

# Exploratory data analysis

Fraida Fund

## In this notebook

In this notebook:

- We practice using `pandas` to read in and manipulate a data set
- We learn a basic “recipe” for exploratory data analysis and apply it to an example

## Introduction

The first step in applying machine learning to a real problem is *finding* or *creating* an appropriate data set with which to train your model.

### What makes data “good”?

What makes a good data set?

- **Size:** the more *samples* are in the data set, the more examples your machine learning model will be able to learn from, and the better it will do. Often, a simple machine learning model trained on a large data set will outperform a “fancy” model on a small data set.
- **Quality:** Are there *predictive* features in the data? Are no values (or very few values) missing, noisy, or incorrect? Is the scenario in which the data collected similar to the scenario in which your model will be used? These are examples of questions that we might ask to evaluate the quality of a data set.

One of the most important principles in machine learning is: **garbage in, garbage out**. If the data you use to train a machine learning model is problematic, or not well suited for the purpose, then even the best model will produce useless predictions.

### Purpose of exploratory data analysis

Once we have identified one or more candidate data sets for a particular problem, we perform some *exploratory data analysis*. This process helps us

- detect and possibly correct mistakes in the data
- check our assumptions about the data
- identify potential relationships between features
- assess the direction and rough size of relationships between features and the target variable

Exploratory data analysis is important for understanding whether this data set is appropriate for the machine learning task at hand, and if any extra cleaning or processing steps are required before we use the data.

### “Recipe” for exploratory data analysis

We will practice using a basic “recipe” for exploratory data analysis.

1. Learn about your data
2. Load data and check that it is loaded correctly
3. Visually inspect the data
4. Compute summary statistics
5. Explore the data further and look for potential issues

Every exploratory data analysis is different, as specific characteristics of the data may lead you to explore different things in depth. However, this “recipe” can be a helpful starting point.

### Example: Brooklyn Bridge pedestrian data set

The Brooklyn Bridge is a bridge that connects Brooklyn and Manhattan. It supports vehicles, pedestrians, and bikers.



Support you are developing a machine learning model to predict the volume of pedestrian traffic on the Brooklyn Bridge. There is a dataset available that you think may be useful as training data: [Brooklyn Bridge Automated Pedestrian Counts dataset](#), from the NYC Department of Transportation.

We will practice applying the “recipe” for exploratory data analysis to this data.

We will use the `pandas` library in Python, which includes many powerful utilities for managing data. You can refer to the [pandas reference](#) for more details on the `pandas` functions used in this notebook.

### Learn about your data

The first step is to learn more about the data:

- Read about *methodology* and *data codebook*
- How many rows and columns are in the data?
- What does each variable mean? What units are data recorded in?
- How was data collected? Identify sampling issues, timeliness issues, fairness issues, etc.

This is the step that is most often skipped, but it is by far the most essential!

For the Brooklyn Bridge dataset, you can review the associated documentation on the NYC Data website:

- [NYC Data Website](#)

- Data dictionary

## Load data and check that it is loaded correctly

The next step is to load the data in preparation for your exploratory data analysis.

First, we will import some useful libraries:

- In Python - libraries add powerful functionality
- You can import an entire library (`import foo`) or part (`from foo import bar`)
- You can define a nickname, which you will use to call functions of these libraries (many libraries have “conventional” nicknames)

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# set up notebook to show all outputs, not only last one
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

Now we are ready to read in our data!

Our data is in CSV format, so will use the `read_csv` function in pandas to read in our data.

Function documentation: [pandas reference](#)

```
pandas.read_csv(filepath_or_buffer,
                 sep=',', header='infer',
                 names=None,
                 ...)
```

`read_csv` is for “flat” text files, where each data point is on another row, and the fields in a row are separated by some delimiter (e.g. comma). Other pandas functions exist for loading other kinds of data (read from database, Excel file, etc.)

```
url = 'https://data.cityofnewyork.us/api/views/6fi9-q3ta/rows.csv?accessType=DOWNLOAD'
df = pd.read_csv(url)
```

We will want to verify that the data was loaded correctly. For *tabular* data, we can start by looking at a few rows of data with the `head` function. (For data that is not tabular, such as image, text, or audio data, we might start by looking at a few random samples instead.)

```
df.head()
```

	hour_beginning	location	Pedestrians	Towards Manhattan	\
0	04/30/2019 12:00:00 AM	Brooklyn Bridge	3	3	
1	12/31/2019 10:00:00 PM	Brooklyn Bridge	10	9	
2	12/31/2019 11:00:00 PM	Brooklyn Bridge	2	0	
3	12/31/2019 09:00:00 PM	Brooklyn Bridge	12	0	
4	04/01/2019 03:00:00 AM	Brooklyn Bridge	1	0	

	Towards Brooklyn	weather_summary	temperature	precipitation	lat	\
0	0	NaN	NaN	NaN	40.708164	
1	1	cloudy	42.0	0.0005	40.708164	
2	2	cloudy	42.0	0.0004	40.708164	

3	12	cloudy	42.0	0.0036	40.708164
4	1	clear-night	36.0	0.0000	40.708164

	long	events	Location1
0	-73.999509	NaN	(40.7081639691088, -73.9995087014816)
1	-73.999509	NaN	(40.7081639691088, -73.9995087014816)
2	-73.999509	NaN	(40.7081639691088, -73.9995087014816)
3	-73.999509	NaN	(40.7081639691088, -73.9995087014816)
4	-73.999509	NaN	(40.7081639691088, -73.9995087014816)

One thing to look for in the output above, that is easily missed: verify that column names and row names are loaded correctly, and that the first row of real data is actually data, and not column labels.

