Università degli studi di Verona
Dipartimento di Informatica

**Relazione finale de corso "Big Data":
Interrogazioni sul dataset inerente alle Verona Card ed ai dati delle stazioni meteo utilizzando PySpark**

A.A. 2021/2022

Studentessa:
Anna Dalla Vecchia - VR472021

Professore:
Damiano Carra

Indice

Capitolo 1

Spark

Apache Spark è un framework open source di elaborazione parallela sviluppato per migliorare le prestazioni delle applicazioni che processano ed analizzano Big Data. Permette infatti l'elaborazione di dati distribuiti in un tempo minore rispetto agli strumenti di elaborazione dei database tradizionali. Spark supporta diverse API in Scala, Python, Java, R oltre alla presenza di funzioni di SQL già implementate che hanno contribuito alla sua diffusione.

Un'applicazione Spark è strutturata in tre parti principali: Driver, Esecutori e un Cluster manager. Il driver è costituito dall'applicazione e da una sessione Spark, quest'ultima si occupa di suddividere l'operazione che il programma deve eseguire, in diversi compiti, che verranno successivamente gestiti da ciascun esecutore. Gli esecutori, anche detti nodi, eseguono l'attività commissionata dal driver. Il cluster manager, infine, si occupa di gestire le risorse disponibili, come viene suddiviso il lavoro ed eseguire il programma stesso.

Il principale metodo di astrazione fornito da Spark è l'RDD (Resilient Distributed Dataset), esso è essenzialmente una collezione di elementi partizionati tra i nodi del cluster che possono essere utilizzati in parallelo. Inoltre, per garantire un accesso più rapido ai dati, l'rdd permette di mantenere gli elementi nel cluster una volta caricati grazie al metodo `persist`.

Si noti che per avere una maggiore efficienza, tutte le operazioni eseguite da Spark sono *lazy*, ovvero le operazioni non sono eseguite immediatamente bensì, vengono memorizzate le trasformazioni da applicare ad un determinato dataset e vengono eseguite nel momento in cui viene richiesto il risultato, ad esempio con un print. In questo modo viene restituito al driver solo il risultato finale, senza il rallentamento del passaggio dei risultati intermedi più grandi.

Capitolo 2

Introduzione al dataset

In questo lavoro viene utilizzato un dataset ottenuto da una selezione di alcune specifiche parti del database relazionale inerente al turismo del comune di Verona. In particolare, per svolgere le interrogazioni richieste, prendiamo in considerazione le due tabelle con un numero maggiore di tuple, ovvero la tabella delle Verona Card e la tabella contenente i dati del meteo delle stazioni veronesi. Di entrambe, sono esportati i dati in formato `csv` illustrati più nel dettaglio nelle sezioni successive.

2.1 Verona Card

Questa tabella è formata da 2.416.839 righe contenenti i dati delle strisciate delle Verona Card (VC). La tabella presenta i seguenti campi:

- **attivazione** di tipo `Date` contenente la data di attivazione della VC.
- **id_vc** di tipo `Character Varying` contenente il codice identificativo alfanumerico della VC.
- **istante** di tipo `Character Varying` contente la data e l'orario di strisciata della VC nel seguente formato '06/06/2014 11:24'.
- **profilo** di tipo `Character Varying` contenente la durata della VC, queste infatti possono avere la durata di 24, 48 o 72 ore.
- **poi** di tipo `Character Varying` contenente l'identificativo del punto di interesse (POI - Point Of Interest) in cui avviene la strisciata.
- **classid** di tipo `Integer` contenente l'identificativo della strisciata proveniente direttamente dal server ceh raccoglie le presenze.

2.2 Stazioni meteorologiche

Questa tabella è formata da 21.868.825 righe contenenti i rilevamenti, ogni 20 minuti, delle stazioni meteorologiche del territorio veronese. La tabella è formata dai seguenti campi:

- **data** di tipo `Date` contenente la data della rilevazione.
- **ora** di tipo `Character Varying` contenente l'ora della rilevazione nei seguenti formati '00.00' o '00:00'.
- **t** di tipo `Numeric` contenente la temperatura rilevata.
- **ur** di tipo `Integer` non utile al fine del progetto.
- **pr** di tipo `Numeric` contenente la pressione rilevata.
- **wind** di tipo `Numeric` contenente la velocità del vento.
- **wind_dir** di tipo `Character Varying` contenente la direzione in cui soffia il vento.
- **rain** di tipo `Numeric` contenente i millimetri di pioggia rilevati.
- **dp** di tipo `Numeric` non utile al fine del progetto.
- **idstazione** di tipo `Integer` contenente il codice identificativo della stazione metereologica.

Capitolo 3

Interrogazioni

Di seguito viene illustrata l'esecuzione delle varie interrogazioni.

3.1 Preparazione dati Verona Card

Per poter eseguire le interrogazioni presentate nelle sezioni successive mediante Spark, vengono caricati i dati della Verona Card ignorando l'header del file ottenuto dall'esportazione dal database.

Essendo i dati estratti dal database provenienti direttamente dal server del comune di Verona, vi è la necessità di eseguire una "pulizia". Infatti, al fine di ottenere consistenza nei risultati delle interrogazioni che si andranno ad eseguire, è necessario rispettare ad esempio l'univocità dell'identificativo delle singole Verona card. È possibile infatti notare nella Fig. 3.1 come, all'interno della tabella Verona Card, originariamente, siano presenti più righe con profili diversi associate alla stessa VC (stesso id).

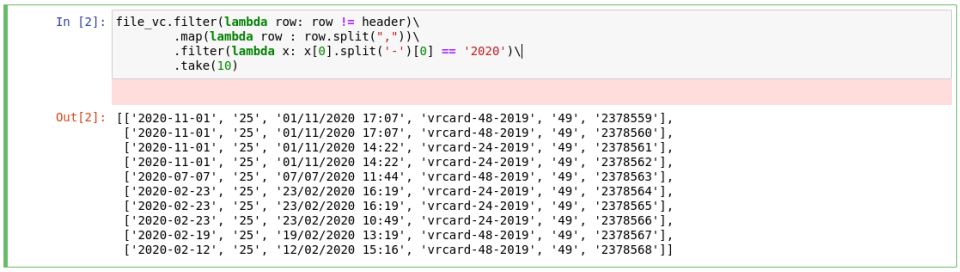

Figura 3.1: Esempio inconsistenza identificativo Verona Card

Per la consistenza delle statistiche, non consideriamo tali tuple in quanto, mancando un identificativo consistente, non è possibile risalire al collegamento tra le varie righe della tabella. A seguito di queste considerazioni, viene eseguita una "pulizia" della tabella mediante il codice presentato nel

Listing 3.1. Oltre al `filter` per selezionare dei dati con `id_vc` consistente viene eseguito un `map` per eliminare l'header del file ed infine un altro `map` per eliminare i dati che non verranno utilizzati nelle interrogazioni.

```
base_dir = 'dataset/'
filename_vc = base_dir + 'veronaCard.csv'
file_vc = sc.textFile(filename_vc)
header = file_vc.first()
data_vc = file_vc.filter(lambda row: row != header)\
            .map(lambda row : row.split(","))\
            .filter(lambda x: x[0].split('-')[0] != '2020')\
            .map(lambda x: (x[0], x[1], x[2], x[4]))

# data, id_vc, istante, poi
#('2017-11-29', '0435803A9C4C81', '29/11/2017 12:07', '61')

data_vc.persist()
```

Listing 3.1: Caricamento dei dati delle Verona Card

3.2 Q1

> Per ogni Verona Card, il numero di punti di interesse (Points of Interest, PoI) visitati, da cui costruire la sua distribuzione (l'asse x contiene il numero di PoI, l'asse y la percentuale di Verona Card usate per quel numero di PoI).

Per poter eseguire questa interrogazione l'idea è quella di estrarre dall'rdd `data_vc`, ottenuto dalla "pulizia" dei dati presentata nella sezione precedente, solo le informazioni necessarie ed utilizzarle come chiave per il conteggio. Visto il problema segnalato nel paragrafo precedente (Fig. 3.1), prima di selezionare solo l'identificativo della VC come chiave, viene eseguito un ulteriore controllo illustrato in Fig. 3.2. In `no_attivazione` è presente il numero di righe selezionate prendendo come chiave solo l'identificativo, in `si_attivazione` la chiave è composta dall'identificativo e dall'attivazione. Essendo che le due selezioni non ottengono lo stesso numero di righe vuol dire che lo stesso `id_vc` è presente con date di attivazione diverse (un identificativo viene utilizzato più volte nel medesimo arco temporale). Per tali ragioni, si utilizzerà come chiave la coppia `id_vc`, `attivazione`.

Per l'esecuzione dell'interrogazione, viene creata la coppia chiave valore necessaria per contare quanti PoI ogni VC ha visitato. In particolare, per contare quante volte ogni VC viene usata è possibile utilizzare il metodo `map` che permette di assegnare il valore numerico 1 ad ogni chiave (`id_vc`, `attivazione`) trovata nell'rdd. Quindi, mediante il metodo `reduceByKey`, vengono sommate le occorrenze della stessa chiave. Il risultato parziale viene mostrato in Fig. 3.3.

```
In [3]: no_attivazione = data_vc.map(lambda x: (x[1])).distinct().count()
        si_attivazione  = data_vc.map(lambda x: (x[1],x[0])).distinct().count()

        print('Gli id_vc sono univoci? ')
        if no_attivazione == si_attivazione:
              print('True')
        else:
              print('\tNO\ncon attivazione:',si_attivazione, '\nsenza attivazione:',no_attivazione)

        [Stage 6:================================================>          (4 + 1) / 5]

        Gli id_vc sono univoci?
               NO
        con attivazione: 671026
        senza attivazione: 482801
```

Figura 3.2: Controllo identificativo Verona Card come chiave

```
Out[4]: [(('04C5653A9C4C80', '2017-10-05'), 5),
         (('045F673A9C4C80', '2017-10-05'), 5),
         (('04039E429C4C81', '2017-12-28'), 6),
         (('04C58A429C4C80', '2017-10-31'), 4),
         (('04B30A429C4C84', '2017-11-29'), 6)]
```

Figura 3.3: Risultato intermedio salvato nella variabile `visit_vc` in cui il primo valore rappresenta la coppia identificativo della VC, data di attivazione ed il secondo il numero di PoI visitati

