

Lab 12: Introduction to dataCommons

Collaborators

Write names in this cell:

```
# Run this cell to set up your notebook
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

import numpy as np
import pandas as pd
import sklearn
import matplotlib.pyplot as plt
import seaborn as sns

import plotly.offline as py
import plotly.graph_objs as go

sns.set()
sns.set_context("talk")
py.init_notebook_mode(connected=False)
```



So far in DS100, you have largely worked with data represented in tabular forms such as Pandas dataframes. However, there are many different ways of expressing data that go beyond the table. In this lab, we'll present [dataCommons](#), a tool that represents data as a *knowledge graph* instead of a table. We will:

1. Discuss what a knowledge graph is and how it compares to tabular data you've seen so far
2. Introduce dataCommons and how to ingest its data for analysis
3. Use these statistics to build a simple linear model that predicts the prevalence of obesity in US Cities.

Note that dataCommons is also an early release project. You may find some bugs! If so please post on Piazza and we'll respond as soon as possible. You can also email antaresc@berkeley.edu for assistance.

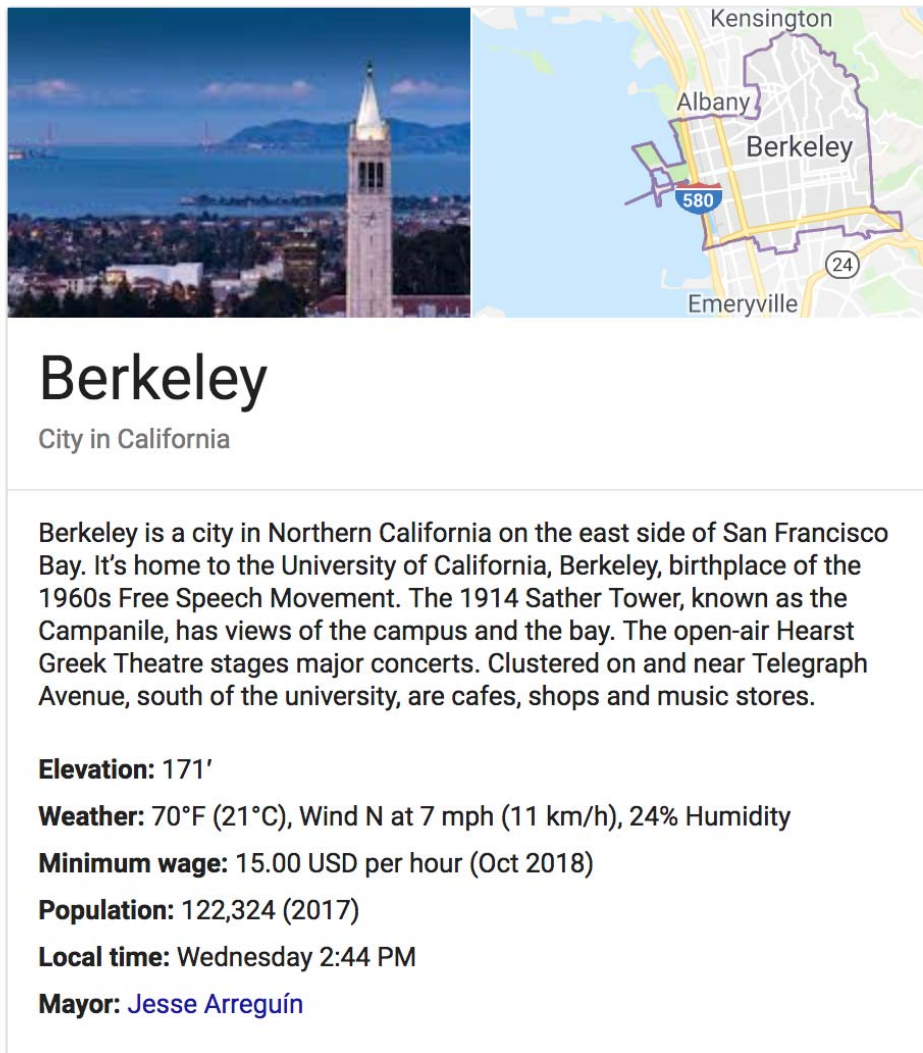
Handy Resources for this Lab

Here is a list of links that you may find handy when completing this lab.

