Alma Mater Studiorum · Università di Bologna

SCUOLA DI INGEGNERIA E ARCHITETTURA
CAMPUS DI CESENA
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

ANALISI DI DATI DI TRAIETTORIA SU PIATTAFORMA BIG DATA

Relatore:
Prof. Matteo Golfarelli

Presentata da:
Luca Campana

Correlatore:
Dott. Matteo Francia

Anno Accademico 2017-2018
Sessione I

Ai miei genitori,

per il loro instancabile sostegno

Indice

Capitolo 1

Introduzione

L'evoluzione delle tecnologie di geolocalizzazione e di comunicazione *wireless* ha generato, in questi ultimi anni, la creazione di una grande quantità di dati geografici: sensori *GPS*, *RFID* e *device* mobili producono senza soluzione di continuità dati relativi allo spazio geografico in cui sono localizzati. Nonostante le difficoltà legate alla loro gestione, questi dati rappresentano una preziosa risorsa e sono difatti utilizzati con successo nel campo dello *spatial data mining* per estrarre conoscenza implicita e *pattern* interessanti non esplicitamente rappresentati.

Nell'ambito dei dati spaziali risultano particolarmente rilevanti i dati di traiettoria, cioè quei dati spazio-temporali, prodotti da entità in movimento dotate di appositi sensori, che rappresentano il percorso di un oggetto attraverso lo spazio in funzione del tempo. L'analisi di questi dati ha richiesto, viste le caratteristiche peculiari, lo sviluppo di tecniche *ad hoc* di *mining*, in grado di rendere il processo di estrazione delle informazioni sia efficace che efficiente. Queste tecniche permettono di inferire una serie di nozioni che vanno oltre il concetto di posizione spazio-temporale e che sono di fondamentale importanza in molti ambiti applicativi, come quello della sicurezza, del trasporto o dell'ecologia.

Se il valore di questi dati è indubbio, la loro gestione, come detto, risulta difficile. Lo sviluppo tecnologico sta radicalmente trasformando le caratteristiche dei dati spaziali: i volumi, l'eterogeneità e la frequenza con cui arrivano eccedono le capacità delle tecnologie tradizionali. Si può quindi parlare oramai di *Spatial Big Data* [12], cioè di dati spaziali

con caratteristiche proprie dei *Big Data*. Questo *shift* qualitativo e quantitativo rende necessaria l'adozione di tecnologie innovative, adatte a gestire la complessità che ne deriva: in questo contesto, risulta quindi naturale pensare alle tecnologie *Big Data* come potenziale veicolo del necessario rinnovamento tecnologico e come strumento per colmare il *gap* tecnologico creatosi.

Il contributo principale di questa tesi è lo studio e lo sviluppo di un'applicazione per l'analisi di dati di traiettoria su piattaforma *Big Data*. L'applicazione è in grado di estrarre, attraverso l'utilizzo di tecniche di *trajectory mining* su dati di traiettoria reali, una serie di informazioni, come l'abitazione, il luogo di lavoro o i luoghi frequentati, legate alle persone a cui i dati si riferiscono. Inoltre, vuole rappresentare uno studio sull'applicabilità delle tecnologie *Big Data* all'ambito dello *spatial data mining*. Lo sviluppo dell'applicazione si inserisce all'interno di un più ampio progetto, a cui il Business Intelligence Group partecipa, il cui scopo è l'estrazione di *pattern* di comportamento tra i cittadini della città di Milano, al fine di valutare l'impatto, sugli abitanti della città, di un programma di rinnovamento urbano.

La restante parte della tesi è organizzata come segue. Nel Capitolo 2 si introducono alcune delle principali tecnologie *open-source* del mondo *Big Data* e si analizza una libreria per l'elaborazione e l'analisi di dati spaziali, GeoSpark. Nel Capitolo 3 si descrivono il processo di *trajectory mining* e le relative tecniche e si approfondiscono alcuni algoritmi di *clustering* spazio-temporale e di *clustering grid/density-based*. Nel Capitolo 4 si illustrano le fasi salienti del processo di sviluppo dell'applicazione e gli algoritmi su cui si basa, concludendo con l'analisi dei risultati ottenuti. Nel Capitolo 5 si riportano conclusioni e sviluppi futuri.

Capitolo 2

Le tecnologie

La quantità di dati disponibili, pubblici e privati, è cresciuta negli ultimi decenni in modo esponenziale: solo nel 2013 i dati generati nel mondo ogni giorno erano circa 2.5 *exabyte* [1]. Tra il 2013 e il 2020 la quantità di dati globali crescerà da 4.4 *zettabyte* a 44 *zettabyte*, mentre nel 2025 si prevede un volume dati globale di 163 *zettabyte* [1]. Questa costante crescita è dovuta a diversi fattori, tutti legati al forte processo di digitalizzazione a cui stiamo assistendo da anni: la migrazione delle attività economiche verso Internet, l'esplosione dei social network, la crescita del numero di *device* mobili connessi e il diffondersi dell'*Internet of Things*. La disponibilità di una così massiccia quantità di dati ha accresciuto il bisogno di dotarsi di tecnologie in grado di gestirli in modo efficiente e di modelli di analisi in grado di sfruttarli in modo efficace: le tradizionali modalità sono infatti difficilmente in grado di gestire dati con queste caratteristiche.

2.1 *Big Data*

Con il termine *Big Data* si intendono due concetti distinti ma strettamente connessi tra loro. Originariamente i *Big Data* sono stati infatti definiti come dati strutturati, non strutturati o semi-strutturati così complessi e così grandi in volume da richiedere l'utilizzo di tecniche e tecnologie innovative. Negli ultimi tempi il termine ha però assunto una diversa sfumatura, andando a riferirsi più all'analisi e ai metodi di analisi predittiva che

alla dimensione dei dati stessi. L'elevato volume non è quindi la sola caratteristica dei *Big Data*: possiamo infatti individuare almeno 4 dimensioni (le cosiddette *V's of Big Data*) coperte dai *Big Data*.

- **Volume**: i dati hanno un elevato volume che non ne permette l'elaborazione con le tradizionali tecnologie (per esempio i *database* convenzionali).
- **Velocity**: sensori, *device* mobili e *IoT* producono dati ad una frequenza maggiore rispetto ai normali sistemi informativi.
- **Variety**: i dati sono estremamente eterogenei sia nel formato sia nel contenuto informativo.
- **Veracity**: la qualità è eterogenea e spesso non è disponibile un preciso schema.

2.1.1 Architettura *Big Data*

Un'architettura *Big Data* è un'architettura concepita per gestire la memorizzazione, l'elaborazione e l'analisi di dati troppo complessi o grandi in volume per le tradizionali architetture. Negli ultimi anni l'utilizzo dei dati e il contesto attorno ad essi è cambiato profondamente: il costo per la memorizzazione dei dati è calato notevolmente mentre sono cresciuti i mezzi attraverso i quali i dati possono essere collezionati. I dati possono infatti giungere sotto forma di *stream* ad alta frequenza o più lentamente come *chunk* di grandi dimensioni: in entrambi i casi lo scopo finale è comunque quello di analizzare questi dati con tecniche avanzate o utilizzarli in attività di *machine learning*. Il fine delle architetture *Big Data* è quello allora di risolvere le problematiche legate a queste nuove sfide. Di seguito si descrivono i componenti che possono andare a comporre un'architettura *Big Data*.

- *Data sources*: tutte le soluzioni *Big Data* necessitano di almeno una fonte di dati. Questa fonte può essere ad esempio un *database* relazionale, un insieme di file statici o un insieme di *device IoT*.

- *Data storage*: i dati utilizzati nelle fasi di *processing* sono solitamente memorizzati su file in modo distribuito, andando a costituire il cosiddetto *data lake*.
- *Batch processing*: data la mole di dati la fase di *processing* può durare a lungo e non essere quindi interattiva. In questa fase i dati vengono letti dal *data lake*, processati e poi salvati nuovamente su file.
- *Real-time ingestion*: se l'architettura prevede una fonte di dati *real-time* è necessario gestire i dati mano a mano che arrivano. I dati possono essere memorizzati e utilizzati in una fase successiva o inseriti in *buffer* per permettere l'elaborazione in tempo reale.
- *Stream processing*: i dati gestiti nella fase di *ingestion* vengono processati e preparati per l'analisi mano a mano che arrivano.
- *Machine learning*: i dati processati possono essere utilizzati come *input* per algoritmi di *machine learning*.
- *Analytical data store*: i dati dopo essere stati processati sono spesso memorizzati in formati differenti per facilitare l'utilizzo di *tool* analitici, come *database* relazionali o NoSQL.
- *Analysis and reporting*: lo scopo delle soluzioni *Big Data* è solitamente quello di favorire la comprensione dei dati. L'architettura può allora includere un ulteriore *layer* di modellazione dati per facilitare l'analisi e il *reporting*: i dati possono ad esempio essere organizzati come cubi multidimensionali da utilizzare in sessioni OLAP.

Nelle seguenti sezioni si vanno ad analizzare alcuni *framework* per architetture *Big Data* sviluppati per l'elaborazione di grandi quantità di dati su ambienti distribuiti. Nello specifico si analizzeranno Apache Hadoop, in quanto cuore di un ampio ecosistema di tecnologie *open source* per il calcolo distribuito, e Apache Spark, in quanto tecnologia di elaborazione *Big Data* flessibile, efficiente e *general purpose*. Infine si approfondirà GeoSpark, un'estensione di Spark capace di gestire e interrogare in modo efficiente grandi quantità di dati spaziali.

2.2 Hadoop

Apache Hadoop è una piattaforma software *open source* per la memorizzazione e l'elaborazione distribuita di grandi insiemi di dati su *cluster*. Fa parte dell'Apache Project e fu creata originariamente da Yahoo, dopo la pubblicazione di alcune ricerche da parte di Google [7]. È concepito per lavorare su una o migliaia di macchine, ognuna con un proprio *storage* locale e una propria computazione locale. Piuttosto che fare affidamento sull'hardware, Apache Hadoop è in grado di individuare e gestire fallimenti dell'esecuzione direttamente a livello applicativo, assicurando disponibilità e affidabilità anche su *cluster* costituiti da hardware *commodity*.

Il cuore di Apache Hadoop è costituito dal componente di memorizzazione dati, chiamato HDFS, e dal componente di elaborazione dati, che implementa il modello di programmazione MapReduce. Hadoop in particolare è in grado di suddividere i dati in grandi blocchi distribuiti tra i vari nodi di calcolo per permettere ad ogni nodo di processare i dati assegnatogli senza dover effettuare lunghi trasferimenti in rete, secondo il concetto di *data locality*. In questo modo la fase di computazione risulta più rapida. Apache Hadoop è composto da diversi moduli, elencati qui di seguito.

- **Hadoop Common**: offre le librerie necessarie agli altri moduli Hadoop.
- **HDFS**: è il *file system* distribuito che permette di memorizzare i dati su più nodi.
- **Hadoop MapReduce**: rappresenta un'implementazione del modello computazionale MapReduce.
- **Hadoop YARN**: è la piattaforma responsabile della gestione delle risorse del *cluster*.

2.2.1 Hadoop Distributed File System

Hadoop Distributed File System (HDFS) è il *file system* distribuito su cui si basa Hadoop: ha il compito di fornire un servizio di memorizzazione dati scalabile e *fault-tolerant* su

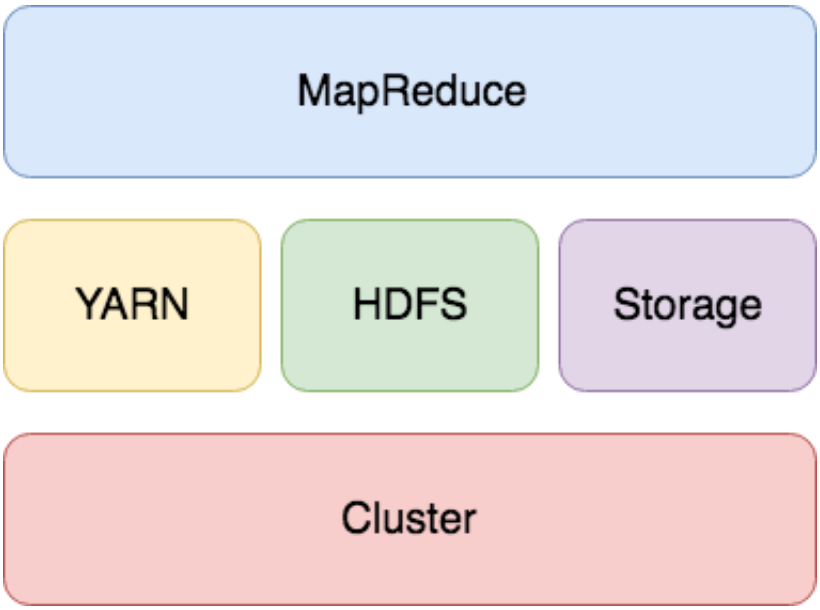


Figura 2.1: Architettura di Hadoop.

commodity hardware. I dati in Hadoop sono suddivisi in blocchi di grandi dimensioni (128 MB di default) che vengono distribuiti sui vari nodi del *cluster* e replicati due volte su più macchine. In caso di malfunzionamento di un nodo, HDFS è in grado di bilanciare in modo trasparente le repliche dei dati.

2.2.2 MapReduce

MapReduce è un modello di programmazione e, nella sua implementazione Hadoop, un *framework* concepito per facilitare l'elaborazione parallela di grandi volumi di dati su una grande quantità di nodi: come modello di programmazione è ispirato alle operazioni di *map* e *reduce*, tipiche della programmazione funzionale. Possiamo pensare ad un'operazione di MapReduce come l'esecuzione successiva di 3 passi, descritti di seguito.

- **Map**: prende in *input* un oggetto chiave-valore e ritorna una collezione di coppie chiave-valore.

- **Shuffle**: le coppie chiave-valore in *output* della fase precedente vengono recuperate da un *master* e raggruppate per chiave.

- **Reduce**: prende in *input* una chiave e una lista di valori e li combina.

L'operazione di *mapping* può essere facilmente parallelizzata in quanto ogni applicazione della funzione f di *mapping* avviene in modo isolato e indipendente: nella pratica significa che ogni nodo potrà applicare la funzione di *map* f sui propri dati locali. L'operazione di *reduce* è invece maggiormente vincolata al concetto di *data locality*: ciò significa che i valori associati ad una medesima chiave dovranno essere raggruppati su un singolo nodo (operazioni di *shuffling*) prima che la funzione di *reduce* g possa essere eseguita su ogni gruppo di valori in parallelo, nel caso in cui la funzione g non sia associativa.

Il vantaggio dell'approccio MapReduce, rispetto ad altri sistemi di computazione parallela, è che fornisce un *framework* ad alto livello per operare su grandi quantità di dati in modo parallelo, nascondendo al programmatore tutti i dettagli non necessari: permette quindi di concentrarsi solo sulla computazione da eseguire sui dati, lasciando al *framework* il compito di occuparsi di come effettivamente eseguire la computazione.

2.2.3 Limitazioni di Hadoop

Hadoop ha notevolmente semplificato la gestione della computazione distribuita su *cluster* ma presenta alcuni evidenti limiti a causa del modello di programmazione MapReduce e della sua architettura: infatti, nonostante sia in grado di facilitare lo sviluppo di applicazioni parallele, non tutti gli algoritmi possono essere parallelizzati su Hadoop in modo efficiente. Ad esempio algoritmi iterativi, che accedono più volte allo stesso insieme di dati, risultano su Hadoop poco efficienti a causa delle numerose operazioni di *I/O* su disco eseguite [17].

2.3 Spark

Spark è un *framework* di calcolo distribuito scritto in Scala, *open source* e ispirato al paradigma MapReduce, compatibile con l'ecosistema Hadoop. Apache Spark nasce con lo scopo di permettere una esecuzione più efficiente di alcune particolari applicazioni su *cluster* di hardware *commodity*, superando così le limitazioni di Hadoop.