Infine, per poter contare quante VC sono state utilizzate lo stesso numero di volte, viene sfruttata nuovamente la tecnica `MapReduce`. In particolare viene messa come chiave il numero di PoI visitati da ogni VC, ovvero il valore numerico rappresentato in Fig. 3.3. Si procede in modo analogo a quanto fatto precedentemente con l'aggiunta di: un `map` in cui viene calcolata la percentuale, un `sortByKey` per ordinare l'rdd secondo la chiave e infine di un `collect`, al fine di semplificare la visualizzazione grafica riportata in seguito. La parte di codice appena descritta può esser visualizzata nel Listing 3.2.

```
visit_vc = data_vc.map(lambda x: ((x[1],x[0]), 1))\
                  .reduceByKey(lambda a,b: a+b)

tot_vc = visit_vc.count() #numero totale di verona card

visit_vc_same = visit_vc.map(lambda x: (x[1],1))\
                  .reduceByKey(lambda a,b: a+b)\
                  .map(lambda x: (x[0],x[1]/tot_vc * 100))\
                  .sortByKey()\
                  .collect()
```

Listing 3.2: Esecuzione Q1

Nel Listing 3.3 viene riportato il codice per la stampa del grafico. Sull'asse delle x viene riportato il numero di POI visitati, sull'asse delle y viene riportata la percentuale di VC usate per quel numero di PoI. Di seguito in Fig. 3.4 viene mostrato il risultato finale.

```
import matplotlib.pyplot as plt

```

```
perc_vc = [ x[1] for x in visit_vc_same]
visit_number = [x[0] for x in visit_vc_same]

plt.suptitle('PoI visitati da ogni VC')
plt.xlabel('# visite per verona card')
plt.ylabel('Percentuale')

plt.grid(ls = '-')
plt.plot(visit_number, perc_vc)
for x,y in zip(visit_number, perc_vc):
    plt.text(x,y, str(round(y,2)))

plt.show()
```

Listing 3.3: Stampa del grafico di Q1

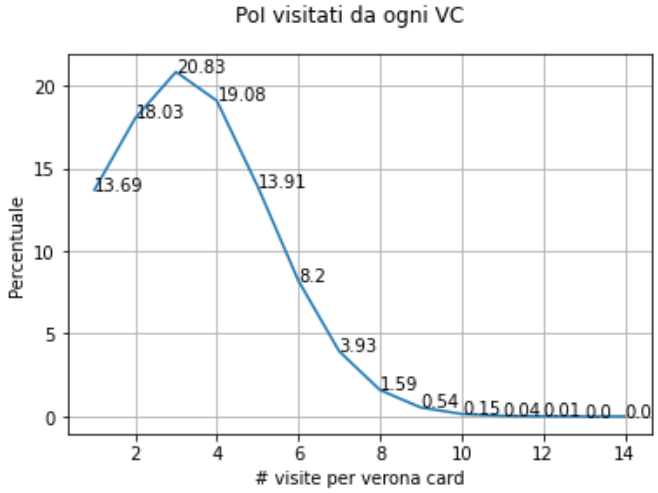

Figura 3.4: Risultato dell'interrogazione Q1

3.3 Q2

> Per ogni PoI, la distribuzione del numero di visite giornaliere (l'asse x indica il range di visitatori, l'asse y la percentuale di giorni che quel PoI ha ricevuto un numero di visitatori all'interno di quel range).

Per risolvere questa interrogazione, è possibile prima di tutto andare a calcolare il numero di presenze giornaliere per ogni PoI. Successivamente, si va ad identificare quanti giorni presentano lo stesso numero di presenze per il medesimo PoI. Poiché il numero di affluenze è molto vario anche per lo stesso PoI, se si facesse un `reduceByKey` sul numero puntuale di visitatori

giornalieri, si avrebbe che ogni affluenza ha una percentuale prossima allo zero. Infatti, la probabilità di avere lo stesso numero di affluenze in due giorni diversi per il medesimo PoI, e supponendo che il numero massimo di affluenze giornaliere sia limitato a N_MAX sarebbe:

$$\mathcal{P}[\text{2 giorni con la stessa affluenza}] = \frac{1}{N\_MAX} * \frac{1}{N\_MAX} = \frac{1}{N\_MAX^2}$$

se $N\_MAX = 100$ avremmo una probabilità uguale a 1×10^{-3}. Contando però che non è possibile stabile a priori il valore massimo di visitatori giornalieri, la probabilità e di conseguenza le percentuali, si abbassano ulteriormente. Per queste considerazioni e al fine di ottenere un'analisi statistica consistente, si è deciso di suddividere il range di visitatori giornalieri in un certo numero di bucket prestabiliti (in questo caso di studio 15). L'assegnamento ad ogni bucket avviene secondo la funzione riportata nel Listings 3.4 nella quale, dato il numero di visitatori, il massimo ed il minimo numero di visitatori per un determinato PoI, viene restituito il bucket di appartenenza.

```
1 def getBucket(visitors, max_v, min_v):
2     if max_v == visitors:
3         return number_bucket
4
5     bucket = round((max_v - min_v) / number_bucket,0)
6     return int((visitors - min_v) / bucket) + 1
```

Listing 3.4: Funzione per la suddivisione in bucket

Per l'esecuzione dell'interrogazione, viene quindi creata la coppia chiave-valore necessaria a contare quante visite giornaliere ha avuto un determinato PoI. In particolare, la chiave che viene creata nel `map` è composta dal numero identificativo del PoI e dal giorno in cui è stata registrata la presenza. A questa chiave viene mappato il valore numerico 1 necessario al `reduceByKey` per sommare il numero di presenze. La porzione di codice appena descritta è riportata nel Listing 3.5 ed un esempio del risultato intermedio viene invece presentato in Fig. 3.5.

```
1 daily_visitors = data_vc.map(lambda x: ((x[3],x[2].split(' ')
      [0]), 1))\
2                         .reduceByKey(lambda a,b: a+b)
```

Listing 3.5: Creazione rdd in cui ad ogni giorno di ogni PoI viene assegnata l'affluenza totale

Per poter rappresentare l'affluenza di ogni PoI, è necessario salvare in una lista gli identificativi dei PoI presenti. Mediante un ciclo for è possibile filtrare, grazie al metodo `filter` di PySpark, solo i dati riguardanti il PoI in analisi. Attraverso il metodo `map` successivamente, si snellisce il contenuto dell'rdd andando a togliere l'informazione dell'identificativo del PoI.

Sucessivamente viene salvato il massimo ed il minimo numero di visite giornaliere del PoI necessarie alla funzione `get_bucket`, descritta in Lisi-

```
Out[9]: [(('61', '19/11/2017'), 157),
         (('61', '04/10/2017'), 201),
         (('61', '27/11/2017'), 60),
         (('61', '25/10/2017'), 136),
         (('61', '24/10/2017'), 105)]
```

Figura 3.5: Risultato parziale in cui il primo valore rappresenta la coppia identificativo del PoI, data della registrazione della presenza ed il secondo il numero di visitatori

tings 3.4, per poter calcolare il bucket di appartenenza di ogni singolo giorno. Viene inoltre memorizzato il totale di giorni per poter poi calcolare la percentuale di giorni con lo stesso numero di visite.

Si procede quindi con il calcolo delle affluenze, con una procedura simile a quanto utilizzato per l'interrogazione della sottosezione 3.2. Attraverso il `map` si crea la coppia chiave-valore (bucket di appartenenza, 1) necessaria al `reduceByKey` per poter trovare il numero di giorni con un numero di affluenze appartenenti allo stesso bucket. Si calcola quindi, attraverso un altro `map`, la percentuale di giorni con un numero di visite nello stesso range. Infine, con il `sortByKey` e con il `collect` viene creata la lista ordinata utilizzata per disegnare il grafico del risultato. La porzione di codice appena descritta è riportata nel Listing 3.6 e il grafico di un PoI si può osservare in Fig. 3.6, per gli altri grafici rimando al notebook inviato in allegato a questa relazione (reperibile anche al seguente link `https://github.com/4nnina/bigData_project/blob/main/Progetto.ipynb`).

```
1  import matplotlib.pyplot as plt
2
3  list_poi = data_vc.map(lambda x: x[3]).distinct().collect()
4
5  for poi in list_poi:
6      count_daily = daily_visitors.filter(lambda x: x[0][0] ==
       poi)\
7              .map(lambda x: (x[0][1],x[1]))
8
9      max_visitors = count_daily.map(lambda x: x[1]).max()
10     min_visitors = count_daily.map(lambda x: x[1]).min()
11     tot_visitors = count_daily.count()
12
13     group_same_count_visitors = count_daily.map(lambda x: (
       getBucket(x[1],max_visitors,min_visitors), 1))\
14             .reduceByKey(lambda a,b: a+b)\
15             .map(lambda x: (x[0],x[1] / tot_visitors * 100))\
16             .sortByKey()\
17             .collect()
18
19     bucket = max(int(max_visitors/number_bucket), 1)
20     visitors = [ x[1] for x in group_same_count_visitors]
21     bucket_list = [ x[0]*bucket  for x in group_same_count
       _visitors]
```

```
22
23      x_labels = range(0, max_visitors,bucket)
24      plt.xticks(x_labels)
25
26      plt.suptitle('Distribuzione visite PoI numero {}'.format(
        poi))
27      plt.xlabel('Range visitatori')
28      plt.ylabel('Percentuale di giorni')
29      plt.grid()
30      plt.plot(bucket_list,visitors)
31      for x,y in zip(bucket_list,visitors):
32          plt.text(x,y, str(round(y,2)))
33      plt.show()
```

Listing 3.6: Esecuzione e produzione dei grafici di Q2

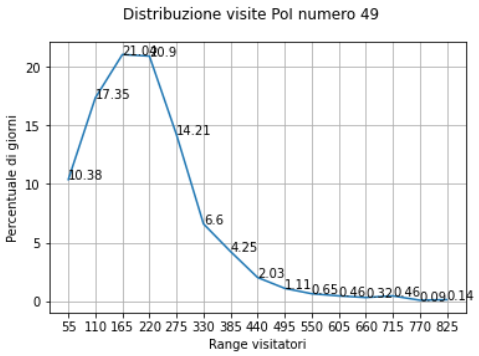

Figura 3.6: Risultato interrogazione Q2 per il PoI 49, Arena di Verona

3.4 Preparazione dati stazioni meteorologiche

Analogamente a quanto fatto per il dataset delle VC, vengono in questa fase caricati i dati del meteo. Come visto in precedenza il dataset inerente alle stazioni meteorologiche è provvisto di informazioni come la pressione, il vento e la sua direzione non necessarie al fine di svolgere le interrogazioni trattate in queto progetto e pertanto eliminate dal dataset. Vengono tenute solo la data e l'ora (i minuti non sono rilevanti e vengono scartati) del rilevamento, la temperatura ed infine i millimetri di pioggia già convertiti in `float`. Come per le VC, i dati di questo dataset sono ricavati direttamente dai dati inviati dalle centraline meteorologiche veronesi e presentano delle anomalie. Si può notare in Fig. 3.7 l'orario con valori non validi e in Fig. 3.8 millimetri di pioggia negativi e valori di temperatura atipiche.

