# Bicycle Sharing Demand

November 29, 2018

## 1 Class Project - Bike Sharing Demand

Prepared By: Enrique Castillo, Erika Estrada, Walter Carbajal, Vrezh Khalatyan

### 1.1 # Predicting amount of bicycles to be used on a given day

**Bike sharing throughout the world has started to become increasingly popular throughout the past decade. Large transportation companies have begun to "hop on the train" in order to gain some of the market before it gets to large. However, aside from getting people around their cities in a much easier and healthier way. The bike sharing systems/programs have the potential to generate a large amount of data which can function as a sensor network, which can be used for studying mobility in a city.**

**Kaggle hosted a friendly competetion to have participants find the best machine learning algorithm to predict the amount of bikes used on a given day.**

**We are given hourly rental data spanning two years. The training set is comprised of the first 19 days of each month, while the test set is the 20th to the end of the month.**

**Our task: "You must predict the total count of bikes rented during each hour covered by the test set, using only information available prior to the rental period.**

**In general, our goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features.**

**We use methods such as *Cross Validation* to assess the predictive performance of the models and to judge how they perform outside the sample to a new data set also known as test data.**

**The motivation to use *Cross Validation* techniques is that when we fit a model, we are fitting it to a training dataset, Without *Cross Validation*, we only have information on how does our model perform to our in-sample data. Ideally, we would like to see how does the model perform when we have new data in terms of accuracy of its predictions.**

### 1.2 Data Fields

- **datetime:** *hourly date + timestamp*

- **season:** *1 = spring, 2 = summer, 3 = fall, 4 = winter*
- **holiday:** *whether the day is considered a holiday*
- **workingday:** *whether the day is neither a weekend nor holiday*
- **weather:**
  - 1: Clear, Few clouds, Partly cloudy, Partly cloudy
  - 2: Mist + Cloudy, Mist + Broken clouds, Mist + Few clouds, Mist
  - 3: Light Snow, Light Rain + Thunderstorm + Scattered clouds, Light Rain + Scattered clouds
  - 4: Heavy Rain + Ice Pallets + Thunderstorm + Mist, Snow + Fog
- **temp:** *temperature in Celsius*
- **atemp:** *"feels like" temperature in Celsius*
- **humidity:** relative humidity
- **windspeed:** *wind speed*
- **casual:** *number of non-registered user rentals initiated*
- **registered:** *number of registered user rentals initiated*
- **count:** *number of total rentals*

### 1.3 Analyzing Our Data

We will start our attempt with this project by seeing the type of data we are dealing with.

First we import the necessary modules to read the data. In this case, we will use the pandas and numpy library.

```
In [1]: import pandas as pd
        import numpy as np
```

Next, we will import our dataset from our local directory.

```
In [2]: bicycle_df = pd.read_csv('train.csv')
```

The first thing and the easiest thing is to preview the first five rows of data in out dataset. So we use...

```
In [3]: # bicycle_df.head() # omitted to reduce space on report (see .iypnb)
```

Here, some questions about that dataset start to come to mind. Questions such as:

- What are the max values of season, holiday, workingday weather?
- What can we do to datetime to make it more useful?
- Do we need the casual and registered labels?
- Is there any categorical features?

To answer the first question, we use the dataframe's describe() method...

```
In [4]: # bicycle_df.describe() # omitted to reduce space on report (see .iypnb)
```

Analyzing the results of the describe() method, we see that:

- **Season** has integer values of 1,2,3,4 (Winter, Spring, Summer, Fall)
- **Holiday** has integer values of 1 or 0 (Holiday or non-holiday)
- **Working Day** has integer values 1 or 0 (Work Day or not a Work Day)
- **Weather** has integer values 1,2,3,4 each number representing a specific weather condition (described in the Data Field section above)

Based of these results, we can determine that these four fields must be categorial features. Therefore, we can conclude that our dataset has a mixture of numerical and categorical features, which may complicate our results later in this project.

Now, we must check whether the suggested fields above need to be converted into categorical types. To do this, we use the dataframe's dtypes() method to check the each column's data type...

```
In [5]: # bicycle_df.dtypes # omitted to reduce space on report (see .iypnb)
```

As shown above, we will need to convert the features mentioned above into categorical features. We will do this later on in our project.

