Università degli studi di Verona

Dipartimento di Informatica

Interrogazione di un Dataset rappresentante le strisciate delle Verona Card e dati delle stazioni meteorologiche con PySpark

Big Data

A.A. 2022/2023

Studentessa:

Elisa Acciari - VR478828

Professore:

Damiano Carra

Indice

1. Abstract

Questo progetto sarà strutturato in 3 fasi:

1. Interrogazione dei dati rappresentanti le strisciate delle VeronaCard
2. Interrogazione dei dati delle stazioni meteorologiche di Verona
3. Addestramento di un modello di classificazione che predica correttamente i giorni di pioggia e le temperature.

Nel primo punto andremo ad elaborare i dati andando innanzitutto a fare una pulizia di questi e a salvare solo i campi che ci interessano per l'analisi, per poi memorizzarli in un rdd, e procederemo successivamente con le seguenti interrogazioni:

1. Trovare il giorno con più accessi alle strutture di interesse e quello con meno e analizzare, per il giorno con più accessi, il numero di accessi ai POI durante le varie ore del giorno.
2. Verificare i valori di concentrazione di visitatori nei vari POI prendendo in considerazione prima tutti i giorni registrati e successivamente solo il giorno con maggiori accessi.

Nel secondo punto andremo sempre a ripulire i dati, catturando i campi di interesse, a memorizzarli in un rdd e ad eseguire le query in questo ordine:

- Query relative alla pioggia
  1. Dati i giorni con più e meno accessi, verificare i livelli di pioggia per quei giorni.
  2. Analizzare la concentrazione di visitatori nei vari POI in un giorno di pioggia e no.
- Query relative alla temperatura
  1. Dati i giorni con più e meno accessi, verificare i livelli di temperatura per quei giorni.
  2. Analizzare la concentrazione di visitatori nei vari POI in un giorno con alte temperature e con basse temperature

Infine, come terza fase, andremo ad addestrare un modello di classificazione che riesca a prevedere in quali giorni pioverà o meno e i livelli di temperatura che potrebbero verificarsi.

Inizialmente eseguiamo una previsione su due classi di Temperatura e due classi di Pioggia e vediamo quanto il mio modello è accurato.

- Temperatura

    - Classe 0: temperature <= 10
    - Classe 1: temperature > 10
- Pioggia
    - Classe 0: livelli di pioggia <= 6
    - Classe 1: livelli di pioggia > 6

Successivamente aumentiamo il numero delle classi ed osserviamo se riesce a prevedere in modo ottimo anche in casi non binari.

- Temperatura
    - Classe 0: temperature < 0
    - Classe 1: 0 <= temperature < 10
    - Classe 2: 10 <= temperature < 20
    - Classe 3: temperature >= 20
- Pioggia
    - Classe 0: pioggia < 2
    - Classe 1: 2 <= pioggia < 4
    - Classe 2: 4 <= pioggia < 6
    - Classe 3: 6 <= pioggia < 10
    - Classe 4: pioggia >= 10

L'algoritmo che andrò ad utilizzare è Logistic Regression, un algoritmo di classificazione utilizzato in pySpark per la classificazione binaria o multiclasse, usato con dati in formato rdd.
Ho scelto questo modello in quanto si comporta molto bene nei casi di classificazione lineare, ovvero quando il confine decisionale del modello è in grado di separare perfettamente i campioni tra le classi ed è proprio il nostro caso dato che le classi che andremo a trattare sono classi linearmente separabili.

Uno dei vantaggi della Logistic Regression in pySpark è la sua semplicità e la sua velocità di esecuzione su grandi quantità di dati.

In particolare, per addestrare il mio modello andrò ad utilizzare la libreria `LogisticRegressionWithLBFGS` che utilizza l'algoritmo di ottimizzazione LBFGS (Limited-memory Broyden-Fletcher-Goldfarb-Shanno) per minimizzare la funzione di costo del modello di regressione logistica sui dati di addestramento.

Questo algoritmo di ottimizzazione è particolarmente utile per la regressione logistica in quanto gestisce in modo efficiente problemi di grandi dimensioni.

Come funziona:

- utilizzo il metodo LogisticRegressionWithLBFGS**.train()** per addestrare il modello sui dati di training etichettati
- successivamente utilizzo il metodo LogisticRegressionWithLBFGS**.predict()** per prevedere le label dei dati di testing che il modello non ha mai visto prima.

2. Spark

Spark è un framework open source di elaborazione parallela utilizzato per l'elaborazione di grandi volumi di dati, con elevata velocità di elaborazione.

Uno dei vantaggi di Spark rispetto ad altri framework come Hadoop è la sua velocità di elaborazione, grazie all'utilizzo della tecnologia in-memory computing, che consente di mantenere i dati in RAM, anziché effettuare scritture e letture dal disco.

2.1 Architettura di Apache Spark

Apache Spark è organizzato secondo una struttura master/slave, ovvero una struttura di elaborazione distribuita in cui il processo principale, chiamato "master", coordina il lavoro dei processi secondari, chiamati "slave" o "worker".

Questa organizzazione di master/slave aiuta a distribuire l'elaborazione su più nodi del cluster, migliorando le prestazioni dell'elaborazione dei dati. Ciò significa che gli utenti possono elaborare grandi quantità di dati in parallelo su diversi nodi di calcolo all'interno del cluster, ottimizzando l'utilizzo delle risorse e riducendo i tempi di elaborazione complessivi.

Tale architettura è composta da tre elementi principali: driver, executors e gestore cluster.

2.1.1 Driver

Il driver, che funge da master, è il processo principale dell'applicazione, che gestisce l'intera operazione di elaborazione dei dati in Spark.
Il driver controlla il gestore cluster, programma il lavoro degli esecutori e comunica con loro per coordinare le attività di elaborazione.

È costituito dal programma e una sessione Spark.

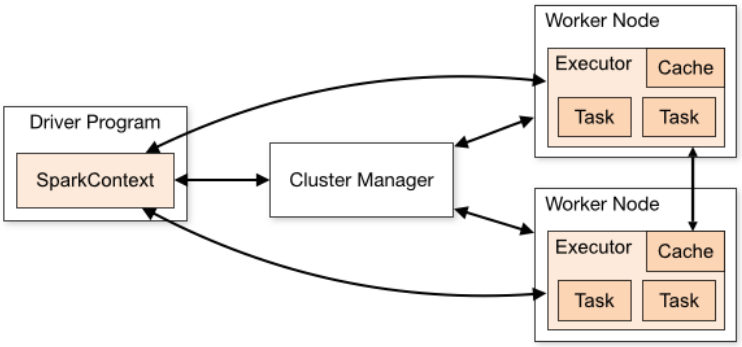
2.1.2 Executors

Gli esecutori sono i processi secondari, ovvero i worker, che effettuano l'effettiva elaborazione dei dati. Essi ricevono le istruzioni dal master e restituiscono i risultati allo stesso.

2.1.3 Gestore del cluster

Il gestore del cluster comunica sia con il driver che con gli esecutori per:

- Gestire l'allocazione delle risorse
- Gestire la divisione del programma
- Gestire l'esecuzione del programma

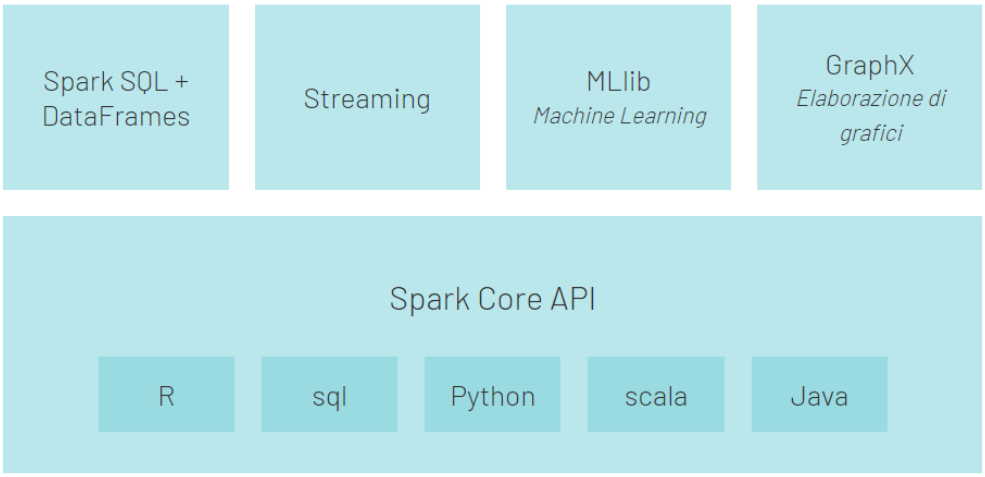


In definitiva, le applicazioni Spark vengono eseguite come insiemi indipendenti di processi su un cluster, coordinati dall'oggetto **SparkContext** nel programma principale (denominato *programma driver)*.
Nello specifico, per essere eseguito su un cluster, SparkContext può connettersi a diversi tipi di *gestori di cluster* che allocano le risorse tra le applicazioni. Una volta connesso, Spark acquisisce *esecutori* sui nodi del cluster e, successivamente, SparkContext invia *le attività* agli esecutori per l'esecuzione.

2.2 Ecosistema Spark

Il suo ecosistema è formato da:

- **Spark Core**, è il cuore del framework e fornisce le funzionalità fondamentali per la gestione dei dati. È il motore generico sottostante alla piattaforma Spark, sul quale sono basate tutte le altre funzionalità. Fornisce capacità di calcolo in memoria per offrire velocità, un modello di esecuzione generalizzato per supportare svariate applicazioni, e API Java, Scala e Python per facilitare lo sviluppo.
- **Spark SQL**, è il modulo Spark utilizzato per lavorare con dati strutturati. Consente di eseguire query di dati strutturati all'interno dei programmi Spark, utilizzando SQL o un'API DataFrame.
- **Spark Streaming**, consente di realizzare applicazioni di analisi interattive e potenti su dati storici e flussi in tempo reale, ereditando da Spark la facilità d'uso e la tolleranza agli errori.
- **MLlib** è la libreria Spark per il machine learning con strumenti che rendono il ML pratico, scalabile e facile da usare. MLlib contiene molti algoritmi di apprendimento comuni, come la classificazione, la regressione e il clustering.
- **GraphX**, è un motore per l'elaborazione di grafici costruito su Spark che offre una modalità interattiva per costruire, trasformare e analizzare dati strutturati in forma di grafico su larga scala.



