DEPLOY DI UNA PIPELINE DI REAL-TIME DATA INGESTION E ANALYTICS CON AWS IOT CORE, AMAZON KINESIS E AMAZON SAGEMAKER

Amazon Kinesis Data Firehose | Amazon S3 | AWS Glue | AWS IoT Core | ETL | Internet of Things (IoT)

beSharp | 4 Febbraio 2021

Introduzione

Il Machine Learning sta rapidamente entrando a far parte della nostra vita quotidiana. Sempre più software e dispositivi sono oggi in grado di connettersi ad internet e di gestire autonomamente routine e attività di tutti i giorni senza l'intervento umano. Si pensi ad esempio alla domotica, alle luci e ai riscaldamenti *smart* o ai robot che puliscono i pavimenti in autonomia senza difficoltà alcuna anche in ambienti domestici complessi pieni di ostacoli.

In questo contesto, le informazioni che possiamo raccogliere dai dispositivi connessi sono infinite. Il costo contenuto di acquisizione del dato e della potenza di calcolo necessaria ad elaborare grandi quantità di informazioni hanno reso accessibile l'applicazione del Machine Learning ai più diversi casi d'uso. Uno dei più interessanti riguarda sicuramente l'ingestion e l'analisi real-time dei dati provenienti da dispositivi connessi.

In questo articolo, descriveremo una soluzione basata sui servizi gestiti di AWS per l'elaborazione in tempo reale di elevati volumi di dati provenienti da uno o più dispositivi connessi IoT e mostreremo come realizzare una pipeline completa di real-time Data Ingestion e Analytics.

Esploreremo alcuni concetti chiave relativi all'apprendimento automatico, all'ETL, alla pulizia dei dati e alla preparazione del data lake.

Prima di passare alla progettazione del codice e dell'infrastruttura, però, facciamo un breve riepilogo su alcuni concetti chiave relativi a Machine Learning, ETL, pulizia e preparazione dei dati, creazione dei data lake ed IoT. Partiamo!

IoT, Machine Learning e Data Transformation: concetti chiave

L'IoT si è evoluto rapidamente grazie alla diminuzione dei costi dei sensori intelligenti e alla diffusione di metodologie come analisi in tempo reale, apprendimento automatico e sistemi integrati.

Naturalmente, anche i settori tradizionali dei sistemi embedded, delle reti di sensori wireless, dei sistemi di controllo e dell'automazione contribuiscono al mondo dell'IoT.

Machine Learning

Il Machine Learning è nato come un'**evoluzione dell'intelligenza artificiale**. Il Machine Learning tradizionale richiede ai programmatori di scrivere euristiche complesse e difficili da mantenere per eseguire un compito tradizionalmente umano (ad esempio il riconoscimento del testo nelle immagini) utilizzando un computer.

Grazie al ML, è il sistema stesso che impara le relazioni tra i dati.

Per esempio, in un'ipotetica partita di scacchi, basterà fornire un set di dati di caratteristiche riguardanti le partite di scacchi e il modello imparerà a giocare da solo.
Tutto ciò acquista ancora più rilevanza se lo si pensa in un **contesto distribuito** dove la previsione **dovrà scalare**.

Data Transformation

In una pipeline di Machine Learning, i dati devono essere uniformi, ovvero standardizzati. Le differenze nei dati possono derivare dalla loro provenienza da fonti eterogenee, da "database schema" differenti o flussi di importazione dei dati diversi.

La trasformazione dei dati o flusso di ETL (Estrazione, Trasformazione, Caricamento) è quindi un passaggio essenziale in tutte le pipeline di ML. I dati standardizzati non sono solo essenziali nell'addestramento del modello di ML, ma sono anche molto più facili da analizzare e visualizzare nella fase preliminare di **data discovery**.

Per le attività di pulizia e formattazione del dato sono generalmente utilizzate librerie come Scipy Pandas o simili.

