05 - Working with Categorical Data

May 25, 2021

1 05 - Working with Categorical Data

This notebook is based on an example from the book "Python Data Science Handbook".

In this notebook, we demonstrate how to work with categorical data (using the example model of linear regression).

The objective of the machine learning model is to predict the number of bicycle trips across Seattle's Fremont Bridge based on weather, weak day, and other factors.

2 Bicycle data

Prepare a variable for the shared data directory.

[1]:
```
from pathlib import Path
import os

data_dir = str(Path.home()) + r'/coursematerial/GIS/GeoDataScience'
```

[2]:
```
import pandas as pd
counts = pd.read_csv(os.path.join(data_dir, r'FremontBridge.csv'),␣
 ↪index_col='Date', parse_dates=True)

counts.head()
```

[2]:
```
                     Fremont Bridge Total  Fremont Bridge East Sidewalk  \
Date
2019-11-01 00:00:00                  12.0                           7.0
2019-11-01 01:00:00                   7.0                           0.0
2019-11-01 02:00:00                   1.0                           0.0
2019-11-01 03:00:00                   6.0                           6.0
2019-11-01 04:00:00                   6.0                           5.0

                     Fremont Bridge West Sidewalk
Date
2019-11-01 00:00:00                           5.0
2019-11-01 01:00:00                           7.0
```

```
2019-11-01 02:00:00                          1.0
2019-11-01 03:00:00                          0.0
2019-11-01 04:00:00                          1.0
```

The data originates from an automated bicycle counter that is installed on the Fremont Bridge in Seattle, which has inductive sensors on the east and west sidewalks of the bridge. The dataset contains time stamps (Date), the number of bicycles counted on the east sidewalk (Fremont Bridge East Sidewalk), the number of bicycles counted on the west sidewalk (Fremont Bridge West Sidewalk), and the total number (Fremont Bridge Total). As you can see from the first column, Date is the index of the DataFrame and it is of type **DatetimeIndex**.

Since we want to predict daily bicycle trips, but the bicycle counts are on an hourly basis, we aggregate the hourly data to daily data with the **resample()** method of the pandas class DataFrame. As first argument we provide **'d'** for daily and then the method **sum()** to sum up the values per day. You can think of **resample()** like a convenient group by as in SQL, which works on the time series index of the DataFrame, and which regards the specified time range (here days). For each group, the **sum()** operation is applied as an aggregate function.

[3]:
```
counts.resample('d').sum()
```

[3]:

Date	Fremont Bridge Total	Fremont Bridge East Sidewalk	Fremont Bridge West Sidewalk
2012-10-03	7042.0	3520.0	3522.0
2012-10-04	6950.0	3416.0	3534.0
2012-10-05	6296.0	3116.0	3180.0
2012-10-06	4012.0	2160.0	1852.0
2012-10-07	4284.0	2382.0	1902.0
…	…	…	…
2021-04-26	2411.0	929.0	1482.0
2021-04-27	2242.0	911.0	1331.0
2021-04-28	2382.0	974.0	1408.0
2021-04-29	2701.0	1159.0	1542.0
2021-04-30	1776.0	791.0	985.0

```
[3132 rows x 3 columns]
```

Since we only use the 'Fremont Bridge Totel' column, we extract it by indexing with the column name. We use double parenthesis to get a DataFrame, and not a Series object. In order to have an easier to remember column name, we rename the only column to 'Total'.

Since each bicycle is typically counted twice, on the east and the west sidewalk, we divide the number by two. (Although it seems that people are also not completely passing the bridge. But we ignore this phenomenon.)

[4]:
```
daily = counts.resample('d').sum()[['Fremont Bridge Total']]

daily /= 2.0

daily = daily.rename(columns={'Fremont Bridge Total' : 'Total'})

daily.head()
```

[4]:

Date	Total
2012-10-03	3521.0
2012-10-04	3475.0
2012-10-05	3148.0
2012-10-06	2006.0
2012-10-07	2142.0

Our assumption is that the number of bicycle trips are dependent on the day of the week and not on the date itself. Therefore, we get the day of the week with the **dayofweek** attribute from the DataFrame **index** (which as you remember is a DatetimeIndex).

[5]:
```
daily.index.dayofweek
```

[5]:
```
Int64Index([2, 3, 4, 5, 6, 0, 1, 2, 3, 4,
            …
            2, 3, 4, 5, 6, 0, 1, 2, 3, 4],
           dtype='int64', name='Date', length=3132)
```

So, what we have is categorical data with values 0 to 6 for the respective days of the week. We cannot just use these values as input, because it would not make any sense to multiply them with any weight. Day 4 would, e.g., go into the linear combination as double the value as day 2, and 6 even three times as day 2.

