**ETL Project: Renewables Revisited**

Graham Penrose

Scott Clark

12/21/19

The goal of this project was to build out a comprehensive database for how energy flows in California. We established a solid set of data on renewable energy in California with our first class project, but we wanted to continue to expand the work. We sought out additional ways to gather data based on the skills taught in class with three particular goals in mind: find renewable energy information for after 2017, incorporate non-renewable energy production data, and identify hourly demand data.

We were successful in locating the data that we sought from the California Independent System Operator (CASIO) and the US Energy Information Administration (USEIA). CAISO provides more than 80% of energy in California, so it serves as a good proxy for how California is making progress on increasing renewable energy production. We pulled production data from CAISO by scraping but were not able to access demand data without API access, despite multiple requests to CAISO. However, CAISO reports information to the federal government, so by using USEIA API we were able to locate the missing information. The data collected was transformed and loaded into a relational PostgreSQL database.

**Extract**

Demand Data: We received USEIA API access and used it to pull all the available hourly demand data for CAISO specifically. The JSON data was in an easy to use form with all hourly data listed within a single nested dictionary key. We used the pandas module to read the data into a dataframe.

Production Data: Obtaining the comprehensive production data required scraping daily report files on the CAISO website using beautifulsoup, requests, and pandas. In the scraping code, the requests module navigated to a webpage that holds daily reports based off the date given in the url string in a loop that changed the date parameter in the url for each iteration. The returned text data wasn’t within a “<table>” tag, so using pandas to read the html directly wouldn’t work. To overcome this, we used the StringIO module to transform the returned text into a “.txt” file in memory and used pandas read\_csv to read from that temporary “.txt” file. Using this method, we split the displayed data into two tables and appended them to a dictionary with the corresponding date for the hourly data tables as the key. This required some less dynamic, hard coded parameters to be set on the way pandas would read the temporary “.txt” file.

While the scraping worked, there were still challenges with time-out errors when trying to access the data. In a loop, the scraping code hit the CAISO website about 1,300 times and turned the displayed data table into two dataframes. Sometimes, the code would error out due to the website not responding quick enough for the default "requests" module. This error was addressed by adding "with requests.get(url) as r:" to the code in order to open and close the session within every loop, which seemed to help. Opening and closing the session within each loop allowed the code to run, however sometimes it would still timeout and throw an error. We didn’t find a perfect solution to this, other than just waiting for a minute or two then re-running the code.

In both demand and production data, there was a stretch of data missing from 2019. It is unclear why this data was missing, but if CAISO failed to produce data for those dates, it makes sense that it was also missing from USEIA datasets. We obtained data from July 1, 2015 to the current day from the above sources.

**Transform**

Demand Data: There were two minor issues with the USEIA data. The timestamp was not in a format that pandas would recognize, as it had a string on the end of it that did not pertain to time. The string was split off and the timestamp worked. The second issue was that there were a handful of duplicate timestamps in the USEIA dataset. As timestamps were used as a primary key in our database, duplicates prevented us from loading our data into pdAdmin. This was resolved by using drop\_duplicates, keeping only one of the timestamps.

Production Data: Of of the dates we collected, there were only two days that seemed to be generated with errors in the data and made them impossible to scrape. We just added a couple lines to the scraping code that would skip those dates when looping through and grabbing the data. The scraping loop would fill two dictionaries with keys of each date scraped and values of the collected dataframes. Then, we essentially stacked the dataframes within each dictionary on top of each other to build both an hourly renewable production table, and an hourly total production table. Finally, some column manipulation was needed to get the table formatted in the way we wanted our database structured.

For both production and demand data, we used the datetime module to break timestamps into the date and hour categories we needed for our relational database. We structured into First Normal form by removing duplicates and ensuring a single value per field.

**Load**

APostgreSQL database with three schema and five tables was built.

* Production:
  + Hourly renewable production by renewable energy source (solar, wind, etc.)
  + Hourly production by all energy sources (solar, wind, nuclear, etc.)
* Demand:
  + Hourly demand
* Comparison
  + Hourly percent of production provided by renewable sources
  + Hourly percentage of demand met by renewable sources

We related all tables on the timestamp, which served as the primary key.

To bring it all together, we made a simple flask app with a styled home page and four possible routes that served as API’s. The purpose of this API is to allow a user to have access to the data we collected without having to do our full ETL process. A user could pull data from either the “daily percent of production provided by renewable sources” table or the “daily percentage of demand met by renewable sources” table. They could either pass an individual date or the start and end dates of a date range to pull from the two comparison tables. All the routes have an optional url parameter of “units” where a user could change the returned data to be in GWh rather than MWh.