- The [notebook](#) used for the dataCommons demo.
- A [tutorial](#) for getting started with dataCommons query API
- The dataCommons [documentation page](#)

Knowledge Graphs

You may have noticed that sometimes when you search on Google, Bing, or Yahoo for a topic like "Berkeley, CA", the search engine displays a panel containing metadata about the result like so:



The image shows a search result panel for Berkeley, California. At the top, there are two images: on the left, a photograph of the Sather Tower (Campanile) at the University of California, Berkeley, with the city and bay in the background; on the right, a map showing Berkeley's location relative to neighboring cities like Kensington, Albany, Emeryville, and the San Francisco Bay Area, with highways 580 and 24 marked. Below the images, the word "Berkeley" is prominently displayed in a large, bold font, followed by "City in California" in a smaller font. A descriptive paragraph follows, detailing Berkeley's location, its status as the birthplace of the 1960s Free Speech Movement, and mentioning key landmarks like the Sather Tower and Hearst Greek Theatre. Below this, several key facts are listed in a structured format: Elevation (171'), Weather (70°F, Wind N at 7 mph, 24% Humidity), Minimum wage (15.00 USD per hour), Population (122,324 in 2017), Local time (Wednesday 2:44 PM), and Mayor (Jesse Arreguin).

Berkeley
City in California

Berkeley is a city in Northern California on the east side of San Francisco Bay. It's home to the University of California, Berkeley, birthplace of the 1960s Free Speech Movement. The 1914 Sather Tower, known as the Campanile, has views of the campus and the bay. The open-air Hearst Greek Theatre stages major concerts. Clustered on and near Telegraph Avenue, south of the university, are cafes, shops and music stores.

Elevation: 171'

Weather: 70°F (21°C), Wind N at 7 mph (11 km/h), 24% Humidity

Minimum wage: 15.00 USD per hour (Oct 2018)

Population: 122,324 (2017)

Local time: Wednesday 2:44 PM

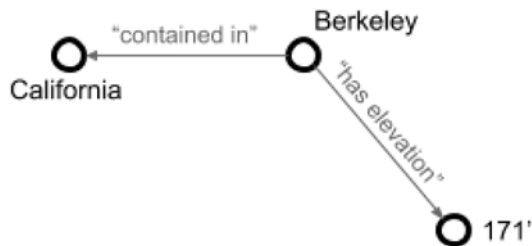
Mayor: [Jesse Arreguin](#)

These panels are built dynamically by querying *knowledge graphs*, an alternate way of representing data that captures the relation between [entities](#) that one hopes to describe. The idea behind knowledge graphs are as follows. Suppose we have a qualitative statement like:

Berkeley is contained in the State of California

We can think of "Berkeley" and "California" as entities while "contained in" as a property relating these entities together. Specifically, "Berkeley" is related to "California" by the relation specified by "contained in". We can also think of quantitative data in this model as well. The fact that "Berkeley" has an elevation of 171' (52 meters) is a statement relating the entity "Berkeley" with the literal "171'" through the property defined by "has elevation".

A natural way of representing these entities and their relationships is to model them as a network or [graph](#). This representation is called a *knowledge graph* and consists of nodes representing entities and edges labeled by relations between the entities. In the above example, our statements would translate to the following knowledge graph:



Knowledge graphs have become a ubiquitous model for storing data and you're likely to have already encountered them through search panels like that above! As stated previously, the information in these panels are stored in knowledge graphs. For this specific examples, Google/Bing/Yahoo each have an entity representing "Berkeley, CA" in their knowledge graphs with edges to entities representing its elevation, population, and mayor [Jesse Arreguín](#).

