

UNIVERSITÀ DEGLI STUDI DI NAPOLI "FEDERICO II"

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

CORSO DI BIG DATA ENGINEERING - LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

ANNO ACCADEMICO 2022-2023

HOMEWORK 3

Big Data Engineering

Acquisizione streaming dati da Open Meteo, memorizzazione su HDFS, analitiche con PySpark

Professore:

Ing. Vincenzo Moscato

Studenti:

Antonio Romano M63001315

Andriy Korsun M63001275

Giuseppe Riccio M63001314

Michele Cirillo M63001293

1 Raccolta e Preprocessing	1
1.1 Sorgente dei dati	1
1.1.1 Composizione API URL	1
1.1.2 I parametri meteo scelti	2
2 Configurazioni	3
2.1 Kafka	3
2.2 Hadoop	4
2.3 Spark	5
2.4 La configurazione completa	5
2.5 Architettura complessiva	8
2.5.1 Producer	8
2.5.1.1 Implementazione	9
2.5.2 Consumer	10
2.5.2.1 Implementazione	10
3 Analytics sullo stream dei dati	13
3.1 Visualizzazione del tempo corrente	14
3.1.1 Implementazione	14
3.1.2 Risultati	15
3.2 Andamento temperatura (reale ed apparente) in una finestra temporale	15
3.2.1 Implementazione	16
3.2.2 Risultati	16
3.3 Andamento umidità in una finestra temporale	17
3.3.1 Implementazione	17
3.3.2 Risultati	18
3.4 Andamento della velocità del vento in una finestra temporale	18
3.4.1 Implementazione	18
3.4.2 Risultati	19
3.5 Andamento della temperatura media giornaliera	20
3.5.1 Implementazione	20
3.5.2 Risultati	21
3.6 Andamento probabilità di pioggia e precipitazione effettiva	21
3.6.1 Implementazione	21
3.6.2 Risultati	24
3.7 Andamento della pressione atmosferica in una finestra temporale	24
3.7.1 Implementazione	25
3.7.2 Risultati	25
3.8 Esecuzione definitiva: Kafka	26
4 Conclusioni	27
4.1 Conclusioni	27
Elenco delle figure	28
Elenco delle tabelle	29
Bibliografia	30

1 Raccolta e Preprocessing

Contenuti

1.1	Sorgente dei dati	1
1.1.1	Composizione API URL	1
1.1.2	I parametri meteo scelti	2

L'Homework richiede, tramite Apache Kafka, di gestire l'acquisizione di uno stream di dati (e.g., messaggi da un web server, misure da una rete di sensori, etc.) da una data sorgente (anche simulata) e la successiva memorizzazione in un file di log su HDFS. Utilizzando poi uno degli strumenti del primo homework (pyspark, pig o hive), effettuare delle query sullo stream di dati, mostrandone i risultati su un'apposita dashboard.

1.1 Sorgente dei dati

Il data source scelto per l'Homework 3 è **Open Meteo**¹; questa API utilizza le previsioni meteo globali. Nel caso in esame **si è scelto NOAA GFS**². Si raccolgono dati meteo per le città più importanti di **Italia**. Per il modello **GFS**, l'aggiornamento delle previsioni è effettuato ogni 6 ore, il meteo corrente ogni ora.

Modello meteorologico	Regione	Risoluzione spaziale	Risoluzione temporale	Lunghezza giorni previsione	Frequenza di aggiornamento
GFS	Globale	0.11 ° (13 km)	Hourly, 3-hourly after 120 hours	16 days	Every 6 hours

Tabella 1.1: Data Source: Open-Meteo GFS

Open-Meteo.com, infatti, è un servizio online che fornisce dati meteorologici e previsioni utilizzando modelli meteorologici globali. Offre una vasta gamma di informazioni tra cui temperature, umidità, velocità del vento, precipitazioni e altro ancora. Gli utenti possono accedere ai dati meteorologici tramite una semplice API e utilizzarli per scopi come l'analisi dei dati, la creazione di applicazioni o la visualizzazione delle previsioni meteorologiche.

1.1.1 Composizione API URL

L'endpoint API accetta una coordinata geografica, un elenco di variabili meteorologiche e risponde con previsioni meteo orarie in formato **JSON** fino a 16 giorni.

Come si può notare dal seguente URL, vengono indicate le variabili scelte:

API URL `https://api.open-meteo.com/v1/gfs?latitude=40.88&longitude=14.52&hourly=temperature_2m,relativehumidity_2m,apparent_temperature,pressure_msl,precipitation,precipitation_probability,visibility,windspeed_10m¤t_weather=true&past_days=3&forecast_days=3&timezone=auto`

¹<https://open-meteo.com/en/docs/gfs-api>: API meteo forecasting free

²Modello di Global Forecast System

Parametro	Formato	Default	Descrizione
latitudine, longitudine	Virgola mobile		Coordinate geografiche WGS84 della posizione
elevation	Virgola mobile		Elevazione utilizzata per il ridimensionamento statistico.
hourly	Matrice di stringhe		Un elenco di variabili meteorologiche che devono essere restituite.
daily	Matrice di stringhe		Un elenco di aggregazioni variabili meteorologiche giornaliere.
current.weather	Bool	true/false	Includi le condizioni meteorologiche correnti nell'output JSON.
temperature.unit	String	Celsius	Temperatura
windspeed.unit	String	kmh	Altre unità di velocità del vento: ms, mph e kn
precipitation.unit	String	millimetro	Altre unità di quantità di precipitazioni: inch
timeformat	String	iso8601	Formato orario giornaliero
timezone	String	GMT	I dati vengono restituiti a partire alle 00:00 ora locale.
past.days	Numero intero	0	Se past.days è impostato, è possibile restituire i dati meteo passati.
forecast.days	Numero intero (0-16)	7	Sono possibili fino a 16 giorni di previsione.
start_date end_date	Stringa (aaaa-mm-gg)		Intervallo di tempo per ottenere i dati meteorologici.
cell.selection	String	terra	

Tabella 1.2: Composizione dell'API URL

1.1.2 I parametri meteo scelti

Nel caso in esame, si è voluto scegliere soltanto alcune variabili meteo al fine di costruire un'analisi previsionale e storica settimanale semplice.

Variabile	Unità	Descrizione
temperature_2m	°C (°F)	Temperatura dell'aria a 2 metri dal suolo
relativehumidity_2m	%	Umidità relativa a 2 metri dal suolo
apparent_temperature	°C (°F)	Temperatura percepita calcolata con combinazione vento, umidità e radiazione solare
windspeed_10m	km/h (mph, m/s, nodi)	Velocità del vento a 10 o 80 metri dal suolo.
meteocode	Codice WMO	Condizioni meteorologiche come codice numerico.
precipitation	mm (pollici)	Precipitazione totale (pioggia, rovesci, neve) somma delle ore precedenti
precipitation_probability	%	La probabilità di precipitazione si basa su modelli meteorologici.
visibility	Metri	Distanza di visualizzazione in metri. Influenzato da nuvole basse, umidità. Max 24 km
pressure_msl	hPa	Pressione atmosferica ridotta al livello medio del mare (msl)

Tabella 1.3: Parametri scelti

Contenuti

2.1	Kafka	3
2.2	Hadoop	4
2.3	Spark	5
2.4	La configurazione completa	5
2.5	Architettura complessiva	8
2.5.1	Producer	8
2.5.2	Consumer	10

La configurazione seguente è stata fatta su una macchina Windows avente la Windows Subsystem Linux (WSL). Vengono installate opportunamente e interconnesse le piattaforme: Kafka, Hadoop e Spark

2.1 Kafka

Apache Kafka è una piattaforma di **Big Data Streaming** distribuita, ad alta velocità e scalabile, progettata per la gestione di flussi di dati in tempo reale. Funziona come un sistema di messaggistica public-subscriber, in cui i produttori (**publisher**) di dati inviano i messaggi ai topic e i consumatori (**subscriber**) possono leggere da uno o più topic.

