Chapter 1

Bike sharing analysis

Overview

In this chapter, we analyze data from bike sharing services, and identify usage patterns, depending on time features and weather conditions. Furthermore, topics like visual analysis, hypothesis testing and time series analysis are introduced, and their applications to the problem are presented. By the end of the chapter, the reader should be comfortable in working with time series data, as well as applying some of the main techniques to such type of problems.

Introduction

Nowadays, bike sharing is a fundamental service in the urban mobility sector. In fact, it is easily accessible (as no driving license is required), cheaper than normal car sharing services (since bike maintenance and insurance are substantially cheaper than automobile ones), and finally, it is often faster way to commute within the city. Therefore, understanding the driving factors of bike sharing requests is essential for both companies and users. From company’s perspective, identifying the expected bike demand in a specific area, within a specific time frame can significantly increase revenue and customers’ satisfaction. Moreover, bike relocation can be optimized, which would further reduce operational costs. From users’ perspective, probably the most important factor is bike availability, which we can easily see aligns with companies’ interests.

In this chapter, we will analyze bike sharing data, available for the period between 1st of January 2011 and 31st of December 2012. The data is aggregated on an hourly basis, hence no initial and final locations of the single rides are available, but only the total number of rides per hour. Nevertheless, additional meteorological information is available in the data, which could serve us as driving factor for identifying total request for a specific time frame (bad weather conditions could have a substantial impact on bike sharing demand).

Note the although the conducted analysis is related to bike sharing, the provided techniques could be easily transferred to other time of sharing business models, such as car or scooter sharing.

Initial data analysis

In this first part, we load the data and perform some initial exploration on it. The main goal of the presented steps is to acquire some basic knowledge about the data, how the various features are distributed, if there are missing values in it and so on.

Let us first import the relevant for the analysis python libraries and the data itself

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

%matplotlib inline

hourly\_data = pd.read\_csv('data/hour.csv')

A good practice is the check the size of the data we are loading, as well as the number of missing values of each column, and some general statistics about the numerical columns

print(f"Shape of data: {hourly\_data.shape}")

print(f"Number of missing values in the data:{hourly\_data.isnull().sum().sum()}")

which returns the following output:

Shape of data: (17379, 17)

Number of missing values in the data: 0

In order to get simple statistics on the various columns, in a pandas.Dataset, we can use the describe method:

hourly\_data.describe().T

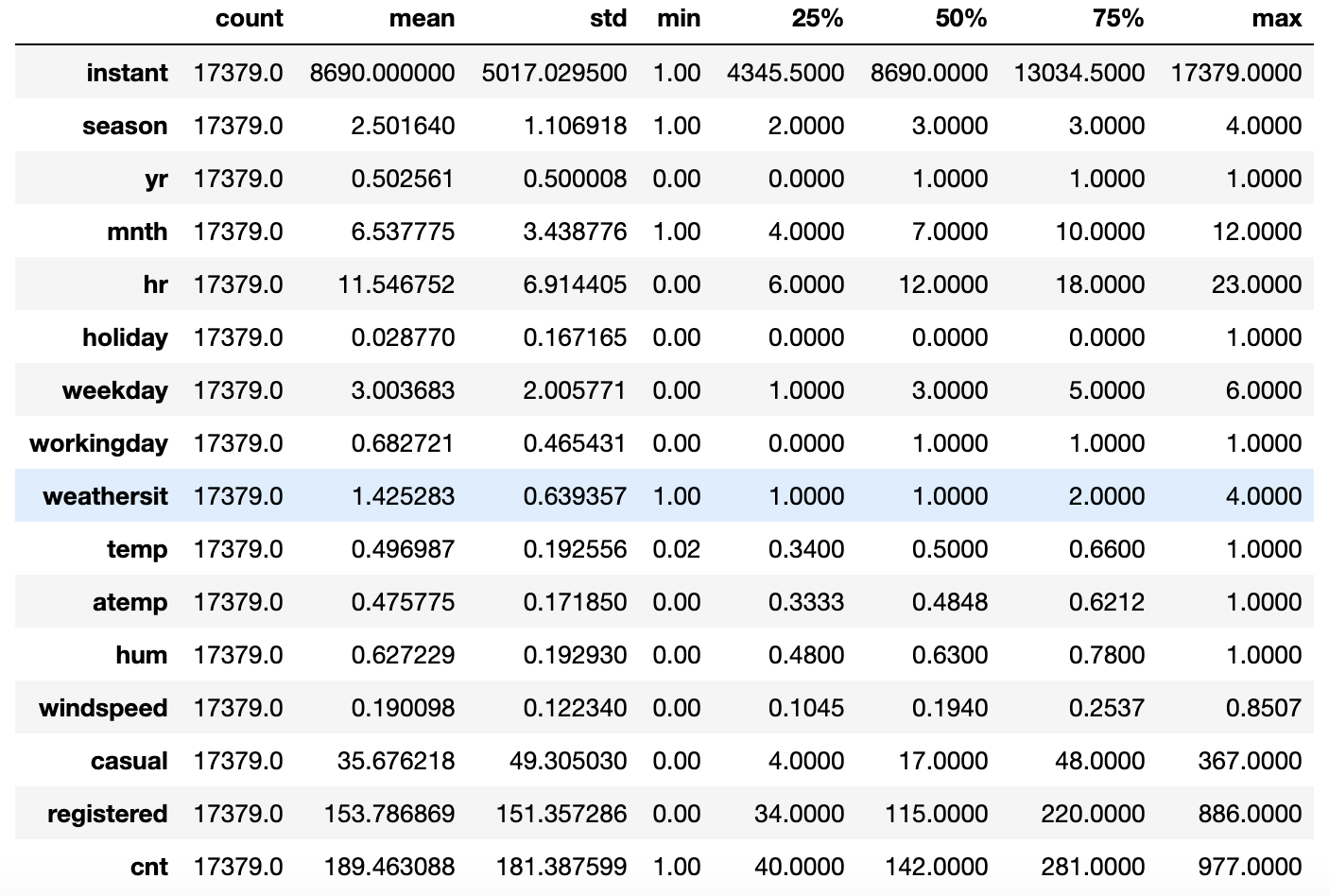


Figure 1.1: Output of the describe() method

Note that the T after the describe() method gets the transpose of the resulting dataset, hence the columns become rows and vice versa.

According to the description of the original data, provided in the Readme.txt file, we can split the columns into three main groups:

* **temporal features**: containing information about the time at which record was registered. This group contains the columns **dteday**, **season**, **yr**, **mnth**, **hr**, **holiday**, **weekday** and **workingday**.
* **weather related features**: columns containing information about the weather conditions. Columns **weathersit**, **temp**, **atemp**, **hum** and **windspeed** are included in this group.
* **record related features**: columns containing information about the amount of records, for the specific hour and date. In this group are included **casual**, **registered** and **cnt** columns.

Note that we did not include the first column **instant** in any of the previously mentioned groups. The reason for that is that it is and index column, and will be excluded from our analysis, as it does not contain any relevant for our analysis information.

Data preprocessing

In this part, we perform some preprocessing steps, which will allow us to transform the data into more human readable format. Note that data preprocessing and wrangling is one of the most important parts in data analysis. In facts, a lot of hidden patterns and relationship might arise, when data is transformed in the correct way. Furthermore, some machine learning algorithms might not even converge, or provide an erroneous result, when fed with badly preprocessed data. In other cases, deriving insights from normalized data might be difficult for a human to understand the results, hence it is good practice to transform the data before presenting the results.

In this use case, the data is already normalized and ready for analysis, nevertheless, some of its columns are hard to interpret from human perspective. Before proceeding further, we will perform some basic transformations on the columns, which will allow us to have more understandable analysis at later stage.

Exercise 1.01: Transforming temporal and weather features

In the first part of the exercise, we are going to encode the temporal features into a more human readable format. The **seasons** column contains values from 1 to 4, which encode respectively Winter, Spring, Summer and Fall seasons, the **yr** column, originally containing the values 0 and 1 represents 2011 and 2012, while the **weekday** column contains values from 0 to 6, each one representing a day of the week (0: Sunday, 1: Monday, … , 6: Saturday).

1. As a first step, we create a copy of the original dataset:

preprocessed\_data = hourly\_data.copy()

1. In the next step, we map the **season** variable from numerical to nicely encoded categorical one. In order to do that, we create a python dictionary, which contains the encoding, then exploit the apply and lambda functions:

seasons\_mapping = {1: 'winter', 2: 'spring', 3: 'summer', 4: 'fall'}

preprocessed\_data['season'] = preprocessed\_data['season'].apply(lambda x: seasons\_mapping[x])

1. Same logic applies for the **yr** column:

yr\_mapping = {0: 2011, 1: 2012}

preprocessed\_data['yr'] = preprocessed\_data['yr'].apply(lambda x: yr\_mapping[x])

1. And finally, for the **weekday** column:

weekday\_mapping = {0: 'Sunday', 1: 'Monday', 2: 'Tuesday', 3: 'Wednesday', 4: 'Thursday', 5: 'Friday', 6: 'Saturday'}

preprocessed\_data['weekday'] = preprocessed\_data['weekday'].apply(lambda x: weekday\_mapping[x])

Let us now proceed with encoding the weather related columns (**weathersit**, **hum** and **windspeed**). According to the provided information with the data, the **weathersit** column represents the current weather conditions, where 1 stands for clear weather with few clouds, 2 represents cloudy weather, 3 relates to light snow or rain, and 4 stands for heavy snow or rain. The **hum** column stands for the current normalized air humidity, with values from 0 to 1 (hence we will multiply the values of this column by 100, in order to obtain percentages). Finally, the **windspeed** column represents the windspeed, again normalized from values between 0 and 67 m/s.

1. As next step, we encode the **weathersit** values:

weather\_mapping = {1: 'clear', 2: 'cloudy', 3: 'light\_rain\_snow', 4: 'heavy\_rain\_snow'}

preprocessed\_data['weathersit'] = preprocessed\_data['weathersit'].apply(lambda x: weather\_mapping[x])

1. Finally, we rescale the **hum** and **windspeed** columns

preprocessed\_data['hum'] = preprocessed\_data['hum']\*100

preprocessed\_data['windspeed'] = preprocessed\_data['windspeed']\*67

1. We can visualize the results from our transformation, by calling the sample() method on the newly created dataset:

cols = ['season', 'yr', 'weekday', 'weathersit', 'hum', 'windspeed']

preprocessed\_data[cols].sample(10, random\_state=123)

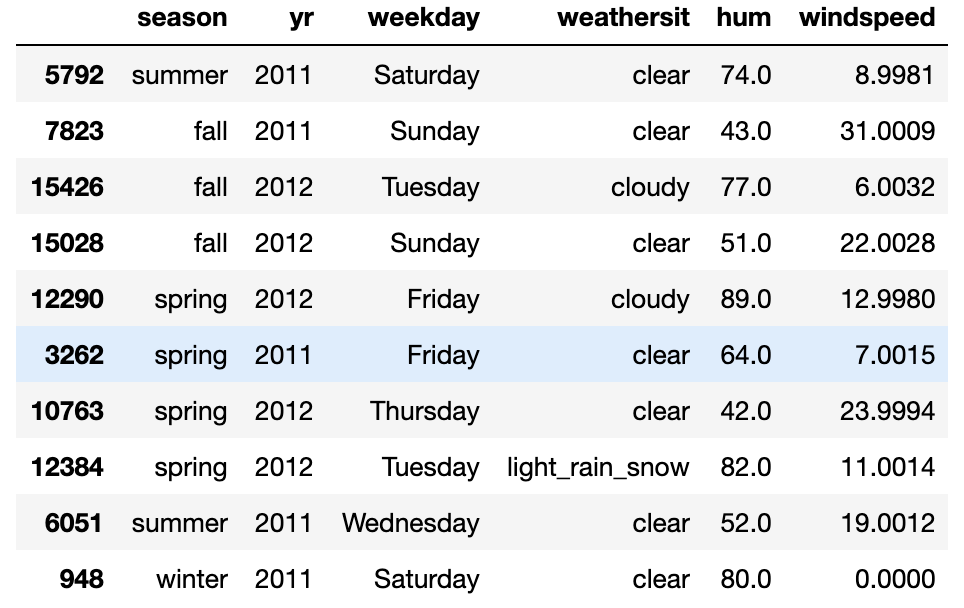


Figure 1.2: Result from variables transformation

Note that in the Exercise, we did not transform the **temp** and **atemp** columns (the true and perceived temperatures, respectively). The reason for that is that they assume only positive values in the original dataset (hence we do not know when a negative temperatures occurred), furthermore, as their scales are different (the maximum value, registered in the true temperature is 41 degrees, while for the perceived one is 67), we do not want to modify their relations (hours, at which true temperature is greater than perceived one, and viceversa).

**Registered vs casual use analysis**

We begin our analysis of the single features, by focusing on the two main one: the number rides by registered users, versus the number of rides by casual ones. These numbers are represented in the **registered** and **casual** columns respectively, with **cnt** column representing the sum of the registered and casual rides. We can easily verify the last statement for each entry in the dataset, by using the assert statement:

assert (preprocessed\_data.casual + preprocessed\_data.registered == preprocessed\_data.cnt).all(), \

'Sum of casual and registered rides not equal to total number of rides'

First step in analyzing the two columns is to look at their distributions. This can be done easily with the seaborn library:

sns.distplot(preprocessed\_data['registered'], label='registered')

sns.distplot(preprocessed\_data['casual'], label='casual')

plt.legend()

plt.xlabel('rides')

plt.title("Rides distributions")

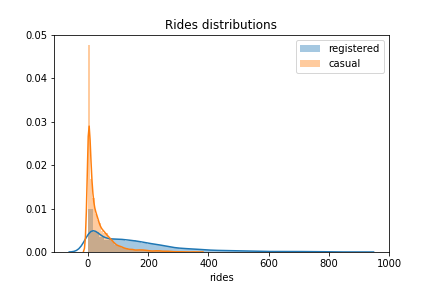


Figure 1.3: Distributions of registered vs casual rides

From Figure 1.3, we can easily see that registered users perform way more rides than casual ones. Furthermore, we can see that the two distributions are skewed to the right, with frequent occurrences, often having zero or small number or rides (probably overnight) and few entries with high number of registered rides.

Let us focus now on the evolution of rides over time, we can analyze the number of rides each day, with the following piece of code:

plot\_data = preprocessed\_data[['registered', 'casual', 'dteday']]

plot\_data = plot\_data.groupby('dteday').sum().plot(figsize=(10,6))

ax.set\_xlabel("time");

ax.set\_ylabel("number of rides per day");

Here we first take subset of the original **preprocessed\_data** dataset, afterwards we compute the total number of rides for each day, by first grouping the data by **dteday** column, and then summing the single entries for the **casual** and **registered** columns. The result of the code snipper is given in the following figure:

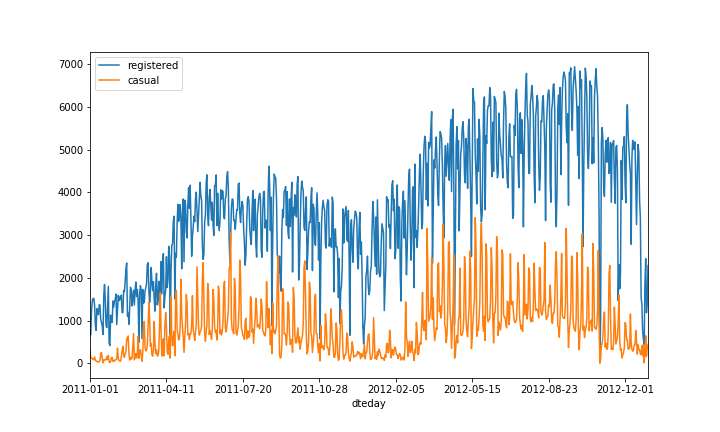


Figure 1.4: Number of rides per day

As we can see from the above figure, number of registered rides is always and significantly above the number of casual rides per day. Furthermore, we can observe that during winter months, the overall number of rides decreases (which is totally in line with our expectations, as bad weather and low temperatures play a negative impact on ride sharing services). Note that there is quite a lot of variance in the time series of the rides in Figure 1.4, one way to smoothen the curves is to take the tolling mean and standard deviation of the two time series, and plot those instead. This is what we are doing in the next code snipper:

plot\_data = preprocessed\_data[['registered', 'casual', 'dteday']]

window = 7

temp\_data = preprocessed\_data[['registered', 'casual', 'dteday']]

temp\_data = temp\_data.groupby('dteday').sum()

rolling\_means = temp\_data.rolling(window).mean()

rolling\_deviations = temp\_data.rolling(window).std()

ax = rolling\_means.plot(figsize=(10,6))

ax.fill\_between(rolling\_means.index, rolling\_means['registered'] + 2\*rolling\_deviations['registered'], rolling\_means['registered'] - 2\*rolling\_deviations['registered'], alpha = 0.2)

ax.fill\_between(rolling\_means.index, rolling\_means['casual'] + 2\*rolling\_deviations['casual'], rolling\_means['casual'] - 2\*rolling\_deviations['casual'], alpha = 0.2)

ax.set\_xlabel("time")

ax.set\_ylabel("number of rides per day")

which returns the following plot:

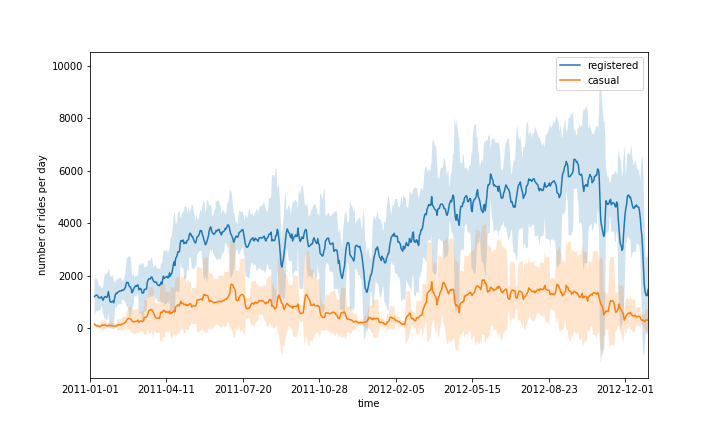


Figure 1.5: Rolling mean and standard deviation of rides

In order to compute the rolling statistics (mean and standard deviation), we use the rolling() function, in which we attach mean() and std(). This is a handy way to compute rolling statistics on time series, in which only recent entries account for computing them.

Let us now focus on the distributions on the requests over the separate hours and days of the week. We would expect certain time patterns to arise, as bike requests should be more frequent during certain hours of the day, depending on the day of the week. This analysis can be easily done by leveraging various functions from the **seaborn** package, as is shown in the following code snipper:

plot\_data = preprocessed\_data[['hr', 'weekday', 'registered', 'casual']]

plot\_data = plot\_data.melt(id\_vars=['hr', 'weekday'], var\_name='type', value\_name='count')

grid = sns.FacetGrid(plot\_data, row='weekday', col='type', height=3, aspect=2,

row\_order=['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday'])

grid.map(sns.barplot, 'hr', 'count', alpha=0.5)

Let us focus on the melt() function, applied on a pandas dataset. It will create a new dataset, in which values are grouped by the columns **hr** and **weekday**, while creating two new columns: **type** (containing the values **casual** and **registered)** and **count** (containing the respective counts for the types **casual** and **registered**).

The function seaborn.FacetGrid() will create new grid of plots, with rows corresponding to the different days of the week, and columns corresponding to the types. Finally, the map() function is applied to each element of the grid, creating the respective plots. The produced plot is shown in Figure 1.6. We can immediately note that over working days, the highest number of rides for registered users take place around 8AM in the morning, and 6PM in the evening. This is totally in line with our expectations, as most probably registered users use the bike sharing service for commuting. On the other hand, casual usage of bike sharing services over working days is quite limited, as the plot shows.

During the weekend, we can see that rides distributions change for both casual and registered users. Still registered rides are more frequent than casual ones, but both the distributions have the same shape, almost uniformly distributed during the time interval 11AM – 6PM.

As a conclusion, we could claim that most of the usage of bike sharing services occurs during working days, right before and after standard working time (9 to 5).

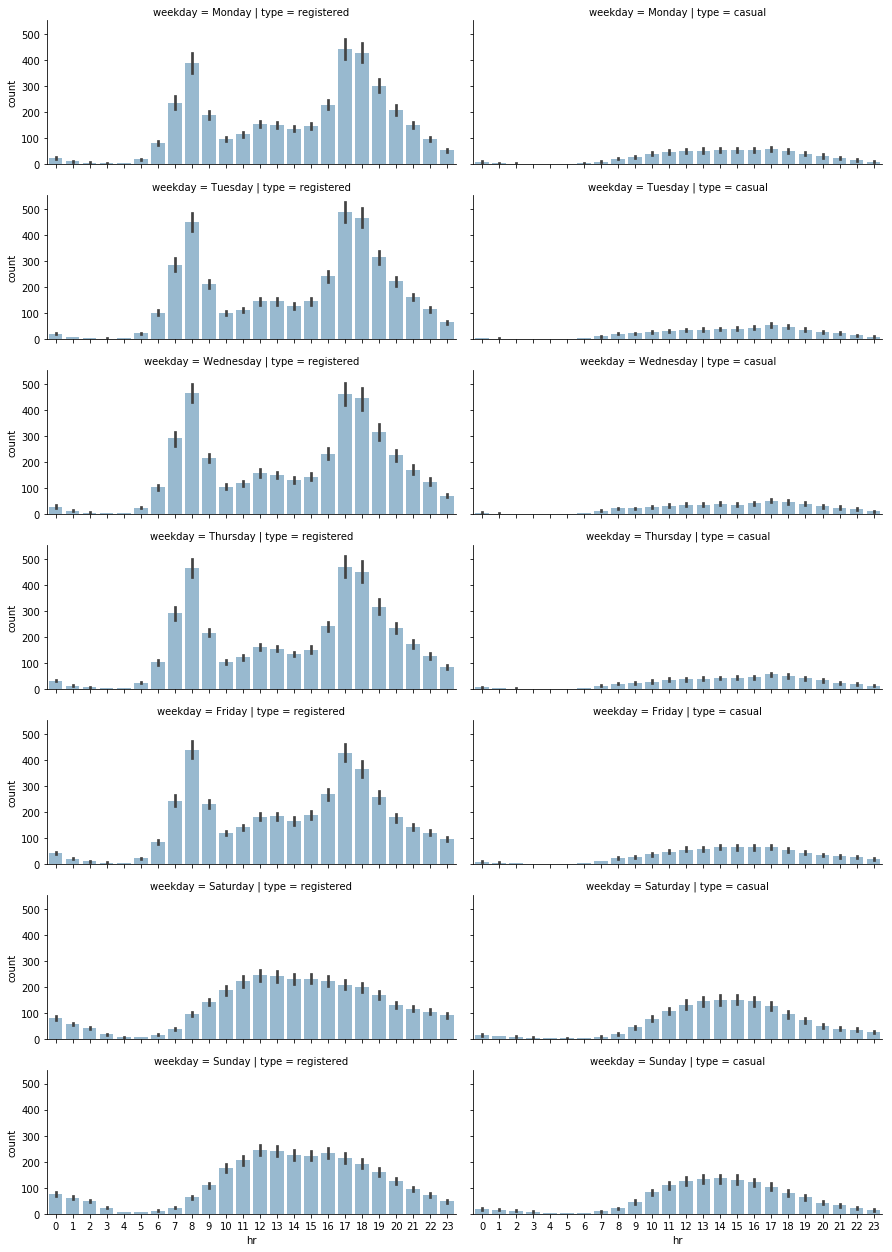


Figure 1.6: Distribution of rides on daily and hourly basis

Exercise 1.02: Analyzing season impact on rides

In this exercise, we will investigate the impact of the different seasons over the total number of rides. Our goal is to create grid plots, similar to the one in Figure 1.6, in which number of rides will be distributed over hours and weekdays, based on the current season.