– **NumPy**: libreria utilizzata per la gestione di array multidimensionali. Generalmente utilizzata per le fasi di import e lettura di un dataset.

- **Pandas Dataframe**: libreria utilizzata per la gestione di dati in formato tabulare. Colleziona data

La Pipeline

Per la soluzione che andremo a realizzare faremo largo uso dei servizi gestiti messi a disposizione da AWS. Ecco un semplice schema infrastrutturale raffigurante gli attori principali nella nostra Pipeline di ML:

Entriamo nel merito di ciascun servizio.

La pipeline sarà organizzata in 5 fasi principali: **ingestion**, preparazione del **data lake**, **trasformazione**, **training** e **inferenza**.
Per la **fase di ingestion**, i dati saranno raccolti dai dispositivi connessi utilizzando **AWS IoT Core**, un servizio che permette di connettere i dispositivi ad AWS senza dover gestire server o complessità di comunicazione. I dati collezionati saranno poi inviati utilizzando il protocollo MQTT per minimizzare il code da scrivere e la banda richiesta. Con IoT Core è possibile anche gestire l'**autenticazione dei device**.

AWS IoT Core – Per concessione di AWS

Per mandare le informazioni al nostro data lake su Amazon S3, utilizzeremo Amazon Kinesis Data Firehose e la feature che permette la lettura di messaggi IoT Core.

Per trasformare i dati e renderli disponibili per Amazon SageMaker, utilizzeremo invece AWS Glue, il servizio di ETL managed in grado di trovare, preparare e combinare tra di loro i dati, per l'analisi, il machine learning e il deploy dell'applicativo. Mettendo a disposizione tutti questi strumenti, esso permette di analizzare grandi moli di dati in pochi minuti, anziché in mesi.
Infine, vedremo come utilizzare gli algoritmi di Amazon SageMaker, in particolare **DeepAR,** per "istruire" e deployare il modello per l'inferenza.

Ingestion: da IoT Core a Kinesis Firehose

È il momento di connettere i nostri dispositivi di test attraverso le feature di AWS IoT Core.

Connettere un nuovo dispositivo

Seguiamo i passaggi descritti nel wizard per connettere i dispositivi.

Gli obiettivi di questa fase sono:

1. Creare un **AWS IoT Thing**
2. Scaricare il codice richiesto direttamente sul nostro dispositivo per permettere la connessione con AWS.

Stabilire una connessione con AWS è importante anche per permettere a Kinesis Firehose di leggere i messaggi mandati da AWS IoT Core. Ricordiamo che il dispositivo che stiamo connettendo necessiterà di una connessione TCP pubblica sulla porta 8883.

Dal wizard, selezioniamo Linux come sistema operativo e un SDK (nel nostro caso Node.js):

A questo punto, diamo un nome al nostro dispositivo e otteniamo il nostro kit di connessione contenente:

- L'SDK selezionata

Riassunto delle proprietà di un "Thing"

Una volta scaricato il kit, inizializziamo un nuovo progetto Node.js e installiamo **AWS-IoT-device-SDK.** In questo modo, i node module richiesti verranno installati. Dopodiché sarà possibile lanciare lo script **start.sh** incluso, aggiungendo tutti i certificati scaricati nel kit nella stessa directory del progetto.