Spark in particolare ha l'obiettivo di ottimizzare l'esecuzione di applicazioni che utilizzano più volte lo stesso *working set* attraverso multiple operazioni parallele. I casi più tipici sono i seguenti.

- Algoritmi iterativi, in cui una funzione è applicata più volte al medesimo insieme di dati.
- Analisi interattive, in cui l'utente esegue più *query* su uno stesso *dataset* con lo scopo di analizzarlo ed esplorarlo.

Grazie al *caching* dei dati in memoria e a specifiche ottimizzazioni Spark risulta da 10 fino a 100 volte più veloce rispetto a Hadoop MapReduce [17].

Apache Spark richiede un *cluster manager* e uno *storage* distribuito. Come *cluster manager* Spark supporta sia la modalità *standalone* (utilizzando il proprio *cluster manager*) che l'utilizzo di Hadoop YARN o Apache Mesos. Per quanto riguarda lo *storage* distribuito Spark è in grado di lavorare su HDFS, Amazon S3, Cassandra, OpenStack Swift e altri. Spark implementa inoltre una propria modalità di memorizzazione dati pseudo-distribuita, solitamente utilizzata nelle fasi *testing* e sviluppo, in grado di utilizzare il *file system* locale.

Spark è suddiviso nei seguenti moduli:

- **Spark Core**: è il modulo principale e fornisce i servizi per l'invio di *task*, le operazioni di *I/O* e lo *scheduling*, esposti tramite un'interfaccia disponibile per più linguaggi (Java, Python, Scala e R).
- **Spark SQL**: fornisce supporto per l'elaborazione di dati strutturati.
- **Spark Streaming**: comprende una serie di funzionalità per l'analisi di *stream* di dati.
- **MLlib**: è il *framework* utilizzato per il *machine learning* distribuito.
- **GraphX**: è il modulo utilizzato per il *graph processing* distribuito.

2.3.1 Architettura

Spark utilizza un'architettura *master/worker*, caratterizzata da due tipologie principali di processi chiamati *driver* (*master*) ed *executor* (*worker*) e da un *cluster manager*, come visibile in figura 2.2. Ogni applicazione Spark consiste in un *driver program* e vari *executor*.

- **Driver program**: è il processo che esegue la funzione `main()` e che crea lo SparkContext.
- **Executor**: è un processo lanciato su un nodo *worker*, in grado di eseguire *task*, mantenere i dati in memoria o su disco e ritornare i risultati della computazione al *driver*.
- **Cluster manager**: è un servizio esterno necessario per acquisire risorse sul *cluster*.

Le applicazioni Spark diventano un insieme di processi indipendenti sul *cluster*, coordinati dallo SparkContext. Prima dell'esecuzione, SparkContext si connette a uno dei *cluster manager* disponibili (ad esempio YARN), che allocherà le risorse necessarie all'applicazione. Una volta connesso, Spark acquisisce gli *executor* disponibili sui nodi e manda a ognuno di questi sia il codice dell'applicazione che i *task* da eseguire. Ogni *executor* eseguirà poi le operazioni necessarie sulla partizione di dati a propria disposizione.

2.3.2 Modello di programmazione

L'astrazione principale utilizzata da Spark Core è chiamata *Resilient Distribuited Dataset* (RDD) e rappresenta una collezione in sola lettura di oggetti partizionati sui nodi del *cluster* che possono essere ricostruiti nel caso in cui la partizione venga persa. Sugli RDD possono inoltre essere eseguite vari tipi di operazioni ad alto livello.

RDD

Gli elementi di un RDD non devono necessariamente esistere nella memoria fisica e possono essere ricostruiti in caso di fallimento di un nodo, assicurando così tolleranza ai guasti. Spark permette di creare un RDD in 4 modi.

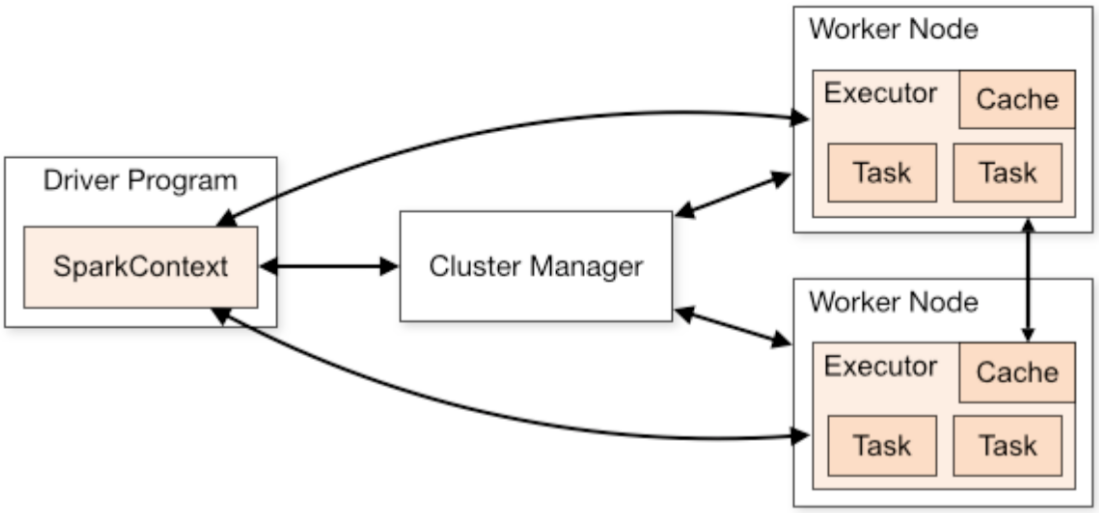


Figura 2.2: Architettura di Spark. Immagine da [13].

- Partendo da un file, salvato ad esempio su HDFS.
- Parallelizzando una collezione dati di Scala (ad esempio un *array*), suddividendola cioè tra più nodi del *cluster*.
- Trasformando un RDD esistente, cioè applicando funzioni sugli oggetti dell'RDD.
- Cambiando la modalità di persistenza di un RDD.

Operazioni sugli RDD

Gli RDD supportano due tipi di operazioni, chiamate *transformation* (trasformazione) e *action* (azione). Le operazioni di *transformation* creano un nuovo *dataset* a partire da quello esistente mentre le operazioni di *action* ritornano un valore dopo che è terminata la computazione sull'insieme dei dati. Tutte le operazioni di trasformazione sono *lazy* in Spark, nel senso che vengono eseguite solo nel momento in cui un'operazione di azione richiede un risultato: questo approccio rende Spark più efficiente. Normalmente ogni RDD trasformato viene ricalcolato ogni volta che viene applicata un'azione su di esso: per evitare di eseguire più volte le stesse operazioni è possibile salvare in memoria l'RDD trasformato,

rendendo così l'esecuzione dell'azione più rapida. Le operazioni di trasformazione possono essere di due tipi, come riportato di seguito.

- *Narrow*: l'RDD in *output* ha partizioni originate da una singola partizione dell'RDD da cui deriva. Trasformazioni di tipo *narrow* sono ad esempio le operazioni `map()` e `filter()`.

- *Wide*: i dati necessari per calcolare una singola partizione in un RDD possono essere sparsi in più partizioni dell'RDD padre. È il caso di trasformazioni come `groupByKey()` o `reduceByKey()`. In questo caso Spark dovrà effettuare un'operazione di *shuffling*, trasferendo dati attraverso il *cluster*.

Persistenza degli RDD

Una delle funzionalità più importanti di Spark è la memorizzazione di un insieme di dati in memoria. Normalmente ogni RDD viene ricalcolato ogni volta che è applicata su di esso un'azione. Persistendo un RDD, ogni nodo memorizza le partizioni del *dataset* a cui è associato per riutilizzarle nel momento in cui vengono applicate altre azioni sul medesimo *dataset*. Ciò consente a Spark di essere molto più efficiente e veloce nel caso di algoritmi iterativi o di analisi interattive. Ogni RDD può essere salvato utilizzando diverse modalità di memorizzazione: i dati possono ad esempio essere salvati in memoria o su disco.

2.3.3 Spark SQL

Spark SQL è il modulo di Spark per l'elaborazione di dati strutturati. Diversamente dalle *API* fornite da Spark RDD, l'interfaccia di Spark SQL è in grado di eseguire ulteriori ottimizzazioni grazie al maggior numero di informazioni in possesso sui dati e sulla computazione da eseguire. Sia Spark Core che Spark SQL utilizzano comunque lo stesso motore computazionale, dunque lo sviluppatore è libero di usare differenti *API*, anche nel medesimo programma. Spark SQL permette di esprimere interrogazioni in linguaggio SQL e di interfacciarsi con Hive.

Dataset

Un Dataset è una collezione distribuita di dati. L'interfaccia Dataset è stata introdotta a partire da Spark 1.6 e mantiene tutti i benefici degli RDD (*strong typing* e possibilità di utilizzare funzioni *lambda*), a cui si aggiungono le ottimizzazioni di Spark SQL. Un Dataset può essere creato a partire oggetti JVM e poi manipolato attraverso funzioni di trasformazione come ad esempio `map()` e `filter()`.

DataFrame

Un DataFrame è un Dataset organizzato in colonne, ognuna delle quali associata a un'etichetta. È concettualmente equivalente a una tabella di un *database* relazionale e può essere creato a partire da file dati strutturati, tabelle Hive, *database* esterni o RDD. L'interfaccia Dataframe è disponibile per Scala, Java, Python e R. A differenza di RDD, non fornisce le funzionalità di *type-checking* a tempo di compilazione.

Hive e Spark SQL

Apache Hive è il progetto di *data warehouse* costruito su Hadoop che permette di analizzare, interrogare e ottenere informazioni di sintesi dai dati. Hive fornisce una interfaccia simile a quella SQL (HiveQL) per le interrogazioni sui dati, permettendo di utilizzare un approccio dichiarativo e rendendo trasparente il sottostante livello Java: si occupa infatti anche della conversione delle interrogazioni in *job* MapReduce o Spark. Le tabelle utilizzate da Hive sono concettualmente identiche a quelle dei tipici *database* relazionali.

A livello architetturale, i componenti principali di Hive sono i seguenti.

- *Metastore*: memorizza i metadati di ogni tabella, come schema e localizzazione.
- *Driver*: è il componente centrale dell'architettura. Riceve le *query* HiveQL, le esegue e monitora il progresso dell'esecuzione.
- *Compiler*: è invocato dal *driver* alla ricezione di una interrogazione HiveQL. Converte le *query* in un *AST*, ne controlla la correttezza, e genera un piano di esecuzione (DAG) interpretabile da Hadoop MapReduce o Spark.

- *Optimizer*: esegue delle trasformazioni sul piano di esecuzione con il fine di ottimizzarne l'esecuzione.

- *CLI*, *Web GUI*, e Thrift Server: rappresentano le interfacce di interazione tra l'utente e Hive. Thrift server in particolare permette di interagire con Hive attraverso i protocolli JDBC or ODBC.

Spark SQL supporta la sintassi di HiveQL ed è in grado di eseguire letture e scritture sui dati memorizzati su Hive. Spark SQL fornisce inoltre una serie di *API* che permettono di persistere il contenuto di un DataFrame come tabelle Hive in modo diretto.

2.3.4 Prestazioni

Spark risulta notevolmente più veloce rispetto ad Hadoop nell'esecuzione di algoritmi iterativi, grazie alla possibilità di memorizzare i dati in memoria. Se infatti la prima iterazione viene eseguita in tempi paragonabili a quelli di Hadoop, dalla seconda in poi il tempo necessario per ogni iterazione cala notevolmente, grazie al meccanismo di *caching* dei dati in memoria. Vale lo stesso discorso per le interrogazioni interattive: la prima *query* ha lo stesso tempo di esecuzione su Spark e Hadoop, ma dalla seconda in poi Spark risulta nettamente più rapido rispetto ad Hadoop [17].

2.4 GeoSpark

GeoSpark è un *framework* per l'elaborazione *in-memory* di dati spaziali in ambiente distribuito e fornisce una serie di astrazioni per caricare, processare e analizzare dati spaziali: l'obiettivo di GeoSpark è quello di fornire una piattaforma in grado di gestire grandi moli di dati e di processarli in tempi rapidi, per fornire all'utente risultati nel più breve tempo possibile. GeoSpark nasce come estensione di Spark Core: l'idea è quella di sfruttare ed estendere gli RDD per supportare tipi di dati spaziali, indici e operazioni su dati spaziali in modo efficiente. GeoSpark è composto da tre moduli, elencati di seguito.

- **Core**: basato su Spark Core, mette a disposizione gli *Spatial Resilient Distributed Datasets* e operatori per effettuare interrogazioni.
- **SQL**: è l'interfaccia SQL di GeoSpark, basata su Spark SQL.
- **Viz**: fornisce un'interfaccia per la visualizzazione dati.

2.4.1 Architettura

GeoSpark presenta un'architettura a 3 livelli. Il primo livello è quello di Apache Spark e include tutte le operazioni e astrazioni discusse nelle precedenti sezioni. Il secondo livello è definito *Spatial Resilient Distributed Dataset* (SRDD) *Layer* ed estende gli RDD per supportare oggetti geometrici e operazioni su di essi. Il terzo livello è chiamato *Spatial Query Processing Layer* ed estende l'SRDD *Layer* per permettere l'esecuzione di interrogazioni spaziali su insiemi di dati di grandi dimensioni.

2.4.2 Spatial RDD layer

L' SRDD *Layer* estende Spark con gli SRDD per supportare efficacemente il partizionamento di dati spaziali tra i nodi e introduce una serie di trasformazioni e azioni per gli SRDD, in grado di fornire un'interfaccia intuitiva per lo sviluppo di software di analisi spaziale.

Spatial RDD

In GeoSpark sono definiti diversi tipi di RDD spaziali, come estensione dei tradizionali RDD di Spark, per la gestione e il partizionamento dei dati spaziali. Geospark utilizza per la memorizzazione di oggetti spaziali la JTS Topology Suite, una libreria Java per la creazione e la manipolazione di geometrie vettoriali: ogni oggetto spaziale è memorizzato come punto, rettangolo o poligono. GeoSpark fornisce inoltre una serie di operazioni per gli SRDD, appoggiandosi sulle *API* di Apache Spark, come ad esempio `Overlap()`, per trovare tutti gli oggetti intersecati tra geometrie, o `Union()`, che ritorna un poligono come unione di tutti i poligoni di un SRDD.

SRDD Indexing

GeoSpark offre una serie di indici per ottimizzare l'esecuzione di *query* spaziali, come Quad-Tree o R-Tree, forniti da estensioni degli SRDD, gli *Spatial IndexRDD*. L'utilizzo di questi indici permette di ottenere prestazioni migliori durante l'esecuzione di operazioni particolarmente onerose, come ad esempio operazioni di *spatial join* [16]. La creazione di uno *Spatial IndexRDD* può essere decisa in modo esplicito dall'utente o affidata a GeoSpark, che in modo dinamico è in grado di determinare la convenienza o meno di un indice tenendo conto dell'*overhead* (in termini di spazio e tempo) dell'indice, nonché delle caratteristiche dell'interrogazione e dell'insieme dati.

2.4.3 Spatial Query Processing Layer

Lo *Spatial Query Processing Layer* ha lo scopo di permettere l'utilizzo di interrogazioni spaziali su insieme dati di grandi dimensioni. L'esecuzione delle *query* è ottimizzata da GeoSpark attraverso il partizionamento dello spazio dei dati, gli indici spaziali e la computazione in memoria. Di seguito sono riportate le principali tipologie di interrogazioni supportate.

- **Spatial Range Query**: ritorna tutti gli oggetti che cadono all'interno di una determinata *query window*.
- **Spatial Join Query**: esegue un *join* tra oggetti spaziali sulla base della loro relazione spaziale.
- **Spatial KNN Query**: dato un punto P, ritorna i K oggetti più vicini a P.