We should also check the shape of the data frame - the number of rows and columns. This, too, should be checked against our assumptions about the data from the NYC Data website.

```
df.shape
```

```
(16057, 12)
```

Check the names of the columns and their data types:

```
df.columns
df.dtypes
```

```
Index(['hour_beginning', 'location', 'Pedestrians', 'Towards Manhattan',
      'Towards Brooklyn', 'weather_summary', 'temperature', 'precipitation',
      'lat', 'long', 'events', 'Location1'],
      dtype='object')
```

```
hour_beginning    object
location          object
Pedestrians       int64
Towards Manhattan int64
Towards Brooklyn int64
weather_summary   object
temperature       float64
precipitation     float64
lat               float64
long              float64
events            object
Location1         object
dtype: object
```

We can also get a quick summary with `info()`;

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 16057 entries, 0 to 16056
Data columns (total 12 columns):
hour_beginning    16057 non-null object
location          16057 non-null object
```

```

Pedestrians      16057 non-null int64
Towards Manhattan 16057 non-null int64
Towards Brooklyn 16057 non-null int64
weather_summary  16041 non-null object
temperature      16041 non-null float64
precipitation    16041 non-null float64
lat              16057 non-null float64
long             16057 non-null float64
events           1124 non-null object
Location1        16057 non-null object
dtypes: float64(4), int64(3), object(5)
memory usage: 1.5+ MB

```

pandas infers the data type of each column automatically from the contents of the data.

If the data type of a column is not what you expect it to be, this can often be a signal that the data needs cleaning. For example, if you expect a column to be numeric and it is read in as non-numeric, this indicates that there are probably some samples that include a non-numeric value in that column. (The [NYC Data website](#) indicates what type of data *should* be in each column, so you should reference that when checking this output. )

We have a date/time column that was read in as a string, so we can correct that now:

```

df['hour_beginning'] = pd.to_datetime(df['hour_beginning'])
df.info()

```

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 16057 entries, 0 to 16056
Data columns (total 12 columns):
hour_beginning      16057 non-null datetime64[ns]
location            16057 non-null object
Pedestrians         16057 non-null int64
Towards Manhattan   16057 non-null int64
Towards Brooklyn    16057 non-null int64
weather_summary     16041 non-null object
temperature         16041 non-null float64
precipitation       16041 non-null float64
lat                 16057 non-null float64
long                16057 non-null float64
events              1124 non-null object
Location1           16057 non-null object
dtypes: datetime64[ns](1), float64(4), int64(3), object(4)
memory usage: 1.5+ MB

```

And once we have done that, we can order the data frame by time:

```

df = df.sort_values(by='hour_beginning')
df.head()

```

	hour_beginning	location	Pedestrians	Towards Manhattan	\
8846	2017-10-01 00:00:00	Brooklyn Bridge	44		30
9473	2017-10-01 01:00:00	Brooklyn Bridge	30		17
10098	2017-10-01 02:00:00	Brooklyn Bridge	25		13
10733	2017-10-01 03:00:00	Brooklyn Bridge	20		11
11527	2017-10-01 04:00:00	Brooklyn Bridge	18		10

	Towards Brooklyn		weather_summary	temperature	precipitation \
8846	14		clear-night	52.0	0.0001
9473	13		partly-cloudy-night	53.0	0.0002
10098	12		partly-cloudy-night	52.0	0.0000
10733	9		partly-cloudy-night	51.0	0.0000
11527	8		partly-cloudy-night	51.0	0.0000

	lat	long	events	Location1
8846	40.708164	-73.999509	NaN	(40.7081639691088, -73.9995087014816)
9473	40.708164	-73.999509	NaN	(40.7081639691088, -73.9995087014816)
10098	40.708164	-73.999509	NaN	(40.7081639691088, -73.9995087014816)
10733	40.708164	-73.999509	NaN	(40.7081639691088, -73.9995087014816)
11527	40.708164	-73.999509	NaN	(40.7081639691088, -73.9995087014816)

You may notice that the `hour_beginning` variable includes the full date and time in one field. For our analysis, it would be more useful to have separate fields for the date, month, day of the week, and hour.

We can create these additional fields by assigning the desired value to them directly - then, observe the effect:

```
df['hour'] = df['hour_beginning'].dt.hour
df['month'] = df['hour_beginning'].dt.month
df['date'] = df['hour_beginning'].dt.date
df['day_name'] = df['hour_beginning'].dt.day_name()
```

```
df.head()
```

	hour_beginning	location	Pedestrians	Towards Manhattan \
8846	2017-10-01 00:00:00	Brooklyn Bridge	44	30
9473	2017-10-01 01:00:00	Brooklyn Bridge	30	17
10098	2017-10-01 02:00:00	Brooklyn Bridge	25	13
10733	2017-10-01 03:00:00	Brooklyn Bridge	20	11
11527	2017-10-01 04:00:00	Brooklyn Bridge	18	10

	Towards Brooklyn		weather_summary	temperature	precipitation \
8846	14		clear-night	52.0	0.0001
9473	13		partly-cloudy-night	53.0	0.0002
10098	12		partly-cloudy-night	52.0	0.0000
10733	9		partly-cloudy-night	51.0	0.0000
11527	8		partly-cloudy-night	51.0	0.0000

	lat	long	events	Location1 \
8846	40.708164	-73.999509	NaN	(40.7081639691088, -73.9995087014816)
9473	40.708164	-73.999509	NaN	(40.7081639691088, -73.9995087014816)
10098	40.708164	-73.999509	NaN	(40.7081639691088, -73.9995087014816)
10733	40.708164	-73.999509	NaN	(40.7081639691088, -73.9995087014816)
11527	40.708164	-73.999509	NaN	(40.7081639691088, -73.9995087014816)

	hour	month	date	day_name
8846	0	10	2017-10-01	Sunday
9473	1	10	2017-10-01	Sunday
10098	2	10	2017-10-01	Sunday
10733	3	10	2017-10-01	Sunday

```
11527      4      10  2017-10-01  Sunday
```

