SECONDO PROGETTO PER IL CORSO DI BIG DATA, 2018/2019
GRUPPO : MAR-KEEN

Vincenzo Martello (487518)
Shamri Makeen Mohamed (474975)

Analisi di Big Data applicata all'agricoltura di precisione

SOMMARIO

ABSTRACT

In questa relazione si descriverà il lavoro svolto durante il secondo progetto per il corso di Big Data. È stato scelto il topic 8 : *Analisi di dati IoT*, in particolare si analizzeranno dati relativi all'agricoltura di precisione che sono stati raccolti da alcuni campi di noccioleti dell'azienda agricola Vignola, situati a Caprarola, in provincia di Viterbo. In particolare i dati sono prelevati da una stazione meteo e da dei sensori IoT posti nel sottosuolo a diverse profondità.

L'obiettivo principale è stato quello di implementare delle analisi batch su questi dati, in particolare prevedere l'evapotraspirazione tramite i dati meteo. Quest'ultima è una misura fondamentale nel campo dell'agricoltura di precisione, in particolare per gestire al meglio l'irrigazione in maniera automatica. Per questa analisi ci siamo basati su alcuni paper sull'agricoltura di precisione che verranno citati nelle fonti, ad esempio il FAO 56 che tratta tutte le formule sull'irrigazione e sul drenaggio standardizzate.

Inoltre abbiamo implementato ulteriori job per calcolare dei report relativi ai vari dati, ad esempio un job per calcolare dei possibili malfunzionamenti nei sensori, oppure job riassuntivi delle varie misurazioni su un certo arco temporale (settimane, mesi).

A corredo di queste analisi abbiamo implementato un'architettura tipica di big data tramite Docker, per automatizzare il deploy dell'architettura stessa su una qualsiasi piattaforma (Linux o anche Windows).

Il codice del progetto è disponibile al seguente link.

CONTESTO E OBIETTIVI

Il contesto, come già accennato, è quello di un'azienda agricola che si è dotata di una infrastruttura IoT per raccogliere dati che possano aiutare a migliorare le decisioni degli esperti (agronomi) o prevedere fenomeni non ancora accaduti. In particolare ci sono state fornite due sorgenti di dati, sotto forma di due file CSV, contenenti dati rispettivamente della stazione meteo e dei sensori sotterranei.

I dati sono delle tipiche misurazioni meteorologiche, campionate ogni 15 minuti e vanno dal mese di ottobre 2018 al mese di giugno 2019. Ad esempio abbiamo la temperatura media, temperature minime e massime, umidità relativa, velocità del vento e millimetri di

pioggia. Per quanto riguarda i sensori abbiamo misurazioni di temperatura e umidità. Nel paper sul progetto *PANTHEON*, da cui è partito il nostro lavoro è già descritta una proposta di architettura per la gestione dei dati, e noi abbiamo cercato di basarci su di essa nello sviluppare la nostra proposta. Inoltre abbiamo esaminato le varie tecniche di analisi dei dati proposte nel paper *Machine learning in agriculture : a review* e tramite di esso abbiamo trovato dei riferimenti a degli articoli per l'analisi dei dati meteorologici, che ben si adeguavano alla natura dei nostri dati. Il nostro punto di arrivo è stato dunque il paper *Using MARS, SVM, GEP and empirical equations for estimation of monthly mean reference evapotranspiration*, da cui abbiamo preso spunto per analizzare i nostri dati e fare delle previsioni.

Abbiamo scelto questi riferimenti perché sono i più adatti alla tipologia di dati a nostra disposizione. Nell'agricoltura di precisione sono state proposte numerosissime analisi differenti, ma necessitano di altri tipi di dati, ad esempio immagini delle piantagioni, che nel nostro caso non erano a disposizione.

Nello specifico l'**evapotraspirazione** è una misura di quanta acqua evapora dalla superficie del terreno e traspira attraverso il fogliame. È dunque una misura per comprendere quanta acqua sta "perdendo" la coltivazione, è come se fosse la misura duale alle precipitazione. Una alta evapotraspirazione indica dunque una coltivazione che ha maggiore necessità di acqua rispetto ad un'altra coltivazione con evapotraspirazione più bassa. Tramite la stima corretta di questo valore ed altri indicatori si può effettivamente calcolare il fabbisogno di acqua di una coltivazione e quindi eventualmente tarare un sistema di irrigazione automatico.