The solution is to make each category its own input feature, e.g. features named Monday, Tuesday, …, Sunday, and give each feature either the value 0 or 1, depending on the respective day of the week. For example, a date that corresponds to a Friday would result in values 0 (Monday), 0 (Tuesday), 0 (Wednesday), 0 (Thursday), 1 (Friday), 0 (Saturday), and 0 (Sunday). And a Tuesday in values 0, 1, 0, 0, 0, 0, 0. This is also called a one-hot-encoding.

In a linear model, the 7 values are multiplied by 7 weights that are learned from the training data. But since 6 features out of the 7 always have the value 0, only 1 feature (the actual day of the

week) is actually used. The effect is that for each day of the week, an individual weight is learned, and only this weight is used for the data with the same day of the week.

In the following, we store the values 1 or 0 in the columns with the name 'days', if the day of the week for this date is equal to this day.

[6]:
```
days = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
for i in range(7):
    daily[days[i]] = (daily.index.dayofweek == i).astype(float)

daily.head(10)
```

[6]:

Date	Total	Mon	Tue	Wed	Thu	Fri	Sat	Sun
2012-10-03	3521.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
2012-10-04	3475.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
2012-10-05	3148.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0
2012-10-06	2006.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
2012-10-07	2142.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
2012-10-08	3537.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
2012-10-09	3501.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
2012-10-10	3235.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
2012-10-11	3047.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
2012-10-12	2011.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0

Now, the algorithm can learn from all Monday data records the weight that is multiplied by Mon, and from all Tuesday data records the weight that is multiplied by Tue, etc. Of course, also taking into consideration the other input features like the weather. **This way to encode categorical data in one-hot-encoding is a very important concept in machine learning.**

We also assume that the number of bicycle trips are different, if the day is a holiday or not. We can use the holiday data from the pandas **USFederalHolidayCalender** class. We therefore construct an object of this class, and extract a holiday index (of type DatetimeIndex) with the **holidays()** method that takes the start and end date.

[7]:
```
from pandas.tseries.holiday import USFederalHolidayCalendar

cal = USFederalHolidayCalendar()

holidays = cal.holidays('2012', '2022')

print(holidays)
```

```
DatetimeIndex(['2012-01-02', '2012-01-16', '2012-02-20', '2012-05-28',
               '2012-07-04', '2012-09-03', '2012-10-08', '2012-11-12',
               '2012-11-22', '2012-12-25',
               …
               '2021-01-18', '2021-02-15', '2021-05-31', '2021-07-05',
               '2021-09-06', '2021-10-11', '2021-11-11', '2021-11-25',
```

```
               '2021-12-24', '2021-12-31'],
              dtype='datetime64[ns]', length=101, freq=None)
```

Using this Index, we can construct a Series object with one column named 'holiday' that always contains the value 1. The Series contains the date for each holiday as index, and the value 1 (because all days in this Series object is a holiday).

[8]:
```
holidays_series = pd.Series(1, index=holidays, name='Holiday')

holidays_series.head()
```

[8]:
```
2012-01-02    1
2012-01-16    1
2012-02-20    1
2012-05-28    1
2012-07-04    1
Name: Holiday, dtype: int64
```

We join the daily DataFrame with the holidays_series object (like in SQL), which is possible since both the DataFrame object and the Series objects have an index of dates.

[9]:
```
daily = daily.join(holidays_series)

daily.head()
```

[9]:

Date	Total	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Holiday
2012-10-03	3521.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	NaN
2012-10-04	3475.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	NaN
2012-10-05	3148.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	NaN
2012-10-06	2006.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	NaN
2012-10-07	2142.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	NaN

The data records with the same date index are joined. But as you can see, there are quite a lot of NaN values, where there are data records in daily that could not be joined with data records of the holidays series object. We therefore fill all nan values with the method **fillna()** with 0. The parameter **inplace** determines if the DataFrame object is directly changed or first copied, the copy changed, and returned by the function. Here, we want to fill the values in place.

[10]:
```
daily['Holiday'].fillna(0, inplace=True)

daily.head()
```

[10]:

Date	Total	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Holiday
2012-10-03	3521.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
2012-10-04	3475.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
2012-10-05	3148.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
2012-10-06	2006.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0

```
2012-10-07  2142.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0      0.0
```

Now, we have an input feature for holidays, which the algorithm can learn to multiply with another parameter to increase (or decrease) the number of bicycle trips for a holiday in comparison to a non-holiday. Since there are too few holidays, we do not differentiate if the holiday is a Monday, Tuesday, etc.