For data that is recorded at regular time intervals, it is also important to know whether the data is complete, or whether there are gaps in time. We will use some helpful pandas functions:

- `pd.to_datetime`
- `pd.date_range`

First, we will use `date_range` to get the list of hour intervals that we expect to find in the dataset. Then, we will find the difference between this list and the actual list of hour intervals in the dataset - these are missing intervals.

```
# get beginning and end of date range
min_dt = df.hour_beginning.min()
max_dt = df.hour_beginning.max()
print(min_dt)
print(max_dt)
```

```
2017-10-01 00:00:00
2019-12-31 23:00:00
```

```
# then identify the missing hours
expected_range = pd.date_range(start = min_dt, end = max_dt, freq='H' )
missing_hours = expected_range.difference(df['hour_beginning'])
print(missing_hours)
```

```
DatetimeIndex(['2018-08-01 00:00:00', '2018-08-01 01:00:00',
              '2018-08-01 02:00:00', '2018-08-01 03:00:00',
              '2018-08-01 04:00:00', '2018-08-01 05:00:00',
              '2018-08-01 06:00:00', '2018-08-01 07:00:00',
              '2018-08-01 08:00:00', '2018-08-01 09:00:00',
              ...,
              '2018-12-31 14:00:00', '2018-12-31 15:00:00',
              '2018-12-31 16:00:00', '2018-12-31 17:00:00',
              '2018-12-31 18:00:00', '2018-12-31 19:00:00',
              '2018-12-31 20:00:00', '2018-12-31 21:00:00',
              '2018-12-31 22:00:00', '2018-12-31 23:00:00'],
              dtype='datetime64[ns]', length=3672, freq=None)
```

We had the expected number of rows (the output of `shape` matched the description of the data on the NYC Data website), but the data seems to be missing samples from August 2018 through December 2018, which is worth keeping in mind if we decide to use it:

```
pd.unique(missing_hours.date)
```

```
array([datetime.date(2018, 8, 1), datetime.date(2018, 8, 2),
       datetime.date(2018, 8, 3), datetime.date(2018, 8, 4),
       datetime.date(2018, 8, 5), datetime.date(2018, 8, 6),
       datetime.date(2018, 8, 7), datetime.date(2018, 8, 8),
       datetime.date(2018, 8, 9), datetime.date(2018, 8, 10),
       datetime.date(2018, 8, 11), datetime.date(2018, 8, 12),
       datetime.date(2018, 8, 13), datetime.date(2018, 8, 14),
       datetime.date(2018, 8, 15), datetime.date(2018, 8, 16),
       datetime.date(2018, 8, 17), datetime.date(2018, 8, 18),
```





```
datetime.date(2018, 12, 5), datetime.date(2018, 12, 6),
datetime.date(2018, 12, 7), datetime.date(2018, 12, 8),
datetime.date(2018, 12, 9), datetime.date(2018, 12, 10),
datetime.date(2018, 12, 11), datetime.date(2018, 12, 12),
datetime.date(2018, 12, 13), datetime.date(2018, 12, 14),
datetime.date(2018, 12, 15), datetime.date(2018, 12, 16),
datetime.date(2018, 12, 17), datetime.date(2018, 12, 18),
datetime.date(2018, 12, 19), datetime.date(2018, 12, 20),
datetime.date(2018, 12, 21), datetime.date(2018, 12, 22),
datetime.date(2018, 12, 23), datetime.date(2018, 12, 24),
datetime.date(2018, 12, 25), datetime.date(2018, 12, 26),
datetime.date(2018, 12, 27), datetime.date(2018, 12, 28),
datetime.date(2018, 12, 29), datetime.date(2018, 12, 30),
datetime.date(2018, 12, 31)], dtype=object)
```

This is also a good time to look for rows that are missing data in some columns (“NA” values), that may need to be cleaned.

We can see the number of NAs in each column by summing up all the instances where the `isnull` function returns a True value:

```
df.isnull().sum()
```

```
hour_beginning      0
location            0
Pedestrians         0
Towards Manhattan  0
Towards Brooklyn   0
weather_summary     16
temperature         16
precipitation       16
lat                 0
long                0
events             14933
Location1           0
hour                0
month               0
date                0
day_name            0
dtype: int64
```

There are some rows of data that are missing weather, temperature, and precipitation data. We can see these rows with

```
df[df['temperature'].isnull()]
```

	hour_beginning	location	Pedestrians	Towards Manhattan	\
12271	2018-03-11 02:00:00	Brooklyn Bridge	0	0	
12796	2018-05-13 00:00:00	Brooklyn Bridge	98	69	
482	2019-01-06 00:00:00	Brooklyn Bridge	3	3	
2604	2019-01-09 00:00:00	Brooklyn Bridge	3	3	
2140	2019-01-14 00:00:00	Brooklyn Bridge	0	0	
3951	2019-01-16 00:00:00	Brooklyn Bridge	7	2	
5562	2019-02-02 00:00:00	Brooklyn Bridge	0	0	

7696	2019-03-05 00:00:00	Brooklyn Bridge	2	0
2944	2019-03-10 02:00:00	Brooklyn Bridge	0	0
0	2019-04-30 00:00:00	Brooklyn Bridge	3	3
4198	2019-05-02 00:00:00	Brooklyn Bridge	3	2
5962	2019-05-08 00:00:00	Brooklyn Bridge	3	2
5277	2019-06-17 01:00:00	Brooklyn Bridge	0	0
5934	2019-09-06 00:00:00	Brooklyn Bridge	2	1
5206	2019-09-17 00:00:00	Brooklyn Bridge	2	2
701	2019-11-03 01:00:00	Brooklyn Bridge	0	0

