For week 35:

Data Cleaning: Remove New York and Borough Name from all addresses. New York is understood; Borough name is stored in its own column.

Add columns for RENTAL\_COUNT (number of transactions per location) and LOCKER\_COUNT based on number of LOCKER\_BOX\_DOOR values.

The dashboard has a selector for BOROUGH. With only 33 unique locations in this data sets, it is easy to naviage to any location even when all BOROUGH’s are shown.

This dashboard groups all data by gender. Top row left has a world choropleth map with selected country highlighted. Top row right is a histogram of user counts, binned by AGE\_RANGE, EDUCATION, or YEARS\_EXPERIENCE. I experimented with Plotly Studio for ideas, but have developed this mostly from scratch.

Bottom row left is a pareto chart showing the top 10 of CS\_LANG, AI\_ASST, or AI\_FEATURE. I made a design decision to place the highest numbered category on the bottom instead of the top to steer clear of the legend. The bottom row right is a barbell chart made with px.line, for direct companion of Male vs Famale responses.

Dataset has 608 columns, but many are part of a multi-column ‘dummies’ format groups, which uses separate columns for each unique answer to the same question. The dummies format is also considered to be a sparse dataframe, and is why the dataset is so wide

Polars concat\_list was used to gather the CS\_LANG, AI\_ASST, and AI\_FEATURE values into 3 columns with datatype of list. Polars support for list and array types is far more extensive and efficient than the pandas equivalent. I use the number of answers from each user by category to weight the values. For example, when users list 4 programming languages, each one is given a weight of ¼. An index is added to give each user a unique ID, before exploding the dataframe to separate the list values into separate rows, similar to a melt or unpivot.

The cleaned dataset is saved as a parquet file, with all datatypes optimized for storage size. String categories are cast as Categorical, float columns are cast as Float32, Integer columns cast as UInt8. The app checks for the parquet file, and it is used as the global dataframe. If not found the app will reads the csv file, cleans it, and saves the data frame as parquet for future runs. This allows me to upload a much smaller parquet file to Plotly Cloud and avoids having to read the csv file and clean it every time.

I hope you enjoy this dashboard and appreciate any feedback or suggestions.

Here is a link to Plotly Cloud hosted dashboard:

Here are a few screenshots:

Here is the code:

.

When designing a dashboard with a multi-level pulldown scheme (i.e., cascading dropdowns) for geographic data like State, County, City, and Zip Code, you need to ensure that the design is user-friendly, intuitive, and performance-optimized. Here’s a strategy to approach this:

**1. User Flow & Interaction Design**

* **State Dropdown:**  
  The user will first select a state from a list. This dropdown should be the default visible one.
* **County Dropdown:**  
  Once a state is selected, the County dropdown should dynamically update to show only the counties within the selected state.
* **City Dropdown:**  
  After the county is selected, the City dropdown should show only the cities within that county.
* **Zip Code Dropdown:**  
  Finally, once the city is selected, the Zip Code dropdown should populate with the zip codes associated with that city.

**2. Dynamic Filtering & Dependencies**

* **Cascading Effect:**  
  Use a cascading effect where each dropdown’s options are dependent on the selection of the previous dropdown. For instance:
  + Selecting a state filters the available counties.
  + Selecting a county filters the available cities.
  + Selecting a city filters the available zip codes.
* **Loading States:**  
  Make sure that each dropdown displays a "loading" state until the data is fully fetched for the next level. You could use a spinner or an "Updating..." text as a visual cue.
* **Clear Selections:**  
  If a user goes back a level (e.g., selects a new state), the lower-level dropdowns (county, city, zip code) should reset or be cleared.

**3. Data Source & Performance**

* **Efficient Data Retrieval:**  
  To avoid overwhelming the user or the system, fetch data dynamically (on-demand) based on user selections. This means instead of loading all possible states, counties, cities, and zip codes at once, only load the relevant data for each level.
* **Use AJAX or Fetch API:**  
  Load data asynchronously (AJAX or Fetch API) when a user selects a dropdown option, to prevent page reloads and improve responsiveness.
* **Server-Side Filtering:**  
  If the dataset is large, implement server-side filtering for each dropdown level. For instance, when a state is selected, a request should be sent to the server to fetch only the counties in that state.

**4. Usability & Accessibility**

* **Pre-select First Option:**  
  If possible, pre-select the first option in each dropdown to minimize the number of actions the user needs to take.
* **Clear Instructions:**  
  Make sure the purpose of each dropdown is clear, either with labels, tooltips, or placeholder text in each dropdown.
* **Keyboard Navigation:**  
  Ensure that users can navigate through the dropdowns with the keyboard (tabbing through each dropdown).
* **Mobile Optimization:**  
  Ensure the dropdowns are mobile-friendly. On mobile devices, use touch-friendly controls, like single-select or scrollable lists.

**5. Handling Edge Cases**

* **No Data Available:**  
  If a user selects a state that doesn’t have any counties, or a county that has no cities, the corresponding dropdowns should show a message like “No data available.”
* **Dynamic Error Handling:**  
  If the data for any dropdown fails to load due to a network issue or server error, ensure that the user is notified gracefully, e.g., "Unable to load counties. Please try again."
* **Data Integrity:**  
  Make sure the data for each level is complete and accurate. For instance, zip codes should be assigned correctly to cities, counties, and states.

**6. Visual Design and Layout**

* **Dropdown Size:**  
  Ensure that the dropdowns are large enough to be easily clickable and readable. Avoid overwhelming users with too many visible options at once.
* **Grouping Options:**  
  If the list of options is large (for instance, all states), group them alphabetically or by regions (e.g., “Northeast,” “Midwest,” etc.) to make selection easier.
* **Space Management:**  
  Use a compact layout so the dropdowns do not crowd the screen. Consider making each dropdown collapse or expand as needed, so they don’t occupy too much space.

**7. Data Preloading for Speed**

* **Use Caching:**  
  To reduce load times, consider caching data on the client side (e.g., in localStorage or session) for previously selected regions.
* **Lazy Load for Cities and Zip Codes:**  
  Instead of preloading all cities and zip codes for all counties, consider using a lazy load approach where cities and zip codes are only loaded after the county is selected.

**8. Optional Enhancements**

* **Search Functionality:**  
  Allow users to search within each dropdown if there are a lot of items. A search box inside the dropdown can help users find what they're looking for faster.
* **Autocomplete:**  
  For large lists, you could implement an autocomplete feature (especially for zip codes) to speed up data entry.
* **Visualization:**  
  If applicable, you could show a map or a chart dynamically based on the selections (e.g., showing population or demographic data of the selected zip code or city).