Abbiamo sviluppato il nostro esempio partendo dal codice di **device-example.js** come semplice base per capire come connettere un dispositivo ad AWS IoT:

```
const deviceModule = require('aws-iot-device-sdk').device;
const cmdLineProcess = require('aws-iot-device-sdk/examples/lib/cmdline');

processPollutionData = (args) => {

   // Device properties which are needed
   const device = deviceModule({
       keyPath: args.privateKey,
       certPath: args.clientCert,
       caPath: args.caCert,
       clientId: args.clientId,
       region: args.region,
       baseReconnectTimeMs: args.baseReconnectTimeMs,
       keepalive: args.keepAlive,
       protocol: args.Protocol,
       port: args.Port,
       host: args.Host,
       debug: args.Debug
   });

   const minimumDelay = 250; // ms
   const interval = Math.max(args.delay, minimumDelay);

   // Send device information
   setInterval(function() {
```

```
        // Prepare Data to be sent by the device
```



```
        };

        device.publish('', JSON.stringify(payload));
    }, interval);

    // Device callbacks, for the purpose of this example we have put
    // some simple console logs
    device.on('connect', () => { console.log('connect'); });
    device.on('close', () => { console.log('close'); });
    device.on('reconnect', () => { console.log('reconnect'); });
    device.on('offline', () => { console.log('offline'); });
    device.on('error', (error) => { console.log('error', error); });
    device.on('message', (topic, payload) => {
console.log('message', topic, payload.toString());
    });
}

// this is a precooked module from aws to launch
// the script with arguments
module.exports = cmdLineProcess;

// Start App
if (require.main === module) {
    cmdLineProcess('connect to the AWS IoT service using MQTT',
        process.argv.slice(2), processPollutionData);
```

Importiamo i moduli di Node.js necessari a connettere i nostri dispositivi ad AWS e di pubblicare su un canale a noi rilevante. è possibile leggere i dati dai sensori dei dispositivi in qualunque modo, ad esempio, nel caso in cui un device possa scrivere le informazioni in una specifica destinazione sul disco, basterà leggere e rendere i dati una stringa utilizzando **device.publish('<YOUR_TOPIC>', JSON.stringify(payload))**.

L'ultima parte di codice chiama semplicemente la funzione principale al fine di mandare le informazioni alla console.

Per eseguire lo script, utilizziamo lo script start.sh incluso nel development kit **assicurandoci di puntare al nostro codice e non al codice di esempio fornito da AWS**

Nota: per la natura esemplificativa dell'articolo, il codice del dispositivo che utilizziamo è semplificato. Consigliamo di non utilizzarlo per un ambiente di produzione.

Per testare il funzionamento di ciò che abbiamo realizzato fin qui, accediamo alla console AWS IoT, entriamo nella sezione **Test** dalla sidebar sulla sinistra e inseriamo il nome del nostro topic.

Clicchiamo su “Subscribe to topic” e, se tutto è correttamente setuppato, dovremmo vedere

è il momento di connettere Kinesis Firehose per cominciare a inviare i dati ad Amazon S3.

Kinesis Firehose

Trasferire i dati raccolti dai dispositivi, arricchire il data lake e migliorare il modello è estremamente importante per evitare il problema chiamato Concept Drift**,** un problema che si verifica al **graduale disallineamento del modello deployato rispetto ai dati reali**. Questo succede in quando i dati storici non sarebbero in grado di rappresentare un problema nel frattempo evoluto.

Per risolvere il problema dobbiamo assicurare un logging efficiente e capire quando intervenire sul modello, ad esempio effettuando nuovamente il training o aggiornando la versione per poi rideployarla. Definiamo quindi una “action” di Kinesis Firehose specifica per registrare automaticamente e trasportare ciascun messaggio MQTT distribuito dal dispositivo, direttamente su Amazon S3, in modo da alimentare il nostro data lake con dati sempre aggiornati.

Creiamo lo stream di Firehose

Per creare lo stream di Firehose, cerchiamo “Kinesis firehose” nella search bar, selezioniamolo e clicchiamo su “Create delivery stream”, come mostrato in figura:

Selezioniamo un nome valido in “Delivery stream name”, “Direct PUT or other sources” nella sezione “Sources” e, nella pagina successiva, lasciamo tutto come da default. Convertiremo i dati in S3 più tardi. Infine, nell’ultima pagina, selezioniamo **S3** come destinazione e aggiungiamo eventualmente un prefisso ai dati inseriti nel bucket. Clicchiamo su “Next” per creare lo stream.