	Towards Brooklyn	weather_summary	temperature	precipitation \
12271	0	NaN	NaN	NaN
12796	29	NaN	NaN	NaN
482	0	NaN	NaN	NaN
2604	0	NaN	NaN	NaN
2140	0	NaN	NaN	NaN
3951	5	NaN	NaN	NaN
5562	0	NaN	NaN	NaN
7696	2	NaN	NaN	NaN
2944	0	NaN	NaN	NaN
0	0	NaN	NaN	NaN
4198	1	NaN	NaN	NaN
5962	1	NaN	NaN	NaN
5277	0	NaN	NaN	NaN
5934	1	NaN	NaN	NaN
5206	0	NaN	NaN	NaN
701	0	NaN	NaN	NaN

	lat	long	events \
12271	40.708164	-73.999509	Daylight Saving Time starts
12796	40.708164	-73.999509	Mother's Day
482	40.708164	-73.999509	NaN
2604	40.708164	-73.999509	NaN
2140	40.708164	-73.999509	NaN
3951	40.708164	-73.999509	NaN
5562	40.708164	-73.999509	NaN
7696	40.708164	-73.999509	NaN
2944	40.708164	-73.999509	Daylight Saving Time starts
0	40.708164	-73.999509	NaN
4198	40.708164	-73.999509	NaN
5962	40.708164	-73.999509	NaN
5277	40.708164	-73.999509	NaN
5934	40.708164	-73.999509	NaN
5206	40.708164	-73.999509	NaN
701	40.708164	-73.999509	Daylight Saving Time ends

	Location1	hour	month	date \
12271	(40.7081639691088, -73.9995087014816)	2	3	2018-03-11
12796	(40.7081639691088, -73.9995087014816)	0	5	2018-05-13
482	(40.7081639691088, -73.9995087014816)	0	1	2019-01-06
2604	(40.7081639691088, -73.9995087014816)	0	1	2019-01-09
2140	(40.7081639691088, -73.9995087014816)	0	1	2019-01-14
3951	(40.7081639691088, -73.9995087014816)	0	1	2019-01-16
5562	(40.7081639691088, -73.9995087014816)	0	2	2019-02-02

7696	(40.7081639691088, -73.9995087014816)	0	3	2019-03-05
2944	(40.7081639691088, -73.9995087014816)	2	3	2019-03-10
0	(40.7081639691088, -73.9995087014816)	0	4	2019-04-30
4198	(40.7081639691088, -73.9995087014816)	0	5	2019-05-02
5962	(40.7081639691088, -73.9995087014816)	0	5	2019-05-08
5277	(40.7081639691088, -73.9995087014816)	1	6	2019-06-17
5934	(40.7081639691088, -73.9995087014816)	0	9	2019-09-06
5206	(40.7081639691088, -73.9995087014816)	0	9	2019-09-17
701	(40.7081639691088, -73.9995087014816)	1	11	2019-11-03

	day_name
12271	Sunday
12796	Sunday
482	Sunday
2604	Wednesday
2140	Monday
3951	Wednesday
5562	Saturday
7696	Tuesday
2944	Sunday
0	Tuesday
4198	Thursday
5962	Wednesday
5277	Monday
5934	Friday
5206	Tuesday
701	Sunday

pandas includes routines to fill in missing data using the `fillna` function ([reference](#)). We will fill these using the “forward fill” method, which carries the last valid observation forward to fill in NAs.

(Note: this makes sense only because we already sorted by date, and it’s reasonable to expect adjacent hours to have similar weather!)

```
df['temperature'] = df['temperature'].fillna(method="ffill")
df['precipitation'] = df['precipitation'].fillna(method="ffill")
df['weather_summary'] = df['weather_summary'].fillna(method="ffill")
```

Now we can count the NAs again and find that there are only missing values in the `events` column. This is the expected result, since there are many days with no event.

```
df.isnull().sum()
```

hour_beginning	0
location	0
Pedestrians	0
Towards Manhattan	0
Towards Brooklyn	0
weather_summary	0
temperature	0
precipitation	0
lat	0
long	0
events	14933
Location1	0

```
hour          0
month         0
date          0
day_name      0
dtype: int64
```

## Visually inspect data

Now we are ready to visually inspect the data.

