**Student (s) Number as per student card:**

_____ _____

_____ _____

_____ _____

**Course Title:** MSc in Data Analytics

**Lecturer Name:** Dr Shazia A Afzal

**Module/Subject Title:** Machine Learning and Pattern Recognition

**Assignment Title:** Supervised Machine Learning - Regression

**No of Words:** 2905

# Table of Contents


Intuition ..... 3
Dataset source ..... 3
SECTION 1: Choice of dependent and independent variables and selection of algorithm ..... 3
Choice of dependent and independent variables ..... 3
Variable ..... 3
Independent variable ..... 3
Dependent variable ..... 3
Selection of Algorithm ..... 4
Linear Regression ..... 4
SECTION 2: Data Preparation ..... 5
Data pre-processing ..... 7
SECTION 3: Feature Selection ..... 17
Variance inflation factor (VIF) ..... 17
Mathematical rule ..... 17
Type conversion ..... 17
P – value ..... 20
Mathematical rule ..... 20
SECTION 4: Model Development and Evaluation ..... 22
Model development ..... 22
Prediction ..... 22
Evaluation ..... 23
SECTION 5: Model Comparison ..... 24
Regularization ..... 24
Lasso ..... 24
Ridge Regression ..... 27
ElasticNet ..... 28
Optimization Algorithm ..... 28
Stochastic Gradient Descent (SGD) ..... 28
Conclusion ..... 29
References ..... 30

# Intuition

Understanding the dataset description, we are attempting to determine when bikes are more commonly used based on temperature, humidity, season, month, year, weekdays, holidays, windspeed, and weather conditions.

# Dataset source

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

# SECTION 1: Choice of dependent and independent variables and selection of algorithm

## Choice of dependent and independent variables

### Variable

A variable is a property of an object. A variable can be independent or dependent on another variable.

### Independent variable

A variable that is unaffected by other variables, it also has no relationship with other variables. For example, except for the cnt column, all the variables in the dataset we worked on are independent variables because they do not rely on other features to determine their value.

| | instant | dteday | season | yr | mnth | holiday | weekday | workingday | weathersit | temp | atemp | hum | windspeed | casual | registered | cnt |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 2011-01-01 | 1 | 0 | 1 | 0 | 6 | 0 | 2 | 0.344167 | 0.363625 | 0.805833 | 0.160446 | 331 | 654 | 985 |
| **1** | 2 | 2011-01-02 | 1 | 0 | 1 | 0 | 0 | 0 | 2 | 0.363478 | 0.353739 | 0.696087 | 0.248539 | 131 | 670 | 801 |
| **2** | 3 | 2011-01-03 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0.196364 | 0.189405 | 0.437273 | 0.248309 | 120 | 1229 | 1349 |
| **3** | 4 | 2011-01-04 | 1 | 0 | 1 | 0 | 2 | 1 | 1 | 0.200000 | 0.212122 | 0.590435 | 0.160296 | 108 | 1454 | 1562 |
| **4** | 5 | 2011-01-05 | 1 | 0 | 1 | 0 | 3 | 1 | 1 | 0.226957 | 0.229270 | 0.436957 | 0.186900 | 82 | 1518 | 1600 |

### Dependent variable

A variable that is influenced by other factors. It also has a relationship with other variables. Label column is another name for the dependent variable. The cnt column, for instance, is our dependent variable because it is affected by independent variables such as season, yr, mnth, holiday, weekday, workingday, weathersit, temp, atemp, hum, windspeed, casual, and registered.

## Selection of Algorithm

Here we are applying linear regression to our model. Before we apply linear regression to our model, there are some rules we must follow.

## Linear Regression

(Tibshirani et al., 2021). Simple linear regression is a very simple method for predicting a quantitative response Y from a single predictor variable X. It is presumptively assumed that X and Y have a linear relationship. Linear regression is a method for determining the best straight line fit to the provided data, i.e., the best linear correlation between the independent and dependent variables.

For univariate,

$$y = mx + c$$

For multivariate,

$$y = \beta_0 x_0 + \beta_1 x_1 + C$$

Where y = Dependent variable

m = Slope of the line

x = Independent variable

c = Intercept of the line

C = Intercept of the line

$\beta_0$ , $\beta_1$ = slope of $x_0$ and $x_1$

$x_0$ , $x_1$ = Independent variable which are contributing to predicting the y value

## Assumptions

- X (independent variables) should be correlated with Y (dependent variable).
- The residuals mean should be zero.
- Error terms are not allowed to be correlated to one another.
- X residuals must be uncorrelated.
- (Codecademy, 2008). The variance of the error term must be constant.
- No multicollinearity i.e., no relationships with the features themselves.
- Error terms are supposed to be normally distributed when plotted on graph.

If and only if these assumptions apply, we could use a linear regression algorithm.

## SECTION 2: Data Preparation

Data preparation comprises both information gathering and data cleansing. We collect data from a variety of sources in data gathering. Cleaning data involves removing null and missing values.

```
In [1]: import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        import seaborn as sns
        from sklearn.preprocessing import StandardScaler, MinMaxScaler
        from sklearn.linear_model import Ridge, RidgeCV, Lasso, LassoCV,ElasticNet , ElasticNetCV ,LinearRegression
        from sklearn.model_selection import train_test_split
        from pandas_profiling import ProfileReport
```

Importing all the required libraries. Numpy is used for numerical operations, while pandas are used for creating, reading, updating, and deleting objects. The data is visualized using Matplotlib, seaborn, and ProfileReport. Standardizing and normalizing are accomplished using StandardScalar and MinMaxScalar, respectively. We utilized Lasso, Ridge, and ElasticNet for regularization. LinearRegression is utilized because we choose this approach to apply to our model, and train_test_split is used to separate the data into train and test data.

```
In [2]: df = pd.read_csv("day.csv")
```

```
In [3]: df.head()
```

Out[3]:

| | instant | dteday | season | yr | mnth | holiday | weekday | workingday | weathersit | temp | atemp | hum | windspeed | casual | registered | cnt |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2011-01-01 | 1 | 0 | 1 | 0 | 6 | 0 | 2 | 0.344167 | 0.363625 | 0.805833 | 0.160446 | 331 | 654 | 985 |
| 1 | 2 | 2011-01-02 | 1 | 0 | 1 | 0 | 0 | 0 | 2 | 0.363478 | 0.353739 | 0.696087 | 0.248539 | 131 | 670 | 801 |
| 2 | 3 | 2011-01-03 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0.196364 | 0.189405 | 0.437273 | 0.248309 | 120 | 1229 | 1349 |
| 3 | 4 | 2011-01-04 | 1 | 0 | 1 | 0 | 2 | 1 | 1 | 0.200000 | 0.212122 | 0.590435 | 0.160296 | 108 | 1454 | 1562 |
| 4 | 5 | 2011-01-05 | 1 | 0 | 1 | 0 | 3 | 1 | 1 | 0.226957 | 0.229270 | 0.436957 | 0.186900 | 82 | 1518 | 1600 |

Using the pandas function read_csv, read data from day.csv file. With the exception of the dteday column, we can see that most of the column values are numerical.

```
In [4]: # checking rows and columns
        df.shape
```

```
Out[4]: (731, 16)
```

Checking the dataset's dimensions.

```
In [5]: # checking for missing values
        df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 731 entries, 0 to 730
Data columns (total 16 columns):
 #   Column      Non-Null Count  Dtype  
---  ------      --------------  -----  
 0   instant     731 non-null    int64  
 1   dteday      731 non-null    object 
 2   season      731 non-null    int64  
 3   yr          731 non-null    int64  
 4   mnth        731 non-null    int64  
 5   holiday     731 non-null    int64  
 6   weekday     731 non-null    int64  
 7   workingday  731 non-null    int64  
 8   weathersit  731 non-null    int64  
 9   temp        731 non-null    float64
 10  atemp       731 non-null    float64
 11  hum         731 non-null    float64
 12  windspeed   731 non-null    float64
 13  casual      731 non-null    int64  
 14  registered  731 non-null    int64  
 15  cnt         731 non-null    int64  
dtypes: float64(4), int64(11), object(1)
memory usage: 91.5+ KB
```

There are no missing values, as evidenced by the count of each column.

```
In [6]: # checking for any null values
        df.isna().sum()
```

```
Out[6]: instant       0
        dteday        0
        season        0
        yr            0
        mnth          0
        holiday       0
        weekday       0
        workingday    0
        weathersit    0
        temp          0
        atemp         0
        hum           0
        windspeed     0
        casual        0
        registered    0
        cnt           0
        dtype: int64
```

examining the total number of null values in each column.

## Data pre-processing

Before training the model, we must ensure that there are no missing, null, or category data for our linear regression model, as our algorithm requires numerical values to predict the outcome. This data pre-processing phase is required since it increases the learning and accuracy of our model.

