FE 550 - Team Project Final Submission

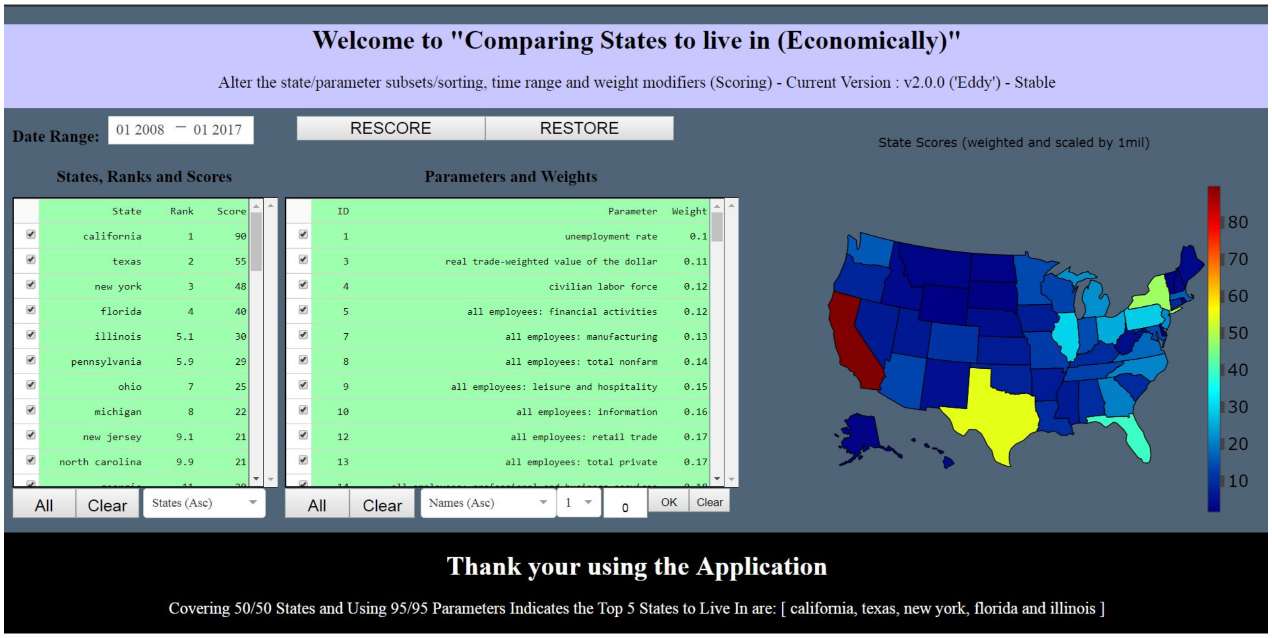
Wyatt Marciniak, Yoseph Borai, Xiaochi Ma, Lucas Eisenberg

May 12, 2019

Contents

Overview and Motivation

This project was designed to answer the question of “Which State is the Best to Work and Live In”. That is a tough question to answer, for anyone, but with the data resources available today, we can create a tool to help out just a bit. We introduce “Comparing States to Live In (Economically)”



Welcome to "Comparing States to live in (Economically)"

Alter the state/parameter subsets/sorting, time range and weight modifiers (Scoring) - Current Version : v2.0.0 ('Eddy') - Stable

Date Range: 01 2008 – 01 2017

RESCORE | RESTORE

States, Ranks and Scores

	State	Rank	Score
✓	california	1	90
✓	texas	2	55
✓	new york	3	48
✓	florida	4	40
✓	illinois	5.1	30
✓	pennsylvania	5.9	29
✓	ohio	7	25
✓	michigan	8	22
✓	new jersey	9.1	21
✓	north carolina	9.9	21

All | Clear | States (Asc)

Parameters and Weights

	ID	Parameter	Weight
✓	1	unemployment rate	0.1
✓	3	real trade-weighted value of the dollar	0.11
✓	4	civilian labor force	0.12
✓	5	all employees: financial activities	0.12
✓	7	all employees: manufacturing	0.13
✓	8	all employees: total nonfarm	0.14
✓	9	all employees: leisure and hospitality	0.15
✓	10	all employees: information	0.16
✓	12	all employees: retail trade	0.17
✓	13	all employees: total private	0.17

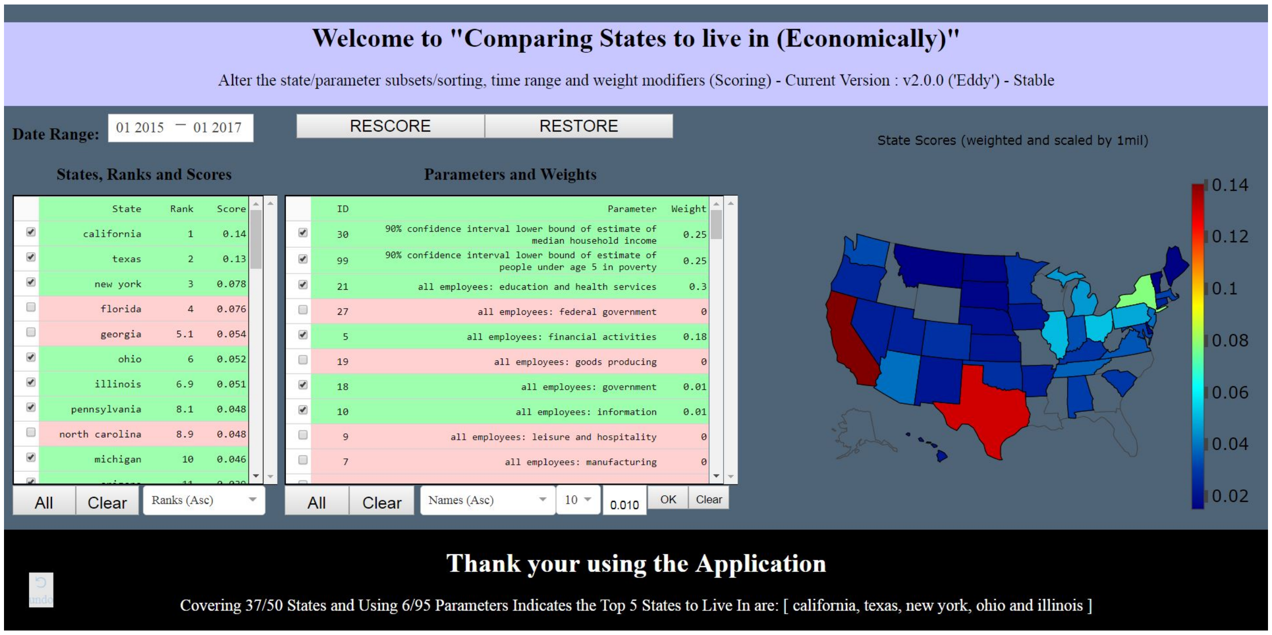
All | Clear | Names (Asc) | 1 | 0 | OK | Clear

State Scores (weighted and scaled by 1mil)

Thank your using the Application

Covering 50/50 States and Using 95/95 Parameters Indicates the Top 5 States to Live In are: [california, texas, new york, florida and illinois]

This application holds 95 data sets that span over 9 years (2008 - 2017) and all 50 states evenly (This is approx. 717,000 data points). It allows for custom viewing controls, sorting, weighting and a map-based visualization to summarize your results. It is stable and runnable *(see ‘Issues, concerns and notices’ at report end). As an example of the power this application has, we offer an example run showcasing all components edited by the user, as well as rescored (by the user - button click event - discussed below). See below figure (and note the footer (bottom black bar) always tells you the current summary result of the top 5 states by score that you have NOT deselected, otherwise it skips the states until 5 are found in the proper order:



Welcome to "Comparing States to live in (Economically)"

Alter the state/parameter subsets/sorting, time range and weight modifiers (Scoring) - Current Version : v2.0.0 ('Eddy') - Stable

Date Range: 01 2015 – 01 2017

RESCORE | RESTORE

States, Ranks and Scores

	State	Rank	Score
✓	california	1	0.14
✓	texas	2	0.13
✓	new york	3	0.078
☐	florida	4	0.076
☐	georgia	5.1	0.054
✓	ohio	6	0.052
✓	illinois	6.9	0.051
✓	pennsylvania	8.1	0.048
☐	north carolina	8.9	0.048
✓	michigan	10	0.046

All | Clear | Ranks (Asc)

Parameters and Weights

	ID	Parameter	Weight
✓	30	90% confidence interval lower bound of estimate of median household income	0.25
✓	99	90% confidence interval lower bound of estimate of people under age 5 in poverty	0.25
✓	21	all employees: education and health services	0.3
☐	27	all employees: federal government	0
✓	5	all employees: financial activities	0.18
☐	19	all employees: goods producing	0
✓	18	all employees: government	0.01
✓	10	all employees: information	0.01
☐	9	all employees: leisure and hospitality	0
☐	7	all employees: manufacturing	0

All | Clear | Names (Asc) | 10 | 0.010 | OK | Clear

State Scores (weighted and scaled by 1mil)

Thank your using the Application

Covering 37/50 States and Using 6/95 Parameters Indicates the Top 5 States to Live In are: [california, texas, new york, ohio and illinois]

Resources

For this application, we used *Python* 3.7.0 / 3.7.2 and utilized the *PyCharm* IDE for script writing, application design and project file managment. We used an array of standard packages as well as more advanced utilities for higher level operations (such as multi-processing for data retrival and manipulation in real-time):

Table 1: Key Python Modules by Operation (Documentation is Linked)

Operation	Dependencies
Core Utilities	*Numpy* | *Scipy* | *Certifi* | *Math*
Web Scraping Data Sets	*Beautiful Soup* | *Requests* | *JSON (json)*
Optimizing Data Reading/Operations	*Multiprocessing*
Data Handling (fetch/clean/store/call)	*Pandas* | *Openpyxl* | *Datetime*
Application Design	*Plotly* | *Dash (Plotly)*

All packages listed above are open-source languages contained within the Python virtual environment and requirements.txt file found in the root of this project's source code.

