Published in EPFL Extension School

Michael Notter Follow
Feb 18 · 14 min read · Listen

Advanced exploratory data analysis (EDA) with Python

How to quickly get a handle on almost any tabular dataset

[Find the code to this article here.]

. . .

Getting a good feeling for a new dataset is not always easy, and takes time. However, a good and broad exploratory data analysis (EDA) can help a lot to understand your dataset, get a feeling for how things are connected and what needs to be done to properly process your dataset.

In this article, we will touch upon multiple useful EDA routines. However, to keep things short and compact we might not always dig deeper or explain all of the implications. But in reality, spending enough time on a proper EDA to fully understand your dataset is a key part of any good data science project. As a rule of thumb, **you probably will spend 80% of your time in data preparation and exploration and only 20% in actual machine learning modeling**.

Investigation of structure, quality and content

Overall, the EDA approach is very iterative. At the end of your investigation you might discover something that will require you to redo everything once more. That is normal! But to impose at least a little bit of structure, I propose the following structure for your investigations:

1. **Structure investigation**: Exploring the general shape of the dataset, as well as the data types of your features.
2. **Quality investigation**: Get a feeling for the general quality of the dataset, with regards to duplicates, missing values and unwanted entries.
3. **Content investigation**: Once the structure and quality of the dataset is understood, we can go ahead and perform a more in-depth exploration on the features values and look at how different features relate to each other.

But first we need to find an interesting dataset. Let's go ahead and load the road safety dataset from OpenML.

```
from sklearn.datasets import fetch_openml

# Download the dataset from openml
dataset = fetch_openml(data_id=42803, as_frame=True)

# Extract feature matrix X and show 5 random samples
df_X = dataset["frame"]
```

```
# Show size of the dataset
df_X.shape

>>> (363243, 67)


import pandas as pd

# Count how many times each data type is present in
the dataset
pd.value_counts(df_X.dtypes)
```

1.1. Structure of non-numerical features

Data types can be numerical and non-numerical. First, let's take a closer look at the **non-numerical** entries.

	Accident_Index	Sex_of_Driver	Date	Time	Local_Authority_(Highway)	LSOA_of_Accident_Location
0	201501BS70001	1.0	12/01/2015	18:45	E09000020	E01002825
1	201501BS70002	1.0	12/01/2015	07:50	E09000020	E01002820
2	201501BS70004	1.0	12/01/2015	18:08	E09000020	E01002833
3	201501BS70005	1.0	13/01/2015	07:40	E09000020	E01002874
4	201501BS70008	1.0	09/01/2015	07:30	E09000020	E01002814

Even though `Sex_of_Driver` is a numerical feature, it somehow was stored as a non-numerical one. This is sometimes due to some typo in data recording. These kind of things need to be taken care of during data preparation.

Once this is taken care of, we can use the `.describe()` function to investigate how many unique values each non-numerical feature has and with which frequency the most prominent value is present - using the code `df_X.describe(exclude="number")` :

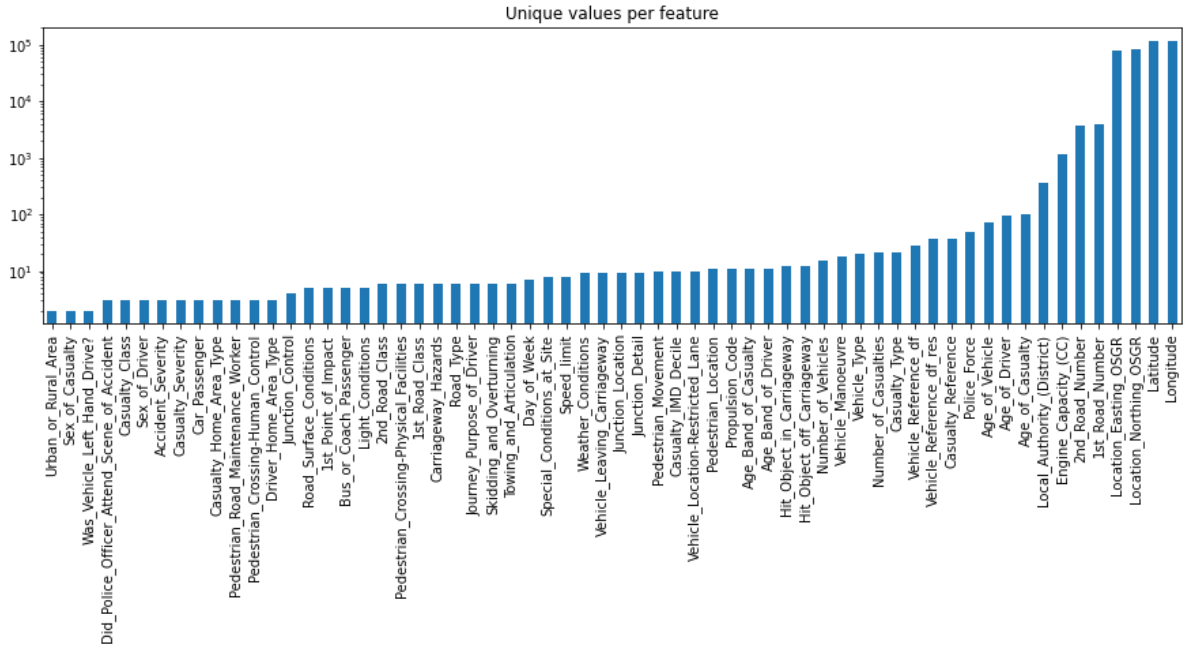
	Accident_Index	Date	Time	Local_Authority_(Highway)	LSOA_of_Accident_Location
count	363243	319866	319822	319866	298758
unique	140056	365	1439	204	25979
top	201543P296025	14/02/2015	17:30	E10000017	E01028497
freq	1332	2144	2972	8457	1456

1.2. Structure of numerical features

Next, let's take a closer look at the numerical features. More precisely, let's investigate how many unique values each of these feature has. This process will give us some insights about the number of **binary** (2 unique values), **ordinal** (3 to ~10 unique values) and **continuous** (more than 10 unique values) features in the dataset.

```
# For each numerical feature compute number of unique
entries
unique_values = df_X.select_dtypes(
  include="number").nunique().sort_values()

# Plot information with y-axis in log-scale
unique_values.plot.bar(logy=True, figsize=(15, 4),
                       title="Unique values per
```



1.3. Conclusion of structure investigation

At the end of this first investigation, we should have a better understanding of the general structure of our dataset. Number of samples and features, what kind of data type each feature has, and how many of them are binary, ordinal, categorical or continuous. For an alternative way to get such kind of information you could also use `df_X.info()` or `df_X.describe()`.

2. Quality Investigation

Before focusing on the actual content stored in these features, let's first take a look at the general quality of the dataset. The goal is to have a global view on the dataset with regards to things like *duplicates*, *missing values* and *unwanted entries* or *recording errors*.

2.1. Duplicates

Duplicates are entries that represent the same sample point multiple times. For example, if a measurement was registered twice by two different people. Detecting such duplicates is not always easy, as each dataset might have a unique identifier features (e.g. an index number or recording time that is unique to each new sample). So you might want to ignore them first. And once you are aware about the number of duplicates in your dataset, you can simply drop them with `.drop_duplicates()`.

```
feature
n_duplicates = df_X.drop(labels=["Accident_Index"],
axis=1).duplicated().sum()
print(f"You seem to have {n_duplicates} duplicates in
your database.")

>>> You seem to have 22 duplicates in your database.

#  Extract column names of all features, except
'Accident_Index'
columns_to_consider = df_X.drop(labels=
["Accident_Index"], axis=1).columns
```

2.2. Missing values

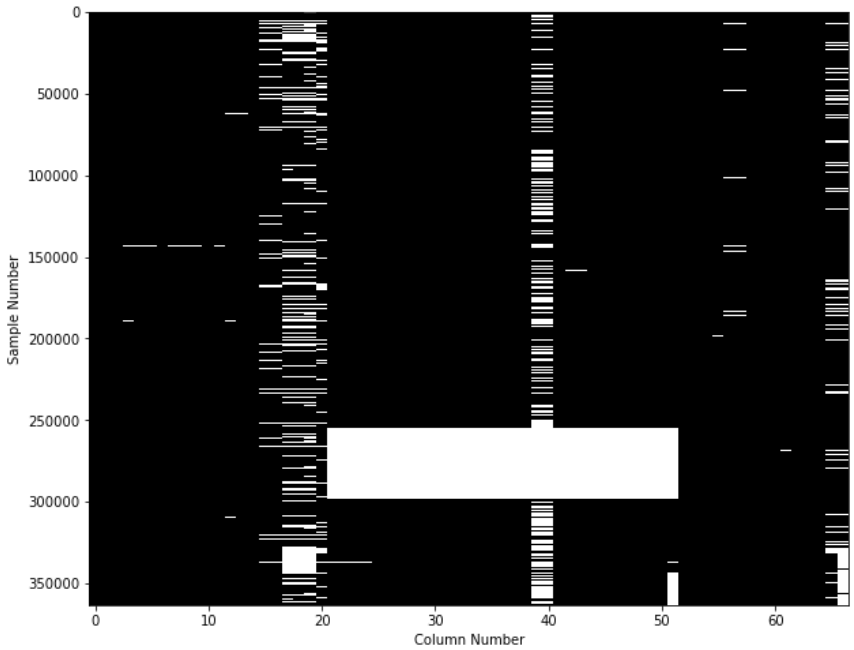
Another quality issue worth to investigate are **missing values**. Having some missing values is normal. What we want to identify at this stage are big holes in the dataset, i.e. samples or features with a lot of missing values.

2.2.1. Per sample

To look at number of missing values per sample we have multiple options. The most straight forward one is to simply visualize the output of `df_X.isna()`, with something like this:

```
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 8))
plt.imshow(df_X.isna(), aspect="auto",
interpolation="nearest", cmap="gray")
plt.xlabel("Column Number")
plt.ylabel("Sample Number");
```



This figure shows on the y-axis each of the 360'000 individual samples, and on the x-axis if any of the 67 features contains a missing value. While this is already a useful plot, an even better approach is to use the missingno library, to get a plot like this one:

From both of these plots we can see that the dataset has a huge hole, caused by some samples where more than 50% of the feature values are missing. For those samples, filling the missing values with some replacement values is probably not a good idea.

Therefore, let's go ahead and drop samples that have more than 20% of missing values. The threshold is inspired by the information from the 'Data Completeness' column on the right of this figure.

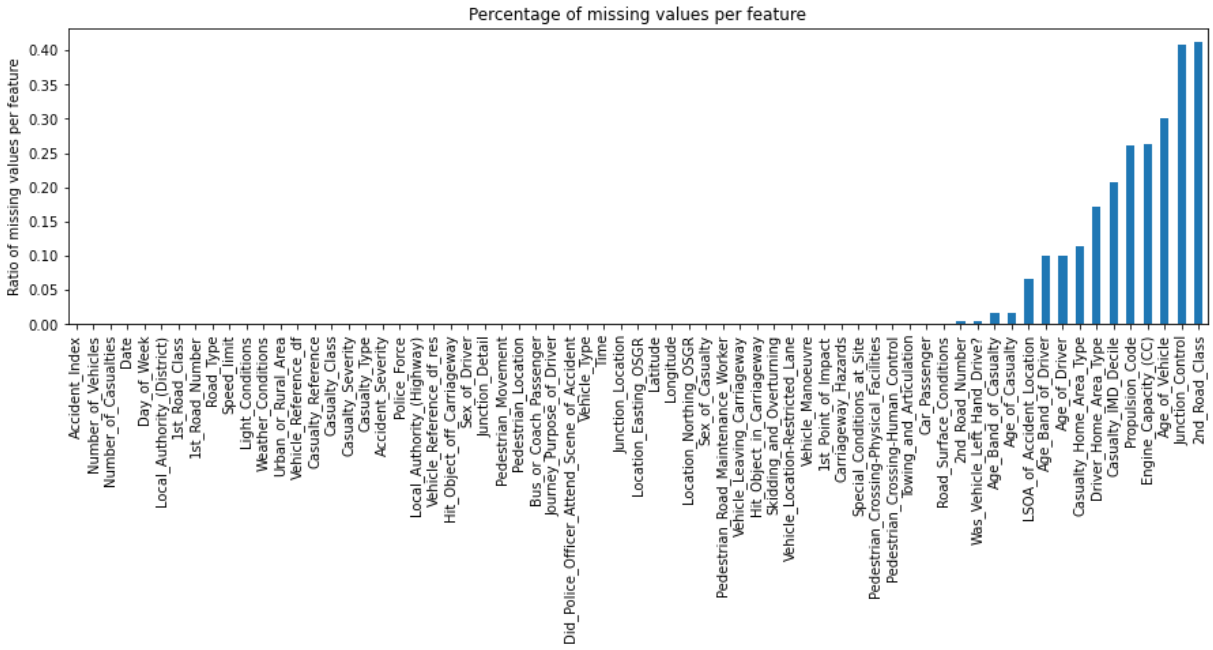
```
df_X = df_X.dropna(thresh=df_X.shape[1] * 0.80,
axis=0).reset_index(drop=True)
df_X.shape

>>> (319790, 67)
```

2.2.2. Per Feature

As a next step, let's now look at the number of missing values per feature. For this we can use some `pandas` trickery to quickly identify the ratio of missing values **per feature**.

```
df_X.isna().mean().sort_values().plot(
    kind="bar", figsize=(15, 4),
    title="Percentage of missing values per feature",
    ylabel="Ratio of missing values per feature");
```



From this figure we can see that most features don't contain any missing values. Nonetheless, features like `2nd_Road_Class`, `Junction_Control`, `Age_of_Vehicle` still contain quite a lot of missing values. So let's go ahead and remove any feature with more than 15% of missing values.

```
df_X = df_X.dropna(thresh=df_X.shape[0] * 0.85,
axis=1)
df_X.shape
```

2.2.3. Small side note

Missing values: There is no strict order in removing missing values. For some datasets, tackling first the features and than the samples might be better. Furthermore, the threshold at which you decide to drop missing values per feature or sample changes from dataset to dataset, and depends on what you intend to do with the dataset later on.

