Data Engineering Trial Task – Taiyo.AI

Problem Statement

Objective:

Find, standardize, and continuously update data regarding construction and infrastructure projects and tenders in the state of California.

Part 1: Research and Data Sourcing

Task: Research and identify 5-10 reliable data sources about construction and infrastructure projects and tenders in California.

Methodology: Use a combination of online research and language models (e.g., OpenAI's GPT models) to identify these sources. Explicitly state how and why you used GPT or similar models in your research process.

Part 2: Data Extraction and Standardization

Task: From the provided data sources and your own list, suggest methods to scrape data using language model-based tools like OpenAI API, Mistral 7B, Llama2, or other open-source models.

Requirements: Demonstrate how you can build data products (DPs) to scrape data from multiple sources. Standardize the scraped data according to the guidelines provided in Data Standards List.

Part 3: Automation and Continuous Updating

Task: Propose a system for automating the data scraping and standardization processes.

Implementation:

Part 1: Research and Data Sourcing:

In construction and infrastructure development, access to accurate and timely data is paramount for stakeholders to make informed decisions, allocate resources effectively, and streamline project management processes. As the construction landscape evolves with technological advancements and shifting market dynamics, the importance of reliable data sources becomes increasingly evident. This introduction delves into the diverse array of data sources available for gathering information on construction projects and tenders in California,

shedding light on their respective advantages, drawbacks, and implications for data-driven decision-making.

Data Sources:

Government Websites serve as primary repositories of information on public infrastructure projects and tenders. These official platforms offer comprehensive insights into government-funded initiatives, ranging from transportation and utilities to public buildings and urban development. While these websites provide valuable data, challenges arise from the fragmentation of information across various departments and the variability in data formats and accessibility protocols between different government entities. Industry Publications and News Websites offer valuable insights into construction trends, project announcements, and industry developments. These sources provide a broader perspective on the construction landscape, facilitating market analysis and trend forecasting. However, reliance on publications and news websites may pose challenges in terms of data consistency, depth, and access restrictions to premium content. Construction Industry Databases cater specifically to the needs of construction professionals, aggregating information on projects, contractors, suppliers, and tenders. These databases offer sophisticated search functionalities and filters for targeted data retrieval. Nonetheless, access to comprehensive databases may require subscription fees, and data quality may vary between platforms, necessitating careful validation.

Public Procurement Portals established by government agencies and municipalities offer transparency in the procurement process by publishing invitations to bid, request for proposals (RFPs), and awarded contracts. While these portals enhance visibility into public procurement activities, navigating multiple portals and reconciling data formats pose logistical challenges, particularly for researchers and analysts seeking consolidated datasets. Trade Associations and Professional Networks foster collaboration and knowledge exchange among construction industry stakeholders. These platforms provide valuable resources, insights, and networking opportunities. However, data sourced from trade associations and professional networks may lack standardization and comprehensiveness, limiting its utility for comprehensive analysis.

Social Media and Online Forums serve as informal channels for sharing real-time updates, discussions, and user-generated content related to construction projects and tenders. While these platforms offer agility and immediacy in data dissemination, extracting structured data from unstructured sources presents computational challenges, necessitating advanced natural language processing (NLP) techniques for data extraction and analysis.

List Of Data Sources Used:

1. ConstructConnect - https://www.constructconnect.com/construction-near-me/california-construction-projects

2. ElkGrove - https://www.elkgrovecity.org/southeast-policy-area/development-projects
3. CityOfSanrafael - https://www.cityofsanrafael.org/major-planning-projects-2/
4. SantaMariaGroup - https://www.santamariagroup.com/projects
5. Highways.dot.gov - https://highways.dot.gov/federal-lands/projects/ca
6. Publicworks.com - https://ocip.ocpublicworks.com/service-areas/oc-infrastructure-programs/projects-and-studies

Part 2: Data Extraction and Standardization:

Data extraction and standardization for construction and infrastructure projects, the use of language model-based tools such as OpenAI API, Mistral 7B, and Llama2 offers promising avenues for automating the process and ensuring consistency in data collection. Leveraging these tools, alongside open-source models, enables the development of robust data products (DPs) capable of scraping data from diverse sources while adhering to standardized formats and guidelines.

Scraping Methods:

Utilizing language model-based tools for data scraping involves several approaches tailored to the characteristics of each data source:

1. **Web Scraping:** Language model-based tools can be employed to scrape structured data from websites hosting project information, tenders, and procurement portals. By parsing HTML content and applying NLP techniques, these tools can extract relevant data fields such as project names, descriptions, timelines, and locations.
2. **Textual Data Mining:** For unstructured data sources such as industry publications, news articles, and social media platforms, language models play a crucial role in mining textual data for key insights. By applying text summarization, sentiment analysis, and named entity recognition (NER) techniques, these tools can identify relevant information and extract structured data points for further analysis.
3. **API Integration:** Many government websites and industry databases offer Application Programming Interfaces (APIs) for accessing structured data programmatically. Language model-based tools can interface with these APIs to retrieve data in real-time, enabling dynamic updates and seamless integration with data products.

Standardization Process:

Standardizing scraped data according to predefined guidelines ensures consistency and interoperability across datasets. The standardization process involves the following steps:

1. **Data Cleaning:** Remove inconsistencies, duplicates, and irrelevant information from scraped datasets to ensure data integrity and accuracy.