Data Sets

I. Initial Attempts

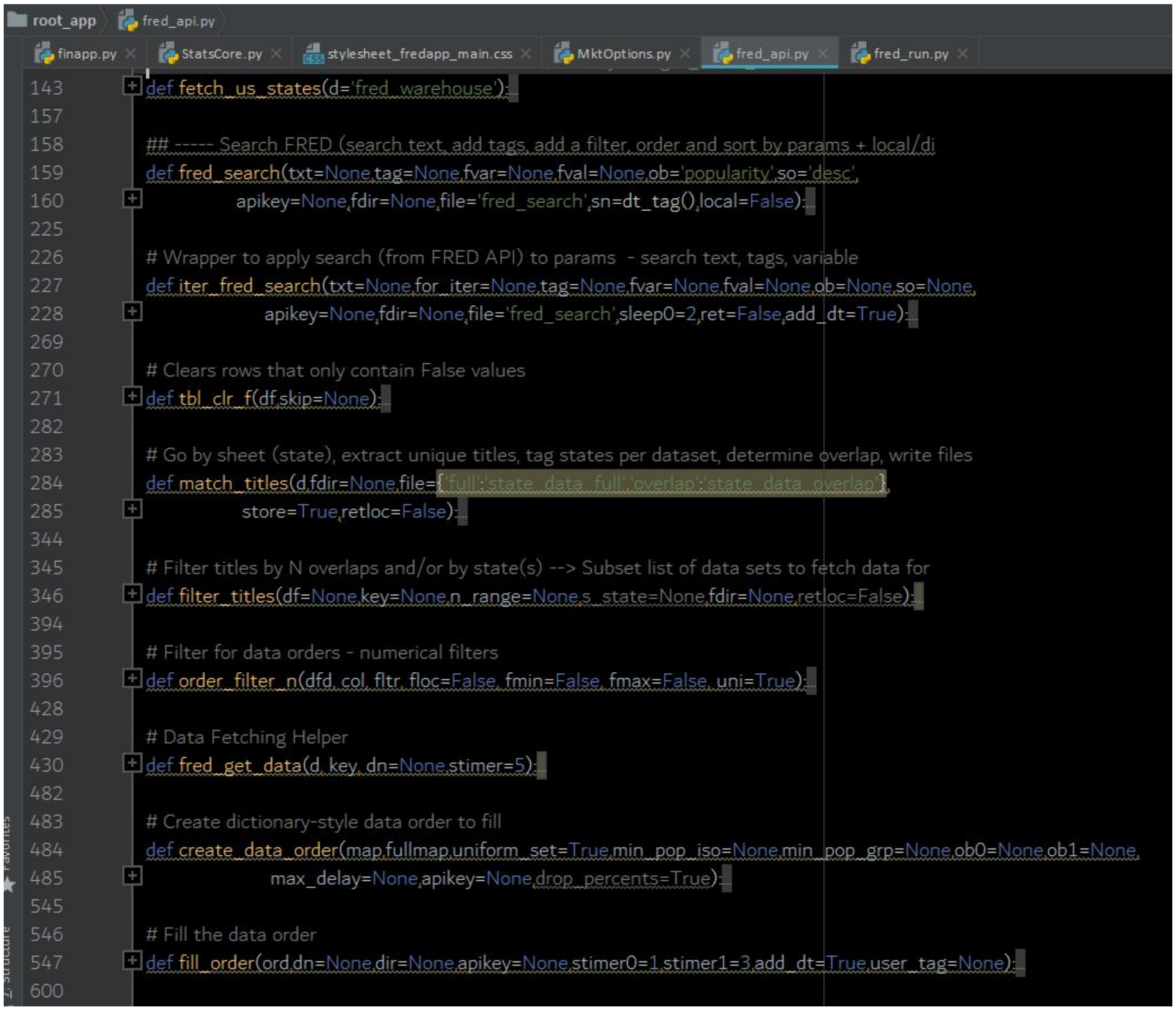
To create this application, we needed to find data sets that covered the 50 US States over a reasonable period of time. More importantly, those data sets needed to be diverse enough to cover a range of user preferences from specfic industries to broader statistics and indicators. We faced difficulty in finding non-economic and/or non-financial data sets due to a lack of both recorded data as well as publicly facing content to acquire. Some candidate sites (that were not used) were the *IRS* website, the *BEA* (US Bureau of Economic Analysis) and the national *Census* records. These data sets were either incomplete, too poorly maintained for implementation at this time or, most commonly, the data was static for single/current year metrics and the one that were of value, overlapped with our main data source.

II. FRED database API

We sourced all of our data from the *FRED* (Federal Reserve Economic Data) database. There are over 500,000 US and foriegn data series available, but more importantly for pur project, they included state level data, across all 50 states and over a large period of time.
The API written for the project mostly contans wrapper functions (functions that call other functions with modifications or organize an algoorithm using other resources) that extend the core *API* avaialble for free public use. The site allows for user downloads from the UI as well as web-scraping protocols, but the dedicated API service provides a stable, and officially established, pipeline of data in a fixed format. This is a key element to have to ensure data security, accuracy and reliability (in terms of fetching) if you can find and use one. We found monthly data to be the best frequency for realistic testing.