Also, until now we only addressed the big holes in the dataset, not yet how we would fill the smaller gaps. This is content for another post.

2.3. Unwanted entries and recording errors

Another source of quality issues in a dataset can be due to unwanted entries or recording errors. It's important to distinguish such samples from simple outliers. While outliers are data points that are unusual for a given feature distribution, **unwanted entries or recording errors are samples that shouldn't be there in the first place**.

For example, a temperature recording of 45°C in Switzerland might be an outlier (as in 'very unusual'), while a recording at 90°C would be an error. Similarly, a temperature recording from the top of Mont Blanc might be physical possible, but most likely shouldn't be included in a dataset about Swiss cities.

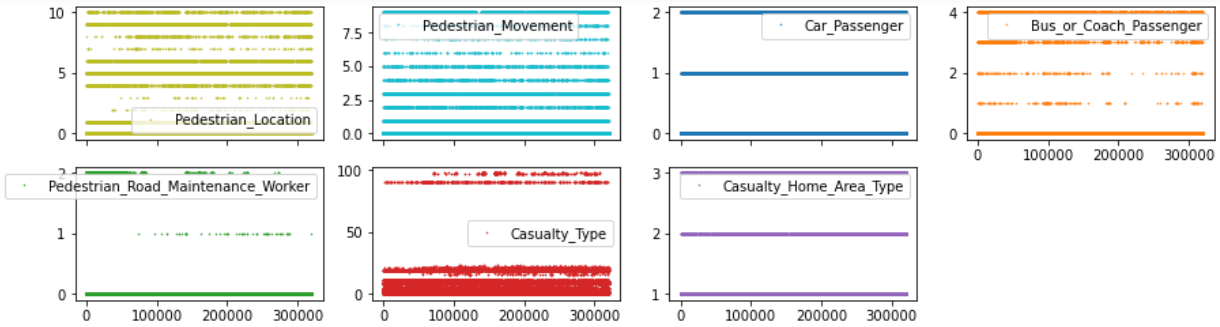
Of course, detecting such errors and unwanted entries and distinguishing them from outliers is not always straight forward and depends highly on the dataset. One approach to this is to take a global view on the dataset and see if you can identify some very unusual patterns.

2.3.1. Numerical features

To plot this global view of the dataset, at least for the numerical features, you can use pandas' `.plot()` function and combine it with the following parameters:

- `lw=0`: `lw` stands for line width. `0` means that we don't want to show any lines
- `marker="."`: Instead of lines, we tell the plot to use `.` as markers for each data point
- `subplots=True`: `subplots` tells `pandas` to plot each feature in a separate subplot
- `layout=(-1, 4)`: This parameter tells `pandas` how many rows and columns to use for the subplots. The `-1` means "as many as needed", while the `2` means to use 2 columns per row.
- `figsize=(15, 30), markersize=1`: To make sure that the figure is big enough we recommend to have a figure height of roughly the number of features, and to adjust the `markersize` accordingly.

So what does this plot look like?



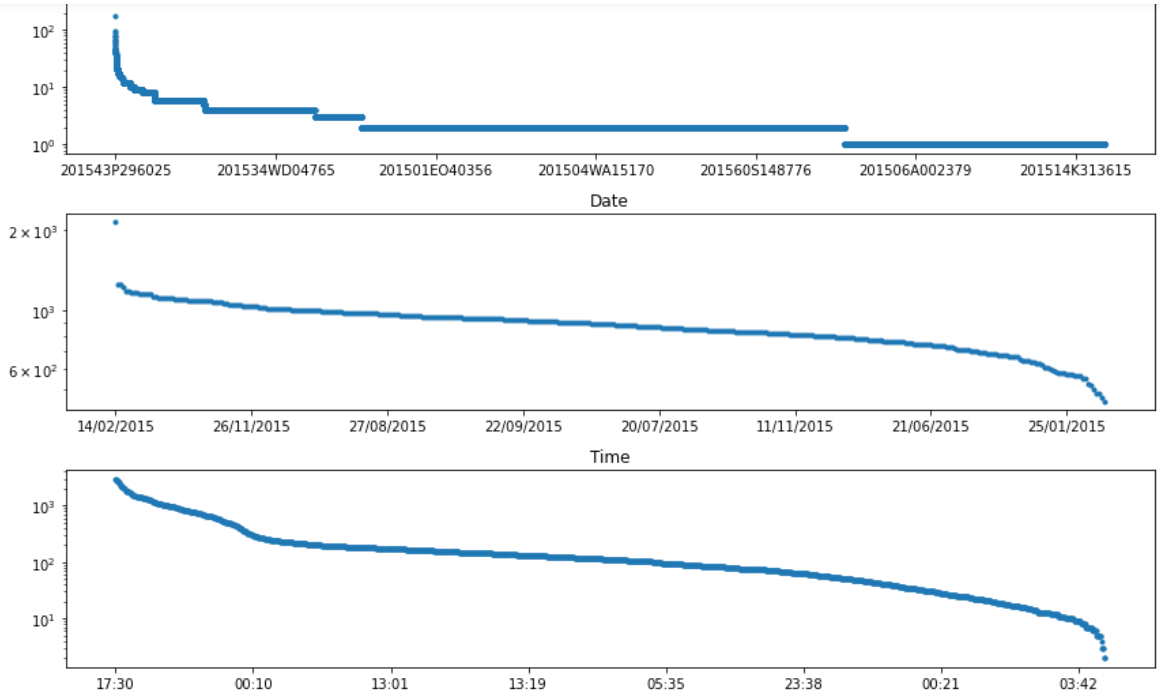
Each point in this figure is a sample (i.e. a row) in our dataset and each subplot represents a different feature. The y-axis shows the feature value, while the x-axis is the sample index. These kind of plots can give you a lot of ideas for data cleaning and EDA. Usually it makes sense to invest as much time as needed until you're happy with the output of this visualization.

2.3.2. Non-numerical features

Identifying **unwanted entries** or **recording errors** on non-numerical features is a bit more tricky. Given that at this point, we only want to investigate the general quality of the dataset. So what we can do is take a general look at how many unique values each of these non-numerical features contain, and how often their most frequent category is represented. To do so, you can use: `df_X.describe(exclude=["number", "datetime])`

	Accident_Index	Date	Time	Local_Authority_(Highway)	LSOA_of_Accident_Location
count	319790	319790	319746	319790	298693
unique	123645	365	1439	204	25977
top	201543P296025	14/02/2015	17:30	E10000017	E01028497
freq	1332	2144	2969	8457	1456

There are multiple ways for how you could potentially streamline the quality investigation for each individual non-numerical features. None of them is perfect, and all of them will require some follow up investigation. But for the purpose of showcasing one such a solution, what we could do is loop through all non-numerical features and plot for each of them the number of occurrences per unique value.



