Data Engineering: Empowering Data Scientists in the Face of Missing Data

Colmeman
6 min read · Aug 13

Introduction

In the vast field of data science, the art of extracting insights from disparate data sources is essential. But what happens when these structured databases are absent? This is where data engineering plays a pivotal role, serving as a crucial tool for data scientists to methodically collect, refine and systematically store information from the far reaches of the web. This narrative embarks on a comprehensive exploration of the data engineering process, illuminated by a pertinent example involving a fictional scooter company christened Gans.

The Data Engineering Challenge

Faced with a shortage of easily accessible databases, the strategic importance of data engineering takes on a new dimension. This complex undertaking involves the capture of data from the web, followed by its careful structuring to facilitate its incorporation into a form suitable for comprehensive analysis. The symbiotic interplay between data engineering and data science is the backbone of the operationalisation of data from unconventional channels.

The Practical Example: Gans Scooter Company

As we embark on a virtual journey, we are immersed in an illustrative scenario that encapsulates the essence of data engineering. Within this scenario, Gans, a fictional scooter company, emerges as a central character. The company wants to gain comprehensive insights into cities, real-time weather forecasts and critical flight

data. The catch is that there is no existing database. This complex scenario serves as a canvas to demonstrate the dynamic way in which data engineering confronts, navigates and ultimately overcomes such challenges.

Phase I: Establishing the Foundational Fabric — The Local Pipeline

As we begin our journey, we delve into the initial phase — the establishment of a robust basic framework. Python takes on the role of a master craftsman, facilitating the construction of a local data pipeline.

Web Scraping: Unveiling Hidden Nuggets

Web scraping is emerging as a key technique for extracting data from online sources. With Python's Beautiful Soup library as a loyal ally, the digital realm is dissected, revealing hidden pockets of valuable data. In the context of our scenario, the web scraping methodology allows the extraction of insights about cities such as Berlin, Paris and Madrid. Information on coordinates, population demographics and country affiliations is meticulously extracted from the digital tapestry of Wikipedia.

```
from bs4 import BeautifulSoup as BS
import requests
import pandas as pd

urls =  ["https://en.wikipedia.org/wiki/Berlin",
        "https://en.wikipedia.org/wiki/Paris",
        "https://en.wikipedia.org/wiki/Madrid"]

responses = []

for url in urls:
    responses.append(requests.get(url))


for response in responses:
    if response.status_code != 200:
        print(f"There is an error {response.status_code}")
    else:
        print(f"The response status is: {response.status_code}")

soups = []

for response in responses:
    soups.append(BS(response.content, 'html.parser'))

data_df = pd.DataFrame()
```

```
city_names = []
for soup in soups:
    city_names.append(soup.find('span', class_='mw-page-title-main').get_text()
data_df['city'] = city_names

country_names = []
for soup in soups:
    for s in soup.select("table.infobox tbody tr th"):
        if s.text == "Country":
            country_name = s.next_sibling.text.strip()
            country_names.append(country_name)
            break

data_df['country'] = country_names
```

API Integration: Establishing Data Connectivity

APIs, those digital conduits that facilitate access to data, play a vital role. Python's robust 'requests' library proves to be a trusted accomplice, enabling the streamlined extraction of data from diverse APIs. Notably, these APIs include real-time sources, underscoring their relevance. The Open Weather Map API is used in the quest for real-time weather forecasts.

```
import requests
import pandas as pd
from datetime import datetime
import pytz


API_key = "YOUR API KEY"
weathers = []
cities =  ['berlin', 'paris', 'madrid']
for city in cities:
  weathers.append(requests.get(f"http://api.openweathermap.org/data/2.5/forecas

def get_weather_loop(cities):

  tz = pytz.timezone("Europe/Berlin")
  now = datetime.now().astimezone(tz)

  weather_dict = {'city': [],
                  'country': [],
                  'forecast_time': [],
                  'outlook': [],
                  'detailed_outlook': [],
                  'temperature': [],
                  'temperature_feels_like': [],
```

```
                    'clouds': [],
                    'rain': [],
                    'snow': [],
                    'wind_speed': [],
                    'wind_deg': [],
                    'humidity': [],
                    'pressure': []}

  for city in cities:
    url =  (f"http://api.openweathermap.org/data/2.5/forecast?q={city}&appid={A
    response = requests.get(url)
    weather_json = response.json()

    for i in weather_json['list']:
      weather_dict['city'].append(weather_json['city']['name'])
      weather_dict['country'].append(weather_json['city']['country'])
      weather_dict['forecast_time'].append(i['dt_txt'])
      weather_dict['outlook'].append(i['weather'][0]['main'])
      weather_dict['detailed_outlook'].append(i['weather'][0]['description'])
      weather_dict['temperature'].append(i['main']['temp'])
      weather_dict['temperature_feels_like'].append(i['main']['feels_like'])
      weather_dict['clouds'].append(i['clouds']['all'])
      try:
        weather_dict['rain'].append(i['rain']['3h'])
      except:
        weather_dict['rain'].append('0')
      try:
        weather_dict['snow'].append(i['snow']['3h'])
      except:
        weather_dict['snow'].append('0')
      weather_dict['wind_speed'].append(i['wind']['speed'])
      weather_dict['wind_deg'].append(i['wind']['deg'])
      weather_dict['humidity'].append(i['main']['humidity'])
      weather_dict['pressure'].append(i['main']['pressure'])

  return pd.DataFrame(weather_dict)

cities_weather = get_weather_loop(cities)
```

