[Website (https://www.coleridgeinitiative.org)](https://www.coleridgeinitiative.org)

Rayid Ghani, Frauke Kreuter, Julia Lane, Adrianne Bradford, Alex Engler, Nicolas Guetta Jeanrenaud, Graham Henke, Daniela Hochfellner, Clayton Hunter, Brian Kim, Avishek Kumar, Jonathan Morgan, Ekaterina Levitskaya, Benjamin Feder.

to be updated on export

Inference

In this notebook, we go over these main concepts:

- The use of survey weights (or projection factors) to get accurate statistics from the sample.
- Imputation to account for missing values.

Python Setup

As always, we start by importing any packages we need, as well as creating our connection to the database.

In [ ]:
```
# data manipulation
import pandas as pd

# database connection
from pyathenajdbc import connect

# statistics package for calculating survey weights
from statsmodels.stats.weightstats import DescrStatsW
import statsmodels.api as sm

# plotting
import matplotlib.pyplot as plt
%matplotlib inline

# show full contents of a cell in pandas
pd.set_option('display.max_colwidth',-1)
```

In [ ]:
```
#database connection
conn = connect(s3_staging_dir = 's3://usda-iri-2019-queryresults/',
               region_name = 'us-gov-west-1',
               LogLevel = '0',
               workgroup = 'workgroup-iri_usda')
```

Survey Weights (Projection Factors)

The IRI data include survey weights (or projection factors; in this class we use a variable called `projection61k`), which allow us to produce estimates for the total U.S. population based on survey data from a random sample of the population. In general, survey weights are used because the sample isn't necessarily taken evenly from the population. Sometimes, researchers decide to intentionally oversample from certain subpopulations in order to make sure they have enough people from that group. Re-weighting is also done after the fact to adjust for non-response or other factors that may reduce our sample (in our case, adjustments were made to account for not all households having reliable purchasing data).

The projection factors indicate how many households are represented by the household in the sample. In order to obtain estimates that represent all the households in the United States, we need to multiply the household purchase quantities and expenditures by the projection factors.

> Note: In terms of household purchase quantities, random-weight products (perishable products that are typically sold in bulk or by the unit) lack the quantity information - therefore, for the random-weight portion, it is only possible to project expenditures and not the purchase quantities.

Certain types of households (including younger households, lower income households, and households with children) are less likely to report purchases consistently enough to be included in the static panel. Although weights are provided to weight the data to ensure that the distribution of households demographics reflects the make-up of the U.S. population, it is likely that households that report regularly enough to be included in the static panel have different attitudes toward diet and health than does the general population (see Muth et al., 2013, [2]). These differences in attitudes could influence purchase behaviors.

> Note: In addition, in some cases, the data are being projected from a relatively small pool of reporters (particularly for households with heads under age 35). Weights are also not provided with the *InfoScan store data*; therefore, it is not possible to calculate nationally representative estimates, and the analyses of the data is only representative of the subset of stores reflected in the data.

In this section we will compute statistics with and without sampling weights to show how the results differ and demonstrate why using weights is so important.

Part 1: What is the proportion of WIC-households with enough purchasing data vs. total households in 2017?

As a reminder, the survey weights are stored in the `projection61k` variable. First, let's look at the proportion of WIC-households with enough purchasing history without applying weights.

In [ ]:
```
# Select distinct households, WIC flag variable (wic_june), and projecti
on61k above zero in 2017

query = '''
SELECT DISTINCT(panid), projection61k, wic_june
FROM iri_usda.demo_all
WHERE projection61k > 0 AND year = '2017';
'''
hh = pd.read_sql(query, conn)
```

In [ ]:
```
# Create a new variable for whether the household is WIC or not
hh['is_wic'] = hh.wic_june == 1
```

In [ ]:
```
# View first rows of a dataframe
hh.head()
```

In [ ]:
```
# Find total number of households records in 2017 by calling a 'len' fun
ction
len(hh)
```

In [ ]:
```
# Find how many records with WIC flag = True in 2017 by subsetting a dat
aframe and calling a `len` function
len(hh[hh['is_wic'] == True])
```

In [ ]:
```
# Find the proportion without the weights
len(hh[hh['is_wic'] == True]) / len(hh)
```

Now let's look at the results after applying weights.