Ontologies

It's important to consider how to query knowledge graphs for information, and to do so, we'll need to answer these questions:

1. What kind of entities can the knowledge graph contain
2. What kind of relations are defined between the entities

To answer these questions, a knowledge graph is equipped with an *ontology* or a restricted vocabulary that defines a set of *types* that entities can be categorized as and *properties* that relate entities of given types. There are many ontologies standardized across the web. One example is the [Schema.org](#) vocabulary which defines *types* including "[State](#)" and "[City](#)," as well as *properties* including "[containedInPlace](#)" and "[elevation](#)".

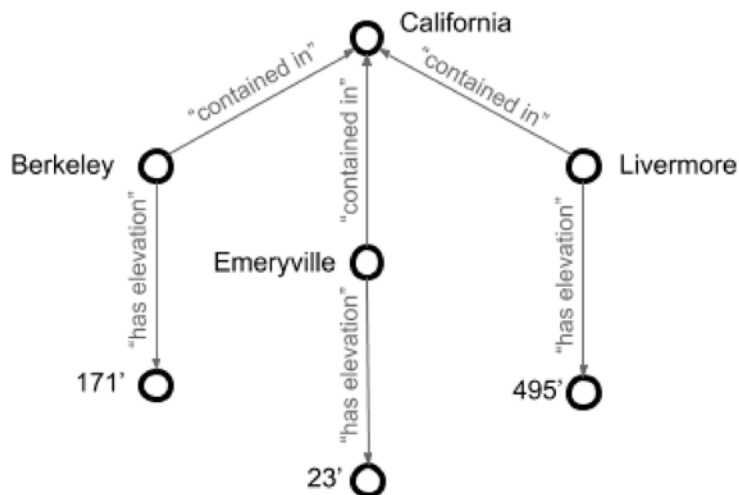
Using this vocabulary, we can express the information contained in the panel above as the statements

- Berkeley is containedInPlace of the State of California
- Berkeley has elevation "171 feet"

Caveat: In Schema.org, "containedInPlace" and "elevation" don't relate directly to "City" or "State". This is not too important for the lab, but they instead relate indirectly through entities like "[Place](#)" and "[GeoCoordinates](#)" and through the property "[geo](#)". These choices are sometimes made so that the vocabulary can easily be extended as new use cases arise. You can read more about how Schema.org models entities and properties [here](#)!

Converting to a Table

Even though knowledge graphs have a nice representation as a network, most of our statistical tools operate on tables! Luckily for us, we can convert between graphical and tabular view. Suppose our knowledge graph contains a number of statements regarding the elevation of cities in California.



We would really like a table where each row contains a city, elevation pair like so:

	City	Elevation
0	Berkeley	171
1	Emeryville	23
2	Livermore	495

Because the ontology tells us what kinds of entities the knowledge graph contains, we can create a single column by asking for everything of type "City"!

	City
0	Berkeley
1	Emeryville
2	Livermore

The ontology also tells us that the "elevation" property relates cities to text literals representing the city's elevation. We can construct the elevation column by querying, for each row, the entity related to the city in that row via "elevation":

	City	Elevation
0	Berkeley	171
1	Emeryville	23

"Emeryville" has "elevation" "23"

▼ Introducing dataCommons

[DataCommons](#) is an open knowledge graph that connects data from different sources (US Census, NOAA, Bureau of Labor Statistics, and more) under a single, unified representation. It contains statements about real world objects like

- The [University of California Berkeley](#) is a university located in [Berkeley, CA](#)
- [Berkeley, CA](#) has an elevation of 52 meters (171 feet)
- The [population of all persons in Maryland](#) has a total [count](#) of 5,959,902.

In the graph view, our nodes would be entities like [Berkeley](#). Every node has a type corresponding to what the node represents. For example, the [University of California Berkeley](#) is a [CollegeOrUniversity](#). Properties between entities are represented by edges between these nodes. For example, the statement "the University of California is located in the State of Berkeley" is represented in the graph as two nodes: "the University of California" and "Berkeley" with an edge labeled "[location](#)" pointing from UC Berkeley to Berkeley.