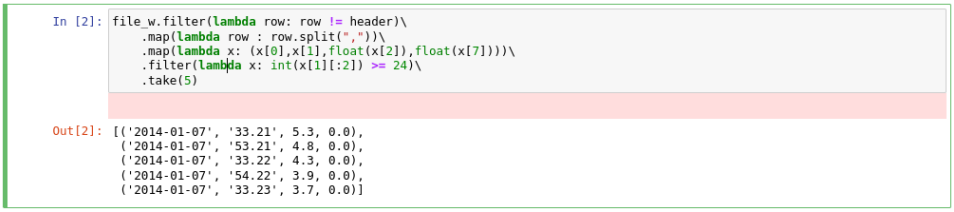

Figura 3.7: Esempi orari non validi

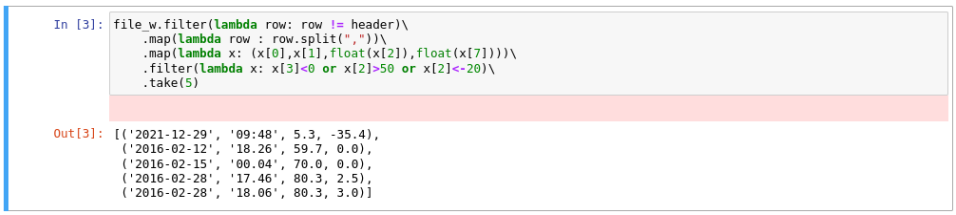

Figura 3.8: Esempio millimetri di pioggia negativi e temperature anomale

Sono eseguiti diversi `filter` per poter risolvere queste anomalie prima di rendere consistente l'rdd. Il codice per il caricamento e per la pulizia dei dati è riportato nel Listing 3.7.

```
base_dir = 'dataset/'

filename_w = base_dir+"meteo.csv"
file_w = sc.textFile(filename_w)

header = file_w.first()

data_weather = file_w.filter(lambda row: row != header)\
        .map(lambda row : row.split(","))\
        .map(lambda x: (x[0],int(x[1][:2]),float(x[2]),float(x
    [7])))

data_w = data_weather.filter(lambda x: x[1] < 24)\
             .filter(lambda x: x[3] > 0)\
             .filter(lambda x: x[2] < 50 and x[2] > -20)

# data, ora, t, rain
#('2016-02-08', 11, 10.7, 10.3)

data_w.persist()
```

Listing 3.7: Caricamento dai delle stazioni meteo veronesi previo filtraggio dei dati

Le interrogazioni 3.5 e 3.6 avendo entrambe il parametro X e Y rispettivamente personalizzabile sono state implementate come due funzioni da poter chiamare e modificare attraverso il loro argomento.

3.5 C1

> Giorni in cui ha piovuto per almeno X ore / giorni in cui ha piovuto meno di X ore.

Per risolvere questa consegna, che richiede una selezione sul numero di ore di pioggia in un giorno, si inizia calcolando la media di millimetri di pioggia nelle varie ore della giornata. Questo viene svolto grazie ad un `map` per creare la coppia chiave-valore in cui la chiave è formata dalla data e dall'ora, e il valore dai millimetri di poggia. A seguire, grazie al `reduceByKey`, viene eseguita la media dei valori per ogni chiave definita.

Poichè gli effettivi millimetri di pioggia non sono di nostro interesse, per contare le ore di pioggia in una giornata, si è deciso di utilizzare un ulteriore `reduceByKey`. In particolare viene applicato questo metodo ad un rdd in cui tramite il `map`, fromiamo una chiave costituita dal giorno e come valore il numero 1 se la media dei millimetri di pioggia oraria risulta > 0, il valore numerico 0 altrimenti. In questo modo, otteniamo un rdd con il giorno e il numero di ore in cui è stata rilevata pioggia. Un esempio del risultato è riportato in Fig. 3.9 e la parte di codice corrispondente è riportata nel Listing 3.8.

```
rain_hour = data_w.map(lambda x: ((x[0],x[1]),x[3]) )\
                .reduceByKey(lambda a,b : (a+b) / 2)\
                .map(lambda x: (x[0][0], 1 if x[1]>0 else 0))\
                .reduceByKey(lambda a,b: a+b)
```

Listing 3.8: Codice per il calcolo delle ore di pioggia in un giorno

```
Out[16]: [('2016-01-11', 24),
          ('2016-01-18', 24),
          ('2016-02-12', 24),
          ('2016-06-02', 24),
          ('2016-08-05', 24),
          ('2016-08-16', 24),
          ('2016-09-30', 24),
          ('2016-10-03', 24),
          ('2016-10-29', 24),
          ('2016-11-21', 24)]
```

Figura 3.9: Risultato del Listing 3.8, ore di pioggia in un giorno

Guardando Fig. 3.9 si può notare come le ore di pioggia in tutti i giorni stampati siano 24 ovvero, secondo il nostro risultato, ad ogni ora ci sarebbe stata una perturbazione. Se si prova a fare una media delle ore di pioggia durante i giorni dell'anno essa risulta incredibilmente alta (Fig. 3.10).

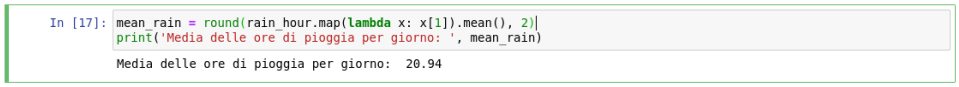

Figura 3.10: Media ore di pioggia giornaliere

Questa anomalia è causata dalla selezione sull'ora di pioggia (Listing 3.8, riga 3). Infatti, basta cambiare il valore di threshold di pioggia per ottenere un risultato più realistico. Data la sensibilità differente dei vari sensori delle stazioni meteorologiche, esistono varie classificazioni per le perturbazioni. In particolare possiamo riassumerle nei seguenti punti:

- 1 - 2 mm/h pioggia debole
- 2 - 4 mm/h pioggia leggera
- 4 - 6 mm/h pioggia moderata
- 6 - 10 mm/h pioggia forte
- 10 - 30 mm/h rovescio
- $>$ 30 mm/h nubifragio

A seguito di questa considerazione, si può concludere che fino ad 1 mm di pioggia in un'ora non è considerata una perturbazione. Perciò, la soglia per decretare se in un'ora ha piovuto o meno, non sarà più 0 ma sarà alzata ad 1 (Fig. 4.6).

Si procede definendo la funzione `splitRain` che prende come argomenti l'rdd e il discriminante X per le ore di pioggia e restituisce una tupla contenente due rdd, uno contenente i giorni in cui ha piovuto più di X ore ed uno con i giorni in cui ha piovuto meno di X ore. In Listing 3.9 è possibile vederne il codice.

```
1 def splitRain(rdd, hour_rain):
2     over_X = rdd.filter(lambda x: x[1] >= hour_rain)
3     under_X = rdd.filter(lambda x: x[1] < hour_rain)
4
5     return (over_X, under_X)
```

Listing 3.9: Funzione per l'esecuzione di C1

A seguito in Listing 3.10 viene riportato un esempio di esecuzione in cui la X è la media delle ore di pioggia giornaliere calcolato con il discriminante dei pioggia posto a 1mm. Il risultato del codice si può osservare in Fig. 3.12.

```
In [17]: rain_hour = data_w.map(lambda x: ((x[0],x[1]),x[3]) )\
                    .reduceByKey(lambda a,b : (a+b) / 2)\
                    .map(lambda x: (x[0][0], 1 if x[1]>=1 else 0))\
                    .reduceByKey(lambda a,b: a+b)
         rain_hour.take(10)

Out[17]: [('2016-01-11', 19),
          ('2016-01-18', 23),
          ('2016-02-12', 19),
          ('2016-06-02', 19),
          ('2016-08-05', 24),
          ('2016-08-16', 11),
          ('2016-09-30', 0),
          ('2016-10-03', 11),
          ('2016-10-29', 19),
          ('2016-11-21', 20)]

In [18]: mean_rain = round(rain_hour.map(lambda x: x[1]).mean(), 2)
         print('Media delle ore di pioggia per giorno: ', mean_rain)

         Media delle ore di pioggia per giorno:  11.31
```

Figura 3.11: Ore di pioggia giornaliere a seguito della modifica della threshold

```
over_X, under_X = splitRain(rain_hour, mean_rain)

import matplotlib.pyplot as plt

labels = 'Rain\'s hours >= {}'.format(mean_rain), 'Rain\'s
    hours < {}'.format(mean_rain)
sizes = [over_X.count(),under_X.count()]

plt.pie(sizes, labels=labels, autopct='%1.1f%%',
        shadow=True, startangle=90)

plt.show()
```

Listing 3.10: Esecuzione C1 con X uguale alla media ore di pioggia giornaliere

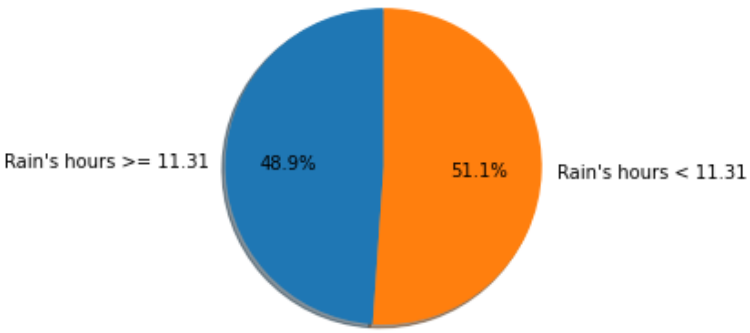

Figura 3.12: Grafico rappresentante il risultato di C1 con X uguale alla media delle ore di pioggia per giorno

3.6 C2

> Giorni in cui la temperatura media era superiore a Y / giorni in cui era inferiore a Y.

In questa interrogazione si chiede di ottenere l'elenco dei giorni in cui si ha avuto una temperatura maggiore di una soglia Y e l'elenco dei giorni in cui è minore di Y. Si possono notare varie analogie con l'interrogazione precedente in cui veniva chiesto di dividere i giorni secondo le ore di pioggia.

