Max Tjen

5/11

CSDS 133

Final Project Report

Introduction

My project revolves around the amount of rented public bikes per hour in Seoul, South Korea, depending on a number of variables with most being weather related. The data that I am using was released by the city of Seoul, who monitored the dependent variable conditions and how many bikes were rented per hour. Public bike rental sharing has gained a lot of popularity in the past ten years, mostly due to its convenience and pricing, leading to many major cities now have some sort of bike sharing system in place that is available to the public. While the concept is relatively simple, a problem that has arisen with the growing popularity is supply and demand. Because there has been an increase in the amount of people renting the bikes, there are times where there are not bikes available to be rented. As such, my project is attempting to predict how many bikes tend to be rented by hour based on some variables. It would help to solve the problem by providing insight as to approximately how many bikes should be available to be rented. In doing so, it would make it so there would be less instances where there are not any available bikes to rent.

Reference: The Kitchener Today, Ontario, Canada

https://www.kitchenertoday.com/local-news/increased-participation-decline-in-availability-leading-to-widespread-bicycle-shortage-3587366

Data Description and Analysis

After doing basic exploratory data analysis, it was found that the original dataset had 8,760 samples, with each possessing 14 features. Nine of the features provided were various weather elements at the time, including temperature, rainfall, and wind speed. The other five features covered more categorical data, such as the hour, season, and whether or not it is a holiday. Likely because the data was collected by the city, there were no null values in the data, so each sample had values for every feature. The data types of the values were also relatively straightforward, with the majority of features having numeric values. Furthermore, those that were not a numeric data type are easy to make numeric, as there are only a couple possible values - as such, the values can be easily changed into numeric values. Most of the features did not have too many unique samples, but that is not much of a concern as the area where the data is from has relatively consistent weather. When looking at the descriptive statistics of the data, almost all of the features except for visibility has a relatively small range. Similarly, most of the features had small standard deviations, telling that there is not a ton of variability in the data. To check the variability even more, the skew of the data was checked and it was found that nearly all of the features had very low skew values that were close to zero. A correlation heatmap was used to evaluate the data, revealing that some of the features were decently correlated. There were two features that did have a very high correlation though, with those being temperature and dew point temperature. Using figures to see each feature's value distribution revealed that almost all of the features are regularly distributed. However, a couple of the features had a number of high values that may outweigh the smaller values. These same features seemed to also have a decent amount of outliers which could create issues when modeling. Overall though, it appears that most of the data will be good for modeling.

	Hour	Temperature(°C)	Humidity(%)	Wind speed (m/s)	Visibility (10m)	Dew point temperature(°C)	Solar Radiation (MJ/m2)	Rainfall(mm)	Snowfall (cm)	Seasons	Holiday	Functioning Day	Rented Bike Count
count	8760.00	8760.00	8760.00	8760.00	8760.00	8760.00	8760.00	8760.00	8760.00	8760.00	8760.00	8760.00	8760.00
mean	11.50	12.88	58.23	1.72	1436.83	4.07	0.57	0.15	0.08	2.50	0.05	0.97	704.60
std	6.92	11.94	20.36	1.04	608.30	13.06	0.87	1.13	0.44	1.11	0.22	0.18	645.00
min	0.00	-17.80	0.00	0.00	27.00	-30.60	0.00	0.00	0.00	1.00	0.00	0.00	0.00
25%	5.75	3.50	42.00	0.90	940.00	-4.70	0.00	0.00	0.00	2.00	0.00	1.00	191.00
50%	11.50	13.70	57.00	1.50	1698.00	5.10	0.01	0.00	0.00	3.00	0.00	1.00	504.50
75%	17.25	22.50	74.00	2.30	2000.00	14.80	0.93	0.00	0.00	3.00	0.00	1.00	1065.25
max	23.00	39.40	98.00	7.40	2000.00	27.20	3.52	35.00	8.80	4.00	1.00	1.00	3556.00

