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

Corso di Laurea Magistrale in Ingegneria Informatica

Modelli e Tecniche per Big Data

Progetto 5 - FlickrAnalytics

Candidato:

Ivan Scuderi 216635

Relatori:

Prof. Paolo Trunfio

Prof. Fabrizio Marozzo

Anno Accademico 2020-2021

Indice

Capitolo 1: Introduzione

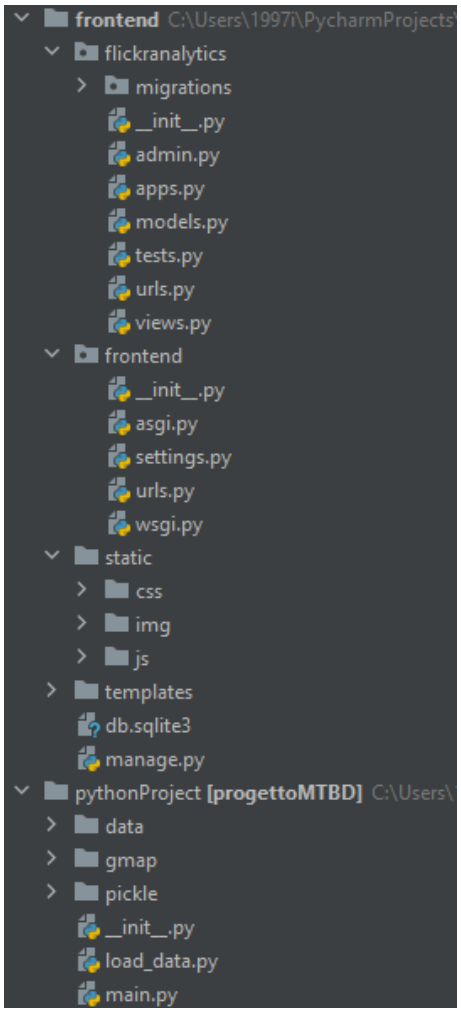
1.1: Descrizione del Progetto e Scopo

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. Siccome in primo luogo non avevo alcuna informazione relativamente alla struttura del dataset su cui avrei dovuto lavorare ed alla semantica delle varie colonne, 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 e Architettura

Per quanto riguarda le tecnologie impiegate per lo sviluppo di tale progetto, come da traccia, 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***. Per quanto riguarda ulteriori tecnologie impiegate si nominano: le librerie ***pandas***, ***numpy***, ***seaborn*** e ***matplotlib*** impiegate per una prima fase di preprocessing del dataset e

visualizzazione dei risultati. 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 il *framework* ***Apache Arrow*** che mi ha permesso l'interoperabilità tra *Spark* e le librerie *Python* sopracitate, in modo tale da poter effettuare una conversione, efficace e priva di *overhead* dovuto alla serializzazione, dei dati e degli oggetti impiegati tra sistemi scritti in linguaggi di programmazione differenti. Sostanzialmente mi ha quindi permesso la conversione di *Dataframe Spark* (gestiti dalla *JVM*) in *Dataframe pandas* e viceversa. 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.



Si procede ora a descrivere l'**architettura** 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 di progetto '*pythonProject',* che consiste nel '*backend*' dell'applicazione, in cui si conservano le informazioni relativamente alla locazione dei dati sul sistema e si implementano le varie interrogazioni, 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. Più nello specifico lo script '*main.py*' contiene tutti i metodi adibiti al caricamento, analisi dei dati e costruzione dei risultati, questo viene importato ed impiegato sul '*frontend'* nello script '*view.py*' che costituisce appunto la porzione '*View'* dell'architettura M.T.V, *'Model, Template, View'*, Django (sostanzialmente rappresenta il '*Controller'* nel pattern M.V.C). Si andranno a descrivere più nel dettaglio nei prossimi capitoli le strutture interne dei singoli moduli analizzandone funzionalità e scopi, andando a trattare nello specifico le query svolte e le varie scelte progettuali adottate nel corso dello sviluppo e del deploy della piattaforma.

Capitolo 2: Modulo Backend

2.1: Struttura e Scelte Progettuali

Prima di approfondire nello specifico la struttura dei programmi e dei metodi che costituiscono tale modulo, si introducono alcune scelte progettuali adottate nel corso dello sviluppo, in particolare sono state riscontrate diverse difficoltà tecniche durante la fase di lettura del file *json* contenete il *dataset*. Più precisamente non è stato possibile impiegare le primitive di *PySpark* per effettuare tale operazione, per via di alcuni errori sulla composizione degli elementi del file. Inoltre anche l'utilizzo del metodo *'read_json'* della libreria *pandas*, per effettuare il caricamento diretto di questo, non ha portato risultati per via dell'elevata dimensione del *dataset* (circa 3 GB). Infatti tale metodo lavora caricando interamente in memoria il file di cui si intende effettuare la lettura ma, per via di alcune limitazioni del mio ambiente di sviluppo, non mi è stato possibile effettuare la lettura diretta. Per questo motivo ho optato per l'impiego di solo alcuni blocchi del dataset, in particolare ho sviluppato uno script utilizzando le già note librerie *Python* che permetta, dato un valore numerico in input, di caricare le porzioni del file desiderate.

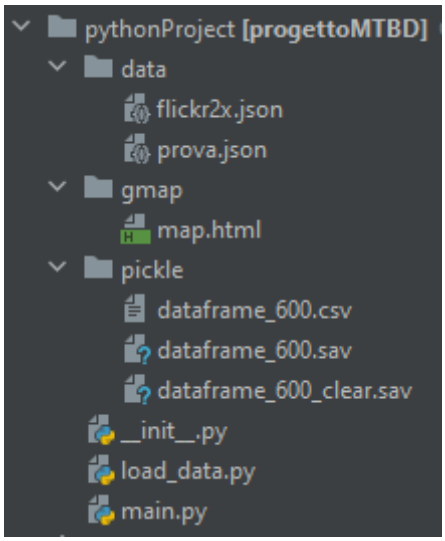
```
def loadjson(chunks=600):
    ##DATASET CONTENENTE INFORMAZIONI RIGUARDO POST INERENTI AL SOCIALWEB FLICKR IN FORMATO JSON
    data = pd.read_json(r"C:\Users\1997i\PycharmProjects\pythonProject\data\flickr2x.json",
                        lines=True, chunksize=100, orient="records")
```

Lo *script* sopracitato è contenuto nel file '*load_data.py*' e viene implementato all'interno del metodo '*loadjson*'. L'utente, tramite l'interazione permessa dall'interfaccia grafica web, ha la possibilità di selezionare un numero preciso di 'chunks' dal file su cui vuole effettuare le analisi, ognuna di queste porzioni corrisponde, come specificato dai valori '*chunksize*' e '*orient*', a cento elementi del file *json*, che costituiranno i *record* del *dataset* su cui successivamente si andrà a lavorare (in sostanza, se come nell'esempio si specifica il valore 600, si otterrà un *Dataframe* della dimensione di 600x100 elementi,

ossia 60'000 *record*) . Inoltre tale metodo, oltre a permettere il caricamento dei dati, effettua una fase preliminare di *preprocessing* sul *dataset* così ottenuto mediante l'impiego della libreria *Python pandas*, al fine di eliminare eventuali attributi ridondanti, effettuare sostituzione di valori nulli o poco significativi e convertire il tipo di eventuali colonne. Una volta terminata tale procedura di pulizia dei dati, si procede a salvare su disco il *dataset* ottenuto, sia sotto forma di file in formato '*csv*', sia tramite la serializzazione fornita dalla libreria Pyhton ***pickle***, sotto forma di *Dataframe pandas*.

