Speed up your Data Cleaning and Preprocessing with klib

Customized and very easily applicable functions with sensible default values

Andreas Kanz Aug 6, 2020 · 8 min read ★

Image by Author

TL;DR
The klib package provides a number of very easily applicable functions with sensible default values that can be used on virtually any DataFrame to assess data quality, gain insight, perform cleaning operations and visualizations which results in a much lighter and more convenient to work with Pandas DataFrame.

Over the past couple of months I've implemented a range of functions which I frequently use for virtually any data analysis and preprocessing task, irrespective of the dataset or ultimate goal.

These functions require nothing but a Pandas DataFrame of any size and any datatypes and can be accessed through simple one line calls to gain insight into your data, clean up your DataFrames and visualize relationships between features. It is up to you if you stick to sensible, yet sometimes conservative, default parameters or customize the experience by adjusting them according to your needs.

This package is not meant to provide an Auto-ML style API. Rather it is a collection of functions which you can — and probably should — call every time you start working on a new project or dataset. Not only for your own understanding of what you are dealing with, but also to produce plots you can show to supervisors, customers or anyone else looking to get a higher level representation and explanation of the data.

Installation Instructions

Install klib using pip:

```
pip install --upgrade klib
```

Alternatively, to install with conda run:

```
conda install -c conda-forge klib
```

What follows is a workflow and set of best practices which I repeatedly apply when facing new datasets.

Quick Outline

- Assessing Data Quality
- Data Cleaning
- Visualizing Relationships

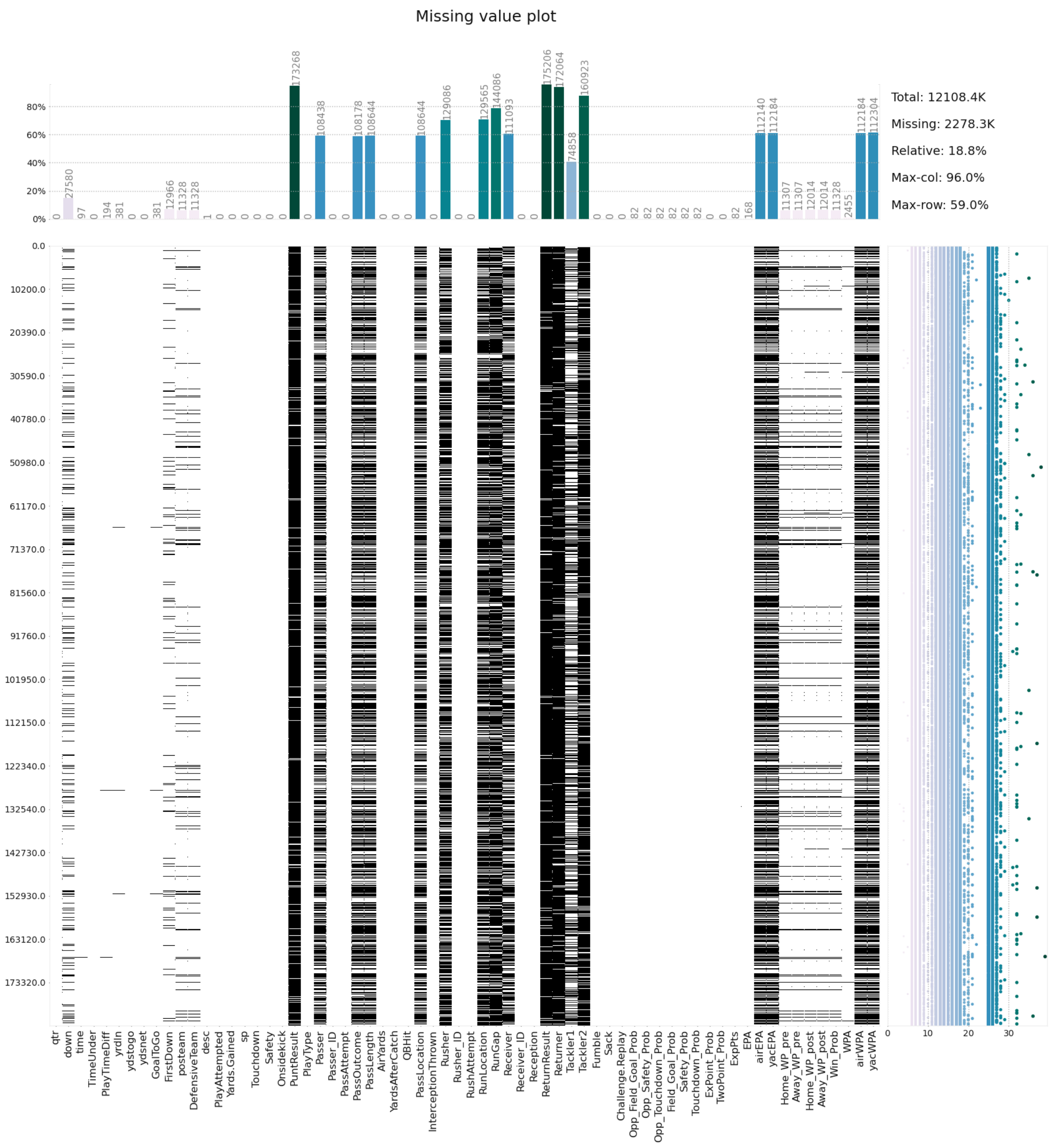
The data used in this guide is a slightly truncated version of the NFL Dataset found on Kaggle. You can download it here or use any arbitrary data you want to follow along.