Creiamo la IoT Rule

Per utilizzare lo stream creato, occorre prima connetterlo con AWS IoT tramite una **IoT Rule**. l’IoT Rule autorizzerà Kinesis a ricevere i messaggi e a scriverli nel bucket S3. Per configurare AWS IoT per mandare messaggi a Firehose abbiamo eseguito i seguenti passaggi:

1. Durante la creazione della regola, nella console di AWS IoT, scegliamo “Add action” nella sezione “Set one or more actions”.
2. Scegliamo “Send a message to an Amazon Kinesis Firehose stream”.

5. Come Separator scegliamo un carattere da inserire tra i record, ad esempio una virgola.
6. Per il nome del ruolo IAM, scegliamo “Create a new role”.
7. Selezioniamo “Add action”.

Ecco un esempio di come apparirà la regola che andreamo a creare:

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM ''",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "firehose": {
                    "deliveryStreamName": "",
                    "roleArn": "arn:aws:iam:::role/"
                }
            }
        ]
    }
}
```

Se avremo svolto correttamente tutti i passaggi, cominceremo a veder comparire i dati nel bucket:

Apriamo uno dei file caricati nel bucket e... ecco i file generati dai nostri device!

Datalake: S3

Amazon S3 è il servizio di storage ideale per costruire data lake. Con una possibilità di scalare pressoché illimitata, un data lake basato su Amazon S3 per l’analisi dei big data, presenta diversi benefici.

L’architettura dati centralizzata di S3 semplifica la creazione di un ambiente multi-tenant in cui più utenti possono utilizzare il proprio strumento di analisi di Big Data su un insieme comune di dati.

Inoltre, S3 si integra perfettamente con altri servizi Amazon come Amazon Athena, Amazon

di dati "moccati" da un device di test per eseguire semplici algoritmi di forecasting.

Processo di ETL: AWS Glue

Anche se i dati vengono salvati su Amazon S3 quasi in tempo reale, non sono ancora sufficienti per consentirci di gestire un modello Amazon SageMaker. Come abbiamo spiegato nell'introduzione, infatti, i dati devono essere preparati e quando si tratta di algoritmi **AWS SageMaker predefiniti,** è necessario tenere presente alcune impostazioni di default.

Ad esempio SageMaker non accetta headers e, nel caso in cui volessimo definire un **training supervisionato**, dobbiamo anche mettere la "ground truth" come prima colonna del dataset.

In questo semplice esempio abbiamo utilizzato Glue Studio per trasformare i dati grezzi nel bucket S3 di sorgente in file di parquet strutturati da salvare in un Bucket di output dedicato. Il bucket di output verrà utilizzato da Sagemaker come origine dati.

Attiviamo il Crawler appena creato, cliccando su "Run crawler".

Il prossimo step è configurare un job di Glue Studio, utilizzando il "Catalog" come sorgente

d’ingresso dei dati.

Per farlo, ecco qui gli step che abbiamo seguito:

1. Scegliamo “Create and manage jobs” dalla dashboard di AWS Glue Studio.
2. Nella pagina “Manage Jobs”, scegliamo le opzioni di Origine e Destinazione aggiunte alle proprietà del grafico. Quindi, scegliamo S3 sia come sorgente che come destinazione.

3. Premiamo il pulsante “Create” per avviare il processo di creazione del lavoro.

Ora vedremo un grafico a tre nodi che rappresenta i passaggi coinvolti nel processo ETL. Quando AWS Glue viene istruito a leggere da un’origine dati S3, creerà anche uno schema interno, chiamato **Glue Data Catalog**.

Il grafico ETL

Per configurare il source node, clicchiamo su di esso nel grafico:

Seleziona il database e la tabella del crawler

La stessa cosa può essere fatta per il nodo di trasformazione: cliccando su di esso, è possibile definire quale tipo di trasformazione si vuole applicare ai dati di input. Qui puoi anche verificare che il JSON sia stato importato correttamente:

La mappatura automatica generata da AWS Glue

Infine, possiamo selezionare il nodo di destinazione, specificando di nuovo S3 come destinazione e utilizzando .parquet come formato di output.

Le proprietà del nodo di destinazione

Ora dobbiamo impostare i parametri del lavoro ETL dato il grafico dei nodi appena creato. Andiamo


