Università degli Studi di Milano - Bicocca
Dipartimento di Informatica, Sistemistica e Comunicazione
Corso di Laurea Magistrale in Data Science

Un algoritmo di semplificazione delle poligonali geografiche di impianti industriali

Relatore: Andrea Maurino

Correlatore: Francesco De Cassai

Tesi di Laurea Magistrale di:
Silvia Gloria Tamburini
Matricola 813117

Anno Accademico 2020-2021

A Michele,

che continua a fidarsi di me

quando io non ci riesco

Indice

Elenco delle tabelle

Elenco delle figure

Capitolo 1

Introduzione

I dati relativi alle coordinate spaziali di un impianto industriale comprendono generalmente diverse decine di strutture, ciascuna descritta da centinaia di punti; e questo comporta che, al momento della visualizzazione dell'impianto su un software, siano necessari diversi minuti per elaborare le varie forme e mostrarle a schermo. Risulta quindi di interesse aziendale sviluppare algoritmi specifici per semplificare queste forme, riducendo il numero di punti da visualizzare, ma mantenendo il più possibile inalterata la forma di partenza.

Il presente lavoro, partendo dal contesto della *map generalization* e in particolare della *line simplification*, considera un approccio per step successivi che, in base a un livello di dettaglio da mantenere (indicato in metri), rimuove un adeguato numero di punti, permettendo una perdita minima del patrimonio informativo. L'approccio è stato testato per mezzo di metriche definite appositamente, che considerano la perdita informativa dell'algoritmo sia in termini di differenze di area, sia in termini di differenze di forma (considerando lo scarto massimo tra la figura originale e la nuova figura).

Il pacchetto *GeoSampling* che produce la semplificazione è originale e scritto in Python, e fa largo uso di librerie standard quali Pandas, MatplotLib e Shapely. Il codice completo è stato pubblicato in un pacchetto apposito su GitHub, ed è disponibile al link [1]. La tesi è in collaborazione con Accenture, che ha fornito i dati degli impianti (più di 400.000 punti).

Il presente lavoro sarà così strutturato:

- Nel capitolo 2, si farà una breve presentazione del contesto più generale della *map simplification* e della *line generalization* all'interno del quale questa tesi si muove;

- Il capitolo 3 consisterà in una estesa descrizione del codice che è stato scritto e implementato;
- Il capitolo 4 presenterà i test che sono stati eseguiti e i risultati che sono stati ottenuti sul dataset che è stato condiviso;
- Infine, il capitolo 5 riassumerà le conclusioni raggiunte e i possibili miglioramenti che si potrebbero apportare in sviluppi futuri di questo codice.

Si osservi che in questo documento non è stato riportato il codice completo, per evitare di appesantirlo: si rimanda alla cartella pubblica di GitHub [1] per qualsiasi problema legato al codice specifico.

Capitolo 2

Revisione della letteratura

Il contesto più generale all'interno del quale si muove questo lavoro è quello della *map generalization* e in particolare della *line simplification*. In queste pagine, si vuole effettuare una breve e non comprensiva *review* della letteratura che si è trovata sull'argomento, seguendo in particolare un approccio *top-bottom*:

- Nella sezione 2.1, si introduce il contesto più generale della *map generalization*, spiegando che cos'è e come viene tipicamente effettuata;
- Nella sezione 2.2, concentrandosi sul *come* viene effettuata la generalizzazione cartografica, si fornisce un breve elenco dei diversi operatori presi in considerazione dalla letteratura;
- Nella sezione 2.3, ci si concentra infine sull'operazione di semplificazione, il più rilevante per questo lavoro di tesi, che sarà descritto più nel dettaglio.

2.1 *Map generalization*

2.1.1 Definizione

La *map generalization* è una operazione cartografica in cui le *features* del mondo reale o di una mappa vengono sintetizzate e astratte in rappresentazioni più adatte a mappe a scale più ridotte. L'operazione permette al cartografo di enfatizzare alcuni fenomeni e processi geografici, e de-enfatizzarne altri [2].

In particolare, si possono fornire tre diverse definizioni di *map generalization* a seconda dei tre contesti più comuni in cui il termine viene utilizzato. I tre contesti sono stati introdotti da [3], e sono rappresentati schematicamente nella figura 2.1.

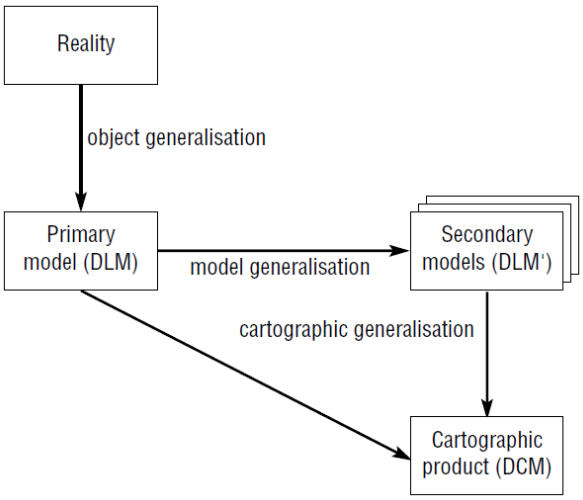


Figura 2.1 – Tipologie di *map generalization*. L'immagine è presa da [4]

Object generalization È l'operazione in cui viene costruito un database geografico primario a partire dal mondo reale. Si tratta quindi di partire dall'infinita complessità presente nel mondo reale ed effettuare un processo di *astrazione* che converta tale complessità in una rappresentazione cartografica significativa, singola e mirata, utilizzabile e utile alla scala data e allo scopo che ci si è prefissati [5]. La *map generalization* è richiesta in questo caso per rappresentare nel database solo le informazioni rilevanti per l'uso richiesto [4].

Model generalization È l'operazione in cui viene costruito un database geografico secondario a partire da quello primario, modificandolo per particolari obiettivi o scale specifiche. Il contesto più comune è quello in cui si hanno dei dati geografici raccolti con un certo livello di dettaglio informativo, e si desidera portarli a un livello di dettaglio inferiore, enfatizzando gli elementi più importanti ed eliminando quelli superflui, ma mantenendo le relazioni logico-spaziali tra gli elementi [4].

Cartographic generalization È l'operazione in cui viene costruita una mappa, cioè una rappresentazione cartografica del database geografico, di solito con uno scopo o un target specifico.

Il termine *map generalization* è più comunemente utilizzato per intendere la *cartographic generalization* [4], tuttavia in questo contesto ci si è interessati all'argomento soprattutto rispetto alla *model generalization*, che viene effettuata so-

prattutto per ridurre il tempo di esecuzione degli algoritmi o lo spazio necessario in memoria per salvare i dati.

2.1.2 Framework concettuale

La *map generalization* è in generale un processo complesso e olistico, per cui quindi non esiste un'unica soluzione *one-click* che separi in maniera automatica le informazioni geografiche essenziali dai dettagli irrilevanti o eccessivi. Tuttavia, si tratta di una operazione di estrema importanza: nelle parole del cartografo Arthur Robinson [6], «l'atto della generalizzazione dà alla mappa la sua ragione di esistere».

Sono stati quindi sviluppati dei *framework* per la *map generalization*, che da una parte descrivono il processo nella sua interezza, e dall'altra ne individuano gli step fondamentali. Quello più diffuso, citato e utilizzato, soprattutto nella letteratura americana [2], è il framework sviluppato da McMaster e Shea [7], [8], mostrato schematicamente nella figura 2.2.

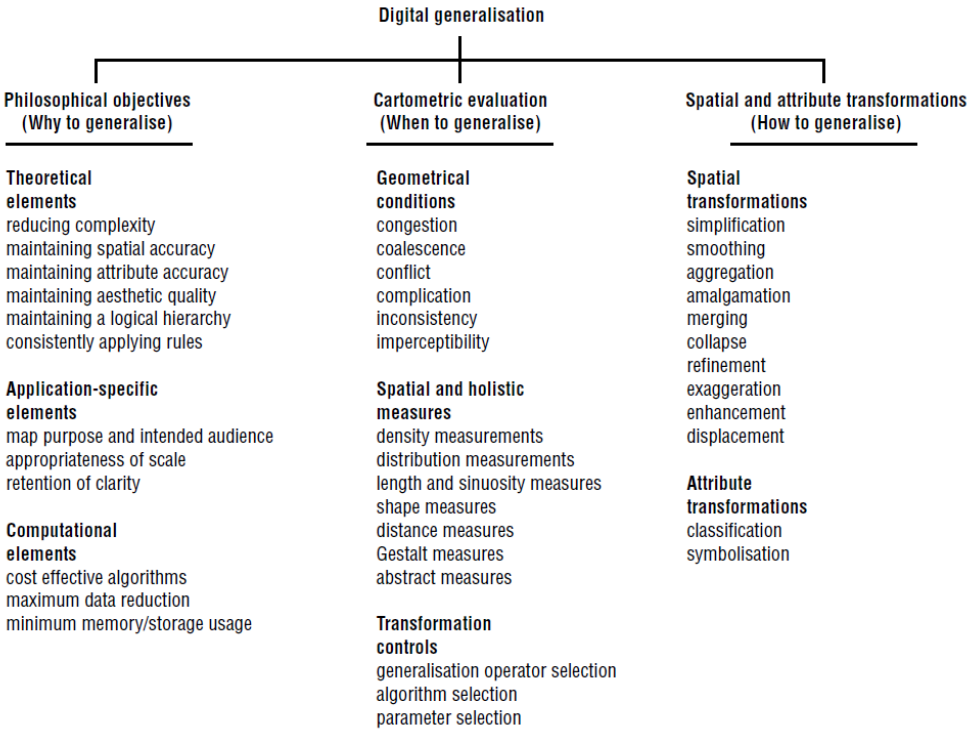


Figura 2.2 – Framework per la *map generalization* proposto da McMaster e Shea (immagine presa da [4]

In particolare, il framework suddivide il processo di generalizzazione in tre aree: gli obiettivi della generalizzazione (il *perché*), la valutazione delle condizioni

geometriche e geografiche che suggeriscono di generalizzare (il *dove*) e la selezione di appropriati operatori, algoritmi e funzioni che risolvano i problemi messi in evidenza (il *come*).

2.2 Operatori di *map generalization*

Ci si concentra qui in particolare sul terzo ramo del framework proposto da McMaster e Shea: il *come* svolgere la generalizzazione, e quindi un elenco di operatori.

Si osservi in particolare qui la differenza tra *operatore* e *algoritmo*: l'operatore definisce la trasformazione dei dati che si vuole raggiungere (l'obiettivo, se si vuole), mentre l'algoritmo è la funzione che implementa effettivamente la particolare trasformazione [4]. Questo significa che gli operatori sono indipendenti dal modello dei dati utilizzato (dati raster o dati vettoriali, ad esempio), mentre gli algoritmi sono necessariamente dipendenti dalla specifica rappresentazione scelta.

Non solo esistono diversi operatori di *map generalization*, ma esistono anche diverse classificazioni di questi operatori, che non concordano tra di loro. Per esempio, gli autori dell'articolo [2] hanno confrontato tredici diverse liste di operatori, formulate tra il 1962 e il 2007, compreso quello di McMaster e Shea [7], e hanno osservato, come anche altri in precedenza [9], che i diversi esperti di *map generalization* non fanno uso di un lessico comune e molti termini presenti in letteratura sono utilizzati in più modi, a volte contraddittori. Inoltre, diverse classificazioni di operatori sono incomplete, in quanto in alcune mancano degli operatori, e le definizioni spesso non sono sufficientemente chiare o si sovrappongono con altre definizioni [4].

Di conseguenza, un qualsiasi elenco di operatori è destinato ad essere incompleto e non condiviso dall'intera comunità degli studiosi sull'argomento, e perciò non si desidera in questo contesto tentare di fornire una panoramica completa degli operatori esistenti. Tuttavia, con il solo scopo di dare un'idea al lettore, si fornisce di seguito un breve elenco parziale, mentre nella figura 2.3 le diverse operazioni sono mostrate visivamente (le immagini sono prese da [2]).

Selection, filter, omission, eliminate Rimuovere da una mappa alcuni elementi geografici, che possono essere sia interi *layer* (come quello geologico, quello delle strade o la posizione delle città), sia singoli elementi (per esempio: tra le milioni di città nel mondo, quali cinquanta mostrare in una cartina mondiale).

Simplify Data una linea o un perimetro di un'area, rimuovere alcuni vertici che la compongono, pur mantenendo inalterata la forma *overall*.

Smoothing Data una linea o un perimetro di un'area, darle un aspetto meno *rough*, più curvo e meno angolato, anche aggiungendo punti;

Merge, combine Sostituire alcune *feature* simili con un unico elemento che li rappresenti che abbia la stessa dimensione (per esempio, convertire tanti edifici vicini in un unico edificio che rappresenti l'intero complesso abitativo);

Collapse, symbolize Sostituire una *feature* con un elemento che lo rappresenti che abbia una dimensione minore (per esempio, trasformare una città bi-dimensionale in un punto mono-dimensionale).

Displace Spostare leggermente la posizione di un oggetto rispetto a dove dovrebbe essere in merito alla scala, per migliorarne la visualizzazione;

Enhance Aggiungere dettagli a una *feature* per renderla maggiormente visibile;

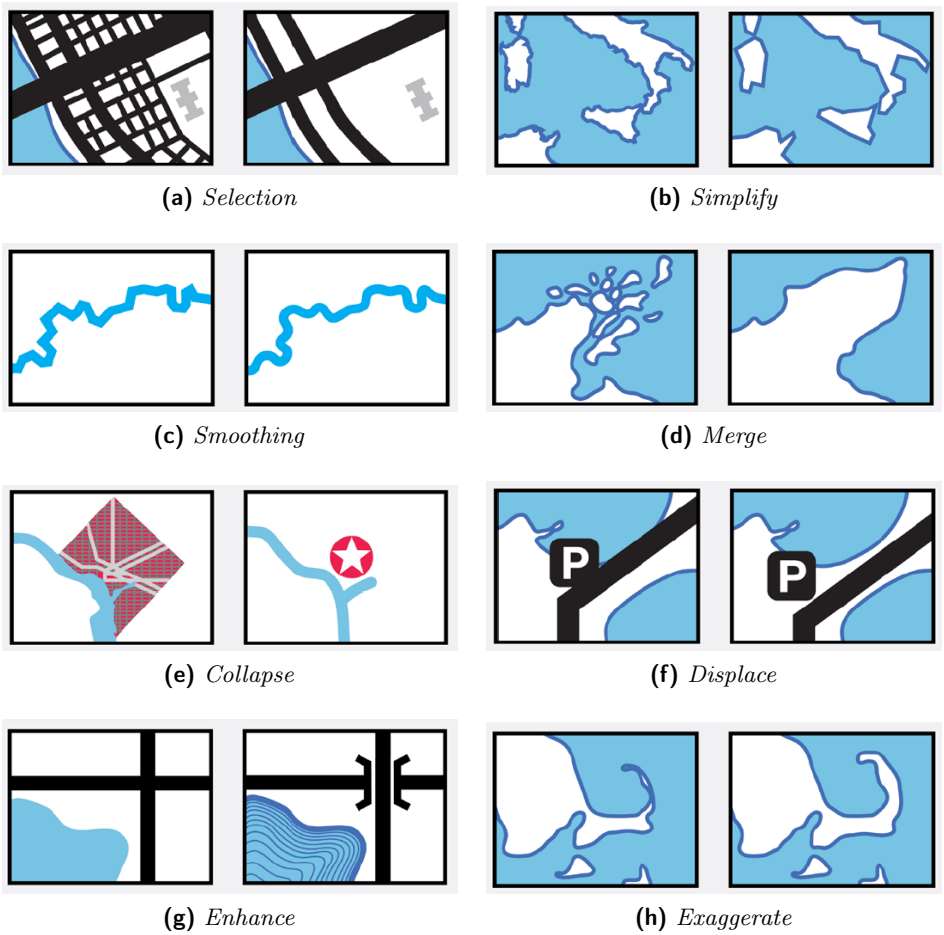
Exaggerate Amplificare una porzione di una *feature* rispetto a quanto dovrebbe essere, per renderla maggiormente visibile ed enfatizzare una sua particolare caratteristica.

2.3 *Simplification*

Tra i vari operatori di *map generalization*, di cui alcuni sono stati presentati nella sezione 2.2, l'unico che si può considerare realmente *condiviso*, essendo presente in tutte le classificazioni prese in considerazione da [2], è l'operatore di *simplification*, o semplificazione. Questo, insieme al grande numero di algoritmi disponibili per svolgere questa operazione, mostra la grande importanza che ha questo operatore nel contesto della *map generalization*.

2.3.1 Definizione

L'operatore di semplificazione è tra i più utilizzati e riconosciuti in letteratura, e la sua definizione è cambiata nel tempo. Se infatti le prime classificazioni di operatori, come [10] (1962), [11] (1974), [12] (1978) e [13] (1987) si mantengono molto vaghe nella definizione, descrivendo la semplificazione come un'operazione che elimina dettagli non necessari da una *feature* pur mantenendone il carattere generale, le classificazioni più recenti, come [14] (1989), [8] (1992), [15] (2005), [16]



(a) *Selection* **(b)** *Simplify*

(c) *Smoothing* **(d)** *Merge*

(e) *Collapse* **(f)** *Displace*

(g) *Enhance* **(h)** *Exaggerate*

Figura 2.3 – Alcuni operatori di *map generalization*

(2007) e [17] (2007), tendono invece a considerare una definizione più precisa, per cui l'operatore di semplificazione deve esplicitamente *ridurre il numero di punti* che costituiscono una linea o il perimetro di un poligono.

In queste classificazioni più recenti si trova anche una distinzione tra l'operazione di semplificazione, che ha lo scopo di rimuovere i punti, e l'operazione di *smoothing*, con cui è a volte confusa, che invece ha l'obiettivo di rimuovere le piccole variazioni nella geometria di una *feature* per migliorarne l'aspetto, rendendolo meno angolato e meno frattale. Sebbene l'idea dietro allo *smoothing* possa sembrare simile alla semplificazione, in realtà questa operazione di solito viene effettuata *aggiungendo* punti: per esempio, creando punti intermedi, o permettendo che due o più punti si connettano in maniera non lineare [2]

2.3.2 Algoritmi

Il contesto della *line simplification*, in particolare per dati vettoriali, è notoriamente l'area della *map generalization* più studiata e per cui esistono più algoritmi [2] [4]. Questo è giustificato dal fatto che le linee, e in particolare le *polylines* (linee spezzate costituite da più segmenti connessi tra loro) sono gli elementi geografici più importanti e comuni, ma possono avere comunque forme molto complesse, spesso di natura frattale (si pensi ad esempio alle anse di un fiume, o a una linea di costa), e perciò è importante costruire algoritmi che semplifichino adeguatamente la linea mantenendo le anse principali sia in termini di posizione del punto di curvatura, sia in termini di contrasto nel grado di curvatura [18].

Tra i tanti algoritmi che sono stati formulati, il più *naive* è sicuramente il cosiddetto algoritmo *n-th point*, che consiste sostanzialmente nel mantenimento del primo punto, dell'ultimo punto e dell'n-esimo punto di una linea: se ne può vedere un esempio di applicazione in figura 2.4. È un algoritmo estremamente efficiente (tempo di computazione $O(n)$), ma non buono dal punto di vista dell'accuratezza, non prendendo in considerazione nessuna informazione sulla curvatura della linea.

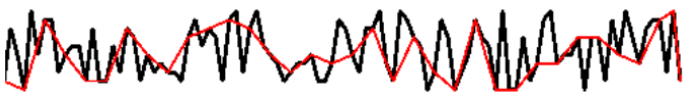


Figura 2.4 – Algoritmo *n-th point*

Come si può immaginare, l'*n-th point algorithm* non è tra gli algoritmi più diffusi e utilizzati; in questo gruppo invece rientrano i quattro approcci messi a disposizione dalla funzione `SimplifyPolygon` di arcGIS [19], mostrati nella

figura 2.5, in cui sono applicati a una *polyline* chiusa: gli algoritmi Douglas-Peucker, Visvalingam-Whyatt, Wang-Muller e Zhou-Jones.

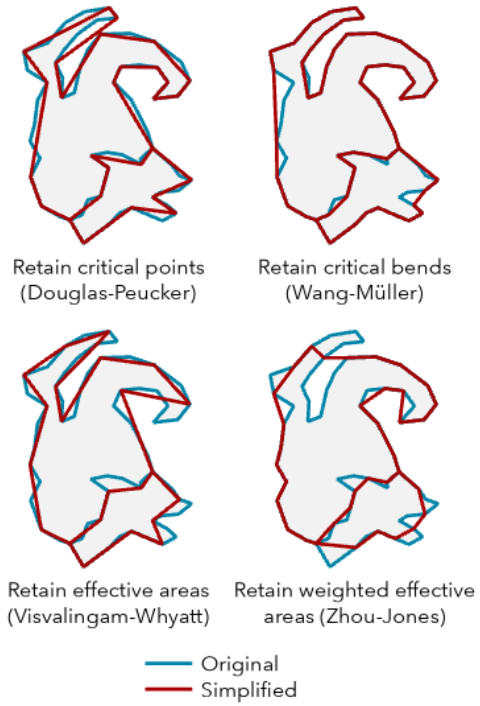


Figura 2.5 – Approcci più utilizzati di *line simplification*

Di seguito si fornirà una breve descrizione dei primi due algoritmi; nella successiva parte della sezione, invece, si farà una breve descrizione degli altri due, e si completerà la sezione con una breve carrellata non esaustiva di altri algoritmi di *line simplification* trovati in letteratura.

2.3.2.1 Douglas-Peucker algorithm

Il metodo di *line simplification* di gran lunga più famoso, conosciuto e utilizzato è l'algoritmo di Douglas-Peucker [20], talvolta chiamato anche Ramer-Douglas-Peucker perché Ramer ne ha ideato uno simile più o meno nello stesso momento [21]. Oltre ad avere una implementazione in arcGIS e in QGIS, è già presente anche una implementazione di questo algoritmo fatta in C++ [22] e in Python [23]. L'algoritmo procede nel seguente modo:

- Si parte connettendo il primo punto A e l'ultimo punto B della linea;
- Si calcolano le distanze tra la linea e tutti i vertici intermedi, e si considera in particolare quello più distante C;

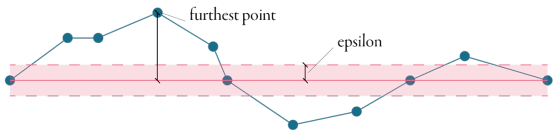


Figura 2.6 – Algoritmo Douglas-Peucker (immagine presa da [24]).

