Pratyusha Duvvuri (dpratyus), Ike Kilinc (aykilinc), Diego Martinez (diegom), Joel Miller (jgmiller)

17-480/780: Project Proposal

API Design - Fall 2020 - Team08

CDC Wonder API

Description / Documentation | Sample Code

API Description

The CDC Wonder API is a query tool from the Centers for Disease Control (CDC) that provides access to a collection of online databases for the analysis of public health data. These databases collect statistics from the United States such as births, detailed or compressed mortality rates, and causes of death.

To access the data in these databases, the API exposes an endpoint which users of the API can access through an HTTP POST request which contains an XML document that specifies all of the parameters needed to define the specific data that the user wishes to receive from the database.

However, this API does have some shortcomings that become readily apparent when formatting this XML parameter document and when parsing the XML response. These limitations take many forms: there is excessive, unnecessary verbosity both in specifying unnecessary parameters and in boilerplate code always necessary to parse and visualize the response data, there are hidden limitations and undocumented cases in which the API fails to return the expected data, and there are dozens of parameters with cryptic names that are impossible to make sense of and make the API very unapproachable.

Sample Code

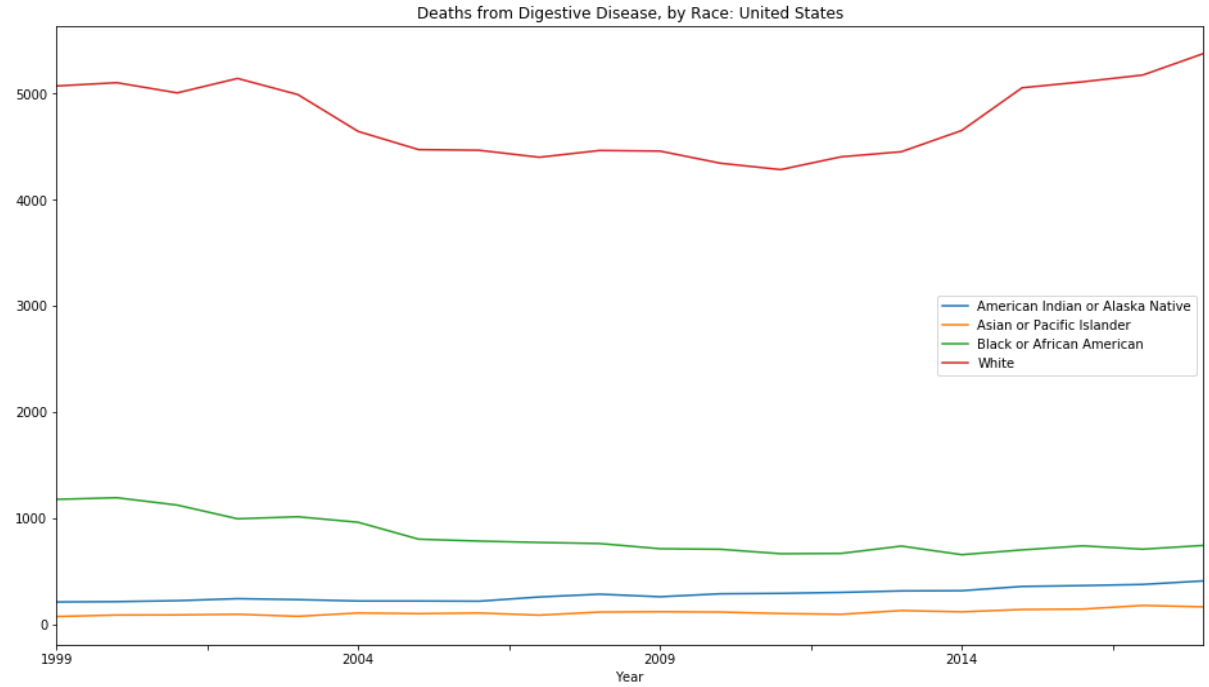
```
# One of many examples where users must needlessly append "*None*" or "*All*" to every
XML request field. Cumbersome.
b_parameters = {
    "B_1": "D76.V1-level1",
    "B_2": "D76.V8",
    "B_3": "*None*",
    "B_4": "*None*",
    "B_5": "*None*"
}

m_parameters = {
    "M_1": "D76.M1",   # Deaths, must be included
    "M_2": "D76.M2",   # Population, must be included
    "M_3": "D76.M3",   # Crude rate, must be included
    #"M_31": "D76.M31",        # Standard error (crude rate)
    #"M_32": "D76.M32"         # 95% confidence interval (crude rate)
    "M_41": "D76.M41", # Standard error (age-adjusted rate)
    "M_42": "D76.M42"  # 95% confidence interval (age-adjusted rate)
}

f_parameters = {
    "F_D76.V1": ["*All*"], # year/month
    "F_D76.V10": ["*All*"], # Census Regions - dont change
# How is a user or a later viewer of this code supposed to intuitively know what these
codes are (without significant Googling)?
    "F_D76.V2": ["K00-K92"], # ICD-10 Codes
    "F_D76.V27": ["*All*"], # HHS Regions - dont change
    "F_D76.V9": ["*All*"] # State County - dont change
}

i_parameters = {
    "I_D76.V1": "*All* (All Dates)",  # year/month
    "I_D76.V10": "*All* (The United States)", # Census Regions - dont change
    "I_D76.V2": "K00-K92 (Diseases of the digestive system)", # ICD-10 Codes
    "I_D76.V27": "*All* (The United States)", # HHS Regions - dont change
    "I_D76.V9": "*All* (The United States)" # State County - dont change
}

v_parameters = {
    "V_D76.V1": "",         # Year/Month
    "V_D76.V10": "",        # Census Regions
    "V_D76.V11": "*All*",   # 2006 Urbanization
    "V_D76.V12": "*All*",   # ICD-10 130 Cause List (Infants)
    "V_D76.V17": "*All*",   # Hispanic Origin
    "V_D76.V19": "*All*",   # 2013 Urbanization
    "V_D76.V2": "",         # ICD-10 Codes
    "V_D76.V20": "*All*",   # Autopsy
    "V_D76.V21": "*All*",   # Place of Death
    "V_D76.V22": "*All*",   # Injury Intent
    "V_D76.V23": "*All*",   # Injury Mechanism and All Other Leading Causes
    "V_D76.V24": "*All*",   # Weekday
    "V_D76.V25": "*All*",   # Drug/Alcohol Induced Causes