2.4.4 GeoSpark SQL

GeoSpark SQL nasce come alternativa ai tradizionali *database* relazionali spaziali (ad esempio PostGIS/PostgreSQL), non in grado di fornire *performance* adeguate se utilizzati su grandi volumi di dati. GeoSpark SQL offre sia un'interfaccia SQL per eseguire

interrogazioni spaziali sui dati, sia un motore computazionale efficiente e in grado di lanciare più operazioni in parallelo, utilizzando la tecnologia offerta da Hive e Spark.

2.4.5 Prestazioni

GeoSpark è in grado di eseguire interrogazioni spaziali in tempi ridotti rispetto a *framework* simili sviluppati su Hadoop, come ad esempio SpatialHadoop, grazie alla computazione *in-memory* [16]. La figura 2.3 mette a confronto i tempi di esecuzione di un'operazione di *spatial join* su *dataset* di differenti dimensioni: GeoSpark risulta più prestante rispetto a SpatialHadoop su entrambi gli insieme dati, a prescindere dagli indici utilizzati.

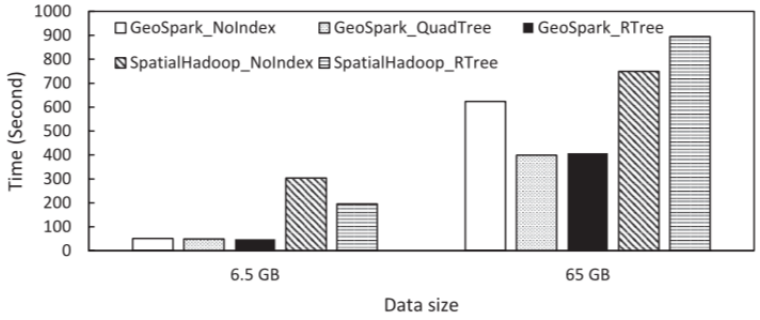


Figura 2.3: *Performance* di GeoSpark e SpatialHadoop per *spatial join* su *dataset* diversi. Immagine da [16].

GeoSpark SQL risulta più veloce nelle *query kNN* rispetto a PostGIS/PostgreSQL, come mostrato in figura 2.4, e ESRI Spatial Framework for Hadoop [8]. Inoltre, comparato sempre a PostGIS/PostgreSQL, GeoSpark esegue in meno della metà del tempo operazioni di *spatial join* [8], come mostrato in figura 2.5. GeoSpark SQL risulta particolarmente efficiente quindi in operazioni ad elevato costo computazionale, grazie alla computazione *in-memory* e alle capacità di calcolo parallelo di Spark [8].

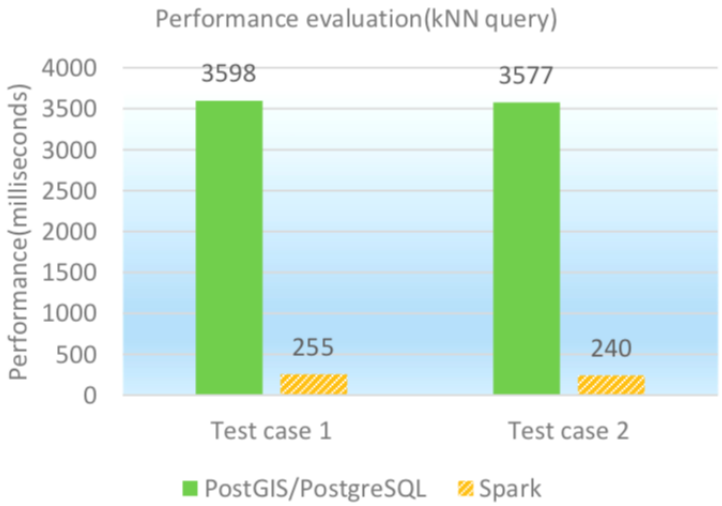


Figura 2.4: Confronto tra GeoSpark e PostGIS/PostgreSQL nell'esecuzione di *query kNN* su *dataset* di punti (Test case 1) e su *dataset* di poligoni (Test case 2). Immagine da [16].

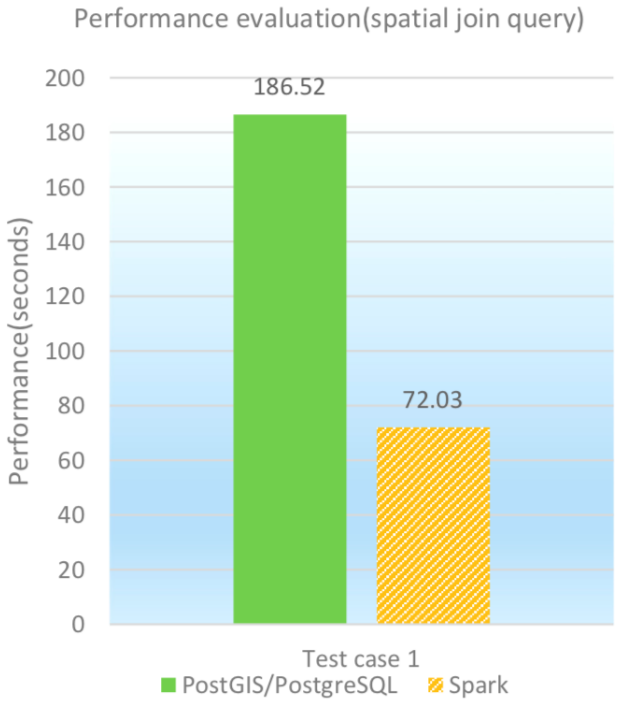


Figura 2.5: Confronto tra GeoSpark e PostGIS/PostgreSQL nell'esecuzione di una *query* di *spatial join*. Immagine da [16].

Capitolo 3

Le tecniche per l'analisi di dati di traiettoria

Il diffondersi di *device* capaci di geolocalizzarsi e lo sviluppo di nuove tecniche per l'acquisizione della posizione ha generato negli ultimi anni una grande disponibilità di dati di traiettorie spaziali, relativi ad oggetti in movimento nello spazio geografico. Questa disponibilità ha reso necessaria la messa a punto di tecniche di analisi dati *ad hoc*, in grado di fornire soluzioni a problematiche negli ambiti, ad esempio, del trasporto, dell'ecologia, della sorveglianza e della sicurezza [10]. In questa sezione si introdurrà il concetto di *trajectory mining* e si analizzeranno alcuni specifici algoritmi di *clustering* spazio-temporale, facendo riferimento a studi e lavori già presenti in letteratura. Infine si presenterà PatchWork, un performante algoritmo di *clustering grid/density-based* per la piattaforma Apache Spark.

3.1 Processo di analisi in letteratura

Il processo di *mining* delle traiettorie può essere considerato come parte di un *framework* più ampio [10], che spazia dalla creazione e gestione dei dati, fino all'estrazione di informazioni. Come mostrato in figura 3.1, gli oggetti in movimento dotati di appositi sensori e in grado di produrre dati spaziali possono essere di diversi tipi, ad esempio un'automobile,

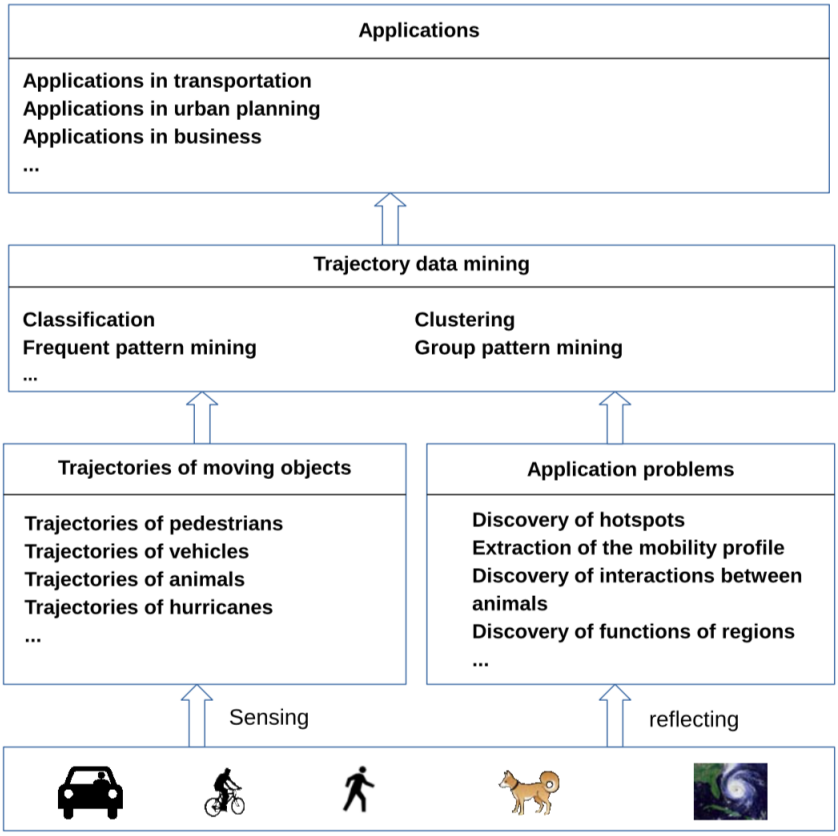


Figura 3.1: Esempio di *framework* per *trajectory mining*. Immagine da [10].

un essere umano o un animale: ognuno di questi oggetti genera, muovendosi nello spazio, un insieme di traiettorie. I dati ottenuti attraverso il monitoraggio di questi oggetti hanno una valenza doppia: da una parte stimolano dubbi e domande che vengono formalizzate in problemi applicativi, dall'altra rappresentano una preziosa fonte di informazioni. L'applicazione di tecniche di *trajectory data mining* su questi dati rappresenta la fase cruciale dell'intero processo: permette infatti di estrarre informazioni dai dati e quindi di fornire soluzioni a questi problemi, rilevanti in molti ambiti applicativi.

3.1.1 Traiettorie spaziali e dati di traiettoria

Una traiettoria spaziale è la rappresentazione del percorso che un oggetto esegue attraverso lo spazio in funzione del tempo [19], come susseguirsi di punti spazio-temporali ordinati in modo cronologico. Più formalmente, una traiettoria può essere rappresentata come

$T = \langle p_1 ... p_n \rangle$ dove $p_k = (id_k, loc_k, t_k, A_k)$ è la posizione k, id_k è l'identificativo, loc_k rappresenta la locazione, t_k il tempo di registrazione della posizione e A_k è una lista opzionale di attributi descrittivi [10].

La componente di località spaziale di ogni dato di traiettoria può assumere forme diverse, a seconda della tecnologia utilizzata: le più tipiche sono GPS (*Global Positioning System*), GSM (*Global System for Mobile Communications*) e tecnologie di *geo-social networking*, a cui si aggiungono WiFi e RFID [10]. I dati GPS, ad esempio, sono rappresentati come coordinate geografiche ordinate in modo cronologico, mentre quelli GPS sono sequenze di celle GPS ordinate lungo la dimensione del tempo.

3.1.2 Trajectory data mining

Il *data mining* rappresenta l'attività di estrazione di informazioni da insiemi di dati e fa parte del più ampio processo di *knowledge discovery*, cioè l'estrazione di conoscenza attraverso l'interpretazione delle informazioni. La disponibilità di grandi quantità di dati spaziali ha stimolato lo studio di nuovi metodi di *mining*: le attività del *trajectory data mining* sono simili a quelle del *data mining* tradizionale, ma le tecniche utilizzate sono state adattate per gestire al meglio le specificità dei dati spaziali [18].

Possiamo considerare l'attività di *trajectory data mining* come l'applicazione di più passi successivi. Alla base di tutto ci sono ovviamente i dati, organizzati in traiettorie e ricavati da sensori dedicati. I dati devono spesso subire una fase di *preprocessing* prima di poter essere utilizzati: tipicamente in questa fase si utilizzano tecniche di riduzione del rumore o di compressione di traiettorie. Se il volume di dati è grande si possono generare indici spaziali, per rendere le fasi di *retrieving* e di interrogazione delle traiettorie più performanti. Infine, si applicano sui dati metodi di *mining*, come ad esempio algoritmi di *clustering* o metodi di classificazione, con lo scopo di estrarne informazioni utili alla risoluzione di uno o più problemi applicativi. Di seguito si riportano brevemente le principali classi di metodi utilizzati nell'attività di *trajectory mining*.

- *Clustering*: metodi di *clustering* possono essere applicati sulle traiettorie spaziali a due livelli; a livello di un insieme di traiettorie, ottenendo *cluster* come insiemi di

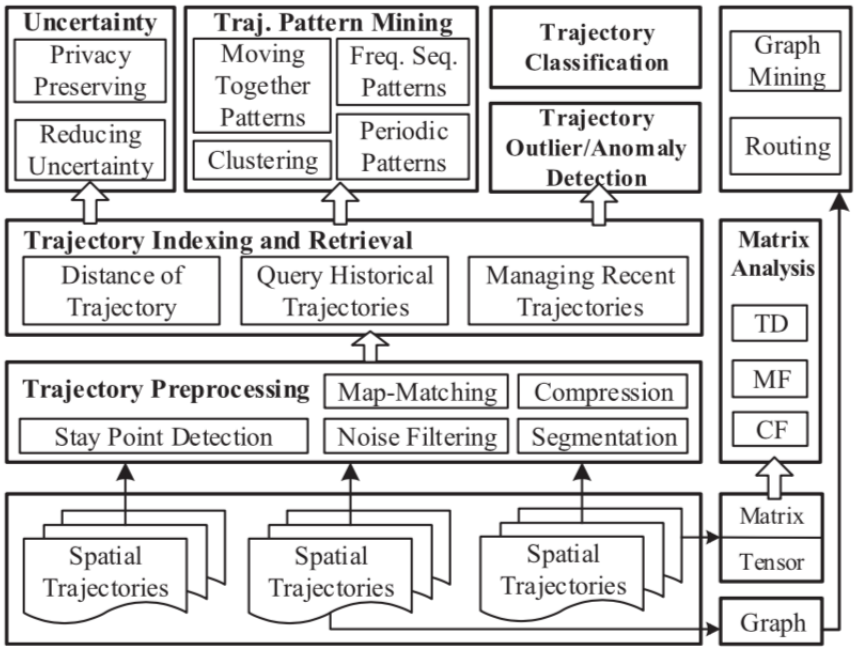


Figura 3.2: Paradigma del *trajectory mining*. Immagine da [18].

traiettorie, o a livello di singole traiettorie, ottenendo *cluster* come insiemi di singoli punti della traiettoria. Attualmente lo stato dell'arte degli algoritmi di *clustering* per traiettorie è rappresentato da estensioni di tradizionali algoritmi di *clustering* [10], modificati opportunamente per supportare una misura di similarità (o distanza) coerente con l'obiettivo dell'analisi e con la natura dei dati spaziali.

- Classificazione: gli algoritmi di classificazione si occupano di assegnare una classe predefinita a ciascuna traiettoria in base a determinate caratteristiche della stessa, come la velocità, la durata o la lunghezza. Molto spesso l'attività di classificazione delle traiettorie avviene in seguito all'applicazione di altri metodi, come *clustering* o riduzione del rumore.

- *Outlier detection*: ha l'obiettivo di rilevare traiettorie che si discostano dalla maggior parte delle traiettorie del *dataset*, cioè si focalizza sulla ricerca di *pattern* anomali.

- Predizione: i metodi di predizione hanno lo scopo di predire la futura locazione di un oggetto in movimento a partire dall'analisi delle traiettorie esistenti. Metodi di

predizione permettono ad esempio di prevedere la destinazione di un utente durante i suoi prossimi movimenti.

- *Pattern mining*: ha lo scopo di scoprire e descrivere *pattern* nascosti nelle traiettorie, considerando sia la dimensione spaziale, sia quella temporale. Il *pattern mining* può riguardare, ad esempio, la ricerca di movimenti ripetuti periodicamente, l'analisi di percorsi che vengono eseguiti frequentemente o l'estrazione di *pattern* a partire dai percorsi di più oggetti che si muovono in gruppo.