Similarly, the ICAO API emerges as a key player, providing real-time flight data relevant to nearby airports.

```
import requests
import pandas as pd
from datetime import datetime, date, timedelta
from pytz import timezone
```

```
icao_list = get_airports_loop(data_df)

def tomorrows_flight_arrivals(icao_list):
  today = datetime.now().astimezone(timezone('Europe/Berlin')).date()

  tomorrow = (today + timedelta(days=1))
  list_for_df = []
  for icao in icao_list:
    if icao is None: continue
    times = [["00:00","11:59"],["12:00","23:59"]]
    for time in times:
      url = f"https://aerodatabox.p.rapidapi.com/flights/airports/icao/{icao}/{
      querystring = {"withLeg":"true","direction":"Arrival","withCancelled":"fa
      headers = {
          'x-rapidapi-host': "aerodatabox.p.rapidapi.com",
          'x-rapidapi-key': 'Your API Key'
          }
      response = requests.request("GET", url, headers=headers, params=querystr
      if response.status_code == 204 : continue
      flights_json = response.json()
      for flight in flights_json['arrivals']:
        flights_dict = {}
        flights_dict['arrival_icao'] = icao
        flights_dict['arrival_time_local'] = flight['arrival'].get('scheduledT
        flights_dict['arrival_terminal'] = flight['arrival'].get('terminal', No
        flights_dict['departure_city'] = flight['departure']['airport'].get('na
        flights_dict['departure_icao'] = flight['departure']['airport'].get('ic
        flights_dict['departure_time_local'] = flight['departure'].get('schedul
        flights_dict['airline'] = flight['airline'].get('name', None)
        flights_dict['flight_number'] = flight.get('number', None)
        flights_dict['data_retrieved_on'] = datetime.now().astimezone(timezone(
        list_for_df.append(flights_dict)

  return pd.DataFrame(list_for_df)
```

Streamlining Data Management with Pandas, MySQL, and sqlalchemy

Once we've neatly organized all the required data in Pandas dataframes (our go-to), the next step is setting up a MySQL database in our chosen RDBMS. Here, we create distinct tables for different data categories: cities, weather, airports, and flights. These tables use primary and foreign keys to establish relationships and maintain data integrity.

For shifting dataframe data, we rely on the `sqlalchemy` library. With a simple import and request, our data seamlessly moves into existing MySQL tables. But `sqlalchemy` isn't just for data movement – it effortlessly brings data back to our notebook for

analysis. This library's versatility streamlines interactions between Python and MySQL, making data management a breeze.

```
schema="mydb"
host="127.0.0.1"
user="root"
password="Your Password"
port=3306
con = f'mysql+pymysql://{user}:{password}@{host}:{port}/{schema}'

cities_airports_flights_tomorrow.to_sql('flights', # renamed from "iss_positior
                                        if_exists='append',
                                        con=con,
                                        index=False)
```

Phase II: Transitioning to Cloud Terrain — The Cloud Pipeline

With the infrastructure of the local pipeline firmly in place, we traverse the trajectory towards the cloud — a realm synonymous with scalability, automation, and expansive possibilities.

Amazon.com Inc., Apache License 2.0 <http://www.apache.org/licenses/LICENSE-2.0>, via Wikimedia Commons

AWS RDS: Navigating Cloud-Based Database Management

In the arena of cloud computing, Amazon Web Services (AWS) unfurls the Relational Database Service (RDS) as a cardinal offering. This service provides a cloud-native platform for database management. By embracing AWS's MySQL

instance, a transition transpires from the constraints of physical hardware towards an ethereal domain characterized by data integrity, accessibility, and streamlined management.

AWS Lambda: Enabling Orchestrated Code Execution

AWS Lambda emerges as a bastion for serverless code execution. Within this context, Python code thrives, liberated from the confines of conventional servers. Additionally, pivotal libraries such as SQLAlchemy find their home as layers, bolstering code efficiency and performance. Subsequently, data operations seamlessly transpire within the domain of a MySQL database interlinked with the cloud.

Automation through AWS EventBridge: Choreographing Timely Execution

Automation, the hallmark of streamlined data engineering, becomes a reality through AWS EventBridge. This orchestration tool facilitates the automated execution of Lambda functions on a predetermined schedule. In this manner, a choreographed dance of data collection, processing, and assimilation transpires at specified intervals, ensuring a perpetually refreshed dataset.

Reflections and Considerations

Amidst the allure of cloud services lies a sobering caveat. As one ventures deeper, financial considerations come into play. It is imperative to reflect upon the potential costs associated with cloud utilization, ensuring a balanced approach that aligns technological aspirations with fiscal prudence.

Culmination: The Data Symphony

In conclusion, this journey into the realms of data engineering and its harmonious coexistence with data science reveals the latent potential of unconventional data sources. The story of Gans, the scooter company, highlights the central role that data engineering plays in overcoming the lack of structured databases. As an agent of transformation, data engineering effectively transforms raw, disparate data into actionable insights. In an ever-evolving technological landscape, data engineering stands resolute, seamlessly weaving fragmented pieces of data into a coherent tapestry of knowledge.

Data Engineering | Data Science

Edit profile

Written by Colmeman

2 Followers

Recommended from Medium