- Se quello più distante è entro una tolleranza massima ε, l'algoritmo può terminare;
- Se invece quello più distante è oltre una tolleranza massima, viene aggiunto alla semplificazione, così che ora la linea semplificata sia composta dai punti A, C, B.
- Si considerano i segmenti AC e CB come il nuovo segmento AB: si calcolano le distanze tra la linea e tutti i vertici intermedi, e si considerano quelli più distanti, eccetera.
- L'algoritmo termina quando tutti i vertici della linea spezzata originale sono distanti al massimo ε dalla linea semplificata.

Il primo e il secondo step dell'algoritmo sono rappresentati nella figura 2.6.

L'algoritmo è molto rapido e permette di semplificare facilmente una linea mantenendo i cosiddetti *punti critici*, cioè i punti più distanti rispetto alla baseline; tuttavia, a volte può generare linee che si auto-intersecano, e per questo sono state sviluppate altre versioni che correggono questo problema [25], anche se si sconsiglia di utilizzarlo su linee troppo complesse (come quelle di costa).

2.3.2.2 Visvalingam-Whyatt algorithm

Un altro degli algoritmi più noti e conosciuti per la procedura di *line simplification*, spesso accostato e confrontato con il Douglas-Peucker algorithm descritto nella sezione 2.3.2.1, è l'algoritmo di Visvalingam-Whyatt [26], che procede nel seguente modo:

- Si considerano tutti i triangoli sottesi dai punti della linea considerati a tre a tre;
- Si calcola l'area di ognuno e si prende il punto che costituisce il vertice centrale del triangolo di area minima;
- Se l'area del triangolo è minore di una certa soglia ε, il punto centrale del triangolo viene eliminato;

- I triangoli vengono poi ricalcolati, e si cerca di nuovo il triangolo di area minima, e così via.

Una rappresentazione del calcolo di tutti i triangoli sottesi dai punti della linea è mostrata in figura 2.7.

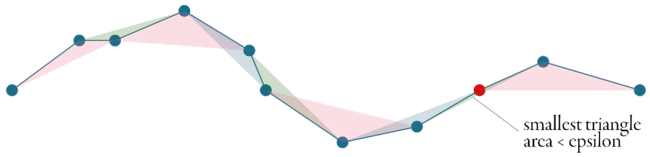


Figura 2.7 – Algoritmo di Visvalingham-Whyatt (immagine presa da [24]).

2.3.2.3 Altri algoritmi

Tra gli altri algoritmi molto citati in letteratura, si ritrova l'algoritmo di Li-Openshaw [27], basato sulla definizione dello *Smallest Visible Object*, tale che se ci fosse un oggetto più piccolo sullo schermo, il nostro occhio lo percepirebbe come un semplice punto; l'algoritmo di Wang-Muller [28], che riconosce le curvature più importanti ed elimina quelle meno importanti e l'algoritmo di Zhou-Jones [29], una correzione dell'algoritmo di Visvalingam-Whyatt che prende in considerazione anche diverse metriche di valutazione delle aree.

Tuttavia, oltre ai più famosi, molti altri algoritmi sono stati sviluppati e testati su vari dataset.

Per esempio, l'algoritmo in [30] è sicuramente tra i più vecchi sviluppati (1969). Gli algoritmi in [31] e [32] considerano i punti due a due e considerano se ci sono punti da rimuovere nelle *strip* che partono dal prolungamento del segmento costituito dai punti, con eventualmente un constraint di lunghezza, L'algoritmo [33] che usa un approccio topologicamente coerente per semplificare i punti di un poligono. Il metodo di [34] utilizza un approccio basato sui minimi quadrati per rimuovere i punti. L'idea di Perkal [35] di utilizzare un *ε-cicle rolling approach* è stata utilizzata anche da [36] e da [37]. Alcuni algoritmi sfruttano la *Delaunay triangulation* per i punti che compongono la linea da semplificare, ed effettuano la semplificazione in base ai triangoli che si formano nel processo: tra questi, ci sono [38], [39] e [40]. Un altro approccio esplorato, sempre geometrico, è quello che sfrutta i diagrammi di Voronoi, come quello descritto in [41]. Si trovano algoritmi basati sul metodo *oblique-dividing curve*, in cui una curva dalla forma simile a una sinusoide viene approssimata con una spezzata obliqua dipendente dalla tangente a ogni ansa della sinusoide [42] e tecniche basate sulle SOM [43].

Sono stati studiati approcci ibridi che mescolano la *line simplification* con altre operazioni di *map generalization*, come lo *smoothing* o l'*enhancement* [44] [45]. Recentemente sono stati sviluppati anche nuovi algoritmi per le immagini raster, che convertono le immagini in bianco e nero e, tramite operazioni di *image processing*, riconoscono il boundary dell'immagine e identificano i punti da mantenere e da eliminare [18].

La lista termina qui, ma non è da considerarsi comprensiva: si spera di essere riusciti a dare l'idea della vastità della letteratura sull'argomento.

2.3.2.4 Considerazioni

Poiché l'argomento è stato così tanto trattato, è difficile sviluppare un metodo realmente innovativo, e questo lavoro di tesi non aspira a farlo – e anzi, come si vedrà nel capitolo 3, in realtà i metodi suggeriti per effettuare la *line simplification* sono piuttosto elementari.

Tuttavia, si osserva che la stragrande maggioranza dei metodi di *line simplification* sviluppati si basano sul task di semplificare linee rappresentanti *feature* naturali, come fiumi e linee di costa: meno metodi sono specifici per forme artificiali, come costruzioni o, in questo caso, impianti industriali. Questo probabilmente è dovuto al fatto che, essendo le figure meno *rough*, sono relativamente più facili da gestire, e di conseguenza non sono necessari approcci particolarmente complicati. Tuttavia, il task risulta comunque di interesse aziendale, e per questo motivo in questo lavoro ci si è focalizzati su questo obiettivo, con lo scopo di trovare la migliore semplificazione possibile per il dataset che è stato fornito.

Capitolo 3

Descrizione del codice sviluppato

Nel capitolo che segue, si fornirà una estesa descrizione del codice sviluppato, di come funziona, delle funzioni chiamate e degli algoritmi implementati e utilizzati. Il codice è stato raccolto in un pacchetto di Python chiamato *GeoSampling*, ed è disponibile in una cartella pubblica di GitHub al link [1] e, di conseguenza, non sarà riportato interamente nel presente lavoro di tesi.

3.1 Ambiente di sviluppo

Si è scelto di lavorare in Python [46], un linguaggio di programmazione di alto livello orientato agli oggetti open-source di tipo *general-purpose*, estremamente diffuso in molti contesti e in particolare in quello della gestione, manipolazione e analisi dei dati.

In particolare, tra le varie distribuzioni di Python, si è scelto di utilizzare la distribuzione di *Anaconda* [47], molto diffusa nel contesto della Data Science, che oltre a Python installa automaticamente anche una serie di pacchetti e librerie utili per l'analisi dei dati, tra cui Numpy, Pandas, Scikit-Learn, Matplotlib, SciPy e TensorFlow, e gli ambienti di lavoro di Spyder e Jupyter Notebook. Tra i due, si è preferita l'opzione del *Jupyter Notebook* per la sua interattività e rapidità nel visualizzare i risultati del codice, che compensa le minori e meno *user-friendly* funzionalità di *debugging*

Il codice è stato quindi sviluppato e testato su diversi notebook di Jupyter (`.ipynb`), e infine è stato formalizzato e salvato in file Python (`.py`) scritti utilizzando l'editor *Visual Studio Code*, file che poi sono stati condivisi su GitHub. Non sono stati invece condivisi i Jupyter Notebook, per evitare di condividere dati confidenziali.

Il codice fa uso di diverse librerie standard, di cui alcune presenti nella *Python Standard Library* (che quindi non necessitano di installazione), altre già presenti all'interno della distribuzione di Anaconda, e altre ancora appositamente scaricate per il progetto. Di seguito, l'elenco delle librerie esplicitamente importate dai codici:

- *Math* [48], libreria appartenente alla *Python Standard Library* contenenente semplici funzioni matematiche;
- *Itertools* [49], libreria appartenente alla *Python Standard Library* contenente funzioni per generare iteratori specifici e ulteriori rispetto a quelli generalmente permessi da Python, che possono ciclare per esempio sulle permutazioni di una lista;
- *Collections* [50], libreria appartenente alla *Python Standard Library* contenente funzioni per generare container ulteriori rispetto a quelli generalmente permessi da Python (liste, dizionari, insiemi), tra cui per esempio un dizionario che si ricorda l'ordine di inserimento degli elementi e lo mantiene;
- *Copy* [51], libreria appartenente alla *Python Standard Library* contenente funzioni per effettuare copie *deep*, che non sono quindi solo un riferimento all'oggetto di partenza, ma che costruiscono nuovi oggetti che sono copie di quelli di partenza;
- *NumPy* [52], libreria presente in Anaconda dedicata all'algebra lineare, che introduce strutture dati utili per gestire vettori e matrici e metodi per manipolarli;
- *Pandas* [53], [54], libreria presente in Anaconda dedicata alla manipolazione dei dati tabellari, che introduce le strutture dati delle serie e dei dataframe e metodi per creali, selezionare dati, manipolarli, gestirli, eccetera;
- *Scikit-learn* [55], libreria presente in Anaconda dedicata ai modelli di Machine Learning, che introduce strutture dati e metodi per tutti i principali e più accreditati algoritmi per il Machine Learning;
- *MatplotLib* [56], libreria presente in Anaconda dedicata alla creazione di visualizzazioni, che introduce strutture dati e metodi per generare plot, istogrammi, boxplot e tanto altro;
- *Seaborn* [57], libreria presente in Anaconda basata su Matplotlib, che fornisce un'interfaccia high-level per generare visualizzazioni a partire da dataframe di Pandas;

- *Shapely* [58], libreria appositamente scaricata di geometria euclidea, costruita per creare e manipolare oggetti geometrici (punti, linee, poligoni) nel piano cartesiano.

3.2 Figura di esempio

Un aspetto importante dello sviluppo del codice qui descritto è che il codice è stato sviluppato per gran parte facendo riferimento a *una sola figura*, rappresentata nell'immagine 3.1 (con coordinate anonimizzate). L'algoritmo ha quindi seguito una tecnica di rimozione dei punti che ha lavorato *figura per figura*, concentrandosi in particolare sulle figure a molti punti. L'intero dataset è stato fornito successivamente, e grazie ad esso sono state effettuate alcune correzioni e alcuni *debug* delle funzioni, ma non è stata modificata la struttura fondamentale che esegue il task richiesto di sampling.

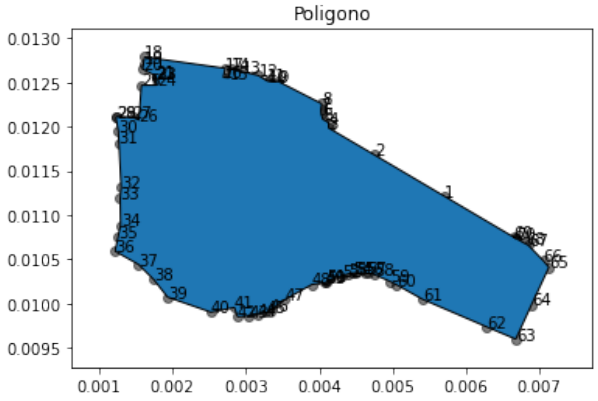


Figura 3.1 – Figura di esempio

Questa figura ha le seguenti caratteristiche, che sono state prese in considerazione per lo sviluppo del codice:

- Ha $N = 70$ punti;
- A parte qualche imprecisione sui bordi, è fondamentalmente *convessa*.
- Molti dei punti che la costituiscono sono superflui e facilmente eliminabili, perché sono sovrapposti ad altri punti o si trovano sulla stessa linea definita da altri punti;
- Si possono identificare diversi gruppi di punti a densità elevata che si possono sostituire con un unico punto, distanziati da altri punti relativamente isolati.

Nel seguito, tutte le spiegazioni delle funzioni saranno corredate da un'immagine che fornirà un esempio di funzionamento della funzione, che riguarderà sempre questa figura di esempio.

3.3 Struttura del pacchetto

Come si è detto, tutte le funzioni sviluppate sono state raccolte in un apposito pacchetto chiamato *GeoSampling*, pubblicato su GitHub al link [1].

Si consideri tuttavia che il pacchetto, sebbene sia stato condiviso, non è pronto all'utilizzo da parte della community dei programmatori in Python, per le seguenti ragioni:

- Il pacchetto è commentato e presenta una *docstring* per ogni funzione, ma la lingua in cui questi elementi sono scritti è l'italiano;
- Il pacchetto è dotato di un *readme* in inglese molto minimale;
- Nessuna funzione presenta controlli di *input check* che verificano la consistenza dei parametri di input, e si assume dunque che l'utilizzatore conosca bene le funzioni richieste;
- Non è stato diffuso l'elenco di pacchetti necessari con le relative versioni;
- Non è stata prodotta nessuna documentazione per il codice, escluso (se si vuole) questo lavoro di tesi;
- Non sono stati resi pubblici dei file di esempio sul funzionamento del pacchetto.

Nell'immagine 3.2 sono schematizzati i file Python e le funzioni che costituiscono il pacchetto sviluppato.

Come si vede dal codice dei colori dell'immagine 3.2, le funzioni contenute nel pacchetto riguardano quattro *topic* fondamentali:

- C'è una parte dedicata al *data import* (arancione), che contiene la funzione `db_preparation` che effettua un *preprocessing* dei dati iniziali per trasformarli nel formato di input che le funzioni sviluppate successivamente si aspettano; e la funzione `anonimization` che ha l'obiettivo di anonimizzare le coordinate dei punti di un impianto industriale per poterle mostrare pubblicamente (ad esempio, in questo lavoro di tesi);

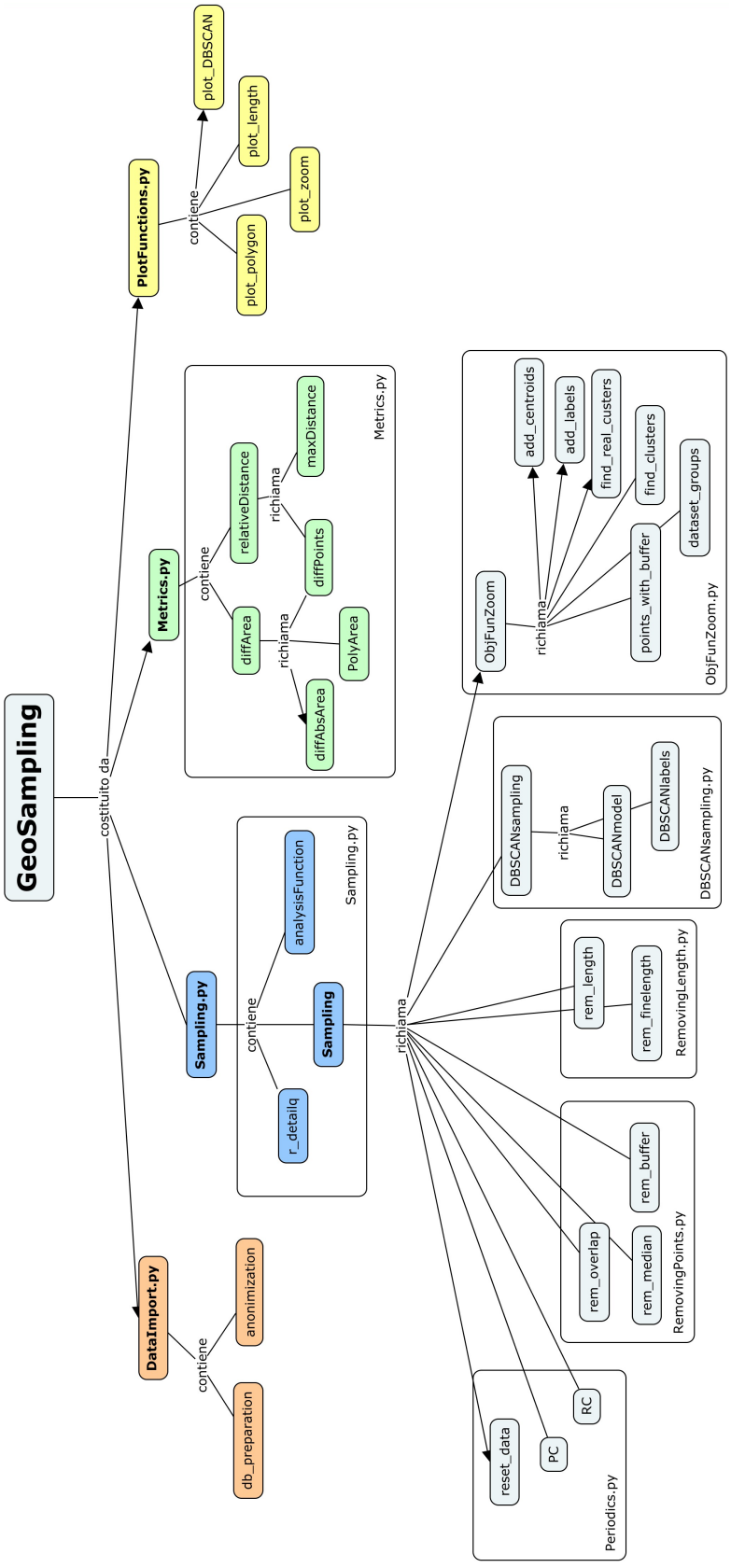


Figura 3.2 – Schema delle funzioni del pacchetto *GeoSampling*

- C'è una parte dedicata al *plotting* (giallo), cioè funzioni che fanno uso di Matplotlib e Seaborn per plottare il poligono (*plot_polygon*) e per plottare alcune anteprime dei metodi di sampling: *plot_zoom*, che rappresenta attorno a ogni punto un cerchio di raggio pari al dettaglio indicato (utile per il metodo dello zoom, si veda la sezione 3.4.4.6); *plot_length*, che rappresenta un istogramma delle lunghezze dei segmenti, utile per calcolare il dettaglio (si veda la sezione 3.5); e *plot_DBSCAN*, che mostra i risultati del clustering attuato dal DBSCAN (utile per il metodo del DBSCAN, si veda la sezione 3.4.4.6);

- C'è una grossa parte dedicata al *sampling* (azzurro), cioè le funzioni che effettuano il sampling, che saranno descritte estensivamente nella sezione 3.4;

- C'è una sezione dedicata alle *metriche* (verde), cioè le funzioni che valutano la bontà del sampling effettuato, che saranno descritte nella sezione 3.6.

Nelle sezioni successive di questo capitolo, si descriveranno le parti dedicate al *sampling* (nelle sezioni 3.4 e 3.5) e alle *metriche*, nella sezione 3.6; nel capitolo 4, invece, si descriverà brevemente la funzione di *data import* utilizzata per processare il dataset fornito.

3.4 La funzione di sampling

La parte principale del codice è la funzione *Sampling*, che si trova nel file Python `Sampling.py`.

3.4.1 Parametri di input

La funzione ha la seguente intestazione:

```
def Sampling(dati, detail = 10, par_identity = False,
        clustering = False, ifzoom = False, par_zoomed = 0.5,
        par_buffer = 0.5, par_length = 2,
        finelength = True, minPoints = 4, analysis = False)
```

La funzione si aspetta in input un dataframe di *Pandas*, qui chiamato `dati`, che contiene le coordinate spaziali degli N punti che costituiscono il poligono. In particolare, il dataset atteso è così strutturato:

- Ha $N + 1$ righe, indicizzate come $0, 1, \ldots, N$, rappresentanti ciascuno un punto; in particolare, la riga 0 è uguale alla riga N, e l'ultimo punto effettivo corrisponde alla riga $N - 1$;

- I punti del dataset sono considerati come *ordinati*, nel senso che il punto corrispondente all'indice i e il punto corrispondente all'indice $i + 1$ sono considerati come adiacenti. Tuttavia, l'ordinamento è invariante rispetto a una permutazione ciclica: considerando la sequenza di punti consecutivi $ABCD$, si ha che i seguenti ordinamenti sono equivalenti:$BCDA$, $CDAB$, $DABC$.

- Ha *due* colonne: una colonna chiamata *Latitude*, contenente la latitudine dei punti, indicata in gradi, dove la latitudine Nord è positiva e la latitudine Sud è negativa; e una colonna chiamata *Longitude*, contenente la longitudine dei punti, anche questa indicata in gradi, dove la longitudine Est è positiva e la longitudine Ovest è negativa.

A parte il dataframe di input, nella tabella 3.1 sono riassunti tutti i parametri della funzione, di cui ciascuno è modificabile.

3.4.2 Descrizione *high-level*

La funzione opera nel seguente modo:

- Il parametro di input più importante è `detail`, un valore in metri che indica il «livello di dettaglio» del sampling: sostanzialmente, è la distanza entro la quale due punti sono considerati come se fossero lo stesso punto.

- Innanzitutto i dati vengono resettati, in modo da far sì che, se il poligono rappresentato ha N punti; il dataset abbia N punti: questo viene fatto tramite la funzione *`reset_data`* del modulo `Periodics.py`, che rimuove l'ultima riga del dataset, che come si è detto ci si aspetta possegga come ultima riga una copia del primo punto.

- Vengono per prima cosa rimossi i punti sovrapposti. La funzione che effettua questa rimozione si chiama *`rem_overlap`* e sarà descritta nella sezione 3.4.4.2.

- Se il parametro `clustering` è stato posto a `True`, viene applicato un metodo di *clustering* dei punti: i punti vengono raggruppati in alcune zone ad alta densità, riassumibili con il punto più vicino al loro centroide, separate da alcuni punti isolati, lasciati inalterati. Le funzioni che applicano questo metodo si chiamano *`DBSCANsampling`* e *`ObjFunZoom`*, e sono descritte nella sezione 3.4.4.6. In entrambi questi casi, il raggruppamento dei punti dipende dal dettaglio moltiplicato per il parametro `par_zoomed`.