3.1.3 Classi di problemi trattabili

Le classi di problemi che possono essere trattate con tecniche di *trajectory mining* sono di notevole interesse. Di seguito ne sono riportate alcune.

- Caratterizzazione di oggetti in movimento: lo scopo è quello di ricavare informazioni che descrivano il movimento degli oggetti. Una istanza di questa classe di problemi è, ad esempio, la profilazione del movimento di individui, per capire quali luoghi prediliga un essere umano o un animale.

- Scoperta di relazioni sociali: attraverso lo studio delle traiettorie si cerca di capire come due o più individui interagiscono tra loro. Un esempio tipico può essere la comprensione dell'interazione tra classi animali, come preda e predatore.

- Caratterizzazione di luoghi o regioni: analizzando gli spostamenti attraverso determinati luoghi e regioni, si vogliono ottenere informazioni in grado di descrivere questi luoghi e queste regioni.

- Riconoscimento di eventi sociali: il *trajectory mining* può essere utilizzato per identificare raduni ed eventi sociali. In certi casi è possibile determinare anche il tipo di evento.

- Predizioni basate sulle traiettorie: l'obiettivo è quello di riuscire a effettuare previsioni, ad esempio sulla destinazione di un oggetto in movimento o su potenziali situazioni di traffico automobilistico, a partire dall'analisi di traiettorie spaziali.

- Raccomandazioni basate sulle traiettorie: le raccomandazioni basate sulle traiettorie sono ricavate a partire dall'analisi della cronologia degli spostamenti. L'assunto di fondo è che persone con interessi e gusti simili abbiano anche *pattern* di mobilità simili.

3.2 Algoritmi di *clustering* spazio-temporale

Il *clustering* spazio-temporale è un processo di raggruppamento di oggetti in base alle loro caratteristiche spaziali e temporali. Pur essendo un campo relativamente nuovo nell'ambito del *data mining*, ha acquisito una notevole importanza a causa delle sue ricadute soprattutto nell'ambito della *Geographic Information Science* (GIScience) [9]. Di seguito sono presentati due algoritmi per il *clustering* spazio-temporale di dati, nell'ottica di studiare le possibili modalità di estrazione di "luoghi di interesse per un utente" a partire dai suoi dati spazio-temporali.

3.2.1 DJ-Cluster

DJ-Cluster (*Density Join Cluster*) [20] è un algoritmo di ricerca dei "luoghi di interesse" di un utente: con luogo di interesse si possono intendere la propria casa, il luogo di lavoro o ad esempio i locali frequentati. A partire dai dati spazio-temporali degli utenti, attraverso l'applicazione di tecniche di *preprocessing* e di algoritmi di *clustering*, DJ-Cluster è in grado di ricavare il dizionario dei luoghi di interesse di una persona, cioè il suo *personal gazetteer*. DJ-Cluster è un algoritmo di *clustering density* e *join-based* che nasce come estensione di DBSCAN [4], con il fine di superare alcune limitazioni di quest'ultimo. DBSCAN è infatti estremamente sensibile ai parametri *Eps* e *MinPoints*, risultando in alcune situazioni poco performante [20]. DJ-Cluster si basa sull'idea di densità del vicinato (*density-based neighborhood*), cioè sul numero dei punti a distanza minore di un raggio di dimensione *Eps* da un punto dell'insieme dati. DJ-Cluster calcola per ogni punto il vicinato: se non ci sono punti o i punti sono in numero inferiore a *MinPoints* allora il punto è annotato come punto di *noise*. Se invece i punti sono in numero uguale o

maggiore di *MinPoints*, i punti del vicinato andranno a costituire un nuovo *cluster*, se tra i punti del vicinato non è presente nemmeno un punto già appartenente ad un *cluster* esistente, o saranno uniti ad un *cluster* già esistente, se tra i punti del vicinato è presente almeno un punto già appartenente ad un *cluster*. In questo ultimo caso l'insieme di punti è definito come *density-joinable*, cioè può essere unito ad un *cluster* esistente attraverso un'operazione di *join*.

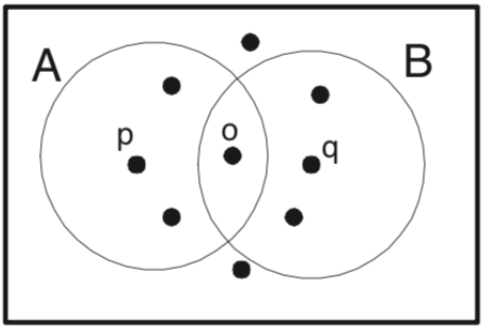


Figura 3.3: Esempio di relazione *density-joinable*. Immagine da [20].

Più formalmente il *density-based neighborhood* N di un punto p, denotato con $N(p)$, è definito come:

$$N(p) = \{q \in S \mid dist(p, q) \leq Eps\} \tag{3.1}$$

dove S è l'insieme dei punti, q è un qualunque punto dell'insieme dei punti, Eps è il raggio del cerchio con centro p. Due *density-based neighborhood* distinti possono formare un nuovo *cluster* se *density-joinable*: $N(p)$ è *density-joinable* con $N(q)$, denotato con $J(N(p), N(q))$, se esiste un punto o contenuto in $N(p)$ e $N(q)$, come mostrato in figura 3.3. Possiamo quindi definire un *cluster* C *density* e *join-based* come segue:

$$\forall p \in S, \forall q \in S, \exists N(p), N(q) : \exists J(N(p), N(q)) \tag{3.2}$$

DJ-Cluster ha una complessità computazionale di $O(n^2)$ se non si utilizza un indice spaziale e di $O(n \log n)$ se si utilizza un indice R-tree [20].

Preprocessing temporale dei dati

L'applicazione del solo algoritmo DJ-Cluster sui dati spazio-temporali grezzi genera una grande quantità di *cluster* (cioè luoghi) poco interessanti ai fini della creazione di un *personal gazetteer*, come semafori o attraversamenti pedonali. Questi *cluster* si formano perché, ad esempio, una strada può essere percorsa spesso per andare a lavoro o all'università e i punti di stop lungo questa generano un accumulo di dati poco significativi. Per migliorare la qualità dei risultati e contemporaneamente aumentare le *performance* dell'algoritmo, gli autori hanno eseguito sui dati due operazioni preliminari di *processing*.

La prima operazione è stata l'eliminazione di tutti i punti con velocità maggiore di 0: la velocità è fornita per ogni punto direttamente dai sensori GPS, come stima a partire dalla distanza percorsa tra due punti successivi. In questo modo sono stati rimossi molti punti poco interessanti, rilevati ad esempio mentre si guida un'automobile. La seconda operazione ha riguardato invece la rimozione dei punti troppo vicini rispetto alla lettura precedente: in questo modo si ottiene una riduzione della cardinalità dell'insieme dei punti, permettendo all'algoritmo di ottenere *performance* migliori.

3.2.2 ST-DBSCAN

ST-DBSCAN (*Spatio-Temporal* DBSCAN) [3] è un algoritmo di *clustering density-based*, nato come estensione di DBSCAN [4]. La caratteristica più interessante di ST-DBSCAN è la capacità di eseguire *clustering* su dati spazio-temporali a partire da attributi spaziali, temporali e non spaziali. DBSCAN utilizza un solo parametro di distanza *Eps* per definire la similarità tra due punti, limitando di fatto la misura ad una sola dimensione. ST-DBSCAN utilizza invece due metriche di distanza, $Eps1$ e $Eps2$, per definire la similarità sulla base di differenti caratteristiche di densità: $Eps1$ è utilizzato per definire la similarità sulla dimensione spaziale, andando quindi a misurare la vicinanza tra due punti in termini geografici, mentre $Eps2$ viene usato per definire la similarità tra due punti in base alle loro caratteristiche non spaziali.

ST-DBSCAN ha, rispetto a DBSCAN, altre due importanti caratteristiche. In primo luogo è in grado di individuare punti di *noise* anche su *cluster* di differente densità,

sulla base del calcolo del grado di densità di ogni *cluster*. In secondo luogo, ST-DBSCAN evita che gli oggetti posti ai bordi dei *cluster* siano eccessivamente differenti tra loro per caratteristiche non spaziali, confrontando le caratteristiche degli oggetti candidati ad entrare in un *cluster* con la media degli oggetti del *cluster*. La complessità computazionale media di ST-DBSCAN è uguale a quella di DBSCAN, quindi pari a $O(n \log n)$ [3].

Algoritmo

ST-DBSCAN richiede quattro parametri di input, $Eps1$, $Eps2$, $MinPoints$ e Δ_ϵ. $Eps1$ è il parametro relativo alla distanza spaziale, ad esempio misurata tramite latitudine e longitudine, mentre $Eps2$ è il parametro di distanza per gli attributi non spaziali. $MinPoints$ è il minimo numero di punti che devono essere presenti entro le distanze definite da $Eps1$ e $Eps2$. Δ_ϵ rappresenta invece una soglia di differenza tra il valore di un determinato attributo di un oggetto candidato alla partecipazione ad un *cluster* e la media dei valori, per quell'attributo, degli oggetti del medesimo *cluster*.

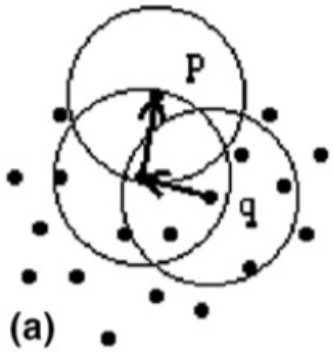


Figura 3.4: Esempio di *density-reachability*: il punto p è *density-reachable* dal punto q. Immagine da [3].

L'algoritmo prende in considerazione il primo punto p e determina se ha un numero sufficiente di vicini, considerando i valori degli attributi $Eps1$, $Eps2$ e $MinPoints$: in caso positivo il punto p viene etichettato come punto *core* e viene creato un nuovo *cluster*. Tutti i punti *directly density-reachable* dal punto p, cioè quelli a distanza minore di $Eps1$ e $Eps2$ da p, sono aggiunti al *cluster*. L'algoritmo colleziona poi in modo iterativo tutti i punti *density-reachable* dal punto p (vedi figura 3.4). Se un oggetto non è di tipo *noise*,

non appartiene ad un *cluster* e la differenza tra la media dei valori del *cluster* e il nuovo valore è inferiore a Δ_ϵ, allora viene assegnato al *cluster* corrente. L'algoritmo seleziona poi il punto seguente dal *dataset* e ripete il medesimo processo fino a che tutti i punti non sono stati processati.

Per supportare gli aspetti temporali, i dati spazio-temporali vengono dapprima filtrati mantenendo solo i vicini temporali e i loro corrispondenti valori spaziali: due oggetti sono vicini temporali se i valori di questi oggetti sono osservati in unità temporali consecutive, come giorni consecutivi nello stesso anno o nello stesso giorno in anni consecutivi. Solo a questo punto i valori spaziali e non-spaziali di un punto p vengono confrontati con i corrispettivi valori degli altri punti.

3.3 PatchWork

PatchWork [6] è un algoritmo di *clustering* distribuito *density-based* e *grid-based* implementato sulla piattaforma di calcolo distribuito Apache Spark. Come altri algoritmi *density-based*, presenta alcune caratteristiche peculiari che lo rendono adatto ad un uso in ambito *data mining*: non necessita di un numero di *cluster* definito a priori, può scoprire *cluster* di dimensione arbitraria ed è in grado di identificare efficacemente punti di *noise* e *outliers*.

A differenza di altre implementazioni di algoritmi di *clustering*, come ad esempio DJ-Cluster, presenta una complessità computazionale lineare: ciò lo rende particolarmente adatto alla gestione di grandi quantità di dati e all'utilizzo su piattaforme di elaborazione *Big Data*.

3.3.1 Parametri

PatchWork offre un serie di parametri definibili dall'utente che permettono di adattare l'esecuzione dell'algoritmo alle proprie esigenze.

- **CellSize (ϵ)**: ϵ rappresenta il parametro che determina la dimensione delle celle su ciascuna delle D dimensioni dello spazio. È concettualmente simile a Eps, ma permette di decidere un valore differente per ogni dimensione in modo indipendente.
- **MinPoints**: è il numero minimo di punti che una cella deve contenere per poter essere considerata come parte di un *cluster*.
- **Ratio**: esprime una soglia di densità necessaria per poter espandere un *cluster*.
- **MinCell**: permette di filtrare *cluster* formati da un numero di celle troppo esiguo.

3.3.2 L'algoritmo

L'algoritmo, come prima cosa, suddivide lo spazio dei punti in una griglia multi-dimensionale, in modo da poter identificare più efficientemente regioni ad elevata densità di punti. Ad ogni punto viene poi associata una cella, cioè un ipercubo dello spazio multi-dimensionale identificato da un $CellID$, della griglia: preso un punto $P = (p_1, \ldots, p_i, \ldots, p_D) \in \Re^D$, P appartiene alla cella K tale che $K = (k_1, \ldots, k_i, \ldots, k_D)$ e $\forall i \, [1, D] \, , k_i = \lfloor p_i / \varepsilon_i \rfloor$. Essendo il processo di associazione di un punto ad una cella indipendente rispetto alle caratteristiche dei restanti punti, questa parte dell'algoritmo può essere eseguita in modo parallelo su più macchine. Questa fase di associazione tra punti e celle viene eseguita attraverso un'operazione di *mapping*, di complessità $O(1)$, su un RDD contenente i punti: per ogni punto P della collezione, la funzione di *mapping* ritorna una tupla ($cellID(P)$,1), dove $cellID(P)$ rappresenta una cella della griglia.

$$map(P(p_1, \ldots, p_D)) \Rightarrow ((\lfloor p_1/\varepsilon_i \rfloor, \ldots, \lfloor p_D/\varepsilon_i \rfloor), 1) \tag{3.3}$$

Le tuple ($cellID(P)$,1) ottenute vengono raggruppate per $cellID(P)$ e poi ridotte attraverso l'utilizzo dell'operatore di somma. Il risultato è una collezione di m_ϵ tuple $(cellID(P), density)$, dove $density$ rappresenta il numero totale di punti presenti in una data cella.

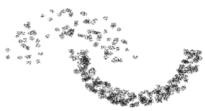
L'algoritmo, a questo punto, crea i *cluster* utilizzando la collezione di celle e le relative densità: essendo $m_\epsilon << n$, dove n è il numero totale dei punti, questa operazione

avrà un costo computazionale estremamente limitato. Se è stato specificato un valore di $minPoints$, l'algoritmo preliminarmente rimuove le celle con $Density(C) < minPoints$. Ordina poi le celle in modo decrescente in base alla densità e crea il primo *cluster* C utilizzando la prima cella, cioè quella a densità maggiore. Infine esplora le celle C_i contigue e le unisce al *cluster* C se hanno una densità sufficiente, cioè se $Density(C_i) > Density(C) * Ratio$. Questo step viene ripetuto per tutte le celle della collezione. Definendo il parametro $MinCell$, i *cluster* trovati possono essere filtrati in base al numero di celle che comprendono: in questo modo è possibile considerare solo i *cluster* sufficientemente estesi nello spazio.

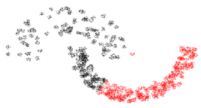
3.3.3 Risultati e prestazioni

PatchWork è stato confrontato con altri algoritmi di *clustering* implementati sulla piattaforma Apache Spark, nello specifico Spark-DBSCAN [14] e MLlib k-means [11]. PatchWork, data la sua natura *density-based*, risulta adatto alla rilevazione di *cluster* di forma arbitraria, a differenza di MLlib k-means, come mostrato nelle figure 3.5f e 3.5h. In particolare i risultati su *cluster* di dimensione arbitraria sono paragonabili a quelli di Spark-DBSCAN [6], implementazione dell'algoritmo DBSCAN per Spark, come mostrato nelle figure 3.5h e 3.5g. A differenza di quest'ultimo, PatchWork è in grado di filtrare i *cluster* a ridotta estensione spaziale, attraverso la definizione del parametro *MinCell*, come mostrato nelle figure 3.5o e 3.5p.