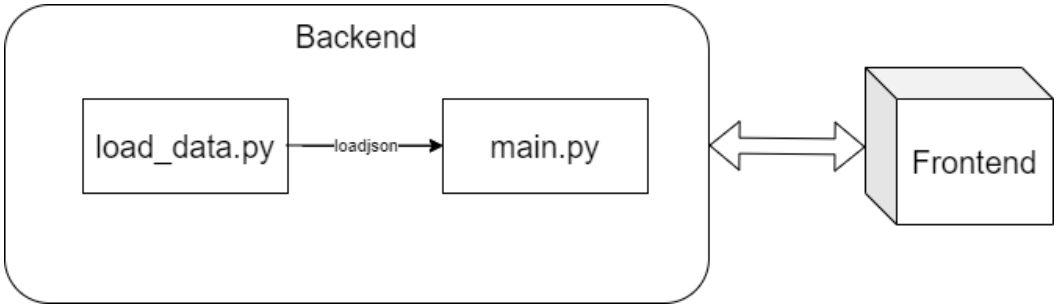
```
##NUMERO DI CHUNK MASSIMI CHE SI VOGLIONO ESTRARRE DAL FILE FLICKR2X.JSON
stop = chunks

PICKLE_PATH = "C:\\Users\\1997i\\PycharmProjects\\pythonProject\\pickle"
namePickle = "dataframe_"+str(stop)+".sav"
namePickleClear = "dataframe_"+str(stop)+"_clear.sav"
nameCsv = "dataframe_"+str(stop)+".csv"
pathPickle = os.path.join(PICKLE_PATH, namePickle)
pathCsv = os.path.join(PICKLE_PATH, nameCsv)
pathPickleClear = os.path.join(PICKLE_PATH, namePickleClear)

if os.path.exists(pathCsv):
    print("Il file csv del dataset formato dai chunks indicati esiste già: ", pathCsv)
    return (pathCsv, pathPickleClear)
```

Tenendo presente il fatto che non è consentito all'utente specificare un punto di partenza per la lettura e caricamento del file, è stato possibile implementare un controllo sull'eventuale presenza di un *dataset* già elaborato, e formato dal numero specificato di elementi, per restituirlo prontamente senza dover impiegare ulteriore tempo per processare ed ottenere una risorsa già disponibile.



Per quanto riguarda la struttura del backend questa è costituita da varie cartelle in cui vengono inseriti eventuali file già a disposizione nel progetto, o creati durante la fase di esecuzione, in particolare: la cartella ***data*** contiene il file *json* su cui si andrà a lavorare, più un secondo file '*prova.json*', utilizzato unicamente per estrarre i nomi delle colonne e impiegarle al fine della costruzione di un Dataframe pandas

vuoto, su cui poi poter effettuare l'*'append'* delle varie porzioni estratte dal file principale. La cartella ***pickle*** contiene tutti i file ottenuti al seguito dell'esecuzione del metodo sopra descritto, proprio in tale cartella si effettua il controllo se un determinato dataset esiste già per poi restituirne il *path* assoluto. Infine la cartella ***gmap*** ha lo scopo di conservare i file in formato html risultanti dall'esecuzione delle query che lavorano impiegando i servizi Google per la geolocalizzazione di elementi sul motore Maps, che verranno poi aperti sul browser tramite la libreria *Python* ***webbrowser***.



La struttura del progetto vede quindi il *frontend*, con il quale si interfaccerà l'utente, impiegare i servizi di elaborazione dei dati implementati all'interno dei metodi del file '*main.py'*, che rappresenta il centro di tutto l'applicativo e che, a sua volta, impiegherà la primitiva precedentemente descritta facente parte dello script '*load_data.py*' per il caricamento e la pulizia del dataset. Importante è il fatto che la, presenza del file *'__init__.py'* nella directory di progetto '*pythonProject'* del modulo *backend,* permette di far riconoscere dall'interprete *Python* questa cartella come *package* importabile, in modo da poter eseguire tale *import* nella porzione del *frontend* relativo alla gestione delle '*views' Django*. Si andrà quindi successivamente a trattare più nello specifico il *core* della piattaforma andando a descrivere nel dettaglio le *query* implementate, facendo riferimento alle informazioni ottenute ed ai possibili risultati in ambito di analisi dei dati che si possono trarre.

2.2: Il Core del Progetto - main.py

```
GMAP_PATH = r"C:\Users\1997i\PycharmProjects\pythonProject\gmap"
IMG_PATH = r"C:\Users\1997i\PycharmProjects\frontend\static\img"

apikeys = "AIzaSyBAfHsJ7miedYS1yoU66VVoPA9fdAuv9DA"

spark_home = os.getenv("SPARK_HOME")
sc = SparkContext(master="local[*]", appName="SimpleApp")
spark = SparkSession.builder.appName("SimpleApp").getOrCreate()
sc.getConf().set('spark.sql.caseSensitive', 'true')
spark.conf.set("spark.sql.execution.arrow.pyspark.enabled", "true")
```

Come detto precedentemente tale file *Python* costituisce il punto focale di tutto il sistema, contiene le varie importazioni di librerie e metodi che permettono il collegamento con la componente *Spark* e con i servizi della *Google Cloud Platform* per la visualizzazione tramite *Google Maps*. In particolare si può vedere come per l'appunto si vada ad impiegare una chiave per l'accesso alle *API Google*, fornitami in seguito alla registrazione ai servizi offerti di cui necessitavo l'utilizzo. Le operazioni preliminari effettuate sono necessarie alla creazione di uno ***SparkContext*** che funga da ponte di collegamento per il cluster *Spark*, costituito da un unico nodo ossia la macchina su cui effettuo il *deploy* della piattaforma. Viene inoltre creato un oggetto di tipo ***SparkSession***, che mi permetterà di accedere a funzionalità del *framework Spark*, quali l'utilizzo di *Dataframe Spark* e la possibilità di effettuare interrogazioni sui dati mediante l'uso dello *SparkSQL*. Infine si configura opportunamente tale oggetto in modo da rendere possibile l'utilizzo della componente *Apache Arrow* e tutti i benefici da questa permessi. Nel proseguo della trattazione si andranno a descrivere maggiormente nello specifico i metodi contenuti all'interno di questo programma, esposti come interfaccia grazie alla quale l'utente, mediante il *frontend*, ha la possibilità di elaborare i dati ed ottenere e visualizzare i risultati.