Note that the statement "UC Berkeley is a CollegeOrUniversity" is built directly into the graph using the "[typeOf](#)" property linking "UC Berkeley" to a node representing type "CollegeOrUniversity". Indeed, one way of "equipping" a knowledge graph with an ontology is to add nodes representing words in the ontology directly into the knowledge graph!

Datacommons closely follows the Schema.org data model and uses a superset of Schema.org vocabulary to provide a common set of types and properties.

The dataCommons Browser

The dataCommons [browser](#) provides a human-readable way of exploring data in the graph. Searching in the browser for an entity like [Berkeley](#), takes you to a page listing all properties related to the entity (i.e. nodes connected to the given entity via incoming and outgoing edges). Each entry in this list contains the property's name (for example [elevation](#)), the value of the property ('52'), and the *provenance* of the statement ([Wikidata](#)).

One of the first topics discussed in DS100 is the importance of data quality since more data does not always lead to a more accurate model! Because dataCommons unifies data from multiple sources, it is important to publish where the data is coming from. Thus when exploring data from dataCommons, it is important to consider the provenance before ingesting the data for an analytic task.

An important property for all entities is the **dcid**. The **dcid** (dataCommons identifier) is a unique identifier assigned to each entity in the knowledge graph. With this identifier, you will be able to search for and query information on the given entity using the Python API which we'll describe next. The **dcid** is listed at the top of the page next to "About: " and also in the list of properties.

Python API

The Python [Query API](#) provides functions for users to extract structured information from Datacommons programmatically as [Pandas](#) Dataframes. This makes integrating dataCommons into common statistical workflows very easy. Datacommons has already been installed on datahub, so to get started, load the client as follows:

```
!pip install --upgrade -q datacommons

import datacommons

dc = datacommons.Client()
```

You'll need to authenticate with your @berkeley.edu email before proceeding. Once done, we can begin looking at how to ingest data from dataCommons to build statistical models!

▼ Case Study: Prevalence of Obesity in 500 US Cities

As you've already seen in this course, data cleaning and curation can be a tricky task made even more so when you have to join different datasets into a single data frame. An important feature of dataCommons is the ability to seamlessly join across multiple data sources in one query which we'll demonstrate with the following case study.

Obesity is well known to correlate with health factors such as high blood pressure, but is also known to correlate with economic factors such as low-income, unemployment, etc [1][2]. The Center for Disease Control (CDC) provides prevalence percentages on health conditions such as [obesity](#), [high blood pressure](#), and [high cholesterol](#) for approximately 500 major cities in the US (e.g. [San Francisco](#), [New York](#), and [Austin](#)). Meanwhile, the US Bureau of Labor Statistics provides [unemployment rates](#) while the US Census provides [poverty rates](#) for most cities across the United States.

Even though these statistics come from different datasets across different government agencies with different storage formats, dataCommons surfaces each of these in a single, uniform knowledge graph. In fact, you can see this in the [browser](#) by looking at the *provenance* column. Let's use the data in dataCommons to create a linear regression model that incorporates variables:

- Prevalence of high blood pressure
- Unemployment rate
- Percent of population living with income below the poverty line

to predict the prevalence of obesity in the 500 cities that the CDC provides data for. One thing you may note is that the US Census also provides employment statistics (you can see this by navigating to the "employment" and "employmentStatus" sections for [San Francisco](#) and observing the different provenances). Our choice of using statistics from the Bureau of Labor Statistics is purely demonstrative, but it would be interesting to see if similar results can be reproduced using US Census employment statistics.

To get started, we'll need the cities that CDC provides data for. The dataCommons Python API provides [read_dataframe](#), a function that accesses cached dataframes. We've created a frame to get you started on this case study.

```
data = dc.read_dataframe('CDC-Cities-37ef5131')
data.head(5)
```

	City	CityName
0	City	Text
1	dc/5n6xfw1	Kansas City
10	dc/1c7ts02	Jonesboro
100	dc/rj1gww3	Medford
101	dc/lygh3t	Honolulu

Notice that the top row contains the type of the contents in each column. We'll need to keep that until we're done querying data.