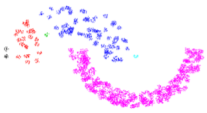
PatchWork è stato testato anche sul *dataset* SFPD Incidents: il *dataset* comprende 1,7 milioni di crimini avvenuti a San Francisco ed è fornito dal San Francisco Police Department (SFPD) Crime Incident Reporting System. Per testare PatchWork sono state considerate 3 dimensioni: latitudine, longitudine e categoria di crimine. La figura 3.6 mostra i risultati sul *dataset* di tre algoritmi di *clustering*. MLLib k-means suddivide il *dataset* in k=7 partizioni di uguale dimensione, mentre Spark-DBSCAN crea un unico grande *cluster* che comprende quasi tutta la città. PatchWork crea invece un *cluster* più grande e molti *cluster* di media dimensione. Il vantaggio di PatchWork rispetto a DBSCAN è quello di permettere di definire valori differenti di *Eps* per ogni dimensione, rendendo



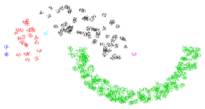
(a) *Dataset* Jain



(b) MLlib K-Means su *dataset* Jain

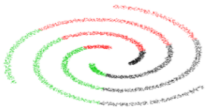


(c) Spark-DBSCAN su *dataset* Jain

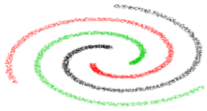


(d) PatchWork su *dataset* Jain

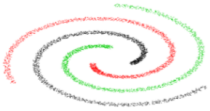
(e) *Dataset* Spiral



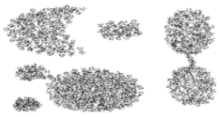
(f) MLlib K-Means su *dataset* Spiral



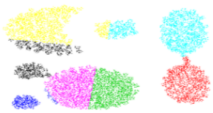
(g) Spark-DBSCAN su *dataset* Spiral



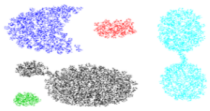
(h) PatchWork su *dataset* Spiral



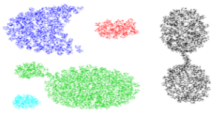
(i) *Dataset* Aggregation



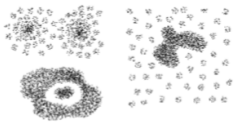
(j) MLlib K-Means su *dataset* Aggregation



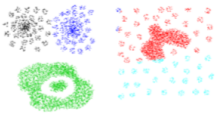
(k) Spark-DBSCAN su *dataset* Aggregation



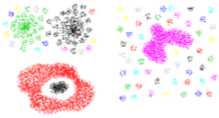
(l) PatchWork su *dataset* Aggregation



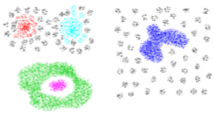
(m) *Dataset* Compound



(n) MLlib K-Means su *dataset* Compound



(o) Spark-DBSCAN su *dataset* Compound

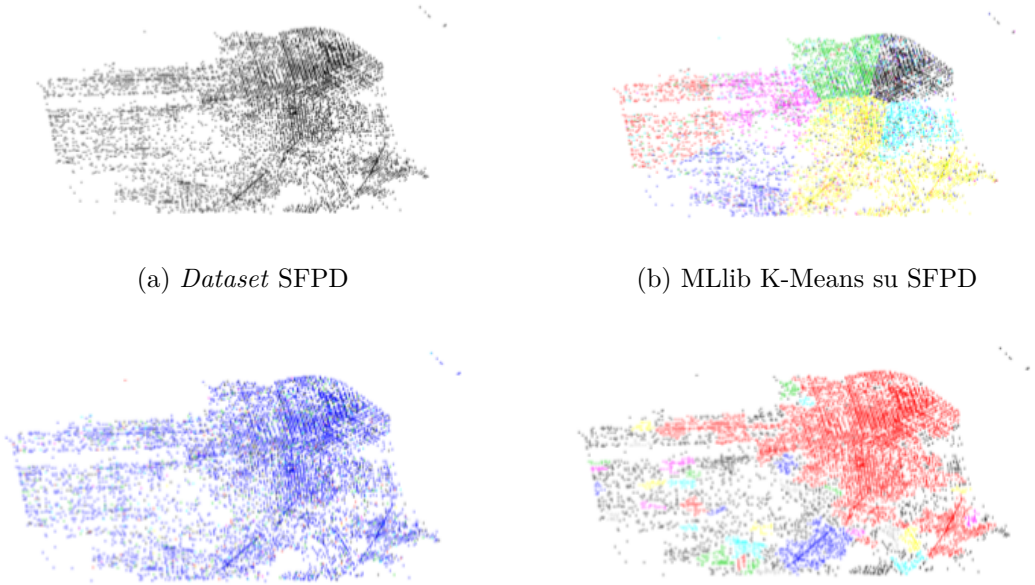


(p) PatchWork su *dataset* Compound

Figura 3.5: Confronto tra MLLib K-Means, Spark-DBSCAN e PatchWork su *dataset* sintetici. Immagine da [6].

così più semplice la modellazione del *dataset* [6].

Per il *clustering* di insiemi di dati di grandi dimensioni, superiori al milione di oggetti, PatchWork risulta essere più veloce rispetto ad altre implementazioni *K-means* o *density-based* su Spark grazie alla complessità di computazione lineare [6].



(a) *Dataset* SFPD

(b) MLlib K-Means su SFPD

(c) Spark-DBSCAN su SFPD

(d) PatchWork su SFPD

Figura 3.6: Confronto su *dataset* SFPD. Immagine da [6].

Capitolo 4

Il prototipo realizzato

In questa sezione viene presentato il prototipo realizzato all'interno del Business Intelligence Group a supporto del programma "Lacittàintorno" di Fondazione Cariplo: il programma si propone di di sviluppare e migliorare il benessere e la qualità della vita per gli abitanti dei quartieri intorno al centro storico di Milano, attraverso iniziative economiche, culturali e creative. L'obiettivo del prototipo è quello allora di aiutare a realizzare un'analisi dei profili e dei luoghi di interesse delle persone che si spostano su due dei quartieri coinvolti nell'iniziativa, Adriano e Chiaravalle, con il fine di ripetere l'analisi a seguito dell'attività di riqualificazione e di attestare i cambiamenti nelle persone interessate da tali quartieri.

4.1 Il prototipo

Il prototipo si pone l'obiettivo di aiutare la comprensione delle abitudini e degli interessi dei cittadini coinvolti dal progetto a partire dall'elaborazione e dall'analisi di dati di traiettoria spaziale grezzi, raccolti su un campione della popolazione che frequenta o abita a Milano. Analizzando i dati relativi al movimento di queste persone con tecniche di *trajectory mining*, si vogliono quindi identificare dei *pattern*, relativi agli spostamenti, ai luoghi frequentati e alle abitudini, tra i cittadini interessati dai due quartieri.

Ad alto livello il prototipo può essere visto come l'applicazione di passi successivi di elaborazione dei dati di traiettoria, in cui l'*output* di un passo rappresenta l'*input* per il

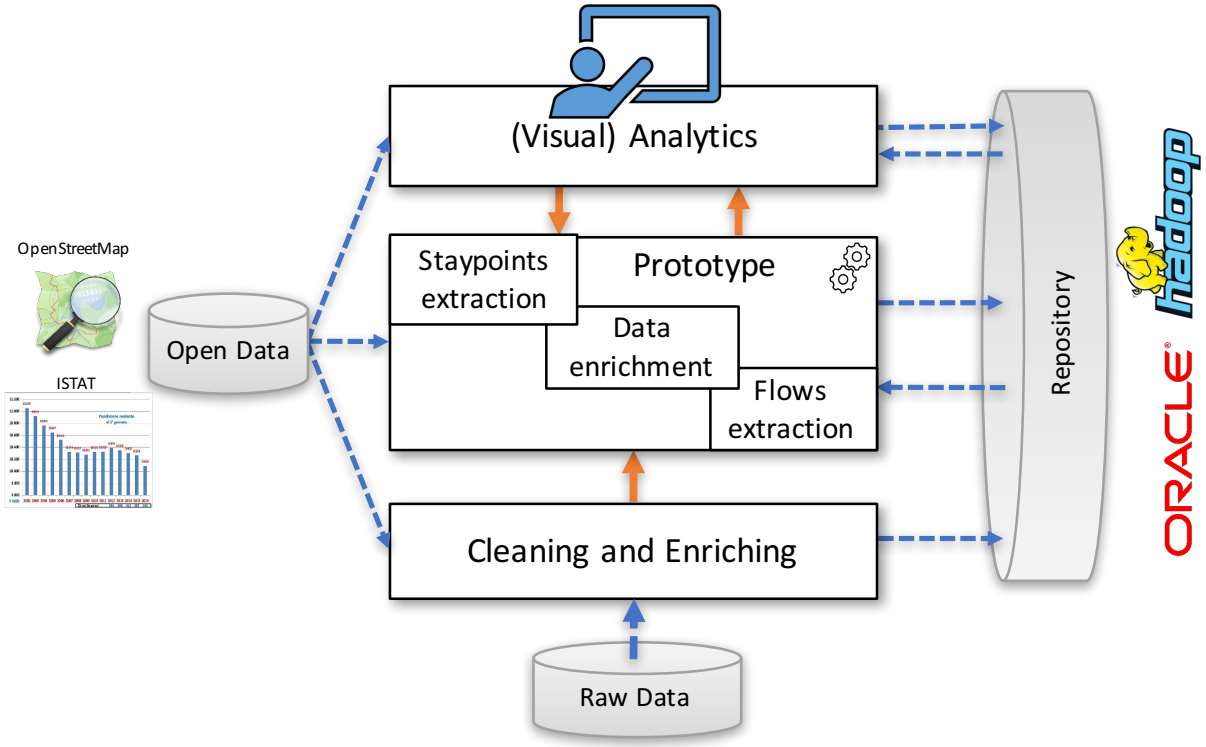


Figura 4.1: Architettura del progetto. Le frecce di colore blu rappresentano flussi fisici di dati, quelle di colore arancione flussi logici tra processi.

passo successivo, realizzando così una sorta di *pipeline* di processi, il cui fine è l'estrazione di informazioni utili ai fini dell'analisi. Questo insieme di processi si inserisce all'interno di un *framework* più ampio, mostrato in figura 4.1, che copre le necessità dell'intero progetto. A monte del prototipo realizzato, abbiamo una base dati che conserva e mette a disposizione i dati grezzi alla fase di pulizia e arricchimento dei dati. I dati processati vengono quindi caricati in un *repository* e messi così a disposizione degli altri componenti del *framework*. A valle rispetto al prototipo è stata implementata un'interfaccia Web per la *visual analytics*, con lo scopo di facilitare l'attività di analisi dati.

4.1.1 Requisiti

A partire dalle richieste degli utenti del progetto, sono stati individuati alcuni requisiti. Il prototipo deve permettere agli utenti di rispondere a una serie di domande riguardanti le persone che frequentano i due quartieri interessati nella ricerca, come ad esempio: in quale

quartiere lavorano gli abitanti di un determinato quartiere? Quanto tempo impiegano per raggiungere il luogo di lavoro? Quali ristoranti frequentano? Ogni quanto vanno al ristorante? A che tipo di attività partecipano? Il prototipo deve essere in grado quindi di inferire dai dati una serie di informazioni che permettano di tracciare un profilo delle persone coinvolte nel programma.

Analizzate le richieste degli utenti, sono stati individuati una serie di requisiti più prettamente funzionali, elencati di seguito.

1. Il prototipo deve essere in grado di determinare in che quartiere una persona abita. Questo requisito è necessario per poter circoscrivere l'analisi solo sulle persone che effettivamente abitano nei due quartieri coinvolti.

2. Il prototipo deve essere in grado di determinare in che quartiere una persona lavora. Questo requisito è necessario per poter capire dove lavorano le persone che abitano nei due quartieri coinvolti.

3. Il prototipo deve essere in grado di determinare quali posti una persona frequenta e con che frequenza. Questo requisito è necessario per poter tracciare un profilo della persone coinvolte e per capire quali attività prediligono o a quali partecipano.

4. Il prototipo deve essere in grado di estrarre i flussi di movimento delle persone attraverso i quartieri, in relazione a dove lavorano, a dove abitano, a che posti frequentano e tenendo conto dei giorni e delle ore in cui si muovono.

Data la quantità di dati da processare, è inoltre necessario che il prototipo gestisca i dati in modo efficiente: il tempo di esecuzione dell'intera elaborazione deve essere contenuto e compatibile con le esigenze di coloro che dovranno analizzarne i risultati.

4.1.2 Analisi

Dall'analisi dei requisiti emerge come il *core* dell'applicativo sia l'analisi della componente spaziale e temporale dei dati. L'analisi della componente spaziale permette infatti di localizzare i luoghi considerati come "interessanti" (ad esempio la casa o il luogo di lavoro)

da una persona e contestualmente di poter registrare i movimenti attraverso i quartieri. L'analisi della componente temporale aiuta a determinare, con un maggiore grado di certezza, la semantica di un luogo considerato come "interessante" e rende possibile l'estrazione dei flussi temporali di movimento attraverso i quartieri. Possiamo definire uno *staypoint* come "un luogo in cui un utente si è fermato per un certo periodo di tempo" [15]. Il fatto che l'utente vi si sia fermato per un tempo più o meno lungo induce a due riflessioni. Innanzitutto possiamo ritenere che quel luogo geografico rivesta una certa importanza per l'utente e che dunque possa essere un suo potenziale "luogo interessante". In secondo luogo, dato che la persona ha stazionato nel luogo, possiamo pensare che avrà generato un elevato numero di *ping* nei dintorni di quel luogo.

Architettura

A livello strutturale possiamo pensare il prototipo come composto da 4 componenti logici, come mostrato in figura 4.2, ognuno avente un ruolo specifico nell'elaborazione dei dati sulle dimensioni spaziale e temporale e nella gestione degli stessi. La separazione delle responsabilità e dei compiti tra più elementi ha ragioni sia ingegneristiche che funzionali. A livello ingegneristico la separazione in componenti indipendenti, aventi ognuno un preciso *task*, garantisce il rispetto del principio di *Separation of Concerns* (*SoC*), promuovendo una generale semplificazione dello sviluppo e della manutenzione del software. A livello funzionale, invece, permette di eseguire al bisogno solo porzioni della *pipeline*, senza dover necessariamente processare da capo l'intero *dataset* nel caso in cui, ad esempio, sia necessario apportare modifiche solo ad uno dei processi terminali. L'interazione tra questi componenti si realizza infatti attraverso la condivisione dei dati elaborati, che vengono prodotti da un componente, memorizzati in modo persistente e letti da quello successivo.

