Predicting Bike Sharing Demand

A Machine Learning Approach

By: Dustin Bruce and Dillon Hughes

Abstract:

In this project, we aimed to predict bike-sharing demand using a dataset containing historical data on the number of hourly bike rentals, weather conditions, and other relevant features. The primary challenge was to preprocess the data, engineer relevant features, and train an accurate predictive model. To solve this problem, we employed the XGBoost algorithm, which is known for its efficiency and effectiveness in handling structured data. We utilized Optuna, a hyperparameter optimization library [2], to fine-tune the model's performance by searching for the best hyperparameters. Additionally, we employed 5-fold cross-validation to ensure robust model evaluation. The model's performance was assessed using the root mean squared logarithmic error (RMSLE) metric. Our final model achieved a minimized RMSLE score through the identified optimal hyperparameters, providing accurate bike-sharing demand predictions, which can be useful for bike-sharing service providers in planning and managing their resources.

Introduction:

The increasing popularity of bike-sharing systems across the globe has revolutionized urban transportation by providing a flexible, eco-friendly, and cost-effective alternative to traditional modes of transportation. Bike-sharing services have been implemented in various cities, offering numerous benefits such as reduced traffic congestion, lower greenhouse gas emissions, and enhanced public health due to increased physical activity. However, managing bike-sharing systems effectively requires accurate demand prediction, which remains a significant challenge for service providers.Bike-sharing demand prediction is a complex problem due to the multifaceted nature of factors influencing bike usage, such as weather conditions, time of day, day of the week, and seasonality. Accurate demand prediction is crucial for service providers, as it enables them to optimize fleet management, allocate resources efficiently, and ensure a satisfactory user experience. Inaccurate predictions can lead to bike unavailability or overcrowding at docking stations, resulting in customer dissatisfaction and reduced system utilization.Various machine learning techniques have been employed to address this problem, including linear regression, decision trees, random forests, and gradient boosting machines. Recently, the XGBoost algorithm has gained popularity due to its effectiveness in handling structured data and its ability to provide feature importance analysis, making it a suitable choice for predicting bike-sharing demand. Hyperparameter optimization is another crucial aspect of model development, as it can greatly enhance model performance by identifying the best set of

hyperparameters.In this study, we tackle the bike-sharing demand prediction problem using an XGBoost model with Optuna for hyperparameter optimization. We preprocess the dataset, engineer relevant features, and employ 5-fold cross-validation to ensure robust model evaluation. The model's performance is assessed using the root mean squared logarithmic error (RMSLE) metric. Our approach aims to provide accurate bike-sharing demand predictions, which can aid service providers in planning and managing their resources more effectively.

Methodology:

Preprocessing

In the preprocessing step, we first focus on the 'datetime' column. As the original 'datetime' column contains a string representation of the date and time, we convert it into a datetime object using the Pandas library's 'to_datetime()' function. This conversion allows us to easily extract relevant time-related features that can potentially improve the model's predictive performance. After converting the 'datetime' column, we proceed to extract various time-related features from it. These features include the hour, day, day of the week, month, and year. We believe that these time-based features can capture the temporal patterns and trends in bike sharing demand. For instance, the demand for bikes might vary depending on the time of day, with higher demand during rush hours or lower demand during nighttime. Similarly, demand can also vary depending on the day of the week, with weekends possibly having different demand patterns compared to weekdays.Next, we create interaction features that combine the extracted time-based features with other existing features in the dataset. Examples of such interaction features include 'hour_workingday', 'hour_temp', and 'hour_humidity'. These features can help the model capture the combined effects of multiple features, such as the effect of the hour of the day and the working day status on bike demand.Finally, we perform one-hot encoding for categorical features such as 'season', 'weather', and 'year'. One-hot encoding is a widely used technique for transforming categorical variables into a binary format, allowing machine learning algorithms to better handle these features. We use the Pandas library's 'get_dummies()' function to generate one-hot encoded columns for the categorical features and drop the original columns. At the end of the preprocessing step, we have a dataset with a rich set of numerical and one-hot encoded categorical features that is ready for feature scaling and subsequent model training.

