Jacob Weissman

Thesis Reflection Paper

In this project, I tackled two objectives. First, I created a way to easily plot weather data. While there are sites that offer vast amounts of historical weather data, it is uncommon to find visualization tools for comparing different locations and years. I created a tool using MatPlotLib that allows users to easily accomplish this task. Second, I used machine learning to tackle a practical weather question: can weather predict Citi Bike usage? If the answer was yes, then my model could optimize the activities of Citi Bike rebalancing trucks, bike commuters, and traffic police. I built a model using scikit-learn based on weather features that can predict how many Citi Bike rides there will be on a given day with $R^2 = 0.741$ and mean-squared error (MSE) = 0.111 squared standard deviations, a significant improvement over the baseline model performance of $R^2 = 0.491$ and MSE = 0.218.

My first task was to build a way to collect valid input from the user. A date range consists of a start date and an end date, and an interval is a date range with a zip code. To uphold the single responsibility principle, I utilized several functions and layers of validation when collecting input from the user (the relevant files are found in the ‘user’ folder). The user’s start date and end date are validated on their own and relative to each other. When valid start and end dates are detected, the date range is accepted. and the user is asked to provide a list of zip codes (which are also validated) for the given date range. This completes the first set of intervals.

The user is allowed to keep entering intervals to graph, with several caveats. API calls are made using Visual Crossing, which offers free API calls up to 1,000 credits (weather data for single days) per day. Therefore, the maximum number of credits in a single API call is set as a

global parameter in ‘descriptors.py’ to 100. The program (‘user/user_input.py’) that is responsible for collecting user intervals is built to gracefully handle this restriction, as well as instruct the user when incorrect or invalid input is detected. The program consistently prints the current query cost to the user. When the user exceeds the total, the program blocks the interval, informs the user what went wrong, and asks for a new interval (starting again from the start date). The program is also capable of handling exceptional cases, such as the user choosing the same date range twice.

To uphold the data visualization principle of presenting information as simply as possible, all date ranges must be of the same length. This is to avoid having intervals of different lengths on a graph, which is complicated and confusing. To ensure this, when the user picks additional start dates, the end dates are automatically calculated, validated, and printed for them.

The output dictionary containing date ranges and zip codes can then be used by ‘query.py’ to generate and run weather queries through API calls. A query must be made for each date range and individual zip code, so for each date range the corresponding list of zip codes is looped through to generate queries. For data protection purposes, I placed the API key in a .gitignore file. The raw output from ‘query.py’ is a dictionary, which is converted to a dataframe using ‘convert_to_dataframe.py’. After parsing through the dictionary, the dataframe is created that has a row for each day. To map the days back to intervals for plotting purposes, the interval is also saved for each row. Additionally, if a ‘clean’ dataframe is desired, a function is available to perform additional processing. This includes converting sunrise and sunset into a single ‘daylight’ metric, as well as dropping extraneous columns. The dataframe that is returned can be plotted using files in the ‘plotting’ folder.

‘plotting/plot_one_metric.py’ is responsible for all of the plotting in this project. There are several key facts I considered in my approach. I wanted to gain familiarity with MatPlotLib for future use, so I did not want to use any wrapper on top of it. An interval has a start date MM-DD, a year, and a zip code. I utilized colors of lines, linestyles, and datapoint markers to differentiate between these three components. In all graphing cases, the x-axis is time and the y-axis is the metric that the user has chosen to graph.

‘plot_one_metric.py’ consists of two main functions. ‘plot_metric_by_single_interval’ is responsible for plotting cases where there is only one date range. In this case, the x-axis is used to present dates. ‘plot_metric_by_multiple_intervals’ is responsible for plotting cases with more than one date range. In this case, the x-axis is days, and colors are used to distinguish between date ranges. The colors associated with each date are plotted on the graph below the actual lines, for ease of reading. I found using a legend of colors was a lot more cumbersome, and it made colors and dates feel much less intuitive. My altered approach is less cumbersome but requires a ‘hack’ of MatPlotLib. I had to calculate a boundary below the minimum point for each graph, and use this (in addition to y-range information) to buffer and plot each date range.

The graphing feels clean and intuitive. For example, when a single year or zip code is used, it is removed from the legend and placed in the x-axis description. Colors are assigned to dates when there is more than one date; otherwise, I prioritized intervals being clearly distinguishable when assigning the “differentiators” (colors, linestyles, and markers) to components of the interval. The case where a date range crosses January 1st required special handling to avoid the graph breaking down. When dates are used as colors, they are assigned according to a gradient (found in ‘colors.py’), which allows users to easily visualize how weather

changes throughout different dates in the year. Ultimately, users have a clean and easy way of visualizing complex comparisons.