È in grado di gestire flussi di dati ad alta velocità e garantire una bassa latenza, rendendolo adatto per applicazioni in cui è necessario elaborare e analizzare grandi quantità di dati in tempo reale, come la gestione di eventi, l'elaborazione di log, l'analisi dei dati in streaming e altro ancora. Di seguito verrà analizzata la sua architettura:

- **Producer:** rappresenta un'applicazione o un sistema che genera e invia i messaggi ai Kafka Topic. I producer scrivono i messaggi in uno o più topic specificati;
- **Topic:** è una categoria o un canale a cui vengono pubblicati i messaggi; ogni topic è suddiviso in una o più partizioni proprio per renderlo strutturato e leggibile il più possibile;
- **Partition:** una sequenza ordinata e immutabile di messaggi all'interno di un topic. Ogni partizione può essere distribuita su diversi nodi di un cluster Kafka. L'utilizzo di partizioni consente di scalare orizzontalmente la capacità di elaborazione dei messaggi;
- **Consumer:** un'applicazione o un sistema che legge ed elabora i messaggi da uno o più topic;
- **Broker:** un server Kafka che gestisce la memorizzazione e la replica dei dati. Ogni broker è responsabile di una o più partizioni dei topic e funge da intermediario tra i producer e i consumer;
- **ZooKeeper:** è un sistema di coordinamento distribuito utilizzato da Kafka per la gestione delle informazioni di configurazione e il mantenimento dello stato dei nodi del cluster. ZooKeeper tiene traccia dei broker attivi, delle partizioni dei topic e degli offset dei consumer.

2.2 Hadoop

Come già fatto nell'**HW1**, si descrive **Hadoop**. Esso è un framework software open source che consente l'elaborazione distribuita di grandi set di dati su cluster. È progettato per scalare da singoli server a migliaia di macchine, ognuna delle quali offre capacità di calcolo e storage locali. Il core di Hadoop è basato su dei componenti principali, quali:

1. **Hadoop Distributed File System (HDFS):** Rappresenta il file system distribuito che archivia grandi quantità di dati sui nodi del cluster. Proprietà principali dell'HDFS sono:
 - **Architettura distribuita:** I dati in HDFS sono suddivisi in blocchi di dimensioni fisse (di solito 128 MB) che sono distribuiti su tutti i nodi nel cluster Hadoop. Ciò rende i dati processabili in parallelo, aumentando l'efficienza e la velocità di elaborazione;
 - **Fault Tolerance:** HDFS è progettato per resistere ai guasti;
 - **Scalabilità.**
2. **Hadoop MapReduce:** Modello di programmazione per l'elaborazione di grandi volumi di dati in parallelo. Funziona dividendo un set di dati in piccole parti che possono essere elaborate in parallelo su diversi nodi del cluster:
 - **Fase Map:** Viene preso un set di dati di input e convertito in un altro set di dati, dove gli elementi individuali sono suddivisi in coppie chiave/valore. In questa fase, i dati vengono filtrati e ordinati;
 - **Fase Reduce:** L'output della fase Map viene ulteriormente elaborato. Vengono prese le coppie chiave/valore dalla fase Map e combinate in un set più piccolo di coppie chiave/valore. L'output della fase Reduce non è altro che un set di dati che ha subito una sorta di aggregazione o sintesi rispetto all'input originale, costituito da coppie chiave/valore (proprio come l'output della fase Map) ma con un numero molto ridotto di chiavi uniche.

Le principali proprietà di Hadoop, alcune ripetibili a partire dalle proprietà dei suoi componenti principali precedentemente enunciate, sono:

- **Scalabilità:** Hadoop è progettato per essere estremamente scalabile. Può eseguire operazioni su un gran numero di nodi del cluster e processare una grossa mole di dati;
- **Fault Tolerance:** Hadoop è in grado di rilevare e gestire i guasti. Se un nodo fallisce durante l'esecuzione di un'attività, quella parte del lavoro viene automaticamente spostata su un altro nodo per prevenire l'interruzione dell'elaborazione;
- **Flessibilità di dati e dello storage:** Hadoop non richiede la pre-elaborazione dei dati per salvarli. Può gestire qualsiasi tipo di dato, strutturati e non strutturati, come testo, immagine, audio, video;
- **Velocità di elaborazione:** Hadoop è in grado di processare una grande mole di dati molto rapidamente poiché è in grado di elaborarli in parallelo, grazie anche alla capacità di eseguire compiti su più nodi del cluster;
- **Data Protection:** Viene garantita la protezione dei dati grazie alle proprietà di redundancy e fault tolerance. È possibile, inoltre, replicare i dati sui vari nodi della rete, prestando attenzione alla consistenza di essi;
- **Flessibilità rispetto ai sistemi operativi:** Qualunque sistema operativo utilizzino i vari nodi va sempre bene dato che Hadoop non richiede una conoscenza di alto livello del sistema operativo.

2.3 Spark

Anche in questo caso, si avvale della teoria già descritta nell'HW1. **Apache Spark** è un framework open-source di elaborazione di dati distribuiti, che fornisce un'interfaccia per programmare cluster con API parallele e richiede dati distribuiti. È stato sviluppato per essere veloce e generale, consentendo di sviluppare applicazioni su larga scala per l'elaborazione ed il processamento di flussi di dati. Spark viene utilizzato per diversi motivi, tra cui:

- **Scalabilità:** gestione di grandi volumi di dati distribuiti su un cluster di nodi, sfruttando la potenza di calcolo di computer per elaborare e analizzare dati;
- **Velocità:** grazie alla sua architettura di memoria in-cache, Spark è in grado di elaborare dati molto più velocemente rispetto ad altri framework di elaborazione di dati distribuiti come Hadoop MapReduce;
- **Flessibilità:** supporto a diversi linguaggi di programmazione (Scala, Java, Python e R); offre diverse librerie per l'analisi di dati, come MLlib per il machine learning, GraphX per il calcolo su grafi e Spark Streaming per l'elaborazione di flussi di dati in tempo reale;
- **Facilità d'uso:** le API di Spark sono facili da usare e consentono di scrivere codice più pulito e conciso.

Nel caso in esame, **non viene utilizzata** la manipolazione degli RDD descritta nell'HW1 ma viene utilizzato PySpark per condurre le attività di analisi dei dati. Il concetto alla base del modello di programmazione di PySpark è quello di utilizzare un insieme di operazioni per manipolare i dati attraverso gli RDD (Resilient Distributed Dataset). Gli RDD rappresentano un insieme di dati distribuiti e immutabili e costituiscono l'astrazione principale di PySpark.

Tuttavia, nel caso specifico, non si fa uso diretto degli RDD e delle trasformazioni e azioni disponibili nelle API di Spark, ma viene utilizzata la vasta gamma di funzionalità per l'elaborazione e l'analisi dei dati in modo distribuito e scalabile che esso mette a disposizione.

2.4 La configurazione completa

Si descrive di seguito il procedimento di configurazione completa delle piattaforme **Hadoop, Kafka e Spark**. Per la guida completa, si rimanda alla **pagina Medium** [1] contenente l'articolo sviluppato dagli autori del presente elaborato.