    "V_D76.V27": "",        # HHS Regions
```

```
    "V_D76.V4": "*All*",    # ICD-10 113 Cause List
    "V_D76.V5": ["15-24", "25-34", "35-44"], # Ten-Year Age Groups
    "V_D76.V5": "*All*",
    "V_D76.V51": "*All*",   # Five-Year Age Groups
    "V_D76.V51": ["45-49", "50-54", "55-59", "60-64"],
    "V_D76.V52": "*All*",   # Single-Year Ages
    "V_D76.V6": "00",       # Infant Age Groups
    "V_D76.V7": "*All*",    # Gender
    "V_D76.V8": "*All*",    # Race
    "V_D76.V9": ""          # State/County
}

o_parameters = {
    "O_V10_fmode": "freg",    # Use regular finder and ignore v parameter value
    "O_V1_fmode": "freg",     # Use regular finder and ignore v parameter value
    "O_V27_fmode": "freg",    # Use regular finder and ignore v parameter value
    "O_V2_fmode": "freg",     # Use regular finder and ignore v parameter value
    "O_V9_fmode": "freg",     # Use regular finder and ignore v parameter value
    "O_aar": "aar_std",       # age-adjusted rates
    "O_aar_pop": "0000",      # population selection for age-adjusted rates
    "O_age": "D76.V5",        # select age-group (e.g. ten-year, five-year,
single-year, infant groups)
    "O_javascript": "on",     # Set to on by default
    "O_location": "D76.V9",   # select location variable to use (e.g. state/county,
census, hhs regions)
    "O_precision": "1",       # decimal places
    "O_rate_per": "100000",   # rates calculated per X persons
    "O_show_totals": "false",  # Show totals for
    "O_timeout": "300",
    "O_title": "Digestive Disease Deaths, by Age Group",    # title for data run
    "O_ucd": "D76.V2",        # select underlying cause of death category
    "O_urban": "D76.V19"      # select urbanization category
}

vm_parameters = {
    "VM_D76.M6_D76.V10": "",        # Location
    "VM_D76.M6_D76.V17": "*All*",   # Hispanic-Origin
    "VM_D76.M6_D76.V1_S": "*All*",  # Year
    "VM_D76.M6_D76.V7": "*All*",    # Gender
    "VM_D76.M6_D76.V8": "*All*"     # Race
}

# These never change, but must always be included...
misc_parameters = {
    "action-Send": "Send",
    "finder-stage-D76.V1": "codeset",
    "finder-stage-D76.V1": "codeset",
    "finder-stage-D76.V2": "codeset",
    "finder-stage-D76.V27": "codeset",
    "finder-stage-D76.V9": "codeset",
    "stage": "request"
}
```

```
# Far too much boilerplate to simply format the request.
def createParameterList(parameterList):
    """Helper function to create a parameter list from a dictionary object"""

    parameterString = ""

    for key in parameterList:
        parameterString += "<parameter>\n"
        parameterString += "<name>" + key + "</name>\n"

        if isinstance(parameterList[key], list):
            for value in parameterList[key]:
                parameterString += "<value>" + value + "</value>\n"
        else:
            parameterString += "<value>" + parameterList[key] + "</value>\n"

        parameterString += "</parameter>\n"

    return parameterString

xml_request = "<request-parameters>\n"
xml_request += createParameterList(b_parameters)
xml_request += createParameterList(m_parameters)
xml_request += createParameterList(f_parameters)
xml_request += createParameterList(i_parameters)
xml_request += createParameterList(o_parameters)
xml_request += createParameterList(vm_parameters)
xml_request += createParameterList(v_parameters)
xml_request += createParameterList(misc_parameters)
xml_request += "</request-parameters>"

import requests

# Step out of the 1950's, a single API call for this breadth of information is
unnecessary and amalgamates the numerous purposes for requesting the data into one