We can see that the most frequent accident (i.e. `Accident_Index`), had more than 100 people involved. Digging a bit deeper (i.e. looking at the individual features of this accident), we could identify that this accident happened on February 24th, 2015 at 11:55 in Cardiff UK. A quick internet search reveals that this entry corresponds to a luckily non-lethal accident including a minibus full of pensioners.

The decision for what should be done with such rather unique entries is once more left in the the subjective hands of the person analyzing the dataset. Without any good justification for WHY, and only with the intention to show you the HOW - let's go ahead and remove the 10 most frequent accidents from this dataset.

2.4. Conclusion of quality investigation

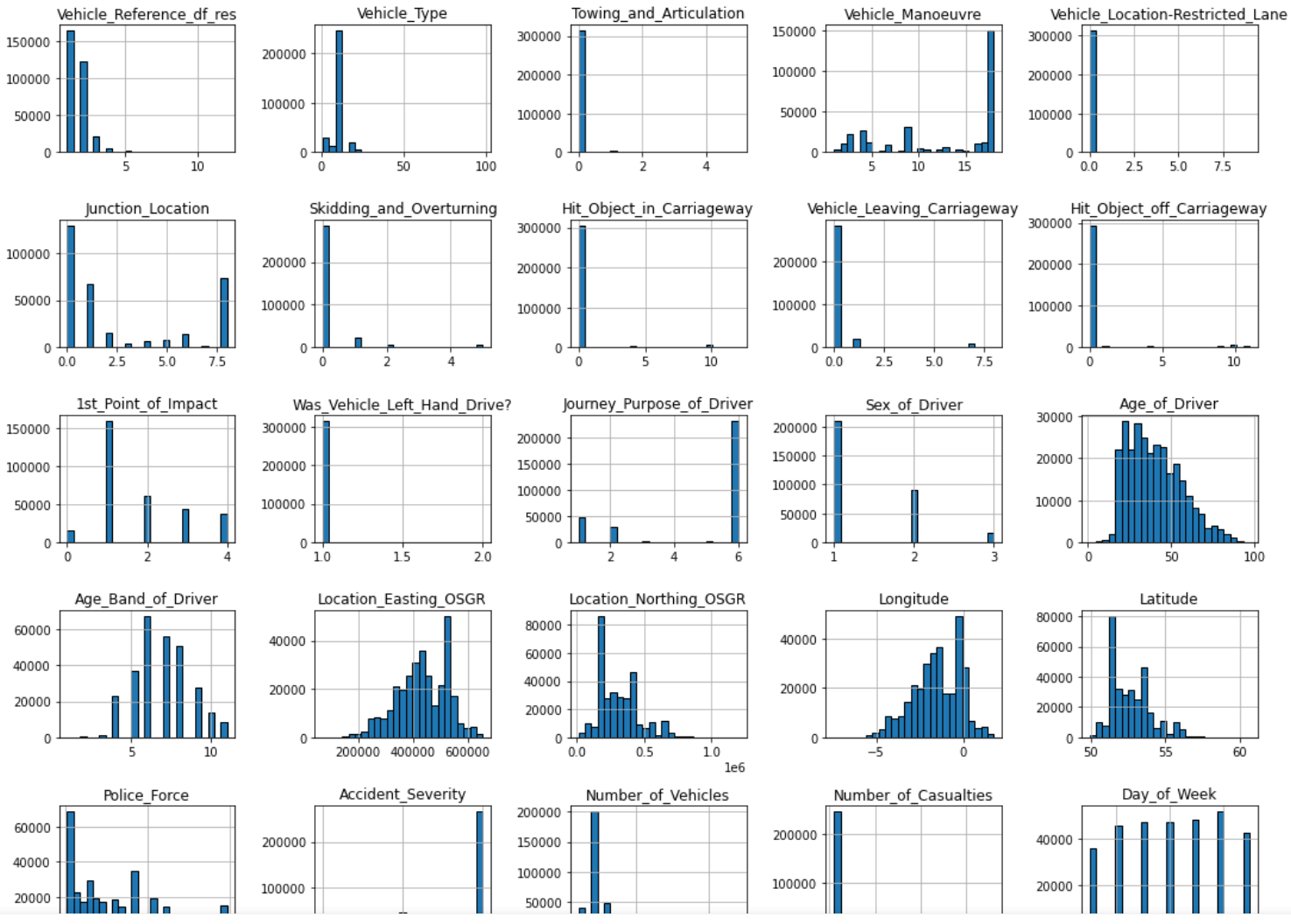
At the end of this second investigation, we should have a better understanding of the general quality of our dataset. We looked at duplicates, missing values and unwanted entries or recording errors. It is important to point out that we didn't discuss yet how to address the remaining missing values or outliers in the dataset. This is a task for the next investigation, but won't be covered in this article.

