UNIVERSITÀ DELLA CALABRIA
DIPARTIMENTO DI
INGEGNERIA INFORMATICA,
MODELLISTICA, ELETTRONICA
E SISTEMISTICA
DIMES

Corso di Laurea Magistrale in Ingegneria Informatica

Progetto Esame:

Sistemi Distribuiti e Cloud Computing

Candidato:

Ivan Scuderi 216635

Relatori:

Prof. Domenico Talia

Prof. Loris Belcastro

Anno Accademico 2020-2021

Indice

Capitolo 1: Introduzione

1.1: Descrizione del Progetto e Scopo

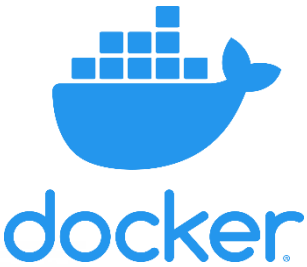
Il progetto da me svolto ha come obiettivo lo sviluppo di un applicativo web al fine di effettuare query e analisi su un *dataset* contenete elementi in formato *json*, che fanno riferimento a post geotaggati estratti dalla piattaforma "***Flickr***". Tale piattaforma permette agli utenti iscritti di condividere foto e video personali sul proprio profilo, come contenuti privati o pubblici, per rimanere in contatto con i propri amici e conoscenti, con una forma di comunicazione simile ad un blog. Scelta progettuale è stata quella di adottare un'analisi di tipo '*black box*' , ovvero sviluppare inizialmente delle interrogazioni sui dati che mi permettessero di inquadrare il contesto specifico di riferimento, per poi allargare lo scopo e cercare di estrarre informazioni più o meno rilevanti dal punto di vista pratico. La trattazione continuerà andando a descrivere le tecnologie impiegate, per poi passare all'analisi dell'architettura, delle scelte progettuali e delle varie componenti del sistema.

1.2: Tecnologie Impiegate

Per quanto riguarda le tecnologie impiegate per lo sviluppo di tale progetto, ho "costruito" un cluster Spark sulla mia macchina locale utilizzando una tecnologia di containerizzazione chiamata ***Docker*** (di cui parlerò in maniera più dettagliata nella sezione relativa alla descrizione dell'architettura del sistema), sui nodi da me implementati ho utilizzato il framework open source ***Apache Spark*** per il lato '*backend*' del sistema, impiegando in particolare la sua versione per il linguaggio ***Python*** ovvero ***PySpark***. Tramite l'utilizzo del modulo ***Spark SQL*** è stata possibile la scrittura delle query, mentre i dati sono stati caricati negli oggetti ***Dataframe Spark***. Centrale nello sviluppo è stato l'impiego del framework ***Apache Livy***, questo è un servizio che facilita le interazioni su un cluster Spark andando ad implementare un'interfaccia server *REST* per la sottomissione di richieste e *query*, anche in questo caso si è scelta

l'installazione di tale servizio su di un *container* in maniera tale da rendere il sistema il più modulare e distribuito possibile, mentre per l'implementazione del client ho fatto uso della libreria ***pylivy***.

Per quanto riguarda ulteriori tecnologie impiegate si nominano: le librerie ***pandas***, ***seaborn*** e ***matplotlib*** impiegate per effettuare varie operazioni accessorie sul *dataset* e visualizzare in maniera grafica i risultati delle *query*. La libreria ***gmplot***, grazie alla quale ho potuto usufruire dei servizi della *Google Cloud Platform* relativi alla geolocalizzazione tramite piattaforma *Google Maps*, utilizzati per visualizzare i risultati delle *query* che impiegavano informazioni relative a latitudine e longitudine. Infine, per quanto riguarda il '*frontend*', ho utilizzato il *framework* ***Django*** scritto in *Python* per lo sviluppo della piattaforma web impiegando, per la parte grafica e di creazione dei *template*, il *framework* ***Bootstrap***, che consiste in una raccolta di strumenti unificata contenente una serie di modelli di progettazione *HTML*, *CSS* e *JavaScript* per le varie componenti dell'interfaccia grafica vera e propria del sistema.

