**Introduction**

This problem revolves around predicting the number of rented bikes from a bike rental shop using data which includes climate conditions and time information. Since the target variable, the count of rented bikes, is a continuous variable, and we are interested in making a prediction in a continuous state space, we will be looking at regression machine learning algorithms to solve the problem.

**Methods**

To start, we use the Linear Regressor algorithm as a baseline model to compare the results against. This is because Linear Regression is a robust, but simple algorithm, so we would expect it to perform at a good level without any configurations. This is preferable to sklearn’s Dummy Regressor since the Linear Regression algorithm is likely to produce a viable model to predict the count of rental bikes. Thus, should provide suitable competition to any proposed model.

To evaluate the models, we will be using the as the performance metric. The of a model is defined as the proportion of variance in the dependent variable that we can predict from the independent variables. Mathematically, for observation , prediction and mean observed value, we have that

The benefit of using this metric is that, unlike Root Mean Squared Error (RMSE), it is not attached to any units, thus we can get a clear view of the model performance without needing to give context to a figure. However, a downside of the that the figure does not provide us with any more information about the residuals since it is detached from the predictions, so it is important to be careful of an overfitted model.

To choose the hyperparameters, we will be using the Optuna framework to assist us with optimisation. Optuna is a Bayesian approach to finding the optimal parameters which deals with the search problem probabilistically. It uses a Tree-structured Parzen Estimator which sequentially builds models to estimate hyperparameter performance based on past trials, and thus selects new hyperparameters to evaluate. What this does is provide a search algorithm that performs better than the more commonly used optimisers such as a Randomised Search or Grid Search in a clean structure.

Hyperparameters are algorithm parameters that can be used to adjust the learning process in a variety of ways. They can be split into two types of hyperparameters: ones to control the model complexity and ones to control the regularisation of the model. To effectively tune these hyperparameters, assess performance and determine robustness, we split the data using a 60/20/20 split to use for training, hyperparameter validation and performance testing.

**Baseline Results**

Chart, bar chart

Description automatically generatedAs displayed in Figure 1, we found that the baseline produced the following for the training and validation datasets respectively: 0.6847 and 0.6595. These scores are close, suggesting we have avoided mass-overfitting, and show a suitably large level of performance. We will be utilising the XGBoost and Random Forest algorithms to model the problem. To start, we fitted each model to the training set without any hyperparameter adjustments. Since the evaluation of these models will give us an indication of what hyperparameters we should be adjusting to get the best out of the model. An overfit model would mean we would want to reduce the model complexity and increase regularisation with the opposite required for an underfit model.

As you can see in Figure 1 both the XGBoost and Random Forest models perform at a higher level than the baseline. However, this plot does show that we have overfitted the models since there is a great difference in between the train and validation datasets in both cases. Thus, we need to increase the regularisation and reduce the model complexity.

**Hyperparameter Tuning**

We created an OptunaSearchCV instance for each model to help tune the hyperparameters. OptunaSearchCV includes a cross-validation component in which the training data was randomly split into ten folds to improve the robustness of the model. For each search, we performed 200 trials whereby each trial would select a value for each hyperparameter from the given options, train the model using these hyperparameters – with the cross-validation element – and produce the mean value of the cross-validation scores. OptunaSearchCV uses these scores and the Tree-structured Parzen Estimator sampler to finalise the maximal score.