2.2.1: Il metodo – start

```
def start(c=600)
```

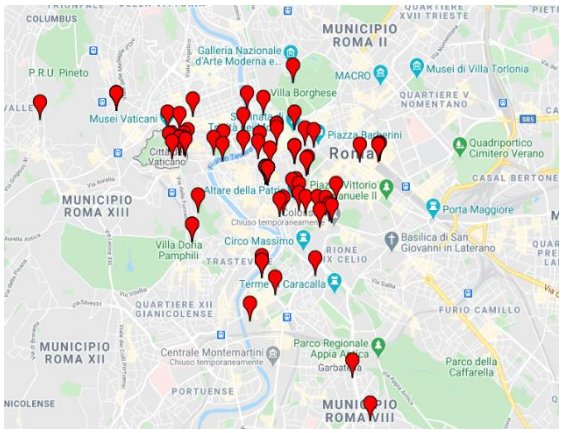
Questo primo metodo permette di utilizzare all'interno del file '*main.py*' le funzionalità del programma '*load_data.py*', infatti risulta essere molto importante poiché permette la restituzione dell'oggetto *DataFrame Spark* costituito dal numero di *chunk* specificati in input dall'utente. Tale oggetto è cruciale per l'applicazione dei successivi metodi.

```
path = load_data.loadjson(c)[1]
df = pickle.load(open(path, 'rb'))
##USO APACHE ARROW IN PYSPARK TRAMITE PYARROW PER CARICARE UN DATAFRAME PANDAS SULLA JVM OTTENEDO UN DATAFRAME SPARK
sdf = spark.createDataFrame(df).cache()
return sdf
```

2.2.2: Il metodo – plot_in_gmap

```
def plot_in_gmap(sparkdf, elements=100)
```

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, impiegando a tal fine il metodo '*scatter*' della libreria *gmplot*. Inoltre l'utente può specificare il numero di elementi che desidera visualizzare poiché, siccome la dimensione del dataset, nonostante la riduzione dovuta al suo eventuale non completo utilizzo, risulta comunque essere molto elevata, andare a visualizzare tutti gli elementi può produrre un risultato poco

interpretabile. Come si può sostanzialmente intuire dall'esempio di cui sopra, i dati fanno tutti riferimento a contenuti prodotti nella città di Roma.

2.2.3: Il metodo – show_placeid_on_map

```
def show_placeid_on_map(sdf, number=4)
```

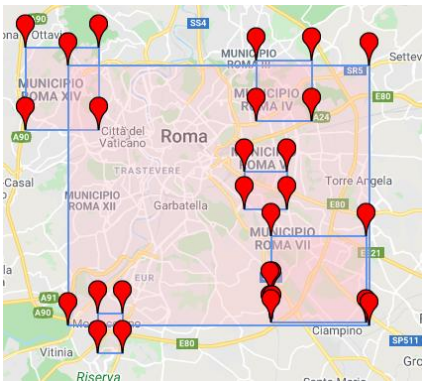
Come per la precedente, anche la seguente query ha lo scopo di analizzare le caratteristiche ed il contesto dei dati con cui ci si trova a lavorare. Più nello specifico la presenza di un attributo chiamato '*placeId* ', poteva suggerire il fatto che in qualche modo si andasse a identificare per mezzo di una stringa il luogo specifico, o la zona, a cui un determinato contenuto faceva riferimento. Per poter controllare la veridicità di tale affermazione, la soluzione è stata quella di effettuare raggruppamenti dei dati sui valori di tale attributo, andando a calcolare per ognuno di questi il massimo e minimo valore

```
latlonplaceMax = sdf.groupby('placeId').max('latitude', 'longitude')
latlonplaceMin = sdf.groupby('placeId').min('latitude', 'longitude')
latlonplace = latlonplaceMax.join(latlonplaceMin, 'placeId', 'inner')
```

di latitudine e longitudine. Una volta fatto ciò è stato possibile effettuare una *join* dei *dataset* risultanti, per arrivare ad ottenere gli estremi in termini di punti, delle zone a cui i diversi '*placeId*' si riferiscono.

```
latitude = [dictrow['max(latitude)'], dictrow['max(latitude)'], dictrow['min(latitude)'],
            dictrow['min(latitude)']]
longitude = [dictrow['max(longitude)'], dictrow['min(longitude)'], dictrow['min(longitude)'],
             dictrow['max(longitude)']]
gmap.scatter(latitude, longitude, color='r', marker=True, size=5)
gmap.polygon(latitude, longitude, face_color='pink', edge_color='cornflowerblue', edge_width=2)
```

La visualizzazione avviene andando a segnare su mappa tali estremi e riempiendo lo spazio in essi contenuto tramite l'impiego del metodo '*polygon*' della libreria *gmplot.* Di seguito si mostrano esempi del risultato prodotto.

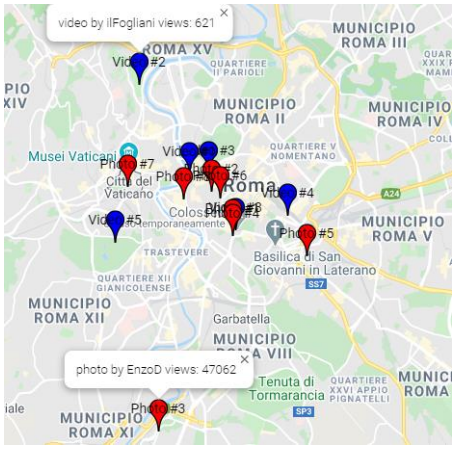


Come si può notare l'esempio di sinistra farebbe pensare al fatto che la supposizione iniziale fosse corretta, in realtà indagando più nello specifico si possono evidenziare chiare intersezioni o addirittura sovrapposizioni di zone. Per questo motivo si è deciso di non impiegare tale attributo per interrogazioni più complesse, poiché avrebbe potuto portare a risultati erronei.