2.3 Vantaggi Spark

- **Velocità**. Il vantaggio principale di Spark è la sua velocità. La potenza distintiva di Spark deriva dalla sua elaborazione **in-memory** .
  Ogni volta che concludo un'operazione, metto i dati in RAM; quindi, accelera esponenzialmente le velocità di accesso ai dati, poiché le informazioni archiviate nella RAM sono disponibili istantaneamente, mentre i dati conservati sui dischi sono soggetti ai limiti imposti dalle velocità di rete e disco, anche se la RAM è una memoria volatile.
- **Facilità d'uso**: Spark fornisce un'interfaccia di programmazione semplice e facile da usare
- **Flessibilità**: può essere utilizzato per una varietà di attività, che abbiamo già descritto sopra, come: l'apprendimento automatico, l'elaborazione in streaming ecc...
- **Tolleranza ai guasti**: fornisce tolleranza ai guasti integrata, il che significa che può eseguire il ripristino dai guasti dei nodi senza perdere dati o richiedere un intervento manuale.
- **Scalabilità**: Spark è progettato per essere altamente scalabile e può essere eseguito su cluster di migliaia di nodi.
- Community: Spark ha una community ampia e attiva di sviluppatori che contribuiscono al suo sviluppo e creano nuove applicazioni e strumenti.
- Unifica la pipeline andando a sviluppare tutti i tool in un unico framework, ovvero integra più fasi di elaborazione dei dati in un unico lavoro Spark coeso.

2.4 RDD (Resilient Distributed Datasets)

Spark RDD è la struttura dati fondamentale in Apache Spark, che rappresenta una raccolta immutabile e distribuita di insieme di record che può essere memorizzato ed elaborato su più nodi all'interno di un cluster. Gli RDD possono essere creati da varie origini dati come HDFS, file system locale e dalla parallelizzazione di una raccolta di dati esistente.

Gli RDD consentono a Spark di elaborare grandi dataset in parallelo, utilizzando il concetto di partizionamento, ovvero la suddivisione dei dati in “pezzi” che possono essere elaborati in modo indipendente su nodi diversi del cluster. Sono tolleranti ai guasti, il che significa che possono eseguire il ripristino dai guasti senza la necessità di un intervento manuale, in quanto ogni partizione viene replicata su diversi nodi del cluster.

D'altra parte, ***Spark DataFrame*** è una raccolta distribuita di dati organizzati in colonne denominate, simile a una tabella in un database relazionale. Fornisce un'API più user-friendly rispetto agli RDD, che consente agli utenti di interrogare e manipolare dati strutturati utilizzando una sintassi simile a SQL.

Sebbene Spark DataFrame sia più facile da usare, gli RDD svolgono ancora un ruolo importante in Spark perché forniscono maggiore flessibilità e controllo sulle operazioni di elaborazione dei dati.

2.4.1 Operazioni sui RDD

Gli RDD sono costruiti e manipolati attraverso 2 operazioni:

- **Trasformazioni**: posso combinare qualsiasi operatore e combinazioni di questi (map, filter, join …)
  L'applicazione di una trasformazione a un RDD produce un nuovo RDD (poiché gli RDD sono immutabili)
  Per avere una maggiore efficienza, tutte le operazioni eseguite da Spark sono ***lazy***, ovvero le operazioni non sono eseguite immediatamente, ma vengono memorizzate le trasformazioni che verranno poi eseguite effettivamente nel momento in cui viene chiesto il risultato, ovvero attraverso le *azioni*.
  In questo modo viene restituito al driver solo il risultato finale, senza il rallentamento del risultato dei passaggi intermedi.
- **Azioni**: operazioni per ottenere il dato finale (count, collect, save, …)

Inoltre, per garantire un accesso più rapido ai dati, possiamo utilizzare il comando **persist** su una rdd che permette di mantenere quest'ultima in ram per un accesso più veloce.

3. Implementazione codice

3.1 Inizializzare l'ambiente Spark

Andiamo ad utilizzare l'API Python per Apache Spark, ovvero PySpark, che consente di scrivere applicazioni Spark utilizzando il linguaggio di programmazione Python.
Utilizziamo Google Colab per eseguire il programma.

La prima cosa da fare è **installare pyspark** attraverso:

```
!pip install pyspark
```

Dopodiché, creiamo uno **SparkContext**.
In Apache Spark, SparkContext è il punto di ingresso per la creazione di RDD (Resilient Distributed Datasets) e l'esecuzione di operazioni su di essi. Rappresenta la connessione ad un cluster Spark ed è responsabile del coordinamento dell'esecuzione delle attività nel cluster.

```
import pyspark
sc = pyspark.SparkContext(appName="mySparkApp")
```

“mySparkApp” sarà il nome dell'applicazione.

Infine, ci connettiamo all’account di Google Drive tramite:

```
from google.colab import drive
drive.mount('/content/drive')
```

Ci posizioniamo sulla cartella del progetto ed iniziamo la vera e propria elaborazione dei dati.

3.2 Interrogazione dati delle Verona Card

Come prima fase vado ad interrogare i dati rappresentanti gli ingressi delle Verona Card.

Vado a generare un rdd attraverso lo SparkContext dal file di testo ‘dati.csv’, contenente i dati relativi alle strisciate delle Verona Card.

```
filename= "/content/drive/MyDrive/BigData/dati.csv"
fileRdd = sc.textFile(filename)
```

3.2.1 Pulizia dei dati

I dati inizialmente hanno il seguente schema:

nome campo	significato
Data	Data di ingresso
Ora	Ora ingresso
POI	Point of interest (ovvero la struttura che viene visitata)
device	Codice del dispositivo su cui è stata fatta la strisciata
serialVC	Seriale VeronaCard
dataActivation	Data di attivazione
in	number
number	number
profile	Tipo di VeronaCard (24-ore, 48-ore, ecc...)

In particolare, i dati nell’rdd si presentano in questo modo:

```
fileRdd.take(5)
```

```
['30-12-14,18:40:38,Tomba Giulietta,40,049B31523F3880,30-12-14,in,1,24 Ore',
 '30-12-14,18:40:36,Tomba Giulietta,40,049B26523F3880,30-12-14,in,1,24 Ore',
 '30-12-14,18:29:31,Casa Giulietta,27,04C316523F3884,30-12-14,in,1,72 Ore',
 '30-12-14,18:29:30,Casa Giulietta,27,04C315523F3884,30-12-14,in,1,72 Ore',
 '30-12-14,18:29:15,Casa Giulietta,27,045955523F3880,30-12-14,in,1,24 Ore']
```

I campi che mi interessano saranno:

Data	Ora	POI	serialVC	profile
'30-12-14'	'18:40:38'	'Tomba Giulietta'	'049B31523F3880'	'24 Ore'

Prima di andare effettivamente a selezionare questi campi, vado a ripulire il set di dati. In particolare, vado a:

1. Convertire la data nel formato AAAA-MM-DD, per una migliore visualizzazione
2. Correggere le incongruenze nel campo *profile*, in quanto alcune istanze mostrano un diverso formato del tipo: *'vrcard-24'*
   Quindi applico all'rdd una funzione ***profile()*** che mi permette di ottenere i seguenti risultati:
   a. da vrcard-24 → a 24 Ore
   b. da vrcard-48 → a 48 Ore
   c. da vrcard-72 → a 72 Ore
3. Una volta corrette le incongruenze vado a selezionare i campi sopra descritti ed eseguo *persist* per rendere più veloce l'accesso a **data_vr** in quanto sarà un rdd a cui accederò molte volte.

I passaggi sono riportati di seguito.

```
[9]  data= fileRdd.map(lambda row : row.split(","))
```

```
[12] data_vr = profile(data)
```

```
[13] data_vr = data_vr.map(lambda x: (datetime.strptime(x[0][0:-2] + '20' + x[0][-2:8], "%d-%m-%Y")\
                                      .strftime("%Y-%m-%d"), x[1], x[2], x[4], x[8]))
     data_vr.persist()
     data_vr.take(3)
```

```
     [('2014-12-30', '18:40:38', 'Tomba Giulietta', '049B31523F3880', '24 Ore'),
      ('2014-12-30', '18:40:36', 'Tomba Giulietta', '049B26523F3880', '24 Ore'),
      ('2014-12-30', '18:29:31', 'Casa Giulietta', '04C316523F3884', '72 Ore')]
```

Mostro che effettivamente la correzione del campo *profile* è avvenuta selezionando l'id di una Verona card che presentava il formato sbagliato in data ed osserviamo che in data_vr (ovvero l'rdd corretto) l'instanza con lo stesso id presenta il dato giusto.

```
[15] d_error = data.filter(lambda x: x[4] == '04AB1DE2185084')
     d_error.first()
```

```
     ['30-12-18',
      '18:32:51',
      'Casa Giulietta',
      '28',
      '04AB1DE2185084',
      '30-12-18',
      'in',
      '1',
      'vrcard-24-2019']
```

```
[16] d_correct = data_vr.filter(lambda x: x[3] == '04AB1DE2185084')
     d_correct.first()
```

```
     ('2018-12-30', '18:32:51', 'Casa Giulietta', '04AB1DE2185084', '24 Ore')
```

3.2.2 Query 1

`Trovare il giorno con più accessi alle strutture di interesse e quello con meno e analizzare per ognuno il numero di accessi durante le ore del giorno.`

1. Il primo punto da implementare è *trovare il giorno con più accessi alle strutture di interesse e quello con meno.*

Vado quindi a creare una nuova rdd andando ad applicare un map e poi un reduceByKey a data_vr per estrapolare solo i giorni e per contare il numero di istanze che sono registrate per ogni giorno, ottenendo il numero totale di accessi per ogni giorno.

Calcolo successivamente il giorno con accessi massimi e minimi con le operazioni di max e min.

```
[371] day = data_vr.map(lambda x: (x[0],1)).reduceByKey(lambda a,b: a+b)
      day_max_visit = day.max(lambda x:x[1])
      day_min_visit = day.min(lambda x:x[1])

      print("Giorno con più accessi: {} \nNumero di accessi: {} \n".format(day_max_visit[0], day_max_visit[1]))

      print("Giorno con meno accessi: {} \nNumero di accessi: {}".format(day_min_visit[0], day_min_visit[1]))
```

```
Giorno con più accessi: 2016-10-30
Numero di accessi: 8258

Giorno con meno accessi: 2020-11-02
Numero di accessi: 2
```

2. Successivamente devo *analizzare, per il giorno con massimi accessi, il numero di accessi durante le varie ore del giorno.*

Quindi vado a filtrare i dati in data_vr, prendendo solo quelle istanze per le quali il giorno corrisponde al giorno con massimi accessi.

