spazio metrico usando le distanze a coppie.

Lo scopo dei metodi di Multi-Dimensional Scaling (MDS) è proprio quello di trovare una rappresentazione di n punti in uno spazio Euclideo che preservi il più possibile le distanze a coppie tra i punti nello spazio metrico originale. Esistono diverse versioni degli algoritmi MDS, e le più comuni sono disponibili nella libreria `scikit-learn`.

Calcoliamo una rappresentazione MDS (classica) della matrice D.

Input:

```
from sklearn.manifold import MDS
embedding = MDS(n_components=3, dissimilarity='precomputed')
X_transformed = embedding.fit_transform(D)
X_transformed.shape
```

Output:

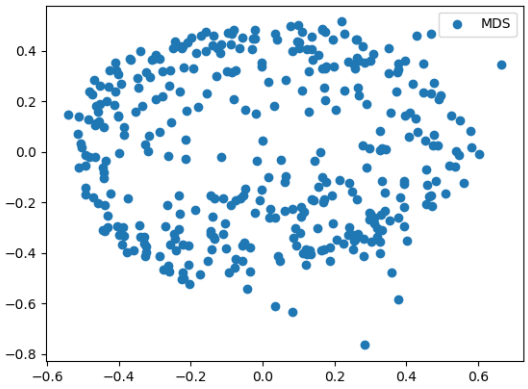
```
(370, 3)
```

Ora possiamo rappresentare questa configurazione, ad esempio sui primi due assi.

Input:

```
fig = plt.figure()
plt.scatter(X_transformed[:, 0], X_transformed[:, 1], label='MDS')
```

Output:



È importante ricordare che il metodo MDS fornisce un'immersione dei dati in uno spazio Euclideo che approssima la matrice delle distanze.

La funzione `AlphaComplex()` calcola direttamente l'albero dei simplessi che rappresenta il complesso Alpha:

```
alpha_complex = gd.AlphaComplex(points=X_transformed)
st_alpha = alpha_complex.create_simplex_tree()
```

La nuvola di punti $X_{\text{transformed}}$ appartiene a uno spazio Euclideo, e così anche il complesso Alpha.

Input:

```
st_alpha.dimension()
```

Output:

```
3
```

Come per il complesso di Vietoris-Rips, gli n punti sono tutti vertici del complesso Alpha.

Input:

```
print(st_alpha.num_vertices())
```

Output:

```
370
```

Notare che il numero di simplessi nel complesso Alpha è molto minore rispetto a quello del complesso di Vietoris-Rips calcolato in precedenza.

Input:

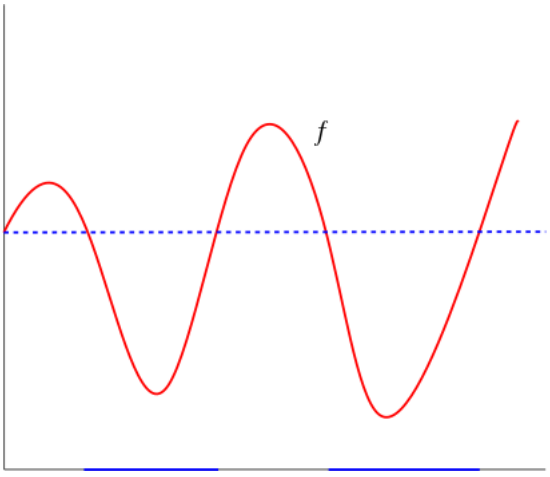
```
print(st_alpha.num_simplices())
```

Output:

```
9329
```

9 Complessi cubicali

In molti casi nell'analisi dei dati si vuole studiare la topologia degli insiemi di sottolivello di una funzione.



Sopra è riportato un esempio di funzione definita su un sottoinsieme di $\mathbb{R}$. In generale la funzione f è definita su un sottoinsieme di $\mathbb{R}^d$.
Un possibile approccio per studiare la topologia degli insiemi di sottolivello di f è definire una griglia regolare su $\mathbb{R}^d$ e quindi un complesso filtrato con vertici posizionati su questa griglia, usando f come funzione di filtrazione. Un complesso filtrato basato su griglia di questo tipo è chiamato *complesso cubicale* in GUDHI. Vediamo la costruzione di questo complesso.

9.1 Costruzione

Esistono vari costruttori che possono essere utilizzati per definire un complesso cubicale. Eccone un esempio:
Input:

```
from GUDHI import CubicalComplex
import numpy as np
cc = CubicalComplex(top_dimensional_cells=np.array([[1., 8., 7.],
                                                    [4., 20., 6.],
                                                    [6., 4., 5.]]))
print(f"Cubical complex is of dimension {cc.dimension()} - {cc.num_simplices()} simplices.")
```