Figure 1: Descriptive Statistics

```
Hour                      -1.26e-03
Temperature(°C)           -1.75e-01
Humidity(%)                6.86e-02
Wind speed (m/s)           8.94e-01
Visibility (10m)          -6.95e-01
Solar Radiation (MJ/m2)    1.51e+00
Rainfall(mm)               1.46e+01
Snowfall (cm)              8.29e+00
Seasons                    3.41e-02
Holiday                    4.22e+00
Rented Bike Count          1.14e+00
dtype: float64
```

Figure 2: Feature Skewness

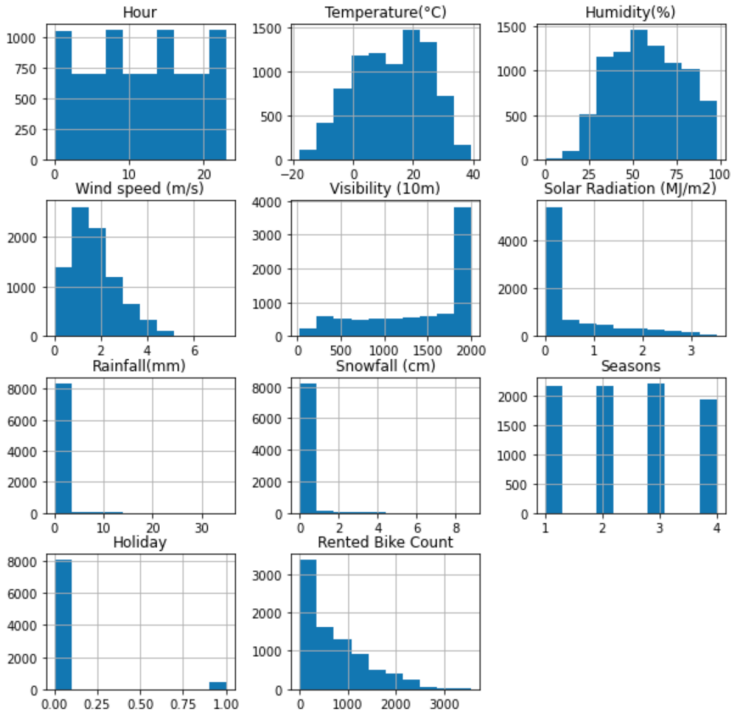

Figure 3: Histogram Plots

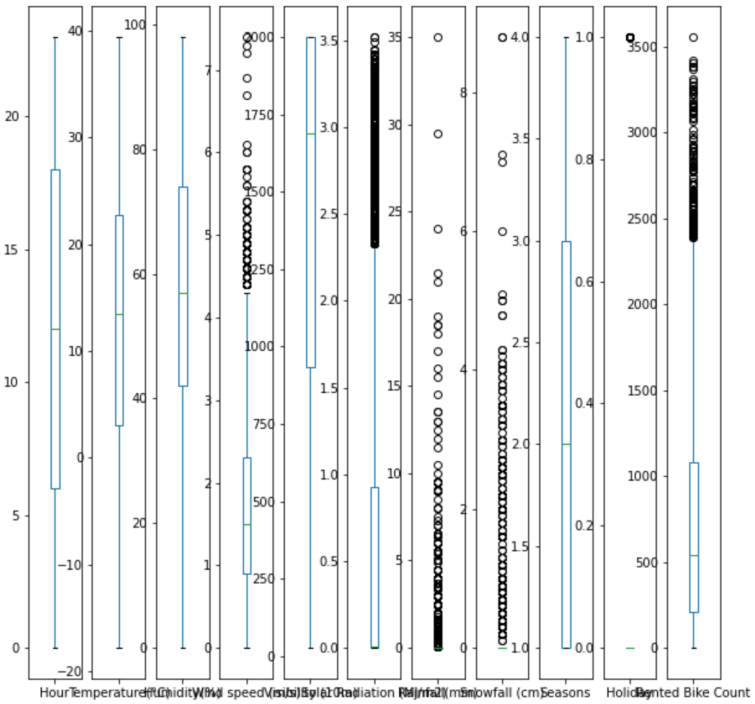

Figure 4: Box Plots

As a whole, the data is well behaved, where there is barely any skewness and the important features are well distributed. To fully prepare the data set for modeling though, there are a couple of actions that I would take to make the model more accurate. The first thing that I would do is to make all of the values binary so that the model can read and interpret them. In doing so, each feature can be compared with the others and the graphics would convey more useful and effective information. Following that, I will remove any features that have a high correlation with another feature, in this case dew point temperature, along with features that cannot easily be read and have little influence over the model results, which is the date feature. In doing so, the model's efficiency would be improved as it would not have to evaluate two very similar features and values. I would also remove samples where the functioning day value is 0, as the data would not have a consistent baseline. After that, I would remove the functioning day feature as all of the samples would have the same value of 1. I would not do anything to treat the outliers as the only feature that influences the target feature is solar radiation. Because of how the data is distributed, there are only a lot of outliers because the majority of the data has little to no radiation while there are a decent amount that have higher values of it.

	Hour	Temperature(°C)	Humidity(%)	Wind Speed(m/s)	Visibility(10m)	Solar Radiation(MJ/m2)	Rainfall(mm)	Snowfall(cm)	Season	Holiday	Rented Bike Count