Assessing the Data Quality

Determining data quality before starting to work on a dataset is crucial. A quick way to achieve that is to use the missing value visualization of klib, which can be called as easy as follows:

```python
import klib
import pandas as pd

df = pd.read_csv("NFL_DATASET.csv")
klib.missingval_plot(df)
```

klib_missing_val_plot hosted with ♡ by GitHub view raw



Default representation of missing values

This single plot already shows us a number of important things. Firstly, we can identify columns where all or most of the values are missing. These are candidates for dropping, while those with fewer missing values might benefit from imputation.

Secondly, we can often times see patterns of missing rows stretching across many features. We might want to eliminate them first before thinking about dropping potentially relevant features.

And lastly, the additional statistics at the top and the right side give us valuable information regarding thresholds we can use for dropping rows or columns with many missing values. In our example we can see that if we drop rows with more than 30 missing values, we only lose a few entries. At the same time, if we eliminate columns with missing values larger than 80% the four most affected columns are removed.

A quick note on performance: Despite going through about 2 million entries with 66 features each, the plot takes only seconds to create.

Data Cleaning

With this insight, we can go ahead and start cleaning the data. With klib this is as simple as calling *klib.data_cleaning()*, which performs the following operations:

- **cleaning the column names:**
  This unifies the column names by formatting them, splitting, among others, CamelCase into camel_case, removing special characters as well as leading and trailing white-spaces and formatting all column names to *lowercase_and_underscore_separated*. This also checks for and fixes duplicate column names, which you sometimes get when reading data from a file.
- **dropping empty and virtually empty columns:**
  You can use the parameters *drop_threshold_cols* and *drop_threshold_rows* to adjust the dropping to your needs. The default is to drop columns and rows with more than 90% of the values missing.
- **removes single valued columns:**
  As the name states, this removes columns in which each cell contains the same value. This comes in handy when columns such as "year" are included while you're just looking at a single year. Other examples are "download_date" or indicator variables which are identical for all entries.
- **drops duplicate rows:**
  This is a straightforward drop of entirely duplicate rows. If you are dealing with data where duplicates add value, consider setting *drop_duplicates=False.*
- Lastly, and often times most importantly, especially for **memory reduction** and therefore for speeding up the subsequent steps in your workflow, *klib.data_cleaning()* also **optimizes the datatypes** as we can see below.

```
df_cleaned = klib.data_cleaning(df)
```

klib_data_cleaning hosted with ♡ by GitHub view raw

```
Shape of cleaned data: (183337, 62) - Remaining NAs: 1754608

Changes:
Dropped rows: 123
     of which 123 duplicates. (Rows: [22257, 25347, 26631, 30310,
33558, 35164, 35777, ..., 182935, 182942, 183058, 183368, 183369])
Dropped columns: 4
     of which 1 single valued. (Column: ['play_attempted'])
Dropped missing values: 523377
Reduced memory by at least: 63.69 MB (-68.94%)
```

You can change the verbosity of the output using the parameter *show=None, show="changes" or show="all"*. Please note that the memory reduction indicates a very conservative value (i.e. less reduction than is actually achieved), as it only performs a shallow memory check. A deep memory analysis slows down the function for larger datasets but if you are curious about the "true" reduction in size you can use the *df.info()* method as shown below.

```
df.info(memory_usage='deep')

dtypes: float64(25), int64(20), object(21)
memory usage: 256.7 MB
```