Vado a convertire la stringa relativa al tempo in un elemento *date time* e ad estrapolare l'ora, andando poi a raggruppare con reduceByKey le istanze per ora.

Colleziono tutto in una lista e la ordino per ora crescente, quest'ultimo passo servirà per il plottaggio dell'asse x nel grafico.

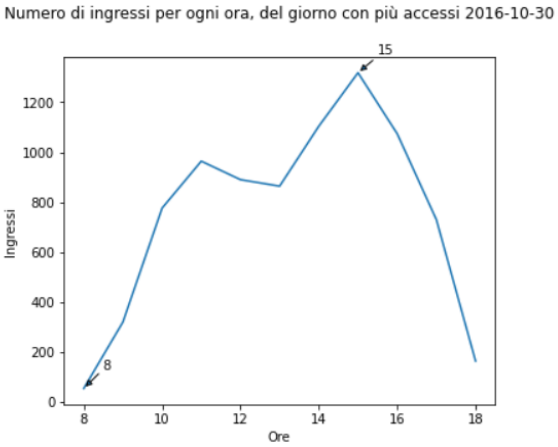
Ottengo una lista **dv_max** con elementi `(ora, numero di accessi)`.

```
[21] daily_visitors_max = data_vr.filter(lambda x: x[0] == day_max_visit[0] ) \
                         .map(lambda x: (datetime.strptime(x[1], '%H:%M:%S').time().hour ,1))\
                         .reduceByKey(lambda a,b: a+b)\

     dv_max = daily_visitors_max.collect()
     dv_max= sorted(dv_max)
```

```
[22] dv_max

     [(8, 54),
      (9, 320),
      (10, 776),
      (11, 964),
      (12, 890),
      (13, 864),
      (14, 1104),
      (15, 1318),
      (16, 1074),
      (17, 730),
      (18, 164)]
```

Plot

Ottengo il seguente grafico, in cui osserviamo l'andamento di accessi durante le varie ore del giorno e marchiamo le ore con massimi e minimi accessi, ovvero rispettivamente le **15:00** e le **8:00**.



3.2.3 Query 2

`Verificare i valori di concentrazione di visitatori nei vari poi, prendendo in considerazione prima` **`tutti i giorni`** `registrati e successivamente solo il` **`giorno con maggiori accessi.`**

Funzioni

- Funzione ***access*** che restituisce una lista del numero di accessi ai vari POI, dove l'indice della lista corrisponde ad un particolare POI.
  Questa lista mi servirà per fare il plot del numero di accessi.

```
[20] def access(poi, visit):
        j=0
        list_visit= []
        n= len(poi)
        for i in range(n):
            try:
                visit[j][0]
            except IndexError:
                if len(poi) > len(visit):
                    list_visit.extend(n*[0])

                return list_visit

            if(poi[i] == visit[j][0]):
                list_visit.append(visit[j][1])
                j+=1
            else:
                list_visit.append(0)
            n-=1
        return list_visit
```

- Funzione ***percent*** che mi calcola la percentuale di ogni elemento all'interno della lista
  Mi servirà sempre per il plot: le percentuali le visualizzeremo nell'asse a destra

```
[21] def percent(list_):
        list_percent=[]
        sum_ = sum(list_)
        for i in list_:
            list_percent.append(i/sum_)
        return list_percent
```

Inizialmente estrapolo i vari Point of Interest in una lista secondo l'ordine alfabetico.

```
[23] poi = data_vr.map(lambda x: x[2]).distinct().collect()
     poi = sorted(poi)
```

```
[31] poi
```

```
['Arena',
 'Casa Giulietta',
 'Castelvecchio',
 'Duomo',
 'Giardino Giusti',
 'Museo Conte',
 'Museo Lapidario',
 'Museo Miniscalchi',
 'Museo Radio',
 'Museo Storia',
 'Palazzo della Ragione',
 'San Fermo',
 'San Zeno',
 'Santa Anastasia',
 'Sighseeing',
 'Teatro Romano',
 'Tomba Giulietta',
 'Torre Lamberti',
 'Verona Tour']
```

Poi mi concentro sui seguenti due punti nella query:

1. Numero di accessi ai POI considerando tutti i gironi registrati
2. Numero di accessi ai POI nel giorno con maggiori visite

1. Numero di accessi ai POI considerando tutti i gironi registrati

Estrapolo quindi una lista di elementi `(str: 'nome POI', int: Numero accessi')` e li ordino in ordine alfabetico rispetto ai POI.

```
[23] d_visit = data_vr.map(lambda x: (x[2], 1))\
                .reduceByKey(lambda a,b: a+b)\
                .sortBy(lambda x: x[0]) \
                .collect()
```

Calcoliamo ora il **numero di accessi medio** per ogni POI, dividendo per il numero di giorni presi in considerazione, per fare poi anche un confronto con i risultati relativi al giorno con massime visite.

```
[24] day_tot= data_vr.map(lambda x: x[0]).distinct().count()
     l_visit =[]
     for p in d_visit:
       l_visit.append((p[0], int(p[1]/day_tot)))
```

```
[25] l_visit
```

```
[('Arena', 371),
 ('Casa Giulietta', 325),
 ('Castelvecchio', 239),
 ('Duomo', 177),
 ('Giardino Giusti', 28),
 ('Museo Conte', 0),
 ('Museo Lapidario', 34),
 ('Museo Miniscalchi', 0),
 ('Museo Radio', 0),
 ('Museo Storia', 13),
 ('Palazzo della Ragione', 98),
 ('San Fermo', 78),
 ('San Zeno', 72),
 ('Santa Anastasia', 195),
 ('Sighseeing', 0),
 ('Teatro Romano', 133),
 ('Tomba Giulietta', 79),
 ('Torre Lamberti', 250),
 ('Verona Tour', 0)]
```

Successivamente applico la funzione **access** che mi crea una lista in cui ogni elemento di indice *i* nella lista *l_visite* corrisponde al numero di accessi al Point of interest di indice *i* nella lista poi.

```
[35] lista_visite = access(poi, l_visite)
     lista_visite
```

```
[371, 325, 239, 177, 28, 0, 34, 0, 0, 13, 98, 78, 72, 195, 0, 133, 79, 250, 0]
```

2. Numero di accessi ai POI nel giorno con maggiori visite

Sappiamo che il giorno con più visite è stato salvato in `day_max_visit[0]`, quindi faccio gli stessi passaggi del punto precedente andondo a selezionare però le istanze che mi identificano questo giorno in particolare.

```
l_max_visit = data_vr.filter(lambda x: x[0] == day_max_visit[0]) \
                .map(lambda x: (x[2], 1)) \
                .reduceByKey(lambda a,b: a+b) \
                .sortBy(lambda x: x[0]) \
                .collect()
```

```
l_max_visit
```

```
[('Arena', 1456),
 ('Casa Giulietta', 1268),
 ('Castelvecchio', 890),
 ('Duomo', 448),
 ('Giardino Giusti', 130),
 ('Museo Lapidario', 198),
 ('Museo Radio', 4),
 ('Museo Storia', 44),
 ('Palazzo della Ragione', 594),
 ('San Fermo', 212),
 ('San Zeno', 176),
 ('Santa Anastasia', 520),
 ('Teatro Romano', 816),
 ('Tomba Giulietta', 462),
 ('Torre Lamberti', 1040)]
```

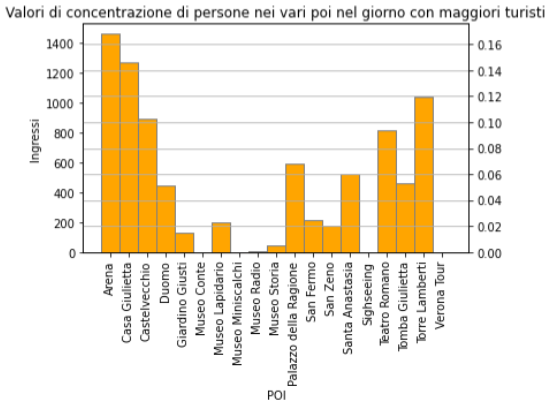
```
list_max_visit = access( poi, l_max_visit )
print(list_max_visit)
```

```
[1456, 1268, 890, 448, 130, 0, 198, 0, 4, 44, 594, 212, 176, 520, 0, 816, 462, 1040, 0]
```

Plot

Nei seguenti due istogrammi gli assi rappresentano:

- **asse x** rappresenta i vari POI
- **asse y** il numero di ingressi
- **terzo asse** le percentuali.



Possiamo notare una lieve differenza tra i due grafici: in particolare, nel giorno con più accessi rispetto a tutti gli altri giorni abbiamo una riduzione degli accessi a Santa Anastasia e al palazzo della Ragione e un aumento degli accessi al Teatro Romano. Oltre al fatto che ovviamente l'asse x relativo al plot del giorno con più visite presenterà valori più alti rispetto all'altro plot.

3.3 Interrogazione dati meteorologici

Anche qui andiamo a creare il nostro rdd attraverso Spark Context con i dati delle stazioni meteorologiche.

```
[34] filename= "/content/drive/MyDrive/BigData/meteo.csv"
     fileRdd_weather = sc.textFile(filename)
     fileRdd_weather.take(3)
```

```
['data,ora,t,ur,pr,wind,wind_dir,rain,dp,idstazione',
 '2016-02-08,11.54,10.7,82,1015.5,4.26,ESE,10.30,7.8,243',
 '2016-02-08,11.50,9.0,97,1015.5,7.97,SSW,19.80,8.5,351']
```

Le istanze si presenteranno con i seguenti campi:

data	ora	t	ur	pr	wind	wind_dir	rain	dp	idstazione
2016-02-08	11.54	10.7	82	1015.5	4.26	ESE	10.30	7.8	243

Quelli che interessano a noi saranno:

data	ora	t	rain
2016-02-08	11.54	10.7	10.30

3.3.1 Pulizia dei dati

1. Iniziamo ad eliminare la prima riga, rappresentante i nomi dei vari campi
2. Rendiamo l'ora nel formato giusto, ovvero HH:MM e successivamente filtriamo le ore in modo da avere solo gli orari in cui effettivamente ci sono visite
3. Convertiamo in float i dati relativi alla temperatura e alla precipitazione
4. Selezioniamo i dati della temperatura tra i -20 e i 50 gradi
5. Selezioniamo i dati di pioggia che siano maggiori di 0
6. Selezioniamo i campi che ci interessano

```
[120] index = fileRdd_weather.first()

      data_weather = fileRdd_weather.filter(lambda row: row != index)\
                     .map(lambda row : row.split(","))\
                     .map(lambda x: (x[0], x[1][:2] + ':' + x[1][3:], float(x[2]), float(x[7])))\
                     .filter(lambda x: (x[1] > '08:00') and (x[1] < '22:00') and (x[2] < 50 and x[2] > -20) and (x[3]>0 and x[3]<60))\
                     .map(lambda x: (x[0], error(x[1]), x[2], x[3] ))
                     #x[0] : data , x[1]: ora, x[2]: temp, x[3] : rain
      data_weather.persist()
      data_weather.take(3)
```

```
[('2016-02-08', '11:54', 10.7, 10.3),
 ('2016-02-08', '11:50', 9.0, 19.8),
 ('2016-02-08', '11:53', 8.7, 10.2)]
```

Anche su questa rdd *data_weather* applico persist, in quanto dovrò andare ad accedervi molte volte.