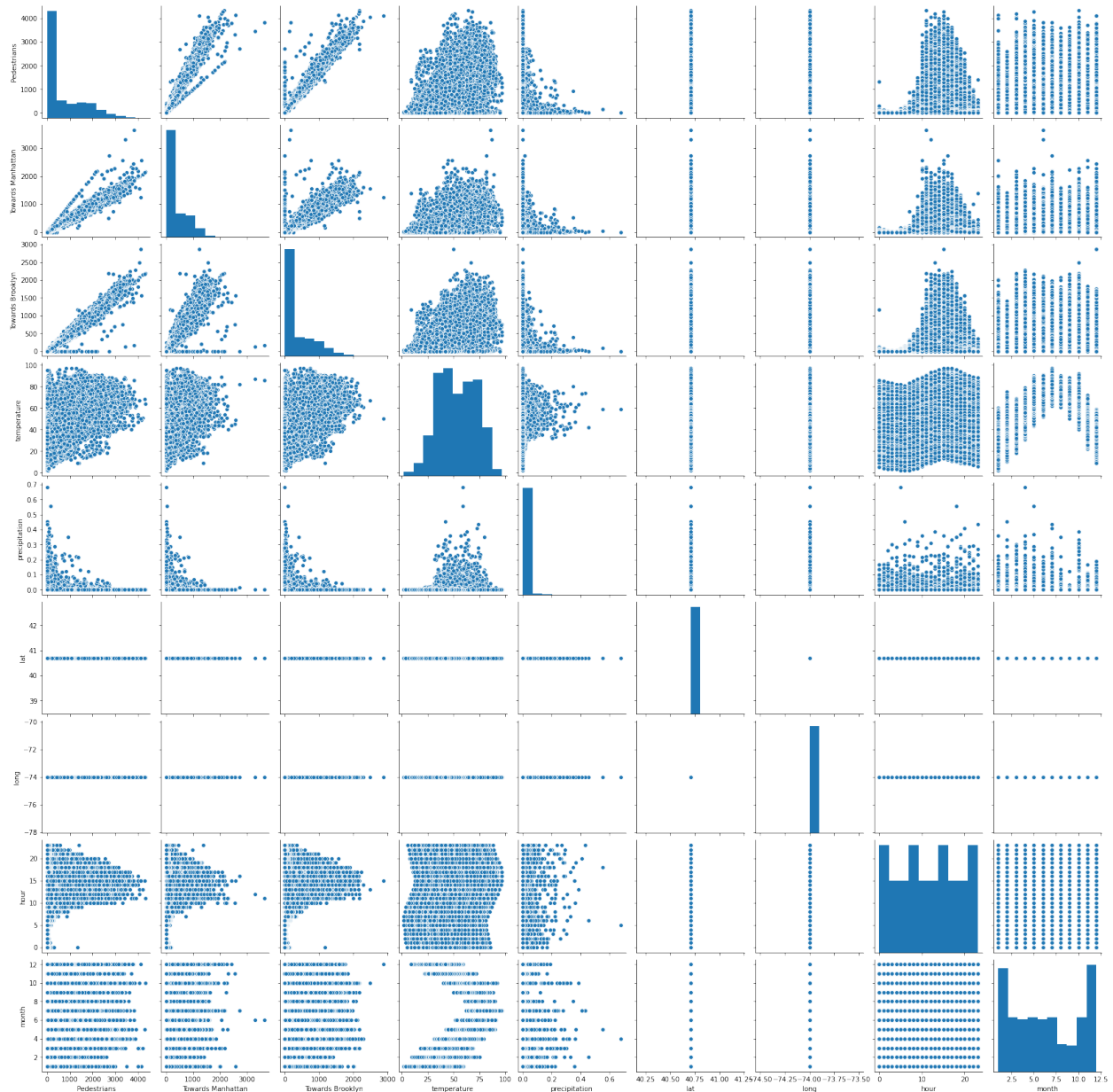
For tabular data, and especially tabular data with many numeric features, it is often useful to create a *pairplot*. A pairplot shows pairwise relationships between all numerical variables. It is a useful way to identify:

- features that are predictive - if there is any noticeable relationship between the target variable and any other variable.
- features that are correlated - if two features are highly correlated, we may be able to achieve equally good results just using one of them.

We can create a “default” pairplot with

```
sns.pairplot(df)
```

```
<seaborn.axisgrid.PairGrid at 0x7f63ffb0d0d0>
```



Here, each pane shows one numerical variable on the x-axis and another numerical variable on the y-axis, so that we can see if a relationship exists between them. The panes along the diagonal shows the empirical distribution of values for each feature in this data.

But, it is difficult to see anything useful because there is so much going on in this plot. We can improve things somewhat by:

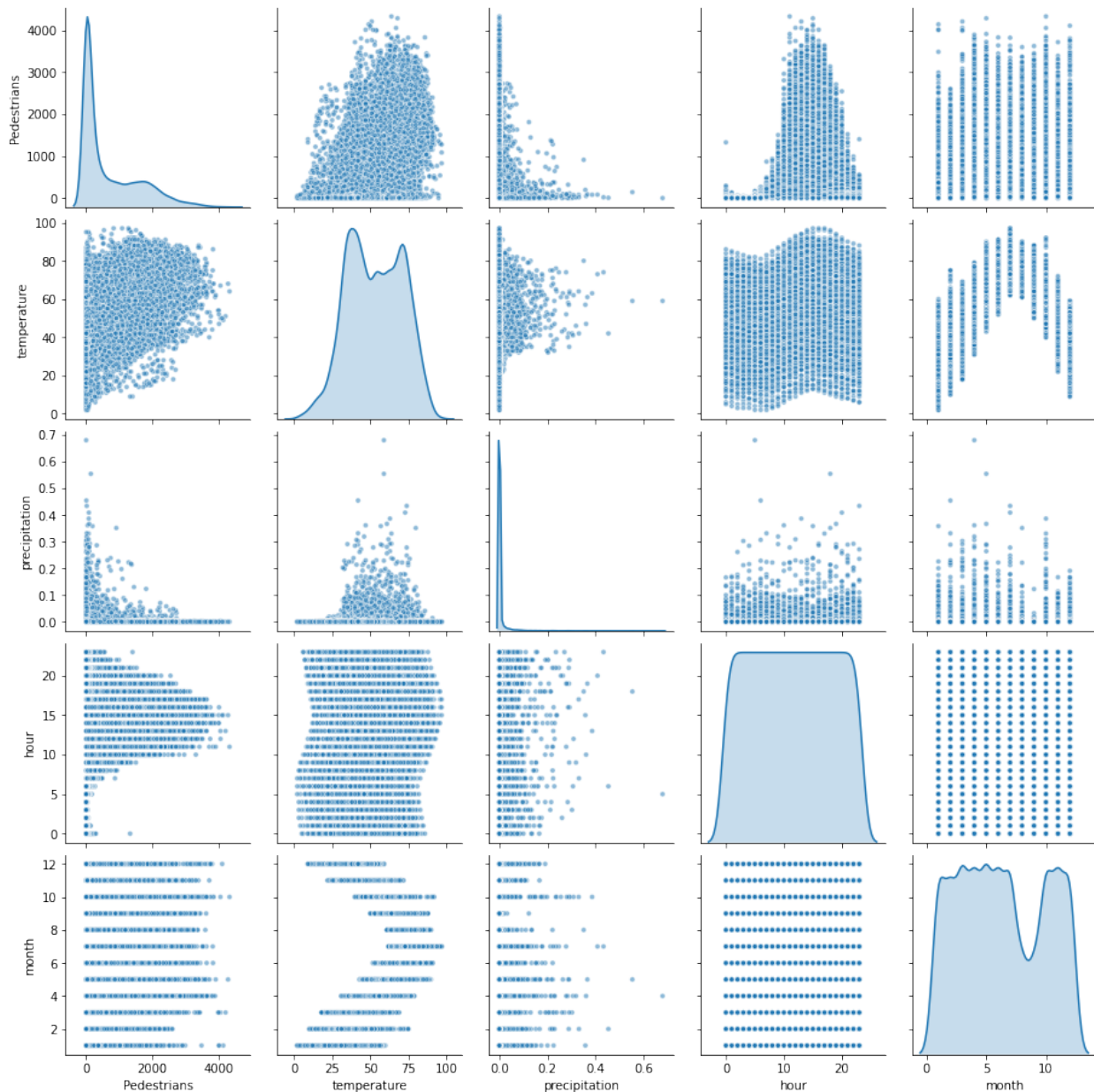
- specifying only the variables we want to include, and excluding variables that don't contain useful information, such as `lat` and `long`, and
- making the points on the plot smaller and partially transparent, to help with the overplotting.