As we can see, pandas assigns 64 bits of storage for each float and int. Additionally, 21 columns are of type "object", which is a rather inefficient way to store data. After data cleaning, the **memory usage drops to only 58.4 MB, a reduction of almost 80%!** This is achieved by converting, where possible, *float64* to *float32*, and *int64* to *int8*. Also, the dtypes *string* and *category* are utilized. The available parameters such as *convert_dtypes, category*, *cat_threshold* and many more allow you to tune the function to your needs.

```
df_cleaned.info(memory_usage='deep')

dtypes: category(17), float32(25), int8(19), string(1)
memory usage: 58.4 MB
```

Lastly, we take a look at the column names, which were actually quite well formatted in the original dataset already. However, **after the cleaning process, you can rely on lowercase and underscore-connected column names**. While not advisable to avoid ambiguity, this now allows you to use *df.yards_gained* instead of *df["Yards.Gained"]*, which can be really useful when doing quick lookups or when exploring the data for the first time.

```
Some column name examples:
Yards.Gained --> yards_gained
PlayAttempted --> play_attempted
Challenge.Replay --> challenge_replay
```

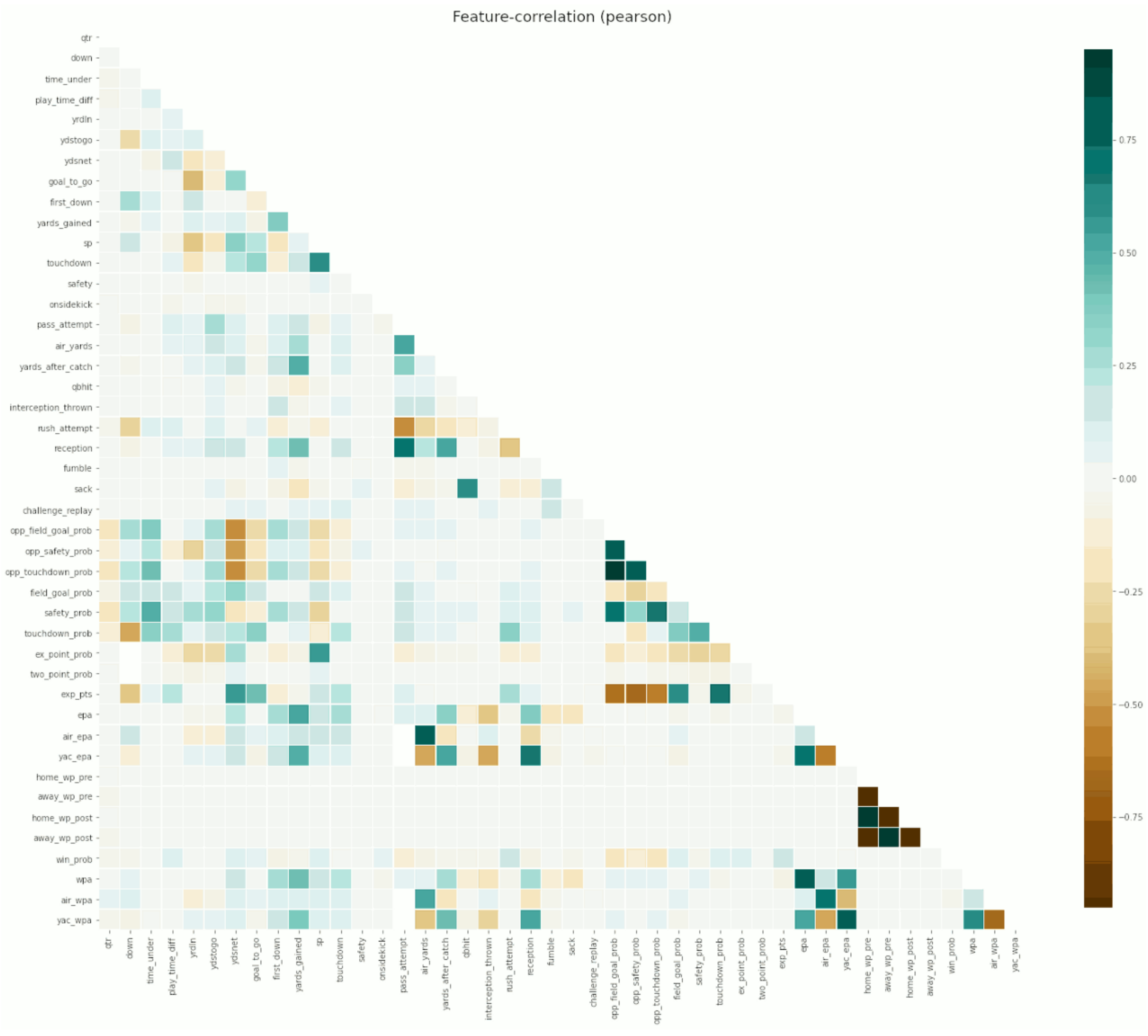
Ultimately, and to sum it all up: we find that not only have the column names been neatly formatted and unified, but also that the features have been converted to more efficient datatypes. With the relatively milde default settings, only 123 rows and 4 columns, of which one column was singled valued, have been eliminated. This leaves us with a lightweight DataFrame of shape: (183337, 62) and 58 MB memory usage.