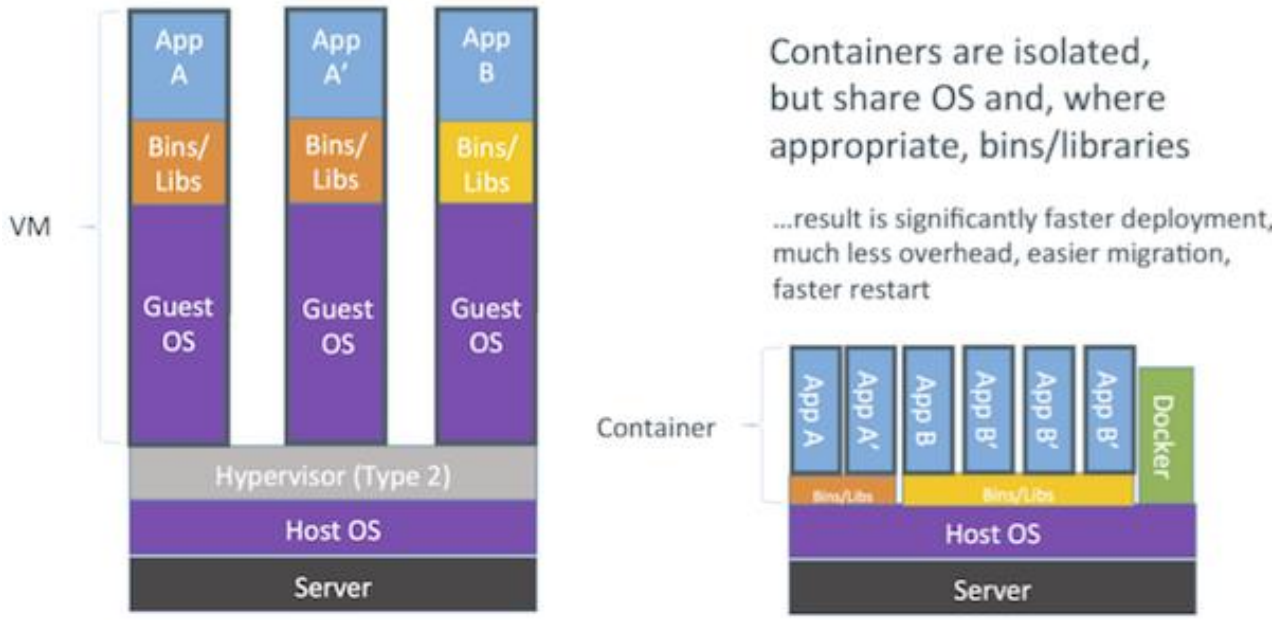


1.3: Docker ed Architettura

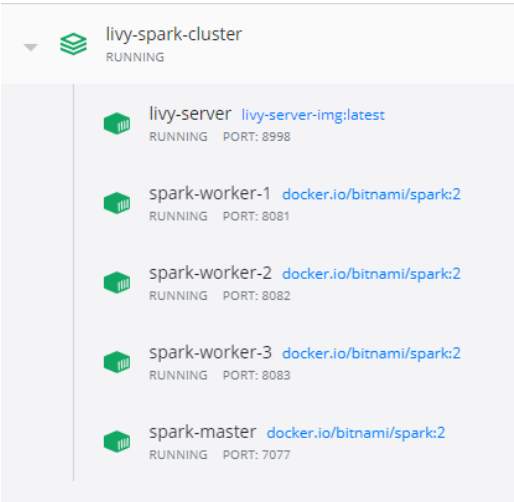
Come prima cosa si procede alla descrizione della tecnologia *Docker* e a cosa si fa riferimento quando si parla di *container*, andando anche a descrivere il numero e le tipologie di nodi utilizzati nel progetto, insieme a ciò sarà ovviamente descritta anche l'architettura del sistema implementato.

Docker è una piattaforma software che permette di creare, testare e distribuire applicazioni con la massima rapidità, andando a raccogliere il software in unità standardizzate chiamate ***container*** che offrono tutto il necessario per la corretta esecuzione del codice (librerie, strumenti di sistema, codice, e molto altro). Sostanzialmente un *container* può essere visto come una macchina a sé stante, si può infatti "installare" un container partendo da un'immagine preconfigurata ottenendo quindi un ambiente pronto all'uso. Nel mio caso ho utilizzato l'immagine di partenza "*bitnami/spark*" (reperita dalla piattaforma *Docker Hub*), che si basa su un *OS Linux* molto leggero, per l'implementazione dei vari *worker* e *slave* del cluster *Spark*, andando a configurarla opportunamente per poter includere all'interno del nodo anche una versione del server *Apache Livy* già pronta all'uso; è infatti possibile la scrittura di uno *script* chiamato "*Dockerfile*" per poter modificare a proprio piacimento immagini già esistenti andandone a costruire di nuove. Come si può capire da tale premessa generale *Docker* è uno strumento di virtualizzazione molto potente e flessibile che negli ultimi anni sta prendendo piede fortemente nel mondo del *Cloud Computing*, tramite l'utilizzo di *container* i vari provider di servizi *cloud* hanno la possibilità di massimizzare l'utilizzo delle proprie macchine poiché sullo stesso nodo il numero di *container* allocabili è maggiore rispetto al numero di istanze di macchine virtuali, per via dell'elevata leggerezza dei primi sia dal punto di vista di memoria *RAM* richiesta che di *storage*. Fino a qualche anno fa la principale soluzione in ambito della virtualizzazione era l'utilizzo degli *hypervisor*, strumenti software che si occupano di gestire l'infrastruttura virtuale creata a partire dalle risorse fisiche disponibili, la containerizzazione invece utilizza un approccio differente che consiste nel virtualizzare non l'hardware ma solamente il sistema operativo, permettendo quindi ai diversi *container* di condividere eventuali

file e librerie di sistema. La containerizzazione è perciò un approccio moderno nello sviluppo e rilascio del software basato su ***microservizi*** e nello strumento *Docker* è stata implementata mediante l'utilizzo delle funzionalità di isolamento delle risorse del *kernel Linux*, come ad esempio *cgroup* e *namespace*.



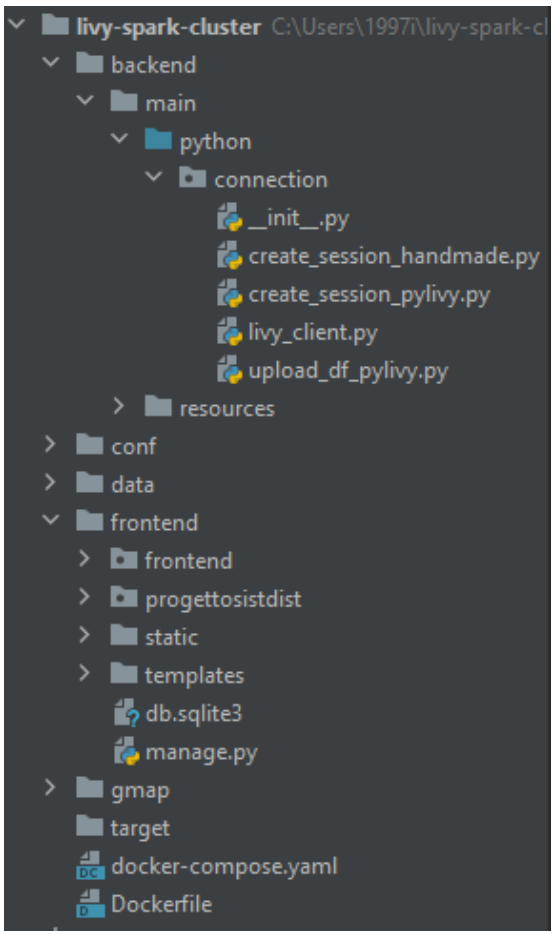
Si procederà ora al trattamento dell'architetture progettuale, come prima cosa è necessaria la descrizione più nel dettaglio del cluster impiegato. Come detto già in precedenza si è deciso l'utilizzo di un cluster *Spark* impiegando per la sua declinazione *PySpark*, il cluster in particolare è composto da:



- 1 nodo *spark-master*;
- 3 nodi *spark-worker* (ad ognuno sono stati riservati 2 GB di memoria RAM per l'esecuzione dei task ed 1 core della CPU);
- 1 nodo per il *livy-server*.

Ognuno di questi componenti consiste in un container in esecuzione e la comunicazione tra loro avviene tramite la definizione di una "*docker network*": ossia una rete costruita in fase di istanziazione dei nodi e basata su indirizzi non routabili, ossia indirizzi IP che non esistono all'esterno della rete stessa e che vengono impiegati unicamente per permettere la

comunicazione tra i vari *container*. La comunicazione con l'esterno avviene tramite l'IP del *localhost* sulle varie porte specificate in fase di configurazione del cluster tramite *Docker*, ad esempio come si può vedere dall'immagine di cui sopra è possibile comunicare con il *livy-server* interrogando la porta 8998 su cui appunto sarà in ascolto il processo relativo allo specifico *container*. Infine per quanto riguarda il *frontend*, è stato sviluppato tramite l'impiego del *framework Django* sulla mia macchina locale ed è possibile accedere alla piattaforma web sulla porta 8000.