We'll also change the histograms on the diagonal, which show the frequency of values for each variable, into a density plot which shows the same information in a more useful format.

```
sns.pairplot(df,
              vars=['Pedestrians', 'temperature', 'precipitation', 'hour', 'month'],
              diag_kind = 'kde',
```

```
plot_kws={'alpha':0.5, 'size': 0.1})
```

```
<seaborn.axisgrid.PairGrid at 0x7f63fcf561c0>
```



We are mainly interested in the top row of the plot, which shows how the target variable (Pedestrians) varies with the temperature, precipitation levels, and hour. However, it is also useful to note relationships between features. For example, there is a natural relationship between the time of data and the temperature, and between the month and the temperature.

### Summary statistics

Now, we are ready to explore summary statistics. The “five number summary” - extremes (min and max), median, and quartiles - can help us gain a better understanding of the data. We can use the `describe` function in `pandas` to compute this summary.

```
df.describe()
```

	Pedestrians	Towards Manhattan	Towards Brooklyn	temperature	\
count	16057.000000	16057.000000	16057.000000	16057.000000	
mean	687.106309	334.772436	352.286853	53.205892	
std	862.244605	417.807545	456.624509	18.036476	
min	0.000000	0.000000	0.000000	2.000000	
25%	16.000000	9.000000	5.000000	39.000000	
50%	227.000000	112.000000	111.000000	53.000000	
75%	1254.000000	611.000000	632.000000	69.000000	
max	4330.000000	3657.000000	2872.000000	97.000000	

	precipitation	lat	long	hour	month
count	16057.000000	1.605700e+04	1.605700e+04	16057.000000	16057.000000
mean	0.004613	4.070816e+01	-7.399951e+01	11.499346	6.347076
std	0.023389	7.105649e-15	1.421130e-14	6.922682	3.544812
min	0.000000	4.070816e+01	-7.399951e+01	0.000000	1.000000
25%	0.000000	4.070816e+01	-7.399951e+01	5.000000	3.000000
50%	0.000000	4.070816e+01	-7.399951e+01	11.000000	6.000000
75%	0.000000	4.070816e+01	-7.399951e+01	17.000000	10.000000
max	0.680400	4.070816e+01	-7.399951e+01	23.000000	12.000000

We are especially interested in Pedestrians, the target variable, so we can describe that one separately:

```
df['Pedestrians'].describe()
```

```
count    16057.000000
mean      687.106309
std       862.244605
min        0.000000
25%       16.000000
50%      227.000000
75%     1254.000000
max      4330.000000
Name: Pedestrians, dtype: float64
```

For categorical variables, we can use `groupby` to get frequency and other useful summary statistics.

For example, we may be interested in the summary statistics for Pedestrians for different weather conditions:

```
df.groupby('weather_summary')['Pedestrians'].describe()
```

	count	mean	std	min	25%	50%	\
weather_summary							
clear-day	3127.0	1386.569875	861.890079	0.0	611.50	1401.0	
clear-night	3755.0	102.689214	206.438992	0.0	2.00	19.0	
cloudy	2383.0	540.437684	727.986539	0.0	9.00	141.0	
fog	76.0	234.473684	307.735795	0.0	15.00	110.0	
partly-cloudy-day	3169.0	1422.154307	844.930127	0.0	699.00	1433.0	
partly-cloudy-night	2508.0	93.880383	173.265652	0.0	3.00	23.0	
rain	920.0	256.165217	421.571020	0.0	10.00	67.5	
sleet	14.0	117.928571	157.850204	0.0	7.25	28.0	

snow	93.0	195.473118	292.630818	0.0	16.00	77.0
wind	12.0	668.333333	682.617067	0.0	8.00	596.5
	75%	max				
weather_summary						
clear-day	1982.5	4330.0				
clear-night	93.5	1779.0				
cloudy	944.5	3894.0				
fog	276.5	1321.0				
partly-cloudy-day	2008.0	4286.0				
partly-cloudy-night	97.0	1522.0				
rain	311.0	2727.0				
sleet	254.5	404.0				
snow	258.0	1561.0				
wind	1010.0	1910.0				

Make special note of the `count` column, which shows us the prevalence of different weather conditions in this dataset. There are some weather conditions for which we have very few examples.

