Results

Anthony Mascari

Western Governor’s University

C964: Capstone

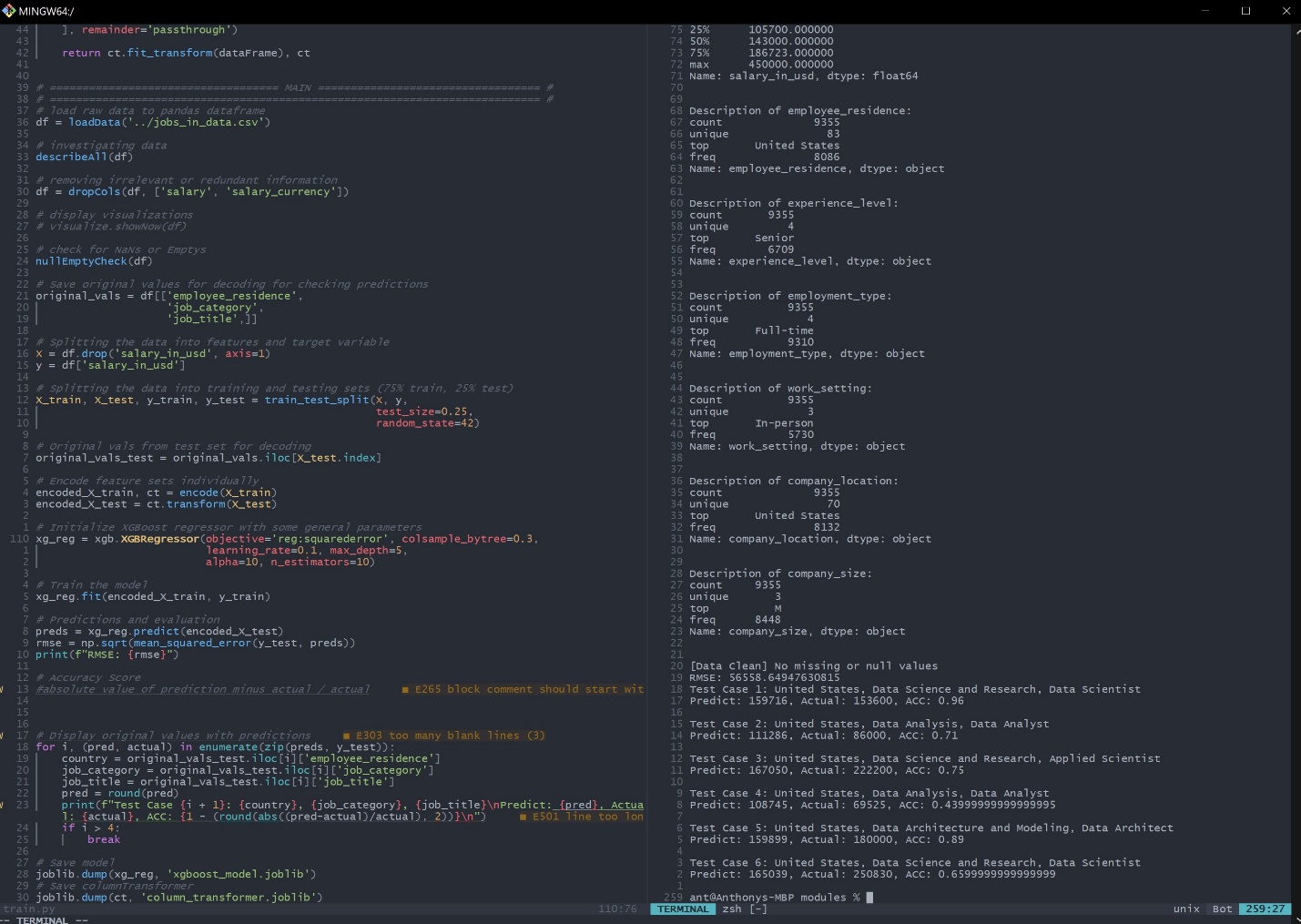
# Overview

Throughout the testing process there were a variety of different stages and challenges that varied in complexity. The process began with an initial look into the raw data in order to observe what features may play an important role in the model’s performance. From there the model was tested and iterated upon to improve accuracy through the tuning process. After the model was tuned, it was saved and functionality was implemented in order to use the model with a parameterized function. After the function was created an API was developed in order to provide a front end a way of selecting available options (and only available options to avoid input validation errors). The front end was then able to query the API with parameters relevant to input for the model and the API returns a prediction to the front end. Additionally functionality was added to provide three customized charts for the user inquiry. These charts are produced on the server side and sent as bytecode to the frontend for display.