Output:

```
Cubical complex is of dimension 2 - 49 simplices.
```

9.2 Crater dataset

Utilizziamo il dataset 'crater' per illustrare le filtrazioni di complessi cubicali.
Nel nostro esempio il point cloud è in $\mathbb{R}^2$ ed è composto da un anello centrale e quattro cluster agli angoli. Invece di definire direttamente un complesso di Vietoris-Rips o un complesso Alpha sul point cloud, vogliamo studiare gli insiemi di livello superiore di un estimatore di densità calcolato sul point cloud. Possiamo anche visualizzare la densità stimata con uno stimatore kernel 2D utilizzando il modulo Seaborn (la documentazione dello stimatore KDE si trova al link: https://kde.org/documentation/). Per farlo usiamo il seguente codice:

```
import numpy as np
import GUDHI as gd
import pickle as pickle
from pylab import *
from sklearn.neighbors import KernelDensity
import seaborn as sns

# Carica il dataset
f = open("./datasets/crater_tuto", "rb")
crater = pickle.load(f)
f.close()

# Scatter plot del point cloud
plt.scatter(crater[:, 0], crater[:, 1], s=0.1)
```

```
plt.xlabel("X")
plt.ylabel("Y")
plt.show()

# Kernel Density Estimate
sns.kdeplot(
    x=crater[:, 0],
    y=crater[:, 1],
    shade=True,
    cmap="PuBu",
    bw_method=0.3
)
plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```

Listing 4: Codice Python per stima della densità e visualizzazione del point cloud

Output 1:

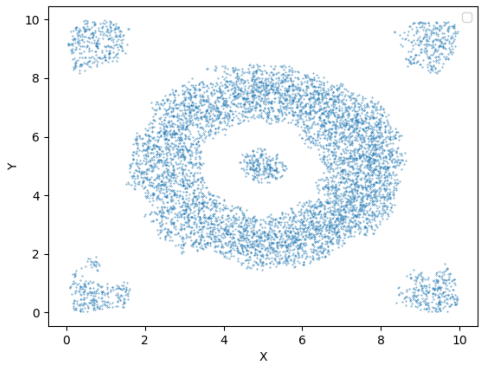


Fig 1.17: Nuvola di punti in $\mathbb{R}^2$

Output 2:

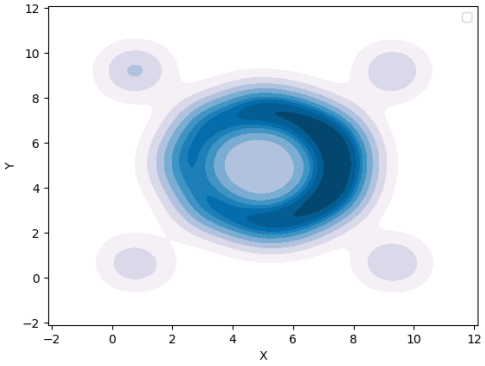


Fig 1.18: Density Estimation of Crater Dataset

9.3 Complesso cubicale per il dataset "crater"

Per prima cosa, definiamo una griglia regolare in $\mathbb{R}^2$.

Input:

```
xval = np.arange(0, 10, 0.05)
yval = np.arange(0, 10, 0.05)
nx = len(xval)
ny = len(yval)
```

Successivamente, adattiamo un kernel density estimator standard 2-dimensionale ai dati utilizzando scikit-learn.

Input:

```
kde = KernelDensity(kernel='gaussian', bandwidth=0.3).fit(crater)
positions = np.array([[u, v] for u in xval for v in yval])
```

La filtrazione del complesso cubicale considera gli insiemi di sottolivello della funzione di filtrazione. Per studiare gli insiemi di livello superiore della densità, dobbiamo prendere l'opposto del KDE. Definiamo il valore di filtrazione in ogni vertice come segue.

Input:

```
filt_values = -kde.score_samples(X=positions)
```

L'intervallo di valori di filtrazione è:

Input:

```
print(min(filt_values), max(filt_values))
```

Output:

```
3.2652121885802146 26.527204685017267
```

Notiamo che i valori di filtrazione sono positivi perché il metodo `score_samples()` della classe `KernelDensity` restituisce il log-densità, che per definizione è sempre negativo.
Ora siamo pronti a calcolare la filtrazione del complesso cubicale basata sulla griglia $[xval, yval]$ usando i valori immagazzinati in `filt_values` come valori di filtrazione.

Input:

```
cc_density_crater = gd.CubicalComplex(
    dimensions=[nx, ny],
    top_dimensional_cells=filt_values
)
```

Notiamo che un complesso cubicale non è un oggetto di tipo simplex tree.

Input:

```
type(cc_density_crater)
```

Output:

`GUDHI.cubical_complex.CubicalComplex`

Tuttavia, la classe `CubicalComplex` ha metodi simili.

Input:

```
cc_density_crater.dimension()
```

Output:

```
2
```

Input:

```
cc_density_crater.num_simplices()
```

Output:

```
160801
```

10 Barcodes e diagrammi di persistenza

In questa sezione studiamo come calcolare i barcodes e i diagrammi di persistenza di una filtrazione definita partendo da un dataset. Inoltre utilizziamo la distanza di Bottleneck per confrontare i diagrammi di persistenza.

10.1 Protein binding dataset

Utilizziamo lo stesso dataset della Sezione 8.1. Dopo aver costruito il complesso di Vietoris-Rips, calcoliamo la persistenza sul simplex tree utilizzando il modulo `persistence()`.

```
import numpy as np
import pandas as pd
import GUDHI as gd
from sklearn import manifold
from pylab import *

path_file = "./datasets/Corr_ProteinBinding/"
files_list = [
    '1anf.corr_1.txt',
    '1ez9.corr_1.txt',
    '1fqa.corr_2.txt',
    '1fqb.corr_3.txt',
    '1fqc.corr_2.txt',
    '1fqd.corr_3.txt',
    '1jw4.corr_4.txt',
    '1jw5.corr_5.txt',
    '1lls.corr_6.txt',
    '1mpd.corr_4.txt',
    '1omp.corr_7.txt',
    '3hpi.corr_5.txt',
    '3mbp.corr_6.txt',
    '4mbp.corr_7.txt'
```

```
]

# Load correlation matrices
corr_list = [
    pd.read_csv(
        path_file + u,
        header=None,
        sep='\s+'  # Use sep='\s+' to avoid warning
    ) for u in files_list
]

# Compute distance matrices
dist_list = [1 - np.abs(c) for c in corr_list]

# Select the first distance matrix and print the first rows
D0 = dist_list[0]

skeleton_protein = gd.RipsComplex(
    distance_matrix = D0.values,
    max_edge_length = 0.8
)

Rips_simplex_tree_protein0 = skeleton_protein.create_simplex_tree(max_dimension =
    2)
BarCodes_Rips0 = Rips_simplex_tree_protein0.persistence()
for i in range(20):
    print(BarCodes_Rips0[i])
```

L'oggetto `BarCodes_Rips0` è la lista dei barcodes: ogni elemento nella lista è una tupla $(\text{dim}, (b, d))$ dove `dim` è una dimensione, `b` è il parametro di nascita e `d` è il parametro di morte. Stampiamo i primi 20 elementi della lista.

Output:

```
(1, (0.07963602000000003, 0.35798637))
(1, (0.12677510000000003, 0.39508646999999997))
(1, (0.26003449999999995, 0.5273952))
(1, (0.07943339999999999, 0.31429881000000004))
(1, (0.08248586999999996, 0.30429980999999995))
(1, (0.11378021999999999, 0.31171713999999995))
(1, (0.07726765000000002, 0.26078758))
(1, (0.09107215000000002, 0.25065161999999996))
(1, (0.0709843, 0.22765623000000001))
(1, (0.09347402000000005, 0.24999733000000002))
(1, (0.07013614000000001, 0.22504734000000004))
(1, (0.08752541000000003, 0.20355559))
(1, (0.21541215000000002, 0.32814707))
(1, (0.06835270000000004, 0.17527247))
(1, (0.08857625000000002, 0.19539684000000002))
(1, (0.08241111999999995, 0.18353136999999997))
(1, (0.10362273, 0.20264340000000003))
(1, (0.09289979999999998, 0.19181444000000003))
(1, (0.09581541999999998, 0.19310879999999997))
(1, (0.09541275000000005, 0.18175165000000004))
```

Fig 1.19: Primi 20 barcodes

Queste 20 caratteristiche topologiche hanno dimensione 1 e corrispondono a cicli di dimensione 1. Abbiamo accesso agli intervalli di persistenza per ogni dimensione utilizzando il metodo

`persistence_intervals_in_dimension()`, ad esempio per la dimensione 0.

Input:

```
Rips_simplex_tree_protein0.persistence_intervals_in_dimension(0)
```

L'ultimo barcode $(0.0, \infty)$ muore all'infinito. Finalmente, possiamo stampare i punti `(birth,death)` nel cosidetto diagramma di persistenza aggiungendo la seguente riga al codice precedente.

Input:

```
gd.plot_persistence_diagram(BarCodes_Rips0)
```

Output:

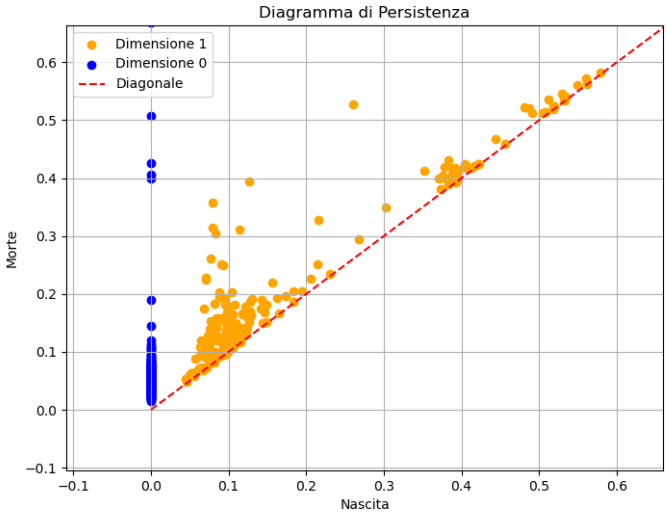


Fig 1.20: Diagramma di persistenza per la prima configurazione di proteine

In questa rappresentazione, le caratteristiche di dimensione 0 sono i punti blu (buchi di dimensione 0, ossia componenti connesse). L'ultima componente connessa muore all'infinito nella filtrazione (punto blu in alto). Le caratteristiche di dimensione 1 sono rappresentate in arancione.
Le caratteristiche topologiche più persistenti sono quei punti che si trovano lontano dalla diagonale. Si noti che questa rappresentazione non indica quali punti sono 'all'origine' di una data caratteristica. Inoltre, una data caratteristica topologica (ossia una classe di omologia che corrisponde a un buco di dimensione d) è per definizione una classe di cicli definiti sul punto nube e quindi può essere rappresentata da più cicli.

10.2 Distanza di Bottleneck

Calcoliamo la filtrazione del complesso di Vietoris-Rips per un'altra configurazione di proteine e il barcode per questa filtrazione.

Input:

```
D1 = dist_list[1]
```

```
skeleton_protein1 = gd.RipsComplex(
    distance_matrix = D1.values,
    max_edge_length = 0.8
)

Rips_simplex_tree_protein1 = skeleton_protein1.create_simplex_tree(max_dimension = 2)

BarCodes_Rips1 = Rips_simplex_tree_protein1.persistence()
```

Il diagramma di persistenza è:

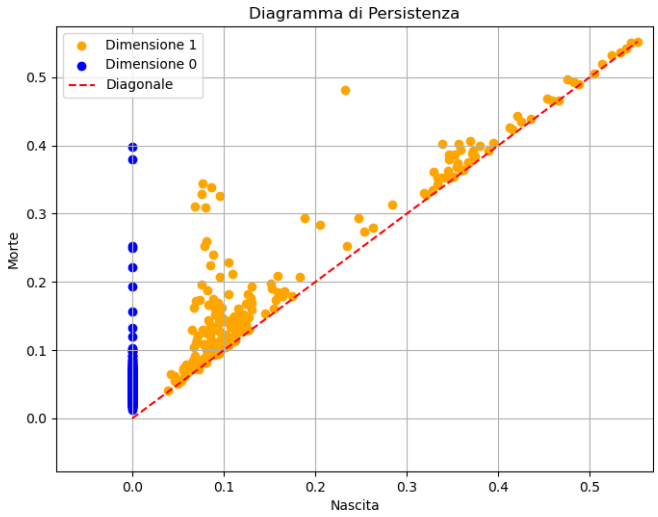


Fig 1.21: Diagramma di persistenza per la seconda configurazione di proteine

La distanza di Bottleneck tra i due diagrammi di persistenza può essere calcolata utilizzando la funzione `bottleneck_distance()`. La distanza di Bottleneck è calcolata per dimensione (per la dimensione 1 nell'esempio qui sotto). Possiamo fornire come argomento della funzione gli intervalli di persistenza per una data dimensione, che possono essere calcolati utilizzando la funzione `persistence_intervals_in_dimension()`.

Input:

```
I0 = Rips_simplex_tree_protein0.persistence_intervals_in_dimension(1)
I1 = Rips_simplex_tree_protein1.persistence_intervals_in_dimension(1)

gd.bottleneck_distance(I0, I1)
```

Output:

```
0.05052142999999998
```

Per impostazione predefinita, la funzione utilizza un algoritmo costoso per calcolare la distanza di Bottleneck esatta. È anche possibile calcolare una distanza di Bottleneck approssimata (ulteriore errore di approssimazione fornito come argomento), che è solitamente molto più veloce da calcolare.

Input:

```
gd.bottleneck_distance(I0, I1, 0.01)
```

Output:

```
0.05061900451758505
```

10.3 MDS sulle distanze di Bottleneck

Un approccio semplice per confrontare le configurazioni MPB è quello di usare le distanze di Bottlenexk tra le configurazioni. Calcoliamo prima la filtrazione del complesso di Vietoris-Rips per ogni MPB. Successivamente calcoliamo la matrice delle distanze di Bottleneck di dimensioni 0 e 1 e applichiamo il metodo Multidimensional Scaling, dalla libreria scikit-learn, per visualizzare le distanze di Bottleneck.

