In this coursework, we were provided with data (coursework_other.csv), which consisted of the number of rental bikes out for rent at each hour under the "Rented Bike count" column. My first aim is to understand the data given to us and perform some pre-processing functions. I have loaded the dataset into my jupyter notebook in the form of a data frame using the help of a library called "Pandas". Pandas is a valuable tool used for python data analysis and data manipulation.

1. Understanding the data:

After loading the data, my first step was to determine the magnitude of the correlation between the columns. It was clear that our target column's values were varying from other columns, so I tried to find out their relationships by plotting different types of graphs. The following pattern was noticed:

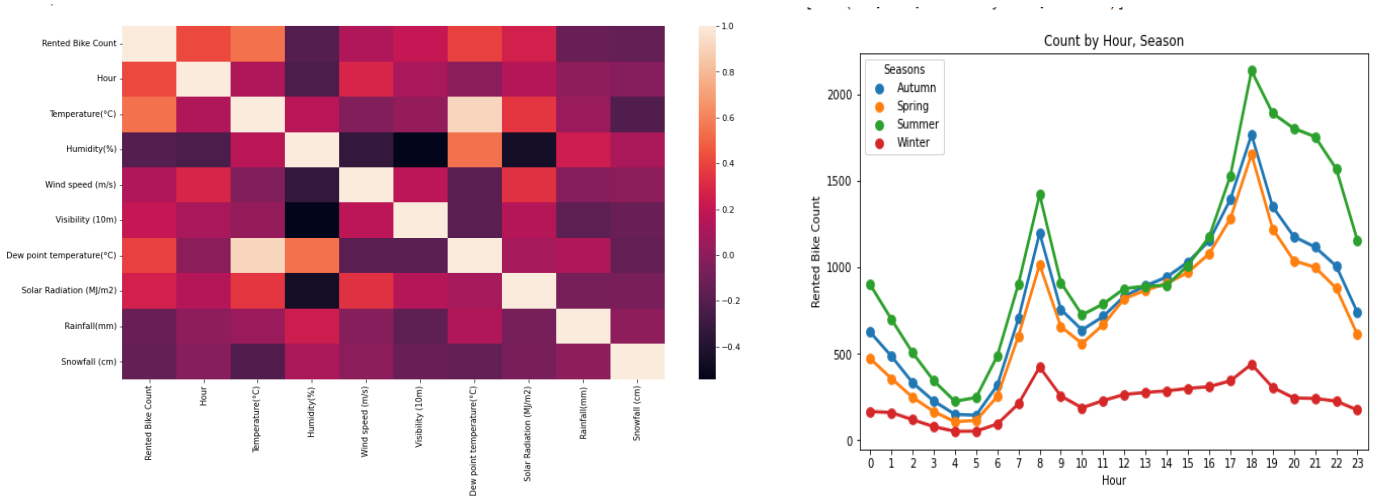

Fig: Confusion matrix for correlation and plot to understand the relationship between the frequency of bike count and seasons.

The lighter the colour between the two columns, the stronger their correlation. By looking at this, I realised that there is much correlation between our target column, i.e., the "Rented Bike Count", and the other columns.

Q) The kind of algorithm to use (e.g., classification/regression/clustering)?

Our target column, i.e., the "Rented Bike Count", came under regression problem. The Rented Bike count depended on other "independent" variables like 'Hour', 'Temperature', 'Season' etc. Hence by standard definition, predicting the Rented Bike count comes under regression.

Before going any further to train our data, all our data should be numerical, but the challenge faced was that the "Seasons" were in the categorical format. However, a critical column affected our target column, as shown in the above graph. Other woo columns named "Holiday" and "Functioning Day" were in categorical format, and they did seem to affect our target column in some way. Hence, we can overcome this challenge by replacing the absolute values using label encoding. All my data required for training was in numerical value doing this.

2. Train Test Split:

We first divide our data into train and test sets in an 80:20 ratio for our train test split. The same setting is further done in our train test, divided into train and validation sets. The reason for doing this was to make sure our best model performs well with unseen data(test data) and does not overfits our trained data.

3. Implementation of models:

3.1 Sklearn.DummyRegressor(Baseline model)

I have decided to choose a dummy regressor as my baseline model as it was suggested. I trained my model using the dummy regressor with the strategy parameter set as "Mean". This means that the value of our target column will be predicted based on the mean of the target column present in our training set.