Tabella 3.1 – I parametri della funzione `Sampling`

Parametro	*Default*	*Che cosa modifica*
`detail`	`10`	Livello di dettaglio del sampling, indicato in metri.
`par_identity`	`False`	Variabile booleana che indica se la funzione che rimuove i punti sovrapposti, descritta nella sezione 3.4.4.2, deve effettuare un check che richiede un'uguaglianza esatta delle coordinate, o un'uguaglianza dei loro arrotondamenti a quattro cifre decimali.
`clustering`	`False`	Variabile booleana che indica se si desidera applicare uno dei due metodi di clustering, descritti nella sezione 3.4.4.6.
`ifzoom`	`False`	Variabile booleana che, se `clustering` è posto a `True`, indica se si vuole utilizzare come funzione di clustering il metodo dello zoom (parametro posto a `True`), oppure il metodo basato sul DBSCAN (parametro posto a `False`).
`par_zoomed`	`0.5`	Numero che indica un parametro moltiplicativo del dettaglio, specifico per le funzioni di rimozione dei punti che utilizzano il metodo del clustering, spiegato nella sezione 3.4.4.6.
`par_buffer`	`0.5`	Numero che indica un parametro moltiplicativo del dettaglio, specifico per la funzione di rimozione dei punti che utilizza il metodo del buffer, spiegato nella sezione 3.4.4.4.
`par_length`	`2`	Numero che indica un parametro moltiplicativo del dettaglio, specifico per la funzione di rimozione dei punti che utilizza il metodo della lunghezza dei segmenti, spiegato nella sezione 3.4.4.5.
`finelength`	`True`	Variabile booleana che, per quanto riguarda la funzione di rimozione dei punti che utilizza il metodo della lunghezza dei segmenti, indica se bisogna usare la funzione che effettua la rimozione grezza, o la rimozione fine, spiegate nella sezione 3.4.4.5.
`minPoints`	`4`	Numero che indica il minimo dei punti da mantenere.
`analysis`	`False`	Variabile booleana che indica se, oltre ad effettuare il sampling, si desidera plottare il risultato di ogni fase del sampling.

- Secondariamente, vengono rimossi i punti che si trovano in corrispondenza della media dei loro punti adiacenti, tramite la funzione *rem_median*, descritta nella sezione 3.4.4.3.

- In seguito, vengono rimossi i punti che si trovano all'interno di un *buffer* dei loro punti adiacenti, buffer che ha un raggio pari al dettaglio moltiplicato per il parametro `par_buffer`. La funzione che effettua questa rimozione si chiama *rem_buffer* ed è descritta nella sezione 3.4.4.4.

- Vengono poi rimossi i punti che costituiscono un segmento troppo corto, per cui la lunghezza minima è determinata dal dettaglio moltiplicato per il parametro `par_length`. Le funzioni che effettuano questa rimozione si chiamano *rem_length* e *rem_finelength* e sono descritte nella sezione 3.4.4.5.

- Viene infine effettuata una ulteriore iterazione della funzione *rem_buffer*, descritta nella sezione 3.4.4.4, per effettuare alcune ultime correzioni al sampling.

Nel caso in cui un qualunque metodo di rimozione restituisca un poligono con meno di `minPoints` punti (dove `minPoints` è stato posto pari a 4), il metodo viene scartato, e i punti che costituiscono il poligono vengono resettati allo stato che avevano prima dell'applicazione del metodo. Infine, il parametro `analysis` si utilizza in fase di testing per mostrare i risultati delle successive applicazioni delle funzioni.

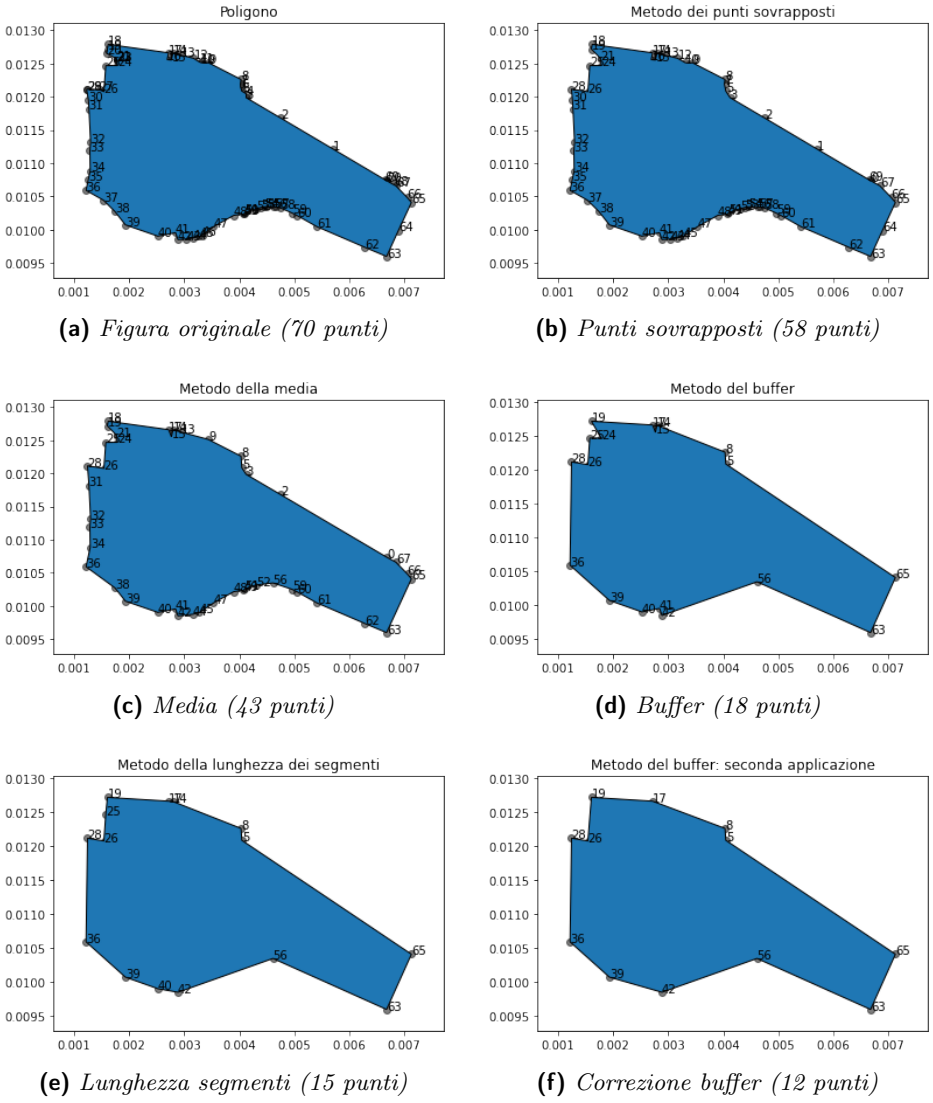
Nella sezione 3.4.4 si spiegheranno nel dettaglio le funzioni che permettono queste rimozioni di punti.

3.4.3 Risultati sulla figura di esempio

Per dare un'idea del risultato del procedimento di sampling sulla figura di esempio, si può fare riferimento all'immagine 3.3, in cui la funzione *Sampling* è stata applicata con i parametri di default descritti nella tabella 3.1, e ponendo `detail` pari a 20 metri.

Nella tabella 3.2, sono invece riassunti i risultati numerici dei punti tolti dal sampling, dove:

- *Step* contiene un nome riassuntivo dello step effettuato. Nelle formule che seguono, per brevità, il nome sarà sostituito dall'indice i, con $i = 0, 1, \ldots, 5$, dove $i = 0$ corrisponde allo step chiamato *Originale*, $i = 1$ corrisponde allo step chiamato *Sovrapposti*, eccetera.



(a) *Figura originale (70 punti)*

(b) *Punti sovrapposti (58 punti)*

(c) *Media (43 punti)*

(d) *Buffer (18 punti)*

(e) *Lunghezza segmenti (15 punti)*

(f) *Correzione buffer (12 punti)*

Figura 3.3 – Applicazione della funzione ***`Sampling`*** alla figura di esempio, con i parametri di default e `detail = 20`

- *Punti step* indica il numero di punti che la figura ha una volta eseguito quello step. Nelle formule che seguono, questo valore sarà indicato con v_i, dove i è l'indice corrispondente allo step;

- *Punti rimossi step* indica quanti punti sono stati rimossi rispetto allo step precedente. Nelle formule che seguono, questo valore sarà indicato con r_i, e si ha che:

$$r_i = v_{i-1} - v_i$$

- *Punti rimossi totali* indica quanti punti sono stati rimossi fino a quel punto. Nelle formule che seguono, questo valore sarà indicato con t_i, e si ha che:

$$t_i = t_{i-1} + r_i$$

Tabella 3.2 – Punti rimossi dal sampling sulla figura di esempio, con i parametri di default e `detail = 20`

Step	*Punti step*	*Punti rimossi step*	*Punti rimossi totali*
Originale	70	-	-
Sovrapposti	58	12	12
Media	43	15	27
Buffer	18	25	52
Lunghezza	15	3	55
Correzione	12	3	58

Come si può vedere, la figura di esempio è strutturata in modo tale che già le rimozioni dei punti sovrapposti (*Sovrapposti*) e dei punti che si trovano sulla stessa linea di altri punti (*Media*) permettono di effettuare un sampling significativo (vengono rimossi 27 punti, il 38.5% del totale); tuttavia il grosso della rimozione viene svolto dal metodo del buffer, che da solo rimuove 25 punti (il 35.7% del totale), mentre gli ultimi due step rimuovono meno punti ma completano il sampling, restituendo una figura pulita. Alla fine il sampling ha rimosso 58 punti (l'83% del totale), e la figura ottenuta è qualitativamente simile, nella forma, a quella originale. Quando si descriveranno le metriche costruite nella sezione 3.6, mostrerà anche un esempio di applicazione su questo sampling.

3.4.4 Funzioni di rimozione dei punti

In questa sezione saranno descritte estensivamente tutte le funzioni scritte di rimozione dei punti.

Si userà in particolare questa convenzione: queste funzioni suddividono l'insieme dei punti di un poligono nelle categorie «da rimuovere» e «da mantenere», secondo un criterio che verrà descritto, e poi eliminano dall'elenco dei punti che costituiscono il poligono i punti considerati come «da rimuovere».

3.4.4.1 Scelte comuni

Come si vedrà nel seguito, la maggioranza delle funzioni descritte per la rimozione dei punti opera direttamente sulle coordinate dei punti, che sono espresse in gradi. Perciò, quando è stato necessario, si sono effettuate le necessarie conversioni da metri a gradi e da gradi a metri. La conversione considera un modello terrestre di sfera perfetta, e avviene secondo le seguenti formule:

$$L = \frac{2\pi RA}{360} \tag{3.1}$$

$$A = \frac{360L}{2\pi R} \tag{3.2}$$

essendo L una lunghezza in metri, A una lunghezza in gradi ed $R = 6371$ Km $= 6,371 \times 10^6$ m il raggio terrestre.

Inoltre, la maggioranza delle funzioni descritte segue un metodo *iterativo*, che considera i punti a gruppi di n, dove tipicamente $n = 2$ oppure $n = 3$, e rimuove il secondo punto. In questi casi, si sono sempre effettuate le scelte di rimozione *in-place* e di condizioni al contorno periodiche, che per via della loro pervasività saranno descritte di seguito. Nel seguito in particolare si considererà un gruppo pari a $n = 2$ punti; tuttavia, valgono considerazioni analoghe per un gruppo pari a $n = 3$ punti.

Rimozione *in-place* Per quanto riguarda la rimozione *in-place*, considerando i punti adiacenti A, B, C, si definisce qui una rimozione *in-place* una rimozione che, se all'iterazione i ha considerato un gruppo di $n = 2$ punti adiacenti A, B e ha rimosso il punto B, all'iterazione $i + 1$ considera un gruppo di $n = 2$ punti adiacenti A, C, eventualmente per rimuovere il punto C. Una rimozione non *in-place* effettuata all'iterazione i richiederebbe invece di considerare, all'iterazione $i+1$, i punti B, C. Si è scelta una rimozione *in-place* per una maggiore coerenza: nell'esempio dell'iterazione $i + 1$, una rimozione non *in-place* considererebbe se rimuovere il punto C basandosi su un punto non più esistente, il punto B.

Condizioni periodiche Le condizioni periodiche sono invece necessarie per evitare di assegnare ruoli privilegiati al primo punto, indicato qui con P_0, e all'ultimo punto, indicato qui con P_{N-1}, per cui la condizione di essere «primo» o «ul-

timo» è arbitraria, essendo l'ordinamento dei punti invariante per permutazioni cicliche.

Infatti, si consideri un ciclo che considera per ogni iterazione i un gruppo di $n = 2$ punti, per eventualmente rimuovere il secondo punto.

Se non venissero imposte le condizioni periodiche, la prima iterazione avrebbe $i = 0$ e considererebbe il gruppo formato dai punti adiacenti P_0, P_1, in cui il punto sotto analisi è il punto P_1; mentre l'ultima iterazione dovrebbe avere $i = N - 2$ e considerare il gruppo formato dai punti adiacenti P_{N-2}, P_{N-1}, mettendo sotto analisi il punto P_{N-2}. I punti P_0 e P_N non verrebbero quindi mai messi sotto analisi.

Per questa ragione, si sono imposte condizioni periodiche: a partire da una dataset di N righe, indicizzate come $0, \ldots, N-1$, che rappresentano un poligono di N punti, si è creato un dataframe di $N+2$ righe, indicizzate come $-1, 0, \ldots, N$, in cui le righe $0, \ldots, N-1$ sono uguali al dataframe di partenza, e in più la riga -1 è uguale alla riga $N-1$ e la riga N è uguale alla riga 0. Questa operazione viene in particolare svolta dalla funzione `PC` del modulo `Periodics.py`.

Inoltre, le condizioni periodiche vengono rimosse alla fine di ogni applicazione delle funzioni di rimozione dei punti, tramite la funzione `RC` del modulo `Periodics.py`, che rimuove la prima e l'ultima riga del dataset.

3.4.4.2 Metodo dei punti sovrapposti

Questa sezione è dedicata a spiegare il funzionamento della funzione `rem_overlap`, che rimuove i punti secondo il *metodo dei punti sovrapposti*, che verrà qui spiegato.

Considerando due punti adiacenti A, B, il *metodo dei punti sovrapposti* qui definito considera il punto B come «da rimuovere» se coincide con il punto A.

In particolare, sono stati introdotti due criteri diversi, distinti dal parametro `identity` (chiamato poi `par_identity` nella funzione `Sampling`).

Infatti, se `identity` è pari a `True`, si considera un criterio di uguaglianza perfetta, fino all'ultima cifra decimale; ossia, se $latA, latB$ sono le latitudini dei punti A, B e $lonA, lonB$ le loro longitudini, si ha che:

$$latA = latB$$
$$lonA = lonB$$

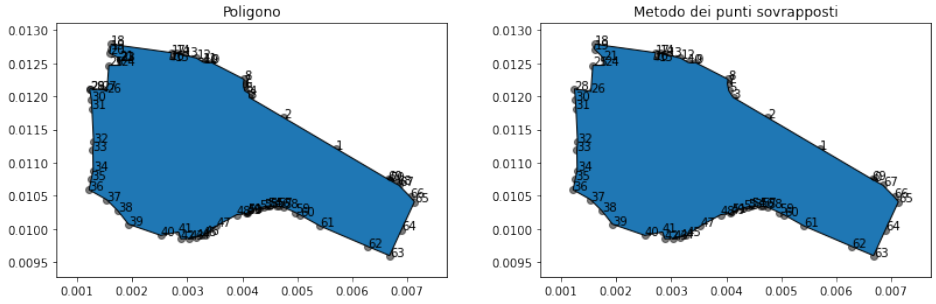
Invece, se `identity` è pari a `False`, si considera un criterio di uguaglianza entro la quarta cifra decimale; ossia, si è deciso di arrotondare ogni numero a una

accuratezza `acc` pari a 4, tramite la funzione ***round***, e si ha quindi che:

$$\texttt{round}(latA) = \texttt{round}(latB)$$
$$\texttt{round}(lonA) = \texttt{round}(lonB)$$

La funzione ***rem_overlap***, quindi, considera il dataset di input, considera tutte le righe a gruppi di due, e rimuove il secondo punto se rispetta questa condizione. Si sono effettuate in particolare le scelte di rimozione *in-place* e di condizioni al contorno periodiche, già spiegate nella sezione 3.4.4.1.

Si osserva in figura 3.4 l'esempio di applicazione del metodo dei punti sovrapposti a partire dalla figura di esempio originale. Come ci si aspetterebbe, la figura è rimasta identica nell'aspetto, e non sono immediatamente visibili i punti rimossi.



(a) *Situazione di partenza* **(b)** *Metodo dei punti sovrapposti*

Figura 3.4 – Applicazione del metodo dei punti sovrapposti alla figura di esempio

3.4.4.3 Metodo della media

Questa sezione è dedicata a spiegare il funzionamento della funzione ***rem_median***, che rimuove i punti secondo il *metodo della media*, che verrà qui spiegato.

Considerando tre punti adiacenti A, B, C, essendo $latA, latB, latC$ le tre rispettive latitudini e $lonA, lonB, lonC$ le tre rispettive longitudini, il *metodo della media* qui definito considera il punto B come «da rimuovere» se, ponendo:

$$medLat = \frac{latA + latC}{2}$$
$$medLong = \frac{lonA + lonC}{2}$$

si ha che:

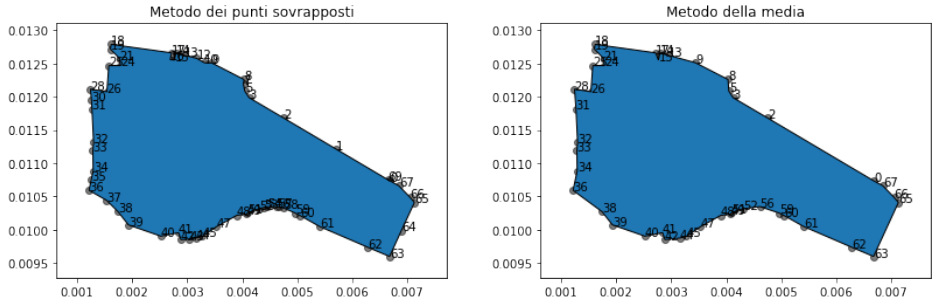
$$latB = medLat$$
$$lonB = medLong$$

Più precisamente, poiché i numeri sono memorizzati in Python con 16 cifre decimali e dunque non si avrà mai una uguaglianza perfetta, perché la condizione sia applicabile computazionalmente parlando, si è deciso di *arrotondare* i valori, tramite la funzione ***round***, a un valore di `acc = 4` cifre decimali, così che di fatto la condizione sia:

$$\texttt{round}(latB) = \texttt{round}(medLat)$$
$$\texttt{round}(lonB) = \texttt{round}(medLong)$$

La funzione ***rem_median***, quindi, considera il dataset di input, considera tutte le righe a gruppi di tre, e rimuove il punto mediano se rispetta questa condizione. Si sono effettuate anche in questo caso le scelte di rimozione *in-place* e di condizioni al contorno periodiche, spiegate nella sezione 3.4.4.1.

Si osserva in figura 3.5 l'esempio di applicazione del metodo della media a partire dalla figura di esempio originale, a cui è stato applicato il metodo dei punti sovrapposti. Come si può vedere, sono stati eliminati alcuni punti che si trovavano esattamente sulla linea definita da altri punti: si riconoscono a occhio il punto 1 e il punto 64.



(a) *Situazione di partenza* **(b)** *Metodo della media*

Figura 3.5 – Applicazione del metodo della media alla figura di esempio, a cui è stato precedentemente applicato il metodo dei punti sovrapposti

3.4.4.4 Metodo del buffer

Questa sezione è dedicata a spiegare il funzionamento della funzione `rem_buffer`, che rimuove i punti secondo il *metodo del buffer*, che verrà qui spiegato.

Si considerano tre punti adiacenti A, B, C. Dato il segmento AC che unisce i punti A, C, si costruisce il *buffer* di raggio r del segmento AC, e si verifica se il punto B è contenuto all'interno di questo buffer: se è vero, il punto viene rimosso; altrimenti, viene mantenuto.

In particolare, dato un segmento AB e un raggio r, il *buffer* del segmento AB di raggio r è così costruito [59]:

- Si trovano le due rette parallele al segmento AB, ai due lati, distanti r dal segmento di partenza, chiamate AB_{left} e AB_{right};
- Si disegnano due semi-cerchi centrati in A e in B, di raggio r, chiamati A_{arc} e B_{arc};
- Si connettono, nell'ordine, AB_{left}, A_{arc}, AB_{right} e B_{arc}, per costruire un poligono.

Nell'immagine 3.6 è rappresentato il processo di creazione del buffer appena spiegato.

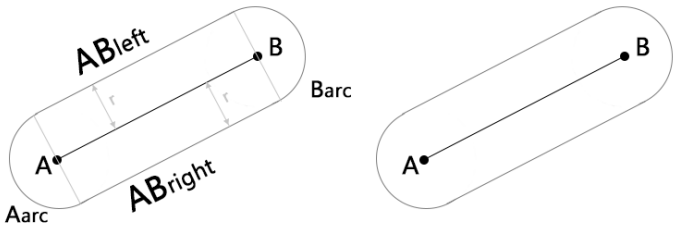


Figura 3.6 – Costruzione del buffer di raggio r per il segmento AB

Per la costruzione del buffer, non si è scritto l'algoritmo da zero ma si è utilizzato il metodo `buffer` della libreria *Shapely* [58], scelto perché *well-established* e rapido.

La funzione `rem_buffer`, quindi, considera il dataset di input, considera tutte le righe a gruppi di tre, e rimuove il secondo punto se rispetta questa condizione. Si sono anche in questo caso effettuate le scelte di rimozione *in-place* e di condizioni al contorno periodiche, per cui si rimanda alla sezione 3.4.4.1.