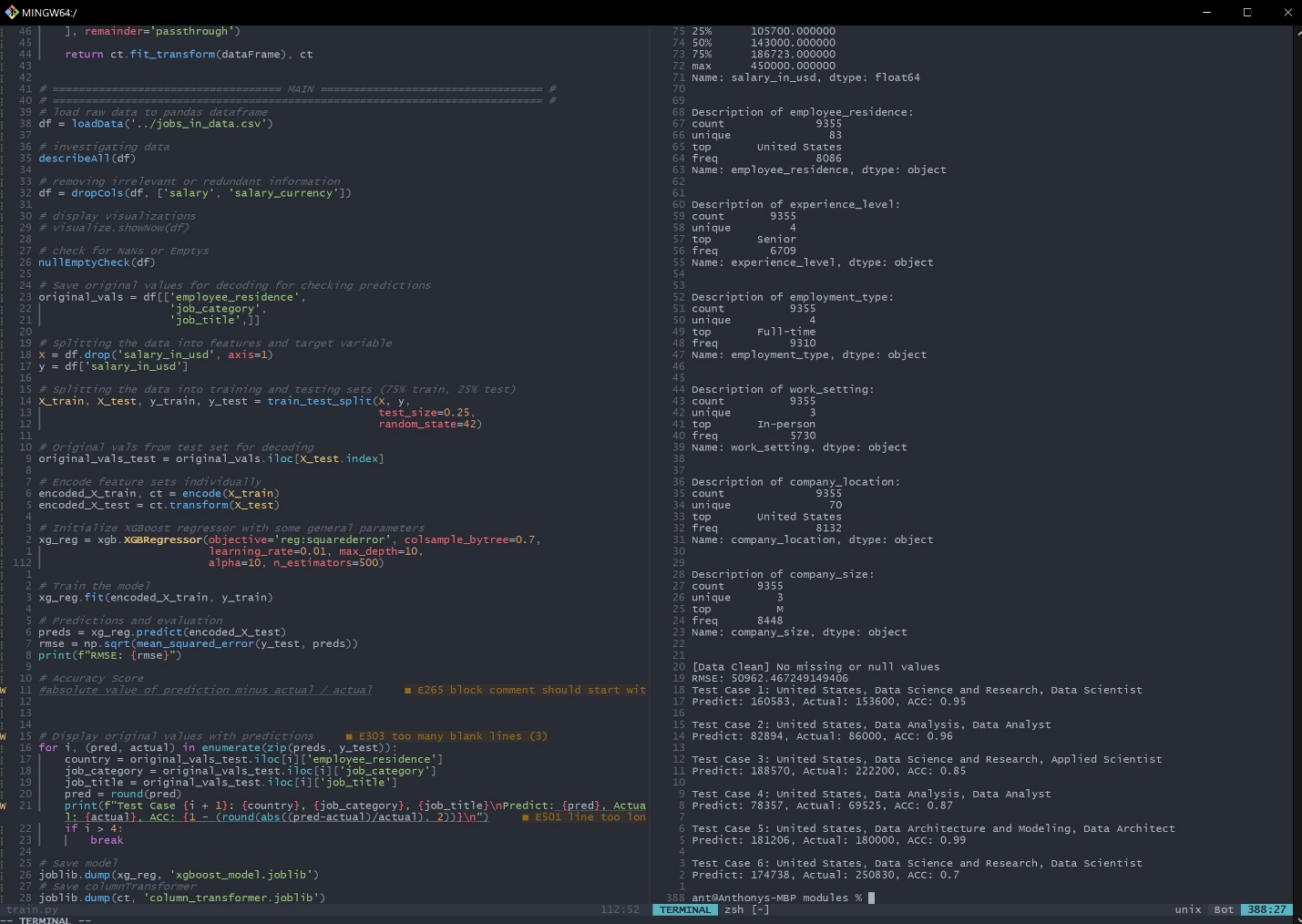
# Model Development

Model development was the first major area in developing this product that went through testing, revisions, and optimizations. There were many testing cycles and iterations. In the screen captures below it is apparent how drastically turning parameters within the model affect performance. In the first screen capture (*Figure 1.1*) you may view one set of parameters on the left split window with the results printed to the console in the right window. Note how these parameters and results vary dramatically from the results on the next screen capture where the model has been finely tuned (*Figure 1.2*). In the console window to the right in each screenshot you may view the input, the salary prediction for the input and then you can see the accuracy score derived from the difference between the prediction and the actual value from the dataset.