The dteday column is transformed to datetime type in this case because we are attempting to break this one column into two independent columns such as year and month.

```
In [7]: # converting dteday column to datetime type
        df['dteday'] =  pd.to_datetime(df['dteday'],format='%Y-%m-%d')
        df['dteday']
```

```
Out[7]: 0     2011-01-01
        1     2011-01-02
        2     2011-01-03
        3     2011-01-04
        4     2011-01-05
              ...
        726   2012-12-27
        727   2012-12-28
        728   2012-12-29
        729   2012-12-30
        730   2012-12-31
        Name: dteday, Length: 731, dtype: datetime64[ns]
```

```
In [8]: # creating a separate columns for year and month from dteday column
        df['year'] = pd.DatetimeIndex(df['dteday']).year
        df['month'] = pd.DatetimeIndex(df['dteday']).month
```

```
In [9]: df.head()
```

Out[9]:

| instant | dteday | season | yr | mnth | holiday | weekday | workingday | weathersit | temp | atemp | hum | windspeed | casual | registered | cnt | year | month |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2011-01-01 | 1 | 0 | 1 | 0 | 6 | 0 | 2 | 0.344167 | 0.363625 | 0.805833 | 0.160446 | 331 | 654 | 985 | 2011 | 1 |
| 2 | 2011-01-02 | 1 | 0 | 1 | 0 | 0 | 0 | 2 | 0.363478 | 0.353739 | 0.696087 | 0.248539 | 131 | 670 | 801 | 2011 | 1 |
| 3 | 2011-01-03 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0.196364 | 0.189405 | 0.437273 | 0.248309 | 120 | 1229 | 1349 | 2011 | 1 |
| 4 | 2011-01-04 | 1 | 0 | 1 | 0 | 2 | 1 | 1 | 0.200000 | 0.212122 | 0.590435 | 0.160296 | 108 | 1454 | 1562 | 2011 | 1 |
| 5 | 2011-01-05 | 1 | 0 | 1 | 0 | 3 | 1 | 1 | 0.226957 | 0.229270 | 0.436957 | 0.186900 | 82 | 1518 | 1600 | 2011 | 1 |

We removed the yr and mnth columns since we extracted the year and month from the dateday column, which is more accurate.

```
In [10]: # Dropping yr and mnth column as we extracted year and month from dteday column which is more accurate
         df.drop(columns=['yr','mnth'],inplace = True)

In [11]: df.head()
```

Out[11]:

| | instant | dteday | season | holiday | weekday | workingday | weathersit | temp | atemp | hum | windspeed | casual | registered | cnt | year | month |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 2011-01-01 | 1 | 0 | 6 | 0 | 2 | 0.344167 | 0.363625 | 0.805833 | 0.160446 | 331 | 654 | 985 | 2011 | 1 |
| **1** | 2 | 2011-01-02 | 1 | 0 | 0 | 0 | 2 | 0.363478 | 0.353739 | 0.696087 | 0.248539 | 131 | 670 | 801 | 2011 | 1 |
| **2** | 3 | 2011-01-03 | 1 | 0 | 1 | 1 | 1 | 0.196364 | 0.189405 | 0.437273 | 0.248309 | 120 | 1229 | 1349 | 2011 | 1 |
| **3** | 4 | 2011-01-04 | 1 | 0 | 2 | 1 | 1 | 0.200000 | 0.212122 | 0.590435 | 0.160296 | 108 | 1454 | 1562 | 2011 | 1 |
| **4** | 5 | 2011-01-05 | 1 | 0 | 3 | 1 | 1 | 0.226957 | 0.229270 | 0.436957 | 0.186900 | 82 | 1518 | 1600 | 2011 | 1 |

The additional variable holiday is being dropped because the workingday section has all the necessary data. Dropping the dteday column because we already have a year and a month, as well as the fact that we cannot operate with non-numerical columns and the instant column is a non-essential column.

```
In [12]: #Dropping the excess variable holiday as the workingday section covers sufficient data that is required.
         df.drop(columns=['holiday'],inplace=True)

In [13]: # Dropping the dteday as we have year and month, also we cannot work on non numerical columns
         # and instant column is irrelevant column
         df.drop(columns=['instant','dteday'],inplace=True)

In [14]: df.head()
```

Out[14]:

| | season | weekday | workingday | weathersit | temp | atemp | hum | windspeed | casual | registered | cnt | year | month |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 6 | 0 | 2 | 0.344167 | 0.363625 | 0.805833 | 0.160446 | 331 | 654 | 985 | 2011 | 1 |
| **1** | 1 | 0 | 0 | 2 | 0.363478 | 0.353739 | 0.696087 | 0.248539 | 131 | 670 | 801 | 2011 | 1 |
| **2** | 1 | 1 | 1 | 1 | 0.196364 | 0.189405 | 0.437273 | 0.248309 | 120 | 1229 | 1349 | 2011 | 1 |
| **3** | 1 | 2 | 1 | 1 | 0.200000 | 0.212122 | 0.590435 | 0.160296 | 108 | 1454 | 1562 | 2011 | 1 |
| **4** | 1 | 3 | 1 | 1 | 0.226957 | 0.229270 | 0.436957 | 0.186900 | 82 | 1518 | 1600 | 2011 | 1 |

To eliminate confusion, the columns have been renamed.

```
In [15]: #renaming columns
         df.rename(columns={'hum':'humidity','cnt':'count'},inplace=True)

In [16]: df.head()
```

Out[16]:

| | season | weekday | workingday | weathersit | temp | atemp | humidity | windspeed | casual | registered | count | year | month |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 6 | 0 | 2 | 0.344167 | 0.363625 | 0.805833 | 0.160446 | 331 | 654 | 985 | 2011 | 1 |
| **1** | 1 | 0 | 0 | 2 | 0.363478 | 0.353739 | 0.696087 | 0.248539 | 131 | 670 | 801 | 2011 | 1 |
| **2** | 1 | 1 | 1 | 1 | 0.196364 | 0.189405 | 0.437273 | 0.248309 | 120 | 1229 | 1349 | 2011 | 1 |
| **3** | 1 | 2 | 1 | 1 | 0.200000 | 0.212122 | 0.590435 | 0.160296 | 108 | 1454 | 1562 | 2011 | 1 |
| **4** | 1 | 3 | 1 | 1 | 0.226957 | 0.229270 | 0.436957 | 0.186900 | 82 | 1518 | 1600 | 2011 | 1 |

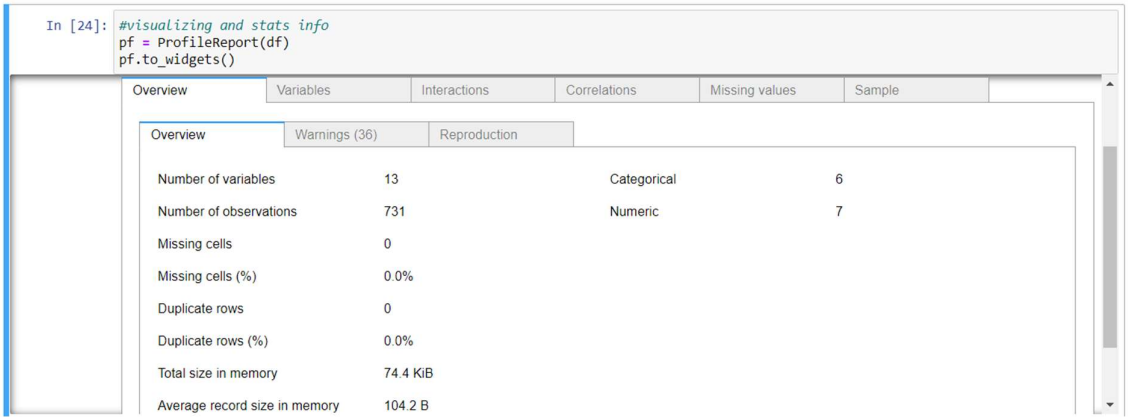
According to the dataset description, the ordinal and nominal columns are being converted into categorical columns and transforming year into numerical column.

```
In [17]: #converting to categorical columns
         labels = {1:'winter',2:'spring',3:'summer',4:'fall'}
         df['season'] = df['season'].map(labels)

In [18]: labels = {1:'Clear',2:'Mist',3:'Light Snow',4:'Heavy Rain'}
         df['weathersit'] = df['weathersit'].map(labels)

In [19]: labels = {1:'Working_day',0:'Holiday'}
         df['workingday'] = df['workingday'].map(labels)

In [20]: #converting to numerical column
         labels = {2012:1,2011:0}
         df['year'] = df['year'].map(labels)

In [21]: #converting month into categorical feature
         labels = {1:'Jan',2:'Feb',3:'Mar',4:'Apr',5:'May',6:'June',7:'July',8:'Aug',9:'Sep',10:'Oct',11:'Nov',12:'Dec'}
         df['month'] = df['month'].map(labels)

In [22]: #converting weekday into categorical feature
         labels = {0:'Mon',1:'Tue',2:'Wed',3:'Thu',4:'Fri',5:'Sat',6:'Sun'}
         df['weekday'] = df['weekday'].map(labels)

In [23]: #now the dataset should look more meaningful
         df.head()
```