2.2.4: Il metodo – top_n_post_per_views

```
def top_n_post_per_views(sdf, n=10, photo=True, video=True)
```

Con la seguente interrogazione si vanno ad analizzare in maniera più specifica i dati, nel tentativo di ottenere risultati che possano essere più o meno interessanti dal punto di vista pratico. In particolare si indaga su quali tipologie di post fruttino maggiori visualizzazioni all'interno della piattaforma *Flickr*. L'utente ha la possibilità di ricercare, secondo un dato valore numerico in input, i migliori post per visualizzazioni, potendo filtrare a seconda della tipologia di contenuto. L'attributo '*media*' specifica per l'appunto tale informazione e può assumere due soli valori: '*photo*' oppure '*video*'. Si può quindi



specificare se andare a ricercare i migliori elementi per entrambi, o escludere determinate tipologie di post. Ovviamente la visualizzazione avviene su mappa, nel tentativo di evidenziare luoghi in cui è più probabile che si ottenga maggior seguito in base al contenuto scelto. Come ci si poteva aspettare, anche con un campione non troppo

numeroso, la predominanza è marcata nei pressi delle grandi attrazioni turistiche presenti nella capitale. Si può vedere dall'esempio come vengano distinti i vari post mediante l'impiego di colorazioni differenti per i *marker*, in base alle diverse tipologie di media, inoltre è possibile conoscere l'utente che ha condiviso il rispettivo elemento ed il numero di visualizzazioni da questo ottenute.

2.2.5: Il metodo – top_n_directions

```
def top_n_directions(sdf, n=3, np=10)
```

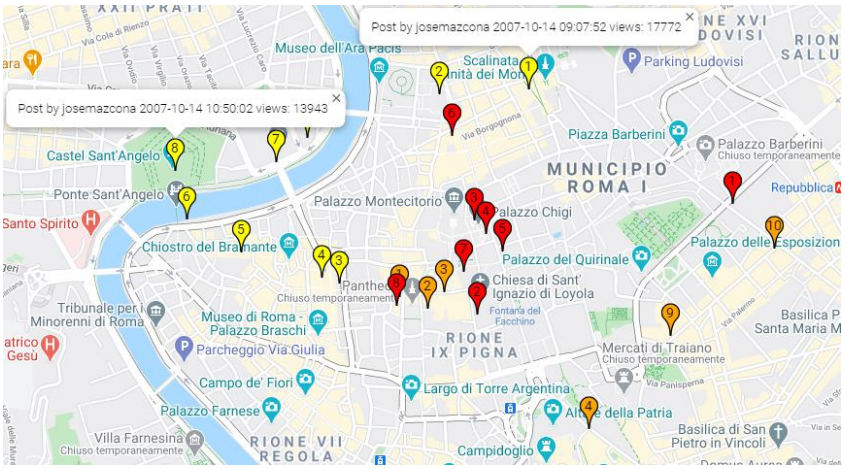
La *query* in questione è stata pensata con lo scopo di aiutare nell'analisi dei *pattern* di mobilità frequenti nel *dataset*, al fine di ricercare le attrazioni turistiche che maggiormente vengano visitate in intervalli di tempo consecutivi, andando a selezionare a questo scopo i post degli utenti 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*'.

```
owner = sdf.groupby('owner').agg(sum('views').alias('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.

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

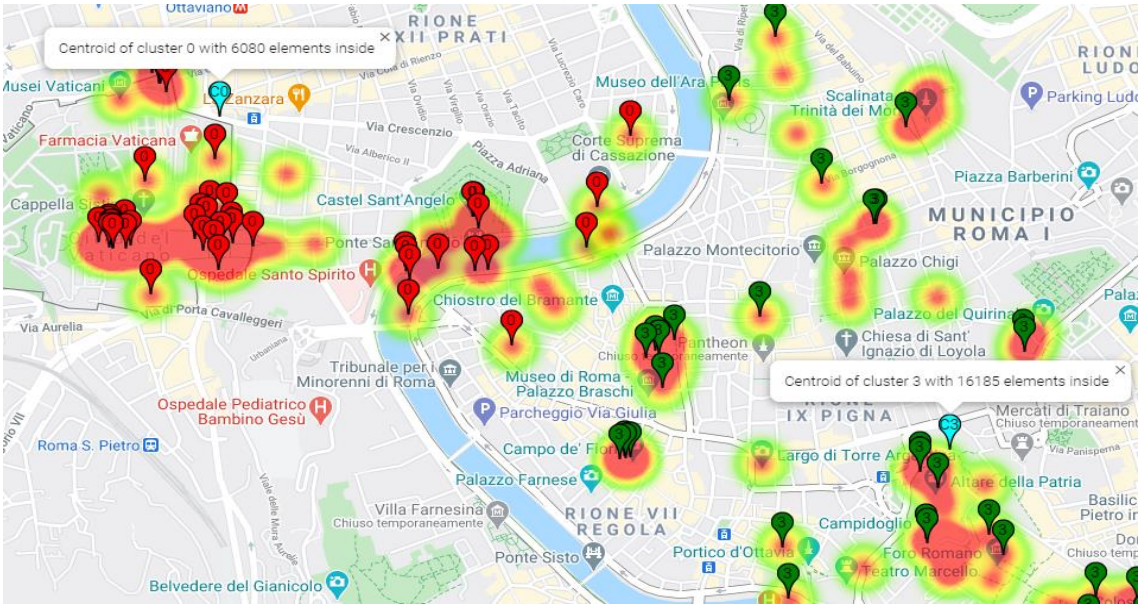
Il risultato evince come sia possibile, calibrando opportunamente i valori in input ed il numero di post visualizzati su mappa per ognuno dei top utenti, tracciare dei veri e propri percorsi di attrazioni turistiche più frequentemente visitate in maniera consecutiva nel tempo. Questo può essere utile al fine di migliorare o introdurre determinati servizi di *tour* o visite guidate all'interno della città.



2.2.6: Il metodo – kmean_heatmap

```
def kmeans_heatmap(sdf, k=5, np=100)
```

La seguente interrogazione 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 in input 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.



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(data)
```

