Case Study - Bike Sharing

Context:

Bike-sharing systems are a new generation of traditional bike rentals where the whole process from membership, rental and return has become automatic. Through these systems, the user can easily rent a bike from a particular position and return to another position. Currently, there are about over 500 bike-sharing programs around the world which are composed of over 500 thousand bicycles. Today, there exists a great interest in these systems due to their important role in traffic, environmental, and health issues.

Problem Statement:

'Travel Along' is a new bike-sharing company and wants to expand its customer count and provide better services at a reasonable cost. They have conducted several surveys and collated the data about weather, weekends, holidays, etc. from the past 2 years.

As a recently hired data scientist at 'Travel Along', you have been asked to analyze the patterns in the data and figure out the key areas which can help the organization to grow and manage the customer demands. Further, you need to use this information to predict the count of bikes shared so that the company can take prior decisions for surge hours.

Objective:

- What are the different factors which affect the target variable? What business recommendations can we give based on the analysis?
- How can we use different ensemble techniques - Bagging, Boosting, and Stacking to build a model to predict the count of bikes rented?

Data Description:

The bike-sharing rental process is highly correlated to the environmental and seasonal settings. For instance, weather conditions, precipitation, the day of week, season, the hour of the day, etc. can affect the rental behaviors.

- instant: record index
- dteday : date
- season : season (1:spring, 2:summer, 3:fall, 4:winter)

- yr : year (0: 2011, 1:2012)
- mnth : month (1 to 12)
- hr : hour (0 to 23)
- holiday : whether the day is holiday or not
- weekday : day of the week
- workingday : if day is neither weekend nor holiday then 1, otherwise is 0.
- weathersit :
  - 1: Clear, Few clouds, Partly cloudy
  - 2: Mist + Cloudy, Mist + Broken clouds, Mist + Few clouds, Mist
  - 3: Light Snow, Light Rain + Thunderstorm + Scattered clouds, Light Rain + Scattered clouds
  - 4: Heavy Rain + Ice Pallets + Thunderstorm + Mist, Snow + Fog
- temp : Normalized temperature in Celsius. The values are divided by 41 (max)
- atemp: Normalized feeling temperature in Celsius. The values are divided to 50 (max). The “feel like” temperature relies on environmental data including the ambient air temperature, relative humidity, and wind speed to determine how weather conditions feel to bare skin.
- hum: Normalized humidity. The values are divided by 100 (max)
- windspeed: Normalized wind speed. The values are divided by 67 (max)
- casual: count of casual users
- registered: count of registered users
- cnt: count of total rental bikes including both casual and registered

Note: The first section of the notebook is the section that has been covered multiple times in the previous case studies. For this discussion, this part can be skipped and we can directly refer to this summary of observations from EDA.

Overview of the dataset

Let's start by importing libraries we need.

In [1]:
```
import warnings
warnings.filterwarnings("ignore")

import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import BaggingRegressor,RandomForestRegressor, GradientBoosti
from xgboost import XGBRegressor
from sklearn import metrics
from sklearn.model_selection import GridSearchCV, train_test_split
```

In [2]:
```
#Loading dataset
```

```python
data=pd.read_csv("hour.csv")
```

View the first 5 rows of the dataset.

In [3]:
```python
data.head()
```

Out[3]:

	instant	dteday	season	yr	mnth	hr	holiday	weekday	workingday	weathersit	temp	ate
0	1	2011-01-01	1	0	1	0	0	6	0	1	0.24	0.2
1	2	2011-01-01	1	0	1	1	0	6	0	1	0.22	0.2
2	3	2011-01-01	1	0	1	2	0	6	0	1	0.22	0.2
3	4	2011-01-01	1	0	1	3	0	6	0	1	0.24	0.2
4	5	2011-01-01	1	0	1	4	0	6	0	1	0.24	0.2

Check data types and number of non-null values for each column.

In [4]:
```python
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 17379 entries, 0 to 17378
Data columns (total 17 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   instant     17379 non-null  int64
 1   dteday      17379 non-null  object
 2   season      17379 non-null  int64
 3   yr          17379 non-null  int64
 4   mnth        17379 non-null  int64
 5   hr          17379 non-null  int64
 6   holiday     17379 non-null  int64
 7   weekday     17379 non-null  int64
 8   workingday  17379 non-null  int64
 9   weathersit  17379 non-null  int64
 10  temp        17379 non-null  float64
 11  atemp       17379 non-null  float64
 12  hum         17379 non-null  float64
 13  windspeed   17379 non-null  float64
 14  casual      17379 non-null  int64
 15  registered  17379 non-null  int64
 16  cnt         17379 non-null  int64
dtypes: float64(4), int64(12), object(1)
memory usage: 2.3+ MB
```

- We can see that there are total of 17 columns and 17,379 rows in the dataset.
- All columns' data type is either integer or float except one column - 'dteday' which is of the object type.

- The number of non-null values of each column is equal to the number of total rows in the dataset i.e. no null value. We can further confirm this using `isna()` method.

In [5]:
```
data.isna().sum()
```

Out[5]:
```
instant       0
dteday        0
season        0
yr            0
mnth          0
hr            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
```

- There are no missing values in the data.

Summary of the dataset

In [6]:
```
# Summary of continuous columns
data[['temp','atemp','hum','windspeed','cnt']].describe().T
```

Out[6]:

	count	mean	std	min	25%	50%	75%	max
temp	17379.0	0.496987	0.192556	0.02	0.3400	0.5000	0.6600	1.0000
atemp	17379.0	0.475775	0.171850	0.00	0.3333	0.4848	0.6212	1.0000
hum	17379.0	0.627229	0.192930	0.00	0.4800	0.6300	0.7800	1.0000
windspeed	17379.0	0.190098	0.122340	0.00	0.1045	0.1940	0.2537	0.8507
cnt	17379.0	189.463088	181.387599	1.00	40.0000	142.0000	281.0000	977.0000

- The mean and median value of temperature is approx 0.50
- The mean and median value for 'atemp' is approx 0.47 and 0.48 respectively
- The mean and median value for 'hum' is approx 0.627 and 0.63 respectively
- Wind speed has some extreme values at the right end
- The target variable seems to have skewed distribution as higher values are on the right. We will explore this further.

Number of unique values in each column

In [7]:
```python
data.nunique()
```

Out[7]:
```
instant       17379
dteday          731
season            4
yr                2
mnth             12
hr               24
holiday           2
weekday           7
workingday        2
weathersit        4
temp             50
atemp            65
hum              89
windspeed        30
casual          322
registered      776
cnt             869
dtype: int64
```

- We can drop 'instant' column as it is an ID variable and will not add value to the model.
- We can drop 'dteday' column as it just contains dates of 731 days i.e. 2 years. This will not add value to the model.

In [8]:
```python
#Dropping two columns from the dataframe
data.drop(columns=['instant','dteday'], inplace=True)
```

Number of observations in each category

In [9]:
```python
cat_cols=['season','yr','holiday','workingday','weathersit']

for column in cat_cols:
    print(data[column].value_counts())
    print('-'*30)
```

```
3    4496
2    4409
1    4242
4    4232
Name: season, dtype: int64
------------------------------
1    8734
0    8645
Name: yr, dtype: int64
------------------------------
0    16879
1      500
Name: holiday, dtype: int64
------------------------------
1    11865
0     5514
Name: workingday, dtype: int64
------------------------------
1    11413
2     4544
3     1419
4        3
Name: weathersit, dtype: int64
------------------------------
```

- The number of observations from year 0 i.e. 2011 is slightly more than the number of observations from year 1 i.e. 2012.
- As expected, the number of non-holidays and working days are much higher than the number of holidays and non-working days respectively.
- We have only 3 observations where weathersit=4 and most common is 1 i.e. clear or partly cloudy.

EDA

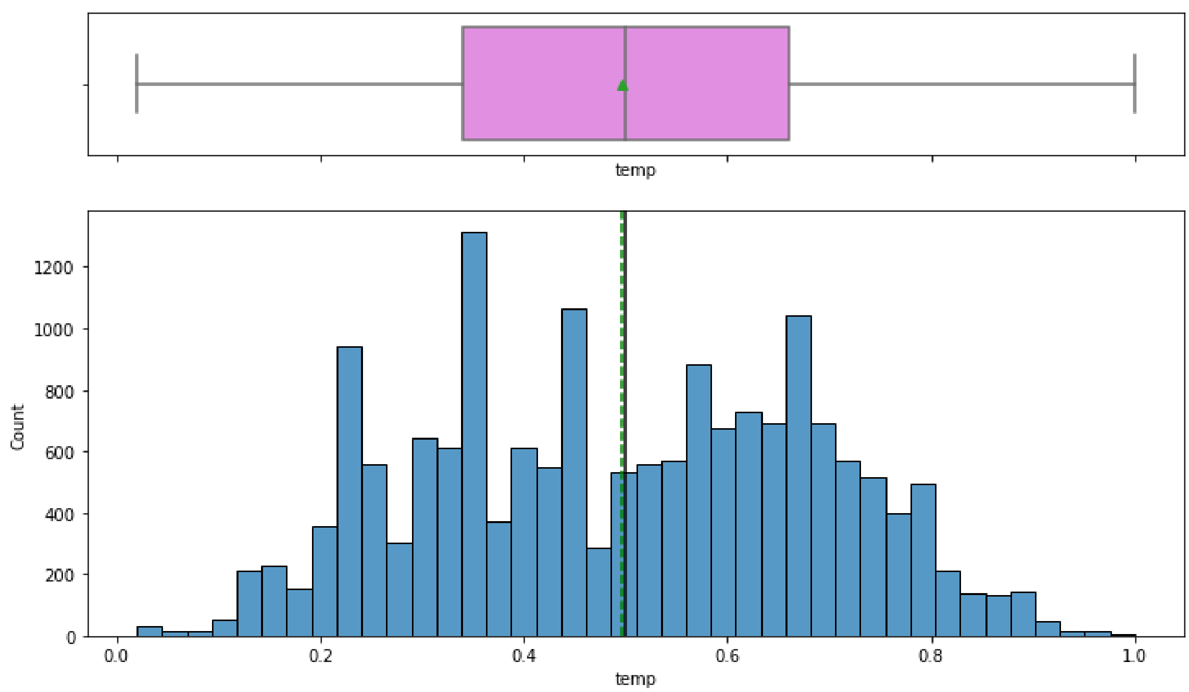
Univariate analysis

In [10]:
```
# function to plot a boxplot and a histogram along the same scale.


def histogram_boxplot(data, feature, figsize=(12, 7), kde=False, bins=None):
    """
    Boxplot and histogram combined

    data: dataframe
    feature: dataframe column
    figsize: size of figure (default (12,7))
    kde: whether to show the density curve (default False)
    bins: number of bins for histogram (default None)
    """
    f2, (ax_box2, ax_hist2) = plt.subplots(
        nrows=2,  # Number of rows of the subplot grid= 2
```

```
        sharex=True,  # x-axis will be shared among all subplots
        gridspec_kw={"height_ratios": (0.25, 0.75)},
        figsize=figsize,
    )  # creating the 2 subplots
    sns.boxplot(
        data=data, x=feature, ax=ax_box2, showmeans=True, color="violet"
    )  # boxplot will be created and a star will indicate the mean value of the col
    sns.histplot(
        data=data, x=feature, kde=kde, ax=ax_hist2, bins=bins, palette="winter"
    ) if bins else sns.histplot(
        data=data, x=feature, kde=kde, ax=ax_hist2
    )  # For histogram
    ax_hist2.axvline(
        data[feature].mean(), color="green", linestyle="--"
    )  # Add mean to the histogram
    ax_hist2.axvline(
        data[feature].median(), color="black", linestyle="-"
    )  # Add median to the histogram
```

Observations on temperature

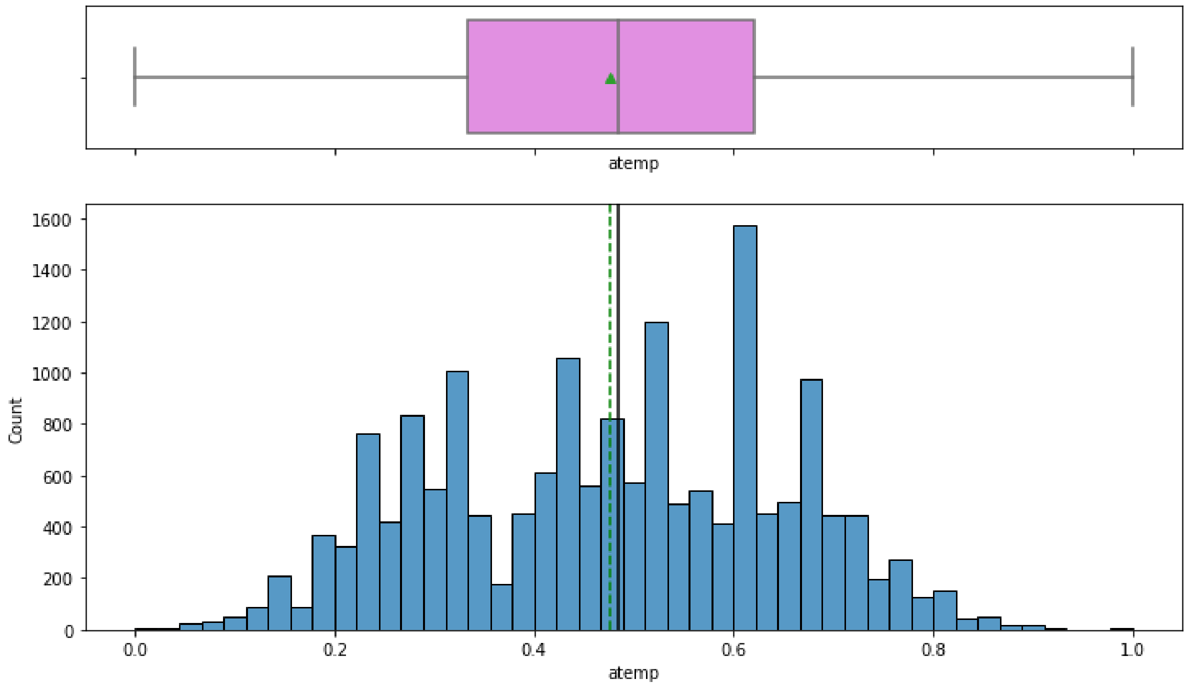
In [11]:
```
histogram_boxplot(data, "temp")
```



- The temperature has an approx symmetric distribution with mean and median equal to 0.5
- As evident from the boxplot, there are no outliers in the distribution for this variable