Next, lets take a quick look at the dataset's volume...

```
In [6]: bicycle_df.shape
```

```
Out[6]: (10886, 12)
```

For our training set, we have **10,886** entries of data that we will use to train our machine learning models to predict the amount of bicycles used on a given hour of a day.

To help us visualize the data, we will plot the data based on the **amount of bicycles used throughout the seasons** and the **amount of bicycles used hourly**. First we will import the necessary models to accomplish this...

```
In [7]: import matplotlib.pyplot as plt
        import seaborn as sn
        import calendar

        from datetime import datetime
```

Now we re-import our dataset into a new dataframe...

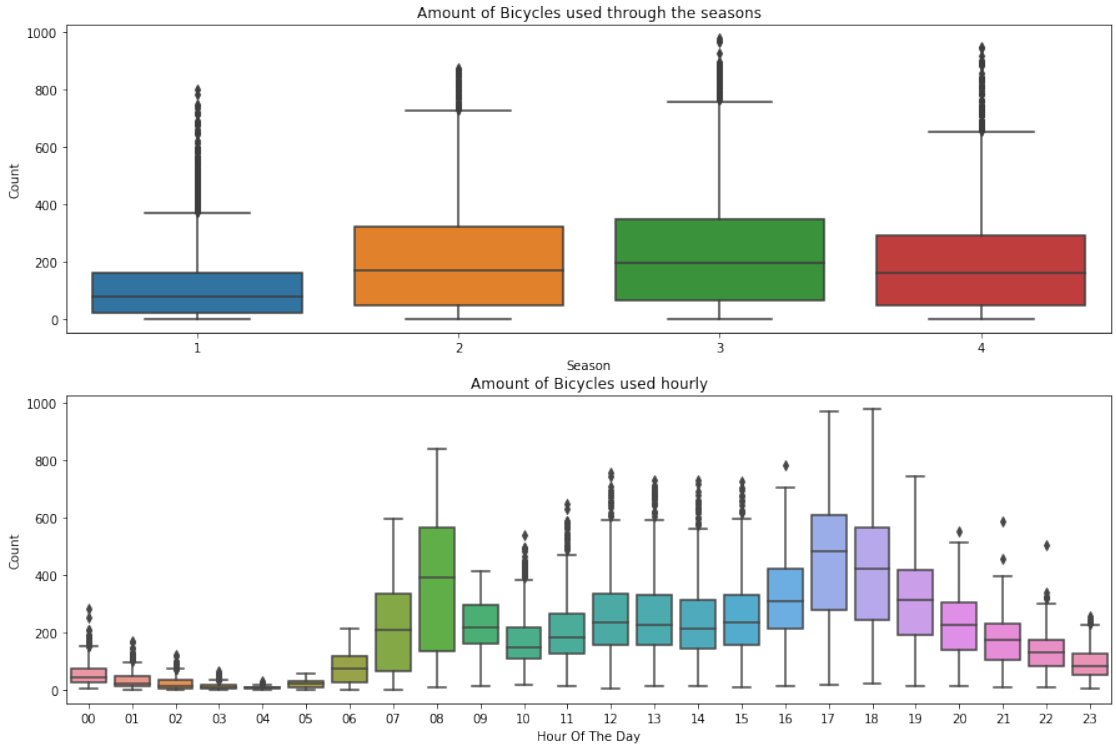
```
In [8]: modified_bicycle_df = pd.read_csv('train.csv')
```

Next, we use the *datetime* feature to seperate it into *date*, *hour*, and *month* features to better allow us to visualize the data and drop it from this dataframe instance since we don't need it here anymore...

```
In [9]: modified_bicycle_df['date'] = modified_bicycle_df.datetime.apply(lambda x : x.split()[0
        modified_bicycle_df['hour'] = modified_bicycle_df.datetime.apply(lambda x : x.split()[1
        modified_bicycle_df['month'] = modified_bicycle_df.date.apply(lambda dateString : calen

        modified_bicycle_df = modified_bicycle_df.drop(['datetime'],axis=1)
```

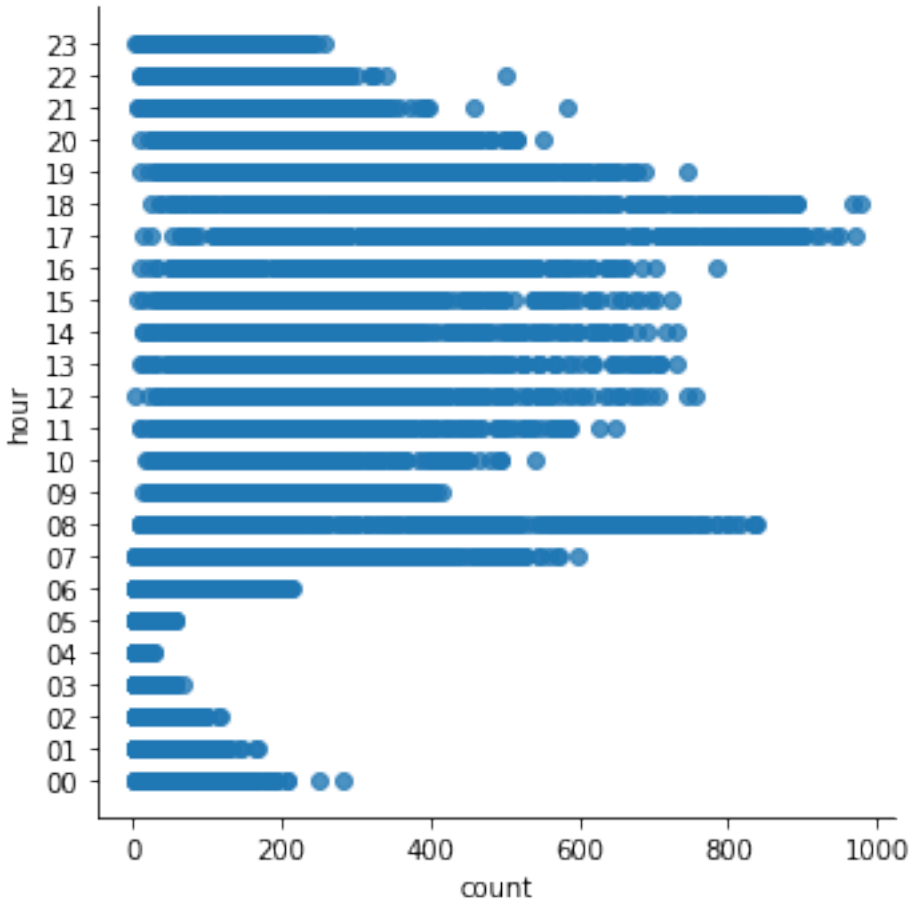
Finally, we will plot the data into two box graphs...

```
In [10]: fig, axes = plt.subplots(nrows=2,ncols=1)

         fig.set_size_inches(15, 10)

         sn.boxplot(data=modified_bicycle_df,
                    y="count",
                    x="season",
                    orient="v",
                    ax=axes[0])

         sn.boxplot(data=modified_bicycle_df,
                    y="count",
                    x="hour",
                    orient="v",
                    ax=axes[1])

         axes[0].set(xlabel='Season',
                     ylabel='Count',
                     title="Amount of Bicycles used through the seasons")

         axes[1].set(xlabel='Hour Of The Day',
                     ylabel='Count',
                     title="Amount of Bicycles used hourly")

Out[10]: [Text(0,0.5,'Count'),
          Text(0.5,0,'Hour Of The Day'),
          Text(0.5,1,'Amount of Bicycles used hourly')]
```




Lets use different graph to read the busiest hour of day with these bicycles...

```
In [11]: df = pd.DataFrame()
         df['count'] = modified_bicycle_df['count']
         df['hour'] = modified_bicycle_df['hour']

         sn.lmplot('count', 'hour', data=df, fit_reg=False)

Out[11]: <seaborn.axisgrid.FacetGrid at 0x1948a8294a8>
```




Here we can clearly see that people use the bicycles the most at **hours 17 & 18**

After analyzing our data, we will begin to use different machine learning algorithms and start our attempt at predicting our **count** label.

First, lets import the necessary modules we will use across all our different algorithms..

```
In [12]: from sklearn.model_selection import cross_val_score
         from sklearn.preprocessing import scale
         from sklearn.model_selection import train_test_split
         from sklearn.metrics import accuracy_score
         from sklearn.metrics import r2_score
         from matplotlib.pyplot import bar
```