Dropping unnecessary columns

In this portion of the code, we prepare the dataset for machine learning by separating the input features (X) from the target variable (y). To create the input features matrix (X), we drop unnecessary columns from the dataset. These columns include 'datetime', which has already been used to extract time-related features, and 'count', which is the target variable. Additionally, we drop the 'casual' and 'registered' columns, as they provide a direct breakdown of the 'count' column and are not available in the test set. By removing these columns, we ensure that our model relies only on the relevant features for predicting bike demand.For the target variable (y),

we use the 'count' column from the dataset, which represents the total number of bike rentals in a given hour. However, instead of using the raw count values, we apply a logarithmic transformation by adding 1 to the count values and then taking the natural logarithm using NumPy's 'log1p()' function. This transformation helps in mitigating the effects of outliers and scaling the target variable to a more manageable range. It also aligns with the evaluation metric of the competition, which is the root mean squared logarithmic error (RMSLE).

Define Numeric Columns for Scaling

In this step of the code, we define a list of numeric columns that require scaling before feeding them into the machine learning model. The numeric columns include 'temp', 'atemp', 'humidity', 'windspeed', 'hour', 'day', 'day_of_week', 'month', 'hour_workingday', 'hour_temp', and 'hour_humidity'. These columns represent a mix of original features from the dataset (e.g., 'temp', 'humidity', 'windspeed') and the engineered time-related and interaction features (e.g., 'hour', 'day', 'day_of_week', 'month', 'hour_workingday', 'hour_temp', 'hour_humidity').

Scaling the numeric features is an essential preprocessing step, as it ensures that all features are on a comparable scale, preventing any single feature from dominating the model due to its larger range of values. This is particularly important for algorithms like XGBoost that are sensitive to the scale of input features. By defining this list of numeric columns, we can subsequently apply the desired scaling technique, such as MinMax scaling, to normalize the range of these features and improve the model's performance.

Scale Numeric Columns

In this portion of the code, we scale the numeric columns that we previously defined in the `numeric_columns` list. To do this, we use the `MinMaxScaler` from the scikit-learn library, which is a popular method for feature scaling. MinMax scaling scales the numeric features by transforming them into a specific range, typically [0, 1]. This is achieved by subtracting the minimum value of the feature and then dividing by the range of the feature (maximum value minus minimum value).We first instantiate the MinMaxScaler by creating a variable named `scaler`. Next, we apply the MinMax scaling to the specified numeric columns in our input features matrix (X) using the `fit_transform()` method. This method computes the minimum and maximum values of each feature in the dataset, fits the scaler to these values, and then scales the features by transforming them into the desired range. The transformed values replace the original values in the `X` matrix.By scaling the numeric columns in this way, we ensure that all features have a comparable range of values, allowing the XGBoost algorithm to better capture their contributions to the target variable without any single feature dominating due to its larger scale.

Train-Test Split

In this part of the code, we perform a train-test split on our preprocessed dataset to create separate training and testing sets for our model. We use the `train_test_split` function from the scikit-learn library to divide the input features matrix (X) and the target variable (y) into

respective training and testing sets. The `test_size` parameter is set to 0.2, meaning that 20% of the dataset is reserved for testing purposes, while the remaining 80% is used for training. By specifying the `random_state` parameter as 42, we ensure that the data splitting process is reproducible, allowing for consistent results across different runs.The train-test split is a crucial step in evaluating the performance of our model. It allows us to train our model on one subset of the data (training set) and assess its performance on another, unseen subset (testing set). This helps us gain insights into the model's generalization ability and detect potential overfitting or underfitting issues.Additionally, we define a variable `n_folds` and set its value to 5. This variable represents the number of folds we will use in the KFold cross-validation process, which is a technique for assessing the performance of our model more robustly by partitioning the training set into multiple subsets (folds) and iteratively training and validating the model on these subsets. By using 5-fold cross-validation, we aim to obtain a more reliable estimate of our model's performance and reduce the impact of potential data imbalances or biases in the train-test split.