We're using the `DescrStatsW` function from the `statsmodels.stats.weightstats` module. To use it, we simply give it the variable that we want to calculate statistics for (in this case, `hh.is_wic` , which represents whether a household was WIC or not) and provide the weights to be used in the calculation (here, it is in the `projection61k` variable).

> Note: We can also calculate these proportions by hand like we did in the first Data Exploration notebook, but here we will use the provided function.

```
In [ ]: # Create weighted stats using DescrStatsW function
        weighted_stats = DescrStatsW(hh.is_wic, weights = hh.projection61k)
```

```
In [ ]: # Look at the mean (this is the same as proportion since it's 1 if WIC a
        nd 0 if not)
        weighted_stats.mean
```

After applying weights, you can see how the proportion estimate changed.

Checkpoint 1: WIC-eligible households in 2017 with and without weights

Remember that in the Data Exploration notebook we defined a WIC-eligible household as a household which is currently not a member of a WIC program but is eligible based on income.

Find the count and proportion of 2017 households with enough purchasing history that are WIC-eligible with and without weights.

The code below shows how to find the unweighted proportion.

```
In [ ]: # Use the code from the Data Exploration notebook to get a subset of WIC
        -eligible households in 2017
        # with associated weights
        query = """
        SELECT panid, projection61k
        FROM iri_usda.demo_all
        WHERE ac in (1,4,5,7) and
            (
            (hhinc <= 5 and hhsize = 1) or (hhinc = 6 and hhsize < 3) or (hhinc
         = 7 and hhsize < 4) or
            (hhinc = 8 and hhsize < 5) or (hhinc = 9 and hhsize < 6) or (hhinc =
        10 and hhsize < 8) or
            (hhinc = 11 and hhsize = 8)
            )
            and wic_june != 1 and year = '2017' and projection61k > 0;
        """

        wic_eligible_hh = pd.read_sql(query, conn)
```

```
In [ ]: # Total number of WIC-eligible households with projection61k above zero
         in 2017
        len(wic_eligible_hh)
```

The proportion of WIC-eligible households without weights can be found using the following:

```
In [ ]: len(wic_eligible_hh) / len(hh)
```

What is the proportion of WIC-eligible households after applying weights?

```
In [ ]: # Create an `is_eligible` flag in the original dataframe
        hh['is_eligible'] = hh.panid.isin(wic_eligible_hh.panid) == 1
```

```
In [ ]: # Fill in the ... with the proper variables
        weighted_wic_eligible = DescrStatsW(..., weights = ...)
```

```
In [ ]: weighted_wic_eligible.mean
```

Part 2: Applying weights to WIC-household expenditures in 2017

Let's look at differences in total purchases of WIC-households in 2017 before and after applying weights.

```
In [ ]: # Let's pull a table with total purchases per WIC-household in 2017

        query = '''
        SELECT *
        FROM iri_usda_2019_db.panid_expense
        '''

        wic_expenditures = pd.read_sql(query, conn)

        wic_expenditures.head() # View the dataframe
```

Let's look at a distribution of a non-weighted variable by using function `.quantile()` and looking at the 10th, 25th, 50th, 75th, and 90th percentiles.

```
In [ ]: wic_expenditures.quantile([.1,.25,.5,.75,.9])
```

Now let's calculate the same distribution, but this time with weights, using the same `DescrStatsW` function as in Part 1.

```
In [ ]: wic_expenditures_w = DescrStatsW(wic_expenditures.total_wic_purchase, we
        ights = wic_expenditures.projection61k, ddof = 0)
```

```
In [ ]: wic_expenditures_w.quantile([.1,.25,.5,.75,.9]).reset_index().rename(col
        umns={'p':'percentile',0:'total_wic_purchase'})
```

To summarize, applying weights allow us to get more accurate estimates in terms of representation of a general population. In this example, the variance in the values is not very high, so the difference between weighted and non-weighted results is not as big, as in the previous example with the proportion.