Correlation Plots

Once the initial data cleaning is done, it makes sense to take a look at the relationships between the features. For this we employ the function **klib.corr_plot()**. Setting the *split* parameter to *"pos"*, *"neg"*, *"high"* or *"low"* and optionally combining each setting with a *threshold*, allows us to dig deeper and highlight the most important aspects.

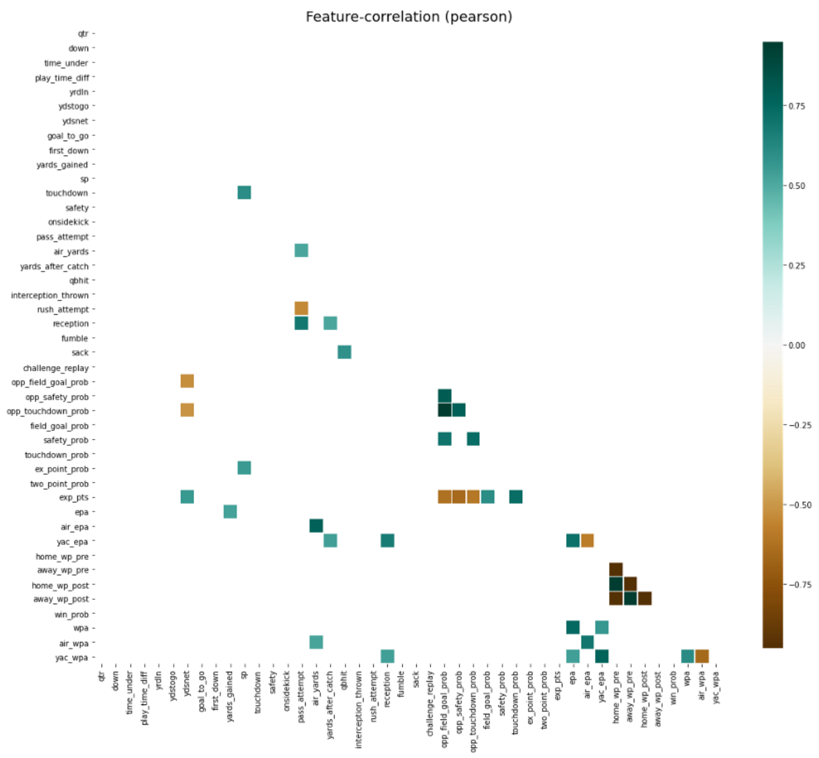
```
The gif below shows the output of the following three function calls:

klib.corr_plot(df_cleaned, annot=False, figsize=(15,12))
klib.corr_plot(df_cleaned, split='pos', annot=False, figsize=(15,12))
klib.corr_plot(df_cleaned, split='neg', annot=False, figsize=(15,12))
```

klib_corr_plot hosted with ♡ by GitHub view raw



Correlation plots

At a glance, we can identify a number of interesting relations. Similarly, we can easily zoom in on correlations above any given threshold, let’s say |0.5|. Not only does this allow us to spot features which might be causing trouble later on in our analysis, it also shows us that there are quite a few highly negatively correlated features in our data. Given sufficient domain expertise, this can be a great starting point for some feature engineering!