1	0	-5.2	37	2.2	2000	0.0	0.0	0.0	1	0	254
2	1	-5.5	38	0.8	2000	0.0	0.0	0.0	1	0	204
3	2	-6.0	39	1.0	2000	0.0	0.0	0.0	1	0	173
4	4	-6.0	36	2.3	2000	0.0	0.0	0.0	1	0	78
5	5	-6.4	37	1.5	2000	0.0	0.0	0.0	1	0	100

Figure 5: Head of Data After Cleaning

Feature Selection and Algorithm Testing

To prepare the data for feature and algorithm selection, the data must first be made numerical so that the testing can be done accurately.

```
# replace string values with numerical boolean values
bikesDF = bikesDF.replace(['No Holiday', 'Holiday', 'Winter', 'Spring',
                           'Summer', 'Autumn', 'No', 'Yes'],
                          [0, 1, 1, 2, 3, 4, 0, 1])
bikesDF.head()
```

Figure 6: Make Values Numerical

To figure out what features are the most important, I will use the feature importance attribute using extra trees regressor as well as recursive feature elimination. The extra trees regressor returns values that tell how important the features are to predicting the target value, helping to figure out which features are the most influential. Similarly, recursive feature elimination ranks the features by importance that helps easily tell which features are more useful than others.

```
# Feature Importance
# smaller values -> less important

from sklearn.ensemble import ExtraTreesRegressor
from sklearn.feature_selection import RFE
from sklearn.linear_model import LogisticRegression

array = bikesDF.values
X = array[:,0:10]
Y = array[:,10]
# feature extraction
model = ExtraTreesRegressor()
model.fit(X, Y)
print(model.feature_importances_)

print()

# Feature Extraction with RFE
model = LogisticRegression()
rfe = RFE(model, 5)
fit = rfe.fit(X, Y)
print("Num Features: {}".format(fit.n_features_))
print("Selected Features: {}".format(fit.support_))
print("Feature Ranking: {}".format(fit.ranking_))
```

Figure 7: Feature Importance Code

```
[0.11990546 0.19578243 0.18231671 0.1804376  0.16419295 0.10141149
 0.01696216 0.01214163 0.01934493 0.00750465]
```

Figure 8: Extra Trees Regressor Results

```
Num Features: 5
Selected Features: [ True  True  True False  True False False False  True False]
Feature Ranking: [1 1 1 3 1 4 2 5 1 6]
```