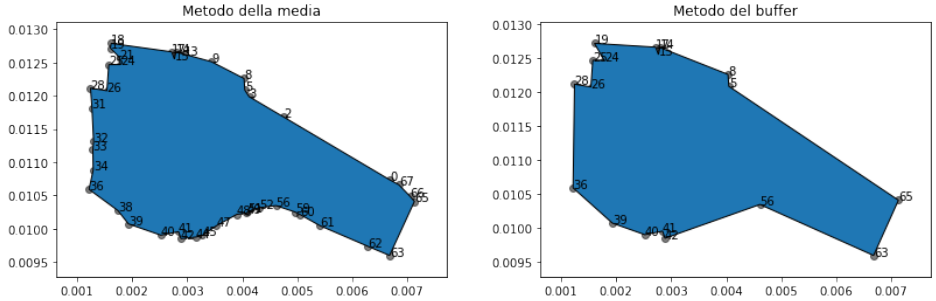
Per quanto invece riguarda il *raggio* del buffer, si è deciso di renderlo dipendente dal dettaglio (`detail`) per mezzo di un parametro moltiplicativo `par_buffer`, ponendo quindi:

$$r = \texttt{detail} \cdot \texttt{par\_buffer}$$

In particolare, il valore di default di `par_buffer` è posto a `0.5`, così che il buffer attorno a un punto (cioè, il cerchio di raggio r centrato nel punto) abbia diametro pari a `detail`.

Si noti inoltre che, essendo `detail` un valore considerato in metri, per rendere il calcolo del buffer computazionalmente coerente con i valori delle coordinate dei punti, in gradi, si effettuata la conversione da metri a gradi, secondo la formula (3.2) a pagina 26.

Si osserva in figura 3.7 l'esempio di applicazione del metodo del buffer a partire dalla figura di esempio originale, a cui è stato applicato il metodo dei punti sovrapposti e il metodo della media. Come si può vedere, questo metodo elimina moltissimi punti della figura: si osservano ad esempio i punti $31, 32, 33, 34$ e i punti dal 42 al 63.



(a) *Situazione di partenza* **(b)** *Metodo della media*

Figura 3.7 – Applicazione del metodo del buffer alla figura di esempio, confrontato con la figura a cui è stato applicato il metodo dei punti sovrapposti e della media.

3.4.4.5 Metodo della lunghezza dei segmenti

Questa sezione è dedicata invece a spiegare il funzionamento delle funzioni nel file Python `RemovingLength.py`, dedicate alla rimozione dei punti secondo il metodo della *lunghezza dei segmenti*. In particolare si sono ideate due tecniche ascrivibili a questo metodo: una che verrà qui chiamata *rimozione grezza*, contenuta nella funzione *`rem_length`*; e un'altra, qui chiamata *rimozione fine*, contenuta nella funzione *`rem_finelength`*.

Rimozione grezza Siano dati i punti adiacenti A, B, C e si considera in particolare il segmento AB. Se $\overline{AB} < l_{min}$, dove l_{min} è una data lunghezza minima, l'estremo B viene rimosso e all'iterazione successiva si considera il segmento AC;

altrimenti, l'estremo B non viene rimosso e all'iterazione successiva si considera il segmento BC.

La lunghezza del segmento AB è stata valutata tramite il metodo `length` della libreria Shapely [58], che si basa sulla distanza euclidea.

La funzione *`rem_length`*, quindi, considera il dataset di input, considera tutte le righe a gruppi di due, e rimuove il secondo punto se rispetta questa condizione. Si sono anche in questo caso effettuate le scelte di rimozione *in-place* e di condizioni al contorno periodiche, per cui si rimanda alla sezione 3.4.4.1.

Per quanto invece riguarda la *lunghezza minima* presa in considerazione dall'algoritmo, si è deciso di renderla dipendente dal dettaglio (`detail`) per mezzo di un parametro moltiplicativo `par_length`, ponendo quindi:

$$r = \texttt{detail} \cdot \texttt{par\_length}$$

In particolare, il valore di default di `par_length` è posto a `2`.

Anche in questo caso, essendo `detail` un valore considerato in metri, per rendere il calcolo della lunghezza computazionalmente coerente con i valori delle coordinate dei punti (in gradi), si è effettuata la conversione da metri a gradi, seguendo l'equazione (3.2) a pagina 26.

Rimozione fine La rimozione fine è un metodo di rimozione dei punti sulla base della lunghezza dei segmenti che i punti costituiscono, che funziona in modo analogo alla rimozione grezza (con condizioni periodiche, rimozione *in-place*, lunghezza minima dipendente dal dettaglio e conversione da metri a gradi, come già spiegato in precedenza), ma effettua una modifica nella condizione di rimozione dei punti, motivata dalla possibilità che si verifichi la situazione descritta nella figura 3.8.

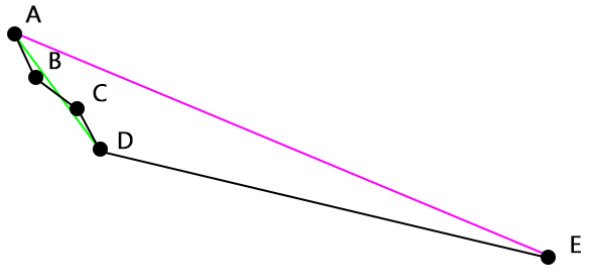


Figura 3.8 – Situazione problematica per la rimozione grezza

Si considerino i punti A, B, C, D, E come in figura 3.8, e si supponga che i segmenti AB, BC, CD rispettino tutti la condizione di essere più corti della

lunghezza minima, mentre si ha che $DE > l_{min}$. Applicando *rem_length* a questo poligono, verrebbero eliminati i punti B, C, D e si avrebbe un unico segmento che connette A con E. Si osserva in figura l'effetto che questo avrebbe sul poligono (segmento rosa): meglio sarebbe invece non rimuovere il punto D, così che ci sia un unico segmento che connette A con D (segmento verde) e il segmento DE rimane invece inalterato. Per effettuare questa correzione, si è ideato un algoritmo qui denominato di *rimozione fine*, di seguito descritto.

Siano dati i punti adiacenti A, B, C, D, E e si considerano in particolare i punti A, B, C, i segmenti AB e BC, e la lunghezza minima l_{min}. Si considerano i seguenti casi:

- Se $\overline{BC} > l_{min}$, allora non si eliminano né A né B, e si considerano i punti adiacenti successivi C, D, E;
- Se $\overline{BC} < l_{min}$, allora si hanno i seguenti sottocasi:
  - Se $\overline{AB} < l_{min}$, allora si elimina il punto B e si considerano i punti adiacenti successivi A, C, D;
  - Se $\overline{AB} > l_{min}$, allora non si elimina il punto B, e si considerano i punti adiacenti successivi B, C, D.

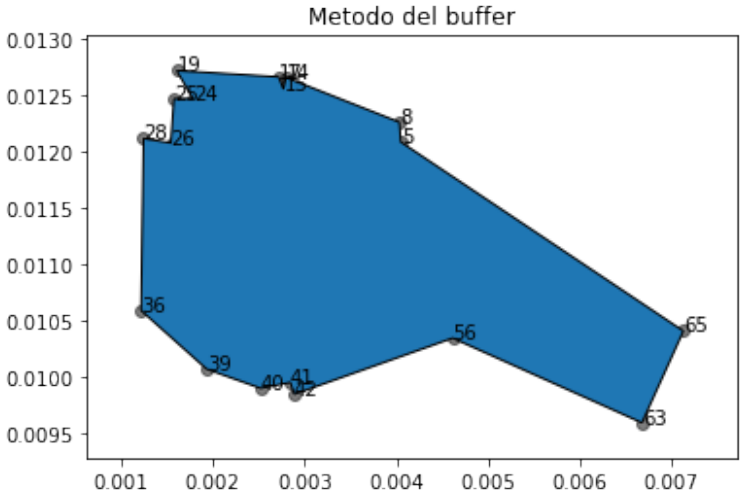
Confronto tra le funzioni Si osserva nella figura 3.9 la differenza tra l'applicazione di *rem_length* e l'applicazione di *rem_finelength* a partire dalla stessa situazione, corrispondente all'applicazione dei metodi dei punti sovrapposti, della media e del buffer.

Come si può vedere, il metodo della lunghezza dei segmenti modificato con *rem_finelength* non elimina piccoli segmenti importanti per il mantenimento della forma, risultanti in una maggiore conservazione del patrimonio informativo.

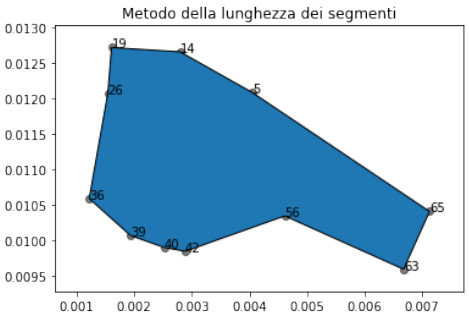
3.4.4.6 Metodo del clustering

Oltre ai metodi finora descritti, relativamente banali nel loro sviluppo, si è provato anche a implementare un metodo di sampling basato sul clustering, che si basa sull'assunzione che i punti siano distribuiti nel perimetro della figura in modo tale che siano riconoscibili alcuni *cluster* molto densi, separati da alcuni punti isolati. Questa assunzione è vera per la figura di esempio, come si è visto nella sezione 3.2.

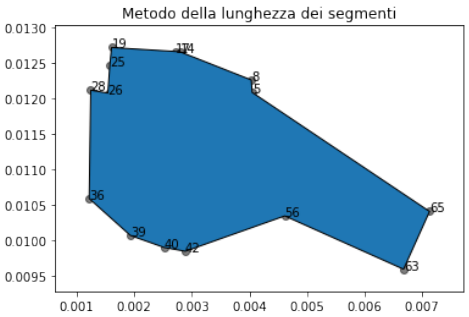
Il metodo di clustering qui scritto, che viene attivato nella funzione *Sampling* solo se il parametro `clustering` è posto a `True`, si basa quindi sull'idea di individuare i cluster maggiormente densi di punti e nel sostituirli con il punto più vicino al loro centroide, lasciando invece i punti isolati inalterati.



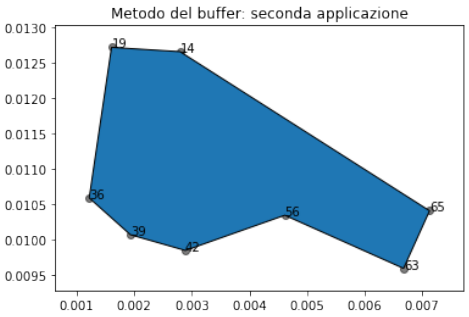
(a) *Situazione di partenza*



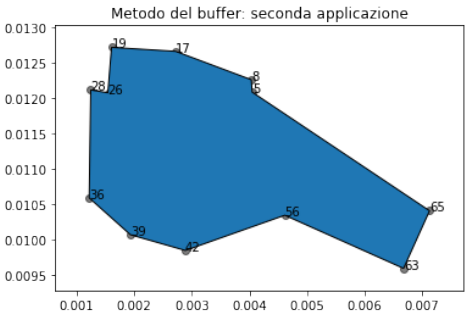
(b) `rem_length`



(c) `rem_finelength`



(d) *Finale (con* `rem_length`*)*



(e) *Finale (con* `rem_finelength`*)*

Figura 3.9 – Differenze tra applicare *`rem_length`* e *`rem_finelength`*

Si sono in particolare considerati due approcci: innanzitutto si è implementata una funzione personale, basata sull'idea di ingrandire i punti con un raggio pari al dettaglio e di considerare nello stesso clustering i punti intersecanti; e poi si è confrontato questo approccio manuale con il metodo di clustering del DBSCAN, in particolare nella sua versione implementata nella libreria di Scikit-learn [60]. Si descriveranno di seguito entrambi gli approcci.

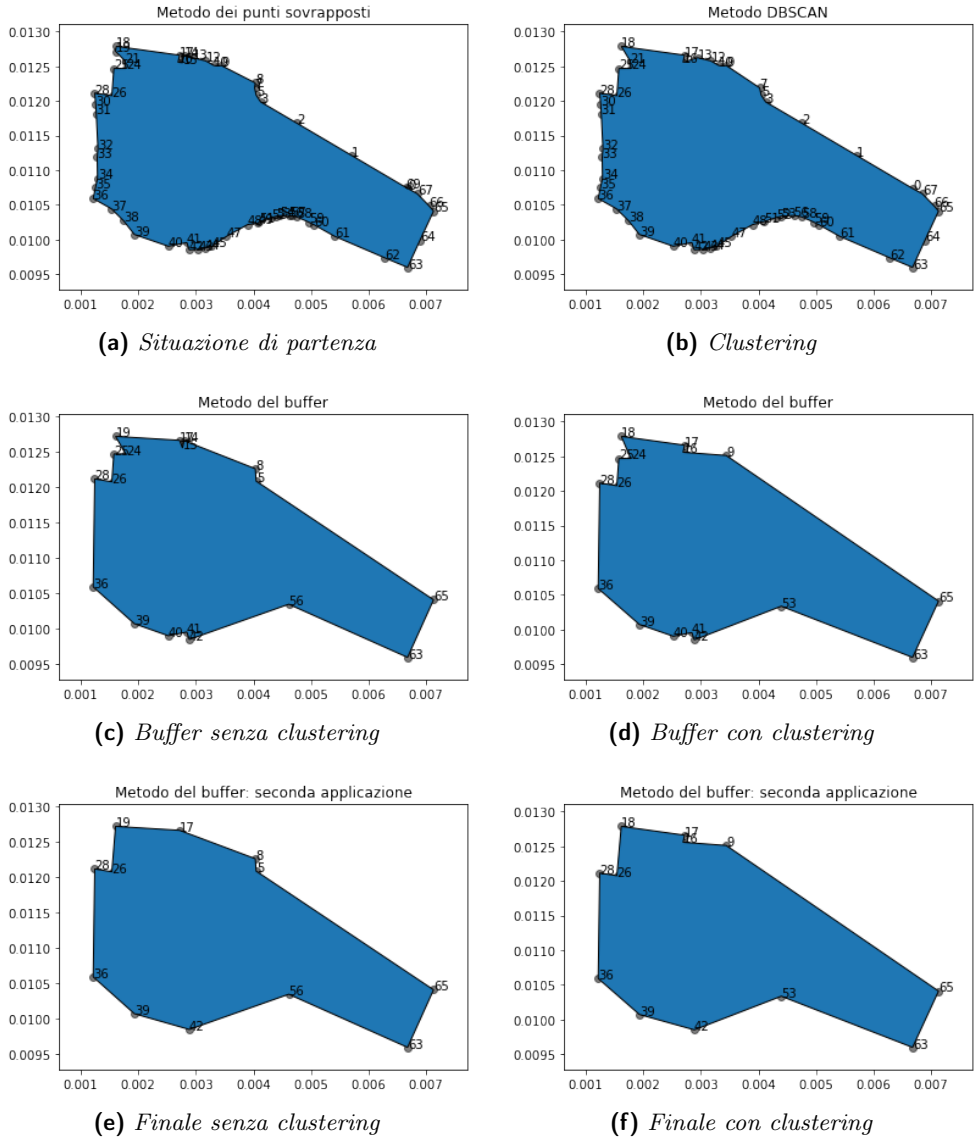
A titolo esemplificativo, si può vedere nella figura 3.10 che effetto ha applicare un metodo di clustering rispetto a non applicarlo. Come figura di riferimento si è presa la figura originale una volta eliminati i punti sovrapposti (figura 3.10a), mentre nella figura 3.10b si può osservare l'eliminazione dei punti tramite clustering, applicata subito dopo lo step dei punti sovrapposti. Nelle figure 3.10c e 3.10d si possono confrontare i risultati dell'applicare il metodo della media e del buffer al solo step dei punti sovrapposti o alla figura con punti sovrapposti e DBSCAN; e nelle figure 3.10e e 3.10f si possono invece confrontare i risultati finali del sampling dopo l'ultima correzione del metodo del buffer, applicato sia al sampling con clustering sia al sampling senza clustering.

Come si può vedere, sulla figura di esempio il metodo del clustering non ha un effetto così incisivo di per sé (nello specifico ha eliminato 3 punti); tuttavia, i suoi piccoli effetti, che in questo caso hanno riguardato uno smussamento della figura nella regione comprendente i punti 3-8, vengono amplificati dai metodi successivi e in particolare dal metodo del buffer, le cui applicazioni alla fine restituiscono una figura dalla forma leggermente diversa rispetto a quella che si avrebbe senza clustering. Poiché si è osservato nel caso della figura di esempio che la forma restituita dalla funzione di sampling con il clustering sembra meno aderente all'originale rispetto a quella senza il clustering, si è scelto di rendere l'applicazione della funzione facoltativa, regolabile tramite il parametro `clustering`.

Metodo dello zoom Questa parte è dedicata a spiegare il funzionamento delle funzioni nel file Python `ObjFunZoom.py`, dedicate alla rimozione dei punti secondo il *metodo dello zoom*, implementato in particolare nella funzione *`ObjFunZoom`*. Il metodo implementato è stato chiamato in questo modo perché l'idea è quella di effettuare uno zoom sulla figura, dando bi-dimensionalità a ciascun punto.

Volendone dare una descrizione *high-level*, questa funzione esegue i seguenti passaggi:

- Innanzitutto chiama la funzione *`points_with_buffer`*, che costruisce attorno a ogni punto P_i il *buffer* b_i di raggio r, dove per *buffer* di un punto si intende il cerchio centrato nel punto e con raggio r. In particolare, il raggio r si considera dipendente dal dettaglio (`detail`) per mezzo del parametro



(a) *Situazione di partenza* **(b)** *Clustering*

(c) *Buffer senza clustering* **(d)** *Buffer con clustering*

(e) *Finale senza clustering* **(f)** *Finale con clustering*

Figura 3.10 – Applicazione del metodo di clustering alla figura di esempio a cui è stato precedentemente applicato il metodo dei punti sovrapposti

moltiplicativo `par_zoom`, ponendo quindi:

$$r = \texttt{detail} \cdot \texttt{par\_zoom}$$

- Secondariamente, viene chiamata la funzione ***dataset_groups*** che, per ogni buffer, calcola tutte le intersezioni con tutti gli altri buffer. In particolare, per ogni i, j, con $i \neq j$, questa funzione verifica se $b_i \cap b_j$, e se sì, ottiene per ogni punto P_i l'elenco dei punti P_j con il cui buffer si interseca.
- In terzo luogo, vengono chiamate le due funzioni ***find_clusters*** e poi ***find_real_clusters***, che calcolano il raggruppamento in clusters, basandosi sulla regola che, se $b_i \cap b_j$, allora b_i e b_j appartengono allo stesso cluster;
- Poi, tramite la funzione ***add_labels***, a ogni cluster viene assegnata la sua corrispondente label;
- Infine viene chiamata la funzione ***add_centroids***, che per ogni cluster calcola il suo centroide, calcolando latitudine e longitudine media, ed effettua la sostituzione del cluster con il centroide. Ovvero, dati i punti $P_{i1}, \ldots, P_{in}$ appartenenti al cluster i, e dato il centroide c_i, indicate con $d(P_{ij}, c_i)$ le distanze tra i punti P_{ij} e il centroide c_i, con $j = 1, \ldots, n$, si calcola:

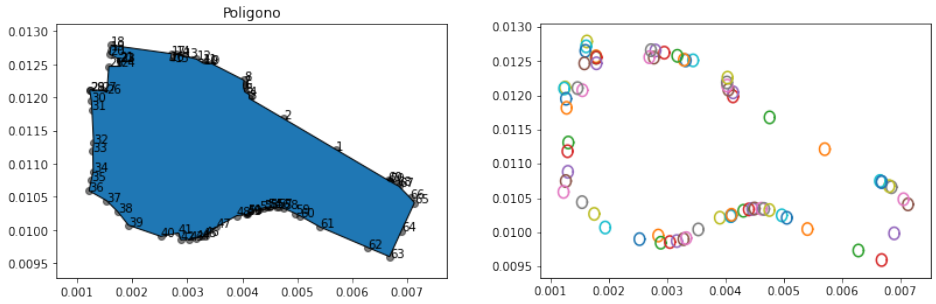
  $$P_{i\bar{j}} = \text{argmin}_j \, d(P_{ij}, c_i)$$

  e i punti P_{ij} vengono sostituiti con il punto $P_{i\bar{j}}$.

Nella figura 3.11 vengono mostrati tutti i buffer b_i dei punti creati dalla funzione ***points_with_buffer***, applicati alla figura di esempio. Si possono riconoscere alcuni cerchi isolati, e gruppi di cerchi intersecati: l'obiettivo delle funzioni successive è quello di inserire tutti i cerchi che si intersecano tra loro in un unico cluster, e sostituire quei punti con il centroide del cluster, lasciando inalterati i punti separati.

Non si descriverà di seguito il codice riga per riga, rimandando per qualsiasi dubbio o perplessità alla cartella GitHub del progetto [1], in cui il codice è presente nella sua interezza, adeguatamente commentato. Tuttavia, si riportano di seguito alcuni *highlights* che si reputano degni di nota.

Per quanto riguarda la funzione ***dataset_groups***, dati i buffer b_i, b_j, per trovare tutte le intersezioni b_i, b_j con $i \neq j$, si è effettuato un ciclo non sulle coppie b_i, b_j, ma sulle *combinazioni* b_i, b_j, utilizzando la funzione `itertools.combinations` presente nel pacchetto *itertools* di Python [61].



(a) *Figura di esempio* **(b)** *Buffer per ogni punto*

Figura 3.11 – Esempio di calcolo del buffer per ogni punto della figura di esempio

Sempre per quanto riguarda la funzione *dataset_groups*, ottenuto un dataframe che raccogliesse tutte le intersezioni tra i buffer b_i, b_j, ci si è ridotti ad un dataframe in termini solo dei punti P_i, grazie ad una operazione di `groupby` a cui è stato applicata la funzione di conversione in `set` di Python. Il dataset risultante dalla funzione *dataset_groups* è rappresentato nella figura 3.12a.

