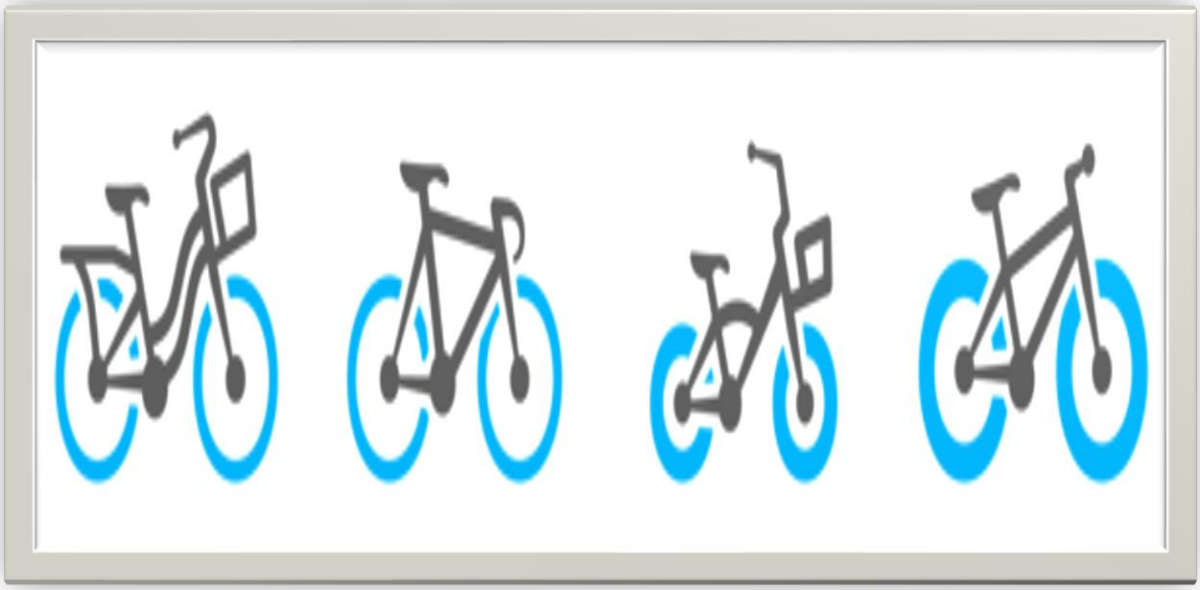


Machine Learning

Predicting Bike Sharing Demand on an Hourly Basis

Professor: Mrs. P. Velardi

Participants:

Samin Hamidi 1732304

Leen Shi 1739246

Contents

Introduction

Bike Sharing Systems

These new and attractive systems that are new generations of traditional bike rentals, turn the process of obtaining membership, rental, and bike return into an automatic one. Kiosks are scattered across the city and each person starts its journey by borrowing a bicycle from them. They should return the bike after reaching their destination. This process is alleviated by Bike Sharing Systems. Currently there are over 500 bike-sharing programs around the world which is composed of over 500 thousand bicycles. Today, there exists great interest in these systems due to their important role in traffic, environmental and health issues.

Data generated by these systems makes them attractive to researchers because the duration of travel, departure location, arrival location, and time elapsed is explicitly recorded. Bike sharing systems therefore function as a sensor network, which can be used for studying mobility in a city.

In this competition, participants are asked to combine historical usage patterns with weather data in order to forecast bike rental demand in the Capital Bikeshare program in Washington, D.C.

Dataset

In this data set, historical usage patterns and weather data are combined. The goal is to predict the demand of bike rental in an hourly basis.

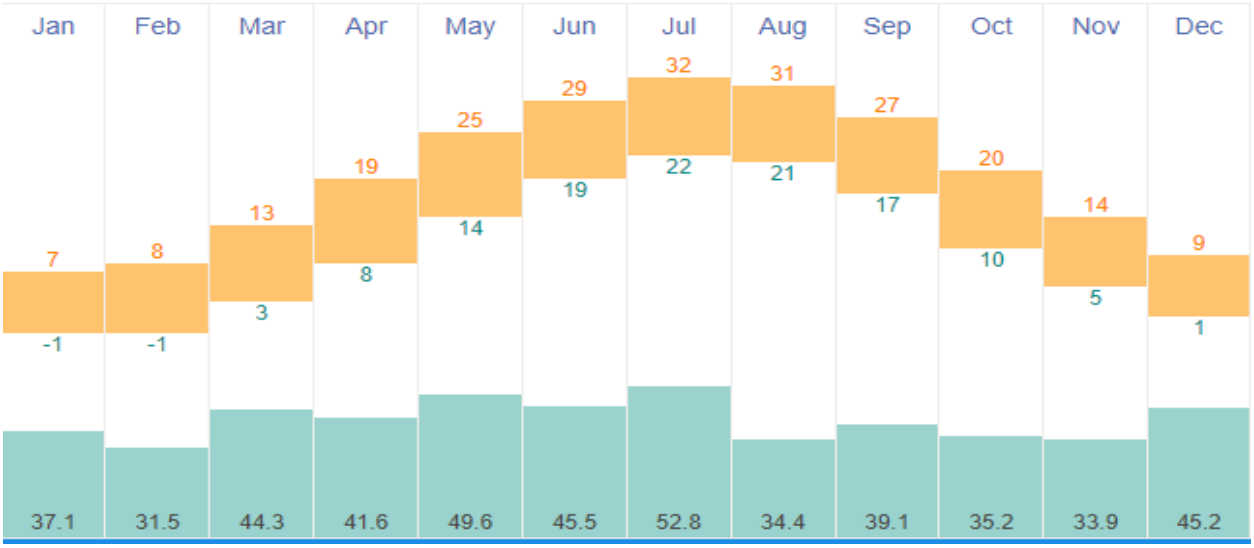
Historical usage patterns are produced by Capital Bikeshare. The city location has been Washington, D.C.

Dataset provides hourly rental data which spans two years. We work with the training set. It has been comprised of the first 19 days of each month.

Weather information has been gathered through this website: http://www.freemeteo.com.

To be able to understand the dataset better we need to have some knowledge about annual weather average of Washington DC and its suburb.

The table below gives us a glimpse of annual weather averages in Washington DC. These values are based on weather reports collected during 1985-2015.



Spring and fall are mild to warm, while winter is chilly with annual snowfall averaging 15.5 inches (39 cm). Winter temperatures average around 38 °F (3 °C) from mid-December to mid-February. Summers are hot and humid with a July daily average of 79.8 °F (26.6 °C) and average daily relative humidity around 66%, which can cause moderate personal discomfort. The combination of heat and humidity in the summer brings very frequent thunderstorms, some of which occasionally produce tornadoes in the area.

Our dataset has 10886 columns (attributes) and 12 rows (instances). Count is the target(dependent) variable. We have four categorical variables in the form of integer. 3 continuous numerical and 4 discrete numerical cases. Datetime is an object.

In [663]:
```
# Reading training set
dataset = pd.read_csv(r"C:\Users\samin\Desktop\Dataset\train.csv")
```

In [664]:
```
#Number of Rows and Columns
print(dataset.shape)
```

```
(10886, 12)
```

In [665]:
```
dataset.head(2)
```

Out[665]:

	datetime	season	holiday	workingday	weather	temp	atemp	humidity	windspeed	casual	registered	count
0	1/1/2011 0:00	1	0	0	1	9.84	14.395	81	0.0	3	13	16
1	1/1/2011 1:00	1	0	0	1	9.02	13.635	80	0.0	8	32	40

In [666]:
```
dataset.dtypes
```

Out[666]:
```
datetime       object
season          int64
holiday         int64
workingday      int64
weather         int64
temp          float64
atemp         float64
humidity        int64
windspeed     float64
casual          int64
registered      int64
count           int64
dtype: object
```

Below, we add the description of each attribute:

Attribute	Description	Type	
datetime	Hourly date+ timestamp	object	
season	1= spring, 2=summer, 3=fall, 4= winter	int64	categorical
holiday	Whether the day is considered a holiday	int64	categorical
workingday	Whether the day is neither a weekend nor holiday	int64	categorical
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	int64	categorical
temp	Temperature in Celsius	float64	Continuous numerical
atemp	"feels like" temperature in Celsius	float64	Continuous numerical
humidity	Relative humidity	int64	Discrete numerical
windspeed	Wind speed	float64	Continuous numerical
Casual	Number of non-registered user rentals initiated	int64	Discrete numerical
Registered	Number of registered user rentals initiated	int64	Discrete numerical
Count	Number of total rentals (Target Variable)	int64	Discrete numerical

What is out task?

We are in charge of predicting bike demand in the future. This is obviously very important for predicting the revenues of the company and planning infrastructure improvements.

Tools

To go through this project, we decided to use **Anaconda**. Anaconda is a free and open source distribution of Python language for data science and machine learning related applications. The goal of Anaconda is simplifying package management.

We installed Python 3.0. **Jupyter notebook** was the mean of documenting our process and experimentations.

We used **scikit learn** extensively. It is a free software machine learning library for the Python programming language and is designed to operate with numerical and scientific libraries **NumPy** and **SciPy**.

Pandas also was used. It is used for data manipulation and analysis. It offers data structures and operations for manipulating numerical tables and time series.

We used **Matplotlib** for plotting and **seaborn** for embellishment of those plot. Matplotlib is a plotting library and its numerical mathematics extension NumPy. Seaborn is a graphic library built on top of Matplotlib. It allows us to make our charts prettier and facilitates some of the common data visualization needs.