Si procede quindi con il calcolo della temperatura media giornaliera. Inizialmente è calcolata la temperatura media oraria grazie ad un `mapReduce` in cui la chiave è formata dal giorno e dall'ora, mentre il valore è rappresentato dalla temperatura. Successivamente prendendo come chiave solo il giorno viene calcolata la media della temperatura giornaliera. Il codice è presentato nel Listing 3.11, le righe 5 e 6 della porzione di codice presentata calcolano la media giornaliera della temperatura che verrà utilizzata in seguito come discriminante per l'esecuzione dell'interrogazione.

```
1 temp_day = data_w.map(lambda x: ((x[0],x[1]),x[2]) )\
2             .reduceByKey(lambda a,b : round((a + b)/2, 2))\
3             .map(lambda x: (x[0][0],x[1]))\
4             .reduceByKey(lambda a,b : round((a + b)/2, 2))
5 mean_temp = round(temp_day.map(lambda x: x[1]).mean(), 2)
6 print('Temperatura media giornaliera: ', mean_temp)
```

Listing 3.11: Codice per il calcolo della temperatura media oraria e del discriminante `mean_temp`

Viene quindi definita la funzione `splitTemp` che prende come argomenti l'rdd e la Y della nostra interrogazione e restituisce due rdd, uno con l'elenco dei giorni in cui c'è stata una temperatura media superiore alla soglia, l'altro con i giorni in cui è stata inferiore. In Listing 3.12 è presentato il codice.

```
1 def splitTemp(rdd, temp):
2     over_Y = rdd.filter(lambda x: x[1] >= temp)
3     under_Y = rdd.filter(lambda x: x[1] < temp)
4
5     return (over_Y, under_Y)
```

Listing 3.12: Funzione per l'esecuzione di C2

Infine, nel Lisitng 3.13, è presentata la porzione di codice in cui viene utilizzata la funzione, ponendo Y uguale alla media della temperature giornaliere, e stampato il grafico a torta (Fig. 3.13).

```
1 over_Y, under_Y = splitTemp(temp_day, mean_temp)
2
3 import matplotlib.pyplot as plt
4
5 labels = 'Temperature >= {}'.format(mean_temp), ' Temperature <
      {}'.format(mean_temp)
```

```
sizes = [over_Y.count(),under_Y.count()]

plt.pie(sizes, labels=labels, autopct='%1.1f%%',
        shadow=True, startangle=90)

plt.show()
```

Listing 3.13: Esecuzione C2 con Y uguale alla media delle temperature giornaliere

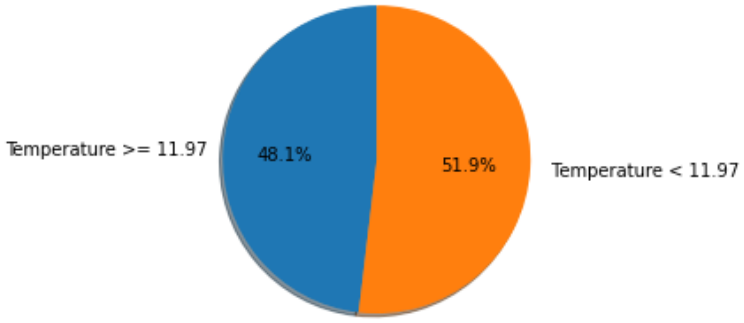

Figura 3.13: Grafico rappresentante il risultato di C2 con Y uguale alla media della temperatura giornaliera

3.7 Q1 e Q2 in base a C1

Prima di procedere con le interrogazioni successive bisogna porre attenzione al formato della data nei due dataset. Mentre per le stazioni meteorologiche il giorno è originariamente salvato nel database relazionale con il classico formato `Date` (YYYY-MM-DD), nel dataset delle VC l'istante di utilizzo della stessa è salvato in una stringa e la data si presenta nel formato DD/MM/YYYY HH:MM. Per poter rendere compatibile il confronto dei due dataset è stata quindi implementata la funzione `getDate` che estrae dall'istante della VC la data e la converte nel formato utilizzato dalla data delle stazioni meteorologiche. L'orario viene trascurato in quanto non rilevante ai fini delle interrogazioni. In Fig. 3.14 viene riportata la funzione e un esempio per maggiore chiarezza.

```
In [27]: def getDate(date):
             day, month, year = date.split(' ')[0].split('/')
             return '-'.join([year, month, day])

         print('Prima: ', data_vc.take(1))
         data_vc = data_vc.map(lambda x: (x[0], x[1], getDate(x[2]), x[3]))
         print('Dopo: ', data_vc.take(1))

         Prima:  [('2017-11-29', '0435803A9C4C81', '29/11/2017 12:07', '61')]
         Dopo:  [('2017-11-29', '0435803A9C4C81', '2017-11-29', '61')]
```

Figura 3.14: Funzione di standardizzazione data

Q1 + C1

Per ogni Verona Card, il numero di punti di interesse (Points of Interest, PoI) visitati, da cui costruire la sua distribuzione (l'asse x contiene il numero di PoI, l'asse y la percentuale di Verona Card usate per quel numero di PoI) discriminando i giorni in cui ha piovuto per almeno X ore / giorni in cui ha piovuto meno di X ore.

Per eseguire questa interrogazione viene eseguito un cambiamento rispetto all'esecuzione della query Q1 (3.2). Mentre in precedenza venivano contate le occorrenze di visite totali in cui una VC veniva utilizzata, per questa interrogazione e per la successiva (3.8) il conteggio è fatto sul numero di visite giornaliere che una VC esegue. Questo per poter permettere di applicare la discriminante della temperatura e della pioggia.

Procediamo quindi con l'illustrazione dell'esecuzione dell'interrogazione. Per prima cosa viene eseguito il conteggio del numero di strisciate giornaliere per ciascuna VC grazie ad un `map`, in cui la chiave è formata dalla data e dall'identificativo della VC, ed un `reduceByKey`. Con un successivo `map` si va a creare l'rdd per il `join` con il dataset del meteo, si va infatti a mettere come chiave solo la data e come valore la coppia identificativo VC e la somma appena eseguita. Il risultato di questa operazione è salvato nella variabile `visit_vc` di Listing 3.14 e si può vedere in Fig. 3.15.

```
Out[29]: [('2017-10-05', ('04B28C4A9C4C80', 1)),
          ('2017-10-05', ('04E279429C4C80', 5)),
          ('2017-10-05', ('045F673A9C4C80', 5)),
          ('2017-11-29', ('04952C3A9C4C80', 6)),
          ('2017-10-30', ('0438A0429C4C80', 2))]
```

Figura 3.15: Rdd in cui è presente per ogni giorno e per ogni VC quanti PoI sono stati visitati

Grazie alla funzione `splitRain` (Listing 3.9), implementata durante l'esecuzione di C1, è possibile recuperare facilmente recuperata la suddivisione in giorni in cui ha piovuto più di X ore e meno di X ore, anche in questo caso X viene posto uguale alla media giornaliera di ore di pioggia. A questo punto è possibile eseguire il `join` tra l'rdd dei giorni in cui ha piovuto di più e l'rdd del numero di PoI visitati giornalmente da ogni VC. Analogamente a quanto eseguito in precedenza viene eseguito il `join` con i giorni in cui ha piovuto meno. Per entrambi gli rdd risultanti, attraverso un primo `map` si estrae l'informazione riguardante l'identificativo della VC ed il numero di PoI visitati, con il secondo `map` si prepara l'rdd al `reduceByKey` nella quale si conta quante VC hanno visitato lo stesso numero di PoI. Si procede alla preparazione della stampa sul grafico del risultato attraverso un `map` per il calcolo della percentuale, grazie al totale salvato prima del `mapReduce`, un

`sortByKey` per l'ordinamento ed il `collect` per convertire il risultato in una lista. L'esecuzione di queste operazioni è riportata nel Listing 3.14.

```
visit_vc = data_vc.map(lambda x: ((x[2],x[1]), 1))\
                .reduceByKey(lambda a,b: a+b)\
                .map(lambda x: (x[0][0],(x[0][1],x[1])))
over_X, under_X = splitRain(rain_hour, mean_rain)

visit_vc_over = visit_vc.join(over_X)\
                .map(lambda x: x[1][0])
tot_vc_over = visit_vc_over.map(lambda x: x[0]).distinct().
    count()
list_over = visit_vc_over.map(lambda x: (x[1],1))\
            .reduceByKey(lambda a,b: a+b)\
            .map(lambda x: (x[0],x[1] / tot_vc_over * 100))\
            .sortByKey()\
            .collect()

visit_vc_under = visit_vc.join(under_X)\
                .map(lambda x: x[1][0])
tot_vc_under = visit_vc_under.map(lambda x: x[0]).distinct().
    count()
list_under = visit_vc_under.map(lambda x: (x[1],1))\
            .reduceByKey(lambda a,b: a+b)\
            .map(lambda x: (x[0],x[1] / tot_vc_under * 100))\
            .sortByKey()\
            .collect()
```

Listing 3.14: Preparazione dati VC per l'esecuzione del join ed esecuzione di Q1+C1

In Listing 3.15 è riportato il codice per la stampa su grafico, il grafico ottenuto è riportato in Fig. 3.16.

```
import matplotlib.pyplot as plt

perc_vc_over = [ x[1] for x in list_over]
perc_vc_under = [ x[1] for x in list_under]
visit_number_over = [x[0] for x in list_over]
visit_number_under = [x[0] for x in list_under]

plt.suptitle('PoI visitati da ogni VC in un giorno')
plt.xlabel('# visite per verona card')
plt.ylabel('Percentuale')

plt.grid(ls = '-')
plt.plot(visit_number_over,perc_vc_over, color='b', label= '
    rain >= X')
plt.plot(visit_number_under,perc_vc_under, color='g', label= '
    rain < X')
plt.legend()

plt.show()
```

Listing 3.15: Stampa del risultato dell'interrogazione Q1+C1

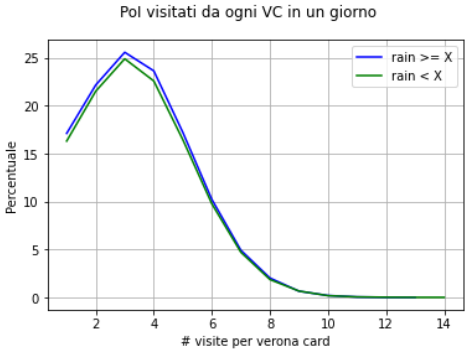

Figura 3.16: Grafico risultante dal Listing 3.15

Q2 + C1

Per ogni PoI, la distribuzione del numero di visite giornaliere (l'asse x indica il range di visitatori, l'asse y la percentuali di giorni che quel PoI ha ricevuto un numero di visitatori all'interno di quel range) discriminando i giorni in cui ha piovuto per almeno X ore / giorni in cui ha piovuto meno di X ore.