As with the day of day features, also the Holiday feature is of type categorical data.

Our third assumption is that the hours of daylight would affect how many people ride their bicycle. The following code uses some standard astronomical calculation to compute the hours of daylight for a given date and latitude value. And the results are stored with the daily DataFrame.

[11]:
```
import numpy as np
from datetime import datetime

def hours_of_daylight(date, axis=23.44, latitude=47.61):
    """Compute the hours of daylight for the given date"""
    days = (date - datetime(2000, 12, 21)).days
    m = (1. - np.tan(np.radians(latitude))
         * np.tan(np.radians(axis) * np.cos(days * 2 * np.pi / 365.25)))
    return 24. * np.degrees(np.arccos(1 - np.clip(m, 0, 2))) / 180.

daily['Daylight_hrs'] = list(map(hours_of_daylight, daily.index))

daily.head()
```

[11]:

Date	Total	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Holiday	Daylight_hrs
2012-10-03	3521.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	11.277359
2012-10-04	3475.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	11.219142
2012-10-05	3148.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	11.161038
2012-10-06	2006.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	11.103056
2012-10-07	2142.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	11.045208

Let us plot the daylight hours stored in the DataFrame.

[12]:
```
daily.tail()
```

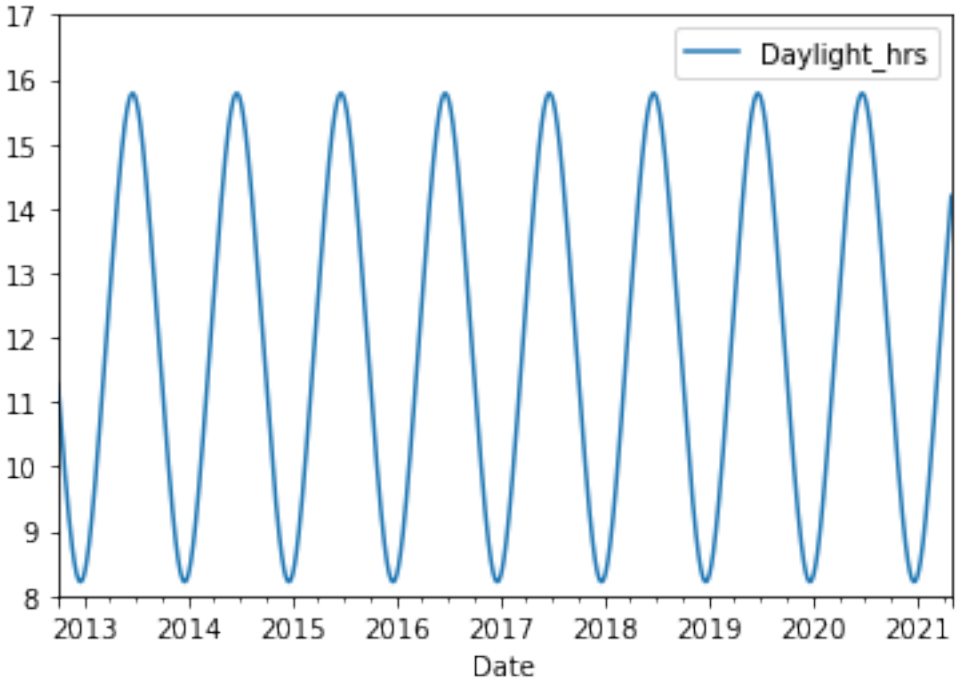
[12]:

Date	Total	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Holiday	Daylight_hrs
2021-04-26	1205.5	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	13.980697
2021-04-27	1121.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	14.033671
2021-04-28	1191.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	14.086250
2021-04-29	1350.5	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	14.138418
2021-04-30	888.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	14.190158

[13]:
```
%matplotlib inline
import matplotlib.pyplot as plt

daily[['Daylight_hrs']].plot()
plt.ylim(8, 17)
```

[13]: (8.0, 17.0)



3 Weather data

The weather dataset is from the weather station close to Seattle Tacoma International Airport made available from NOAA and contains quite a number of weather information, but also a lot of invalid values (-9999).