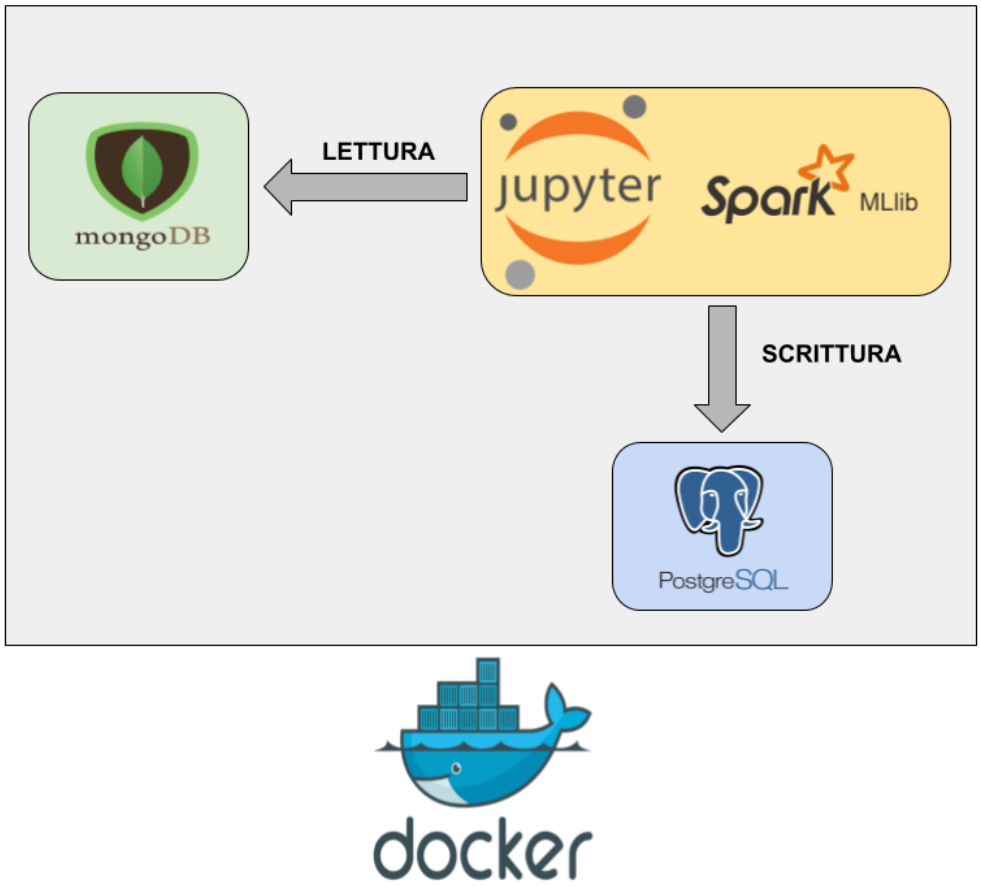
Questo valore è talmente importante che il paper 56 della FAO è effettivamente dedicato alla sua stima e delle relative stime meteo necessarie a calcolare l'evapotraspirazione stessa. È bene dire che non esiste una formula esatta, le varie formule presenti nel paper sono tutte empiriche, e proprio per questo il machine learning può essere utile per definire nuovi modelli per il calcolo di questo valore, che magari siano più accurati dei metodi empirici stessi.

Oltre a questo calcolo, come già detto, ci siamo concentrati sulla definizione di un'architettura per l'analisi batch di dati da sensori IoT. Abbiamo deciso di memorizzare i dati su MongoDB, essendo le fonti di dati variegate (come citato nel paper su Pantheon), analizzare i dati tramite Spark core e fare previsioni con SparkMLlib e infine memorizzare i vari report su un db relazionale, Postgres, essendo i report stessi strutturati e molto

contenuti (dunque per essi non è necessario un sistema di memorizzazione distribuito e schemaless, come i tipici noSQL). In ogni caso l'architettura verrà descritta meglio nella prossima sezione.

ARCHITETTURA

L'architettura è riassunta nella seguente immagine:



I blocchi rappresentano ognuno un diverso container dell'architettura. Come si può vedere vi è un container per MongoDB, un altro contenente uno Spark engine e un'installazione di Jupyter Notebook, utile per eseguire i vari job Spark in maniera interattiva, anziché lanciare il job con il classico comando ***spark-submit***. Infine vi è un container con Postgres, utile per memorizzare i risultati dei vari job. Inoltre ci sono altri due container dell'architettura non mostrati, perché non aventi delle capacità funzionali. Sono i container con delle istanze di Mongo-Express e Adminer, due interfacce web dalle quali è possibile visualizzare rispettivamente le varie collezioni presenti su una certa

istanza di MongoDB e su una certa istanza di Postgres. L'architettura è istanziata tramite Docker che permette ai vari container di stare in un ambiente isolato, ma anche di comunicare tra di loro. In particolare si è sfruttato ***docker-compose***, uno strumento che permette di avviare una qualsiasi architettura a partire da una file .yml che la descrive. Il seguente stralcio di codice racchiude il docker-compose.yml da noi utilizzato:

```
version: '3'

services:

 pyspark:
  image: jupyter/all-spark-notebook:latest
  container_name: jupyter_pyspark
  working_dir: /home/jovyan/code
  ports:
  - 8888:8888/tcp
  - 4040:4040/tcp
  volumes:
  - ./job:/home/jovyan/code


 mongo:
  image: mongo
  container_name: mongodb
  restart: always
  environment:
   MONGO_INITDB_ROOT_USERNAME: root
   MONGO_INITDB_ROOT_PASSWORD:
mongodb


 mongo-express:
  image: mongo-express
```

```
    container_name: mongo_express
    restart: always
    ports:
      - 8081:8081
    environment:

ME_CONFIG_MONGODB_ADMINUSERNAME:
root

ME_CONFIG_MONGODB_ADMINPASSWORD:
mongodb


  postgres:
    image: postgres:11.3
    container_name: postgres
    ports:
    - "5432:5432/tcp"
    restart: always
    environment:
      POSTGRES_PASSWORD: password
      POSTGRES_DB: report


  adminer:
    image: adminer
    restart: always
    ports:
      - 8085:8088
```