Out[23]:

| | season | weekday | workingday | weathersit | temp | atemp | humidity | windspeed | casual | registered | count | year | month |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | winter | Sun | Holiday | Mist | 0.344167 | 0.363625 | 0.805833 | 0.160446 | 331 | 654 | 985 | 0 | Jan |
| 1 | winter | Mon | Holiday | Mist | 0.363478 | 0.353739 | 0.696087 | 0.248539 | 131 | 670 | 801 | 0 | Jan |
| 2 | winter | Tue | Working_day | Clear | 0.196364 | 0.189405 | 0.437273 | 0.248309 | 120 | 1229 | 1349 | 0 | Jan |
| 3 | winter | Wed | Working_day | Clear | 0.200000 | 0.212122 | 0.590435 | 0.160296 | 108 | 1454 | 1562 | 0 | Jan |
| 4 | winter | Thu | Working_day | Clear | 0.226957 | 0.229270 | 0.436957 | 0.186900 | 82 | 1518 | 1600 | 0 | Jan |

For data visualization, we attempted to limit the number of lines. When we use ProfileReport to visualize our dataset, we can see the columns overview, variables, interactions, correlations, missing values, and sample. In Overview, we have the number of columns, the number of rows, the number of missing values, the number of duplicate rows, the number of categorical columns, and the number of numerical columns. Each individual column statistical data is available under the variables tab.




The graphs between two columns and their relationships can be seen in the interaction tab.

Overview | Variables | Interactions | Correlations | Missing values | Sample

- season
- weekday
- workingday
- weathersit
- temp
- atemp
- humidity
- windspeed
- casual
- registered

**count**

count

Real number ($\mathbb{R}_{\geq 0}$)

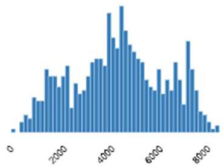
HIGH CORRELATION

HIGH CORRELATION

HIGH CORRELATION

HIGH CORRELATION

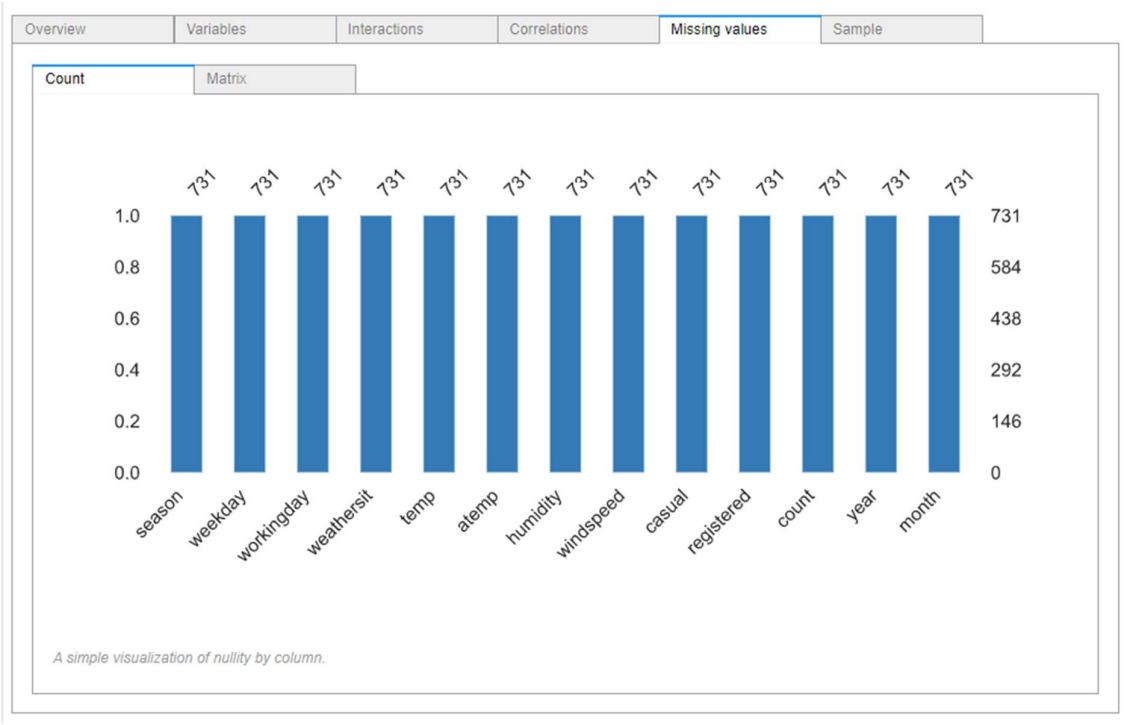
| | | | |
|---|---|---|---|
| Distinct | 696 | Minimum | 22 |
| Distinct (%) | 95.2% | Maximum | 8714 |
| Missing | 0 | Zeros | 0 |
| Missing (%) | 0.0% | Zeros (%) | 0.0% |
| Infinite | 0 | Negative | 0 |
| Infinite (%) | 0.0% | Negative (%) | 0.0% |
| Mean | 4504.348837 | Memory size | 5.8 KiB |



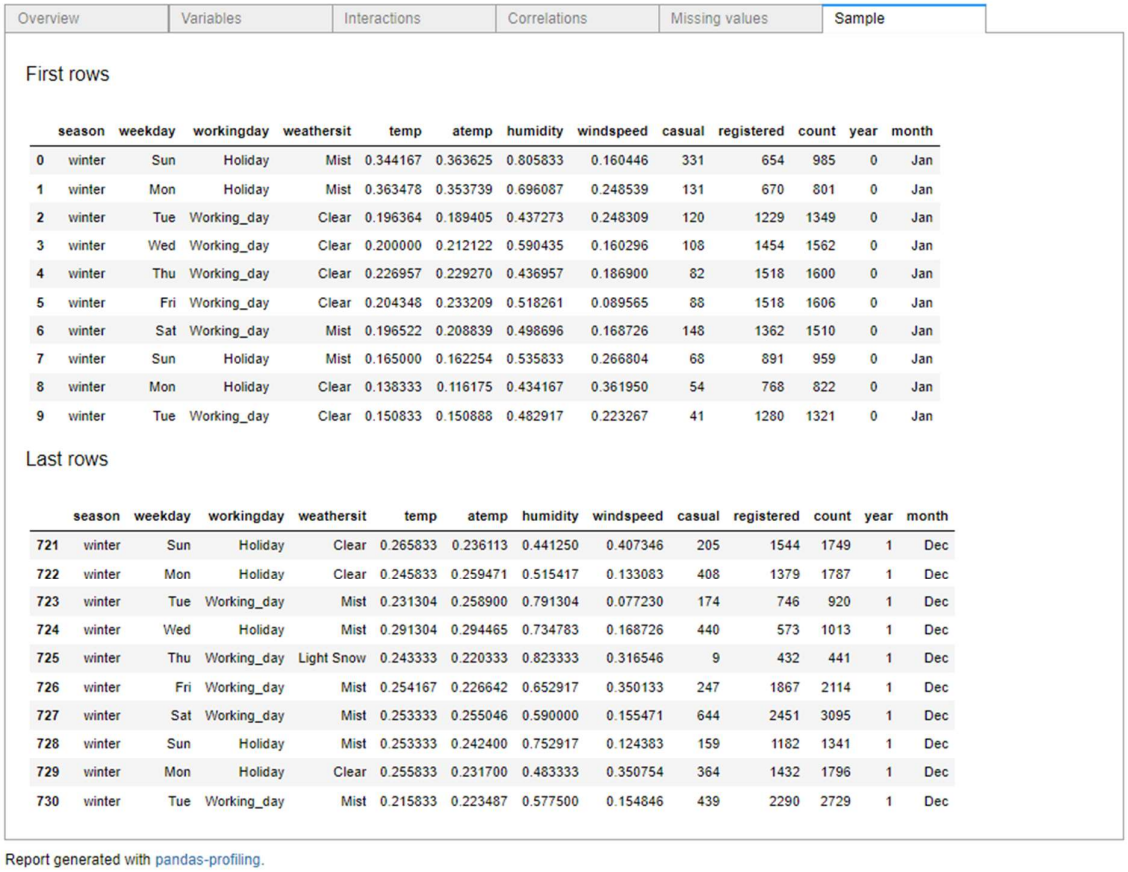

Toggle details

In correlations tab, we can see whether there is a positive or negative correlation between each variable or column; there are several correlations like as Persons, Spearman's, and so on. We attempted to demonstrate some.