Q) The metric to use to measure the performance of the model

For measuring the performance of all the models, we will be implementing, I have decided to go with 'r2_score'. 'r2_score' is also known as the coefficient of determination. Its mathematical formula is:

$$R^2=1-(RSS/TSS)$$

RSS=sum of the square of residuals. TSS=total sum of

Unlike most other scores, r2_score can be damaging, and by the terms and definition, r2_score is a type of an accuracy metric

Result: The r2_score for the dummy regressor was found to be -0.00035. It is okay for r2_score to have a negative value because the model used here is arbitrarily worse and not the best fit for our data.

3.2 Linear Regression Model:

When it comes to a regression question, the first and the simplest model that comes to my mind to implement is linear regression. In this model, the target is predicted by linear approximation. The model predicts the results based on our training data in a linear fashion.

Result: The r2_score for linear regression was found to be 0.5423. This implies that our model has an accuracy of 54.23%, which is not bad but not the best accuracy either.

Our next step will be to set the parameter in such a way so that we can get the best accuracy for this model. This method is also known as **Hyperparameter tuning**. Unfortunately, linear regression does not have hyperparameters, or this model does not. Its other variants, such as ridge or lasso, do, but a simple linear regression does not have any hyperparameters of such kind. Hence, we do not experiment any further with this model.

3.3 KNeighborsRegressor

Also known as KNN, the target column is predicted based on the local interpolation of targets associated with the nearest neighbours in this algorithm. Before fitting our data to the model, we first scale our data. KNN is a distance-based algorithm, and all such distance-based algorithms are affected by the scales of the variables. The Columns in our training set have different scales, resulting in our model having poor scores. Hence, to avoid that, we scale all our columns with the help of the MinMaxScaler found in sklearn. This estimator scales and translates each feature individually such that it is in the given range on the training set, e.g., between zero and one. Once this is done, we then fit our scaled values into the model and predict based on these scales' values, and finally, the predicted values are compared with our validation set(y_test) in the form of r2_score. After following these steps, the r2_score was found to be 0.7675.

Hyperparameter Tuning on KNN:

Hyperparameter is a parameter whose value is used to control the learning process. The reason to perform hyperparameter tuning on our algorithm is to get the most optimised hyperparameters for getting the best results for our learning algorithm.

Q) How to choose the hyperparameters of your model?

We can use the get_params () method to know all the parameters of our model.

By calling the get_parms () function, we get an output of our model's parameters. After looking at the parameters, I decided to work mainly on 'leaf_size', 'n_neighbours' and the 'p-value'. The reason is quite simple: these values are integer values, and the accuracy mainly varies based on integer values than other values. Hence, I aimed to run various tests on different combinations of these parameters. Thus, I assigned some range of values for my hyperparameter with the help of the NumPy library and then ran those parameters in a GridSearchCV.

leaf_size = list of numbers ranging from 5-20, n_neighbors = list of numbers ranging from 5-20, p= [1,2]

To understand grid search, we first need to understand cross-validation. Cross-validation is a resampling method that uses different data portions to test and train a model on various iterations. What GridSearchCV does is, is it applies an exhaustive search over all the values specified for our hyperparameters for our model and omits the best setting for our model. It helps loop through predefined hyperparameters and fit your estimator (model) on your training set. So, in the end, you can select the best parameters from the listed hyperparameters. After fitting the grid search, the best parameters were found and implemented again in our model.

Result: After applying our model's best parameters, our r2_score was 0.7945. This means that our accuracy jumped to 79.45%, and there was a difference of almost 3% in our accuracy compared to the model with no hyperparameter tuning.

The critical question is why we choose these specific values of our hyperparameters. The answer is to avoid overfitting our model. There are some values of the hyperparameters which can give us even more accuracy, but then our model gets too used to our training data which is also known as overfitting. This is where GridSearchCV comes into play, where it understands the problem of overfitting and returns us values that give us the best parameters without facing any overfitting. The graphs of our hyperparameters against our mean_test_score.