We reimport the data to keep data for pre-examination and for implementation separate. Similar to visualizing the data, we will split the *datetime* feature into four features (*year, month, day, hour*) instead of three and then drop the *datetime* feature after we have extracted all the necessary data we need...

```
In [13]: train_data = pd.read_csv("train.csv")
         train_data["year"] = train_data.datetime.apply(lambda x : x.split()[0].split("-")[0])
```

```
train_data["month"] = train_data.datetime.apply(lambda x : x.split()[0].split("-")[1])
train_data["day"] = train_data.datetime.apply(lambda x : x.split()[0].split("-")[2])
train_data["hour"] = train_data.datetime.apply(lambda x : x.split()[1].split(":")[0])
train_data = train_data.drop('datetime', axis=1)
```

Next, we create our **feature matrix** and the **label vector**. One other thing we do here is scale our feature matrix to help normalize our dataset which can offer better results...

```
In [14]: features= train_data.drop(['casual','registered','count'], axis=1)
         scaled_features = scale(features)
         target = train_data['count']
```

```
In [15]: X_train, X_test, y_train, y_test = train_test_split(scaled_features,
                                                             target,
                                                             test_size=0.3,
                                                             random_state=2)
```

We will begin by first using some classifiers and analyzing their results.

### 1.4 DecisionTreeClassifier

First up, we'll the DecisionTreeClassifier. Lets import the necessary module and create our instance of this model...

```
In [16]: from sklearn.tree import DecisionTreeClassifier

         my_decisionTree = DecisionTreeClassifier(random_state=2,
                                                  max_depth = 10)
```

Next, lets train the model and get the accuracy score..

```
In [17]: my_decisionTree.fit(X_train, y_train)
         y_predict_dt = my_decisionTree.predict(X_test)
         score_dt = accuracy_score(y_test, y_predict_dt)
         print(score_dt)
```

```
0.022351500306184935
```

Finally, we then try 10-fold *Cross Validation*

```
In [18]: accuracy_list = cross_val_score(my_decisionTree,
                                          scaled_features,
                                          target,
                                          cv=10)
         accuracy_cv = accuracy_list.mean()
         print(accuracy_cv)
```

```
C:\Users\wcarb\Anaconda3\lib\site-packages\sklearn\model_selection\_split.py:605: Warning: The
  % (min_groups, self.n_splits)), Warning)
```

```
0.020222767683661436
```

```
In [19]: test_score = r2_score(y_test, y_predict_dt)
         print('The R^2 score is {}'.format(test_score))
```

```
The R^2 score is 0.6645823715846187
```

### 1.5 RandomForestClassifier

Next up, we will try using the Random Forest Classifier. First we import the necessary modules and create our instance of our model...

```
In [20]: from sklearn.ensemble import RandomForestClassifier

         my_RandomForest = RandomForestClassifier(n_estimators = 100,
                                                  bootstrap = True,
                                                  random_state=3)
```

Then, we will use the same training and testing sets used above and train our Random Forest Classifier. We also calculate the accuracy score of our prediction...

```
In [21]: my_RandomForest.fit(X_train, y_train)
         y_predict_RF = my_RandomForest.predict(X_test)
         accuracy_RF =accuracy_score(y_test, y_predict_RF)
         print("Prediction Accuracy using only RandomForest Classifier: " + str(accuracy_RF))
```

```
Prediction Accuracy using only RandomForest Classifier: 0.0192896509491733
```

Finally, we use 10-fold cross validation and calculate our new accuracy score...

```
In [22]: rf_accuracy_list = cross_val_score(my_decisionTree,
                                            scaled_features,
                                            target,
                                            cv=10,
                                            scoring='accuracy')
         accuracy_rf = rf_accuracy_list.mean()

         print("Prediction Accuracy with RandomForest Using Cross_Validation: " + str(accuracy_
```

```
C:\Users\wcarb\Anaconda3\lib\site-packages\sklearn\model_selection\_split.py:605: Warning: The
  % (min_groups, self.n_splits)), Warning)
```

```
Prediction Accuracy with RandomForest Using Cross_Validation: 0.020222767683661436
```

```
In [23]: test_score = r2_score(y_test, y_predict_RF)
         print('The R^2 score is {}'.format(test_score))
```

```
The R^2 score is 0.6008186779703486
```

Let's stop here with the classifiers and analyze the results we've gotten so far. Using the Decision Tree Classifier, we only achieved a accuracy score of **2.2%** without *Cross Validation* and **2%** with it! The same could be said for the Random Forest Classifier, where we achieved **1.9%** without *Cross Validation* and **2%** with it!

Because of these results, we can conclude that using Classifiers for our particular dataset will not work out since some of our features are not continuous!