Per quanto concerne l'implementazione dell'applicativo, questa consta sostanzialmente di due moduli: un modulo '*frontend*' contenuto nell'omonima cartella che, come dice il nome, racchiude tutte le classi e gli script necessari al funzionamento dell'interfaccia, la cui struttura è stata generata tramite l'impiego dei comandi *Django* standard. Ed un secondo modulo, contenuto nella cartella '*backend*', in cui si implementano le varie interrogazioni andando ad utilizzare le varie classi e metodi della librerie *pylivy* per la costruzione del *client REST* per l'accesso al *cluster*, e andando ad effettuare ove richiesta la visualizzazione tramite *Google Maps*, o costruendo e salvando nell'apposita *directory* sul *frontend* eventuali grafici ottenuti grazia all'utilizzo delle sopracitate librerie *matplotlib* e *seaborn*.

Capitolo 2: Modulo Backend

2.1: Struttura e Comunicazione con il Cluster

Come detto precedentemente, tramite l'utilizzo del framework *Apache Livy* installato su di un nodo del *cluster*, è disponibile un'interfaccia server di tipo *REST* grazie alla quale si possono sottomettere delle richieste ed effettuare elaborazioni su un cluster *Spark* appositamente configurato. Ciò che invece è necessario implementare è quindi il client che debba andare ad utilizzare tali servizi, per fare ciò ho impiegato la libreria *Python* chiamata ***pylivy***, libreria che permette di implementare client al fine di effettuare l'esecuzione di codice remoto su cluster *Spark*. Il core del modulo *backend* è costituito dal file '*livy_client.py*' , in cui sono stati implementati vari metodi d'utilità che saranno poi richiamati dal *frontend*, in particolare centrali sono le funzioni:

- ***create_client()***, si impiega l'oggetto *LivyClient* della libreria *pylivy* che prende in input l'URL della macchina su cui è in esecuzione il server *Livy* al fine di stabilirne la connessione.
- ***create_session(client)***, funzione che prende in input un oggetto di tipo *LivyClient,* costruisce una sessione di lavoro sul server *Livy* e restituisce l'identificativo della sessione necessario al fine dell'esecuzione delle query, per fare ciò si impiega l'omonimo metodo '*create_session*' dell'oggetto sopracitato specificando come tipologia di sessione '*PYSPARK*'. Importante è inoltre il fatto che una volta stabilita la sessione di lavoro, è necessario effettuare l'upload del *dataset* (che si trova sulla macchina locale) sul cluster per poterci lavorare, questo viene fatto mediante l'utilizzo di un ulteriore metodo dell'oggetto *LivyClient* chiamato appunto '*upload*'.

```
s = client.create_session(kind=SessionKind.PYSPARK, spark_conf=conf)
s_id = s.session_id
```

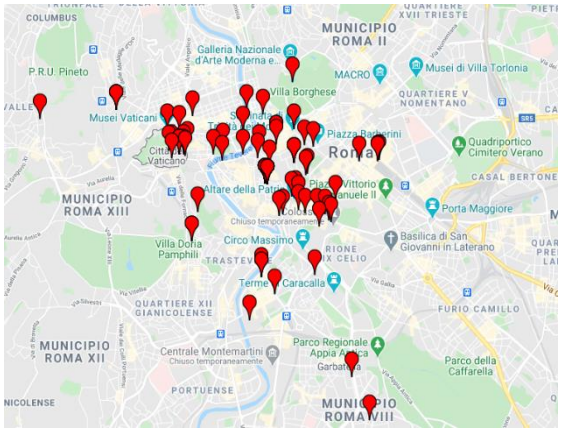
- ***submit_statement(client, session_id, statement, download_name)***, prima di spiegare il significato di tale funzione è necessario descrivere cosa si intende per '*statement*', come già detto *pylivy* permette l'esecuzione di codice remoto su di un cluster *Spark*, questo avviene andando ad effettuare chiamate *HTTP* di tipo *POST* sul server *Livy* passando come contenuto del messaggio una stringa contenete il codice che si vuole eseguire. Con il termine '*statement*' si fa quindi riferimento appunto al codice che si ha la necessità di eseguire sul cluster, siccome nella fase di upload del dataset si specifica anche un nome per tale risorsa, si ha la possibilità di effettuare operazioni sui dati rifacendosi in ogni statement al nome impiegato per il dataset. Secondo lo stesso meccanismo, alla fine dell'esecuzione della *query* il risultato verrà salvato in una variabile come oggetto di tipo *Spark Dataframe*, è possibile ottenere tale risultato mediante l'utilizzo del metodo '*download*' dell'oggetto *LivySession*, a cui si passerà in fase di creazione l'URL del server *Livy* e l'identificativo della sessione, il risultato verrà scaricato come oggetto *Dataframe pandas* specificando il nome della variabile sul cluster. In definitiva tale funzione sottomette lo '*statement*' in input sul cluster grazie all'oggetto '*client*' per effettuarne l'esecuzione, attende il completamento di tale operazione, costruisce l'oggetto di tipo *LivySession* tramite il '*session_id*' in input, infine effettua il download del risultato contenuto sul cluster nella variabile '*download_name*'.
- ***delete_session(client, session_id)***, funzione molto importante che viene richiamata al terminare del lavoro del client al fine di eliminare la sessione sul server *Livy* e conseguentemente liberare risorse sul cluster.

In sostanza queste funzioni vanno a definire l'intero ciclo di lavoro: si parte andando a costruire il client, quando sarà necessario si inizializza una sessione sul server e si effettua *l'upload* del *dataset* su cui si vorrà lavorare, si vanno ad effettuare le varie *query* ottenendo i risultati ed infine si elimina la sessione.

2.2: Statement e Visualizzazione dei Risultati

Al fine di rendere il codice sul *frontend* il più semplice ed ordinato possibile, sono state sviluppate delle funzioni accessorie che restituiscono la stringa che costituisce lo *statement* che si vuole mandare in esecuzione ed il nome della variabile all'interno del codice su cui verrà salvato il risultato. Siccome sono state sviluppate un totale di cinque query, si avrà lo stesso numero di funzioni sopracitate ed inoltre si avranno anche ulteriori funzioni che permettono la visualizzazione o l'ottenimento dei risultati sotto forma di grafico o come pagina web del servizio Google Maps. Nelle sezioni successive si andranno a descrivere i vari *statement* e le strategie e modalità di visualizzazione dei risultati.

