Università Politecnica delle Marche
Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

Apache Kafka e Spark Structured Streaming: Predizione di guasti mediante l'analisi di Weibull

Gruppo di lavoro:
Dianel Ago
Matricola S1094844

Xhulio Limani
Matricola S1094086

Docente:
Prof. Domenico Potena

Anno Accademico 2021-2022

Indice

1 Introduzione

1.1 Obiettivo

L'obiettivo del progetto é quello di utilizazre Apache Kafka e Spark structured Sreaming per simulare l'invio e la ricezione di dati in near-real-time sui quali applicare, sempre in near-real-time, l'analisi di Weibull per stimare la probabilita di rottura di alcuni componenti/sensori e predirre la loro durata di vita rimanente.

1.2 Kafka e Spark Structure Streaming

Vediamo cosa sono Kafka e Spark Structured Streaming.

Kafka è un sistema di messaggistica produttore-consumatore distribuito, usato per data ingestion in tempo reale che permette di rendere disponibili le informazioni ai consumatori in modo parallelo, con tolleranza ai guasti. Questo lo rende adatto alla costruzione di code di dati in tempo reale che spostano in modo affidabile i dati tra diversi sistemi.

I dati in Kafka sono organizzati in argomenti che vengono suddivisi in partizioni per il parallelismo. Ogni partizione è una sequenza ordinata e immutabile di record e può essere considerata come un registro. I produttori aggiungono record alla coda di questi registri e i consumatori leggono i registri.

Più consumatori possono iscriversi a un argomento e ricevere i record in arrivo. Un cluster Kafka conserva tutti i record pubblicati, indipendentemente dal fatto che siano stati consumati o meno, per un periodo di conservazione configurabile.

Structured Streaming fornisce un'API unificata per lo streaming e il batch che consente di visualizzare i dati pubblicati su Kafka come DataFrame.

É un motore di elaborazione del flusso basato sul motore SQL di Spark. Quando si usa lo streaming strutturato è possibile scrivere query di streaming nello stesso modo in cui si scrivono le query batch.

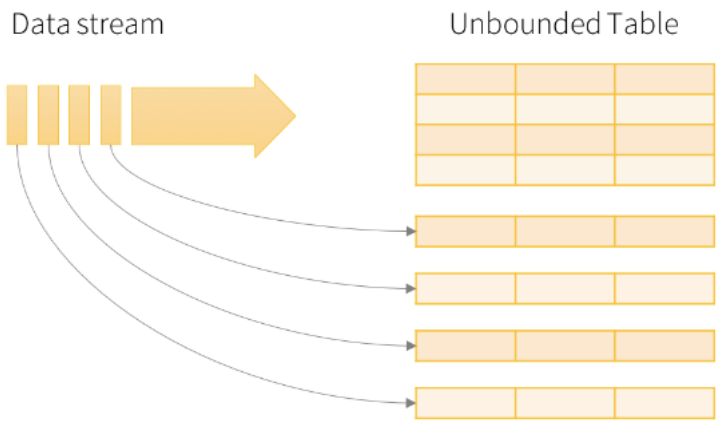


Figura 1: Unbounded Table

Quindi si fa riferimento a una query sull'input che genererà la tabella dei risultati.

A ogni ciclo di trigger, verranno aggiunte nuove righe alla tabella di Input, sulla quale possono essere fatte diverse operazioni di preprocesing dei dati per arrivare a una o piu tabelle derivanti come tabelle di output.

Inoltre si possono ricevere dati "vecchi" e può anche mantenere in memoria uno stato parziale delle aggregazioni per un lungo periodo di tempo (grazie alla funzionalita Window) , in modo che gli eventi che arrivano con ritardo possano essere aggregati in input.

Per non sovraccaricare troppo la memoria con la funzione withwatermark é possibile dare un tempo limite come finestra temporale, dopo il quale i dati in ritardo non vengono piu considerati.

1.3 Dataset usato

Useremo un popolare set di dati disponibile nel repository di dati della NASA, chiamato PHM08.

Si tratta di una raccolta di dati introdotta per la prima volta nel 2008 per una competizione challenge alla prima conferenza su Prognostics and Health Management. È una serie temporale multivariata, che contiene 100 motori turbofan, in cui i dati di ciascun motore hanno misurazioni prese da 21 sensori. Ogni motore inizia a funzionare normalmente e termina con un guasto.

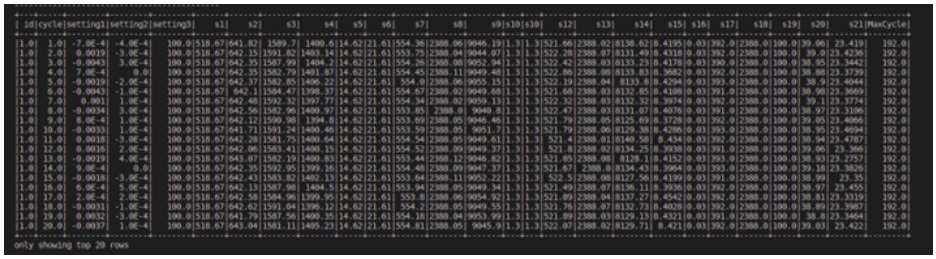


Figura 2: Tabella Dataset

I dati possono essere considerati come provenienti da una flotta di motori dello stesso tipo. Ogni riga del set di dati rappresenta un ciclo operativo di un motore specifico identificato dall' id del motore e dal numero di ciclo. Ci sono più voci per ogni motore per rappresentare i diversi tempi di vita. Altre colonne rappresentano diverse caratteristiche, 3 impostazioni operative e 21 sensori.

Per il valore del ciclo corrispondente all'ultimo valore di ID, il motore viene dichiarato non sano. Per esempio il motore con ID=1 arriva a un massimo di 192 cicli, questo significa che al ciclo numero 192 il motore viene dichiarato rotto.

La stima della vita utile residua (RUL) dei beni critici è una sfida importante nella manutenzione basata sulle condizioni.

Ciò contribuisce a migliorare l'affidabilità, a ridurre i guasti delle macchine e i costi di manutenzione.

Ad esempio, se il primo motore (ID= 1) ha 192 eventi distinti di serie temporali (quindi il ciclo massimo = 192), la colonna dei cicli assumera valori in ordine da 1 a 192, mentre il RUL inizierà con 192 e scenderà fino a 1, pouche il RUL= ciclo_massimo - ciclo_attuale.

Nelle industrie ci sono vari componenti importanti che devono essere monitorati. La RUL è una delle informazioni più importanti per l'utente finale, ma spesso i dati di supporto non sono disponibili.

Utilizzando l'analisi di Weibull, questa lacuna può essere affrontata.

Ai fini del nostro progetto il dataset é stasto splittato in 80:20, ovvero abbiamo utilizzato un 80% per il train ed il rimanente 20% per il test.

1.4 Analisi di Weibull

In molti settori é utile stimare l'affidabilita dei vari componenti nel corso del tempo. Questa ha numerose applicazioni nell' ingegneria, nelle aziende farmaceutiche , nel settore finanziario e anche meteorologico.

L'affidabilità implica il funzionamento corretto di una componente e può essere valutata, o meglio stimata, soltanto attraverso procedure probabilistiche.

I modelli matematici che permettono di svolgere un'analisi affidabilistica sono innumerevoli. Il modello affidabilistico di Weibull é uno di questi.

Nel caso di componenti soggetti a fenomeni di fatica, usura, di corrosione e, in genere, a fenomeni di guasto successivi a fasi di rodaggio e di vita utile (in cui il guasto è random), il tasso di guasto cresce al passare del tempo a causa dell'invecchiamento. Nel modello di Weibull, la probabilità cumulata di guasto, o probabilità che un componente si guasti entro un certo istante t, è definita dall'espressione:

$$F(t) = 1 - e^{-}((t - to)/\eta)^{\beta} \tag{1}$$

e la probabilità che un componente sopravviva fino all'istante t è data da:

$$R(t) = e^{-}((t - to)/\eta)^{\beta} \tag{2}$$

Il parametro η prende il nome di vita caratteristica della distribuzione, o parametro di scala, ad è quel valore per il quale la probabilità cumulata di guasto, o l'inaffidabilità, è del 63%.

Il parametro β è definito parametro di forma. Tale parametro definisce invece la forma della distribuzione. Pertanto se β=1, il tasso di guasto è costante; se β>1 , il tasso di guasto è crescente con il tempo; se β<1 , il tasso di guasto è decrescente con il tempo.

Il parametro t0 è detto parametro di posizione o di locazione ed indica il tempo minimo a cui il componente può subire un guasto:

- t0=0 vuol dire che esso si può rompere da subito;
- t0≠0 significa che il guasto potrebbe accadere solo da un certo istante in poi;

Anche η, β sono parametri maggiori di zero, che definiscono matematicamente la forma della curva di affidabilità.

La probabilità che un componente si guasti ad un dato istante t, quindi la distribuzione guasto, é:

$$f(t) = (\beta/\eta) * (t - to/\eta)^{(\beta-1)} e^{-(t-to/\eta)^{\beta})} \tag{3}$$

Si definisce tasso di guasto $\lambda(t)$ in percentuale:

$$\lambda(t) = \beta/\eta((t - to)/\eta)^{(\beta-1)} \tag{4}$$

Il tempo medio di funzionalità prima dell'occorrenza del guasto (Mean Time To Failure) nella teoria di Weibull è data dall'espressione:

$$MTTF = to + \eta * \Gamma(1 + 1/\beta) \tag{5}$$

dove Γ é la funzione Gamma

Nella nostra analisi per stimare i parametri di Shape,Scala e il paramentro di locazione abbiamo utilizzato la libreria "reliability" per Python3 poiche mette a disposizioni molte funzioni, é di facile utilizzo ed ha una buona precisione. Inoltre ci permette di decidere se fare un Analisi di Weibull a due o tre fattori, con lo stesso set di dati.

2 Architettura e Implementazione

2.1 Architettura

Figura 3: Architettura

La nostra architettura si basa su un semplice Cluster hadoop, composto da un master e due nodi, poiche é risaputo che HDFS permette di immagazzinare un'enorme quantità di dati, in modo da ottimizzare le operazioni di archiviazione e accesso a un ristretto numero di file di grandi dimensioni. Noi abbiamo sfruttato questa proprieta per imagazzinare il dataset di train e l'output necessario per un analisi grafica. Inoltre su ogni macchina del cluster sono stati configurati Spark e Kafka.

Per simulare i dati prodotti dai diversi sensori abbiamo usato Kafka in questo modo: un nodo del cluster non fa altro che leggere ogni riga del dataset di test e la immette del Topic con intervalli di latenza di circa 100 ms.

Dall'altra parte della rete ci sara il master in ascolto sul topic sulla porta dedicata alla comunicazione.

Grazie a Spark structured streaming, il master elaborerá i dati provenienti da Kafka in near-real-time in fase di pre procesing per poi fare l'analisi di Weibull.

Ai fini del funzionamento del progetto, i componenti del cluster devono far parte della stessa rete.

Il progetto e' stato testato sulla rete dell'Univpm.

2.2 Implementazione dell' architettura

Per simulare l'invio dei dati é stato creato uno script python che gira su un nodo del claster il quale attraverso Kafka, crea un topic dove immettere i dati.