▼ Querying for StatisticalPopulations and Observations

The first thing we want is a column containing the prevalence of Obesity statistics for the cities in the City column. Let's understand how dataCommons stores statistical data. There are two types of entities of interest: [StatisticalPopulations](#) and [Observations](#).

A StatisticalPopulation defines a collection of things of a certain type (i.e the set of all [Persons in Kansas City](#)). Given a population, we can have different Observations such as one corresponding to the [population of Kansas City between 2011-2015](#). For this task, we'll want the population of all [Persons in Kansas City with chronic condition obesity](#). To get more specific populations, we can define different constraints on our StatisticalPopulations such as `chronicCondition='Obesity'`.

The API defines functions allowing us to fetch data over these two types. We can use the [get_populations](#) function to get the populations we want. Given a Pandas dataframe containing a column `seed_col_name` of dcids representing where the populations are located (for example the 'City' column in our current dataframe `data`), returns a dataframe containing a new column labeled `new_col_name` containing dcids corresponding to StatisticalPopulations constrained by the `population_type` and any additional kwargs.

The new column is constructed in exactly the same way as we discussed in the Converting to a Table section above. For each dcid in the `seed_col_name` column, the function queries for a StatisticalPopulation located in that entity. A nice thing to note is that because the API functions return Pandas dataframes, you'll be able to apply many of the tools you've already encountered in DS100 to analyze the data after querying is finished. Consider the following example for querying populations of persons with 'chronicCondition' of 'Obesity'


```
data = dc.get_populations(pd_table=data,
                          seed_col_name='City',
                          new_col_name='Person/Obesity',
                          population_type='Person',
                          max_rows=500,
                          chronicCondition='Obesity')
data.head(5)
```

	City	CityName	Person/Obesity
0	City	Text	Population
1	dc/5n6xfw1	Kansas City	dc/p/8cfdc4pn9tl2f
2	dc/1c7ts02	Jonesboro	dc/p/vqqhfbrz7l765
3	dc/rj1gww3	Medford	dc/p/mscwg43zbd2g3
4	dc/lygh3t	Honolulu	dc/p/3redng4jvmsx6

Next, we want the prevalence percentages associated with this population. We use the [get_observations](#) function to get the 'crudePrevalence' for each population referred to in our Person/Obesity column and specify that we only wish to include statistics from 2015. Compare the function call below to the browser page for the [population in Kansas City with obesity](#). At the bottom, you can find the Observation node for 2015 'crudePrevalence'.

```
data = dc.get_observations(pd_table=data,
                           seed_col_name='Person/Obesity',
                           new_col_name='Obesity',
                           start_date='2015-01-01',
                           end_date='2015-01-01',
                           measured_property='crudePrevalence',
                           stats_type='percent',
                           max_rows=500)

data.head(5)
```



	City	CityName	Person/Obesity	Obesity
0	City	Text	Population	Observation
1	dc/5n6xfw1	Kansas City	dc/p/8cfdc4pn9tl2f	37.1
2	dc/1c7ts02	Jonesboro	dc/p/vqqhfbrz7l765	33
3	dc/rj1gww3	Medford	dc/p/msc wd43zbd2g3	29.1
4	dc/lygh3t	Honolulu	dc/p/3redng4jvmsx6	21.7

We note that temporal data in dataCommons is currently provided at yearly and monthly granularity. Start dates and end dates are currently fixed to be the first date of the year/month respectively. Let's now query for the other statistics we need!