My second aim was to create a model to predict Citi Bike usage. I adhered to the standard machine-learning pipeline, ensuring that I followed best practices throughout. My initial goal was a model capable of predicting both the number of rides and the total length of all rides for a given day (referred to as the 'target variables'), but I pivoted to focusing on only the number of rides after poor preliminary performance for total ride length. I started by randomly selecting three start stations (among stations with > ~30 rides per day) in the Jersey City Citi Bike dataset: Newport PATH (07310), South Waterfront Walkway (07030), and Marin Light Rail (07302). I downloaded all of the data for each of these stations between 2022-11-01 and 2023-10-31, which gave me a row for every single ride. I performed preprocessing by dropping unnecessary columns and renaming columns as needed (bike preprocessing can be found in 'ml_preprocessing/bike.py'). I then performed feature extraction by summing rows over dates so that I had a row for each date with the total number of rides and total length.

For the weather, I queried and saved the weather data for each of the three start station zip codes over the entire date range (weather preprocessing can be found in 'ml_preprocessing/weather.py'). Additionally, scaling was performed to convert range-10 features (UV Index) and range-100 features (humidity, cloud cover, precipitation cover) to range-1 features. I also added a binary workday feature, which indicated if a day was a weekend / federal holiday or a weekday. Finally, in 'ml_preprocessing/dataframe.py', one large dataframe was created where each row contained the date, zip code, number of rides, total ride length, and all available features for that day.

The dataframe was then split into smaller datasets using ‘ml_normalize/split.py’ with a 60:20:20 split into training, validation, and testing. Feature selection was also performed later in the project, which required creating additional datasets. This was to ensure that the training and validation datasets were not biased by the selection of features that had already been seen. Of the non-test data, 10% was used for the feature selection tuning dataset. The other 90% was split 75:25 between the training and validation feature selection datasets. Finally, a non-feature selection train_validation and a feature selection train_validation dataset were created purely for hyperparameter optimization, and later in the project, for training models for the test dataset.

After the datasets were split, they had to be fit and transformed before being used for machine learning. ‘Fitting’ refers to customizing a scaler on a particular dataset. Standard scalers were used to transform the following features: 'daylight', 'tempmax', 'tempmin', 'temp', 'feelslikemax', 'feelslikemin', 'feelslike', 'dew', 'windspeed', 'pressure', 'visibility'. Standard scalers were chosen since they would be expected to somewhat follow a normal distribution, and ‘zero’ values were not meaningful. Min-max scalers were used to transform the following features: 'precip', 'snow', 'snowdepth'. In this case, since the distribution would not be normal (most values would be zero) and the zero values were meaningful, min-max scaling was used.

When training and testing a model, the test dataset should not have a bearing on the training of the model. To prevent data leakage, fitting and transforming were performed by fitting on any dataset that would be used for training, and transforming on the corresponding ‘testing’ dataset. Exact details can be found in ‘main’ in ‘ml_normalize/split.py’. In doing so, no testing was ever performed on a dataset that was used for fitting. For example: fit on train_validation, transform on test; fit on train, transform on validation. Anomalies were

removed from any dataset used for training to ensure optimal training. Importantly, this was only after fitting and transforming. To avoid providing an artificial boost to performance, the corresponding datasets used for testing did not have anomalies removed.

The target variables also had to be transformed to ensure that learning could properly take place. Some stations have more riders merely due to the stations always being more popular. The model would perform poorly if it learned on unscaled target variables and attempted to attribute large popularity differences in the targets to weather. Therefore, target variables were fit and scaled using a robust scaler based on their zip codes.

I then created the predictor sets (all of which can be found in 'descriptors.py'). Each predictor set is a subset of the available features for prediction. Part of building a model is also figuring out which features should be dropped to ensure optimal performance. The simplest predictor set was the 'all features' set, which contains all available features. The second set is the 'baseline' set, which is used for comparison purposes (since it does not contain weather data) and contains only the 'daylight' and 'is_work_day' features. The third set was the 'domain expert' set, which I created based on picking features that I thought would be most predictive while avoiding picking redundant features with a higher-performing counterpart.

I then built the actual machine-learning models. I utilized modular code whenever possible, including creating a generic hyperparameter search and predictor in 'ml_learning/general.py'. I built the remaining functionality for conducting a hyperparameter search and model training/prediction of linear models in 'lr.py', multilayer perceptron models in 'mlp.py', and random forest models in 'rf.py'. The general flow was to perform a hyperparameter search on the train_validation dataset and train on the training dataset.

Preliminary testing (using the validation dataset) was performed on each of the feature sets for each of the models and on each of the target variables (‘number of rides’ and ‘total ride length’). Performance was very low on total ride length (R^2 = ~0.25 in early testing). While I initially hoped I could model total ride length, this is a difficult problem to model. It involves multiple random variables: will a person get on the bike, and then how long will they stay on the bike. It would require its own project to model. Therefore, I decided to focus only on the number of rides. Preliminary testing was also performed to see if principle component analysis would boost performance, but it worsened performance and was abandoned.