Ricordiamo che i dati sono provenienti da un file csv immagazinato su hadoop.

Di seguito è riportato il codice del produttore per lo streaming dei dati :

```
from sqlite3 import Time
import string
from kafka import KafkaProducer
import time
import json
```

```
from json import dumps
from datetime import datetime
import os
import pandas as pd
import csv
import subprocess
from io import StringIO



indice=0
DATA_DIR='/home/diago/Desktop/max_life_test.csv'

train_df= pd.read_csv(DATA_DIR,header=None)
#train_df= pd.read_csv(DATA_DIR,header=None)

train_df.columns=['id','cycle','setting1','setting2','setting3','s1','s2'
,'s3','s4','s5','s6','s7','s8','s9','s10','s11','s12','s13',
's14','s15','s16','s17','s18','s19','s20','s21','MaxCycle']

def get_partizione(key,all,avalaible):    #due partizioni
    if indice%2==0:
       #print("partizione 0")
       return 0
    else:
       #print("partizione 1")
       return 0

def json_serialize(data):
    return json.dumps(data).encode('utf-8')



if __name__== "__main__":

    while 1==1:
        my_producer = KafkaProducer(
        bootstrap_servers = ['xhulio:9092'],
        value_serializer = json_serialize,
        partitioner=get_partizione
        )
        for i in range(0,20630):
        #for line in cat.stdout:
                    riga=train_df.loc[i,:].to_json()
                    utenti_registrati=json.loads(riga)
                    indice=indice+1
                    print(utenti_registrati)
                    my_producer.send("univpma",utenti_registrati)
                    time.sleep(0.1)
```

Ai fini di leggere questi dati, nella macchina master del cluster viene eseguito uno script python dove, avvalendosi dell' interazione tra Kafka e Spark structured streaming, i dati vengono letti per essere filtrati in fase di pre-procesing.

Di seguito è riportato il codice di spark structured streaming che funge da consumatore :

```
##############################
# STREAMING
##############################

bootstrapServers = "localhost:9092"
subscribeType = "subscribe"
topics = "univpma"

lines = spark\
        .readStream\
        .format("kafka")\
        .option("kafka.bootstrap.servers", bootstrapServers)\
        .option("header",True)\
        .option(subscribeType, topics)\
        .option("startingOffsets", "earliest")\
        .load()

linesbis=lines\
        .selectExpr("CAST(value AS STRING) as json")\
        .withColumn("id", jsonparse(col("json"), lit("id")))\
        .
        .
        .
        .select("id","cycle","setting1","setting2","setting3","s1","s2",
        "s3","s4","s5","s6","s7","s8","s9","s10","s10","s12",
        "s13","s14","s15","s16","s17","s18","s19","s20","s21","MaxCycle")
```

Per poter leggere correttamente i dati é necessario specificare il nome del topic e la porta dove kafka li pubblica. Porta e nome del topic devono entrambi coincidere con quelli inseriti per il produttore.

I dati non arrivano cosi come li inviamo, per questo é necessario fare un "cast" per aggiustare e settare la struttura dei dati.

Facendo una query con pysparksql, grazie all funzione ".writestream()" possiamo vedere che il risulato dello streaming prende questa forma:

Figura 4: Tabella Input

Se entriamo nel contesto del progetto, possiamo dire che a questo punto i dati inviati dai sensori arrivano al nodo master, il quale fará delle future eleborazioni, in modo chiaro ben strutturato e senza perdite.

2.3 Fase di training

In questa fase del progetto il dataset preso dalla Nasa é stato da noi diviso in due e in questa sezione ci occuperemeno del file che abbiamo usato per il training, ovvero quello che usa 80% dei dati di partenza e che abbiamo memorizzato all'interno del cluster Hadoop in modo che tutti i nodi possano disporre del contenuto.

Abbiamo usato pyspark sql per elaborare le informazioni estratte da Hadoop e ricavarci un primo modello per la nostra analisi di Weibull.

Prima di tutto viene avviata una sessione per creare un entry point in modo da poter lavorare con Spark utilizzando le API per dataset e dataframe.

Una sessione spark può essere creata utilizzando il metodo getOrCreate() come mostrato nel codice:

```
##############################
# TRAINING
##############################
DIR="/Hadoop_File/max_life_train1.csv"

spark = SparkSession\
        .builder\
        .appName("SparkStructuredWeibull")\
        .master("local[*]")\
        .config("spark.jars.packages",
        "org.apache.spark:spark-sql-kafka-0-10_2.12:3.0.0")\
        .config("spark.jars.packages",
        "graphframes:graphframes:0.8.2-spark3.0-s_2.12")\
        .config("spark.jars.repositories", "https://repos.spark-packages.org")\
        .config("spark.sql.streaming.statefulOperator.checkCorrectness.enabled",
        "false")\
        .getOrCreate()
colonne=['id','cycle'.............','MaxCycle']

schema=StructType()\
  .add("id",IntegerType(),True)\
  .
  .
  .
  .add("s21",FloatType(),True)\
  .add("MaxCycle",IntegerType(),True)
df = spark.read.format("csv").option("header",False).schema(schema).load(DIR)
```

Abbiamo deciso di fare un analisi di Weibull a 3 fattori poiché da uno studio del dataset a nostra disposizione abbiamo visto che i motori non si rompono in quasliasi momento, ovvero tutti i motori presenti nel dataset si rompono dopo il ciclo numero 100.