**XGBoost**

With regards to hyperparameters, we start with the XGBoost model. Two intuitive ways to control model complexity were min\_child\_weight, to control how finely the data is split for each leaf, and n\_estimators, the number of decision trees generated in the boosting process. We decided to tune n\_estimators as opposed to using the early\_stopping\_rounds hyperparameter in XGBoost’s *fit* method to maintain consistency across the cross-validation folds. We also chose to tune subsample and colsample\_bytree to introduce a stronger element of randomness into training. Subsample denotes the proportion of observations to be randomly sampled for each tree and colsample\_bytree is similar, but it indicates the fraction of columns to be randomly sampled for each tree. This was done so that the training data was more representative of the entire population such that we do not draw conclusions specific to the training data. Moreover, we chose three regularisation hyperparameters: gamma, reg\_lambda and reg\_alpha. Each hyperparameter regularises the model at a different stage. Gamma controls the complexity of the gradient boosting; reg\_alpha smooths the prediction line by penalising the features that cause the loss function to increase (L1 regularisation); and reg\_lambda, L2 regularisation, regularises the weights of the features. Finally, we also looked at tuning the learning\_rate of the model since a high rate will cause the model the take larger steps in the training’s optimisation phases and thus avoid overfitting.

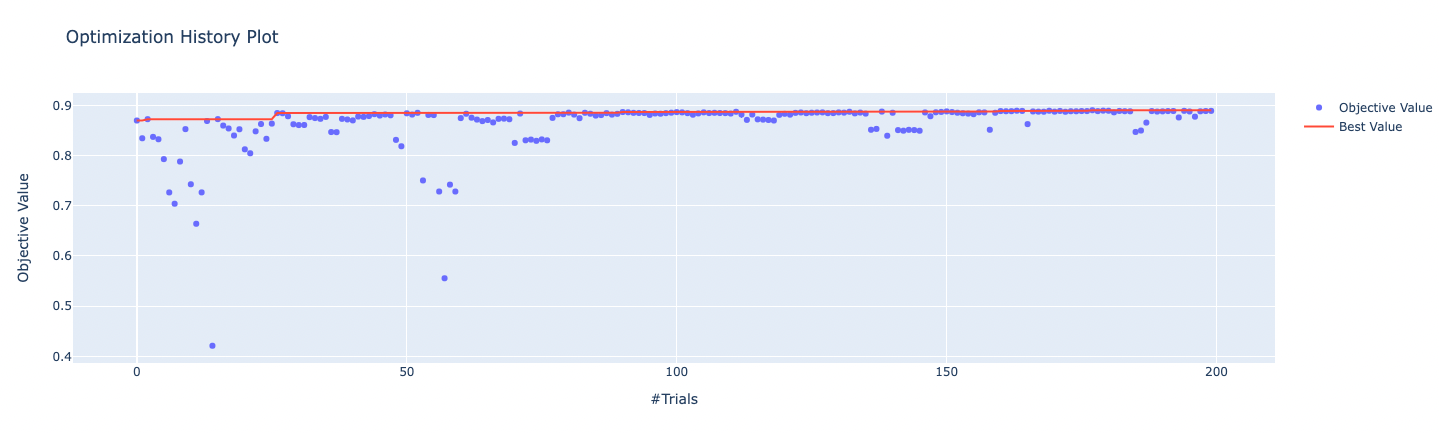
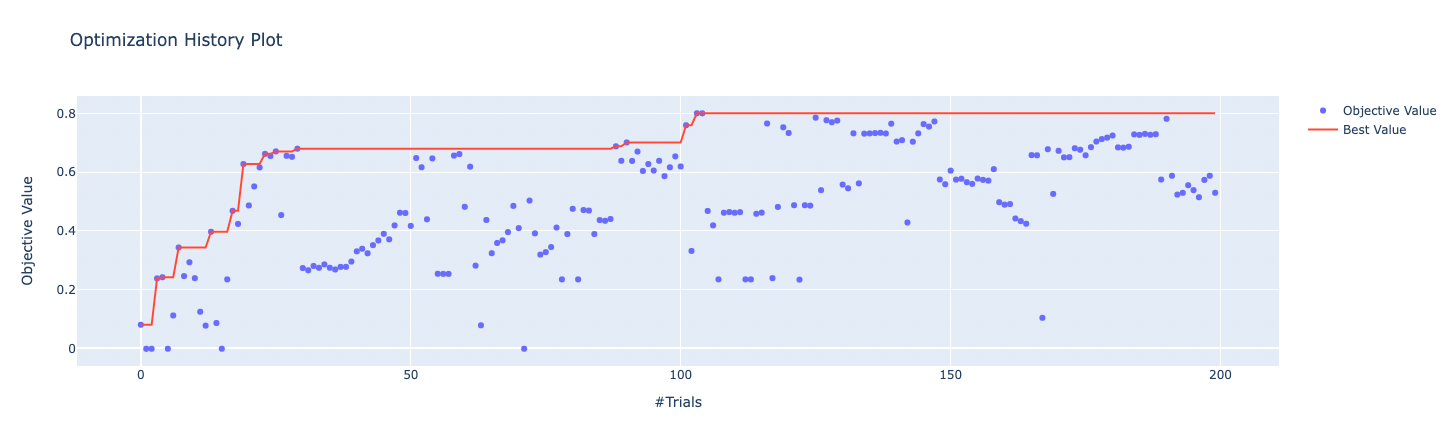