1. Let us first start with combining the hours and seasons, we create a subset of the initial data by selecting the **hr**, **season**, **registered** and **casual** columns:

plot\_data = preprocessed\_data[['hr', 'weekday', 'registered', 'casual']]

1. Next step is to unpivot the data from wide to long format:

plot\_data = plot\_data.melt(id\_vars=['hr', 'season'], var\_name='type', value\_name='count')

1. We define the seaborn FacetGrid, in which rows represent the different seasons:

grid = sns.FacetGrid(plot\_data, row='season', col='type', height=2.5, aspect=2.5, row\_order=['winter', 'spring', 'summer', 'fall'])

Note that we are also specifying the desired order of rows here

1. Finally, we apply the seaborn.barplot() function to the FacetGrid elements:

grid.map(sns.barplot, 'hr', 'count', alpha=0.5)

The resulting plot is shown in Figure 1.7

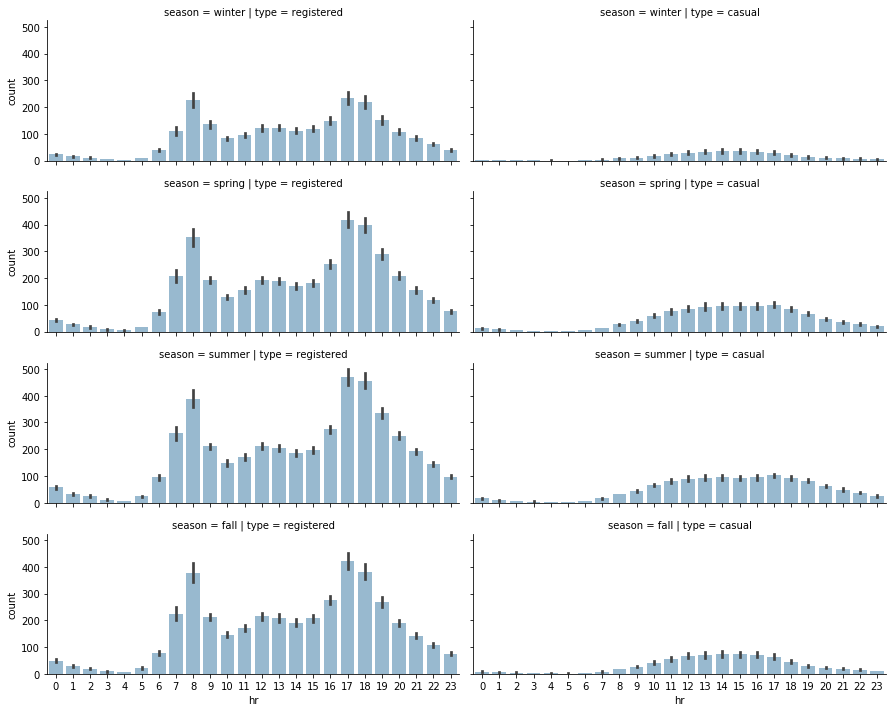


Figure 1.6: Distribution of rides on seasonal level

For the second part of the exercise (distribution of rides on a weekday basis), we proceed as in the first part:

1. First, we create a subset of the initial preprocessed data, containing only the relevant columns (**weekday**, **season**, **registered** and **casual**):

plot\_data = preprocessed\_data[['weekday', 'season', 'registered', 'casual']]

1. We again unpivot the data from wide to long format, but this time using **weekday** and **season** as grouping variables:

plot\_data = plot\_data.melt(id\_vars=['weekday', 'season'], var\_name='type', value\_name='count')

1. FacetGrid object is created, using the seaborn.FacedGrid() function:

grid = sns.FacetGrid(plot\_data, row='season', col='type', height=2.5, aspect=2.5, row\_order=['winter', 'spring', 'summer', 'fall'])

This step is similar to step 3.

1. Finally, we apply the seaborn.barplot() function to each of the elements in the FacetGrid object.

grid.map(sns.barplot, 'weekday', 'count', alpha=0.5,

order=['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday'])

Note that we are also specifying the order of the days of the week, which is passed as a parameter to the seaborn.barplot() function. The resulting plot is shown in the following figure:

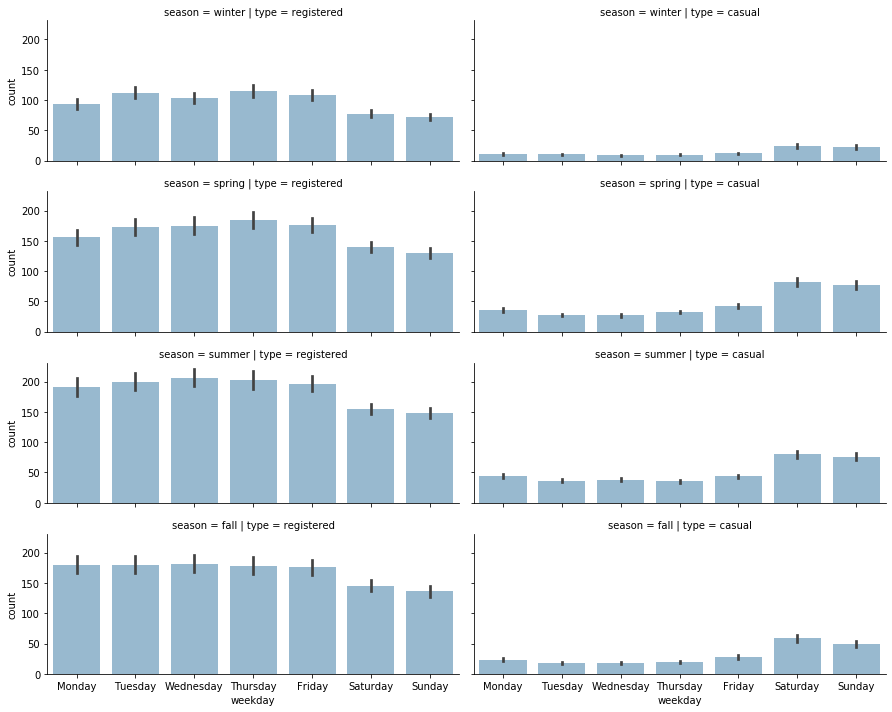


Figure 1.7: Distribution of rides over days of the week

An interesting pattern occurs from the analysis conducted in Exercise 1.02: there is a decreasing number of registered rides over the weekend (compared to the rest of the week), while the number of casual rides increases. This could enforce our initial hypothesis, that registered customers user the bike sharing service mostly for commuting (which could be the reason for the decreasing number of registered rides over the weekend), while casual customers use the service mostly occasionally over the weekend. Of course, such a conclusion cannot be based solely on plot observations, but has to be backed by statistical tests, which is the topic of our next section.

Hypothesis tests

Hypothesis testing is a branch of the inferential statistics, in which a general conclusion can be done about a large group (a population), based on analysis and measurements performed on a smaller group (a sample). A typical example could be making an estimation of the average height of a country’s citizens (in this case, the population), based on measurements performed on thousand people (the sample). Hypothesis testing tries to address the question “Is certain hypothetical value in line with the value obtained by direct measurements or not?”.

Although various statistical tests are known in literature and in practice, the general idea can be summarized in the following steps

* **Definition of Null and Alternative Hypothesis:** In this first step, a null hypothesis is defined (say, the average country’s population height is 175cm). This is the hypothesis which is going to be tested by the statistical test. The alternative hypothesis consists in the complement statement of the null one (in our example, the average height is not 175cm). The null and alternative hypothesis are always one complement to the other.
* **Identifying appropriate test statistic:** a test statistic is a quantity, whose calculation is based on the sample, and whose value is the basis for accepting or rejecting the null hypothesis. In most of the cases, it can be computed by the following formula:

where the *sample statistic* is the statistic value, computed on the sample (in our case, the average height of the thousand people), the *value under null hypothesis* is the value, assuming that the null hypothesis holds (in this case 175 cm), and the *standard error of the sample statistic* is the standard error in the measurement of the sample. Once the test statistic is identified and computed, we have to decide what type of probability distribution it follows. In most of the cases, the following probability distributions will be used:

* + student’s t-distribution (for t-test)
  + standard normal or z-distribution (for z-test)
  + chi-squared distribution (for chi-squared test)
  + F-distribution (for F-tests)

Choosing which distribution to use depends on the sample size, and the type of test. As a rule of thumb, if the sample size is greater than 30, we can expect that the assumptions of the central limit theorem hold, and the test statistic follows a normal distribution (hence use z-test). For more conservative approach, or samples with less than 30 entries, a t-test should be used (with test statistic following a student t-distribution).

* **Specifying significance level:** once the test statistic has been calculated, we have to decide whether we can reject the null hypothesis or not. In order to do that, we specify a significance level, which is the probability of rejecting a true null hypothesis. In most of the cases, a level of significance of 5% is used. This means that we accept that there exist 5% probability that we reject the null hypothesis, while being true (for a more conservative approach, we could always use 1% of even 0.5%). Once a significance level is specified, we have to compute the rejection points, which are values, with which the test statistic is compared. If it is larger than the specified rejection point(s), we can reject the null hypothesis and assume that the alternative hypothesis is true. We can distinguish two separate cases here:
  + two sided tests: tests, in which the null hypothesis assumes that the value “is equal to” a predefined value. For example: average height of the population is equal to 175cm. In this case, if we specify a significance level of 5%, then we have two critical values (one positive and one negative), with the probability of the two tails summing up to 5%. For example, if we assume that the sample mean of the height follows a normal distribution, the two critical values will be -1.96 and 1.96, and we will not reject the null hypothesis, if:
  + one sided tests: tests, in which the null hypothesis assumes that the value is “greater than” or “less than” a predefined value (for example: the average height is less than 175cm). In that case, if we specify a significance level of 5%, we will have only one critical value, with probability at the tail equal to 5%. In that case, assuming that the test statistic follows a normal distribution, the critical value will be 1.645 and we will reject the null hypothesis if:

Note that quite often, instead of computing the critical values of a certain significance level, we refer to the **p-value** of the test. The p-value is the smallest level of significance, at which the null hypothesis can be reject. The p-value provides also the probability of obtaining the observed sample statistic, assuming that the null hypothesis is correct. If the obtained p-value is less than the specified significance level, we can reject the null hypothesis, hence the p-value approach is in practice, an alternative (and most of the time, more convenient) way to perform hypothesis testing.

Let us now provide a practical example on performing hypothesis testing with python.

Exercise 1.03: Estimating average registered rides

In this exercise, we will show how to perform hypothesis testing on our bike sharing dataset.

1. First, we start with computing the average number of registered rides per hour. Note that this value will serve in formulating our null hypothesis, as here we are explicitly computing the population statistic (i.e. average number of rides). In most of the cases, such quantities are not directly observable, and in general, we only have an estimation for the population statistics.

population\_mean = preprocessed\_data.registered.mean()

1. Suppose now that we perform certain measurements, trying to estimate the true average number of rides, performed by registered users. For example, we could register all the rides during the summer of 2011 (this is going to be our sample):

sample = preprocessed\_data[(preprocessed\_data.season == "summer") & (preprocessed\_data.yr == 2011)].registered

1. At this point, we have to specify our significance level. A standard value is 0.05, i.e. when performing the statistical test, if the p-value, obtained by the statistical test is less than 0.05, we can reject the null hypothesis with at least 95%. The following code snipper shows how to perform that:

from scipy.stats import ttest\_1samp

test\_result = ttest\_1samp(sample, population\_mean)  
print(f"Test statistic: {test\_result[0]}, p-value: {test\_result[1]}")

The result of the previous test returns a p-value of 0.00049, which is less than the predefined critical value. Hence we can reject the null hypothesis and assume that the alternative hypothesis is correct.

1. Note that we have to make an important observation here: we computed the average number of rides on the true population, hence the value computed by the statistical test should be the same, so why we rejected the null hypothesis. The answer to that question lies in the fact that our sample is not a true representation of the population, but rather a biased one. In fact, we selected only entries from the summer of 2011. Hence neither data from the full year is present, nor entries from 2012. In order to show how such mistakes can compromise the results of statistical tests, let us perform again the test, but this time taking as sample 5% of the registered rides (selected randomly). The following code snipper performs that:import random

random.seed(111)

sample\_unbiased = preprocessed\_data.registered.sample(frac=0.05)

test\_result\_unbiased = ttest\_1samp(sample\_unbiased, population\_mean)

print(f"Unbiased test statistic: {test\_result\_unbiased[0]}, p-value: {test\_result\_unbiased[1]}")

This time, the computed p-value is equal to 0.49, which is way larger than the critical 0.05. Hence, we cannot reject the null hypothesis.

Note that when performing statistical tests, it is important to have unbiased sample of the data, otherwise, the results can be compromised.

Quite often, when performing statistical tests, we want to compare certain statistics on two different groups (for example the average height between women and men), and estimate whether there is a statistically significant difference, between the values obtained in the two groups. Let us denote with and the hypothetical means of the two groups, then we will have:

* null hypothesis:
* alternative hypothesis

Let us denote with and the sample means (the means obtained from the two groups), then the test statistic takes the form:

where and are the number of samples in the two groups, while is the pooled estimator of the common variance, computed as

with and being the variances of the two groups. Note that the test statistic in this case follows a student t-distribution with degrees of freedom.

As in the previous case, most of the times we don’t have to compute the test statistics by our self, as python already provides handy functions for that, plus the alternative approach of accepting or rejecting the null hypothesis, using the p-value is still valid.

Let us now focus on a practical example of how to perform a statistical test between two different groups. In the previous section, we observed graphically, that registered users tend to perform more rides during the working days, than the weekend. In order to assess this statement, we will perform a hypothesis test, in which we will test whether the mean of registered rides during working days is the same as during weekend. This is done in the following exercise.