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(k).setSeed(42).setInitMode('k-means||')
```

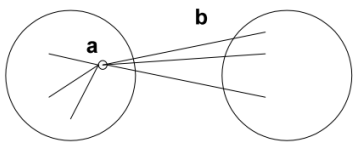
4. È stato quindi possibile effettuare l'addestramento del modello così ottenuto, per poi andarlo ad applicare sul *DataFrame* in maniera tale da associare ad ogni elemento il *cluster* a cui fa riferimento.

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

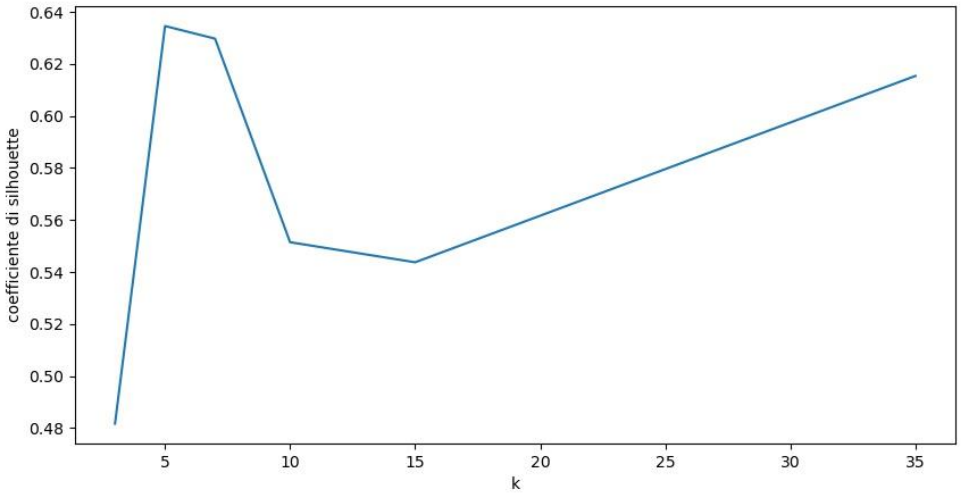
Viene inoltre calcolato, grazie all'impiego di un oggetto di tipo ***ClusteringEvaluetor***, il *coefficiente di silhouette*, ossia una misura di validità della clusterizzazione effettuata; tale informazione viene ovviamente restituita all'utente in maniera tale da avere un indicazione sulla bontà del 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**:**

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



Facendo sempre riferimento all'uso di algoritmi di *data mining*, viene inoltre restituito all'utente, nella sezione dell'interfaccia relativa alla query finora trattata, un'analisi grafica qualitativa sui valori del parametro '*K*': come si evince il risultato più significativo (sempre in termini di coefficiente di silhouette) si ha con un numero di centroidi pari a 5, mentre all'aumentare di questi si nota come aumenta anche la bontà della clusterizzazione. Questo però è abbastanza prevedibile e poco significativo, poiché all'aumentare dei centroidi è normale che i coefficienti di valutazione vadano a restituire valori sempre più alti tendenti ad uno, risulta perciò essere un risultato ingannevole.

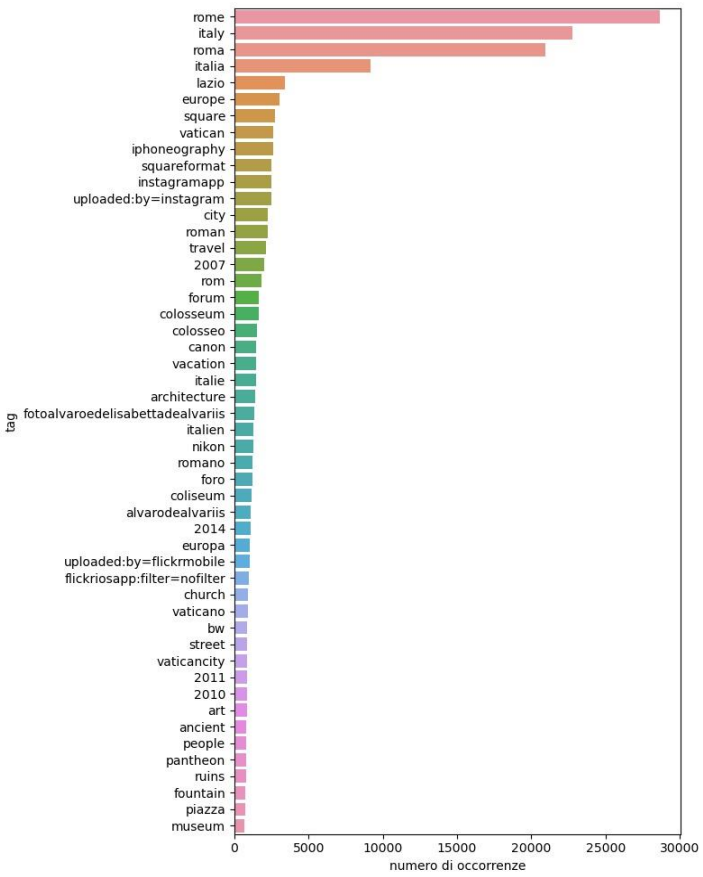


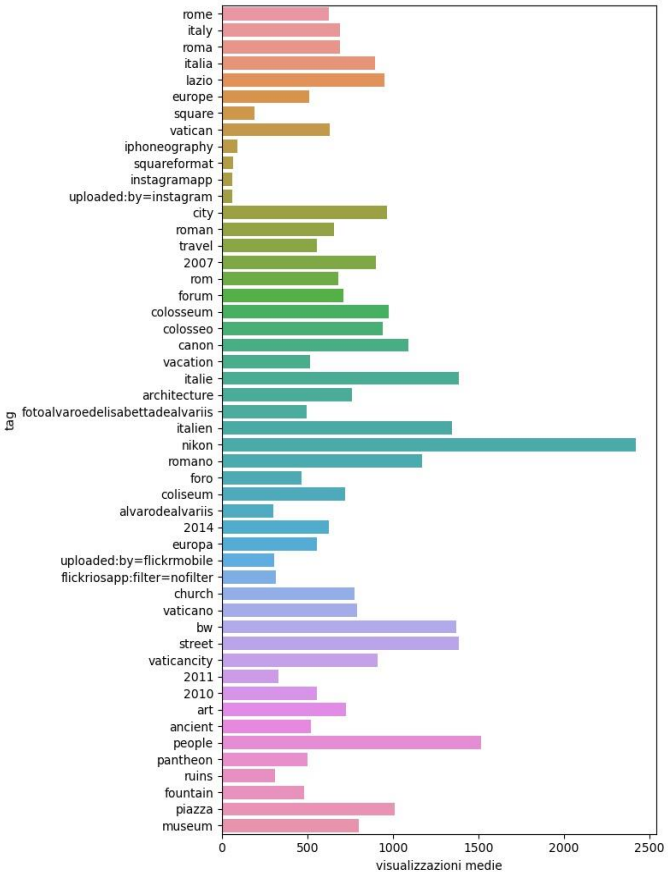
2.2.7: Il metodo – flattened_tags_count_mean

```
def flattened_tags_count_mean(sdf, 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. 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