**Figure 1.1** *Initial model, poorly tuned model accuracy.*



**Figure 1.2** *Tuned model, greatly improved accuracy.*



# API Implementation

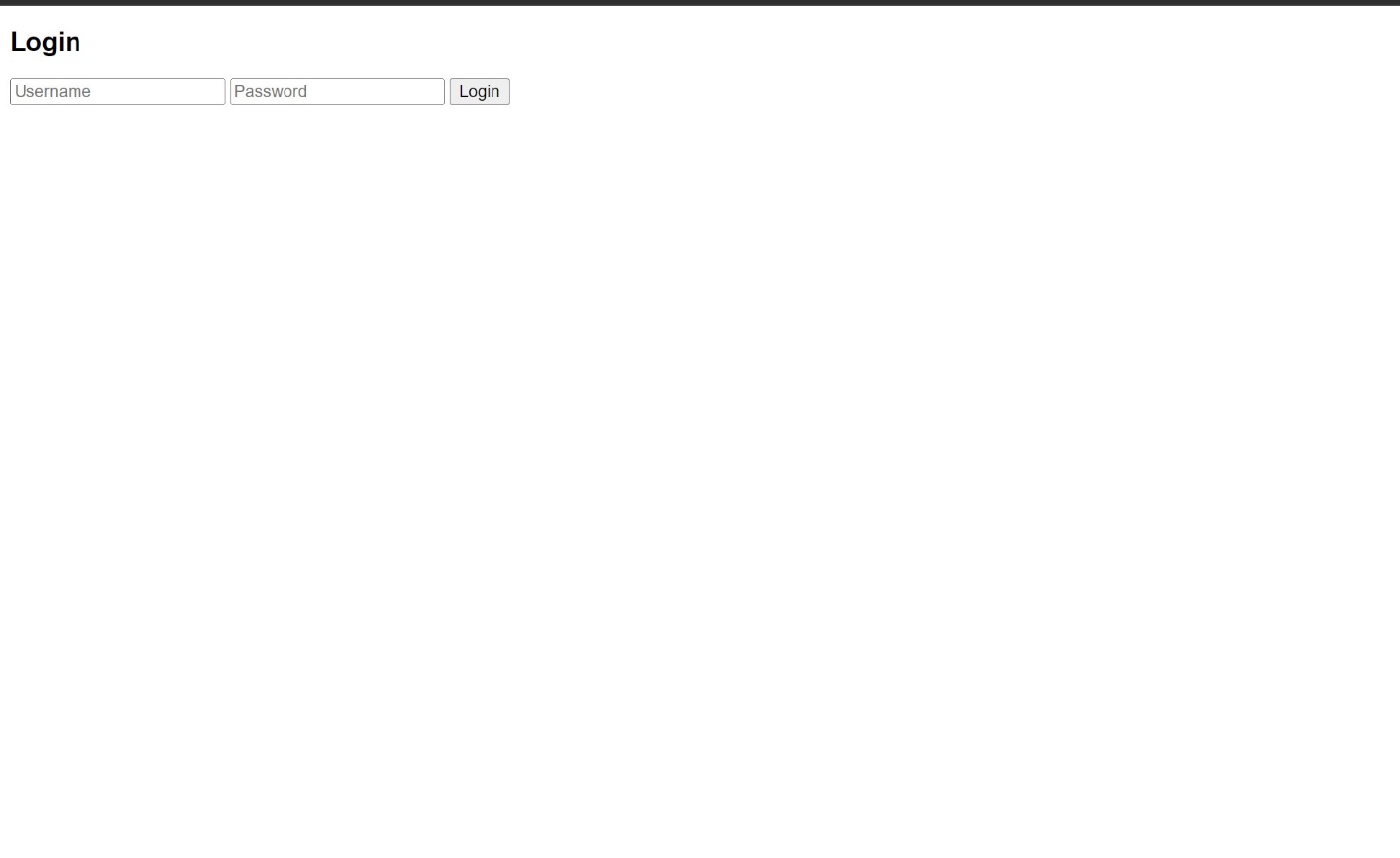
After the model was tuned and saved, the API was created to allow a front-end to communicate with the model and use a variety of different features. The API also went through a variety of tests (via curl requests), revisions (rewrites) and optimizations. A screenshot including the structure and endpoints of the API is visible below (*Figure 2.1*). Note how an endpoint is provided which supplies the options for each of the categories the model is using to make predictions. This was done in order to provide the front end a list of options to create dropdown menus from where the user can select items to input into the model without having to be overly concerned about input validation or user errors.