Exercise 1.04: Hypothesis testing on registered rides

1. The first step in this exercise, is to formulate the null hypothesis. As we mentioned earlier, we are interested in identifying whether there is a statistically significant difference between registered rides during working and weekend days. Hence, our null hypothesis is that the average number of rides for registered users during working days is the same as the average number of rides during the weekends. Alternative hypothesis in that case is that those values are different.
2. Once the null hypothesis is established, we have to collect data for the two groups. This is done if the following code snipper:

weekend\_days = ['Saturday', 'Sunday']

weekend\_data = preprocessed\_data.registered[preprocessed\_data.weekday.isin(weekend\_days)]

workingdays\_data = preprocessed\_data.registered[~preprocessed\_data.weekday.isin(weekend\_days)]

1. Finally, we perform the two sample t-test by using the scipy.stats.ttest\_ind function:

from scipy.stats import ttest\_ind

test\_res = ttest\_ind(weekend\_data, workingdays\_data)

print(f"Statistic value: {test\_res[0]}, p-value: {test\_res[1]}")

The resulting p-value from this test is less than 0.00001, way below the standard critical 0.05 value. As a conclusion, we can reject the null hypothesis and confirm that our initial observation is correct: there is a statistically significant difference between number of rides performed during working and weekend days.

1. Finally, we plot the distributions of the two samples:

sns.distplot(weekend\_data, label='weekend days')

sns.distplot(workingdays\_data, label='working days')

plt.legend()

plt.xlabel('rides')

plt.title("Rides distributions")

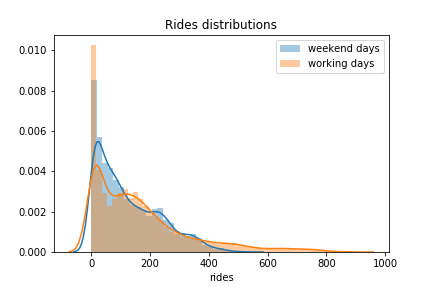


Figure 1.8: Distribution of registered rides: working days vs weekend

1. Same type of hypothesis testing can be performed for validating the second assumption from the last section: casual users perform more rides during the weekend. In this case, our null hypothesis is that the average number of rides during working days is the same as the average number of rides during the weekend, both performed only by casual customers. Alternative hypothesis then will result in statistically significant difference between the average number of rides, between the two groups.

weekend\_days = ['Saturday', 'Sunday']

weekend\_data = preprocessed\_data.casual[preprocessed\_data.weekday.isin(weekend\_days)]

workingdays\_data = preprocessed\_data.casual[~preprocessed\_data.weekday.isin(weekend\_days)]

test\_res = ttest\_ind(weekend\_data, workingdays\_data)

print(f"Statistic value: {test\_res[0]}, p-value: {test\_res[1]}")

The p-value, returned from the previous code snipper is 0, which is a strong evidence against the null hypothesis. Hence, we can conclude that also casual customers behave differently over the weekend (in this case, tend to use more the bike sharing service).

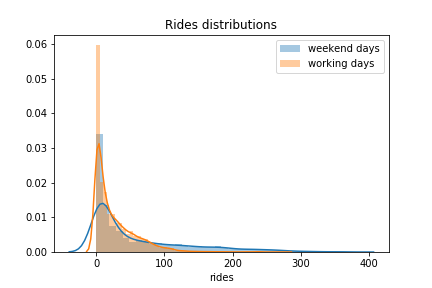


Figure 1.8: Distribution of casual rides: working days vs weekend

Weather related features

Let us now focus on the analysis on the group of features, representing the weather conditions. Our expectation is to observe a strong dependency on those features, of the current number of rides, as bad weather can significantly influence bike sharing services.

The weather features, we identified earlier, are the following:

* **weathersit:** categorical variable, representing the current weather situation. We encoded that variable with the following four values:
  + **clear:** representing clear weather, or few clouds
  + **cloudy:** representing mist or cloudy weather
  + **light\_rain\_snow:** light rain or snow is present
  + **heavy\_rain\_snow:** heavy rain on snow is present
* **temp:** normalized temperature in Celsius. Values are divided by 41, which means that the highest registered temperature in the data is 41°C (corresponding to 1 in our dataset).
* **atemp:** normalized feeling temperature in Celsius. Values are divided by 50, which means that the highest registered temperature in the data is 50°C (corresponding to 1 in our dataset).
* **hum:** humidity level in percentages.
* **windspeed:** windspeed in m/s.

From the provided description, we can see that most of the weather-related features assume continuous values (except **weathersit**). Furthermore, as both our variables in interest (**casual** and **registered** number of rides) are also continuously distributed, first and most common way to measure the relationship between two different continuous variables is to measure their correlation.

Correlation (also known as Pearson’s correlation) is a statistic that measures the degree to which two random variables move in relation to each other. In practice, it provides a numerical measure (scaled between -1 and 1), through which we can identify how much one of the variables would move in one direction, assuming that the other one moves. Let us denote with and the two random variables, the correlation coefficient between and is denoted with and is computed by the formula:

where and denote the mean of the two variables. A positive correlation between and means that increasing one of the values, will increase also the other one, while a negative correlation means that increasing one of the values, will decrease the other one.

Let us provide a practical example on computing the correlation between two variables. As we want to compare several variables, it makes sense to define a function, which performs the analysis between the single variables, as we want to follow the “don’t repeat yourself” principle.

def plot\_correlations(data, col):

# get correlation between col and registered rides

corr\_r = np.corrcoef(data[col], data["registered"])[0,1]

ax = sns.regplot(x=col, y="registered", data=data,

scatter\_kws={"alpha":0.05},

label=f"Registered rides (correlation: {corr\_r:.3f})")

# get correlation between col and casual rides

corr\_c = np.corrcoef(data[col], data["casual"])[0,1]

ax = sns.regplot(x=col, y='casual', data=data,

scatter\_kws={"alpha":0.05},

label=f"Casual rides (correlation: {corr\_c:.3f})")

#adjust legend alpha

legend = ax.legend()

for lh in legend.legendHandles:

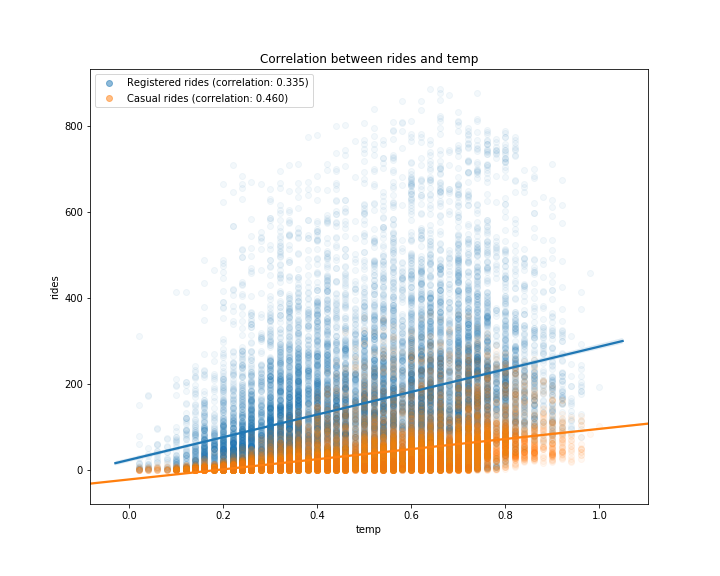
lh.set\_alpha(0.5)

ax.set\_ylabel("rides")

ax.set\_title(f"Correlation between rides and {col}")

return ax

Applying the previously defined function to the four columns (**temp**, **atemp**, **hum** and **windspeed**) returns the following figure:



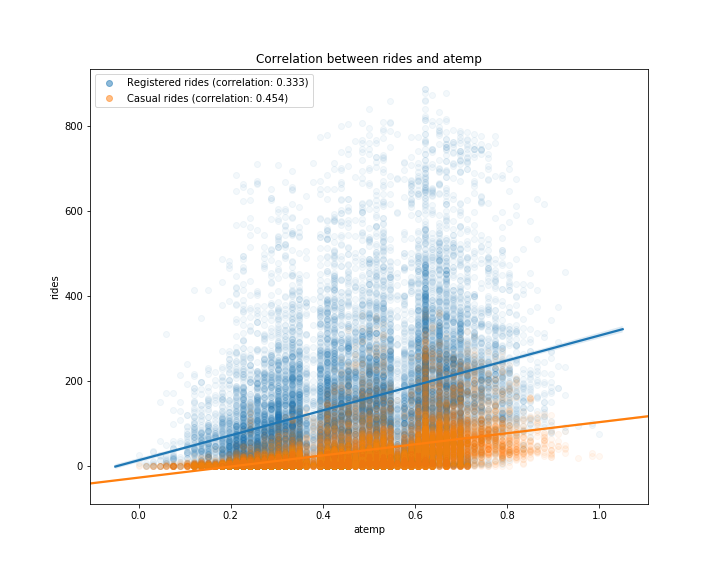
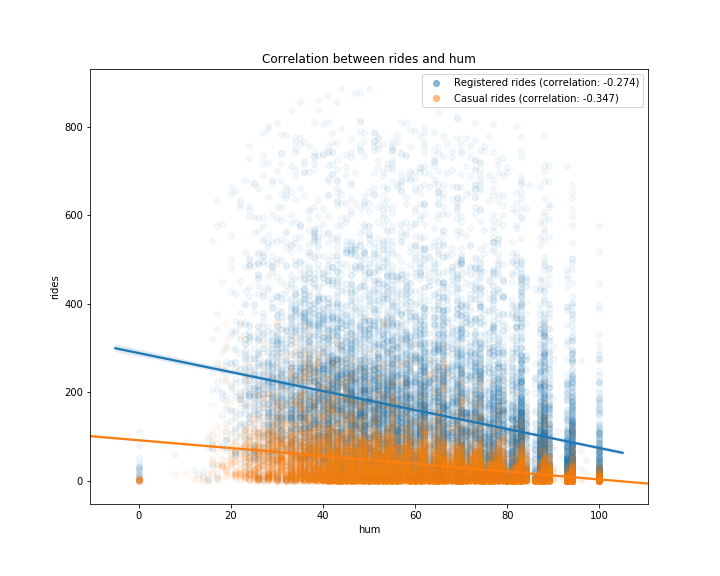


Figure 1.9: Correlations between rides and temp/atemp features



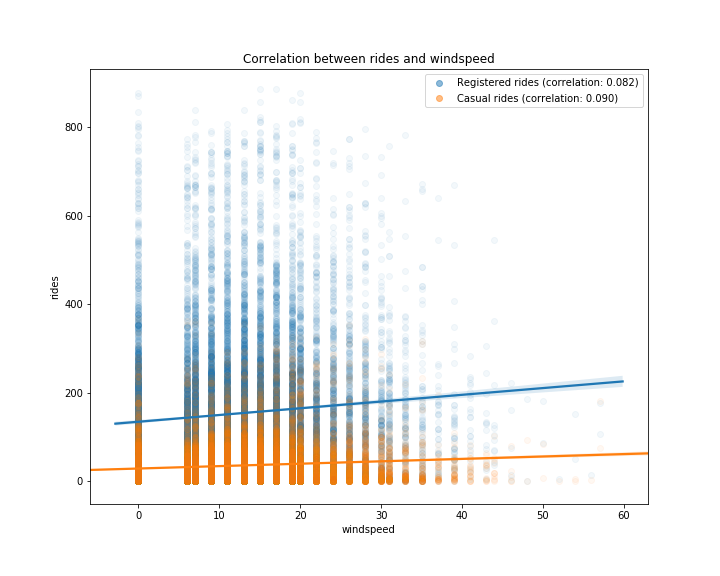


Figure 1.10: Correlations between rides and hum/windspeed features

From Figures 1.9, we can observe that higher temperatures have a positive impact on number of rides (correlation between registered/casual rides and temperature is respectively 0.335 and 0.46). Note that as the values in **registered** column are widely spread with respect to the different values in **temp**, we have a lower correlation, compared to the **casual** column. Same pattern can be observed in Figure 1.10, in which humidity level has a negative correlation with both type of rides (-0.274 for **registered** and -0.347 for **casual**). This means that with high level of humidity (mist or rain), customers will tend not to use the bike sharing service.

One of the major drawbacks of the correlation coefficient is its assumption of linear relationship between the two random variables. This is quite a strong assumption, as most of the time, relationships in nature are not linear. A measure, which generalizes the Pearson’s correlation to monotonic relationships between two variables is the Spearman rank correlation.

Let us illustrate the difference between the two measures in the following example.

Exercise 1.05: Difference between Pearson and Spearman correlation

In this exercise, we investigate the difference between the Pearson’s correlation (in which a linear relationship between the two variables is assumed), and the Spearman’s correlation (in which only monotonic relationship is required). For a sake of better presenting the difference between the two measures, we will create a synthetic data, which will serve our purpose.