Figure 2



After 200 trials, the optimised hyperparameters produced an XGBoost model which was evaluated at the following for train and validation respectively: 0.9852 and 0.8782. As you can see from Figure 2, the search algorithm found the area in which the optimal values lay very quickly – note that the objective value is the .Figure 3 shows the hyperparameters trialled by Optuna with darker coloured points showing us the later trials. As we can see, a lot of hyperparameters were tested, but the algorithm hones into the optimal areas very quickly. The max\_depth is a notable graph since later trials do seem to be spread a little more evenly across the values. We can use Figures 2 and 3 in conjunction to see how the performance changed with different suggestions of hyperparameters – the change in performance of trials 70-76 is due to the search algorithm exploring a max\_depth of two instead of five.

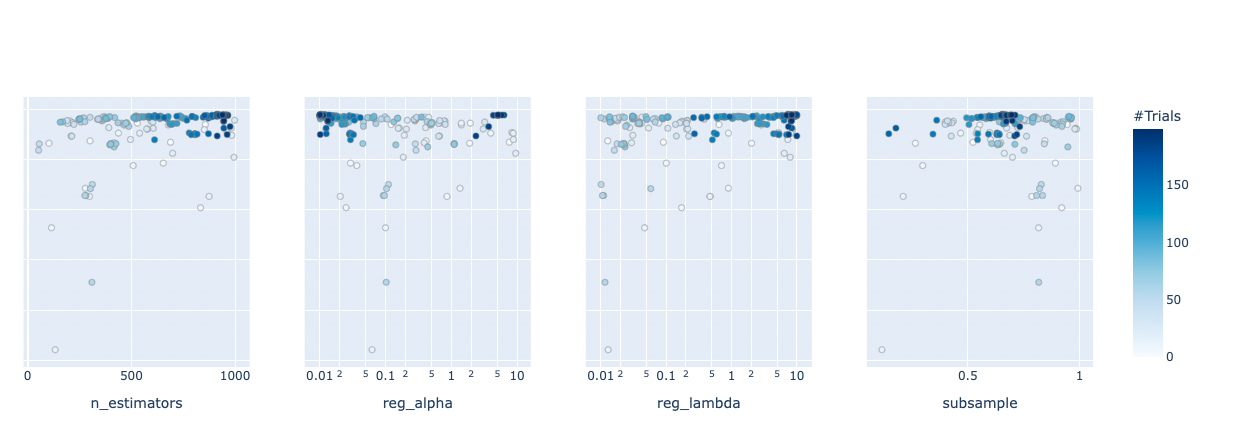
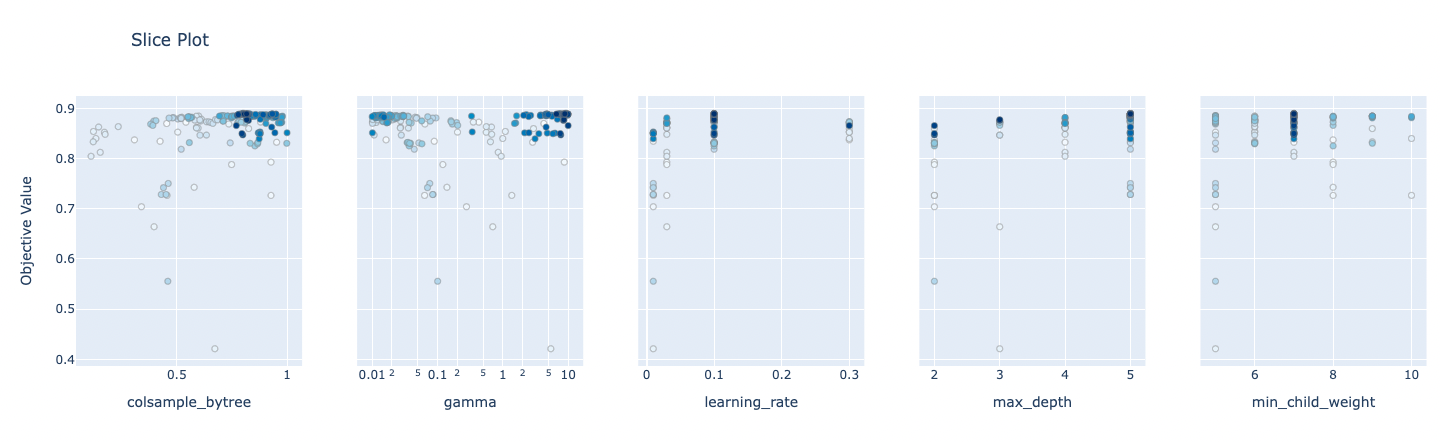
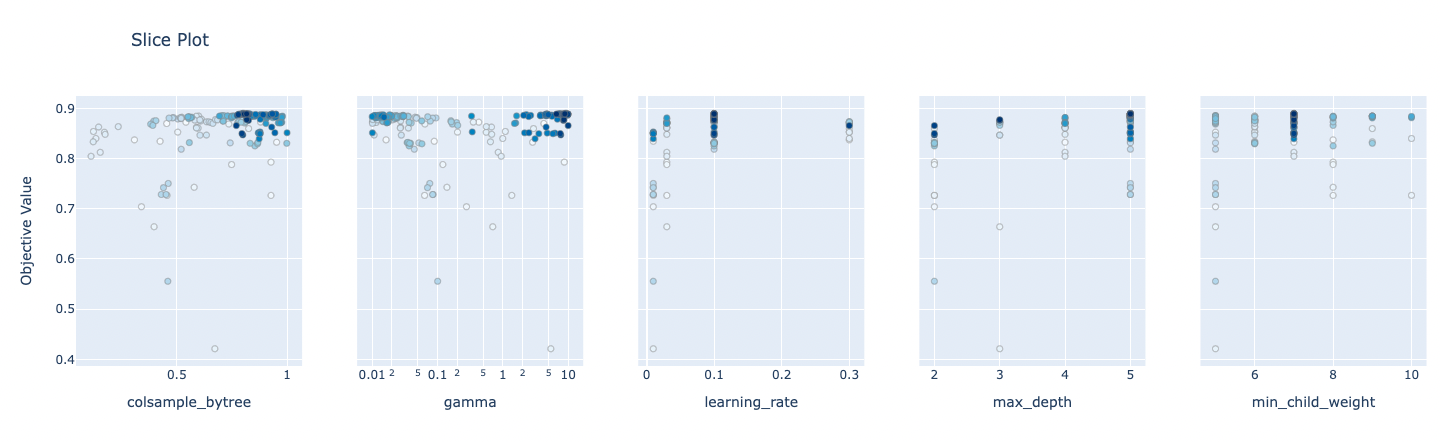


Figure 3



Moreover, Figure 4 shows us which hyperparameters had the greatest impact on the fit of the model. We can see subsample and learning\_rate are frontrunners, but max\_depth and n\_estimators also show to have a large impact on results. The values suggest a better model than the baseline, but the model is still overfitting.