Figure 9: RFE Results

With those rankings, I created five dataframes by removing various features, depending on the feature importance results. Following that, I used negative mean squared error scoring to figure out which of the dataframes performed the best.

```
# create new dataframes with varying amount of features

fourFeatureDF = bikesDF.drop(columns = ['Hour', 'Solar Radiation(MJ/m2)', 'Rainfall(mm)',
                                        'Snowfall(cm)', 'Season', 'Holiday'])

fiveFeatureDF = bikesDF.drop(columns = ['Solar Radiation(MJ/m2)', 'Rainfall(mm)',
                                        'Snowfall(cm)', 'Season', 'Holiday'])

sixFeatureDF =  bikesDF.drop(columns = ['Rainfall(mm)', 'Snowfall(cm)', 'Season', 'Holiday']

sevenFeatureDF =  bikesDF.drop(columns = ['Rainfall(mm)', 'Snowfall(cm)', 'Holiday'])
```

Figure 10: Dataframe Creation

```
array = fourFeatureDF.values
X = array[:,0:4]
Y = array[:,4]
scoring = 'neg_mean_squared_error'
kfold = KFold(n_splits = 10, random_state = 7, shuffle = True)
model = LinearRegression()
results = cross_val_score(model, X, Y, cv = kfold, scoring = scoring)
print(results.mean())

print()

array = fiveFeatureDF.values
X = array[:,0:5]
Y = array[:,5]
results = cross_val_score(model, X, Y, cv = kfold, scoring = scoring)
print(results.mean())

print()

array = sixFeatureDF.values
X = array[:,0:6]
Y = array[:,6]
results = cross_val_score(model, X, Y, cv = kfold, scoring = scoring)
print(results.mean())

print()

array = sevenFeatureDF.values
X = array[:,0:7]
Y = array[:,7]
results = cross_val_score(model, X, Y, cv = kfold, scoring = scoring)
print(results.mean())

print()

array = bikesDF.values
X = array[:,0:10]
Y = array[:,10]
results = cross_val_score(model, X, Y, cv = kfold, scoring = scoring)
print(results.mean())
```

Figure 11: Dataframe Testing

Amount of Features	Result
4	-227293.7587809633
5	-215973.20309479948
6	-216227.47225276334
7	-215046.1107389483
10	-211325.22523529647

Figure 12: Dataframe Testing Results

After conducting those tests it was found that the original ten feature dataframe performed the best. Because the same scoring method was used for each dataframe, the results could easily be interpreted to figure out which was the one to move forward with for model development. With that dataframe, I ran six different regression algorithms to determine what algorithms may potentially work the best for me to model the data: linear regression, ridge regression, lasso regression, elastic net regression, decision tree regressor, and support vector regression. I again used negative mean squared error as the scoring method for each algorithm. In doing so, I would be able to figure out the best performing algorithm to move forward with.

```
array = bikesDF.values
X = array[:,0:10]
Y = array[:,10]
kfold = KFold(n_splits = 10, random_state = 7)
scoring = 'neg_mean_squared_error'

model = LinearRegression()
results = cross_val_score(model, X, Y, cv = kfold, scoring = scoring)
print(results.mean())

print()

model = Ridge()
results = cross_val_score(model, X, Y, cv = kfold, scoring = scoring)
print(results.mean())

print()

model = Lasso()
results = cross_val_score(model, X, Y, cv = kfold, scoring = scoring)
print(results.mean())

print()

model = ElasticNet()
results = cross_val_score(model, X, Y, cv = kfold, scoring = scoring)
print(results.mean())

print()

model = DecisionTreeRegressor()
results = cross_val_score(model, X, Y, cv = kfold, scoring = scoring)
print(results.mean())

print()

model = SVR()
results = cross_val_score(model, X, Y, cv = kfold, scoring = scoring)
print(results.mean())
```