1. Let us start with defining our random variables. We will create an variable, which will represent our independent variable, and two dependent ones: and , given by:

where represents a noise component, normally distributed with mean 0 and standard deviation of 0.1

# define random variables

x = np.linspace(0,5, 100)

y\_lin = 0.5\*x + 0.1\*np.random.randn(100)

y\_mon = np.exp(x) + 0.1\*np.random.randn(100)

1. We can compute the Pearson’s and Spearman’s correlations by using the pearsonr() and spearmanr() functions, in the scipy.stats module:

# compute correlations  
corr\_lin\_pearson = pearsonr(x, y\_lin)[0]

corr\_lin\_spearman = spearmanr(x, y\_lin)[0]

corr\_mon\_pearson = pearsonr(x, y\_mon)[0]

corr\_mon\_spearman = spearmanr(x, y\_mon)[0]

1. Note that both the pearsonr() and spearmanr() functions return a two dimensional array, in which the first value is the respective correlation, while the second one is the p-value of a hypothesis test, in which the null hypothesis assumes that the computed correlation is equal to zero. This is quite handy at times, as we not only compute the correlation, but also test its statistical significance against being zero.
2. Finally, we visualize both the data and the computed correlations:

class User

# visualize variables

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10,5))

ax1.scatter(x, y\_lin)

ax1.set\_title(f"Linear relationship\n

Pearson: {corr\_lin\_pearson:.3f},

Spearman: {corr\_lin\_spearman:.3f}")

ax2.scatter(x, y\_mon)

ax2.set\_title(f"Monotonic relationship\n

Pearson: {corr\_lin\_pearson:.3f},

Spearman: {corr\_lin\_spearman:.3f}")

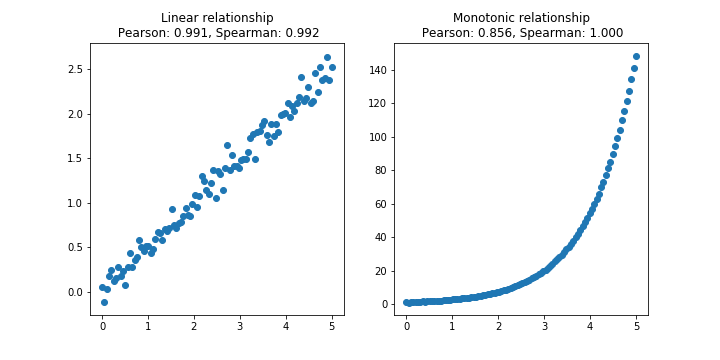


Figure 1.11: Difference between Pearson and Spearman correlations

1. As we can see from the above figure, when the relationship between the two variables is linear (figure on the left), the two correlation coefficients are very similar. In the monotonic (figure on the right), the linear assumption of the Person correlation fails, and although the correlation coefficient is still quite high (0.856), it is not capable to capture the perfect relationship between the two variables. On the other side, Spearman correlation coefficient is 1, which means that it succeeds the capture the almost perfect relationship between the two variables

Let us return to our bike sharing data, and investigate the relationship between the different variables, in light of the difference between the two correlation measures. First, we define a function, which on provided data and column, computes the Pearson and Spearman correlation coefficients with the **registered** and **casual** rides:

# define function for computing correlations

def compute\_correlations(data, col):

pearson\_reg = pearsonr(data[col], data["registered"])[0]

pearson\_cas = pearsonr(data[col], data["casual"])[0]

spearman\_reg = spearmanr(data[col], data["registered"])[0]

spearman\_cas = spearmanr(data[col], data["casual"])[0]

return pd.Series({"Pearson (registered)": pearson\_reg,

"Spearman (registered)": spearman\_reg,

"Pearson (casual)": pearson\_cas,

"Spearman (casual)": spearman\_cas})

Note that the previously defined function returns a pandas.Series() object, which will be used in creating a new dataset, containing the different correlations:

# compute correlation measures between different features

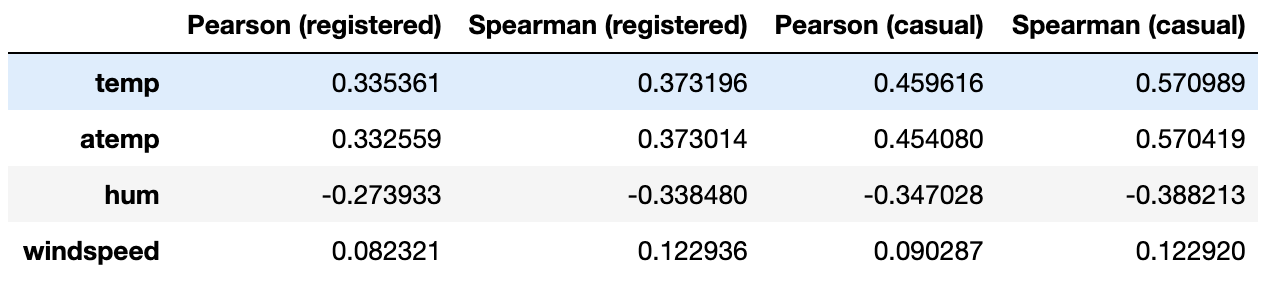
cols = ["temp", "atemp", "hum", "windspeed"]

corr\_data = pd.DataFrame(index=["Pearson (registered)", "Spearman (registered)",

"Pearson (casual)", "Spearman (casual)"])

for col in cols:

corr\_data[col]= compute\_correlations(preprocessed\_data, col)

Result is shown below:  


As we can observe, for most of the variables, the Pearson and Spearman correlation coefficient are close enough (some non-linearity in to be expected). The most striking difference between the two coefficients occur when comparing **temp** (and **atemp**) and **casual** columns. More precisely, the Spearman correlation is quite higher, meaning that there is a significant evidence toward nonlinear relatively strong and positive relationship. An interpretation of this result is that casual customers are keener to use the bike sharing service when temperatures are higher. We have already seen from previous analysis, that casual customers ride mostly during the weekend, and do not rely on bike sharing services for their commuting to work. This conclusion is again confirmed by the higher relationship with temperature, as opposed to registered customers, whose rides have lower correlation with temperature.

One last, but quite useful analysis, when performing comparison between different continuous features, is the correlation matrix plot. It allows the analyst to quickly visualize possible relationship between the different features and identify potential clusters with highly correlated features. The next code snipper does that:

# plot correlation matrix

cols = ["temp", "atemp", "hum", "windspeed", "registered", "casual"]

plot\_data = preprocessed\_data[cols]

corr = plot\_data.corr()

fig = plt.figure(figsize=(10,8))

plt.matshow(corr, fignum=fig.number)

plt.xticks(range(len(plot\_data.columns)), plot\_data.columns)

plt.yticks(range(len(plot\_data.columns)), plot\_data.columns)

plt.colorbar()

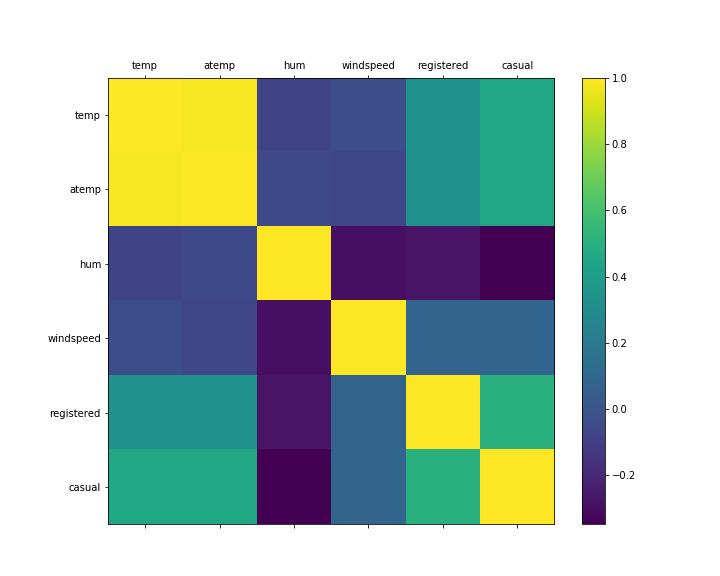


Figure 1.12: Correlation matrix between continuous weather features and rides

In the last part of this section, we are going to investigate deeper the impact of weather conditions (i.e. the **weathersit** column) on the number of rides, combined with the rest of the features.

Exercise 1.06: Full analysis on weather features

In this exercise, we combine the impact of the weather conditions on the number of rides. We also investigate the correlations between the various weather features and number of rides, and test the statistical significance of the computed correlation, against the null hypothesis of having a correlation equal to zero.

1. First, we need to define a function, which returns the combined plots between a selected weather feature and rides, based on a specific weather condition. We also include the computation of the correlation values, and their respective p-values (using the Pearson correlation test). We use as a reference the function plot\_correlations() defined previously:

def plot\_correlations\_new(data, col, weather\_cond):

# extract data for the specific weather condition

plot\_data = data[data['weathersit'] == weather\_cond]

# get correlation between col and registered rides

pearson\_corr\_r = pearsonr(plot\_data[col], plot\_data["registered"])

ax = sns.regplot(x=col, y="registered", data=plot\_data,

scatter\_kws={"alpha":0.05},

label=f"Registered (corr: {pearson\_corr\_r[0]:.3f}, p-val: {pearson\_corr\_r[1]:.3f})")

# get correlation between col and casual rides

pearson\_corr\_c = pearsonr(plot\_data[col], plot\_data["casual"])

ax = sns.regplot(x=col, y="casual", data=plot\_data,

scatter\_kws={"alpha":0.05},

label=f"Casual (corr: {pearson\_corr\_c[0]:.3f}, p-val: {pearson\_corr\_c[1]:.3f})")

# adjust legend alpha

legend = ax.legend()

for lh in legend.legendHandles:

lh.set\_alpha(0.5)

ax.set\_xlabel("")

ax.set\_ylabel("")

ax.set\_title(f"{col} | {weather\_cond}")

return ax

Note that in the plot\_correlations() function, we are first filtering the data, based on the provided weather condition. Then we compute the Person correlation, between the **registered** and **casual** columns, and the specified column as input in the function. Finally, we plot the filtered data and include the computed correlations.

1. As next step, we need to create a 4x4 grid plot, in which each column represents a different weather condition (**clear**, **cloudy**, **light\_rain\_snow** and **heavy\_rain\_snow**), and each row represents a weather column (**temp**, **atemp**, **hum** and **windspeed**). For this reason, we rely on the enumerate() python function, which iterates on both objects and their indexes, in a provided collection of objects. The following code snipper produces the plot in Figure 1.13

weather\_conditions = preprocessed\_data.weathersit.unique()

columns = ["temp", "atemp", "hum", "windspeed"]

plt.figure(figsize=(20,30))

for col\_index, col in enumerate(columns):

for row\_index, weather\_cond in enumerate(weather\_conditions):

plot\_number = row\_index + col\_index\*4 + 1

plt.subplot(4,4,plot\_number)

plot\_correlations\_new(preprocessed\_data, col, weather\_cond)

Note that in order to produce the grid plot, we use the pyplot.subplot(n, m, k) function, which creates the kth plot in an n x m grid plot.

1. Last step is to interpret the results from Figure 1.13. First of all, we can see that the most highly correlated feature with the **registered** and **casual** rides features are the **temp** and **atemp** features (almost similar one to each other, as the first one is the true temperature, while the second one is the perceived one). This result is not surprising, as it is already in line with our expectations. Furthermore, **casual** rides’ correlation is higher than the **registered** one. This result can be interpreted as registered users, being less dependent on the current weather and temperature conditions, than casual ones. Note that for all three weather conditions (**clear**, **cloudy** and **light\_rain\_snow**) the p-values of the computed correlations are practically zero (which means strong significance against the null hypothesis of having a correlation equal to zero), while the p-values of the correlations with the last weather condition, **heavy\_rain\_snow**, are way larger than 0.05. This means that even if we obtain a correlation coefficients different from zero, we cannot reject the null hypothesis of having a correlation equal to zero. Possible reason for having such a large p-value is probably not enough samples in the data, containing bad weather conditions.

Let us now focus on the **hum** column. It’s correlations with the **registered** and **casual** rides are negative in all three weather conditions, meaning that humidity level has a negative impact on bike sharing services. Of course, such a conclusion is mostly confirming our common sense, in which people tent to rely on alternative transportation services, when raining. Also in this case, correlations computed under **heavy\_rain\_snow** weather conditions are not reliable, as the p-values of the Pearson tests are larger than the critical 0.05.

Finally, the **windspeed** column has a very small impact on bike sharing services, as it’s correlations with the two rides columns is quite low (especially under **cloudy** and **light\_rain\_snow** weather conditions).

This concludes our analysis on the weather columns, and their impact on the number of rides. In the next section, we will exploit a more advanced techniques for time dependent features, known as time series analysis.