La procedura per risolvere questa interrogazione segue il ragionamento fatto per Q2 in particolare il Listing 3.6 con l'aggiunta della distinzione dei giorni in cui ha piovuto più di X ore e meno di X ore come avvenuto per l'interrogazione appena risolta (3.7) mediate il `join`. Viene ad ogni modo riportato il codice in Listing 3.16. In Fig. 3.17 viene riportato il grafico, per gli altri grafici rimando al notebook inviato in allegato a questa relazione.

```
1  import matplotlib.pyplot as plt
2
3  list_poi = data_vc.map(lambda x: x[3]).distinct().collect()
4
5  daily_visitors = data_vc.map(lambda x: ((x[3],x[2].split(' ')
       [0]), 1))\
6                  .reduceByKey(lambda a,b: a+b)
7
8  over_X, under_X = splitRain(rain_hour, mean_rain)
9
10 for poi in list_poi:
11     count_daily = daily_visitors.filter(lambda x: x[0][0] ==
       poi)\
12                 .map(lambda x: (x[0][1],x[1]))
13
14     max_visitors = count_daily.map(lambda x: x[1]).max()
15     min_visitors = count_daily.map(lambda x: x[1]).min()
16
17     over_join = count_daily.join(over_X)
18     under_join = count_daily.join(under_X)
19
```

```
20      tot_days_over = over_join.count()
21      tot_days_under = under_join.count()
22
23      over_list = over_join.map(lambda x: (getBucket(x[1][0],
        max_visitors,min_visitors), 1))\
24              .reduceByKey(lambda a,b: a+b)\
25              .map(lambda x: (x[0], x[1] / tot_days_over * 100))\
26              .sortByKey()\
27              .collect()
28
29      under_list = under_join.map(lambda x: (getBucket(x[1][0],
        max_visitors,min_visitors), 1))\
30              .reduceByKey(lambda a,b: a+b)\
31              .map(lambda x:(x[0], x[1] / tot_days_under * 100))\
32              .sortByKey()\
33              .collect()
34
35      bucket = max(int(max_visitors/number_bucket), 1)
36
37      visitors_over = [ x[1] for x in over_list]
38      visitors_under = [ x[1] for x in under_list]
39      bucket_over = [ x[0]*bucket  for x in over_list]
40      bucket_under = [ x[0]*bucket  for x in under_list]
41
42      x_labels = range(0, max_visitors,bucket)
43      plt.xticks(x_labels)
44
45      plt.suptitle('Distribuzione visite poi numero {}'.format(
        poi))
46      plt.xlabel('Range visitatori')
47      plt.ylabel('Percentuale di giorni')
48      plt.grid()
49      plt.plot(bucket_over,visitors_over,color='b',label= 'rain
        >= X')
50      plt.plot(bucket_under,visitors_under,color='g',label= 'rain
         < X')
51      plt.legend()
52      plt.show()
```

Listing 3.16: Esecuzione e stampa Q2+c1

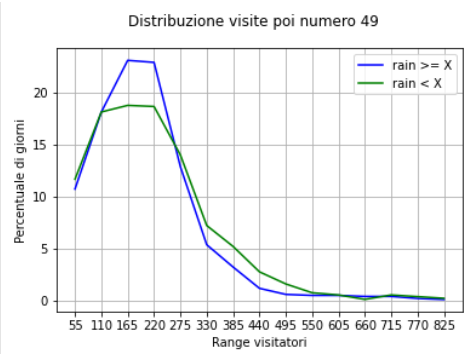

Figura 3.17: Risultato interrogazione Q2+C1 per il PoI 49, Arena di Verona

3.8 Q1 e Q2 in base a C2

Le interrogazioni a seguire sono analoghe a Q1+C1 e Q2+C1, differiscono solo per la chiamata della funzione di divisione dei giorni. Viene infatti utilizzata `splitTemp` descritta in 3.12. Per completezza vengono riportati i Listing e le immagini analoghe.

Q1 + C2

Per ogni Verona Card, il numero di punti di interesse (Points of Interest, PoI) visitati, da cui costruire la sua distribuzione (l'asse x contiene il numero di PoI, l'asse y la percentuale di Verona Card usate per quel numero di PoI) discriminando i giorni in cui la temperatura media era superiore a Y / giorni in cui era inferiore a Y.

```
1  visit_vc = data_vc.map(lambda x: ((x[2],x[1]), 1))\
2              .reduceByKey(lambda a,b: a+b)\
3              .map(lambda x: (x[0][0],(x[0][1],x[1])))
4
5  over_Y, under_Y = splitTemp(temp_day, mean_temp)
6
7  visit_vc_over = visit_vc.join(over_X)\
8              .map(lambda x: x[1][0])
9
10 tot_vc_over = visit_vc_over.map(lambda x: x[0]).distinct().
       count()
11
12 list_over = visit_vc_over.map(lambda x: (x[1],1))\
13             .reduceByKey(lambda a,b: a+b)\
14             .map(lambda x: (x[0],x[1] / tot_vc_over * 100))\
15             .sortByKey()\
16             .collect()
17
18
```

```
visit_vc_under = visit_vc.join(under_X)\
                 .map(lambda x: x[1][0])

tot_vc_under = visit_vc_under.map(lambda x: x[0]).distinct().
    count()

list_under = visit_vc_under.map(lambda x: (x[1],1))\
             .reduceByKey(lambda a,b: a+b)\
             .map(lambda x: (x[0],x[1] / tot_vc_under * 100))\
             .sortByKey()\
             .collect()
```

Listing 3.17: Esecuzione Q1+C2

```
import matplotlib.pyplot as plt

perc_vc_over = [ x[1] for x in list_over]
perc_vc_under = [ x[1] for x in list_under]
visit_number_over = [x[0] for x in list_over]
visit_number_under = [x[0] for x in list_under]

plt.suptitle('PoI visitati da ogni VC in un giorno')
plt.xlabel('# visite per verona card')
plt.ylabel('Percentuale')

plt.grid(ls = '-')
plt.plot(visit_number_over,perc_vc_over, color='r', label = '
    temperature >= Y')
plt.plot(visit_number_under,perc_vc_under, color='b', label = '
    temperature < Y')
plt.legend()

plt.show()
```

Listing 3.18: Stampa Q1+C2

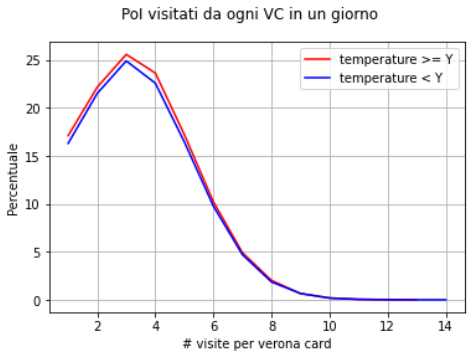

Figura 3.18: Grafico del risultato di Q1+C2

Q2 + C2

Per ogni PoI, la distribuzione del numero di visite giornaliere (l'asse x indica il range di visitatori, l'asse y la percentuali di giorni che quel PoI ha ricevuto un numero di visitatori all'interno di quel range) discriminando i giorni in cui la temperatura media era superiore a Y / giorni in cui era inferiore a Y.

```
import matplotlib.pyplot as plt

list_poi = data_vc.map(lambda x: x[3]).distinct().collect()

daily_visitors = data_vc.map(lambda x: ((x[3],x[2].split(' ')[0]), 1))\
                .reduceByKey(lambda a,b: a+b)

over_Y, under_Y = splitTemp(temp_day, mean_temp)

for poi in list_poi:
    count_daily = daily_visitors.filter(lambda x: x[0][0] == poi)\
                    .map(lambda x: (x[0][1],x[1]))
    max_visitors = count_daily.map(lambda x: x[1]).max()
    min_visitors = count_daily.map(lambda x: x[1]).min()

    over_join = count_daily.join(over_Y)
    under_join = count_daily.join(under_Y)

    tot_days_over = over_join.count()
    tot_days_under = under_join.count()

    over_list = over_join.map(lambda x: (getBucket(x[1][0], max_visitors,min_visitors), 1))\
            .reduceByKey(lambda a,b: a+b)\
            .map(lambda x: (x[0], x[1] / tot_days_over * 100))\
            .sortByKey()\
            .collect()

    under_list = under_join.map(lambda x: (getBucket(x[1][0], max_visitors,min_visitors), 1))\
            .reduceByKey(lambda a,b: a+b)\
            .map(lambda x:(x[0], x[1] / tot_days_under * 100))\
            .sortByKey()\
            .collect()

    bucket = max(int(max_visitors/number_bucket), 1)

    visitors_over = [ x[1] for x in over_list]
    visitors_under = [ x[1] for x in under_list]
    bucket_over = [ x[0]*bucket  for x in over_list]
    bucket_under = [ x[0]*bucket  for x in under_list]

```

```
41      x_labels = range(0, max_visitors,bucket)
42      plt.xticks(x_labels)
43
44      plt.suptitle('Distribuzione visite poi numero {}'.format(
        poi))
45      plt.xlabel('Range visitatori')
46      plt.ylabel('Percentuale di giorni')
47      plt.grid()
48      plt.plot(bucket_over,visitors_over,color='r', label = '
        temperature >= Y')
49      plt.plot(bucket_under,visitors_under,color='b', label = '
        temperature < Y')
50      plt.legend()
51
52      plt.show()
```

Listing 3.19: Esecuzione e stampa di Q1+C2

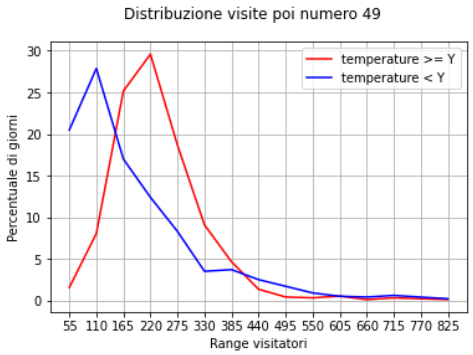

Figura 3.19: Risultato interrogazione Q2+C2 per il PoI 49, Arena di Verona

Capitolo 4

Richiesta supplementare

> Se assumiamo che in uno specifico giorno i dati delle centraline non siano stati registrati, è possibile risalire all'informazione se in tale giorno ha piovuto / non ha piovuto o ad una stima della temperatura media guardando la distribuzione dell'occupazione dei PoI o guardando l'uso delle Verona Card?