To find out if there is a missing value, go to the missing tab. Fortunately, there are no missing values.



The top ten columns and bottom ten columns of the dataset were visible under the sample tab.



Overview | Variables | Interactions | Correlations | Missing values | Sample

First rows

| | season | weekday | workingday | weathersit | temp | atemp | humidity | windspeed | casual | registered | count | year | month |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | winter | Sun | Holiday | Mist | 0.344167 | 0.363625 | 0.805833 | 0.160446 | 331 | 654 | 985 | 0 | Jan |
| 1 | winter | Mon | Holiday | Mist | 0.363478 | 0.353739 | 0.696087 | 0.248539 | 131 | 670 | 801 | 0 | Jan |
| 2 | winter | Tue | Working_day | Clear | 0.196364 | 0.189405 | 0.437273 | 0.248309 | 120 | 1229 | 1349 | 0 | Jan |
| 3 | winter | Wed | Working_day | Clear | 0.200000 | 0.212122 | 0.590435 | 0.160296 | 108 | 1454 | 1562 | 0 | Jan |
| 4 | winter | Thu | Working_day | Clear | 0.226957 | 0.229270 | 0.436957 | 0.186900 | 82 | 1518 | 1600 | 0 | Jan |
| 5 | winter | Fri | Working_day | Clear | 0.204348 | 0.233209 | 0.518261 | 0.089565 | 88 | 1518 | 1606 | 0 | Jan |
| 6 | winter | Sat | Working_day | Mist | 0.196522 | 0.208839 | 0.498696 | 0.168726 | 148 | 1362 | 1510 | 0 | Jan |
| 7 | winter | Sun | Holiday | Mist | 0.165000 | 0.162254 | 0.535833 | 0.266804 | 68 | 891 | 959 | 0 | Jan |
| 8 | winter | Mon | Holiday | Clear | 0.138333 | 0.116175 | 0.434167 | 0.361950 | 54 | 768 | 822 | 0 | Jan |
| 9 | winter | Tue | Working_day | Clear | 0.150833 | 0.150888 | 0.482917 | 0.223267 | 41 | 1280 | 1321 | 0 | Jan |

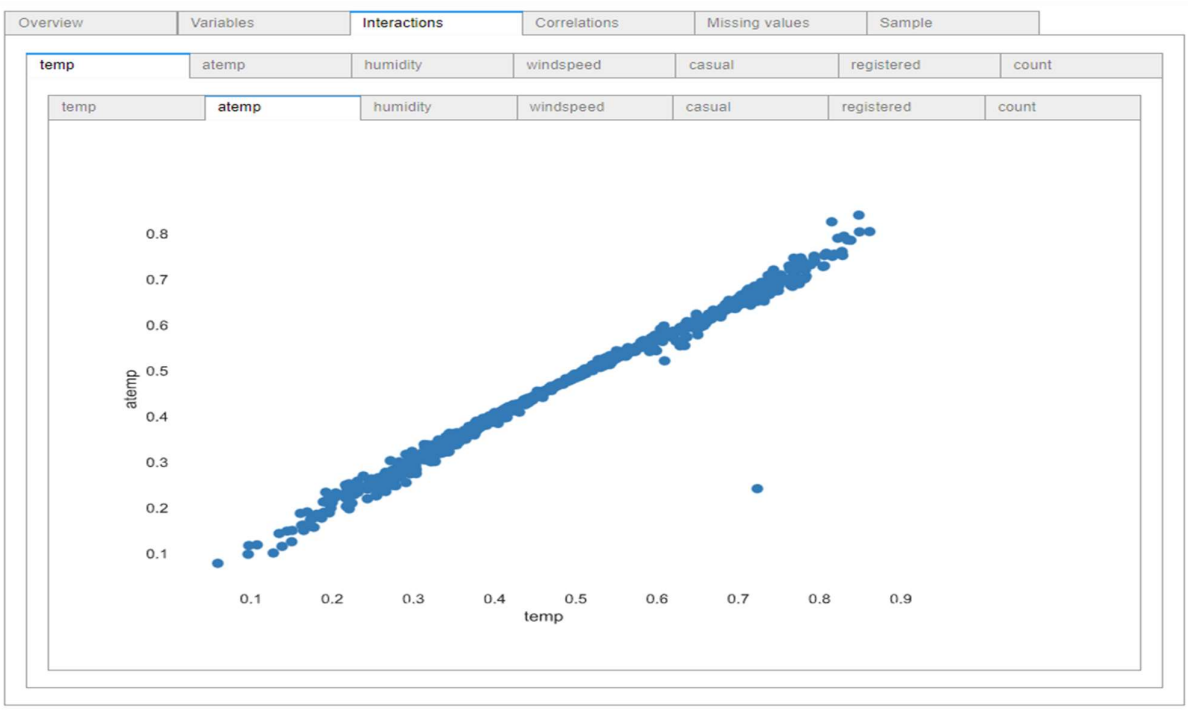
Last rows

| | season | weekday | workingday | weathersit | temp | atemp | humidity | windspeed | casual | registered | count | year | month |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 721 | winter | Sun | Holiday | Clear | 0.265833 | 0.236113 | 0.441250 | 0.407346 | 205 | 1544 | 1749 | 1 | Dec |
| 722 | winter | Mon | Holiday | Clear | 0.245833 | 0.259471 | 0.515417 | 0.133083 | 408 | 1379 | 1787 | 1 | Dec |
| 723 | winter | Tue | Working_day | Mist | 0.231304 | 0.258900 | 0.791304 | 0.077230 | 174 | 746 | 920 | 1 | Dec |
| 724 | winter | Wed | Holiday | Mist | 0.291304 | 0.294465 | 0.734783 | 0.168726 | 440 | 573 | 1013 | 1 | Dec |
| 725 | winter | Thu | Working_day | Light Snow | 0.243333 | 0.220333 | 0.823333 | 0.316546 | 9 | 432 | 441 | 1 | Dec |
| 726 | winter | Fri | Working_day | Mist | 0.254167 | 0.226642 | 0.652917 | 0.350133 | 247 | 1867 | 2114 | 1 | Dec |
| 727 | winter | Sat | Working_day | Mist | 0.253333 | 0.255046 | 0.590000 | 0.155471 | 644 | 2451 | 3095 | 1 | Dec |
| 728 | winter | Sun | Holiday | Mist | 0.253333 | 0.242400 | 0.752917 | 0.124383 | 159 | 1182 | 1341 | 1 | Dec |
| 729 | winter | Mon | Holiday | Clear | 0.255833 | 0.231700 | 0.483333 | 0.350754 | 364 | 1432 | 1796 | 1 | Dec |
| 730 | winter | Tue | Working_day | Mist | 0.215833 | 0.223487 | 0.577500 | 0.154846 | 439 | 2290 | 2729 | 1 | Dec |

Report generated with pandas-profiling.

As mentioned in assumptions of linear regression, trying to avoid multicollinearity for the independent variables. Checking the plots of temp and atemp using ProfileReport from pandas profiling, you can see that there is a substantial correlation between them, as well as casual and registered. As a result, the columns listed below have been removed.

```
In [25]: # To avoid multicollinearity
         # Dropping atemp, casual, registered
         df.drop(columns=['atemp', 'casual', 'registered'],inplace = True)
```




Report generated with pandas-profiling.


Report generated with pandas-profiling.

```
In [26]: df.head()
```

Out[26]:

| | season | weekday | workingday | weathersit | temp | humidity | windspeed | count | year | month |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | winter | Sun | Holiday | Mist | 0.344167 | 0.805833 | 0.160446 | 985 | 0 | Jan |
| 1 | winter | Mon | Holiday | Mist | 0.363478 | 0.696087 | 0.248539 | 801 | 0 | Jan |
| 2 | winter | Tue | Working_day | Clear | 0.196364 | 0.437273 | 0.248309 | 1349 | 0 | Jan |
| 3 | winter | Wed | Working_day | Clear | 0.200000 | 0.590435 | 0.160296 | 1562 | 0 | Jan |
| 4 | winter | Thu | Working_day | Clear | 0.226957 | 0.436957 | 0.186900 | 1600 | 0 | Jan |

There are still categorical columns in the preceding example, and we need to do something about them because our model can forecast data based on numerical values. Using the pandas function get dummies, convert these categorical columns to numbers. This function adds 1 everywhere the value is. For example, in the month column, wherever Jan is indicated as 1 and where it is not designated as 0. Apply to the remaining categorical columns as well, such as month, season, weekday, workingday, and weathersit.

```
In [27]: #creating dummy indicators for season, weekday, workingday, weathersit, month
         seasons = pd.get_dummies(df['season'],drop_first=True)
         week_day = pd.get_dummies(df['weekday'],drop_first=True)
         working_day = pd.get_dummies(df['workingday'],drop_first=True)
         weather = pd.get_dummies(df['weathersit'],drop_first=True)
         month = pd.get_dummies(df['month'],drop_first=True)
```