Define Objective Function for Optuna

In this code snippet, we define the objective function for Optuna, a hyperparameter optimization library. The objective function, named `objective`, takes a trial object as input and uses it to suggest the hyperparameters for the XGBoost model. Optuna iteratively explores the hyperparameter search space by evaluating the model's performance with different hyperparameter combinations, aiming to find the optimal set of hyperparameters that minimizes the objective function.Within the `objective` function, we define a dictionary named `params` that stores the hyperparameters suggested by Optuna for the current trial. The `trial.suggest_*` methods are used to sample hyperparameters from specific ranges or distributions. For instance, 'n_estimators' is sampled from integers in the range of 100 to 1000 with a step size of 100, and 'learning_rate' is sampled from a logarithmic scale between 0.001 and 0.1. Other hyperparameters, such as 'max_depth', 'min_child_weight', 'subsample', 'colsample_bytree', 'gamma', 'reg_alpha', and 'reg_lambda', are also sampled using their respective sampling methods. Once the hyperparameters are suggested, we create an instance of the XGBRegressor model with the sampled hyperparameters, the 'reg:squarederror' objective, and additional parameters 'n_jobs' and 'random_state' set to -1 and 42, respectively. The 'n_jobs' parameter is set to -1 to allow the model to use all available CPU cores for parallel computation, while the 'random_state' parameter ensures reproducibility of the results.

After the model is initialized with the suggested hyperparameters, it will be trained and evaluated using KFold cross-validation, which is performed in the subsequent lines of the `objective` function (not shown in the code snippet). The objective function will return the mean RMSLE score across all folds, which Optuna will use to guide the search for the optimal hyperparameters.

Initialize K-Fold cross validation

In this step of the code, we initialize the KFold cross-validation process by creating an instance of the KFold class from the scikit-learn library. Cross-validation is a technique used to assess

the performance of a machine learning model more robustly, by iteratively training and validating the model on different subsets (folds) of the training data. This helps to obtain a more reliable estimate of the model's performance and reduces the impact of potential data imbalances or biases in the train-test split.The KFold class is instantiated with three parameters: n_splits, shuffle, and random_state. The n_splits parameter, which is set to the previously defined variable n_folds, determines the number of folds (subsets) the training data will be divided into. In this case, we use 5-fold cross-validation, meaning that the training data will be split into 5 equally-sized folds. The shuffle parameter, set to True, ensures that the data is randomly shuffled before splitting it into folds, which helps to maintain an even distribution of the target variable across the folds. Lastly, the random_state parameter is set to 42, ensuring that the random shuffling process is reproducible, allowing for consistent results across different runs of the code.

Perform K-Fold cross validation

In this code snippet, the KFold cross-validation process is performed on the training data using the previously initialized `kf` object. We first create an empty list named `rmsle_scores` to store the Root Mean Squared Logarithmic Error (RMSLE) scores for each fold of the cross-validation.

The for loop iterates over each combination of training and validation indices provided by the `kf.split(X_train)` method. For each iteration, the training data (`X_train`) and target labels (`y_train`) are split into training and validation folds using the current set of indices. This creates the `X_train_fold`, `X_val_fold`, `y_train_fold`, and `y_val_fold` subsets for the current fold.Next, the XGBoost model (`xgb_model`) is trained on the `X_train_fold` and `y_train_fold` subsets using the `fit` method. Once the model is trained, predictions are made on the `X_val_fold` subset using the `predict` method, resulting in the `y_val_pred` array.

Since the target labels were log-transformed earlier in the pipeline, we need to revert this transformation to obtain the actual bike count predictions. We apply the inverse of the log1p transformation, which is the exponential function minus 1 (`expm1`), to both `y_val_fold` and `y_val_pred`, resulting in the `y_val_exp` and `y_val_pred_exp` arrays, respectively.With the actual and predicted bike counts in their original scale, we calculate the RMSLE score for the current validation fold using the `mean_squared_log_error` function from scikit-learn, followed by taking the square root of the result. The calculated RMSLE score for the current fold is then appended to the `rmsle_scores` list.This cross-validation process is repeated for all folds, resulting in a list of RMSLE scores for each fold. These scores will be used to calculate the mean RMSLE across all folds, which serves as the objective value for the Optuna. hyperparameter optimization process. After performing the KFold cross-validation for each trial, we obtain a list of RMSLE scores for each fold, which are stored in the `rmsle_scores` list. We then calculate the mean RMSLE across all folds using the `np.mean` function and assign it to the `mean_rmsle` variable.Finally, the mean RMSLE value is returned from the `objective` function, which will be used by Optuna to evaluate the performance of each combination of hyperparameters that it suggests. The goal of Optuna is to minimize this RMSLE value by finding the optimal set of hyperparameters.