III. API Source Code Architecture [fred_api.py (Also: app_utilities.py)]

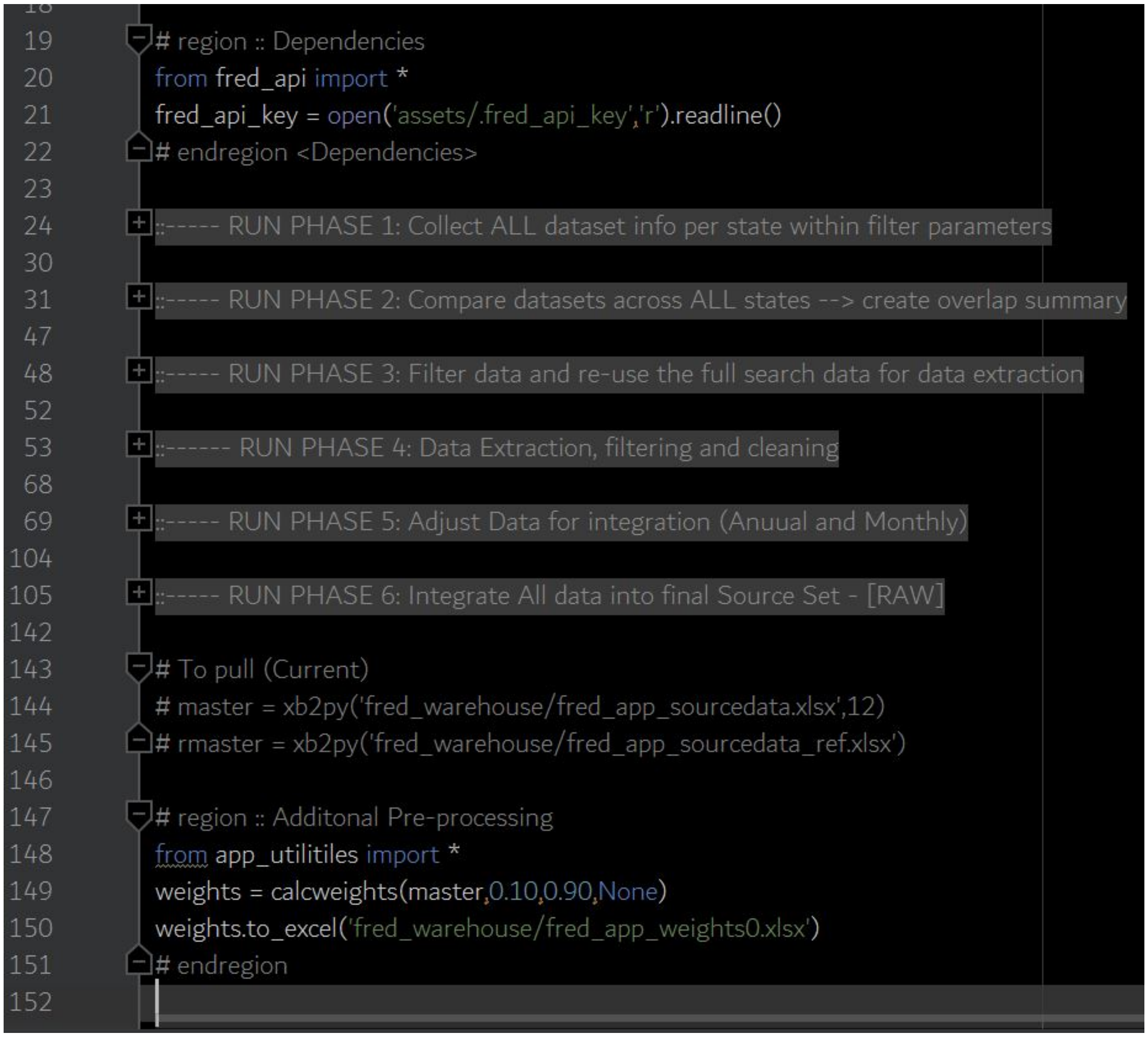
The Architecture was designed specifically for this application but it has the components needed to integrate into a generalized framework (future work). The core of the library is seen in the below architecture. The functionalities utilize the free API (key needed - but free) from FRED and wraps useful utilities to gather state data specifically (also accounting for overlapping date ranges, missing obervations and data that is delayed by too great a period)



```
def fetch_us_states(d='fred_warehouse'):...

## ----- Search FRED (search text, add tags, add a filter, order and sort by params + local/di
def fred_search(txt=None,tag=None,fvar=None,fval=None,ob='popularity',so='desc',
             apikey=None,fdir=None,file='fred_search',sn=dt_tag(),local=False):...

# Wrapper to apply search (from FRED API) to params  - search text, tags, variable
def iter_fred_search(txt=None,for_iter=None,tag=None,fvar=None,fval=None,ob=None,so=None,
                apikey=None,fdir=None,file='fred_search',sleep0=2,ret=False,add_dt=True):...

# Clears rows that only contain False values
def tbl_clr_f(df,skip=None):...

# Go by sheet (state), extract unique titles, tag states per dataset, determine overlap, write files
def match_titles(d,fdir=None,file={'full':'state_data_full','overlap':'state_data_overlap'},
              store=True,retloc=False):...

# Filter titles by N overlaps and/or by state(s) --> Subset list of data sets to fetch data for
def filter_titles(df=None,key=None,n_range=None,s_state=None,fdir=None,retloc=False):...

# Filter for data orders - numerical filters
def order_filter_n(dfd, col, fltr, floc=False, fmin=False, fmax=False, uni=True):...

# Data Fetching Helper
def fred_get_data(d, key, dn=None,stimer=5):...

# Create dictionary-style data order to fill
def create_data_order(map,fullmap,uniform_set=True,min_pop_iso=None,min_pop_grp=None,ob0=None,ob1=None,
                 max_delay=None,apikey=None,drop_percents=True):...

# Fill the data order
def fill_order(ord,dn=None,dir=None,apikey=None,stimer0=1,stimer1=3,add_dt=True,user_tag=None):...
```

IV. API Operation Analysis [fredrun.py]

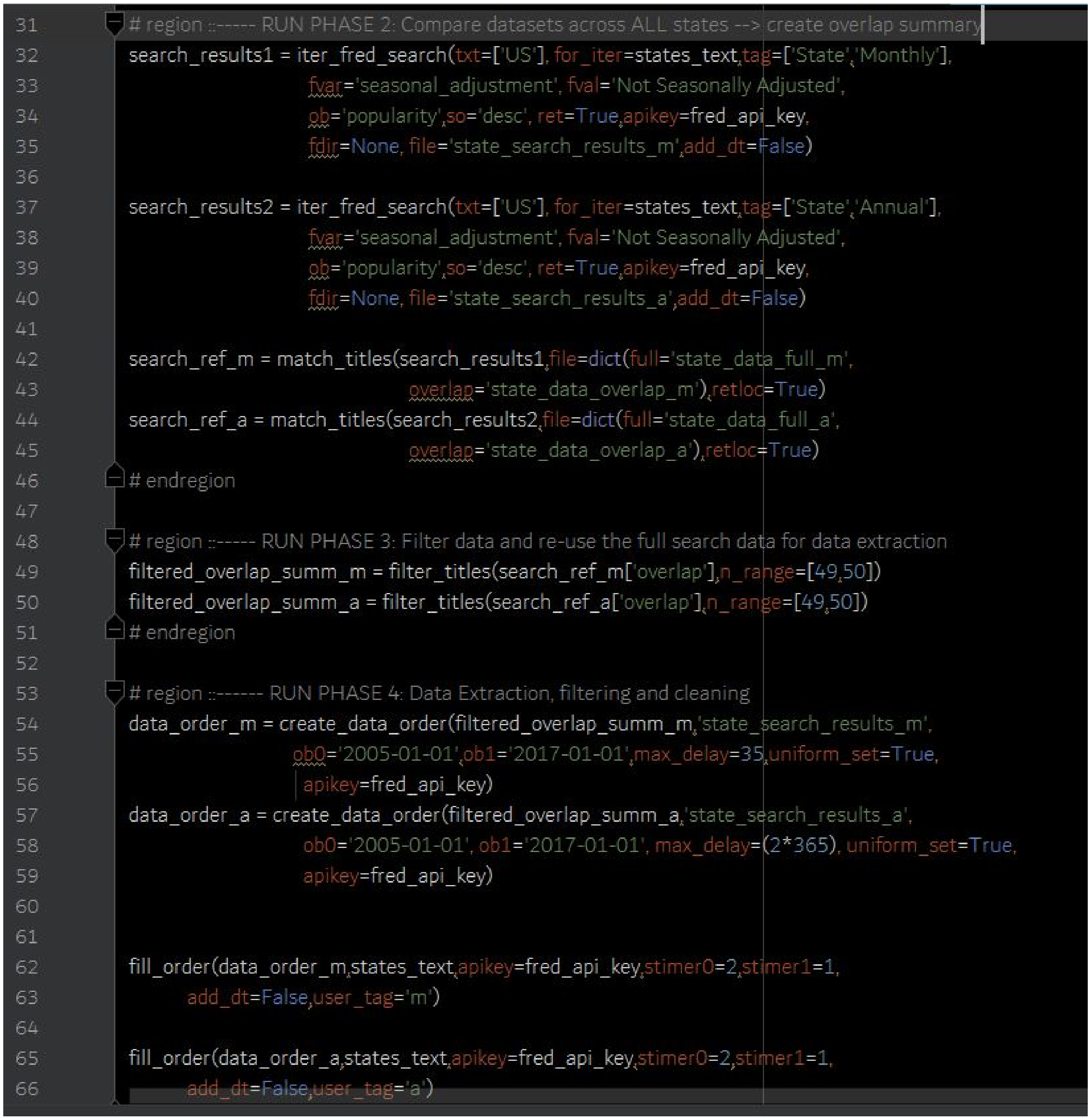
The main runner (script running the function/algorithm/etc...) is written as a step by step algorithm so users can review it piece by piece to follow along. The achitecture shows us the progression from acquiring the state name/abbreviation data to filtering searches in the FRED database to final aggregation and storage. See the outlined runner architecture below:



```
# region :: Dependencies
from fred_api import *
fred_api_key = open('assets/.fred_api_key','r').readline()
# endregion <Dependencies>

::----- RUN PHASE 1: Collect ALL dataset info per state within filter parameters

::----- RUN PHASE 2: Compare datasets across ALL states --> create overlap summary

::----- RUN PHASE 3: Filter data and re-use the full search data for data extraction

::------ RUN PHASE 4: Data Extraction, filtering and cleaning

::----- RUN PHASE 5: Adjust Data for integration (Anuual and Monthly)

::----- RUN PHASE 6: Integrate All data into final Source Set - [RAW]

# To pull (Current)
# master = xb2py('fred_warehouse/fred_app_sourcedata.xlsx',12)
# rmaster = xb2py('fred_warehouse/fred_app_sourcedata_ref.xlsx')

# region :: Additonal Pre-processing
from app_utilitiles import *
weights = calcweights(master,0.10,0.90,None)
weights.to_excel('fred_warehouse/fred_app_weights0.xlsx')
# endregion