2. **Normalization:** Standardize data formats, units of measurement, and naming conventions to facilitate data integration and analysis. For example, standardizing date formats to YYYY-MM-DD or converting currency values to a common currency.
3. **Entity Recognition:** Apply NLP techniques to identify and classify entities such as project names, locations, and stakeholders within the scraped data. This step facilitates data organization and categorization for downstream analysis.
4. **Schema Mapping:** Map scraped data fields to a predefined schema or data model, aligning with the guidelines provided in Data Standards List. This step ensures consistency in data representation and facilitates data interoperability across different sources and systems.

By implementing these methods and adhering to standardized practices, data engineers can effectively extract and standardize construction and infrastructure project data from multiple sources, enabling stakeholders to make informed decisions and drive innovation in project management and planning processes.

Data Standards List (Attributes):

Attribute Type: Generic

1. **original_id:**
   Description: Unique identifier generated by Taiyo (UUID functionality).
   Expectations: Expect the column to exist, expect column values to be of type UUID, and expect column values to be unique.

2. **aug_id:**
   Description: Augmented identifier.
   Expectations: Expect the column to exist, expect column values to be of type UUID, and expect column values to be unique.

3. **country_name:**
   Description: Name of the country.
   Expectations: Expect the column to exist, expect column values to be of type string, and expect column values to be unique.

4. **country_code:**
   Description: ISO 3-letter country code.
   Expectations: Expect the column to exist, expect column values to be of type string, and expect column values to be unique.

5. **map_coordinates:**
   Description: Geo Point of the region formatted as a JSON object.
   Expectations: Expect the column to exist, expect column values to be valid GeoJSON, and expect column values to be unique.

6. **url:**
   Description: URL of the website source.

Expectations: Expect the column to exist, expect column values to be valid URLs, and expect column values to be unique.

7. **region_name:**
   Description: Region name for a country according to World Bank Standards.
   Expectations: Expect the column to exist, and expect column values to be in a predefined set of region names.

8. **region_code:**
   Description: Region code for a region according to World Bank Standards.
   Expectations: Expect the column to exist, and expect column values to be in a predefined set of region codes.

Attribute Type: Projects and Tender

1. **title:**
   Description: Title for the tender/project.
   Expectations: Expect the column to exist, and expect column values to be in a predefined set.

2. **description:**
   Description: Summary description of the tender/project.
   Expectations: Expect the column to exist, and expect column values to be in a predefined set.

3. **status:**
   Description: Current status of the tender/project.
   Expectations: Expect the column to exist, and expect column values to be in a predefined set.

4. **Stages:**
   Description: Current stage of the tender/project.
   Expectations: Expect the column to exist, and expect column values to be in a predefined set.

5. **date:**
   Description: Date on which the information was first recorded or published.
   Expectations: Expect the column to exist, and expect column values to match a specified format.

6. **procurementMethod:**
   Description: Procurement method used for the relevant works, goods, or services.
   Expectations: Expect the column to exist, and expect column values to be in a predefined set.

7. **budget:**
   Description: Total upper estimated value of the procurement.

Expectations: Expect the column to exist, and expect column values to be in a predefined set.

8. **currency:**
   Description: Currency for each amount, specified using the uppercase 3-letter currency code from ISO4217.
   Expectations: Expect the column to exist, and expect column values to be valid currency codes.

9. **buyer:**
   Description: Entity whose budget will be used to pay for goods, works, or services related to a contract.
   Expectations: Expect the column to exist.

10. **sector:**
    Description: High-level categorization of the main sector related to the procurement process.
    Expectations: Expect the column to exist, and expect column values to be in a predefined set.

11. **subsector:**
    Description: Further subdivision of the sector the procurement process belongs to.
    Expectations: Expect the column to exist.

12. **data_source :**
    Description: source where the project found.
    Expectations: Expect the column to exist.

In certain instances, the selected data sources for construction and infrastructure projects in California might exhibit inconsistencies or lack essential variables. In such scenarios, manual data extraction from supplementary sources like Google or government databases becomes necessary to address these gaps and ensure data completeness. However, when dealing with a large volume of records, manual extraction becomes challenging and time-intensive, posing difficulties in efficiently handling and reconciling the extensive dataset. Therefore, a careful balance between automated extraction tools and manual intervention is crucial to effectively manage the data extraction process, ensuring both accuracy and efficiency in capturing the required information for analysis and decision-making purposes.

Code and Explanation:

Libraries Used:

```
import re
import uuid
import time
import requests
```

```
import schedule
import pandas as pd
from bs4 import BeautifulSoup
```