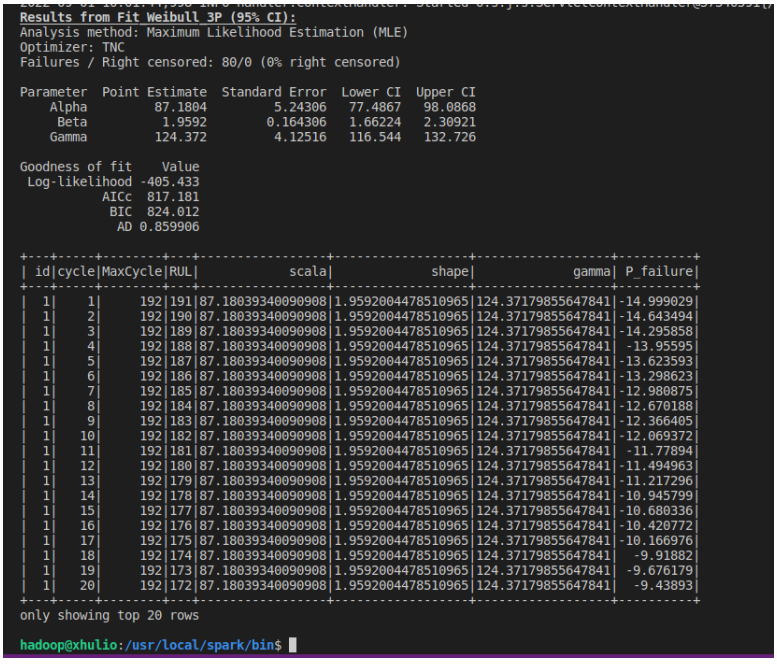
Inoltre, testando i vari parametri e visualizzando i vari grafici abbiamo notato che la Weibull a 3 parametri ci restituiva un modello migliore, dove i punti del dataset seguivano il modello risultante.

Questo ci permette di poter fissare il parametro di locazione gamma.

Quanto detto si puo failmente riscontrare nel grafico in figura [6] della PDF di weibull riportato sotto, con il quale vogliamo evidenziare la differenza della distribuzione di Weibull con due o tre parametri.

```
df_filtered=df.select('id','cycle','MaxCycle')
df_filtered=df_filtered.withColumn("RUL",col('MaxCycle')-col('cycle'))
df_failure=df_filtered.where(col('RUL')==0).select('id','cycle','RUL')
failure_dataset=df_failure.select('cycle')
list_failure=failure_dataset.toPandas().values.reshape(-1)
wb=Fit_Weibull_3P(failures=list_failure,show_probability_plot=False,
print_results=False)
df_final=df_filtered.withColumn("scala",lit(wb.alpha))
df_final=df_final.withColumn("shape",lit(wb.beta))
df_final=df_final.withColumn("gamma",lit(wb.gamma))
df_final=df_final.withColumn("P_failure",p_failure3p(col('cycle'),
col('scala'),col('shape'),col('gamma')))
```

Dalla tabella di output in figura [5] possiamo vedere come la probabilita di fallimento per i primi gamma (parametro di locazione) cicli di ogni motore non viene considerata (ovvero minore di zero) dalla distribuzione di weibull a 3 fattori.



```
Results from Fit_Weibull_3P (95% CI):
Analysis method: Maximum Likelihood Estimation (MLE)
Optimizer: TNC
Failures / Right censored: 80/0 (0% right censored)

Parameter  Point Estimate  Standard Error  Lower CI  Upper CI
    Alpha         87.1804         5.24306   77.4867   98.0868
     Beta          1.9592        0.164306   1.66224   2.30921
    Gamma         124.372         4.12516   116.544   132.726

Goodness of fit    Value
 Log-likelihood -405.433
           AICc  817.181
            BIC  824.012
             AD 0.859906

+---+-----+--------+---+-----------------+-----------------+-----------------+----------+
| id|cycle|MaxCycle|RUL|            scala|            shape|            gamma| P_failure|
+---+-----+--------+---+-----------------+-----------------+-----------------+----------+
|  1|    1|     192|191|87.18039340090908|1.9592004478510965|124.37179855647841|-14.999029|
|  1|    2|     192|190|87.18039340090908|1.9592004478510965|124.37179855647841|-14.643494|
|  1|    3|     192|189|87.18039340090908|1.9592004478510965|124.37179855647841|-14.295858|
|  1|    4|     192|188|87.18039340090908|1.9592004478510965|124.37179855647841| -13.95595|
|  1|    5|     192|187|87.18039340090908|1.9592004478510965|124.37179855647841|-13.623593|
|  1|    6|     192|186|87.18039340090908|1.9592004478510965|124.37179855647841|-13.298623|
|  1|    7|     192|185|87.18039340090908|1.9592004478510965|124.37179855647841|-12.980875|
|  1|    8|     192|184|87.18039340090908|1.9592004478510965|124.37179855647841|-12.670188|
|  1|    9|     192|183|87.18039340090908|1.9592004478510965|124.37179855647841|-12.366405|
|  1|   10|     192|182|87.18039340090908|1.9592004478510965|124.37179855647841|-12.069372|
|  1|   11|     192|181|87.18039340090908|1.9592004478510965|124.37179855647841| -11.77894|
|  1|   12|     192|180|87.18039340090908|1.9592004478510965|124.37179855647841|-11.494963|
|  1|   13|     192|179|87.18039340090908|1.9592004478510965|124.37179855647841|-11.217296|
|  1|   14|     192|178|87.18039340090908|1.9592004478510965|124.37179855647841|-10.945799|
|  1|   15|     192|177|87.18039340090908|1.9592004478510965|124.37179855647841|-10.680336|
|  1|   16|     192|176|87.18039340090908|1.9592004478510965|124.37179855647841|-10.420772|
|  1|   17|     192|175|87.18039340090908|1.9592004478510965|124.37179855647841|-10.166976|
|  1|   18|     192|174|87.18039340090908|1.9592004478510965|124.37179855647841|  -9.91882|
|  1|   19|     192|173|87.18039340090908|1.9592004478510965|124.37179855647841| -9.676179|
|  1|   20|     192|172|87.18039340090908|1.9592004478510965|124.37179855647841|  -9.43893|
+---+-----+--------+---+-----------------+-----------------+-----------------+----------+
only showing top 20 rows

hadoop@xhulio:/usr/local/spark/bin$
```

Figura 5: Tabella finale di training

Dalla figura finale [7] inoltre possiamo vedere il grafico della CDF, e vediamo che i dati relativi alla rottura dei motori seguono il modello di Weibull.