[14]:
```
weather = pd.read_csv(os.path.join(data_dir, 'BicycleWeather.csv'),␣
↪index_col='DATE', parse_dates=True)

weather.head()
```

[14]:

DATE	STATION	STATION_NAME
2012-01-01	GHCND:USW00024233	SEATTLE TACOMA INTERNATIONAL AIRPORT WA US
2012-01-02	GHCND:USW00024233	SEATTLE TACOMA INTERNATIONAL AIRPORT WA US

```
2012-01-03  GHCND:USW00024233  SEATTLE TACOMA INTERNATIONAL AIRPORT WA US
2012-01-04  GHCND:USW00024233  SEATTLE TACOMA INTERNATIONAL AIRPORT WA US
2012-01-05  GHCND:USW00024233  SEATTLE TACOMA INTERNATIONAL AIRPORT WA US

            PRCP  SNWD  SNOW  TMAX  TMIN  AWND  WDF2  WDF5  …  WT17  WT05  \
DATE                                                          …
2012-01-01     0     0     0   128    50    47   100    90  … -9999 -9999
2012-01-02   109     0     0   106    28    45   180   200  … -9999 -9999
2012-01-03     8     0     0   117    72    23   180   170  … -9999 -9999
2012-01-04   203     0     0   122    56    47   180   190  … -9999 -9999
2012-01-05    13     0     0    89    28    61   200   220  … -9999 -9999

            WT02  WT22  WT04  WT13  WT16  WT08  WT18  WT03
DATE
2012-01-01 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999
2012-01-02 -9999 -9999 -9999     1     1 -9999 -9999 -9999
2012-01-03 -9999 -9999 -9999 -9999     1 -9999 -9999 -9999
2012-01-04 -9999 -9999 -9999     1     1 -9999 -9999 -9999
2012-01-05 -9999 -9999 -9999 -9999     1 -9999 -9999 -9999

[5 rows x 25 columns]
```

And also give out the last days to see at what day the data ends. We have to remember this when we join the data with the daily bicycle data.

[15]:
```
weather.tail()
```

[15]:
```
                     STATION                                STATION_NAME  \
DATE
2015-08-28  GHCND:USW00024233  SEATTLE TACOMA INTERNATIONAL AIRPORT WA US
2015-08-29  GHCND:USW00024233  SEATTLE TACOMA INTERNATIONAL AIRPORT WA US
2015-08-30  GHCND:USW00024233  SEATTLE TACOMA INTERNATIONAL AIRPORT WA US
2015-08-31  GHCND:USW00024233  SEATTLE TACOMA INTERNATIONAL AIRPORT WA US
2015-09-01  GHCND:USW00024233  SEATTLE TACOMA INTERNATIONAL AIRPORT WA US

            PRCP  SNWD  SNOW  TMAX  TMIN  AWND  WDF2  WDF5  …  WT17  WT05  \
DATE                                                          …
2015-08-28     5     0     0   233   156    26   230   240  … -9999 -9999
2015-08-29   325     0     0   222   133    58   210   210  … -9999 -9999
2015-08-30   102     0     0   200   128    47   200   200  … -9999 -9999
2015-08-31     0     0     0   189   161    58   210   210  … -9999 -9999
2015-09-01    58     0     0   194   139 -9999 -9999 -9999  … -9999 -9999

            WT02  WT22  WT04  WT13  WT16  WT08  WT18  WT03
DATE
2015-08-28 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999
2015-08-29 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999
```

```
2015-08-30 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999
2015-08-31 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999
2015-09-01 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999

[5 rows x 25 columns]
```

We will use the precipitation (PRCP), the minimum temperature (TMIN) and the maximum temperature (TMAX), and calculate the average temperature from the latter two.

Since the temperature is stored as integer values in 1/10 degree Celsius (meaning that 233 is 23.3°C), we convert the TMIN and TMAX to degree Celsius (as floating point data type), compute the average temperature, and store it in the DataFrame in the column 'Temp (C)'.

[16]:
```
weather['Temp (C)'] = 0.5 * (weather['TMIN'] / 10 + weather['TMAX'] / 10)

weather.head()
```

[16]:

DATE	STATION	STATION_NAME	PRCP	SNWD	SNOW	TMAX	TMIN	AWND	WDF2	WDF5	…	WT05	WT02	WT22	WT04	WT13	WT16	WT08	WT18	WT03	Temp (C)
2012-01-01	GHCND:USW00024233	SEATTLE TACOMA INTERNATIONAL AIRPORT WA US	0	0	0	128	50	47	100	90	…	-9999	-9999	-9999	-9999	-9999	-9999	-9999	-9999	-9999	8.90
2012-01-02	GHCND:USW00024233	SEATTLE TACOMA INTERNATIONAL AIRPORT WA US	109	0	0	106	28	45	180	200	…	-9999	-9999	-9999	-9999	1	1	-9999	-9999	-9999	6.70
2012-01-03	GHCND:USW00024233	SEATTLE TACOMA INTERNATIONAL AIRPORT WA US	8	0	0	117	72	23	180	170	…	-9999	-9999	-9999	-9999	-9999	1	-9999	-9999	-9999	9.45
2012-01-04	GHCND:USW00024233	SEATTLE TACOMA INTERNATIONAL AIRPORT WA US	203	0	0	122	56	47	180	190	…	-9999	-9999	-9999	-9999	1	1	-9999	-9999	-9999	8.90
2012-01-05	GHCND:USW00024233	SEATTLE TACOMA INTERNATIONAL AIRPORT WA US	13	0	0	89	28	61	200	220	…	-9999	-9999	-9999	-9999	-9999	1	-9999	-9999	-9999	5.85

[5 rows x 26 columns]

Precipitation is also stored in 1/10 mm, so we convert it directly in place to inches, so that we also have floating point values. (It is not really necessary for the machine learning task, but the example shows nicely how to perform and store calculations in place within a DataFrame.)

[17]:
```
weather['PRCP'] /= 254

weather.head()
```

[17]:
```
                     STATION                                 STATION_NAME  \
DATE
2012-01-01  GHCND:USW00024233  SEATTLE TACOMA INTERNATIONAL AIRPORT WA US
2012-01-02  GHCND:USW00024233  SEATTLE TACOMA INTERNATIONAL AIRPORT WA US
2012-01-03  GHCND:USW00024233  SEATTLE TACOMA INTERNATIONAL AIRPORT WA US
2012-01-04  GHCND:USW00024233  SEATTLE TACOMA INTERNATIONAL AIRPORT WA US
2012-01-05  GHCND:USW00024233  SEATTLE TACOMA INTERNATIONAL AIRPORT WA US

                PRCP  SNWD  SNOW  TMAX  TMIN  AWND  WDF2  WDF5  …  WT05  \
DATE                                                              …
2012-01-01  0.000000     0     0   128    50    47   100    90  … -9999
2012-01-02  0.429134     0     0   106    28    45   180   200  … -9999
2012-01-03  0.031496     0     0   117    72    23   180   170  … -9999
2012-01-04  0.799213     0     0   122    56    47   180   190  … -9999
2012-01-05  0.051181     0     0    89    28    61   200   220  … -9999

            WT02  WT22  WT04  WT13  WT16  WT08  WT18  WT03  Temp (C)
DATE
2012-01-01 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999      8.90
2012-01-02 -9999 -9999 -9999     1     1 -9999 -9999 -9999      6.70
2012-01-03 -9999 -9999 -9999 -9999     1 -9999 -9999 -9999      9.45
2012-01-04 -9999 -9999 -9999     1     1 -9999 -9999 -9999      8.90
2012-01-05 -9999 -9999 -9999 -9999     1 -9999 -9999 -9999      5.85

[5 rows x 26 columns]
```

And we also add another column that indicates, if it is a dry day without precipitation at all (PRCP == 0). The machine learning model has then a chance to use the precipitation value (as quantitative data) itself, but also the categorical information if there was precipitation at all.

[18]:
```
weather['Dry Day'] = (weather['PRCP'] == 0).astype(int)

weather.head()
```

[18]:
```
                     STATION                                 STATION_NAME  \
DATE
2012-01-01  GHCND:USW00024233  SEATTLE TACOMA INTERNATIONAL AIRPORT WA US
2012-01-02  GHCND:USW00024233  SEATTLE TACOMA INTERNATIONAL AIRPORT WA US
2012-01-03  GHCND:USW00024233  SEATTLE TACOMA INTERNATIONAL AIRPORT WA US
2012-01-04  GHCND:USW00024233  SEATTLE TACOMA INTERNATIONAL AIRPORT WA US
2012-01-05  GHCND:USW00024233  SEATTLE TACOMA INTERNATIONAL AIRPORT WA US

                PRCP  SNWD  SNOW  TMAX  TMIN  AWND  WDF2  WDF5  …  WT02  \
```

```
DATE                                                                  …
2012-01-01  0.000000     0     0   128    50    47   100    90  … -9999
2012-01-02  0.429134     0     0   106    28    45   180   200  … -9999
2012-01-03  0.031496     0     0   117    72    23   180   170  … -9999
2012-01-04  0.799213     0     0   122    56    47   180   190  … -9999
2012-01-05  0.051181     0     0    89    28    61   200   220  … -9999

            WT22  WT04  WT13  WT16  WT08  WT18  WT03  Temp (C)  Dry Day
DATE
2012-01-01 -9999 -9999 -9999 -9999 -9999 -9999 -9999      8.90        1
2012-01-02 -9999 -9999     1     1 -9999 -9999 -9999      6.70        0
2012-01-03 -9999 -9999 -9999     1 -9999 -9999 -9999      9.45        0
2012-01-04 -9999 -9999     1     1 -9999 -9999 -9999      8.90        0
2012-01-05 -9999 -9999 -9999     1 -9999 -9999 -9999      5.85        0

[5 rows x 27 columns]
```