In [661]:
```
import numpy as np
import pandas as pd
from math import sqrt

import matplotlib.pyplot as plt
import seaborn as sns
get_ipython().run_line_magic('matplotlib', 'inline')
```

In [662]:
```
from sklearn.cross_validation import train_test_split

from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor, AdaBoostRegressor, BaggingRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.svm import SVR

from sklearn.model_selection import cross_val_score
from sklearn.metrics import mean_squared_error
```

Data Preprocessing

Leakage Attributes

In this dataset we have two leakage variables: Casual and Registered. The task of finding such cases in a database is usually tricky. Leakage variables are those that have been added to the dataset from outside and happens to have the information we want to predict. Here, sum of 'casual' and 'registered' in each row is equal to the value of 'count', our target variable. We need to eliminate these two as their existence make our predictions overly optimistic. In fact, absurd and invalid.

```
In [1515]: #Dropping leakage variables: casual and registered. Their sum will tell us the number of counts(target), so they have to be dropp
           dataset = dataset.drop(["casual"], 1)
           dataset = dataset.drop(["registered"], 1)
```

Missing Values

Missing values in a dataset makes the predictions less exact. We need to check to see if any columns contain any missing values. There are different methods to handle such cases. They can fall into two categories: deletion and imputation. These are all part of data manipulation and data cleaning.

Fortunately, our data set does not contain any missing values.

```
In [1517]: #Checking out to see if we have any missing values in each column?
           dataset.info()
```

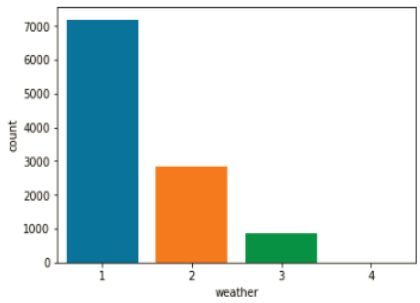
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10886 entries, 0 to 10885
Data columns (total 10 columns):
datetime      10886 non-null object
season        10886 non-null int64
holiday       10886 non-null int64
workingday    10886 non-null int64
weather       10886 non-null int64
temp          10886 non-null float64
atemp         10886 non-null float64
humidity      10886 non-null int64
windspeed     10886 non-null float64
count         10886 non-null float64
dtypes: float64(4), int64(5), object(1)
memory usage: 850.5+ KB
```

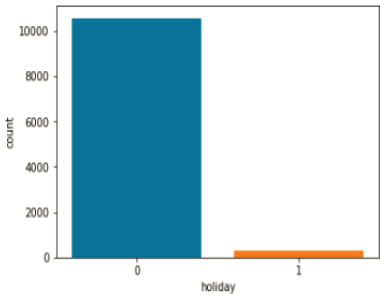
```
In [1518]: dataset.isnull().values.any()
Out[1518]: False
```

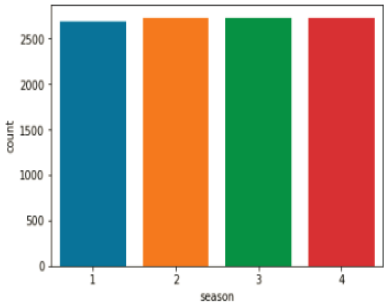
Unbalanced Attributes (Class Imbalance)

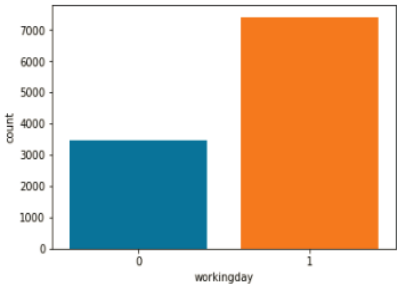
When a column is categorical, and the values of a category outnumber the value of other categories, this attribute can be called an unbalanced attribute and the issue a class imbalance.

We draw the count plot of our categorical attributes to have an idea of their frequency in the dataset.









Among these, holiday is an unbalanced attribute. More than 10000 instances fall under class 0 (not a holiday). An unbalanced attribute in monotone thus do not contribute to predicting the target. We drop this variable.

```
In [1639]: # dropping 'holiday'
           dataset = dataset.drop(["holiday"], 1)
```

Datetime Variable

	datetime	season	holiday	workingday	weather	temp	atemp	humidity	windspeed	casual	re
0	1/1/2011 0:00	1	0	0	1	9.84	14.395	81	0.0	3	
1	1/1/2011 1:00	1	0	0	1	9.02	13.635	80	0.0	8	

In this dataset we have an **object** column called **datatime**. It has its own specific format and contains information about hour, day, month and year. We can extract this information out of datetime by creating four separate columns that contain data about hour, day, month and year respectively.

After creating these columns, we can drop datetime form our dataframe.

In [1532]:
```
# Feature Engineering
# 1) datetime: extracting information out of this feature