Per risolvere questa richiesta si è deciso di procedere con un approccio empirico-sperimentale.. Come eseguito per le interrogazioni precedenti, si è provveduto al caricamento dei dati con il filtraggio degli stessi, il codice è riportato in Listings 4.1.

```
1  def getDate(date):
2      day, month, year = date.split(' ')[0].split('/')
3      return '-'.join([year, month, day])
4
5  base_dir = 'dataset/'
6  filename_vc = base_dir + 'veronaCard.csv'
7  filename_w = base_dir + 'meteo.csv'
8
9  file_vc = sc.textFile(filename_vc)
10 file_w = sc.textFile(filename_w)
11
12 header = file_vc.first()
13
14 data_VC = file_vc.filter(lambda row: row != header)\
15             .map(lambda row : row.split(","))\
16             .filter(lambda x: x[0].split('-')[0] != '2020')\
17             .map(lambda x: (x[0], x[1], getDate(x[2]), x[4]))
18
19 data_VC.persist()
20
21 data_W = file_w.map(lambda row : row.split(","))\
22             .map(lambda x: (x[0],int(x[1][:2]),float(x[2]),
       float(x[7])))\
23             .filter(lambda x: x[1] < 24)\
```

```
            .filter(lambda x: x[3] > 0)\
            .filter(lambda x: x[2] < 50 and x[2] > -20)

data_W.persist()
```

Listing 4.1: Caricamento dataset e rimozione dati non consistenti

A seguito del caricamento vengono estratte due date da entrambi gli rdd (Listing 4.2), il 27 Luglio 2019, giornata calda di pioggia ed il 10 Novembre 2018, giornata fredda senza precipitazioni. L'estrazione del meteo dei due giorni sono state ricavate allo stesso modo in cui il dataset del meteo veniva preparato per le interrogazioni C1 e C2. Viene comunque riportato il codice in Listing 4.3 con il risultato ottenuto in Fig. 4.1.

```
day1 = '2019-07-27'
day2 = '2018-11-10'

data_w = data_W.filter(lambda x: x[0] != day1 and x[0] != day2)
day1_w = data_W.filter(lambda x: x[0] == day1)
day2_w = data_W.filter(lambda x: x[0] == day2)

data_vc = data_VC.filter(lambda x: x[0] != day1 and x[0] !=
    day2)
day1_vc = data_VC.filter(lambda x: x[2] == day1)
day2_vc = data_VC.filter(lambda x: x[2] == day2)
```

Listing 4.2: Estrazione date di cui si vuole ricavare il meteo

```
def getTestData(rdd):
    rain = rdd.map(lambda x: ((x[0],x[1]),x[3]) )\
                .reduceByKey(lambda a,b : (a+b) / 2)\
                .map(lambda x: (x[0][0], 1 if x[1]>=1 else 0))\
                .reduceByKey(lambda a,b: a+b)\
                .collect()
    temp = rdd.map(lambda x: ((x[0],x[1]),x[2]) )\
                .reduceByKey(lambda a,b : round((a + b)/2, 2))\
                .map(lambda x: (x[0][0],x[1]))\
                .reduceByKey(lambda a,b : round((a + b)/2, 2))\
                .collect()
    return rain, temp
rain1, temp1 =  getTestData(day1_w)
rain2, temp2 =  getTestData(day2_w)
```

Listing 4.3: Calcolo della temperatura media e delle ore di pioggia delle date estratte dal dataset

```
Giorno  2019-07-27
ore di pioggia:  [('2019-07-27', 7)]
temperatura media:  [('2019-07-27', 27.61)]

Giorno  2018-11-10
ore di pioggia:  [('2018-11-10', 0)]
temperatura media:  [('2018-11-10', 11.79)]
```

Figura 4.1: Ore di pioggia e temperatura media giornaliera ottenute

Vengono quindi preparati i dati per la suddivisione dei giorni grazie alle funzioni `splitRain` e `splitTemp` (Listings 3.9 e 3.12 rispettivamente) in giorni di pioggia e giorni senza perturbazioni ($X = 1$ ora di pioggia), in giorni caldi ed in giorni freddi ($Y = 20$°C). Il codice viene riportato in Listing 4.4

```
rain_hour = data_w.map(lambda x: ((x[0],x[1]),x[3]) )\
                .reduceByKey(lambda a,b : (a+b) / 2)\
                .map(lambda x: (x[0][0], 1 if x[1]>=1 else 0))\
                .reduceByKey(lambda a,b: a+b)

temp_day = data_w.map(lambda x: ((x[0],x[1]),x[2]) )\
                .reduceByKey(lambda a,b : round((a + b)/2, 2))\
                .map(lambda x: (x[0][0],x[1]))\
                .reduceByKey(lambda a,b : round((a + b)/2, 2))

day_rain, day_no_rain = splitRain(rain_hour, 1)
day_hot, day_cold = splitTemp(temp_day,20)
```

Listing 4.4: Suddivisione dei giorni secondo C1 e C2

Viene quindi eseguita l'interrogazione Q1+C1 con l'aggiunta al grafico dell'andamento delle visite per VC dei giorni selezionati precedentemente. Il codice è riportato in Listing 4.5 e il grafico ottenuto in Fig. 4.2

```
def execQ1(rdd, tot):
    list_visit = rdd.map(lambda x: (x[1],1))\
                .reduceByKey(lambda a,b: a+b)\
                .map(lambda x: (x[0],x[1] / tot * 100))\
                .sortByKey()\
                .collect()
    return list_visit

visit_vc = data_vc.map(lambda x: ((x[2],x[1]), 1))\
                .reduceByKey(lambda a,b: a+b)\
                .map(lambda x: (x[0][0],(x[0][1],x[1])))

#PIOGGIA
visit_vc_over = visit_vc.join(day_rain)\
                            .map(lambda x: x[1][0])

tot_vc_over = visit_vc_over.map(lambda x: x[0]).distinct().
    count()

list_over = execQ1(visit_vc_over, tot_vc_over)

#SOLE
visit_vc_under = visit_vc.join(day_no_rain)\
                            .map(lambda x: x[1][0])

tot_vc_under = visit_vc_under.map(lambda x: x[0]).distinct().
    count()

list_under = execQ1(visit_vc_under, tot_vc_under)
```

```
28
29 #GIORNI NUOVI
30 #1
31 visit_vc1 = day1_vc.filter(lambda x: x[2] == day1)\
32                     .map(lambda x: ((x[2],x[1]), 1))\
33                     .reduceByKey(lambda a,b: a+b)
34 tot1 = visit_vc1.count()
35
36 list1 = execQ1(visit_vc1, tot1)
37
38 #2
39 visit_vc2 = day2_vc.filter(lambda x: x[2] == day2)\
40                     .map(lambda x: ((x[2],x[1]), 1))\
41                     .reduceByKey(lambda a,b: a+b)
42 tot2 = visit_vc2.count()
43
44 list2 = execQ1(visit_vc2, tot1)
45
46 #PLOT
47 import matplotlib.pyplot as plt
48
49 perc_vc_over = [ x[1] for x in list_over]
50 perc_vc_under = [ x[1] for x in list_under]
51 visit_number_over = [x[0] for x in list_over]
52 visit_number_under = [x[0] for x in list_under]
53
54 perc_vc1 = [ x[1] for x in list1]
55 visit_number1 = [x[0] for x in list1]
56 perc_vc2 = [ x[1] for x in list2]
57 visit_number2 = [x[0] for x in list2]
58
59 plt.suptitle('PoI visitati da ogni VC in un giorno')
60 plt.xlabel('# visite per verona card')
61 plt.ylabel('Percentuale')
62
63 plt.grid(ls = '-')
64 plt.plot(visit_number_over,perc_vc_over, color='b', label= '
       rain >= X')
65 plt.plot(visit_number_under,perc_vc_under, color='g', label= '
       rain < X')
66 plt.plot(visit_number1, perc_vc1, color='orange', label= day1)
67 plt.plot(visit_number2, perc_vc2, color='pink', label= day2)
68 plt.legend()
69
70 plt.show()
```

Listing 4.5: Esecuzione Q1 + C1 con l'aggiunta al grafico dei due giorni di cui si vuole ricavare il meteo

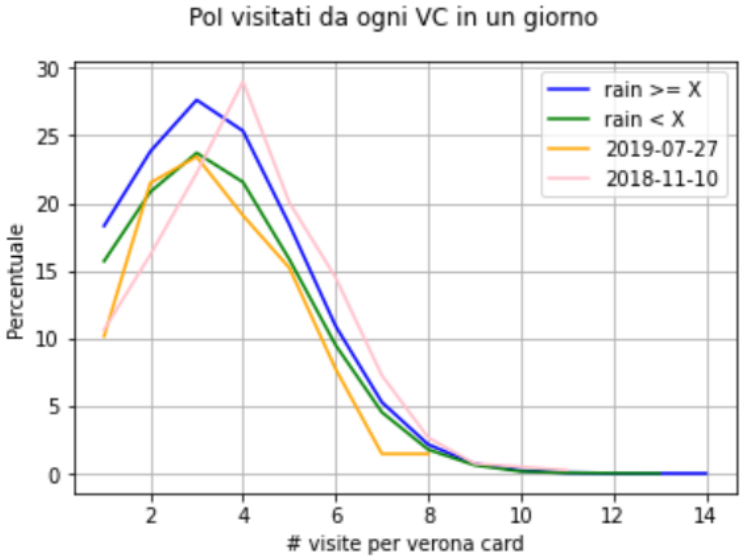

Figura 4.2: Grafico ottenuto da Q1 + C1 con $X = 1$

Si può notare che l'andamento delle due curve per giorni sereni (in verde) e di pioggia (in blu) sia leggermente differente. Nei giorni di pioggia circa il 27% del numero totale di VC hanno visitato 3 PoI, percentuale che per lo stesso numero di PoI scende a meno del 25% nei giorni senza perturbazioni. Vedendo l'affluenza dei due giorni presi in analisi si direbbe che il 27 Luglio 2019 (in arancio) sia stato un giorno sereno, il picco in 3 PoI visitati corrisponde infatti ai giorni con meno di un'ora di pioggia. Il 10 Novembre 2018 (in rosa) un giorno di pioggia essendo che poco meno del 30% delle VC ha visitato 4 PoI.
Purtroppo il meteo reale (Fig. 4.1) non rispecchia le conclusioni ricavate dal grafico, in Luglio infatti abbiamo un giorno di pioggia ed in Novembre un giorno senza perturbazioni. Se ci basassimo esclusivamente su questo grafico quindi **non sarebbe possibile predirre se in un giorno ha piovuto o meno**.

Viene ora eseguita l'interrogazione Q1+C2. Il codice utilizzato per risolvere la richiesta è analogo a Listing 4.5 fatta eccezione sul parametro di `join`. Per il codice che porta alla produzione del grafico in Fig. 4.3 riporto quindi al notebook in allegato reperibile anche al seguente link `https://github.com/4nnina/bigData_project/blob/main/bonus.ipynb`.