Il componente "Staypoints Extractor" realizza il primo passo di elaborazione e opera quindi direttamente sui dati di traiettoria delle persone. Il suo compito principale è quello di estrarre, per ogni persona, una serie di *staypoints* e di arricchirli sia in senso semantico, sia in senso temporale. Estratti gli *staypoint* per un dato utente, il componente "Staypoints Extractor" fornisce un'interpretazione dello *staypoint*, cioè lo arricchisce semanticamente:

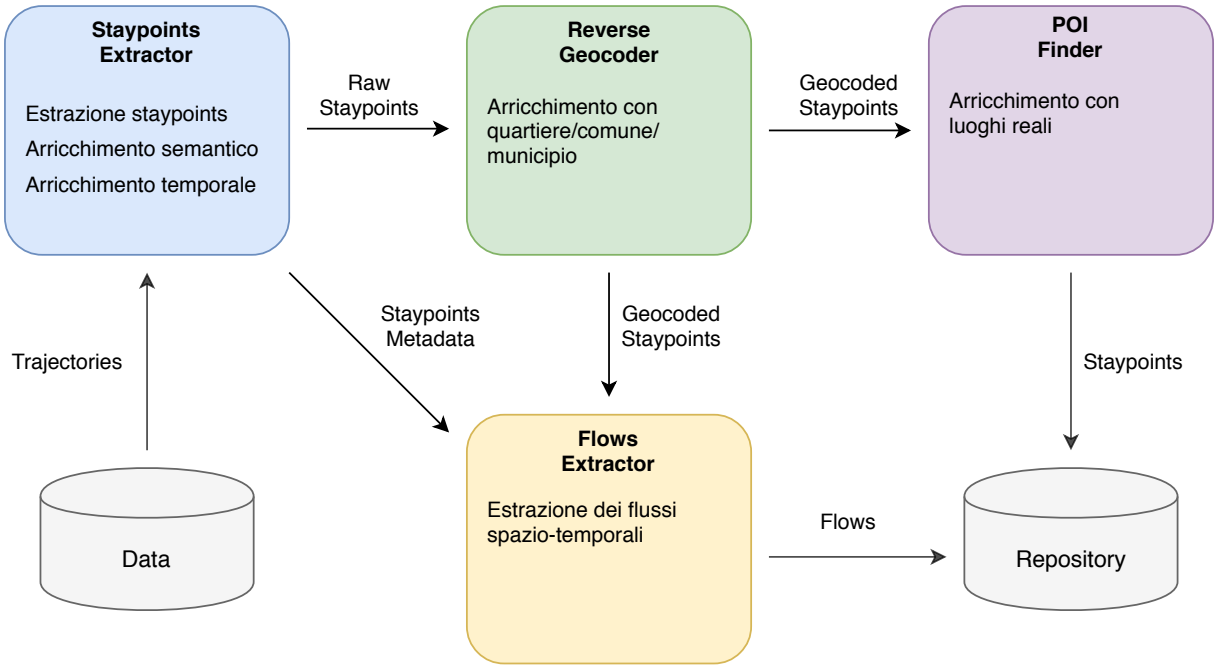


Figura 4.2: Architettura del prototipo.

determina infatti se è più probabile che uno *staypoint* possa essere il luogo di lavoro della persona, la sua abitazione o un generico luogo che frequenta. Inoltre, lavorando sulla dimensione temporale dei dati, arricchisce ogni *staypoint* estratto con informazioni temporali, creando così una relazione tra il luogo geografico e i giorni e le ore in cui la persona ha passato del tempo in quel luogo. Il componente "Staypoints Extractor" produce quindi, al termine della sua elaborazione, due artefatti per ogni persona: una lista di *staypoint* come luoghi geografici arricchiti semanticamente e una serie di metadati che descrivono quando il luogo è stato frequentato. I due artefatti sono stati definiti nell'immagine 4.2 rispettivamente come "Raw Staypoints" e "Staypoints Metadata".

La prima fase di *reverse geocoding* è affidata al componente "Reverse Geocoder", che si occupa di assegnare agli *staypoint* di ogni persona un quartiere, un municipio e un comune, sovrapponendo agli *staypoint* un *layer* di informazioni relative a specifiche entità territoriali e effettuando l'arricchimento a partire dal rapporto di condivisione dello spazio geografico tra le entità territoriali e gli *staypoint*. Il prodotto di questa elaborazione è allora una lista di *staypoint* arricchiti con informazioni sul territorio.

Il componente "POI Finder" utilizza il prodotto dell'elaborazione del componente "Reverse Geocoder" per effettuare un'ulteriore processo di arricchimento e *reverse geocoding*: il suo compito è quello di realizzare un *mapping* tra uno *staypoint* e uno o più *point of interest* (*POI*), cioè luogo specifici (ad esempio un monumento, una chiesa, un ristorante o un parco) che possono essere considerati come interessanti o utili. Si passa così da una nozione geografica di *staypoint* ad una legata invece a luoghi del territorio.

L'ultimo componente coinvolto è definito come "Flows Extractor" e ha il compito di estrarre i flussi di movimento delle persone attraverso i quartieri della città. A partire dagli *staypoint*, arricchiti semanticamente e con le informazioni relative alle entità territoriali, e dai metadati temporali associati agli *staypoint*, estrae i flussi degli spostamenti tra un quartiere e l'altro, determinando ore e giorni della settimana in cui avvengono gli spostamenti e considerando gli spostamenti anche relativamente alla semantica dello *staypoint* (abitazione, luogo di lavoro o luogo frequentato).

I prodotti finali della *pipeline* descritta, cioè l'insieme degli *staypoint* arricchiti e i flussi di movimento, vengono infine memorizzati su un *repository* e resi quindi disponibili agli altri componenti del progetto.

Modellazione dati

I prodotti della elaborazione del prototipo sono stati modellati per formalizzarne la struttura e guidare la successiva fase di implementazione. La scrittura e lettura di dati rappresenta, come detto, la modalità di interazione tra i vari componenti del progetto e dunque la loro modellazione riveste una fase importante del processo di analisi.

In figura 4.3 è mostrato il *data model* degli *staypoint* arricchiti. Ogni *staypoint* è associato ad una singola persona, rappresentata dall'attributo *Id Utente* e caratterizzato da una posizione, espressa come coppia di coordinate geografiche latitudine/longitudine, che permette di collocarlo all'interno dello spazio. L'attributo *Cardinalità* è il numero di *ping* che generano lo *staypoint*. L'attributo *Nome Feature* definisce invece il tipo di *staypoint*, cioè l'abitazione della persona, il luogo di lavoro o un luogo che frequenta

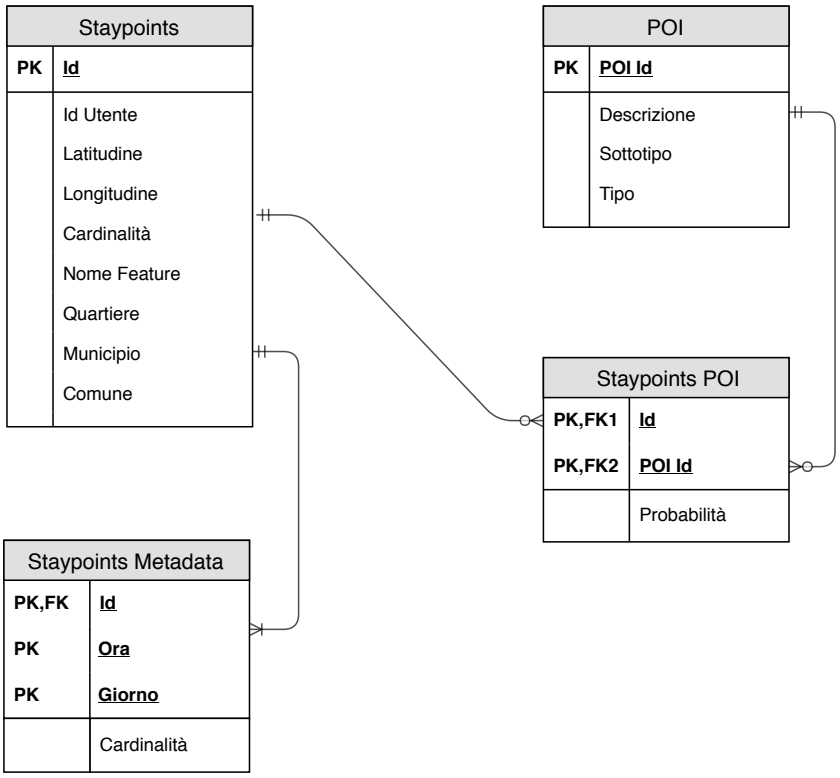


Figura 4.3: *Data model* degli *staypoint*.

abitualmente. Ogni *staypoint* è inoltre associato a un quartiere, un municipio o un comune della città metropolitana di Milano.

L'oggetto *Staypoints Metadata* descrive la componente temporale dello *staypoint*, cioè quando il luogo è stato frequentato. Ogni *staypoint* sarà allora associato a coppie ora/-giorno che indicano il giorno della settimana e l'ora in cui è avvenuta la visita. L'attributo *Cardinalità* è il numero di *ping* che costituiscono lo *staypoint*, data un'ora e un giorno.

Ad ogni *staypoint* possono essere associati anche un insieme di probabili *POI*. Ogni *POI* è caratterizzato da un attributo *Descrizione*, che può essere la descrizione o il nome proprio del luogo, un attributo *Sottotipo*, che indica il tipo di luogo (ristorante, hotel, supermercato ecc.) e un attributo *Tipo*, che descrive la categoria del luogo (luogo pubblico, luogo legato all'attività della ristorazione, negozio ecc.). L'attributo *Probabilità* rappresenta infine la probabilità che lo *staypoint* corrisponda effettivamente a quel dato *POI*.

L'oggetto *Flussi*, mostrato in figura 4.4, modella i movimenti delle persone attraverso i quartieri della città, in relazione agli *staypoint* e al tempo. Gli attributi *Quartiere Da*

Flussi	
PK	Quartiere Da
PK	Nome Feature Da
PK	Giorno Da
PK	Ora Da
PK	Quartiere A
PK	Nome Feature A
PK	Giorno A
PK	Ora A
	Cardinalità

Figura 4.4: *Data model* dei flussi.

e *Quartiere A* rappresentano rispettivamente il quartiere di partenza e di arrivo, *Nome Feature Da* e *Nome Feature A* il tipo di *staypoint* di partenza e di arrivo, *Giorno Da* e *Giorno A* il giorno della settimana di partenza e di arrivo e *Ora Da* e *Ora A* l'ora di partenza e di arrivo. *Cardinalità* è il numero di persone distinte che hanno effettuato uno spostamento tra due *staypoint* rispettivamente di tipo *Nome Feature Da* e *Nome Feature A*, fissati due giorni e due ore, una di partenza e una di arrivo, e due quartieri, uno di partenza e uno di arrivo.

4.1.3 Scelte tecnologiche

Le scelte tecnologiche legate alla realizzazione del prototipo sono state fortemente influenzate dai risultati di precedenti test eseguiti su una piattaforma tradizionale Oracle. I tempi di esecuzione delle elaborazioni su questa piattaforma sono infatti risultati eccessivamente alti: l'intero processo di *processing* poteva impiegare anche molteplici ore per terminare, a causa della notevole mole di dati coinvolta. Questi tempi di esecuzione non sono ovviamente compatibili con le esigenze dei clienti del progetto. Data la grande quantità di dati da elaborare si è pensato quindi di realizzare il prototipo su una piattaforma di calcolo *Big Data*, in quanto adatta per definizione a gestire operazioni su grandi moli di dati. Pertanto, la volontà di contenere i tempi di esecuzione ha giocato un ruolo fondamentale

in questa scelta.

Nello specifico, si è deciso di sviluppare il prototipo utilizzando la piattaforma Apache Spark: l'ecosistema di Spark offre infatti una libreria avanzata e performante per la gestione e il *processing* dei dati spaziali, GeoSpark. Come visto nella sezione 2.4, GeoSpark ha infatti prestazioni migliori rispetto a librerie simili basate su altre piattaforme. La scelta di Apache Spark e GeoSpark è quindi sembrata la più naturale considerati i requisiti di progetto e la tipologia di dati da elaborare. A fianco di Spark e GeoSpark si è inoltre deciso di utilizzare Apache Hive, per permettere una memorizzazione strutturata dei dati elaborati e per facilitarne il conseguente trasferimento sul *repository* condiviso.

4.1.4 Implementazione

Di seguito si riportano le fasi salienti del processo di implementazione del prototipo.

Algoritmo di estrazione degli *staypoint*

La fase di estrazione degli *staypoint* è uno dei momenti cruciali dell'intera elaborazione, in quanto influenza in modo determinante la qualità dei risultati delle fasi successive. Come detto in precedenza, uno *staypoint* rappresenta un luogo geografico in cui una persona si è fermata in modo ricorrente o per un determinato periodo di tempo: l'idea di fondo per identificare gli *staypoint* è quella allora di considerare la distribuzione dei punti che compongono le traiettorie nello spazio. Se una persona si è fermata in un determinato luogo per un certo periodo di tempo o con frequenza avrà generato un elevato numero di dati spaziali (cioè *ping*) nelle vicinanze di quel luogo. Dunque, i luoghi geografici corrispondenti ai centroidi delle posizioni spaziali a più alta densità possono essere considerati come gli *staypoint* di una data persona, come esemplificato in figura 4.5. Ispirandosi al lavoro di Zhou et al., descritto nella sezione 3.2.1, si è implementato l'algoritmo di estrazione degli *staypoint* come applicazione di due passi di elaborazione successivi sui dati di traiettoria di ciascuna persona, uno di *preprocessing* e uno di *clustering*.

La fase di *preprocessing* consiste nel filtrare i dati disponibili per velocità , accuratezza e quantità per persona. In questa fase vengono eliminate tutte le posizioni la cui velocità

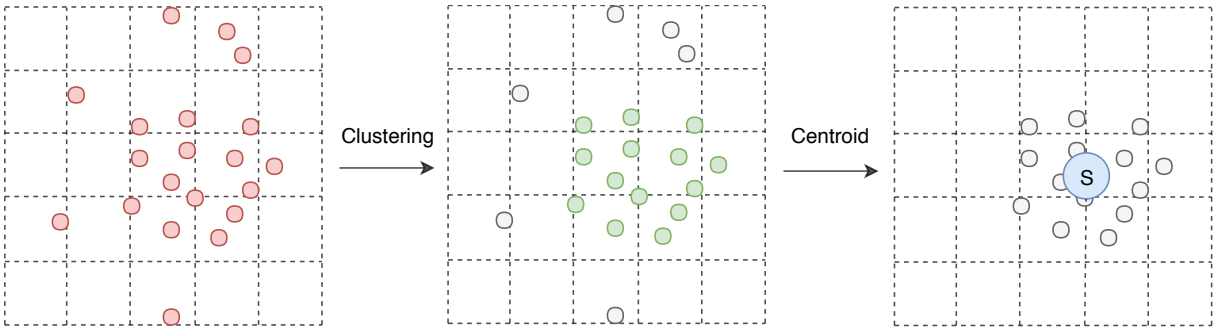


Figura 4.5: Esempio di estrazione di uno *staypoint* (nella figura rappresentato dalla lettera S) a partire da un gruppo di posizioni.

è superiore ad una certa soglia, fissata a 2 km/h: questo permette di rimuovere tutti quei dati poco significativi relativi, ad esempio, a spostamenti in automobile. Si ottiene così un *dataset* caratterizzato da sole posizioni statiche, relative ad assenza di moto o a movimenti effettuati a piedi. I dati vengono inoltre filtrati per accuratezza, eliminando tutte le posizioni con valore di accuratezza superiore a 30 metri, per ridurne il rumore. Infine vengono rimossi tutte le persone, e le relative posizioni, con un numero di *ping* associato inferiore a 100. Il risultato di questa fase di *preprocessing* è quindi un insieme di dati di traiettoria come posizioni statiche ad elevata accuratezza.