Next we will begin using Regressors and see how they perform with out dataset.

### 1.6 KNeighborsRegressor

Lets import the KNeighborsRegressor from the sklearn.neighbors package and create an instance of our model...

```
In [24]: from sklearn.neighbors import KNeighborsRegressor

         my_algorithm = KNeighborsRegressor(n_neighbors=75)
```

Finally, lets train the machine learning algorithm...

```
In [25]: accuracy_list = cross_val_score(my_algorithm,
                                         scaled_features,
                                         target,
                                         cv=10)
         accuracy_cv = accuracy_list.mean()
         print(accuracy_cv)
```

```
0.27422459123592324
```

```
In [26]: my_algorithm.fit(X_train, y_train)
         y_predict_KNR = my_algorithm.predict(X_test)
         test_score = r2_score(y_test, y_predict_KNR)
         print('The R^2 score is {}'.format(test_score))
```

```
The R^2 score is 0.5084543878223611
```

With only a **27.4%** accuracy score, we can conclude that this particular algorithm is very inefficient with our particular dataset since it grouping the data through multiple categories creates too many clusters, therefore causing error within our test.

### 1.7 DecisionTreeRegressor

Our next model is the Decision Tree Regressor. First lets use the normalized feature matrix and split the data into training and testing sets...

```
In [27]: from sklearn.tree import DecisionTreeRegressor
         dtr = DecisionTreeRegressor()
```

Next, we train the model, use 10-fold *Cross Validation*, and calculate the accuracy score based on this...

```
In [28]: dtr = dtr.fit(X_train, y_train)

         y_predict =dtr.predict(X_test)

         accuracy_rfr_cv = cross_val_score(dtr, scaled_features, target, cv=10).mean()
         print('The accuracy of our prediction using cross validation is  {}%'
               .format(round(accuracy_rfr_cv * 100, 2)))
```

```
The accuracy of our prediction using cross validation is  67.88%
```

Finally we calculate the R^2 score...

```
In [29]: test_score = r2_score(y_test, y_predict)
         print('The R^2 score is {}'.format(test_score))
```

```
The R^2 score is 0.8745227801786325
```

Analyzing our Decision Tree Regressor, we managed to get a accuracy score of **67.88%** using *Cross Validation* and a R^2 score of **0.871**! This is better than our KNeighborRegressor model. However, our Random Forest Regressor still manages to out perform the Decision Tree Regressor as will soon find out.

### 1.8 RandomForestRegressor

Using the RandomForestRegressor, lets first create an instance of the mode, train it and predict.

```
In [30]: from sklearn.ensemble import RandomForestRegressor

         my_random_forest_reg = RandomForestRegressor(random_state = 0,
                                                      max_depth = 20,
                                                      n_estimators = 150)
         my_random_forest_reg.fit(X_train, y_train)
         y_predict = my_random_forest_reg.predict(X_test)
```

We then calculate the accuracy of the results using 10-fold cross-validation, similar as used in our KNeighborsRegressor model...

```
In [31]: accuracy_RF_CV = cross_val_score(my_random_forest_reg,
                                          features,
                                          target,
                                          cv=10).mean()
         print('The accuracy of our prediction using cross validation is  {}%'
               .format(round(accuracy_RF_CV * 100, 2)))
```

```
The accuracy of our prediction using cross validation is  79.68%
```