Another categorical variable is `events`, which indicates whether the day is a holiday, and which holiday. Holidays have very different pedestrian traffic characteristics from other days.

```
df.groupby('events')['Pedestrians'].describe()
```

	count	mean	std	min	\
events					
Black Friday	48.0	723.583333	952.014814	0.0	
Christmas Day	48.0	832.312500	1199.381546	0.0	
Christmas Eve	48.0	705.520833	945.112444	0.0	
Cinco de Mayo	48.0	807.750000	1047.286392	3.0	
Columbus Day (regional holiday)	44.0	694.181818	854.264712	0.0	
Daylight Saving Time ends	48.0	548.687500	719.950003	0.0	
Daylight Saving Time starts	48.0	504.500000	708.192515	0.0	
Easter Monday	24.0	581.916667	704.003515	0.0	
Easter Sunday	48.0	1321.812500	1443.738832	0.0	
Father's Day	48.0	930.645833	836.469111	0.0	
Halloween	48.0	566.104167	789.258533	0.0	
Independence Day	48.0	749.604167	886.326983	0.0	
Labor Day	24.0	513.666667	540.116869	0.0	
Martin Luther King Jr. Day	48.0	195.166667	281.788387	0.0	
Memorial Day	48.0	1314.333333	1346.292282	0.0	
Mother's Day	48.0	334.895833	430.048896	1.0	
New Year's Day	48.0	845.395833	1229.824148	2.0	
New Year's Eve	48.0	827.062500	1159.014556	0.0	
Presidents' Day (regional holiday)	48.0	535.541667	622.282927	0.0	
St. Patrick's Day	48.0	747.458333	864.023344	0.0	
Tax Day	48.0	567.625000	626.338316	0.0	
Thanksgiving Day	48.0	564.708333	783.503098	0.0	
Valentine's Day	48.0	451.479167	509.012979	0.0	
Veterans Day	48.0	631.833333	803.411114	0.0	
Veterans Day observed	24.0	421.083333	477.603703	2.0	
	25%	50%	75%	max	
events					



Black Friday	4.00	172.5	1515.50	2913.0
Christmas Day	6.00	82.5	1480.75	3807.0
Christmas Eve	10.75	113.0	1406.50	2625.0
Cinco de Mayo	64.25	408.5	848.25	3390.0
Columbus Day (regional holiday)	28.75	332.0	878.75	2587.0
Daylight Saving Time ends	14.75	170.0	983.00	2311.0
Daylight Saving Time starts	10.00	123.5	814.25	2232.0
Easter Monday	15.25	380.5	850.25	2242.0
Easter Sunday	55.00	410.5	2809.50	3894.0
Father's Day	111.75	807.5	1831.50	2128.0
Halloween	13.75	144.0	803.00	2465.0
Independence Day	32.00	259.0	1305.50	2727.0
Labor Day	18.50	283.5	1080.50	1486.0
Martin Luther King Jr. Day	5.50	41.5	281.25	955.0
Memorial Day	53.75	745.0	2752.00	3657.0
Mother's Day	49.50	169.5	494.50	1693.0
New Year's Day	28.25	149.0	1353.00	4141.0
New Year's Eve	1.75	178.0	1517.00	3587.0
Presidents' Day (regional holiday)	27.00	138.5	1237.50	1648.0
St. Patrick's Day	11.25	221.5	1557.00	2617.0
Tax Day	16.75	284.5	1208.75	1910.0
Thanksgiving Day	5.00	112.0	1093.50	2298.0
Valentine's Day	17.50	193.0	1053.00	1448.0
Veterans Day	11.75	159.0	1172.50	2265.0
Veterans Day observed	56.75	124.5	878.75	1269.0

It can be useful to get the total pedestrian count for the day of a holiday, rather than the summary statistics for the hour-long intervals. We can use the `agg` function to compute key statistics, including summing over all the samples in the group:

```
df.groupby('events').agg({'Pedestrians': 'sum'})
```

	Pedestrians
events	
Black Friday	34732
Christmas Day	39951
Christmas Eve	33865
Cinco de Mayo	38772
Columbus Day (regional holiday)	30544
Daylight Saving Time ends	26337
Daylight Saving Time starts	24216
Easter Monday	13966
Easter Sunday	63447
Father's Day	44671
Halloween	27173
Independence Day	35981
Labor Day	12328
Martin Luther King Jr. Day	9368
Memorial Day	63088
Mother's Day	16075
New Year's Day	40579
New Year's Eve	39699
Presidents' Day (regional holiday)	25706
St. Patrick's Day	35878

Tax Day	27246
Thanksgiving Day	27106
Valentine's Day	21671
Veterans Day	30328
Veterans Day observed	10106

## Explore relationships and look for issues

Finally, let's further explore relationships between likely predictors and our target variable. We can group by `day_name`, then call the `describe` function on the `Pedestrians` column to see the effect of day of the week on traffic volume:

```
df.groupby('day_name')['Pedestrians'].describe()
```

	count	mean	std	min	25%	50%	75%	max
day_name								
Friday	2280.0	696.521053	845.244195	0.0	17.0	243.5	1318.00	3722.0
Monday	2304.0	642.983941	777.944829	0.0	12.0	232.0	1232.00	3657.0
Saturday	2280.0	943.185965	1159.857344	0.0	22.0	241.5	1894.50	4330.0
Sunday	2305.0	753.213015	947.772750	0.0	19.0	206.0	1452.00	3894.0
Thursday	2280.0	601.263158	728.067954	0.0	16.0	214.0	1102.25	3173.0
Tuesday	2328.0	599.210911	731.047235	0.0	14.0	232.5	1122.50	4141.0
Wednesday	2280.0	574.956140	694.807586	0.0	16.0	217.0	1050.00	3807.0