Initialize the Optuna Study

This code initializes an Optuna study using the optuna.create_study function. The direction parameter is set to 'minimize', indicating that we want to minimize the objective function value. The sampler parameter is set to TPESampler, which is a Tree-structured Parzen Estimator sampler that is commonly used for hyperparameter optimization tasks. After initializing the study, the study.optimize method is called to start the hyperparameter optimization process. The objective function is passed as the first argument to this method, indicating that this function will be optimized. The n_trials parameter is set to 100, which specifies the maximum number of trials (i.e., evaluations of the objective function) that the optimization process will perform.During the optimization process, Optuna will suggest a set of hyperparameters and evaluate their performance by calling the objective function. Based on the RMSLE score returned by the objective function, Optuna will update its internal model and suggest a new set of hyperparameters to evaluate in the next trial. This process continues until the specified number of trials (n_trials) is reached. At the end of the optimization process, Optuna will return the set of hyperparameters that produced the best RMSLE score found during the search. This set of hyperparameters will be used to train the final model on the entire training dataset.

The Best Hyperparameters

After the hyperparameter optimization process is completed, Optuna returns the best set of hyperparameters that produced the lowest RMSLE score during the search. The `study.best_params` attribute is used to obtain this set of hyperparameters, which is stored in the `best_params` variable. The `best_params` variable is then printed to the console using the `print` function, which outputs the best hyperparameters that were found during the hyperparameter optimization process. This information is useful for understanding which hyperparameters contributed the most to the final model's performance and can be used for future experiments or to fine-tune the model further.

Train the model with the best hyperparameters.

After the best hyperparameters have been obtained from the hyperparameter optimization process, the final XGBoost model is trained using these hyperparameters. The `XGBRegressor` class is used to initialize the model, with the best hyperparameters passed as keyword arguments using the `**` syntax.The `objective` parameter is set to `'reg:squarederror'`, indicating that the model is trained using mean squared error as the loss function. The `n_jobs` parameter is set to `-1`, which means that all available CPU cores are used to parallelize the training process. The `random_state` parameter is set to `42`, which ensures that the results are reproducible.Once the model is initialized, the `fit` method is called on the training dataset (`X_train` and `y_train`) to train the model with the best hyperparameters. The resulting `best_xgb_model` can then be used to make predictions on new, unseen data.

Analyze the feature importances

After training the final XGBoost model, the feature importances are analyzed to gain insights into which features have the most significant impact on the model's predictions. The `feature_importances_` attribute of the `best_xgb_model` object is used to extract the importance scores for each feature.The feature importances are then stored in a Pandas Series object called `feature_importances` with the feature names set as the index. The `sort_values` method is used to sort the features by importance in descending order, resulting in `feature_importances_sorted`.The resulting `feature_importances_sorted` can be printed or visualized to gain insights into which features are most important for predicting bike-sharing demand. This information can be used to inform further feature engineering or to help stakeholders understand which factors are most influential in determining bike-sharing usage.Print the sorted feature importances and Print the best RMSLE score. After analyzing the feature importances, the sorted feature importances (`feature_importances_sorted`) are printed using the `print` function. Next, the best RMSLE score is printed using the `study.best_value` attribute of the Optuna `study` object. The `best_value` attribute returns the best score obtained during the hyperparameter tuning process. The RMSLE is a measure of the accuracy of the model, and a lower value indicates a better-performing model. The RMSLE score is printed along with the number of folds used for cross-validation (`n_folds`) using the `print` function.

Working on the Test Set

Next, we read in the test dataset using the `pd.read_csv()` function and assign it to the `test_df` DataFrame. The `test.csv` file contains the same features as the `train.csv` file, except for the `count`, `casual`, and `registered` columns, which are the target variables we are trying to predict. Next, the `test_df['datetime']` column is converted to a datetime format using the `pd.to_datetime()` function, as was done with the `train.csv` file. Time-related features, including `hour`, `day`, `day_of_week`, `month`, and `year`, are then extracted from the datetime format using the `dt` accessor in pandas. This step is necessary to create the same set of features in the test set as in the training set, which is required for the model to make accurate predictions on the test set.