Finally, again we calculate the R^2 score for our model

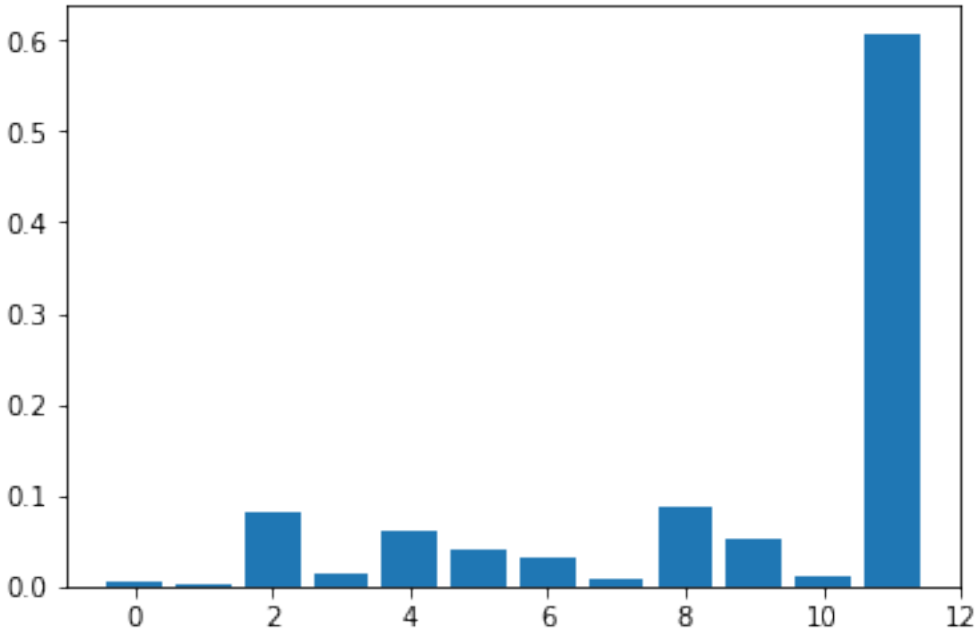
```
In [32]: test_score = r2_score(y_test, y_predict)
         print('The R^2 score is {}'.format(test_score))
```

```
The R^2 score is 0.9387967343298419
```

As we can see Random forest regressor is a much better algorithm for this problem given a accuracy score of **79.68%** using *Cross Validation* and a R^2 score of **0.94**! Next, we will find the least important features and remove them to see if we can improve on the score of Random forest regressor

```
In [33]: from matplotlib import pyplot
         my_random_forest_reg.fit(X_train, y_train)
         importance = my_random_forest_reg.feature_importances_
         names = list(features)
         bar(range(len(importance)), importance)
         print (sorted(zip(map(lambda x: round(x, 4), importance), names), reverse=True))
         pyplot.show()
```

```
[(0.6077, 'hour'), (0.0871, 'year'), (0.0812, 'workingday'), (0.0605, 'temp'), (0.0518, 'month
```




Now we will only include the 6 best features...

```
In [34]: new_features = features.drop(['humidity',
                                       'season',
                                       'weather',
                                       'day',
                                       'windspeed',
                                       'holiday'],
                                      axis=1)
```

We normalize the new features

```
In [35]: new_scaled_features = scale(new_features)
```

Now, we use 10-fold Cross Validation with only the six best features...

```
In [36]: accuracy_RF_CV = cross_val_score(my_random_forest_reg,
                                           new_scaled_features,
                                           target,
                                           cv=9).mean()

         print('The accuracy of our prediction using cross validation is  {}%'
               .format(round(accuracy_RF_CV * 100, 2)))
```

```
The accuracy of our prediction using cross validation is  76.77%
```

Finally, lets calculate the R^2 score for this model

```
In [37]: test_score = r2_score(y_test, y_predict)
         print('The R^2 score is {}'.format(test_score))
```

```
The R^2 score is 0.9387967343298419
```

We can see that we've only only lost a few meausures in our metric score (**79.68% -> 76.77%**) but we gain a lot of speed time in computing by removing the 6 least important features

For the following algorithms (Linear Regression, AdaBoot Regressor, Bagging Regressor), we trained and tested with the same feature engineering methods:

- extracting the date features from 'datetime'
- normalizing the feature values
- manually changing negative predictions to zero (since count must be a non-negative number)

For some of these algorithms, we also applied extra data manipluation that improves that particular model.

Here's the function we'll use to remove negative predictions...

```
In [38]: def zero_min(predictions):
             i = 0
             for val in predictions:
```

```
            if val < 0:
                predictions[i] = 0
            i += 1
        return predictions
```

Dataframe for these algorithms...

```
In [39]: modified_bicycle_df = pd.read_csv("train.csv")

         modified_bicycle_df["date"] = modified_bicycle_df.datetime.apply(lambda x : x.split()
         modified_bicycle_df["hour"] = modified_bicycle_df.datetime.apply(lambda x : x.split()
         modified_bicycle_df["weekday"] = modified_bicycle_df.date.apply(lambda dateString : da
         modified_bicycle_df["month"] = modified_bicycle_df.date.apply(lambda dateString : date
         modified_bicycle_df["year"] = modified_bicycle_df.date.apply(lambda dateString : datet

         categoryVariableList = ["hour","weekday","month","season","holiday","workingday"]
         for var in categoryVariableList:
             modified_bicycle_df[var] = modified_bicycle_df[var].astype("category")

         modified_bicycle_df = modified_bicycle_df.drop(["datetime"],axis=1)

         features = ['season', 'holiday', 'workingday', 'weather', 'temp', 'atemp',
                     'humidity', 'windspeed', 'hour', 'weekday', 'month', 'year']

         # feature matrix and label vector
         X = modified_bicycle_df[features]
         y = modified_bicycle_df['count'] # ignoring casual and registered

         # normalizing the data
         scaled_X = scale(X)
```