```

V. Core Fetch, Process and Clean Operation [fredrun.py]

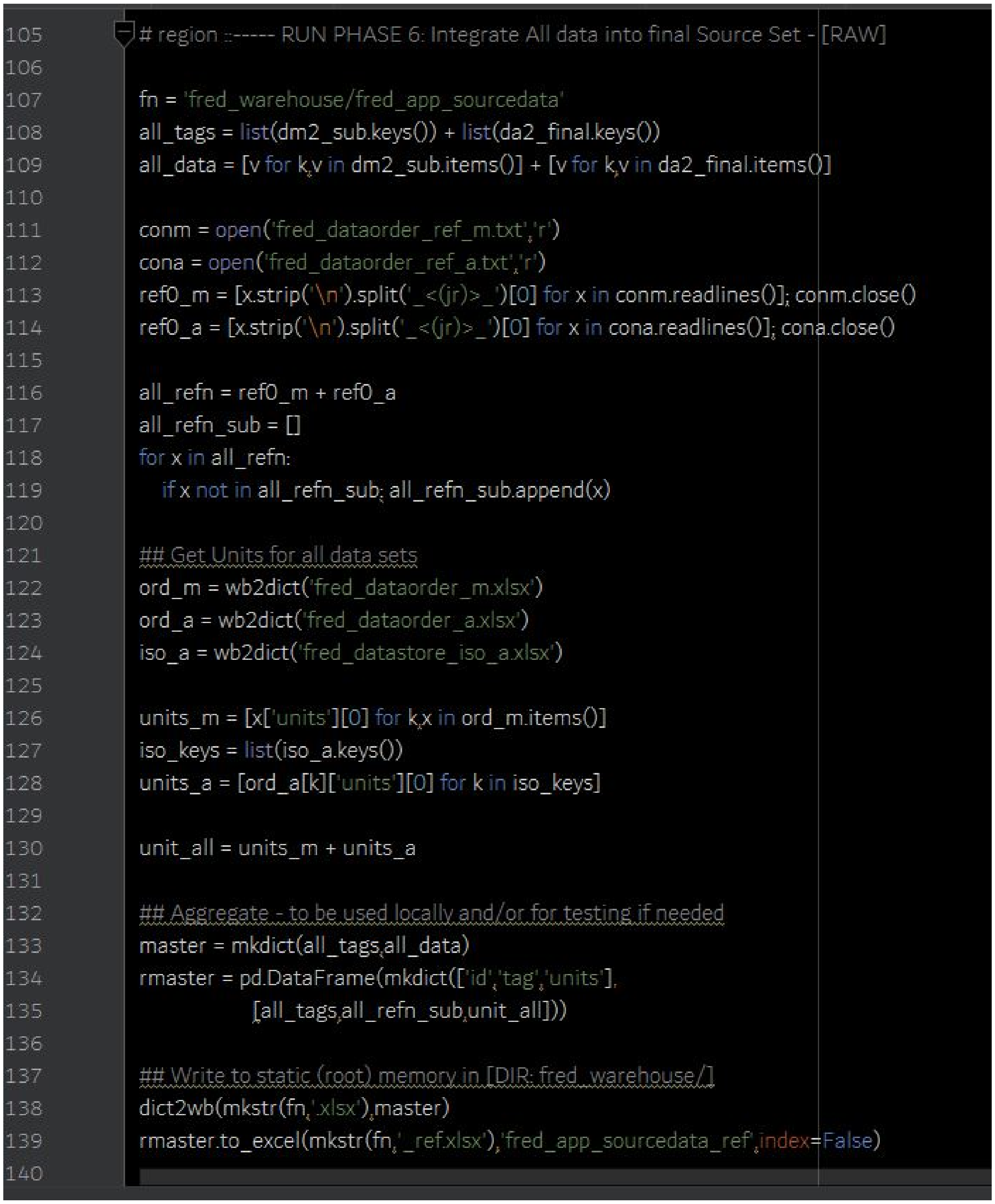
The operation below is the crucial operational piece of the API algorithm operation. We can see in Phase 2 the algorithm collects search results from the FRED database directly. In Phase 3, the data sets (some states had over 1200 to themselves alone) are combined into a unique list (single data set names that are not duplicated), generalized for all state names and then compared. A summary is returned (and stored, as all the data is) which shows all unique data sets and a large True/False table of whether or not a state had that data set recorded for themselves. Finally, in phase 4 we collect a uniform set of data for each state per the overlap analysis. In the end, we compiled 95 data sets. We needed to collect annual and monthly data sets and generate evenly spaced steps of data to normalize the annual data with the monthly. By doing this, we were able to expand from about 23 data sets to the current 95. See this key segment screenshoted below:



```
# region ::----- RUN PHASE 2: Compare datasets across ALL states --> create overlap summary
search_results1 = iter_fred_search(txt=['US'], for_iter=states_text,tag=['State','Monthly'],
                        fvar='seasonal_adjustment', fval='Not Seasonally Adjusted',
                        ob='popularity',so='desc', ret=True,apikey=fred_api_key,
                        fdir=None, file='state_search_results_m',add_dt=False)

search_results2 = iter_fred_search(txt=['US'], for_iter=states_text,tag=['State','Annual'],
                        fvar='seasonal_adjustment', fval='Not Seasonally Adjusted',
                        ob='popularity',so='desc', ret=True,apikey=fred_api_key,
                        fdir=None, file='state_search_results_a',add_dt=False)

search_ref_m = match_titles(search_results1,file=dict(full='state_data_full_m',
                                    overlap='state_data_overlap_m'),retloc=True)
search_ref_a = match_titles(search_results2,file=dict(full='state_data_full_a',
                                    overlap='state_data_overlap_a'),retloc=True)
# endregion

# region ::----- RUN PHASE 3: Filter data and re-use the full search data for data extraction
filtered_overlap_summ_m = filter_titles(search_ref_m['overlap'],n_range=[49,50])
filtered_overlap_summ_a = filter_titles(search_ref_a['overlap'],n_range=[49,50])
# endregion

# region ::------ RUN PHASE 4: Data Extraction, filtering and cleaning
data_order_m = create_data_order(filtered_overlap_summ_m,'state_search_results_m',
                       ob0='2005-01-01',ob1='2017-01-01',max_delay=35,uniform_set=True,
                        apikey=fred_api_key)
data_order_a = create_data_order(filtered_overlap_summ_a,'state_search_results_a',
                        ob0='2005-01-01', ob1='2017-01-01', max_delay=(2*365), uniform_set=True,
                        apikey=fred_api_key)


fill_order(data_order_m,states_text,apikey=fred_api_key,stimer0=2,stimer1=1,
        add_dt=False,user_tag='m')

fill_order(data_order_a,states_text,apikey=fred_api_key,stimer0=2,stimer1=1,
        add_dt=False,user_tag='a')
```

VI. Creating the final data resources [fredrun.py]

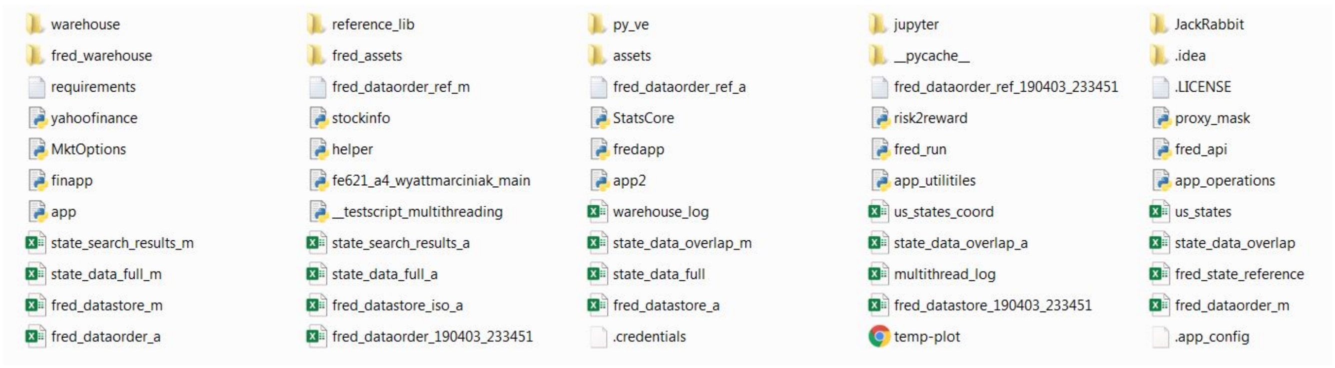
Here, we show the final data sets (that will be referenced for use by the application) being cleaned and created before storing. We use (.xlsx) Excel Workbooks as functional 'databases' where the sheets delimit parameters, usually. This helps manage large memory resources and with clustered data in the root we can easily call to it and populate the application with data.



```
# region ::----- RUN PHASE 6: Integrate All data into final Source Set - [RAW]

fn = 'fred_warehouse/fred_app_sourcedata'
all_tags = list(dm2_sub.keys()) + list(da2_final.keys())
all_data = [v for k,v in dm2_sub.items()] + [v for k,v in da2_final.items()]

conm = open('fred_dataorder_ref_m.txt','r')
cona = open('fred_dataorder_ref_a.txt','r')
ref0_m = [x.strip('\n').split('_<(jr)>_')[0] for x in conm.readlines()]; conm.close()
ref0_a = [x.strip('\n').split('_<(jr)>_')[0] for x in cona.readlines()]; cona.close()

all_refn = ref0_m + ref0_a
all_refn_sub = []
for x in all_refn:
    if x not in all_refn_sub: all_refn_sub.append(x)

## Get Units for all data sets
ord_m = wb2dict('fred_dataorder_m.xlsx')
ord_a = wb2dict('fred_dataorder_a.xlsx')
iso_a = wb2dict('fred_datastore_iso_a.xlsx')

units_m = [x['units'][0] for k,x in ord_m.items()]
iso_keys = list(iso_a.keys())
units_a = [ord_a[k]['units'][0] for k in iso_keys]

unit_all = units_m + units_a

## Aggregate - to be used locally and/or for testing if needed
master = mkdict(all_tags,all_data)
rmaster = pd.DataFrame(mkdict(['id','tag','units'],
              [all_tags,all_refn_sub,unit_all]))

## Write to static (root) memory in [DIR: fred_warehouse/]
dict2wb(mkstr(fn,'.xlsx'),master)
rmaster.to_excel(mkstr(fn,'_ref.xlsx'),'fred_app_sourcedata_ref',index=False)
```