Figure 13: Algorithm Testing Code

Regression Algorithm	Result
Linear	-227293.7587809633
Ridge	-227293.73364004432
Lasso	-227294.37075451267
Elastic Net	-227214.71837038547
Decision Tree	-150205.54421985854
Support Vector	- 468500.6643208548

Figure 14: Algorithm Testing Results

From that testing, it was clear that using the support vector regression algorithm would not work the best. It appeared that the decision tree regressor algorithm performed the best, but I ran an extra test with the remaining five algorithms using r^2 scoring just to make sure.

```
array = bikesDF.values
X = array[:,0:10]
Y = array[:,10]
scoring = 'r2'
kfold = KFold(n_splits = 10, random_state = 7, shuffle = True)

# pipeline
DTRestimator = []
DTRestimator.append(('Standard', StandardScaler()))
DTRestimator.append(('Regressor', DecisionTreeRegressor()))
model = Pipeline(DTRestimator)
results = cross_val_score(model, X, Y, cv = kfold, scoring = scoring)
print('Decision Tree:',results.mean())

LRestimator = []
LRestimator.append(('Standard', StandardScaler()))
LRestimator.append(('Linear Regression', LinearRegression()))
model = Pipeline(LRestimator)
results = cross_val_score(model, X, Y, cv = kfold, scoring = scoring)
print('Linear:',results.mean())

Restimator = []
Restimator.append(('Standard', StandardScaler()))
Restimator.append(('Ridge', Ridge()))
model = Pipeline(Restimator)
results = cross_val_score(model, X, Y, cv = kfold, scoring = scoring)
print('Ridge:',results.mean())

Lestimator = []
Lestimator.append(('Standard', StandardScaler()))
Lestimator.append(('Lasso', Lasso()))
model = Pipeline(Lestimator)
results = cross_val_score(model, X, Y, cv = kfold, scoring = scoring)
print('Lasso:',results.mean())
```

```
Decision Tree: 0.7439090639480395
Linear: 0.5353705511803393
Ridge: 0.535371385634241
Lasso: 0.535344638565112
```

Figure 15: Secondary Algorithm Testing Code and Results

It became obvious that the decision tree regressor algorithm performed vastly better than any of the other regression algorithms. To ensure that the decision tree regressor algorithm was the best performing algorithm, I ran it another time to check the accuracy where the performance was confirmed.

Regression Algorithm	Result
Linear	53.53%
Ridge	53.54%
Lasso	53.53%

Decision Tree	74.39%

Figure 16: Secondary Algorithm Testing Results

Initial Model Training

With the results from the stack of regression algorithms, I decided to move forward with the decision tree regressor model. To perform the initial model training, I used a pipeline to minimize data leakage when the model runs. I also fully switched to r^2 scoring permanently for the rest of the project to keep a consistent scoring method for my runs. When getting the results of the initial model training, I used KFold cross validation to make sure that all of the data in the dataset is used as well as randomly shuffled so that there is not any overfitting of the model

```
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.pipeline import Pipeline
from sklearn.tree import DecisionTreeRegressor

array = bikesDF.values
X = array[:,0:10]
Y = array[:,10]

# pipeline
DTRestimator = []
DTRestimator.append(('Regressor', DecisionTreeRegressor()))
model = Pipeline(DTRestimator)
scoring = 'r2'

kfold = KFold(n_splits = 10, random_state = 7, shuffle = True)
results = cross_val_score(model, X, Y, cv = kfold, scoring = scoring)
print(results, results.mean())
```

Figure 17: Initial Model Training Code

Fold	R^2 Result
1	0.6854664

2	0.72208787
3	0.7183766
4	0.7514351
5	0.79821383
6	0.74976878
7	0.75308572
8	0.79659906
9	0.7495641
10	0.74651215
Average	**0.7471109619536863** **⇒ 74.7%**

Figure 18: Fold and Average Results of Initial Model Training

When breaking down the iterations of the decision tree regressor, it appears to generally be in the 70's in terms of accuracy. Given that, it is good that there is not a lot of variance in the folds, as it means that the model is performing consistently with the dataset. From the results of the model folds, it was confirmed that the decision tree regressor algorithm was the right one to move forward with as the results were consistent with the results from the stack of algorithms.