### 1.9 Linear Regression

First, lets create our instance of the model...

```
In [40]: from sklearn.linear_model import LinearRegression

         lin_reg = LinearRegression()

         X_train, X_test, y_train, y_test = train_test_split(X,
                                                             y,
                                                             test_size=0.25,
                                                             random_state=4)
```

Let's train our model and predict the our label with the testing set...

```
In [41]: lin_reg.fit(X_train, y_train)
         y_predict = lin_reg.predict(X_test)
```

Next, lets remove any negative predictions...

```
In [42]: y_predict = zero_min(y_predict)
```

Finally, our evaluation of our Linear Regression model...

```
In [43]: accuracy_list = cross_val_score(lin_reg, X, y, cv=10)
         accuracy = accuracy_list.mean()
         score = r2_score(y_test, y_predict)

         print("Accuracy:", accuracy)
         print("R2:", score)
```

```
Accuracy: 0.13754153769369504
R2: 0.413746544859943
```

Since this is a regression model and some of our data is categorical, we can try to turn those values into dummy variables. Let's repeat what we just did above with this new method...

```
In [44]: linear_dummies_X = pd.get_dummies(X) # all features and dummies
         print(X.shape)

         # normalizing the data
         linear_scaled_X = scale(linear_dummies_X)

         # train and test
         X_train_linear, X_test_linear, y_train_linear, y_test_linear = train_test_split(linear
                                                                                         y,
                                                                                         test_s
                                                                                         rando



         lin_reg = LinearRegression()
         lin_reg.fit(X_train_linear, y_train_linear)
         y_predict = lin_reg.predict(X_test_linear)

         # adjust negative predictions
         y_predict = zero_min(y_predict)

         # evaluation
         accuracy_list = cross_val_score(lin_reg, linear_scaled_X, y, cv=10)
         accuracy = accuracy_list.mean()
         score = r2_score(y_test_linear, y_predict)

         print("Accuracy:", accuracy)
         print("R2:", score)
```

```
(10886, 12)
Accuracy: 0.3771558554656015
R2: 0.7047931511643091
```

### 1.10 AdaBoost Regressor

Creating an AdaBoost Regressor instance and spliting the dataset...

```
In [45]: from sklearn.ensemble import AdaBoostRegressor

         ada_boost = AdaBoostRegressor(n_estimators=20, random_state=3)

         X_train, X_test, y_train, y_test = train_test_split(scaled_X, y, test_size=0.25, rando
```

Training our model...

```
In [46]: ada_boost.fit(X_train, y_train)
         y_predict = ada_boost.predict(X_test)
```

Adjusting negative predictions...

```
In [47]: y_predict = zero_min(y_predict)
```

Our model's evaluation...

```
In [48]: accuracy_list = cross_val_score(ada_boost, scaled_X, y, cv=10)
         accuracy = accuracy_list.mean()
         score = r2_score(y_test, y_predict)

         print("Accuracy:", accuracy)
         print("R2:", score)
```

```
Accuracy: 0.042574166586473616
R2: 0.6199934385645652
```