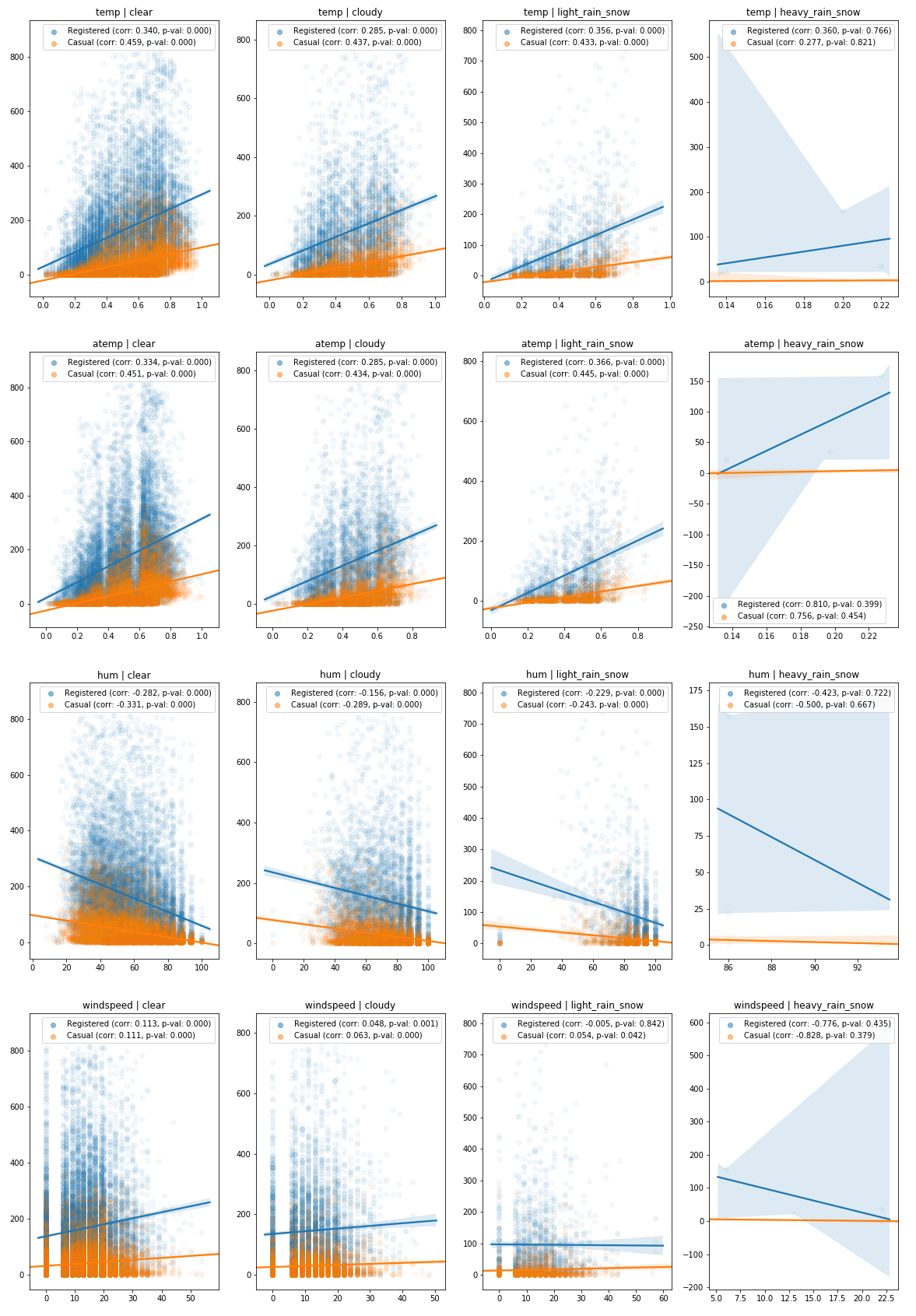


Figure 1.13: Correlations between temporal features, filtered by weather condition

Time series analysis

In this section, we perform a time series analysis on the rides columns (**registered** and **casual**) in the bike sharing dataset.

A time series is a sequence of observations, equally spaced in time and in a chronological order. A typical examples of time series are stock prices, yearly rainfalls, or the number of customers, using a specific transportation service every day. When observing time series, their fluctuation might result random, but often it exhibits certain patterns (for example highs and lows of ocean’s tides, or hotels’ prices in a proximity of fairs).

When studying time series, an important concept is the notion of stationarity. A time series is said to be *strongly stationary,* if all aspects of its behavior do not change in time. In other words, given a time series , for each and , the distributions of and are the same. In practice, strong stationarity is quite a restrictive assumption, and in most of the cases, not satisfied. Furthermore, for most of the methods illustrated later in this section to work, it is enough to have a time series which is *weakly stationary*, i.e. its mean, standard deviation and covariance do not change in time. More precisely, given

* (a constant) for every
* (a constant) for every
* for every and and some function

In order to check stationarity in practice, we can rely on one of the following methods (in most of the cases, we perform both):

* **Plotting rolling statistics:** in this method, we plot the rolling mean and standard deviation of the analyzed time series, and assure if those values fluctuate around a constant ones.
* **Dickey-Fuller stationarity test**: a statistical test, in which the null hypothesis is that the time series is nonstationary. Hence, when performing the test, a small p-value would be a high evidence against the time series being nonstationary.

In practice, we rely on both the techniques, as plotting the rolling statistics is not a rigorous approach. Let us define a utility function, which will perform both the tests for us:

# define function for plotting rolling statistics and ADF test for time series

from statsmodels.tsa.stattools import adfuller

def test\_stationarity(ts, window=10, \*\*kwargs):

# create dataframe for plotting

plot\_data = pd.DataFrame(ts)

plot\_data['rolling\_mean'] = ts.rolling(window).mean()

plot\_data['rolling\_std'] = ts.rolling(window).std()

# compute p-value of Dickey-Fuller test

p\_val = adfuller(ts)[1]

ax = plot\_data.plot(\*\*kwargs)

ax.set\_title(f"Dickey-Fuller p-value: {p\_val:.3f}")

We also need to extract the daily **registered** and **casual** rides from our preprocessed data:

# get daily rides

daily\_rides = preprocessed\_data[["dteday", "registered", "casual"]]

daily\_rides = rides.groupby("dteday").sum()

# convert index to DateTime object

daily\_rides.index = pd.to\_datetime(daily\_rides.index)

Applying the previously defined test\_stationarity() function, to the daily rides produces the following plots:

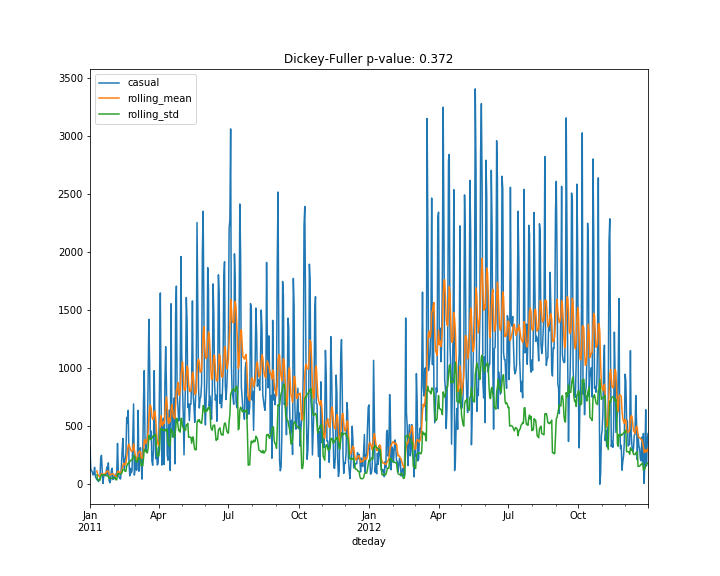
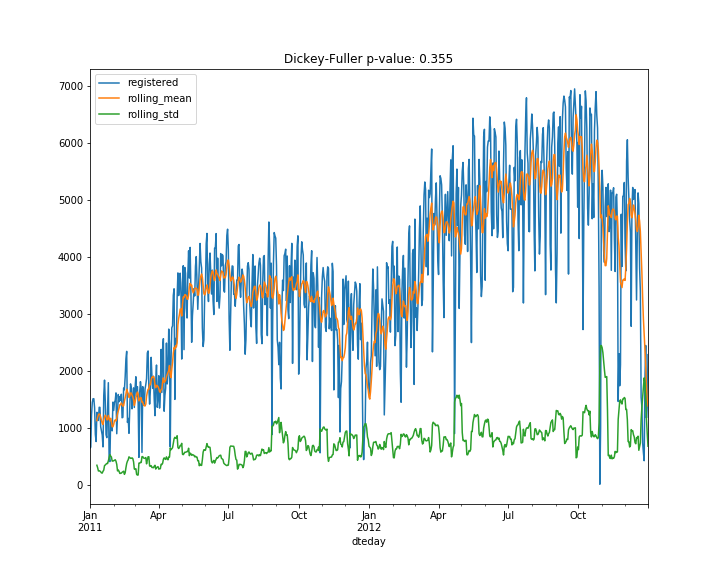


Figure 1.14: Stationarity test results for aggregated daily registered and casual rides

From the performed tests, we can see that both the moving average and standard deviations are not stationary, furthermore the Dickey-Fuller test returns values of 0.355 and 0.372 for the **registered** and **casual** columns, respectively. This is a strong evidence that the time series are not stationary, and we need to process them, in order to obtain a stationary one.

A common way to detrend a time series and make it stationary is to subtract either its rolling mean or its last value, or to decompose it into a components, containing its trend, seasonality and residual components. Let us first check if the time series are stationary, by subtracting their rolling means and last values:

# subtract rolling mean

registered = daily\_rides["registered"]

registered\_ma = registered.rolling(10).mean()

registered\_ma\_diff = registered - registered\_ma

registered\_ma\_diff.dropna(inplace=True)

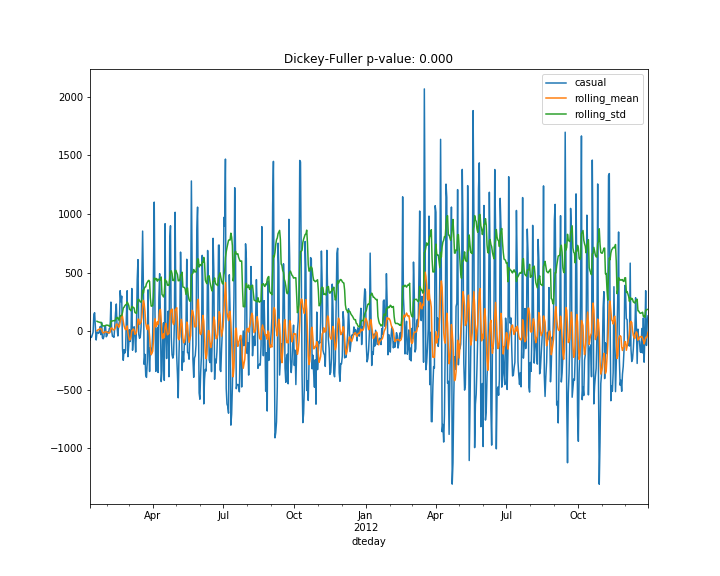
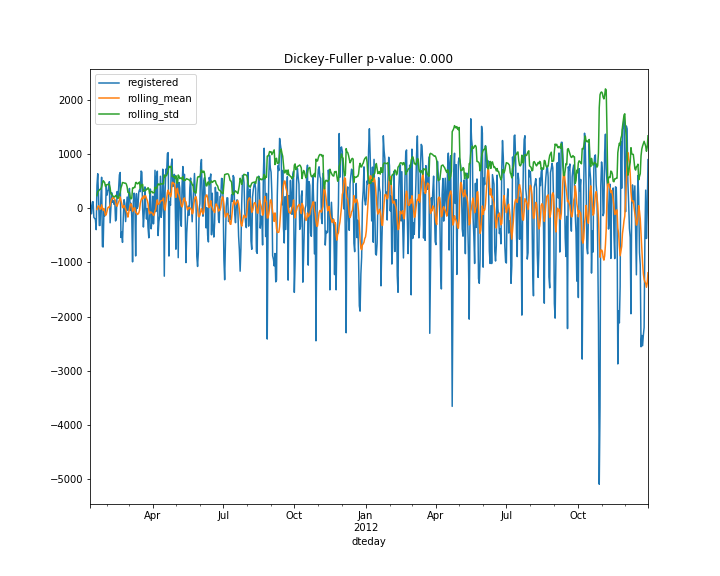
casual = daily\_rides["casual"]

casual\_ma = casual.rolling(10).mean()

casual\_ma\_diff = casual - casual\_ma

casual\_ma\_diff.dropna(inplace=True)