Model Performance Tuning

In this stage of the project, I ran various bagging and boosting ensemble methods to try and improve the accuracy of the final model. In particular, I used bagged decision tree regressor, random forest regressor, extra trees regressor, ada boost regressor, and gradient boost regressor.

To figure out which one performed the best to boost the accuracy of the model, I used a baseline accuracy of my selected algorithm and then found the difference between the performance of the ensemble method and the baseline performance.

Boost Method	Result	Percent Change
Original (Decision Tree Regressor)	74.74%	-
Bagged Decision Tree Regressor	86.33%	+11.59%
Random Forest Regressor	86.47%	+11.73%
Extra Trees Regressor	**86.86%**	**+12.12%**
Ada Boost Regressor	55.30%	-19.44%
Gradient Boost Regressor	84.71%	+9.97%

Figure 19: Ensemble Method Results

With the results of the various bagging and boosting algorithms, I decided to choose extra trees regressor as the ensemble technique that I would move forward with. While a couple of the other methods returned good amounts of accuracy increase, the extra trees regressor performed the best with an accuracy of 86.8%, a 12.1% increase when compared to the original algorithm I came in with (decision tree regressor), which returned an accuracy of 74.7%. Although the bagged decision tree regressor and random forest regressor both performed well, extra trees regressor performed better than both of them by at least 0.39%, a relatively sizable gap given the performance gains.

```
# initial stack of base and ensemble methods

from sklearn.model_selection import KFold
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import ExtraTreesRegressor

array = bikesDF.values
X = array[:,0:10]
Y = array[:,10]

kfold = KFold(n_splits = 10, random_state = 7, shuffle = True)

modelOne = DecisionTreeRegressor()
modelTwo = ExtraTreesRegressor()

# results for each model
resultsOne = cross_val_score(modelOne, X, Y, cv = kfold, scoring = scoring)
resultsTwo = cross_val_score(modelTwo, X, Y, cv = kfold, scoring = scoring)

print('DTR:', resultsOne.mean() * 100)
print('ETR:', resultsTwo.mean() * 100)

# get the mean of both model scores as a combined performance
meanScore = (resultsOne.mean() + resultsTwo.mean()) / 2
print('Average:', meanScore * 100)
```

```
DTR: 74.48868858173032
ETR: 86.83396164243592
Average: 80.66132511208312
```

Figure 20: Initial Code Stack and Results of Stacked Base Algorithm and Ensemble Technique

After running my base algorithm of decision tree regressor with the ensemble technique of extra trees regressor, the averaged performance was 80.66%. Following that, I began to do performance tuning for my base algorithm. However in my situation, the grid search method of tuning was not working correctly for some reason, as it would return the worst performing parameter value. As such, I had to manually change the parameter value each time to find the optimal value. The parameters that I attempted to tune were criterion, splitter, min_samples_leaf, min_weight_fraction, random_state, max_leaf_nodes, max_depth, min_samples_split, and max_features. Most of these parameters dealt with how the tree made decisions when run, in turn affecting its performance. While I tested nine different parameters, only three of them ended up making a difference, as for the others the base value already

optimized the algorithm's performance. The three parameters that did impact the score were max_depth, min_samples_split, and max_features.