Non c'è molto da specificare nel file, vengono specificati i nomi delle immagini da cui avviare i container e poi vengono mappate alcune porte da i container al sistema host,

ma non è obbligatorio farlo, è necessario se ad esempio si vuole accedere ai vari servizi esternamente. Una delle funzioni utili di docker-compose è quella di poter assegnare un nome ai vari container. I container possono comunicare tra loro tramite il container_name anziché con l'IP (che non è noto a priori) e questo permette di avere un codice più flessibile, perché appunto il programmatore non deve ricavare l'ip dei vari container. Inoltre è possibile specificare ulteriori parametri da noi non specificati perché non necessari, ad esempio generare una overlay network per questa applicazione, oppure aumentare il numero di repliche per i vari container (quest'ultima opzione potrebbe essere utile se servissero più repliche di Spark ad esempio). Abbiamo utilizzato tutte le immagini ufficiali prelevate da DockerHub, per poter sfruttare una documentazione più completa e di maggiore affidabilità.

In fase iniziale, anziché usare Docker, avevamo utilizzato un'architettura differente. In particolare avevamo scelto di utilizzare *Spark on Yarn*, cioè Spark che si appoggia per la memorizzazione su HDFS e comunica con il cluster Hadoop tramite Yarn, il tutto installato localmente su una macchina. In effetti questo è possibile e facilmente configurabile dopo aver installato Spark e Hadoop. Inoltre questo tipo di configurazione di Spark è adatta per job batch, in quanto HDFS permette una maggiore affidabilità e resistenza ai guasti rispetto a Spark eseguito solo in memoria. Per quanto riguarda il codice dei nostri vari job e predizioni non cambia molto tra le due configurazioni, se non le prime righe per acquisire la sorgente dei dati, anziché leggerli da MongoDB, se vogliamo usare Spark on Yarn basta specificare nello script come sorgente un path su HDFS.

I nostri job sono stati eseguiti sia sull'architettura Spark on Yarn, sia sull'architettura contenitorizzata. Alla fine abbiamo scelto l'architettura con Docker essendo più portabile. Infatti basta avere una installazione di Docker Engine e il file docker-compose.yml per avviare e far funzionare correttamente questa architettura in qualsiasi macchina. Nel caso di Spark on Yarn servirà una macchina locale o un cluster in cui bisognerà configurare manualmente Hadoop e Spark, oltre ad installare Postgres per salvare i risultati dei job oppure salvare i risultati direttamente su HDFS.

Nell'architettura su container abbiamo deciso di utilizzare MongoDB come sistema di memorizzazione dei dati in input sia per avvicinarci all'architettura descritta nel paper su Pantheon, sia perché configurare Spark on Yarn su Docker non è l'ideale, non esistendo alcuna immagine ufficiale per HDFS (Hadoop su Docker è ancora sperimentale) anche perché in generale il sistema Hadoop e Spark sono più adatti a lavorare su dei cluster già

preconfigurati (ad esempio forniti come servizi cloud). Per quanto riguarda MongoDB invece è presente una immagine ufficiale su DockerHub che è anche configurabile ad esempio specificando il numero di repliche, come è possibile per tutti i container.

Per poter salvare i dati su MongoDB è stato necessario un piccolo adattamento dei dati stessi. Infatti tipicamente MongoDB usa un formato dati JSON per importare ed esportare i dati. Essendo i nostri dati in CSV li abbiamo convertiti in JSON con la libreria pandas di Python e poi importati su MongoDB tramite il comando **mongoimport**, in due collezioni diverse station (per i dati della stazione) e nodes (per i dati dai sensori). In generale questa conversione e importazione manuale non è necessaria perché nel caso generale ci sarà un processo software che trasferisce i dati prelevati dai sensori verso il sistema di memorizzazione scelto. Abbiamo deciso che ogni riga del CSV dovesse corrispondere ad un documento di una collezione nel seguente formato:

{ “colonna_1!”: valore_colonna_1, “colonna_2”: valore_colonna_2 , ...,”colonna_n”: valore_colonna_n }

Per convertire i csv in JSON e salvarli su MongoDB (una volta avviata l’applicazione) basta eseguire lo script **saveDatasetToMongoDB.sh** , purché si abbiano i due csv in una cartella “data” e purché si abbia un ambiente python con pandas.

ESECUZIONE DEI JOB CON SPARK

Con l’ausilio di Spark abbiamo fatto due job per calcolare informazioni che ipotizziamo possano essere significative nel contesto dell’agricoltura di precisione. Il dataset su cui si eseguono i job è quello relativo alle stazioni meteo ed è letto da MongoDB. I job si trovano nel notebook denominato ***SparkJobsSecondProject***.

Ecco cosa fanno i due job in questione:

- Job *Weekly_Report*: calcolo, su base settimanale, di report relativi ad un anno di informazioni su temperature e piogge;
- Job *Anomalies_Detection*: rilevamento di anomalie, ovvero punti del dataset in cui non sono stati rilevati dei valori dai sensori.

Job *Weekly_Report*

Sul sito *Arsial* sono disponibili i dati di temperature minime, massime e medie raccolte nelle zone agricole del Lazio. Traendo ispirazione da questi dati, abbiamo cercato di fare

un job che estraesse proprio questo tipo di informazioni dal dataset che abbiamo a disposizione.

L'obiettivo è rappresentare con un dataframe, in cui ogni riga corrisponde a un periodo temporale di una settimana, le seguenti informazioni: millimetri di pioggia totale, numero di giorni della settimana in cui ha piovuto, media delle temperature (medie, minime e massime), massimo delle temperature (minime e massime), minimo delle temperature (minime e massime).

La funzione che realizza il job è ***weekly_report*** e prende come parametro di input l'anno del quale si vogliono fare queste analisi dei dati settimanali. La strategia è quella di filtrare il dataframe *stazioneDf* togliendo le righe che non fanno parte dell'anno passato in input. Successivamente si eseguono due raggruppamenti con il comandi *groupBy*. Il primo raggruppamento lo si fa per giorni; il motivo di questo raggruppamento sui giorni è calcolare la somma dei millimetri di pioggia totale che sono caduti ogni giorno ed inoltre etichettare come 1 (ovvero True) i giorni che hanno superato una certa soglia, 0 (ovvero False) altrimenti. Abbiamo deciso di impostare la soglia maggiore o uguale a 2 mm. Vengono inoltre calcolate su base giornaliera il massimo delle temperature (minime e massime), il minimo delle temperature (minime e massime), la somma delle temperature (medie, minime e massime). Per il momento abbiamo calcolato la somma delle temperature e non la media perché questa verrà calcolata nella seconda *groupBy*; infatti sarebbe inesatto calcolarla ora. Il secondo raggruppamento è quindi quello che aggrega su base settimanale. Questa volta per le temperature si calcola la media per i valori medi, massimi e minimi. Si calcolano di nuovo le temperature massime, minime e i millimetri di pioggia totali, con in più anche il numero di giorni in cui ha piovuto nella settimana.

Nelle seguenti due immagini si può vedere come è fatto il dataframe, denominato *df_weekly,* risultante da questo job (la funzione ha preso come input l'anno 2019):

Settimana	Pioggia_Totale	Giorni_di_Pioggia	Temperatura_Media_Settimanale	Massima_Temperatura_Massima	Minima_Temperatura_Massima
1	0.0	0	4.924027777777777	12.98	-1.14
2	1.44	0	3.3936309523809522	11.45	-4.75
3	2.24	1	4.929493293591654	14.0	-1.98
4	15.520000000000007	4	4.232757575757576	12.44	-1.12
5	4.400000000000001	0	3.5415994020926753	11.46	-3.73
6	16.719999999999995	3	8.05752976190476	15.54	-0.86
7	0.0	0	7.591895522388061	18.12	-1.62
8	0.08	0	7.32372197309417	17.35	-1.02
9	0.04	0	8.48891207153502	20.42	0.59
10	0.16	0	10.170238095238094	19.48	1.96
11	0.44000000000000006	0	9.541222056631891	18.72	-2.29
12	0.76	0	10.285692995529061	21.64	1.58
13	0.04	0	11.399029850746267	23.57	1.82
14	5.640000000000002	1	10.912556053811656	23.15	-0.28
15	1.92	0	9.369388059701494	20.17	-0.55
16	1.72	0	10.857973174366618	23.28	2.64
17	3.0000000000000004	1	14.587544642857143	26.97	4.46
18	3.7600000000000007	1	11.715402985074627	21.32	3.6
19	7.520000000000003	1	10.902389867841412	20.8	1.35
20	6.480000000000001	1	12.005591451292247	21.39	5.78

Temperatura_Massima_Media	Massima_Temperatura_Minima	Minima_Temperatura_Minima	Temperatura_Minima_Media
5.174409722222221	12.46	-1.7	4.682673611111111
3.6338988095238096	10.82	-5.32	3.1701636904761896
5.20703427719821	13.64	-2.42	4.668360655737704
4.382757575757576	11.62	-1.33	4.098439393939394
3.74593423019432	9.98	-4.14	3.351315396113603
8.2403125	14.2	-1.4	7.887693452380951
7.865940298507463	17.58	-2.1	7.328223880597016
7.6161285500747375	16.67	-1.38	7.046188340807174
8.776557377049182	19.56	-0.3	8.213368107302534
10.420416666666666	18.24	1.27	9.935357142857143
9.79338301043219	17.56	-2.67	9.303889716840539
10.540342771982115	20.81	0.94	10.047779433681072
11.725417910447762	22.62	1.38	11.095985074626864
11.220149476831093	21.98	-0.94	10.612062780269058
9.725507462686567	18.33	-0.92	9.035597014925372
11.170357675111774	22.06	2.38	10.570014903129659
14.875416666666668	26.34	4.07	14.313958333333332
12.05849253731343	20.39	3.14	11.396343283582091
11.025192731277535	20.42	1.32	10.78225220264317
12.11133697813121	20.53	5.76	11.8987027833002