dataset['datetime'] = pd.to_datetime(dataset['datetime'])
dataset['Hour'] = dataset['datetime'].apply(lambda time: time.hour)
dataset['Day'] = dataset['datetime'].apply(lambda time: time.dayofweek)
dataset['Month'] = dataset['datetime'].apply(lambda time: time.month)
dataset['year'] = dataset['datetime'].apply(lambda time:time.year)
```

In [1533]:
```
# A glimpse of new variables added
dataset.head(2)
```

Out[1533]:

	datetime	season	workingday	weather	temp	atemp	humidity	windspeed	count	Hour	Day	Month	year
0	2011-01-01 00:00:00	1	0	1	9.84	14.395	81	0.0	2.833213	0	5	1	2011
1	2011-01-01 01:00:00	1	0	1	9.02	13.635	80	0.0	3.713572	1	5	1	2011

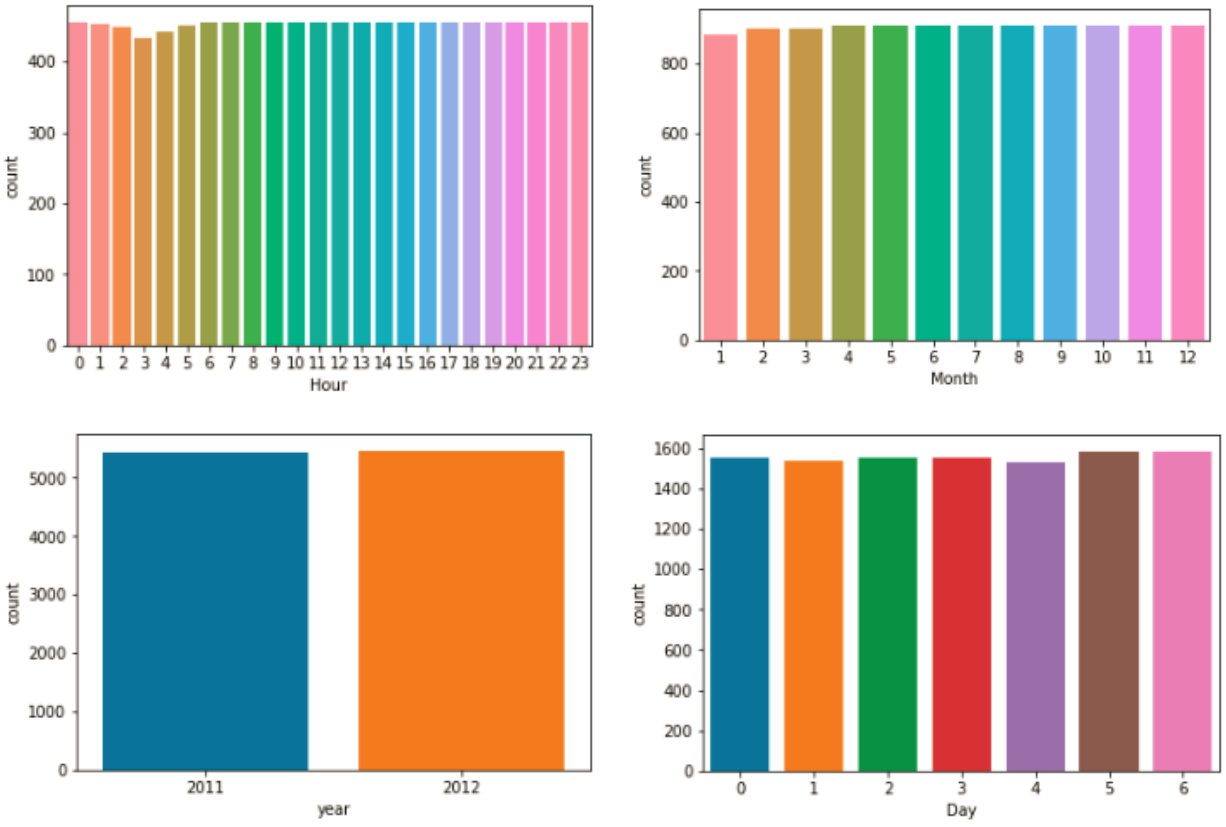
In [1534]:
```
# Useful information has been extracted out of datetime, now we can drop it
dataset = dataset.drop(["datetime"], 1)
```

In [1535]:
```
# Our dataset now:
dataset.head(2)
```

Out[1535]:

	season	workingday	weather	temp	atemp	humidity	windspeed	count	Hour	Day	Month	year
0	1	0	1	9.84	14.395	81	0.0	2.833213	0	5	1	2011
1	1	0	1	9.02	13.635	80	0.0	3.713572	1	5	1	2011

Checking to see if new columns added are balanced or not?



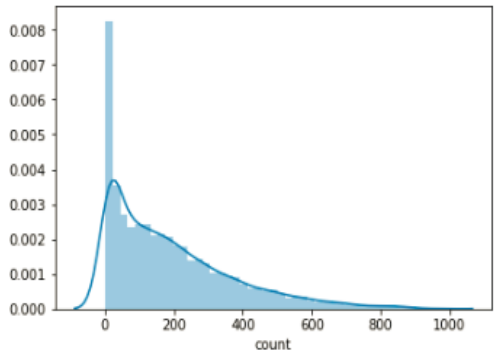
As we see in the count plots, all four new variables are balanced. So, we keep them all.

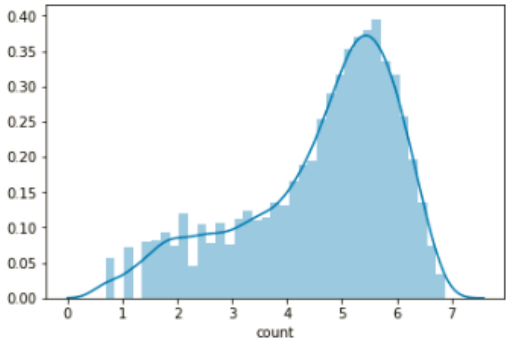
Skewness

Skewness is asymmetry in a statistical distribution, in which the curve appears distorted or skewed either to the left or to the right. Skewness can be quantified to define the extent to which a distribution differs from a normal distribution.

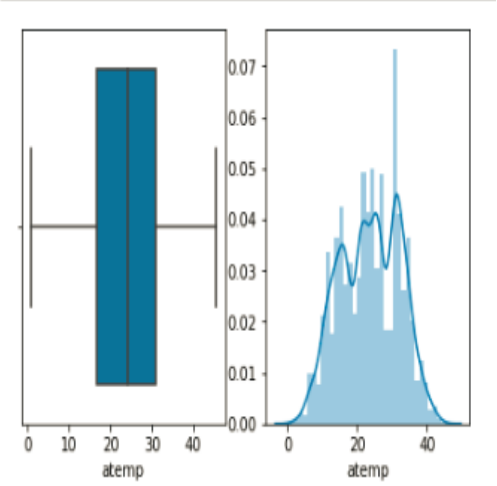
To reduce the skewness, we can perform a log transform. There are other ways to reduce skewness including boxcox which has been provided by scipy. We decided to work with the log transform because it is a straightforward method**.**

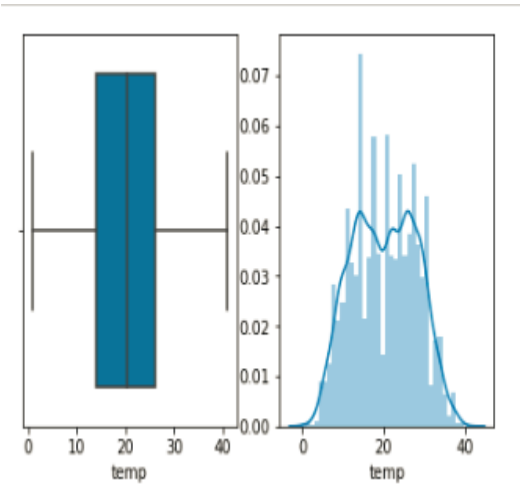
We can check the skewness of our target variable and if its distribution is not normal we will take its log.

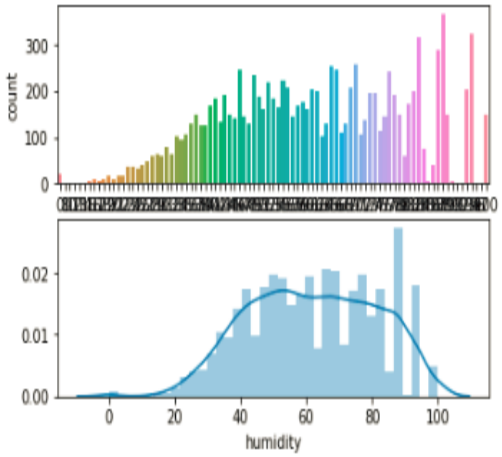


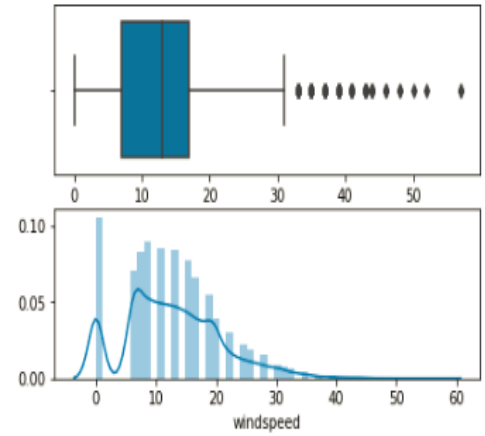


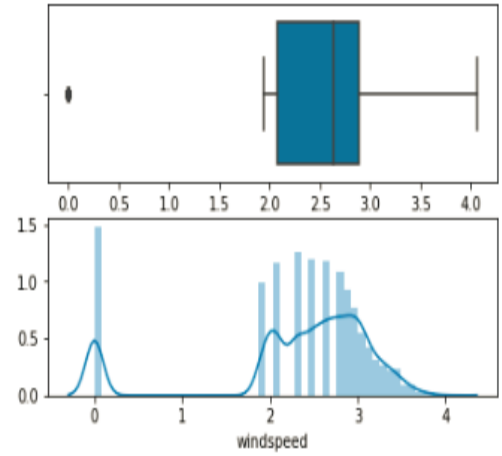
It has been suggested to take log of those skewed independent variables in case the skew of dependent attribute has been calculated. Among our continuous numerical values (temp, atemp, humidity and windspeed), temp, atemp and humidity have an almost normal distribution, so we leave them as they are. We take the log of humidity and windspeed.











In the left, we see the result after taking the log from windspeed column.

Categorical into Numeric

Data in a column can be of numerical or categorical type. Categorical attributes are those divided into categories (levels). Categories may be nominal or ordinal. Ordinal levels follow a logical order. In nominal categories finding such an order is difficult.

In our dataset, we have four categorical features: season, weather, holiday and workingday. After dropping 'holiday' due to its class imbalance, the rest are remaining.

Their levels are provided in the form of integer values instead of string names. We decided to map these levels into their names and then use a method to turn them into numerical data.

First, we turn them into their intuitive form, replacing each category with its name. For example: 1: "spring", 2: "Summer", 3: "Fall", 4: "Winter". Then use one-hot encoding. Pandas provides the very useful **get_dummies** method on a DataFrame, which does what we want.

One-hot encoding is the default way of turning categorical data into numeric. With this method we encode the categorical variable as a **one-hot vector**, i.e. a vector where only one element is non-zero, or *hot*. With one-hot encoding, a categorical feature becomes an array whose size is the number of possible choices for that features.

Another method called label-encoding also exist, but it is used when categories of one attribute follow a logical order. In those algorithms where the weight of each attribute matters, this form of labeling becomes important. For other models like decision tree, labeling does not make a difference.

For each categorical variable we create a one-hot encoded data frame. Then concatenate them all to our dataset. After this step, we can drop the original columns: season, workingday and weather.

In [697]:
```
# We have three categorical (in form of integer) variables: Season (4 levels), workingday(2 levels) and weather(4 levels)
# We can turn them into dummy variables (one-hot encoding) (the standard method of turning categorical data into numerical)
# First, for each categorical attribute we create a new one-hot encodded dataframe and then at the end concatanate them together.

dummy1 = pd.get_dummies(dataset['season'])
dummy2 = pd.get_dummies(dataset['workingday'])
dummy3 = pd.get_dummies(dataset['weather'])

dataset = pd.concat([dataset, dummy1, dummy2, dummy3], axis=1)

dataset.head(10)
```

Out[697]:

	season	workingday	weather	temp	atemp	humidity	windspeed	count	Hour	Day	...	Fall	Spring	Summer	Winter	Workingday	off-day	awful	excel
0	Spring	off-day	excelent	9.84	14.395	81	0.000000	2.833213	0	5	...	0	1	0	0	0	1	0	
1	Spring	off-day	excelent	9.02	13.635	80	0.000000	3.713572	1	5	...	0	1	0	0	0	1	0	
2	Spring	off-day	excelent	9.02	13.635	80	0.000000	3.496508	2	5	...	0	1	0	0	0	1	0	
3	Spring	off-day	excelent	9.84	14.395	75	0.000000	2.639057	3	5	...	0	1	0	0	0	1	0	
4	Spring	off-day	excelent	9.84	14.395	75	0.000000	0.693147	4	5	...	0	1	0	0	0	1	0	
5	Spring	off-day	good	9.84	12.880	75	1.946367	0.693147	5	5	...	0	1	0	0	0	1	0	
6	Spring	off-day	excelent	9.02	13.635	80	0.000000	1.098612	6	5	...	0	1	0	0	0	1	0	
7	Spring	off-day	excelent	8.20	12.880	86	0.000000	1.386294	7	5	...	0	1	0	0	0	1	0	
8	Spring	off-day	excelent	9.84	14.395	75	0.000000	2.197225	8	5	...	0	1	0	0	0	1	0	
9	Spring	off-day	excelent	13.12	17.425	76	0.000000	2.708050	9	5	...	0	1	0	0	0	1	0	

10 rows × 22 columns

In [698]:
```
#Drop season, holiday, workingday and weather