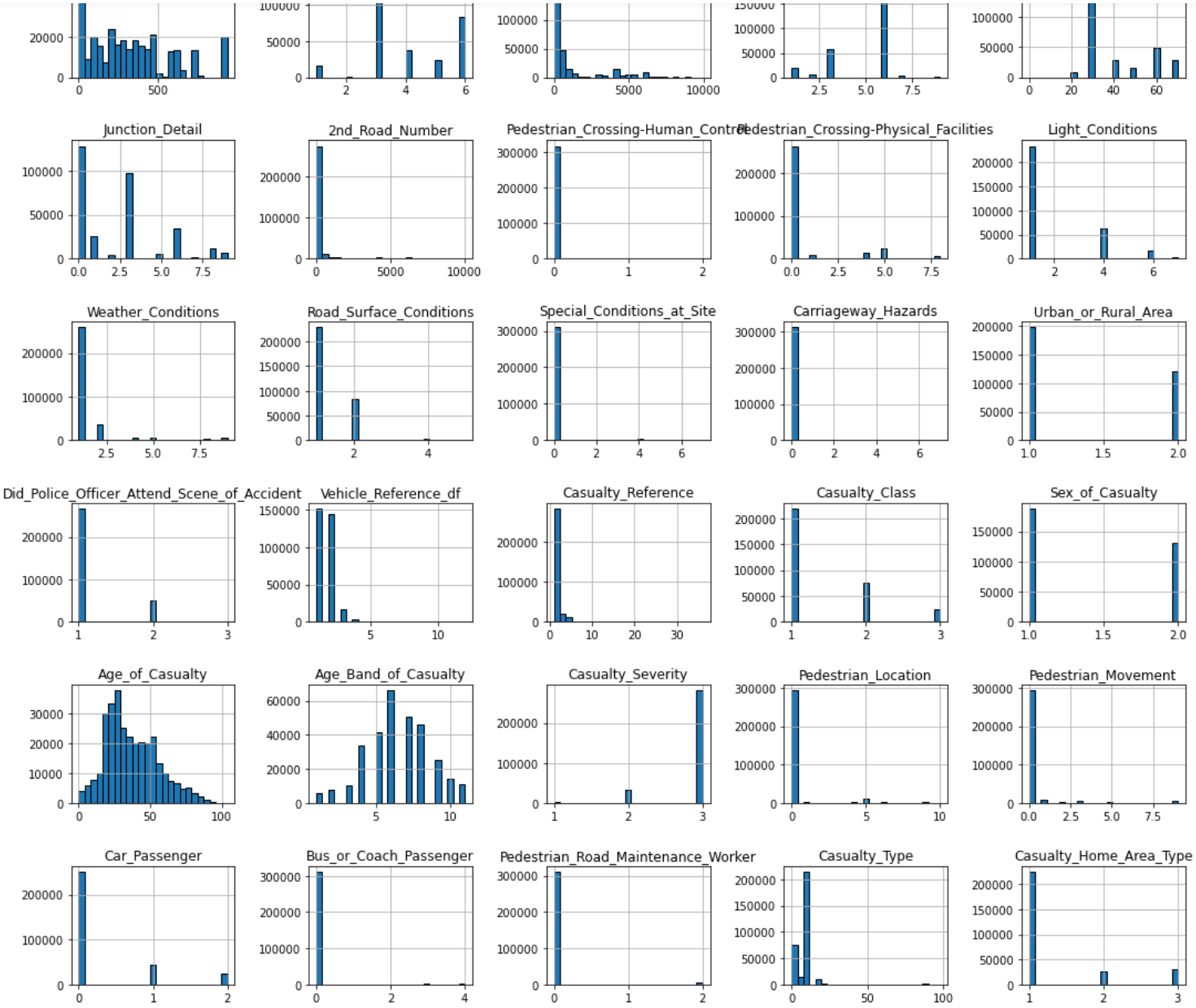
3. Content Investigation

Up until now we only looked at the general structure and quality of the dataset. Let's now go a step further and take a look at the actual content. In an ideal setting, such an investigation would be done feature by feature. But this becomes very cumbersome once you have more than 20–30

3.1. Feature distribution

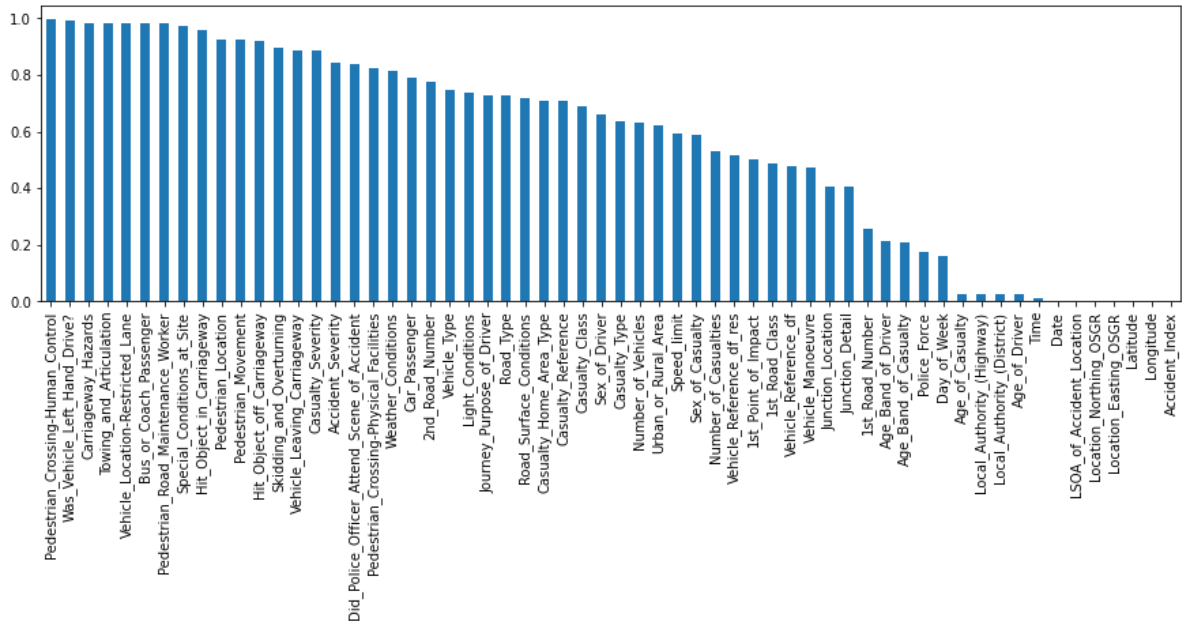
Looking at the value distribution of each feature is a great way to better understand the content of your data. Furthermore, it can help to guide your EDA, and provides a lot of useful information with regards to data cleaning and feature transformation. The quickest way to do this for numerical features is using histogram plots. Luckily, `pandas` comes with a builtin histogram function that allows the plotting of multiple features at once.





There are a lot of very interesting things visible in this plot. For example…

Most frequent entry: Some features, such as `Towing_and_Articulation` or `Was_Vehicle_Left_Hand_Drive?` mostly contain entries of just one category. Using the `.mode()` function, we could for example extract the ratio of the most frequent entry for each feature and visualize that information.



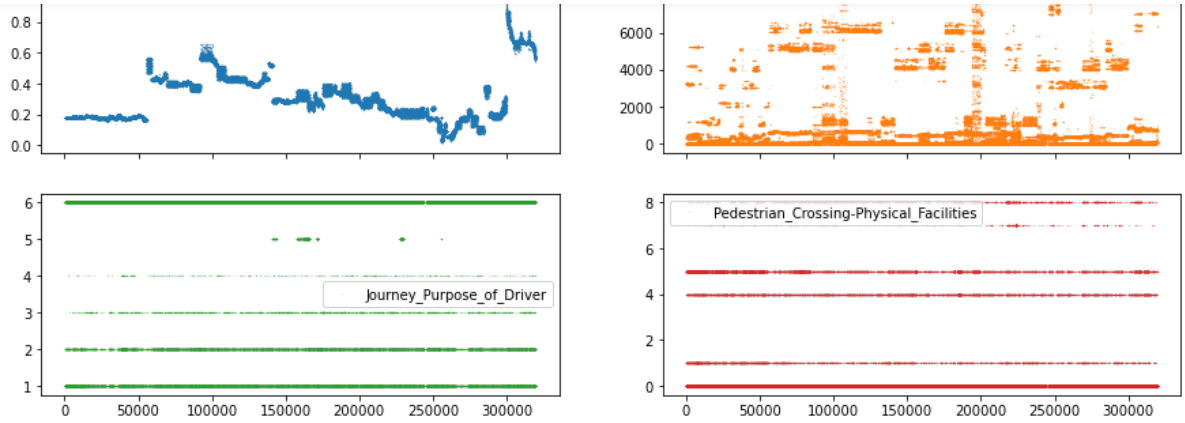
Skewed value distributions: Certain kind of numerical features can also show strongly non-gaussian distributions. In that case you might want to think about how you can transform these values to make them more normal distributed. For example, for right skewed data you could use a log-transformation.

3.2. Feature patterns

Next step on the list is the investigation of feature specific patterns. The goal of this part is two fold:

1. Can we identify particular patterns within a feature that will help us to decide if some entries need to be dropped or modified?
2. Can we identify particular relationships between features that will help us to better understand our dataset?

Before we dive into these two questions, let's take a closer look at a few 'randomly selected' features.



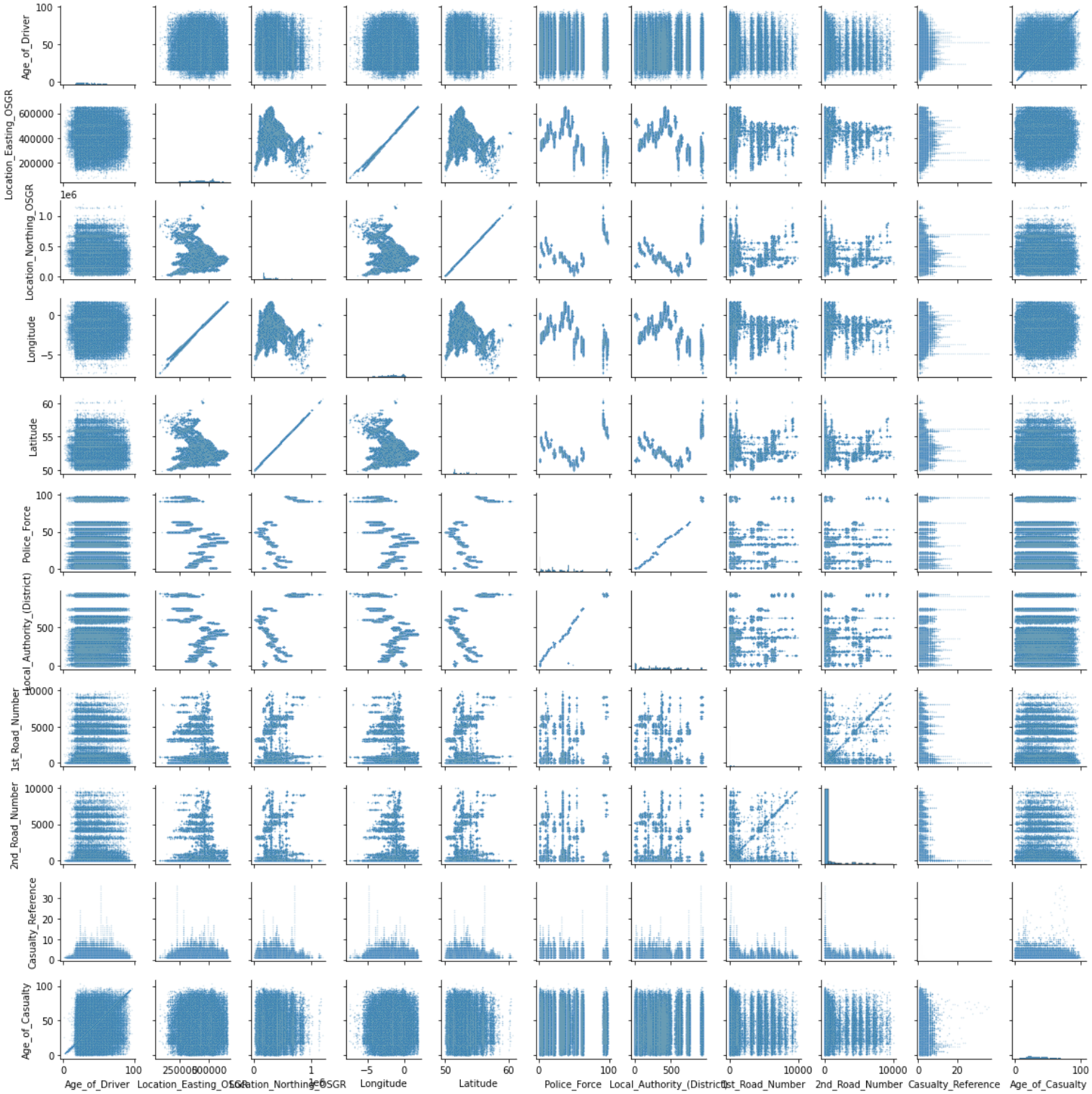
In the top row, we can see features with continuous values (e.g. seemingly any number from the number line), while in the bottom row we have features with discrete values (e.g. 1, 2, 3 but not 2.34).

While there are many ways we could explore our features for particular patterns, let's simplify our option by deciding that we treat features with less than 25 unique features as **discrete** or **ordinal** features, and the other features as **continuous** features.

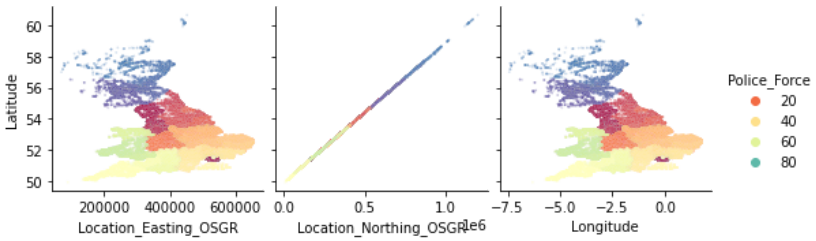
3.2.1. Continuous features

Now that we have a way to select the continuous features, let's go ahead and use seaborn's `pairplot` to visualize the relationships between these features. **Important to note**, seaborn's pairplot routine can take a long time to create all subplots. Therefore we recommend to not use it for more than ~10 features at a time.

Given that in our case we only have 11 features, we can go ahead with the pairplot. Otherwise, using something like `df_continuous.iloc[:, :5]` could help to reduce the number of features to plot.



There seems to be a strange relationship between a few features in the top left corner. `Location_Easting_OSGR` and `Longitude`, as well as `Location_Easting_OSGR` and `Latitude` seem to have a very strong linear relationship.



Knowing that these features contain geographic information, a more in-depth EDA with regards to geolocation could be fruitful. However, for now we will leave the further investigation of this pairplot to the curious reader and continue with the exploration of the discrete and ordinal features.

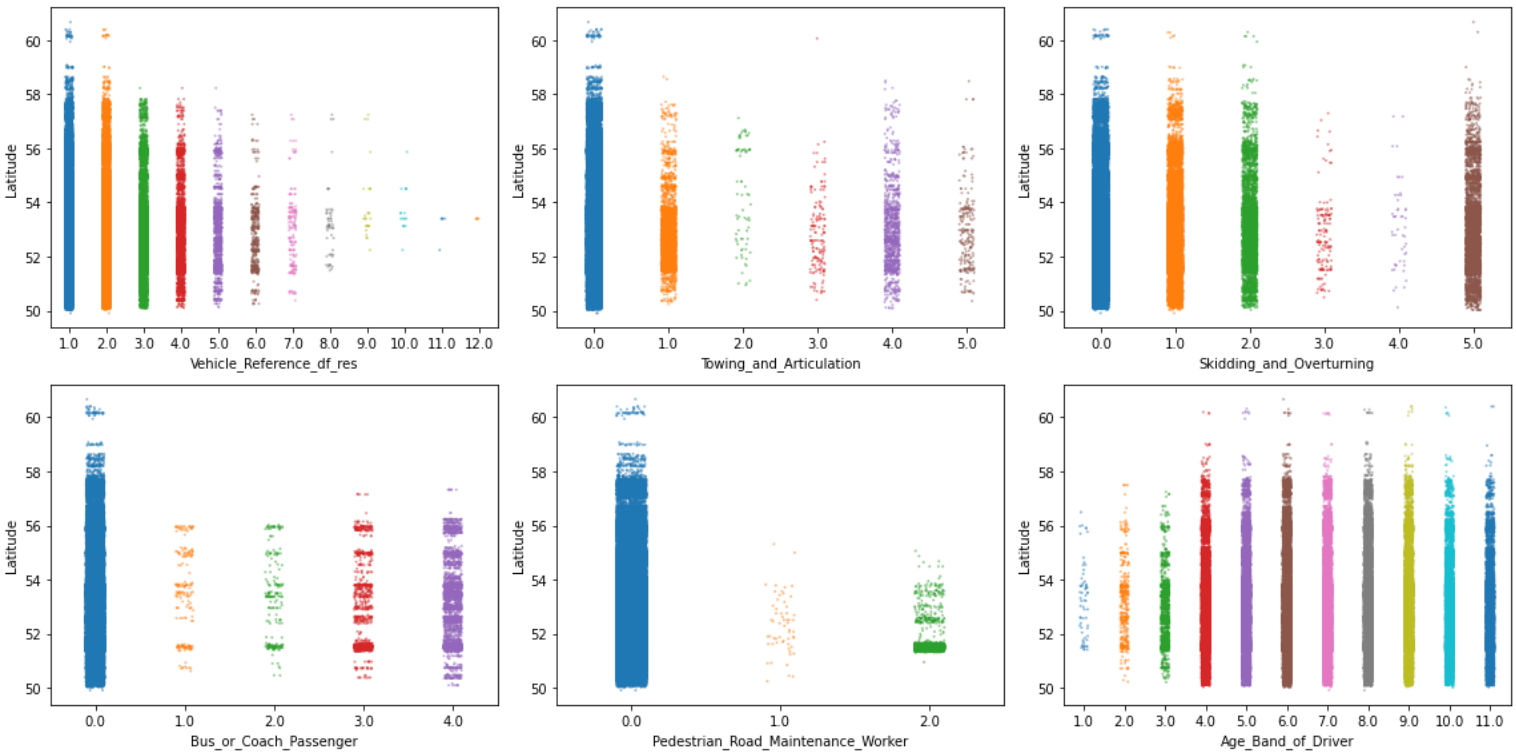
3.2.2. Discrete and ordinal features

Finding patterns in the discrete or ordinal features is a bit more tricky. But also here, some quick pandas and seaborn trickery can help us to get a general overview of our dataset. First, let's select the columns we want to investigate.

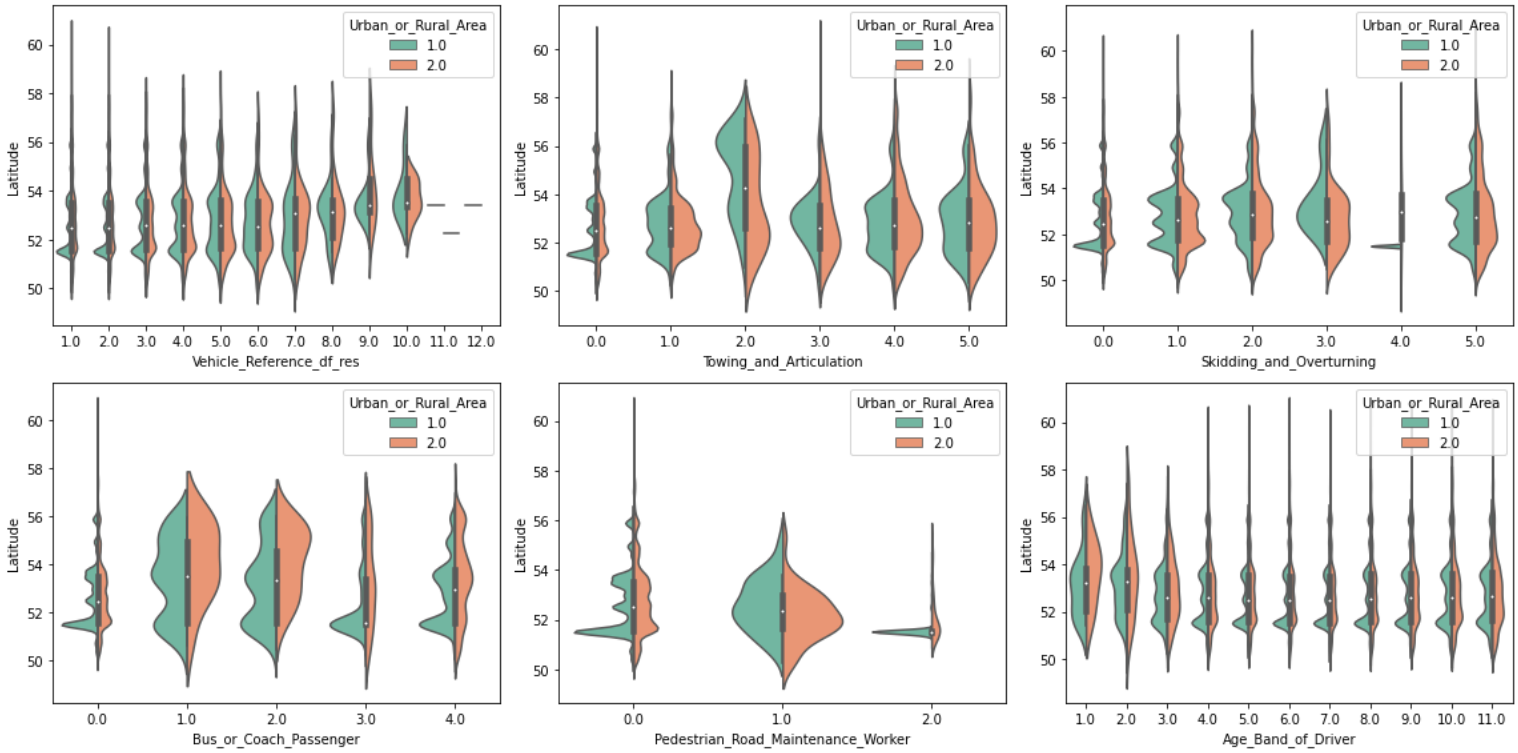
As always, there are multiple way for how we could investigate all of these features. Let's try one example, using seaborn's `stripplot()` together with a handy `zip()` for-loop for subplots.

Note, to spread the values out in the direction of the y-axis we need to chose one particular (hopefully informative) feature. While the 'right' feature can help to identify some interesting patterns, usually any continuous feature should do the trick. The main interest in this kind of plot is to see how many samples each discrete value contains.

feature to stretch the values over the y-axis.



These kind of plots are already very informative, but they obscure regions where there are a lot of data points at once. For example, there seems to be a high density of points in some of the plots at the 52nd latitude. So let's take a closer look with an appropriate plot, such as `violineplot` (or `boxenplot` or `boxplot` for that matter). And to go a step further, let's also separate each visualization by `Urban_or_Rural_Area`.

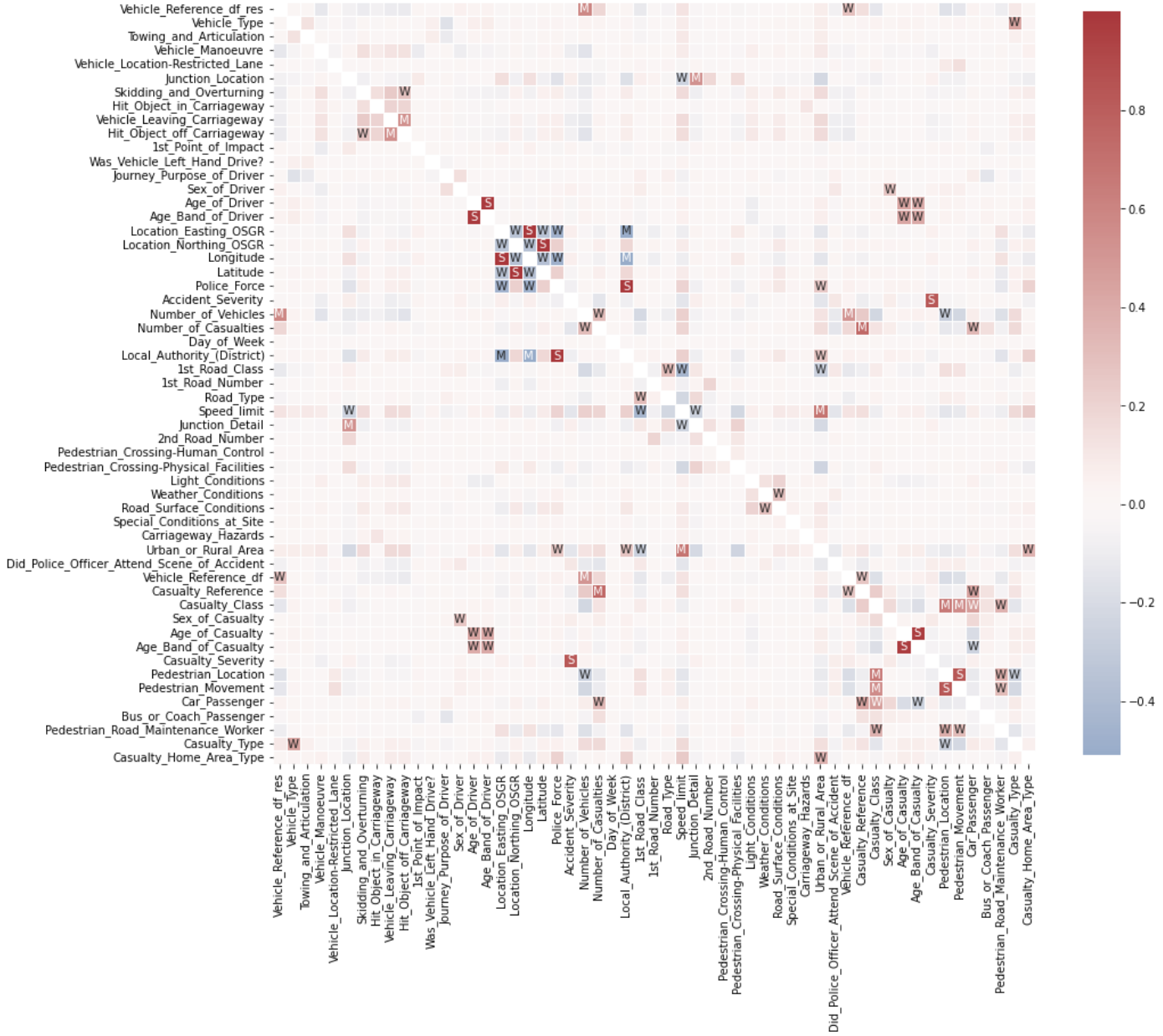


Interesting! We can see that some values on features are more frequent in urban, than in rural areas (and vice versa). Furthermore, as suspected, there seems to be a high density peak at latitude 51.5. This is very likely due to the more densely populated region around London (at 51.5074°).

3.3. Feature relationships

Last, but not least, let's take a look at relationships between features. More precisely how they correlate. The quickest way to do so is via pandas' `.corr()` function. So let's go ahead and compute the feature to feature correlation matrix for all numerical features.

Note: Depending on the dataset and the kind of features (e.g. ordinal or continuous features) you might want to use the `spearman` method instead of the `pearson` method to compute the correlation. Whereas the **Pearson** correlation evaluates the linear relationship between two continuous variables, the **Spearman** correlation evaluates the monotonic relationship based on the ranked values for each feature. And to help with the interpretation of this correlation matrix, let's use seaborn's `.heatmap()` to visualize it.



This looks already very interesting. We can see a few very strong correlations between some of the features. Now, if you're interested actually ordering all of these different correlations, you could do something like this:

As you can see, the investigation of feature correlations can be very informative. But looking at everything at once can sometimes be more confusing than helpful. So focusing only on one feature with something like `df_X.corrwith(df_X["Speed_limit"])` might be a better approach.

Furthermore, correlations can be deceptive if a feature still contains a lot of missing values or extreme outliers. Therefore, it is always important to first make sure that your feature matrix is properly prepared before investigating these correlations.

3.4. Conclusion of content investigation

At the end of this third investigation, we should have a better understanding of the content in our dataset. We looked at value distribution, feature patterns and feature correlations. However, these are certainly not all possible content investigation and data cleaning steps you could do. Additional steps would for example be outlier detection and removal, feature engineering and transformation, and more.

A proper and detailed EDA takes time! It is a very iterative process that often makes you go back to the start, after you addressed another flaw in the dataset. This is normal! It's the reason why we often say that 80% of any data science project is data preparation and EDA.

Take home message

Be mindful that an in-depth EDA can consume a lot of time. And just because something seems interesting doesn't mean that you need to follow up on it. Always remind yourself what the dataset will be used for and tailor your investigations to support that goal. Sometimes it is also ok, to just do a quick-and-dirty data preparation and exploration. This will allow you to move on to the data modeling part rather quickly, and to establish a few preliminary baseline models and perform some informative results investigation.

. . .

Originally published at https://miykael.github.io/blog/2022/advanced_eda/.

Read more from EPFL Extension School