– Installazione di Hadoop

– Installazione di Java

- *Aggiornare il sistema operativo:*
`sudo apt-get update`
- *Installare Java versione 8:*
`sudo apt-get install openjdk-8-jdk`

– Installazione di SSH

- *Installare SSH:*
`sudo apt-get install ssh`
- *Avviare il servizio SSH:*
`sudo service ssh start`

- Download di **Hadoop**
 - Scaricare Hadoop-3.3.2:
`wget https://dlcdn.apache.org/hadoop/common/hadoop-3.3.2/hadoop-3.3.2.tar.gz`
 - Estrarre l'archivio:
`tar -xvzf hadoop-3.3.2.tar.gz`
 - Rinominare la cartella:
`mv hadoop-3.3.2 hadoop`
- Preparazione dell'esecuzione
 - Trovare il percorso di installazione di Java:
`dirname $(dirname $(readlink -f $(which java)))`
 - Modificare il file `etc/hadoop/hadoop-env.sh` inserendo il percorso di Java:
`export JAVA_HOME=<percorso ottenuto precedentemente>`
 - Eseguire il comando:
`bin/hadoop`
- **Configurazione** in modalità pseudo-distribuita, utilizzare il comando `nano` per ogni modifica
 - File `etc/hadoop/core-site.xml`

```
<configuration>
  <property>
    <name>fs.defaultFS</name>
    <value>hdfs://localhost:9000</value>
  </property>
</configuration>
```
 - File `etc/hadoop/hdfs-site.xml`

```
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
  <property>
    <name>dfs.webhdfs.enabled</name>
    <value>true</value>
  </property>
  <property>
    <name>dfs.permissions</name>
    <value>>false</value>
  </property>
</configuration>
```
- Avvio di **Hadoop**
 - Formattazione del file system:
`bin/hdfs namenode -format`

- Avvio del servizio HDFS:

```
sbin/start-dfs.sh
```

– Installazione di Kafka

- Configurazione di **JAVA_HOME**

- Modificare il file `~/.bashrc`:

```
nano ~/.bashrc
```

- Aggiungere la seguente riga alla fine del file:

```
export JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-amd64
```

- Salvare il file e applicare le modifiche:

```
source ~/.bashrc
```

- Download del pacchetto binario di **Kafka**

- Scaricare Kafka-3.2.3:

```
wget https://dlcdn.apache.org/kafka/3.2.3/kafka_2.13-3.2.3.tgz
```

- Estrarre l'archivio:

```
tar -xvzf kafka_2.13-3.2.3.tgz
```

- Rinominare la cartella:

```
mv kafka_2.13-3.2.3 kafka
```

- Avvio dell'ambiente **Kafka**

- Avviare il servizio ZooKeeper:

```
$KAFKA_HOME/bin/zookeeper-server-start.sh
```

```
$KAFKA_HOME/config/zookeeper.properties
```

- Aprire un nuovo terminale WSL e avviare il server Kafka:

```
$KAFKA_HOME/bin/kafka-server-start.sh
```

```
$KAFKA_HOME/config/server.properties
```

A valle dell'avvio dell'ambiente **Kafka** è possibile eseguire prima il **consumer.py** e poi il **producer.py** nel caso in cui si è scelto di implementare entrambi in Python.

– Installazione di Spark

- Download del pacchetto binario di **Spark**

- Scaricare Spark 3.2.4 senza Hadoop pre-compilato:

```
wget https://dlcdn.apache.org/spark/spark-3.2.4  
/spark-3.2.4-bin-without-hadoop.tgz
```

- Estrarre l'archivio:

```
tar -xvzf spark-3.2.4-bin-without-hadoop.tgz
```

- Rinominare la cartella:

```
mv spark-3.2.4-bin-without-hadoop spark
```

- Configurazione delle variabili di ambiente di **Spark**

- Modificare il file `~/.bashrc`:

```
nano ~/.bashrc
```

- Aggiungere le seguenti righe alla fine del file:

```
export SPARK_HOME=<percorso dove si trova Spark>
export PATH=$SPARK_HOME/bin:$PATH
export SPARK_DIST_CLASSPATH=$(hadoop classpath)
```
- Salvare il file e applicare le modifiche:

```
source ~/.bashrc
```
- Configurazione predefinita di **Spark**
 - Creare il file di configurazione predefinito di Spark:

```
cp $SPARK_HOME/conf/spark-defaults.conf.template
$SPARK_HOME/conf/spark-defaults.conf
```
 - Modificare il file per aggiungere alcune configurazioni:

```
nano $SPARK_HOME/conf/spark-defaults.conf
```
 - Assicurarsi di aggiungere la seguente riga alla fine del file:

```
spark.driver.host localhost
```
 - Copiare i file di configurazione di Hadoop (core-site.xml e hdfs-site.xml) nella cartella di configurazione di Spark:

```
cp $HADOOP_HOME/etc/hadoop/core-site.xml $SPARK_HOME/conf/
cp $HADOOP_HOME/etc/hadoop/hdfs-site.xml $SPARK_HOME/conf/
```

La configurazione a questo punto è pronta, si può passare all'esecuzione del **producer.py**, del **consumer.py** e dello **script** di raccolta dei dati immagazzinati dall'HDFS e Analytics per la visualizzazione dei risultati. Quest'ultimo procedimento è stato fatto con la piattaforma **Streamlit**.

2.5 Architettura complessiva

Configurando opportunamente la WSL come fatto negli step della configurazione, è possibile eseguire il workflow:

Sorgente API $\xrightarrow{\text{Cattura}}$ **Kafka** $\xrightarrow{\text{Salva}}$ **HDFS** $\xrightarrow{\text{Mostra}}$ **Dashboard**

2.5.1 Producer

A partire dall'**API sorgente**, raccoglie i dati e li invia sotto forma di "messaggio" al Topic. Più precisamente, il seguente script ha l'obiettivo di leggere un elenco di città da un file JSON, ottenere i dati meteorologici per ogni città da un'API di open-meteo, e inviare questi dati a un topic Kafka:

- Viene letto il file 'capoluoghi.json', poi caricato il suo contenuto in un dizionario capoluoghi;
- Viene creato un oggetto Session di Requests. Le sessioni sono utilizzate per persistere parametri tra le richieste;
- Viene creato un oggetto KafkaProducer, che si connette al server Kafka in esecuzione su 'localhost:9092'. Questo producer utilizzerà un value_serializer che convertirà ogni messaggio in una stringa JSON e lo codificherà in UTF-8 prima di inviarlo;
- Per ogni elemento nel dizionario 'capoluoghi', vengono eseguite le seguenti operazioni:
 - Vengono estratte le informazioni di nome, latitudine e longitudine;

- Viene costruito un URL API per open-meteo.com utilizzando le coordinate di latitudine e longitudine. Questa API restituirà dati meteorologici per la posizione specificata (in base al capoluogo selezionato);
- Viene inviata una richiesta GET a questa URL;
- Vengono estratti i dati JSON dalla risposta;
- Vengono infine inviati questi dati al topic Kafka specificato;
- Terminato il ciclo, un ultimo messaggio è uguale a "EOF".

2.5.1.1 Implementazione

Di seguito l'implementazione del Producer in **Python**:

```
Spark
Spark

KAFKA_TOPIC = 'MeteoTopic'

with open('capoluoghi.json') as file:
    capoluoghi = json.load(file)

with requests.Session() as s:
    producer = KafkaProducer(bootstrap_servers=['localhost:9092'],
                             value_serializer=lambda x: dumps(x).encode('utf-8'))

    for capoluogo in capoluoghi['capoluoghi']:
        print(capoluogo)
        nome = capoluogo['nome']
        latitudine = capoluogo['latitudine']
        longitudine = capoluogo['longitudine']

        API_URL =
            f"https://api.open-meteo.com/v1/gfs?latitude={latitudine}&longitude={longitudine}&hourly=temperature_2m,relativehumidity_2m,apparent_temperature,presure_msl,precipitation,precipitation_probability,visibility,windspeed_10m&current_weather=true&past_days=3&forecast_days=3&timezone=auto"

        response = s.get(API_URL)
        data = response.json()

        producer.send(KAFKA_TOPIC, value=data)

    producer.send(KAFKA_TOPIC, 'EOF')
```

Tabella 2.1: Implementazione Producer - Spark

2.5.2 Consumer

Il Consumer rimane in attesa del Producer che invia il "messaggio" e li carica nell'**HDFS di Hadoop**. Più precisamente, il seguente script ha l'obiettivo di leggere un elenco di nomi di città da un file JSON, quindi leggere i messaggi da un topic Kafka. Ogni messaggio viene scritto in un nuovo file JSON in HDFS, il cui nome corrisponde al nome della città nell'elenco:

- Viene creato un oggetto `KafkaConsumer` che si connette a un server Kafka in esecuzione su 'localhost:9092' e si sottoscrive al topic 'MeteoTopic';
- Viene definito il percorso di base in HDFS dove i dati verranno scritti;
- Viene letto il file 'capoluoghi.json' e salvato il suo contenuto nella variabile `capoluoghi`;
- Vengono inseriti i valori associati alla chiave 'nome' in `nomi_città`;
- Iterazione sui messaggi nel Consumer con l'uso di un contatore (sarà utilizzato per tracciare il numero di messaggi consumati da Kafka); Per ogni messaggio:
 - Viene generato il percorso HDFS completo, includendo il nome della città corrente (preso dalla lista `nomi_città`) e l'estensione '.json'. Viene successivamente aperto, in modalità scrittura, un file con il percorso generato;
 - Viene scritto il messaggio decodificato nel file, ed incrementato poi il contatore;
 - Chiude il file HDFS;
 - Il ciclo termina quando il messaggio decodificato è uguale a "EOF".