dataset = dataset.drop(["season"],1 )
dataset = dataset.drop(["workingday"],1 )
dataset = dataset.drop(["weather"],1 )

dataset.head(2)
```

Out[698]:

	temp	atemp	humidity	windspeed	count	Hour	Day	Month	year	Fall	Spring	Summer	Winter	Workingday	off-day	awful	excelent	good	not-good
0	9.84	14.395	81	0.0	2.833213	0	5	1	2011	0	1	0	0	0	1	0	1	0	0
1	9.02	13.635	80	0.0	3.713572	1	5	1	2011	0	1	0	0	0	1	0	1	0	0

Univariate Analysis of Dependent vs. Independent Attributes

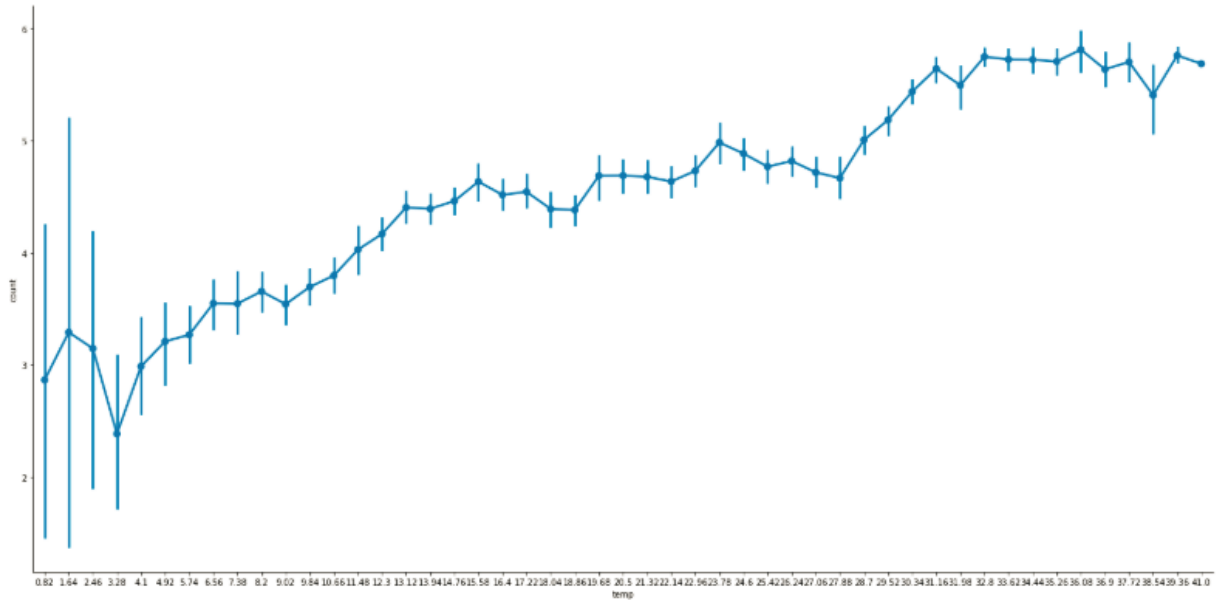
Analyzing the relationship between independent variables and count (dependent variable)

Through matplotlib and seaborn library we can visualize the role of each independent variable on the dependent attribute. Plotting makes understanding data easier.

To draw these plot, we use seaborn's factor plot:

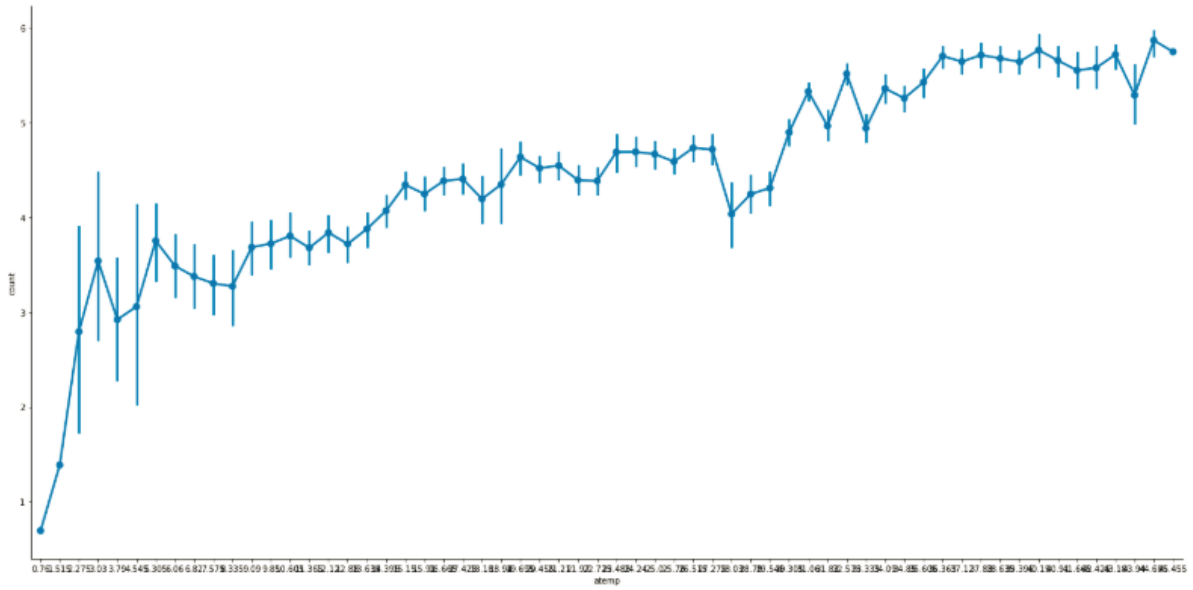
```
In [1129]: sns.factorplot(x="temp", y="count",
                          data=dataset, kind="point", size=10, aspect=2)
```

Temp



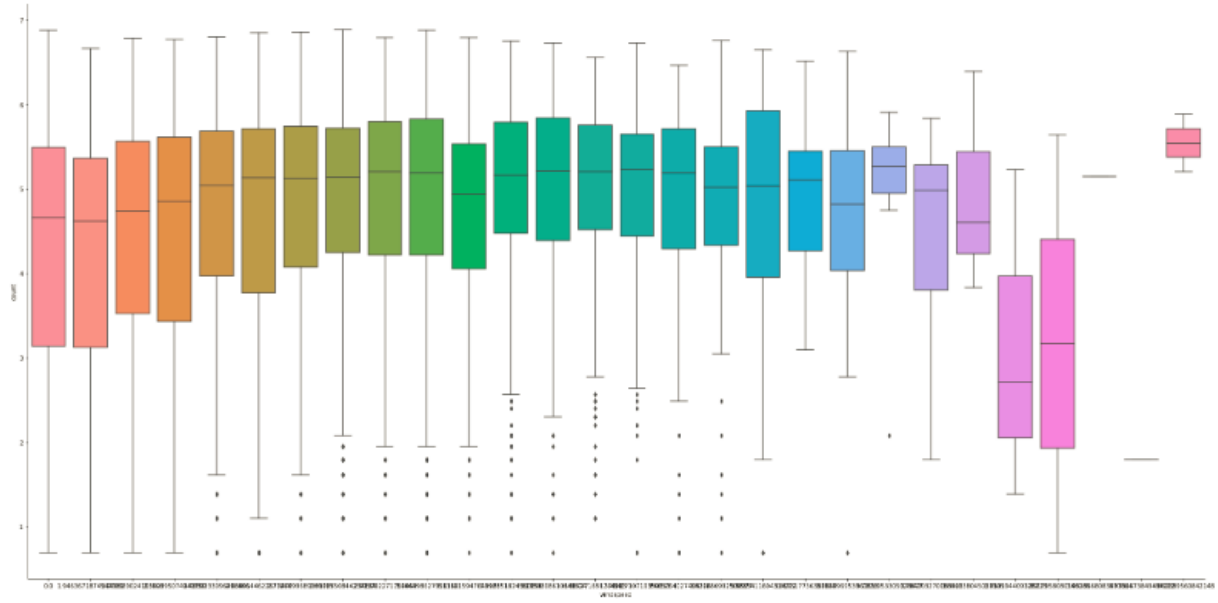
This plot shows there is a positive linear relationship between temp and count. The higher the temperature gets, the higher is the demand.

Atemp



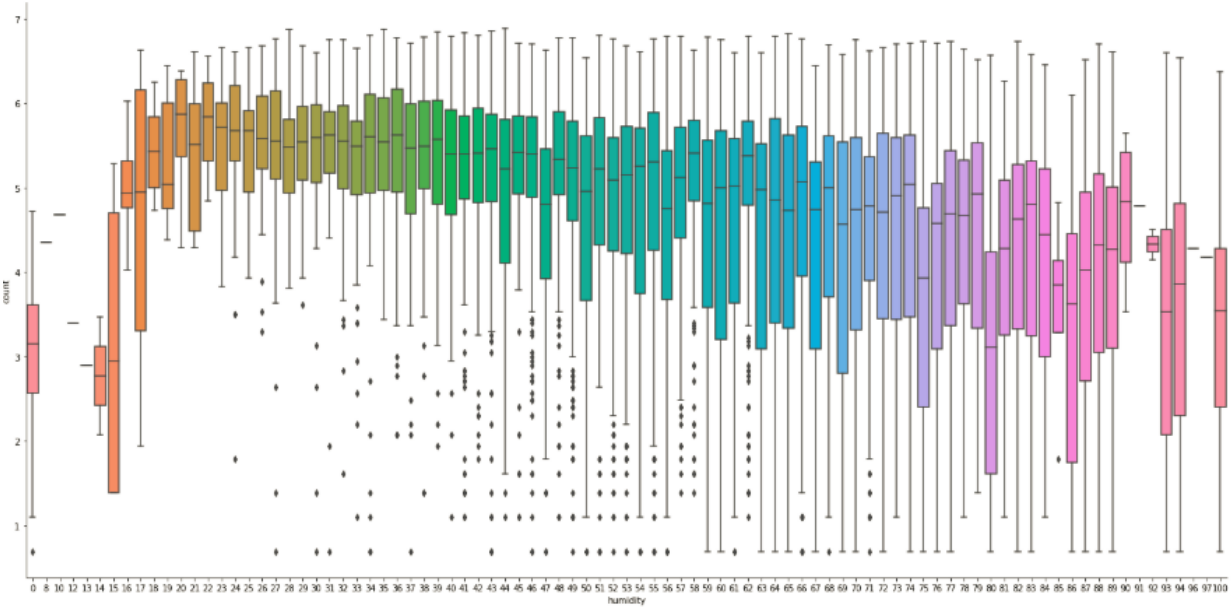
Atemp also shows a similar linear relationship to count like temp.

Windspeed



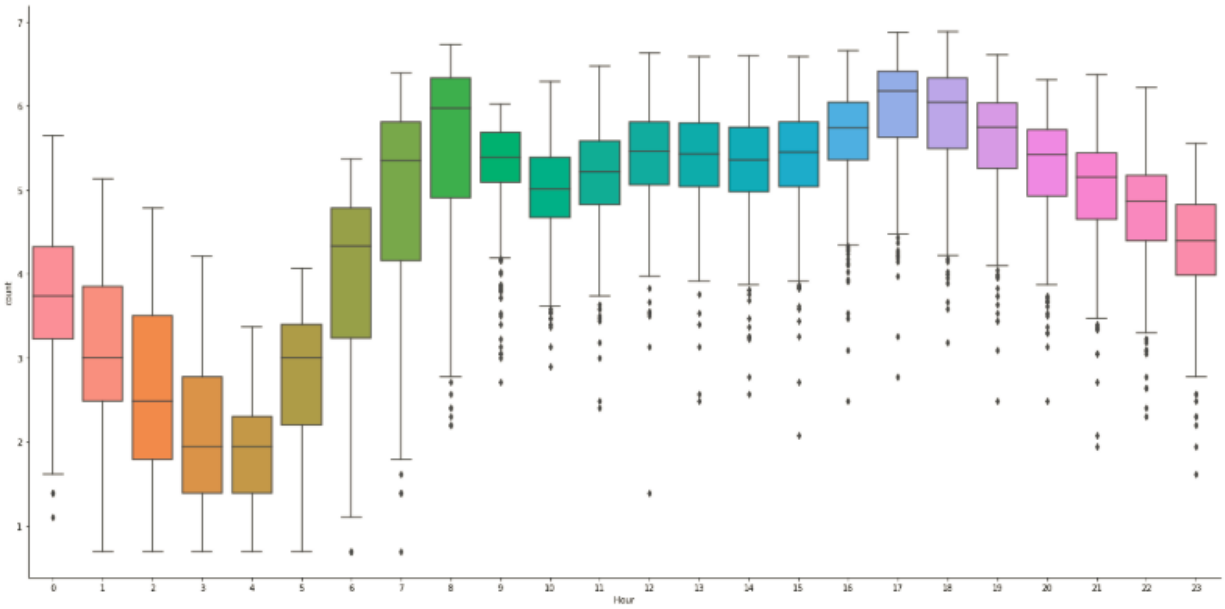
We cannot infer much from this graph. Just when the windspeed is very high, the demand experiences a decrease.

Humidity



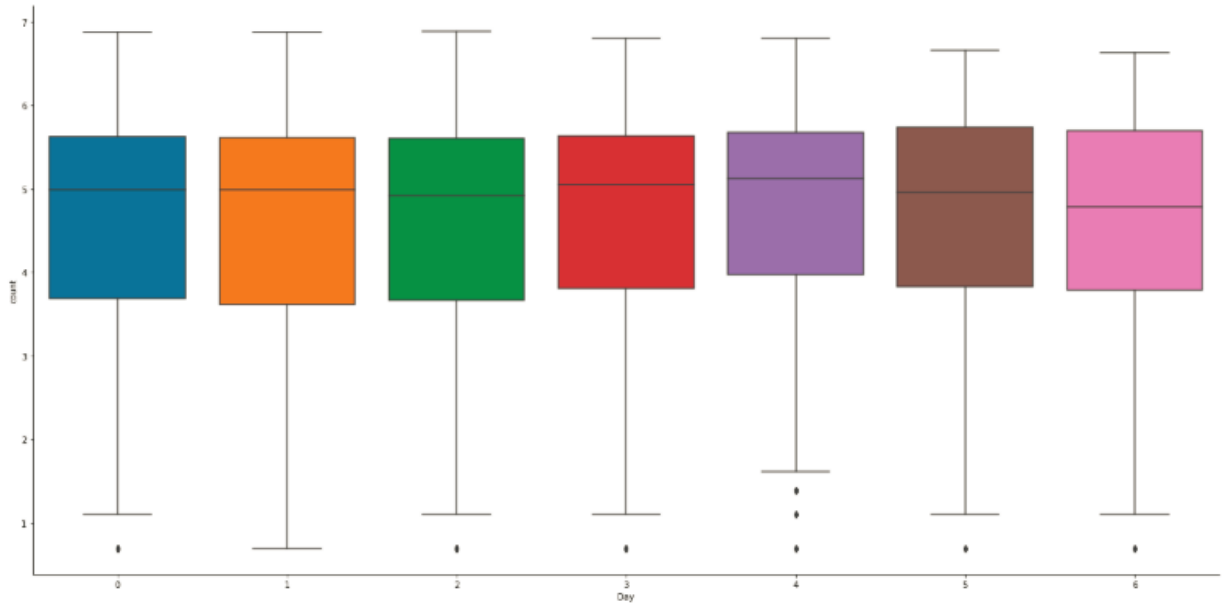
Humidity has a slightly negative relationship with target variable count. We wait until drawing the correlation matrix and observing the correlation value.

Hour



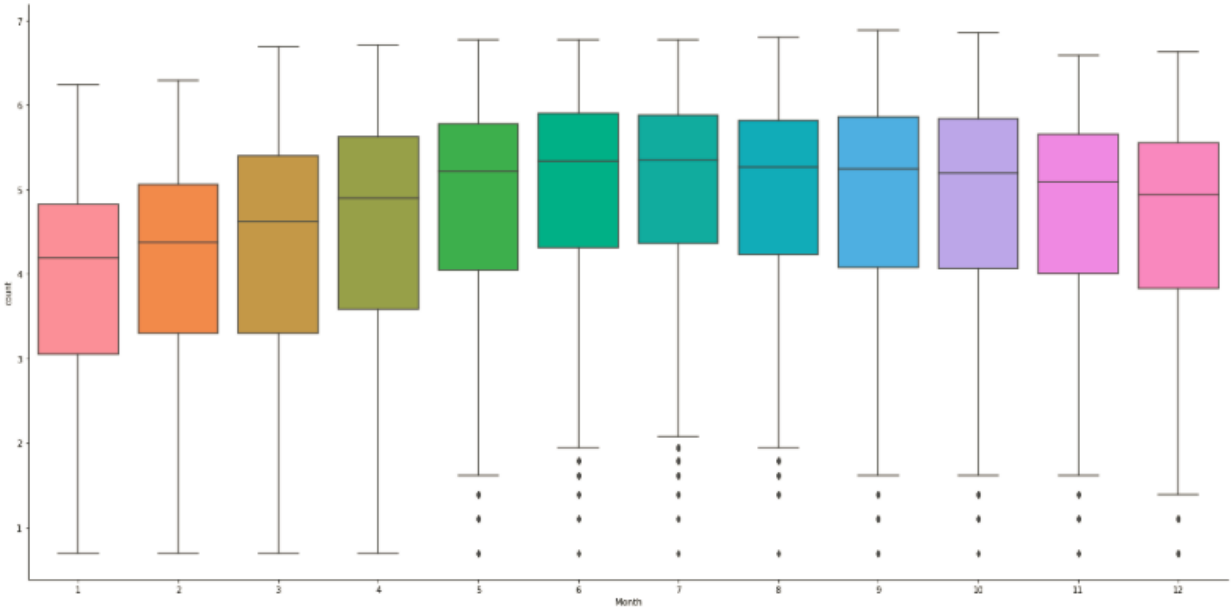
We can observe from this plot that in the early hours in the morning (6, 7 and 8) the demand for bikes sees an increase. This makes sense as people wants to commute to work and they need a mean to get there. Also, in the afternoon (16, 17 and 18),we see a higher demand. In these hours people are usually coming back home so, looking for a vehicle to get them there.