Funzioni

- Funzione ***mean_x*** che mi calcola la media dei valori di temperatura o di pioggia dell'ora associata

Input
x : lista di elementi `['ora', lista di elementi]`
Output
`list_`: lista con elementi `['ora', media degli elementi per quell'ora]`

```
[37] def mean_x(x):
         list_=[]
         for i in x:
             n= len(i[1])
             list_.append((i[0],sum(i[1])/n))

         return list_
```

- Funzione ***hour_x*** che mi estrae da una lista le ore e i valori di temperatura o pioggia di ogni ora e li inserisce in liste differenti.
  Input
  list_: lista con elementi `('ora', valore di temperatura o pioggia medio)`
  Output
  `ho, x`: ho `[lista di tutte le ore]`, x `[lista di tutti i valori medi di ogni ora]`

```
[38] def hour_x(list_):
         ho = []
         x = []
         for i in list_:
             ho.append(i[0])
             x.append(i[1])
         return ho, x
```

3.3.2 Query Pioggia

Spiegazione valori pioggia:

- pioggia debole (1 – 2 mm/h)
- pioggia leggera (2 – 4 mm/h)
- pioggia moderata (4 – 6 mm/h)
- pioggia forte (6 - 10 mm/h)
- rovescio (> 10 mm/h)

I campi che mi interessano sono:

data	ora	rain
'2016-02-08'	'11:54'	10.3

Li seleziono:

```
[40] data_rain = data_weather.map(lambda x: (x[0], x[1] ,x[3]))
```

3.3.2.1 Query 1

Trovare i giorni con più e meno accessi e verificare i livelli di ***pioggia*** per quei giorni.

Dalla prima query sappiamo che i giorni con più accessi e con meno corrispondono a:

```
[39] print(day_max_visit)
     print(day_min_visit)

     ('2016-10-30', 8258)
     ('2020-11-02', 2)
```

Quindi ora andiamo a selezionare le istanze nell'rdd relativa al meteo che corrispondono a tali giorni

```
rain_day_max = data_rain.filter(lambda x: x[0] == day_max_visit[0])
rain_day_min = data_rain.filter(lambda x: x[0] == day_min_visit[0])
```

Per una migliore visualizzazione dei valori di pioggia durante le varie ore, vado a raggruppare questi valori per l'ora corrispondente ed eseguo una media su questi, in modo tale da trovare il **valore di precipitazione media per ogni ora**.

```
rain_day_max = rain_day_max.map(lambda x: (x[1][:2], x[2])).groupByKey()\
          .map(lambda x : (x[0], list(x[1])))\
          .collect()
rain_day_min = rain_day_min.map(lambda x: (x[1][:2], x[2])).groupByKey() \
          .map(lambda x : (x[0], list(x[1])))\
          .collect()

# calcolo il valore medio per ogni ora del giorno
lrain_mean_max=  mean_x(rain_day_max)
lrain_mean_min= mean_x(rain_day_min)

# ordino con orari crescenti
lrain_mean_max= sorted(lrain_mean_max, key=itemgetter(0))
lrain_mean_min= sorted(lrain_mean_min, key=itemgetter(0))
```

La lista ***rain_day_max*** sarà formata da elementi (ora, lista dei valori di pioggia):

```
>> rain_day_max


[('19', [10.3, 7.8, 9.5,...]),
 ('15', [0.3, 0.3, 0.3,... ])
 ... ]
```

Ottengo due liste *lrain_mean_max, lrain_mean_min* contenenti per ogni ora, il valore medio di precipitazione rispettivamente per il giorno con maggiori accessi e per il giorno con meno.

Ad esempio, per ***lrain_mean_max*** abbiamo:

```
[47] lrain_mean_max
```

```
[('08', 0.5693181818181822),
 ('09', 0.5599315068493156),
 ('10', 0.5477272727272732),
 ('11', 0.5477272727272732),
 ('12', 0.5581699346405233),
 ('13', 0.5475000000000004),
 ('14', 0.5506756756756761),
 ('15', 0.5532467532467538),
 ('16', 0.5490625000000006),
 ('17', 0.5525641025641032),
 ('18', 0.5684210526315795),
 ('19', 0.5689542483660136),
 ('20', 0.5695512820512827),
 ('21', 0.5683006535947718)]
```

Estraggo le ore e i livelli di pioggia per i due giorni, i quali mi serviranno per il plot

```
time_max, rain_max = hour_x(lrain_mean_max)
time_min, rain_min = hour_x(lrain_mean_min)
```

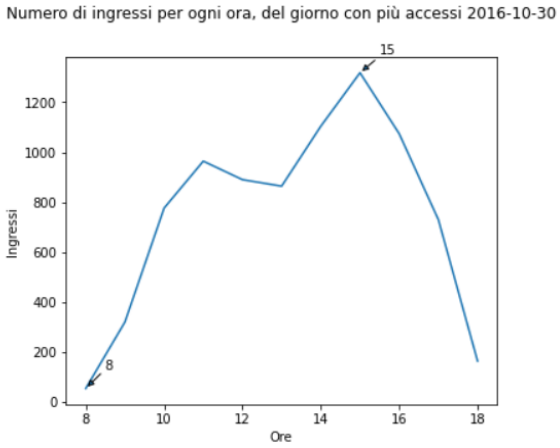
Plot

Visualizziamo due grafici rappresentanti l'andamento delle precipitazioni nei giorni con massimi e minimi accessi.

- **asse x:** valore di precipitazione
- **asse y:** ore del giorno (in particolare solo le ore in cui ci sono effettivamente visite)

Notiamo che nel grafico relativo al giorno 2016-10-30, ovvero al giorno con più accessi, abbiamo un andamento crescente di pioggia dalle ore 14:00.

Infatti, facendo un confronto con il grafo relativo al comportamento delle **visite del giorno con più accessi** (plot della Query 1 dell'interrogazione delle Verona Card), ovvero:



osserviamo due cose:

- che il picco massimo di accessi si ha alle 15 che corrisponde ad un valore di pioggia leggera, come vediamo nel plot sovrastante
- successivamente abbiamo una decrescita di visite, che corrisponde alle ore in cui la pioggia aumenta.

Quindi notiamo una relazione direttamente proporzionale fra il numero di visite e i valori di pioggia.

3.3.2.2 Query 2

`Analizzare la concentrazione di visitatori nei vari POI in un giorno di pioggia forte e con pioggia debole`

Questa volta, a differenza della query precedente in cui prendevo due giorni separati, andrò a considerare tutti i giorni registrati.

Per ogni giorno, raggruppo in una lista **m_rain** ogni livello di pioggia relativo a quel giorno:

```
m_rain = data_rain.map(lambda x: (x[0], x[2]))\
          .groupByKey()\
          .map(lambda x : (x[0], list(x[1])))\
          .collect()
```

Ottengo m_rain con elementi `(giorno, lista dei livelli di pioggia per quel giorno)`:

```
>> m_rain


[('2016-01-11', [0.3, 2.8, 0.5,...]),
 ('2019-05-16', [0.3, 0.3, 0.3,... ])
 ... ]
```

Applico poi la funzione ***mean_x*** che mi va a calcolare la media di valori di ogni lista ottenendo così in ***list_mean_rain***, per ogni giorno, il livello medio di pioggia, con elementi `(giorno, valore medio di pioggia)`

```
lista_mean_rain=  mean_x(m_rain)
lista_mean_rain= sorted(lista_mean_rain, key=itemgetter(0))
```

```
>> list_mean_rain


[('2016-01-11', 9.715892923998556),
 ('2016-01-18', 8.354323374340956),
 ... ]
```

Questa lista mi servirà ora per andare a filtrare i giorni con vari livelli medi di pioggia.

Ho scelto di selezionare i giorni con pioggia leggera/debole e pioggia forte superiore ai 15 mm/h.

```
# filtro i giorni di pioggia
lista_day_rain = []
lista_day_no_rain = []
for x in lista_mean_rain:
  if(x[1] > 15):
    lista_day_rain.append(x[0])
  elif (x[1] < 5):
    lista_day_no_rain.append(x[0])
```

```
# numero di giorni di pioggia forte
day_rain=len(set(lista_day_rain))
print(day_rain)

# numero di giorni con pioggia debole
day_no_rain= len(set(lista_day_no_rain))
print(day_no_rain)
```

```
94
1899
```

Vediamo intanto che i giorni di pioggia forte sono molto minori rispetto ai giorni di pioggia debole.

In **lista_day_rain** e **lista_day_no_rain** inserisco tutte le date che corrispondono ai giorni di pioggia forte e ai giorni di pioggia debole rispettivamente in modo da poter andare a selezionare le istanze relative in data_vr.

Ad esempio, **lista_day_rain** sarà composta dai seguenti elementi:

```
>> list_day_rain

['2014-01-05',
 '2014-01-14',
 '2014-01-31',
 ... ]
```

Query2 - Pioggia forte

Calcolo per tutti i giorni con pioggia forte, il numero di visite ad ogni POI fatte in quei giorni.