convoluted hodgepodge.
url = "https://wonder.cdc.gov/controller/datarequest/D76"
response = requests.post(url, data={"request_xml": xml_request,
"accept_datause_restrictions": "true"})

# There is no understanding of WHY an error took place, only that it SOME arbitrary
error was present.
if response.status_code == 200:
    data = response.text
else:
    print("something went wrong")

# BeautifulSoup library facilitates parsing of XML response
import bs4 as bs

# This library facilitates 2-dimensional array operations and visualization
import pandas as pd
```

```
# All of this boilerplate just to format the response data into a Pandas Dataframe?? A
user should not need to do all this work that the API ~absolutely~ could and should be
doing for them.
def xml2df(xml_data):
    """ This function grabs the root of the XML document and iterates over
        the 'r' (row) and 'c' (column) tags of the data-table
        Rows with a 'v' attribute contain a numerical value
        Rows with a 'l attribute contain a text label and may contain an
        additional 'r' (rowspan) tag which identifies how many rows the value
        should be added. If present, that label will be added to the following
        rows of the data table.

        Function returns a two-dimensional array or data frame that may be
        used by the pandas library."""

    root = bs.BeautifulSoup(xml_data,"lxml")
    all_records = []
    row_number = 0
    rows = root.find_all("r")

    for row in rows:
        if row_number >= len(all_records):
            all_records.append([])

        for cell in row.find_all("c"):
            if 'v' in cell.attrs:
             try:
               all_records[row_number].append(float(cell.attrs["v"].replace(',','')))
             except ValueError:
               all_records[row_number].append(cell.attrs["v"])

             else:
               if 'r' not in cell.attrs:
                    all_records[row_number].append(cell.attrs["l"])
               else:
                    for row_index in range(int(cell.attrs["r"])):
                       if (row_number + row_index) >= len(all_records):
                          all_records.append([])
                          all_records[row_number + row_index].append(cell.attrs["l"])
                       else:
                          all_records[row_number + row_index].append(cell.attrs["l"])
        row_number += 1
    return all_records

data_frame = xml2df(data)

df = pd.DataFrame(data=data_frame, columns=["Year", "Race", "Deaths", "Population",
"Crude Rate", "Age-adjusted Rate", "Age-adjusted Rate Standard Error"])

df.head()

# Load matplotlib for plotting and instruct jupyter to display figures inline
from matplotlib import pyplot as plt
%matplotlib inline
```

```
# Group total number of deaths by year
df.groupby(by=['Year']).sum()['Deaths'].plot(title='Deaths from Digestive Disease:
United States');
fig = plt.gcf()
fig.set_size_inches(16, 9)

# Create mult-line chart for deaths by race

# Store figure and axis for shared plot
fig, ax = plt.subplots()

# Store labels for all race groups
labels = []

# For each group in the groupby object, grab the 'Race' label and create a line plot
for it
for key, grp in df.groupby(['Race']):
    ax = grp.plot(ax=ax, kind='line', x='Year', y='Deaths')
    labels.append(key)

# Set the labels for each line using the group labels
lines, _ = ax.get_legend_handles_labels()
ax.legend(lines, labels, loc='best')

# Configure chart size and title
fig = plt.gcf()
fig.set_size_inches(16, 9)
plt.title("Deaths from Digestive Disease, by Race: United States");
```