Day



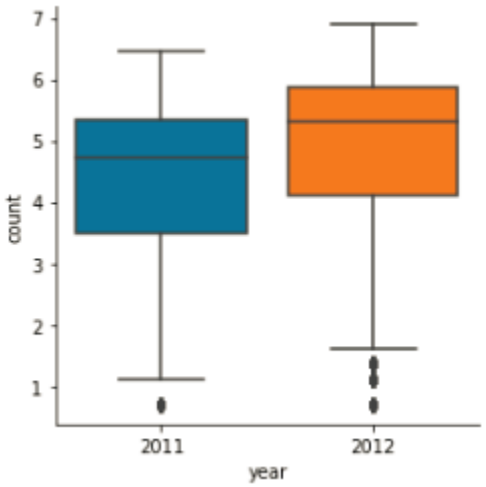
This plot doesn't show much.

Month



we cannot see a sharp increase in the demand in months 4 to 10. We can say that in the first three months and last two months the demand is lower. It makes sense because in these months climate is severe.

Year



In year 2012, the demand sees an increase. We can interpret it this way: people are becoming more familiar with bike sharing systems and seeing them as a convenient way of commuting throughout the city.

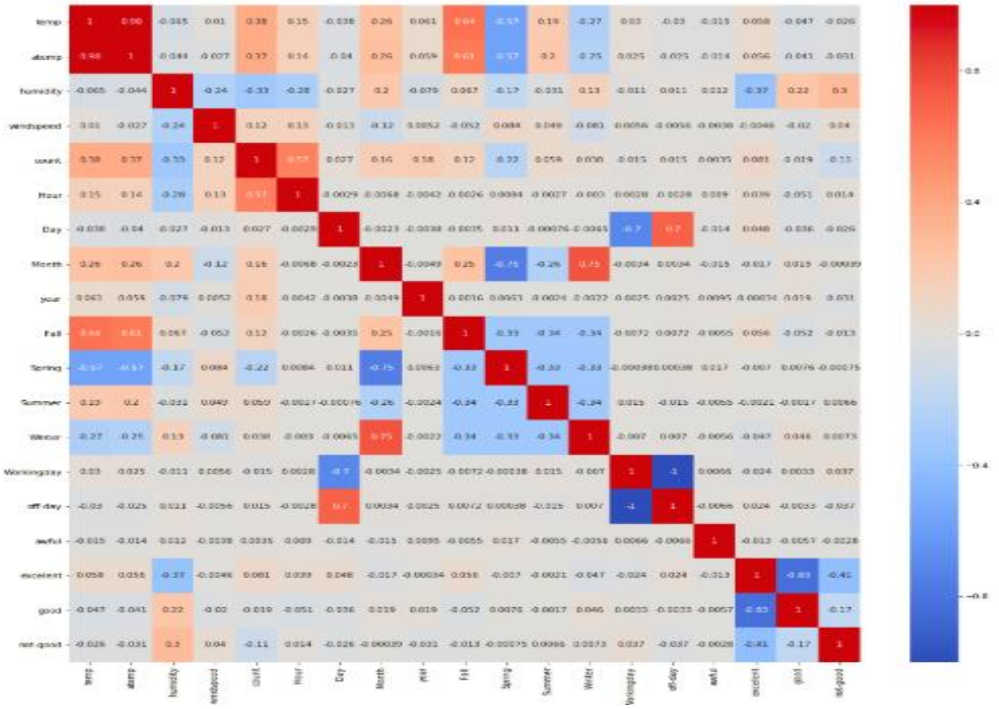
Correlation and Multicollinearity

The term correlation refers to a mutual relationship or association between quantities. There are several methods for calculating the correlation, each measuring different types of strength of association.

It is important to understand possible correlations in the data (very important in regression models). Strongly correlated predictors, phenomenon referred to as **multicollinearity**, will cause coefficient estimates to be less reliable.

We use Pandas and seaborn to create a correlation matrix and then a heatmap. DataFrame.corr () supports different correlation methods. We choose the default method which is pearson correlation coefficient. Pearson is a measure of the linear correlation between two variables *X* and *Y*. It has a value between +1 and −1, where 1 is total positive linear correlation, 0 is no linear correlation, and −1 is total negative linear correlation.

After creating the correlation matrix, we use seaborn.heatmap() which plots rectangular data as a color-encoded matrix.



Observing the correlation between dependent and all the independent variables, we see that the highest correlation value is 0.38 (count-temp). This is not a very high correlation and we can see that our model is not a linear model. However, we decided to apply LinearRegression Model in the modeling phase, but our expectation is not high.

Multicollinearity

We drop those columns that have a multicollinearity >0.80.