4 Join bicycle and weather data

Both datasets (DataFrames) contain an index based on dates, which allows us to join the two DataFrames. First, we check the range of both datasets.

[19]:
```
print('Bicyle: ', daily.index.min(), '---', daily.index.max())
print('Weather:', weather.index.min(), '---', weather.index.max())
```

```
Bicyle:  2012-10-03 00:00:00 --- 2021-04-30 00:00:00
Weather: 2012-01-01 00:00:00 --- 2015-09-01 00:00:00
```

Because the date range of the bicycle data goes beyond the weather data, we have to slice the bicycle data to end at the last day for which we have weather data. Otherwise we have bicycle data records with NaN values for the weather.

(It does not matter that the date range of the weather data starts earlier, because we use the bicycle data as the base data to join the weather with. Therefore, all data records of bicycle are contained in the resulting DataFrame, but not all from the weather data. It is kind of like a LEFT OUTER JOIN in SQL.)

[20]:
```
daily_sliced = daily.loc[:'2015-9-01']

print('Bicyle: ', daily_sliced.index.min(), '---', daily_sliced.index.max())
```

```
Bicyle:  2012-10-03 00:00:00 --- 2015-09-01 00:00:00
```

Now we can join the two DataFrames, adding precipitation, average temperature, and the dry day information to the bicycle DataFrame.

[21]:
```
bicycle = daily_sliced.join(weather[['PRCP', 'Temp (C)', 'Dry Day']])
bicycle.head()
```

[21]:
```
             Total  Mon  Tue  Wed  Thu  Fri  Sat  Sun  Holiday  Daylight_hrs  \
Date
2012-10-03  3521.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0      0.0     11.277359
2012-10-04  3475.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0      0.0     11.219142
2012-10-05  3148.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0      0.0     11.161038
2012-10-06  2006.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0      0.0     11.103056
2012-10-07  2142.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0      0.0     11.045208

            PRCP  Temp (C)  Dry Day
Date
2012-10-03   0.0     13.35        1
2012-10-04   0.0     13.60        1
2012-10-05   0.0     15.30        1
2012-10-06   0.0     15.85        1
2012-10-07   0.0     15.85        1
```

[22]:
```
bicycle.tail()
```

[22]:
```
             Total  Mon  Tue  Wed  Thu  Fri  Sat  Sun  Holiday  Daylight_hrs  \
Date
2015-08-28  2653.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0      0.0     13.418591
2015-08-29   699.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0      0.0     13.362212
2015-08-30  1213.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0      0.0     13.305611
2015-08-31  2823.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0      0.0     13.248802
2015-09-01  2876.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0      0.0     13.191795

                PRCP  Temp (C)  Dry Day
Date
2015-08-28  0.019685     19.45        0
2015-08-29  1.279528     17.75        0
2015-08-30  0.401575     16.40        0
2015-08-31  0.000000     17.50        1
2015-09-01  0.228346     16.65        0
```

And finally, we add another (floating point) column that gives a percentage of the days passed per year starting with 0.0 at day 1, increasing by 1/365 per day. This allows us to measure any observed annual increase or decrease in daily crossings.

[23]:
```
bicycle['Year'] = (bicycle.index - bicycle.index[0]).days / 365.

bicycle.head()
```

[23]:
```
             Total  Mon  Tue  Wed  Thu  Fri  Sat  Sun  Holiday  Daylight_hrs  \
Date
2012-10-03  3521.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0      0.0     11.277359
2012-10-04  3475.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0      0.0     11.219142
2012-10-05  3148.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0      0.0     11.161038
```

```
2012-10-06  2006.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0      0.0     11.103056
2012-10-07  2142.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0      0.0     11.045208

            PRCP  Temp (C)  Dry Day      Year
Date
2012-10-03   0.0     13.35        1  0.000000
2012-10-04   0.0     13.60        1  0.002740
2012-10-05   0.0     15.30        1  0.005479
2012-10-06   0.0     15.85        1  0.008219
2012-10-07   0.0     15.85        1  0.010959
```