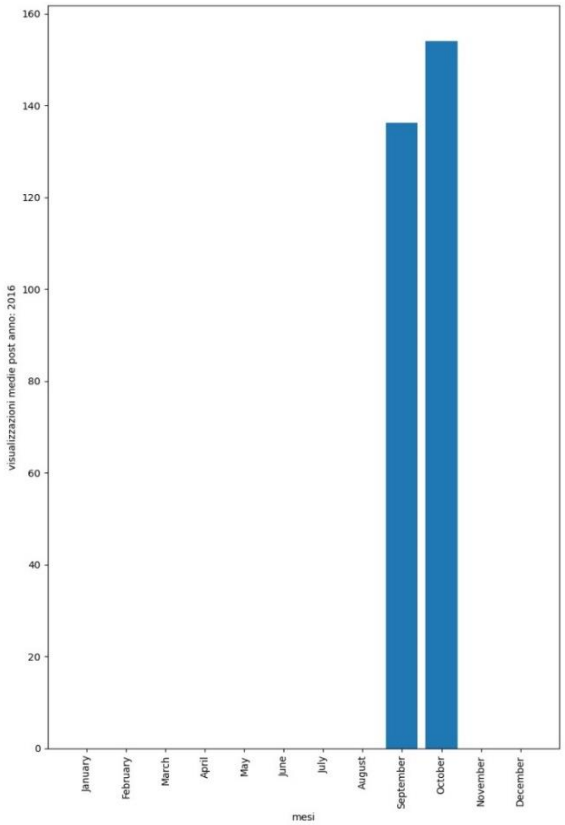
```
data = sdf.groupby('tags').agg(count('tags').alias('count'),
                               mean('views').alias('mean_views'))
tag = data.select(explode('tags').alias('tag'), 'count', 'mean_views')
tag = tag.groupBy('tag').agg(sum('count').alias('count'),
                             mean('mean_views').alias('mean_views'))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

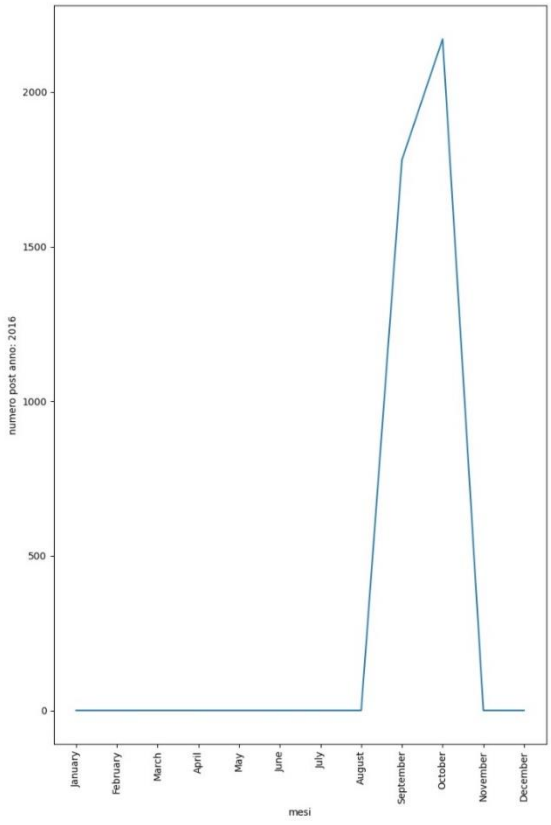
tag, un DataFrame con all'interno nuove righe in cui compare tale valore come elemento singolo. Infine si esegue nuovamente il raggruppamento, ma questa volta su tale colonna risultante, andando ad effettuare la somma delle occorrenze precedentemente calcolate, ed un ulteriore calcolo delle visualizzazioni medie, sempre impiegando il risultato precedente.

2.2.8: Il metodo – year_stats

```
def year_stats(sdf, year)
```

L'interrogazione restituisce diverse informazioni relative allo specifico anno di interesse dell'utente, in particolare permette di conoscere il numero di post per mese condivisi nell'anno e le visualizzazioni medie che questi hanno ottenuto. Può essere interessante al fine di monitorare, tramite i risultati ottenuti, l'affluenza turistica che si ha nella città e soprattutto i mesi in cui questa risulta essere più marcata, in modo da poter attuare politiche che ne permettano lo sviluppo avendo informazioni sui periodi dell'anno più popolari. Di seguito si mostra un esempio di risultato.





Al fine dell'esecuzione di tale *query*, è stato necessario l'implementazione di due specifiche '*user defined function*' tramite il meccanismo dei *decorator Python*. Infatti, essendo la colonna '*dateTaken'* un oggetto di tipo '*datetime'*, è stato necessario andare a costruire un nuovo attributo tramite estrazione del parametro '*year'* contenuto in tale oggetto.

```
@f.udf
def udf_year(dateTaken):
```

Tale funzione sostanzialmente prende in input un' intera colonna di riferimento, andando ad eseguire i passi definiti per ogni elemento in essa contenuto, per poterla applicare ed aggiungere l'attributo risultante al *dataset* di partenza, è stato necessario l'utilizzo della funzione *Spark* '*withColumn'*.

```
data = sdf.withColumn('year', udf_year('dateTaken'))
```

Analogamente ai passaggi precedentemente effettuati, con lo scopo di estrarre e proporre una visualizzazione non numerica ma testuale dei differenti mesi, è stata definita ed applicata nello stesso modo un ulteriore *UDF*.

2.2.9: Il metodo – owner_overview

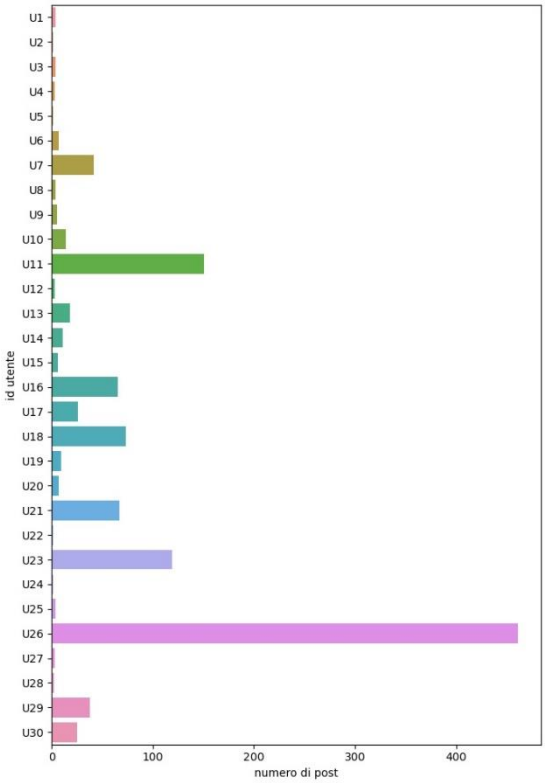
```
def owner_overview(sdf, n=20)
```

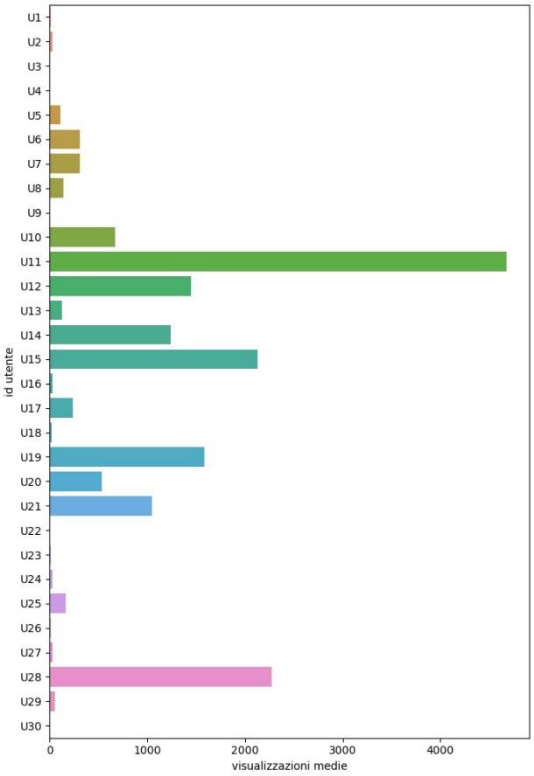
L'ultima interrogazione implementata permette la restituzione di varie info relative agli utenti attivi campionati in base al valore in input, ossia coloro i quali hanno condiviso contenuti sulla piattaforma recentemente. Queste informazioni risultano essere:

- 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.

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

- Grafico a barre che mostra il numero totale di post per tali utenti.
- Grafico a barre che mostra le visualizzazione medie per post condiviso.





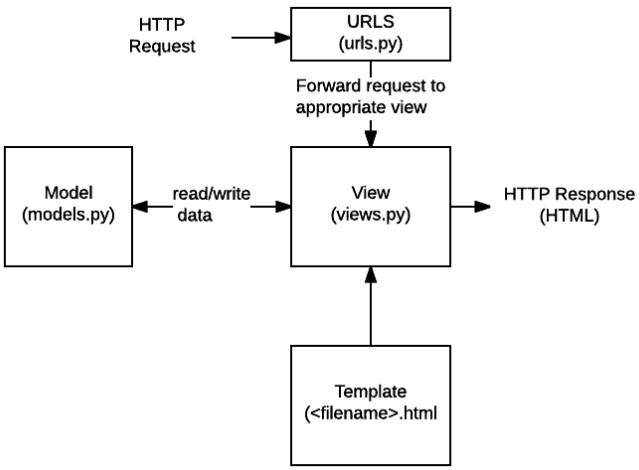
Per ottenere tutto ciò è stato necessario andare ad effettuare un ordinamento delle righe del *dataset* secondo l'attributo '*dateTaken'*, applicando questo su una '*Window' Spark*. È stato quindi possibile lavorare su tutte le righe del dataset operando solo sulla colonna specificata dalla finestra,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