Vado quindi filtrare le istanze che corrispondono ai giorni di pioggia in list_day_rain e a collezionare in una lista ***visit_rain*** tutti gli elementi del tipo `(POI, numero di accessi)`:

```
visit_rain= data_vr.filter(lambda x: x[0] in lista_day_rain)\
              .map(lambda x: (x[2], 1))\
              .reduceByKey(lambda a,b: a+b)\
              .sortBy(lambda x: x[0])\
              .collect()
```

Successivamente vado a dividere i numeri di accessi ad un POI per il numero di giorni di pioggia, ottenendo così la media di accessi ad ogni POI in ***l_visit*** :

```
l_visit =[]
for p in visit_rain:
  l_visit.append((p[0], int(p[1]/day_rain)))
```

```
l_visit
```

```
[('Arena', 112),
 ('Casa Giulietta', 101),
 ('Castelvecchio', 81),
 ('Duomo', 56),
 ('Giardino Giusti', 6),
 ('Museo Conte', 0),
 ('Museo Lapidario', 11),
 ('Museo Miniscalchi', 0),
 ('Museo Radio', 0),
 ('Museo Storia', 3),
 ('Palazzo della Ragione', 34),
 ('San Fermo', 26),
 ('San Zeno', 22),
 ('Santa Anastasia', 61),
 ('Teatro Romano', 42),
 ('Tomba Giulietta', 24),
 ('Torre Lamberti', 79),
 ('Verona Tour', 0)]
```

Applico la funzione ***access*** che, ricordiamo, restituisce una lista del numero di accessi ai vari POI (in questo caso ***list_rain***), dove l'indice di ogni elemento corrisponde ad un particolare POI.

```
list_rain = access(poi, l_visit)
list_rain
```

```
[112, 101, 81, 56, 6, 0, 11, 0, 0, 3, 34, 26, 22, 61, 0, 42, 24, 79, 0]
```

Query2 - Pioggia leggera

Il procedimento è complementare a quello relativo ai giorni di pioggia forte, solo che in questo caso andrò a selezionare i giorni in ***list_day_no_rain***, ovvero relativi ai giorni di pioggia debole/leggera.

```
day_vr_no_rain= data_vr.filter(lambda x: x[0] in lista_day_no_rain)\
                .map(lambda x: (x[2], 1))\
                .reduceByKey(lambda a,b: a+b)\
                .sortBy(lambda x: x[0])\
                .collect()
l_visit =[]
for p in day_vr_no_rain:
  l_visit.append((p[0], int(p[1]/day_no_rain)))

l_visit
```

```
[('Arena', 155),
 ('Casa Giulietta', 135),
 ('Castelvecchio', 98),
 ('Duomo', 73),
 ('Giardino Giusti', 11),
 ('Museo Conte', 0),
 ('Museo Lapidario', 14),
 ('Museo Miniscalchi', 0),
 ('Museo Radio', 0),
 ('Museo Storia', 5),
 ('Palazzo della Ragione', 40),
 ('San Fermo', 32),
 ('San Zeno', 30),
 ('Santa Anastasia', 80),
 ('Teatro Romano', 55),
 ('Tomba Giulietta', 34),
 ('Torre Lamberti', 103),
 ('Verona Tour', 0)]
```

```
list_no_rain = access(poi, l_visit)
list_no_rain
```

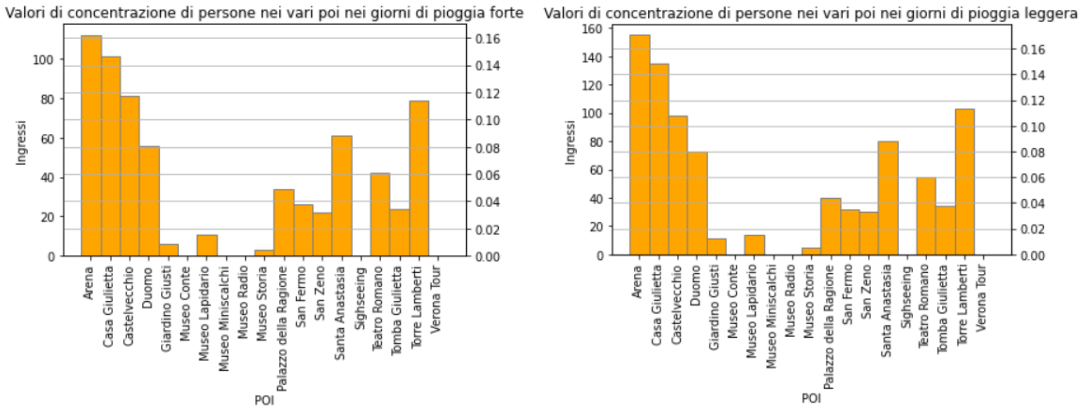
```
[155, 135, 98, 73, 11, 0, 14, 0, 0, 5, 40, 32, 30, 80, 0, 55, 34, 103, 0]
```

Plot

Nei seguenti due istogrammi gli assi rappresentano:

- **asse x** rappresenta i vari POI
- **asse y** il numero di ingressi

- **terzo asse** le percentuali.



Come possiamo notare dai due grafi sopra, le differenze **in media** sono minime, a parte che sul numero di accessi ai vari POI, il quale sarà più elevato nei giorni di pioggia leggera rispetto che in quelli con pioggia forte.

3.3.3 Query Temperatura

I campi che mi interessano sono:

data	ora	t
'2016-02-08'	'11:54'	10.3

Li seleziono:

```
[61] # seleziono  x[0]: giorno , x[1]: ora, x[2]: temperatura
     data_temp = data_weather.map(lambda x: (x[0], x[1] ,x[2]))
```

3.3.3.1 Query 1

Trovare i giorni con più e meno accessi e verificare i livelli di ***temperatura*** per quei giorni.

Eseguo operazioni complementari alla Query 1 delle Query relative alla pioggia.

Quindi seleziono i giorni con visite massime e minime e successivamente calcolo la media per trovare il **valore di temperatura media per ogni ora**.

```
[62] temp_day_max = data_temp.filter(lambda x: x[0] == day_max_visit[0])\
                    .map(lambda x: (x[1][:2], x[2]))\
                    .groupByKey()\
                    .map(lambda x : (x[0], list(x[1])))\
                    .collect()


     temp_day_min = data_temp.filter(lambda x: x[0] == day_min_visit[0])\
                    .map(lambda x: (x[1][:2], x[2]))\
                    .groupByKey()\
                    .map(lambda x : (x[0], list(x[1])))\
                    .collect()

     # faccio la media dei valori per ogni ora
     ltemp_mean_max=  mean_x(temp_day_max)
     ltemp_mean_min= mean_x(temp_day_min)

     # ordino per ora crescente
     ltemp_mean_max= sorted(ltemp_mean_max, key=itemgetter(0))
     ltemp_mean_min= sorted(ltemp_mean_min, key=itemgetter(0))
```

La lista temp_day_max sarà composta da elementi `(ora, lista di valori di temperatura)`:

```
>> temp_day_max


[('19',  [12.1, 12.4, ... ]),
 ('09',  [7.3, 11.8, ...]),
 ... ]
```

La lista **ltemp_mean_max** sarà così costruita:

```
ltemp_mean_max
```

```
[('08', 9.43560606060606),
 ('09', 12.032191780821918),
 ('10', 14.168831168831165),
 ('11', 15.802597402597405),
 ('12', 16.97581699346405),
 ('13', 17.821874999999988),
 ('14', 18.26216216216216),
 ('15', 17.76428571428571),
 ('16', 16.544375000000002),
 ('17', 14.253846153846153),
 ('18', 12.436184210526308),
 ('19', 11.36928104575163),
 ('20', 10.630769230769236),
 ('21', 9.952941176470588)]
```

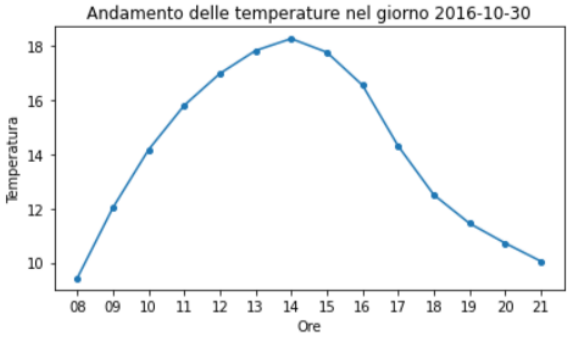
Estrapolo sempre le ore e i valori medi per ogni ora dei due giorni

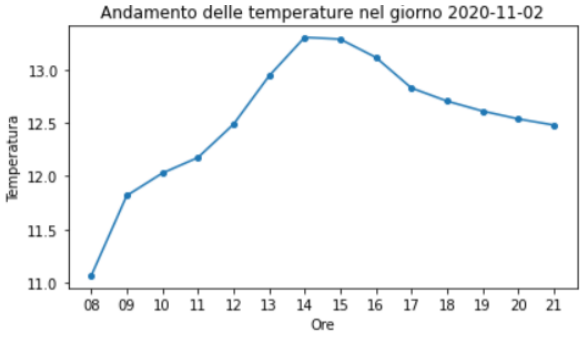
```
[65] time_temp_max, temp_max = hour_x(ltemp_mean_max)
     time_temp_min, temp_min = hour_x(ltemp_mean_min)
```

Plot

Visualizziamo due grafici rappresentanti l'andamento delle temperature nei giorni con massimi e minimi accessi.

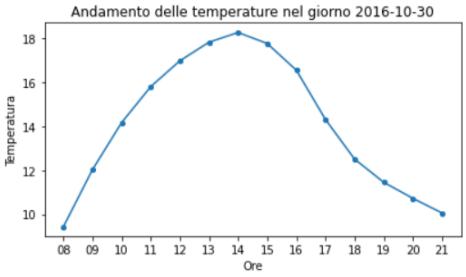
- **asse x:** valori di temperatura
- **asse y:** ore del giorno (in particolare solo le ore in cui ci sono effettivamente visite)



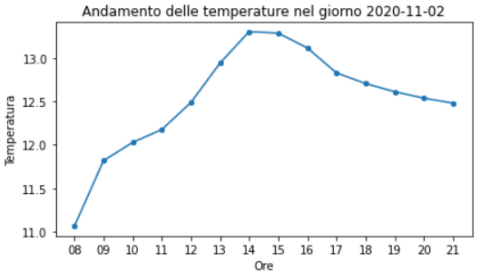


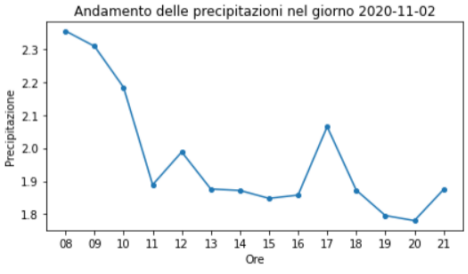
Confrontiamo i precedenti due plot con quelli relativi alla Query 1 nella sezione Pioggia:

- Per il giorno con più accessi osserviamo che le temperature diminuiscono verso le ore 15:00 (a sinistra) e la stessa ora corrisponde anche all'aumento di pioggia nel grafico delle precipitazioni (a destra).