2.5.2.1 Implementazione

Di seguito l'implementazione del Consumer in **Python**:


<div><div> Spark</div><pre>consumer = KafkaConsumer('testTopic', bootstrap_servers=['localhost:9092']) hdfs_base_path = 'hdfs://localhost:9000/nome/bigdata/' with open('capoluoghi.json') as file: capoluoghi = json.load(file) nomi_citta = [] for capoluogo in capoluoghi['capoluoghi']: nomi_citta.append(capoluogo['nome']) count = 0 for message in consumer: values = message.value.decode('utf-8') if values == '"EOF"': break hdfs_path = f"{hdfs_base_path}{nomi_citta[count]}.json" with hdfs.open(hdfs_path, 'wt') as f: f.write(values) count += 1 f.fs.close()</pre></div>
--

Tabella 2.2: Implementazione Consumer - Spark

Inoltre, viene creato uno script aggiuntivo per l'analisi dei flussi di dati ricevuti utilizzando **PySpark**. Su questo script viene creata una dashboard utilizzando **Streamlit**. Nel capitolo delle Analytics, verranno mostrate e discusse ognuna di esse, con una breve descrizione ed implementazione.

Nella Figura sottostante, si mostra l'architettura complessiva realizzata per l'esecuzione di questo Homework:

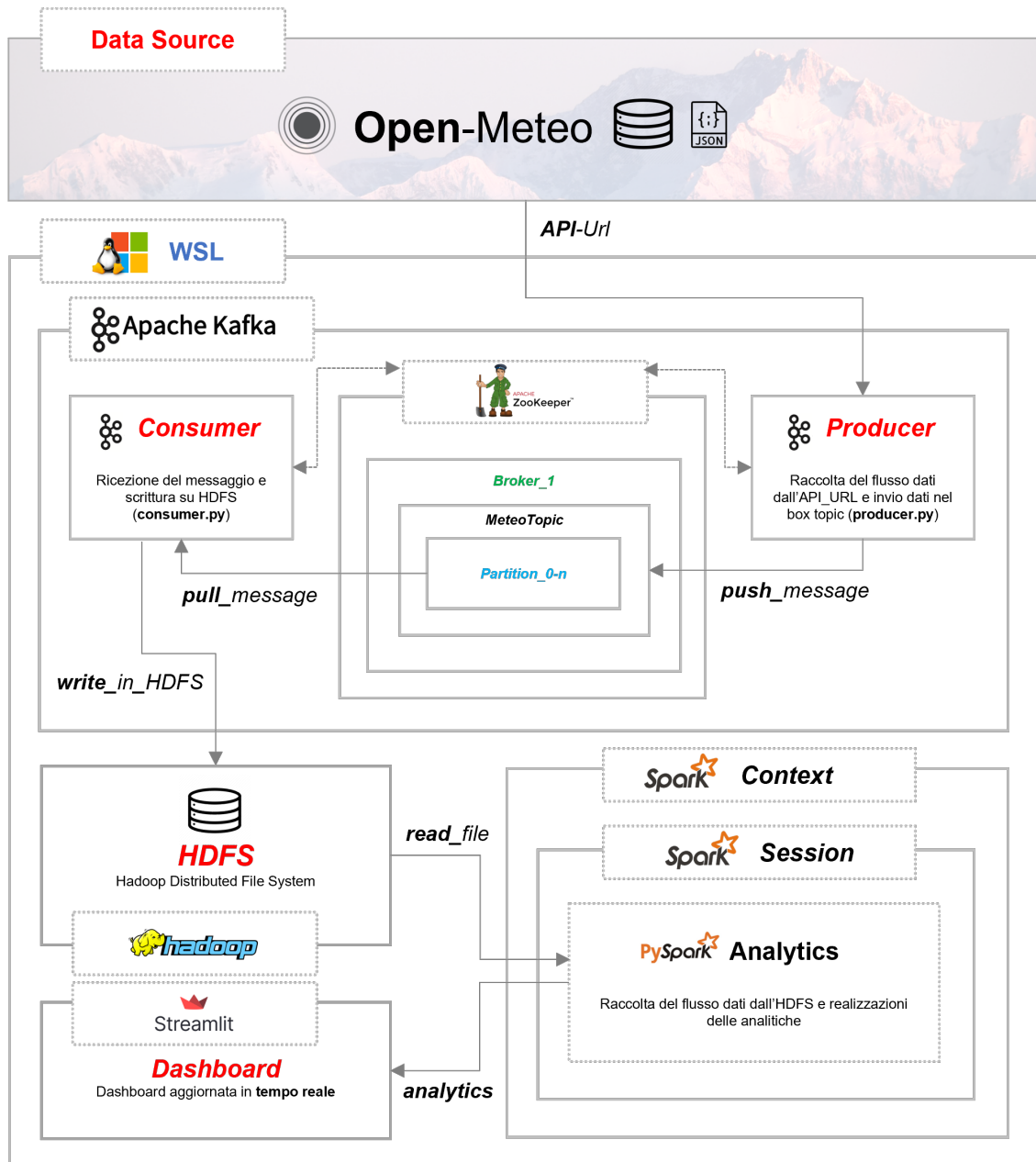


Figura 2.1: Architettura complessiva - Kafka, Hadoop, Spark, Streamlit

3 Analytics sullo stream dei dati

Contenuti

3.1	Visualizzazione del tempo corrente	14
3.1.1	Implementazione	14
3.1.2	Risultati	15
3.2	Andamento temperatura (reale ed apparente) in una finestra temporale	15
3.2.1	Implementazione	16
3.2.2	Risultati	16
3.3	Andamento umidità in una finestra temporale	17
3.3.1	Implementazione	17
3.3.2	Risultati	18
3.4	Andamento della velocità del vento in una finestra temporale	18
3.4.1	Implementazione	18
3.4.2	Risultati	19
3.5	Andamento della temperatura media giornaliera	20
3.5.1	Implementazione	20
3.5.2	Risultati	21
3.6	Andamento probabilità di pioggia e precipitazione effettiva	21
3.6.1	Implementazione	21
3.6.2	Risultati	24
3.7	Andamento della pressione atmosferica in una finestra temporale	24
3.7.1	Implementazione	25
3.7.2	Risultati	25
3.8	Esecuzione definitiva: Kafka	26

In questo capitolo, verranno discusse le **Analytics** realizzate per la visualizzazione, valutazione ed analisi su quelle che sono le condizioni climatiche dei capoluoghi italiani. Tramite una dashboard realizzata in **Streamlit**, è possibile selezionare ogni capoluogo nel caso si volessero visionare informazioni su ognuno di essi presi singolarmente. Si ricorda, brevemente, che le informazioni che verranno mostrate sulla dashboard includono:

- **temperature_2m**: temperatura dell'aria a 2 metri dal suolo;
- **relativehumidity_2m**: umidità relativa dell'aria a 2 metri dal suolo;
- **apparent_temperature**: temperatura percepita;
- **windspeed_10m**: velocità del vento a 10 metri dal suolo;
- **precipitation**: quantità totale di precipitazione;
- **precipitation_probability**: probabilità di precipitazione;
- **pressure_msl**: pressione atmosferica ridotta al livello medio del mare (msl).

Si noti: I risultati delle successive Analytics si basano su dati raccolti il giorno **29/05/2023 alle ore 09:00** con 3 giorni di forecasting e 3 giorni passati.