```
user = user.select("*").withColumn('id', row_number().over(Window.orderBy(desc('lastPost'))))
```

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, ossia '*flattened_tags_count_mean'*.

Capitolo 3: Modulo Frontend

3.1: Framework Django e Componenti Utilizzati



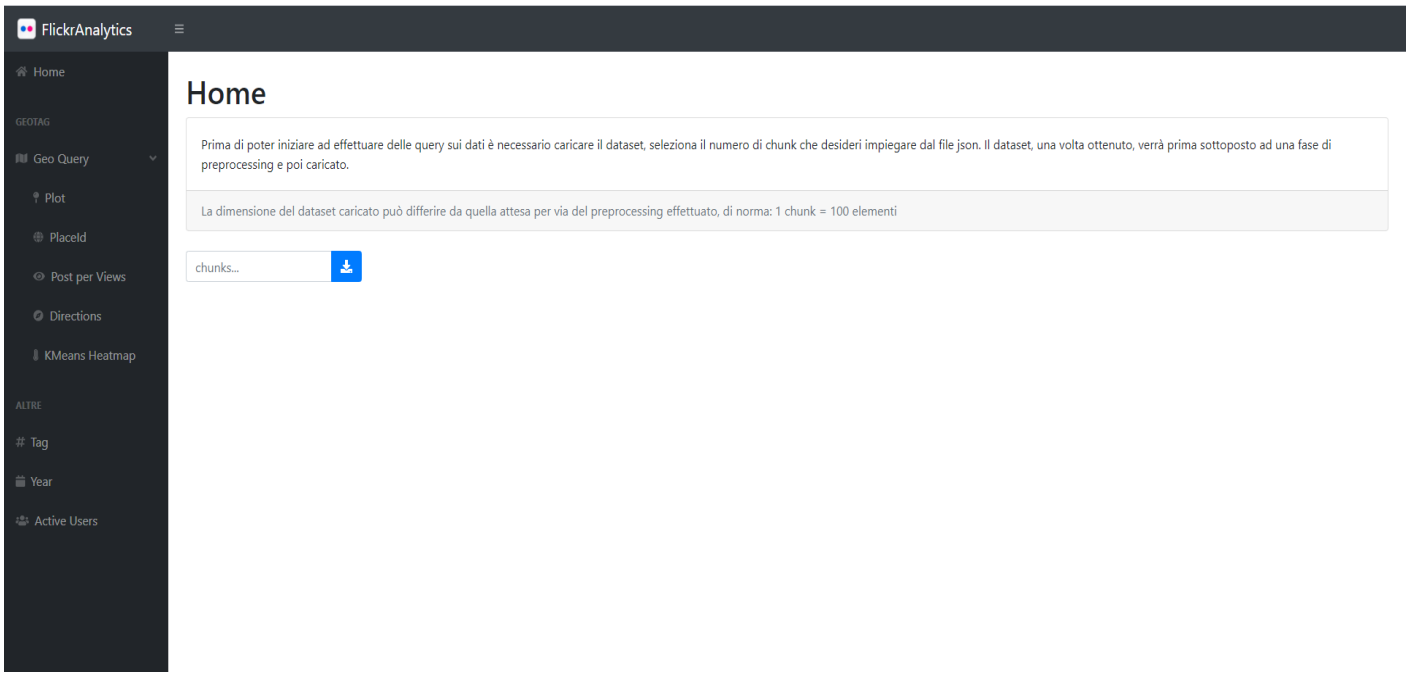
Django è un *framework Python open source* per lo sviluppo di applicazioni web che permette di gestire in maniera automatica la costruzione dell'impalcatura sottostante, in modo tale da slegare lo sviluppatore da tale onere, potendosi concentrare sulla scrittura dell'app e sulla logica implementativa. 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.

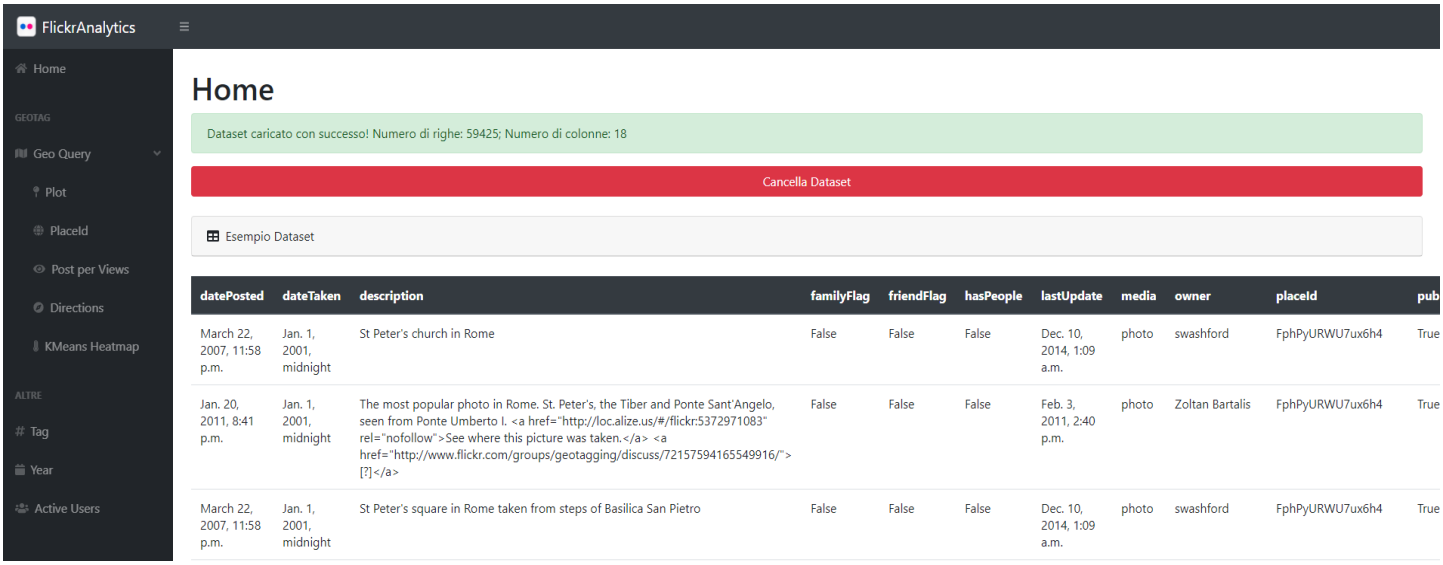
- **Model**: componente che gestisce l'accoppiamento tra oggetti *Pyhton* ed elementi all'interno del *database*, siccome la porzione *backend* del sistema *FlickrAnalytics* è totalmente sviluppato in *Spark* non è stata necessaria l'implementazione di tale modulo.
- **Template**: sostanzialmente fa 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 *FlickrAnalytics* 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*.

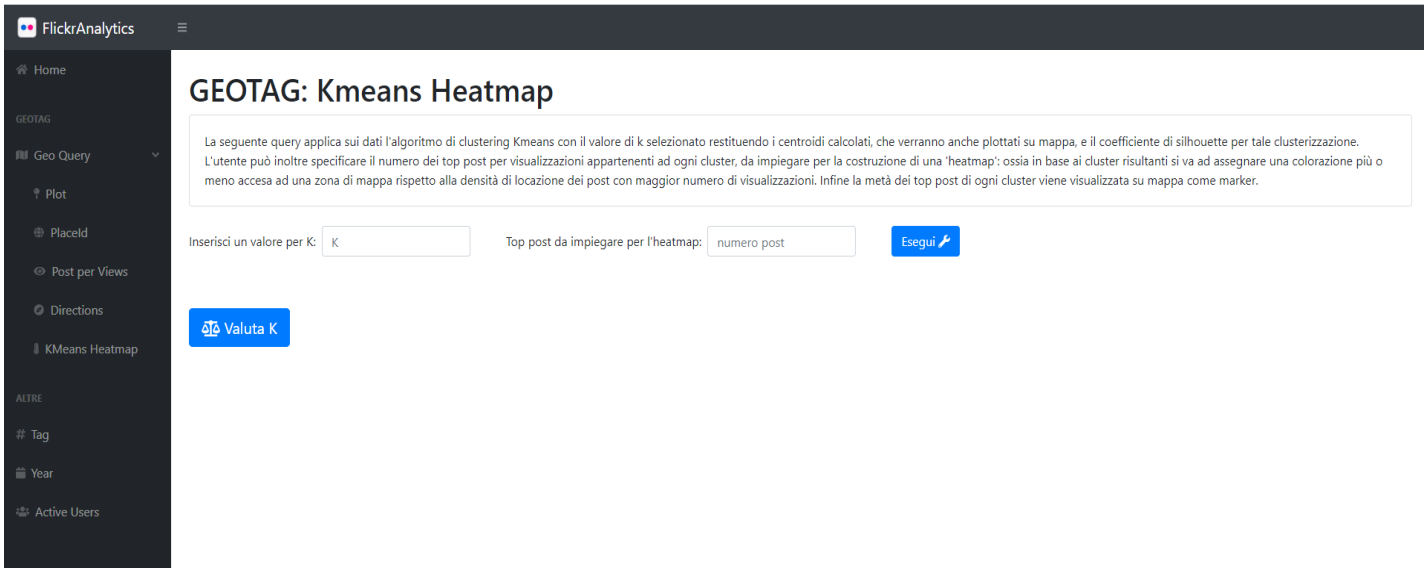
3.2: FlickrAnalytics GUI