```
In [1154]: #temp and atemp: We drop atemp
dataset = dataset.drop(["atemp"], 1)
#High Multi-collinearity between good and pleasant(-0.83). We drop good
dataset = dataset.drop(["good"], 1)

dataset.head(2)
```

Out[1154]:

	temp	humidity	windspeed	count	Hour	Day	Month	year	Fall	Spring	Summer	Winter	Workingday	off-day	awful	excelent	not-good
0	9.84	81	0.0	2.833213	0	5	1	2011	0	1	0	0	0	1	0	1	0
1	9.02	80	0.0	3.713572	1	5	1	2011	0	1	0	0	0	1	0	1	0

Dropping count

Before starting modeling, we need to keep the target variable in a separate column and then drop 'count' from dataset.

```
In [1155]: #Preparing our dataset for model selection
#Keep target variable 'count' in a column called y and then drop it from dataset.
y=dataset['count']
dataset = dataset.drop(["count"], 1)
```

Model Selection

Model selection is the task of selecting a ML model from a set of candidate models, given the data.

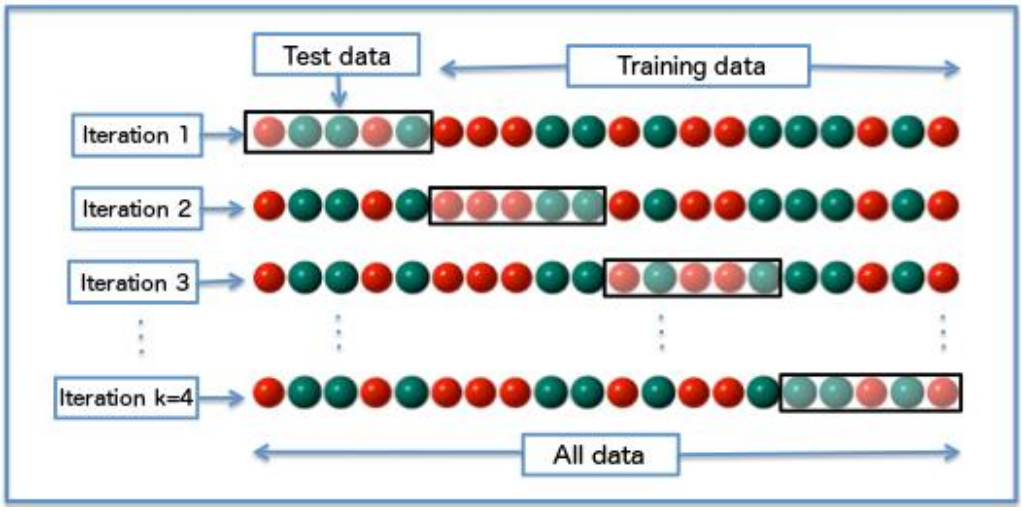
Now, to be able to select a model, we first need a way to evaluate it. Cross-validation is a technique to evaluate predictive models by partitioning the original sample into a training set to train the model, and a test set (validation set) to evaluate it.

There are different methods to do cross validation. Two most prominent ones are Train_Test split and K-fold. In the former, dataset is divided into a training and a test set. This takes place in a random way. The percentage of training set to testing set is usually 80 to 20 or 70 to 30.

In k-fold **cross-validation**, the original sample is randomly partitioned into k equal sized subsamples. Evaluation score is calculated for each division and them the mean of the results is returned.

Each method has their own advantages. Train_Test split is useful when we have computational constraints. Some of the algorithms that avoid overfitting like Random Forest and Lasso work well with this way of cross validation. But, if we can afford it, k-fold cross validation gives us a more reliable result. Iteration on the training and testing process multiple times, changing the train and test dataset distribution, helps in validating the model effectiveness properly.

Picture below shows how k divisions take place.



We choose K-Fold as our cross-validation model.

The question that arises here is how to choose k?

There exists a paper by Ron Kohavi (1995) that investigates this subject. The basic idea is that a lower K is usually cheaper and more biased. Larger K is more expensive, less biased, but can suffer from large variability. This is often cited with the conclusion to use k=10.

We choose k=5 to reduce the computational price.

Evaluation Metrics

To calculate the error (distance between predicted and actual target value) we use two metrics: MSE (Mean Squared Error) and RMSE (Root Mean Squared Error).

MSE is one of the metrics provided by sckit learn's cross_validation library. By taking the root we have RMSE.

MSE

MSE (Mean Squared Error) measures the average squared difference between the estimated values and what is estimated. MSE is a loss function, corresponding to the **expected value** of the squared error loss. MSE is a measure of the quality of an estimator—it is always non-negative, and values closer to zero are better.

$$MSE = \frac{\sum_{i=1}^{N}(Predicted_i - Actual_i)^2}{N}$$

MSE incorporates both the variance of the estimator (how widely spread the estimates are from one data sample to another) and its bias (how far off the average estimated value is from the truth). We can use

this metric to evaluate the same mode on different datasets or to evaluate different models on the same dataset.

To compute MSE we use **sklearn.metrics.mean_squared_error**.

RMSE

RMSD is the square root of the average of squared errors.

$$RMSE = \sqrt{MSE}$$

The effect of each error on RMSD is proportional to the size of the squared error; thus, larger errors have a disproportionately large effect on RMSD.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N}(Predicted_i - Actual_i)^2}{N}}$$

RMSD is a measure of accuracy, to compare forecasting errors of different models for a particular dataset and not between datasets, as it is scale-dependent (unlike MSE which encompasses both bias and variance of an estimator).

RMSD is always non-negative, and a value of 0 (never achieved in practice) would indicate a perfect fit to the data. In general, a lower RMSD is better than a higher one.

Algorithms

Linear Regression

To be able to predict the value of a dependent variable we are looking to find the relationship between a dependent and one or more independent variables. The Former is called simple linear regression and the latter, multiple linear regression. Multiple linear regression is the most common method of linear analysis used and it is the case with our model.

There are two main ways to perform linear regression in Python — with Statsmodels and scikit-learn. It is also possible to use the Scipy library. The last one is not as common as the other two.

We use scikit-learn linearRegression from linear_model.

One of the methods defined for LinearRegression is .score. It returns R-squared or the coefficient of determination. R-squared is the percentage of the response variable variation that is explained by a linear model. In general, the higher the R-squared, the better the model fits your data.

R-squared has its own limitations. It *cannot* determine whether the coefficient estimates, and predictions are biased, which is why we must assess the residual plots (MSE and RMSE metrics). R-squared does not indicate whether a regression model is adequate. You can have a low R-squared value for a good model, or a high R-squared value for a model that does not fit the data!

***class* sklearn.linear_model.LinearRegression(*fit_intercept=True*, *normalize=False*, *copy_X=True*, *n_jobs=1*)**

K-Nearest Neighbors Regressor

The basic **Nearest Neighbor** (NN) algorithm is a simple Algorithm. It can be used both for classification and regression tasks. The intuition behind NN is this: similar examples should have similar outputs.

K-Nearest Neighbors is a slightly different version of NN. In KNN, we are given a sample x and predict by finding the k nearest samples (neighbors) in the algorithm. The advantage of KNN compared to NN is that it handles outliers well. Basic NN doesn't perform well with outliers because it has high variance, meaning that its predictions can vary a lot depending on which samples appears in the dataset.

To do classification, after finding the k nearest samples, KNN takes the most frequent label of their labels. For regression, it is possible to take the **mean** or **median** of the k neighbors or solve a linear regression problem on the neighbors.

This algorithm works very well when our feature space has low dimensions and we have enough data (this is the case with our dataset here) because we have enough nearby points to get a good answer. When the dimension rises, the performance becomes worst, this is due to the fact that the distance measure loses its meaning when the dimensions increases significantly.

In scikit learn we have the class and the function KNeighborsRegressor. We apply the model with its default values.

***class* sklearn.neighbors.KNeighborsRegressor(*n_neighbors=5*, *weights='uniform'*, *algorithm='auto'*, *leaf_size=30*, *p=2*, *metric='minkowski'*, *metric_params=None*, *n_jobs=1*, ***kwargs*)**

Parameters	Value	Description	
n_neighbors	5	Number of neighbors to use by default for kneighbors queries.	