```
import numpy as np
import pandas as pd
import GUDHI as gd
import matplotlib.pyplot as plt
from sklearn.manifold import MDS

path_file = "./datasets/Corr_ProteinBinding/"
files_list = [
    '1anf.corr_1.txt',
    '1ez9.corr_1.txt',
    '1fqa.corr_2.txt',
    '1fqb.corr_3.txt',
    '1fqc.corr_2.txt',
    '1fqd.corr_3.txt',
    '1jw4.corr_4.txt',
    '1jw5.corr_5.txt',
    '1lls.corr_6.txt',
    '1mpd.corr_4.txt',
    '1omp.corr_7.txt',
    '3hpi.corr_5.txt',
    '3mbp.corr_6.txt',
    '4mbp.corr_7.txt'
]

# Carica le matrici di correlazione
corr_list = [
    pd.read_csv(
        path_file + u,
        header=None,
        sep='\s+'  # Usa sep='\s+' per eliminare l'avviso
    ) for u in files_list
]

# Calcola le matrici di distanza
dist_list = [1 - np.abs(c) for c in corr_list]

# Calcola gli intervalli di persistenza per tutte le matrici di distanza
```

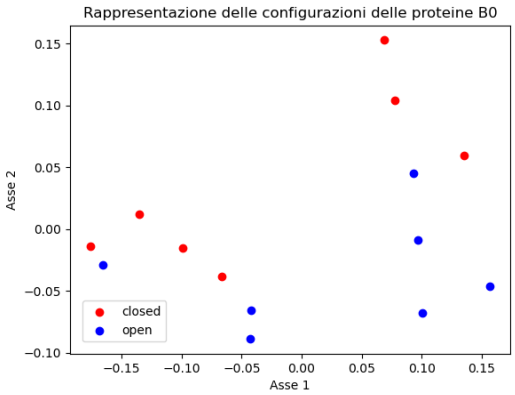
```
persistence_list0 = []
persistence_list1 = []

for i, d in enumerate(dist_list):
    print(i)  # Utilizzato per il debug
    rips_complex = gd.RipsComplex(
        distance_matrix=d.values,
        max_edge_length=0.8
    )
    simplex_tree = rips_complex.create_simplex_tree(max_dimension=2)
    simplex_tree.compute_persistence()
    persistence_list0.append(simplex_tree.persistence_intervals_in_dimension(0))
    persistence_list1.append(simplex_tree.persistence_intervals_in_dimension(1))

l = len(files_list)
B0 = np.zeros((l, l))
B1 = np.zeros((l, l))

# Calcola le matrici di distanza di bottleneck
for i in range(l):
    for j in range(i):
        B0[i, j] = gd.bottleneck_distance(persistence_list0[i], persistence_list0[j
    ])
        B1[i, j] = gd.bottleneck_distance(persistence_list1[i], persistence_list1[j
    ])

B0 = B0 + B0.T  # Matrice simmetrica
B1 = B1 + B1.T  # Matrice simmetrica

# Applicare il Multi-Dimensional Scaling (MDS)
mds = MDS(
    n_components=2,
    max_iter=3000,
    eps=1e-9,
    dissimilarity="precomputed",
    n_jobs=1
)
pos = mds.fit(B0).embedding_

# Visualizzazione dei risultati
plt.scatter(pos[0:7, 0], pos[0:7, 1], color='red', label="closed")
plt.scatter(pos[7:l, 0], pos[7:l, 1], color='blue', label="open")
plt.legend(loc=3, borderaxespad=1)
plt.title("Rappresentazione delle configurazioni delle proteine B0")  # Aggiungi
    titolo
plt.xlabel("Asse 1")  # Aggiungi etichetta per l'asse
plt.ylabel("Asse 2")  # Aggiungi etichetta per l'asse
plt.show()  # Mostra il grafico

pos = mds.fit(B1).embedding_
plt.scatter(pos[0:7, 0], pos[0:7, 1], color = 'red' , label = "closed")
plt.scatter(pos[7:l, 0], pos[7:l, 1], color = 'blue', label = "open")
```

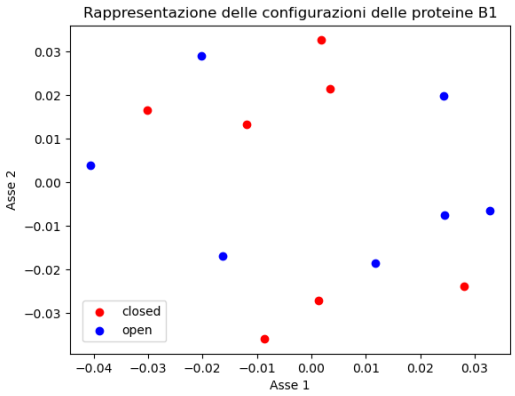
```
87 plt.legend(loc = 3, borderaxespad = 1)
88 plt.title("Rappresentazione delle configurazioni delle proteine B1")  # Aggiungi
       titolo
89 plt.xlabel("Asse 1")  # Aggiungi etichetta per l'asse
90 plt.ylabel("Asse 2")  # Aggiungi etichetta per l'asse
91 plt.show()  # Mostra il grafico
```

Listing 5: Codice Python per la rappresentazione MDS delle configurazioni proteiche

Output:

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
```





11 Sperimentazione

11.1 Root Mean Square Deviation (RMSD)

Prima di iniziare la sperimentazione, introduciamo il concetto di misura RMSD.
La *Root Mean Square Deviation* (RMSD) è una misura statistica utilizzata per quantificare la dissimilarità tra due set di dati, ed è particolarmente usata in bioinformatica e chimica computazionale per confrontare la conformazione spaziale di strutture molecolari o proteiche. La RMSD calcola la radice quadrata della media dei quadrati delle differenze tra le posizioni corrispondenti degli atomi di due strutture. La RMSD tra due configurazioni di atomi, $\{r_1, r_2, \ldots, r_n\}$ e $\{r'_1, r'_2, \ldots, r'_n\}$, è data dalla seguente espressione:

$$\text{RMSD} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} |\mathbf{r}_i - \mathbf{r}'_i|^2}$$

dove:

- n è il numero di atomi nelle due strutture,
- r_i è la posizione dell'atomo i nella prima struttura,
- r'_i è la posizione dell'atomo i nella seconda struttura,
- $|r_i - r'_i|$ è la distanza euclidea tra le posizioni corrispondenti degli atomi.