La funzione che identifica i clusters è in due parti, chiamate *find_clusters* e *find_real_clusters*. A partire dal dataset restituito da *dataset_groups*, la prima funzione restituisce la lista di clusters in parte visibile nella figura 3.12b, in cui sono presenti i clusters finali ma anche alcuni loro sottoinsiemi; mentre la seconda funzione trasforma prima ogni set di quella lista nel suo sovrainsieme con più elementi, come in figura 3.12c e poi elimina i doppioni, come in figura 3.12d.

Infine, quelli finora chiamati come *centroidi* sono calcolati, nella funzione *add_centroids*, facendo la media delle coordinate dei punti del cluster. L'operazione è valida anche per i cluster con un solo elemento.

Metodo del DBSCAN Il metodo del DBSCAN, implementato nel file Python `DBSCANsampling.py` e in particolare nella funzione *DBSCANsampling*, consiste nell'applicare il metodo di clustering DBSCAN ai punti del poligono, e in seguito sostituire a ogni cluster il suo centroide, come svolto in precedenza con il metodo dello zoom.

In particolare, tra le varie tipologie di clustering (k-means, k-medoids, mixture models, Self Organizing Maps, graph-based...), data la necessità di ottenere un risultato basato sulla densità dei punti, si è scelto il metodo *density-based* del DBSCAN, e in particolare la versione implementata da Scikit-learn [60], molto ottimizzata e rapida anche in caso di molti punti, nonostante in questo caso non sia necessario. L'algoritmo, tra le altre caratteristiche, ha il vantaggio di richiedere relativamente pochi paramtri e di non aver bisogno di *domain knowledge*, come

	Name_A	Points
0	0	{67, 68, 69}
1	1	{}
2	2	{}
3	3	{4, 5, 6}
4	4	{5, 6, 7}
...	...	...
65	65	{66}
66	66	{}
67	67	{68}
68	68	{69}
69	69	{}

70 rows × 2 columns

(a) *Dataset di* `dataset_groups`

```
[{0, 67, 68, 69},
 {1},
 {2},
 {3, 4, 5, 6, 7, 8},
 {4, 5, 6, 7, 8},
 {5, 6, 7, 8},
 {6, 7, 8},
 {7, 8},
 {8},
 {9, 10, 11, 12},
 {10, 11, 12},
 {11, 12},
 {12},
 {13, 14, 15, 16, 17},
 {14, 15, 16, 17},
 {15, 16, 17},
 {16, 17},
 {17},
 {18, 19, 20, 21, 22, 23, 24, 25},
 {19, 20, 21, 22, 23, 24, 25},
 {20, 21, 22, 23, 24, 25},
 {21, 22, 23, 24}
```

(b) *Clusters di* `find_clusters`

```
[{0, 67, 68, 69},
 {1},
 {2},
 {3, 4, 5, 6, 7, 8},
 {3, 4, 5, 6, 7, 8},
 {3, 4, 5, 6, 7, 8},
 {3, 4, 5, 6, 7, 8},
 {3, 4, 5, 6, 7, 8},
 {3, 4, 5, 6, 7, 8},
 {9, 10, 11, 12},
 {9, 10, 11, 12},
 {9, 10, 11, 12},
 {9, 10, 11, 12},
 {13, 14, 15, 16, 17},
 {13, 14, 15, 16, 17},
 {13, 14, 15, 16, 17},
 {13, 14, 15, 16, 17},
 {13, 14, 15, 16, 17},
 {18, 19, 20, 21, 22, 23, 24, 25},
 {18, 19, 20, 21, 22, 23, 24, 25},
 {18, 19, 20, 21, 22, 23, 24, 25},
 {18, 19, 20, 21, 22, 23, 24, 25},
 {18, 19, 20, 21, 22, 23, 24, 25}
```

(c) *Primi clusters di* `find_real_clusters`

```
[{0, 67, 68, 69},
 {1},
 {2},
 {3, 4, 5, 6, 7, 8},
 {9, 10, 11, 12},
 {13, 14, 15, 16, 17},
 {18, 19, 20, 21, 22, 23, 24, 25},
 {26, 27},
 {28, 29, 30, 31},
 {32, 33},
 {34, 35, 36},
 {37},
 {38},
 {39},
 {40},
 {41, 42, 43, 44, 45, 46},
 {47},
 {48, 49, 50, 51},
 {52, 53, 54, 55, 56, 57, 58},
 {59, 60},
 {61},
 {62},
 {63},
 {64},
 {65, 66}]
```

(d) *Clusters finali di* `find_real_clusters`

Figura 3.12 – Highlights del codice di *ObjFunZoom*

ad esempio il numero di clusters totali richiesto dal metodo delle k-medie [62].

Volendo brevemente riassumere i passaggi di un algoritmo di DBSCAN, innanzitutto è necessario classificare ciascun punto nelle tre categorie di *core point*, *border point* e *noise point*, come di seguito:

- Un *core point* è un punto nel dataset tale per cui, entro una distanza massima di `eps`, sono presenti almeno altri `minPoints` punti;
- Un *boundary point* è un punto che non rispetta la condizione per essere un *core point*, ma è tale che entro una distanza di `eps` è presente almeno un *core point*;
- Un *noise point* è un punto che non rispetta la condizione di essere un *core point*, e il suo *core point* più distante è oltre una distanza di `eps`.

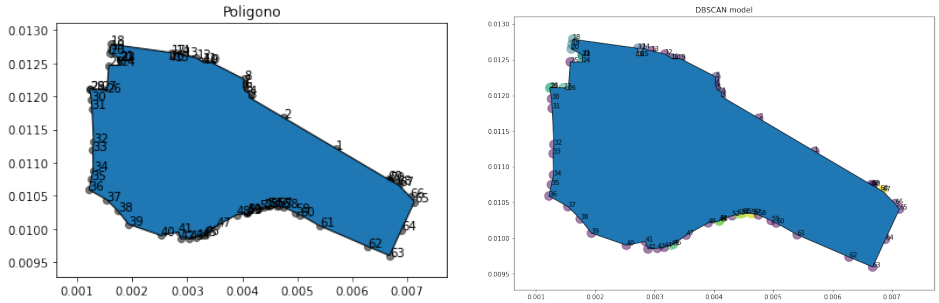
A questo punto, un *cluster* si può effettuare ricorsivamente: si parte da un *core point*, si prendono tutti i suoi vicini che sono *core points*, si aggiungono tutti i *loro* vicini che sono *core points*, eccetera; e poi, ai cluster costituiti dai *core points*, si aggiungono i rispettivi *boundary points* più vicini. I *noise points* vengono invece lasciati isolati, l'algoritmo di Scikit-learn assegna loro l'etichetta -1.

Per implementare l'algoritmo, sono richiesti due parametri: `eps`, la distanza massima del vicinato entro la quale calcolare la densità; e `minPoints`, il numero di punti che devono essere presenti nel vicinato di un punto perché questo sia considerato un *core point*. In questo contesto, per rendere la funzione paragonabile a quanto sviluppato con il metodo di zoom, si è scelto di porre `minPoints` pari a 2 (che, includendo il punto stesso, significa richiedere la presenza di un solo altro punto nel vicinato per considerare il punto un *core point*), mentre si è posto `eps` dipendente dal dettaglio `detail`, in maniera analoga al metodo dello zoom, come segue:

$$\texttt{eps} = \texttt{detail} \cdot \texttt{par\_zoom}$$

La funzione *`DBSCANsampling`* implementata, quindi, innanzitutto richiama la funzione *`DBSCANmodel`*, che effettua il clustering; poi viene chiamata la funzione *`DBSCANlabels`*, che di fatto ha lo scopo di sostituire la label -1 che l'algoritmo automaticamente assegna ai *noise points* con una label univoca per ogni punto; e infine viene chiamata la funzione *`add_centroids`*, già definita nel metodo dello zoom descritto precedentemente, grazie alla quale vengono calcolati i centroidi e ogni cluster di punti viene sostituito con il suo centroide.

Per avere un'idea di come si comporta il DBSCAN, nella figura 3.13 si può vedere come si comporta l'algoritmo del DBSCAN applicato alla figura di esempio.



(a) *Figura di esempio* **(b)** *DBSCAN clustering*

Figura 3.13 – Algoritmo del DBSCAN applicato alla figura di esempio

Confronto tra i due metodi Dalle spiegazioni effettuate, si può osservare che la funzione di zoom implementata segue una logica molto simile al noto metodo del DBSCAN, e perciò si desidera qui effettuare un confronto preliminare tra i due metodi, applicato alla figura di esempio, per rilevarne eventuali differenze.

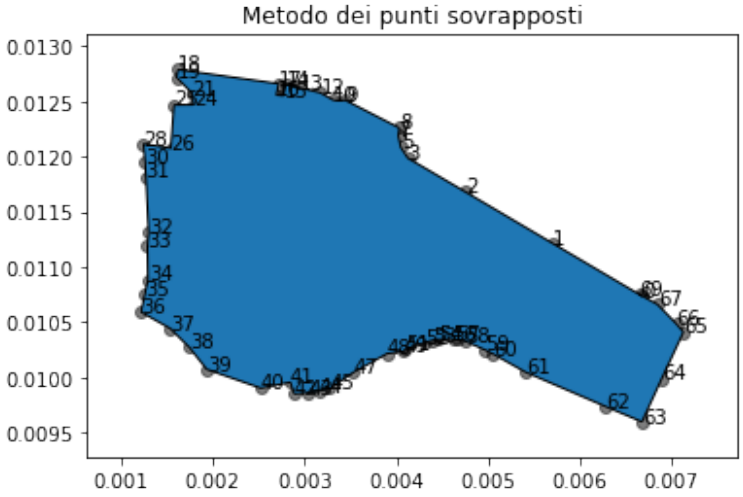
Nell'immagine 3.14 si parte da una figura a cui è stato applicato solo il metodo dei punti sovrapposti, e vengono applicate le due funzioni di clustering tramite il metodo dello zoom e della media; viene anche mostrato il risultato finale, dopo l'applicazione di tutti gli altri metodi.

Come si può vedere, almeno sulla figura di esempio i due metodi sono fondamentalmente equivalenti, e non si registrano differenze né a livello di applicazione della funzione di clustering, né nel risultato finito, cioè una volta applicato anche il metodo del buffer che, come si è visto, tende ad amplificare le eventuali piccole differenze di forma.

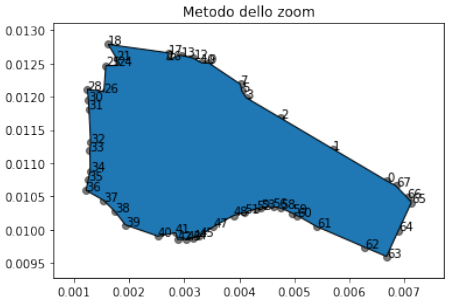
Questo tuttavia non significa che i due metodi siano identici, ed è possibile aspettarsi differenze in test più corposi.

Innanzitutto, è da mettere in rilevanza un problema di efficienza, visto che il metodo dello zoom implementato manualmente (2,6 secondi di esecuzione) risulta circa 10 volte più lento del metodo del DBSCAN (252 ms), che oltre a essere stato costruito apposta per essere molto efficiente anche per dataset grandi [62], sfrutta diverse tecniche di ottimizzazione, tra cui la scrittura di molte funzioni di base in C, tramite Cython.

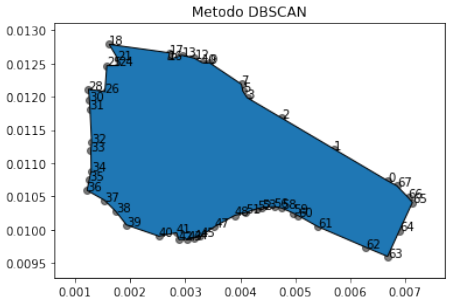
Inoltre, dato il funzionamento delle due funzioni, ci si può aspettare che la funzione di zoom tendi a generare cluster più grandi, mentre la tecnica del DBSCAN sia più conservativa. Infatti, si considerino le coppie di punti A, B e C, D rappresentate in figura 3.15, la cui distanza è pari a $\texttt{eps} + \varepsilon$, essendo $\varepsilon > 0$ ed essendo $\varepsilon < \texttt{eps}$: per queste coppie di punti, il funzionamento delle due funzioni è diverso.



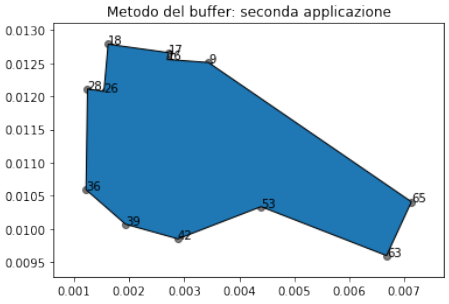
(a) *Situazione di partenza*



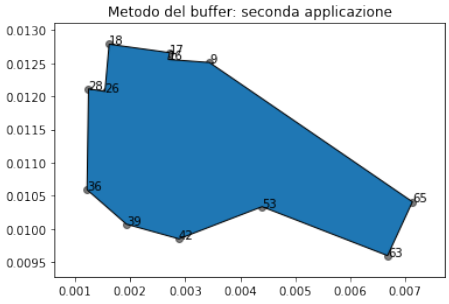
(b) *Metodo dello zoom*



(c) *DBSCAN clustering*



(d) *Finale (zoom)*



(e) *Finale (DBSCAN)*

Figura 3.14 – Confronto tra il metodo dello zoom e il metodo del DBSCAN

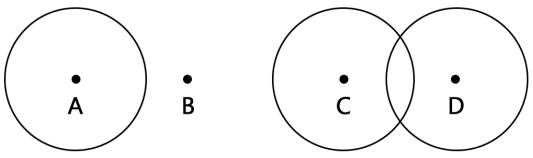


Figura 3.15 – Differenza tra il metodo DBSCAN e il metodo dello zoom

Per quanto riguarda il metodo del DBSCAN, si ha in particolare che B non appartiene al *neighborhood* di A, essendo la distanza tra A e B maggiore di `eps`; tuttavia, secondo il metodo dello zoom, C, D appartengono allo stesso cluster, poiché i loro buffer si intersecano. Di conseguenza, in generale ci si aspetta per il metodo dello zoom cluster con più punti del metodo del DBSCAN.

3.5 Individuazione del dettaglio

Il dettaglio, ovvero il parametro `detail` della funzione *`Sampling`* descritta nella sezione 3.4, è il parametro più importante della funzione di sampling. Nella funzione indica genericamente la distanza entro la quale due punti sono considerati come se fossero lo stesso punto, e di conseguenza dipende fortemente dalla grandezza della figura.

Dopo aver tentato valori di dettaglio fissi, per esempio pari a 10 o 20 metri, si è deciso di costruire un approccio per il dettaglio che dipendesse dalle lunghezze dei segmenti che compongono il poligono.

Per comprendere il tipo di approccio che si è pensato, nella figura 3.16 si può visualizzare la distribuzione delle lunghezze dei segmenti che compongono la solita figura di esempio, e si è osservato che la distribuzione delle lunghezze tende ad essere fortemente asimmetrica verso sinistra, con una coda a destra: molti segmenti corti, pochi segmenti lunghi. Questa distribuzione delle lunghezze dei segmenti è coerente con le precedenti osservazioni sulle distribuzioni dei punti.

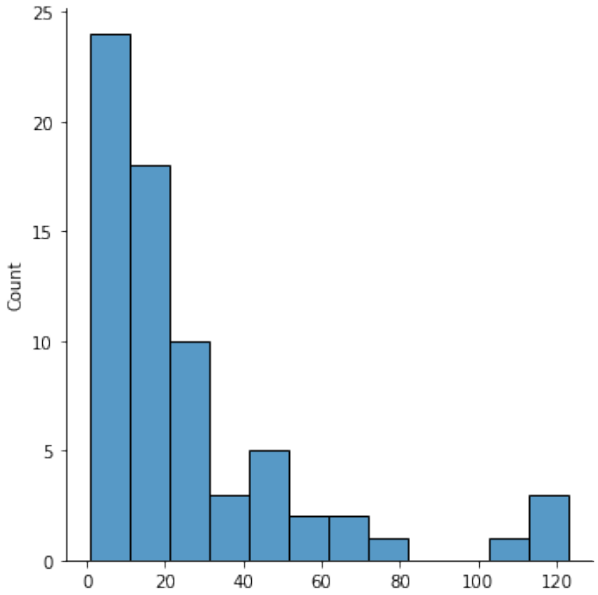


Figura 3.16 – Distribuzione delle lunghezze dei segmenti della figura di esempio

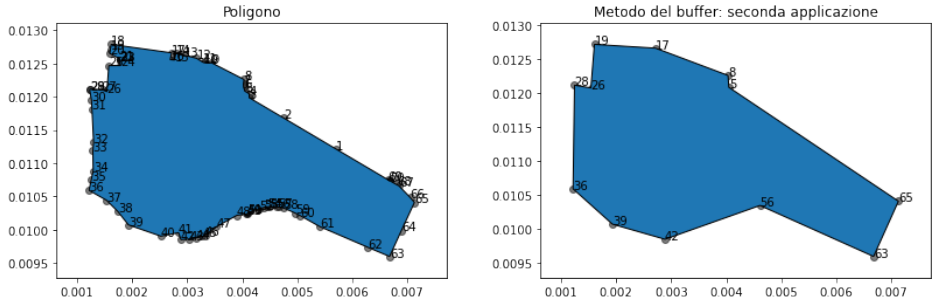
Di conseguenza, dopo aver considerato diverse statistiche come la media, la mediana, la moda e la deviazione standard, si è ritenuto che seguire un approccio basato sui *quantili* potesse costituire l'approccio più rapido e flessibile per l'individuazione del dettaglio.

Di conseguenza, la funzione `r_detailq` calcola tutte le lunghezze dei segmenti che compongono la figura, effettuando una conversione da gradi a metri utilizzando la formula (3.1) a pagina 26, e ne ricava il q-esimo quantile, dove q è un numero tra 0 e 1, approssimato per difetto tramite la funzione `floor` di *Math*. Per esempio, se si pone $q = 0.15$ (valore di default), il q-esimo quantile è il terzo ventile. Infine, si pone il dettaglio pari a questo valore.

3.6 Metriche di valutazione

Nella seguente sezione, saranno descritte le due metriche di valutazione del sampling, implementate nel file Python `Metrics.py`: una basata sulla differenza di area tra la figura semplificata e la figura originale, chiamata `diffArea`; e una basata sulla distanza massima tra la figura semplificata e la figura originale, calcolata relativamente al dettaglio del sampling, chiamata `relativeDistance`.

Per avere un esempio del loro funzionamento, si consideri il sampling sulla figura di esempio rappresentato nella figura 3.17, in cui si passa da 70 punti a 12 punti, con un dettaglio pari a 20 metri. I valori restituiti dalle due metriche sono riportati in tabella 3.3.



(a) *Figura originale* **(b)** *Figura semplificata*

Figura 3.17 – Esempio completo di sampling

3.6.1 Scelte comuni

Le due metriche implementate, sebbene si basino su criteri differenti (uno è l'area e l'altra è la distanza), sono basate su principi comuni.

Tabella 3.3 – Risultati delle metriche sul sampling in figura 3.17

Metrica	*Risultato*
diffArea	0.0443
relativeDistance	1.0952

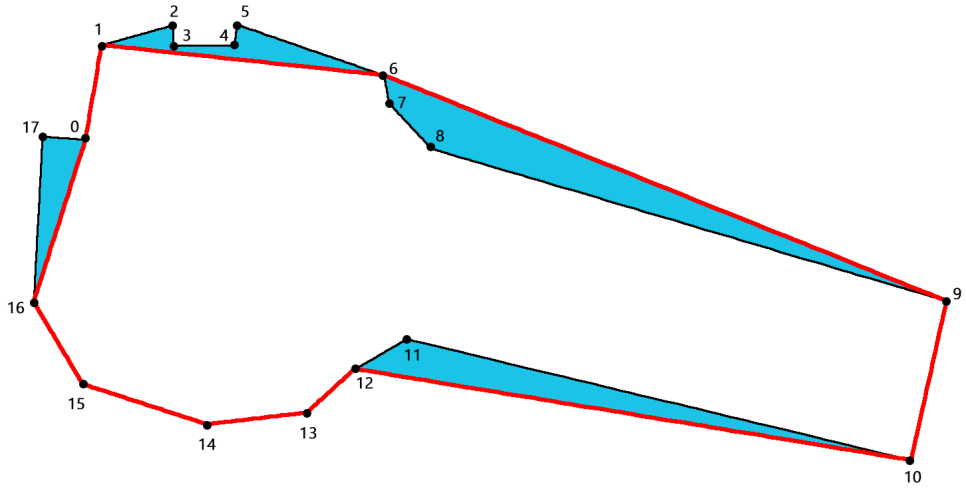


Figura 3.18 – Esempio di funzionamento delle metriche di valutazione

Innanzitutto, entrambe assumono che l'operazione precedente di sampling non abbia aggiunto punti alla figura di partenza. Ovvero, se la figura di partenza è costituita da un insieme I ordinato di N punti P_i, con $i = 0, 1, \ldots, N-1$, dove i è l'indice originale del punto, la figura semplificata è costituita da un insieme I' di $n \leq N$ punti, e si ha che $I' \subseteq I$.

Inoltre, indicando gli elementi di I' come P_j, dove i valori di j sono compresi in un insieme A, per ragioni computazionali si richiede che l'indice dei punti della figura semplificata non sia stato resettato; ovvero, si ha che $A \subseteq \{0, 1, \ldots, N-1\}$. Per un esempio, si consideri la figura 3.18: se i punti della figura originale sono i punti P_i, con $i = 0, 1, \ldots, 17$, i punti della figura semplificata sono i punti P_j, con $j = 0, 1, 6, 9, 10, 12, 13, 14, 15, 16$.

Entrambe le metriche, poi, si basano su un preprocessing comune delle due figure, calcolato nella funzione `diff_points`, che calcola ogni gruppo di punti consecutivi presenti nella figura originale ma non nella copia: questi punti, infatti, assieme ai due punti della figura semplificata adiacente al gruppo da entrambi i lati, formano un nuovo poligono, che qui sarà chiamato per convenzione *difference polygon*. Formalmente, se i punti della figura originale sono indicati con P_i e i punti della figura semplificata sono indicati con P_j, secondo le stesse con-

venzioni precedenti, considerando j, j' indici consecutivi, si considerano i punti $P_{jj'}: \{P_{j+1}, \ldots, P_{j'-1}\}$, con $P_{jj'} \subset I$.