3.1 Visualizzazione del tempo corrente

La prima Analytic ha come obiettivo quello di visualizzare il tempo corrente, che ci fornisce una rappresentazione visiva immediata delle condizioni meteorologiche attuali mostrata a lato della dashboard.

3.1.1 Implementazione

Di seguito l'implementazione in **PySpark**:

```
Spark

# Seleziona la citta'
info_meteo = openSpark(citta_scelta)

# Ottiene la emoji relativa al Weather code nel JSON
def get_weather_info(weather_value):
    if weather_value == 0:
        emoji = "emoji1"
        title = "Cielo sereno"
    . . .
    return emoji, title
# Mostra l'emoji ed il titolo del meteo corrente
weather_emoji, weather_title = get_weather_info(weather_curr_value)

# Ottiene le emoji del vento corrente
def get_wind_info(wind_value):
    if wind_value == 0:
        emoji = "emoji1"
        title = "Assenza di vento"
    . . .
    return emoji, title
#Mostra l'emoji ed il titolo del vento corrente
wind_emoji, wind_title = get_wind_info(vento_curr_value)

# Ottiene le info sul tipo di vento
def get_wind_type(wind_direction):
    if 337.5 <= wind_direction <= 22.5:
        return "Tramontana - N"
    . . .
    else:
        return "Unknown"
#Mostra il tipo del vento corrente
wind_type = get_wind_type(dir_vento_curr_value)
```

Tabella 3.1: Visualizzazione del tempo corrente - Spark

3.1.2 Risultati

Nella Figura successiva, vengono mostrati i risultati ottenuti all'esecuzione della prima Analytic. Al lato sinistro, viene mostrata l'interfaccia della sidebar che permette di scegliere, attraverso il menù a discesa, tra tutti i capoluoghi italiani. Una volta scelto, verranno mostrate, nel lato destro, le informazioni climatiche relative a quella specifica città.

In questo caso specifico, vengono mostrate le condizioni climatiche a partire dall'ultimo aggiornamento del 2023-05-29T09:00, il cui meteo rileva cielo Sereno, con temperatura di 23.1 °C, Velocità del Vento di 2.2 Km/h (Vento moderato) la cui direzione è di un'angolazione di 189.0° Ostro - S, ovvero soffia da sud verso ovest.

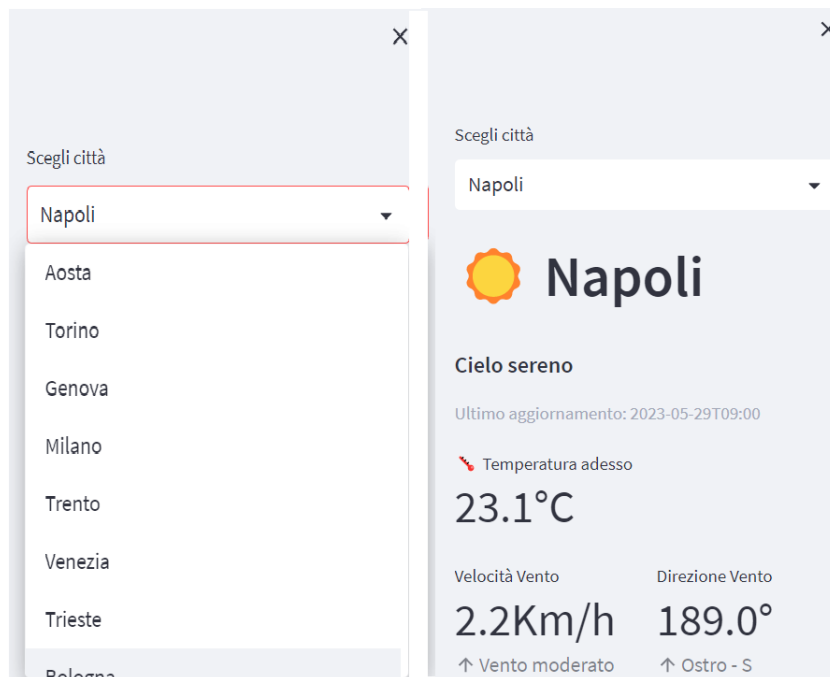


Figura 3.1: Sidebar - Visualizzazione del tempo corrente

3.2 Andamento temperatura (reale ed apparente) in una finestra temporale

L'obiettivo della seconda Analytic è quello di mostrare l'andamento temperatura e temperatura apparente (percepita) in una finestra temporale. Consente inoltre di esaminare le variazioni di queste due grandezze nel corso del tempo e di individuare eventuali pattern o tendenze significative.

In particolare, l'Analytic aiuta a capire come l'impatto dell'umidità, della velocità del vento e di altri fattori meteorologici influenzano la temperatura che l'umano percepisce rispetto ad un sensore.

Ad esempio, se ci fosse alta umidità e bassa temperatura, l'aria verrebbe percepita come pesante e la traspirazione potrebbe essere ostacolata, causando una sensazione di calore superiore rispetto alla temperatura reale.

3.2.1 Implementazione

Di seguito l'implementazione in **PySpark**:

```
Spark

# Esegue la query per ottenere l'andamento della temperatura nel tempo
result_df = info_meteo.select(
    'hourly.time', 'hourly.temperature_2m',
    'hourly.apparent_temperature')

# Converte il risultato in un DataFrame pandas
result_pd = result_df.toPandas()

# Visualizza il line chart utilizzando Plotly Express
fig = px.line(result_pd, x=result_pd['time'][0],
              y=[result_pd['temperature_2m'][0],
                 result_pd['apparent_temperature'][0]],
              title='Andamento della temperatura nel tempo')

# Aggiunge le etichette alle linee plottate
newnames = {'wide_variable_0': 'Temperatura', 'wide_variable_1':
            'Temperatura Apparente'}
fig.for_each_trace(lambda t: t.update(name = newnames[t.name],
                                       legendgroup = newnames[t.name],
                                       hovertemplate='%{y:.1f} C',
                                       line=dict(width=2.5)
                                       )
                  )
```

Tabella 3.2: Andamento temperatura e temperatura apparente in una finestra temporale - **Spark**

3.2.2 Risultati

Nella Figura successiva, viene mostrato il risultato ottenuto dall'esecuzione della seconda Analytic. Essa mostra l'andamento della temperatura nel tempo, in questo caso la linea temporale specificata è nell'intervallo 26-31 Maggio. I picchi di maggior temperatura si hanno nelle ore pomeridiane, nello specifico nell'intervallo orario 12:00 e 15:00.

Quando si parla di temperatura, si fa riferimento ad una quantità di calore presente in un determinato ambiente, dunque è una misura pressoché oggettiva, a differenza della temperatura apparente, intesa come misura soggettiva.

Si può notare inoltre come la temperatura apparente è quasi sempre differente dalla temperatura, in dimostrazione del fatto che quella apparente dipende da differenti fattori, come l'umidità, velocità del vento, esposizione del vento, vestiario.

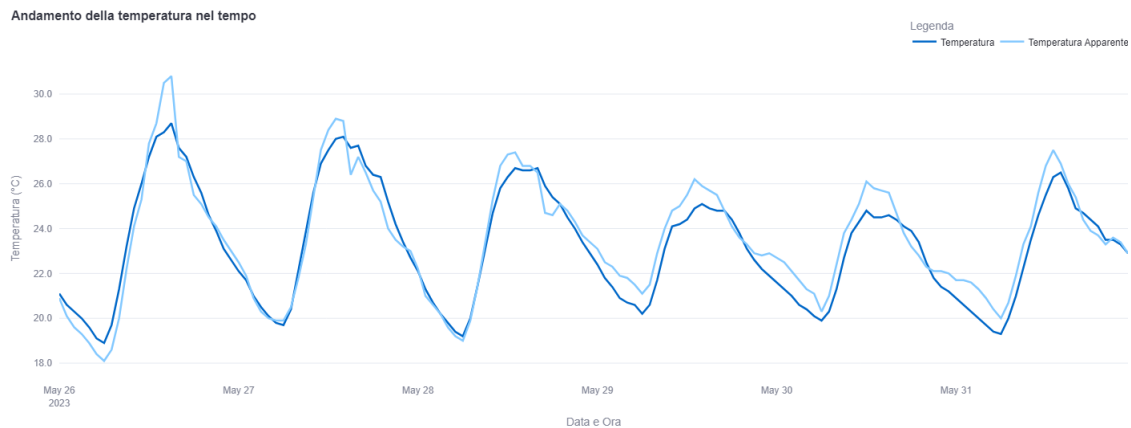


Figura 3.2: Andamento temperatura e temperatura apparente in una finestra temporale

3.3 Andamento umidità in una finestra temporale

In accordo con la precedente Analytic, la terza ha come obiettivo quello di mostrare l'andamento dell'umidità in una finestra temporale. Questa analisi permette di monitorare le variazioni dell'umidità nell'arco del tempo selezionato, fornendo informazioni utili per comprendere le condizioni di umidità relative all'ambiente in esame.