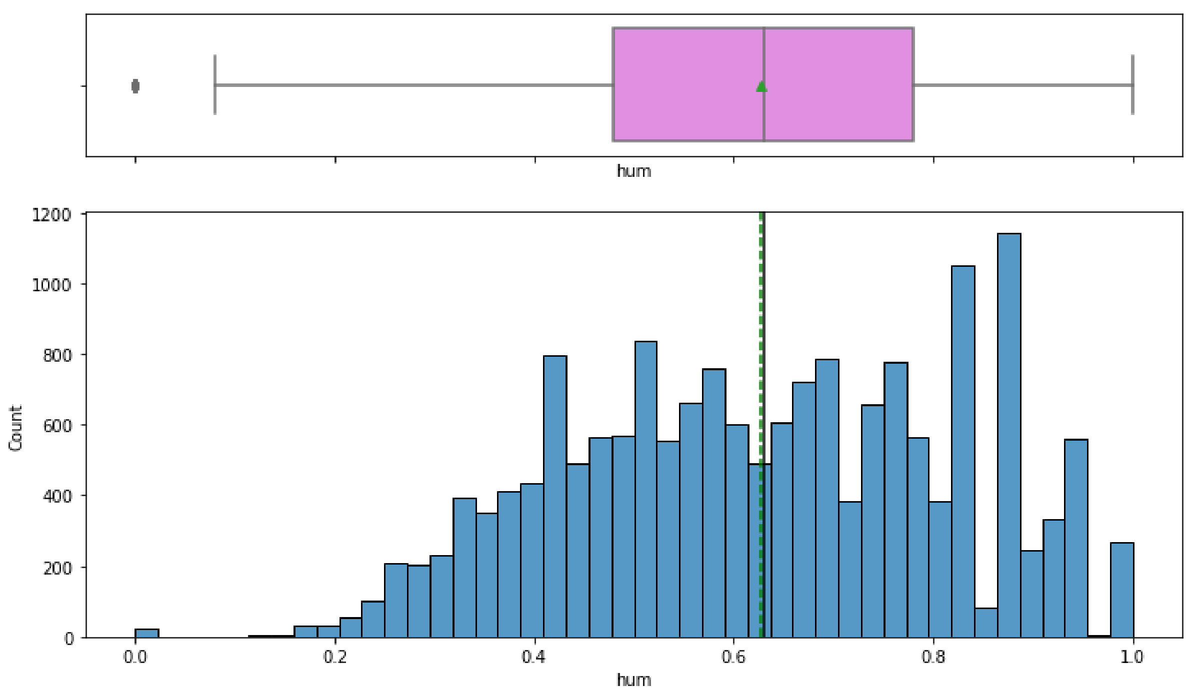
Observations on 'feel like temperature'

In [12]:
```
histogram_boxplot(data, "atemp")
```



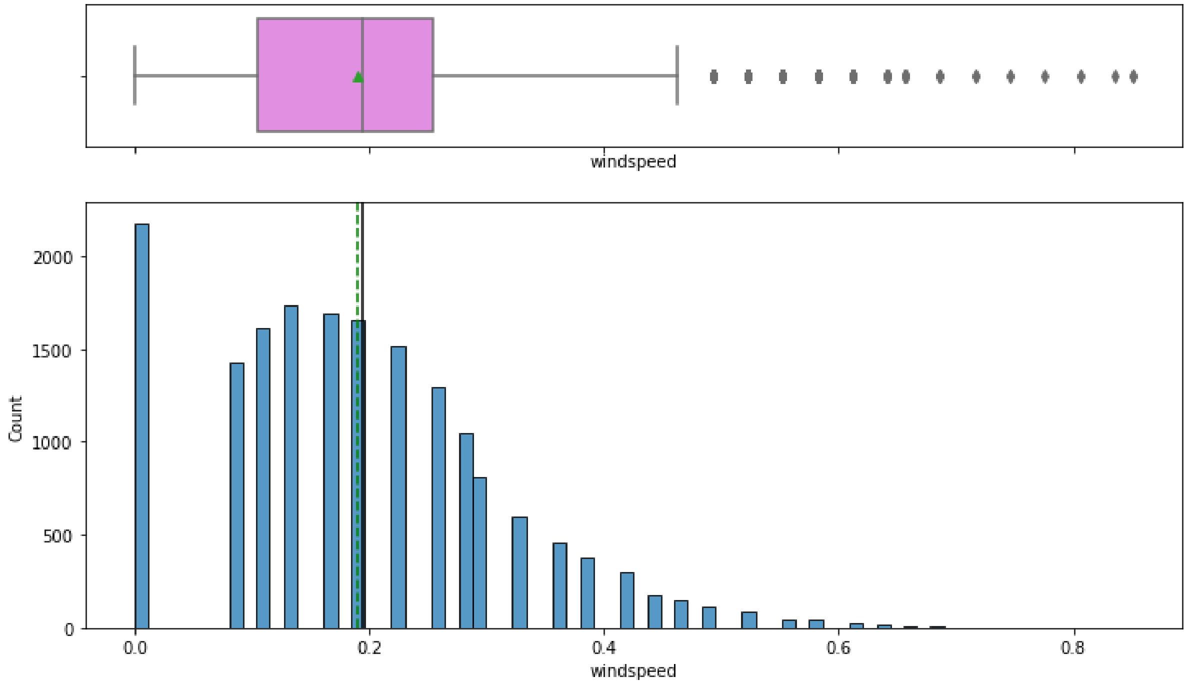
- Same as temperature, the distribution for feel like the temperature is also symmetrically distributed
- There are no outliers in the distribution of these variables

In [13]:
```
histogram_boxplot(data,"hum")
```



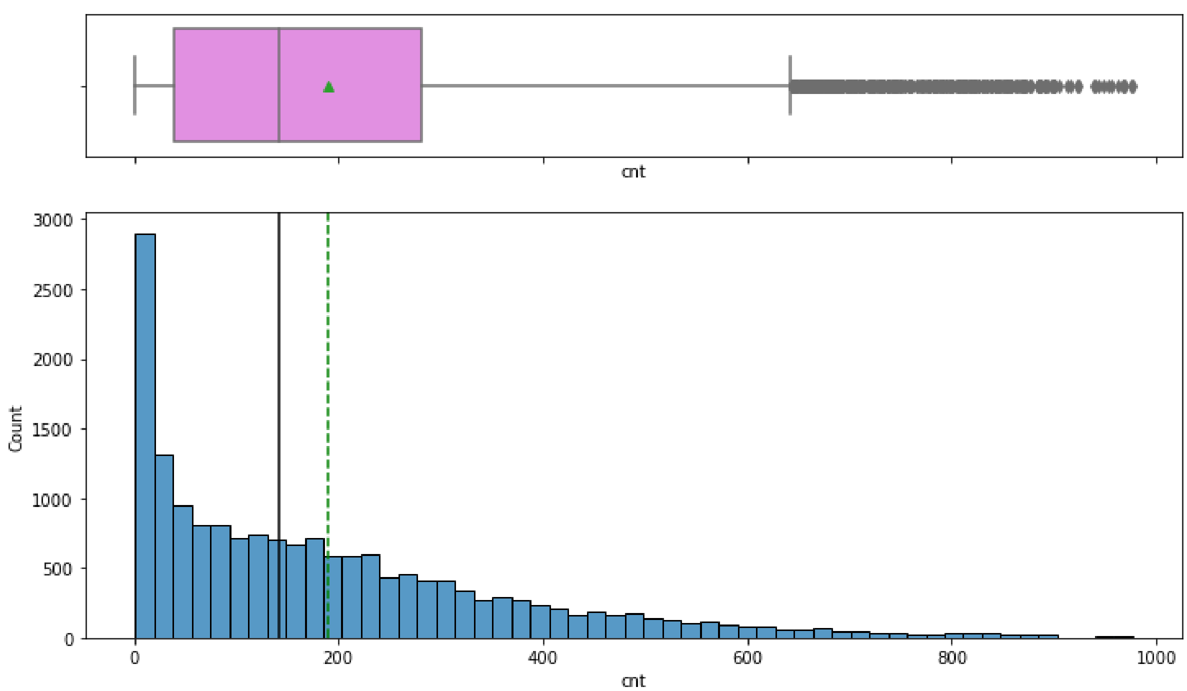
- Most of the values are concentrated in the middle i.e. 0.4 to 0.8
- Humidity with the value equal to 0 is an outlier
- The distribution is approx normally distributed with mean and median equal to 0.63

In [14]:
```
histogram_boxplot(data,'windspeed')
```



- Wind speed has a right-skewed distribution and 0 has the highest count among all observations
- Distribution is not symmetric but mean and median are approx equal with a value equal to 0.19
- There are many outliers in this variable

In [15]:
```
histogram_boxplot(data,'cnt')
```



In [16]:
```
#Top 5 highest values
```

```
data['cnt'].nlargest()
```

Out[16]:

```
14773    977
14964    976
14748    970
14725    968
15084    967
Name: cnt, dtype: int64
```

- The target variable i.e. the count of bikes rented has a right-skewed distribution
- The range of values is very large with many observations being less than 10 counts and some being greater than 900 count
- As evident from the boxplot, there are many outliers

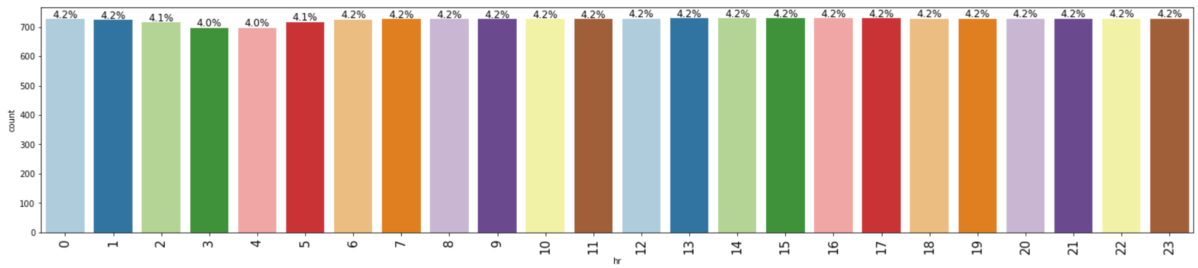
Function to create barplots that indicate percentage for each category

In [17]:

```
# function to create labeled barplots


def labeled_barplot(data, feature, perc=False, n=None):
    """
    Barplot with percentage at the top

    data: dataframe
    feature: dataframe column
    perc: whether to display percentages instead of count (default is False)
    n: displays the top n category levels (default is None, i.e., display all level
    """

    total = len(data[feature])  # length of the column
    count = data[feature].nunique()
    if n is None:
        plt.figure(figsize=(count + 1, 5))
    else:
        plt.figure(figsize=(n + 1, 5))

    plt.xticks(rotation=90, fontsize=15)
    ax = sns.countplot(
        data=data,
        x=feature,
        palette="Paired",
        order=data[feature].value_counts().index[:n].sort_values(),
    )

    for p in ax.patches:
        if perc == True:
            label = "{:.1f}%".format(
                100 * p.get_height() / total
            )  # percentage of each class of the category
        else:
            label = p.get_height()  # count of each level of the category

        x = p.get_x() + p.get_width() / 2  # width of the plot
        y = p.get_height()  # height of the plot
```

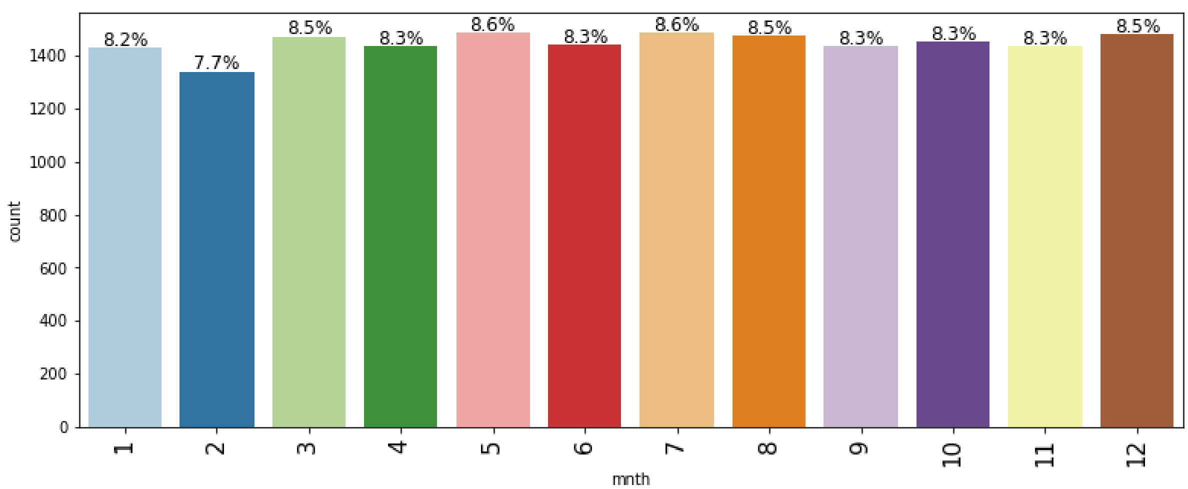
```
        ax.annotate(
            label,
            (x, y),
            ha="center",
            va="center",
            size=12,
            xytext=(0, 5),
            textcoords="offset points",
        )  # annotate the percentage

    plt.show()  # show the plot
```

In [18]:
```
labeled_barplot(data, "hr",perc=True)
```



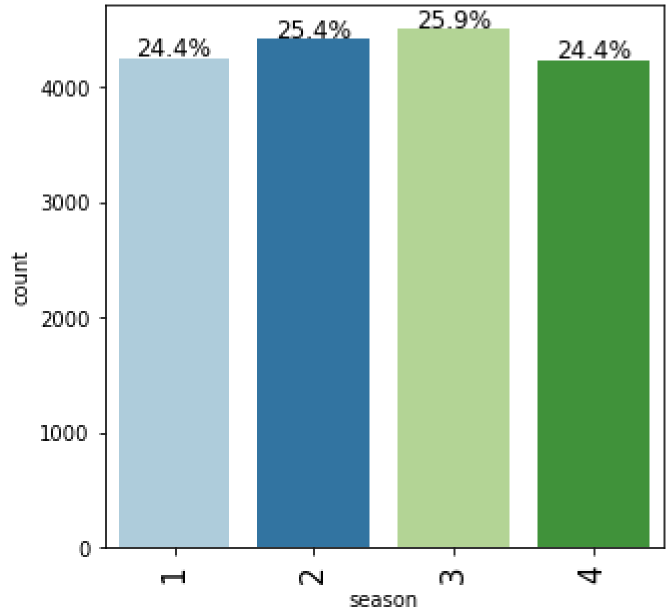
- Each hour i.e. 0 to 23 has approx 4% observations in the data

In [19]:
```
labeled_barplot(data,'mnth',perc=True)
```



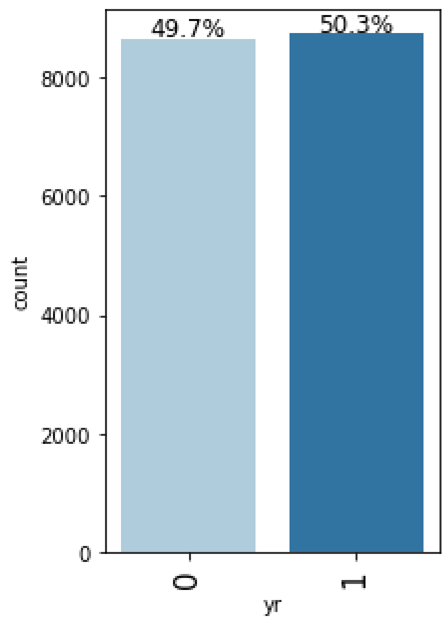
- Each month i.e. 1 to 12 has approx 8.5% observations in the data
- Month 2 has slightly less number of observations compared to other months

In [20]:
```
labeled_barplot(data,'season',perc=True)
```



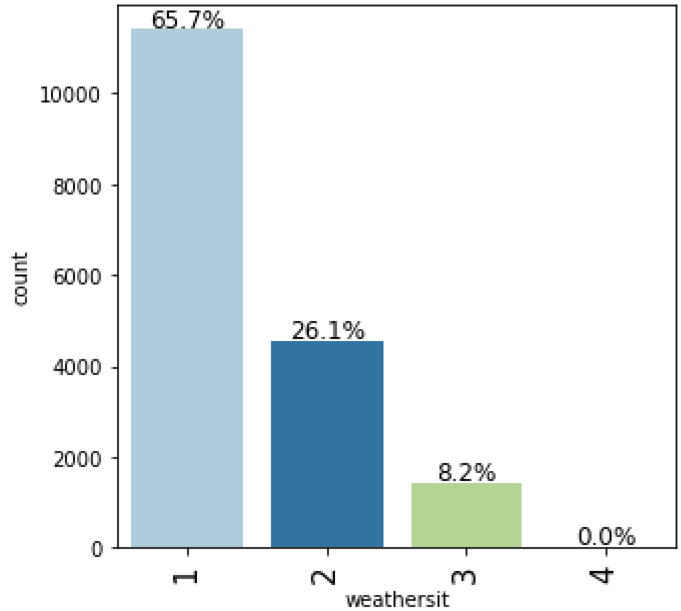
- Each season has approx 24% observations in the data

In [21]:
```
labeled_barplot(data,'yr',perc=True)
```



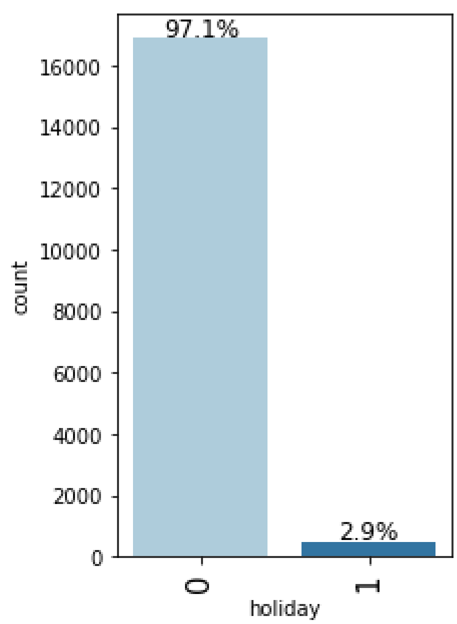
- Both years have approx equal number of observations in the data

In [22]:
```
labeled_barplot(data,'weathersit',perc=True)
```



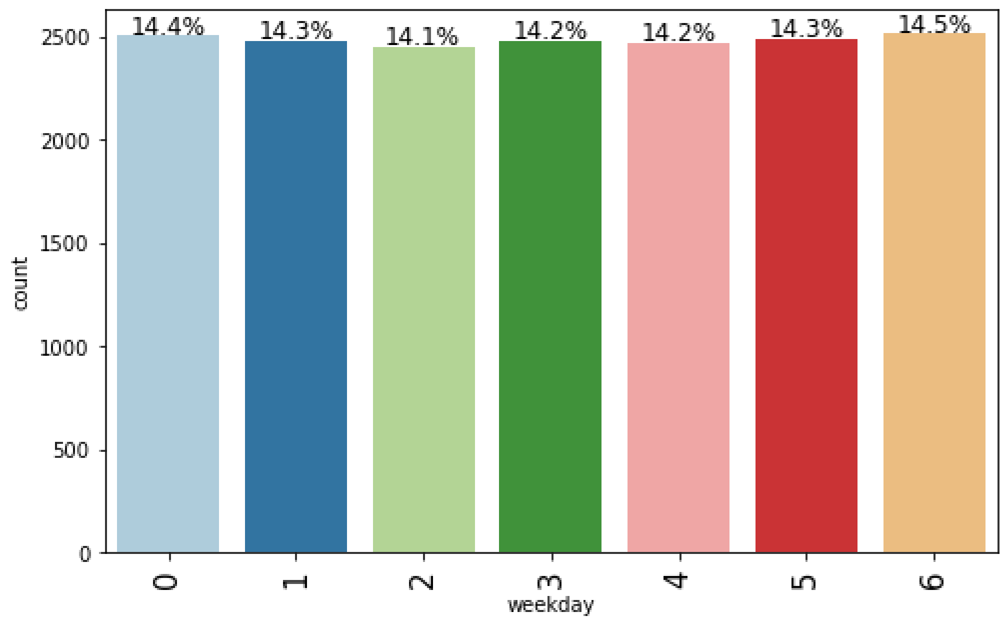
- Season 1 has the highest percentage of observations i.e. 65.7%
- Season 2 and season 3 have 26.1% and 8.2% observations respectively
- We saw earlier that season 4 has only 3 observations in the data. Here, it shows 0% observations due to rounding off.

In [23]:
```
labeled_barplot(data,'holiday',perc=True)
```



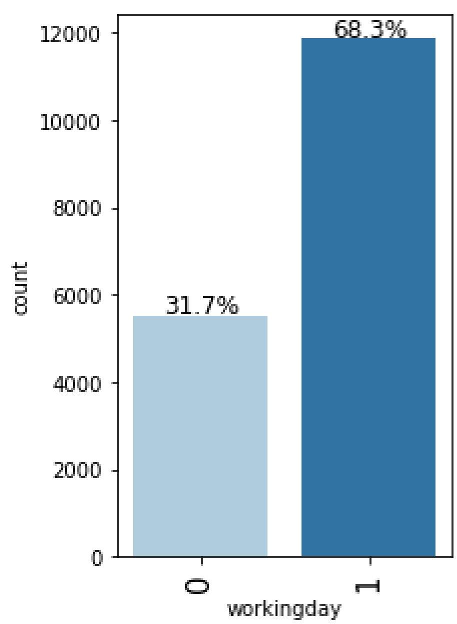
- As expected the percentage for non-holidays is much more than holidays.
- There are 97% non-holidays observations and only 3% for holidays

In [24]:
```
labeled_barplot(data,'weekday',perc=True)
```



- Each weekday i.e. 0 to 6 has approx 14% observations in the data.

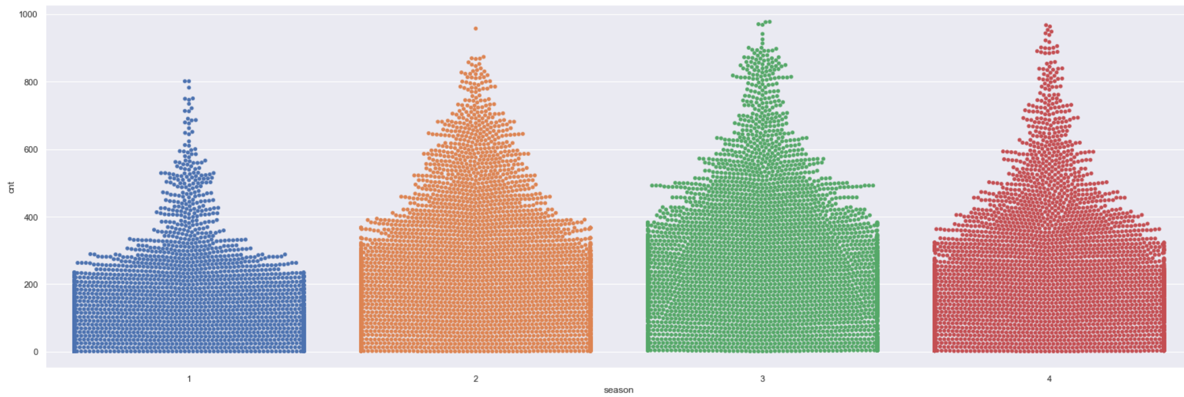
In [25]:
```
labeled_barplot(data,'workingday',perc=True)
```



- As expected, the number of observations for working days is higher than the number of observations for non-working days.
- There are approx 68% observations for working days and 32% observations for non-working days.

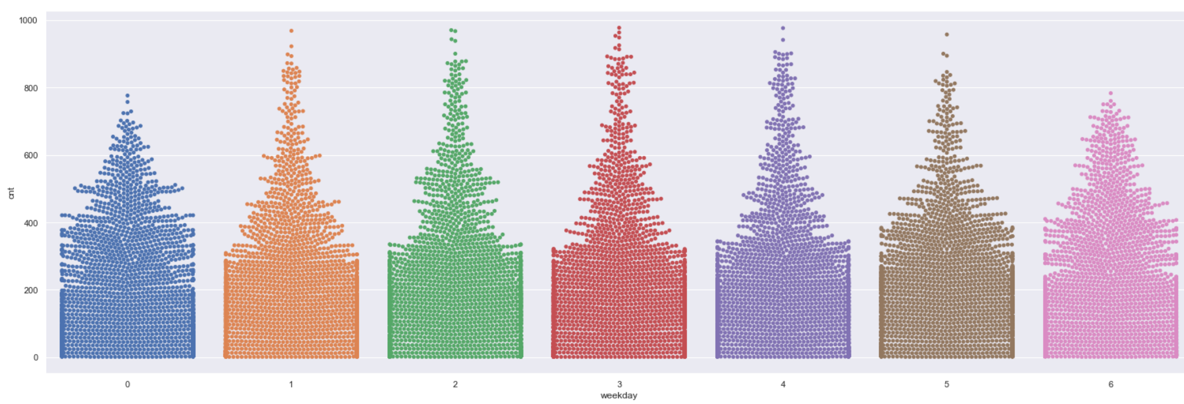
Bivariate analysis

In [26]:
```
sns.set(rc={'figure.figsize':(21,7)})
sns.catplot(x="season", y="cnt", kind="swarm", data=data, height=7, aspect=3);
```



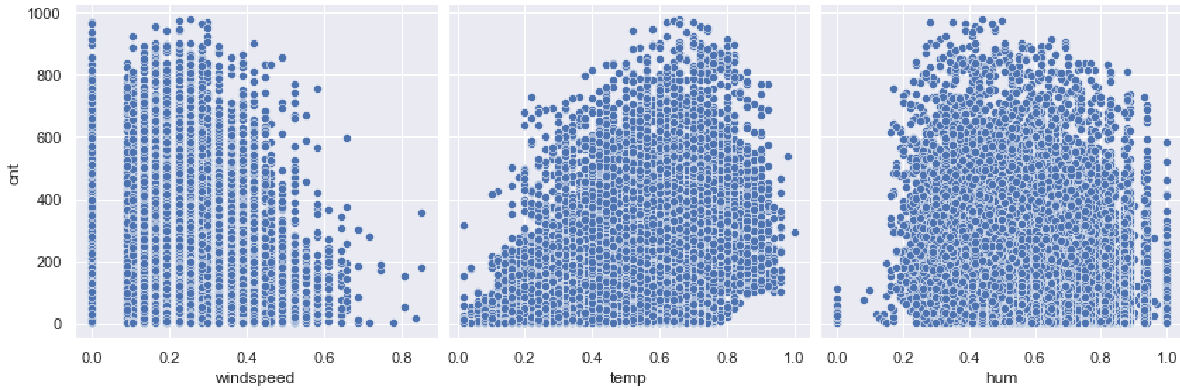
- The lowest number of bikes are rented in the first season
- The highest number of bikes are shared in 3rd season
- This can be due to the relatively high temperature in season 1 i.e. spring as compared to season 3 i.e. fall

In [27]:
```
sns.set(rc={'figure.figsize':(21,7)})
sns.catplot(x="weekday", y="cnt", kind="swarm", data=data, height=7, aspect=3);
```



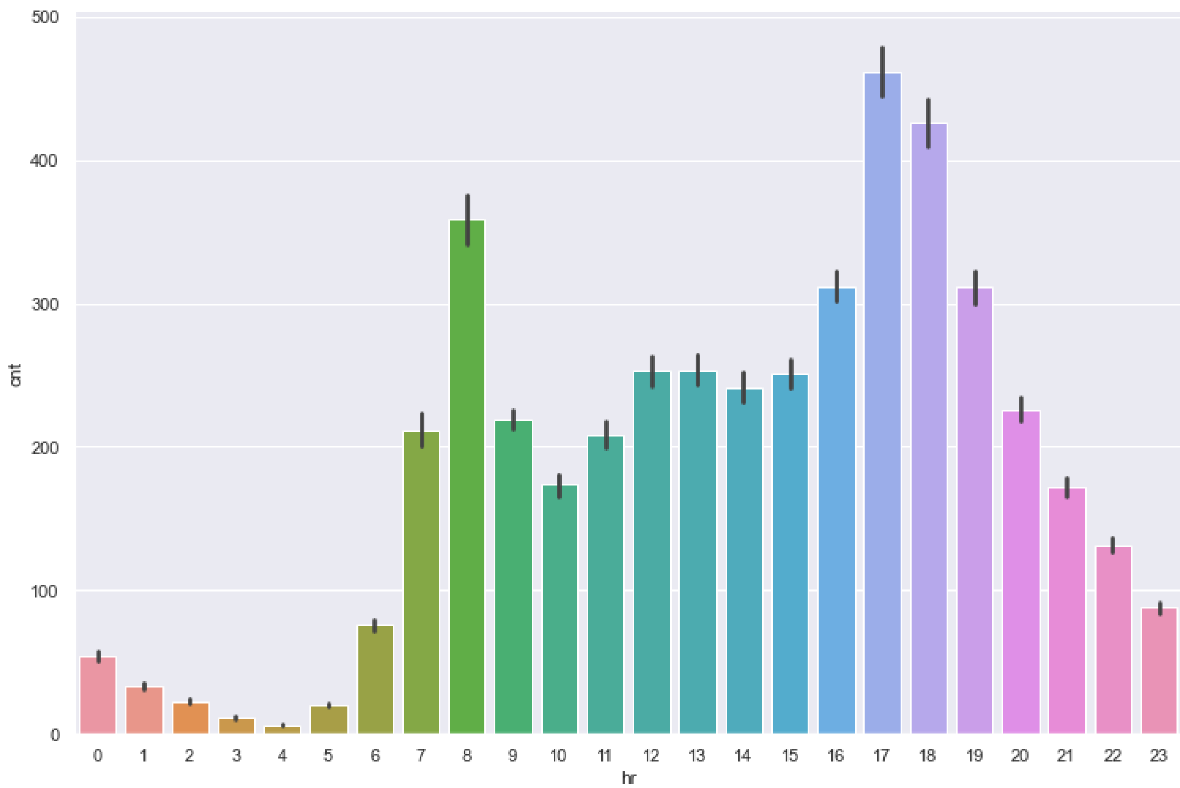
- Weekends i.e. weekday=0 and weekday=6 have a low count of bikes rented and it is less varying.
- Working days have a higher count of bikes rented and have more variation in the count and there are some outliers for days from 1 to 5.
- This can be due to closed schools/offices on weekends.

In [28]:
```
sns.pairplot(
    data,
    x_vars=["windspeed", "temp", "hum"],
    y_vars=["cnt"],
    height=4,
    aspect=1
    );
```



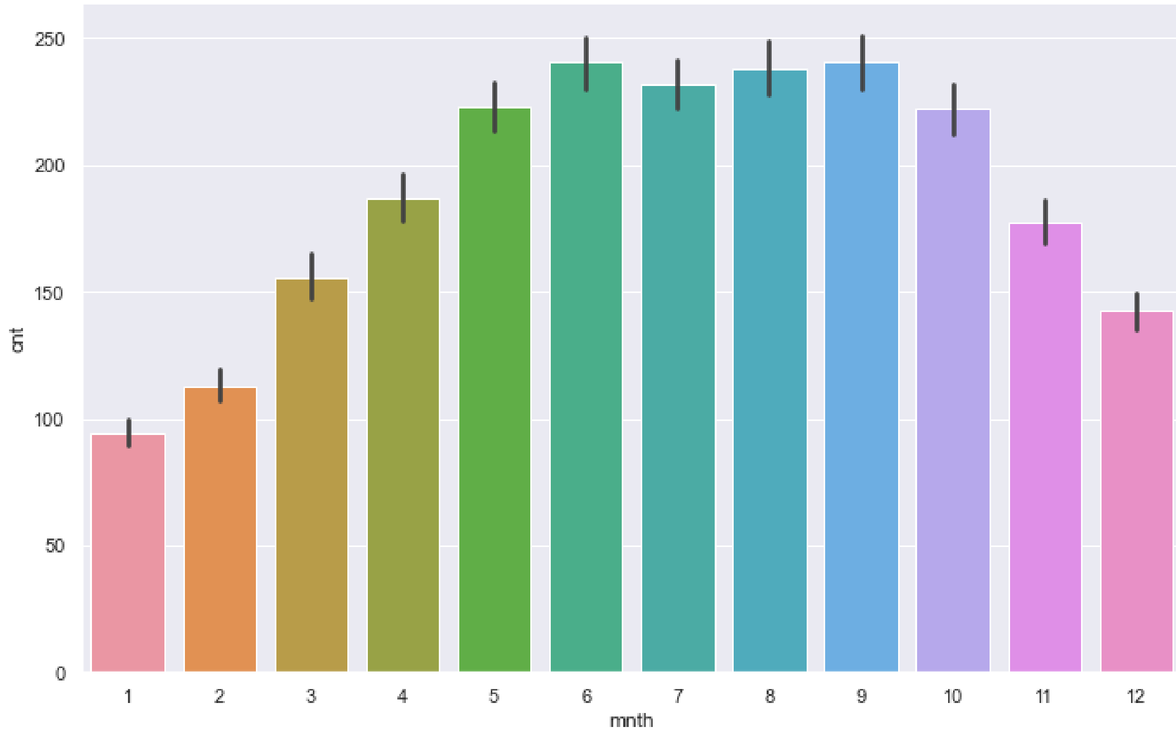
- We can see that count of bikes rented is low when the temperature is very low or very high. The same is true for humidity.
- Count of bikes rented is high when wind speed is low.

In [29]:
```
sns.catplot(x="hr", y="cnt", data=data, kind='bar', height=7, aspect=1.5, estimator
```



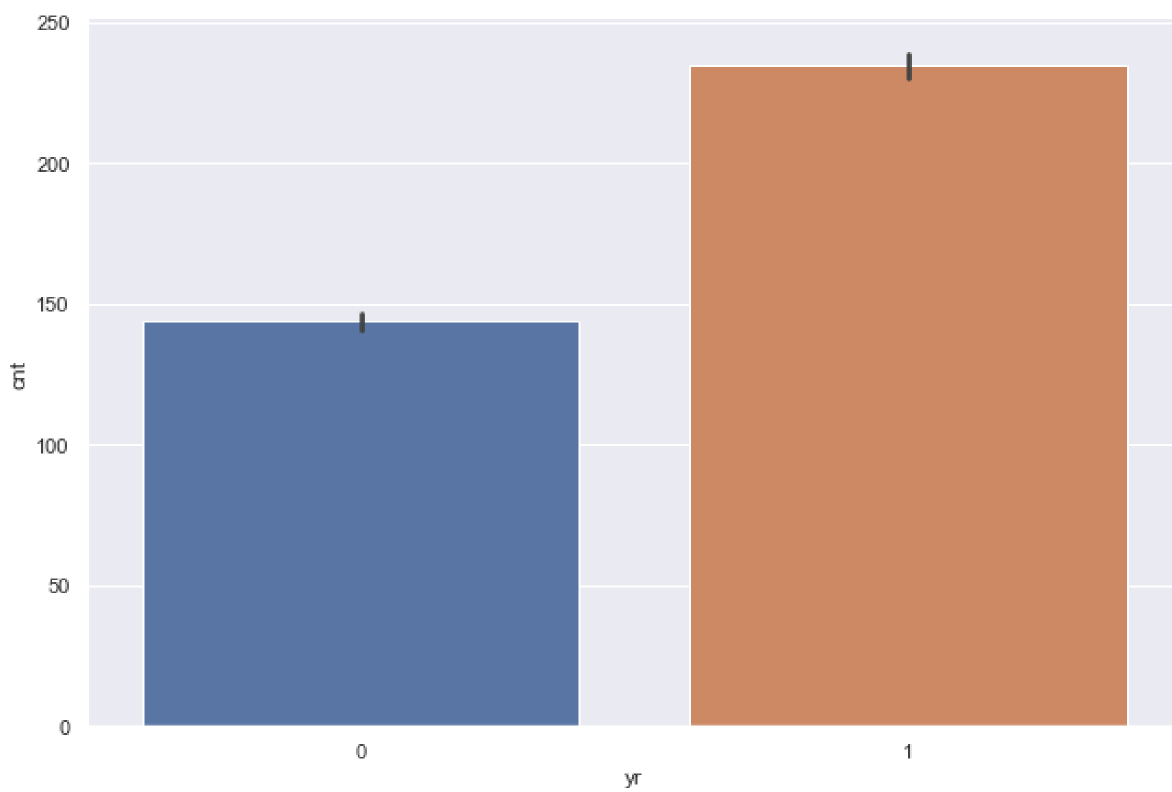
- We can see the average number of bikes rented is high at 8 AM and 5-6 PM, this can be due to office/school/college timings.
- The average number of bikes rented is very low for night time i.e. 12 AM to 5 AM.

In [30]:
```
sns.catplot(x="mnth", y="cnt", data=data, kind='bar', height=6, aspect=1.6, estimat
```



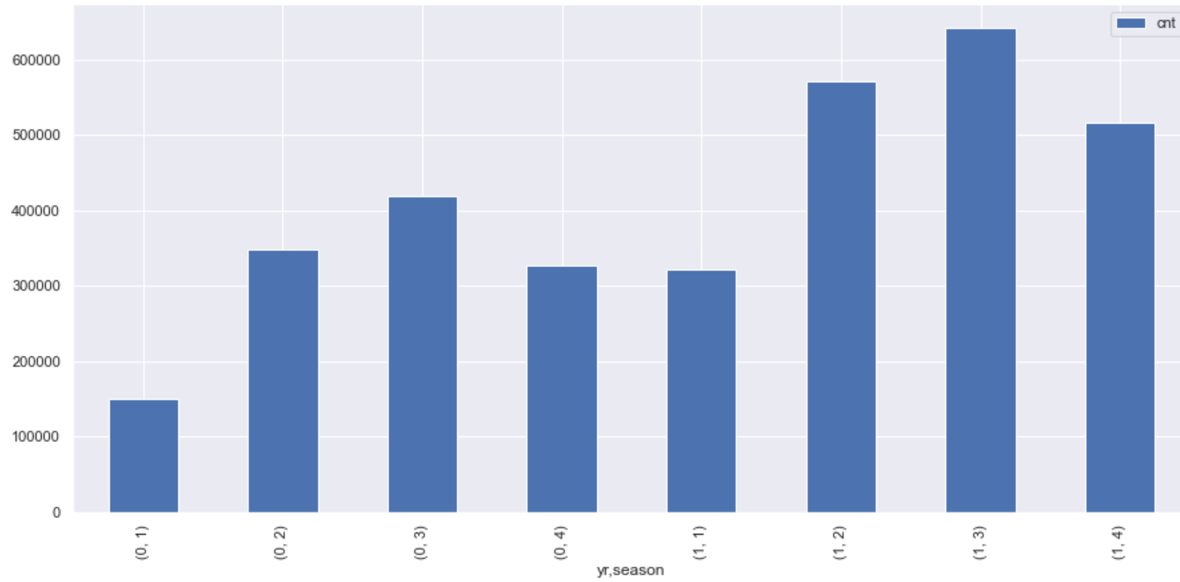
- The average number of bikes rented is low for months - December, January, February. This can be due to the cold weather in these months.
- The average number of bikes rented is consistently high for months from May to October.

In [31]:
```
sns.catplot(x="yr", y="cnt", data=data, kind='bar', size=6, aspect=1.5, estimator=n
```



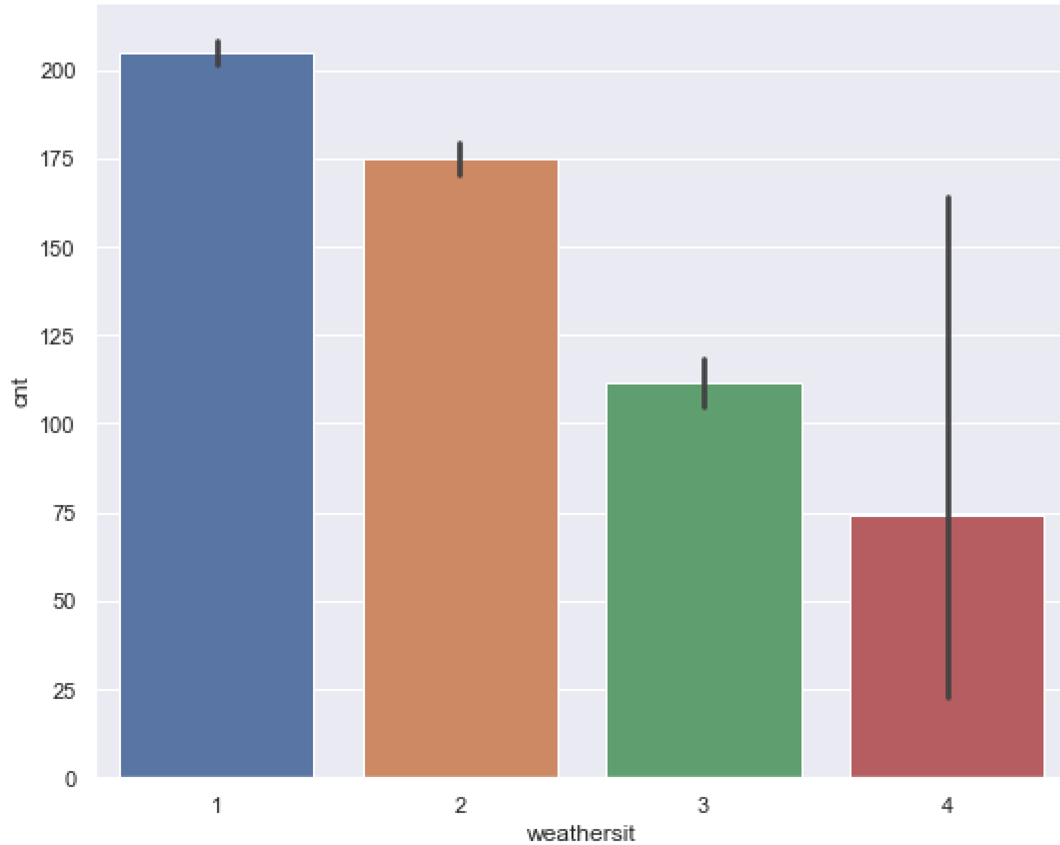
- The average count of bikes rented is high for the year 2012 as compared to 2011.
- Let's check this for each season of both years and observe if the count in each season has increased or in just 1 or 2 seasons.

In [32]:
```
sns.set(rc={'figure.figsize':(15,7)})
pd.pivot_table(data=data, index=['yr', 'season'], values='cnt', aggfunc=np.sum).plo
```



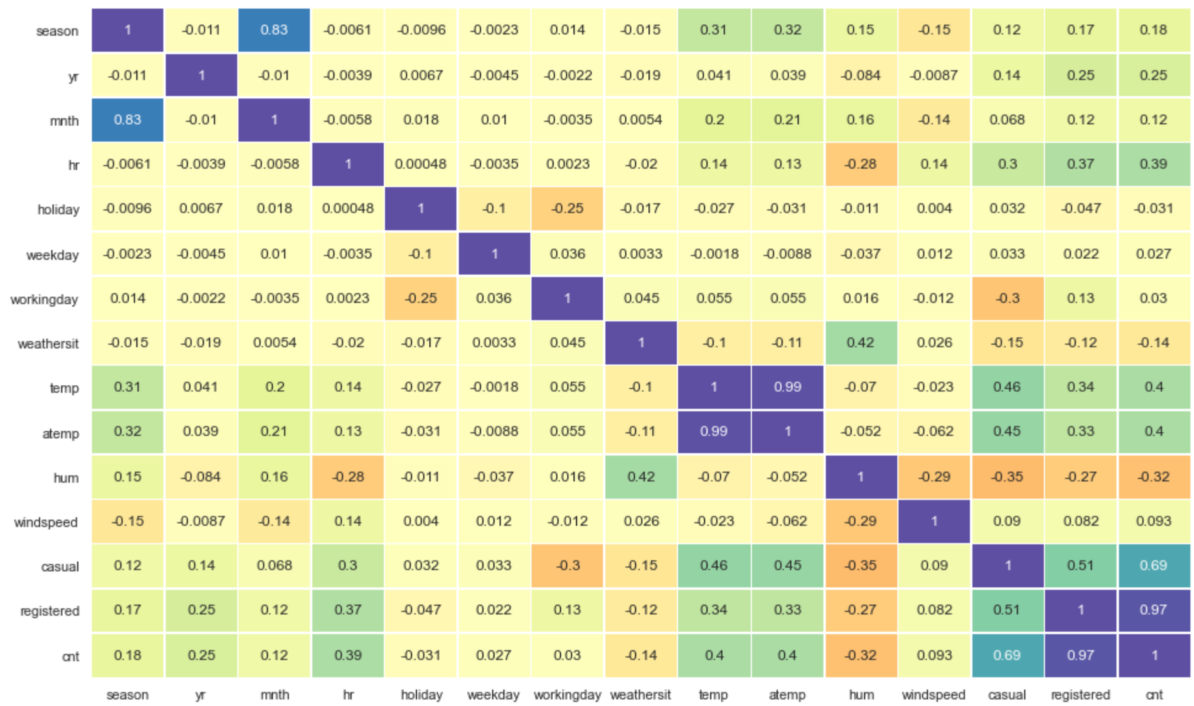
- We can see that number of bikes rented is higher in the year 2012 for each season as compared to seasons in 2011.
- This shows that bike-sharing is becoming more popular with time.

In [33]:
```
sns.catplot(x="weathersit", y='cnt', kind='bar', data=data, size=6, aspect=1.25, es
```



- As expected, the average count of bikes rented is much higher on clear or partly cloudy days compared to snowy or rainy days.
- This implies that the average count of bikes rented is hugely affected by the weather.

In [34]:
```
sns.set(rc={'figure.figsize':(16,10)})
sns.heatmap(data.corr(),
            annot=True,
            linewidths=.5,
            center=0,
            cbar=False,
            cmap="Spectral")
plt.show()
```



- We can see that temperature and feel like temperature are almost perfectly correlated
- Month and season have a high positive correlation among them
- As count is the addition of two columns - Casual and registered. We can drop these two columns because if we have casual and registered count then making a model won't make sense as we can simply add them. We would not have these 2 columns while predicting new observations

Summary of EDA

Data Description:

- Dependent variable is "cnt" which is representing the count of bikes rented and it is of object data type.
- dteday is of object data type while rest of the features are either integer or float.
- There are no missing values in the dataset.

Data Cleaning:

- Instant is an ID variable so it is dropped from the data.
- We can drop 'dteday' column as it just contains dates of 731 days i.e. 2 years. This will not add value to the model.

Observations from EDA:

- `Temperature` : It has an approximately symmetric distribution with mean and median equal to 0.5.

- `Feel like temperature` : It is having a fairly symmetrical distribution with no outliers.
- `Hum` : Humidity is normally distributed with a mean and median equal to 0.63.
- `Windspeed` : It is having a right-skewed distribution. Even though the distribution is unsymmetric but mean and median are approximately equal to 0.19
- `cnt` : The target variable is right-skewed with having a good number of outliers.
- `hr` : It is having equal distribution throughout.
- `mnth` : Each month is having almost 8.5% of observations.
- `season` : Each season has approx 24% of data.
- `yr` : Each year is having 50% of the observations.
- `Weathersit` : Season 1 is having a maximum percentage of observations. (66% almost)
- `holiday` : As expected the percentage for non-holidays is much more than holidays.
- `weekday` : Each weekday is having approx 14% of observations.
- `Workingday` : The number of observations for working days(68%) is higher than the non-working days.
- **Count with season**
  - The lowest number of bikes are rented in the first season and the highest in 3rd season. A high temperature in season 1 might be the reason for this.
- **Count with Weekdays**
  - Weekday 0 and 6 are having a low count of bikes rented. Working days having more count of rented bikes with more variation in the same. The reason might be due to the closing of schools and offices on weekends.
- **Count with windspeed, temp, and hum**
  - When temperature is very low count of bikes rented is very low or very high. While in low wind speed the count is high.
- **Count with hour**
  - At 8 AM and 5-6PM the count of rented bikes is high. This is due to office, school or college timings.
- **Count with month**
  - For December, January, February average number of bikes is low. The expected reason is cold weather. From May to October average number of rented bikes is high.

- **Count with year**
    - In 2012 the count of bikes rented is higher than that in 2011 in each season. This shows the bike-sharing is becoming more popular in 2012.
- **Count with year and season**
    - In clear days(without cloud) the average count of bikes rented is much higher than snowy or rainy days.It shows that weather affects count of rental bikes.
- **Count with weathersit**
    - Temperature and feel like temperature are deeply correlated, Month and season have a high positive correlation.

Actions for data pre-processing:

- As count is the addition of two columns - Casual and registered. We can drop these two columns because if we have casual and registered count then making a model won't make sense as we can simply add them. We would not have these 2 columns while predicting new observations

In [35]:
```
#Dropping columns - casual and registered
data.drop(columns=['casual','registered'], inplace=True)
```

Split the dataset

In [36]:
```
# Separating features and the target column
X = data.drop('cnt', axis=1)
y = data['cnt']
```

In [37]:
```
# Splitting the data into train and test sets in 70:30 ratio
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, random_st
```

In [38]:
```
X_train.shape, X_test.shape
```

Out[38]:
```
((12165, 12), (5214, 12))
```

- We have 12,165 observations in the train set and 5,214 observations in the test set.

Building Models

- We'll fit different models on the train data and observe their performance.
- We'll try to improve that performance by tuning some hyperparameters available for that algorithm.
- We'll use GridSearchCv for hyperparameter tuning and `r_2 score` to optimize the model.

- R-square - `Coefficient of determination` is used to evaluate the performance of a regression model. It is the amount of the variation in the output dependent attribute which is predictable from the input independent variables.
- Let's start by creating a function to get model scores, so that we don't have to use the same codes repeatedly.

In [39]:
```python
# function to compute adjusted R-squared
def adj_r2_score(predictors, targets, predictions):
    r2 = r2_score(targets, predictions)
    n = predictors.shape[0]
    k = predictors.shape[1]
    return 1 - ((1 - r2) * (n - 1) / (n - k - 1))


# function to compute MAPE
def mape_score(targets, predictions):
    return np.mean(np.abs(targets - predictions) / targets) * 100


# function to compute different metrics to check performance of a regression model
def model_performance_regression(model, predictors, target):
    """
    Function to compute different metrics to check regression model performance

    model: regressor
    predictors: independent variables
    target: dependent variable
    """

    # predicting using the independent variables
    pred = model.predict(predictors)

    r2 = r2_score(target, pred)  # to compute R-squared
    adjr2 = adj_r2_score(predictors, target, pred)  # to compute adjusted R-squared
    rmse = np.sqrt(mean_squared_error(target, pred))  # to compute RMSE
    mae = mean_absolute_error(target, pred)  # to compute MAE
    mape = mape_score(target, pred)  # to compute MAPE

    # creating a dataframe of metrics
    df_perf = pd.DataFrame(
        {
            "RMSE": rmse,
            "MAE": mae,
            "R-squared": r2,
            "Adj. R-squared": adjr2,
            "MAPE": mape,
        },
        index=[0],
    )

    return df_perf
```

In [40]:
```python
##  Function to calculate r2_score and RMSE on train and test data
def get_model_score(model, flag=True):
```

```
    '''
    model : classifier to predict values of X

    '''
    # defining an empty list to store train and test results
    score_list=[]

    pred_train = model.predict(X_train)
    pred_test = model.predict(X_test)

    train_r2=metrics.r2_score(y_train,pred_train)
    test_r2=metrics.r2_score(y_test,pred_test)
    train_rmse=np.sqrt(metrics.mean_squared_error(y_train,pred_train))
    test_rmse=np.sqrt(metrics.mean_squared_error(y_test,pred_test))

    #Adding all scores in the list
    score_list.extend((train_r2,test_r2,train_rmse,test_rmse))

    # If the flag is set to True then only the following print statements will be d
    if flag==True:
        print("R-sqaure on training set : ",metrics.r2_score(y_train,pred_train))
        print("R-square on test set : ",metrics.r2_score(y_test,pred_test))
        print("RMSE on training set : ",np.sqrt(metrics.mean_squared_error(y_train,
        print("RMSE on test set : ",np.sqrt(metrics.mean_squared_error(y_test,pred_

    # returning the list with train and test scores
    return score_list
```

Decision Tree Model

In [41]:
```
dtree=DecisionTreeRegressor(random_state=1)
dtree.fit(X_train,y_train)
```

Out[41]:
```
DecisionTreeRegressor(random_state=1)
```

In [42]:
```
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error
```

In [43]:
```
dtree_model_train_perf=model_performance_regression(dtree, X_train,y_train)
print("Training performance \n",dtree_model_train_perf)
```

```
Training performance
       RMSE       MAE  R-squared  Adj. R-squared      MAPE
0  0.442409  0.005754   0.999994        0.999994  0.004256
```

In [44]:
```
dtree_model_test_perf=model_performance_regression(dtree, X_test,y_test)
print("Testing performance \n",dtree_model_test_perf)
```

```
Testing performance
        RMSE        MAE  R-squared  Adj. R-squared      MAPE
0  60.827833  35.107787   0.892227        0.891979  38.03325
```

- The Decision tree model with default parameters is overfitting the train data.

- Let's see if we can reduce overfitting and improve performance on test data by tuning hyperparameters.

Hyperparameter Tuning

In [45]:
```
# Choose the type of classifier.
dtree_tuned = DecisionTreeRegressor(random_state=1)

# Grid of parameters to choose from
parameters = {'max_depth': list(np.arange(2,20)) + [None],
              'min_samples_leaf': [1, 3, 5, 7, 10],
              'max_leaf_nodes' : [2, 3, 5, 10, 15] + [None],
              'min_impurity_decrease': [0.001, 0.01, 0.1, 0.0]
             }

# Type of scoring used to compare parameter combinations
scorer = metrics.make_scorer(metrics.r2_score)

# Run the grid search
grid_obj = GridSearchCV(dtree_tuned, parameters, scoring=scorer,cv=5)
grid_obj = grid_obj.fit(X_train, y_train)

# Set the clf to the best combination of parameters
dtree_tuned = grid_obj.best_estimator_

# Fit the best algorithm to the data.
dtree_tuned.fit(X_train, y_train)
```

Out[45]:
```
DecisionTreeRegressor(max_depth=14, min_impurity_decrease=0.1,
                      min_samples_leaf=5, random_state=1)
```

In [46]:
```
dtree_tuned_model_train_perf = model_performance_regression(dtree_tuned, X_train,y_
print("Training performance \n",dtree_model_train_perf)
```

```
Training performance 
       RMSE       MAE  R-squared  Adj. R-squared      MAPE
0  0.442409  0.005754  0.999994        0.999994  0.004256
```

In [47]:
```
dtree_tuned_model_test_perf = model_performance_regression(dtree_tuned, X_test,y_te
print("Testing performance \n",dtree_tuned_model_test_perf)
```

```
Testing performance 
        RMSE        MAE  R-squared  Adj. R-squared       MAPE
0  54.969957  31.841364  0.911985        0.911782  38.356397
```

- The overfitting is reduced after hyperparameter tuning and the test score has increased by approx 2%.
- RMSE is also reduced on test data and the model is generalizing better than the decision tree model with default parameters.

Plotting the feature importance of each variable

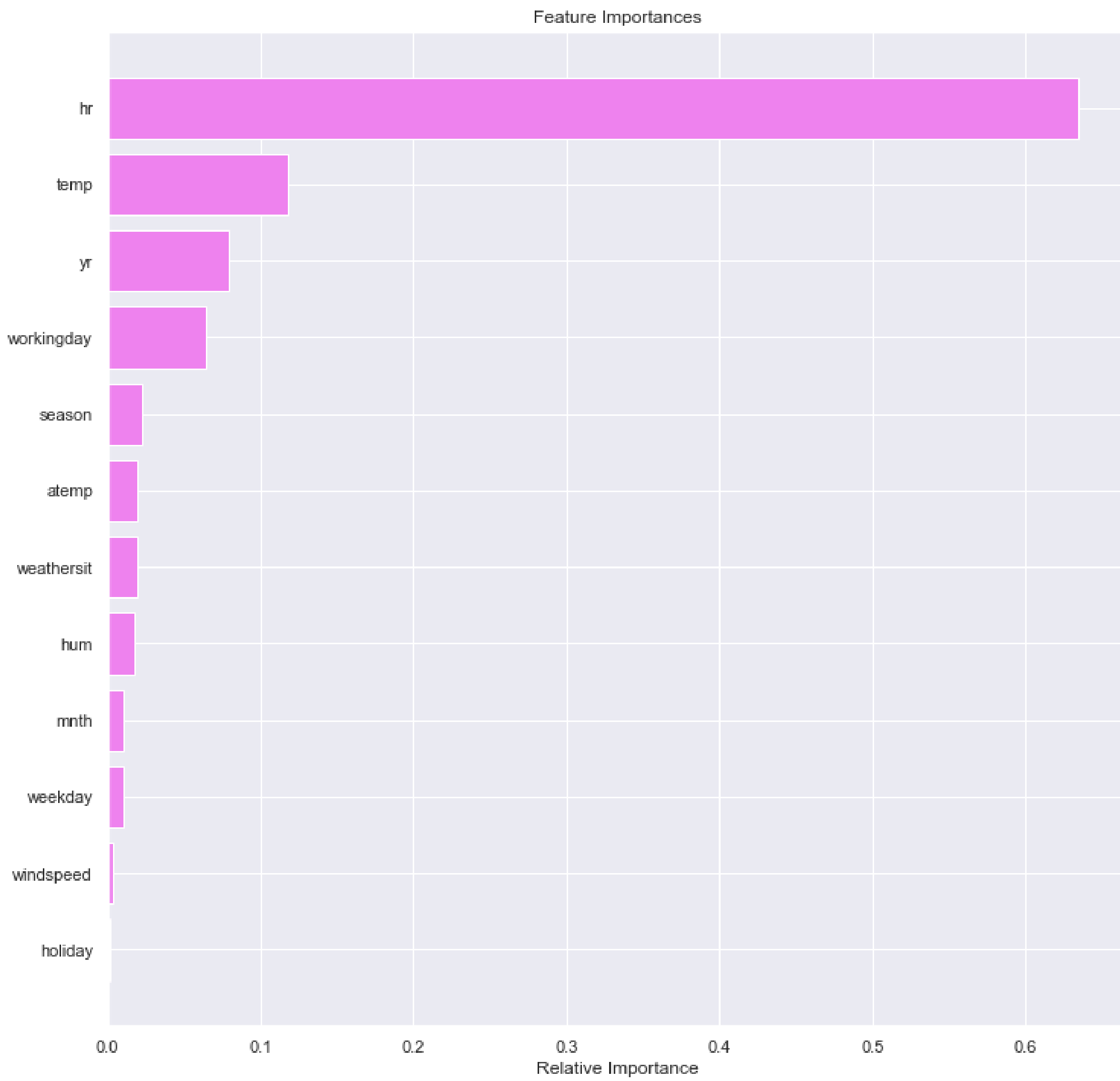
In [48]:
```
# importance of features in the tree building ( The importance of a feature is comp
```

```
#(normalized) total reduction of the criterion brought by that feature. It is also

print(pd.DataFrame(dtree_tuned.feature_importances_, columns = ["Imp"], index = X_t
```

```
                 Imp
hr          0.634942
temp        0.117441
yr          0.079271
workingday  0.063920
season      0.022164
atemp       0.019463
weathersit  0.019460
hum         0.017227
mnth        0.010498
weekday     0.010009
windspeed   0.003882
holiday     0.001724
```

In [49]:

```
feature_names = X_train.columns
importances = dtree_tuned.feature_importances_
indices = np.argsort(importances)

plt.figure(figsize=(12,12))
plt.title('Feature Importances')
plt.barh(range(len(indices)), importances[indices], color='violet', align='center')
plt.yticks(range(len(indices)), [feature_names[i] for i in indices])
plt.xlabel('Relative Importance')
plt.show()
```



- hr is the most important feature, in addition to temp and yr, for tuned decision tree model

Random Forest Model

In [50]:
```
rf_estimator=RandomForestRegressor(random_state=1)
rf_estimator.fit(X_train,y_train)
```

Out[50]: `RandomForestRegressor(random_state=1)`

In [51]:
```
rf_estimator_model_train_perf = model_performance_regression(rf_estimator, X_train,
print("Training performance \n",rf_estimator_model_train_perf)
```

```
Training performance
        RMSE       MAE  R-squared  Adj. R-squared       MAPE
0  16.167421  9.641324   0.991902        0.991894  11.757327
```

In [52]:
```
rf_estimator_model_test_perf = model_performance_regression(rf_estimator, X_test,y_
print("Testing performance \n",rf_estimator_model_test_perf)
```

```
Testing performance
        RMSE        MAE  R-squared  Adj. R-squared       MAPE
0  44.56215  26.192368   0.942159        0.942026  32.748634
```

- Random forest is giving a good r2 score of 94% on the test data but it is slightly overfitting the train data.
- Let's try to reduce this overfitting by hyperparameter tuning.

Hyperparameter Tuning

In [53]:
```python
# Choose the type of classifier.
rf_tuned = RandomForestRegressor(random_state=1)

# Grid of parameters to choose from
parameters = {
                'max_depth':[4, 6, 8, 10, None],
                'max_features': ['sqrt','log2',None],
                'n_estimators': [80, 90, 100, 110, 120]
}

# Type of scoring used to compare parameter combinations
scorer = metrics.make_scorer(metrics.r2_score)

# Run the grid search
grid_obj = GridSearchCV(rf_tuned, parameters, scoring=scorer,cv=5)
grid_obj = grid_obj.fit(X_train, y_train)

# Set the clf to the best combination of parameters
rf_tuned = grid_obj.best_estimator_

# Fit the best algorithm to the data.
rf_tuned.fit(X_train, y_train)
```

Out[53]:
```
RandomForestRegressor(max_features=None, n_estimators=120, random_state=1)
```

In [54]:
```python
rf_tuned_model_train_perf = model_performance_regression(rf_tuned, X_train, y_train
print("Training performance \n",rf_tuned_model_train_perf)
```

```
Training performance
         RMSE       MAE  R-squared  Adj. R-squared      MAPE
0  16.160093  9.622604    0.99191        0.991902  11.73432
```

In [55]:
```python
rf_tuned_model_test_perf = model_performance_regression(rf_tuned, X_test, y_test)
print("Testing performance \n",rf_tuned_model_test_perf)
```

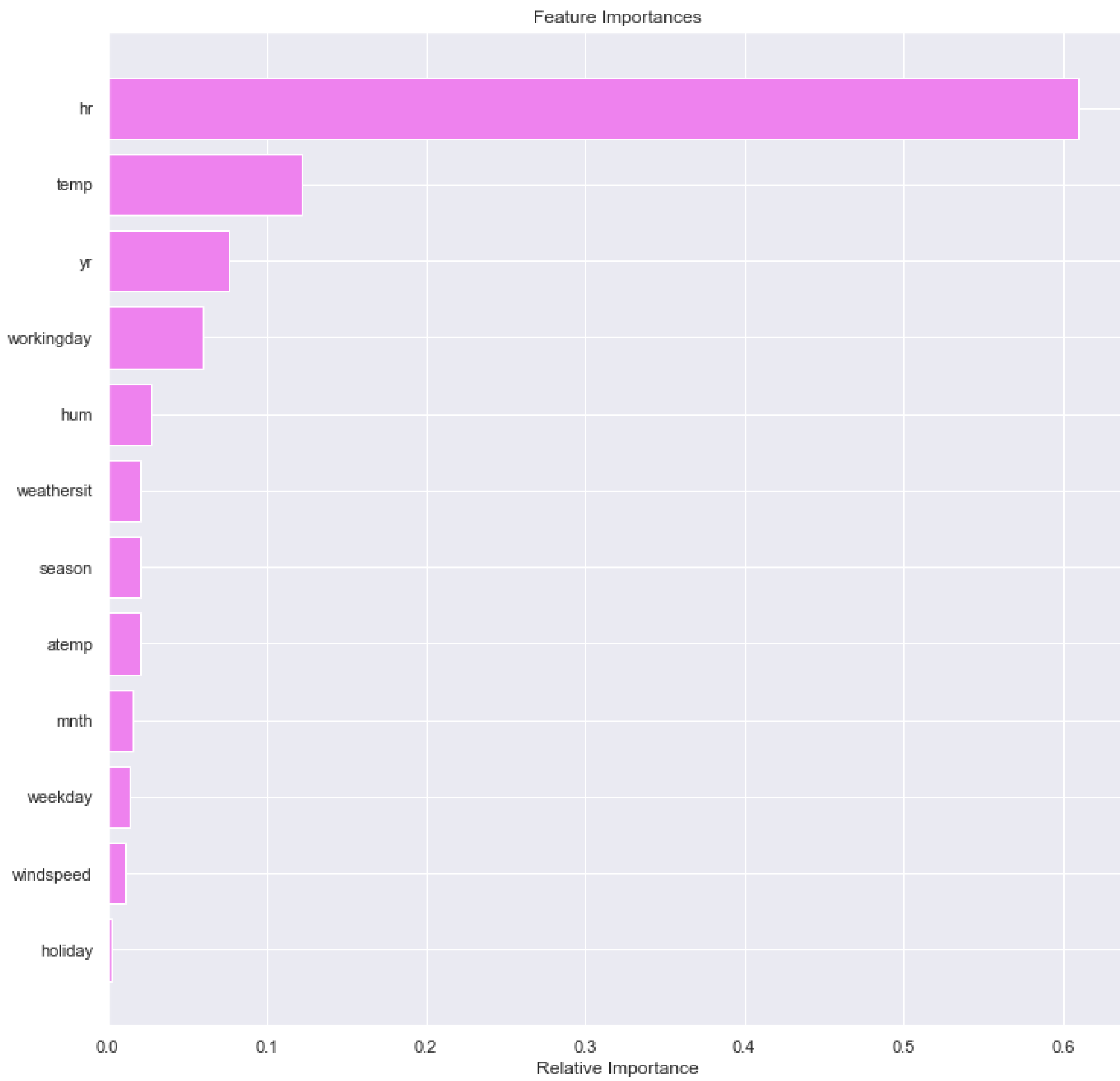
```
Testing performance
       RMSE        MAE  R-squared  Adj. R-squared       MAPE
0  44.5806  26.202363   0.942111        0.941977  32.734172
```

- No significant change in the result. The result is almost the same before or after the hyperparameter tuning.

In [56]:
```
# importance of features in the tree building ( The importance of a feature is comp
#(normalized) total reduction of the criterion brought by that feature. It is also

print(pd.DataFrame(rf_tuned.feature_importances_, columns = ["Imp"], index = X_trai
```

```
                 Imp
hr          0.610116
temp        0.121773
yr          0.076295
workingday  0.059489
hum         0.026844
weathersit  0.020962
season      0.020876
atemp       0.020670
mnth        0.016153
weekday     0.013539
windspeed   0.010510
holiday     0.002774
```

In [57]:
```
feature_names = X_train.columns
importances = rf_tuned.feature_importances_
indices = np.argsort(importances)

plt.figure(figsize=(12,12))
plt.title('Feature Importances')
plt.barh(range(len(indices)), importances[indices], color='violet', align='center')
plt.yticks(range(len(indices)), [feature_names[i] for i in indices])
plt.xlabel('Relative Importance')
plt.show()
```



- hr is the most important feature, in addition to temp and yr, for the tuned random forest model.

Boosting Models

AdaBoost Regressor

In [58]:
```
ab_regressor=AdaBoostRegressor(random_state=1)
ab_regressor.fit(X_train,y_train)
```

Out[58]:
```
AdaBoostRegressor(random_state=1)
```

In [59]:
```
ab_regressor_model_train_perf = model_performance_regression(ab_regressor, X_train,
print("Training performance \n",ab_regressor_model_train_perf)
```

```
Training performance
         RMSE        MAE  R-squared  Adj. R-squared        MAPE
0  104.441843  80.996029   0.662067        0.661733  185.877344
```

In [60]:
```
ab_regressor_model_test_perf = model_performance_regression(ab_regressor, X_test,y_
print("Testing performance \n",ab_regressor_model_test_perf)
```

```
Testing performance
          RMSE        MAE  R-squared  Adj. R-squared        MAPE
0  105.415729  81.468364   0.676321        0.675574  196.403082
```

- AdaBoost is generalizing well but it is giving poor performance, in terms of r2 score as well as RMSE, as compared to the decision tree and random forest model.

Hyperparameter Tuning

In [61]:
```
# Choose the type of classifier.
ab_tuned = AdaBoostRegressor(random_state=1)

# Grid of parameters to choose from
parameters = {'n_estimators': np.arange(10,100,10),
              'learning_rate': [1, 0.1, 0.5, 0.01],
              }

# Type of scoring used to compare parameter combinations
scorer = metrics.make_scorer(metrics.r2_score)

# Run the grid search
grid_obj = GridSearchCV(ab_tuned, parameters, scoring=scorer,cv=5)
grid_obj = grid_obj.fit(X_train, y_train)

# Set the clf to the best combination of parameters
ab_tuned = grid_obj.best_estimator_

# Fit the best algorithm to the data.
ab_tuned.fit(X_train, y_train)
```

Out[61]:
```
AdaBoostRegressor(learning_rate=1, n_estimators=30, random_state=1)
```

In [62]:
```
ab_tuned_model_train_perf = model_performance_regression(ab_tuned, X_train,y_train)
print("Training performance \n",ab_tuned_model_train_perf)
```

```
Training performance
          RMSE        MAE  R-squared  Adj. R-squared        MAPE
0  103.326379  80.468809   0.669247         0.66892  195.189129
```

In [88]:
```
ab_tuned_model_test_perf = model_performance_regression(ab_tuned, X_test,y_test)
print("Testing performance \n",ab_tuned_model_train_perf)
```

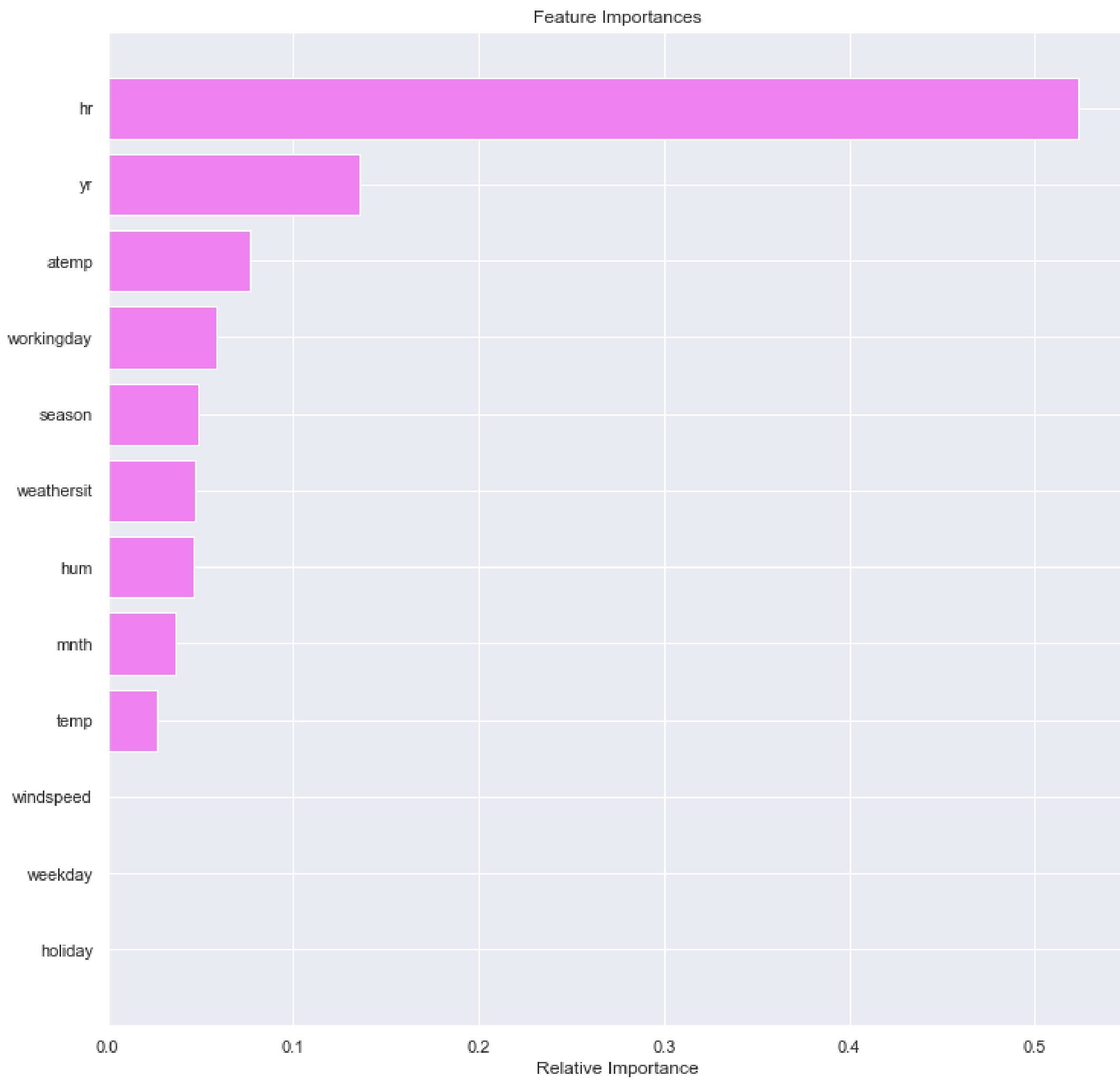
```
Testing performance
          RMSE        MAE  R-squared  Adj. R-squared        MAPE
0  104.430458  81.015165   0.682343         0.68161  207.467888
```

- We can see that there is no significant improvement in the model after hyperparameter tuning.

In [64]:
```
# importance of features in the tree building

print(pd.DataFrame(ab_tuned.feature_importances_, columns = ["Imp"], index = X_trai
```

```
                 Imp
hr          0.523674
yr          0.135837
atemp       0.076841
workingday  0.058274
season      0.049087
weathersit  0.047040
hum         0.046359
mnth        0.036370
temp        0.026518
holiday     0.000000
weekday     0.000000
windspeed   0.000000
```

In [65]:
```
feature_names = X_train.columns
importances = ab_tuned.feature_importances_
indices = np.argsort(importances)

plt.figure(figsize=(12,12))
plt.title('Feature Importances')
plt.barh(range(len(indices)), importances[indices], color='violet', align='center')
plt.yticks(range(len(indices)), [feature_names[i] for i in indices])
plt.xlabel('Relative Importance')
plt.show()
```



- hr is the most important feature here, followed by yr and atemp.

Gradient Boosting Regressor

In [66]:
```
gb_estimator=GradientBoostingRegressor(random_state=1)
gb_estimator.fit(X_train,y_train)
```

Out[66]:
```
GradientBoostingRegressor(random_state=1)
```

In [67]:
```
gb_estimator_model_train_perf = model_performance_regression(gb_estimator, X_train,
print("Training performance \n",gb_estimator_model_train_perf)
```

```
Training performance
        RMSE        MAE  R-squared  Adj. R-squared       MAPE
0  71.87459  48.518039   0.839959        0.839801  92.152544
```

In [68]:
```
gb_estimator_model_test_perf = model_performance_regression(gb_estimator, X_test, y
print("Testing performance \n",gb_estimator_model_test_perf)
```

```
Testing performance
         RMSE        MAE  R-squared  Adj. R-squared        MAPE
0  74.173279  49.691124    0.83975         0.83938  100.149788
```

- Gradient boosting is generalizing well and giving decent results but not as good as random forest.

Hyperparameter Tuning

In [69]:
```
# Choose the type of classifier.
gb_tuned = GradientBoostingRegressor(random_state=1)

# Grid of parameters to choose from
parameters = {'n_estimators': np.arange(50,200,25),
              'subsample':[0.7,0.8,0.9,1],
              'max_features':[0.7,0.8,0.9,1],
              'max_depth':[3,5,7,10]
              }

# Type of scoring used to compare parameter combinations
scorer = metrics.make_scorer(metrics.r2_score)

# Run the grid search
grid_obj = GridSearchCV(gb_tuned, parameters, scoring=scorer,cv=5)
grid_obj = grid_obj.fit(X_train, y_train)

# Set the clf to the best combination of parameters
gb_tuned = grid_obj.best_estimator_

# Fit the best algorithm to the data.
gb_tuned.fit(X_train, y_train)
```

Out[69]:
```
GradientBoostingRegressor(max_depth=7, max_features=0.9, n_estimators=175,
                          random_state=1, subsample=0.7)
```

In [70]:
```
gb_tuned_model_train_perf = model_performance_regression(gb_tuned, X_train,y_train)
print("Training performance \n",gb_tuned_model_train_perf)
```

```
Training performance
         RMSE        MAE  R-squared  Adj. R-squared       MAPE
0  22.272348  14.726501   0.984632        0.984617  23.168969
```

In [71]:
```
gb_tuned_model_test_perf = model_performance_regression(gb_tuned, X_test, y_test)
print("Testing performance \n",gb_tuned_model_test_perf)
```

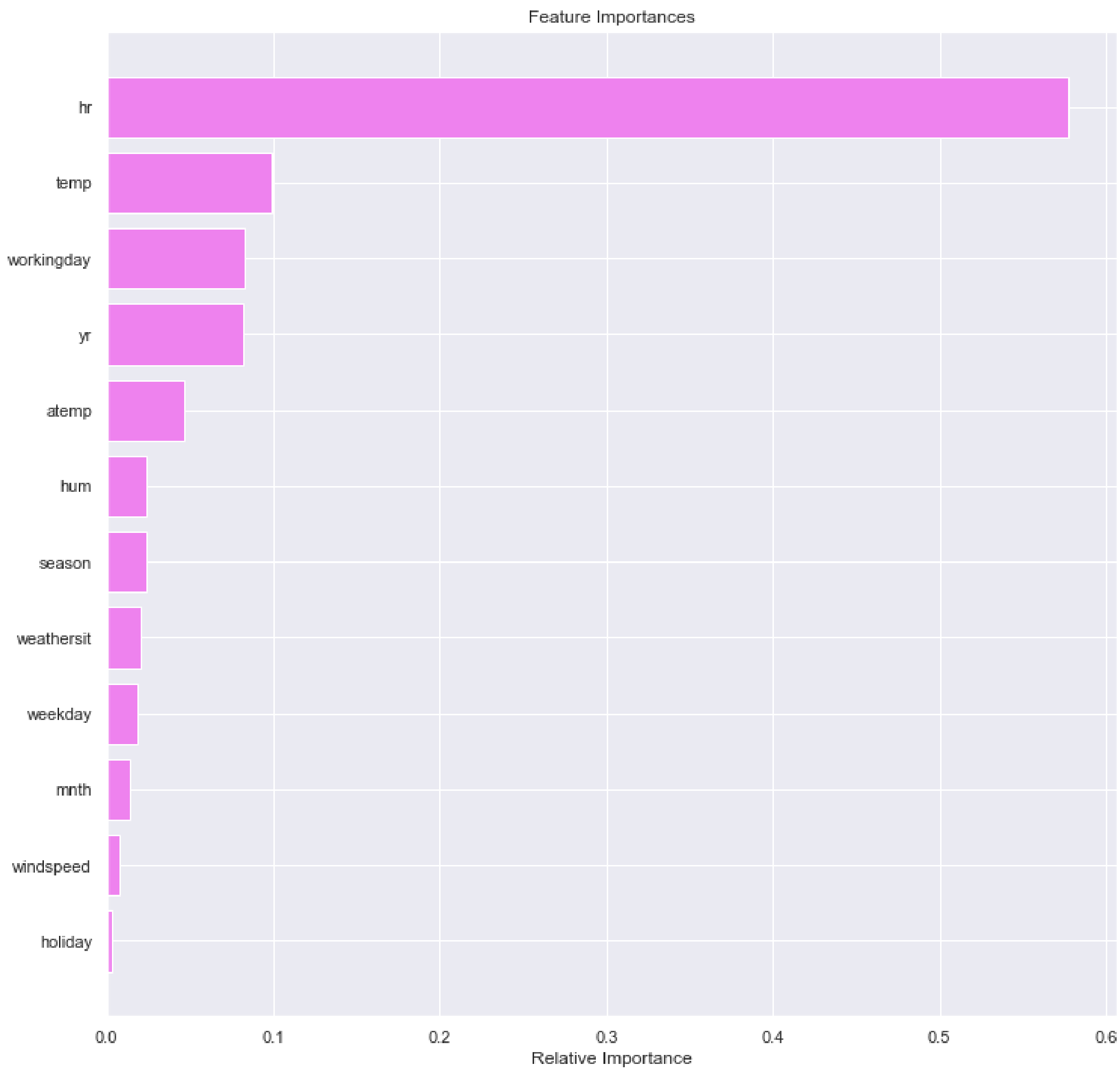
```
Testing performance
         RMSE        MAE  R-squared  Adj. R-squared      MAPE
0  39.076262  24.101884   0.955524        0.955421  36.01227
```

- We can see that the model has improved significantly in terms of r2 score and RMSE.
- The r2 score has increased by approx 12% on the test data.
- RMSE has decreased by more than 30 for the test data.

In [72]:
```
# importance of features in the tree building ( The importance of a feature is comp
#(normalized) total reduction of the criterion brought by that feature. It is also 

print(pd.DataFrame(gb_tuned.feature_importances_, columns = ["Imp"], index = X_trai
```

```
                 Imp
hr          0.577712
temp        0.099255
workingday  0.083006
yr          0.081833
atemp       0.046505
hum         0.024214
season      0.024071
weathersit  0.020767
weekday     0.018315
mnth        0.013716
windspeed   0.007647
holiday     0.002959
```

In [73]:
```
feature_names = X_train.columns
importances = gb_tuned.feature_importances_
indices = np.argsort(importances)

plt.figure(figsize=(12,12))
plt.title('Feature Importances')
plt.barh(range(len(indices)), importances[indices], color='violet', align='center')
plt.yticks(range(len(indices)), [feature_names[i] for i in indices])
plt.xlabel('Relative Importance')
plt.show()
```



- hr is the most important feature
- temp, yr and workingday have almost equal importance

XGBoost Regressor

In [74]:
```
xgb_estimator=XGBRegressor(random_state=1)
xgb_estimator.fit(X_train,y_train)
```

Out[74]:
```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
             colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=-1,
             importance_type='gain', interaction_constraints='',
             learning_rate=0.300000012, max_delta_step=0, max_depth=6,
             min_child_weight=1, missing=nan, monotone_constraints='()',
             n_estimators=100, n_jobs=8, num_parallel_tree=1, random_state=1,
             reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=1,
             tree_method='exact', validate_parameters=1, verbosity=None)
```

In [75]:
```
xgb_estimator_model_train_perf = model_performance_regression(xgb_estimator, X_trai
print("Training performance \n",xgb_estimator_model_train_perf)
```

```
Training performance
         RMSE        MAE  R-squared  Adj. R-squared       MAPE
0  27.214945  17.680308   0.977055        0.977032  29.490325
```

In [76]:
```
xgb_estimator_model_test_perf = model_performance_regression(xgb_estimator, X_test,
print("Testing performance \n",xgb_estimator_model_test_perf)
```

```
Testing performance
         RMSE        MAE  R-squared  Adj. R-squared      MAPE
0  41.356427  25.804078   0.950182        0.950067  41.83796
```

- XGBoost with default parameters is giving almost as good results as the tuned gradient boosting model.

Hyperparameter Tuning

In [77]:
```
# Choose the type of classifier.
xgb_tuned = XGBRegressor(random_state=1)

# Grid of parameters to choose from
parameters = {'n_estimators': [75,100,125,150],
              'subsample':[0.7, 0.8, 0.9, 1],
              'gamma':[0, 1, 3, 5],
              'colsample_bytree':[0.7, 0.8, 0.9, 1],
              'colsample_bylevel':[0.7, 0.8, 0.9, 1]
              }

# Type of scoring used to compare parameter combinations
scorer = metrics.make_scorer(metrics.r2_score)

# Run the grid search
grid_obj = GridSearchCV(xgb_tuned, parameters, scoring=scorer,cv=5)
grid_obj = grid_obj.fit(X_train, y_train)

# Set the clf to the best combination of parameters
xgb_tuned = grid_obj.best_estimator_

# Fit the best algorithm to the data.
xgb_tuned.fit(X_train, y_train)
```

Out[77]:
```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=0.8,
             colsample_bynode=1, colsample_bytree=1, gamma=1, gpu_id=-1,
             importance_type='gain', interaction_constraints='',
             learning_rate=0.300000012, max_delta_step=0, max_depth=6,
             min_child_weight=1, missing=nan, monotone_constraints='()',
             n_estimators=150, n_jobs=8, num_parallel_tree=1, random_state=1,
             reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=1,
             tree_method='exact', validate_parameters=1, verbosity=None)
```

In [78]:
```
xgb_tuned_model_train_perf = model_performance_regression(xgb_tuned, X_train, y_tra
print("Training performance \n",xgb_tuned_model_train_perf)
```

```
Training performance
        RMSE        MAE  R-squared  Adj. R-squared       MAPE
0  24.198253  15.806928   0.981859        0.981842  28.513527
```

In [79]:

```
xgb_tuned_model_test_perf = model_performance_regression(xgb_tuned, X_test, y_test)
print("Testing performance \n",xgb_tuned_model_test_perf)
```

```
Testing performance
        RMSE        MAE  R-squared  Adj. R-squared       MAPE
0  40.738742  25.837159   0.951659        0.951547  43.831514
```

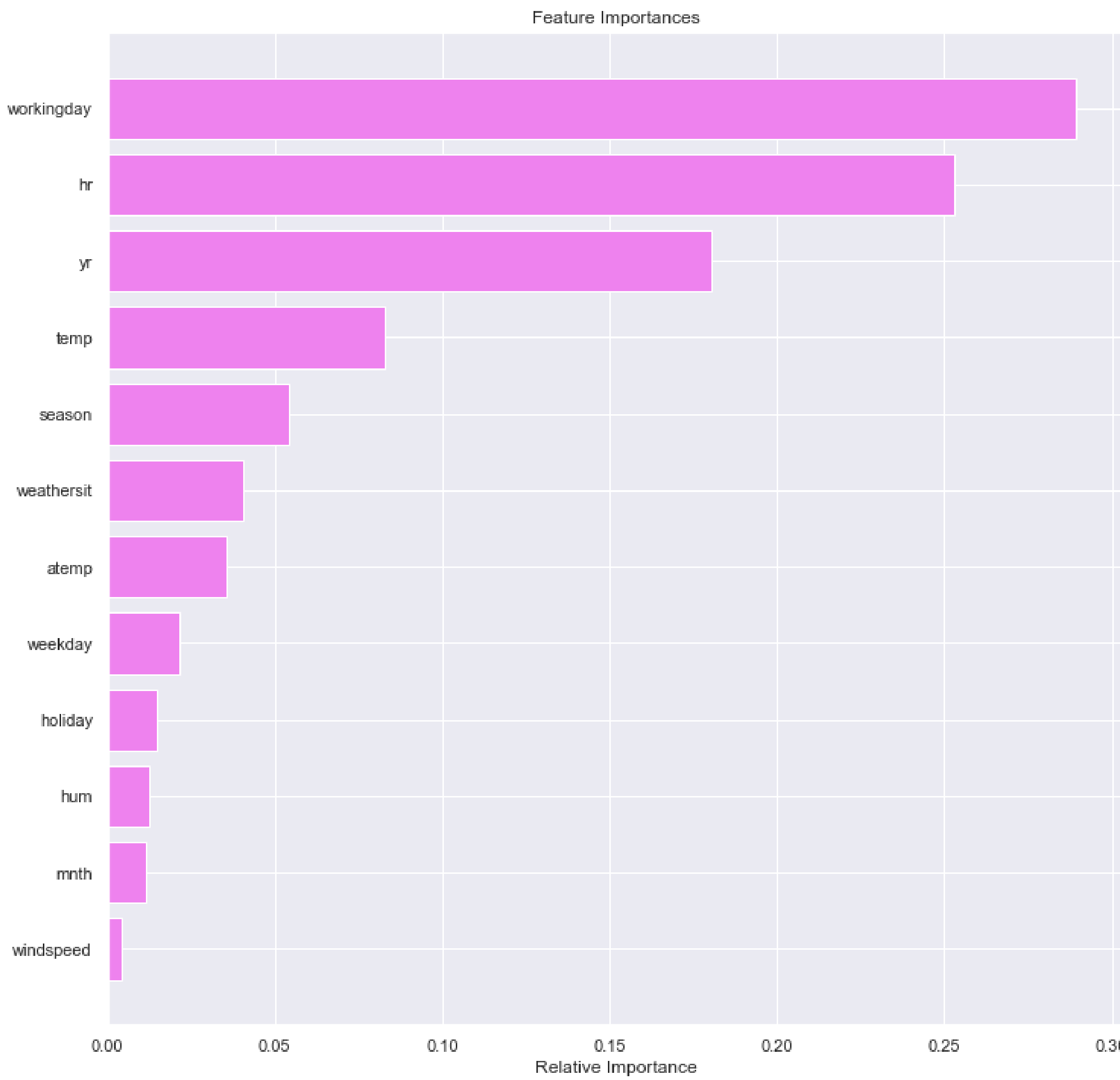
In [80]:

```
# importance of features in the tree building ( The importance of a feature is comp
#(normalized) total reduction of the criterion brought by that feature. It is also

print(pd.DataFrame(xgb_tuned.feature_importances_, columns = ["Imp"], index = X_tra
```

```
                 Imp
workingday  0.289411
hr          0.252782
yr          0.180462
temp        0.082915
season      0.054046
weathersit  0.040526
atemp       0.035576
weekday     0.021362
holiday     0.014707
hum         0.012433
mnth        0.011668
windspeed   0.004111
```

In [81]:

```
feature_names = X_train.columns
importances = xgb_tuned.feature_importances_
indices = np.argsort(importances)

plt.figure(figsize=(12,12))
plt.title('Feature Importances')
plt.barh(range(len(indices)), importances[indices], color='violet', align='center')
plt.yticks(range(len(indices)), [feature_names[i] for i in indices])
plt.xlabel('Relative Importance')
plt.show()
```



- In XGBoost, workingday is the most important feature followed by features - hr and yr

Stacking Model

Now, let's build a stacking model with the tuned models - decision tree, random forest, and gradient boosting, then use XGBoost to get the final prediction.

In [82]:
```
estimators=[('Decision Tree', dtree_tuned),('Random Forest', rf_tuned),
           ('Gradient Boosting', gb_tuned)]
final_estimator=XGBRegressor(random_state=1)
```

In [83]:
```
stacking_estimator=StackingRegressor(estimators=estimators, final_estimator=final_e
stacking_estimator.fit(X_train,y_train)
```

```
Out[83]: StackingRegressor(cv=5,
                  estimators=[('Decision Tree',
                               DecisionTreeRegressor(max_depth=14,
                                                     min_impurity_decrease=0.1,
                                                     min_samples_leaf=5,
                                                     random_state=1)),
                              ('Random Forest',
                               RandomForestRegressor(max_features=None,
                                                     n_estimators=120,
                                                     random_state=1)),
                              ('Gradient Boosting',
                               GradientBoostingRegressor(max_depth=7,
                                                         max_features=0.9,
                                                         n_estimators=175,
                                                         random_state=1,
                                                         subsa...
                                            importance_type='gain',
                                            interaction_constraints=None,
                                            learning_rate=None,
                                            max_delta_step=None,
                                            max_depth=None,
                                            min_child_weight=None,
                                            missing=nan,
                                            monotone_constraints=None,
                                            n_estimators=100, n_jobs=None,
                                            num_parallel_tree=None,
                                            random_state=1, reg_alpha=None,
                                            reg_lambda=None,
                                            scale_pos_weight=None,
                                            subsample=None, tree_method=None,
                                            validate_parameters=None,
                                            verbosity=None))
```

```
In [84]: stacking_estimator_model_train_perf = model_performance_regression(stacking_estimat
         print("Training performance \n",stacking_estimator_model_train_perf)
```

```
Training performance
         RMSE        MAE  R-squared  Adj. R-squared      MAPE
0  23.272893  14.272256    0.98322        0.983204  15.89583
```

```
In [85]: stacking_estimator_model_test_perf = model_performance_regression(stacking_estimato
         print("Testing performance \n",stacking_estimator_model_test_perf)
```

```
Testing performance
         RMSE        MAE  R-squared  Adj. R-squared       MAPE
0  41.413999  24.715628   0.950043        0.949928  29.427916
```

Comparing all models

```
In [86]: # training performance comparison

         models_train_comp_df = pd.concat(
             [dtree_model_train_perf.T, dtree_tuned_model_train_perf.T, rf_estimator_model_t
             ab_regressor_model_train_perf.T,ab_tuned_model_train_perf.T,gb_estimator_model_
             xgb_estimator_model_train_perf.T,xgb_tuned_model_train_perf.T,stacking_estimato
```

```
    axis=1,
)

models_train_comp_df.columns = [
    "Decision Tree",
    "Decision Tree Tuned",
    "Random Forest Estimator",
    "Random Forest Tuned",
    "Adaboost Regressor",
    "Adaboost Tuned",
    "Gradient Boost Estimator",
    "Gradient Boost Tuned",
    "XGB",
    "XGB Tuned",
    "Stacking Classifier"
]

print("Training performance comparison:")
models_train_comp_df
```

```
Training performance comparison:
```

Out[86]:

	Decision Tree	Decision Tree Tuned	Random Forest Estimator	Random Forest Tuned	Adaboost Regressor	Adaboost Tuned	Gradient Boost Estimator	Gradient Boost Tuned
RMSE	0.442409	36.442792	16.167421	16.160093	104.441843	104.430458	71.874590	22.272348
MAE	0.005754	22.525337	9.641324	9.622604	80.996029	81.015165	48.518039	14.726501
R-squared	0.999994	0.958856	0.991902	0.991910	0.662067	0.682343	0.839959	0.984632
Adj. R-squared	0.999994	0.958816	0.991894	0.991902	0.661733	0.681610	0.839801	0.984617
MAPE	0.004256	28.657314	11.757327	11.734320	185.877344	207.467888	92.152544	23.168969

In [91]:

```
# Testing performance comparison

models_test_comp_df = pd.concat(
    [dtree_model_test_perf.T, dtree_tuned_model_test_perf.T, rf_estimator_model_tes
    ab_regressor_model_test_perf.T,ab_tuned_model_test_perf.T,gb_estimator_model_te
    xgb_estimator_model_test_perf.T,xgb_tuned_model_test_perf.T,stacking_estimator_
    axis=1,
)

models_test_comp_df.columns = [
    "Decision Tree",
    "Decision Tree Tuned",
    "Random Forest Estimator",
    "Random Forest Tuned",
    "Adaboost Regressor",
    "Adaboost Tuned",
    "Gradient Boost Estimator",
    "Gradient Boost Tuned",
    "XGB",
    "XGB Tuned",
```

```
    "Stacking Classifier"
]

print("Testing performance comparison:")
models_test_comp_df
```

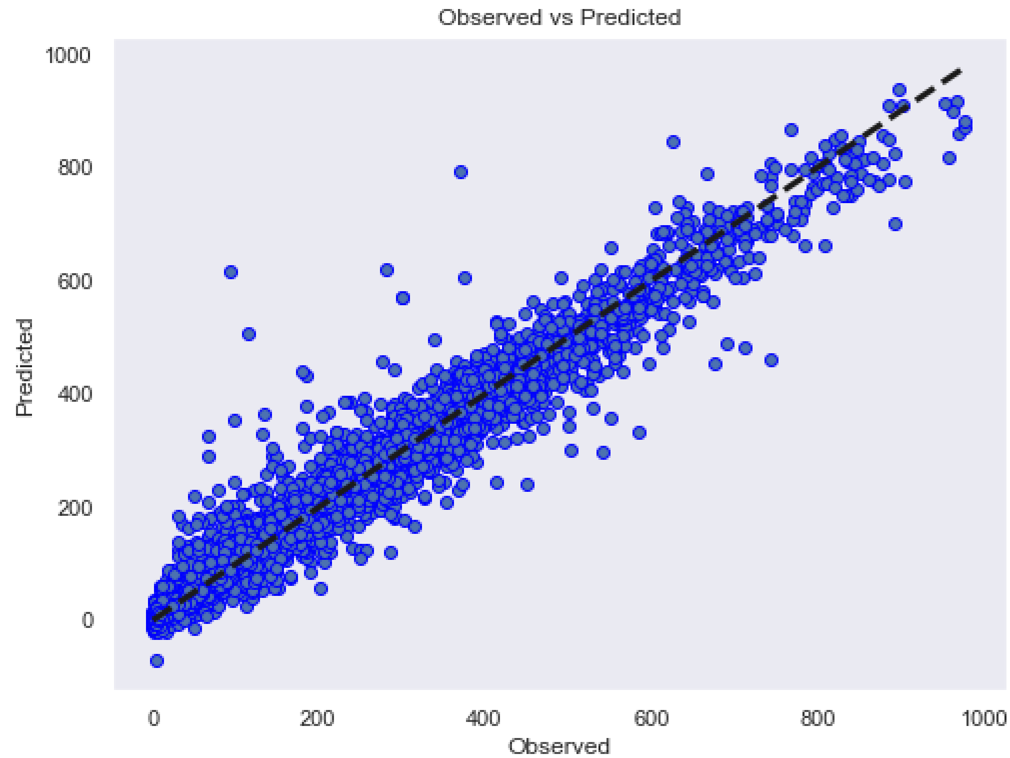
Testing performance comparison:

Out[91]:

	Decision Tree	Decision Tree Tuned	Random Forest Estimator	Random Forest Tuned	Adaboost Regressor	Adaboost Tuned	Gradient Boost Estimator	Gradier Boos Tune
RMSE	60.827833	54.969957	44.562150	44.580600	105.415729	104.430458	74.173279	39.07626
MAE	35.107787	31.841364	26.192368	26.202363	81.468364	81.015165	49.691124	24.10188
R-squared	0.892227	0.911985	0.942159	0.942111	0.676321	0.682343	0.839750	0.95552
Adj. R-squared	0.891979	0.911782	0.942026	0.941977	0.675574	0.681610	0.839380	0.95542
MAPE	38.033250	38.356397	32.748634	32.734172	196.403082	207.467888	100.149788	36.01227

- The tuned gradient boosting model is the best model here. It has the highest r2 score of approx 95.5% and the lowest RMSE of approx 39 on the test data.
- Gradient boosting, XGBoost, and stacking regressor are the top 3 models. They are all giving a similar performance.

In [92]:
```
# So plot observed and predicted values of the test data for the best model i.e. tu
fig, ax = plt.subplots(figsize=(8, 6))
y_pred=gb_tuned.predict(X_test)
ax.scatter(y_test, y_pred, edgecolors=(0, 0, 1))
ax.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'k--', lw=3)
ax.set_xlabel('Observed')
ax.set_ylabel('Predicted')
ax.set_title("Observed vs Predicted")
plt.grid()
plt.show()
```



- We can see that points are dense on the line where predicted is equal to the observed.
- This implies that most of the predicted values are close to the true values with some exceptions as seen in the plot.

Business Recommendations

- We can use this predictive model for any season and environmental parameters (which we know in advance) and can predict the count of the bikes to be rented. The ability to predict the number of hourly users can allow the entities (businesses/governments) that oversee these systems to manage them more efficiently and cost-effectively.
- More bikes can be made available for the fall and winter seasons as the number of bikes rented is high in these seasons.
- As the number of bikes rented is high for day timings compared to night timings, similarly, fall and winter seasons have more surges compared to other seasons. We can choose differential prices of bikes accordingly.
- As most of the rentals are for commuting to workplaces and colleges daily, company can launch more stations near busy workplaces or schools/colleges to reach out to their main customers.
- Number of bikes rented is heavily dependent on the weather. So, we should adjust the number of available bikes in an area based on the weather forecast.
- Maintenance activities for bikes can be done at night due to low usage of bikes during the nighttime.

- Company can provide offers or coupons like a monthly subscription to compensate for the low count on holidays or weekends.

```
In [ ]:
```