For this algorithm, we create dummy variables again. This gives us 57 features, so we can try to perform feature reduction. To find the best number of features, we can use RFECV.

```
In [49]: from sklearn.model_selection import StratifiedKFold
         from sklearn.feature_selection import RFECV
         from sklearn.svm import SVR

         ada_dummies_X = pd.get_dummies(X) # all features and dummies

         # normalizing the data
         ada_scaled_X = scale(ada_dummies_X)

         # train and test
         X_train, X_test, y_train, y_test = train_test_split(ada_scaled_X,
                                                             y,
                                                             test_size=0.25,
                                                             random_state=4)
```

```
        svr = SVR(kernel="linear") # estimator

        rfecv = RFECV(estimator=svr, step=1, cv=StratifiedKFold(2))
        rfecv.fit(X_train, y_train)

        print("Optimal number of features : %d" % rfecv.n_features_)
```

```
C:\Users\wcarb\Anaconda3\lib\site-packages\sklearn\model_selection\_split.py:605: Warning: The
  % (min_groups, self.n_splits)), Warning)


Optimal number of features : 36
```

```
In [50]: from sklearn.decomposition import PCA

        ada_dummies_X = pd.get_dummies(X)
        print(ada_dummies_X.shape)

        # feature reduction
        pca = PCA(n_components=36) # from rfecv
        pca.fit(ada_dummies_X)
        pca_X = pca.transform(ada_dummies_X)
        print(pca_X.shape)

        # train and test
        X_train, X_test, y_train, y_test = train_test_split(pca_X, y, test_size=0.25, random_s

        ada_boost = AdaBoostRegressor(n_estimators=20, random_state=3)

        ada_boost.fit(X_train, y_train)
        y_predict = ada_boost.predict(X_test)

        # adjust negative predictions
        y_predict = zero_min(y_predict)

        # evaluation
        accuracy_list = cross_val_score(ada_boost, pca_X, y, cv=10)
        accuracy = accuracy_list.mean()
        score = r2_score(y_test, y_predict)

        print("Accuracy:", accuracy)
        print("R2:", score)
```

```
(10886, 57)
(10886, 36)
Accuracy: 0.090969278042014
R2: 0.6321831951759266
```

### 1.11 Bagging Regressor

```
In [51]: from sklearn.ensemble import BaggingRegressor

         bagging_scaled_X = scale(X)
         # train and test
         X_train, X_test, y_train, y_test = train_test_split(bagging_scaled_X,
                                                             y,
                                                             test_size=0.25,
                                                             random_state=4)

         bagging = BaggingRegressor(random_state=1)
         bagging.fit(X_train, y_train)
         y_predict = bagging.predict(X_test)

         # adjust negative predictions
         y_predict = zero_min(y_predict)

         accuracy_list = cross_val_score(bagging, bagging_scaled_X, y, cv=10)
         accuracy = accuracy_list.mean()
         score = r2_score(y_test, y_predict)

         print("Accuracy:", accuracy)
         print("R2:", score)

Accuracy: 0.7880198563266602
R2: 0.9407255807536018
```

## 2 Conclusion

After testing plenty of ML algorithms such as

- Decision Tree Classifier
- Random Forest Classifier
- K-Neighbors Regressor
- Decision Tree Regressor
- Random Forest Regressor
- Linear Regression
- AdaBoost Regressor
- Bagging Regressor

There were a couple of things we noticed throughout our project. The first of which was recognizing that in general, *Classifiers* weren't going to give us the results we hoped for due to our dataset. Our dataset composed of a mixture of **categorical** and **numerical** data, along with some **continuous** data. Because of this, we then changed our strategy by then trying models that used *Regression*. **K-Neighbors** immediately showed us that theses models outperformed the *Classifiers* even though the accuracy for our **K-Neighbors** was only about **27%**. The fact that we did only did

a small amount of feature engineering, meant that it was defintely possible to further manipulate the data to increase our accuracies. After trying the next six different ML algorithms, two of them really stood out with their results. **Random Forest Regressor** achieved a accuracy of **79.6%**, however much analyzes of the data had to be done for such as score. **Bagging** achieved nearly the same score at **78.8%** with only normalizing the data and adjusting negative predictions! This means if enough feature engineering and data manipulation, it is possible to achieve an even high accuracy. However, due time restriction we were unable to implement these additonal factors.

Therefore, in the end with a **79.6%** accuracy, our **Random Forest Regression** model would be the best choice to predict the number of bicycles that may be used throughout the day, on a given hour.

## 2.1 Team Contribution

Walter Carbajal

- performed analyzes on Decision Tree Regressor & Random Forest Regressor
- prepared and carefully edited our powerpoint presentation
- overlooked jupyter notebook editing

Enrique Castillo

- performed analyzes on K-Neighbor Regressor
- prepared the jupyter notebook and thorughly wrote descriptions and explaintions on each ML algorithm analyzes

Erika Estrada

- performed analyzes on Linear Regression, AdaBoost Regressor, and Bagging Regressor
- contributed to the analyzes of our Random Forest Regressor

Vrezh Khalatyan

- performed analyzes on Decision Tree Classifer & Random Forest Classifier
- noted the poor performance on Classifiers and our move to Regression models