Thus, we undertake some manual adjustments to max\_depth and learning\_rate to reduce the model complexity – these hyperparameters were chosen due to the impact shown in Figure 4 and that they are intuitive complexity-reduction hyperparameters as they reduce the size of the model. We did consider changing the subsample hyperparameter, but it did not yield any strong results. Figure 5 shows the values that the manual adjustments yielded. Considering both performance and fit, we decided max\_depth = 3 and learning\_rate = 0.03 were the most appropriate values since they are close together, but not at a large detriment to performance.

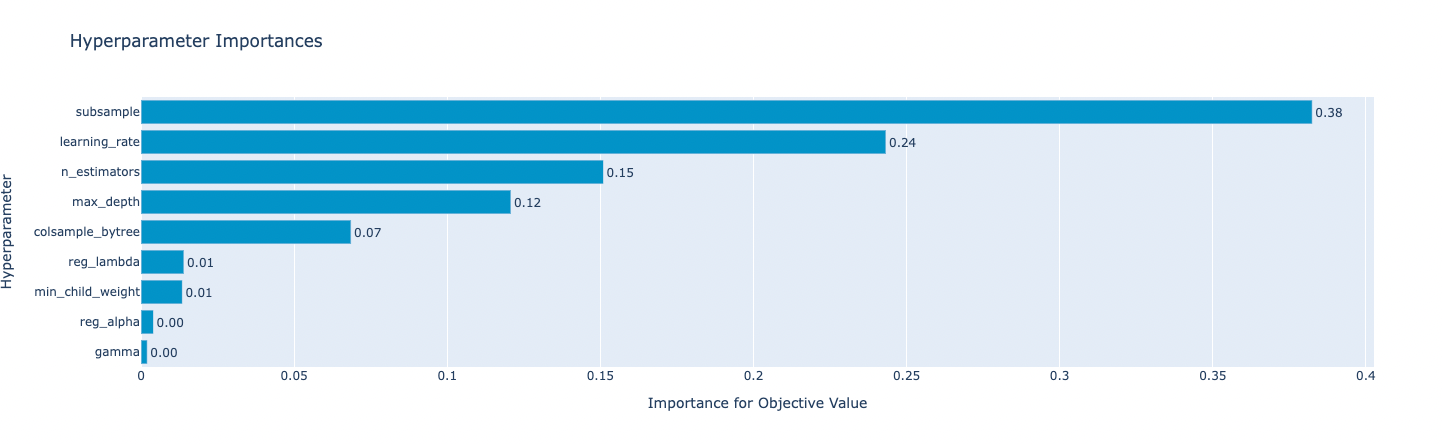


Figure 4

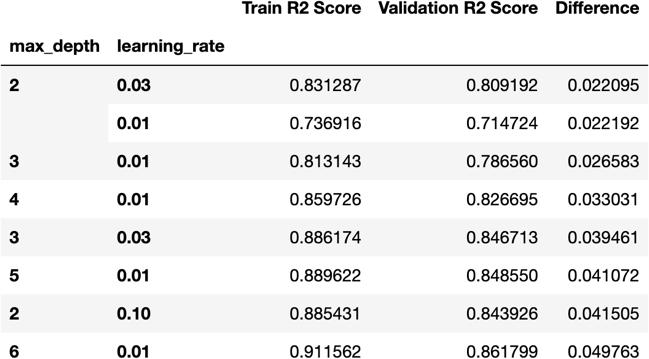


Figure 5

**Random Forest**

With regards to the chosen hyperparameters for the Random Forest model, we also selected max\_depth and n\_estimators – both denoting the same aspect of the model as with XGBoost. However, we also introduced min\_samples\_split and min\_samples\_leaf. Min\_samples\_split controls the minimum number of samples needed for a node to be split and min\_samples\_leaf works in a similar way to XGBoost’s min\_child\_weight whereby it states the minimum number of samples on a leaf node. Random Forest specific hyperparameters we also chose to tune were max\_features, the number of features to use when looking for an optimal split, and bootstrap hyperparameter. This is used to determine if we want to use bootstrap samples to build decision trees which is useful for the model only being representative of the training data.