- Per il giorno con meno accessi invece, dal confronto fra i due grafi, vediamo che per la prima parte della giornata le temperature hanno un andamento crescente fino alle 16, in cui abbiamo un picco di pioggia che provoca un decremento di temperatura.





3.3.3.2 Query 2

`Analizzare la concentrazione di visitatori nei vari POI in un giorno con alte temperature` **`[t>25]`** `e con basse temperature` **`[t<0]`**

I passaggi sono complementari a quelli della Query 2 relativa alla Pioggia.

Creo una lista **m_temp** contenente, per ogni giorno, i valori di temperatura registrati per quel giorno con elementi (`giorno, lista dei livelli di temperatura`)

Calcolo poi la media dei valori di temperatura ed ordino per data crescente, ottenendo **list_mean_temp** con elementi `(giorno, valore medio di temperatura)`

```
m_temp = data_temp.map(lambda x: (x[0], x[2]))\
          .groupByKey()\
          .map(lambda x : (x[0], list(x[1])))\
          .collect()
list_mean_temp=  mean_x(m_temp)
list_mean_temp= sorted(list_mean_temp, key=itemgetter(0))
```

```
>> list_mean_temp


[('2014-01-01', 6.722705601907048),
 ('2014-01-02', 4.704927609855206),
 ... ]
```

```
# mi salvo in due differenti liste i giorni con alte e basse temperature
list_day_high_temp = []
list_day_low_temp = []
for x in list_mean_temp:
  if(x[1] > 25):
    list_day_high_temp.append(x[0])
  elif (x[1] < 0):
    list_day_low_temp.append(x[0])
```

```
# numero di giorni con alte temperature
day_high_temp=len(set(list_day_high_temp))

# numero di giorni con basse temperature
day_low_temp= len(set(list_day_low_temp))
```

Query 2 - Temperature elevate

Seleziono nell'rdd relativa alle VeronaCard, ovvero **data_vr**, i giorni in cui le temperature erano elevate, ovvero i giorni presenti in ***list_day_high_temp***, e creo una lista ***visit_high_temp*** con elementi `('POI', numero di visite)`

```
visit_high_temp= data_vr.filter(lambda x: x[0] in list_day_high_temp)\
                 .map(lambda x: (x[2], 1))\
                 .reduceByKey(lambda a,b: a+b)\
                 .sortBy(lambda x: x[0])\
                 .collect()
```

Calcolo la media del numero di accessi in l_visit e poi applico a quest'ultima la funzione access che ricordo restituisce una lista del numero di accessi ai vari POI, dove l'indice della lista corrisponde ad un particolare POI.

```
l_visit =[]
for p in visit_high_temp:
  l_visit.append((p[0], int(p[1]/day_high_temp)))

list_high_temp = access(poi, l_visit)
list_high_temp
```

```
[87, 78, 63, 45, 6, 0, 8, 0, 0, 2, 23, 20, 17, 48, 0, 32, 15, 63, 0]
```

Query 2 - Temperature basse

Eseguo gli stessi passaggi, ma con i giorni in ***list_day_low_temp***.

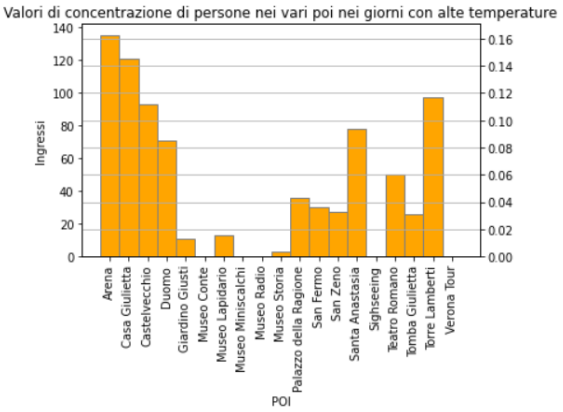
```
visit_low_temp= data_vr.filter(lambda x: x[0] in list_day_low_temp)\
                 .map(lambda x: (x[2], 1))\
                 .reduceByKey(lambda a,b: a+b)\
                 .sortBy(lambda x: x[0])\
                 .collect()

l_visit=[]
for p in visit_low_temp:
  l_visit.append((p[0], int(p[1]/day_low_temp)))

list_low_temp = access(poi, l_visit)
list_low_temp
```

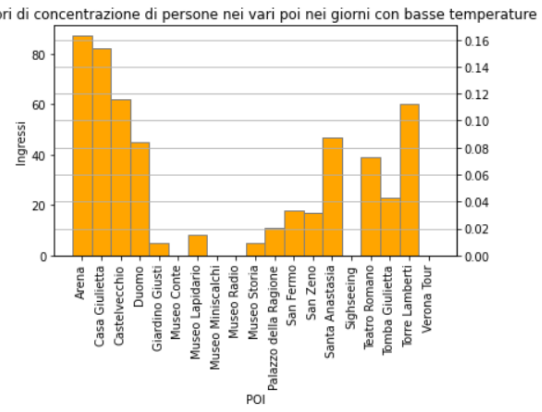
```
[87, 82, 62, 45, 5, 0, 8, 0, 0, 5, 11, 18, 17, 47, 0, 39, 23, 60, 0]
```

Plot

Nei seguenti due istogrammi gli assi rappresentano:

- **asse x** rappresenta i vari POI
- **asse y** il numero di ingressi
- **terzo asse** le percentuali.





Osservando i due plot notiamo che:

1. il numero di accessi nei giorni con temperature sopra i 25 gradi e sotto gli 0 gradi presentano un numero di accessi molto inferiore rispetto al plot in cui siamo andati a considerare tutti i giorni registrati, ovvero:

   Questo ci fa capire che quando le temperature sono molto elevate o molto basse si hanno meno visitatori in media

2. Inoltre, notiamo che la percentuale di accessi ai vari POI non cambia molto tra il grafico relativo ad alte temperature e quello in cui consideriamo tutti i giorni registrati.
3. Invece osserviamo una leggera differenza nell'istogramma relativo ai giorni di basse temperature, in cui il 'Palazzo della Ragione' presenta un numero inferiore di accessi e il 'Museo di Storia' un numero appena superiore rispetto agli altri due istogrammi.

Previsioni

4.1 Preparazione dati

Funzioni

- Funzione ***to_date*** che mi restituisce la data e l'ora da un timestamp
- Funzione ***to_timestamp*** che mi restituisce il timestamp da un elemento del tipo: ('2021-08-23', '18')

```
[80] def to_date(timestamp):
         # Converto il timestamp in un oggetto datetime
       dt_from_timestamp = datetime.fromtimestamp(timestamp)
       date= dt_from_timestamp.strftime('%Y-%m-%d %H:%M')
       return date

     def to_timestamp(date, time):
       # Converto il timestamp in un oggetto datetime
       dt = datetime.strptime(date + ' ' + time +':00', '%Y-%m-%d %H:%M')
       return dt.timestamp()
```

4.1.1 Temperatura

Creo una lista ***dt_temp*** in cui ho come chiave la coppia (data, ora) e come secondo elemento la lista di tutti i valori di temperatura registrati per quella data e quell'ora: `((data, ora), lista)`

```
[81] dt_temp = data_weather.map(lambda x: ((x[0], x[1][:2]), x[2]))\
                .groupByKey()\
                .map(lambda x : (x[0], list(x[1])))\
                .collect()
```

```
>> dt_temp
[(('2021-12-08', '16'), [4.2, 4.5,..]), (('2021-12-27', '16'), [8.2, 8.2, ..]), .... ]
```

Successivamente, vado a calcolare la media di questi valori per ogni (data, ora) creando la lista ***mean_dt*** con elementi `((data, ora), valore medio di temperatura)`:

```
[82] mean_dt = mean_x(dt_temp)
```

```
>> mean_dt
[(('2021-12-08', '16'), 4.17795275590551), (('2021-12-31', '16'), 6.095714285714282), ... ]
```

E poi creo un rdd, ***rdd_mean_temp***, sul quale andrò a basare le mie previsioni relative alla temperatura, che ha come elementi:

data	ora	temp media
'2016-02-08'	'11'	10.3

```
rdd_mean_temp = sc.parallelize(mean_dt)\
                .sortBy(lambda x: x[0])\
                .map(lambda x: (x[0][0], x[0][1], x[1]))
rdd_mean_temp.persist()
```

```
rdd_mean_temp.take(3)
```

```
[('2014-01-01', '08', 7.961363636363637),
 ('2014-01-01', '09', 6.312698412698411),
 ('2014-01-01', '10', 6.155000000000002)]
```

Anche per **rdd_mean_temp** andrò ad applicare *persist*, in quanto la utilizzerò in ognuna delle diverse classificazioni relative alla temperatura.

4.1.2 Pioggia

Come per i passaggi con i dati relativi alla temperatura, creo una lista ***dt_rain*** in cui ho come chiave la coppia (data, ora) e come secondo elemento la lista di tutti i valori di pioggia registrati per quella data e quell'ora `((data, ora), lista)`.

Successivamente vado a calcolare la media dei valori di pioggia per ogni ora.

Ed infine creo un rdd, ***rdd_mean_rain***, sul quale andrò a basare le mie previsioni, che ha come elementi:

data	ora	pioggia media
'2016-02-08'	'11'	10.3

```
dt_rain = data_weather.map(lambda x: ((x[0], x[1][:2]), x[3]))\
          .groupByKey()\
          .map(lambda x : (x[0], list(x[1])))\
          .collect()

mean_dr = mean_x(dt_rain)
rdd_mean_rain = sc.parallelize(mean_dr)\
                .sortBy(lambda x: x[0])\
                .map(lambda x: (x[0][0], x[0][1], x[1]))
rdd_mean_rain.persist()
```

```
rdd_mean_rain.take(3)
```

```
[('2014-01-01', '08', 2.6056818181818184),
 ('2014-01-01', '09', 2.13015873015873),
 ('2014-01-01', '10', 1.7670999999999992)]
```

Anche a **rdd_mean_rain** applico persist per lo stesso motivo di rdd_mean_temp.

4.2 Classificazione con due classi

4.2.1 Temperatura

In questo esempio, abbiamo definito 2 classi di temperatura:

1. basse, con **t <= 10**
2. alte, con **t > 10**

E abbiamo assegnato un'etichetta numerica a ciascuna classe (0 e 1 rispettivamente) con la funzione label_data_temp_2.