La RMSD fornisce un'idea della somiglianza o della differenza tra due configurazioni. Valori di RMSD più piccoli indicano una maggiore somiglianza, mentre valori più grandi suggeriscono una maggiore dissimilarità. Nella pratica, la RMSD è utilizzata per confrontare strutture proteiche allineate, per esempio, per valutare quanto una struttura di riferimento e una struttura predetta siano simili tra loro.
Sebbene la RMSD sia ampiamente utilizzata per misurare la dissimilarità, ha dei limiti, tra cui la sua sensibilità alla grandezza delle strutture e la sua incapacità di catturare eventuali traslazioni o rotazioni tra le strutture.

11.2 SHREC2021

Utilizziamo i metodi visti in questa relazione per comprendere l'importanza della scelta tra il complesso simpliciale di Vietoris-Rips e Alpha nella classificazione delle configurazioni di alcune proteine. I dati sono stati presi da [10] e scaricati da questo link https://github.com/rea1991/SHREC2021. In particolare, allo stesso link troviamo una classificazione delle configurazioni in cluster tale che due configurazioni appartengono allo stesso cluster se hanno lo stesso PDB code. Lavoriamo su 18 configurazioni così scelte: 9 appartengono al cluster numero 1, 6 al cluster numero 2 e 3 al cluster numero 3.
Con il seguente codice costruiamo i complessi di Vietoris-Rips e Alpha di ognuna di queste configurazioni e ne calcoliamo i diagrammi di persistenza stampando il tempo impiegato. Successivamente costruiamo le matrici delle distanze di bottleneck per i complessi di Vietoris-Rips e Alpha di dimensione 0 e 1. Infine clusterizziamo i dati utilizzando il metodo Multidimensional Scaling (MDS) e KMeans. Confontiamo tutti i risultati ottenuti in termini di accuratezza e tempo. Per l'accuratezza utilizziamo il metodo la cui documentazione si trova al link https://scikit-learn.org/dev/modules/generated/sklearn.metrics.balanced_accuracy_score.html.
In seguito, clusterizziamo i dati utilizzando la distanza RMSD come metrica, che è standard in biologia per calcolare la dissimilarità tra due configurazioni proteiche, e confrontiamo i risultati con quelli precedentemente ottenuti.
Ci aspettiamo che i cluster che otterremo non rispetteranno i cluster di partenza in quanto clusterizzare sulla base dei diagrammi di persistenza significa classificare le diverse conformazioni rispetto alla forma e la forma cambia anche da una conformazione ad un'altra della stessa proteina.

```python
import os
import time
import numpy as np
import gudhi as gd
import matplotlib.pyplot as plt
from scipy.spatial import distance_matrix
from sklearn.manifold import MDS
from sklearn.cluster import KMeans
import pandas as pd
from sklearn.metrics import balanced_accuracy_score

# Specifica qui il percorso della cartella contenente i file .off
directory_path = '/content/sample_data/'

# Cluster target iniziali per i file .off
cluster1 = {604, 447, 1512, 1510, 1458, 118, 443, 1534, 867}
cluster2 = {356, 403, 35, 17, 455, 955}
cluster3 = {641, 147, 1430}
cluster_targets = {1: cluster1, 2: cluster2, 3: cluster3}

# Funzione per calcolare il massimo valore finito in un diagramma di persistenza
def replace_inf_with_max(diag):
    finite_vals = [b for b, d in diag if d != float('inf')]
    if finite_vals:
        max_val = max(finite_vals)
        return [(b, max_val if d == float('inf') else d) for b, d in diag]
    else:
        return diag

# Funzione per calcolare la distanza di bottleneck tra due diagrammi di persistenza
def bottleneck_distance(diag1, diag2):
    return gd.bottleneck_distance(diag1, diag2)

# Funzione per calcolare i diagrammi di persistenza usando il complesso di Vietoris
    -Rips
def compute_vietoris_rips_persistence(points):
    rips_complex = gd.RipsComplex(points=points, max_edge_length=2.0)
    simplex_tree = rips_complex.create_simplex_tree(max_dimension=2)
    simplex_tree.compute_persistence()
    return simplex_tree.persistence_intervals_in_dimension(0), simplex_tree.
    persistence_intervals_in_dimension(1)

# Funzione per calcolare i diagrammi di persistenza usando l'Alpha Complex
def compute_alpha_persistence(points):
    alpha_complex = gd.AlphaComplex(points=points)
    simplex_tree = alpha_complex.create_simplex_tree()
    simplex_tree.compute_persistence()
    return simplex_tree.persistence_intervals_in_dimension(0), simplex_tree.
    persistence_intervals_in_dimension(1)

# Funzione per leggere i file .off e ottenere i punti
def read_off_file(file_name):
```

```
    file_path = os.path.join(directory_path, file_name)
    with open(file_path, 'r') as file:
        if file.readline().strip() != 'OFF':
            raise ValueError(f"{file_path} non e' un file .off valido")

        while True:
            line = file.readline().strip()
            if line and not line.startswith('#'):
                break

        n_verts, _, _ = map(int, line.split())
        points = []
        for _ in range(n_verts):
            line = file.readline().strip()
            if line and not line.startswith('#'):
                points.append(list(map(float, line.split())))
    return np.array(points)

# Lista di file .off nella directory specificata
off_files = [f for f in os.listdir(directory_path) if f.lower().endswith('.off')]
n_files = len(off_files)

if n_files == 0:
    raise ValueError("Nessun file .off trovato nella directory specificata.")
else:
    print(f"Trovati {n_files} file .off nella directory '{directory_path}'")

# Matrici per le distanze di bottleneck (dimensione 0 e 1)
bottleneck_vr_dim0 = np.zeros((n_files, n_files))
bottleneck_vr_dim1 = np.zeros((n_files, n_files))
bottleneck_alpha_dim0 = np.zeros((n_files, n_files))
bottleneck_alpha_dim1 = np.zeros((n_files, n_files))

# Calcolo delle persistenze per tutti i file .off
vr_persistences_dim0, vr_persistences_dim1 = [], []
alpha_persistences_dim0, alpha_persistences_dim1 = [], []

for file in off_files:
    points = read_off_file(file)

    start_time = time.time()
    vr_dim0, vr_dim1 = compute_vietoris_rips_persistence(points)
    vr_persistences_dim0.append(replace_inf_with_max(vr_dim0))
    vr_persistences_dim1.append(replace_inf_with_max(vr_dim1))
    print(f"Complesso di Vietoris-Rips per {file} in {time.time() - start_time:.2f}
     secondi")

    start_time = time.time()
    alpha_dim0, alpha_dim1 = compute_alpha_persistence(points)
    alpha_persistences_dim0.append(replace_inf_with_max(alpha_dim0))
    alpha_persistences_dim1.append(replace_inf_with_max(alpha_dim1))
    print(f"Complesso Alpha per {file} in {time.time() - start_time:.2f} secondi")
```

```
101
102 # Calcolo delle distanze di bottleneck
103 for i in range(n_files):
104     for j in range(n_files):
105         bottleneck_vr_dim0[i, j] = bottleneck_distance(vr_persistences_dim0[i],
    vr_persistences_dim0[j])
106         bottleneck_vr_dim1[i, j] = bottleneck_distance(vr_persistences_dim1[i],
    vr_persistences_dim1[j])
107         bottleneck_alpha_dim0[i, j] = bottleneck_distance(alpha_persistences_dim0[i
    ], alpha_persistences_dim0[j])
108         bottleneck_alpha_dim1[i, j] = bottleneck_distance(alpha_persistences_dim1[i
    ], alpha_persistences_dim1[j])
109
110 # Clusterizzazione e confronto per ciascuna matrice di distanza
111 matrices = [
112     (bottleneck_vr_dim0, 'Vietoris-Rips (Dim 0)'),
113     (bottleneck_vr_dim1, 'Vietoris-Rips (Dim 1)'),
114     (bottleneck_alpha_dim0, 'Alpha Complex (Dim 0)'),
115     (bottleneck_alpha_dim1, 'Alpha Complex (Dim 1)')
116 ]
117
118 results = []
119 num_clusters = 3
120
121 for idx, (matrix, matrix_name) in enumerate(matrices):
122     kmeans = KMeans(n_clusters=num_clusters, random_state=42)
123     clusters = kmeans.fit_predict(matrix)
124
125     # Riduzione dimensionale con MDS
126     mds = MDS(n_components=2, dissimilarity='precomputed', random_state=42)
127     coords = mds.fit_transform(matrix)
128
129     # Plot dei cluster
130     plt.figure(figsize=(10, 7))
131     for cluster_id in range(num_clusters):
132         plt.scatter(
133             coords[clusters == cluster_id, 0],
134             coords[clusters == cluster_id, 1],
135             label=f'Cluster {cluster_id + 1}'
136         )
137
138     plt.title(f"Cluster con KMeans e MDS ({matrix_name})")
139     plt.xlabel("MDS Dimension 1")
140     plt.ylabel("MDS Dimension 2")
141     plt.legend()
142     plt.grid(True)
143     plt.show()
144
145 # Mappa dei file ai cluster target
146 file_to_cluster = {}
147 for target, cluster_set in cluster_targets.items():
148     for file_index in cluster_set:
```