La fase successiva è quella di *clustering*: applicando un algoritmo di *clustering density-based* sui dati spaziali pre-processati di ciascuna persona, si ottengono i gruppi distinti di posizioni ad elevata densità, cioè *cluster*, i cui centroidi corrispondono agli *staypoint* della persona. L'implementazione dell'algoritmo di *clustering density-based* scelta per questa fase è PatchWork, descritto nella sezione 3.3, per via della sua complessità lineare in grado di assicurare elevate *performance*. Inizialmente si è pensato di lanciare sequenzialmente istanze dell'algoritmo sugli insiemi di dati spaziali di ciascuna persona, per sfruttare l'ambiente distribuito sottostante. A livello tecnico ciò significa creare un RDD di dati spaziali per ogni utente e lanciare in sequenza un'istanza dell'algoritmo su ognuno di questi RDD; ogni istanza in questo modo viene eseguita in modo distribuito su più macchine del *cluster*. Le caratteristiche dei dati hanno però reso impraticabile questa soluzione: essendo infatti la cardinalità delle posizioni spaziali di ciascuna persona relativamente bassa (in media ad ogni persona sono associati circa 2 000 *ping*) e il numero delle persone distinte

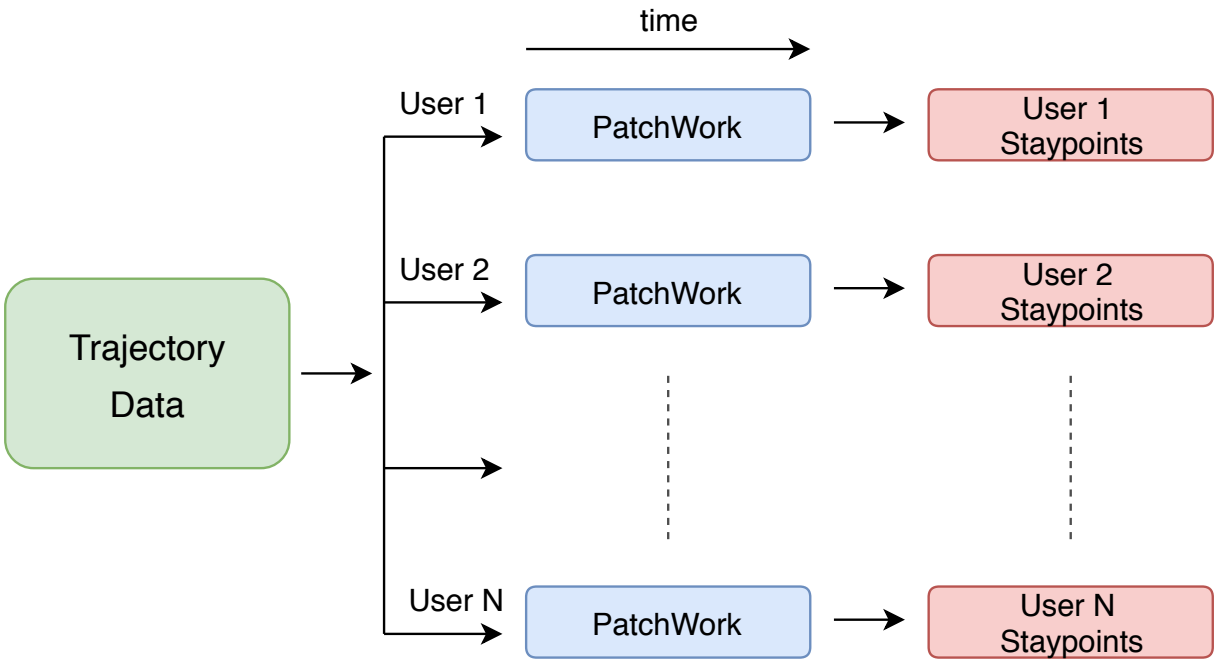


Figura 4.6: Esecuzione della fase di estrazione degli *staypoint* come applicazione *bag-of-tasks*.

elevato (circa 415 000 persone distinte), l'*overhead* generato dall'utilizzo di un RDD distinto per ogni persona supera di gran lunga i vantaggi derivanti dalla parallelizzazione del processo di *clustering* sui dati di ogni persona, rendendo l'esecuzione estremamente lenta e inefficiente. È stato quindi necessario adottare un approccio differente, organizzando l'esecuzione delle istanze dell'algoritmo di *clustering* come applicazione *bag-of-tasks*; l'esecuzione è esemplificata in figura 4.6. Piuttosto che lanciare una singola istanza alla volta, vengono lanciate parallelamente più istanze di PatchWork: ogni istanza diventa quindi un *task*, eseguito da uno degli esecutori appartenenti al *cluster*. Questo approccio è possibile in quanto le istanze sono indipendenti tra loro, nel senso che operano su dati spaziali distinti di persone distinte. La strategia appena descritta permette al processo di estrazione degli *staypoint* di essere molto più efficiente ma ha reso necessaria la modifica di una piccola porzione del codice di PatchWork, per adattarlo alla nuova modalità di esecuzione. Questa modifica non ha comunque modificato la complessità computazionale dell'algoritmo.

Le istanze di PatchWork sono eseguite configurando il parametro $MinPoints$ a 20

e impostando celle bidimensionali quadrate con lato di 55 metri, attraverso la configurazione del parametro $CellSize(\epsilon)$. I parametri $Ratio$ e $MinCell$ sono invece impostati rispettivamente a 0 e 1, in quanto non è risultato necessario considerare, in questa specifica applicazione dell'algoritmo, né la differenza di densità tra celle contigue né un'estensione spaziale minima del *cluster* superiore a quanto già definito dal parametro $CellSize(\epsilon)$. I parametri così impostati garantiscono che ogni *staypoint* estratto contenga almeno 20 rilevazioni, anche se non contigue nel tempo, e quindi che il luogo corrispondente sia stato perlomeno visitato almeno 20 volte da una persona.

Algoritmo di arricchimento semantico degli *staypoint*

L'operazione di arricchimento semantico permette di caratterizzare gli *staypoint*: l'algoritmo determina infatti se è più probabile che uno *staypoint* possa essere il luogo di lavoro della persona, la sua abitazione o un generico luogo che frequenta. L'algoritmo utilizza le informazioni temporali relative alle visite della persona allo *staypoint* per effettuare l'arricchimento semantico: l'idea di fondo è che se un luogo è frequentato soprattutto durante la fascia oraria notturna (dalle 2 alle 6 di notte) è probabile che rappresenti l'abitazione della persona, mentre se è frequentato durante la fascia oraria lavorativa (dalle 9 alle 17) è probabile che corrisponda al luogo di lavoro della persona.

L'algoritmo prende come *input* gli *staypoint* associati ad una persona e le informazioni temporali relative agli orari di visita: per ogni *staypoint* viene calcolata la cardinalità dei *ping* (come numero dei singoli *ping* che compongono il *cluster* da cui è stato estratto) in ognuna delle due fasce orarie, notturna e lavorativa. Lo *staypoint* con cardinalità massima di *ping* nella fascia notturna viene etichettato come luogo dove la persona abita, mentre lo *staypoint* con cardinalità massima nella fascia lavorativa, se non già etichettato come luogo dove la persona abita, viene etichettato come luogo dove la persona lavora. Dunque a ciascuna persona sono associati al più un'abitazione e al più un luogo di lavoro. I restanti *staypoint* sono infine etichettati come luoghi che la persona frequenta.

Arricchimento con quartiere/municipio/comune

L'arricchimento degli *staypoint* con informazioni relative al quartiere, municipio o comune a cui appartiene avviene attraverso un'operazione di *reverse geocoding*: il *reverse geocoding* rappresenta infatti l'attività di associazione tra una posizione spaziale, espressa come coppia latitudine/longitudine, e un luogo, una via o un livello di suddivisione territoriale. Le informazioni relative alle suddivisioni territoriali della città metropolitana di Milano sono state ottenute attraverso *open data* forniti dal Comune di Milano e dalla Regione Lombardia, sotto forma di mappe in formato Shapefile ESRI.

Il *reverse geocoding* sugli *staypoint* viene eseguito avvalendosi dei servizi forniti dalla libreria GeoSpark. Per ogni mappa, dei quartieri, dei municipi e dei comuni, viene effettuata un'operazione di *spatial join* con gli *staypoint*: l'operazione di *spatial join* permette infatti di trasferire attributi da un *layer* ad un altro relativamente alle reciproche relazioni spaziali. Nello specifico, la collezione di *staypoint* viene messa in relazione con le mappe attraverso un'operazione di *join* la cui la clausola ha valore `true` solo se la posizione associata allo *staypoint* si trova completamente all'interno del poligono che rappresenta i confini del quartiere, del municipio o del comune.

Associazione tra *staypoint* e *POI*

Per realizzare l'associazione tra uno *staypoint* e uno o più *POI* si utilizza un *layer* di dati di OpenStreetMap, distribuito in formato Shapefile ESRI, che descrive il territorio italiano attraverso una serie di *feature*, sia relative a elementi naturali (come laghi, fiumi ecc.), sia relative ad artefatti umani (come strade, monumenti ecc.). La porzione di *layer* relativa ai *POI* suddivide i luoghi di interesse in 9 categorie, come riportato nella tabella 4.1.

L'operazione di associazione tra uno *staypoint* e uno o più *POI* è realizzata attraverso l'uso della libreria GeoSpark. Preso uno *staypoint*, le sue coordinate (espresse come coordinate WGS84) sono proiettate sul sistema di coordinate EPSG:32632, che utilizza il metro come unità di misura. La stessa operazione viene eseguita per le coordinate di ogni *POI*. A questo punto vengono cercati tutti i *POI* che si trovano all'interno di un raggio

di 100 metri a partire dallo *staypoint*, cioè i suoi vicini: ad uno *staypoint* possono quindi essere teoricamente associati più *POI* differenti. Ad ogni istanza di *POI* associata ad uno *staypoint* viene infine assegnata una misura di probabilità. Dato l'evento A che uno *staypoint* S corrisponda a un *POI* $P \in N(S)$, la probabilità $Pr(A)$ è calcolata come segue:

$$Pr(A) = \begin{cases} 1 & \text{se } |N(S)| = 1 \\ 1 - \left(\frac{dist(P,S)^2}{\sum_{p \in N(S)} dist(p,S)^2} \right) & \text{se } |N(S)| > 1 \end{cases} \tag{4.1}$$

dove $N(S)$ è l'insieme dei *POI* vicini a S.

Categoria	Descrizione	Esempi di *POI*
Public	Comprende i luoghi pubblici.	Università, scuole, uffici postale.
Health	Comprende luoghi legati alla cura della salute.	Ospedale, farmacie, ambulatori.
Leisure	Comprende luoghi dove tipicamente si spende il tempo libero.	Cinema, teatri, parchi.
Catering	Comprende luoghi legati all'attività di ristorazione.	Ristoranti, pub, bar.
Accomodation	Comprende alloggi.	Hotel, b&b, campeggi.
Shopping	Comprende luoghi dove effettuare acquisti.	Supermercati, librerie, negozi.
Money	Comprende luoghi dove si gestisce denaro.	Banche, ATM.
Tourism	Comprende luoghi turistici e destinati ai turisti.	Monumenti, musei, uffici turistici.
MiscPoi	Comprende luoghi non associabili alle altre categorie.	Bagni, panchine, idranti.

Tabella 4.1: Categorie di *POI* del *layer* dati di OpenStreetMap.

4.2 I dati

I dati a disposizione sono organizzati in un unico *dataset* e rappresentano traiettorie di persone reali come insieme di posizioni spaziali ordinate cronologicamente. Ogni posizione può essere vista come un *ping*, generato dal sensore della persona ad una certa frequenza. Il *dataset* di partenza è costituito da 2 367 846 589 *ping* grezzi relativi a 575 356 persone distinte.

4.2.1 *Preprocessing*

I dati appartenenti al *dataset* di partenza hanno subìto due fasi successive di *processing* prima di essere resi disponibili alle operazioni di elaborazione. La prima fase consiste in cinque operazioni successive di *processing* dati, collocate all'interno della fase di "Cleaning and Enriching" visibile in figura 4.1. Le operazioni sono riportate di seguito.

- *Filtering* degli utenti con un numero di *ping* associati > 100.
- Campionamento dei *ping* con una frequenza di 1 *ping* al minuto, per rendere omogenee le rilevazioni.
- Rimozione dei *ping* con valore di accuratezza > 1000 metri.
- Rimozione dei *ping* registrati al di fuori dell'intervallo temporale da settembre 2017 a dicembre 2017.
- Rimozione di valori nulli.

L'applicazione di questa prima fase di *preprocessing* ha ridotto sia la cardinalità, sia il numero di persone distinte del *dataset* di partenza: il *dataset* processato è costituito infatti da 1 286 592 115 *ping* relativi a 472 221 persone distinte. I *ping* non sono quantitativamente distribuiti in modo uniforme tra le persone: come visibile nel grafico 4.7, la maggior parte delle persone ha un ridotto numero di *ping* associati. Il prototipo utilizza per le proprie elaborazioni i dati di traiettoria provenienti da questo *dataset*.

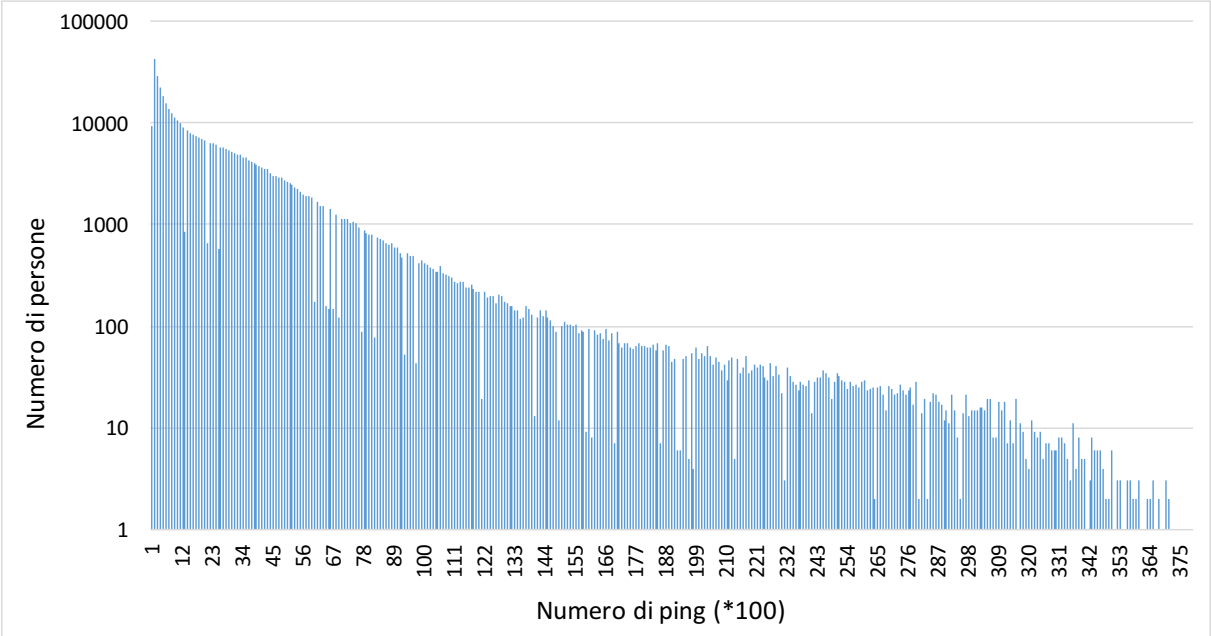


Figura 4.7: Distribuzione delle persone rispetto al numero di *ping*.

La seconda fase di *preprocessing* dei dati, come descritto nella sezione 4.1.4, precede l'attività di estrazione degli *staypoint*. I dati subiscono in questa fase tre operazioni di *filtering*, riportate di seguito.

- Rimozione dei *ping* con valore di accuratezza > 30 metri.
- Rimozione dei *ping* con velocità > 2 km/h.
- Rimozione degli utenti con un numero di *ping* statici associati < 100.

Questo nuovo passo di *filtering* riduce quindi ulteriormente la cardinalità del *dataset*: i *ping* disponibili, che possiamo definire "statici" a seguito del *filtering* per velocità, sono 865 512 283, suddivisi su 414 626 persone distinte. Le fasi di estrazione degli *staypoint* e tutte le successive operano su dati provenienti da quest'ultimo *dataset*.

Fase 1	- Rimozione degli utenti con un numero di *ping* associati < 100. - Campionamento dei *ping* con una frequenza di 1 *ping* al minuto. - Rimozione dei *ping* con valore di accuratezza > 1000 metri. - Rimozione dei *ping* registrati fuori dal periodo settembre-dicembre 2017. - Rimozione di valori nulli.
Fase 2	- Rimozione dei *ping* con accuratezza > 30 metri. - Rimozione dei *ping* con velocità superiore a 2 km/h. - Rimozione degli utenti con un numero di *ping* statici associati < 100.

Tabella 4.2: Attività delle due fasi di *preprocessing* del *dataset*.