2.2.1: statement_lat_lon




```python
def statement_lat_lon():
    # il nome del dataset caricato sul cluster sarà sempre 'data'
    ris_name = "lat_lon"
    statement = textwrap.dedent("""
    lat_lon = data.select('latitude', 'longitude')
    """)
    return statement, ris_name
```

La seguente *query* è stata inserita, come precedentemente detto in merito agli obiettivi iniziali dell'analisi condotta, per poter ottenere informazioni riguardo al contesto specifico dei dati su cui si sta lavorando. Ha infatti lo scopo di impiegare i valori di *latitudine* e *longitudine* presenti come attributi, per andare a visualizzare su mappa in maniera intuitiva, la posizione specifica in cui è stato prodotto il contenuto di ogni post. La visualizzazione avviene passando il *dataframe* risultato dell' esecuzione del metodo '*submit_statement*' (a cui si passa per l'appunto lo *statement* seguente), al metodo '*show_lat_lon_maps*', in cui l'utente può specificare il numero di elementi che desidera visualizzare, siccome la dimensione del dataset risulta essere molto elevata e andare a visualizzare tutti gli elementi

può produrre un risultato poco interpretabile. Per poter usufruire dei servizi della *Google Cloud Platform* relativi a *Google Maps*, si impiega la libreria *gmplot* utilizzando una specifica chiave per l'accesso alle *API Google* fornitami successivamente alla registrazione a tali servizi; in questo caso specifico la visualizzazione su mappa avviene impiegando il metodo '*scatter'* il cui risultato è osservabile nell'immagine di cui sopra. Come si può sostanzialmente intuire dall'esempio, i dati fanno tutti riferimento a contenuti prodotti nella città di Roma.

2.2.2: statement_directions

```
def statement_directions(n_utenti=3, n_post=10):
```

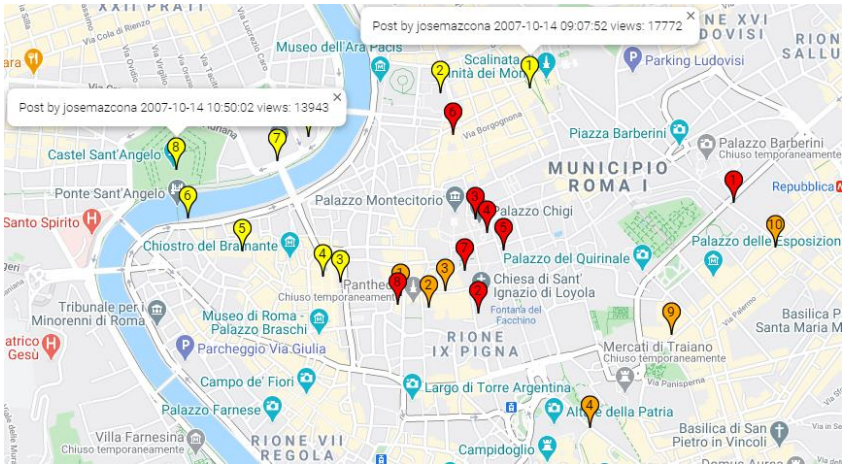
La *query* in questione è stata pensata con lo scopo di aiutare nell'analisi dei *pattern* di mobilità frequenti nel *dataset*, al fine di ricercare le attrazioni turistiche che maggiormente vengano visitate in intervalli di tempo consecutivi, andando a selezionare a questo scopo i post degli utenti più 'popolari'. L'utente può quindi selezionare il numero di '*owner'*, facendo riferimento al nome dell'attributo, con maggiori visualizzazioni sui propri contenuti. Questi sono ottenuti applicando in primo luogo un'aggregazione e poi una funzione aggregata che restituisca la somma per ogni elementi sulla colonna '*views'*. Si ordina poi il risultato in maniera decrescente selezionando il numero di righe specificato in input, inoltre a tale dataset intermedio si va ulteriormente ad applicare un ordinamento in maniera decrescente sull'attributo '*dateTaken',* che, come dice il nome, indica la data in cui è stato prodotto il contenuto del rispettivo post.

```
owner = data.groupby('owner').agg(sum('views').alias('views'))
owner = owner.orderBy(desc('views'))
top = owner.head(""" + str(n_utenti) + """)
```

```
ppowner = ppowner.orderBy(desc('dateTaken'))
```

Il risultato grafico viene prodotto sempre tramite l'utilizzo di un ulteriore metodo che prende in input il *dataframe* ottenuto nella fase di *download* e che a sua volta si poggia sui servizi Maps. Dall'esempio sottostante si evince come sia possibile, calibrando opportunamente i valori in input ed il numero di post visualizzati su mappa per ognuno dei top utenti, tracciare dei veri e propri percorsi di attrazioni turistiche più frequentemente visitate in maniera

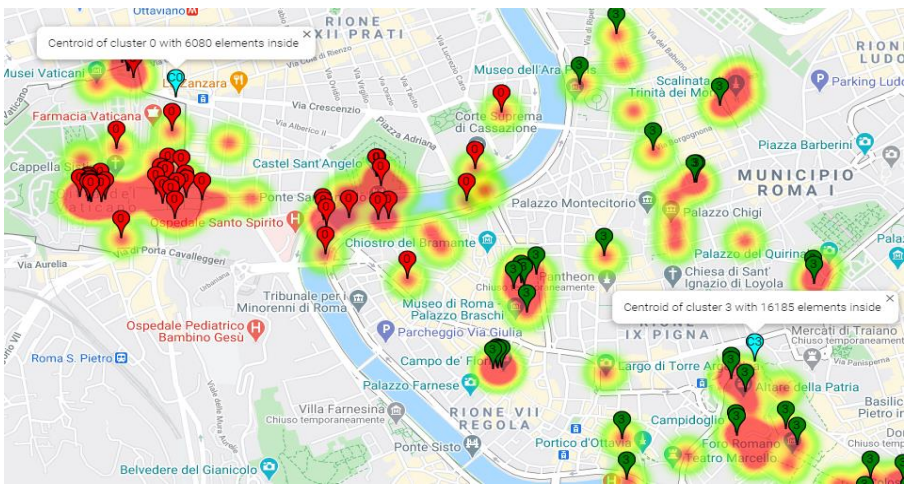
consecutiva nel tempo. Questo può essere utile al fine di migliorare o introdurre determinati servizi di *tour* o visite guidate all'interno della città.