Make predictions on the test dataset using the best XGBoost model from Optuna

Next, we use the `predict` method of the best XGBoost model obtained from the Optuna hyperparameter search to make predictions on the test dataset. The input to the `predict` method is the preprocessed test data `X_test`. The predicted values are stored in the variable `y_test_pred`. These predictions represent the estimated number of bike rentals for each row in the test dataset. The purpose of this step is to evaluate the performance of the model on unseen data, which can help to determine if the model is overfitting to the training data.

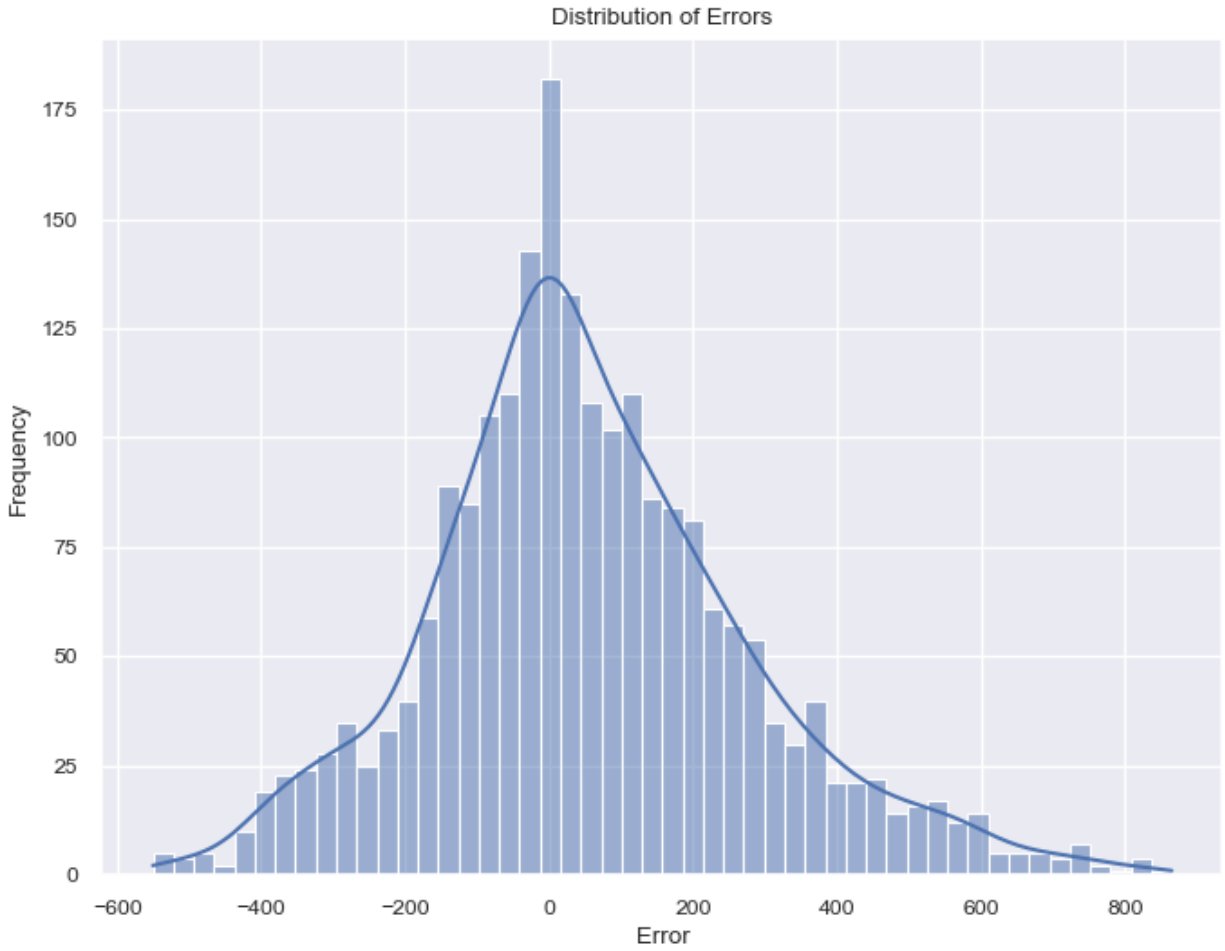
Create a Submission DataFrame

We create a Pandas DataFrame named `submission` with two columns: `datetime` and `count`. The `datetime` column is obtained from the `test_df` DataFrame, which contains the timestamps of the test set. The `count` column is obtained from the predictions made on the test set using the best XGBoost model from Optuna. The `y_test_pred_exp` array contains the predicted bike rental counts in the original scale, so there is no need to perform any additional transformations. Finally, the `submission` DataFrame is saved to a CSV file named `submission.csv` without including the index column. This file can be used to submit the predictions to the Kaggle competition website for evaluation.