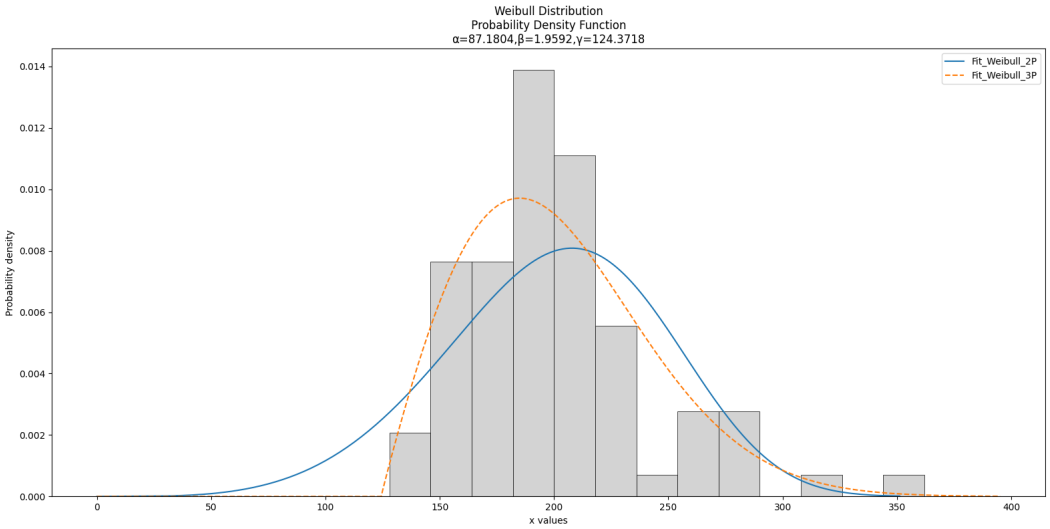


Figura 6: PDF Weibull con due e tre fattori

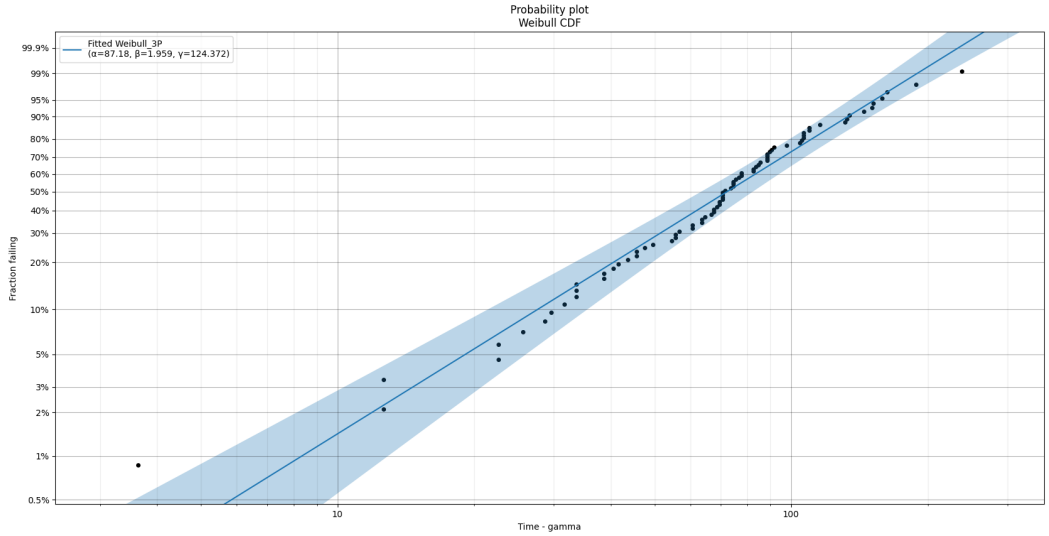


Figura 7: CDF Weibull tre fattori

2.4 Pre procesing dei dati

I dati in input provenienti da Kafka vengono elaborati in modo da poter successivamente lavorare solo con i dati necessari per la nostra analisi.

Ai fini della nostra analisi ci interessa sapere : id motore, il ciclo in corso, il ciclo massimo di vita e quando il RUL ha un valore pari a zero.

```
new_df=linesbis.select('id','cycle','MaxCycle')
new_df=new_df.withColumn("RUL",col('MaxCycle')-col('cycle'))
df_failure=new_df.where(col('RUL')==0).select('id','cycle','RUL')
stream_failure_dataset=df_failure.select('cycle')
df_data_coming=linesbis.select('id','cycle','MaxCycle')
```

il ciclo corrispondente al valore di RUL pari a zero verrá inserito in un altro dataset chiamato "stream failure dataset", il quale useremo poi per fittare i parametri di Weibull.

Spark structured streaming permette di immagazzinare questi dati nella memoria dei driver in modo da poterli successivamente usare in fase di debug.

Quindi abbiamo creato due tabelle: una contenente i valori dell'ultimo ciclo ovvero del numero di ciclo di quando il motore si rompe effettivamente ("Tab coefficenti"), e un altra tabella che contiene id motore, ciclo attuale, e ciclo di vita massimo dei dati in arrivo("Tab dati input").

Queste due tabelle vengono create per semplicita di implementazione del codice Python.

```
query1= stream_failure_dataset\
         .writeStream\
         .outputMode("append")\
         .queryName("Tab_coefficenti")\
         .format("memory")\
         .trigger(processingTime='1 seconds')\
         .start()

query2= df_data_coming\
         .writeStream\
         .outputMode("append")\
         .queryName("Tab_dati_input")\
         .format("memory")\
         .trigger(processingTime='1 seconds')\
         .start()
```

2.5 Fase di test

In questa sezione andremo a vedere come funziona la nostra analisi in near-real-time.

Dopo aver pre processato i dati in arrivo da Kafka, andremo a fare l'analisi di Weibull.

Per ogni ciclo del motore corrente che arriva calcoleremo la sua proabilita di rottura e stimeremo il tempo di vita restante.

La predizione del tempo di vita restante é fatta mediante il calcolo del mmf, ovvere la media di vita dei motori precedentemente analizzati.

Inoltre, man mano che i dati arrivano da Kafka i coefficenti di Weibull si aggiusteranno al meglio in near-real-time per avere una predizione migliore e un adattamento del modello, quindi dei paramentri della Weibull.

In pratica ogni volta che il RUL ha un valore pari a zero, il valore del ciclo corrispondente viene aggiunto su un array, il quale successivamente viene dato come parametro alla funzione di Weibull in modo da verificare se é necessario cambiare i parametri usati fino a quel momento.

Se non é necessario l' analisi continua con i paramentri precedenti altrimenti questi vengono modificati e applicati ai dati in ingresso.

Dall'immagine sottostante [8] possiamo vedere come il modello, anche se lievemente, cambia man mano che arrivano i dati poiché cerca di prendere i paramentri di weibull migliori.

```
whole_failure=np.append(list_failure,list_indici_failure_memoria)
.
.
stream_wb=Fit_Weibull_3P(failures=whole_failure,show_probability_plot
=False,print_results=True)
stream_wb2p=Fit_Weibull_2P(failures=whole_failure,show_probability_plot
=False,print_results=False)
my_df=pd.DataFrame(whole_failure)ci
sparkDF=spark.createDataFrame(my_df)
sparkDF.write.mode("overwrite").csv("hdfs://xhulio:9000/Hadoop_File/array_failure")
.
.
dati_input=dati_input.withColumn("3P_failure",p_failure3p(col('cycle'),
col('3p_scala'),col('3p_shape'),col('3p_gamma')))
.
.
dati_input=dati_input.withColumn("3p_RUL_STIMATO",mttf3p(col('3p_scala'),
col('3p_shape'),col('3p_gamma'))-col('cycle')) # con 3 parametri
dati_input=dati_input.withColumn("stato",when(col('3P_failure')>0.7,"Warning")
.otherwise("OK"))
```

Abbiamo anche voluto confrontare la differenza della distribuzione di Weibull con due e tre fattori, e abbiamo visto che il modello di Weibull a tre paramentri non solo risulta piu adatto e preciso (poco piu), ma nel nostro caso é anche piu corretto nell'interpretazione dei dati come si vede dalla figura sottostante [9].

Come vediamo dalla figura, nonostante tra i due metodi ci sia un errore trascurabile nel predirre il tempo di vita rimanente, c'e una forte differenza nel calcolare la probabilita di rottura.

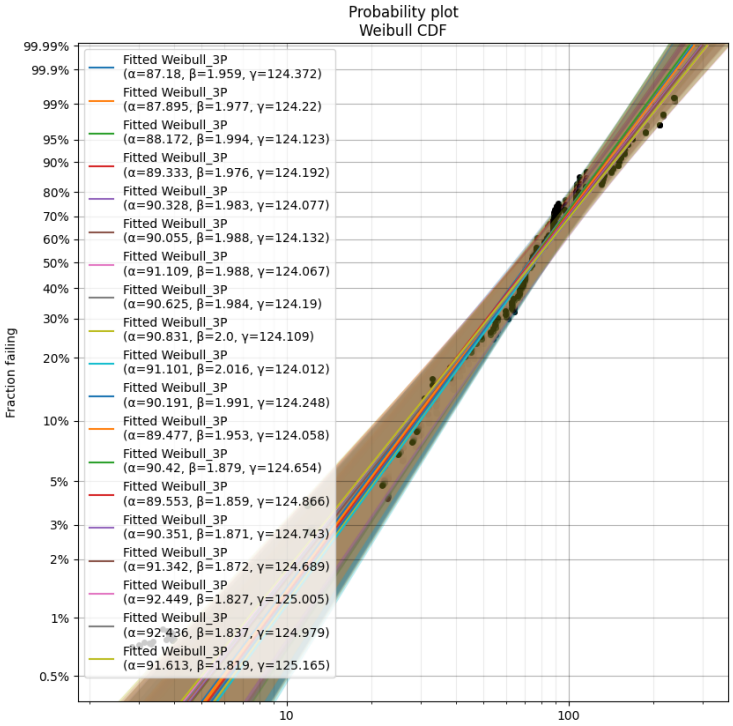


Figura 8: Analisi Weibull in near-realtime

Premesso che il motore numero 88 si rompe al ciclo numero 203, Se facciamo riferimento alla prima riga della tabela la probabilita di rottura calcolata con il metodo a due fattori risulta troppo elevata e se facciamo affidamento a questo paramentro rischiamo di mandare il motore in uno stato di "warning" prima del dovuto. Quando il master rileva che un motore ha una probabilita di rottura superiore del 70% invia un messaggio a uno dei nodi aprendo un topic su Kafka comunicando il numero di id del motore da tenere sotto osservazione [10].

id	cycle	2P_failure	3P_failure	2p_RUL_STIMATO	3p_RUL_STIMATO	stato	2p_errore	3p_errore
88.0	130.0	0.93714607	0.119452916	73.0	74.0	OK	0.046948356807511735	0.04225352112676056
88.0	129.0	0.93579394	0.09995926	74.0	75.0	OK	0.046948356807511735	0.04225352112676056
88.0	128.0	0.9344127	0.080034055	75.0	76.0	OK	0.046948356807511735	0.04225352112676056
88.0	127.0	0.9330018	0.059667744	76.0	77.0	OK	0.046948356807511735	0.04225352112676056
88.0	126.0	0.9315605	0.03885056	77.0	78.0	OK	0.046948356807511735	0.04225352112676056
88.0	125.0	0.9300882	0.017572524	78.0	79.0	OK	0.046948356807511735	0.04225352112676056
88.0	124.0	0.9285843	-0.0041765682	79.0	80.0	OK	0.046948356807511735	0.04225352112676056
88.0	123.0	0.92704797	-0.026407145	80.0	81.0	OK	0.046948356807511735	0.04225352112676056
88.0	122.0	0.9254786	-0.049129862	81.0	82.0	OK	0.046948356807511735	0.04225352112676056
88.0	121.0	0.92387545	-0.07235562	82.0	83.0	OK	0.046948356807511735	0.04225352112676056
88.0	120.0	0.9222379	-0.096095555	83.0	84.0	OK	0.046948356807511735	0.04225352112676056
88.0	119.0	0.920565	-0.120361045	84.0	85.0	OK	0.046948356807511735	0.04225352112676056
88.0	118.0	0.9188562	-0.14516373	85.0	86.0	OK	0.046948356807511735	0.04225352112676056
88.0	117.0	0.9171106	-0.1705155	86.0	87.0	OK	0.046948356807511735	0.04225352112676056
88.0	116.0	0.9153274	-0.1964285	87.0	88.0	OK	0.046948356807511735	0.04225352112676056
88.0	115.0	0.913506	-0.22291517	88.0	89.0	OK	0.046948356807511735	0.04225352112676056
88.0	114.0	0.9116453	-0.24998821	89.0	90.0	OK	0.046948356807511735	0.04225352112676056
88.0	113.0	0.90974456	-0.27766058	90.0	91.0	OK	0.046948356807511735	0.04225352112676056
88.0	112.0	0.90780294	-0.30594558	91.0	92.0	OK	0.046948356807511735	0.04225352112676056
88.0	111.0	0.9058196	-0.33485675	92.0	93.0	OK	0.046948356807511735	0.04225352112676056

only showing top 20 rows

Figura 9: confronto tra i due metodi

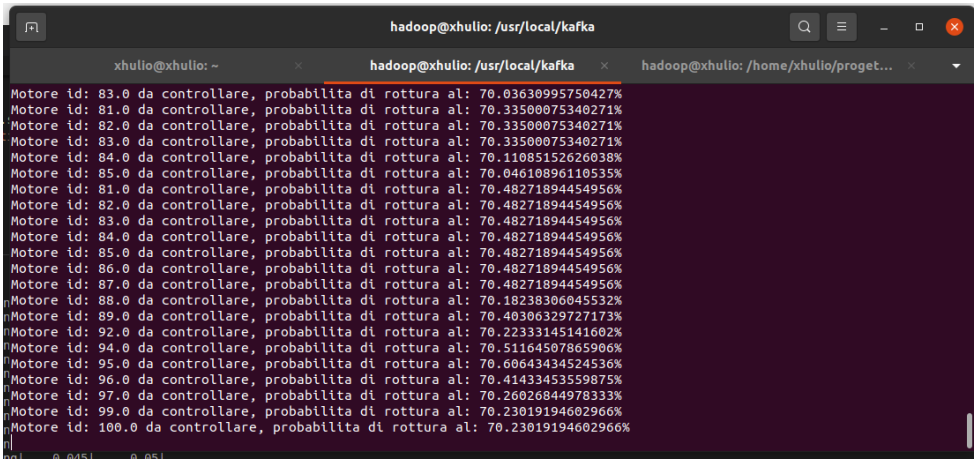


Figura 10: Consumer sulla macchina nodo

3 Conclusioni

Dall' utilizzo di Spark structured streaming e Kafka viene fuori quanto questi due strumenti siano veloci e versatili.

Grazie a Kafka abbiamo potuto mandare informazioni importanti e abbastanza corpose nel nostro caso (un intero dataset) da un dispositivo all'altro con tempi di latenza a nostro dire trascurabili, con una forte affidabilita dei dati poiché tutti i dati inviati sono stati ricevuti, e con una semplicita di programmazione in Python soprendente.

L'unica difficolta riscontrata é stata solo in fase di configurazione, la quale non é stata molto intuitiva data la nostra poca esperienza.

Molto interessante é stato anche l'utilizzo di Spark structured streaming il quale ci ha permesso di lavorare in modo molto scorrevole utilizzando il set di dati a disposizione per le varie operazioni effettuate. Inoltre grazie alle sue molteplcici funzioni ci é stato possibile generare output personalizzabili, ovvero output diversi difronte a esigenze diverse anche se nella nostra applicazione tutto questo non é stato sfruttato al massimo dato che il nostro set di dati non era cosi enorme da farci comprendere tutti i suoi vantaggi.

La programmazione si é basata sulla libreria di Pyspark Sql quindi é stato necessario per noi guardare alcuni concetti basilari su sql.