Avendo l'output di questo job in un formato perfettamente strutturato (di fatto è una relazione con tipi di dato primitivi) ed essendo costituita da un numero molto piccolo di righe (al più pari al numero di settimane in un anno cioè 52) abbiamo deciso di salvare questi dati su un database relazionale, cioè Postgres come spiegato alla sezione precedente. Spark permette con il seguente stralcio di codice molto compatto di caricare un dataframe su un database relazione. L'unico prerequisito è che sul sistema sia stato creato un database, di nome report, che viene creato specificando nel docker-compose.yml la variabile di ambiente **POSTGRES_DB** = report. Ovviamente deve essere presente nell'ambiente Spark il giusto driver JDBC, e noi lo abbiamo importato specificandolo direttamente nella configurazione della SparkSession, come si può vedere nel notebook. Il codice è il seguente :

```
driver = "org.sqlite.JDBC"

url="jdbc:postgresql://postgres/report"

properties = {

        "driver": "org.postgresql.Driver",

        "user": "postgres",

        "password": "password"

}

mode = 'overwrite'

df_weekly.write.jdbc(url=url, table="report_settimanali", mode=mode,
properties=properties)
```

Job *Anomalies_Detection*

Ispezionando la colonna *rad W/mq* del dataset, abbiamo notato che da un certo punto (il quinto giorno di giugno 2019) in poi (ovvero fino alla fine della tabella) non compaiono valori nelle celle della tabella. Questo genere di anomalie è possibile in questo contesto, poiché i dati sono presi da sensori, può capitare che delle volte dei valori non vengano rilevati. Se però l'intervallo in cui i valori non vengono rilevati copre un periodo temporale molto grande, ci può essere il sospetto che i sensori abbiano dei malfunzionamenti.

Ovviamente nel contesto dei Big Data le sorgenti sono molto grandi, quindi non sempre è possibile ispezionare “a mano” i dati come nel nostro caso e proprio per questo abbiamo deciso di definire un job Spark che in qualche modo riportasse in output un report sulle anomalie.

Per questo job il nostro obiettivo è quindi rilevare le anomalie del dataset, ovvero data una colonna della tabella, capire se sono presenti intervalli in cui non compaiono valori e quanto è grande l’intervallo temporale che occupano. La strategia è raggruppare le righe in **blocchi**, ciascuno dei quali indica (con un booleano) record consecutivi che hanno valori nulli (in questo caso si etichettano come True) o non nulli (False) su una certa colonna.

La funzione che realizza il job è ***anomalies_detection*** e prende come input il nome di una colonna del dataset. Per prima cosa si aggiungono quattro colonne ausiliarie al dataframe di partenza, che è il dataset. La prima colonna chiamata *isnull* dice per ogni riga il valore corrispondente della colonna di input è null (True) o no (False), poi si aggiunge la seconda colonna, *lag_isnull,* che sfrutta la funzione di pyspark sql lag e serve per trovare i bordi dei vari periodi di anomalie, cioè serve a capire per ogni riga se la precedente ha un valore nullo sulla colonna di input. La terza colonna, *change*, vale False se i corrispondenti valori di *isnull* e *lag_isnull* sono uguali (entrambi True o entrambi False), True altrimenti. Di fatto ogni riga con change pari a 1 indica la riga in cui si passa da una serie di record con valori nulli a una serie con valori non nulli o viceversa.

La quarta colonna, *block*, ha valori interi che aumentano in maniera incrementale: si parte da x=0; il valore di x rimane costante se il valore corrispondente nella colonna *change* è False, se invece è True, x aumenta di 1. Andando infine a fare una groupBy sulla colonna *block*, si possono individuare i blocchi contigui che hanno valori nulli e non nulli. Si è voluto anche evidenziare i timestamp di inizio (colonna *mindata*) e di fine (colonna *maxdata*) di questi blocchi, nonché la durata temporale (in giorni) che c’è tra i questi time stamp.

Nella seguente immagine c’è un esempio di cosa dà in output questa funzione:

```
df_anomaly = anomalies_detection('rad W/mq')

df_anomaly
```

block	mindata	maxdata	blocco_isnull	Duration
0	2018-10-12 14:25:15	2019-05-06 08:55:28	false	205.7709837962963
1	2019-05-06 13:40:10	2019-06-12 09:25:15	true	36.822974537037034

Preso in input il dataframe e una colonna mi restituisce un una lista di blocchi in cui è indicata la data di inizio e di fine di ogni blocco, un id e un valore blocco_isnull che è true se il blocco ha tutti valori nulli sulla colonna specificata in input. Come si può vedere dall'immagine c'è stato un blocco con tutti valori nulli dal 5 maggio al 6 giugno sulla radiazione solare e questo ci fa fortemente pensare ad una rottura del sensore sulla radiazione.