Osservando sempre come esempio la figura 3.18, i *difference polygons* sono le regioni evidenziate in azzurro. Si consideri per esempio il poligono formato dai punti $P_1, P_2, P_3, P_4, P_5, P_6$: P_1 e P_6 sono due indici consecutivi della figura semplificata, mentre i restanti punti appartengono alla figura originale.

Le metriche di valutazione del sampling partono tutte dal considerare i *difference polygons* sopra definiti, ed effettuano operazioni su di essi.

3.6.2 Metrica della differenza di area

La metrica della differenza di area m_A è calcolata dalla funzione `diffArea`, ed è definita come il rapporto tra l'area A' pari alla somma delle aree dei *difference polygons*, e l'area A del poligono originale. Il calcolo della somma delle aree dei *difference polygons* è svolto in particolare dalla funzione `diffAbsArea`. Formalmente, essendo A_d l'area di ciascuno dei *difference polygons*, si ha che:

$$m_A = \frac{A'}{A} = \frac{\sum_d A_d}{A}$$

Per calcolare l'area di un poligono, si è utilizzato il metodo `object.area` definito da Shapely [63], essendo *well-established* e rapido, che si basa sulla formula dell'area di Gauss [64] (in inglese, *Shoelace formula*), che permette di calcolare l'area di un poligono a partire dalle coordinate dei suoi punti in uno spazio euclideo.

Entrambe le aree A, A' sono calcolate a partire dalle coordinate dei punti, e quindi la loro unità di misura è in gradi al quadrato; tuttavia, poiché m_A è un numero puro, si è ritenuto non necessario convertire le due misure in metri quadrati.

3.6.3 Metrica dello scarto massimo

La metrica dello scarto massimo m_S è calcolata dalla funzione `relativeDistance`, ed è di seguito definita.

Siano D_d ciascuno dei *difference polygons*, e siano P_j^d e $P_j'^d$, come sopra, i due punti estremi di ciascun *difference polygon*, appartenenti alla figura semplificata, mentre si indichi con P_k^d ciascuno dei punti interni al *difference polygon*.

Si consideri il segmento $\overline{P_j^d P_{j'}^d}$ che connette i due punti estremi di ciascun *difference polygon*, e in particolare la retta s definita da questo segmento, e si calcoli la distanza $d_k^d(s, P_k^d)$ tra questa retta e ciascuno dei punti P_k^d.

Si definisce *scarto del difference polygon* S^d la seguente quantità:

$$S^d = \max_k d_k^d(s, P_k^d)$$

Infine, si definisce lo *scarto massimo* S la seguente quantità:

$$S = \max_d s^d$$

La funzione `maxDistance` calcola lo scarto S, innanzitutto in gradi e poi convertendolo in metri tramite la formula (3.1) a pagina (3.1).

Inoltre, per poter confrontare diversi scarti tra di loro, appartenenti a figure con dimensioni diverse, la funzione `relativeDistance` calcola la metrica della differenza di scarto m_S come:

$$m_S = \frac{S}{D}$$

essendo D il dettaglio utilizzato dal sampling.

Capitolo 4

Risultati dei test effettuati

In questa sezione, dopo una breve descrizione del dataset fornito e del preprocessing e dell'esplorazione effettuata, si descriveranno estensivamente tutti i test e i risultati ottenuti dall'algoritmo di sampling con vari valori dei parametri, eseguiti su tutte le figure per cui è stato possibile eseguirli. Si ricorda, come già evidenziato nella sezione 3.2, che il dataset è stato fornito solo successivamente allo sviluppo di buona parte dell'algoritmo.

4.1 Descrizione del dataset

Il dataset fornito è un file `.csv` di 86.191 righe, ciascuna identificante un poligono. Il dataset ha in particolare due colonne: la colonna *ID_EXT*, che contiene un ID alfanumerico unico che assegna un nome al poligono, e una colonna *GEOM*, contenente la descrizione del poligono per mezzo del *Geometry Markup Language*, un linguaggio di markup simile all'XML, che funziona per mezzo di open e closded tag, tramite il quale vengono definiti oggetti geografici. Il separatore tra le colonne è il simbolo `|`.

In particolare, di seguito è mostrato un esempio di codice che definisce un poligono di dodici punti, le cui coordinate sono indicate come Lat$i$ e Lon$i$ e per cui le opzioni nei vari tag sono indicate semplicemente come *key* e *value*.

```
<gml:Polygon key1="value1" key2="value2">
    <gml:outerBoundaryIs>
        <gml:LinearRing>
            <gml:coordinates key1="value1" key2 = "value2">
            Lat1,Lon1 Lat2,Lon2 Lat3,Lon3 Lat4,Lon4
            Lat5,Lon5 Lat6,Lon6 Lat7,Lon7 Lat8,Lon8
            Lat9,Lon9 Lat10,Lon10 Lat11,Lon11 Lat12,Lon12
            </gml:coordinates>
        </gml:LinearRing>
```

```
    </gml:outerBoundaryIs>
</gml:Polygon>
```

Come si può vedere, il codice GML che descrive un poligono è costituito da una serie di *nested tags* con varie opzioni, e all'interno della tag più interna sono presenti le coordinate del punto: la latitudine e la longitudine per ciascun punto sono separati da virgola, mentre uno spazio separa un punto dal successivo. Le opzioni dei tag possono contenere anche link e qualunque tipo di simbolo di punteggiatura.

4.2 Preprocessing del dataset

L'obiettivo del preprocessing del dataset è stato quello di *normalizzare* il dataset; cioè, dalla struttura precedentemente descritta, passare a un dataset con tre colonne: *ID_EXT*, contenente l'identificativo univoco del poligono; e le due colonne *Latitude* e *Longitude*, contenenti le coordinate di ciascun punto. A partire da una struttura simile, il codice di sampling descritto nel capitolo 3 si può applicare separatamente poligono per poligono, estraendolo dal dataset completo con una operazione di indexing e resettando ogni volta l'indice dei punti.

Per fare questa operazione, si è scritto nel pacchetto *GeoSampling*, nel file di Python `DataImport.py`, la funzione *`db_preparation`*, che procede nel seguente modo:

- Innanzitutto, vengono utilizzate le RegEx per rimuovere tutte le tag prima e dopo l'elenco delle coordinate, così da ottenere, nella colonna *GEOM*, una stringa contenente solo le coordinate dei punti che compongono il poligono. Come si è visto, le tag sono del tipo:

  ```
  <gml:KeyWord key1="value1" key2="value2">}
  <gml:Polygon key1="value1" key2="value2">
  ```

  dove `value1` e `value2` possono essere anche link. Perciò, per l'eliminazione sono state utilizzate due RegEx:

  ```
  '</?gml:(?:[a-zA-z=",\.\s])+>'
  '<.+>'
  ```

  La prima RegEx permette di eliminare tutte le tag senza opzioni, o le cui opzioni hanno valori semplici (virgole, punti, spazi), comprese tutte le *end tag*; con la seconda RegEx, invece, vengono eliminate le tag più complesse.

- Si sono poi usate funzionalità di Pandas come la funzione `split` e la funzione `explode` per trasformare l'elenco di coordinate separate da virgole e

da spazi in due colonne *Latitude* e *Longitude* contenenti ciascuna un unico numero.

Poiché la funzione scritta fa uso di funzioni di Pandas ottimizzate per operare velocemente su dataframe anche molto grandi, è molto rapida nell'esecuzione e applicata all'intero dataset, ne restituisce uno costituito da 433.088 punti.

4.3 Esplorazione del dataset

Come si è detto, il dataset è composto da 86.191 poligoni e 433.088 punti. Tutti i poligoni hanno la struttura attesa per quanto riguarda le condizioni al contorno (se il poligono ha N punti, nel dataset ci sono $N + 1$ punti, di cui l'ultimo è uguale al primo). Ci sono 284 ID a cui per un errore non corrisponde nessun punto, che sono stati scartati, e quindi in realtà il dataset è composto da 85.907 poligoni unici e 432.804 punti effettivi.

La stragrande maggioranza dei poligoni è composta da 4 punti (escludendo l'ultimo punto uguale al primo). Si può vedere in particolare nella tabella 4.1 la distribuzione dei punti nei poligoni divisi per intervalli (in cui si è escluso l'ultimo punto uguale al primo) e si osserva che i poligoni con 4 punti sono 85.204, pari al 99.2% dei poligoni totali. Poiché il numero minimo di punti da mantenere in un sampling è stato considerato pari a 4, nei test successivi ci si concentrerà quindi sui poligoni con $N > 5$ punti.

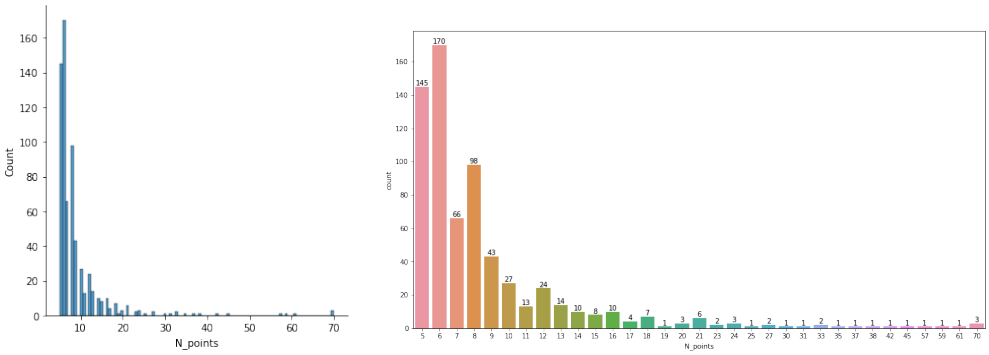
Tabella 4.1 – Distribuzione dei punti dei poligoni

Numero di punti	*Numero di poligoni*
3	31
4	85.204
5+	672

Si noti che, dai valori riportati in tabella 4.1, si può calcolare che le 31 figure con 3 punti (che nel dataset sono in realtà 4) e le 85204 figure con 4 punti (che nel dataset sono in realtà 5) costituiscono in totale 426.144 punti, cioè il 98% del totale. Questo fa sì che lavorare solo sulle figure con $N \geq 5$ sia una grossa limitazione di questo lavoro di tesi, poiché se anche il sampling eliminasse *tutti* i punti corrispondenti a queste figure, il dataset conterrebbe comunque il 98% dei suoi punti.

Nella figura 4.1 è invece riportato, per ogni numero N di punti del poligono, quanti n poligoni ci sono con quei punti, considerando solo $N > 5$. La prima figura mostra tutti i valori di N, e mette in rilevanza i valori di N per cui $n = 0$

(si osserva in particolare una mancanza di poligoni tra $N = 45$ e $N = 57$); mentre la seconda figura mostra solo quei valori di N per cui $n \geq 1$. Si nota che la stragrande maggioranza delle figure è compresa nel range tra $N = 5$ e $N = 9$ (522 figure, corrispondenti al 77% delle figure con $N > 5$).



(a) *Distribuzione continua* **(b)** *Distribuzione non continua*

Figura 4.1 – Distribuzione dei punti nei poligoni, considerando solo i poligoni con più di cinque punti.

Si osserva infine che il numero massimo di punti si ha per $N = 70$, pari al numero di punti della figura di esempio descritta nella sezione 3.2, che quindi si configura come un estremo outlier. Infatti, dato l'insieme dei valori assunti da N, il limite l oltre il quale un valore è da considerarsi un outlier (positivo) è pari a $l = 1.5q_3$, dove q_3 è il terzo quartile; e in questo caso, considerando solo i poligoni con $N \geq 5$, si ha $l = 13.5$. Si può quindi osservare fin da subito che l'algoritmo ha un *bias* di costruzione, dovuto al fatto di essere stato sviluppato basandosi su una figura di esempio che è un estremo outlier del dataset reale.

Infine, si noti che nel dataset sono presenti 3 figure con $N = 70$ punti, ma si tratta in realtà di tre copie della stessa figura, che è la figura di esempio.

4.4 Test effettuati

Si descriveranno in questa sezione i test che sono stati effettuati, che sono consistiti sostanzialmente nell'applicare la funzione *`Sampling`* con parametri diversi a tutte le figure con $N \geq 5$ (che contano in totale 5988 punti).

Si osservi che, tenendo fissato il dataset di partenza, la funzione *`Sampling`* sviluppata permette di modificare in maniera indipendente tutti i parametri elencati nella tabella 3.1. Di seguito, naturalmente, non si sono provate tutte le combinazioni possibili: ci si è piuttosto concentrati su alcuni gruppi di parametri, considerati per rispondere ad alcune specifiche domande relative al funzionamento della funzione.

Nella tabella 4.2 si riassumono tutti i parametri presi in considerazione dai vari test effettuati (dove è presente un trattino, il valore considerato è quello indicato nella colonna *Default*), mentre nella tabella 4.3 si riassumono i risultati ottenuti a livello di rimozione del numero di punti. La riga evidenziata in grigio costituisce il risultato migliore in termini di punti rimossi e di figure ignorate.

Per quanto riguarda la valutazione dei risultati, sono stati utilizzati i riferimenti seguenti.

- Innanzitutto, per ogni test è stato calcolato il numero di punti eliminati rispetto al totale, e sono state fatte valutazioni qualitative delle figure che sono state escluse dal sampling – le figure, in sostanza, per cui $n = N$, dove n è il numero di punti della figura semplificata;

- Solo sulle figure per cui il sampling è stato effettuato, si è fatta una valutazione qualitativa tramite un grafico che rappresenta la bontà del sampling in termini dei punti semplificati. In particolare, il grafico è così costruito: in ascitta c'è il numero di punti N delle figure originali, raggruppate per singoli valori categorici (le figure con $N \geq 30$ saranno considerate aggregate), mentre in ordinata c'è la media del numero n di punti delle figure semplificate. Si considererà qui come «buono» un sampling che in questo grafico restituisce un andamento al più lineare.

- Per approfondire la valutazione precedente, si è creato anche un grafico che, per ogni N, mostra la distribuzione del numero n di punti delle figure semplificate.

- Si è inoltre utilizzato anche un grafico che rappresenta, per ogni N, la distribuzione dei valori della metrica `diffArea`, basata sulla differenza di area, indicando con una *reference line* rossa un valore soglia pari a 0.05: si considera «buono» un sampling che ammette una variazione di area pari al 5%. Tale soglia è stata determinata empiricamente valutando a occhio la bontà del sampling di alcune figure, e in particolare quella di esempio.

Si osservi che, dalle rappresentazioni grafiche dei risultati, è possibile individuare il comportamento specifico della figura di esempio: infatti, dato che le tre figure presenti con $N = 70$ sono tre copie della figura di esempio, è sufficiente cercare nel grafico i valori che si hanno per $N = 70$.

In questa tesi saranno riportati a parole i risultati di tutti i test, mentre saranno mostrate solo le immagini considerate più significative.

Si è scelto invece in questo contesto di non utilizzare per le misure la metrica legata allo scarto massimo, perché la sua misura assoluta non permette il con-

Tabella 4.2 – Parametri presi in considerazione dai test.

Parametro	*Default*	*Detail*	*MinPoints*	*Par. molt.*	*DBSCAN*	*Clust. pers*	
`quantile`	0.15	vari	vari	0.25	0.25	0.25	0.25
`clustering`	False	-	-	-	-	True	True
`ifzoom`	False	-	-	-	-	-	True
`par_buffer`	0.5	-	-	vari	-	-	-
`par_length`	2	-	-	vari	-	-	-
`finelength`	True	-	-	-	False	-	-
`minPoints`	4	-	vari	-	-	-	-

Tabella 4.3 – Risultati dei test in termini di numeri di punti semplificati

Test	*Punti rimossi*	*% totale*	*Figure coinvolte*
Default	2145	36	616
`quantile` 0.25	2379	40	618
Buffer ridotto	2108	35	612
Buffer aumentato	2553	42	603
l_{min} ridotta	2324	38	629
l_{min} aumentata	2404	40	611
Finelength	2549	43	599
DBSCAN	2419	40	645
Clustering	2417	40	645

fronto tra figure, e la sua misura relativa, dipendente dal dettaglio, non consente una valutazione corretta se il dettaglio è sbagliato. La misura di area, invece, fornisce una valutazione indipendente dal dettaglio, e per questo è stata scelta per i test.

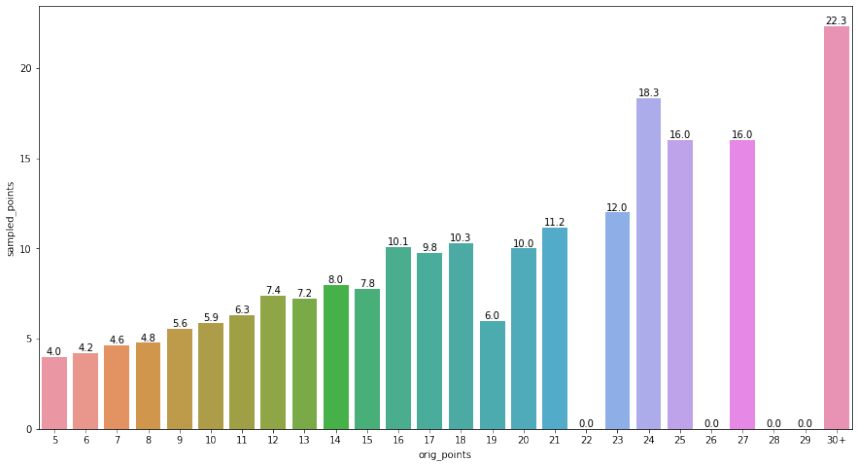
4.4.1 Parametri di default

Il primo test effettuato è stato, naturalmente, quello di applicare i parametri di default a tutte le figure. Sono stati tolti 2140 punti, pari al 36% dei punti totali, e l'algoritmo ha lasciato intatte 56 figure, di cui nessuna oltre gli 8 punti. Considerando quindi le 616 figure rimanenti, nella figura 4.2 sono mostrate le tre valutazioni grafiche dei test: come si può vedere, la semplificazione è lineare nel numero di punti originali, a parte alcune eccezioni di specifiche figure, ed è buona anche a livello di accuratezza, soprattutto per molti punti: la maggioranza delle semplificazioni sono comprese entro la soglia indicata del 5% di variazione di area, e il numero di outlier diminuisce in base al numero di punti, tanto che per $N \geq 19$ non si verificano più figure con una semplificazione oltre la soglia, a parte una.

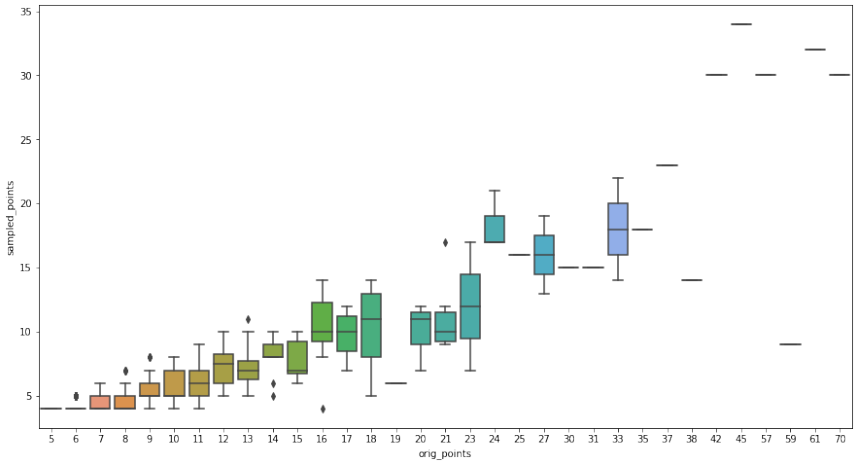
Come ci si poteva aspettare, le figure con meno punti sono quelle più penalizzate da questo algoritmo: non solo perché questo è *biased* verso le figure a molti punti, essendo stato costruito su un estremo outlier del dataset, ma anche perché meno punti ha una figura, e più quei punti sono importanti per determinarne la forma, e dunque meno se ne potranno togliere.

Si ritiene qui interessante mostrare anche le semplificazioni per alcune figure che, dall'immagine 4.2, sembrano essere delle deviazioni dai trend sopra descritti.

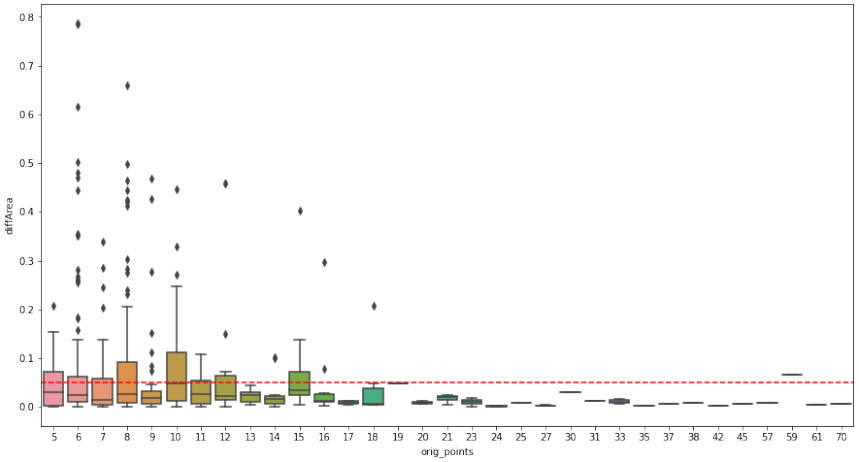
Innanzitutto, si mostrerà il caso $N = 19$, rappresentato da una sola figura, che come si vede è un caso in cui il numero di punti viene diminuito significa-



(a) *Punti semplificati: media*



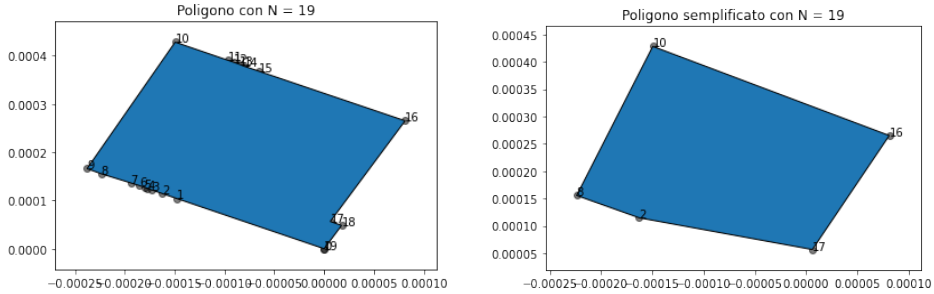
(b) *Punti semplificati: distribuzione*



(c) *Differenze di area*

Figura 4.2 – Risultati dei test: valori di default

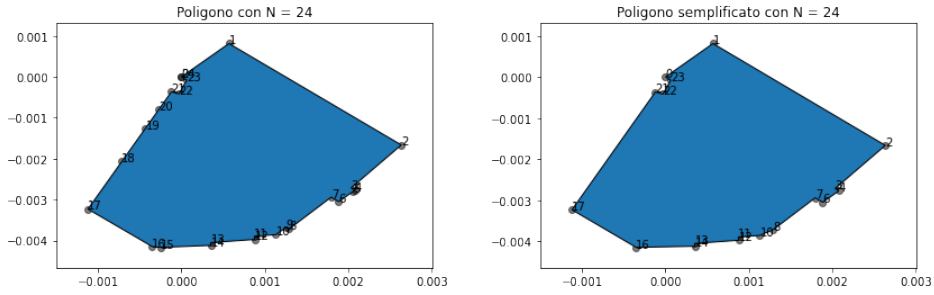
tivamente, pur rimanendo entro il range indicato dalla differenza di area. Come si vede dall'immagine 4.3, questa differenza è dovuta in particolare a una presenza maggiore del normale nella figura di punti quasi allineati, che grazie ai metodi della media e del buffer vengono completamente rimossi, causando il *drop* osservato nel numero di punti.