The resulting time series are tested for stationarity and results are shown in the following figure:



Subtracting the last value can be done in the same way:

# subtract last value

registered = daily\_rides["registered"]

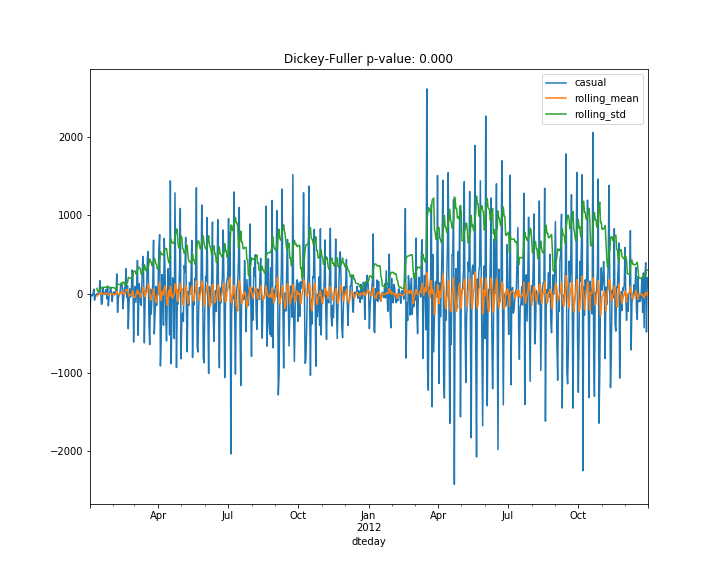
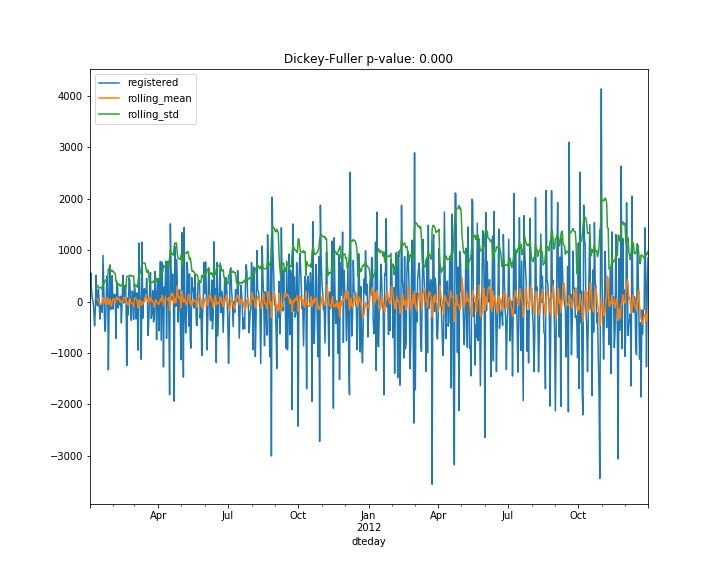
registered\_diff = registered - registered.shift()

registered\_diff.dropna(inplace=True)

casual = daily\_rides["casual"]

casual\_diff = casual - casual.shift()

casual\_diff.dropna(inplace=True)



As we can see, both the techniques returned time series, which are stationary, according to the Dickey-Fuller test. Note that an interesting pattern occurs in the casual series. i.e. rolling standard deviation exhibits a clustering effect, i.e. periods, in which standard deviation is higher, and periods, in which it is lower. This effect is quite common in certain fields (finance for instance) and is known as *volatility clustering*. Possible interpretation, relative to our data is that number of casual rides increase during summer periods, while dropping during the winter.

As we saw from the last analysis, removing both the rolling mean and the last value, returned stationary time series. Let us check also the last-mentioned technique. i.e. the time series decomposition.

Exercise 1.07: Time series decomposition in trend, seasonality and residuals

In this exercise, we exploit the seasonal decomposition in the statsmodel python library, to decompose the number of rides into three separate components: trend, seasonal and residual component

1. First, we use the statsmodel.tsa.seasonal. seasonal\_decompose() method for decomposing the **registered** and **casual** rides:

from statsmodels.tsa.seasonal import seasonal\_decompose

registered\_decomposition = seasonal\_decompose(daily\_rides["registered"])

casual\_decomposition = seasonal\_decompose(daily\_rides["casual"])

1. For each decomposition, the three signals are available in the .trend, .seasonal and .resid variables. Furthermore, the generated decomposition themselves, can provide visual results by calling the .plot() method:

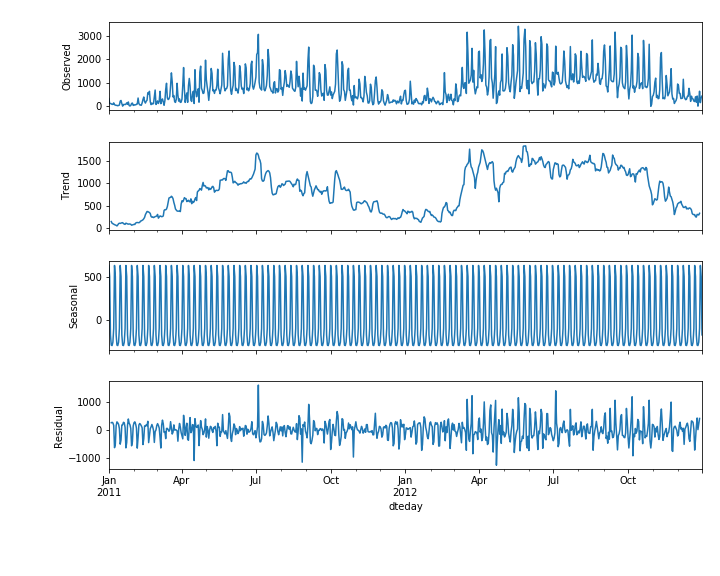
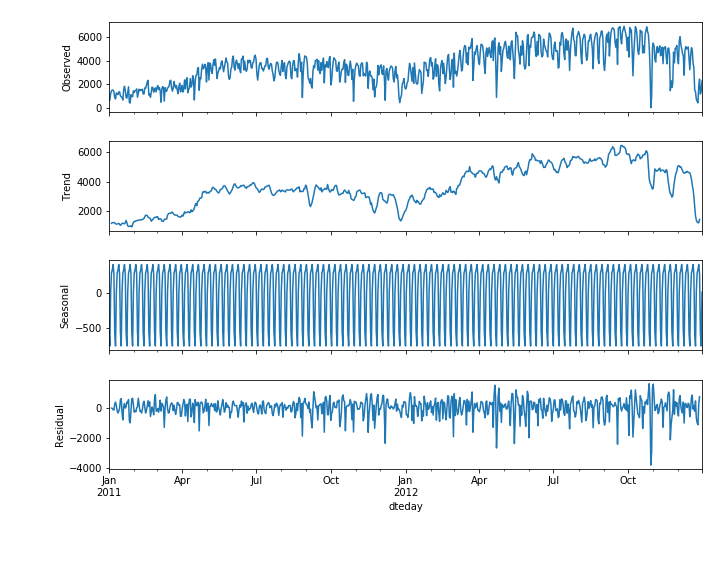
# plot decompositions

registered\_plot = registered\_decomposition.plot()

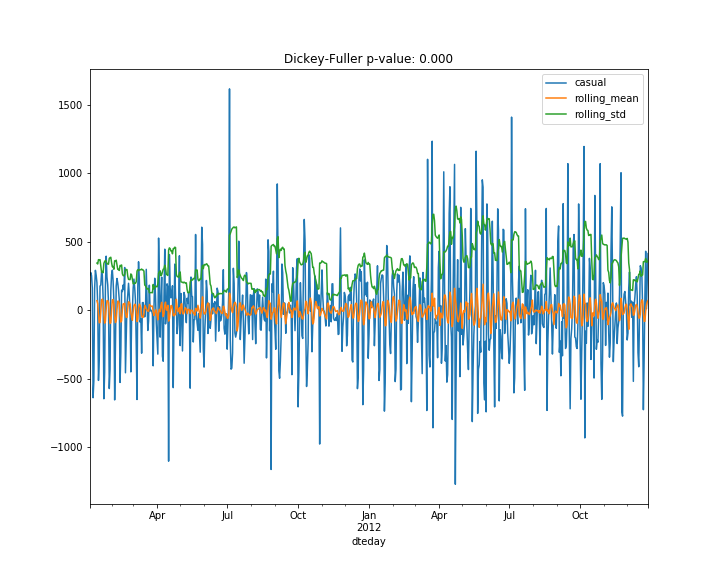
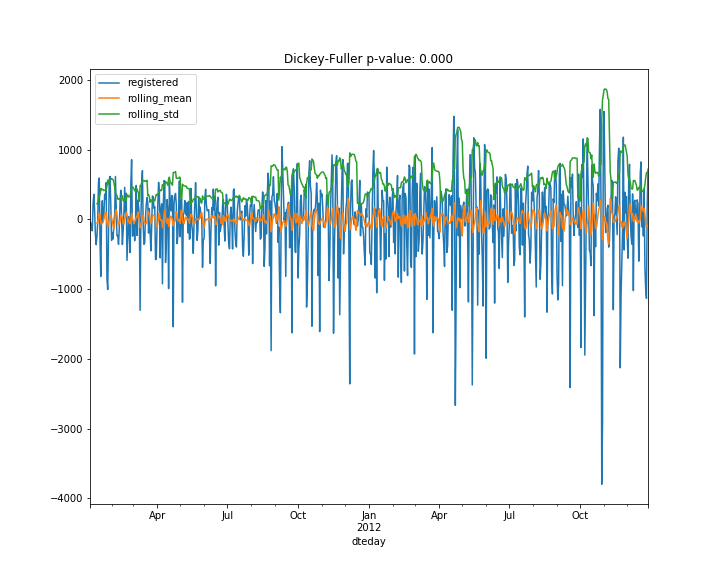
registered\_plot.set\_size\_inches(10, 8)

casual\_plot = casual\_decomposition.plot()

casual\_plot.set\_size\_inches(10, 8)



1. Finally, let us test the obtained residuals for stationarity:



As we can see, the residuals satisfy our stationary test.

A common approach to modeling time series is to assume that past observations somehow influence future ones. For instance, customers, who are satisfied by using the bike sharing service, will more likely recommend it. Hence, by increasing the number of customers and the quality of the service, increases the number of recommendations and thus, the number of new customers. In this way, a positive feedback loop is created, in which current number of rides correlates with its past values. These types of phenomena are the topic of the next section.

ARIMA models

*Auto Regressive Integrated Moving Average* models (ARIMA for short) are a class of statistical models, which try to explain the behavior of a time series by its own past values. Being a class of models, ARIMA models are defined by a set of parameters (p,d,q), each one corresponding to a different component of the ARIMA model:

* **Auto Regressive of order p:** an auto regressive model of order p (AR(p) for short) models the current time series entry, as a linear combination of its last p values. The mathematical formulation is the following:

where is the intercept term, is the lag-I term of the series with the respective coefficient while is the error term (i.e. normally distributed random variable with mean 0 and variance ).

* **Moving Average of order q:** a moving average model of order q (MA(q) for short) attempts to model the current value of the time series, as a linear combination of its past error terms. Mathematically speaking, it has the following formula:  
  As in the Auto Regressive model, represents a bias term, are parameters to be estimated in the model and are the error terms, at times respectively.
* **Integrated component of order d:** the integrated component represents a transformation in the original time series, in which the transformed series is obtained by getting the difference between and , hence:

The integration term is used for detrending the original time series, and making it stationary. Note that we already saw this type of transformation, when we subtracted the previous entry in the number of rides, i.e. we applied an integration term of order 1.

In general, when we apply an ARIMA model of order (p,d,q) to a time series , we obtain the following model:

1. First, we integrate the original time series of order d, hence obtain the new series:
2. Then we apply a combination of AR(p) and MA(q) models, also known as Auto Regressive Moving Average model, or ARMA(p,q) to the transformed series :

Where the coefficients are to be estimated.

A standard method for finding the parameters (p,d,q) of an ARIMA model is to compute the *autocorrelation* and *partial autocorrelation functions* (ACF and PACF for short). The autocorrelation function measures the Pearson correlation between lagged values in a time series, as a function of the lag:

In practice, the ACF measures the complete correlation between the current entry and its past entries, lagged by . Note that when computing the ACF(k), the correlation between with all intermediate values () is not removed. In order to account only for the correlation between and , we often refer to the partial autocorrelation function, which measures only the impact of on .