Backend Framework

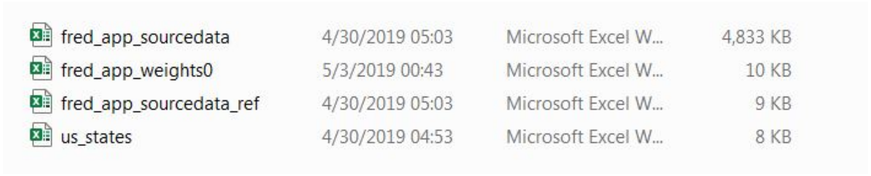
The backend (root) directory is combined with my (Wyatt Marciniak) FE 800 'finapp' root directory. The dependencies are shared and I have been developing functionality for both courses simultanelously using the same parent branch of resources (or at least the pipeline - these will be seperated and only data/files pertaining to this fredapp will be included in the source folder). The root is as follows:



warehouse	reference_lib	py_ve	jupyter	JackRabbit
fred_warehouse	fred_assets	assets	__pycache__	.idea
requirements	fred_dataorder_ref_m	fred_dataorder_ref_a	fred_dataorder_ref_190403_233451	.LICENSE
yahoofinance	stockinfo	StatsCore	risk2reward	proxy_mask
MktOptions	helper	fredapp	fred_run	fred_api
finapp	fe621_a4_wyattmarciniak_main	app2	app_utilitiles	app_operations
app	__testscript_multithreading	warehouse_log	us_states_coord	us_states
state_search_results_m	state_search_results_a	state_data_overlap_m	state_data_overlap_a	state_data_overlap
state_data_full_m	state_data_full_a	state_data_full	multithread_log	fred_state_reference
fred_datastore_m	fred_datastore_iso_a	fred_datastore_a	fred_datastore_190403_233451	fred_dataorder_m
fred_dataorder_a	fred_dataorder_190403_233451	.credentials	temp-plot	.app_config

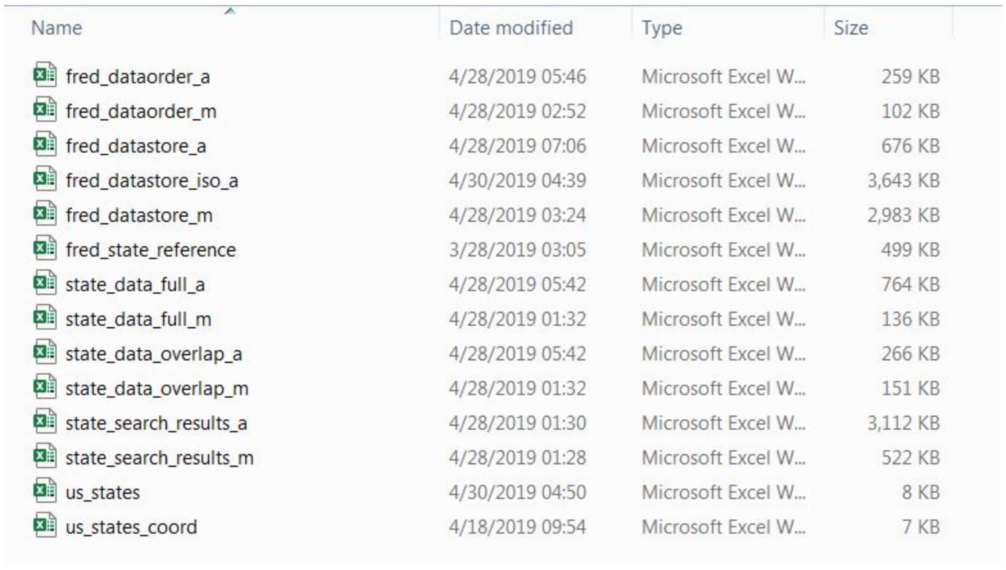
The key files/directories for this application cover any/all (.xlsx) files marked with fred, us, states, etc... as well as the directories:

1. fred_warehouse/ - Holds pre-process data and reference data



fred_app_sourcedata	4/30/2019 05:03	Microsoft Excel W...	4,833 KB
fred_app_weights0	5/3/2019 00:43	Microsoft Excel W...	10 KB
fred_app_sourcedata_ref	4/30/2019 05:03	Microsoft Excel W...	9 KB
us_states	4/30/2019 04:53	Microsoft Excel W...	8 KB

2. fred_assets/ - Holds assets (CSS style sheets for design as well as this report source code/save and the ref_lib/ path to the imges loaded in this report).Shown here is the folder directory fred_assets/backups/ which is where the final, static pre-process data is stored for reference and backup if users damage current data sets being used. The fredapp/API can do this as well but backup copies ensure no issues in operation, especially if deployed and neeeding maintenance. See below:

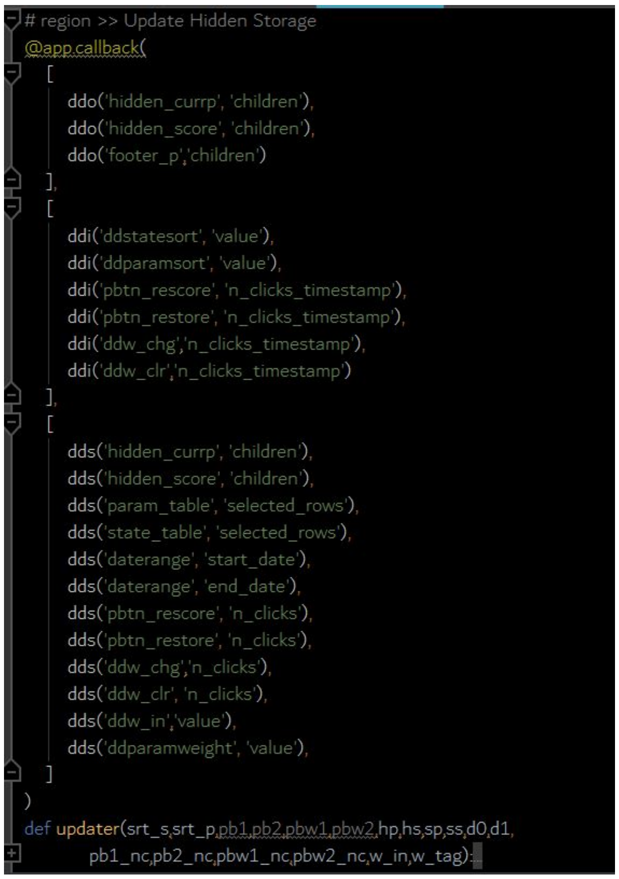


Name	Date modified	Type	Size
fred_dataorder_a	4/28/2019 05:46	Microsoft Excel W...	259 KB
fred_dataorder_m	4/28/2019 02:52	Microsoft Excel W...	102 KB
fred_datastore_a	4/28/2019 07:06	Microsoft Excel W...	676 KB
fred_datastore_iso_a	4/30/2019 04:39	Microsoft Excel W...	3,643 KB
fred_datastore_m	4/28/2019 03:24	Microsoft Excel W...	2,983 KB
fred_state_reference	3/28/2019 03:05	Microsoft Excel W...	499 KB
state_data_full_a	4/28/2019 05:42	Microsoft Excel W...	764 KB
state_data_full_m	4/28/2019 01:32	Microsoft Excel W...	136 KB
state_data_overlap_a	4/28/2019 05:42	Microsoft Excel W...	266 KB
state_data_overlap_m	4/28/2019 01:32	Microsoft Excel W...	151 KB
state_search_results_a	4/28/2019 01:30	Microsoft Excel W...	3,112 KB
state_search_results_m	4/28/2019 01:28	Microsoft Excel W...	522 KB
us_states	4/30/2019 04:50	Microsoft Excel W...	8 KB
us_states_coord	4/18/2019 09:54	Microsoft Excel W...	7 KB

Key Runtime Elements

I. A quick summary of the Dash framework

Dash, from Plotly, is a web-based framework for designing applications that utilize HTML, CSS and Javascript reactivity - for Python. Unlike Flask, which is designed more for actual website architecture, Dash is designed for interactive plots, graphics and applications (including dashboards). Dash integrates with Plotly natively so the power of one of the largest visualization resources across multiple lanquages is the parent to Dash so we have a ton of potential in this area. The framework(s) are 'reactive' which means that they 'listen' for 'events' to occur. To clarify further, reference how we build funtion calls in Dash:



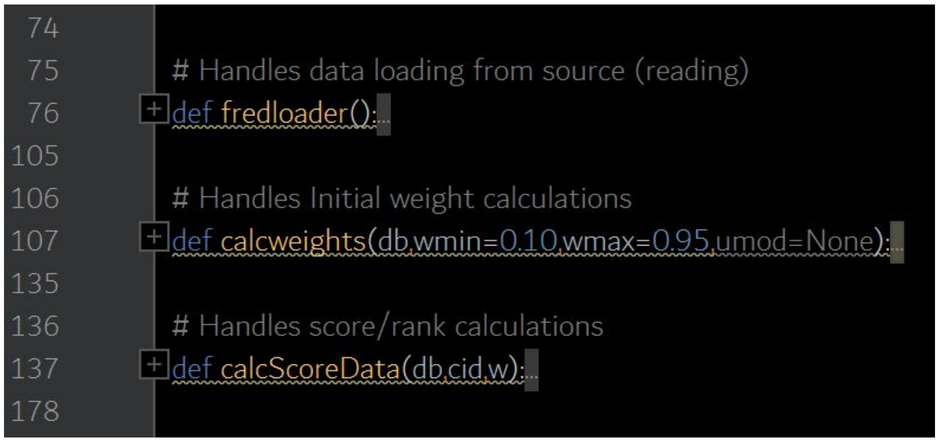
```
# region >> Update Hidden Storage
@app.callback(
    [
        ddo('hidden_currp', 'children'),
        ddo('hidden_score', 'children'),
        ddo('footer_p','children')
    ],
    [
        ddi('ddstatesort', 'value'),
        ddi('ddparamsort', 'value'),
        ddi('pbtn_rescore', 'n_clicks_timestamp'),
        ddi('pbtn_restore', 'n_clicks_timestamp'),
        ddi('ddw_chg','n_clicks_timestamp'),
        ddi('ddw_clr','n_clicks_timestamp')
    ],
    [
        dds('hidden_currp', 'children'),
        dds('hidden_score', 'children'),
        dds('param_table', 'selected_rows'),
        dds('state_table', 'selected_rows'),
        dds('daterange', 'start_date'),
        dds('daterange', 'end_date'),
        dds('pbtn_rescore', 'n_clicks'),
        dds('pbtn_restore', 'n_clicks'),
        dds('ddw_chg','n_clicks'),
        dds('ddw_clr', 'n_clicks'),
        dds('ddw_in','value'),
        dds('ddparamweight', 'value'),
    ]
)
def updater(srt_s,srt_p,pb1,pb2,pbw1,pbw2,hp,hs,sp,ss,d0,d1,
        pb1_nc,pb2_nc,pbw1_nc,pbw2_nc,w_in,w_tag):
```

You are seeing the Output, Input ad State parameter setup of the main updating function which handles all sorting, weighting, etc... and returns repective values to hidden div (HTML) elements. The 'why' in terms of hidden elements is explained belowed but for now, we focus on the chain. This upater returns a tuple of 3 values, to the elements with the IDs found in the 'ddo' (Output) objects. When those output targets are filled, other functions, set to take as Input or 'ddi' the attribute the data delievry affected, will then be activated to run and continue the cycle until all callbacks in the chain are done. Now the app is in 'idle' (standby, waiting for user interaction). Those ddi objects are the event 'triggers' that the functions using them are listening to.

In the above graphic, this function will be called when, for example, 'ddi('pbtn_rescore','n_clicks_timestamp')'is activated. Even if not active, they are used as pipelines but they are used here as input so that whent the button last-clicked data changes, the updater() function is called. For referencing the values of onjects but NOT having them initiate callbacks, we use 'dds' or State objects. In this case, the 'State' of an element or an element's attribute is just its value at that moment. So, if you choose a sort option you would activate the event of one of the 'dd.....sort' parameters, where their curret value is passed and used. Also, '@app.callback' is used as a decorator for runtime parsing locations and all ddo/ddi/dds elements must exist in the layout (not discussed - basic HTML/CSS is more than enough to understand the core methods) or else the appliation will fail to load and, for most users, no error tracebacks will appear. Be sure to watch out for that if you build on the code.

II. Data servicers using Multipricessing for the Application

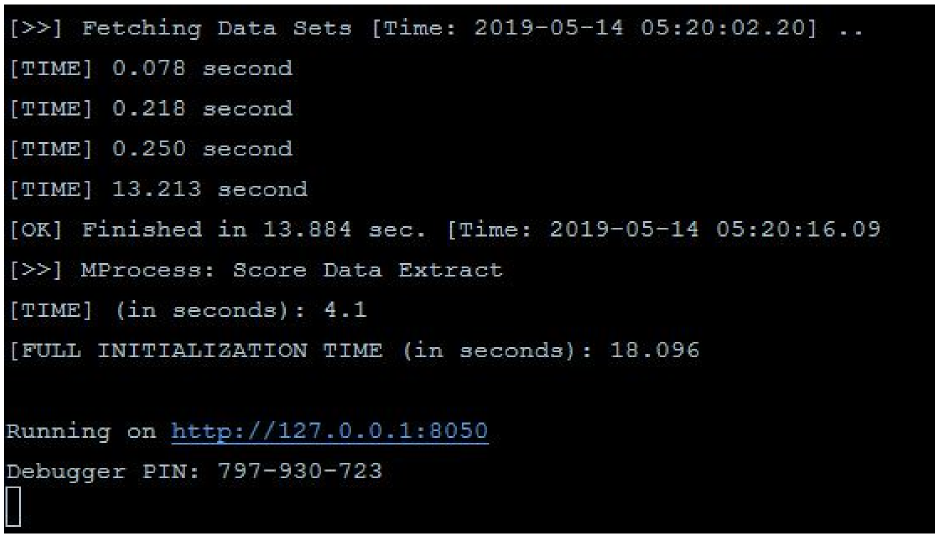
These are simply the functions using multiple thread pools to parallelize the work being done to improve operation lag times closer to non-existant. Please see source code for complete designs.



```
# Handles data loading from source (reading)
def fredloader():...

# Handles Initial weight calculations
def calcweights(db,wmin=0.10,wmax=0.95,umod=None):...

# Handles score/rank calculations
def calcScoreData(db,cid,w):...
```

III. Intialization (Application Start-up)

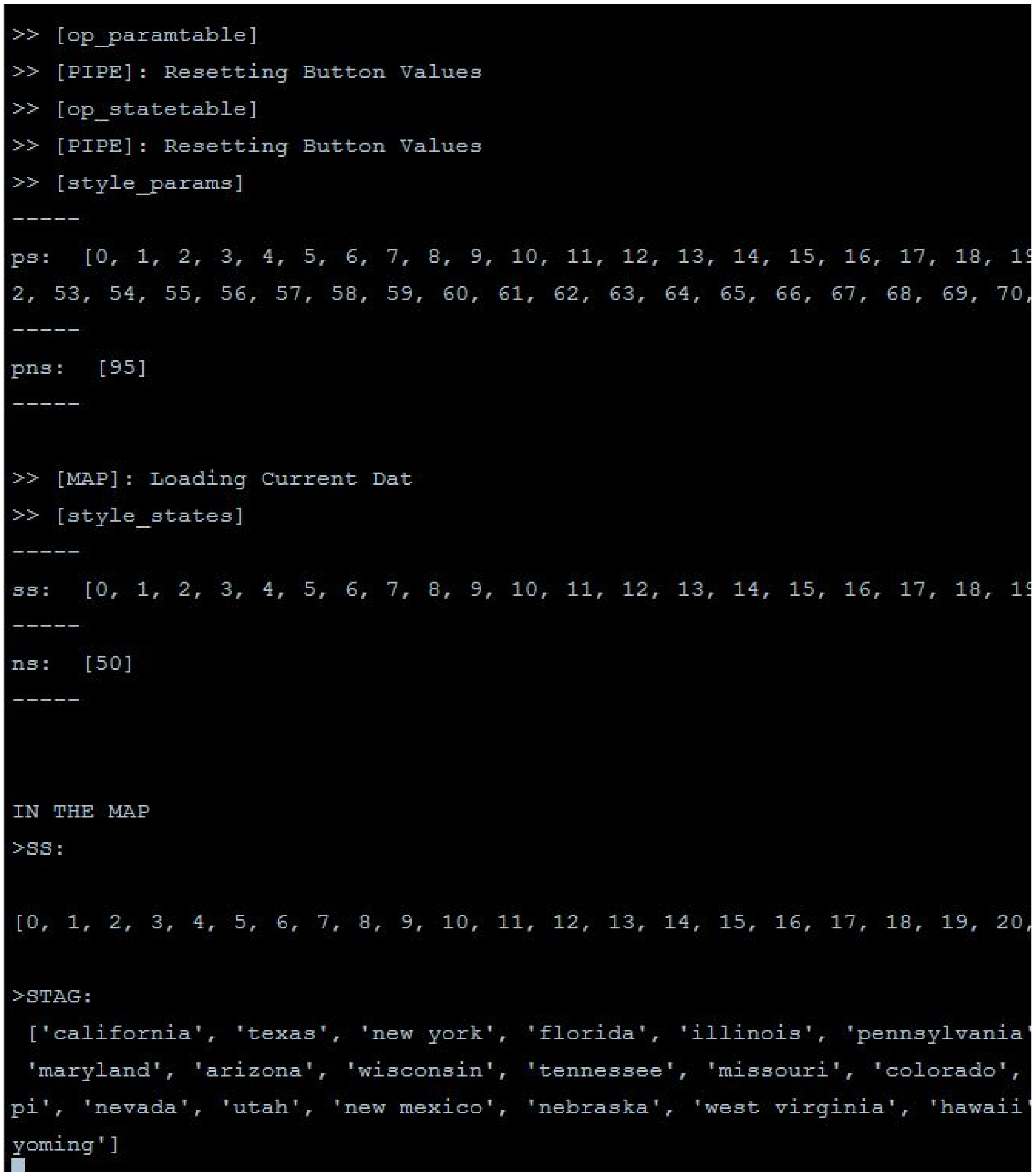
At startup, the application will read the massive amount of source data from memory to hold and parse locally for faster reactivity. As stated before, multiprocessing has saved significant time in startup and data management, as we can see by this screenshot of the backend startup (running/printing in terminal while appliation is alive):



```
[>>] Fetching Data Sets [Time: 2019-05-14 05:20:02.20] ..
[TIME] 0.078 second
[TIME] 0.218 second
[TIME] 0.250 second
[TIME] 13.213 second
[OK] Finished in 13.884 sec. [Time: 2019-05-14 05:20:16.09
[>>] MProcess: Score Data Extract
[TIME] (in seconds): 4.1
[FULL INITIALIZATION TIME (in seconds): 18.096

Running on http://127.0.0.1:8050
Debugger PIN: 797-930-723
```