Anche in questo caso abbiamo scelto di caricare il dataframe su un db relazionale per le stesse ragioni elencate precedentemente, eseguendo il comando:

```
driver = "org.sqlite.JDBC"

url="jdbc:postgresql://postgres/report"

properties = {

        "driver": "org.postgresql.Driver",

        "user": "postgres",

        "password": "password"

}

mode = 'overwrite'

df_anomaly.write.jdbc(url=url, table="guasti", mode=mode, properties=properties)
```

CALCOLO DELL'EVAPOTRASPIRAZIONE

Come già detto l'evapotraspirazione è una misura stimabile tramite differenti formule empiriche, di vario genere e definite già varie decadi fa. Ci sono formule basate sulla temperatura, altre basate sulla radiazione solare, altre sui dati dei venti e altre ancora sulla base di un mix di parametri meteo. Per maggiori riferimenti si veda pag. 4 del paper già citato "*Using MARS, SVM, GEP and empirical equations for estimation of monthly mean reference evapotranspiration*". Abbiamo deciso di utilizzare la formula di Valiantzas, essendo una delle più recenti, ma anche affidabili formule. La formula è la seguente:

$$ET_0 = 0.0393\, R_s \sqrt{T + 9.5} - 0.19\, R_s^{0.6}\, \Phi^{0.15} + 0.048(T + 20)(1 - \tfrac{RH}{100})\, u_2^{0.7}$$

ET_0 sta appunto per evapotraspirazione, R_s è la radiazione solare, T è la temperatura media, RH è l'umidità relativa, u_2 è la velocità del vento a 2 metri e Φ è la latitudine. Come già accennato tutti questi dati sono presenti specificatamente nel dataset, a parte la latitudine che però è ovviamente stimabile sapendo che il campo di nocciole è situato a Caprarola.

Il problema dell'evapotraspirazione è il seguente: essa è difficilmente stimabile se non con strumenti di precisione molto sofisticati, detti lisimetri. Tuttavia le equazioni empiriche non sono esatte, sono appunto empiriche cioè approssimate. Il machine learning, tramite le tecniche di regressione permette di definire dei modelli, cioè delle nuove equazione che possano stimare l'evapotraspirazione, prendendo in input le varie colonne del dataset a scelta , in base a quelle che reputiamo essere le più rappresentative per influenzare la predizione. Nel nostro caso abbiamo usato l'evapotraspirazione di Valiantzas come etichetta per addestrare il nostro modello, non avendo i dati reali di evapotraspirazione. Nel caso ideale si hanno a disposizione i dati meteo e i dati di evapotraspirazione reali e quelli stimati con equazione empiriche e modelli di machine learning e si vogliono confrontare i valori delle equazioni empiriche con i valori stimati del machine learning per vedere quali delle due dà risultati che meglio si avvicinano al valore reale di evapotraspirazione. Il vantaggio del machine learning è quello di poter definire dei nuovi modelli che magari si comportino bene anche in assenza di alcuni parametri, cioè ad esempio si potrebbe definire un modello che stimi

l'evapotraspirazione senza la necessità del valore della velocità del vento e che magari dia comunque delle previsioni simili al valore reale.

Operazioni di data cleaning

Il dataset della stazione è già in un formato semistrutturato, essendo un CSV, tuttavia sono state necessarie delle operazioni di pulizia dei dati, in quanto ci siamo resi conto della presenza di valori nulli in alcuni campi. Infatti una volta letto il dataset con Spark come Spark dataframe, è possibile richiamare il metodo *isNull()* su una colonna specifica del dataframe combinato ad un conteggio per verificare quanti elementi nulli ci sono in una colonna.

Il dataset è stato letto come Spark dataframe, in quanto il connettore di MongoDB (che è la nostra fonte dati di input) per Spark restituisce in output un dataframe che rappresenta la collezione passata come parametro, nel nostro caso la collezione station che rappresenta i dati della stazione. Il dataframe è una struttura ottimizzata per Spark, essendo già strutturata come una tabella, dunque permette di eseguire delle operazioni più efficienti rispetto ad un classico RDD (Spark stesso si occupa in automatico di questa ottimizzazione).

Ci siamo resi conto che nella colonna della radiazione e in quella del vento erano presenti numerose entry nulle. Per quanto riguarda la radiazione solare è possibile stimare la stessa tramite una concatenazione di formule da noi analizzate nel paper FAO 56, tuttavia per poter calcolare le formule era necessario un altro dato purtroppo mancante e cioè le ore di sole durante il giorno . Abbiamo deciso dunque di stimare la radiazione solare in questo modo:

1. Calcolo del valore di radiazione media considerando solo record la cui ora è compresa tra le 5 e le 18;
2. Riempimento dei valori nulli con orario compreso tra le 18 e le 5 con 0, poiché di notte la radiazione è pressoché nulla;
3. Riempimento dei valori nulli con orario compreso tra le 5 e le 18 con la radiazione media calcolata al punto 1.