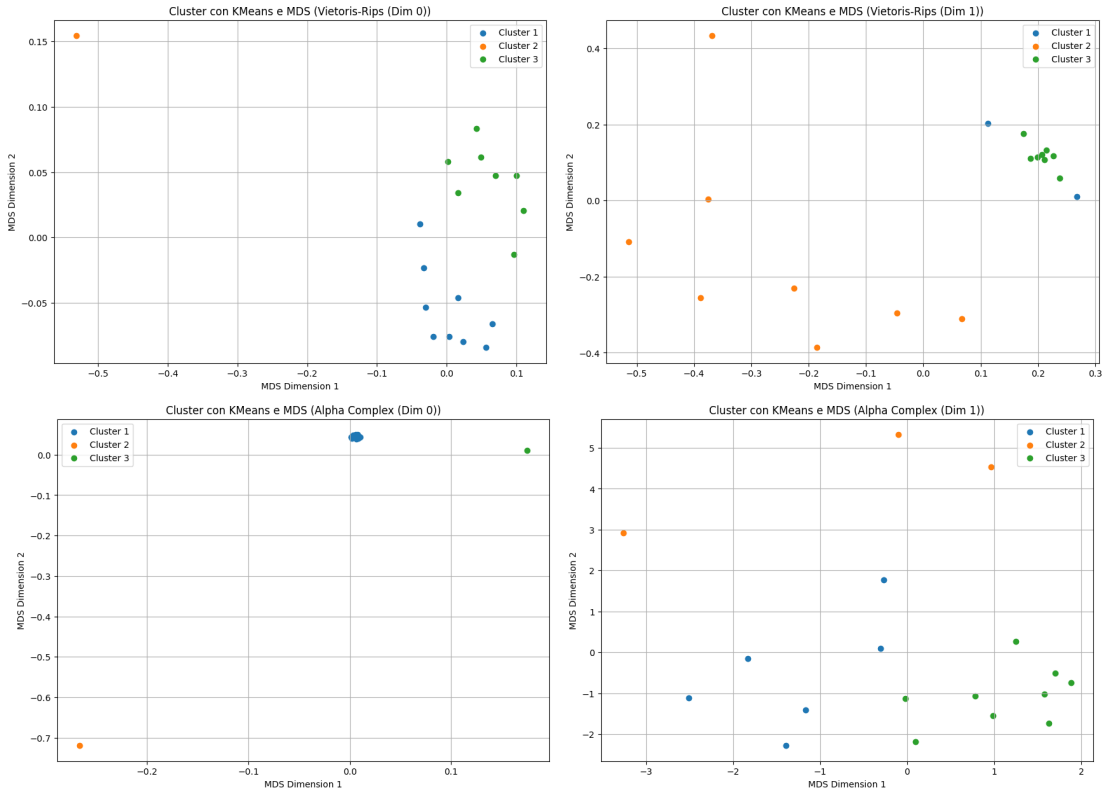
```
149         file_to_cluster[file_index] = target - 1  # I cluster di KMeans iniziano da
        0
150
151 # Creazione delle etichette target in base all'ordine dei file .off
152 true_labels = [file_to_cluster.get(int(file.split('.')[0]), -1) for file in
        off_files]
153 # Verifica che non ci siano file senza cluster associato
154 if -1 in true_labels:
155     raise ValueError("Uno o piu' file .off non ha un cluster target specificato.")
156
157 # Calcolo dell'accuratezza bilanciata per ciascuna matrice
158 for idx, (matrix, matrix_name) in enumerate(matrices):
159     kmeans = KMeans(n_clusters=num_clusters, random_state=42)
160     predicted_labels = kmeans.fit_predict(matrix)
161
162     # Calcolo dell'accuratezza bilanciata
163     balanced_accuracy = balanced_accuracy_score(true_labels, predicted_labels)
164     results.append((matrix_name, balanced_accuracy))
165
166 # Visualizzazione dei risultati
167 for matrix_name, balanced_accuracy in results:
168     print(f"Accuratezza bilanciata per {matrix_name}: {balanced_accuracy:.4f}")
```