Critical Flaw #1: Non-Essential Verbosity

The CDC's Wonder API mandates a broadly unnecessary degree of unimportant work from any user who hopes to employ the CDC's vast swathes of data. Ranging from unreasonably verbose request parameter definitions to masses of boilerplate necessary for processing the data returned in the server response, verbosity dominates the API and overwhelms users.

First evaluating the formatting of the XML request file, we notice that significant sections of the parameter lists provide no additional information and serve only as clutter that lazily forces users to do simple work that the server receiving the API call ought to be handling. For example, the v_parameters header section below contains *over 20* parameters that are specified as their default values of either the empty string or *All*, while only a *single* parameter offers non-default data that will modify the response data.

```
v_parameters = {
    "V_D76.V1": "",         # Year/Month
    "V_D76.V10": "",        # Census Regions
    "V_D76.V11": "*All*",   # 2006 Urbanization
    "V_D76.V12": "*All*",   # ICD-10 130 Cause List (Infants)
    "V_D76.V17": "*All*",   # Hispanic Origin
    "V_D76.V19": "*All*",   # 2013 Urbanization
    "V_D76.V2": "",         # ICD-10 Codes
    "V_D76.V20": "*All*",   # Autopsy
    "V_D76.V21": "*All*",   # Place of Death
    "V_D76.V22": "*All*",   # Injury Intent
    "V_D76.V23": "*All*",   # Injury Mechanism and All Other Leading Causes
    "V_D76.V24": "*All*",   # Weekday
    "V_D76.V25": "*All*",   # Drug/Alcohol Induced Causes
    "V_D76.V27": "",        # HHS Regions
    "V_D76.V4": "*All*",    # ICD-10 113 Cause List
    "V_D76.V5": ["15-24", "25-34", "35-44"], # Ten-Year Age Groups
    "V_D76.V51": "*All*",   # Five-Year Age Groups
    "V_D76.V52": "*All*",   # Single-Year Ages
    "V_D76.V6": "00",       # Infant Age Groups
    "V_D76.V7": "*All*",    # Gender
    "V_D76.V8": "*All*",    # Race
    "V_D76.V9": ""          # State/County
}
```

We intend to significantly reduce the size of the user-defined input by first eliminating all default-valued parameters and requiring only those parameters that a user intends to alter. In the case of this example, a user would only need to specify the Ten-Year Age Groups, while comfortably excluding all the rest. Furthermore, we do intend to overall reduce the masses of structs and XML request formatting that must be conducted on the part of the user by changing the main user-accessible portion of the API to be a single SQL command.

Lastly, we hope to significantly increase the accessibility of the API by going much further than the current API in additionally formatting other operations such as the processing of the API response into a usable format. Currently, the conversion of the XML data into a beautiful and convenient Pandas

Dataframe structure in conducted with the following sample code.

```
def xml2df(xml_data):
    """ This function grabs the root of the XML document and iterates over
        the 'r' (row) and 'c' (column) tags of the data-table
        Rows with a 'v' attribute contain a numerical value
        Rows with a 'l attribute contain a text label and may contain an
        additional 'r' (rowspan) tag which identifies how many rows the value
        should be added. If present, that label will be added to the following
        rows of the data table.

        Function returns a two-dimensional array or data frame that may be
        used by the pandas library."""

    root = bs.BeautifulSoup(xml_data,"lxml")
    all_records = []
    row_number = 0
    rows = root.find_all("r")

    for row in rows:
        if row_number >= len(all_records):
            all_records.append([])

        for cell in row.find_all("c"):
            if 'v' in cell.attrs:
             try:
               all_records[row_number].append(float(cell.attrs["v"].replace(',','')))
             except ValueError:
               all_records[row_number].append(cell.attrs["v"])

             else:
               if 'r' not in cell.attrs:
                    all_records[row_number].append(cell.attrs["l"])
               else:
                    for row_index in range(int(cell.attrs["r"])):
                       if (row_number + row_index) >= len(all_records):
                          all_records.append([])
                          all_records[row_number + row_index].append(cell.attrs["l"])
                       else:
                          all_records[row_number + row_index].append(cell.attrs["l"])
        row_number += 1
    return all_records

data_frame = xml2df(data)
```