▼ Question 1a: Query for Prevalence of High Blood Pressure

The next statistic we want is the unemployment percentage for cities in the City column for 2015. Call `get_populations` and `get_observations` to get a column containing crude prevalence percentages of high blood pressure. Consider this example [population](#) and [observation](#) when writing the queries.

```
### BEGIN SOLUTION
# data = dc.read_dataframe('CDC-Cities-37ef5131')
# data.head(5)

data = dc.get_populations(pd_table=data,
                           seed_col_name='City',
                           new_col_name='Person/Bphigh',
                           population_type='Person',
                           max_rows=500,
                           chronicCondition='Bphigh')

data = dc.get_observations(pd_table=data,
                           seed_col_name='Person/Bphigh',
                           new_col_name='Bphigh',
                           start_date='2015-01-01',
                           end_date='2015-01-01',
                           measured_property='crudePrevalence',
                           stats_type='percent',
                           max_rows=500)

### END SOLUTION

data.head(5)
```



	City	CityName	Person/Obesity	Obesity	Person/Bphigh	Bphigh
0	City	Text	Population	Observation	Population	Observation
1	dc/5n6xfw1	Kansas City	dc/p/8cfdc4pn9tl2f	37.1	dc/p/xd7qbbrfddek6	33.4
2	dc/1c7ts02	Jonesboro	dc/p/vqqhfbrz7l765	33	dc/p/m3yk3n3dk2j16	33.9

▼ Question 1b: Query for Unemployment

Next, we'll want unemployment percentages for each city for 2015 provided by the Bureau of Labor Statistics. Query for the population and observations to get a column of unemployment percentages. Consider this example [population](#) and [observation](#) when writing the queries.

```
### BEGIN SOLUTION
```

```
data = dc.get_populations(pd_table=data,
                          seed_col_name='City',
                          new_col_name='Person/Unemployed',
                          population_type='Person',
                          max_rows=500,
                          employmentStatus='Unemployed')

data = dc.get_observations(pd_table=data,
                           seed_col_name='Person/Unemployed',
                           new_col_name='Unemployed',
                           start_date='2015-01-01',
                           end_date='2015-01-01',
                           measured_property='percent',
                           stats_type='percent',
                           max_rows=500)
```

```
### END SOLUTION
```

```
data.head(5)
```

	City	CityName	Person/Obesity	Obesity	Person/Bphigh	Bphigh
0	City	Text	Population	Observation	Population	Observation
1	dc/5n6xfw1	Kansas City	dc/p/8cfdc4pn9tl2f	37.1	dc/p/xd7qbbrfddek6	33.4
2	dc/1c7ts02	Jonesboro	dc/p/vqqhfbrz7l765	33	dc/p/m3yk3n3dk2j16	33.9
3	dc/rj1gww3	Medford	dc/p/msc wd43zbd2g3	29.1	dc/p/he5pfjqn9s3qg	30.4
4	dc/lygh3t	Honolulu	dc/p/3redng4jvmsx6	21.7	dc/p/1pny281k58bl6	31.5

▼ Question 1c: Query for Poverty Rate

Finally, we'll query for poverty rate. The US Census provides counts for two values: the [population of all persons](#) and the [population of all persons with income below poverty line in the past 12 months](#). We'll query for these two values from the 2011-2015 census and divide the values to get a column of poverty rates later on.

First, query for the populations and observations to get a column of counts of all persons in each City. As an example, consider this [observation](#).

```
### BEGIN SOLUTION
```

```
data = dc.get_populations(pd_table=data,
                          seed_col_name='City',
                          new_col_name='Person',
                          population_type='Person',
                          max_rows=500)

data = dc.get_observations(pd_table=data,
                          seed_col_name='Person',
                          new_col_name='Population',
                          start_date='2011-01-01',
                          end_date='2015-01-01',
                          measured_property='count',
                          stats_type='count',
                          max_rows=500)
```

```
### END SOLUTION
```

```
data.head(5)
```

City	CityName	Person/Obesity	Obesity	Person/Bphigh	Bphigh	Person/L
City	Text	Population	Observation	Population	Observation	
16xfw1	Kansas City	dc/p/8cfdc4pn9tl2f	37.1	dc/p/xd7qbbrfddek6	33.4	dc/p/v2z
c7ts02	Jonesboro	dc/p/vqqhfbrz7l765	33	dc/p/m3yk3n3dk2j16	33.9	dc/p/djj
1gww3	Medford	dc/p/mscwg43zbd2g3	29.1	dc/p/he5pfjqn9s3qg	30.4	dc/p/z
/lygh3t	Honolulu	dc/p/3redng4jvmsx6	21.7	dc/p/1pny281k58bl6	31.5	