2.2.3: statement_kmeans

```
def statement_kmeans(k=5)
```

La seguente interrogazione prende in input un valore numerico intero e impiega al suo interno strumenti della libreria di *Machine Learning* '*pyspark.ml*' per l'applicazione di algoritmi di *data mining* su *DataFrame Spark*. In particolare si è deciso l'utilizzo del famoso algoritmo di clustering ***KMeans***, allo scopo di andare a ricercare il numero di centroidi specificato dall'utente. Grazie all'applicazione di questo algoritmo è stato possibile visualizzare su mappa tali punti, in aggiunta ad alcuni campioni appartenenti ai vari cluster, andando inoltre a definire una '*heatmap*' sul risultato della clusterizzazione, calcolata mediante l'utilizzo dell'omonima funzione della libreria *gmplot*.



Come si può notare dall'esempio di cui sopra, il risultato ottenuto va ad evidenziare la posizione dei diversi centroidi ed alcuni punti appartenenti ad ogni cluster, differenziati in base al colore. Inoltre la colorazione più o meno accesa di determinate zone della mappa, sta ad indicare una presenza maggiore o minore di elementi con un elevato numero di visualizzazioni. L'idea dietro tale interrogazione risulta essere la possibilità di effettuare analisi al fine di determinare la migliore locazione di un servizio, si possono infatti identificare zone, che non siano nei pressi dei principali luoghi turistici, in cui è elevato il traffico di visitatori. Inoltre i centroidi potrebbero dare informazione su dove sia più opportuno andare effettivamente a localizzare tali servizi.

Per poter applicare l'algoritmo *KMeans* è stato prima necessario seguire alcuni passaggi:

1. Come prima cosa è stato costruito un oggetto di tipo ***VectorAssembler*** al cui interno sono state specificate le *features* da impiegare per il calcolo della clusterizzazione.

```
vecAssembler = VectorAssembler(inputCols=["latitude", "longitude"], outputCol="features")
```

2. Successivamente si è applicata la trasformazione dettata dall'oggetto precedentemente discusso, per ottenere un *dataset* in cui le colonne d'interesse fossero raggruppate all'interno della colonna '*features*'.

```
new_sdf = vecAssembler.transform(d)
```

3. Dopo le prime fasi preliminari, è stato possibile costruire l'oggetto *KMeans* vero e proprio, al cui interno è implementato l'omonimo algoritmo di *clustering*, andandolo ad inizializzare con il numero di centroidi specificato dall'utente, selezionando un '*seme random*' univoco per le diverse esecuzioni, ed una modalità di inizializzazione dei centroidi alla prima iterazione basata sull'applicazione preventiva dell'algoritmo ***KMeans++*** (si risolve perciò in primo luogo tale problema ed i centroidi risultanti si impiegano come prima inizializzazione dell'algoritmo classico).

```
kmeans = KMeans().setK(""" + str(k) + """).setSeed(42).setInitMode('k-means||')
```

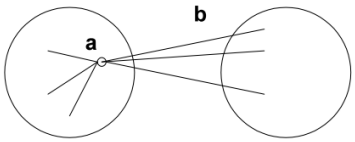
4. È stato quindi possibile effettuare l'addestramento del modello così ottenuto, per poi andarlo ad applicare sul *DataFrame* in maniera tale da associare ad ogni elemento il *cluster* a cui fa riferimento.

```
model = kmeans.fit(new_sdf.select('features'))
transformed = model.transform(new_sdf)
evaluation = evaluator.evaluate(transformed)
```

Viene inoltre calcolato, grazie all'impiego di un oggetto di tipo ***ClusteringEvaluetor***, il *coefficiente di silhouette*, ossia una misura di validità della clusterizzazione effettuata; tale informazione viene ovviamente restituita all'utente in maniera tale da avere un indicazione sulla bontà del risultato ottenuto in termini di algoritmo di *data mining*. Di seguito si definiscono i passaggi per il calcolo di tale metrica.

Per un punto i
- **Sia** C_i **il cluster di** i
- **Calcola:** a_i **= distanza media di** i **dagli altri punti di** C_i
- **Calcola:** b_i **=** *min per ogni cluster* C, $C \neq C_i$ **(distanza media di** i **dai punti del cluster** C**)**
- **Silhouette Coefficient** s_i **per il punto** i**:**