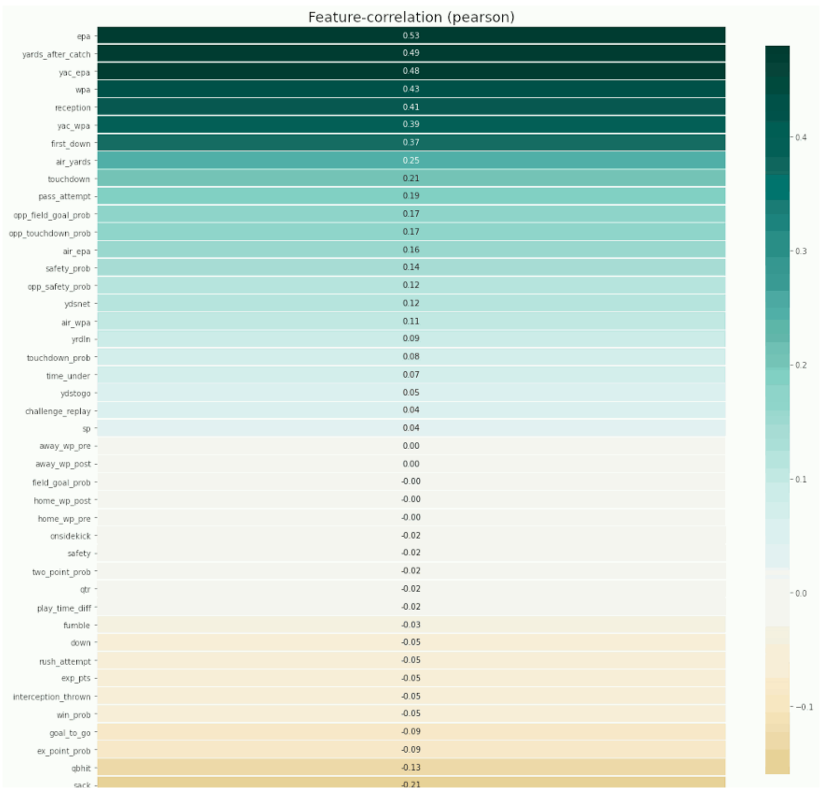


Plot of high absolute correlations

Further, using the same function, we can take a look at the correlations between features and a chosen target. The target column can be supplied as a column name of the current DataFrame, as a separate pd.Series, a np.ndarry or simply as a list.

```
The gif below shows the output of the following three function calls:

klib.corr_plot(df_cleaned, target='yards_gained')
klib.corr_plot(df_cleaned, target='yards_gained', method="spearman")
klib.corr_plot(df_cleaned, target='yards_gained', method="kendall")
```