Results:

The results from the XGBoost model didn't show as much promise as we had hoped, resulting in a final RMSLE score of .42896 (this number is reported from the Kaggle submission, since we were unable to score the model ourselves, the final count was removed from the Test dataset provided). This number indicates only moderate, if not mediocre, prediction power. Since we were unable to score our model on the test data, instead we created a test-train split of the training dataset provided, trained the model with the suggested hyperparameters from before, and tested our model on our mini "test set". The graph below shows the actual vs predicted

values on our internal test set.



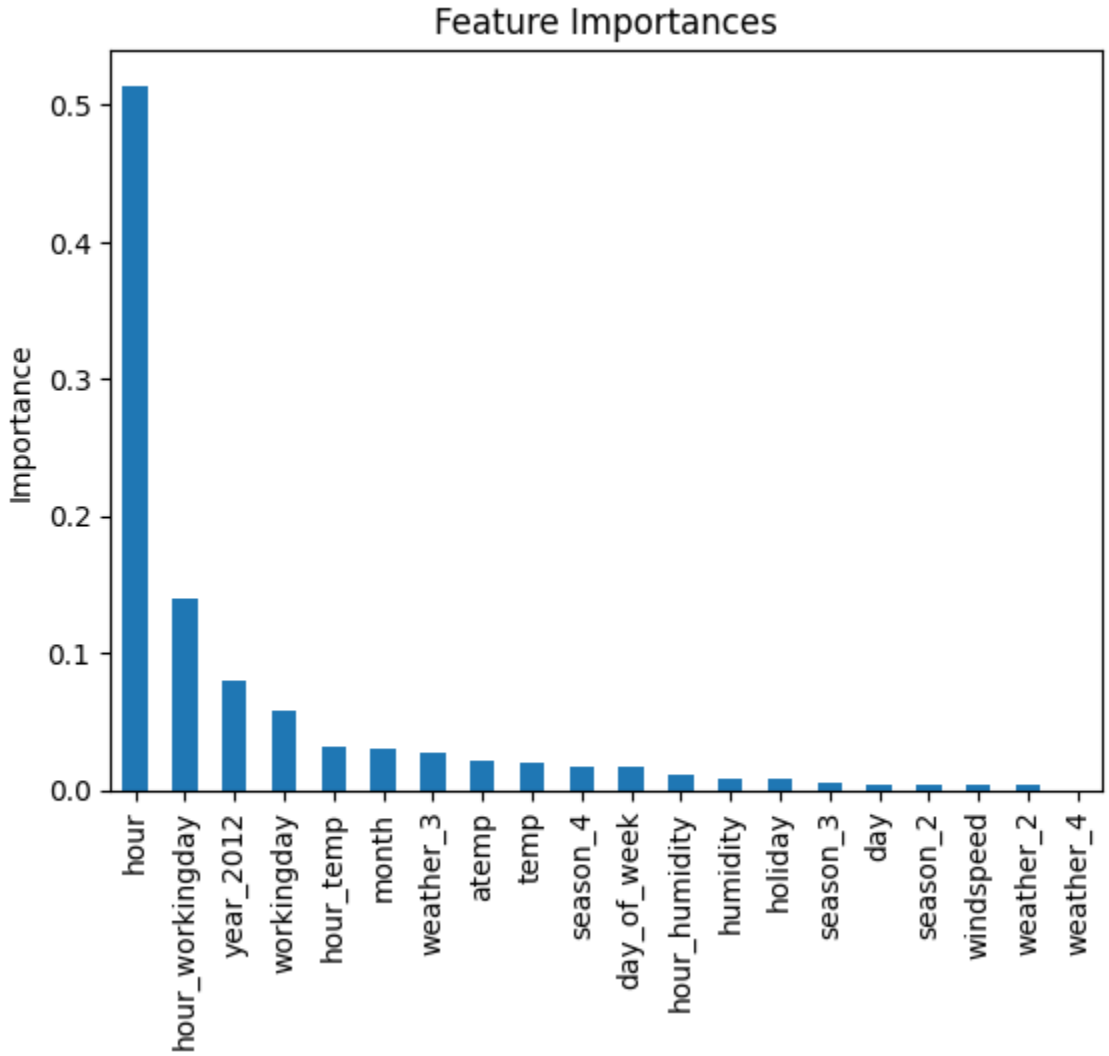
As you can see, the error amounts ranged from -600 to 800, though a majority of the frequency appears to be concentrated between -200 and +200. The maximum actual value in the set was ~1000, making these errors +/-20% of the maximum value.