Next, query for the populations and observations to get a column of counts of all persons with income below the poverty line. As an example, consider this [observation](#)

```
### BEGIN SOLUTION
```

```
data = dc.get_populations(pd_table=data,
                          seed_col_name='City',
                          new_col_name='PersonBelowPoverty',
                          population_type='Person',
                          max_rows=500,
                          povertyStatus='USC_IncomeInThePast12MonthsBelowPovertyLevel')

data = dc.get_observations(pd_table=data,
                          seed_col_name='PersonBelowPoverty',
                          new_col_name='PopulationPoverty',
                          start_date='2011-01-01',
                          end_date='2015-01-01',
                          measured_property='count',
                          stats_type='count',
                          max_rows=500)
```

```
### END SOLUTION
```

```
data.head(5)
```



	City	CityName	Person/Obesity	Obesity	Person/Bphigh	Bphigh
0	City	Text	Population	Observation	Population	Observation
1	dc/5n6xfw1	Kansas City	dc/p/8cfdc4pn9tl2f	37.1	dc/p/xd7qbbrfddek6	33.4
2	dc/1c7ts02	Jonesboro	dc/p/vqqhfbrz7l765	33	dc/p/m3yk3n3dk2j16	33.9
3	dc/rj1gww3	Medford	dc/p/msc wd43zbd2g3	29.1	dc/p/he5pfjqn9s3qg	30.4

▼ Cleaning the Data

Great! We have all the statistics we'll need for this case study queried from three different data sources without having to write additional code to perform the joins. However, we'll still need to perform three data cleanup tasks to prepare the data for analysis:

1. Select the column of city names and all observation columns.
2. Convert the observation columns to numerical types. Since dataCommons is still in its early release stage, there are a couple of oddities such as how numerical observations are stored as text values!
3. Remove any missing rows with missing values.

▼ Question 2: Getting the Final DataFrame

Let's generate the final dataframe. We've provided a helper function that does all the cleanup. Call it with the appropriate arguments to perform the cleanup steps described above. Additionally, add a column that calculates the poverty rate for each city.

```
def select_and_clean(df, info_cols, data_cols):
    """ Selects columns in INFO_COLS + DATA_COLS from DF and performs cleanup
    tasks described above:
    - Remove top row
    - Convert DATA_COLS to numerical types.
    - Remove rows with missing statistics.
    """
    table = df[1:][info_cols + data_cols].copy()
    table[data_cols] = table[data_cols].apply(pd.to_numeric, errors='coerce')
    table = table.dropna()
    return table

# Get the cleaned table
cleaned = data.copy()
### BEGIN SOLUTION

cleaned = select_and_clean(cleaned, ['CityName'], ['Obesity', 'Bphigh', 'Unemployed', 'Popula-
cleaned['PovertyRate'] = cleaned['PopulationPoverty'] / cleaned['Population']

### END SOLUTION
cleaned.head(5)
```



	CityName	Obesity	Bphigh	Unemployed	Population	PopulationPoverty	PovertyRate
1	Kansas City	37.1	33.4	6.9	467990.0	87504.0	0.186978
2	Jonesboro	33.0	33.9	5.6	71576.0	16130.0	0.225355
3	Medford	29.1	30.4	7.1	77579.0	17596.0	0.226814

▼ Exploring the Data

Now's a good time to explore how each variable (high blood pressure prevalence, unemployment rate, and poverty rate) correlate with obesity prevalence. As stated before, previous research has shown that each of these variables tend to correlate positively with the prevalence of Obesity. Does our data show this?

```
# Plot Bphigh vs. Obesity
cleaned.plot.scatter(x='Bphigh', y='Obesity')
plt.title('High Blood Pressure Prevalence v. Obesity Prevalence')
plt.plot()

# Plot Unemployment with Obesity
cleaned.plot.scatter(x='Unemployed', y='Obesity')
plt.title('Unemployment Rate v. Obesity Prevalence')
plt.plot()

# Plot PovertyRate with Obesity
cleaned.plot.scatter(x='PovertyRate', y='Obesity')
plt.title('Poverty Rate v. Obesity Prevalence')
plt.plot()
```





Looks like each of variable does correlate positively with obesity prevalence. As an added bonus, we won't need to transform the data as well. Let's jump right into building our linear regression!