Similarly, we can see the effect of temperature:

```
df.groupby('temperature')['Pedestrians'].describe()
```

	count	mean	std	min	25%	50%	75%	\
temperature								
2.0	3.0	19.333333	25.929391	1.0	4.50	8.0	28.50	
3.0	4.0	16.000000	32.000000	0.0	0.00	0.0	16.00	
4.0	8.0	27.375000	25.767851	0.0	6.25	24.5	40.50	
5.0	5.0	20.000000	41.418595	0.0	0.00	1.0	5.00	
6.0	10.0	54.200000	85.590498	0.0	4.25	10.5	79.75	
...	...	...	...	...	...	...	...	
93.0	3.0	1271.333333	707.043374	455.0	1062.00	1669.0	1679.50	
94.0	4.0	1035.750000	746.532596	101.0	599.75	1202.0	1638.00	
95.0	3.0	476.666667	765.789353	0.0	35.00	70.0	715.00	
96.0	4.0	1161.500000	426.649349	538.0	1104.25	1301.5	1358.75	
97.0	3.0	1063.666667	225.331607	828.0	957.00	1086.0	1181.50	
	max							
temperature								
2.0	49.0							
3.0	64.0							
4.0	77.0							
5.0	94.0							
6.0	275.0							
...	...							
93.0	1690.0							
94.0	1638.0							
95.0	1360.0							

```
96.0      1505.0
97.0      1277.0

[96 rows x 8 columns]
```

And the effect of precipitation:

```
df.groupby('precipitation')['Pedestrians'].describe()
```

```
precipitation      count      mean      std      min      25%      50%      75%  \
0.0000      12338.0  731.183336  901.428828     0.0    15.00   236.0   1390.75
0.0001         226.0  548.982301  652.851843     0.0    52.25   256.0    931.00
0.0002         224.0  753.151786  849.076567     0.0    41.75   347.0   1389.50
0.0003         154.0  705.051948  733.243531     0.0    50.00   493.5   1230.00
0.0004         130.0  717.253846  766.100731     0.0    22.25   438.0   1243.00
...
0.4090          1.0    81.000000         NaN    81.0    81.00    81.0    81.00
0.4340          1.0   18.000000         NaN    18.0    18.00    18.0    18.00
0.4543          1.0    6.000000         NaN     6.0     6.00     6.0     6.00
0.5543          1.0  141.000000         NaN   141.0   141.00   141.0   141.00
0.6804          1.0    0.000000         NaN     0.0     0.00     0.0     0.00

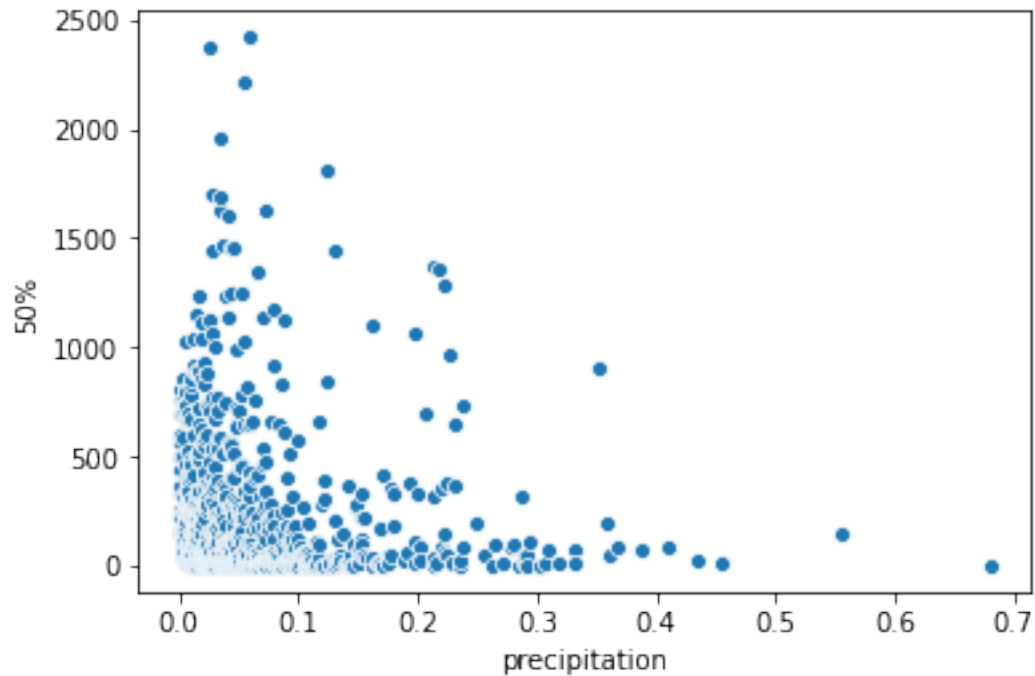
max
precipitation
0.0000      4286.0
0.0001      4330.0
0.0002      3816.0
0.0003      3485.0
0.0004      2733.0
...
0.4090        81.0
0.4340        18.0
0.4543         6.0
0.5543       141.0
0.6804         0.0

[776 rows x 8 columns]
```

We can even plot it separately, by saving it in a new data frame and plotting *that* data frame:

```
df_precip = df.groupby('precipitation')['Pedestrians'].describe()
df_precip = df_precip.reset_index()
sns.scatterplot(data=df_precip, x='precipitation', y='50%')
```

```
<AxesSubplot:xlabel='precipitation', ylabel='50%>
```



We see that certain weather conditions (very high temperature, heavy precipitation, fog) are extremely underrepresented in the dataset. This would be something to consider if, for example, we wanted to use this dataset to predict the effect of extreme weather on pedestrian traffic.

### Next steps

What are some signals that something is wrong with your data?