I then created a fourth set of features using feature selection. I looked at correlations between features as well as between the features and the target variable (‘number of rides’). The six ‘temperature’ features and ‘dew’ all showed ~90% correlation with each other. Average temperature (‘temp’) showed the strongest correlation with the target variable, so I kept only ‘temp’ and dropped the other temperature features. The five features that showed the lowest correlation with the target variable (<0.2 absolute value of correlation) were dropped: ‘is_work_day’, ‘pressure’, ‘humidity’, ‘snow’, and ‘snowdepth’. While ‘maximum wind gust’ showed some positive correlation with the target variable (~0.2), this goes against domain knowledge. Stronger gusts should mean fewer rides, not more. Therefore, the correlation was likely due to noise, so the feature was dropped. Precipitation amount, precipitation coverage, average wind speed, cloud coverage, and visibility all showed some correlation with the target variable and low correlation with other features, and were all kept. Solar radiation, solar energy, and UV index were all >94% correlated with each other and showed some positive correlation (~0.4) with the target variable. UV index was chosen since it is the only feature commonly

available to the public for decision-making. Daylight had a strong positive correlation (~0.6) with the target variable, and was kept. The final results for the selection feature set: 'temp', 'precip', 'precipcover', 'windspeed', 'cloudcover', 'visibility', 'uvindex', 'daylight'.

I then constructed every model (with its optimal hyperparameters based on train_validation data) with every set of predictors. I trained on the training data and tested performance on the validation data. I ran all models with 'R^2' (which scores how much of the target variable variance was correctly captured by the features) as the scoring function followed by MSE (sum of the square of the difference between predictions and actual values of the target variable) as the scoring function. (Note: for MSE, lower is better.) All of the results were saved to a file called '[scoring]_validation_results.txt'. The highest performing pair of model/predictors for both R^2 and MSE was feature selection predictors and a random forest model, with R^2 = 0.774 and MSE = 0.103. Note that the units of MSE are squared standard deviations, since the target variable was standardized. A new model using the train_validation dataset was trained using selection predictors on a random forest model. This final model was tested on the test dataset, and achieved R^2 = 0.741 and MSE = 0.111. The model performance was significantly better than the optimal baseline model (trained on baseline features), which was the random forest for both R^2 and MSE. When retrained on the train_validation dataset, the baseline model achieved R^2 = 0.491 and MSE = 0.218 on the test dataset. Interpreting the MSE results, a MSE of 0.111 means that the predictions were incorrect by, on average, 0.333 standard deviations, while the baseline model was incorrect by, on average, 0.467 standard deviations.

I created a program that allows users to easily utilize all of the tools that I created using the file 'complete_testing/complete_testing.py'. The user's date ranges and zip codes are

collected using ‘user/’, the query is made using ‘query/py’, and the dataframe is created using ‘convert_to_dataframe.py’. The user is then provided a list of metrics available to plot, and can keep plotting metrics for the intervals. Additionally, if the zip codes entered by the user are a subset of the three training zip codes, the user can choose to plot bike activity. In this case, the input data is standardized and transformed on the training scalers. The median, standard deviation, and average for the location are also calculated from training and reported to the user. The dataframe is used to generate a prediction for the target ‘number of rides’, which is reverse scaled back to an actual value based on the zip code. Additionally, the z-score is calculated based on the average and standard deviation (independent of the zip code). The predicted number of rides and the z-score percentile are reported to the user.

There are a few limiting factors in my approach. The most obvious to the user is that they can only generate bike ride data prediction on the three zip codes seen in training. Since only weather features are used to generate the z-score percentile and zip code is not even considered, my model is capable of providing a z-score percentile for any location. However, the model is trained and tested based on three zip codes. I can confidently report performance scores from testing because the data was unseen. However, that only applies to the three zip codes I used in testing! Extending the assumption would require assuming that every other station in the United States, all of which have their own public transportation systems with different reliability, would have the same performance in testing. Since I cannot verify that the z-score percentile has a strong R^2 score backing it up, it would be misleading to report it for other zip codes. Instead, my code makes it very easy to train on new locations and build new models for specific locations.

Another assumption I am forced to make is that predicted weather is accurate when predicting expected bike ride activity. Performance of $R^2 = 0.741$ is only when the weather already happened. In 'complete_testing/complete_testing.py', one can select dates up to 15 days into the future, and therefore view biking predictions 15 days into the future. The performance of 15-day forward predicted activity vs. actual activity is unknown. However, the goal of my project was not to model the weather; this is an entirely separate task. Lastly, I am relying on a single weather data source and a single bike data source. When these sources do not work, there is nothing I can do to compensate. 11 total days over the 365-day interval had zero biking data, and were forced to be dropped. Furthermore, there are many date intervals for which weather data was not properly collected, causing potential graphing issues for users.

When I started this project, I had several goals. I successfully created an easy-to-use weather visualization program, which leverages the massive pool of online weather data to provide users with the ability to graph complex weather comparisons intuitively. I made all of my code public, highly modular, and posted it with a clear 'README' and requirements.txt, allowing future programmers to build on my work and expand my plotting/modeling. Finally, I built a high-performing model that users can leverage with confidence. Citi Bike, which constantly deploys trucks to rebalance stations, can use my tool to predict how and where to deploy trucks. Commuters can determine whether a station is likely to have bikes, and traffic police can determine the number of riders as they anticipate expected biking traffic. I hope my work can make the world more efficient and connected.