However, it is important to keep in mind that when using survey data (due to the specificities with which every particular survey is designed), the weights always need to be applied in order to be able to draw accurate conclusions about the general population.

Missing Data

Sometimes, we have variables with missing (or unknown) data. Instead of dropping those values, there are methods to fill those in, in order to be able to use the data. In this example, we will look at a `mop` in the `trip_all` table, where we have a lot of entries with 0/Unknown values.

Keep in mind that these imputed values will be **approximations**, and must be treated as such. If you choose to impute missing values in your project or future work, you must acknowledge your process and clearly state which variables you imputed values for.

As you remember, in the Data Exploration notebook, we looked at the proportion of payments using WIC benefits. The `mop` variable is only available for year 2017 and contains the following categories:

0 - Unknown
1 - Cash
2 - Check
3 - Credit Card
4 - Debit/ATM
5 - Food Stamps
6 - Gift Card
7 - WIC
8 - Other
9 - Multi

In [ ]:
```
#breakdown of payment counts
qry = '''
select mop, count(*) as count
from iri_usda_2019_db.wic_mop
group by mop
order by count(*) desc
'''
pd.read_sql(qry, conn)
```

We will try to impute the unknown data using the following methods:

- carry on the previous or next entry's value (`forward fill` and `backward fill`);
- find a mode by a household, and impute the mode;
- advanced (optional): use machine learning algorithm (`K-nearest Neighbors`).

Method 1. Forward fill or backward fill the missing values

If we have missing values for a method of payment, but we know the previous or the next entry, we can fill the missing values by using a `forward fill` function (if the previous entry is known) or a `backward fill` function (if the next entry is known).

An Example of How Forward and Backward Fill Work

Method of payment patterns (per household):

Pattern 1
`[None, None, 2, 5, 2, 2, 2]`
The next entries are known -> use `backward fill`
`[2, 2, 2, 5, 2, 2, 2]`

Pattern 2
`[1, 2, 1, 1, 1, None, None]`
The previous entries are known -> use `forward fill`
`[1, 2, 1, 1, 1, 1, 1]`

We created a table `wic_mop` in the `iri_usda_2019_db` database that contains the method of payment in 2017 for every WIC household purchase.

```
In [ ]: # Let's get the methods of payment of WIC-household with enough purchasi
        ng history in 2017
        query = '''
        SELECT *
        FROM iri_usda_2019_db.wic_mop
        '''
        wic_trip = pd.read_sql(query, conn)
```

First, we set the values of `mop` that are 0 to `None` so that it is properly treated as missing.

```
In [ ]: wic_trip.loc[wic_trip['mop'] == '0','mop'] = None
```

Let's check the patterns of methods of payment per household.

```
In [ ]: wic_trip.groupby('panid')['mop'].apply(list).head(5).reset_index()
```

In some cases, we might have households where *none of the methods of payment are known*. Let's filter out those households, as we will not be able to carry a value with either forward fill or backward fill for those households.

> Note: In practice, for such cases we could go a step further and impute for households without any method of payment, e.g., by using a similar demographic group and their methods of payment. However, for now, we will simply show the `forward fill` and `backward fill` functions. We will briefly discuss a method for imputing completely missing values per household in the `K-nearest Neighbor` section towards the end of the notebook.

```
In [ ]: # Return False or True if all the methods of payment are null
        mop_groupby = wic_trip.set_index('panid').isnull().groupby('panid').all
        ().stack().reset_index()
```

How many households are there with no known methods of payment?

```
In [ ]: len(mop_groupby[mop_groupby[0] == True])
```

For now, we will filter these out to show a simple example with `forward fill` and `backward fill` functions.

```
In [ ]: # Save the IDs of those households to a variable
        mop_groupby_true = mop_groupby[mop_groupby[0] == True]

        # Filter out those IDs in the original dataset
        mop_payments = wic_trip[~wic_trip['panid'].isin(mop_groupby_true['panid'
        ])]
```

Now we have households where at least some method of payment information is known.