$$s_i = (b_i - a_i) / max(a_i, b_i)$$

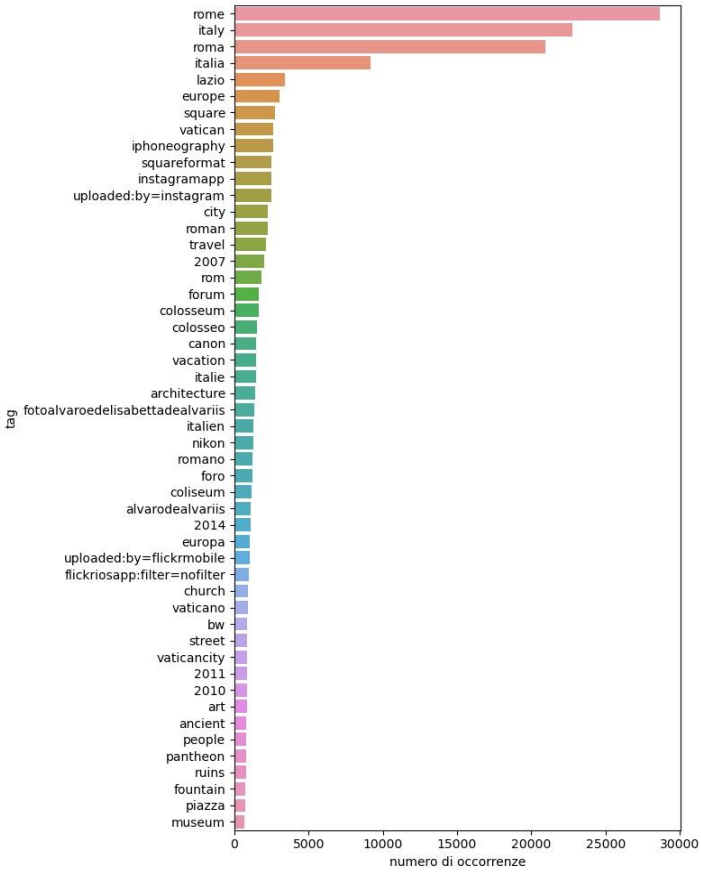


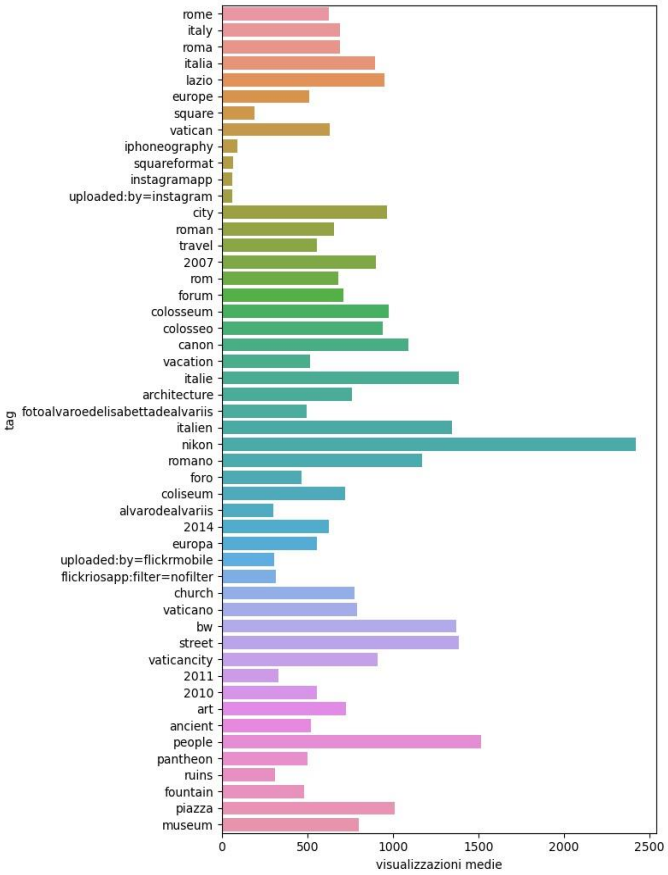
2.2.4: statement_tags

```
def statement_tags(num_esclusi, n=20, order_by_mean=False, escludi=True)
```

La seguente *query* calcola il numero di volte che compare un tag in un determinato post e la media di visualizzazioni che ottengono tali elementi. È inoltre possibile specificare l'ordinamento del risultato intermedio, scegliendo se considerare per l'appunto i valori calcolati di visualizzazioni medie per tag, oppure il numero di occorrenze (si può anche decidere di applicare l'esclusione di tag dal risultato, che non superino una determinata soglia di occorrenze). Lo scopo di tale interrogazione è andare ad analizzare e determinare l'uso di tag che permettano ai diversi contenuti di raggiungere un pubblico più ampio ed averne una maggiore influenza, oppure per analizzare in base a tale tendenza l'effettivo interesse dei visitatori ai vari luoghi turistici della città. La visualizzazione viene effettuata tramite un ulteriore metodo di supporto facendo uso delle librerie *Python matplotlib* e *seaborn*, il risultato è prodotto

sotto forma di grafico a barre che viene salvato su un apposita cartella del *frontend* e mostrato all'utente nell'applicazione web. Di seguito si mostra un esempio di risultato





Per l'implementazione di tale *query* è stato quindi necessario eseguire una '*flattenizzazione*' della colonna '*tags*', poiché ogni elemento di questa è rappresentato da una lista dei tag utilizzati nel relativo post. A tale scopo preventivamente si effettua il raggruppamento sui valori della colonna in questione, andando a calcolare il numero di occorrenze e le visualizzazioni medie. Si utilizza poi la funzione '*explode*' del package '*pyspark.sql.functions*' che restituisce, per ogni valore contenuto nelle liste dei tag, un DataFrame con all'interno nuove righe in cui compare tale valore come elemento singolo.

```
d = data.groupby('tags').agg(count('tags').alias('count'), mean('views').alias('mean_views'))
tag = d.select(explode('tags').alias('tag'), 'count', 'mean_views')
tag = tag.groupBy('tag').agg(sum('count').alias('count'), mean('mean_views').alias('mean_views'))
```

Infine si esegue nuovamente il raggruppamento, ma questa volta su tale colonna risultante, andando ad effettuare la somma delle occorrenze precedentemente calcolate, ed un ulteriore calcolo delle visualizzazioni medie, sempre impiegando il risultato precedente.

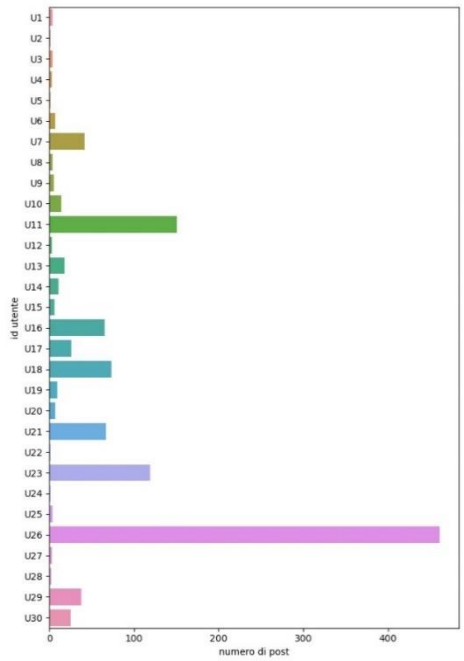
2.2.5: statement_owner

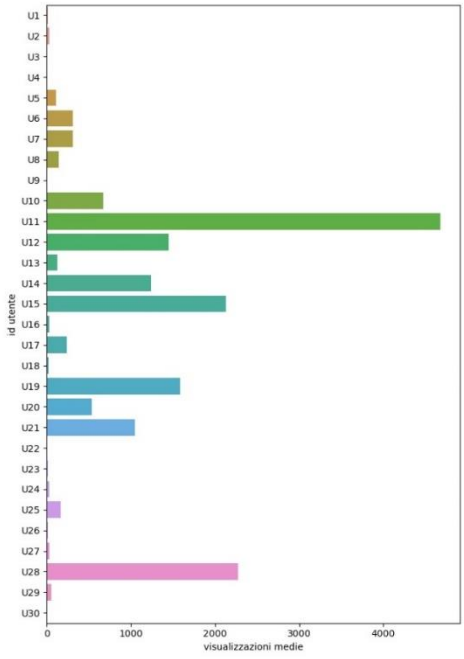
```
def statement_owner(n=20):
```

L'ultima interrogazione implementata permette la restituzione di varie info relative agli utenti 'attivi' campionati in base al valore in input, ossia coloro i quali hanno condiviso contenuti sulla piattaforma recentemente. Queste informazioni risultano essere:

- Username, codice identificativo univoco generato dal sistema, data dell'ultimo post condiviso, tag più usati nei rispettivi post e numero di occorrenze di questi; tali informazioni sono riportate sulla piattaforma web sotto forma tabellare.
- Grafico a barre che mostra il numero totale di post per tali utenti.
- Grafico a barre che mostra le visualizzazione medie per post condiviso.