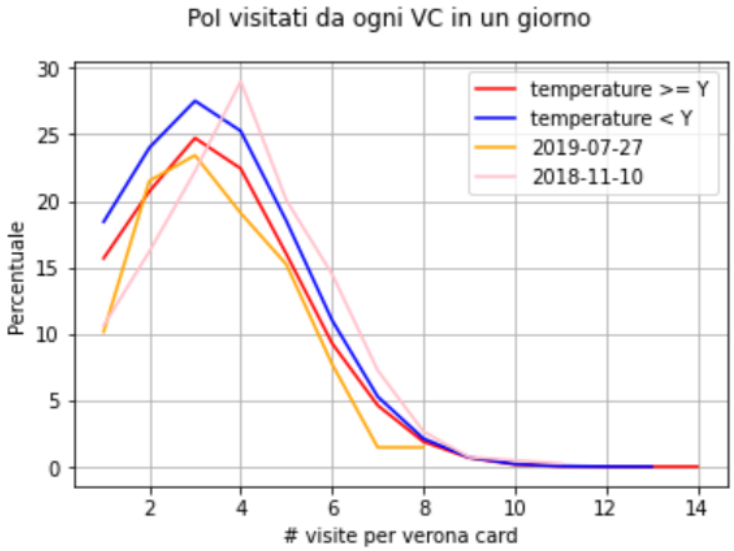

Figura 4.3: Grafico ottenuto da Q1 + C2 con $Y = 20$

Si noti che l'andamento delle visite per VC sia leggermente differente, in particolare nei picchi come nel grafico precedente. Ad una temperature minore (in blu) corrisponde che più del 25% degli utenti ha visitato 3 PoI, ad una temperatura maggiore (in rosso) il 25% degli utenti ha visitato 3 PoI. Nel giorno di Luglio (in arancione) 3 PoI sono stati visitati da poco meno del 25% degli utenti, la curva risulta essere più vicina alla curva della temperatura maggiore, in linea con la temperatura reale.
A Novembre invece si ha che quasi il 30% delle VC ha visitato 4 PoI, e la curva risulta essere simile alla curva delle temperature inferiori in linea alla temperatura reale. Da questo risultato si potrebbe quindi affermare che **è possibile distinguere giorni caldi e giorni freddi** a seconda del numero di PoI visitati da ogni VC al giorno.

Si procede quindi a vedere se è possibile dedurre il meteo guardando la distribuzione dell'occupazione dei PoI. In Listing 4.6 viene riportato il codice per l'esecuzione dell'interrogazione Q2 + C1 con l'aggiunta nel grafico della fascia di appartenenza dei due giorni di cui si vuole ricavare il meteo.

```
1 import matplotlib.pyplot as plt
2
3 list_poi = data_vc.map(lambda x: x[3]).distinct().collect()
4
5 daily_visitors = data_vc.map(lambda x: ((x[3],x[2].split(' ')
      [0]), 1))\
6                          .reduceByKey(lambda a,b: a+b)
7 daily_visitors1 = day1_vc.map(lambda x: ((x[3],x[2].split(' ')
      [0]), 1))\
8                          .reduceByKey(lambda a,b: a+b)
```

```
daily_visitors2 = day2_vc.map(lambda x: ((x[3],x[2].split(' ')[0]), 1))\
                         .reduceByKey(lambda a,b: a+b)

for poi in list_poi:
    count_daily = daily_visitors.filter(lambda x: x[0][0] == poi)\
                                .map(lambda x: (x[0][1],x[1]))
    count_daily1 = daily_visitors1.filter(lambda x: x[0][0] == poi)\
                                .map(lambda x: x[1])\
                                .collect()
    count_daily2 = daily_visitors2.filter(lambda x: x[0][0] == poi)\
                                .map(lambda x: x[1])\
                                .collect()

    max_day = max(count_daily1[0], count_daily2[0])
    min_day = min(count_daily1[0], count_daily2[0])
    max_visitors = max(count_daily.map(lambda x: x[1]).max(), max_day)
    min_visitors = min(count_daily.map(lambda x: x[1]).min(), min_day)

    over_join = count_daily.join(day_rain)
    under_join = count_daily.join(day_no_rain)

    tot_days_over = over_join.count()
    tot_days_under = under_join.count()

    over_list = over_join.map(lambda x: (getBucket(x[1][0], max_visitors,min_visitors), 1))\
                         .reduceByKey(lambda a,b: a+b)\
                         .map(lambda x: (x[0], x[1] / tot_days_over * 100))\
                         .sortByKey()\
                         .collect()

    under_list = under_join.map(lambda x: (getBucket(x[1][0], max_visitors,min_visitors), 1))\
                         .reduceByKey(lambda a,b: a+b)\
                         .map(lambda x: (x[0], x[1] / tot_days_under * 100))\
                         .sortByKey()\
                         .collect()

    bucket = max(int(max_visitors/number_bucket), 1)

    visitors_over = [ x[1] for x in over_list]
    visitors_under = [ x[1] for x in under_list]
    bucket_over = [ x[0]*bucket  for x in over_list]
    bucket_under = [ x[0]*bucket  for x in under_list]

```

```
52      x_labels = range(0, max_visitors,bucket)
53      plt.xticks(x_labels)
54
55      plt.suptitle('Distribuzione visite poi numero {}'.format(
        poi))
56      plt.xlabel('Range visitatori')
57      plt.ylabel('Percentuale di giorni')
58      plt.grid()
59      plt.plot(bucket_over,visitors_over,color='b',label= 'rain
        >= X')
60      plt.plot(bucket_under,visitors_under,color='g',label= 'rain
         < X')
61      plt.axvline(count_daily1[0], color='orange', label= day1)
62      plt.axvline(count_daily2[0], color='pink', label= day2)
63      plt.legend()
64
65      plt.show()
```

Listing 4.6: secuzione Q2 + C1 con l'aggiunta al grafico dei due giorni di cui si vuole ricavare il meteo

Alcuni grafici ottenuti da questa interrogazione sono riportati in Fig. 4.4. Si può notare che i grafici riportati nella prima colonna hanno un andamento molto simile sia nei giorni di pioggia che non, pertanto da questi non si può ricavare alcuna informazione riguardante i giorni di nostro interesse.

La seconda colonna invece, riporta dei grafici con dei picchi in fasce differenti. Essendo l'andamento variabile, si può inoltre considerare per ogni fascia più probabile la precipitazione rappresentata dalla linea con una percentuale maggiore. Contestualizzando maggiormente le informazioni, i PoI riportati nella seconda colonna sono quelli con una maggiore affluenza turistica essendo punti rilevanti per la provincia veronse.
Il PoI 49 corrisponde all'Arena di Verona, si potrebbe dire che a il 27 Luglio (in arancione) sia più probabile che sia stato un giorno di pioggia, il 10 Novembre (in rosa) invece non si ha una significativa differenza ma, si potrebbe dire che è più probabile per qualche punto percentuale che fosse un giorno senza precipitazioni. Queste due affermazioni sono in linea con le reali precipitazioni.
Il PoI 61 corrisponde alla Casa di Giulietta, per la data estiva non si può rilevare la prevalenza di una percentuale sull'altra, per la data autunnale si potrebbe dire che è pochi punti percentuale più probabile la pioggia, in contrasto con il dato raccolto dalle centraline.
Il PoI 71 corrisponde al Museo di Castelvecchio, sia per la data estiva che la data autunnale è più probabile che il giorno abbia avuto delle precipitazioni, questa affermazione non rispecchia la realtà.

A seguito di queste considerazioni possiamo affermare che dall'occupazione dei PoI è **difficile risalire all'informazione riguardante la**

pioggia.

> In conclusione si può affermare che non è possibile risalire alle condizioni della pioggia guardando a distribuzione dell'occupazione dei PoI o guardando l'uso delle Verona Card.

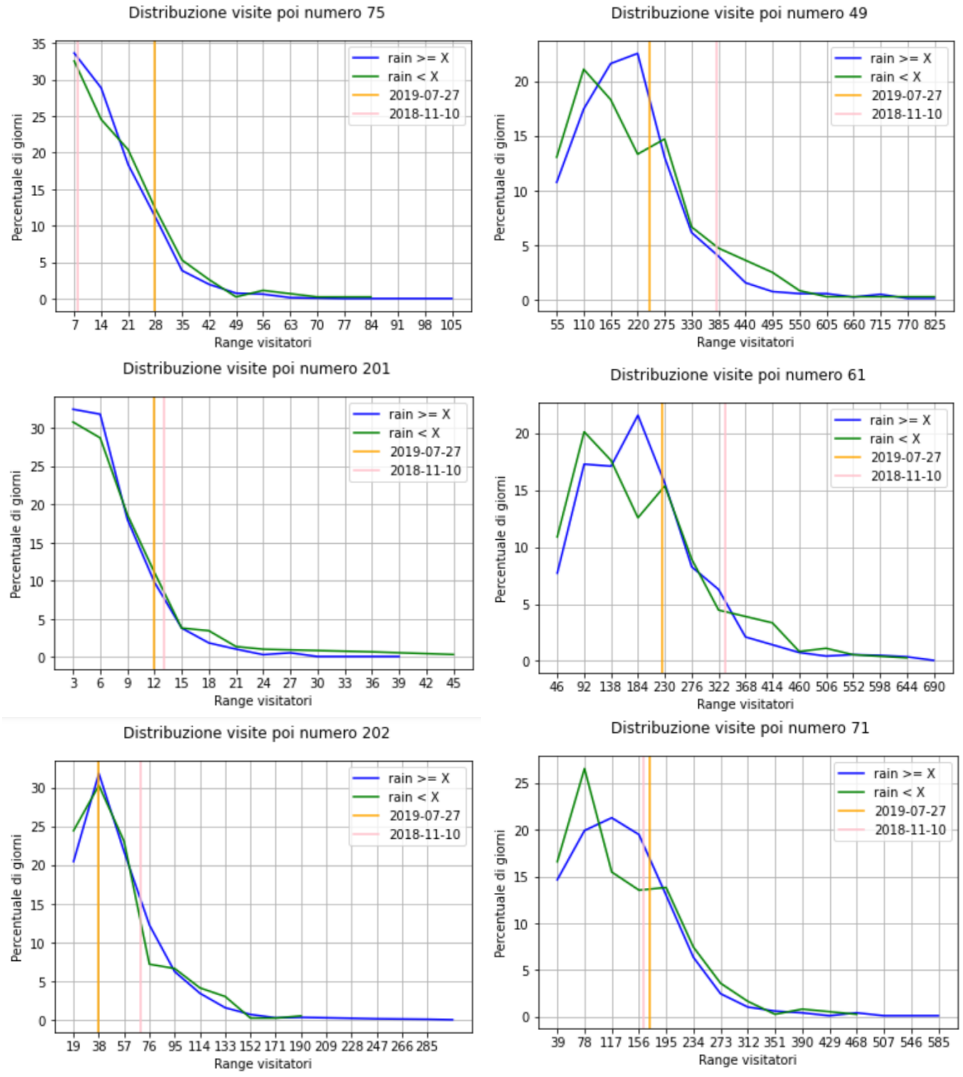

Figura 4.4: Grafici ottenuti da Q2 + C1 con $X = 1$

Viene ora eseguita l'interrogazione Q2 + C2, il codice è analogo al Listing 4.6 ad eccezione del parametro di `join`. Alcuni dei grafici prodotti sono riportati in Fig. 4.5.