klib_feature_corr_plot() hosted with ♡ by GitHub — view raw



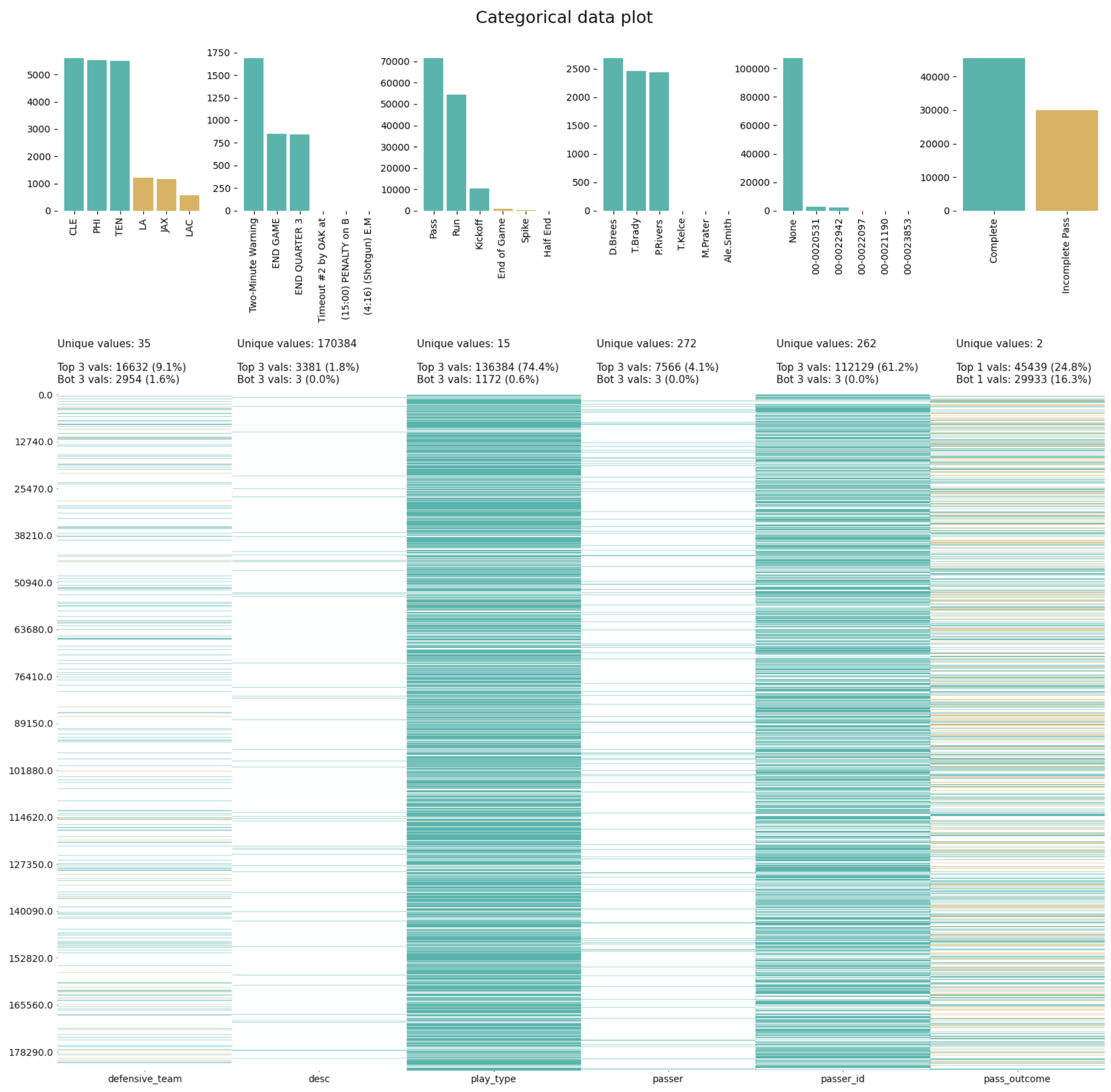
Plot of correlations with the target / label

Just as before it is possible to use a wide range of parameters for customizations, such as removing annotations, changing the correlation method or changing the colormap to match your preferred style or corporate identity.

Categorical data

In a last step in this guide, we take a quick look at the capabilities to visualize categorical columns. The function **klib.cat_plot()** allows to display the top and/or bottom values regarding their frequency in each column. This gives us an idea of the distribution of the values in the dataset what is very helpful when considering to combine less frequent values into a seperate category before applying one-hot-encoding or similar functions. In this example we can see that for the column “play_type” roughly 75% of all entries are made up of the three most frequent values. Further, we can immediately see that “Pass” and “Run” are by far the most frequent values (75k and 55k). Conversely, the plot also shows us that “desc” is made up of 170384 unique strings.

```
klib.cat_plot(df_cleaned)
```



Categorical data plot

The klib package includes many more helpful functions for data analysis and cleaning, not to mention some customized sklearn pipelines, which you can easily stack together using a FeatureUnion and then use with in GridSearchCV or similar. So if you intend to take a shortcut, simply call klib.data_cleaning() and plug the resulting DataFrame into that pipeline. Likely, you will already get a very decent result!

Conclusion

All of these functions make for a very convenient data cleaning and visualization and come with many more features and settings than described here. They are by no means a one fits all solution but they should be very helpful in your data preparation process. klib also includes various other functions, most notably **pool_duplicate_subsets()**, to pool subsets of the data across different features as a means of dimensionality reduction, **dist_plot()**, to visualize distributions of numerical features, as well as **mv_col_handling()**, which provides a sophisticated 3-step process, attempting to identify any remaining information in columns with many missing values, instead of simply dropping them right away.

Data preparation with klib
Fast and simple function calls for efficient data preparation
towardsdatascience.com

> *Note: Please let me know what you would like to see next and which functions you feel are missing, either in the comments below or by opening an issue on GitHub. Also let me know if you would like to see some examples on the handling of missing values, subset pooling or the customized sklearn pipelines.*

Pandas Data Visualization Data Science Data Cleaning Editors Pick