(a) *Figura originale* **(b)** *Figura semplificata*

Figura 4.3 – Semplificazione con i parametri di default per la figura con $N = 19$.

Secondariamente, si mostra nell'immagine 4.4 una figura per il caso $N = 24$, per cui la funzione di sampling sembra rimuovere meno punti di quelli che ci si aspetterebbe. Come si può vedere dall'immagine, questo è dovuto a una struttura a zig-zag della figura, per cui ciascun segmento piccolo (e presumibilmente minore del dettaglio) è seguito da un segmento più grande (presumibilmente maggiore del dettaglio), per cui l'opzione `finelength` mantiene anche il segmento più piccolo.

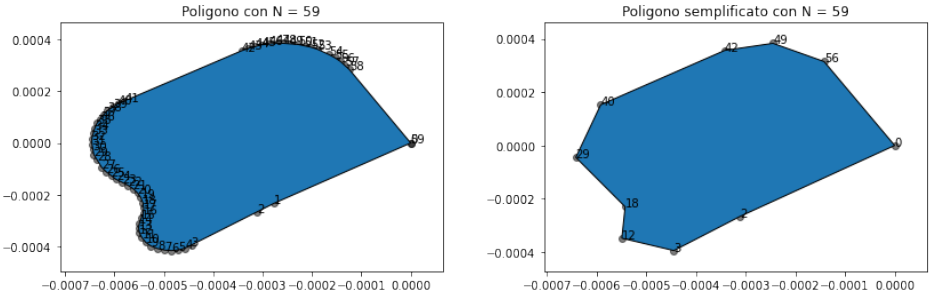


(a) *Figura originale* **(b)** *Figura semplificata*

Figura 4.4 – Semplificazione con i parametri di default per una figura con $N = 24$.

Si ritiene infine interessante mostrare la semplificazione per l'unica figura con $N = 59$, che come si vede dall'immagine 4.2 è caratterizzata da una grossa rimozione di punti, pur con una differenza di area relativa di poco superiore alla soglia. Nell'immagine 4.5 si può vedere la semplificazione applicata nello specifico a questa figura, e si osserva in particolare che la grossa riduzione del

numero di punti è dovuta a un lato curvo della costruzione, che è stato tuttavia adeguatamente semplificato con una spezzata.



(a) *Figura originale* **(b)** *Figura semplificata*

Figura 4.5 – Semplificazione con i parametri di default per la figura con $N = 59$.

4.4.2 Modifica del parametro `detail`

I test successivi che sono stati fatti sono quelli legati al parametro che si è ritenuto il più importante della funzione, quello del dettaglio (`detail`), che in particolare è stato modificato variando il quantile considerato nella funzione *`r_detailq`*, mantenendo tutti gli altri parametri allo stato di default. In particolare, si sono considerati valori di `quantile` pari a 0.1, 0.15 (default), 0.2, 0.25, 0.3, 0.4, 0.5 e 0.6.

Nella tabella 4.4 sono riportate le prime metriche caratteristiche del sampling in base ai diversi valori di `quantile` considerato: il numero di punti rimossi, la relativa percentuale e il numero di figure ignorate dall'algoritmo. Come ci si aspetta, il numero di punti rimossi cresce in modo lineare all'aumentare del quantile, così come la percentuale di punti rimossi; per quanto invece riguarda il numero di figure ignorate, si osserva un andamento più peculiare: all'inizio tende a diminuire, e poi a un certo punto la tendenza si inverte e il numero di figure ignorate inizia ad aumentare.

Si osserva che il numero di figure ignorate dipende da due fattori:

- Se il dettaglio è troppo piccolo rispetto alla figura, le funzioni di sampling non hanno nessun effetto sulla figura, che quindi viene lasciata inalterata;
- Se il dettaglio è troppo grande, le funzioni di sampling hanno un effetto eccessivo sulla figura, distruggendola talmente tanto da soddisfare la condizione di uscita, che fa sì che ogni step venga ignorato.

Di conseguenza, quando il dettaglio è minimo, agisce soprattutto il primo fattore; all'aumentare del dettaglio, il primo fattore diventa meno rilevante, e

Tabella 4.4 – Risultati numerici del sampling per vari valori di `quantile`

`quantile`	*Punti rimossi*	*Percentuale*	*Figure ignorate*
0.1	2004	0.33	66
0.15	2140	0.36	56
0.2	2246	0.37	56
0.25	2377	0.40	54
0.3	2491	0.41	57
0.4	2651	0.44	65
0.5	2793	0.47	74
0.6	2800	0.47	100

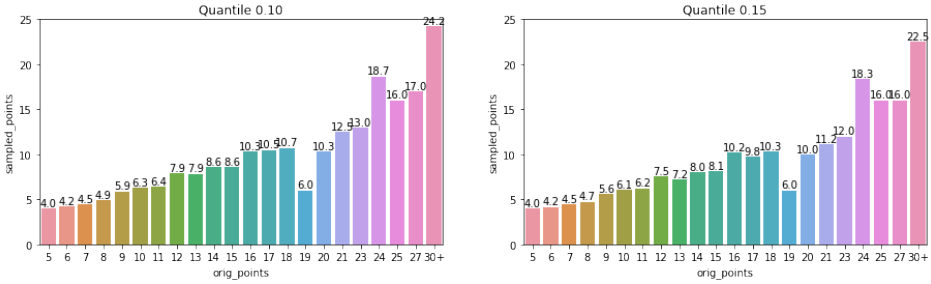
dunque il numero di figure ignorate diminuisce; ma a un certo punto inizia a diventare più rilevante il secondo fattore, e questo spiega l'aumentare delle figure ignorate. Questo suggerisce che il numero di figure ignorate possa essere una buona euristica per capire qual è il dettaglio più corretto, e che il valore di `quantile` più corretto corrisponde a quello che restituisce il numero minore di figure ignorate. In particolare, in questo caso, questa euristica suggerisce `quantile` pari a 0.25. Si suppone inoltre che il parametro `minPoints` sia in grado di regolare la formazione di questo ginocchio: questa ipotesi sarà testata nella sezione 4.4.3.

Nella figura 4.6 si possono vedere i risultati del sampling scorporati per il numero di punti originali delle figure. Si può osservare che l'andamento si mantiene lineare per tutte le figure, con una pendenza che si abbassa man mano che `quantile` aumenta, fino a diventare quasi piatto. Soprattutto per `quantile` alti, diventano evidenti delle irregolarità nell'andamento lineare; tuttavia, queste potrebbero essere dovute alla presenza nel dataset di poche figure con $N > 9$.

Non si mostreranno qui i risultati ottenuti per tutti i quantili provati, ma si desidera mostrare più nello specifico i risultati ottenuti per $q = 0.25$ (il valore suggerito dal numero di figure scartate dall'algoritmo) e per $q = 0.6$ (il massimo valore testato), rispetto al valore di default $q = 0.15$. Si ricordi inoltre che $q = 0.6$ era il valore testato per mostrare i risultati nella sezione 3.4.

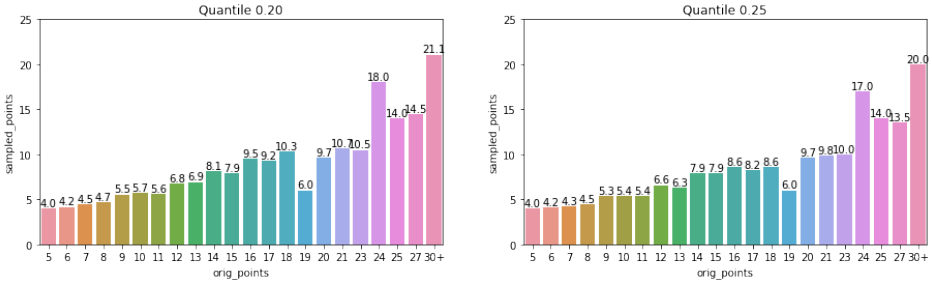
In particolare, nell'immagine 4.7 si possono vedere le differenze tra le distribuzioni della metrica dell'area per i tre quantili citati, e si nota in particolare che il valore di $q = 0.25$ è in effetti migliore anche per quanto riguarda le metriche, soprattutto per le figure con $13 \leq N \leq 18$; mentre il valore di $q = 0.6$, sebbene resti buono con $N > 30$ (compresa la figura per $N = 70$, con cui era stato precedentemente testato), si comporta in modo molto peggiore per tutte le altre figure, e in particolar modo per quelle con $N < 16$.

Date queste considerazioni, per i successivi test si utilizzerà il valore `detail`



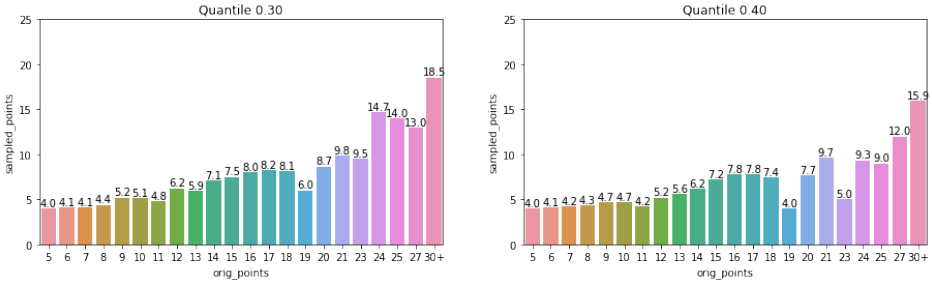
(a) $q = 0.1$

(b) $q = 0.15$



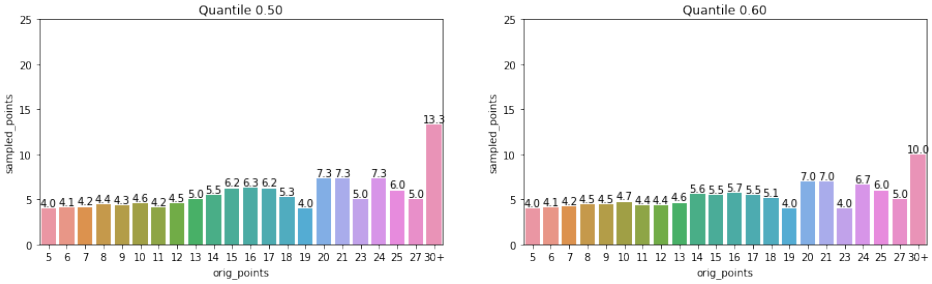
(c) $q = 0.2$

(d) $q = 0.25$



(e) $q = 0.3$

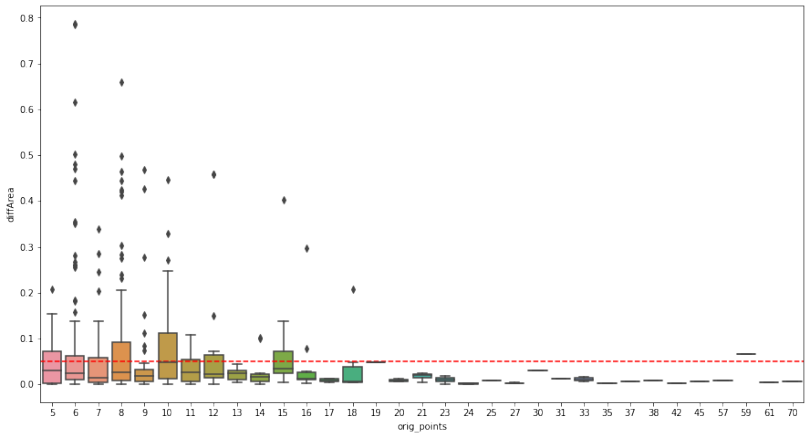
(f) $q = 0.4$



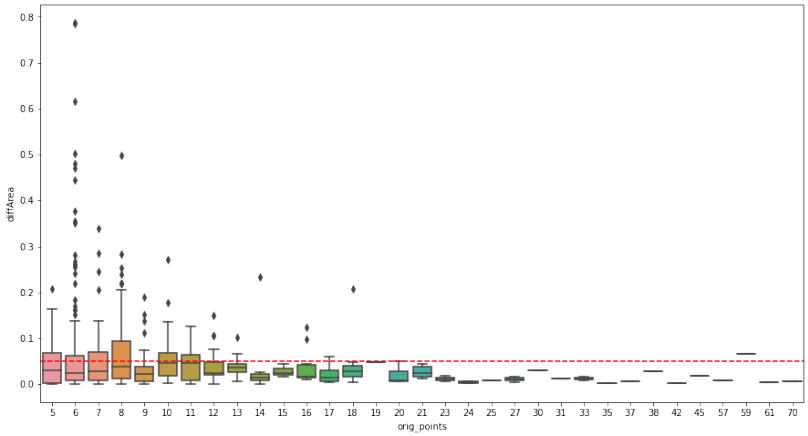
(g) $q = 0.5$

(h) $q = 0.6$

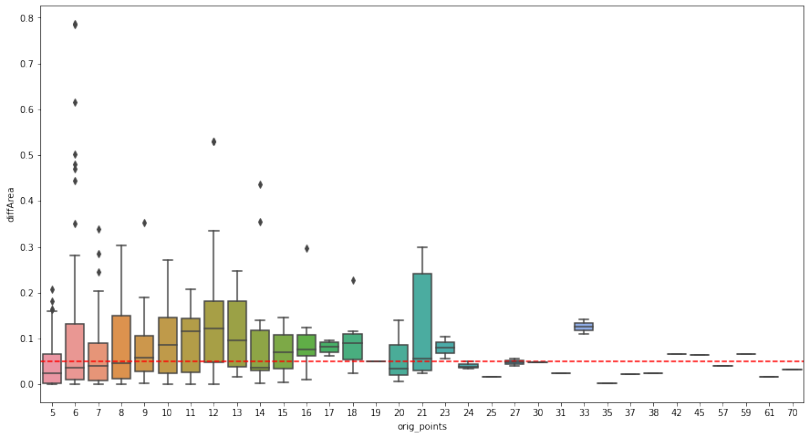
Figura 4.6 – Risultati dei test per diversi valori di `quantile` q



(a) $q = 0.15$



(b) $q = 0.25$



(c) $q = 0.6$

Figura 4.7 – Distribuzione dei valori della metrica dell'area per diversi valori di `quantile`.

pari a 0.25.

4.4.3 Modifica del parametro `minPoints`

Si è supposto nella sezione 4.4.2 di poter modificare la struttura del «ginocchio» formato dall'andamento delle figure eliminate cambiando il parametro `minPoints`. In questa sezione si è verificata questa ipotesi, in particolare ponendo come valori di `minPoints` i numeri $3, 4, 5, 6$, mentre come parametri di `quantile` si sono testati tutti i valori precedentemente controllati nella sezione 4.4.2: 0.1, 0.15, 0.2, 0.25, 0.3, 0.4, 0.5, 0.6.

In particolare, si può vedere nell'immagine 4.8 il numero di figure rimosse per i diversi valori di `minPoints` e `detail` testati, e si può constatare che in realtà l'ipotesi non è confermata. Infatti, l'aumento di `minPoints` permette solo di scartare molte più figure, ma il minimo resta presente solo per `minPoints` pari a 3 e 4, e sempre in corrispondenza dello stesso quantile.

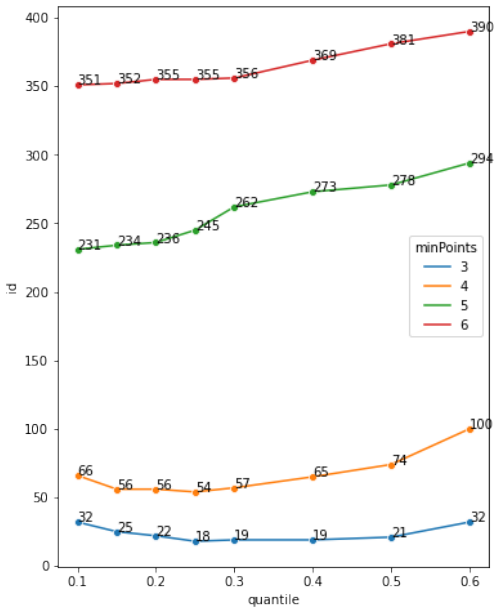


Figura 4.8 – Figure rimosse per diversi valori di `minPoints` e `detail`

4.4.4 Parametri moltiplicativi del dettaglio

Si desidera ora verificare quale sia l'effetto sul risultato finale di modificare i due parametri moltiplicativi del dettaglio, `par_buffer` e `par_length`, i cui valori di default (rispettivamente, 0.5 e 2) sono stati impostati in base a considerazioni geometriche ed euristiche sul funzionamento dell'algoritmo. Si è quindi provato

ad aumentare e diminuire entrambi i valori: per `par_buffer` si sono testati i valori 0.25 e 0.75, mentre per `par_length` si sono testati i valori 1.5 e 2.5.

Come ci si potrebbe aspettare, aumentare ciascuno dei due parametri porta a un sampling che elimina più punti, mentre ridurli significa ottenere un sampling più conservativo; tuttavia si è osservata un'asimmetria nell'importanza dei due parametri, che si riporta graficamente nella figura 4.9 per quanto riguarda le metriche dell'area, ma vale anche per quanto riguarda il numero di punti. Infatti, aumentare o diminuire `prm_buffer` ha un effetto in generale *più forte* sul sampling che aumentare o diminuire `prm_length` il cui contributo, complice probabilmente anche il metodo di `finelength`, è più fine.

Si può quindi concludere che è possibile utilizzare questi due parametri per controllare in maniera più fine il risultato che si vuole ottenere, prendendo in considerazione anche eventuali *constraint*, che potrebbero richiedere ad esempio un numero massimo di punti.

4.4.5 Opzione `finelength`

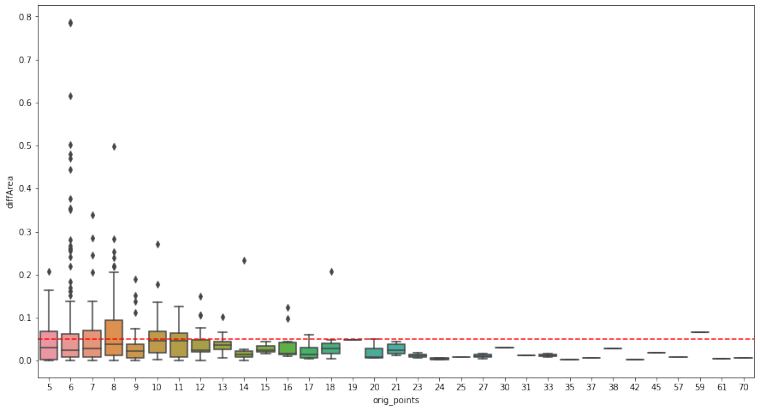
Si desidera poi verificare se il metodo di rimozione fine dei segmenti descritto nella sezione 3.4.4.5 porta a qualche vantaggio significativo nel sampling, e a questo scopo si sono confrontati i risultati del sampling con i parametri di default, ponendo `quantile` pari a 0.25, rispetto ai risultati del sampling con lo stesso `quantile` e ponendo il parametro `finelength` a `False`.

Il risultato è una rimozione di 2549 punti, pari al $42,5\%$, che ha agito su 599 figure (lasciandone fuori 73, tutte comprese tra i 5 e i 9 punti). Il sampling, come era nelle attese, ha tolto un maggior numero di punti ma, come si vede dall'immagine 4.10, è più impreciso: se prima non si verificavano più modifiche di area oltre la soglia per $N \geq 13$, ora questa soglia è $N \geq 25$, e anche per valori grandi ci sono alcuni N il cui corrispondente sampling ha una metrica dell'area oltre la soglia. È quindi confermato che l'opzione `finelength` aiuta a ottenere una maggiore precisione nel sampling, soprattutto in caso di figure di medie e grandi dimensioni.

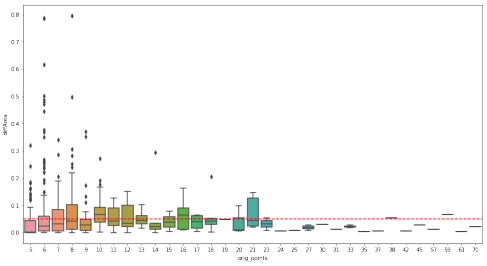
4.4.6 Opzione `clustering`

Come ultimo test, si è infine verificata l'utilità e l'importanza dell'algoritmo di clustering, confrontando il DBSCAN e l'algoritmo personale di clustering con l'algoritmo con i valori di default e `quantile` pari a 0.25.