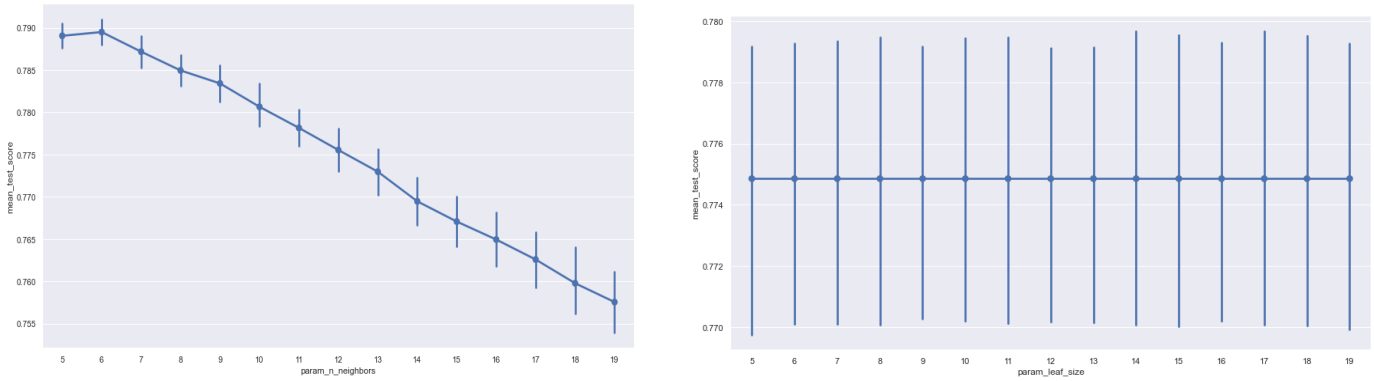

Fig: Accuracies for different settings of n_neighbours and leaf_size

Excellent observation from these figures is that the parameter "n_neighbours" played a significant role in how the scores differ from "leaf_size."

3.4 RandomForestRegressor

A Random Forest Regressor is a combination of multiple Decision Trees. Decision Trees are machine learning algorithm that makes decisions based on conditions formed in a tree-like structure. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. A tree can be seen as a piecewise constant approximation. On fitting the model, the r2_score was found to be 0.7702. Hence, we obtained an accuracy of 77.02% from this model, which is an excellent score.

Hyperparameter Tuning on Random Forest Regressor:

Random forest regressor has parameters that can be tuned. All the names of the parameters can be known by calling the get_params () function. Through this, we can see all the parameters that our model has. After carefully understanding the parameters, I decided to use RandomizedSearchCV. This model defines a grid of hyperparameter ranges and randomly samples from the grid, performing cross-validation with each combination of values. I decided to use these values as my grid:

n_estimators (Number of trees in random forest) = 10 random numbers starting from 400 and endind at 2000

max_features(Number of features to consider at every split) = ['auto', 'sqrt'

max_depth(Maximum number of levels in tree) = 12 numbers starting from 10 and ending at 110 with a gap of 10 and a 'none' value just n case if this parameter doesn't hold much value

min_samples_split(Minimum number of samples required to split a node) = [2, 5, 12]

min_samples_leaf (Minimum number of samples required at each leaf node) = [1, 2, 3]

bootstrap (Method of selecting samples for training each tree) = [True, False]

Result: After running our model with the suggested parameter settings from our RandomisedSearchCV, we obtain an r2_score of 0.8671, i.e. an accuracy of 86.71%, which is the best result so far and has a difference of almost 9% compared to the model with no hyperparameter tuning. Looking at the values of different hyperparameter settings, we can see that some settings achieved an even better score than suggested. However, we do not use them because the model understands that by giving the best accuracy, our Random Forest might be trained only to our trained data and will not perform well on unseen new data. To avoid this overfitting of data, our model yields the best parameters that would work on more generalised data. Here are our accuracies differ with different parameters settings:

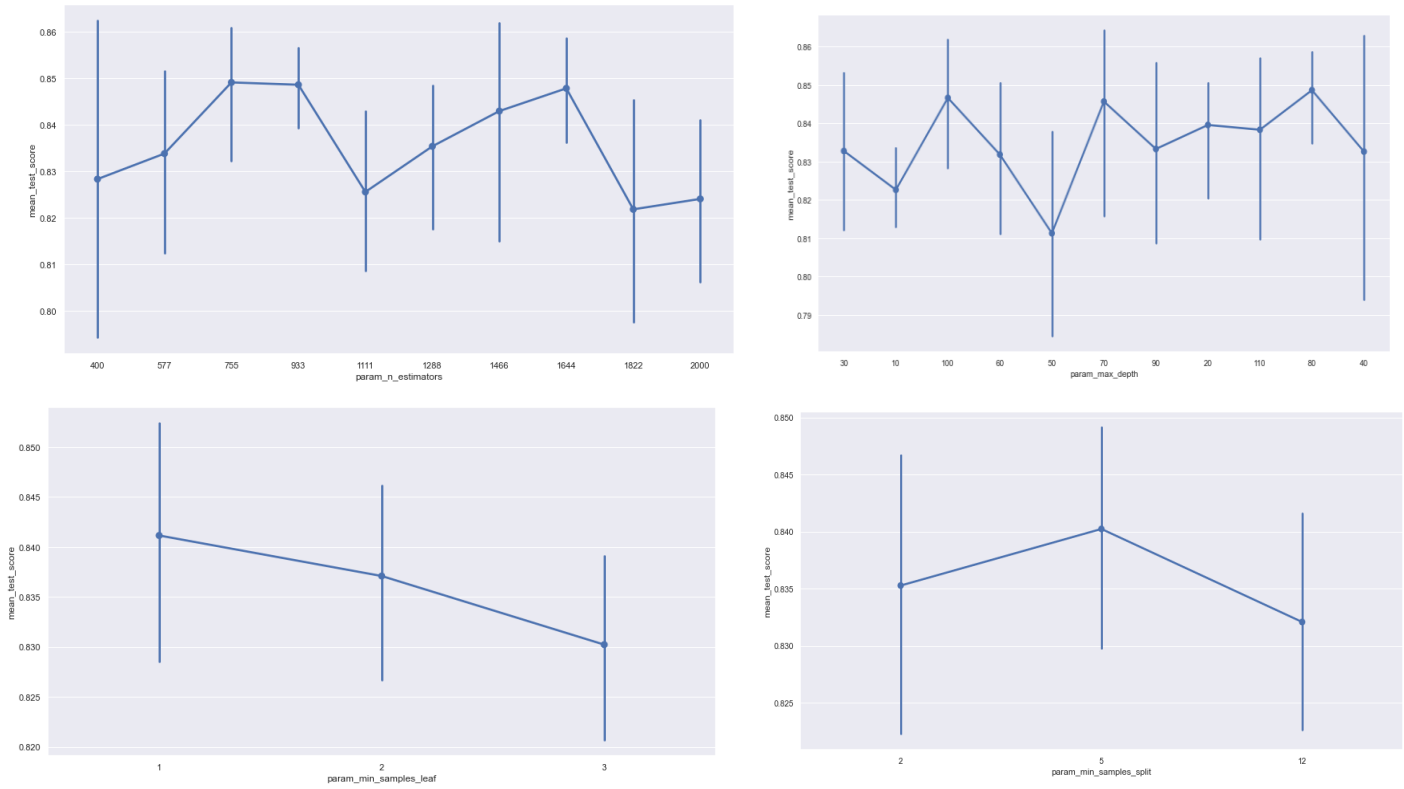

Fig: Accuracies for different settings of n_estimators, max_depth, min_sample_leaves and min_sample_split

Testing our model on unseen data:

Our best model was the Random Forest after applying hyperparameter tuning. Our next step is to determine how our model performs on unseen data. We have kept separate data(test data) in the beginning while splitting our data. This data is unseen, and we will predict the results on that data using our best model and compute our r2_scores.

Our final comparison of r2_score compared to the baseline model is given as follows:

Model(s)	r2_score
DummyRegressor	-0.000358
Linear Regression	0.542320
Knn	0.767505
Knn_with_Hyperarameter_tuning	0.794584
Random Forest	0.770300
Random_forest_with_hyperparameter_tuning	0.867200
Unseen data	0.857041

As shown above, after hyperparameter tuning, the Random Forest Regressor gave us the best r2_score. In contrast, our baseline model gave us the least r2_score, a negative score, as the model was arbitrarily bad for our given data. Coming to our unseen data, our best model performs well on our unseen data as well and hence it would be the suitable model to go with for many any further predictions

Thank you