weights	'uniform'	Weight function used in prediction possible values. 'unifoem': all points in each neighborhood are weighted equally. 'distance': weight points by the inverse of their distance.	
algorithm	'auto'	Algorithm used to compute the nearest neighbors. 'auto' will attempt to decide the most appropriate algorithm based on the values passed fit method.	

Decision Tree Regressor

Decision tree builds regression or classification models in the form of a tree structure. It breaks down a dataset into smaller and smaller subsets while at the same time an associated decision tree is incrementally developed. The result is a tree with **decision nodes** and **leaf nodes**. A decision node has two or more branches, each representing values for the attribute tested. Leaf node represents a decision on the numerical target. The topmost decision node in a tree which corresponds to the best predictor called **root node**.

Decision tree is a rule-based technique. The prediction is done by applying a cascade of rules. This helps the model interpretability.

Decision trees can handle complex relationships (not only simple linear relationships).

***class* sklearn.tree.DecisionTreeRegressor(*criterion='mse', splitter='best', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=None, random_state=None, max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, presort=False*)**

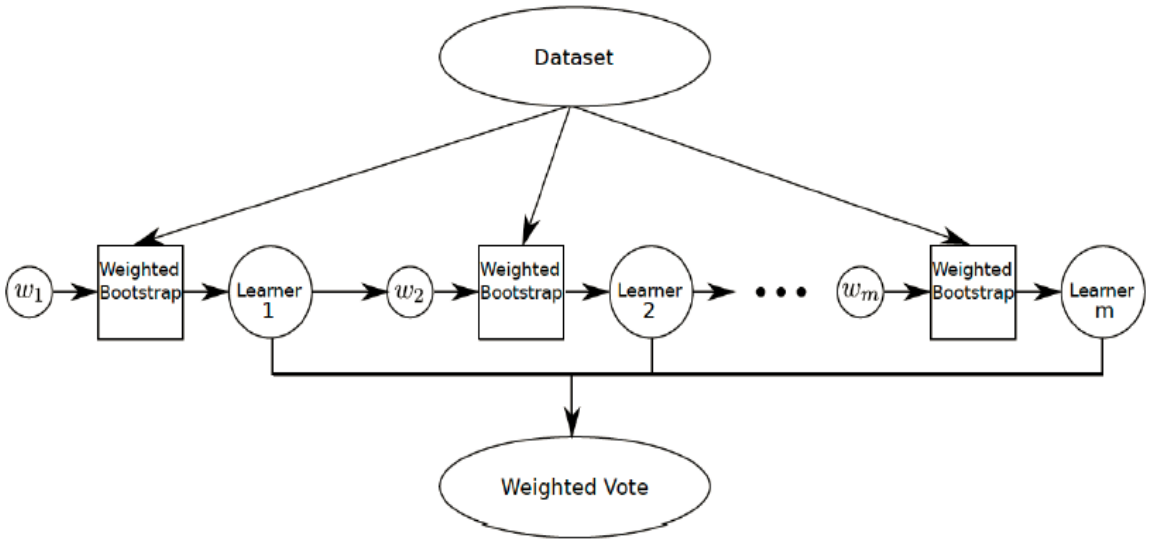
Parameters	Default Value	Description
criterion	mse	The function to measure the quality of a split. Mse is the mean squared error and is equal to variance reduction as feature selection criterion
splitter	best	The strategy used to choose the split at each node.
max_depth	None	The maximum depth of a tree. If none, then nodes are expanded until all leaves are pure or until all

		leaves contain less than min_samples_split samples.
min_samples_split	2	The minimum number of samples required to split an internal node.
Min_samples_leaf	1	The minimum number of samples required to be at a leaf node.
Max_features	None	The number of features to consider when looking for the best split. If None, then max_features=n_features
min_impurity_decrease	0	A node will be split if this split induces a decrease of the impurity greater than or equal to this value.
presort	False	Whether to presort the data to speed up the finding of the best splits in fitting. If the dataset is large, setting this parameter to true will slow down the training process. If the dataset is small or depth is restricted, this may speed up the training operation.

AdaBoost Regressor

Ada Boost Regressor is an ensemble method, a meta-estimator, that first fits a regressor on the original dataset.

Then additional copies of the regressor are fit on the same dataset in which the weights of the instances are adjusted according to the error of the current prediction. As such, subsequent regressors focus more on the difficult classes.



For this dataset we use Ada boost with all its default parameters:

***class* sklearn.ensemble.AdaBoostRegressor(*base_estimator=None*, *n_estimators=50*, *learning_rate= 1.0*, *loss='linear'*, *random_state=None*)**

Parameters	Default Value	Description
base_estimator	DecisionTreeRegressor	The base estimator from which the boosted ensemble is built. The base estimator chosen must support sample weighting.
n_estimators	50	The maximum number of estimators at which boosting is terminated. In case of perfect fit, the learning procedure is stopped early.
learning_rate	1	Learning rate shrinks the contribution of each regressor by learning_rate. There is a trade-off between this parameter and n_estimators.
loss	linear	The loss function to use when updating the weights after each boosting iteration.

Bagging Regressor

Bagging Regressor is an ensemble **meta-estimator** that fits the **base estimator** on **a random** subset of the original dataset. Then the algorithm aggregates their individual predictions (either by voting or by averaging) to form a final prediction.

The randomization and aggregation are ways to reduce the variance of a black-box estimator (e.g., a decision tree).

We use Bagging Regressor model with all its default parameters.

***class* sklearn.ensemble.BaggingRegressor(*base_estimator=None*, *n_estimators=10*, *max_samples=1. 0*, *max_features=1.0*, *bootstrap=True*, *bootstrap_features=False*, *oob_score=False*, *warm_start=Fals e*, *n_jobs=1*, *random_state=None*, *verbose=0*)**

Parameters	Default Value	Descriptions
base_estimator	None	The base estimator to fit on random subsets of the dataset. If None, then the base estimator is a decision tree.
n_estimators	10	The number of base estimators in the ensemble.
max_samples	1.0 (int or float)	The number of samples to draw from X to train each base estimator.
max_features	1.0	The number of features to draw from X to train each base estimator.
bootstrap	True	Whether samples are drawn with replacement.
Bootstrap_features	False	Whether features are drawn with replacement.

Random Forest

One of the ensemble methods which is designed specifically for decision tree regressors or classifiers. Here, as our target variable is a numerical entity, our random forest is a random forest regressor.

Random forest introduces two sources of randomness:

1) Each tree in the ensemble is built from a sample drawn with replacement (i.e., a bootstrap sample) from the training set.
2) When splitting a node during the construction of the tree, the split that is chosen is no longer the best split among all features. Instead, the split that is picked is the best split among a random subset of the features.

What is the effect of this randomness?

Because of this randomness, the bias of the forest usually slightly increases (with respect to the bias of a single non-random tree) but, due to averaging, its variance also decreases, usually more than compensating for the increase in bias, hence yielding an overall better model.

We use this algorithm with its default parameters set in sckit-learn.

Below, we create a table that contains some of the most important parameters embedded in the RandomForest implementation of sklearn.ensemble.

***class* sklearn.ensemble.RandomForestRegressor(*n_estimators=10*, *criterion='mse'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features='auto'*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *bootstrap=True*, *oob_score=False*, *n_jobs=1*, *random_state=None*, *verbose=0*, *warm_start=False*)**

Parameters	**Default values**	**Description**
n_estimators	10	The number of trees in the forest
criterion	mse	The function to measure the quality of a split
max_features	auto	The number of features to consider when looking for the best split If "auto", then max_features=n_features
max_depth	None	The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.
Min_samples_leaf	2	The minimum number of samples required to split an internal node
min_samples_leaf	1	The minimum number of samples required to be at a leaf node
bootstrap	True	Whether bootstrap samples are used when building trees.

To calculate the results, we create a list of all the models we wish to test. Then use cross_val_score to evaluate a score by cross validation. We designate the score object with scoring parameter. All scorer objects follow the convention that higher return values are better than lower return values. Thus metrics that measure the distance between the model and the data like mean_squared_error, are available as neg_mean_squared error. We add the – sign before cross_val_score to get positive results.

In [1269]:
```
#Model selection: Cross-validation
#K-Fold