3.3.1 Implementazione

Di seguito l'implementazione in **PySpark**:

<p>Spark</p> <pre> # Esegue la query per ottenere l'andamento dell'umidità nel tempo result_df = info_meteo.select('hourly.time', 'hourly.relativehumidity_2m') # Converte il risultato in un DataFrame pandas result_pd = result_df.toPandas() # Visualizza il line chart utilizzando Plotly Express fig = px.line(result_pd, x=result_pd['time'][0], y=result_pd['relativehumidity_2m'][0], title='Andamento dell'umidità nel tempo') # Aggiunge l'etichetta alla linea plottata fig.update_traces(name = 'Umidita', hovertemplate='Data: %{x}
 Umidita: %{y} %', line=dict(width=2.5)) </pre>
--

Tabella 3.3: Andamento umidità in una finestra temporale - **Spark**

3.3.2 Risultati

Il risultato dell'esecuzione della terza Analytic viene mostrato in Figura successiva. Si possono notare picchi, in termini percentuali, di umidità superiori al 50% nelle ore notturne e nelle prime ore del mattino, nell'intervallo che intercorre tra le 21:00 e le 08:00 del mattino.



Figura 3.3: Andamento umidità in una finestra temporale

3.4 Andamento della velocità del vento in una finestra temporale

La quarta Analytic ha come obiettivo quello di mostrare l'andamento della velocità del vento in una finestra temporale. Questo tipo di analisi consente di osservare le variazioni della velocità del vento nel periodo preso in considerazione, fornendo informazioni importanti sul comportamento e l'intensità del vento durante quel periodo.

Confrontando l'analisi del vento in combinazione con l'umidità è possibile valutare l'impatto del vento sulla dispersione dell'umidità nell'ambiente e sulla formazione di fenomeni atmosferici come le correnti d'aria e le precipitazioni.

3.4.1 Implementazione

Di seguito l'implementazione in **PySpark**:

```
Spark

# Esegue la query per ottenere l'andamento della velocita del vento nel
  tempo
result_df = info_meteo.select('hourly.time', 'hourly.windspeed_10m')

# Converte il risultato in un DataFrame pandas
result_pd = result_df.toPandas()

# Visualizza il line chart utilizzando Plotly Express
fig = px.line(result_pd, x=result_pd['time'][0],
              y=result_pd['windspeed_10m'][0],
              title='Andamento della velocita del vento nel tempo')

# Aggiunge l'etichetta alla linea plottata
fig.update_traces(
    name = 'Velocita',
    hovertemplate='Data: %{x}<br> Velocita: %{y} km/h',
    line=dict(width=2.5))
```

Tabella 3.4: Andamento della velocità del vento in una finestra temporale - Spark

3.4.2 Risultati

Il risultato dell'esecuzione della quarta Analytic viene mostrato in Figura successiva. Si possono notare come i picchi della velocità del vento son avvenuti tra le ore 14:00 e le 15:00 nei giorni del 28 e del 31 Maggio con velocità di venti tra i 17 e i 18 km/h. A seguire i giorni 29 e 30 Maggio, stessa ora, con velocità dei venti di circa 15 km/h.

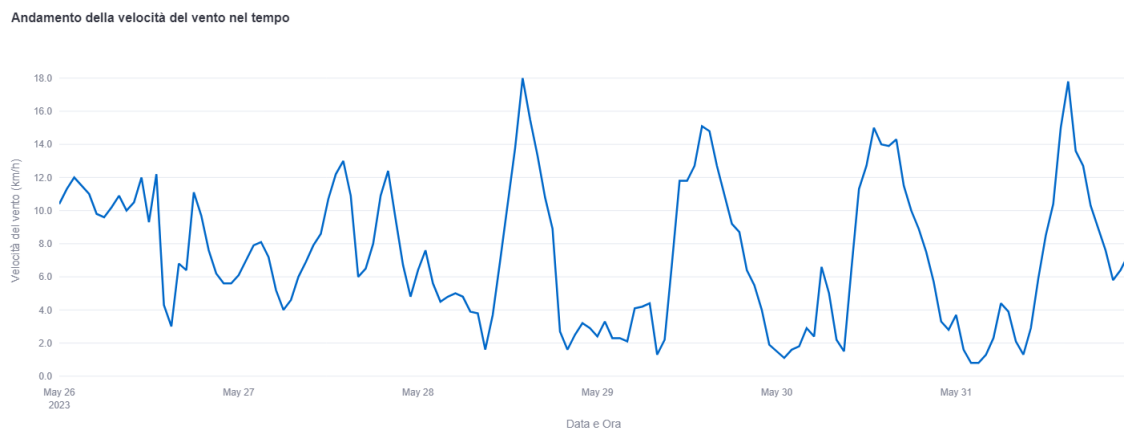


Figura 3.4: Andamento della velocità del vento in una finestra temporale

3.5 Andamento della temperatura media giornaliera

La quinta Analytic ha come obiettivo quello di mostrare l'andamento della temperatura media giornaliera. Quest'analisi fornisce una visione complessiva delle variazioni di temperatura nel corso di una giornata.

3.5.1 Implementazione

Di seguito l'implementazione in **PySpark**:

```
Spark

# Seleziona le colonne "time" e "temperature"
data = info_meteo.select(explode("hourly.time").alias("time"))
temp = info_meteo.select(explode("hourly.temperature_2m").alias("temp"))
# Estrarre la data dalla colonna "time", escludendo l'ora
data_formattata = data.withColumn("date", substring(col("time"), 1, 10))

# Aggiunge un indice ai 2 DataFrame
data_formattata_with_index = data_formattata.withColumn("index",
    monotonically_increasing_id())
temp_with_index = temp.withColumn("index", monotonically_increasing_id())

# Esegue la join basata sull'indice
joined_df = data_formattata_with_index.join(temp_with_index,
    on=["index"], how="inner")
# Seleziona solo le colonne desiderate
result_df = joined_df.select("date", "temp")

# Calcola la temperatura media giornaliera
daily_avg_temp = result_df.groupBy("date") \
    .agg(avg("temp").alias("avg_temp")) \
    .sort(asc("date"))

# Converte il risultato in un DataFrame pandas
result_pd = daily_avg_temp.toPandas()

# Visualizza il line chart utilizzando Plotly Express
fig = px.line(result_pd, x="date", y="avg_temp",
    title="Temperatura media giornaliera")

# Aggiunge l'etichetta alla linea plottata
fig.update_traces(
    name = 'Temperatura media',
    hovertemplate='Data: %{x}<br> Temperatura media: %{y} C',
    line=dict(width=2.5))
```

Tabella 3.5: Andamento della temperatura media giornaliera - Spark

3.5.2 Risultati

Il risultato dell'esecuzione della quinta Analytic viene mostrato in Figura successiva. La temperatura media maggiore nel seguente plot, la si ha avuta per il giorno 27 maggio alle ore 00:00, con un picco di 24.1 °C. Si può notare inoltre come, tra il 27 ed il 30 Maggio, si hanno valori decrescenti di temperatura media.