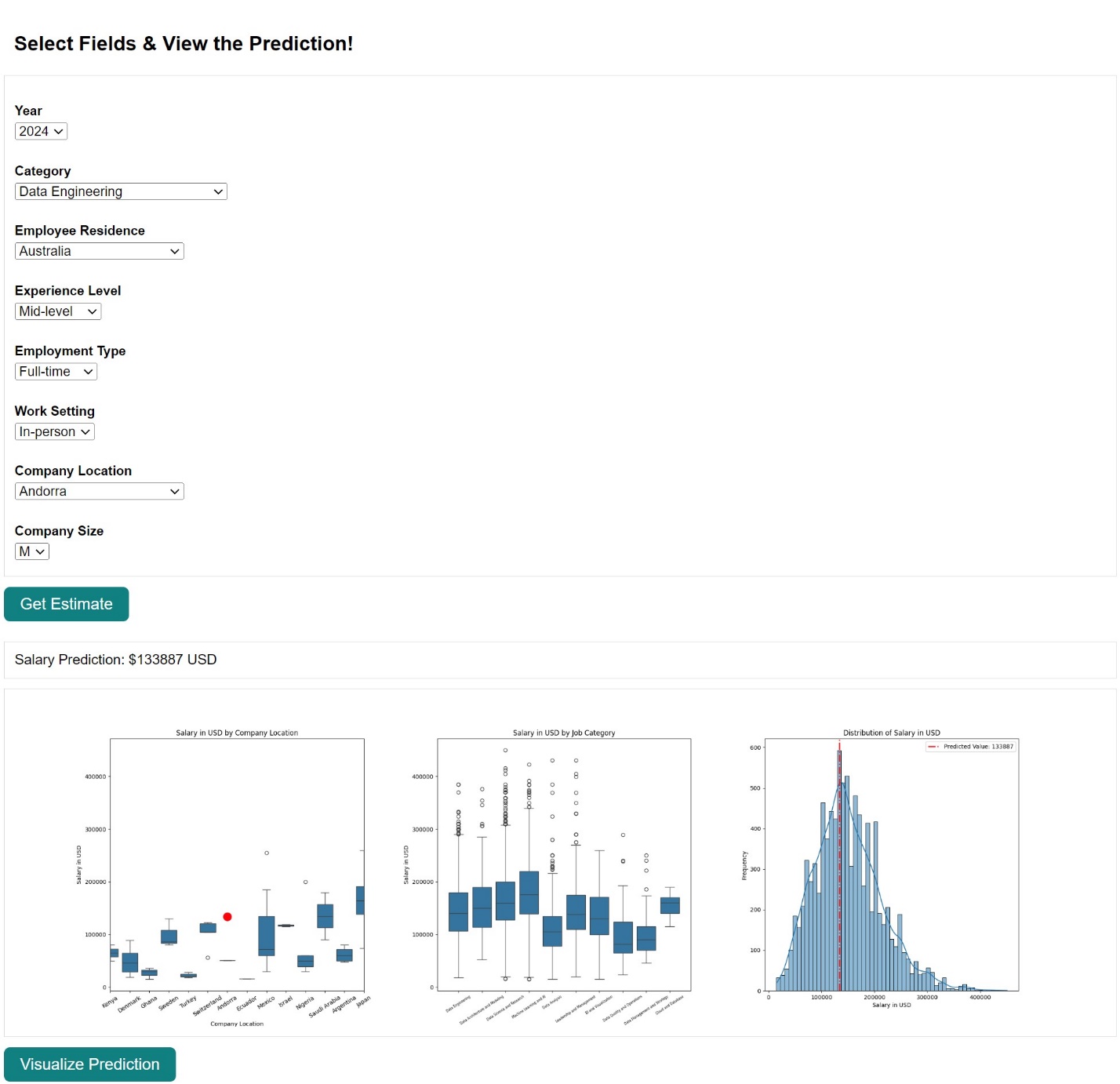
**Figure 2.1** *API endpoint results & implementation*

# Front End Testing & Results

Once the functionality of the API endpoints was tested using curl requests, it was time to create the front end of the application for users to interact with. Below are a few screenshots of the result of the user experience after testing and optimizing.

**Figure 3.1** *Login page*

**

**Figure 3.2** *User Interface*

# Other Performance Optimizations

Throughout the process there were a number of processes that were optimized. One optimization that took place on the back end was reading in the raw data from the CSV only on the start of the program instead of every time the dataframe was referenced by an API call. This enabled the information to be stored in memory and accessed much faster than repeated I/O calls every time data from the csv was needed. Another optimization that was implemented was during the image creation and delivery process from the back end to the front end. Originally, I had implemented a system that wrote the image to disk then transferred the image to the front end after it had been written to disk. This caused a much greater delay than simply storing the image temporarily in memory and transferring the byte data directly to the front end from the memory buffer. This allowed me to entirely skip the slow process of writing and reading images from disk then converting them to byte data.

One last optimization that was made was the functionality to store the unique features for each category in a JSON file on disk. This allows the program to only look through the CSV for unique values once when the program starts then store the information on disk for API calls and/or in memory for any functions that need to return that data to the front end to create the drop-down menus.