U11	Baz 120	Jan. 27, 2017, 8:32 a.m.	['life' 'street' 'city' 'portrait' 'people' 'urban' 'blackandwhite' 'bw' 'italy' 'rome' 'roma' 'monochrome' 'italia' 'faces' 'candid' 'streetphotography' 'streetportrait' 'olympus' 'monotone' 'streetphoto' 'unposed' '45mm' 'omd' 'decisivemoment' 'candidportrait' 'streetphotographer' 'm43' 'streetcandid' 'mft' 'streetphotograph' 'primelens' 'em5' 'candidstreet' 'candidface' 'grittystreetphotography']	29



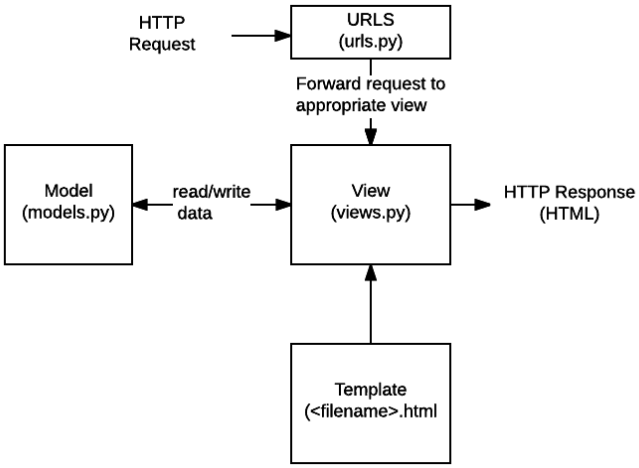


Per ottenere tutto ciò è stato necessario andare ad effettuare un ordinamento delle righe del *dataset* secondo l'attributo '*dateTaken'*, applicando questo su una '*Window' Spark*. È stato quindi possibile lavorare su tutte le righe del dataset operando solo sulla colonna specificata dalla finestra, in maniera tale da potervi associare un indice numerico progressivo, basato sull'ordinamento temporale precedentemente definito.

Lo scopo della seguente interrogazione è quello di permettere all'utente di analizzare i *trend* relativi ai tag più utilizzati di recente e come questi possano portare un proprio contenuto ad essere maggiormente visualizzato. Potendo sfruttare per tale scopo anche le informazioni ottenute dalla *query* precedentemente trattata.

Capitolo 3: Modulo Frontend

3.1: Framework Django e Componenti Utilizzati



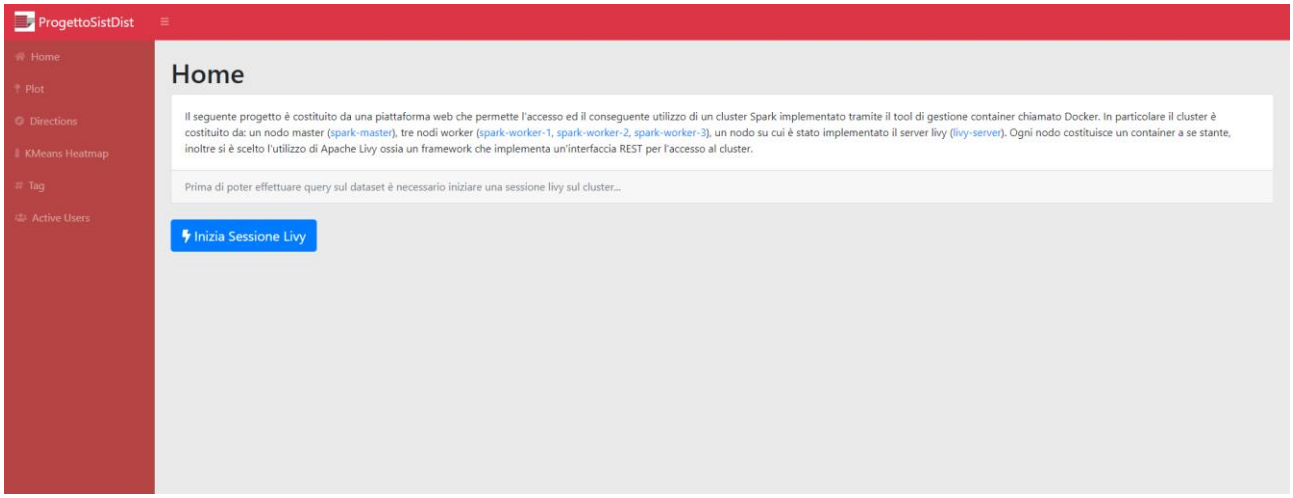
Django è un *framework Python open source* per lo sviluppo di applicazioni web che permette di gestire in maniera automatica la costruzione dell'impalcatura sottostante, in modo tale da slegare lo sviluppatore da tale onere, potendosi concentrare sulla scrittura dell'app e sulla logica implementativa. Come si può notare dallo schema di cui sopra, un applicazione *Django* è costituita da diversi moduli:

- **URLS**: modulo implementato dal file '*urls.py*' in cui si va specificare il *matching* tra l'indirizzo *URL* e il controllore adibito alla sua gestione, riceve quindi una richiesta *HTTP* e procede al suo smistamento.
- **View**: modulo che implementa, all'interno del pattern *MVC,* la componente *Controller*, si definisce a tale scopo un file '*views.py*' in cui ogni metodo in esso presente corrisponderà ad un controllore per uno specifico *path*, all'interno di questi metodi viene sostanzialmente implementata la logica di business dell'applicazione.