Concatenating the columns generated above to the DataFrame. As per the DataFrame, we have a total of 33 columns.

```
In [28]: df = pd.concat([df,seasons,working_day,weather,month,week_day],axis=1)
         df.head()
```

Out[28]:

| | season | weekday | workingday | weathersit | temp | humidity | windspeed | count | year | month | ... | May | Nov | Oct | Sep | Mon | Sat | Sun | Thu | Tue | Wed |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | winter | Sun | Holiday | Mist | 0.344167 | 0.805833 | 0.160446 | 985 | 0 | Jan | ... | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | winter | Mon | Holiday | Mist | 0.363478 | 0.696087 | 0.248539 | 801 | 0 | Jan | ... | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 2 | winter | Tue | Working_day | Clear | 0.196364 | 0.437273 | 0.248309 | 1349 | 0 | Jan | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 3 | winter | Wed | Working_day | Clear | 0.200000 | 0.590435 | 0.160296 | 1562 | 0 | Jan | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 4 | winter | Thu | Working_day | Clear | 0.226957 | 0.436957 | 0.186900 | 1600 | 0 | Jan | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

5 rows × 33 columns

Remove all categorical because it has already been transformed and we no longer need it, leaving us with 28 columns.

```
In [29]: #Dealing with categorical columns(season, weekday, workingday, weathersit, month) as we have dummy
         #indicators for each of these variables
         df.drop(columns=['season', 'weekday', 'workingday', 'weathersit', 'month'],inplace=True)
```

```
In [30]: df.head()
```

Out[30]:

| | temp | humidity | windspeed | count | year | spring | summer | winter | Working_day | Light Snow | ... | May | Nov | Oct | Sep | Mon | Sat | Sun | Thu | Tue | Wed |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.344167 | 0.805833 | 0.160446 | 985 | 0 | 0 | 0 | 1 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0.363478 | 0.696087 | 0.248539 | 801 | 0 | 0 | 0 | 1 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0.196364 | 0.437273 | 0.248309 | 1349 | 0 | 0 | 0 | 1 | 1 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0.200000 | 0.590435 | 0.160296 | 1562 | 0 | 0 | 0 | 1 | 1 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 4 | 0.226957 | 0.436957 | 0.186900 | 1600 | 0 | 0 | 0 | 1 | 1 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

5 rows × 28 columns

Temp, humidity, and windspeed have values between 0 and 1, whereas the rest of the columns have values either 0 or 1, except for the count column, which has values that are far apart from 0 and 1.

Attempting to bring all columns other than dummy columns under one scale, using normalization to do so. The count field spans between 0 and 1 after normalization. Temp, humidity, and windspeed have previously been normalized in accordance with the dataset description.

In [31]:
```
# normalization applied other than the dummy variables
x = ['temp','humidity','windspeed','count']
x
```

Out[31]:
```
['temp', 'humidity', 'windspeed', 'count']
```

In [32]:
```
# normalization as the columns temp, humidity and windspeed are already normalized between 0 and 1
# And count is not normalized, hence we normalize it along with the other columns
normalized = MinMaxScaler()
df[x]= normalized.fit_transform(df[x])
```

In [33]:
```
# Now the data is on the same Scale
df.head()
```

Out[33]:

| | temp | humidity | windspeed | count | year | spring | summer | winter | Working_day | Light Snow | ... | May | Nov | Oct | Sep | Mon | Sat | Sun | Thu | Tue | Wed |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.355170 | 0.828620 | 0.284606 | 0.110792 | 0 | 0 | 0 | 1 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0.379232 | 0.715771 | 0.466215 | 0.089623 | 0 | 0 | 0 | 1 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0.171000 | 0.449638 | 0.465740 | 0.152669 | 0 | 0 | 0 | 1 | 1 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0.175530 | 0.607131 | 0.284297 | 0.177174 | 0 | 0 | 0 | 1 | 1 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 4 | 0.209120 | 0.449313 | 0.339143 | 0.181546 | 0 | 0 | 0 | 1 | 1 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

5 rows × 28 columns

After identifying the independent and dependent variables, extract them from the Dataframe to prepare them for feature selection from the independent variables.

In [34]:
```
#Dropping target variable
x = df.drop(columns=['count'])
x
```

Out[34]:

| | temp | humidity | windspeed | year | spring | summer | winter | Working_day | Light Snow | Mist | ... | May | Nov | Oct | Sep | Mon | Sat | Sun | Thu | Tue | Wed |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.355170 | 0.828620 | 0.284606 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0.379232 | 0.715771 | 0.466215 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | ... | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0.171000 | 0.449638 | 0.465740 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0.175530 | 0.607131 | 0.284297 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 4 | 0.209120 | 0.449313 | 0.339143 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 726 | 0.243025 | 0.671380 | 0.675656 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 727 | 0.241986 | 0.606684 | 0.274350 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | ... | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 728 | 0.241986 | 0.774208 | 0.210260 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 729 | 0.245101 | 0.497001 | 0.676936 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 730 | 0.195259 | 0.593830 | 0.273062 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

731 rows × 27 columns

In [35]:
```
y = df['count']
y
```

Out[35]:
```
0      0.110792
1      0.089623
2      0.152669
3      0.177174
4      0.181546
         ...
726    0.240681
727    0.353543
728    0.151749
729    0.204096
730    0.311436
Name: count, Length: 731, dtype: float64
```

# SECTION 3: Feature Selection

There are two approaches for selecting a feature. The first is the Variance inflation factor, while the second is by p-values.

## Variance inflation factor (VIF)

(Tibshirani et al., 2021). The variance inflation factor quantifies the degree to which the behavior (variance) of an independent variable is affected or inflated by its interaction/correlation with other independent variables. The variance inflation factor provides a quick estimate of how much a variable affects the standard error of the regression.

### Mathematical rule

There is only one criterion for VIF: if the VIF for any column exceeds 10 (VIF > 10), we simply delete the column because it has a higher correlation/contribution to error terms.

### Type conversion

As Y variable has an array type, we change X to an array type as well, because the VIF function accepts array type values.

```
In [36]: arr = x.values
         arr

Out[36]: array([[0.3551696 , 0.82862005, 0.2846062 , ..., 0.        , 0.        ,
                 0.        ],
                [0.37923205, 0.71577069, 0.46621455, ..., 0.        , 0.        ,
                 0.        ],
                [0.1709998 , 0.44963805, 0.4657404 , ..., 0.        , 1.        ,
                 0.        ],
                ...,
                [0.24198597, 0.77420771, 0.21026043, ..., 0.        , 0.        ,
                 0.        ],
                [0.2451011 , 0.49700051, 0.67693615, ..., 0.        , 0.        ,
                 0.        ],
                [0.19525913, 0.59383033, 0.27306151, ..., 0.        , 1.        ,
                 0.        ]])
```

The variance inflation component is being imported from the relevant library. Here, "arr" has all features and "i" contains columns, and these parameters are passed to the variance inflation factor function. The values of vif for each column are listed in the vif column, while the names of the features are listed in the "feature" column.

```
In [37]: from statsmodels.stats.outliers_influence import variance_inflation_factor
         vif_df = pd.DataFrame()

In [38]: vif_df['vif'] = [variance_inflation_factor(arr,i) for i in range(arr.shape[1])]

In [39]: vif_df['feature'] = x.columns
```

```
In [40]: vif_df
```

Out[40]:

| | vif | feature |
|---|---|---|
| 0 | 39.626155 | temp |
| 1 | 39.601188 | humidity |
| 2 | 6.433388 | windspeed |
| 3 | 2.086738 | year |
| 4 | 9.358149 | spring |
| 5 | 10.486915 | summer |
| 6 | 8.625784 | winter |
| 7 | 20.475972 | Working_day |
| 8 | 1.342106 | Light Snow |
| 9 | 2.429744 | Mist |
| 10 | 5.853294 | Aug |
| 11 | 3.832862 | Dec |
| 12 | 4.344264 | Feb |
| 13 | 4.845877 | Jan |
| 14 | 6.199975 | July |
| 15 | 3.027652 | June |
| 16 | 3.009205 | Mar |
| 17 | 2.337887 | May |
| 18 | 4.053268 | Nov |
| 19 | 4.652532 | Oct |
| 20 | 4.590062 | Sep |
| 21 | 5.657446 | Mon |
| 22 | 1.965890 | Sat |
| 23 | 5.634188 | Sun |
| 24 | 1.998173 | Thu |
| 25 | 2.014596 | Tue |
| 26 | 1.986409 | Wed |

Some of the characteristics in the preceding DataFrame have values larger than 10, i.e., VIF > 10. By removing columns like temp, humidity, Working_day, and summer, the remaining features are involved in forecasting the target variable. This is how we decide which features to include. Don't be confused by the dropping of the count column; our initial DataFrame after Data cleaning still retains all its properties, including our target variable, because we didn't use the inplace option to permanently eliminate it.

```
In [41]: x = df.drop(columns=['temp','humidity','Working_day','summer','count'])
         x
```

Out[41]:

| | windspeed | year | spring | winter | Light Snow | Mist | Aug | Dec | Feb | Jan | ... | May | Nov | Oct | Sep | Mon | Sat | Sun | Thu | Tue | Wed |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.284606 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0.466215 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | ... | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0.465740 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0.284297 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 4 | 0.339143 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 726 | 0.675656 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 727 | 0.274350 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 728 | 0.210260 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 729 | 0.676936 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 730 | 0.273062 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

731 rows × 23 columns

After omitting some features, double-check the VIF values to ensure that no features are contributing to error terms, as this is one of our linear regression assumptions.