Ovviamente tale pulizia dei dati è migliorabile, però comunque valida in prima istanza. È migliorabile appunto stimando la durata delle ore di sole e applicando le formule specificate in FAO 56 tramite un altro sensore, oppure calcolando delle medie più

specifiche. Ad esempio per stimare la radiazione solare R_s si può prima stimare la radiazione extraterrestre R_a con questa formula, presente a pag. 46 di FAO 56:

$$Ra = \frac{1440}{\pi} G_{sc}\, dr\, [\, \omega s\, sin(\rho)\, sin(\sigma) + cos(\rho)\, sin(\omega s)]$$

Le variabili elencate in questa formula sono poi calcolabili tramite altre formule ancora (sempre empiriche) specificate. Si tratta essenzialmente di formule che dipendono dal giorno nell'anno e dalla latitudine della località del quale si vuole calcolare la radiazione.

Ad esempio ρ indica la latitudine e σ la declinazione solare. Infatti come è intuibile la radiazione solare dipenderà sia dalla posizione geografica della località, sia dal giorno dell'anno. Una volta stimata R_a si può applicare la radiazione R_s, però giornaliera , anziché ogni 15 minuti come nel nostro caso, con la seguente formula di Hargreaves:

$$R_s = k_{RS} \sqrt{(Tmax - Tmin)}\, R_a$$

K è una costante che dipende dalla posizione della locazione, è pari a 0.16 per località costiere e 0.19 per località più interne, cioè più lontane dal mare., mentre Tmax e Tmin sono le temperature medie e minime registrate nell'arco di un mese. Questo valore di radiazione è come già detto giornaliero, però nel nostro caso l'evapotraspirazione è stata calcolata su base oraria, quindi abbiamo deciso di usare la nostra stima semplificata.

In ogni caso nulla vieta di calcolare i valori di radiazione mancante tramite la formula di Hargreaves e poi calcolare l'evapotraspirazione su base giornaliera. Qualunque sia la scelta, trattandosi di calcoli lineari è facile aggiungere questo valore con Spark con le funzioni predefinite dei vari dataframe, ad esempio fillNa() che riempie automaticamente i valori nulli di una certa colonna.

Per quanto riguarda la colonna della velocità del vento, essendo questo valore più imprevedibile e casuale abbiamo deciso di riempire le colonne con una semplice media e questa probabilmente è l'unica scelta possibile, anche perché la velocità del vento non si può stimare a differenza della radiazione solare, se non prendendo dati di stazioni vicine (che nel nostro caso non erano presenti).

Modelli di predizione

Per stimare l'evapotraspirazione abbiamo sfruttato **SparkMLlib** , una libreria predefinita di Spark che presenta delle funzioni e delle classi predefinite per fare classificazione e regressione. Non abbiamo potuto usare gli specifici modelli del paper, in quanto sono modelli di predizione più complessi non implementati in SparkMLlib, ma abbiamo usato

dei modelli simili per effettuare la predizione. In particolare abbiamo tre modelli di regressione diversi:

1. Linear Regression
2. Decision Tree Regression
3. Gradient-boost tree (GBT) Regression

Come feature per addestrare i modelli abbiamo usato le stesse feature usate per calcolare la formula di Valiantzas, cioè temperatura media, umidità relativa, velocità del vento e radiazione solare. Abbiamo preso un 80% dei dati per addestrare il modello e il 20% per testarlo. Il pre-processing dei dati è stato spiegato alla sezione precedente, mentre per i dati già presenti non è stata necessaria nessuna particolare operazione di pulizia, se non la conversione della radiazione solare Rs da W/ m^2 a MJ/ day in quanto la formula di Valiantzas richiede il secondo tipo di unità di misura. Per fare questo basta modificare la radiazione per una costante, cioè 0.0864 come specificato in FAO 56. Abbiamo aggregato i valori per ora, anziché per minuti con una operazione *groupBy* di Spark, poiché generalmente l'evapotraspirazione si calcola giornalmente, o al più in maniera oraria, visto che i moderni sensori raccolgono i dati anche ora per ora, o come abbiamo visto nel nostro caso ogni 15 minuti. I dati sono stati aggregati facendo la media, come mostrato nel seguente codice:

```
eachHourDf = stazioneDf.select(["data_ora","wind_speed_media","temp1_media","rad W/mq","ur1_media"])\
.groupBy(to_date(stazioneDf['data_ora']).alias("data"),hour(stazioneDf['data_ora'])).agg({"wind_speed_media":"avg","rad W/mq":"avg","ur1_media":"avg"\
                                        ,"temp1_media":"avg"}).\
   withColumnRenamed('avg(rad W/mq)','Rs').withColumnRenamed('avg(temp1_media)','T')\
.withColumnRenamed('avg(ur1_media)','RH').withColumnRenamed('avg(wind_speed_media)','u2')
```