ACF and PACF are in general used for determining the order of integration, when modeling a time series with an ARIMA model. For each lag, the correlation coefficient and level of significance is computed. In general, we aim at integrated series, in which only the first few lags have correlation, greater than the level of significance. We illustrate this in the following exercise:

Exercise 1.08: ACF and PACF plots for registered rides

In this exercise, we plot the autocorrelation and partial autocorrelation functions, for the **registered** number of rides.

1. The python package statsmodels already contains the necessary methods for plotting the ACF and PACF:

from statsmodels.graphics.tsaplots import plot\_acf, plot\_pacf

1. As next step, we define a 3x3 grid, and plot the ACF and PACF for the original series of registered rides, as well as for its first and second order integrated series:

fig, axes = plt.subplots(3, 3, figsize=(25, 12))

# plot original series

original = daily\_rides["registered"]

axes[0,0].plot(original)

axes[0,0].set\_title("Original series")

plot\_acf(original, ax=axes[0,1])

plot\_pacf(original, ax=axes[0,2])

# plot first order integrated series

first\_order\_int = original.diff().dropna()

axes[1,0].plot(first\_order\_int)

axes[1,0].set\_title("First order integrated")

plot\_acf(first\_order\_int, ax=axes[1,1])

plot\_pacf(first\_order\_int, ax=axes[1,2])

# plot first order integrated series

second\_order\_int = first\_order\_int.diff().dropna()

axes[2,0].plot(first\_order\_int)

axes[2,0].set\_title("Second order integrated")

plot\_acf(second\_order\_int, ax=axes[2,1])

plot\_pacf(second\_order\_int, ax=axes[2,2])

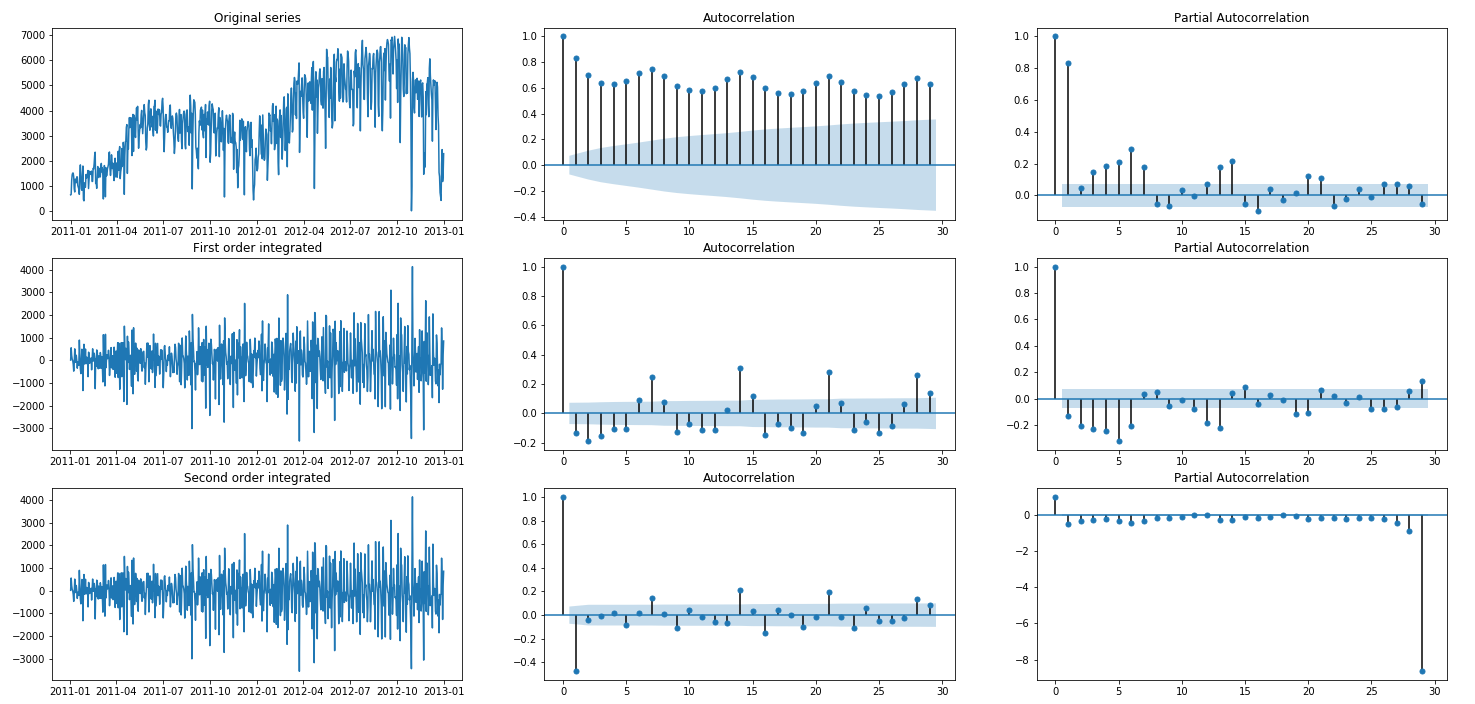


Figure 1.14: Autocorrelation and partial autocorrelation plots of registered rides

1. As we can see from the figure above, the original series exhibits several autocorrelation coefficients, which are above the threshold. The first order integrated series has only few, which makes it a good candidate for further modelling (hence, selecting an ARIMA(p, 1, q) model). Finally, the second order integrated series present a large negative autocorrelation of lag 1, which in general is a sign of too large order of integration.

Let us now focus on finding the model parameters, and the coefficients for an ARIMA(p,d,q) model, based on the observed registered rides. The general approach is to try different combinations of parameters and chose the one which minimizes certain information criterion, for instance the Akaike Information Criterion (AIC) or the Bayesian Information Criterion (BIC):

* **Akaike Information Criterion:**
* **Bayesian Information Criterion:**

where is the number of parameters in the selected model, is the number of samples, while is the log likelihood. As we can see, there is not a substantial difference between the two criteria, and in general, both are used. In case different optimal models are selected, according to the different IC, we tend to find a model in between.

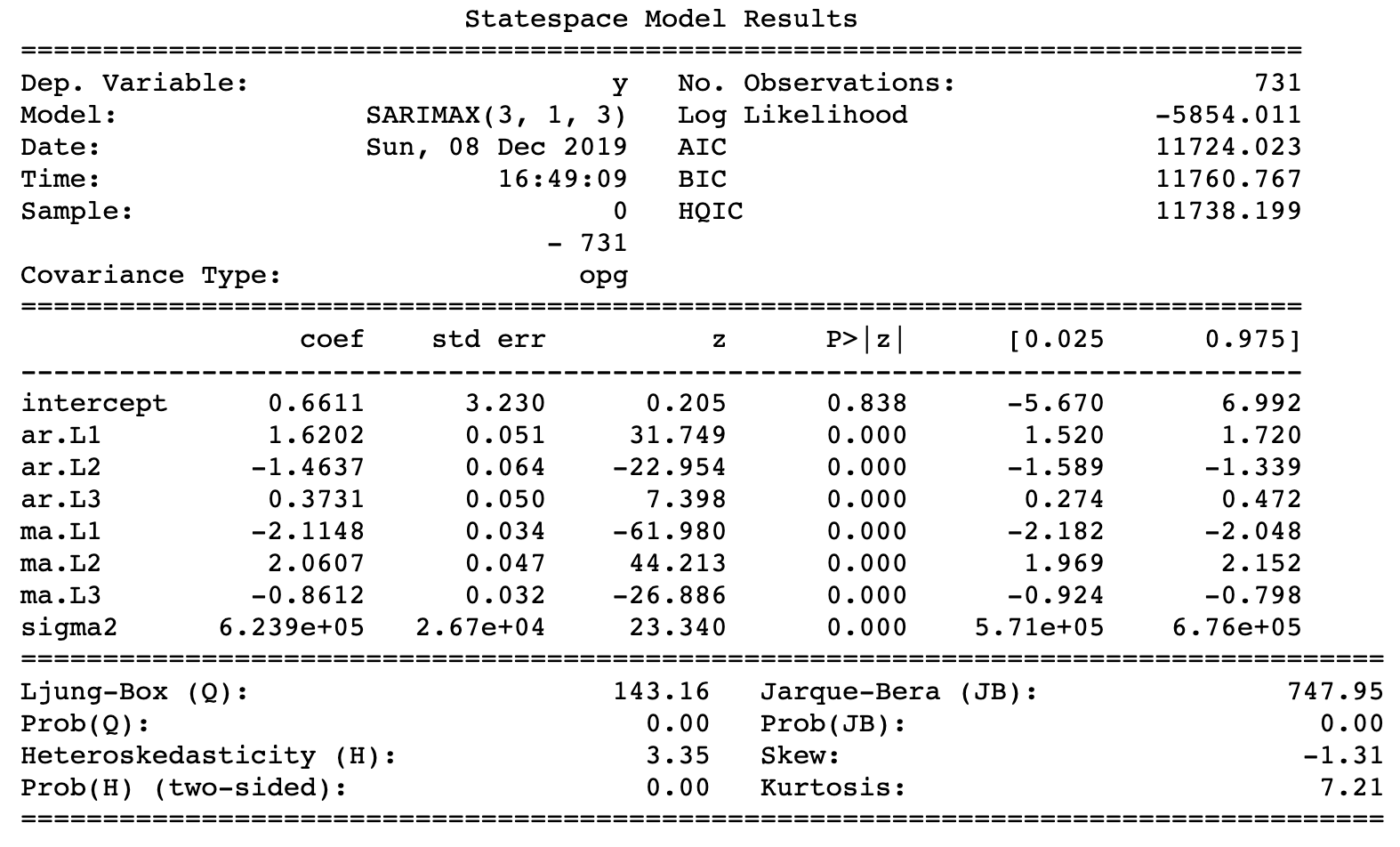
In the next code snipper, we fit an ARIMA(p,d,q) model to the **registered** column:

from pmdarima import auto\_arima

model = auto\_arima(registered, start\_p=1, start\_q=1, max\_p=3, max\_q=3,

information\_criterion="aic")

The python’s pmdarima package has a special function, which automatically finds the best parameters for an ARIMA(p,d,q) model, based on the AIC. Here is the resulting model:



As we can see, the best selected model was ARIMA(3,1,3), with the **coef** column, containing the coefficients for the model itself.

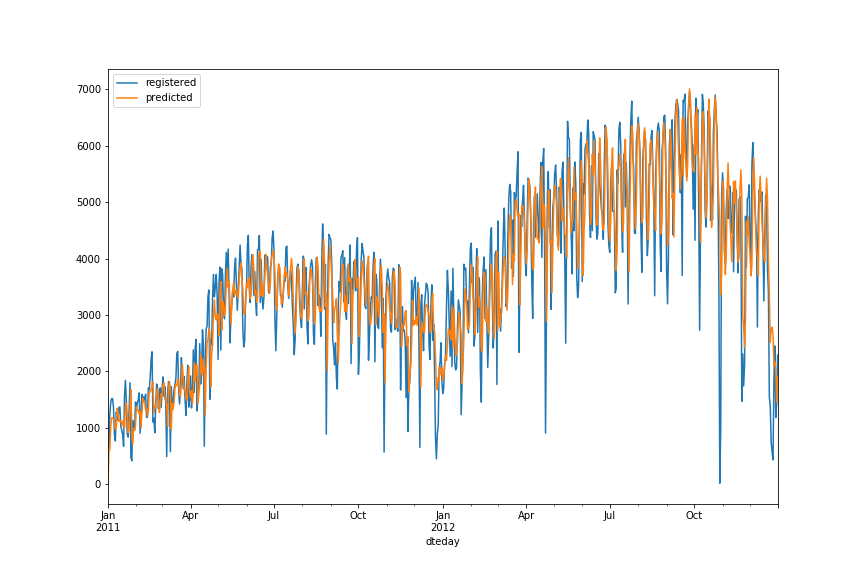
Finally, we can evaluate how well the number of rides are approximated by the model, by using the model.predict\_in\_sample() function:

# plot original and predicted values

plot\_data = pd.DataFrame(registered)

plot\_data['predicted'] = model.predict\_in\_sample()

plot\_data.plot(figsize=(12, 8))



As we can see, the **predicted** column follows the original one quite well, although unable to model correctly large up and downside movements in the **registered** series.

Summary

In this chapter, we studied a business problem, related to bike sharing services. We started with presenting some of the main visual techniques in data analysis, such as barplots, scatter plots and time series visualizations. WE also analyzed customers’ behavior, based on different time frames and weather conditions. We introduced the reader to hypothesis testing, and some of its main applications. Finally, we presented the basics of time series analysis, and how to identify the best time series models, when dealing with nonstationary time series.

Keywords:

bar plots, scatter plots, time series, hypothesis testing, p-value, Pearson correlation, Spearman correlation, stationarity, ARIMA

rivate