[24]:
```
bicycle.tail()
```

[24]:
```
             Total  Mon  Tue  Wed  Thu  Fri  Sat  Sun  Holiday  Daylight_hrs  \
Date
2015-08-28  2653.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0      0.0     13.418591
2015-08-29   699.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0      0.0     13.362212
2015-08-30  1213.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0      0.0     13.305611
2015-08-31  2823.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0      0.0     13.248802
2015-09-01  2876.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0      0.0     13.191795

                PRCP  Temp (C)  Dry Day      Year
Date
2015-08-28  0.019685     19.45        0  2.901370
2015-08-29  1.279528     17.75        0  2.904110
2015-08-30  0.401575     16.40        0  2.906849
2015-08-31  0.000000     17.50        1  2.909589
2015-09-01  0.228346     16.65        0  2.912329
```

To clean up any potential missing data, we drop all rows with NaN values (if there are any).

[25]:
```
print(bicycle.shape)

bicycle.dropna(axis=0, how='any', inplace=True)

print(bicycle.shape)
```

```
(1064, 14)
(1064, 14)
```

Extract input feature columns as X, and target column as y.

[26]:
```
input_column_names = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun',␣
 ↪'Holiday',
                      'Daylight_hrs', 'PRCP', 'Dry Day', 'Temp (C)', 'Year']
X = bicycle[input_column_names]
y = bicycle['Total']
```

5 Train a model and make predictions

Construct and fit a linear regression model, and get the score with the training data. (We do not fit the intercept term by specifying **fit_intercept** as false, because the day per week (Mon, Tue, etc.), which are 1 for exactly one column, will work like a day-specific intercept. It is multiplied by a weight, which is the bias term for each specific day. Remember that the bias is often regarded as one weight that is multiplied with an input feature of value 1. Exactly like we have in this case, but with a bias per day. You can try out to set this parameter to true and observe the change in the model parameters that are printed later.)

Please note: It is not good practice to evaluate a model on the training data. Typically, we should have some test data reserved that the model has not seen. But for the sake of the exercise, and the fact that we do not have that much data to play around with, we use the same data for learning, evaluation, and prediction.

[27]:
```
from sklearn.linear_model import LinearRegression

model = LinearRegression(fit_intercept=False)
model.fit(X, y)

model.score(X, y)
```