```
{
    "Version": "2012-10-17",
    "Statement": [
       {
          "Effect": "Allow",
          "Principal": { "Service": "glue.amazonaws.com" },
          "Action": "sts:AssumeRole"
       }
    ]
}
```

Se tutto è definito correttamente, il job partirà, e contestualmente, inizierà anche la conversione dei dati in formato parquet. I file verranno inseriti nella directory di nostra scelta all'interno del bucket.

File convertiti in parquet

Ottimizzazione del dataset: perché parquet rispetto al CSV

Abbiamo scelto di utilizzare .parquet invece di .csv per il dataset di destinazione.
Il parquet è un formato colonnare altamente compresso, che utilizza l'algoritmo di distruzione e assemblaggio dei record, molto superiore al semplice appiattimento di namespace annidati. Esso presenta i seguenti vantaggi:

- Offre efficienza rispetto ai file basati su righe come CSV. Durante l'interrogazione, l'archiviazione a colonne ignora i dati non rilevanti e la stessa può essere eseguita molto rapidamente.
- Le query di aggregazione richiedono meno tempo rispetto ai database row-oriented, riducendo al minimo la latenza per l'accesso ai dati.
- Apache Parquet può supportare strutture dati nidificate avanzate.
- Parquet è progettato per supportare opzioni di compressione flessibili e schemi di codifica efficienti.
- Apache Parquet funziona al meglio con tecnologie interattive e serverless come AWS Athena, Amazon Redshift e AWS Glue.

Inoltre rispetto al file archiviato in formato .csv abbiamo questi vantaggi in termini di risparmio sui

La fase di machine learning: previsione con Amazon SageMaker

Amazon SageMaker offre 17 algoritmi pronti all'uso che coprono una pletora di argomenti relativi ai problemi di Machine Learning. Nel nostro caso, volevamo semplificare lo sviluppo di un modello per fare previsioni sui dati recuperati dal nostro dispositivo, quindi, invece di mostrare il paradigma **bring your own algorithm**, come nel nostro articolo precedente, questa volta ne useremo uno già pronto.

Come spiegato in precedenza, oltre alla pulizia dei dati, il nostro processo ETL è stato eseguito per trasformare i dati in modo che fossero compatibili con gli algoritmi SageMaker già pronti.

SageMaker API e la libreria di Sklearn offrono metodi per recuperare i dati, chiamare il metodo di training, salvare il modello e distribuirlo in produzione per inferenze real-time o batch.

Iniziamo andando alla pagina di SageMaker e creiamo una nuova istanza notebook, per questo articolo scegliamo una **ml.t3.medium**. Aggiungiamo un nome e creiamo un nuovo **ruolo IAM**.

Lasciamo il resto come predefinito e clicchiamo su "Create notebook".

Creiamo una nuova istanza Notebook

L'accesso è possibile da Jupiter o Jupiter Lab, noi scegliamo il secondo.
Siamo riusciti a mettere in piedi un semplice notebook, che illustra tutti i passaggi coinvolti nell'utilizzo di un algoritmo DeepAR preimpostato da AWS.

Nota: il codice è realizzato esclusivamente per questo articolo e non è pensato per un ambiente di produzione in quanto non vi è alcuna indagine preliminare sui dati e nessuna convalida dei risultati. Tuttavia, tutto il codice presentato è testato e utilizzabile per casi d'uso simili a quello presentato.

Iniziamo importando tutte le librerie necessarie:

```
import time
import io
import math
import random
import numpy as np
import pandas as pd
```

```
import JSON
```


Abbiamo anche impostato le basi per i nostri generatori casuali per garantire la riproducibilità. Dopodiché, dobbiamo recuperare i nostri **file parquet** da **S3** e ottenere da loro un Pandas Dataframe.

```
bucket = ""
data = "output"
model = "model"

sagemaker_session = sagemaker.Session()
role = get_execution_role()

s3_data_path = f"{bucket}/{data}"
s3_output_path = f"{bucket}/{model}/"
```

Inizialmente, prepariamo tutti i percorsi di S3 che verranno utilizzati nel Notebook, generiamo una **sessione SageMaker** e un **ruolo IAM** valido con **get_execution_role()**. Come possiamo vedere SageMaker si prende cura di questi aspetti per noi.

```
from sagemaker.amazon.amazon_estimator import get_image_uri
image_uri = get_image_uri(boto3.Session().region_name, "forecasting-deepar")
```

Nel passaggio precedente abbiamo recuperato il nostro **forecasting Estimator, DeepAR**. Un estimator è una classe in SageMaker in grado di generare, apprendere e testare un modello che verrà poi salvato su S3.

Prima di iniziare a leggere i file parquet aggiungiamo anche un paio di costanti per il nostro esperimento:

```
freq = "H"
prediction_length = 24
context_length = 24 # usually prediction and context are set equal or similar
```

Con **freq** (frequenza) diciamo che vogliamo analizzare la TimeSeries con metriche orarie.
La previsione e la durata del contesto sono impostate su 1 giorno e indicano rispettivamente quante ore vogliamo prevedere in futuro e quante ore in passato utilizzeremo per la previsione.
Di solito, questi valori sono definiti in termini di giorni poiché il dataset è molto più grande.

Abbiamo creato due metodi di supporto per leggere dai file parquet:

```
    if not filepath.endswith('/'):
        filepath = filepath + '/'  # Add '/' to the end

    s3_client = boto3.client('s3')
    s3 = boto3.resource('s3')
    s3_keys = [item.key for item in s3.Bucket(bucket).objects.filter(Prefix=filepath)
               if item.key.endswith('.parquet')]
    if not s3_keys:
        print('No parquet found in', bucket, filepath)

    dfs = [pd_read_s3_parquet(key, bucket=bucket, s3_client=s3_client, **args)
           for key in s3_keys]
    return pd.concat(dfs, ignore_index=True)
```

Quindi leggiamo effettivamente i datasets:

```
# get all retrieved parquet in a single dataframe with helpers functions
df = pd_read_s3_multiple_parquets(data, bucket)
df = df.iloc[:, :8] # get only relevant columns
df['hour'] = pd.to_datetime(df['timestamp']).dt.hour #add hour column for the timeser
ies format

# split in test and training
msk = np.random.rand(len(df)) < 0.8 # 80% mask

# Dividing in test and training
training_df = df[msk]
test_df = df[~msk]
```

Qui manipoliamo il dataset per renderlo utilizzabile con DeepAR che ha il suo formato proprietario. Usiamo df.iloc[:, :8] per mantenere solo le colonne originali senza quelle generate da Glue Schema. Generiamo una nuova colonna **hour** per velocizzare le cose, infine, dividiamo il set di dati in proporzioni 80/20 per l'addestramento e il test.

Quindi riscriviamo temporaneamente i dati su S3 come richiesto da DeepAR, creando file JSON con serie al loro interno.

```
# We need to resave our data in JSON because this is how DeepAR works
# Note: we know this is redundant but is for the article to show how many ways
# there are to transform dataset back and forth from when data is acquired

train_key = 'deepar_training.json'
test_key  = 'deepar_test.json'

# Write data in DeepAR format
```

```
def writeDataset(filename, data):
```



```
                file.write('{'+line+'}\n')
```

Generiamo un JSON in un formato simile a questo:

```
{"start":"2021-02-05 13:00:00","target":[69.0, 56.0, 2.0, …]}
```

Dopodiché, possiamo scrivere i nostri file JSON su S3.