re: Python's regular expression library, used for pattern matching and string manipulation, facilitating text processing tasks.
uuid: Provides functions for generating universally unique identifiers (UUIDs), essential for creating unique identifiers in data processing workflows.
time: Offers functions for working with time-related tasks such as measuring code execution time or adding time delays in scripts.
requests: A Python library for making HTTP requests, commonly used for fetching web pages and interacting with web APIs.
schedule: Enables the scheduling of Python functions to run at specified intervals or times, useful for automating repetitive tasks like data scraping or processing.
pandas: A powerful data manipulation library in Python, widely used for data analysis and manipulation, providing data structures and functions for efficiently handling structured data.
BeautifulSoup: A Python library for parsing HTML and XML documents, facilitating web scraping tasks by providing tools to extract data from web pages easily.

```
# Existing data sources
datasource = [
   {"data_source": "Construct Connect", "url":
"https://www.constructconnect.com/construction-near-me/california-construction-
projects"},
   {"data_source": "ElkGrove", "url": "https://www.elkgrovecity.org/southeast-policy-
area/development-projects"},
   {"data_source": "CityOfSanrafael", "url": "https://www.cityofsanrafael.org/major-
planning-projects-2/"},
   {"data_source": "SantaMariaGroup", "url":
"https://www.santamariagroup.com/projects"},
   {"data_source": "Highways.dot.gov", "url": "https://highways.dot.gov/federal-
lands/projects/ca"},
   {"data_source": "Publicworks.com", "url": "https://ocip.ocpublicworks.com/service-
areas/oc-infrastructure-programs/projects-and-studies"}
]
```

```
# Declaration of Main DataFrame
main_df = pd.DataFrame(columns=["original_id", "aug_id", "country_name",
"country_code", "map_coordinates", "url","data_source", "region_name", "region_code",
"title", "description", "status", "stages", "date",  "procurementMethod", "budget",
"currency", "buyer", "sector", "subsector"])
```

```
#Helper Functions
#This data was collected from chatGPT, Bard and google
#Extra data needed required for below web scraped projects
```

```
ExtrenalProjectDataConstructConnect = [
  {
    "title": "Water / Sewer Construction in Sunol, California",
    "location": "Sunol, California",
    "sector": "Construction (COFOG 1)",
    "subsector": "Water & Sewerage (COFOG 11)",
    "procurementMethod": "Invitation to Bid/Request for Proposals",
    "map_coordinates": "Latitude 37.59438, Longitude -121.88857",
    "region_code": "94586",
    "buyer": "Sewer Contractors",
    "stages": [
      "Pre-construction",
      "Sitework and foundation",
      "Rough framing",
      "Exterior construction",
      "MEP (Mechanical, Electrical, Plumbing)",
      "Finishes and fixtures"
    ]
  },
  {
    "title": "Roads / Highways Construction in Acton, California",
    "location": "Acton, California",
    "sector": "Construction (COFOG 1)",
    "subsector": "Roads & Bridges (COFOG 12)",
    "procurementMethod": "Design-Build/Construction Manager/General Contractor",
    "map_coordinates": "Latitude 37.59438, Longitude -121.88857",
    "region_code": "94586",
    "buyer": "Leonida Builders",
    "stages": [
      "Pre-construction",
      "Sitework and foundation",
      "Rough framing",
      "Exterior construction",
      "MEP (Mechanical, Electrical, Plumbing)",
      "Finishes and fixtures"
    ]
  },
  .....
]
```

#Web Scraping data from constructconnect

```
def ScrapDataConstructConnect(construct_connect_url):
  response = requests.get(construct_connect_url)
  try:
```

```
    if response.status_code == 200:
        soup = BeautifulSoup(response.content, 'html.parser')
        project_list = soup.find_all(class_='project-list')
        for project in project_list:
            table = project.find('table', class_='display compact')
        rows = table.find_all('tr')
        row_data = []
        for row in rows:
            project_details = row.find_all('td')
            row_data.append(project_details)
        row_data1 = []
        for i in range(1, len(row_data), 6):
            subset = row_data[i]
            sublist = []
            for j in range(0, 9):
                td_value = str(subset[j])  # Convert to string
                pattern = re.compile(r'<td>(.*?)</td>')
                matches = re.findall(pattern, td_value)
                cleaned_matches = [match.strip() for match in matches]
                sublist.append(cleaned_matches[0])
            row_data1.append(sublist)
        n = len(main_df);
        for i in range(len(row_data1)):
            title = row_data1[i][0]
            if not any(main_df['title'] == title):
                j = n + i
                index = next((k for k, project in
enumerate(ExtrenalProjectDataConstructConnect) if project['title'] == row_data1[i][0]),
None)
                main_df.loc[j, 'original_id'] = j
                main_df.loc[j, 'aug_id'] = uuid.uuid4()
                main_df.loc[j, 'country_name'] = "California"
                main_df.loc[j, 'country_code'] = "US-CA"
                main_df.loc[j, 'map_coordinates'] =
ExtrenalProjectDataConstructConnect[index]['map_coordinates']
                main_df.loc[j, 'data_source'] = "ConstructConnect"
                main_df.loc[j, 'url'] = construct_connect_url
                main_df.loc[j, 'region_name'] = row_data1[i][1]
                main_df.loc[j, 'region_code'] =
ExtrenalProjectDataConstructConnect[index]['region_code']
                main_df.loc[j, 'title'] = row_data1[i][0]
                main_df.loc[j, 'description'] = row_data1[i][5]
                main_df.loc[j, 'status'] = row_data1[i][2]
                main_df.loc[j, 'stages'] = ExtrenalProjectDataConstructConnect[index]['stages']
                main_df.loc[j, 'date'] = row_data1[i][3]
                main_df.loc[j, 'procurementMethod'] =
ExtrenalProjectDataConstructConnect[index]['procurementMethod']
```

```
                main_df.loc[j, 'budget'] = row_data1[i][8]
                main_df.loc[j, 'currency'] = row_data1[i][8]
                main_df.loc[j, 'buyer'] = ExtrenalProjectDataConstructConnect[index]['buyer']
                main_df.loc[j, 'sector'] = ExtrenalProjectDataConstructConnect[index]['sector']
                main_df.loc[j, 'subsector'] =
    ExtrenalProjectDataConstructConnect[index]['subsector']

            else:
                print("Failed to fetch data from the URL:", construct_connect_url)
        except Exception as e:
print("An error occurred:", e)
```

The given Python function, ScrapDataConstructConnect, is designed to scrape data from the ConstructConnect website. It begins by sending an HTTP GET request to the specified URL. Upon receiving a successful response (status code 200), the BeautifulSoup library is used to parse the HTML content of the webpage. The function then identifies the project list elements and iterates over each project, extracting relevant data such as project title, description, status, date, and budget. This data is processed and added to a Pandas DataFrame named main_df, which serves as the primary data structure for storing the scraped project information. The function also checks for duplicate projects and adds new entries to main_df accordingly. In case of any errors during the scraping process, appropriate error handling is implemented to ensure graceful handling of exceptions. Overall, this function provides a systematic approach to extract, process, and store project data from the ConstructConnect website for further analysis or integration into other applications.