Funzione ***label_data_temp_2***

Utilizza queste label per creare i punti etichettati in modo che il modello possa prevedere la classe corretta.
Mi calcola il timestamp relativo alla data e all'ora dell'input e restituisce un LabeledPoint con classe 1 se la temperatura è maggiore di 10, classe 0 altrimenti

- **Input**
  point: `('data', 'ora', valore di temperatura o pioggia)`
- **Output**
  LabeledPoint : `LabeledPoint(classe, [timestamp, temperatura])`

Osservazioni:

- La conversione in time stamp mi servirà per poterla inserire come float nel vettore di feature di LabeledPoint, poiché richiede il seguente formato:
  LabeledPoint(*`label: float, features: Iterable[float])`*

- L'oggetto "LabeledPoint" è una classe della libreria PySpark, utilizzata per rappresentare i dati etichettati in un formato specifico richiesto dagli algoritmi di apprendimento automatico.

```
[83] def label_data_temp_2(point):
         date, time, temp = point
         tm_stamp = to_timestamp(date,time)
         if temp > 10:
             return LabeledPoint(1, [float(tm_stamp), float(temp)])
         else:
             return LabeledPoint(0, [float(tm_stamp), float(temp)])
```

Applico la funzione *label_data_temp_2* ad ogni istanza dell'RDD **rdd_mean_weather** utilizzando il metodo "map", creando un nuovo RDD di dati etichettati chiamato ***labeled_data***.

```
[191] labeled_data = rdd_mean_temp.map(lambda x: label_data_temp_2(x))
```

```
[192] labeled_data.take(3)
```

```
[LabeledPoint(0.0, [1388563200.0,7.961363636363637]),
 LabeledPoint(0.0, [1388566800.0,6.312698412698411]),
 LabeledPoint(0.0, [1388570400.0,6.155000000000002])]
```

4.2.1.1 Training

Splitto ora i dati in training set e testing set e poi **addestro** il logistic regression model sui dati di training

```
train_data, test_data = labeled_data.randomSplit([0.7, 0.3])
model = LogisticRegressionWithLBFGS.train(train_data, iterations=100, numClasses=2)
```

4.2.1.2 Testing

Prediciamo i valori di pioggia sui dati di test, usando il modello allenato, tramite la funzione predict e salvo in una lista le label predette (**label_predict**).
Estrapolo le vere label del testing set (**label_test_true**), per poi calcolare l'accuratezza confrontandole con quelle predette.

$$accuracy = TP + TN \ / \ TP + TN + FP + FN$$

$$TP: \text{ true positive}$$

$$TN: \text{ true negative}$$

$$FP: \text{ false positive}$$

$$FN: \text{ dalse negative}$$

```
predictions = model.predict(test_data.map(lambda x: x.features))
label_predict= predictions.collect()

# Calcolo l'accuratezza delle previsioni
label_test_true = test_data.map(lambda x: int(x.label)).collect()
label_test_true = [int(x) for x in label_test_true]
accuracy = sum([ int(x==y) for x, y in zip(label_predict, label_test_true)]) / len(label_predict)
print(accuracy)
```

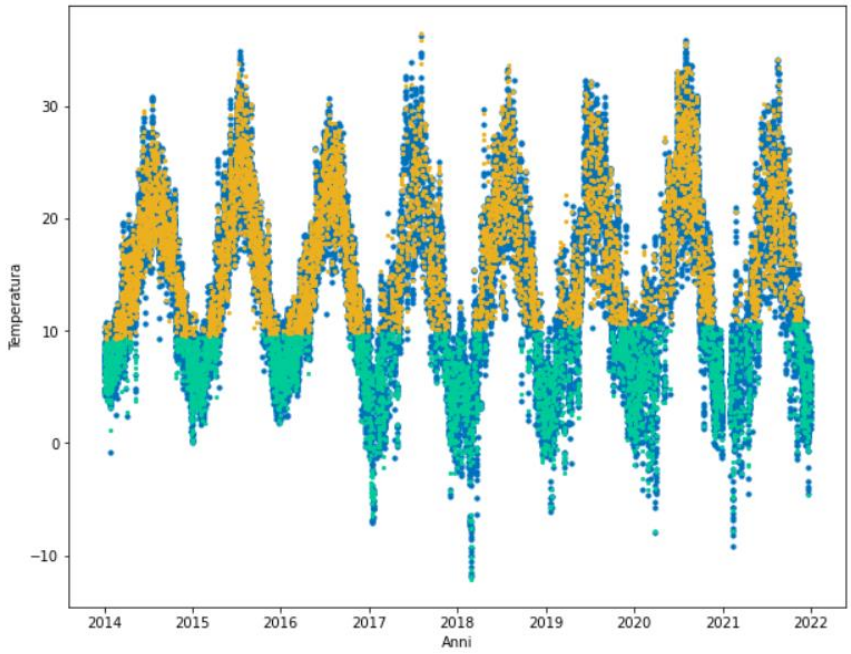
```
0.9807768924302789
```

Per una migliore visualizzazione dei dati e per realizzare il plot di questi punti nello spazio, creo due liste ***test_features*** e ***train_feature***, convertendo il timestamp in formato data in modo da visualizzare i diversi anni nel plot.

Queste due liste avranno elementi: `(array ([data, temperatura]), label_class)`

```
test_features= test_data.map(lambda x: (array([datetime.strptime(to_date(x.features[0]),'%Y-%m-%d %H:%M')\
                                               .date(),x.features[1]]), int(x.label)))\
                                               .collect()
train_features= train_data.map(lambda x: (array([datetime.strptime(to_date(x.features[0]),'%Y-%m-%d %H:%M')\
                                                 .date(),x.features[1]]), int(x.label)))\
                                                 .collect()
```

4.2.1.3 Plot



Osservazioni

Come possiamo vedere nel precedente grafico, il modello di regressione logistica riesce a predire in modo abbastanza accurato (con accuratezza sopra allo 0.9) i giorni con temperatura sopra i 10 gradi (**punti gialli**) e sotto (**punti verdi**).
I **punti blu** rappresentano i dati di training.

4.2.2 Pioggia

In questo esempio andiamo a definire due classi di pioggia:

1. Pioggia leggera/media
2. Pioggia forte

Funzione ***label_data_rain_2***

Mi calcola il timestamp relativo alla data e all'ora dell'input e restituisce un LabelPoint con classe 1 se la pioggia è maggiore di 6, classe 0 altrimenti

- **Input**
  point : `('data', 'ora', valore di pioggia)`
- **Output**
  LabeledPoint : `LabeledPoint(classe, [timestamp, pioggia])`

```
def label_data_rain_2(point):
    date, time, rain = point
    tm_stamp = to_timestamp(date,time)
    if rain < 6:
        return LabeledPoint(0, [float(tm_stamp), float(rain)])
    else:
        return LabeledPoint(1, [float(tm_stamp), float(rain)])
```

Applico poi la funzione **label_data_rain_2**

```
labeled_data = rdd_mean_rain.map(lambda x: label_data_rain_2(x))
labeled_data.take(3)
```

```
[LabeledPoint(0.0, [1388563200.0,2.6056818181818184]),
 LabeledPoint(0.0, [1388566800.0,2.13015873015873]),
 LabeledPoint(0.0, [1388570400.0,1.7670999999999992])]
```

4.2.2.1 Training

Splitto ora i dati in training set e testing set e poi **addestro** il logistic regression model sui dati di training

```
train_data, test_data = labeled_data.randomSplit([0.7, 0.3])
model = LogisticRegressionWithLBFGS.train(train_data, iterations=100, numClasses=2)
```

4.2.2.2 Testing

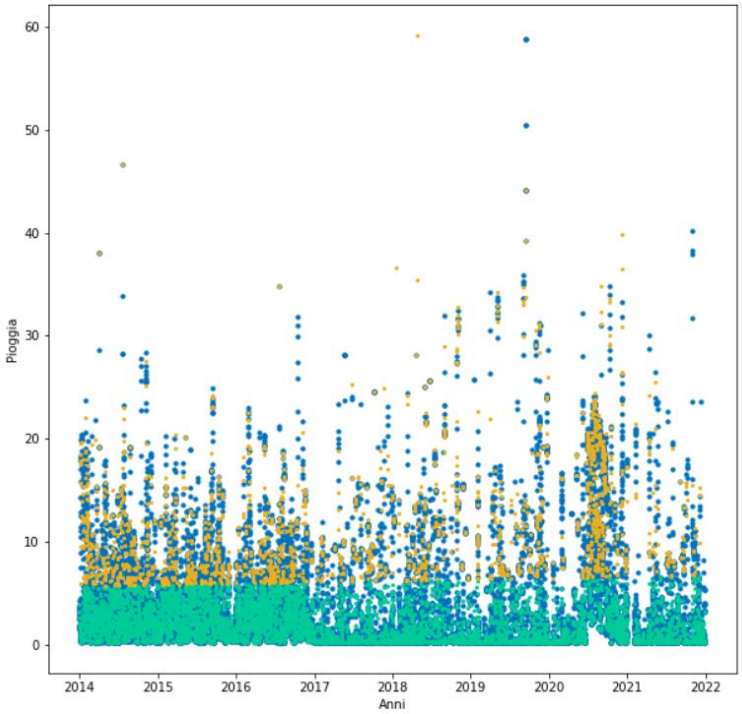
Eseguo passaggi analoghi al testing precedente, andando a calcolare l'accuratezza e andando a creare le due liste ***test_features*** e ***train_feature*** per il plottaggio dei dati.