In [42]:
```
arr = x.values
arr
```

Out[42]:
```
array([[0.2846062 , 0.        , 0.        , ..., 0.        , 0.        ,
        0.        ],
       [0.46621455, 0.        , 0.        , ..., 0.        , 0.        ,
        0.        ],
       [0.4657404 , 0.        , 0.        , ..., 0.        , 1.        ,
        0.        ],
       ...,
       [0.21026043, 1.        , 0.        , ..., 0.        , 0.        ,
        0.        ],
       [0.67693615, 1.        , 0.        , ..., 0.        , 0.        ,
        0.        ],
       [0.27306151, 1.        , 0.        , ..., 0.        , 1.        ,
        0.        ]])
```

In [43]:
```
from statsmodels.stats.outliers_influence import variance_inflation_factor
vif_df = pd.DataFrame()
vif_df['vif'] = [variance_inflation_factor(arr,i) for i in range(arr.shape[1])]
vif_df['feature'] = x.columns
vif_df
```

Out[43]:

| | vif | feature |
|---|---|---|
| 0 | 5.607441 | windspeed |
| 1 | 1.980372 | year |
| 2 | 4.127332 | spring |
| 3 | 7.643340 | winter |
| 4 | 1.111839 | Light Snow |
| 5 | 1.594072 | Mist |
| 6 | 1.778474 | Aug |
| 7 | 1.870574 | Dec |
| 8 | 3.438366 | Feb |
| 9 | 3.653071 | Jan |
| 10 | 1.744273 | July |
| 11 | 1.479249 | June |
| 12 | 2.439700 | Mar |
| 13 | 1.806098 | May |
| 14 | 1.777663 | Nov |
| 15 | 1.843034 | Oct |
| 16 | 1.754608 | Sep |
| 17 | 1.905952 | Mon |
| 18 | 1.901571 | Sat |
| 19 | 1.911272 | Sun |
| 20 | 1.907460 | Thu |
| 21 | 1.913240 | Tue |
| 22 | 1.892578 | Wed |

We can proceed with model training because there are no VIFs greater than 10.

## P – value

A p-value is a statistic that expresses the likelihood that an observed difference may have occurred by chance. The statistical significance of the observed difference is increased by lowering the p-value. The Ordinal Least Square method (OLS) can be used to determine the P value. This function has two essential parameters: formula and DataFrame.

```
In [55]: import statsmodels.formula.api as smf
         lm2 = smf.ols(formula='count~temp+humidity+windspeed+year+spring+summer+winter+Working_day+Light_snow+Mist+Aug+Dec+Feb+Jan+July+
         lm2.summary()
```

| | coef | std err | t | P>\|t\| | [0.025 | 0.975] |
|---|---|---|---|---|---|---|
| Intercept | 0.4071 | 0.045 | 9.100 | 0.000 | 0.319 | 0.495 |
| temp | 0.4143 | 0.038 | 10.896 | 0.000 | 0.340 | 0.489 |
| humidity | -0.1699 | 0.033 | -5.196 | 0.000 | -0.234 | -0.106 |
| windspeed | -0.1633 | 0.023 | -7.202 | 0.000 | -0.208 | -0.119 |
| year | 0.2322 | 0.007 | 34.660 | 0.000 | 0.219 | 0.245 |
| spring | -0.0793 | 0.024 | -3.248 | 0.001 | -0.127 | -0.031 |
| summer | -0.0859 | 0.022 | -3.901 | 0.000 | -0.129 | -0.043 |
| winter | -0.1817 | 0.021 | -8.722 | 0.000 | -0.223 | -0.141 |
| Working_day | 0.0694 | 0.021 | 3.352 | 0.001 | 0.029 | 0.110 |
| Light_snow | -0.2280 | 0.023 | -10.075 | 0.000 | -0.272 | -0.184 |
| Mist | -0.0535 | 0.009 | -6.035 | 0.000 | -0.071 | -0.036 |
| Aug | -0.0059 | 0.028 | -0.214 | 0.831 | -0.060 | 0.049 |
| Dec | -0.0622 | 0.028 | -2.233 | 0.026 | -0.117 | -0.008 |
| Feb | -0.0368 | 0.028 | -1.332 | 0.183 | -0.091 | 0.017 |
| Jan | -0.0525 | 0.028 | -1.844 | 0.066 | -0.108 | 0.003 |
| July | -0.0516 | 0.028 | -1.810 | 0.071 | -0.107 | 0.004 |
| June | 0.0039 | 0.020 | 0.196 | 0.844 | -0.035 | 0.043 |
| Mar | 0.0102 | 0.021 | 0.480 | 0.631 | -0.031 | 0.052 |
| May | 0.0307 | 0.017 | 1.787 | 0.074 | -0.003 | 0.064 |
| Nov | -0.0653 | 0.030 | -2.166 | 0.031 | -0.125 | -0.006 |
| Oct | 0.0074 | 0.029 | 0.252 | 0.801 | -0.050 | 0.065 |
| Sep | 0.0607 | 0.026 | 2.327 | 0.020 | 0.009 | 0.112 |
| Mon | 0.0251 | 0.024 | 1.055 | 0.292 | -0.022 | 0.072 |
| Sat | 0.0050 | 0.012 | 0.405 | 0.685 | -0.019 | 0.029 |
| Sun | 0.0756 | 0.024 | 3.178 | 0.002 | 0.029 | 0.122 |
| Thu | -0.0009 | 0.012 | -0.073 | 0.942 | -0.025 | 0.023 |
| Tue | -0.0196 | 0.013 | -1.558 | 0.120 | -0.044 | 0.005 |
| Wed | -0.0088 | 0.012 | -0.711 | 0.477 | -0.033 | 0.015 |

## Mathematical rule

If $p < 0.05$, the column is significant, which means that more than 95% of the rows in the column are contributing. Otherwise, if $p > 0.05$, the column has no significance, and we remove all ones that meet this criterion.

Remove columns with p > 0.05 since they don't contribute much and have little relevance, such as Aug, Feb, Jan, July, June, Mar, May, Oct, Mon, Sat, Thu, Tue, and Wed, and deep copying the initial DataFrame so that it may be used for model comparison and evaluation ensuring the original DataFrame remains unchanged.

In [56]:
```
# we need to remove p > 0.05 as those doesnt contribute more and also has no significance
# Aug, Feb, Jan, July, June, Mar, May, Oct, Mon, Sat, Thu, Tue, Wed

df1 = df.drop(columns= ['Aug','Feb','Jan','July','June','Mar','May','Oct','Mon','Sat','Thu','Tue','Wed']).copy()
df1
```

Out[56]:

| | temp | humidity | windspeed | count | year | spring | summer | winter | Working_day | Light_snow | Mist | Dec | Nov | Sep | Sun |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.355170 | 0.828620 | 0.284606 | 0.110792 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 1 | 0.379232 | 0.715771 | 0.466215 | 0.089623 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 2 | 0.171000 | 0.449638 | 0.465740 | 0.152669 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0.175530 | 0.607131 | 0.284297 | 0.177174 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0.209120 | 0.449313 | 0.339143 | 0.181546 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 726 | 0.243025 | 0.671380 | 0.675656 | 0.240681 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 727 | 0.241986 | 0.606684 | 0.274350 | 0.353543 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 728 | 0.241986 | 0.774208 | 0.210260 | 0.151749 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 729 | 0.245101 | 0.497001 | 0.676936 | 0.204096 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 730 | 0.195259 | 0.593830 | 0.273062 | 0.311436 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |

731 rows × 15 columns

In [57]:
```
df1.columns
```

Out[57]:
```
Index(['temp', 'humidity', 'windspeed', 'count', 'year', 'spring', 'summer',
       'winter', 'Working_day', 'Light_snow', 'Mist', 'Dec', 'Nov', 'Sep',
       'Sun'],
      dtype='object')
```

These are the features we chose to evaluate our model. Pass the target variable and the rest of the features to the formula parameter.

# SECTION 4: Model Development and Evaluation

## Model development

In model development, the single most critical thing you can do to properly evaluate your model is to avoid training it on the whole dataset. Do not train the model on the complete dataset.

Divide the dataset into training and testing regions. In this case, train test split takes four primary factors into account: features, labels, test size = 0.20, which means that 20% of the data is testing data and the remaining 80% is training data, and random state is utilized to provide same results every time.

```
In [44]: x_train, x_test, y_train, y_test = train_test_split(x,y,test_size=0.20,random_state=100)
```

Calling the LinearRegression constructor to create an object and attempting to train the model with the fit () function.

```
In [45]: lr = LinearRegression()

In [46]: lr.fit(x_train,y_train)
Out[46]: LinearRegression()
```

## Prediction

Testing to see if our model is accurate enough. The predicted value is 0.101, but the actual value is 0.110 and there isn't much of a difference.

```
In [47]: arr[0]
Out[47]: array([0.2846062, 0.       , 0.       , 1.       , 0.       , 1.       ,
                0.       , 0.       , 0.       , 1.       , 0.       , 0.       ,
                0.       , 0.       , 0.       , 0.       , 0.       , 0.       ,
                0.       , 1.       , 0.       , 0.       , 0.       ])

In [48]: #trying to predict the value of y
         lr.predict([arr[0]])
Out[48]: array([0.10122911])

In [49]: # Actual y value
         # y_pred = 0.101 and y_actual = 0.110
         y[0]
Out[49]: 0.11079153244362633
```

Using test data to validate our model and determine its performance score.

## Evaluation

Using R square/Adjusted R square metrics, we found that our model was 75.28 % accurate by using linear regression.

```
In [50]: # R square
         score = lr.score(x_test,y_test)

In [51]: print("Score : {:.2f} %".format(score*100))
         Score : 75.28 %
```

We obtained an adjusted R square score of 83.68 % accuracy by applying the ordinal least squares (OLS) approach.

```
In [58]: import statsmodels.formula.api as smf
         lm2 = smf.ols(formula='count~temp+humidity+windspeed+year+spring+summer+winter+Working_day+Light_snow+Mist+Dec+Nov+Sep+Sun',data=
         lm2.summary()
```

Out[58]:

OLS Regression Results

| | | | |
|---|---|---|---|
| **Dep. Variable:** | count | **R-squared:** | 0.840 |
| **Model:** | OLS | **Adj. R-squared:** | 0.837 |
| **Method:** | Least Squares | **F-statistic:** | 268.3 |
| **Date:** | Wed, 03 Nov 2021 | **Prob (F-statistic):** | 1.63e-273 |
| **Time:** | 19:58:56 | **Log-Likelihood:** | 730.17 |
| **No. Observations:** | 731 | **AIC:** | -1430. |
| **Df Residuals:** | 716 | **BIC:** | -1361. |
| **Df Model:** | 14 | | |
| **Covariance Type:** | nonrobust | | |

```
In [59]: score = lm2.rsquared
         print("R Square Score : {:.2f} %".format(score*100))
         score = lm2.rsquared_adj
         print("Adjusted R Square Score : {:.2f} %".format(score*100))

         R Square Score : 83.99 %
         Adjusted R Square Score : 83.68 %
```

# SECTION 5: Model Comparison

To increase the performance of our model, one can use either regularization or an optimization approach in Model Comparison.

## Regularization

This is a type of regression in which the coefficient estimates are constrained/regularized or shrunk towards zero. All in all, to keep away from overfitting, this technique debilitates learning a more convoluted or adaptable model. (Aurélien Géron, 2019b). Regularization of a linear model is often accomplished by restricting the model's weights. We'll now look at Ridge Regression, Lasso Regression, and Elastic Net, which use three distinct methods to limit the weights. This strategy is used to ensure that our model is consistently accurate.

## Lasso

Regularization techniques such as Lasso regression are used. For more accurate prediction, it is preferred over regression approaches. Shrinkage is utilized in this model. Shrinkage is the process of reducing data values to a single central point known as the mean. Simple, sparse models are supported by the lasso approach (i.e., models with fewer parameters). This type of regression is ideal for models with high degrees of multicollinearity or for automating certain aspects of model selection, such as variable selection/parameter removal.

```
In [61]: df1.head()
```

Out[61]:

| | temp | humidity | windspeed | count | year | spring | summer | winter | Working_day | Light_snow | Mist | Dec | Nov | Sep | Sun |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.355170 | 0.828620 | 0.284606 | 0.110792 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 1 | 0.379232 | 0.715771 | 0.466215 | 0.089623 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 2 | 0.171000 | 0.449638 | 0.465740 | 0.152669 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0.175530 | 0.607131 | 0.284297 | 0.177174 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0.209120 | 0.449313 | 0.339143 | 0.181546 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

Taking a look at the dataset's top 5 rows. The graphs below show that temperature and count are highly correlated. As previously mentioned in the context of Pandas profiling, use the tabs to navigate around the graphs and statistical data.

```
In [62]: pf1 = ProfileReport(df1)
         pf1.to_widgets()
```

Summarize dataset: 100% 28/28 [00:08<00:00, 2.01it/s, Completed]

Generate report structure: 100% 1/1 [00:13<00:00, 13.74s/it]

Overview | Variables | Interactions | Correlations | Missing values | Sample

Overview | Warnings (35) | Reproduction

| | | | |
|---|---|---|---|
| Number of variables | 15 | Numeric | 4 |
| Number of observations | 731 | Categorical | 11 |
| Missing cells | 0 | | |
| Missing cells (%) | 0.0% | | |
| Duplicate rows | 0 | | |
| Duplicate rows (%) | 0.0% | | |
| Total size in memory | 35.8 KiB | | |
| Average record size in memory | 50.2 B | | |

Report generated with pandas-profiling.

```
In [62]: pf1 = ProfileReport(df1)
         pf1.to_widgets()
```

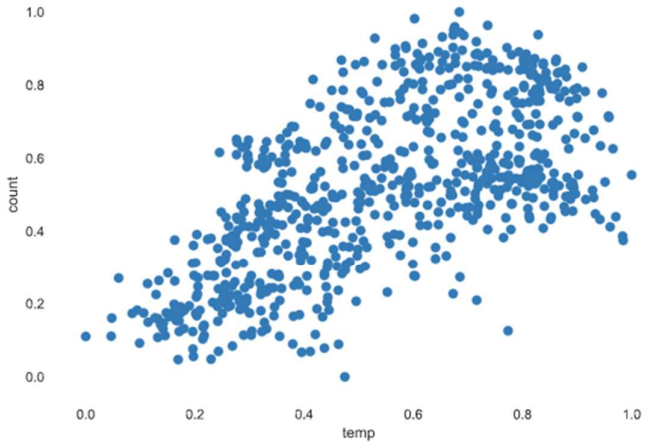
Summarize dataset: 100% 28/28 [00:08<00:00, 2.01it/s, Completed]

Generate report structure: 100% 1/1 [00:13<00:00, 13.74s/it]

Overview | Variables | Interactions | Correlations | Missing values | Sample

temp | humidity | windspeed | count

temp | humidity | windspeed | count




Report generated with pandas-profiling.

The feature selection is done, and we simply need to save the features and labels in some variables.

```
In [63]: x1 = df1.drop(columns=['count'])
         x1
```

Out[63]:

| | temp | humidity | windspeed | year | spring | summer | winter | Working_day | Light_snow | Mist | Dec | Nov | Sep | Sun |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.355170 | 0.828620 | 0.284606 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 1 | 0.379232 | 0.715771 | 0.466215 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 2 | 0.171000 | 0.449638 | 0.465740 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0.175530 | 0.607131 | 0.284297 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0.209120 | 0.449313 | 0.339143 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 726 | 0.243025 | 0.671380 | 0.675656 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 727 | 0.241986 | 0.606684 | 0.274350 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 728 | 0.241986 | 0.774208 | 0.210260 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 729 | 0.245101 | 0.497001 | 0.676936 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 730 | 0.195259 | 0.593830 | 0.273062 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |

731 rows × 14 columns

```
In [64]: y1 = df1['count']
         y1
```

```
Out[64]: 0      0.110792
         1      0.089623
         2      0.152669
         3      0.177174
         4      0.181546
                  ...
         726    0.240681
         727    0.353543
         728    0.151749
         729    0.204096
         730    0.311436
         Name: count, Length: 731, dtype: float64
```

```
In [65]: arr1 = x1.values
         arr1
```

```
Out[65]: array([[0.3551696 , 0.82862005, 0.2846062 , ..., 0.        , 0.        ,
                 1.        ],
                [0.37923205, 0.71577069, 0.46621455, ..., 0.        , 0.        ,
                 0.        ],
                [0.1709998 , 0.44963805, 0.4657404 , ..., 0.        , 0.        ,
                 0.        ],
                ...,
                [0.24198597, 0.77420771, 0.21026043, ..., 0.        , 0.        ,
                 1.        ],
                [0.2451011 , 0.49700051, 0.67693615, ..., 0.        , 0.        ,
                 0.        ],
                [0.19525913, 0.59383033, 0.27306151, ..., 0.        , 0.        ,
                 0.        ]])
```