#Helper Functions
#This data was collected from chatGPT, Bard and google
#Extra data needed required for below web scraped projects

```
ExtrenalProjectDataElkGrove = [
    {
        "title": "Souza Dairy",
        "location": "Elk Grove, California",
        "sector": "Construction (COFOG 1)",
        "subsector": "Single-Family Homes, Site Development",
        "procurementMethod": "Bids by Invitation",
        "map_coordinates": "Latitude 37.59438, Longitude -121.88857",
        "region_code": "94586",
        "date": "01/06/2023",
        "budget": "$28,511.00"
    },
.....
]
```

#Web Scraping data from ElkGrove

```
def ScrapDataElkGrove(ElkGrove_url):
    response = requests.get(ElkGrove_url)
    try:
        if response.status_code == 200:
            soup = BeautifulSoup(response.content, 'html.parser')
            table = soup.find('table', class_='content-table')
            rows = table.find_all('tr')

            row_data = []
            for row in rows[1:]:
                td_data = [td.get_text(strip=True) for td in row.find_all('td')]
                td_data = td_data[:-1]
                ul_tag = row.find('ul')
                list_data = [li.get_text(strip=True) for li in ul_tag.find_all('li')]
                td_data.append(list_data)
                row_data.append(td_data)
            n = len(main_df);
            for i in range(len(row_data)):
                title = row_data[i][1]
                if not any(main_df['title'] == title):
                    j = n + i
                    index = next((k for k, project in enumerate(ExtrenalProjectDataElkGrove) if
project['title'] == title), None)
                    main_df.loc[j, 'original_id'] = j
                    main_df.loc[j, 'aug_id'] = uuid.uuid4()
                    main_df.loc[j, 'country_name'] = "California"
                    main_df.loc[j, 'country_code'] = "US-CA"
                    main_df.loc[j, 'map_coordinates'] =
ExtrenalProjectDataElkGrove[index]['map_coordinates']
                    main_df.loc[j, 'data_source'] = "ElkGrove"
                    main_df.loc[j, 'url'] = ElkGrove_url
                    main_df.loc[j, 'region_name'] = ExtrenalProjectDataElkGrove[index]['location']
                    main_df.loc[j, 'region_code'] =
ExtrenalProjectDataElkGrove[index]['region_code']
                    main_df.loc[j, 'title'] = row_data[i][1]
                    main_df.loc[j, 'description'] = row_data[i][2]
                    main_df.loc[j, 'status'] = row_data[i][4]
                    main_df.loc[j, 'stages'] = row_data[i][5]
                    main_df.loc[j, 'date'] = ExtrenalProjectDataElkGrove[index]['date']
                    main_df.loc[j, 'procurementMethod'] =
ExtrenalProjectDataElkGrove[index]['procurementMethod']
                    main_df.loc[j, 'budget'] = ExtrenalProjectDataElkGrove[index]['budget']
                    main_df.loc[j, 'currency'] = ExtrenalProjectDataElkGrove[index]['budget']
                    main_df.loc[j, 'buyer'] = row_data[i][3]
                    main_df.loc[j, 'sector'] = ExtrenalProjectDataElkGrove[index]['sector']
                    main_df.loc[j, 'subsector'] = ExtrenalProjectDataElkGrove[index]['subsector']
```

```
    else:
        print("Failed to fetch data from the URL:", ElkGrove_url)
  except Exception as e:
    print("An error occurred:", e)
```

The ScrapDataElkGrove function is designed to scrape data from the ElkGrove website. It begins by sending an HTTP GET request to the specified URL. Upon receiving a successful response (status code 200), the BeautifulSoup library is used to parse the HTML content of the webpage. The function then identifies the table containing project data and iterates over each row of the table, extracting relevant information such as project title, description, status, date, and budget. Additionally, it extracts any additional details listed as bullet points under each project and appends them to the project's data. This extracted data is then processed and added to a Pandas DataFrame named main_df, similar to the previous function. Duplicate project titles are checked to avoid redundancy, and new entries are added to main_df accordingly. Proper error handling is implemented to handle exceptions that may occur during the scraping process, ensuring the robustness of the function. Overall, this function provides a systematic approach to extract, process, and store project data from the ElkGrove website for further analysis or integration into other applications.