Algorithms = [LinearRegression(),KNeighborsRegressor(),DecisionTreeRegressor(),BaggingRegressor(),AdaBoostRegressor(), RandomFore
Algorithms_names = [ 'LinearRegression','KNeighborsRegressor','DecisionTreeRegressor','BaggingRegressor','AdaBoostRegressor','Ran

MSE1 = []
RMSE1 = []
Results1 = {}

for Algorithm in range (len(Algorithms)):
    Alg = Algorithms[Algorithm]
    mse = -cross_val_score(Alg, dataset, y, scoring="neg_mean_squared_error", cv = 5)

    MSE1.append(np.mean(mse))
    RMSE1.append(sqrt(np.mean(mse)))


Results1 = {'Modelling Algorithm':Algorithms_names,'MSE':MSE1, 'RMSE':RMSE1}

#Print the result grid
frame = pd.DataFrame(Results1)
frame
```

Out[104]:

	Modelling Algorithm	MSE	RMSE
0	LinearRegression	1.043708	1.021620
1	KNeighborsRegressor	0.613556	0.783298
2	DecisionTreeRegressor	0.347874	0.589809
3	BaggingRegressor	0.233202	0.482910
4	AdaBoostRegressor	0.500135	0.707202
5	RandomForestRegressor	0.234216	0.483959

As the table of results show, RandomForest has the best fit (0.227844). BaggingRegressor (0.237630) and DTRegressor (0.343582) stand in the 2nd and 3rd place respectively. As we expected, LinearRegression model does not perform well on our dataset since there is not much of a linear relationship between independent variables and the target 'count'. KNeighborsRegressor also does not give us a good result compared to the first three. We suspect the reason is the number of columns. For,

the higher the number of dimensions, the more difficult it becomes to make sense out of the concept of distance.

AdaBoostRegressor was the model that disappointed us. We expected a better result from it (comparing it with DecisionTreeRegressor and BaggingRegresor), considering the mechanism of algorithm. We need more investigation and further research to be able to understand what going on underneath.

Hyperparameter Tuning

While model *parameters* are learned during training — such as the slope and intercept in a linear regression — ***hyper-parameters*** must be set before training. All models have a set of sensible default hyper-parameters implemented by Scikit-Learn but these are not guaranteed to be optimal for a problem. The process of selecting the values for a model's hyperparameters in a way that performance increases is called **Parameter Tuning**. In machine learning, parameter tuning, and cross validation tasks are commonly done at the same time in data pipelines.

To perform parameter tuning in python, we leverage the power of scikit-learn again. Scikit-learn provides GridSearchCV which belongs to model_selection **class**. It stands for **grid search cross validation**. Grid search will try all combinations of parameter values and select the set of parameters which provides the most accurate model. GridSearchCV implements a "fit" and a "score" method. Also, other methods such as "predict" if they are implemented in the estimator used. By default, the GridSearchCV's cross validation uses 3-fold KFold or StratifiedKFolddepending on the situation.

Table below shows some of the parameters in GridsearchCV.

Parameters	**Default Value**	**Description**
estimator	No default values. An estimator object must be provided.	This is assumed to implement the scikit-learn estimator interface. Either estimator needs to provide a score function, or scoring must be passed.
param_grid	No default value. A dict or list of dictionaries must be passed to it.	Dictionary with parameters names (string) as keys and lists of parameter settings to try as values, or a list of such dictionaries
scoring	None	

For the most common use cases, we can designate a scorer object with the scoring parameter. All scorer objects follow the convention that higher return values are better than lower return values. Thus, metrics which measure the distance between the model and the data, like metrics.mean_squared_error, are available as neg_mean_squared_error which return the negated value of the metric.

RandomForest

In [105]:
```
from sklearn.grid_search import GridSearchCV

# Create the parameter grid
param_grid = {
    'max_depth': [80, 90, 100],
    'n_estimators': [100, 200, 300]
}

# Model
RF = RandomForestRegressor()

# Instantiate the grid search model
grid_search = GridSearchCV(estimator = RF, param_grid = param_grid, cv = 5, scoring='neg_mean_squared_error')
grid_search.fit(dataset, y)

# View the accuracy score (Scoring scheme for RF algorithm is R squared measure: a measure that tells us how close the data are t
print('Best score for our dataset:', -grid_search.best_score_)

# View the best parameters for the model found using grid search
print('Best depth:',grid_search.best_estimator_.max_depth)
print('Best estimators:',grid_search.best_estimator_.n_estimators)
```

```
Best score for our dataset: 0.2164171135224278
Best depth: 80
Best estimators: 100
```

BaggingRegressor

In [106]:
```
# Create the parameter grid
param_grid = {
    'n_estimators': [10, 20, 30, 40]
}

# Model
BR = BaggingRegressor()

# Instantiate the grid search model
grid_search = GridSearchCV(estimator = BR, param_grid = param_grid, cv = 5, scoring='neg_mean_squared_error')
grid_search.fit(dataset, y)

# View the accuracy score
print('Best score for our dataset:', -grid_search.best_score_)

# View the best parameters for the model found using grid search
print('Best estimators:',grid_search.best_estimator_.n_estimators)
```

```
Best score for our dataset: 0.22201176803812508
Best estimators: 40
```

DecisionTreeRegressor

In [107]:
```
# Create the parameter grid
param_grid = {
    'max_depth': [50, 100, 150, 200],
}

# Model
DT = DecisionTreeRegressor()

# Instantiate the grid search model
grid_search = GridSearchCV(estimator = DT, param_grid = param_grid, cv = 5,  scoring='neg_mean_squared_error')
grid_search.fit(dataset, y)

# View the accuracy score
print('Best score for our dataset:', -grid_search.best_score_)

# View the best parameters for the model found using grid search
print('Best depth:',grid_search.best_estimator_.max_depth)
```

```
Best score for our dataset: 0.3426245881429061
Best depth: 50
```

AdaBoostRegressor

In [108]:
```
# Create the parameter grid
param_grid = {
    'n_estimators': [50 ,100, 150, 200]
}

# Model
AB = AdaBoostRegressor()

# Instantiate the grid search model
grid_search = GridSearchCV(estimator = AB, param_grid = param_grid, cv = 5, scoring= 'neg_mean_squared_error')
grid_search.fit(dataset, y)

# View the accuracy score
print('Best score for our dataset:', -grid_search.best_score_)

# View the best parameters for the model found using grid search
print('Best estimators:',grid_search.best_estimator_.n_estimators)
```

```
Best score for our dataset: 0.4815672330366181
Best estimators: 150
```

Model	MSE (default parameters)	MSE(after parameter tuning)	Improvements?
RandomForest	**0.234216**	**0.216417**	**0.017799**
DecisionTree	**0.347874**	**0.342624**	**0.00525**
AdaBoost	**0.500135**	**0.481567**	**0.018568**
Bagging	**0.233202**	**0.222011**	**0.011191**

All the algorithms saw an improvement in accuracy.

References

http://archive.ics.uci.edu/ml/datasets/Bike+Sharing+Dataset

https://www.kaggle.com/c/bike-sharing-demand

https://en.wikipedia.org/wiki/Washington,_D.C.#cite_note-70

https://www.timeanddate.com/weather/usa/washington-dc/climate

https://www.datascience.com/blog/introduction-to-correlation-learn-data-science-tutorials

http://scikit-learn.org/stable/model_selection.html

https://en.wikipedia.org/wiki/Root-mean-square_deviation

https://en.wikipedia.org/wiki/Mean_squared_error

http://blog.minitab.com/blog/adventures-in-statistics-2/regression-analysis-how-do-i-interpret-r-squared-and-assess-the-goodness-of-fit

scikit-learn documentations:

http://scikit-learn.org/stable/modules/ensemble.html#forest

http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingRegressor.html

http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html

http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html

http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html

http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostRegressor.html

http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

http://scikit-learn.org/stable/modules/model_evaluation.html