IV. Backend Communication - Restarts/Reloads

This is shown to help understand the mountain of data being diplayed frim the terminal as the application is used. The outputs are, mostly, intentional tools for monitoring and testing the application (errors/tracebacks will render in the termnal as well). Below is a typical run-through of the calls in the chain when updating, etc... To decifier thus briefly, at the top, the output tells us the application called the parameter table updter, then coninued in the chain for states, sorting, weights and updates (or errors). The large number lists (and string in other places) are showing the current selected (s) versus not selected (ns) rows for a given table. Use it to monitor status of application or remove them from the source code, there is no penalty on performance. AS sais befoe, however:

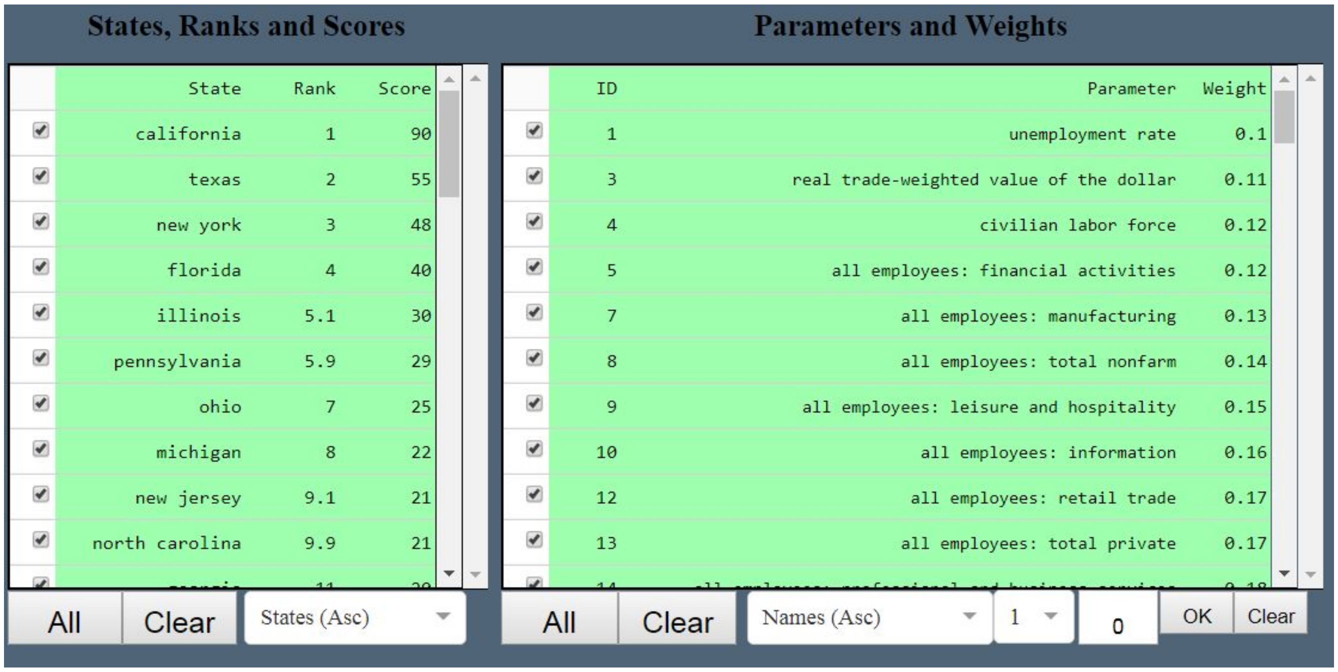


```
>> [op_paramtable]
>> [PIPE]: Resetting Button Values
>> [op_statetable]
>> [PIPE]: Resetting Button Values
>> [style_params]
-----
ps:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19
2, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70,
-----
pns:  [95]
-----


>> [MAP]: Loading Current Dat
>> [style_states]
-----
ss:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19
-----
ns:  [50]
-----



IN THE MAP
>SS:

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,

>STAG:
 ['california', 'texas', 'new york', 'florida', 'illinois', 'pennsylvania'
 'maryland', 'arizona', 'wisconsin', 'tennessee', 'missouri', 'colorado',
pi', 'nevada', 'utah', 'new mexico', 'nebraska', 'west virginia', 'hawaii'
yoming']
```

Features

The application hosts many user-oriented features that allow for full control and customization of the analysis tools. They integrate the backend processing power of Python with the front-end display, designed to minimze hang times while maintaining reactive behaviors. In this section, we will cover the full array of functionalities available as well as some notes on optimization in the backend.

I. Dynamic Data Table Controls



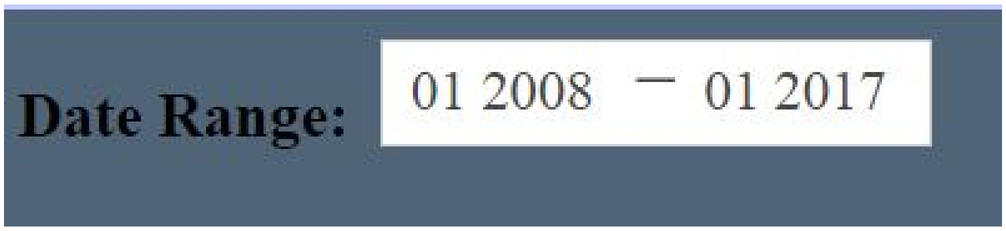
States, Ranks and Scores

	State	Rank	Score
✓	california	1	90
✓	texas	2	55
✓	new york	3	48
✓	florida	4	40
✓	illinois	5.1	30
✓	pennsylvania	5.9	29
✓	ohio	7	25
✓	michigan	8	22
✓	new jersey	9.1	21
✓	north carolina	9.9	21

Parameters and Weights

	ID	Parameter	Weight
✓	1	unemployment rate	0.1
✓	3	real trade-weighted value of the dollar	0.11
✓	4	civilian labor force	0.12
✓	5	all employees: financial activities	0.12
✓	7	all employees: manufacturing	0.13
✓	8	all employees: total nonfarm	0.14
✓	9	all employees: leisure and hospitality	0.15
✓	10	all employees: information	0.16
✓	12	all employees: retail trade	0.17
✓	13	all employees: total private	0.17

The data tables are used as the primary analysis-adjustment tool. They allow the user to vary the states analyzed as well as the parameters used in the scoring, which weights to apply (if any) as well as the date range tool setting the current analysis period from 1 month to 145 months (full period). The tables' rows are selectable, where selected rows are highlighted green and the deselected are highlighted red. The State, Ranks and Scores Table controls the map visualization, while the Parameter table controls the settings for rescoring. The Ranks serve as numerical summaries for reference with the scores becuauase score data drives most of the analysis tools.