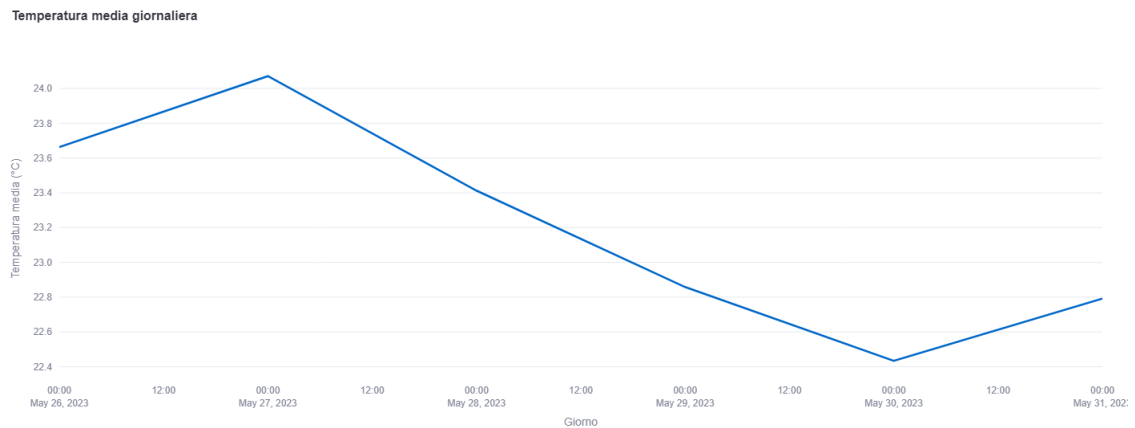


Figura 3.5: Andamento della temperatura media giornaliera

3.6 Andamento probabilità di pioggia e precipitazione effettiva

La sesta Analytic ha come obiettivo quello di valutare l'andamento della probabilità di pioggia e della precipitazione effettiva e come tali andamenti forniscono informazioni utili sulla precisione e l'affidabilità del modello utilizzato per stimare la probabilità di pioggia.

Per lo svolgimento di tale Analytic, si è deciso di analizzare sia l'andamento delle precipitazioni nei giorni precedenti che l'andamento, in termini probabilistici, delle precipitazioni nei giorni futuri.

3.6.1 Implementazione

Di seguito le implementazioni, sia dell'andamento delle precipitazioni nei giorni precedenti che l'andamento probabilistico delle precipitazioni nei giorni successivi in **PySpark**:



```
# Seleziona le colonne "time" e "precipitation"
data = info_meteo.select(explode("hourly.time").alias("time"))
prec = info_meteo.select(explode("hourly.precipitation")
    .alias("precipitation"))

# Aggiunge un indice ai 2 DataFrame
data_with_index = data.withColumn("index", monotonically_increasing_id())
prec_with_index = prec.withColumn("index", monotonically_increasing_id())

# Esegue la join basata sull'indice
joined_df = data_with_index.join(prec_with_index, on=["index"],
    how="inner")

# Seleziona solo le colonne desiderate
result_df = joined_df.select("time", "precipitation")

# Esegue la query per ottenere l'andamento delle precipitazioni nei
    giorni precedenti
filtered_df = result_df.select('time', 'precipitation') \
    .filter(col('time') <= current_time)

# Converti il risultato in un DataFrame pandas
result_pd = filtered_df.toPandas()

# Visualizza il bar chart utilizzando Plotly Express
fig = px.bar(result_pd, x="time", y="precipitation",
    title='Andamento delle precipitazioni nei giorni precedenti')

# Aggiunge l'etichetta alla linea plottata
fig.update_traces(
    name = 'Precipitazioni',
    hovertemplate='Data: %{x}<br> Precipitazioni: %{y} mm')
```

Tabella 3.6: Andamento delle precipitazioni nei giorni precedenti - Spark



```
# Seleziona le colonne "time" e "precipitation_probability"
data = info_meteo.select(explode("hourly.time").alias("time"))
prec_prob = info_meteo.select(explode("hourly.precipitation_probability")
                              .alias("precipitation_probability"))

# Aggiunge un indice ai 2 DataFrame
data_with_index = data.withColumn("index", monotonically_increasing_id())
prec_prob_with_index = prec_prob.withColumn("index",
                                              monotonically_increasing_id())

# Esegue la join basata sull'indice
joined_df = data_with_index.join(prec_prob_with_index, on=["index"],
                                 how="inner")

# Seleziona solo le colonne desiderate
result_df = joined_df.select("time", "precipitation_probability")

# Esegue la query per ottenere l'andamento della probabilita di
# precipitazioni nei prossimi giorni
filtered_df = result_df.select('time', 'precipitation_probability') \
    .filter(col('time') > current_time)

# Converti il risultato in un DataFrame pandas
result_pd = filtered_df.toPandas()

# Visualizza il bar chart utilizzando Plotly Express
fig = px.bar(result_pd, x="time", y="precipitation_probability",
              title='Andamento della probabilita di precipitazioni nel
                    tempo')

# Aggiunge l'etichetta alla linea plottata
fig.update_traces(
    name = 'Probabilita di Precipitazioni',
    hovertemplate='Data: %{x}<br> Probabilita di Precipitazioni: %{y} %')
```

Tabella 3.7: Andamento della probabilità di precipitazioni nei prossimi giorni - Spark

3.6.2 Risultati

Dalle Figure sottostanti ottenute dall'esecuzione della query relativa all'andamento delle precipitazioni, si nota come nei giorni precedenti, nello specifico i giorni 26 Maggio (ore 17:00 e 20:00) e il 27 Maggio (ore 17:00), sono avvenute delle precipitazioni di 0.1 mm, che indicano una quantità di pioggia caduta (misurata in millimetri) molto piccola, e può essere considerata come una pioggia di intensità molto bassa, leggera.

Mentre, per quanto riguarda l'andamento probabilistico delle precipitazioni nei giorni successivi al 29 Maggio (data in cui son state prelevate le informazioni riguardanti le condizioni climatiche), si prevedono piogge su Napoli, con una probabilità superiore alla metà. Vedendo nel dettaglio, questa probabilità aumenta man mano, fino ad arrivare al 68% intorno alle ore 17:00.