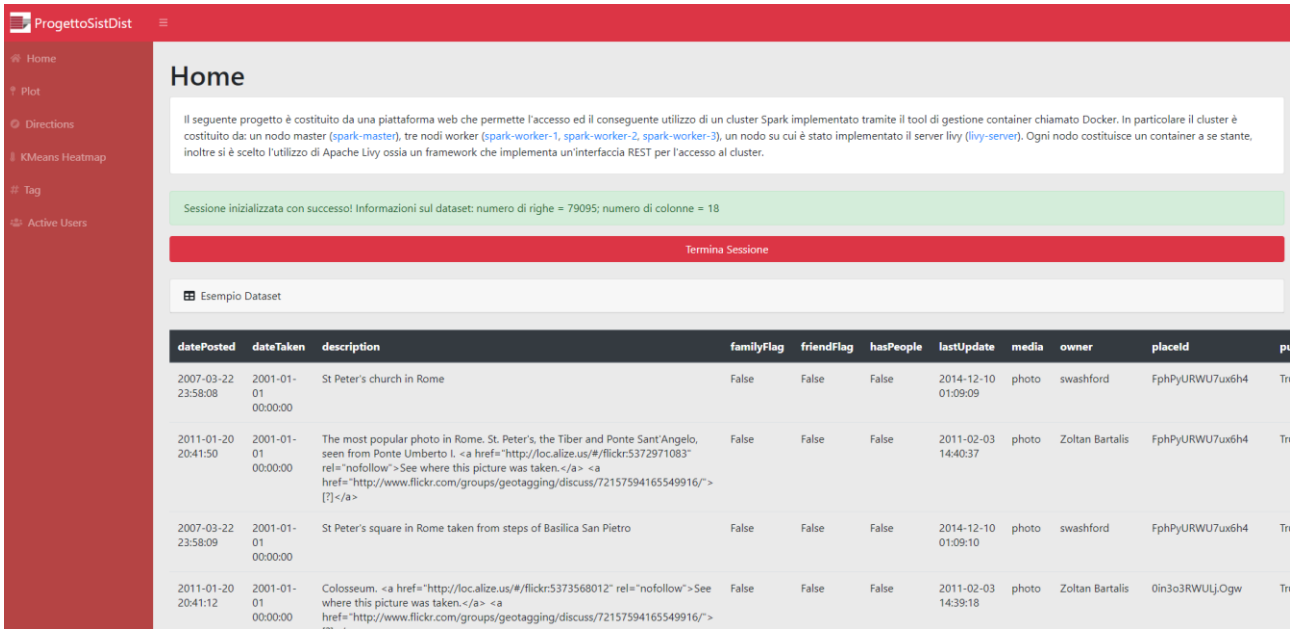
- **Model**: componente che gestisce l'accoppiamento tra oggetti *Pyhton* ed elementi all'interno del *database*, siccome la porzione *backend* del sistema è totalmente sviluppato in *Spark* non è stata necessaria l'implementazione di tale modulo.
- **Template**: sostanzialmente fa riferimento alla porzione *View* del pattern *MVC*, ed è costituito dai file che definiscono la struttura ed il *layout* impiegato per la visualizzazione e restituzione del risultato, a seguito della richiesta *HTTP* inviata al modulo *View*.

Per lo sviluppo del progetto è stata necessaria l'implementazione delle sole componenti: *URL*, *Views* e *Template*. Di cui per l'ultima, come detto inizialmente, si è impiegato il *framework Bootstrap*. Inoltre tutti i metodi di supporto visti nelle sezioni precedenti, vengono richiamati nelle varie *views,* sostanzialmente il flusso di lavoro è il seguente: una volta che un utente si connette alla piattaforma viene creato un *client livy,* successivamente si inizializza una sessione di lavoro sul *cluster* e si carica il *dataset*, fatto ciò l'utente può selezionare dal menu grafico la *query* che da svolgere, ogni query avrà differenti parametri in input e sarà associata al relativo metodo del modulo backend. Al termine dell'esecuzione, si andrà ad effettuare il download del risultato per poi passarlo al rispettivo metodo accessorio per la produzione dell'*output* grafico, questo può essere sotto forma di un ulteriore pagina *HTML* aperta in automatico nel *browser web* dell'utente (nel caso si trattasse di *query* che impiegano servizi *Maps*), oppure semplicemente si andrà ad allegare all'interno della pagina *web* il rispettivo grafico risultante.

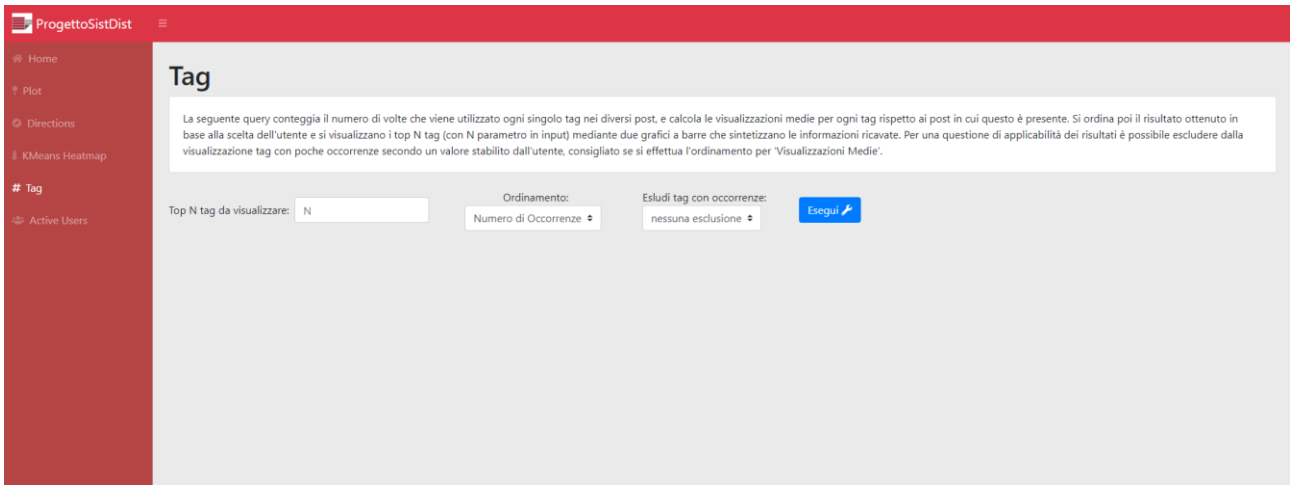
3.2: La GUI del Progetto



L'interfaccia grafica di riferimento per la pagina iniziale si può osservare nell'esempio soprastante, all'avvio dell'applicativo sarà richiesto all'utente, tramite l'apposito pulsante, di iniziare una sessione sul *server livy*, in maniera tale da poter caricare il *dataset* sul *cluster*.



Una volta fatto ciò, viene mostrata una panoramica in forma tabellare dei dati su cui si vuole lavorare, come si può osservare, la barra di navigazione alla sinistra dell'interfaccia costituisce il metodo principale affinché l'utente possa interagire con il *software*. Sono state inserite tante opzioni quante le interrogazioni disponibili sui dati e in ognuna di tali pagine sarà presente un *form* personalizzato, impiegato al fine di inserire l'input della query che si vuole sottoporre.



Ovviamente non è mancata la gestione dei possibili errori di utilizzo della piattaforma da parte dell'utente, in particolare:

- Non è possibile navigare tra le finestre corrispondenti alle varie *query* se prima non si inizializza una sessione.
- È vietato l'inserimento all'interno dei *form* di valori scorretti (negativi o non numerici).
- Non è possibile effettuare l'esecuzione di una *query* se non si compilano tutti i rispettivi *form*.