```
writeDataset(train_key, training_df)
writeDataset(test_key, test_df)

train_prefix   = 'model/train'
test_prefix    = 'model/test'

train_path = sagemaker_session.upload_data(train_key, bucket=bucket, key_prefix=train
_prefix)
test_path  = sagemaker_session.upload_data(test_key,  bucket=bucket, key_prefix=test_
prefix)
```

Usiamo **sagemaker_session.upload_data ()** per questo, passando il percorso di output.
Successivamente, possiamo definire lo stimatore:

```
estimator = sagemaker.estimator.Estimator(
    sagemaker_session=sagemaker_session,
    image_uri=image_uri,
    role=role,
    instance_count=1,
    instance_type="ml.c4.xlarge",
    base_job_name="pollution-deepar",
    output_path=f"s3://{s3_output_path}",
)
```

Passiamo la sessione SageMaker, l'immagine dell'algoritmo, il tipo di istanza e il percorso di output del modello. Abbiamo anche bisogno di configurare alcuni Iperparametri:

```
hyperparameters = {
    "time_freq": freq,
    "context_length": str(context_length),
    "prediction_length": str(prediction_length),
    "num_cells": "40",
    "num_layers": "3",
    "likelihood": "gaussian",
```

```
    "epochs": "20",
```


Questi valori sono presi direttamente dagli esempi AWS ufficiali su DeepAR. Dobbiamo anche passare i due canali, training e test, allo stimatore per avviare il "processo di adattamento" (**fitting process**).

```
data_channels = {"train": train_path, "test": test_path}
estimator.fit(inputs=data_channels)
```

Dopo il training e il test di un modello, è possibile distribuirlo utilizzando un **Real-time Predictor.**

```
# Deploy for real time prediction
job_name = estimator.latest_training_job.name

endpoint_name = sagemaker_session.endpoint_from_job(
    job_name=job_name,
    initial_instance_count=1,
    instance_type='ml.m4.xlarge',
    role=role
)

predictor = sagemaker.predictor.RealTimePredictor(
    endpoint_name,
    sagemaker_session=sagemaker_session,
    content_type="application/json")
```

Il predictor genera un endpoint visibile anche dalla console AWS.

L'endpoint può essere chiamato da qualsiasi applicazione abilitata REST che passa una richiesta con un formato come quello di seguito:

```
{
  "instances": [
    {
       "start": "2021-02-05 00:00:00",
       "target": [88.3, 85.4, ...]
    }
  ],
  "configuration": {
    "output_types": ["mean", "quantiles", "samples"],
    "quantiles": ["0.1", "0.9"],
    "num_samples": 100
  }
}
```

I “targets” sono dei valori di esempio a partire dal periodo impostato in “start” dal quale si vuole

Inferenza in tempo reale: dall'idea alla produzione

L'inferenza in tempo reale si riferisce alla previsione fornita in tempo reale da alcuni modelli. Questo è il tipico caso d'uso di molti sistemi di raccomandazione o generalmente quando la previsione è ad uso singolo. Viene utilizzata quando:

- Abbiamo a che fare con dati **dinamici**.
- Abbiamo richieste di **bassa latenza**.
- Vogliamo previsioni in **tempo reale**.
- È caratterizzata da **un'unica previsione**.

In genere è un pò più complessa da gestire rispetto a ciò che abbiamo fatto nel Notebook ed è tipicamente definita in una pipeline separata, a causa della sua natura di alta disponibilità e tempi di risposta rapidi.

Quando deployamo utilizzando l'API SageMaker è possibile creare un processo di distribuzione molto simile a come viene rilasciata o aggiornata un'applicazione web, tenendo conto di cose come il reindirizzamento del traffico e le tecniche di distribuzione come Blue/Green o Canary.
Vogliamo condividere con voi una guida riassuntiva per entrambi i metodi da provare da soli!

Come deployare

1. Creiamo un modello utilizzando **CreateModelApi.**
2. Creiamo un endpoint HTTPS utilizzando **CreateEndpointConfigApi** inserendo come proprietà:
   - The model
   - The production variants
   - Instance type
   - Instance count
   - Weight
3. Finalizziamo la creazione dell’endpoint utilizzando **CreateEndpointApi**. Passiamo i dati delle due precedenti configurazioni e qualsiasi **tags** a quest’ultima istruzione.

Nota: attraverso le production variants *possiamo implementare diverse strategie di Deploy come A/B e BLUE/GREEN.*

Deploy Blue / Green

Reindirizziamo il traffico su Green. Se Green è ok, con un altro **UpdateEndpointApi** cancelliamo il vecchio modello.

Deploy A / B

Da utilizzare specificatamente se si vuole misurare le performance tra modelli rispetto ad una metrica di alto livello.

1. Creiamo più modelli utilizzando la stessa configurazione.
2. Aggiorniamo o creiamo una configurazione modificando o creando production variants.
3. Settiamo il balancing weights a 50/50.
4. Verifichiamo functionality e performance.
5. Gradualmente cambiamo la % del traffico.

Alla fine escludiamo 1 o più modelli (in questo caso 50/50 uno dei due).

Nota: la proprietà multi-modello per endpoint consente di gestire più modelli contemporaneamente, la memoria della macchina viene gestita automaticamente in base al traffico. Questo approccio può far risparmiare denaro grazie all'uso ottimizzato delle risorse.

Referenze

- https://docs.aws.amazon.com/iot/latest/developerguide/iot-quick-start.html
- https://docs.aws.amazon.com/iot/latest/developerguide/kinesis-firehose-rule-action.html
- https://docs.aws.amazon.com/glue/latest/ug/tutorial-create-job.html
- https://docs.aws.amazon.com/iot/latest/developerguide/topics.html
- https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/CloudWatch_Synthetics_Canaries.html
- https://mqtt.org/
- https://machinelearningmastery.com/gentle-introduction-concept-drift-machine-learning/
- https://parquet.apache.org/

Facciamo il punto

In questo articolo abbiamo visto come sviluppare una pipeline utilizzando le risorse AWS, per acquisire dati da un dispositivo connesso all'ecosistema AWS tramite le funzionalità IoT Core.

Abbiamo visto come l'utilizzo di un set di dati archiviato in parquet sia migliore di uno in semplice formato CSV. Soprattutto ci siamo focalizzati sulle sue maggiori performance in fase di import/export, per le query Athena e di come sia molto più conveniente, in termini di prezzo di AWS S3, grazie alle dimensioni ridotte dei suoi file.

Abbiamo parlato di come SageMaker può essere utilizzato out-of-the-box con il suo set di algoritmi preconfigurati, in particolare, abbiamo visto come implementare la previsione su un set di dati costituito da informazioni sull'inquinamento e sull'ambiente.

Infine, abbiamo visto come mettere in produzione un modello pronto per essere utilizzato, sfruttando l'API di SageMaker per creare una pipeline di distribuzione che tenga conto del problema Concept Drift, permettendo così frequenti aggiornamenti del modello in base all'evoluzione del set di dati nel tempo. Ciò è particolarmente vero per le serie temporali e i modelli di previsione, che migliorano man mano che il set di dati aumenta.

Siamo finalmente giunti alla fine del viaggio, sperando di farvi divertire e, naturalmente, di partire con qualcosa di utile su cui iniziare a lavorare. Come sempre sentiti libero di commentare dandoci le tue opinioni e idee. E i tuoi casi d'uso? Che tipo di dispositivi usi? Connettiti con noi e parlane!

Ci vediamo su su Proud2beCloud tra 14 giorni!

alla grande multinazionale, dal manifatturiero al terziario avanzato, aiutiamo le realtà più all'avanguardia a realizzare progetti innovativi in campo IT.

Get in touch

beSharp.it
proud2becloud@besharp.it