Improvement

With such a lackluster performance from a highly touted model, we must ask ourselves what we could have done better. First of all, at the start of the experiment, we were using an online ipbny notebook editor that was collaborative called DeepNote. However, because this site is relatively lightly used, the processing power it provides was limited by the 2GB of ram (Compared to the 12GB of RAM provided by Kaggle). This hampered our earliest efforts to compile a comprehensive model, as many opportunities were lost due to kernel interrupt when we ran out of RAM.

We also didn't do a very aggressive feature engineering step, and in fact added more columns when we parsed the date-time string used by the original dataset. The reason for this was the relatively small range of a majority of the features. Most features had a feature importance value

that was between 0 and .1, so with no clear losers, we decided to try our hand with nearly the whole feature set. Perhaps we could have had an aggressive Recursive Feature Elimination strategy [4] that would have been able to tease out the correlations and remove some of the features that didn't contribute to the model.



Also, In hindsight, features like temperature and hour of day have intrinsic relationships that, while difficult to tease apart, could result in heavier cumulative inherent weighting of these features. This is further compounded by the inherent correlation between temperature and season, actual temp (atemp in the dataset), and variation with seasons. We could have tried to remove some of these related features to allow a more "clear" feature space [5][6].

We decided that, to bolster our attempts to place highly in the competition, we should use the "latest greatest" technology that we hadn't explicitly covered in our Machine Learning class. This may have led us down the wrong path, as we may not have fully understood the ways that we could have improved the model in nearly all the previous steps. A better strategy would have

been choosing a relatively simple model like a basic Linear Regression model, and adding more layers to the model, or related models with just a degree more complexity.

However, this experiment was still rewarding because we were able to see how a complex ensemble strategy like XGBoost can go off the rails, and the lack of insight provided can be cumbersome. We also learned that feature engineering is especially important with this modeling technique [1][3], which explains why the final model presented wasn't top notch with the small amount of feature engineering we did.

References

1. J. Chen and T. Guestrin, “XGBoost: A Scalable Tree Boosting System,” in Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, Aug. 2016, pp. 785–794, doi: 10.1145/2939672.2939785.
2. K. Takahashi, T. Moriya, S. Sakamoto, and T. Hara, "Optuna: A Next-Generation Hyperparameter Optimization Framework," in Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD ’19), Aug. 2019, pp. 2623–2631, doi: 10.1145/3292500.3330701.
3. M. Lin, Q. Chen, and S. Yan, "Network traffic classification based on XGBoost with optimized feature engineering," Journal of Physics: Conference Series, vol. 1492, no. 1, p. 012089, Apr. 2020, doi: 10.1088/1742-6596/1492/1/012089.
4. Y. Xie, Y. Jiang and X. Zhang, "Traffic Flow Prediction with Deep Learning and Feature Selection," 2018 15th International Conference on Service Systems and Service Management (ICSSSM), 2018, pp. 1-6, doi: 10.1109/ICSSSM.2018.8465954.
5. S. Hong and S. Lee, "Efficient feature selection using correlation-based feature weighting," in IEEE Transactions on Knowledge and Data Engineering, vol. 21, no. 6, pp. 800-14, June 2009, doi: 10.1109/TKDE.2008.141.
6. N. Bhandari, A. Gupta, and H. Ranganathan, “Empirical comparison of feature selection techniques with XGBoost for credit risk assessment,” 2019 IEEE International Conference on Big Data (Big Data), Los Angeles, CA, USA, 2019, pp. 3279-3288, doi: 10.1109/BigData47090.2019.9005951.