L'interfaccia grafica di riferimento per la pagina iniziale si può osservare nell'esempio soprastante, all'avvio dell'applicativo sarà richiesto all'utente di selezionare il numero di *chunk* desiderati, in maniera tale da poter effettuare il caricamento del dataset sulla piattaforma web.

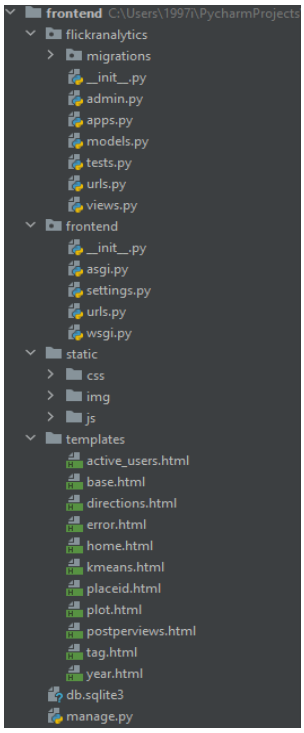


Una volta fatto ciò, viene mostrata una panoramica in forma tabellare dei dati su cui si vuole lavorare, permettendo inoltre all'utente di poter 'Cancellare' tale *dataset* se, all'evenienza, volesse condurre analisi su porzioni più ampie del file *json*. Come si può osservare, la barra di navigazione alla sinistra dell'interfaccia, costituisce il modo principale affinché l'utente possa interagire con il *software*. Sono state inserite tante opzioni quante le interrogazioni disponibili sui dati, inoltre sono state raggruppate le *query* che impiegano visualizzazione dei risultati mediante *Maps* nel menu a tendina '*Geo Query*'. Di seguito si mostra un esempio di pagina web che permetta l'esecuzione di una *query*.



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 effettua il caricamento dei dati.
- 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*.



Bisogna inoltre specificare che, i risultati grafici prodotti dal 'backend' del sistema, vengono salvati automaticamente sul '*frontend* nella cartella: '\static\img\'. Questi vengono eliminati una volta terminata la sessione di lavoro da parte dell'utente.