#Web Scraping data from CityOfSanrafael

```
def ScrapDataCityOfSanrafael(CityOfSanrafael_url):
  response = requests.get(CityOfSanrafael_url)
  try:
    if response.status_code == 200:
      soup = BeautifulSoup(response.content, 'html.parser')
      table = soup.find('table', class_='table')
      rows = table.find_all('tr')
      row_data = []
      for row in rows[1:]:
        td_data = [td.get_text(strip=True) for td in row.find_all('td')]
        td_data = td_data[1:]
        td_with_a = row.find('td')
        if td_with_a:
          a_tag = td_with_a.find('a')
          if a_tag:
            text = a_tag.get_text(strip=True)
            td_data.append(text)
        row_data.append(td_data)

      n = len(main_df);
      for i in range(len(row_data)):
        title = row_data[i][7]
        if not any(main_df['title'] == title):
          j = n + i
```

```
                main_df.loc[j, 'original_id'] = j
                main_df.loc[j, 'aug_id'] = uuid.uuid4()
                main_df.loc[j, 'country_name'] = "California"
                main_df.loc[j, 'country_code'] = "US-CA"
                main_df.loc[j, 'url'] = CityOfSanrafael_url
                main_df.loc[j, 'data_source'] = "CityOfSanrafael"
                main_df.loc[j, 'title'] = title
                main_df.loc[j, 'description'] = row_data[i][0]
                main_df.loc[j, 'status'] = row_data[i][6]
                main_df.loc[j, 'buyer'] = row_data[i][4]

        else:
            print("Failed to fetch data from the URL:", CityOfSanrafael_url)
    except Exception as e:
        print("An error occurred:", e)
```

The ScrapDataCityOfSanrafael function is responsible for scraping data from the CityOfSanrafael website. It begins by sending an HTTP GET request to the specified URL. If the response status code is 200 (indicating success), the BeautifulSoup library is utilized to parse the HTML content of the webpage. The function then identifies the table containing project data and iterates over each row of the table, extracting relevant information such as project title, description, status, and buyer. Additionally, if a hyperlink is present in the row, the text content of the hyperlink is also extracted and appended to the project's data. The extracted data is then processed and added to a Pandas DataFrame named main_df, similar to the previous functions. Duplicate project titles are checked to avoid redundancy, and new entries are added to main_df accordingly. Proper error handling is implemented to handle exceptions that may occur during the scraping process, ensuring the reliability of the function. Overall, this function provides a systematic approach to extract, process, and store project data from the CityOfSanrafael website for further analysis or integration into other applications.

#Web Scraping data from SantaMariaGroup

```
def ScrapDataSantaMariaGroup(SantaMariaGroup_url):
    response = requests.get(SantaMariaGroup_url)
    try:
        if response.status_code == 200:
            soup = BeautifulSoup(response.content, 'html.parser')
            div = soup.find_all('div', class_="fluid-engine")
            div = div[1:-1]
            row_data = []
            for i in div:
                rowi = []
                rowi.append(i.find('h3').text.strip())
                rowdiv = i.find_all('p', class_='sqsrte-small')
                dt = [p.text.strip() for p in rowdiv]
                rowi.append(dt[0])
```

```
            fields_to_extract = ["Location","Status", "Client", "SMG Role"]
            for field in fields_to_extract:
                pattern = re.compile(fr"{field}:\s*(.*?)$", re.IGNORECASE)
                field_value = None
                for text in dt:
                    match = pattern.search(text)
                    if match:
                        field_value = match.group(1)
                        break
                if field_value:
                    rowi.append(field_value)
                else:
                    rowi.append(" ")
            row_data.append(rowi)

        n = len(main_df);
        for i in range(len(row_data)):
            title = row_data[i][0]
            if not any(main_df['title'] == title):
                j = n + i
                main_df.loc[j, 'original_id'] = j
                main_df.loc[j, 'aug_id'] = uuid.uuid4()
                main_df.loc[j, 'country_name'] = "California"
                main_df.loc[j, 'country_code'] = "US-CA"
                main_df.loc[j, 'data_source'] = "SantaMariaGroup"
                main_df.loc[j, 'url'] = SantaMariaGroup_url
                main_df.loc[j, 'title'] = title
                main_df.loc[j, 'description'] = row_data[i][1]
                main_df.loc[j, 'region_name'] = row_data[i][2]
                main_df.loc[j, 'status'] = row_data[i][3]
                main_df.loc[j, 'buyer'] = row_data[i][4]
    else:
        print("Failed to fetch data from the URL:", SantaMariaGroup_url)
except Exception as e:
    print("An error occurred:", e)
```

The ScrapDataSantaMariaGroup function is designed to scrape data from the SantaMariaGroup website. It starts by sending an HTTP GET request to the specified URL and checks if the response status code is 200, indicating success. If successful, the BeautifulSoup library is used to parse the HTML content of the webpage. The function then searches for specific elements containing project information and iterates over each of them. Within each , it extracts the project title, description, location, status, client, and Santa Maria Group (SMG) role. Regular expressions are utilized to extract specific fields from the paragraph text. The extracted data is organized into a list of lists (row_data), with each inner list representing the details of a single project. Duplicate project titles are checked to avoid redundancy, and new entries are added to the main_df DataFrame accordingly. Proper error

handling is implemented to manage exceptions that may occur during the scraping process, ensuring the robustness of the function. Overall, this function provides an effective method to scrape project data from the SantaMariaGroup website for further analysis or integration into other applications.