⇒ **Bolded Row** = best value result

Parameter Value	Result	Percent Change
0	75.51%	-
5	73.45%	-2.06%
10	80.42%	+4.91%
15	76.62%	+1.11%
8	79.80%	+4.29%
9	**80.72%**	**+5.21%**

Figure 21: Results of max_depth Parameter Tuning

Parameter Value	Result	Percent Change
Auto	80.72%	-
5	80.58%	-0.14%
10	80.99%	+0.27%
15	81.24%	+0.52%
20	81.30%	+0.58%
25	81.40%	+0.68%
30	81.40%	+0.68%

35	81.40%	+0.68%
40	81.31%	+0.59%
37	81.35%	+0.63%
33	81.37%	+0.65%
34	81.38%	+0.66%
36	**81.44%**	**+0.72%**

Figure 22: Results of min_samples_split Parameter Tuning

Parameter Value	Result	Percent Change
0	81.44%	-
5	79.64%	-1.80%
8	81.46%	+0.02%
10	81.43%	-0.01%
9	**81.57%**	**+0.13%**
7	80.54%	-0.90%

Figure 23: Results of max_features Parameter Tuning

Although most of the parameters' ideal value was the default value, the three parameters that were found to get better results made a significant impact in the performance of the algorithm. The first was max_depth, which determines the maximum amount of levels the tree can have. It was found that the ideal depth was 9, where accuracy increased by 5.21% to 80.72%. The next parameter that found performance was min_samples_split, which tells the tree how many

samples are required to split a tree node. There was not a ton of accuracy increase, as the ideal amount was 36, which increased accuracy by 0.72% to 81.44%. The final parameter tuned was max_features, which establishes the maximum number of features to consider. 9 features performed the best, finding an extra 0.13% to achieve a total performance of 81.57%.

I also did some parameter tuning afterwards on the boosting ensemble technique and the parameters of the cross validation tool KFold. The extra trees regressor only increased about 0.56% to an accuracy rate of 87.40%.

```
# tuned stack of base and ensemble methods

array = bikesDF.values
X = array[:,0:10]
Y = array[:,10]

kfold = KFold(n_splits = 8, random_state = 5, shuffle = True)

modelOne = DecisionTreeRegressor(max_depth = 9, min_samples_split = 36, max_features = 9)
modelTwo = ExtraTreesRegressor(n_estimators = 400, max_features = 7, min_samples_leaf = 1,
                               min_samples_split = 2, max_depth = 17)

# results for each model
resultsOne = cross_val_score(modelOne, X, Y, cv = kfold, scoring = scoring)
resultsTwo = cross_val_score(modelTwo, X, Y, cv = kfold, scoring = scoring)

print('DTR:', resultsOne.mean() * 100)
print('ETR:', resultsTwo.mean() * 100)

# get the mean of both model scores as a combined performance
meanScore = (resultsOne.mean() + resultsTwo.mean()) / 2
print('Average:', meanScore * 100)
```

Figure 24: Parameter Tuned Code Stack of Stacked Base Algorithm and Ensemble Technique

Test Number	Decision Tree Result	Overall Result
1	81.35%	84.38%
2	80.81%	82.09%
3	81.25%	84.34%
Average	**81.14%**	**83.60%**

Figure 25: Results of Consistency Tests

At the end of parameter tuning, I retrained the performance tuned model with the same training data and ran it three times to ensure consistency. The average accuracy of the decision tree regressor was 81.14%, an increase of 6.40%, and the overall accuracy when averaged with the extra trees regressor was found to be 83.60%. When looking at the individual consistency tests, the accuracy results were consistent throughout, keeping within about 0.5% of each other. The one test that was lower than the other two appears to not just be isolated to the tuned decision tree regressor algorithm, as the averaged result decreased by more, signifying that the tuned extra trees regressor result was also lower than the other tests.