This mountain of boilerplate, among others, ought to be handled by the API and converted into any one of many formats the user might decide upon. In our revised API, we will provide significantly facilitated utilities for converting response data into more valuable, more accessible formats.

Critical Flaw #2: Age Grouping

The Wonder API allows the user to retrieve data from within specific age groups, but this is deceptively complex as there are only specific 10-, 5-, and single-year age groups that the user can specify. So

accessing data from an arbitrary age group involves choosing the correct combination of 10-, 5-, and single-year age groups. This causes sample code for accessing ages 15-44 to look like this:

```
v_parameters = {
    ...
    "V_D76.V5": ["15-24", "25-34", "35-44"], # Ten-Year Age Groups
    "V_D76.V51": "*All*",   # Five-Year Age Groups
    "V_D76.V52": "*All*",   # Single-Year Ages
    ...
}
```

This is incredibly unintuitive in and of itself, but there is another issue on top of this that involves protecting privacy within the dataset that prevent users from accessing any arbitrary dataset. This constraint is not specified and does not have relevant documentation, so we will need to experiment and figure out the requirements.

We intend to enable access to permitted age-grouped data by creating easy to use retrieval methods that document the limitations on what age-grouped data may be retrieved. We intend to take in intuitive input such as a desired age range (from age x to age y) and create the appropriate request automatically. Then we will either return the desired data or helpful exceptions that should allow the user to properly alter their request.

Critical Flaw #3: Lack of Documentation

There is a lack of substantial official API documentation to apply to most use cases. For example, it isn't mentioned that *All* needs to be specified as a "filter" for a category, if you don't want the results to be filtered. For example, for the f_params below,

```
f_parameters = {
    "F_D76.V1": ["*All*"], # year/month
    "F_D76.V10": ["*All*"], # Census Regions - dont change
    "F_D76.V2": ["K00-K92"], # ICD-10 Codes
    "F_D76.V27": ["*All*"], # HHS Regions - dont change
    "F_D76.V9": ["*All*"] # State County - dont change
}
```

there is no way of knowing that this is the format of the API and these are the inputs required, other than looking through the examples to figure out the right way to specify for each of the parameters.

The user also has no way to figure out what parameters the user needs to specify in the first place. The only way to figure out what structure of query works or doesn't is through trial and error.

To fix this issue, we will provide documentation and intuitive wrappers around basic functionality / function names to better explain what our API does. Our API will take care of adding these necessary but unspecified parameters when formatting a request and require only that the user provide specification for the parameters they care about.

Vision for Final Revised API

This API isn't the nicest, clearest, sweetest API… it sucks. The documentation is practically non-existent. It is a nightmare for users to figure out how to use this API just by combing through the two example requests and responses provided on the API website. There is an insane amount of boilerplate and verbosity associated with each request. The response is in XML which the user would have to process and parse, to do anything useful with the response data. The four of us were able to get a better understanding of the API only through a separate github repository with documented sample code!

Our vision is to make a well-documented, SQL-like API that wraps around this REST API and where users can only provide necessary parameters and filters in a few lines of code. The API implementation would take care of the boilerplate associated with making the request and parsing the response into well defined objects, as well as some of the nasty error handling that usually comes from a REST API.

We plan on focusing on the "breadth" aspect of fixing this API, meaning that we will try to minimize problems common to all use cases (such as excessive boilerplate) first. We will then focus on a subset of the data and improving the API to help with accessing and interpreting a specialized subset of data.

References

https://github.com/alipphardt/cdc-wonder-api
https://wonder.cdc.gov/wonder/help/WONDER-API.html
https://github.com/ikekilinc/CDCWonderPy