#Web Scraping data from HighwaysDotGov

```
def ScrapDataHighwaysDotGov(HighwaysDotGov_url):
  response = requests.get(HighwaysDotGov_url)
  try:
    if response.status_code == 200:
      soup = BeautifulSoup(response.content, 'html.parser')
      div = soup.find_all('div', class_='views-element-container')
      div = div[3:]
      project_lst = []
      for i in div:
        table = i.find('tbody')
        status = i.find('h2')
        for tr in table.find_all('tr'):
          td_title = tr.find('td', class_='views-field views-field-title')
          project_name = td_title.text.strip()
          project_link = "https://highways.dot.gov" + td_title.find('a')['href']
          loc = tr.find('td', class_='views-field views-field-field-flh-location')
          loc = loc.text.strip() + ', California'
          project_lst.append([project_name, project_link,loc,status.text.strip()])
      for project in project_lst:
        response1 = requests.get(project[1])
        try:
          if response1.status_code == 200:
            soup = BeautifulSoup(response1.content, 'html.parser')
            section = soup.find('section', class_='section')
            body_text = section.find('div', class_='col-md-7 clearfix')
            body_text = body_text.text.strip()
            description_pattern =
r'^(.*?)(?:\s*Anticipated\s*Timeline|\s*Project\s*Documents:)'
            advertise_pattern = r'Anticipated Timeline\n(.*?)\n'
            description = re.search(description_pattern, body_text,
re.DOTALL).group(1).strip()
            advertise_match = re.search(advertise_pattern, body_text)
            if advertise_match:
              advertise = advertise_match.group(1).strip()
            else:
              advertise = ""
            contact = section.find('section', class_='contact-details')
            contact_div = contact.find('span', class_='field-content')
            buyer = contact_div.text.strip() + ", Federal Highway Administration, USDOT"
```

```
                project.append(description)
                project.append(advertise)
                project.append(buyer)
            else:
                print("Failed to fetch data from the URL:", HighwaysDotGov_url)
        except Exception as e:
            print("An error occurred:", e)

    n = len(main_df)
    for i in range(len(project_lst)):
        title = project_lst[i][0]
        if not any(main_df['title'] == title):
            j = n + i
            main_df.loc[j, 'original_id'] = j
            main_df.loc[j, 'aug_id'] = uuid.uuid4()
            main_df.loc[j, 'country_name'] = "California"
            main_df.loc[j, 'country_code'] = "US-CA"
            main_df.loc[j, 'url'] = project_lst[i][1]
            main_df.loc[j, 'data_source'] = "HighwaysDotGov"
            main_df.loc[j, 'title'] = title
            main_df.loc[j, 'region_name'] = project_lst[i][2]
            main_df.loc[j, 'status'] = project_lst[i][3]
            main_df.loc[j, 'description'] = project_lst[i][4]
            main_df.loc[j, 'date'] = project_lst[i][5]
            main_df.loc[j, 'buyer'] = project_lst[i][6]
    else:
        print("Failed to fetch data from the URL:", HighwaysDotGov_url)
except Exception as e:
    print("An error occurred:", e)
```

The ScrapDataHighwaysDotGov function is designed to scrape data from the HighwaysDotGov website. It begins by sending an HTTP GET request to the specified URL and checks if the response status code is 200, indicating success. If successful, BeautifulSoup is used to parse the HTML content of the webpage. The function then searches for specific elements containing project information and iterates over each of them. Within each , it extracts the project name, link, location, and status. Additionally, it sends a subsequent request to each project link to extract further details such as description, anticipated timeline, and contact information. Regular expressions are employed to extract specific patterns from the retrieved text. The extracted data is organized into a list of lists (project_lst), with each inner list representing the details of a single project. Duplicate project titles are checked to avoid redundancy, and new entries are added to the main_df DataFrame accordingly. Proper error handling is implemented to manage exceptions that may occur during the scraping process, ensuring the robustness of the function. Overall, this function provides an effective method to scrape project data from the HighwaysDotGov website for further analysis or integration into other applications.

#Web Scraping data from Publicworks.com

```
def ScrapDataPublicworks(Publicworks_url):
    response = requests.get(Publicworks_url)
    try:
        if response.status_code == 200:
            soup = BeautifulSoup(response.content, 'html.parser')
            main_div = soup.find('div', class_='block field-blocknodecounty-pagebody')
            div = main_div.find_all('tbody')
            project_lst = []
            for tbody in div:
                tr_tags = tbody.find_all('tr')
                for tr in tr_tags:
                    a_tag = tr.find('a')
                    project_name = a_tag.text.strip()
                    project_href = a_tag['href']
                    td_tags = tr.find_all('td')
                    project_data = [project_href] + [td.text.strip() for td in td_tags if td.text.strip()]
                    project_lst.append(project_data)
            for project in project_lst:
                if not project[0].startswith(("http://", "https://")):
                    project[0] = "https://ocip.ocpublicworks.com/" + project[0]

            for project in project_lst:
                response1 = requests.get(project[0])
                try:
                    if response1.status_code == 200:
                        soup = BeautifulSoup(response1.content, 'html.parser')
                        main = soup.find('main', class_='main-content main-content--with-sidebar')
                        text = main.find('p')
                        if text:
                            project.append(text.text.strip())
                    else:
                        print("Failed to fetch data from the URL:", Publicworks_url)
                except Exception as e:
                    print("An error occurred:", e)

            project_lst[8].insert(2, '')
            project_lst[5].insert(2, '')

            n = len(main_df)
            for i in range(len(project_lst)):
                title = project_lst[i][1]
                if not any(main_df['title'] == title):
                    j = n + i
                    main_df.loc[j, 'original_id'] = j
```

```
            main_df.loc[j, 'aug_id'] = uuid.uuid4()
            main_df.loc[j, 'country_name'] = "California"
            main_df.loc[j, 'country_code'] = "US-CA"
            main_df.loc[j, 'url'] = project_lst[i][0]
            main_df.loc[j, 'data_source'] = "Publicworks.com"
            main_df.loc[j, 'title'] = title
            main_df.loc[j, 'status'] = project_lst[i][len(project_lst[i])-2]
            main_df.loc[j, 'description'] = project_lst[i][len(project_lst[i])-1]
            main_df.loc[j, 'date'] = project_lst[i][2]

    else:
        print("Failed to fetch data from the URL:", Publicworks_url)
except Exception as e:
    print("An error occurred:", e)
```