Train and test datasets are prepared.

```
In [66]: x_train, x_test, y_train, y_test = train_test_split(arr1,y1,test_size=0.20,random_state=100)
```

LassoCV is used to calculate the alpha value for Lasso Regression. The LassoCV class accepts four major arguments: alpha, which is nothing more than a shrinkage factor, cv, which indicates the number of cross validations to perform, max_iter, which is the maximum number of iterations to conduct, and normalize, which is the application of normalization.

```
In [67]: # checking whether our model has consistency or not
         # Regularization and cross validation
         lassocv = LassoCV(alphas=None,cv=50,max_iter=200000,normalize=True)
         lassocv.fit(x_train,y_train)
```

```
Out[67]: LassoCV(cv=50, max_iter=200000, normalize=True)
```

```
In [68]: lassocv.alpha_
```

```
Out[68]: 5.94032207717815e-06
```

Now we take this shrinkage factor and feed it into Lasso Regression. In order to train the model using the fit () method, this is known as hyper parameter tuning. As a result, our model has the smallest slope possible.

```
In [69]: lasso = Lasso(alpha=lassocv.alpha_)
         lasso.fit(x_train,y_train)

Out[69]: Lasso(alpha=5.94032207717815e-06)
```

Lasso regression outperforms linear regression in terms of accuracy, with linear being 75.28 percent accurate and lasso being 78.50 percent accurate.

```
In [70]: score = lasso.score(x_test,y_test)
         print("Lasso Score : {:.2f} %".format(score*100))

         Lasso Score : 78.50 %
```

## Ridge Regression

Ridge regression is a model tuning strategy that can be used to interpret data with multicollinearity. L2 regularization is achieved using this method. When there is a problem with multicollinearity, least-squares are unbiased, and variances are big, the projected values are far from the actual values. Because it uses absolute coefficient values for normalization, Lasso Regression differs from ridge regression. Attempting to select a random alpha value between 0 and 10.

```
In [71]: ridgecv = RidgeCV(alphas=np.random.uniform(0,10,50),cv = 10 , normalize=True)
         ridgecv.fit(x_train,y_train)

Out[71]: RidgeCV(alphas=array([6.1826447 , 6.82301164, 3.251469  , 4.26034477, 9.63628454,
                8.9081008 , 6.35995852, 3.23274121, 7.00307435, 5.31919539,
                7.08400884, 8.66696997, 1.36568887, 5.64283504, 0.24948312,
                4.50973683, 1.64935289, 0.49418116, 1.14848407, 3.98661939,
                0.63664103, 1.96445663, 3.08656749, 5.92421942, 6.30069747,
                0.84409234, 1.29569883, 3.97766383, 2.01516456, 5.63959242,
                7.39089079, 5.68737152, 4.50452416, 9.12513873, 3.35713583,
                0.83526769, 2.51039865, 0.24590512, 4.97973356, 6.85373448,
                7.11072621, 5.15725045, 2.82770932, 7.17321284, 1.88227332,
                2.41966093, 8.07988389, 7.01130004, 8.3125676 , 7.71546657]),
                 cv=10, normalize=True)
```

RidgeCV use the same parameters as LassoCV.

```
In [72]: ridgecv.alpha_

Out[72]: 0.24590511614445365

In [73]: ridge_lr = Ridge(alpha=ridgecv.alpha_)
         ridge_lr.fit(x_train,y_train)

Out[73]: Ridge(alpha=0.24590511614445365)
```

Our alpha is close to zero, so we'll provide it to the Ridge class and use the fit () method to train the model. Our score is nearly identical to that of the Lasso Regression, which is 78.70% accurate.

```
In [74]: score = ridge_lr.score(x_test,y_test)
         print("Lasso Score : {:.2f} %".format(score*100))

         Lasso Score : 78.70 %
```

### ElasticNet

ElasticNet is created by combining Lasso and Ridge. After determining the alpha value for the ElasticNet, get the l1 ratio and pass these arguments to the ElasticNet contructor. Now, train the model with the training dataset and calculate the R square for the ElasticNet , which is our model's score; our model has an accuracy of 78.63 %.

```
In [75]: elastic= ElasticNetCV(alphas=None, cv = 10)
         elastic.fit(x_train,y_train)

Out[75]: ElasticNetCV(cv=10)

In [76]: elastic.alpha_

Out[76]: 0.0001273884940194303

In [77]: elastic.l1_ratio_

Out[77]: 0.5

In [78]: elastic_lr = ElasticNet(alpha=elastic.alpha_ , l1_ratio=elastic.l1_ratio_)

In [79]: elastic_lr.fit(x_train,y_train)

Out[79]: ElasticNet(alpha=0.0001273884940194303)

In [80]: score = elastic_lr.score(x_test,y_test)
         print("Lasso Score : {:.2f} %".format(score*100))

         Lasso Score : 78.63 %
```

We may conclude that our model is consistent based on the three regularization methods because all three models produced nearly identical scores.

## Optimization Algorithm

(Andriy Burkov, 2019). Gradient descent is sensitive to the learning rate used. It is also sluggish when dealing with massive datasets. Fortunately, some important enhancements to this technique have been developed. Minibatch stochastic gradient descent (minibatch SGD) is a variant of the technique that accelerates computation by approximating the gradient using fewer batches (subsets) of training data.

### Stochastic Gradient Descent (SGD)

Gradient Descent has been extended to include stochastic gradient descent. When fresh data is received, any Machine Learning function works with the same goal function to reduce error and generalize. To circumvent the difficulties in Gradient Descent, we choose a tiny number of samples, especially at each stage of the algorithm, we can sample a minibatch selected evenly from the training set. The minibatch size is usually set to be a modest number of instances, ranging from one to a few hundred.

The learning rate is an important parameter for SGD; it is required to lower the learning rate with time. Eta denotes our learning rate, and the L1 or L2 ratio for the optimization method, including the number of cross validations, is passed here.

In [81]:
```
#optimization using stochastic gradient descent
from sklearn.linear_model import SGDRegressor
from sklearn.model_selection import GridSearchCV

sgdr = SGDRegressor(random_state=1, penalty=None)
grid_param = {'eta0': [.0001, .001, .01, .1, 1], 'max_iter': [10000, 20000, 30000, 40000]}

gd_sr = GridSearchCV(estimator=sgdr, param_grid=grid_param, scoring='r2', cv=5)

gd_sr.fit(x_train, y_train)

results = pd.DataFrame.from_dict(gd_sr.cv_results_)
print("Cross-validation results:\n", results)
```

Our model achieves the best score of 81.64 % when eta = 0.1 and maximum iterations = 10000.

In [84]:
```
best_parameters = gd_sr.best_params_
print("Best parameters: ", best_parameters)

best_result = gd_sr.best_score_  # Mean cross-validated score of the best_estimator
print("Best result: ", best_result)

best_model = gd_sr.best_estimator_
print("Intercept: ", best_model.intercept_)
```

```
Best parameters:  {'eta0': 0.1, 'max_iter': 10000}
Best result:  0.8164602099875449
Intercept:  [0.28671983]
```

In [85]:
```
print(pd.DataFrame(zip(x1.columns, best_model.coef_), columns=['Features', 'Coefficients']).sort_values(by=['Coefficients'], asc
```

| | Features | Coefficients |
|---|---|---|
| 0 | temp | 0.352194 |
| 3 | year | 0.239839 |
| 12 | Sep | 0.071406 |
| 13 | Sun | 0.063826 |
| 7 | Working_day | 0.050994 |
| 1 | humidity | 0.024662 |
| 4 | spring | -0.022411 |
| 11 | Nov | -0.034685 |
| 10 | Dec | -0.037041 |
| 5 | summer | -0.053152 |
| 9 | Mist | -0.088130 |
| 2 | windspeed | -0.117133 |
| 6 | winter | -0.190485 |
| 8 | Light_snow | -0.258561 |

# Conclusion

In conclusion, all the above characteristics contribute/highly correlate in predicting the label "count," and individuals ride bikes more frequently on working days and on Sundays when certain parameters such as temperature and humidity are considered.

# References

Aurélien Géron (2019). *Hands-on machine learning with Scikit-Learn and TensorFlow concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, Inc.

Andriy Burkov (2019). *The hundred-page machine learning book*. Quebec, Canada] Andriy Burkov.

Tibshirani, R., Hastie, T., Witten, D. and James, G. (2021). *An Introduction to Statistical Learning: With Applications in R*. Springer.

Codecademy. (2008). *Learn to code - for free | Codecademy*. [online] Available at: https://www.codecademy.com/.