Model Testing with Holdout Data

The final part of the project involves running the tuned models on the 10% of data that was previously held out for final testing. In order to use that data though, it has to be transformed and formatted into the same style as the training data that has been used throughout to prepare the models. The steps taken to prepare the held out data follow the same steps as before, including removing samples on non functioning days, dropping certain features, and making non numeric values numeric.

```
# prepare data for modeling
trainArray = bikesDF.values
testArray = testDF.values

xTrain = trainArray[:,0:10]
yTrain = trainArray[:,10]

xTest = testArray[:,0:10]
yTest = testArray[:,10]

kfold = KFold(n_splits = 8, random_state = 5, shuffle = True)

# sub models
modelOne = DecisionTreeRegressor(max_depth = 9, min_samples_split = 36, max_features = 9)
modelTwo = ExtraTreesRegressor(n_estimators = 400, max_features = 7, min_samples_leaf = 1,
                               min_samples_split = 2, max_depth = 17)

# fit the data
modelOne.fit(xTrain, yTrain)
modelTwo.fit(xTrain, yTrain)

# score each model seperately
scoreOne = modelOne.score(xTest, yTest)
scoreTwo = modelTwo.score(xTest, yTest)

# get the mean of both model scores as a combined performance
meanScore = (scoreOne + scoreTwo) / 2
print(meanScore * 100)
```

Figure 26: Final Code to Test on Held Out Data

Test Number	Result
1	84.13%
2	84.14%
3	84.63%
Average	**84.30%**

Figure 27: Results of Final Tests

Similar to before, I ran the models on the held out data three times to make sure that the results were consistent. As a whole, the average accuracy on the held out data actually performed better than on the training data, mostly due to the final test, ending with an average of 84.30%.

Conclusion and Inferences

From this project, it was found that the number of rented public bikes by hour in Seoul, South Korea could be predicted at a rate of about 84.30%. Luckily, because the data was collected by the city, there were not any null values so it was relatively straightforward when it came to cleaning the data prior to modeling. Similarly, the weather data was pretty normalized and normally distributed because of how consistent the region's weather was. Through the process of the project, it was found that certain weather elements contributed greatly to how many bikes were rented, although nearly all of them did play a role, albeit smaller.

A major problem that has been affecting the bike rental industry is being able to adequately have enough available bikes with the growing popularity and demand. The results of the models are significant because of how it may help companies and cities more accurately predict the amount of bikes may be demanded given various factors. An obstacle that may arise though is that many of the features that help predict the amount are weather related. As such, those are difficult values to predict so the model is more useful when looking at historical data and identifying trends to act on rather than predicting the amount of bikes that may be rented in the immediate future. Although my model does help to predict the amount of rented bikes in a given hour, companies still have to figure out a way to balance having available bikes while not having so many available that at times, many are not in use. With that though, my models and their results can help them create a more accurate picture to assess this problem by evaluating past weather data and determining what they see fit as the amount of bikes should be out and available to people.

Throughout this project, I have learned many useful lessons and skills that I can carry forward in my life. One of the first things that I learned was the difference between classification

and regression models and how each can be leveraged with different datasets. Another thing that I learned is how important and powerful numerical data is for many instances, but particularly for data modeling. While in modeling it is essential for data to be numeric so that values can be compared and contrasted, having numeric data helps to create visuals and convey results that are applicable in nearly any industry. The next lesson that I learned was how not all data features are useful for data modeling and also how important it is to select the right algorithm to use for modeling. I had previously thought that when analyzing data, all the features and values were important to use and that most of the same types of algorithms performed similarly. However, through this project I found that there are certain features and algorithms performed significantly better than others and impacted final modeling. Something else that I learned was the importance of using pipelines when modeling data. I had known a little about data leakage before, but not in the same context as when modeling. Similarly, I learned about having holdout data to use as unseen data to be used when testing the model's accuracy and performance. While I knew that it was important to test the model once finalized, I never realized that it was important to use unseen data so that the model would not have ever seen it, which in turn would unfairly increase the performance. Lastly, I learned about how it was essential to check consistency and the folds when running the models, rather than just look at the final mean score. I did not understand before how there are times when there is a lot of variance between different runs, which would symbolize that the given model is not working too well with the data in predicting the target value.