Tipo	*Ping*	Persone distinte
Ping grezzi	2 367 846 589	575 356
Ping campionati e filtrati (Fase 1)	1 286 592 115	472 221
Ping statici (Fase 2)	865 512 283	414 626

Tabella 4.3: Statistiche relative ai *ping* e alle persone.

4.2.2 Struttura

Tutti i dati di traiettoria utilizzati dal prototipo sono memorizzati nel *cluster* in un'unica tabella Hive: ogni tupla della tabella rappresenta una singola posizione (quindi un singolo *ping*) associata ad una persona. Ogni posizione è descritta dagli attributi riportati qui di seguito.

- *Customid*: identificativo univoco della persona a cui è associata la posizione.
- *Latitude*: latitudine della posizione, espressa secondo il sistema di coordinate WGS84.
- *Longitude*: longitudine della posizione, espressa secondo il sistema di coordinate WGS84.
- *Timest*: marca temporale della rilevazione. È espressa in secondi, secondo il sistema *Unix time*.
- *Speed*: velocità al momento della rilevazione, espressa in metri al secondo.
- *Accuracy*: accuratezza della rilevazione, espressa in metri.

4.3 Risultati

In questa sezione si analizzano i risultati dell'esecuzione del prototipo, tenendo conto sia della conformità dei risultati rispetto ai requisiti, sia dell'efficienza del processo di elaborazione. Il prototipo è stato testato su un *cluster* di 8 macchine del Business Intelligence Group, assegnando a ciascun *executor* 20 GB di memoria e 6 *core*. Come distribuzione di Apache Spark è stata utilizzata una distribuzione Cloudera, versione 2.1.0. Come *resource manager* si è utilizzato YARN.

4.3.1 Risultati dell'elaborazione

L'elaborazione dati del prototipo produce due artefatti principali, un insieme di *staypoint* arricchiti per ciascuna persona e un insieme di flussi di movimento attraverso i quartieri.

Il prototipo estrae totalmente 532 769 *staypoint* a partire dai dati di traiettoria preprocessati, dunque a partire da 865 512 283 *ping* statici suddivisi tra circa 414 626 persone distinte. Questi *staypoint* sono distribuiti tra circa 243 777 persone distinte, quindi circa il 60% delle singole persone viene associato ad almeno uno *staypoint*. Il fatto che il restante 40% delle persone non sia associato ad alcuno *staypoint* è dovuto alla distribuzione spaziale delle loro rilevazioni, non organizzate in gruppi ad elevata densità e dunque considerate come rumore dall'algoritmo di *clustering*, o al numero limitato di posizioni associate, che rende meno probabile la formazione di *cluster* di posizioni. Come mostrato nel grafico 4.8, la quantità di persone per *staypoint* non si distribuisce in modo uniforme: questo fenomeno è conseguenza della distribuzione dei *ping* tra le persone.

I circa 532 000 *staypoint* sono, come visibile in figura 4.9, così suddivisi: 204 000 sono stati interpretati come abitazione di una persona, 97 000 come luogo di lavoro e 231 000 come luogo che una persona frequenta. Dato che ad ogni persona è associato al più uno *staypoint* di tipo "abitazione" e al più uno di tipo "luogo di lavoro", l'algoritmo è stato in grado di associare un'abitazione a più dell'80% delle singole persone con almeno uno *staypoint* e un luogo di lavoro a circa il 40% di queste. È quindi emerso che le persone distinte che abitano a Milano sono 147 000, quelle che lavorano a Milano 38 000 e che

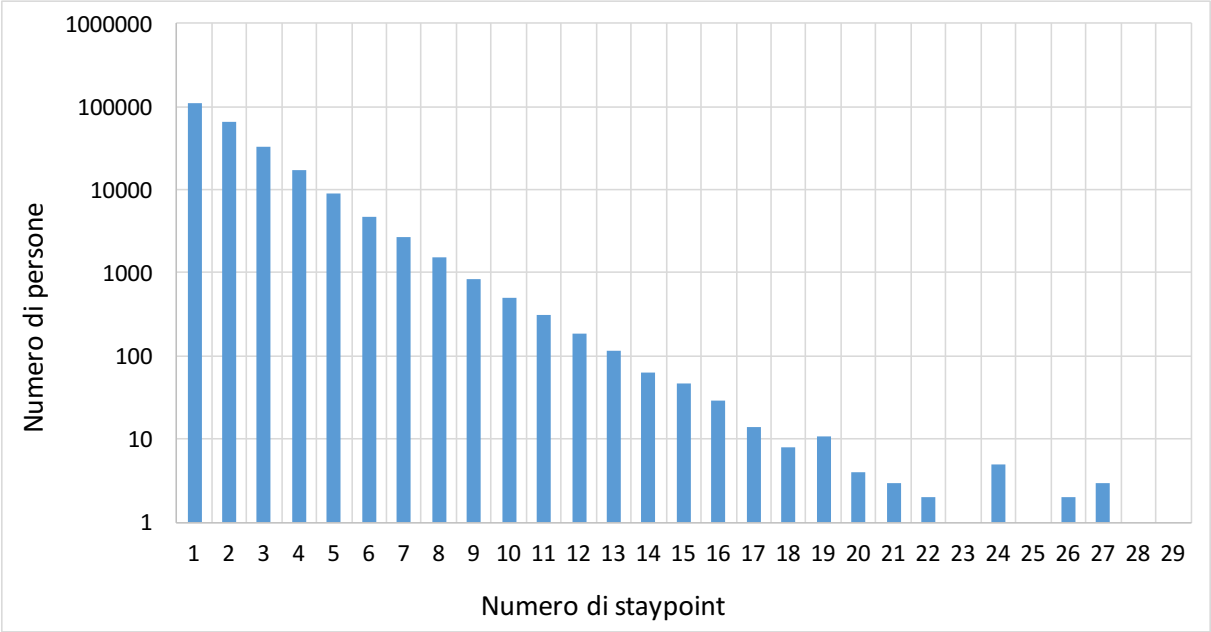


Figura 4.8: Distribuzione delle persone sugli *staypoint*.

abitano e lavorano a Milano 58 000 persone.

Gli *staypoint* sono arricchiti semanticamente con un *layer* OpenStreetMap, come descritto in sezione 4.1.4, prendendo in considerazione però solo quattro tipi di *POI* (ristoranti, pub, bar e teatri), per un totale di 16 000 punti di interesse disponibili: si sono ottenute 230 000 associazioni tra *staypoint* e *POI*.

I flussi generati a partire dall'analisi temporale degli *staypoint* sono invece circa 151 milioni: questo perché ogni flusso è identificato dal quartiere di partenza e quello di arrivo, dalla rispettiva ora e giorno di visita e dal tipo di *staypoint* considerato.

Non essendo disponibile una *ground truth* relativa alle persone coinvolte nel progetto non è stato possibile verificare in modo oggettivo l'accuratezza dei risultati. I risultati dell'elaborazione sono comunque compatibili con quanto espresso durante la fase di analisi dei requisiti e permettono di rispondere alle domande formulate come requisiti utente nella sezione 4.1.1. A partire dai dati di traiettoria di una persona si arriva, in un buon numero di casi, a determinarne abitazione e luogo di lavoro, nonché tipologia di luoghi visitati e frequenza di visita. Si ottiene inoltre un insieme di flussi di movimento, utili per tracciare gli spostamenti delle persone attraverso la città e per determinarne profilo e abitudini.

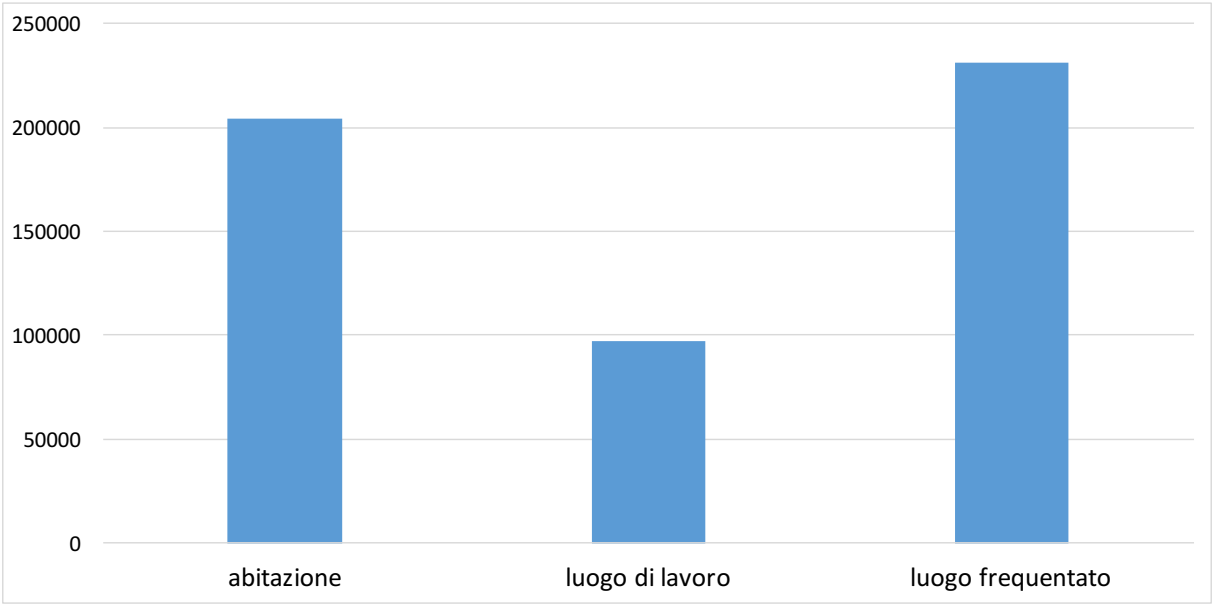


Figura 4.9: Numero di *staypoint* in relazione all'interpretazione fornita dall'algoritmo di arricchimento semantico.

4.3.2 Tempi di elaborazione

L'intera elaborazione impiega circa 43 minuti per terminare. I tempi parziali di ogni fase sono riportati nella tabella 4.4. L'operazione di *clustering* delle posizioni impiega circa 90 secondi dei 6 minuti totali della fase di estrazione e arricchimento semantico/temporale, a conferma della bontà dell'approccio e dell'algoritmo scelto. La fase di arricchimento con quartiere, municipio e comune è quella che incide maggiormente sul tempo totale di esecuzione, in quanto esegue tre *spatial join* (uno per ciascuna mappa) tra la collezione di *staypoint* e le mappe.

Fase	Minuti impiegati
Estrazione, arricchimento semantico/temporale	6
Arricchimento con quartiere/municipio/comune	30
Arricchimento con *POI*	2
Estrazioni flussi	5

Tabella 4.4: Tempi di esecuzione delle fasi di elaborazione.

I tempi di esecuzione dell'intera elaborazione, se confrontati con quelli di test effettuati su piattaforme tradizionali, sono comunque estremamente contenuti e risultano in linea

con quanto stabilito in fase di analisi dei requisiti.

Capitolo 5

Conclusioni

In questa tesi si è presentato un prototipo per l'elaborazione e l'analisi di dati di traiettoria su piattaforma *Big Data*, sviluppato nell'ambito di un più ampio progetto di estrazione di *pattern* di comportamento tra i cittadini della città di Milano. Il prototipo è in grado di inferire dai dati di traiettoria una serie di informazioni, attraverso l'applicazione di processi di arricchimento dati e l'utilizzo di tecniche di *trajectory data mining*. In particolare, il prototipo si dimostra capace di estrarre a partire dai dati di traiettoria di una persona la posizione dell'abitazione, il luogo di lavoro e i luoghi che questa frequenta abitualmente.

Il contributo principale di questa tesi riguarda l'algoritmica per l'attività di *clustering* dei dati spazio-temporali: l'algoritmo di *clustering grid* e *density-based* utilizzato è risultato infatti più efficiente rispetto ai tradizionali algoritmi *density-based* ma egualmente efficace. Inoltre, il prototipo dimostra la validità delle tecnologie *Big Data* applicate a grandi volumi di dati spaziali. I tempi di elaborazione sono infatti estremamente contenuti se confrontati con quelli ottenuti da piattaforme tradizionali e la disponibilità di librerie specifiche per la gestione di dati spaziali contribuisce a semplificare notevolmente il processo di sviluppo.

Il prototipo, naturalmente, può essere ulteriormente esteso e lascia spazio a miglioramenti e sviluppi futuri. Una possibile estensione potrebbe riguardare l'implementazione di algoritmi per l'analisi di traiettorie, al fine di ottenere *pattern* più precisi relativi ai movimenti delle persone attraverso la città. Inoltre, si potrebbe verificare la correttezza dei

risultati del prototipo attraverso il confronto con il campione statistico di un questionario, al momento non disponibile, realizzato come attività collaterale al progetto.

Bibliografia

[1] *Big Data*. https://en.wikipedia.org/wiki/Big_data. (Visitato il 01/06/2018).

[2] *Big data architectures*. https://docs.microsoft.com/en-us/azure/architecture/data-guide/big-data. (Visitato il 01/06/2018).

[3] Derya Birant e Alp Kut. «ST-DBSCAN: An algorithm for clustering spatial-temporal data». In: *Data Knowl. Eng.* 60.1 (2007), pp. 208–221.

[4] Martin Ester et al. «A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise». In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, Oregon, USA*. 1996, pp. 226–231.

[5] Fosca Giannotti et al. «Trajectory pattern mining». In: *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Jose, California, USA, August 12-15, 2007*. 2007, pp. 330–339.

[6] Frank Gouineau, Tom Landry e Thomas Triplet. «PatchWork, a scalable density-grid clustering algorithm». In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*. 2016, pp. 824–831.

[7] *Hadoop*. https://en.wikipedia.org/wiki/Apache_Hadoop. (Visitato il 09/06/2018).

[8] Zhou Huang et al. «GeoSpark SQL: An Effective Framework Enabling Spatial Queries on Spark». In: *ISPRS Int. J. Geo-Information* 6.9 (2017), p. 285.

[9] Slava Kisilevich et al. «Spatio-temporal clustering». In: *Data Mining and Knowledge Discovery Handbook, 2nd ed.* 2010, pp. 855–874.

[10] Jean Damascène Mazimpaka e Sabine Timpf. «Trajectory data mining: A review of methods and applications». In: *J. Spatial Information Science* 13.1 (2016), pp. 61–99.

[11] Xiangrui Meng et al. «MLlib: Machine Learning in Apache Spark». In: *Journal of Machine Learning Research* 17 (2016), 34:1–34:7.

[12] Shashi Shekhar et al. «Spatial big-data challenges intersecting mobility and cloud computing». In: *Proceedings of the Eleventh ACM International Workshop on Data Engineering for Wireless and Mobile Access, MobiDE 2012, Scottsdale, AZ, USA, May 20, 2012*. 2012, pp. 1–6.

[13] *Spark architecture*. https://spark.apache.org/docs/latest/cluster-overview.html. (Visitato il 23/06/2018).

[14] *Spark-DBSCAN*. https://github.com/alitouka/spark_dbscan. (Visitato il 15/06/2018).

[15] Georgios Stylianou. «Stay-point Identification as Curve Extrema». In: *CoRR* abs/1701.06276 (2017).

[16] Jia Yu, Jinxuan Wu e Mohamed Sarwat. «GeoSpark: a cluster computing framework for processing large-scale spatial data». In: *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems, Bellevue, WA, USA, November 3-6, 2015*. 2015, 70:1–70:4.

[17] Matei Zaharia et al. «Spark: Cluster Computing with Working Sets». In: *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*. 2010.

[18] Yu Zheng. «Trajectory Data Mining: An Overview». In: *ACM TIST* 6.3 (2015), 29:1–29:41.

[19] Yu Zheng e Xiaofang Zhou, cur. *Computing with Spatial Trajectories*. Springer, 2011.

[20] Changqing Zhou et al. «Discovering personal gazetteers: an interactive clustering approach». In: *12th ACM International Workshop on Geographic Information Systems, ACM-GIS 2004, November 12-13, 2004, Washington, DC, USA, Proceedings.* 2004, pp. 266–273.