We will use a combination of `forward fill` and `back fill` functions, to fill in for missing values per household.

```
In [ ]: # Forward fill missing values
        mop_payments.loc[:,'mop'] = mop_payments.groupby('panid')['mop'].ffill()
```

```
In [ ]: # Backward fill missing values
        mop_payments.loc[:,'mop'] = mop_payments.groupby('panid')['mop'].bfill()
```

Now let's check the patterns after filling the missing values.

```
In [ ]: mop_payments.groupby('panid')['mop'].apply(list).reset_index().head()
```

All the `None` values have been filled. As mentioned above, for those households which didn't have any method of payment information available, we could still impute by using the method of payment information for the households from a similar demographic group. We could impute using their method of payment patterns or using their most frequent method of payment, as described below.

Impute using mode by household

Another method of filling in the missing values is to find the most frequent method of payment per household (mode), and impute using that mode.

We will use the same dataframe (without the households that do not have any unknown method of payment).

```
In [ ]: mop_payments = wic_trip[~wic_trip['panid'].isin(mop_groupby_true['panid'
        ])]
```

We use the `groupby` function to find the frequency of payment per household.

```
In [ ]: # By default, the groupby function will not include None as a method of
         payment value,
        # will only count the existing values
        mop_count = wic_trip.groupby(['panid','mop'])['mop'].count().reset_index
        (name='count')
```

```
In [ ]: # Get only the most frequent payment method per household
        mop_mode = mop_count.sort_values('count', ascending=False).drop_duplicat
        es(['panid'])

        mop_mode.head()
```

```
In [ ]: # We can now drop the `count` column
        mop_mode = mop_mode.drop('count',axis=1)
```

`Mop_mode` is now our lookup table with the most frequent payment method per household.

We will merge the original table (`mop_payments`) with the look up table. This will create two columns: `mop_x` (original method of payment) and `mop_y` (method of payment from the lookup table).

```
In [ ]: merged = mop_payments.merge(mop_mode, on='panid')
```

Let's take a look at one household as an example. Some IDs have specific values as unique methods of payments, and we will now replace the `None` values in the `mop_x` with the known method of payment from the `mop_y` column.

```
In [ ]: merged[merged['panid'] == '904371770']
```

```
In [ ]: # Replace the None values in the `mop_x` column with the known values in
        the column `mop_y`
        merged.loc[merged['mop_x'].isnull(), 'mop_x'] = merged['mop_y']
```

```
In [ ]: # Now we can drop the `mop_y` column
        merged = merged.drop('mop_y',axis=1)
```

Let's check at the results for the same household. The None values are replaced with the most frequent method of payment for that household.

```
In [ ]: merged[merged['panid'] == '904371770']
```

(Optional) Advanced: Using machine learning to impute values

To impute values, we can also use the machine learning algorithm called the `K-nearest Neighbors`. The principle behind it is quite simple: the missing values can be imputed by values of "closest neighbors" - as approximated by other, known, features.

For example, remember those cases where the methods of payment are completely missing for a given household? For such households, we can approximate their methods of payment by using the methods of payment that belong to other households but in the same demographic group (their 'closest neighbors' in terms of demographic characteristics: the same household size, the same income bracket, the same employment patterns, etc.).

The algorithm calculates the distance between the input values (the missing values) and helps to identify the nearest possible value based on other features (such as known demographic characteristics).

In our case `K-nearest Neighbors` method can be especially useful in cases where no methods of payment are known for a given household (the `forward fill/backward fill` functions or finding the mode will not work).

We will discuss the process for applying methods such as the K-nearest Neighbor classifier in more detail as part of the Machine Learning lecture.

References

1. Muth, M.K., M. Sweitzer, D. Brown, K. Capogrossi, S. Karns, D. Levin, A. Okrent, P. Siegel, and C. Zhen. 2016. "Understanding IRI Household-Based and Store-Based Scanner Data". Technical Bulletin 1942, Economic Research Service.
2. Muth, M.K., S.C. Cates, S.A. Karns, P.Siegel, K.C. Wohlgenant, and C. Zhen. 2013. "Comparing Attitudinal Survey Responses From Proprietary and Government Surveys." Prepared for U.S. Department of Agriculture, Economic Research Service.