▼ Modeling the Data

We'll be predicting the prevalence of obesity with the following linear model.

$$f_{\theta}(x) = \theta_0 + \theta_1(\text{high blood pressure}) + \theta_2(\text{unemployment}) + \theta_3(\text{poverty rate})$$

▼ Question 3: Create the Linear Model

Let's start by creating our training and test sets. In the cell below, assign X to be the sub-table containing high blood pressure prevalences, unemployment rates, and poverty rates and Y to be the column of obesity prevalences. Then split X and Y into training and test sets.

```
### BEGIN SOLUTION
X = cleaned[['Bphigh', 'Unemployed', 'PovertyRate']]
Y = cleaned['Obesity']

x_train, x_test, y_train, y_test = train_test_split(X, Y)
### END SOLUTION
```

Next, fit a linear model to the training set. You can use sklearn's [LinearRegression](#).

```
### BEGIN SOLUTION
model = LinearRegression(fit_intercept=True)
model.fit(x_train, y_train)

### END SOLUTION
print('Model Intercept: {}'.format(model.intercept_))
print('Model Coefficients: {}'.format(model.coef_[0]))
```

```
↳ Model Intercept: 5.134269192582909
   Model Coefficients: 0.6571651428180798
```

▼ Analyzing the Model

We now have a trained model, but how well does it perform?

```
def mse(y_pred, y_true):
    """ Compute the mean squared error of 'y_pred' and 'y_true'. """
    return float(np.sum((y_pred - y_true) ** 2)) / len(y_pred)

train_pred = model.predict(x_train)
test_pred = model.predict(x_test)

print('Training Error: {}'.format(mse(train_pred, y_train)))
print('Test Error: {}'.format(mse(test_pred, y_test)))
```

```
↳ Training Error: 11.653130903136104
   Test Error: 9.275306876393278
```

We can also display a plot of the residuals

```
y_res = (test_pred - y_test) / y_test
```

```
# Plot the results
plt.title("Residual Proportions Plot")
plt.xlabel("Test Data Row Index")
plt.ylabel("resid. / actual obesity prevalence")
plt.scatter(y_res.index, y_res.values)
plt.show()
```

```
↳
```




How well does your model perform? We were able to achieve an MSE for the test set of approximately 10% points from the observed obesity prevalence. Our model was also able to fit the data with the max residual bounded by $\pm 30\%$, which for a simple model considering only three explanatory variables isn't so bad.

Conclusion

Congratulations! Rather than spending a couple of hours just searching for and joining datasets from the CDC, Bureau of Labor Statistics, and US Census Bureau, you've spent one lab session querying dataCommons for the same data and, on top of that, using it to build a linear model. Hopefully dataCommons has made the time-to-analysis much shorter!

You've also created a basic model to predict the prevalence of obesity in approximately 500 states using intuition from a long line of research. The model you've created only uses three explanatory variables, so even though it's not the most accurate at predicting obesity prevalence, it's possible that it can be improved by adding more variables. Obesity is also known to correlate with factors such as [high cholesterol](#) and [diabetes](#), but with the data in dataCommons, we can even consider asking if obesity correlates with stranger factors such as

- How many universities recognized by [collegescorecard](#) dataset are contained in a given city
- The incidence rate of property arson
- The average snowfall in inches

Does adding these variables into your model improve accuracy? Can you think of other variables that correlate with obesity?

Finally, as this project is still in the early stages of release, we're looking for all the feedback we can get. If you're free, we'd very much appreciate it if you fill out this [feedback form](#). We hope that your time has been well spent using this new tool and the dataCommons team wishes to thank you for participating in this lab!