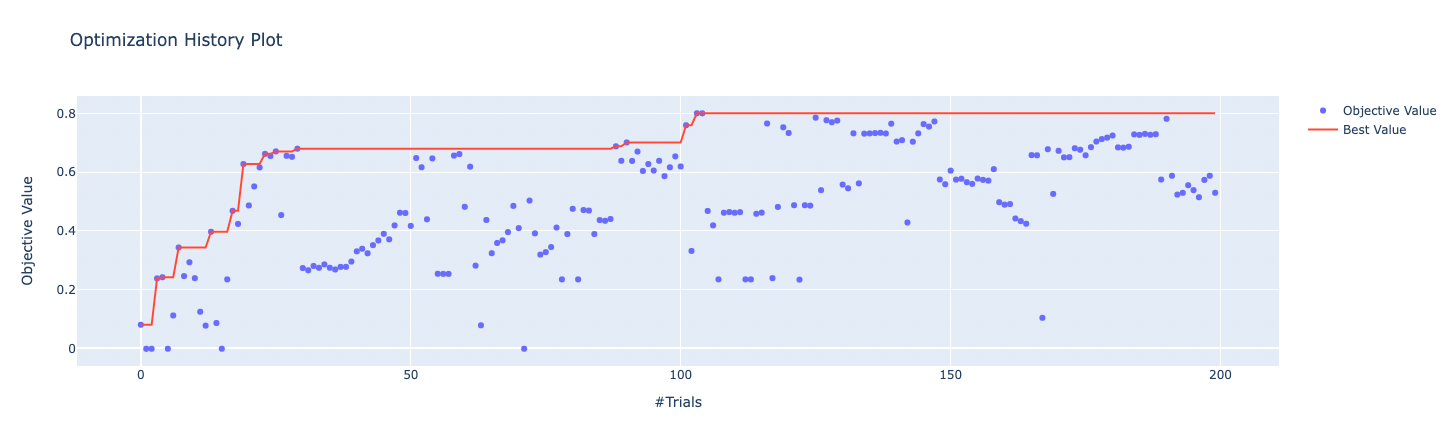
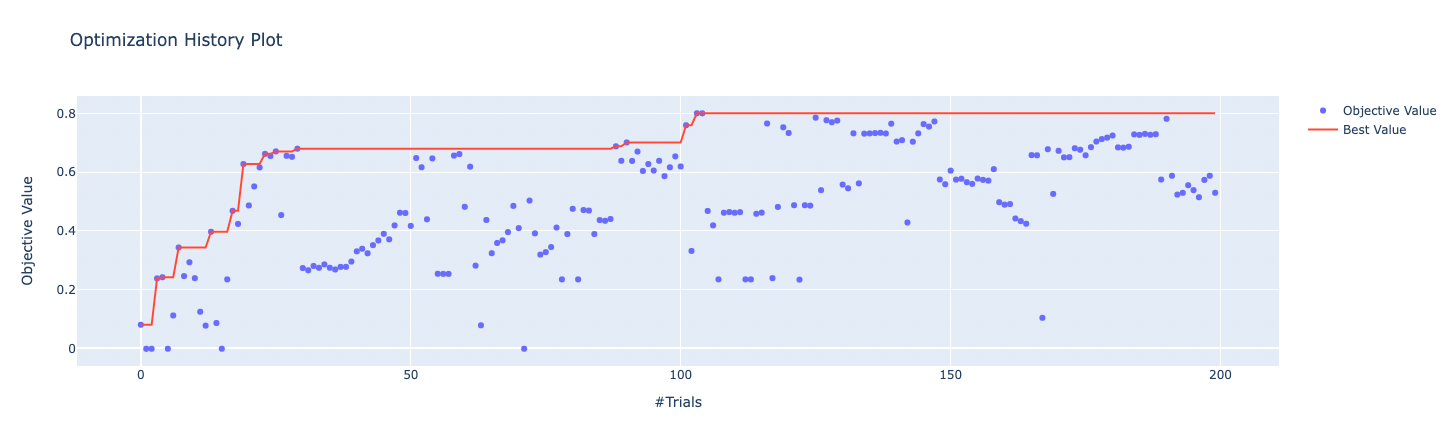