11.3 Osservazioni e conclusioni

Nella prima parte dell'output stampiamo il tempo necessario per la costruzione del complessi simpliciali. In particolare:

```
Trovati 18 file .off nella directory '/content/sample_data/'
Complesso di Vietoris-Rips per 356.off in 29.85 secondi
Complesso Alpha per 356.off in 5.13 secondi
Complesso di Vietoris-Rips per 17.off in 28.40 secondi
Complesso Alpha per 17.off in 4.37 secondi
(...)
Complesso di Vietoris-Rips per 604.off in 36.62 secondi
Complesso Alpha per 604.off in 5.45 secondi
Complesso di Vietoris-Rips per 403.off in 29.04 secondi
Complesso Alpha per 403.off in 5.60 secondi
```

Da ciò capiamo che la costruzione di un complesso Alpha partendo da file .off impiega meno tempo rispetto alla costruzione di un complesso di Vietoris-Rips.
Successivamente otteniamo queste 4 classificazioni:



Con accuratezza:

```
Accuratezza bilanciata per Vietoris-Rips (Dim 0): 0.7222
Accuratezza bilanciata per Vietoris-Rips (Dim 1): 0.1481
Accuratezza bilanciata per Alpha Complex (Dim 0): 0.2963
Accuratezza bilanciata per Alpha Complex (Dim 1): 0.0741
```

Da ciò capiamo che la clusterizzazione ottenuta con il complesso di Vietoris-Rips è molto più accurata rispetto a quella ottenuta con il complesso Alpha.
Se invece adesso consideriamo la distanza RMSD tra le configurazioni di proteine e clusterizziamo in base a questa metrica:

```python
import os
import numpy as np
import trimesh
from scipy.spatial import procrustes
from sklearn.cluster import KMeans
from scipy.spatial.distance import cdist
import matplotlib.pyplot as plt
from sklearn.manifold import MDS
from sklearn.metrics import balanced_accuracy_score

def load_mesh(file_path):
    """Carica una mesh da un file .off e restituisce i vertici."""
    try:
        mesh = trimesh.load(file_path)
        return mesh.vertices
    except Exception as e:
```