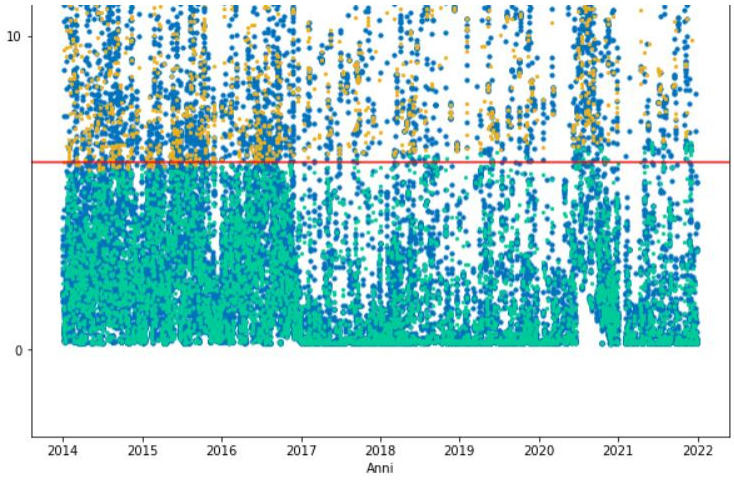
```
predictions = model.predict(test_data.map(lambda x: x.features))
label_predict= predictions.collect()

# Calcolo l'accuratezza delle previsioni
label_test_true = test_data.map(lambda x: int(x.label)).collect()
label_test_true = [int(x) for x in label_test_true]
accuracy = sum([ int(x==y) for x, y in zip(label_predict, label_test_true)]) / len(label_predict)

print(accuracy)
```

```
0.9920562175374275
```

```
test_features= test_data.map(lambda x: (array([datetime.strptime(to_date(x.features[0]),'%Y-%m-%d %H:%M')\
                                               .date(),x.features[1]]), int(x.label)))\
                                               .collect()
train_features= training_data.map(lambda x: (array([datetime.strptime(to_date(x.features[0]),'%Y-%m-%d %H:%M')\
                                                    .date(),x.features[1]]), int(x.label)))\
                                                    .collect()
```

4.2.2.3 Plot





Osservazioni

Nella seconda figura, andando ad ingrandire il grafo e delineando una riga rossa che rappresenta il confine tra le due classi, osserviamo più chiaramente i punti, notando una netta separazione tra i punti sotto la riga (classe di temperatura < 6) e i punti sopra (classe di temperatura >6).

Quindi il mio modello classifica bene i punti in base alla temperatura, con accuratezza del sopra lo 0.9.

4.3 Classificazione con più di due classi

4.3.1 Temperatura

In questo esempio, abbiamo definito 4 classi di temperatura:

1. Sottozero
2. Basse
3. Medie
4. Alte

Funzione ***label_data_temp_4***

Mi calcola il timestamp relativo alla data e all'ora dell'input e restituisce un LabelPoint con la rispettiva classe.

- **Input**
  point : `('data', 'ora', valore di temperatura)`
- **Output**
  LabeledPoint : `LabeledPoint(classe, [timestamp, temperatura])`

```
[127] def label_data_temp_4(point):
        date, time, temp = point
        if temp < 0:
          label = 0 # Temperatura sotto zero
        elif temp < 10:
          label = 1 # Temperatura bassa
        elif temp < 20:
          label = 2 # Temperatura media
        else:
          label = 3 # Temperatura alta

        tm_stamp = to_timestamp(date, time)

        return LabeledPoint(label, [float(tm_stamp), float(temp)])
```

Applico la funzione **label_data_temp_4**

```
labeled_data = rdd_mean_temp.map(lambda x: label_data_temp_4(x))
```

4.3.1.1 Training

Splitto ora i dati in training set e testing set e poi **addestro** il logistic regression model sui dati di training

```
train_data, test_data = labeled_data.randomSplit([0.7, 0.3])
model = LogisticRegressionWithLBFGS.train(train_data, iterations=100, numClasses=2)
```

4.3.1.2 Testing

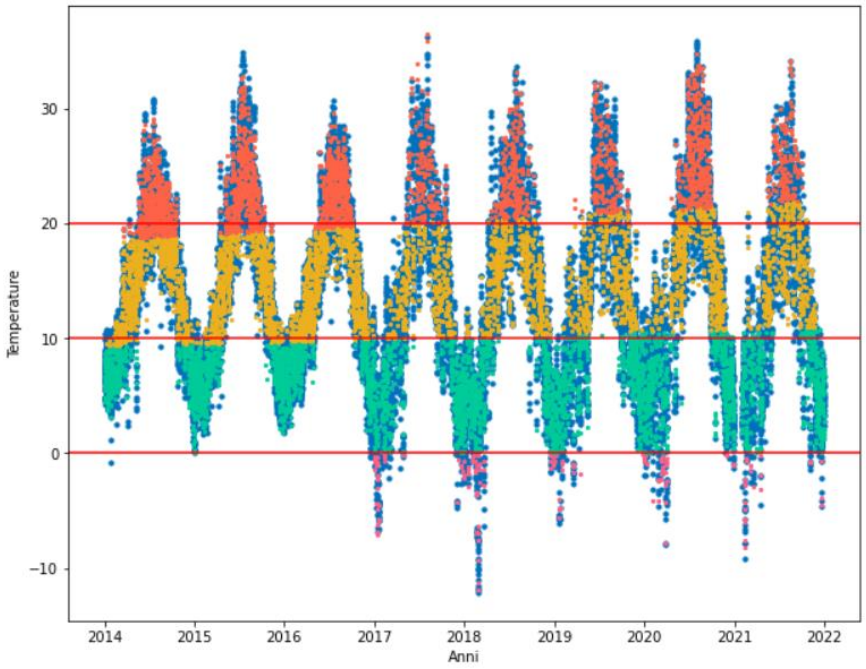
Calcolo l'accuratezza e vado a creare le due liste ***test_features*** e ***train_feature*** per il plottaggio dei dati.

```
predictions = model.predict(test_data.map(lambda x: x.features))

# Calcolo l'accuratezza delle previsioni
label_test_true = test_data.map(lambda x: int(x.label)).collect()
label_predict = predictions.collect()
accuracy = sum([int(x == y) for x, y in zip(label_predict, label_test_true)]) / len(label_predict)

print(accuracy)
```

```
0.945054945054945
```

```
test_features= test_data.map(lambda x: (array([datetime.strptime(to_date(x.features[0]),'%Y-%m-%d %H:%M')\
                                               .date(),x.features[1]]), int(x.label)))\
                                               .collect()
train_features= train_data.map(lambda x: (array([datetime.strptime(to_date(x.features[0]),'%Y-%m-%d %H:%M')\
                                                 .date(),x.features[1]]), int(x.label)))\
                                                 .collect()
```

4.3.1.3 Plot



Osservazioni

Anche qui la Logistic regression riesce ad individuare in modo ottimale le etichette reali dei miei dati, osserviamo infatti una netta separazione dei miei punti, dove le righe rosse rappresentano i confini tra le 4 classi.

Il valore di accuratezza è leggermente diminuito rispetto alla classificazione binaria, ma è comunque buono.

4.3.2 Pioggia

In questo esempio, abbiamo definito 5 classi di livelli di pioggia:

1. Debole
2. Leggera
3. Moderata
4. Forte
5. Nubifragio

Funzione ***label_data_rain_5***

Mi calcola il timestamp relativo alla data e all'ora dell'input e restituisce un LabelPoint con la relativa classe.

- **Input**
  point : `('data', 'ora', valore di pioggia)`
- **Output**
  LabeledPoint : `LabeledPoint(classe, [timestamp, pioggia])`

```
def label_data_rain_5(point):
  date, time, rain = point
  if rain < 2:
    label = 0 # Pioggia debole
  elif rain < 4:
    label = 1 # Pioggia leggera
  elif rain < 6:
    label = 2 # Pioggia moderata
  elif rain < 10:
    label = 3 # Pioggia forte
  else:
    label = 4 # Rovescio

  tm_stamp = to_timestamp(date, time)

  return LabeledPoint(label, [float(tm_stamp), float(rain)])
```

```
labeled_data = rdd_mean_rain.map(lambda x: label_data_rain_5(x))
```

4.3.2.1 Training

Splitto ora i dati in training set e testing set e poi **addestro** il logistic regression model sui dati di training

```
train_data, test_data = labeled_data.randomSplit([0.7, 0.3])
model = LogisticRegressionWithLBFGS.train(train_data, iterations=100, numClasses=2)
```

4.3.2.2 Testing

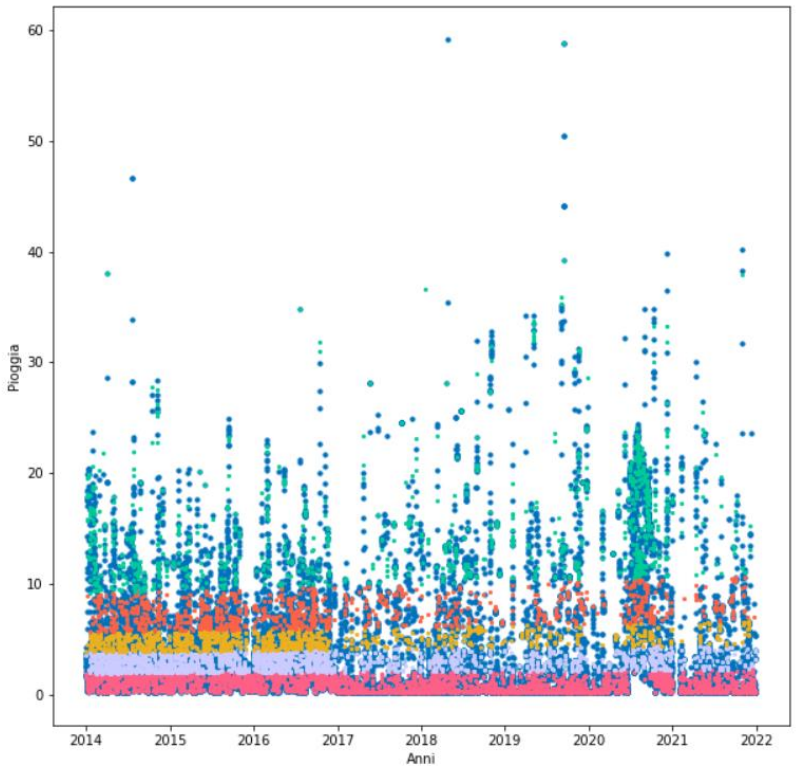
Calcolo l'accuratezza e vado a creare le due liste ***test_features*** e ***train_feature*** per il plottaggio dei dati.

```
predictions = model.predict(test_data.map(lambda x: x.features))
label_test_true = test_data.map(lambda x: int(x.label)).collect()
label_predict = predictions.collect()
accuracy = sum([int(x == y) for x, y in zip(label_predict, label_test_true)]) / len(label_predict)

print(accuracy)
```

```
0.9639466774389013
```

```
test_features= test_data.map(lambda x: (array([datetime.strptime(to_date(x.features[0]),'%Y-%m-%d %H:%M')\
                                               .date(),x.features[1]]), int(x.label)))\
                                               .collect()
train_features= train_data.map(lambda x: (array([datetime.strptime(to_date(x.features[0]),'%Y-%m-%d %H:%M')\
                                                .date(),x.features[1]]), int(x.label)))\
                                                .collect()
```

4.3.2.3 Plot



Osservazioni

Anche qui il mio modello riesce a prevedere in modo ottimale i dati di test in base ai 5 livelli di pioggia con un’accuratezza pari a 0.96.