Tuttavia anche la sua configurazione non é risultata molto intuitiva e veloce, sempre a causa della nostra inesperienza.

Il confronto dei due diversi metodi dell' analisi di Weibull é stato molto interessante.

Per quanto riguarda il metodo con due parametri, quindi ipotizzando che il motore si potesse rompere da un momento all'altro, l'implementazione del modello in fase di training é stato troppo approssimativo, ovvero il risultato non si accostava molto bene alla retta di approssimazione.

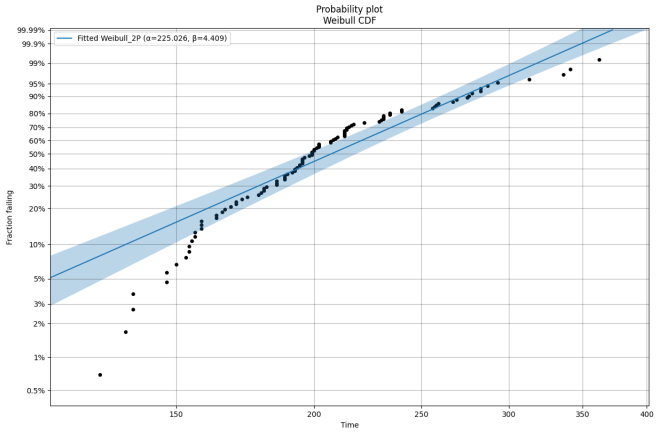


Figura 11: Weibull con 2 parametri

4 Appendice

4.1 Configurazione del sistema

Per la configurazione del sistema per prima cosa siamo partiti dall'installazione di Hadoop. Dopo aver installato tutto si puo accedere ai NameNode di Hadoop tramite l'indirizzo sul browser:

```
http://localhost:9870
```

Una volta finita l'installazione di Hadoop si procede con l'aggiunta dei nodi.

4.1.1 Installazione di Apache Kafka

Per iniziare l'installazione di Apache Kafka prima di tutto bisogna installare Java con i seguenti commandi:

```
sudo apt update
sudo apt install default-jdk
```

Poi andiamo a scaricare ed installare Kafka tramite:

```
wget https://dlcdn.apache.org/kafka/3.2.0/kafka_2.13-3.2.0.tgz
tar xzf kafka_2.13-3.2.0.tgz
sudo mv kafka_2.13-3.2.0 /usr/local/kafka
```

Una volta fatto bisogna creare i SystemD Unit Files per Zookepeer e Kafka:

```
vim /etc/systemd/system/zookeeper.service
```

aggiungendo i codice:

```
[Unit]
Description=Apache Zookeeper server
Documentation=http://zookeeper.apache.org
Requires=network.target remote-fs.target
After=network.target remote-fs.target

[Service]
Type=simple
ExecStart=/usr/local/kafka/bin/zookeeper-server-start.sh
/usr/local/kafka/config/zookeeper.properties
ExecStop=/usr/local/kafka/bin/zookeeper-server-stop.sh
Restart=on-abnormal

[Install]
WantedBy=multi-user.target
```

La stessa cosa anche per Kafka:

```
vim /etc/systemd/system/kafka.service
```

aggiungendo i codice:

```
[Unit]
Description=Apache Kafka Server
Documentation=http://kafka.apache.org/documentation.html
Requires=zookeeper.service

[Service]
Type=simple
Environment="JAVA_HOME=/usr/lib/jvm/java-1.11.0-openjdk-amd64"
ExecStart=/usr/local/kafka/bin/kafka-server-start.sh
/usr/local/kafka/config/server.properties
ExecStop=/usr/local/kafka/bin/kafka-server-stop.sh

[Install]
WantedBy=multi-user.target
```

e alla fine ricaricare il demone systemd per applicare le modifiche

```
systemctl daemon-reload
```

Avviare il servizio Kafka e Zookeeper con i commandi:

```
sudo systemctl start zookeeper
sudo systemctl start kafka
```

Per creare un topic:

```
cd /usr/local/kafka
bin/kafka-topics.sh --create --bootstrap-server localhost:9092
--replication-factor 1 --partitions 1 --topic testTopic

Created topic testTopic.
```

Inviare e ricevere i messagi su Kafka per il produttore:

```
bin/kafka-console-producer.sh --broker-list
localhost:9092 --topic testTopic

>Welcome to kafka
>This is my first topic
>
```

Consumatore:

```
bin/kafka-console-consumer.sh --bootstrap-server
\localhost:9092 --topic testTopic --from-beginning

Welcome to kafka
This is my first topic
```

4.1.2 Installazione di Spark

Installazione dei pacchetti richiesti:

```
sudo apt install curl mlocate git scala -y
```

Scarichiamo l'ultima verisone di spark:

```
curl -O https://archive.apache.org/dist/spark/spark-3.2.0/
spark-3.2.0-bin-hadoop3.2.tgz
```

e lo estrraiamo:

```
sudo tar xvf spark-3.2.0-bin-hadoop3.2.tgz
```

creiamo la directory /opt/spark e muoviamo i file estrrati la cambiando anche i permessi:

```
sudo mkdir /opt/spark
sudo mv spark-3.2.0-bin-hadoop3.2/* /opt/spark
sudo chmod -R 777 /opt/spark
```

e infine modifichiamo il file bashrc:

```
sudo nano ~/.bashrc
```

aggiungendo le due righe seguenti:

```
export SPARK_HOME=/opt/spark
export PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin
```

salviamo i cambiamenti per avere effetto:

```
source ~/.bashrc
```

Per avviare il server master si uttilizza:

```
start-master.sh
```

Per avviare il processo del worker invece:

```
start-slave.sh spark://ServerIPAdress:7077
```