Andamento delle precipitazioni nei giorni precedenti

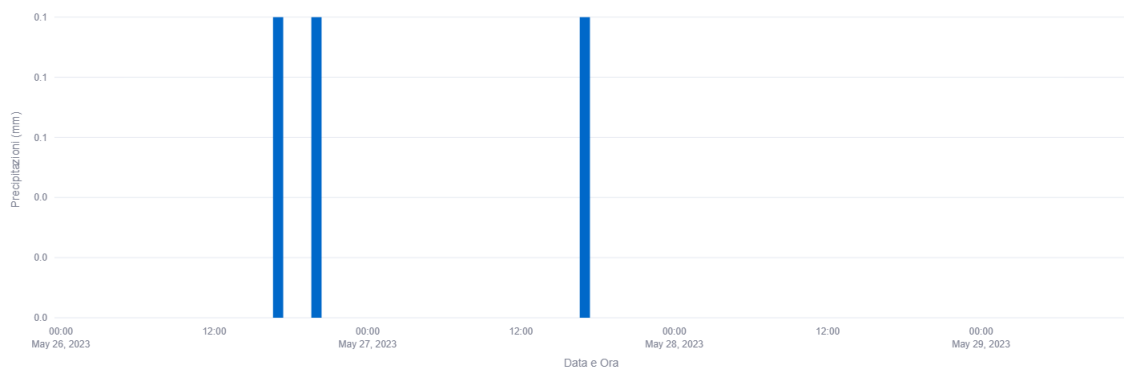


Figura 3.6: Andamento delle precipitazioni nei giorni precedenti

Andamento della probabilità di precipitazioni nel tempo

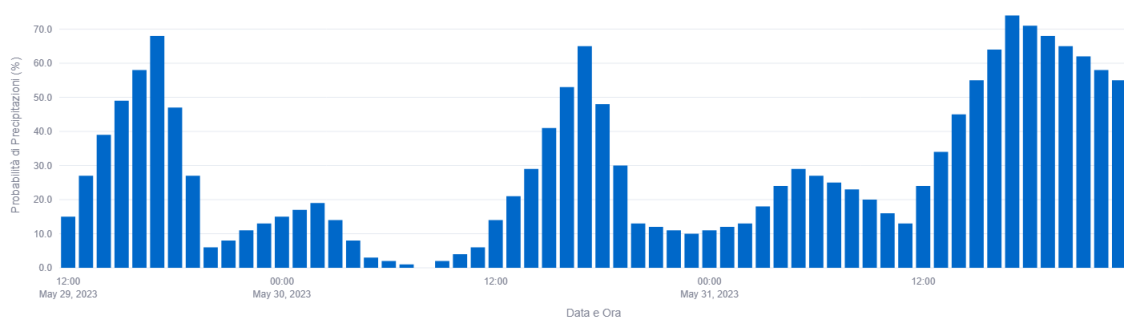


Figura 3.7: Andamento della probabilità di precipitazioni nei prossimi giorni

3.7 Andamento della pressione atmosferica in una finestra temporale

La settima Analytic ha come obiettivo quello di effettuare un'analisi dell'andamento della pressione atmosferica che può contribuire a una migliore comprensione del clima e delle condizioni meteorologiche locali, consentendo quindi di identificare e monitorare potenziali cambiamenti o fenomeni atmosferici significativi.

3.7.1 Implementazione

Di seguito l'implementazione in **PySpark**:

```
Spark

result_df = info_meteo.select('hourly.time', 'hourly.pressure_msl')

# Converti il risultato in un DataFrame pandas
result_pd = result_df.toPandas()

# Visualizza il line chart utilizzando Plotly Express
fig = px.line(result_pd, x=result_pd['time'][0],
              y=result_pd['pressure_msl'][0],
              title='Andamento della pressione nel tempo')

# Aggiunge l'etichetta alla linea plottata
fig.update_traces(
    name = 'Pressione',
    hovertemplate='Data: %{x}<br> Pressione: %{y} hPa',
    line=dict(width=2.5))
```

Tabella 3.8: Andamento pressione in una finestra temporale - **Spark**

3.7.2 Risultati

In Figura sottostante viene mostrato l'andamento della pressione nella finestra temporale specifica. In particolare, la misurazione della pressione atmosferica aiuta a monitorare e prevedere i cambiamenti meteorologici. Nelle previsioni del tempo, la pressione atmosferica viene espressa in hPa (ettaPascal) per indicare le variazioni della pressione atmosferica e la formazione di sistemi meteorologici come cicloni e anticicloni.

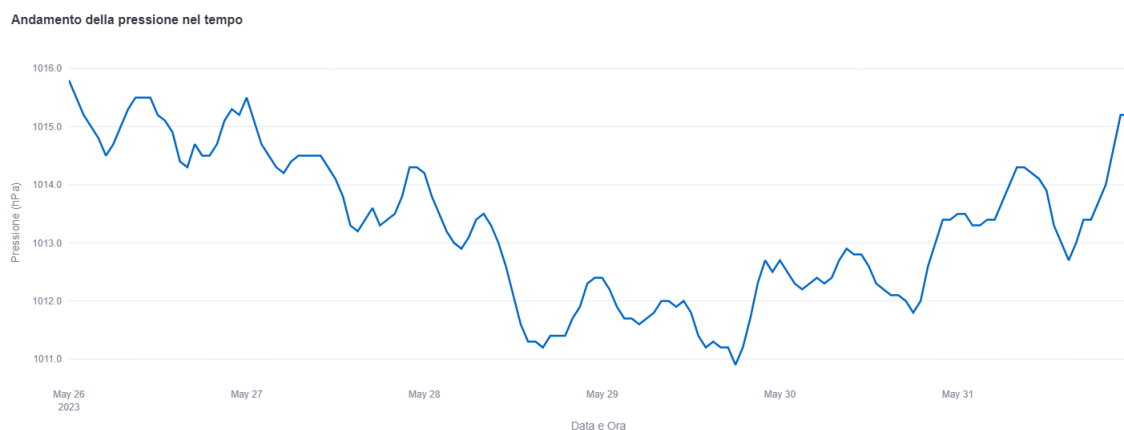


Figura 3.8: Andamento pressione in una finestra temporale

Supponendo di considerare il valore di pressione, del giorno 26 Maggio, di circa 1015.8 hPa, esso può essere considerato come una pressione atmosferica normale. Da notare però che la valutazione della pressione dipende anche dalle condizioni climatiche e zone in cui viene misurata, perché può dipendere anche dall'altitudine (diminuisce con l'aumento di essa), temperatura (se alta, riduce la pressione), umidità (se alta, fa aumentare la pressione) e venti (se forti e se soffiano in direzione ascendente, la pressione si riduce).

3.8 Esecuzione definitiva: Kafka

Si lascia il lettore ad eseguire il seguente codice Python per Kafka:

Kafka

<https://github.com/giuseppericchio/BigData/tree/main/HW3/Kafka>

È possibile inoltre visualizzare il codice della dashboard realizzata su **Streamlit** al seguente link:

<https://github.com/giuseppericchio/BigData/blob/main/HW3/Kafka/dashboard.py>

Contenuti

4.1 Conclusioni	27
---------------------------	----

4.1 Conclusioni

In conclusione, il workflow descritto permette di acquisire dati dall'API sorgente, inviarli a Kafka per il salvataggio nel file system distribuito HDFS, e infine visualizzarli su una dashboard interattiva utilizzando PySpark e Streamlit. Questo approccio offre un modo efficiente per acquisire, gestire, analizzare e presentare i dati meteorologici, o qualsiasi altro tipo di dati, in un ambiente distribuito e scalabile.

In particolare, utilizzando le API del meteo, è stato possibile ottenere dati aggiornati e accurati sulle condizioni meteorologiche. Questi dati possono includere, oltre alle informazioni utilizzate nel contesto di questo homework come temperatura, umidità, velocità del vento, precipitazioni, anche altre informazioni, come raggi UV, orari di alba/tramonto, precipitazioni di neve, utili per poter eseguire future ulteriori analisi.

Elenco delle figure

2.1	Architettura complessiva - Kafka, Hadoop, Spark, Streamlit	12
3.1	Sidebar - Visualizzazione del tempo corrente	15
3.2	Andamento temperatura e temperatura apparente in una finestra temporale	17
3.3	Andamento umidità in una finestra temporale	18
3.4	Andamento della velocità del vento in una finestra temporale	19
3.5	Andamento della temperatura media giornaliera	21
3.6	Andamento delle precipitazioni nei giorni precedenti	24
3.7	Andamento della probabilità di precipitazioni nei prossimi giorni	24
3.8	Andamento pressione in una finestra temporale	25

Elenco delle tabelle

1.1	Data Source: Open-Meteo GFS	1
1.2	Composizione dell'API URL	2
1.3	Parametri scelti	2
2.1	Implementazione Producer - Spark	9
2.2	Implementazione Consumer - Spark	11
3.1	Visualizzazione del tempo corrente - Spark	14
3.2	Andamento temperatura e temperatura apparente in una finestra temporale - Spark	16
3.3	Andamento umidità in una finestra temporale - Spark	17
3.4	Andamento della velocità del vento in una finestra temporale - Spark	19
3.5	Andamento della temperatura media giornaliera - Spark	20
3.6	Andamento delle precipitazioni nei giorni precedenti - Spark	22
3.7	Andamento della probabilità di precipitazioni nei prossimi giorni - Spark	23
3.8	Andamento pressione in una finestra temporale - Spark	25

Bibliografia

- [1] Autori del seguente homework. *Installing Hadoop, Kafka and Spark for Big Data pipeline on WSL2*. 2023.
URL: <https://medium.com/@giuseppe.riccio/installing-hadoop-kafka-and-spark-for-big-data-pipeline-on-wsl2-42eb16dd27af>.
- [2] Autori del seguente homework. *Slide del corso*. 2023.