Figure 6



With the same cross-validation setup, 200 trials tuned the Random Forest model to result in for train and validation respectively: 0.8620 and 0.7909. Figure 6 shows us the optimisation history of the hyperparameter search for this algorithm. We see that the search algorithm took more trials to find the optimal space of hyperparameters with great progress made in trial 101 where the search algorithm found max\_depth = 15 boosted the performance of the algorithm. Figure 7, a snapshot of max\_depth trials, supports this since later trials are either 14 or 15. We also found that min\_samples\_leaf was the more influential hyperparameter on the performance of the model. However, when it came to manual adjustments to correct the minor overfitting, we did not see much change in the results.

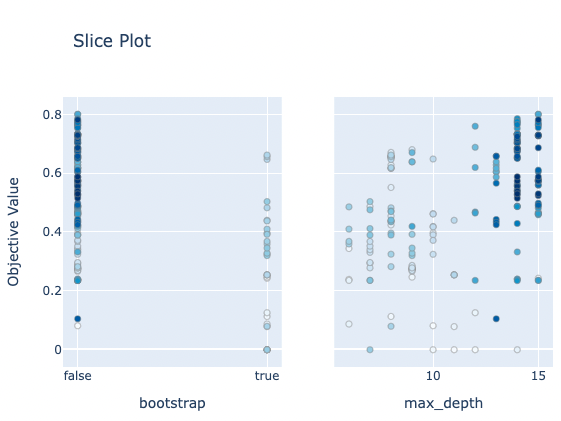
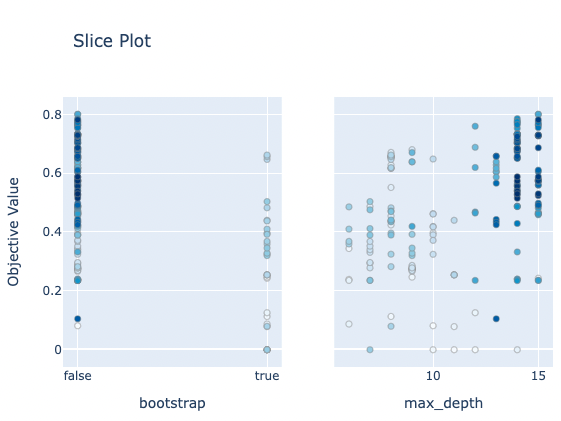


Figure 7

Snapshot

**Results**

Chart, bar chart

Description automatically generatedTo compare the two models fairly, we use the test dataset. This is so that we can be confident that the model we choose is robust enough to generalise to unseen data with a high-performance level. The test dataset qualifies for the unseen data as it was split from the training data before we started modelling and so will display the qualities of each model on unseen data. Figure 8 shows us the performance metrics for each model on the testing dataset. We can see the XGBoost model performs with the highest scores on the unseen dataset with a score of 0.8545 in comparison to the Random Forest and baseline models evaluating to 0.7927 and 0.6664 respectively. We choose the XGBoost model since it performs at a high level on unseen data and this value is close to the performance score when evaluated on the test data. This shows us that the chosen model generalises well to unseen data and predicts the number of rental bikes out each hour with reliable performance.