The ScrapDataPublicworks function is responsible for scraping data from the Publicworks.com website. It begins by sending an HTTP GET request to the specified URL and checks if the response status code is 200, indicating success. If successful, BeautifulSoup is utilized to parse the HTML content of the webpage. The function then navigates to the main element containing the project information and iterates over each <tbody> element within it. Within each <tbody>, it extracts project details such as name, link, and other relevant information. For projects without a complete URL, it appends the base URL to the project link. Subsequently, the function sends a subsequent request to each project link to retrieve additional project details, such as descriptions. The extracted data is organized into a list of lists (project_lst). Certain modifications are made to the project list to accommodate missing data entries or fix inconsistencies. Finally, the function adds the scraped data to the main_df DataFrame, ensuring that duplicate project titles are not included. Proper error handling is implemented to manage any exceptions encountered during the scraping process, ensuring the robustness of the function. Overall, this function provides an effective method to scrape project data from the Publicworks.com website for further analysis or integration into other applications.

#Call for DataSource-1

```
construct_connect_url = next((source["url"] for source in datasource if source["data_source"] == "Construct Connect"), None)
ScrapDataConstructConnect(construct_connect_url)
```

#Call for DataSource-2

```
ElkGrove_url = next((source["url"] for source in datasource if source["data_source"] == "ElkGrove"), None)
ScrapDataElkGrove(ElkGrove_url)
```

#Call for DataSource-3

```
CityOfSanrafael_url = next((source["url"] for source in datasource if source["data_source"] == "CityOfSanrafael"), None)
ScrapDataCityOfSanrafael(CityOfSanrafael_url)
```

#Call for DataSource-4

```
SantaMariaGroup_url = next((source["url"] for source in datasource if source["data_source"] == "SantaMariaGroup"), None)
ScrapDataSantaMariaGroup(SantaMariaGroup_url)
```

#Call for DataSource-5

```
HighwaysDotGov_url = next((source["url"] for source in datasource if source["data_source"] == "Highways.dot.gov"), None)
ScrapDataHighwaysDotGov(HighwaysDotGov_url)
```

#Call for DataSource-6

```
Publicworks_url = next((source["url"] for source in datasource if source["data_source"] == "Publicworks.com"), None)
ScrapDataPublicworks(Publicworks_url)
```

Part 3: Automation and Continuous Updating:

Automating the data scraping and standardization processes involves creating a systematic approach to regularly retrieve and process data from various sources without manual intervention. Here's a detailed explanation of the steps involved in automating this task:

1. **Identify Data Sources:** Begin by identifying the sources from which you want to scrape data. These could include websites, APIs, databases, or any other relevant sources. Ensure that these sources are reliable and provide the necessary information required for your analysis.

2. **Schedule Scraping Jobs:** Implement a scheduling mechanism to periodically execute scraping jobs. This can be achieved using tools like cron jobs, Task Scheduler (for Windows), or cloud-based scheduling services. Schedule the scraping jobs to run at appropriate intervals based on the frequency of data updates on the sources.

3. **Scalable Scraping Infrastructure:** Design a scalable infrastructure to handle the scraping tasks efficiently, especially when dealing with a large volume of data or multiple sources. Consider using cloud computing services like AWS, Google Cloud Platform, or Azure to dynamically scale resources based on demand.

4. **Error Handling and Monitoring:** Implement robust error handling mechanisms to deal with issues such as network errors, connection timeouts, or changes in the website structure. Set up logging and monitoring systems to track the execution of scraping jobs, identify errors, and troubleshoot issues in real-time.

5. **Data Standardization:** Develop scripts or pipelines to standardize the scraped data according to predefined data standards or schemas. This may involve cleaning, deduplicating, and transforming the data into a consistent format suitable for analysis or storage.

6. **Data Storage and Management:** Decide on the storage mechanism for the scraped data. Options include relational databases (e.g., MySQL, PostgreSQL), NoSQL databases (e.g., MongoDB, Cassandra), or data lakes (e.g., Amazon S3, Google Cloud Storage). Choose a solution that aligns with your data requirements and scalability needs.

7. **Integration with ETL Pipelines:** Integrate the scraping and standardization processes into Extract, Transform, Load (ETL) pipelines if you're dealing with a broader data ecosystem. ETL tools like Apache Airflow, Apache NiFi, or custom-built pipelines can automate the flow of data from source to destination seamlessly.

8. **Continuous Improvement:** Regularly review and refine the scraping and standardization processes to adapt to changes in data sources, website structures, or business requirements. Monitor the performance of the automated system and make necessary adjustments to optimize efficiency and accuracy.

By following these steps, you can establish a robust system for automating the data scraping and standardization processes, ensuring that your data remains up-to-date, accurate, and readily available for analysis or other downstream applications.