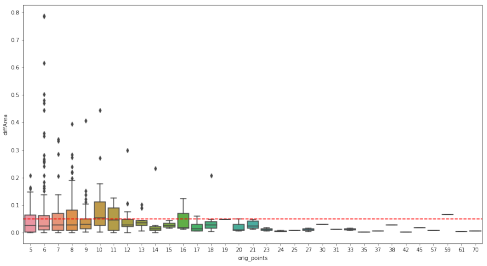
Per quanto riguarda il DBSCAN, il risultato è una rimozione di 2419 punti, pari al 40%, che ha agito su 645 figure (lasciandone fuori 27, tutte comprese tra



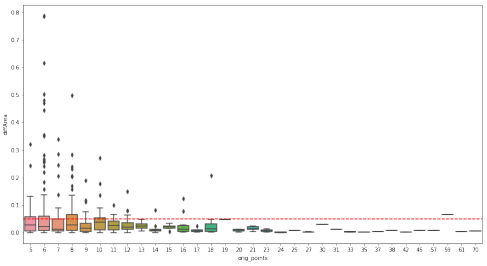
(a) *Situazione di partenza*



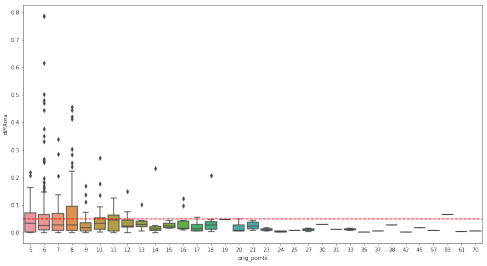
(b) `par_buffer` *aumentato*



(c) `par_length` *aumentato*



(d) `par_buffer` *diminuito*



(e) `par_length` *diminuito*

Figura 4.9 – Effetto che ha modificare i parametri `par_length` e `par_buffer` rispetto ai valori di default, con `quantile` pari a 0.25

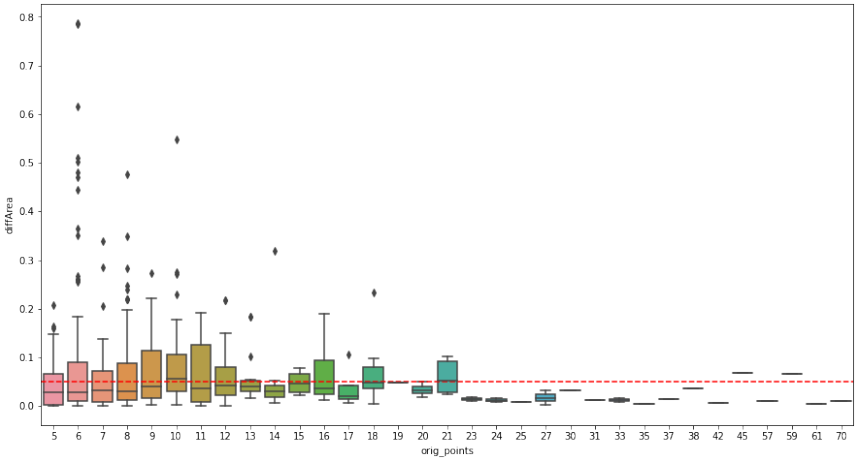
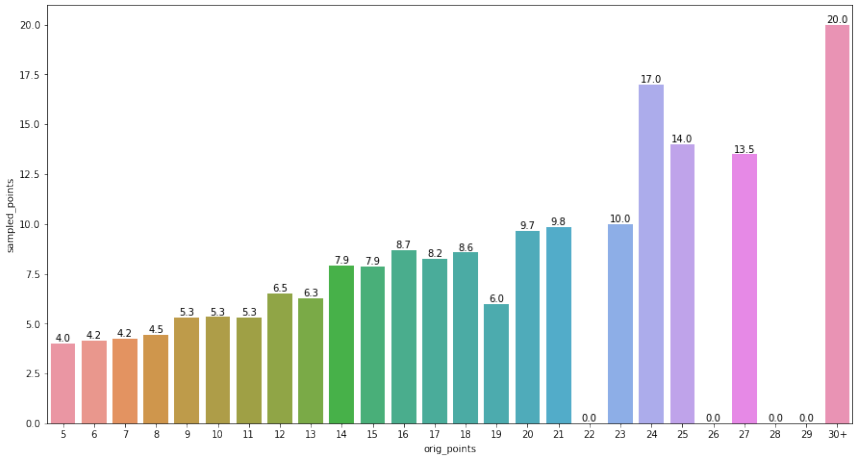
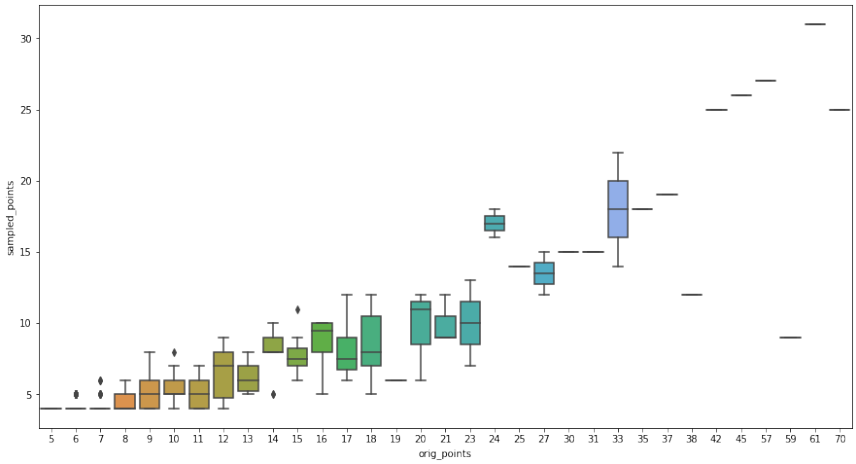


Figura 4.10 – Risultati dei test (differenze di area): test `finelength`

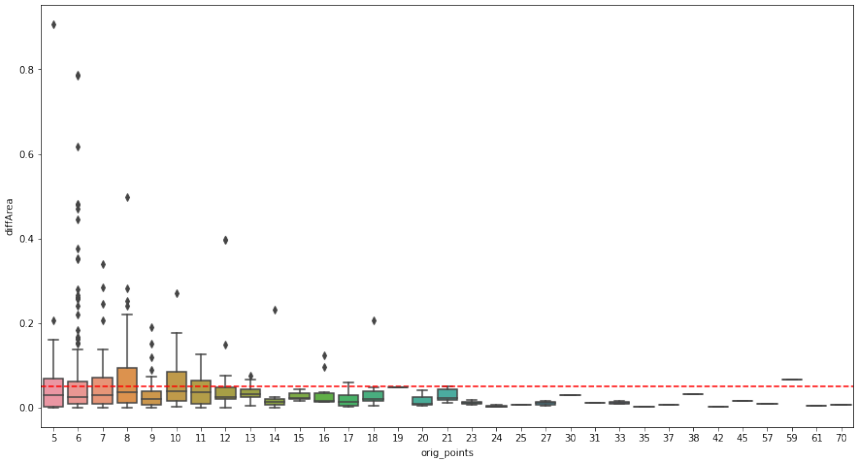
i 5 e i 9 punti): tra tutte le opzioni considerate, si tratta della combinazione migliore di punti rimossi e figure ignorate. In quanto a valutazioni, come si può vedere dalla figura 4.11, i risultati in termini di metriche sono paragonabili, e di conseguenza si può dire che il clustering abbia migliorato i risultati, perché ha permesso di ottenere una maggiore semplificazione su più figure con risultati paragonabili in termini di metriche. Per quanto invece riguarda l'algoritmo personale di clustering, il suo più grande *drawback* è la lentezza, in quanto è circa dieci volte più lento dell'algoritmo di DBSCAN; ma per il resto, i risultati non vengono riportati in questa tesi, perché sono sostanzialmente identici a quelli del DBSCAN, sia a livello di numero di punti semplificati, sia a livello di metriche.



(a) *Punti semplificati: media*



(b) *Punti semplificati: distribuzione*



(c) *Differenze di area*

Figura 4.11 – Risultati dei test con `clustering`

Capitolo 5

Conclusioni

Si riassumeranno in questa sezione le conclusioni che si possono trarre dai risultati ottenuti, anche prendendo in considerazione la revisione della letteratura che è stata svolta.

Innanzitutto, come si è visto questa tesi non ha la pretesa di fornire un codice nuovo per la *line simplification*, visto che la letteratura sull'argomento è molto ricca e molti sono gli approcci già *well-established*. L'obiettivo di questo lavoro era quindi quello di concentrarsi sulla risoluzione di un problema aziendale reale, con dati reali.

In merito a questo, una grossa limitazione di questo lavoro è data dal fatto che il dataset è stato fornito quando l'algoritmo per la rimozione dei punti era già stato in gran parte elaborato; e tale algoritmo ha seguito un approccio di rimozione *figura per figura*, utile soprattutto per figure grandi. Come però si è osservato, il dataset di test è composto per la stragrande maggioranza da quadrilateri; di conseguenza ci si è potuti concentrare solo sullo 0.02% dei poligoni, che contengono solo il 2% dei punti del dataset.

Rispetto alle figure per cui l'algoritmo è stato effettivamente testato, l'algoritmo scritto ottiene risultati buoni sia in termini di mantenimento del numero di punti, sia in termini di conservazione della forma originale (il risultato migliore rimuove il 40% dei punti). I metodi utilizzati per la rimozione sono semplici ma molto flessibili, e il numero di parametri presenti nell'algoritmo permette di effettuare correzioni fini del suo comportamento.

Tra tutti i parametri, quello più importante è sicuramente il dettaglio, per cui si è riusciti a tener conto delle diverse dimensioni delle figure, rendendolo dipendente da una statistica legata alla lunghezza dei segmenti che costituiscono il poligono. Il dettaglio migliore può essere trovato tramite una euristica che minimizza il numero di figure ignorate dall'algoritmo. I parametri moltiplicativi

del dettaglio `par_buffer` e `par_length`, permettono di *fine-tunare* i risultati, insieme alle due opzioni di `finelength` e `clustering` permettono di migliorare i risultati in termini di metriche e di conservazione della forma. L'uso dell'algoritmo del DBSCAN ha migliorato i risultati in termini di punti rimossi senza penalizzare le metriche di bontà del sampling, e l'algoritmo scritto manualmente si comporta in maniera sostanzialmente identica, ma con un maggiore tempo di esecuzione.

Si può quindi concludere che l'obiettivo posto sia stato raggiunto in parte: si può considerare risolto il problema legato alle figure grandi, e ulteriori sviluppi si dovranno concentrare sulla semplificazione dei quadrilateri rimasti. In questo caso, si sconsiglia qui l'approccio figura per figura, che non sarà conservativo della forma; si suggerisce piuttosto una tecnica di clustering *density-based*, come l'algoritmo DBSCAN che riconosca e agglomeri tra loro quadrilateri vicini e li riassuma in un poligono più grosso con più lati.

Bibliografia

[1] Silvia Tamburini. *GeoSampling - Python package for sampling geographical polygons*. URL: `https://github.com/SylviaGreenMiao/GeoSampling`.

[2] Robert E Roth, Cynthia A Brewer e Michael S Stryker. «A typology of operators for maintaining legible map designs at multiple scales». In: *Cartographic Perspectives* 68 (2011), pp. 29–64.

[3] Dietmar Grünreich. «Development of computer-assisted generalization on the basis of cartographic model theory». In: *Gis and Generalization*. CRC Press, 2020, pp. 47–55.

[4] Robert Weibel e Geoffrey Dutton. «Generalising spatial data and dealing with multiple representations». In: *Geographical information systems* 1 (1999), pp. 125–155.

[5] JC Müller e Wang Zeshen. «Area-patch generalisation: a competitive approach». In: *The Cartographic Journal* 29.2 (1992), pp. 137–144.

[6] A. H. Robinson et al. *Elements of Cartography*. New York: John Wiley & Sons, 1995.

[7] K Stuart Shea e Robert B McMaster. «Cartographic generalization in a digital environment: When and how to generalize». In: *Proceedings Auto-Carto 9*. Citeseer. 1989.

[8] Robert Brainerd McMaster e K Stuart Shea. «Generalization in digital cartography». In: Association of American Geographers Washington, DC. 1992.

[9] Monika K Rieger e Michael RC Coulson. «Consensus or confusion: cartographers' knowledge of generalization». In: *Cartographica: The International Journal for Geographic Information and Geovisualization* 30.2-3 (1993), pp. 69–80.

[10] E Raisz. *Principles of Cartography*. New York: McGraw-Hill, 1962.

[11] Henry J Steward. «Cartographic Generalisation Some Concepts and Explanation». In: *Cartographica: The International Journal for Geographic Information and Geovisualization* 11.1 (1974), pp. 1–50.

[12] A. H. Robinson, R. Sale e J Morrison. *Elements of Cartography*. New York: John Wiley & Sons, 1978.

[13] A. De Lucia e R. Black. «Comprehensive approach to automatic feature generalization». In: *Proceedings of the 13th International Cartographic Conference*. Morelia, Mexico, 1987.

[14] Robert B McMaster e Mark Monmonier. «A conceptual framework for quantitative and qualitative raster-mode generalization». In: *Nuclear Physics, Section A* (1989), pp. 390–403.

[15] T.A. Slocum et al. *Thematic Cartography and Geographic Visualization*. Geographic information science. Pearson/Prentice Hall, 2005. ISBN: 9780130351234. URL: https://books.google.it/books?id=2uQYAQAAMAAJ.

[16] Nicolas Regnauld e Robert B McMaster. «A synoptic view of generalisation operators». In: *Generalisation of geographic information*. Elsevier, 2007, pp. 37–66.

[17] Theodor Foerster, Jantien Stoter e Barend Köbben. «Towards a formal classification of generalization operators». In: *Proceedings of the 23rd International Cartographic Conference*. Vol. 4. 10.08. International Cartographic Association Moscow, Russia. 2007, p. 2007.

[18] Yilang Shen, Tinghua Ai e Yakun He. «A new approach to line simplification based on image processing: A case study of water area boundaries». In: *ISPRS International Journal of Geo-Information* 7.2 (2018), p. 41.

[19] arcGIS developers. *Simplify polygon*. URL: https://pro.arcgis.com/en/pro-app/latest/tool-reference/cartography/simplify-polygon.htm (visitato il 21/03/2022).

[20] David H Douglas e Thomas K Peucker. «Algorithms for the reduction of the number of points required to represent a digitized line or its caricature». In: *Cartographica: the international journal for geographic information and geovisualization* 10.2 (1973), pp. 112–122.

[21] Urs Ramer. «An iterative procedure for the polygonal approximation of plane curves». In: *Computer graphics and image processing* 1.3 (1972), pp. 244–256.

[22] Elmar De Koning. *psimpl – generic n-dimensional polyline simplification*. URL: http://psimpl.sourceforge.net/index.html (visitato il 21/03/2022).

[23] Stephan Hügel. *Simplification*. Ver. X.Y.Z. Dic. 2021. DOI: 10.5281/zenodo.5774852. URL: https://github.com/urschrei/simplification.

[24] Martin Fleishmann. *Line simplification algorithms*. URL: https://martinfleischmann.net/line-simplification-algorithms/ (visitato il 21/03/2022).

[25] Alan Saalfeld. «Topologically consistent line simplification with the Douglas-Peucker algorithm». In: *Cartography and Geographic Information Science* 26.1 (1999), pp. 7–18.

[26] Maheswari Visvalingam e James D Whyatt. «Line generalisation by repeated elimination of points». In: *The cartographic journal* 30.1 (1993), pp. 46–51.

[27] Zhilin Li e Stan Openshaw. «Algorithms for automated line generalization1 based on a natural principle of objective generalization». In: *International Journal of Geographical Information Systems* 6.5 (1992), pp. 373–389.

[28] Zeshen Wang e Jean-Claude Müller. «Line generalization based on analysis of shape characteristics». In: *Cartography and Geographic Information Systems* 25.1 (1998), pp. 3–15.

[29] Sheng Zhou e Christopher B Jones. «Shape-aware line generalisation with weighted effective area». In: *Developments in Spatial Data Handling*. Springer, 2005, pp. 369–380.

[30] T. Lang. «Rules for robot draughtsmen». In: *Geographical Magazine* 42 (1969), pp. 50–51.

[31] K. Reumann e A. M. P. Witkam. «Optimizing curve segmentation in computer graphics». In: *Proceedings of International Computing Symposium*. Amsterdam: North-Holland Publishing Company, 1974, pp. 467–472.

[32] Harald Opheim. «Smoothing a digitized curve by data reduction methods». In: (1981).

[33] Mark de Berg, Marc van Kreveld e Stefan Schirra. «Topologically correct subdivision simplification using the bandwidth criterion». In: *Cartography and Geographic Information Systems* 25.4 (1998), pp. 243–257.

[34] Xiaohua Tong et al. «Area-preservation Simplification of Polygonal Boundaries by the Use of the Structured Total Least Squares Method with Constraints». In: *Transactions in GIS* 19.5 (2015), pp. 780–799. DOI: https://doi.org/10.1111/tgis.12130. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/tgis.12130.

[35] J. Perkal. «An attempt at objective generalization». In: *Michigan Inter-University Community of Mathematica Geographer* (1966).

[36] Albert HJ Christensen. «Cartographic line generalization with waterlines and medial-axes». In: *Cartography and Geographic Information Science* 26.1 (1999), pp. 19–32.

[37] Vasilis Mitropoulos et al. «The use of epsilonconvex area for attributing bends along a cartographic line». In: *International Cartographic Conference, la Corona, Spain*. Citeseer. 2005.

[38] Christopher Dyken, Morten Dæhlen e Thomas Sevaldrud. «Simultaneous curve simplification». In: *Journal of geographical systems* 11.3 (2009), pp. 273–289.

[39] Tinghua Ai et al. «Envelope generation and simplification of polylines using Delaunay triangulation». In: *International Journal of Geographical Information Science* 31.2 (2017), pp. 297–319.

[40] Tinghua Ai et al. «A simplification of ria coastline with geomorphologic characteristics preserved». In: *Marine Geodesy* 37.2 (2014), pp. 167–186.

[41] Nabil Mustafa et al. «Dynamic simplification and visualization of large maps». In: *International Journal of Geographical Information Science* 20.3 (2006), pp. 273–302.

[42] Haizhong Qian, Meng Zhang e Fang Wu. «A new simplification approach based on the oblique-dividing-curve method for contour lines». In: *ISPRS International Journal of Geo-Information* 5.9 (2016), p. 153.

[43] Bin Jiang e Byron Nakos. «Line simplification using selforganizing maps». In: *Proceedings of the ISPRS Workshop on Spatial Analysis and Decision Making, Hong Kong, China*. 2003, pp. 3–5.

[44] Robert B McMaster. «The integration of simplification and smoothing algorithms in line generalization». In: *Cartographica: The International Journal for Geographic Information and Geovisualization* 26.1 (1989), pp. 101–121.

[45] Byron Nakos, Julien Gaffuri e Sébastien Mustière. «A transition from simplification to generalisation of natural occurring lines». In: *Proceedings of 11th ICA Workshop on Generalisation and Multiple Representation*. 2008.

[46] Python Software Foundations. *Python*. URL: `https://www.python.org/` (visitato il 11/03/2022).

[47] *Anaconda Software Distribution*. Ver. 2-2.4.0. 2020. URL: `https://docs.anaconda.com/`.

[48] Python Standard Library. *Math - mathematical functions*. URL: https://docs.python.org/3/library/math.html (visitato il 12/03/2022).

[49] Python Standard Library. *Itertools - functions creating iterators for efficient looping*. URL: https://docs.python.org/3/library/itertools.html (visitato il 12/03/2022).

[50] Python Standard Library. *Collections - container datatypes*. URL: https://docs.python.org/3/library/collections.html#module-collections (visitato il 19/03/2022).

[51] Python Standard Library. *Copy – Shallow and deep copy operations*. URL: https://docs.python.org/3/library/copy.html (visitato il 19/03/2022).

[52] Charles R. Harris et al. «Array programming with NumPy». In: *Nature* 585.7825 (set. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[53] The pandas development team. *pandas-dev/pandas: Pandas*. Ver. latest. Feb. 2020. DOI: 10.5281/zenodo.3509134. URL: https://doi.org/10.5281/zenodo.3509134.

[54] Wes McKinney. «Data Structures for Statistical Computing in Python». In: *Proceedings of the 9th Python in Science Conference*. A cura di Stéfan van der Walt e Jarrod Millman. 2010, pp. 56–61. DOI: 10.25080/Majora-92bf1922-00a.

[55] F. Pedregosa et al. «Scikit-learn: Machine Learning in Python». In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[56] John D Hunter. «Matplotlib: A 2D graphics environment». In: *Computing in science & engineering* 9.3 (2007), pp. 90–95.

[57] Michael L. Waskom. «seaborn: statistical data visualization». In: *Journal of Open Source Software* 6.60 (2021), p. 3021. DOI: 10.21105/joss.03021. URL: https://doi.org/10.21105/joss.03021.

[58] Sean Gillies et al. *Shapely: manipulation and analysis of geometric objects*. toblerity.org, 2007. URL: https://github.com/Toblerity/Shapely.

[59] Junfu Fan et al. «Optimization approaches to mpi and area merging-based parallel buffer algorithm». In: *Boletim de Ciências Geodésicas* 20.2 (2014), pp. 237–256.

[60] Scikit-learn developers. *The DBSCAN algorithm*. URL: https://scikit-learn.org/stable/modules/clustering.html#dbscan (visitato il 19/03/2022).

[61] Python Standard Library. *itertools.combinations*. URL: https://docs.python.org/3/library/itertools.html#itertools.combinations (visitato il 19/03/2022).

[62] Martin Ester et al. «A density-based algorithm for discovering clusters in large spatial databases with noise.» In: *kdd*. Vol. 96. 34. 1996, pp. 226–231.

[63] Shapely developers. *Shapely: general attributes and methods*. URL: https://shapely.readthedocs.io/en/stable/manual.html%5C#general-attributes-and-methods (visitato il 20/03/2022).

[64] Wikipedia. *Shoelace formula*. URL: https://en.wikipedia.org/wiki/Shoelace%5C_formula (visitato il 20/03/2022).