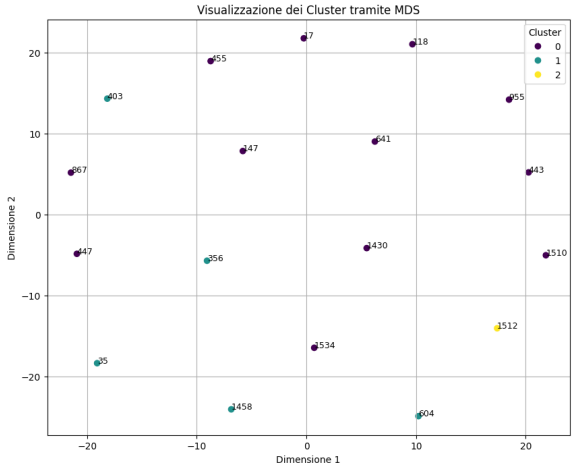
```
        print(f"Errore caricando {file_path}: {e}")
        return None

def calculate_rmsd(vertices1, vertices2):
    """Allinea due insiemi di vertici e calcola l'RMSD."""
    if len(vertices1) != len(vertices2):
        min_size = min(len(vertices1), len(vertices2))
        vertices1 = vertices1[:min_size]
        vertices2 = vertices2[:min_size]

    _, aligned_vertices2, disparity = procrustes(vertices1, vertices2)
    rmsd = np.sqrt(np.sum((vertices1 - aligned_vertices2) ** 2) / len(vertices1))
    return rmsd

# Percorso della directory con i file .off
directory_path = "/content/sample_data"

# Carica i vertici delle mesh
models = {}
for filename in os.listdir(directory_path):
    if filename.endswith(".off"):
        file_path = os.path.join(directory_path, filename)
        vertices = load_mesh(file_path)
        if vertices is not None:
            models[filename] = vertices

# Calcola la matrice RMSD tra ogni coppia di modelli
model_names = list(models.keys())
n_models = len(model_names)
rmsd_matrix = np.zeros((n_models, n_models))

for i in range(n_models):
    for j in range(i + 1, n_models):
        rmsd = calculate_rmsd(models[model_names[i]], models[model_names[j]])
        rmsd_matrix[i, j] = rmsd
        rmsd_matrix[j, i] = rmsd

# Appiattiamo la matrice RMSD e usiamo KMeans per la clusterizzazione
n_clusters = 3  # Imposta il numero di cluster desiderato
kmeans = KMeans(n_clusters=n_clusters, random_state=0).fit(rmsd_matrix)

# Assegna ogni modello al cluster
clusters = {i: [] for i in range(n_clusters)}
for idx, label in enumerate(kmeans.labels_):
    clusters[label].append(model_names[idx])

# Stampa i risultati dei cluster
for cluster_id, model_list in clusters.items():
    print(f"Cluster {cluster_id + 1}: {model_list}")

# Calcola la matrice di distanza tra i modelli (basata sulla matrice RMSD)
distance_matrix = cdist(rmsd_matrix, rmsd_matrix, 'euclidean')
```

```
69
70 # Applica MDS per ridurre la distanza a 2 dimensioni
71 mds = MDS(n_components=2, random_state=0)
72 coordinates = mds.fit_transform(distance_matrix)
73
74 # Crea il grafico
75 plt.figure(figsize=(10, 8))
76
77 # Assegna colori ai cluster predetti
78 scatter = plt.scatter(coordinates[:, 0], coordinates[:, 1], c=kmeans.labels_, cmap=
       'viridis', marker='o')
79
80 # Aggiungi una legenda
81 plt.legend(*scatter.legend_elements(), title="Cluster")
82
83 # Aggiungi etichette ai punti (i nomi dei modelli)
84 for i, model_name in enumerate(model_names):
85     plt.text(coordinates[i, 0], coordinates[i, 1], model_name.split('.')[0],
       fontsize=9)
86
87 # Impostazioni grafiche
88 plt.title('Visualizzazione dei Cluster tramite MDS')
89 plt.xlabel('Dimensione 1')
90 plt.ylabel('Dimensione 2')
91 plt.grid(True)
92 plt.show()
93
94 # Cluster originali (come indici, assegnati manualmente)
95 original_clusters = {
96     0: [604, 447, 1512, 1510, 1458, 118, 443, 1534, 867],    # Cluster 1
97     1: [356, 403, 35, 17, 455, 955],                         # Cluster 2
98     2: [641, 147, 1430]                                      # Cluster 3
99 }
100
101 # Verifica che i modelli siano stati caricati correttamente
102 print(f"Modelli caricati: {model_names}")
103
104 # Creiamo un dizionario che mappa ogni modello al suo cluster originale
105 true_labels = []
106 for model in model_names:
107     found = False
108     for cluster_id, models_in_cluster in original_clusters.items():
109         if int(model.split('.')[0]) in models_in_cluster:  # Supponendo che i nomi
       dei file siano numerici
110             true_labels.append(cluster_id)
111             found = True
112             break
113     if not found:
114         print(f"Attenzione: {model} non trovato nei cluster originali!")
115
116 # Controlla se la lunghezza di true_labels corrisponde alla lunghezza di
       predicted_labels
```

```
117 print(f"Lunghezza true_labels: {len(true_labels)}")
118 print(f"Lunghezza predicted_labels: {len(predicted_labels)}")
119
120 # Calcola l'accuratezza bilanciata solo se le lunghezze sono consistenti
121 if len(true_labels) == len(predicted_labels):
122     accuracy = balanced_accuracy_score(true_labels, predicted_labels)
123     print(f"Balanced Accuracy: {accuracy}")
124 else:
125     print("Errore: le etichette reali e predette hanno lunghezze diverse.")
```

Otteniamo:

```
Cluster 1: ['1430.off', '641.off', '147.off', '17.off', '455.off', '447.off',
            '1534.off', '1510.off', '443.off', '118.off', '867.off', '955.off']
Cluster 2: ['604.off', '35.off', '1458.off', '356.off', '403.off']
Cluster 3: ['1512.off']
```




```
Balanced Accuracy: 0.38888888888888884
```

Osserviamo che abbiamo ottenuto un risultato molto buono per il complesso di Vietoris-Rips (dim 0) e, confrontandolo con la metrica RMSD, lo strumento topologico sembra superiore.

Riferimenti bibliografici

[1] U. Bauer, M. Kerber, F. Roll, and A. Rolle. A unified view on the functiorial nerve theorem and its variations. 2023.

[2] F. Chazal and B. Michel. Tda with python using the gudhi library. building simplicial complexes from distance matrices.

[3] F. Chazal and B. Michel. Tda with python using the gudhi library persistent homology and persistence diagrams.

[4] F. Chazal and B. Michel. Topological data analysis with python and the gudhi library. building simplicial complexes from a point cloud : Rips and alpha complexes.

[5] Pawel Dlotko. Cubical complex user manual.

[6] M. Glisse and V. Rouvreau. Tda with python using the gudhi library. some ways to visualize simplicial complexes.

[7] Violeta Kovacev-Nikolic, Peter Bubenik, Dragan Nikoli´c, and Giseon Heo. Using persistent homology and dynamical distances to analyze protein binding. 2015.

[8] B. Michel. Tda with python using the gudhi library. representing sublevel sets of functions using cubical complexes.

[9] N. Otter, M. A. Porter, U. Tillmann, P. Grindrod, and H. A. Harrington. A roadmap for the computation of persistent homology.

[10] Andrea Raffo, Ulderico Fugacci, Silvia Biasotti, Walter Rocchia, Yonghuai Liu, Ekpo Otu, Reyer Zwiggelaar, David Hunter, Evangelia I. Zacharaki, Eleftheria Psatha, Dimitrios Laskos, Gerasimos Arvanitis, Konstantinos Moustakas, Tunde Aderinwale, Charles Christoffer, Woong-Hee Shin, Daisuke Kihara, Andrea Giachetti, Huu-Nghia Nguyen, Tuan-Duy Nguyen, Vinh-Thuyen Nguyen-Truong, Danh Le-Thanh, Hai-Dang Nguyen, and Minh-Triet Tran. Shrec 2021: Retrieval and classification of protein surfaces equipped with physical and chemical properties. 2021.

[11] Nathaniel Saul and Chris Tralie. Scikit-tda: Topological data analysis for python. 2019.

[12] Christopher Tralie, Nathaniel Saul, and Rann Bar-On. Ripser.py: A lean persistent homology library for python. *The Journal of Open Source Software*, 3(29):925, Sep 2018.

[13] V.Nanda. Lecture notes of the course "computational algebraic topology".