In ogni caso il modello che ha ottenuto un root mean square error (RMSE) più basso sui dati di test (0.60) è il modello GBT, tuttavia questo non significa che sia il migliore perché come abbiamo detto il suo output andrebbe confrontato con i valori reali. Quindi possiamo dire che in realtà il modello GBT è quello che approssima meglio la formula di Valiantzas. Inoltre è possibile ovviamente addestrare questi modelli con meno features, ad esempio solo con le temperature, oppure con temperatura e umidità oppure con

un’altra qualsiasi combinazione di feature e vedere come si comportano. Il tutto è molto semplice da fare, infatti la classe predefinita di SparkMLlib VectorAssembler prende in input un dataframe e una lista di colonne e ritorna una rappresentazione vettoriale delle stesse da dare in input al modello di regressione, quindi modificando la lista di colonne in input al VectorAssembler si possono aggiungere o rimuovere feature dal modello. Le predizioni dei vari modelli, ottenute con la funzione transform (test_set) di un generico modello sono dei dataframe, quindi possono essere salvate sul container su cui gira PostgreSQL come delle tabelle relazionali, con la stessa strategia con cui vengono salvati i report, se ad esempio qualcuno è interessato a confrontare valore predetto con valore empirico ottenuto dalla formula di Valiantzas.

Inoltre Spark MLlib permette di salvare i modelli addestrati tramite la funzione save, in modo tale da poterli riutilizzare in secondo momento, ad esempio fare nuove predizioni. Essendo rappresentati i modelli come dei file parquet, cioè un formato specifico per HDFS e Spark , abbiamo deciso di salvare i modelli nella stessa cartella in cui sono presenti i notebook jupyter. Questa cartella è condivisa tra l’host e il container di jupyter, in questo modo ogni volta che il container viene stoppato la cartella comunque mantiene inalterato il suo contenuto. In tal modo riavviando il container si potrà ricaricare il modello direttamente da questa cartella condivisa. La cartella viene montata grazie al parametro **volumes** nel *docker-compose.yml***.**

RISULTATI

Siamo riusciti ad implementare con successo una architettura per l’analisi in batch di dati provenienti agricoli. Siamo stati in grado di far comunicare tra loro diversi servizi, cioè un database noSQL (MongoDB), un motore di analisi (Spark) e un sistema relazionale (Postgres). Abbiamo ridotto al minimo lo sforzo di installazione grazie a Docker e fatto in modo di poter specificare l’architettura tramite il codice tramite l’approccio Infrastructure-As-Code, così da poter riprodurre questa architettura in più ambienti in maniera rapida.

Abbiamo implementato una architettura che può essere scalabile, aumentando le repliche dei vari servizi. Abbiamo inoltre realizzato dei job che possono essere utili come report per un esperto sul campo (agronomo) e che possono fornire delle predizioni utili (evapotraspirazione) per calcolare altri parametri.

CONCLUSIONI

È stata realizzata con successo una architettura per la gestione di Big Data nell'ambito dell'agricoltura di precisione. L'architettura è stata eseguita in locale, con il minimo numero di container, cioè 1 per ogni servizio, ma Docker comunque permette di aumentare il numero di repliche a piacimento per la scalabilità. Quindi questa architettura può essere adattata a maggiori carichi di lavoro facendone il deploy su un cluster, ad esempio, e aumentando il numero di repliche dei vari servizi e questo si può fare semplicemente modificando il file docker-compose, sul parametro *replicas*.

MongoDB permette poi di avere tante collezioni diverse ed eterogenee tra loro, quindi si potrebbero integrare nuove fonti di dati.

La scelta di un DB relazionale per salvare i report ci è sembrata coerente e crediamo possa esserlo pure definendo nuovi job, in quanto comunque i report generalmente sono dei riassunti strutturati dei dati input e sono molto sintetici.

I modelli di predizione attualmente vengono salvati in una cartella condivisa tra il container di Spark e la macchina locale, ma volendo si potrebbero salvare in un'altra fonte di storage e poi riutilizzarli in altre applicazioni, ad esempio un'interfaccia grafica che prende in input dei dati meteo e tramite il modello predice l'evapotraspirazione.

Inoltre l'evapotraspirazione, come già accennato, può essere integrata con altre misurazioni, ad esempio l'umidità del suolo, l'acqua irrigata sul campo e altri parametri per poter settare un sistema di regolazione automatica dell'irrigazione stessa. Non avendo trovato specifiche formule e non avendo abbastanza dati a disposizione non abbiamo potuto lavorare su questo ulteriore sviluppo.

I vari dati salvati possono poi essere presentati agli utenti (ad esempio gli agronomi) tramite uno strato di presentazione che prende in input ad esempio i dati dal db relazionale e li visualizza in qualche maniera.

FONTI

1) Arsial, sito regione Lazio, temperature settimanali
2) S. Mehdizadeh, J. Behmanesh, K. Khalili "Using MARS, SVM, GEP and empirical equations for estimation of monthly mean reference evapotranspiration"

3) R.Allen,L.Pereira et al. “FAO Irrigation and Drainage Paper NO.56”
4) K. Liakos, P. Busato et al. “Machine Learning in Agriculture: A Review”
5) L. Giustarini, S. Lamprecht, R. Retzlaff, T. Udelhoven, R. Torlone, G. Ulivi, A. Gasparri et al. “PANTHEON: SCADA for Precision Agriculture”
6) Spark MLlib Main Guide
7) Docker Compose
8) PySpark SQL module
9) MongoDB Spark connector
10) Working With Spark Dataframe & JDBC sources