 # A system for automating the data scraping and standardization processes
 # Function to execute the scraping and updating process

```
def run_process():

  #Call for DataSource-1
  construct_connect_url = next((source["url"] for source in datasource if source["data_source"] == "Construct Connect"), None)
  ScrapDataConstructConnect(construct_connect_url)

  # #Call for DataSource-2
  ElkGrove_url = next((source["url"] for source in datasource if source["data_source"] == "ElkGrove"), None)
  ScrapDataElkGrove(ElkGrove_url)

  #Call for DataSource-3
  CityOfSanrafael_url = next((source["url"] for source in datasource if source["data_source"] == "CityOfSanrafael"), None)
  ScrapDataCityOfSanrafael(CityOfSanrafael_url)

  #Call for DataSource-4
  SantaMariaGroup_url = next((source["url"] for source in datasource if source["data_source"] == "SantaMariaGroup"), None)
  ScrapDataSantaMariaGroup(SantaMariaGroup_url)

  #Call for DataSource-5
```

```
    HighwaysDotGov_url = next((source["url"] for source in datasource if source["data_source"] == "Highways.dot.gov"), None)
    ScrapDataHighwaysDotGov(HighwaysDotGov_url)

    #Call for DataSource-6
    Publicworks_url = next((source["url"] for source in datasource if source["data_source"] == "Publicworks.com"), None)
    ScrapDataPublicworks(Publicworks_url)

# Schedule the execution to run every day at a specific time
schedule.every().day.at("00:00").do(run_process)

# Infinite loop to continuously check and run scheduled tasks
while True:
    # Run pending scheduled tasks
    schedule.run_pending()
    # Sleep for 1 minute to avoid high CPU usage
    time.sleep(60)
```

This code snippet outlines a system for automating the data scraping and standardization processes, which aligns with the requirements outlined in Part 3. Here's an explanation of how each component contributes to automating the process:

1. Function to Execute the Scraping and Updating Process (run_process): This function orchestrates the execution of scraping functions for each data source. It retrieves the URLs for each data source from a predefined list (datasource) and calls the respective scraping functions (ScrapDataConstructConnect, ScrapDataElkGrove, etc.). By iterating through each data source and calling the appropriate scraping function, this function automates the process of fetching and updating data from multiple sources.

2. Schedule the Execution to Run Every Day at a Specific Time: Using the schedule library, the script schedules the run_process function to execute every day at midnight (00:00). This scheduling mechanism ensures that the scraping and updating process occurs regularly and consistently without manual intervention.

3. Infinite Loop to Continuously Check and Run Scheduled Tasks: The script enters an infinite loop to continuously check for pending scheduled tasks and execute them. Inside the loop, schedule.run_pending() checks if there are any pending tasks to execute. If any are found, it runs them. After running pending tasks, the script sleeps for 60 seconds (time.sleep(60)) to avoid high CPU usage and unnecessary processing. This loop ensures that the scraping and updating process continues to run indefinitely, periodically fetching and updating data according to the predefined schedule.

Overall, this system automates the data scraping and standardization processes by scheduling the execution of scraping tasks at regular intervals and continuously monitoring and executing scheduled tasks without manual intervention.

To ensure continuous updates of the data sources, the proposed methodology leverages scheduled tasks or cron jobs, which are commonly used in production environments for automating repetitive tasks. Here's how this process works in detail:

How the data sources will be continuously updated:

In the proposed system, scheduled data scraping plays a pivotal role in automating the process of fetching and updating data from various sources. Through scheduled tasks, such as cron jobs or Task Scheduler tasks, the system orchestrates the execution of data scraping functions at predefined intervals. This scheduling ensures that the data retrieval process occurs regularly, allowing for timely updates to be captured from the sources. By setting up these tasks to run daily or at other appropriate intervals, the system can adapt to the varying frequencies at which data updates occur across different sources.

Each data source integrated into the system is assigned a designated scraping function responsible for fetching and updating data from that specific source. These scraping functions are tailored to the structure and characteristics of each source, allowing for efficient extraction of relevant information. By encapsulating the scraping logic within individual functions, the system maintains modularity and flexibility, enabling seamless integration of new data sources or modifications to existing ones. Additionally, this approach facilitates troubleshooting and debugging, as issues related to specific sources can be isolated and addressed independently.

The scheduling mechanism ensures that the scraping functions are executed automatically without requiring manual intervention. This automation streamlines the data update process, freeing up valuable time and resources that would otherwise be spent on manual data retrieval tasks. By adhering to a predefined schedule, the system minimizes the risk of data staleness and ensures that stakeholders have access to the most up-to-date information available. Overall, scheduled data scraping forms the backbone of the system's ability to continuously gather and update data from diverse sources, empowering users with timely insights for decision-making and analysis.

The use of cron jobs or similar scheduling tools for ongoing data updates :

In a production environment, the use of cron jobs or similar scheduling tools is fundamental for automating repetitive tasks such as data scraping and updating. Cron, which stands for "chronograph," is a time-based job scheduler in Unix-like operating systems. It allows system administrators to schedule commands or scripts to run periodically at specified times, dates, or intervals. This capability is essential for ensuring that critical tasks, like fetching and updating data from external sources, occur regularly and consistently without manual intervention. Similarly, on Windows operating systems, Task Scheduler fulfills a comparable role to cron jobs. Task Scheduler enables users to schedule tasks to run at specific times or intervals, offering a user-friendly interface for creating and managing scheduled tasks. Like cron jobs, Task Scheduler allows for the automation of various system tasks, including data scraping and updating processes. By defining the frequency and timing of these

tasks, administrators can ensure that data is fetched and updated at appropriate intervals, maintaining the relevance and accuracy of the information stored in the system.

In the context of the proposed system, cron jobs or Task Scheduler tasks can be configured to execute the script containing the data scraping and updating process (in this case, the run_process function) at the desired frequency. For example, a cron job can be set up to run the script once per day at midnight, ensuring that data from various sources is fetched and updated daily. This automated scheduling mechanism eliminates the need for manual intervention and reduces the risk of human error, enhancing the reliability and efficiency of the data scraping and updating process in a production environment.