Dalle immagini riportate si può notare un andamento differente per tutti i PoI. Si può inoltre osservare che per il giorno estivo, la probabilità di una temperatura maggiore è riportata in tutti i grafici ad eccezione del PoI 201 e 202 in cui gli andamenti in quella data fascia sono molto simili. Per il 10 Novembre invece, in 3 dei 6 grafici riportati (PoI 201, 61,49) si ha nella fascia di appartenenza del giorno un andamento simile sia per i giorni caldi che in quelli più freschi, il PoI 75 è l'unico a riportare la probabilità maggiore per il giorno più freddo, in linea con la temperatura reale, per gli altri invece la temperatura riscontrata risulta opposta alla realtà.

Si può affermare quindi che l'affluenza nei vari PoI **possa aiutare** per risalire all'informazione riguardante la **temperatura media giornaliera**.

In conclusione si può affermare che è possibile risalire alla stima della temperatura guardando la distribuzione dell'occupazione dei PoI o guardando l'uso delle Verona Card.

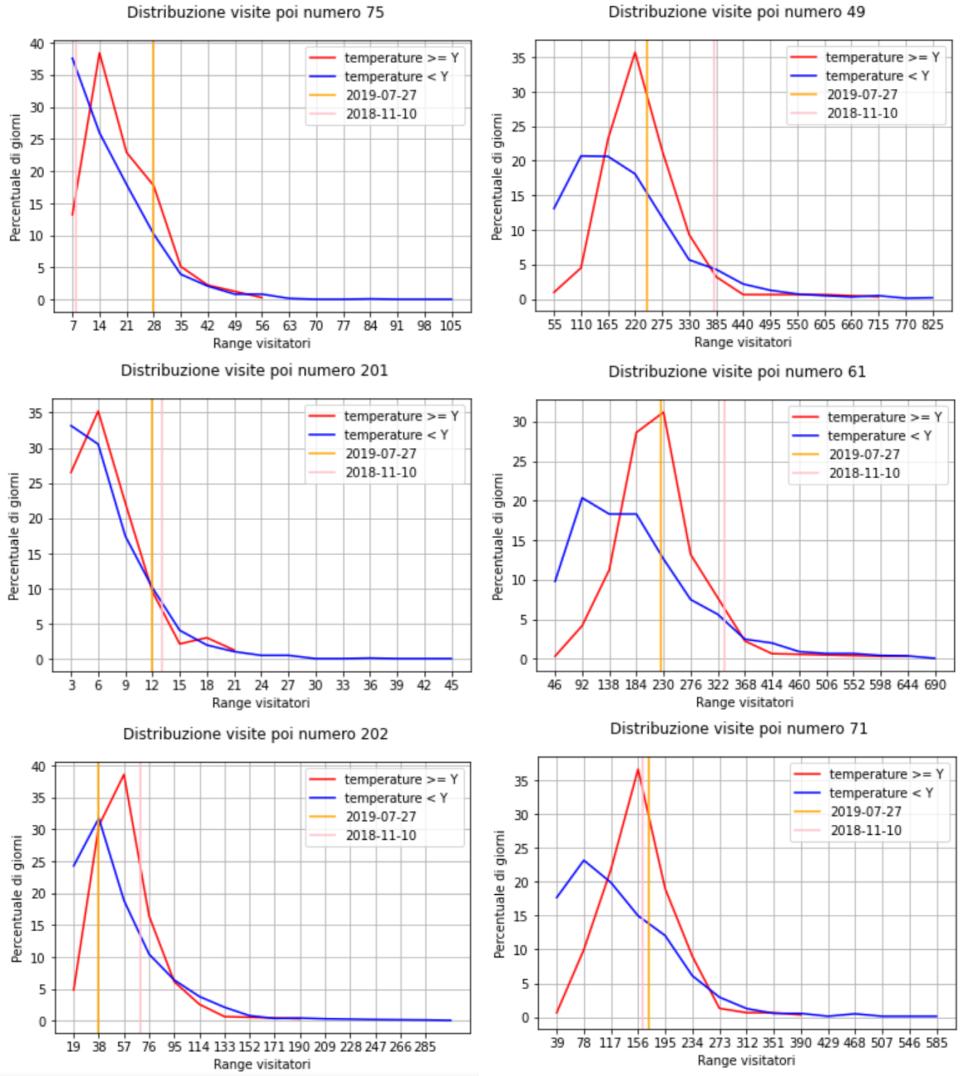

Figura 4.5: Grafici ottenuti da Q2 + C2 con $Y = 20$

4.1 Utilizzo di una tecnica di machine learning per la verifica dei risultati ottenuti

Questo dataset è stato utilizzato anche per un progetto di ricerca il cui scopo verteva alla predizione dell'occupazione dei PoI per creare dei sistemi di predizione per i turisti sull'affollamento in tempo reale. Durante questa attività, sono stati cercati dei metodi di predizioni che fossero affidabili, tra i metodi presi in analisi sono stati ottenuti dei risultati soddisfacenti impiegando le *Random Forest* per un training dataset con poche feature di ingresso e le reti neurali per dataset con un numero maggiore di feature.

Poichè il dataset da cui cerchiamo ricavare l'informazione sulla pioggia e sulla temperatura non ha un numero sostanzioso di features, si è deciso di provare a stimare il meteo con un modello *Random Forest*. È stato utilizzato PySpark per la preparazione dei dati da dare in pasto al modello predittivo. Poichè la previsione della temperatura e dei millimetri di pioggia in modo puntuale richiede metodi più raffinati, si è deciso di creare delle classi di temperatura e di pioggia riportati in Tab. 4.1.

classe	**° di temperatura**		**classe**	**mm di pioggia**
1	temp < 0		1	rain < 1
2	0 <= temp < 10		2	1 <= rain < 2
3	10 <= temp < 20		3	2 <= rain < 4
4	20 <= temp < 30		4	4 <= rain < 6
5	temp > 30		5	6 <= rain < 10
			6	10 <= rain < 30
			7	rain >= 30

Tabella 4.1: Suddivisione in classi della temperatura e della pioggia

È stato creato quindi un test set composto dal 20% dei giorni ed un training set composto dal rimanete 80% dei giorni. Al modello viene passata una lista composta da liste composte nel seguente modo [anno, mese, giorno, poi, presenze] e deve restituire una lista di liste [temperatura, pioggia].
A seguito del training si ottiene che nel test set si ottiene un Mean Absolute Percentage Error (MAPE) del 10% per la temperatura e del 81% per la pioggia, il risultato è una conferma di quanto ottenuto con la distribuzione dell'occupazione dei PoI e guardando l'uso delle Verona Card, è possibile risalire all'informazione sulla temperatura con un margine di errore relativamente basso ma, risalire alle perturbazioni risulta più difficile.

Il codice di quest'ultima parte è riportato nel Listing 4.7 e nel Listing 4.8.

```
1 def get_temp(t):
2     if t < 0:
3         return 1
4     elif t < 10:
```

```
        return 2
    elif t < 20:
        return 3
    elif t < 30:
        return 4
    return 5

def get_rain(r):
    if r < 1: #"pioviggine"
        return 1
    elif r < 2: #"pioggia debole"
        return 2
    elif r < 4: #pioggia leggera
        return 3
    elif r < 6: #"pioggia moderata"
        return 4
    elif r <10: #"pioggia forte"
        return 5
    elif r <30: #"rovescio"
        return 6
    return 7 #"nubifragio"


w_mean = data_w.map(lambda x: ((x[0],x[1]),(x[2],x[3])) )\
    .reduceByKey(lambda a,b :((a[0]+b[0])/2 , (a[1]+b[1])/2))\
    .map(lambda x: (x[0][0],x[1]))\
    .reduceByKey(lambda a,b :((a[0]+b[0])/2 , (a[1]+b[1])/2))\
    .map(lambda x: (x[0],(get_temp(x[1][0]),get_rain(x[1][1])))
    )\
    .sortByKey()

train_w,test_w = w_mean.randomSplit(weights=[0.8, 0.2],seed=1)

n_visit = data_vc.map(lambda x: ((x[3],x[2].split(' ')[0]),1))\
        .reduceByKey(lambda a,b: a+b)\
        .map(lambda x: (x[0][1], (x[0][0], x[1])))

train_visit_weather = n_visit.join(train_w)\
    .map(lambda x: [x[0].split('-'), int(x[1][0][0]), x
    [1][0][1], x[1][1][0], x[1][1][1]])

test_visit_weather = n_visit.join(test_w)\
    .map(lambda x: [x[0].split('-'), int(x[1][0][0]), x
    [1][0][1], x[1][1][0], x[1][1][1]])

train_data = train_visit_weather.map(lambda x: [int(x[0][0]),
    int(x[0][1]), int(x[0][2]), x[1], x[2]]).collect()

train_target = train_visit_weather.map(lambda x: x[3:]).collect
    ()

test_data = test_visit_weather.map(lambda x: [int(x[0][0]), int
    (x[0][1]), int(x[0][2]), x[1], x[2]]).collect()
```

```
52 test_target = test_visit_weather.map(lambda x: x[3:]).collect()
```

Listing 4.7: Preparazione dati per il modello

```
1  import numpy as np
2  from sklearn.ensemble import RandomForestRegressor
3  from sklearn.metrics import mean_absolute_percentage_error
4  from sklearn.metrics import accuracy_score
5  from sklearn.model_selection import train_test_split
6  from sklearn.preprocessing import MinMaxScaler
7  from sklearn.preprocessing import PowerTransformer
8
9  def MAPE(y_true, y_pred):
10     y_true, y_pred = np.array(y_true), np.array(y_pred)
11     return round(np.mean(np.abs((y_true - y_pred) / y_true)) *
       100, 2)
12
13 X_train, X_test, Y_train, Y_test = train_data, test_data,
       train_target, test_target
14
15 n = MinMaxScaler()
16 X_train = n.fit_transform(X_train)
17 X_test = n.transform(X_test)
18
19 forest_reg = RandomForestRegressor(n_estimators=10,
       random_state=0)
20 forest_reg.fit(X_train, Y_train)
21
22 data_predictions = forest_reg.predict(X_test)
23
24 data_prediction_temperature = [x[0] for x in data_predictions]
25 data_prediction_rain = [x[1] for x in data_predictions]
26 Y_test_temperature = [x[0] for x in Y_test]
27 Y_test_rain = [x[1] for x in Y_test]
28
29
30 print('\n----------------result--------------------\n')
31 print('mape temperature: ', MAPE(Y_test_temperature,
       data_prediction_temperature), '%')
32 print('mape rain: ', MAPE(Y_test_rain, data_prediction_rain), '
       %')
```

Listing 4.8: Utilizzo del modello

```
----------result----------

mape temperature:  10.16 %
mape rain:  81.47 %
```

Figura 4.6: Risultato ottenuto dall'esecuzione del codice in Listing 4.8