Date Range: 01 2008 – 01 2017

The date range also contibutes to the score as it sets the range of dates (rows) for averaging the scores. This is done to aggregate the results into list of 50 scores for weighting over any single or multiple time period(s). The weights are found per paramater on the respective table, and the ID column is used for selecting parameters whose weight the user wants to change (explained below)

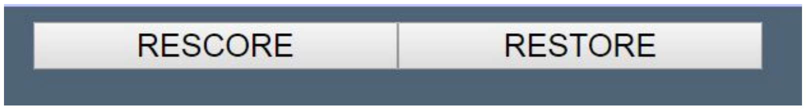
II. Aggregation Buttons

(Referencing the table graphic above) There are 3 elements under each table that are the same. These are the add-all button, clear-all button and the sort-by with asc-desc dropdown menu. They are each isolated to operations on their respective tables and add a huge benefit to quickly aggregate the data pool or wipe it clean and create specific scenarios or aspects to test. They are relativley straightforward in both operation and design.

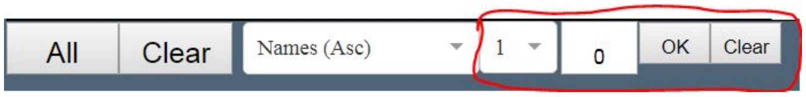
III. Sorting (with maintained selections)

This is an important feature. The tables, can be sorted with the selected cells maintained in the resutls, this means that users can manipulate the data as they wish, selecting/deselecting data until they are satisfied. Reference the figures in part I. Dynamic Data Table Controls above to see the 2 dropdown menus next to the aggregation buttons under the tables. The user has all columns to sort by in asc/desc directions, so they can make selection choices per table and rotate the table to easily isolate the sections they are looking for. The fact that the tables maintain their selected states was tricky, especially when based in a master-updater super function, but it works well and provides an invaluable convenience feature to the user.

IV. Re-weighting, Re-scoring and Restoring

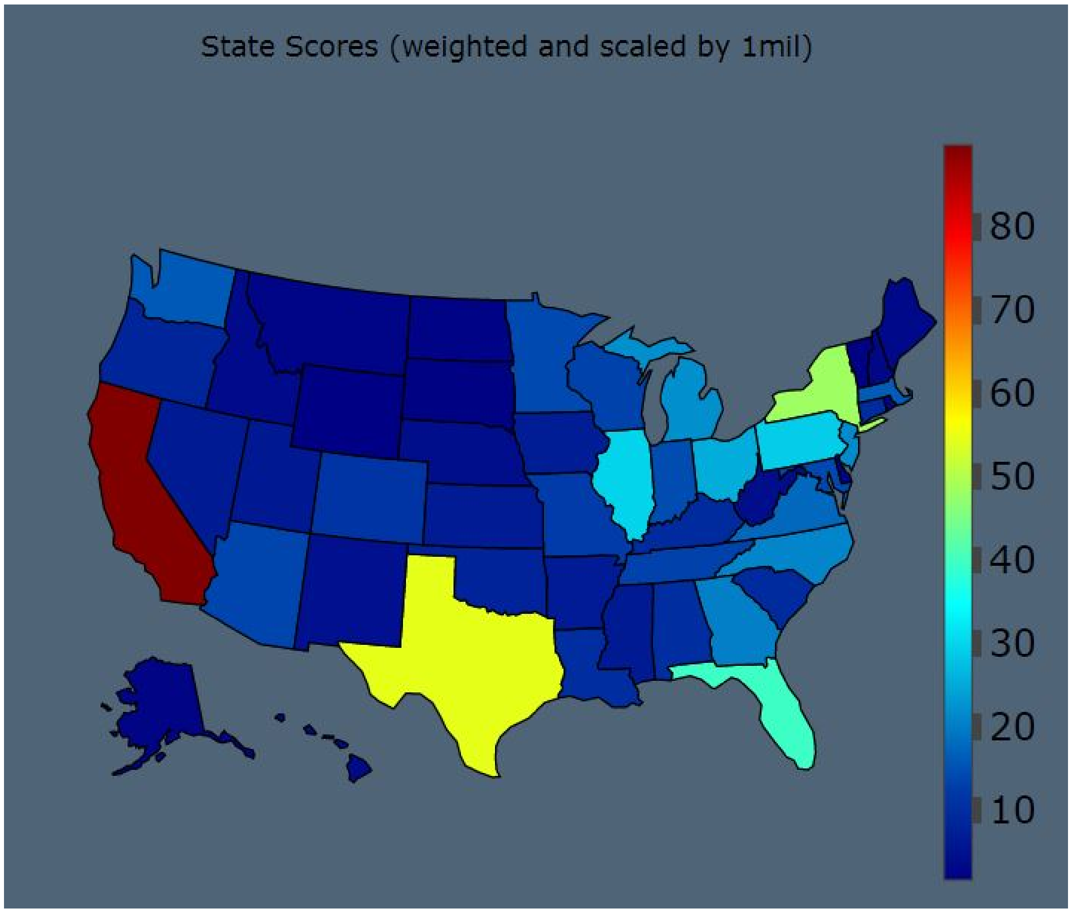


The 'Re' buttons are important here. If the application has to rescore at every update, the application would be slow and break down, especially considering multiple update triggers are simply sorting and map-related updates that have no effect on scores. In this case, the application would be doing the operations with the same results over and over again and with a cost of approx. 4-6 seconds, that cannot be allowed to heppen. For the tables and map, continous updating ismuch lighter and simpler to do. Scoring is a heavy operation (the longest currently during runtime) so the seperation of the scoring makes sense and is necessary. Restore exists to simply reset all data (in tables) back to the startup data.



Finally, the best for last. The user can alter the weights of the parameters as well as the subset used. The startng weights DO NOT SUM TO 100%. This is intentional. Some users may think of weights as ratios, whole numbers or just relative preferences unknown to te deveolopers. That being said, there are no bounds on the weights one can apply, so the user has complete control over what states they want, over what dates, and for any subset (or none) of the data sets with any degree of weighting desired. This is a major component and it is stable as well as it follows the rules of the tables (i.e. it acts like normal data no matter how many times you change it so the integration is seemless). To use it (see above figure), one selects the ID number from the dropdown (and they can input it by hand for quick referencing) and the current weight from the table fills the adjacent input field (currently 0). The user can edit this value however they want (as long as types do not conflict) and set the change by pressing the 'OK' button. The 'Clear' button clears all weights from th table and replaces them with 0. This was implemented for users who want to build their own story from the ground up.

V. Reactive Map Visualization



The map visualization shows the US geographic breakdown of states where each state is shaded to represent its respective score level relative to the range of scores. The important thing to note here is that the score (and color) range re-adjusts for the current state selection (in real time). You can also notice that un-selected states are not colored (which is done on purpose to help bound the area of selection more easily to the eye). This map helps the application capture the audience with easy to digest visual comparisons, and the customization options allow the user to alter the map indefinetly. One can also use the default utilities on the plot object (such as 'box' or 'lasso' group selections, panning/zooming and even the ability to export the plot as a static image plot).

VI. Notes on Optimization

Applications, especially of the data size we are hosting, suffer from speed related issues and make the experience poor and the analysis untrustworthy. To solve this issue, we implemented multiprocssessing in both the initial application data reading as well as the rescoring operations. This helped us cut down start-up times from 60-70 seconds down to about 18-20, which is very siginificant at runtime. For scoring, considering the user will use this feature heavily, we had to speed up an operation the contained the reading, parsing and and aggregating (into score-weighted tables) over 700,000 data points across 95 tables. Those tables are also anywhere from 1 to 50 columns long (states) and 145 rows deep (datetimes - unix). Multi-processing helped tremendously by getting operational delay to, on average, under 5 seconds for a full selection set.

Issues, concerns and notices

At this time, the application needs to implement these additional functionalities:

1. Filters for both the State and Parameter tables (Some source code exists)
2. Statistical analsis plots (Distributions of raw, aggregated data per state/time/param effects)
3. Downloading data ability for users (to extract current test and results)
4. More advanced CSS styling for the look and feel of the application on deployment

In the event the application seems to hang in startup/refresh and the layout (webpage) seems half or partially rendered, a quick hack is to click the 'All' button under the State table, then press the 'Restore' button to fix the layout lag. That has actually been working very well as of this paper submission.

In addition, we would also like to make a few points about the project, running it and working with the source code. Multiprocessing is tricky and we have stable, working code, but (as found in testing) repeated and continuous use, especially with repetitive data calls, rescoring or sorting, the threads that are created can conflict with running ones or endpoints could become compromised (the thread throws a 'nullpointer error' which is the action of pointers pointing to missing/bad data or the use of a null pointer, which goes to null and returns null). It should be 100% fine but in case it does happen, close down your IDE, make sure all the processes are closed in the task-manager processes window and then restart Python. To run, we recommend an IDE (so you can do both) but running the app from the command line clean and simole. The source folder should have everything needed to run the app (no need to retrieve the data again, but the cose is there if you need to). Remember to pip install -r requirements.txt into a Python Virtual Environment to have the dependencies needed to run everything. This can be done by hand as well.

Otherwise, enjoy the app, code and report and please send feedback so we can improve our work. Thank you.