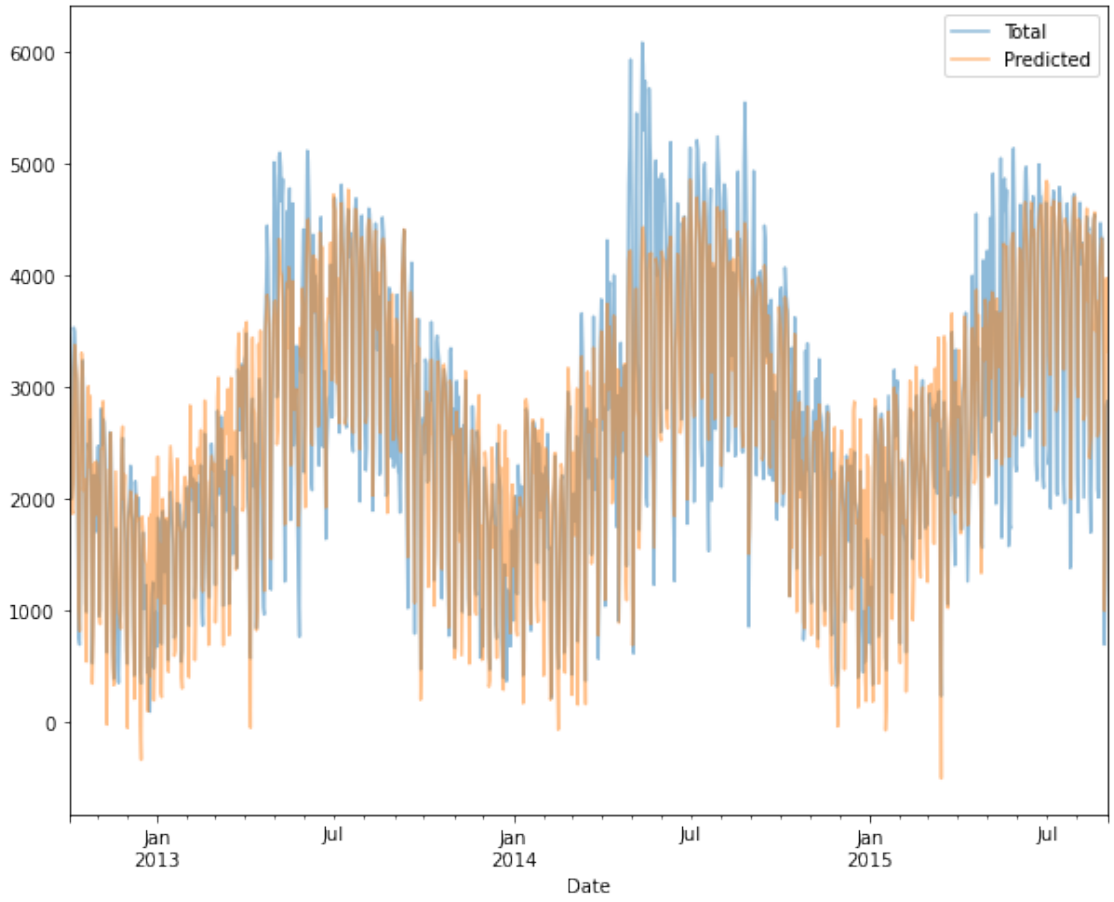
[27]:
```
0.8675358719950574
```

Perform predictions on the same training data and store it in the DataFrame itself.

[28]:
```
bicycle['Predicted'] = model.predict(X)
```

Plot the total (ground truth) and the predicted bicycle trips per day with the plot function of the DataFrame. The parameter **alpha** with the value 0.5 determines that the lines are semi-transparent, so we can better see where the two lines overlap.

[29]:
```
bicycle[['Total', 'Predicted']].plot(figsize=(10, 8), alpha=0.5);
```



It is interesting that the predicted curve follows nicely the seasonal changes, but it is also evident that the predicted values are at times quite far off from the true values (especially in the summer months). Either our features are not complete (i.e., people decide whether to ride are based on more than just these) or there are some nonlinear relationships that we have failed to take into account (e.g., people might ride less at both high and low temperatures).

The rough approximation is sufficient to give us some insights, and we can take a look at the coefficients of the linear model to estimate how much each feature contributes to the daily bicycle count. We therefore extract the parameters from the model, construct a Series object from it for a nicer output, with an index of the columns of the input feature vector X.

[30]:
```
params = pd.Series(model.coef_, index=X.columns)
params
```

[30]:
```
Mon            504.882756
Tue            610.233936
Wed            592.673642
Thu            482.358115
Fri            177.980345
```

```
Sat            -1103.301710
Sun            -1133.567246
Holiday        -1187.401381
Daylight_hrs     128.851511
PRCP            -664.834882
Dry Day          547.698592
Temp (C)          65.162791
Year              26.942713
dtype: float64
```

If we try to interpret these coefficients, then one insight is that the Fremont bridge is mostly used by people going to work. This can be clearly seen by the lower values of bicycle trips towards the end of the week and the negative values on weekends and holidays. Precipitation has a clear negative influence, and the daylight hours probably accounts for the seasonal changes. Looking at the coefficient for year, we could assume that there is a positive trend in the number of people using a bicycle to work.

But how about the uncertainties? To answer this, we evaluate the coefficients using bootstrap re-samplings of the data. (In bootstrap re-sampling, we change the order of the data records and apply standard deviation in this case.)

[31]:
```
from sklearn.utils import resample

np.random.seed(1)

err = np.std([model.fit(*resample(X, y)).coef_ for i in range(1000)], 0)
```

Print out the coefficients (the effect on the model) and the error (standard deviation from the repeated re-sampling and fitting.).

[32]:
```
print(pd.DataFrame({'effect': params.round(0),
                    'error': err.round(0)}))
```

```
              effect  error
Mon            505.0   86.0
Tue            610.0   83.0
Wed            593.0   83.0
Thu            482.0   85.0
Fri            178.0   81.0
Sat          -1103.0   80.0
Sun          -1134.0   83.0
Holiday      -1187.0  163.0
Daylight_hrs   129.0    9.0
PRCP          -665.0   62.0
Dry Day        548.0   33.0
Temp (C)        65.0    4.0
Year            27.0   18.0
```

Besides the observations from above, we can now also see the error or variation of each coefficient.

For example, 129 ± 9 people choose to ride per additional hour of daylight, while the variability of a Friday is much higher with $(178) \pm 81$. Of course, there are many more features to think of that might lead to a better model for predicting bicycle trips across the Fremont bridge.

[ ]: