Advanced exploratory data analysis (EDA)

February 1, 2022

2022

How to quickly get a handle on almost any tabular dataset

[Find the Jupyter Notebook 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.

Having said all this, let's dive right into it!

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"]
```

1. Structure Investigation

Before looking at the content of our feature matrix X, let's first look at the general structure of the dataset. For example, how many columns and rows does the dataset have?

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

```
(363243, 67)
```

So we know that this dataset has 363'243 samples and 67 features. And how many different data types do these 67 features contain?

```
import pandas as pd

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

```
float64    61
object      6
dtype: int64
```

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.

```
# Display non-numerical features
df_X.select_dtypes(exclude="number").head()
```

	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. So let's take care of that:

```
# Changes data type of 'Sex_of_Driver'
df_X["Sex_of_Driver"] = df_X["Sex_of_Driver"].astype("float")
```

Using the `.describe()` function we can also investigate how many unique values each non-numerical feature has and with which frequency the most prominent value is present.

```
df_X.describe(exclude="number")
```

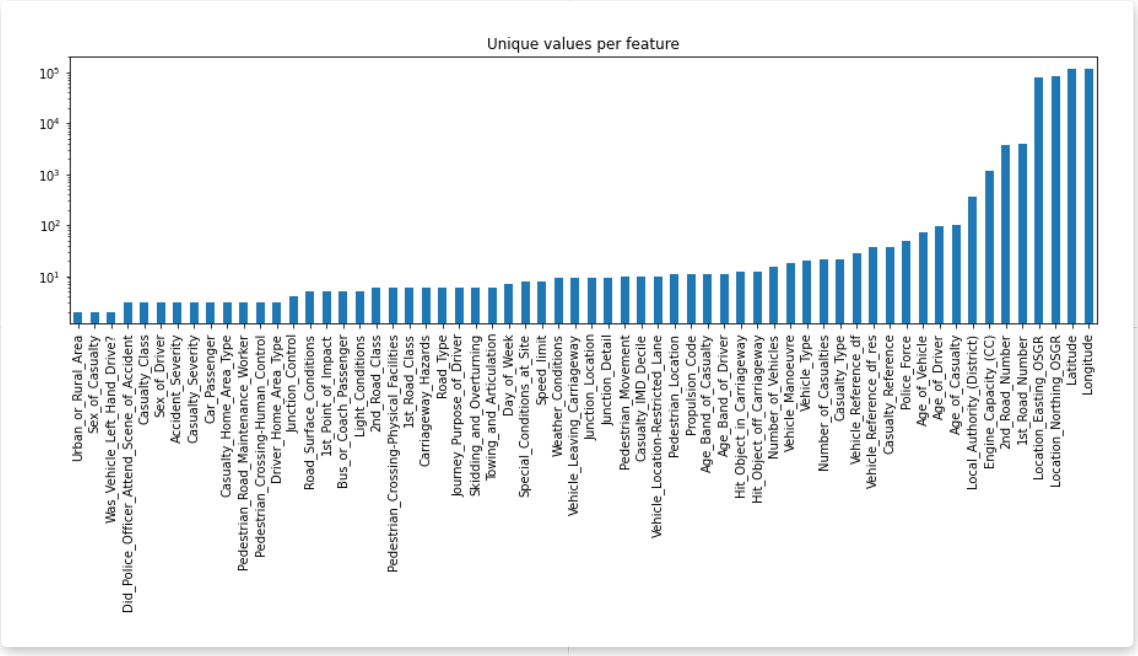
	Accident_Index	Date	Time	Local_Authority_(Highway)	LSOA_of_Accident_Location
count	363243	319866	319822	319866	298758
unique	140056	365	1439	204	25979
top	201543P296025	14/02/2015	17:30	E10000017	E01028497
freq	1332	2144	2972	8457	1456

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 feature");
```



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 (e.g. an index number or recording time that is unique to each new sample) which you might want to ignore first.

```
# Check number of duplicates while ignoring the index 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.
```

To handle these duplicates you can just simply drop them with `.drop_duplicates()`.

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

# Drop duplicates based on 'columns_to_consider'
df_X = df_X.drop_duplicates(subset=columns_to_consider)
df_X.shape
```

```
(363221, 67)
```

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: