Get started Open in app

Follow 563K Followers

You have **2** free member-only stories left this month. Sign up for Medium and get an extra one

Data Cleaning in Python: the Ultimate Guide (2020)

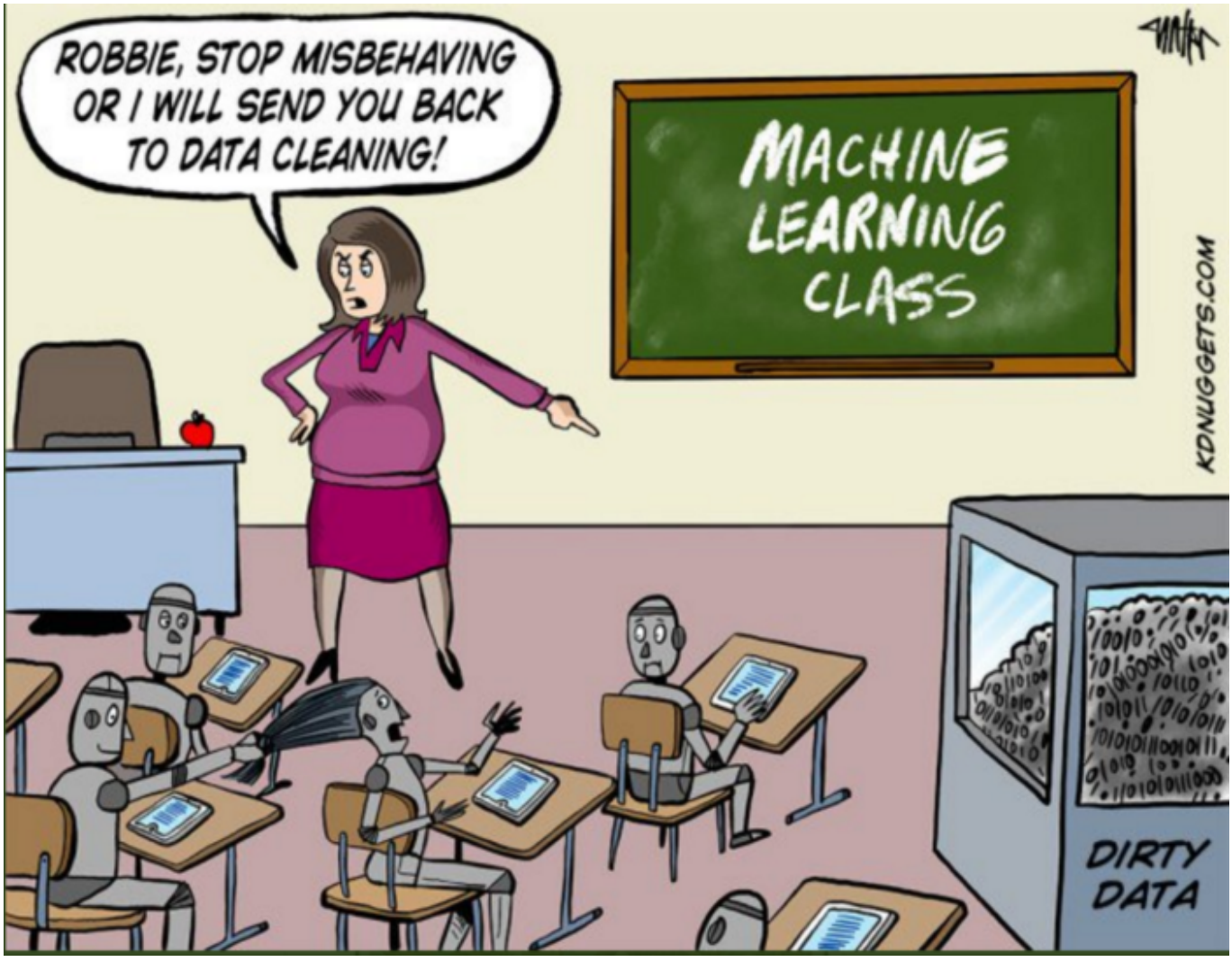
Techniques on what to clean and how.

Lianne & Justin @ Just into Data Feb 3, 2020 · 12 min read ★

Source: Pixabay

> ***Data cleaning or cleansing*** *is the process of detecting and correcting (or removing) corrupt or inaccurate records from a record set, table, or database and refers to identifying incomplete, incorrect, inaccurate or irrelevant parts of the data and then replacing, modifying, or deleting the dirty or coarse data.*

What a long definition! It is certainly not fun and very time-consuming.



Source: kdnuggets.com

To make it *easier*, we created this new complete step-by-step guide in Python. You'll learn techniques on *how to find and clean*:

- Missing Data

- Unnecessary Data — Repetitive Data, Duplicates and more
- Inconsistent Data — Capitalization, Addresses and more

Within this guide, we use the Russian housing dataset from Kaggle. The goal of this project is to predict housing price fluctuations in Russia. We are not cleaning the entire dataset but will show examples from it.

Before we jump into the cleaning process, let's take a brief look at the data.

```
# import packages
import pandas as pd
import numpy as np
import seaborn as sns

import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
import matplotlib
plt.style.use('ggplot')
from matplotlib.pyplot import figure

%matplotlib inline
matplotlib.rcParams['figure.figsize'] = (12,8)

pd.options.mode.chained_assignment = None



# read the data
df = pd.read_csv('sberbank.csv')

# shape and data types of the data
print(df.shape)
print(df.dtypes)

# select numeric columns
df_numeric = df.select_dtypes(include=[np.number])
```

```
# select non numeric columns
df_non_numeric = df.select_dtypes(exclude=[np.number])
non_numeric_cols = df_non_numeric.columns.values
print(non_numeric_cols)
```

read_explore_data.py hosted with ♡ by GitHub view raw

From these results, we learn that the dataset has 30,471 rows and 292 columns. We also identify whether the features are numeric or categorical variables. These are all useful information.

Now we can run through the checklist of “dirty” data types and fix them one by one.

Let’s get started.

Source: GIPHY

Missing data

don't accept missing data.

How to find out?

We cover three techniques to learn more about missing data in our dataset.

- **Technique #1: Missing Data Heatmap**

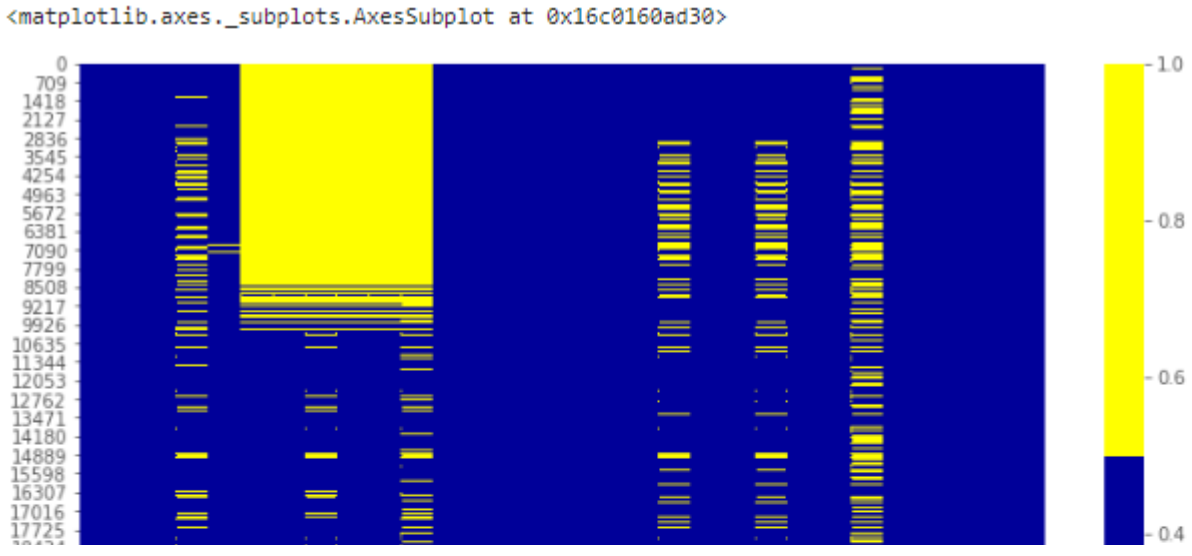
When there is a smaller number of features, we can visualize the missing data via heatmap.

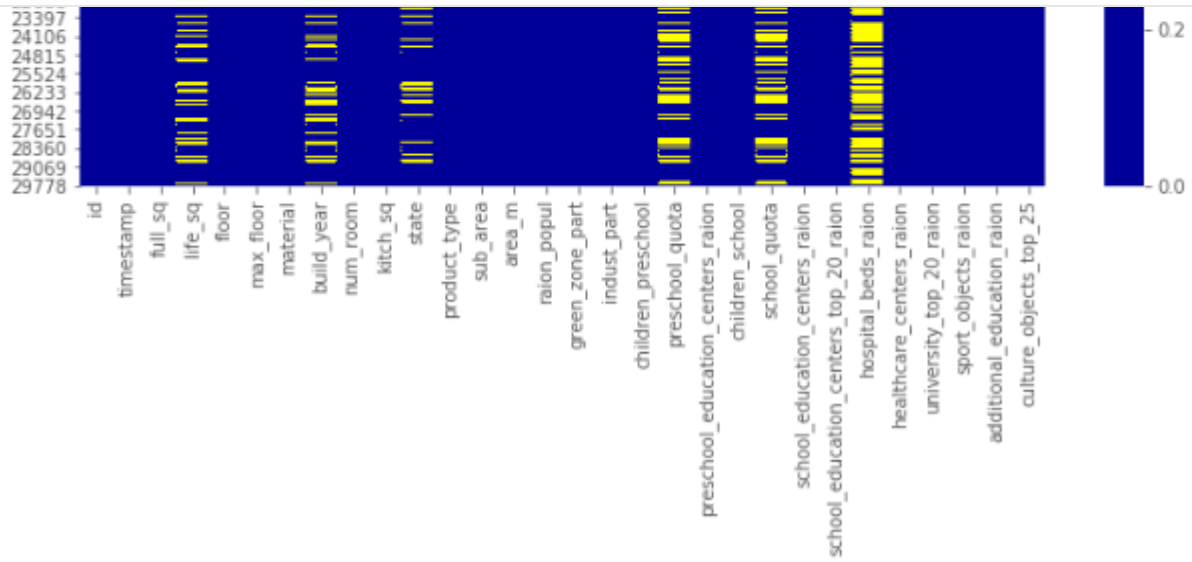
```
cols = df.columns[:30] # first 30 columns
colours = ['#000099', '#ffff00'] # specify the colours - yellow is missing. blue is not missing.
sns.heatmap(df[cols].isnull(), cmap=sns.color_palette(colours))
```

missing_data_find1.py hosted with ♡ by GitHub view raw

The chart below demonstrates the missing data patterns of the first 30 features. The horizontal axis shows the feature name; the vertical axis shows the number of observations/rows; the yellow color represents the missing data while the blue color otherwise.

For example, we see that the *life_sq* feature has missing values throughout many rows. While the *floor* feature only has little missing values around the 7000th row.





Missing Data Heatmap

- **Technique #2: Missing Data Percentage List**

When there are many features in the dataset, we can make a list of missing data % for each feature.

```
# if it's a larger dataset and the visualization takes too long can do this.
# % of missing.
for col in df.columns:
    pct_missing = np.mean(df[col].isnull())
    print('{} - {}%'.format(col, round(pct_missing*100)))
```

missing_data_find2.py hosted with ♡ by GitHub view raw

This produces a list below showing the percentage of missing values for each of the features.

Specifically, we see that the *life_sq* feature has 21% missing, while *floor* has only 1% missing. This list is a useful summary that can complement the heatmap visualization.

```
id - 0.0%
timestamp - 0.0%
full_sq - 0.0%
life_sq - 21.0%
floor - 1.0%
max_floor - 31.0%
```

```
state - 44.0%
product_type - 0.0%
sub_area - 0.0%
area_m - 0.0%
raion_popul - 0.0%
green_zone_part - 0.0%
indust_part - 0.0%
children_preschool - 0.0%
preschool_quota - 22.0%
preschool_education_centers_raion - 0.0%
children_school - 0.0%
school_quota - 22.0%
school_education_centers_raion - 0.0%
school_education_centers_top_20_raion - 0.0%
hospital_beds_raion - 47.0%
healthcare_centers_raion - 0.0%
university_top_20_raion - 0.0%
sport_objects_raion - 0.0%
additional_education_raion - 0.0%
culture_objects_top_25 - 0.0%
```

Missing Data % List — the first 30 features

- **Technique #3: Missing Data Histogram**

Missing data histogram is also a technique for when we have many features.

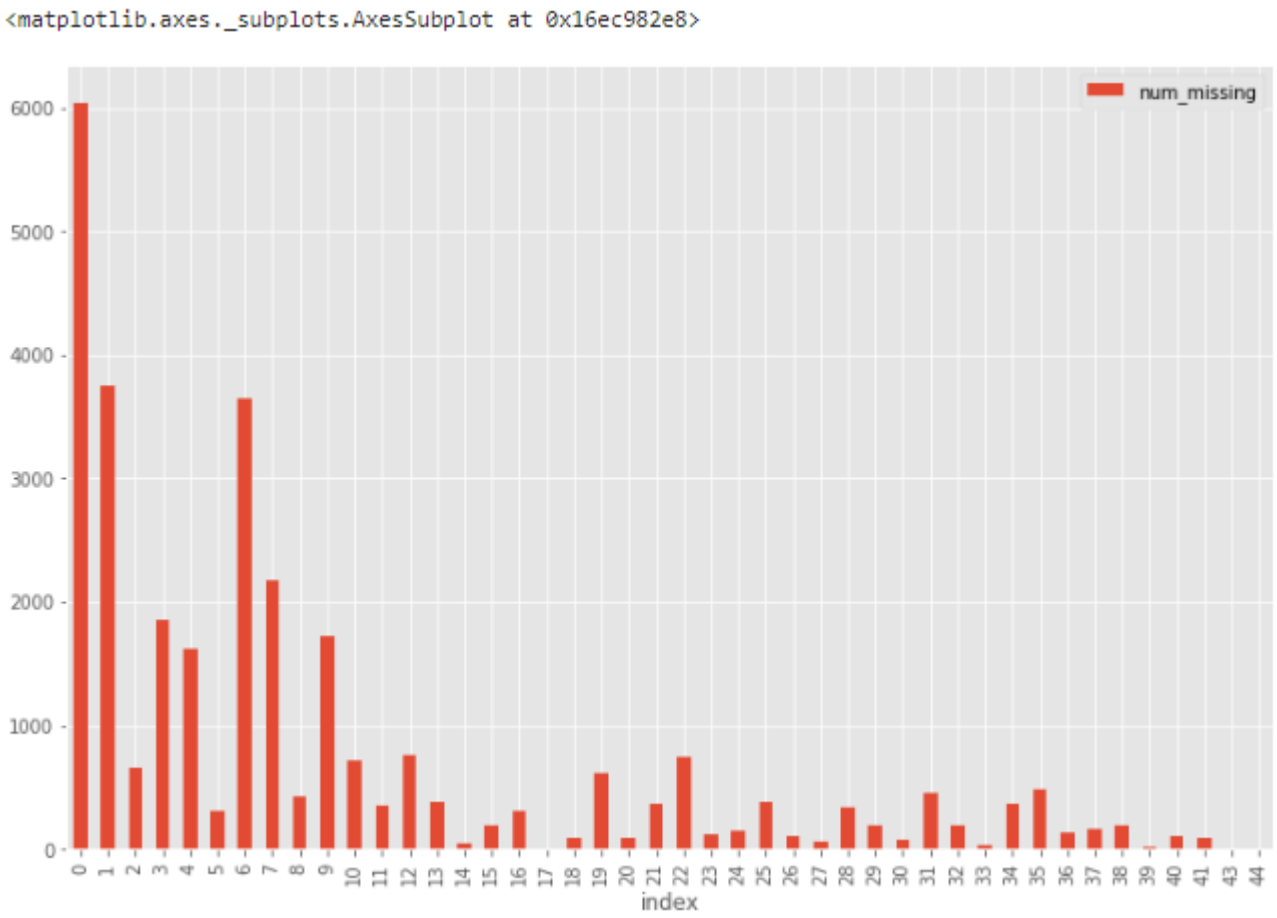
To learn more about the missing value patterns among observations, we can visualize it by a histogram.

```python
# first create missing indicator for features with missing data
for col in df.columns:
    missing = df[col].isnull()
    num_missing = np.sum(missing)

    if num_missing > 0:
        print('created missing indicator for: {}'.format(col))
        df['{}_ismissing'.format(col)] = missing


# then based on the indicator, plot the histogram of missing values
ismissing_cols = [col for col in df.columns if 'ismissing' in col]
df['num_missing'] = df[ismissing_cols].sum(axis=1)

df['num_missing'].value_counts().reset_index().sort_values(by='index').plot.bar(x='index', y='nu
```

missing_data_dropping1.py hosted with ♡ by GitHub view raw

observations.

For example, there are over 6000 observations with no missing values and close to 4000 observations with one missing value.



Missing Data Histogram

What to do?

There are *NO* agreed-upon solutions to dealing with missing data. We have to study the specific feature and dataset to decide the best way of handling them.

Below covers the four most common methods of handling missing data. But, if the situation is more complicated than usual, we need to be creative to use more sophisticated methods such as missing data modeling.

- **Solution #1: Drop the Observation**

Only if we are sure that the missing data is not informative, we perform this. Otherwise, we should consider other solutions.

There could be other criteria to use to drop the observations.

For example, from the missing data histogram, we notice that only a minimal amount of observations have over 35 features missing altogether. We may create a new dataset *df_less_missing_rows* deleting observations with over 35 missing features.

```
# drop rows with a lot of missing values.
ind_missing = df[df['num_missing'] > 35].index
df_less_missing_rows = df.drop(ind_missing, axis=0)
```

missing_data_dropping2.py hosted with ♡ by GitHub view raw

- **Solution #2: Drop the Feature**

Similar to Solution #1, we *only* do this when we are confident that this feature doesn't provide useful information.

For example, from the missing data % list, we notice that *hospital_beds_raion* has a high missing value percentage of 47%. We may drop the entire feature.

```
# hospital_beds_raion has a lot of missing.
# If we want to drop.
cols_to_drop = ['hospital_beds_raion']
df_less_hos_beds_raion = df.drop(cols_to_drop, axis=1)
```

missing_data_dropping3.py hosted with ♡ by GitHub view raw

- **Solution #3: Impute the Missing**

When the feature is a numeric variable, we can conduct missing data imputation. We replace the missing values with the average or median value from the data of the same feature that is not missing.

Using *life_sq* as an example, we can replace the missing values of this feature by its median.

```
# replace missing values with the median.
med = df['life_sq'].median()
print(med)
df['life_sq'] = df['life_sq'].fillna(med)
```

missing_data_imputation1.py hosted with ♡ by GitHub view raw

Moreover, we can apply the same imputation strategy for all the numeric features at once.

```
# impute the missing values and create the missing value indicator variables for each numeric co
df_numeric = df.select_dtypes(include=[np.number])
numeric_cols = df_numeric.columns.values

for col in numeric_cols:
    missing = df[col].isnull()
    num_missing = np.sum(missing)

    if num_missing > 0:  # only do the imputation for the columns that have missing values.
        print('imputing missing values for: {}'.format(col))
        df['{}_ismissing'.format(col)] = missing
        med = df[col].median()
        df[col] = df[col].fillna(med)
```

missing_data_imputation2.py hosted with ♡ by GitHub view raw

```
imputing missing values for: floor
imputing missing values for: max_floor
imputing missing values for: material
imputing missing values for: build_year
imputing missing values for: num_room
imputing missing values for: kitch_sq
imputing missing values for: state
imputing missing values for: preschool_quota
imputing missing values for: school_quota
imputing missing values for: hospital_beds_raion
imputing missing values for: raion_build_count_with_material_info
imputing missing values for: build_count_block
imputing missing values for: build_count_wood
```

```
imputing missing values for: build_count_foam
imputing missing values for: build_count_slag
imputing missing values for: build_count_mix
imputing missing values for: raion_build_count_with_builddate_info
imputing missing values for: build_count_before_1920
imputing missing values for: build_count_1921-1945
imputing missing values for: build_count_1946-1970
imputing missing values for: build_count_1971-1995
imputing missing values for: build_count_after_1995
imputing missing values for: metro_min_walk
imputing missing values for: metro_km_walk
imputing missing values for: railroad_station_walk_km
imputing missing values for: railroad_station_walk_min
imputing missing values for: ID_railroad_station_walk
imputing missing values for: cafe_sum_500_min_price_avg
imputing missing values for: cafe_sum_500_max_price_avg
imputing missing values for: cafe_avg_price_500
imputing missing values for: cafe_sum_1000_min_price_avg
imputing missing values for: cafe_sum_1000_max_price_avg
imputing missing values for: cafe_avg_price_1000
imputing missing values for: cafe_sum_1500_min_price_avg
imputing missing values for: cafe_sum_1500_max_price_avg
imputing missing values for: cafe_avg_price_1500
imputing missing values for: cafe_sum_2000_min_price_avg
imputing missing values for: cafe_sum_2000_max_price_avg
imputing missing values for: cafe_avg_price_2000
imputing missing values for: cafe_sum_3000_min_price_avg
imputing missing values for: cafe_sum_3000_max_price_avg
imputing missing values for: cafe_avg_price_3000
imputing missing values for: prom_part_5000
imputing missing values for: cafe_sum_5000_min_price_avg
imputing missing values for: cafe_sum_5000_max_price_avg
imputing missing values for: cafe_avg_price_5000
```

Luckily, our dataset has no missing value for categorical features. Yet, we can apply the mode imputation strategy for all the categorical features at once.

```
# impute the missing values and create the missing value indicator variables for each non-numeri
df_non_numeric = df.select_dtypes(exclude=[np.number])
non_numeric_cols = df_non_numeric.columns.values

for col in non_numeric_cols:
    missing = df[col].isnull()
    num_missing = np.sum(missing)

    if num_missing > 0:  # only do the imputation for the columns that have missing values.
        print('imputing missing values for: {}'.format(col))
        df['{}_ismissing'.format(col)] = missing

        top = df[col].describe()['top'] # impute with the most frequent value.
        df[col] = df[col].fillna(top)
```

- **Solution #4: Replace the Missing**

For categorical features, we can add a new category with a value such as "_MISSING_". For numerical features, we can replace it with a particular value such as -999.

This way, we are still keeping the missing values as valuable information.

```
# categorical
df['sub_area'] = df['sub_area'].fillna('_MISSING_')


# numeric
df['life_sq'] = df['life_sq'].fillna(-999)
```

missing_data_replace.py hosted with ♡ by GitHub view raw

Irregular data (Outliers)

Outliers are data that is *distinctively* different from other observations. They could be real outliers or mistakes.

How to find out?

Depending on whether the feature is numeric or categorical, we can use different techniques to study its distribution to detect outliers.

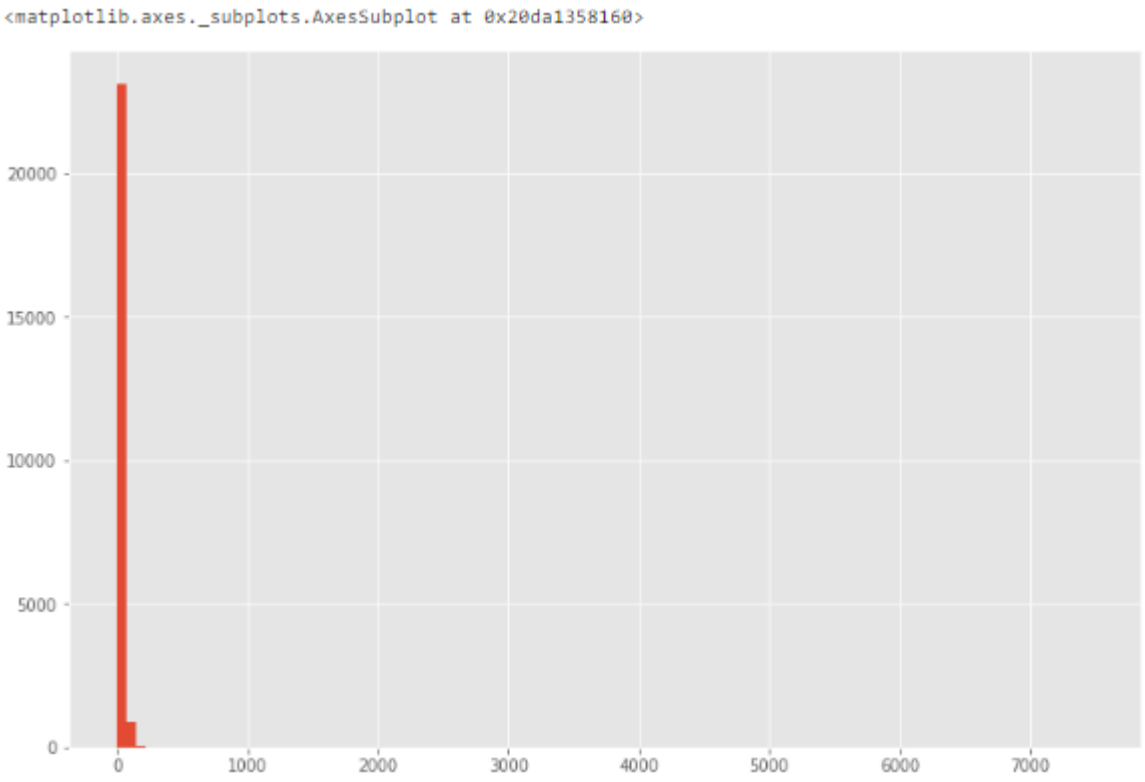
- **Technique #1: Histogram/Box Plot**

When the feature is numeric, we can use a histogram and box plot to detect outliers.

Below is the histogram of feature *life_sq*.

```
# histogram of life_sq.
df['life_sq'].hist(bins=100)
```

The data looks highly skewed with the possible existence of outliers.



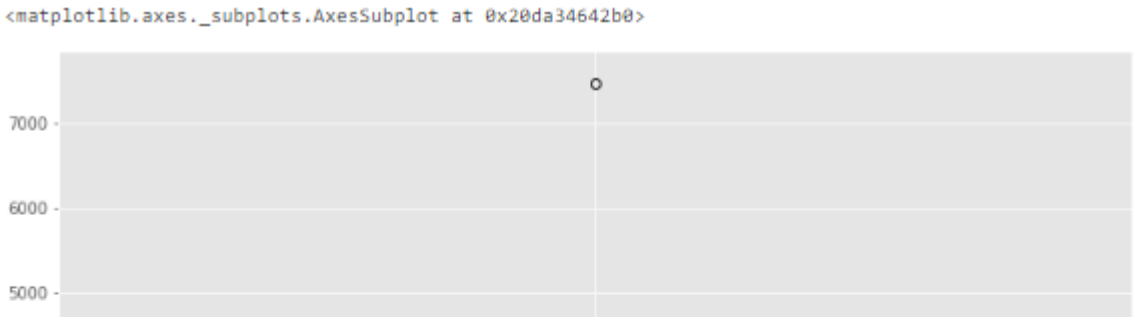
Histogram

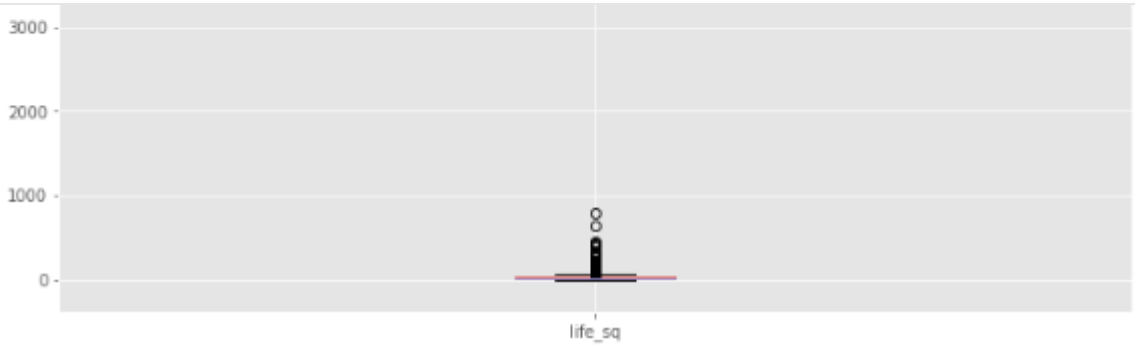
To study the feature closer, let's make a box plot.

```
1  # box plot.
2  df.boxplot(column=['life_sq'])
```

outlier_boxplot.py hosted with ♡ by GitHub view raw

In this plot, we can see there is an outlier at a value of over 7000.





Box Plot

- **Technique #2: Descriptive Statistics**

Also, for numeric features, the outliers could be too distinct that the box plot can't visualize them. Instead, we can look at their descriptive statistics.

For example, for the feature *life_sq* again, we can see that the maximum value is 7478, while the 75% quartile is only 43. The 7478 value is an outlier.

```
1  df['life_sq'].describe()
```

outlier_describe.py hosted with ♡ by GitHub view raw

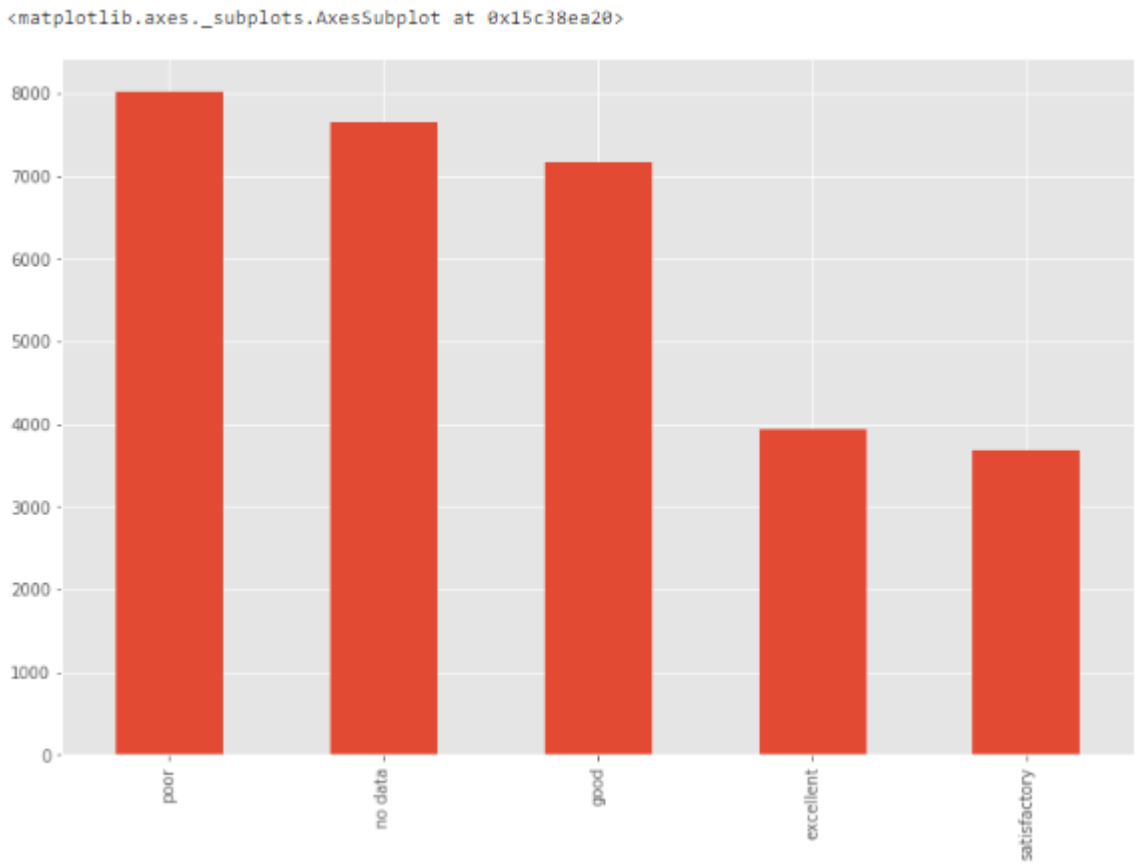
```
count    24088.000000
mean        34.403271
std         52.285733
min          0.000000
25%         20.000000
50%         30.000000
75%         43.000000
max       7478.000000
Name: life_sq, dtype: float64
```

- **Technique #3: Bar Chart**

When the feature is categorical. We can use a bar chart to learn about its categories and distribution.

For example, the feature *ecology* has a reasonable distribution. But if there is a category with only one value called "other", then that would be an outlier.

```
1  # bar chart - distribution of a categorical variable
```



Bar Chart

- **Other Techniques:** Many other techniques can spot outliers as well, such as scatter plot, z-score, and clustering. This article does not cover all of those.

What to do?

While outliers are not hard to detect, we have to determine the right solutions to handle them. It highly depends on the dataset and the goal of the project.

The methods of handling outliers are somewhat similar to missing data. We either drop or adjust or keep them. We can refer back to the missing data section for possible solutions.

which is more straightforward.

All the data feeding into the model should serve the purpose of the project. The unnecessary data is when the data doesn't add value. We cover three main types of unnecessary data due to different reasons.

Unnecessary type #1: Uninformative / Repetitive

Sometimes one feature is uninformative because it has too many rows being the same value.

How to find out?

We can create a list of features with a high percentage of the same value.

For example, we specify below to show features with over 95% rows being the same value.

```
num_rows = len(df.index)
low_information_cols = [] #

for col in df.columns:
    cnts = df[col].value_counts(dropna=False)
    top_pct = (cnts/num_rows).iloc[0]

    if top_pct > 0.95:
        low_information_cols.append(col)
        print('{0}: {1:.5f}%'.format(col, top_pct*100))
        print(cnts)
        print()
```

irrelevant_data.py hosted with ♡ by GitHub view raw

We can look into these variables one by one to see whether they are informative or not. We won't show the details here.

```
oil_chemistry_raion: 99.02858%
no     30175
yes      296
Name: oil_chemistry_raion, dtype: int64
```

```
nuclear_reactor_raion: 97.16780%
no     29608
yes      863
Name: nuclear_reactor_raion, dtype: int64

big_road1_1line: 97.43691%
no     29690
yes      781
Name: big_road1_1line, dtype: int64

railroad_1line: 97.06934%
no     29578
yes      893
Name: railroad_1line, dtype: int64

cafe_count_500_price_high: 97.25641%
0    29635
1      787
2       38
3       11
Name: cafe_count_500_price_high, dtype: int64

mosque_count_500: 99.51101%
0    30322
1      149
Name: mosque_count_500, dtype: int64

cafe_count_1000_price_high: 95.52689%
0    29108
1     1104
2      145
3       51
4       39
5       15
6        8
7        1
Name: cafe_count_1000_price_high, dtype: int64

mosque_count_1000: 98.08342%
0    29887
1      584
Name: mosque_count_1000, dtype: int64

mosque_count_1500: 96.21936%
0    29319
1     1152
Name: mosque_count_1500, dtype: int64
```

What to do?

We need to understand the reasons behind the repetitive feature. When they are genuinely uninformative, we can toss them out.

Unnecessary type #2: Irrelevant

Again, the data needs to provide valuable information for the project. If the features are not related to the question we are trying to solve in the project, they are irrelevant.

How to find out?

For example, a feature recording the temperature in Toronto doesn't provide any useful insights to predict Russian housing prices.

What to do?

When the features are not serving the project's goal, we can remove them.

Unnecessary type #3: Duplicates

The duplicate data is when copies of the same observation exist.

There are two main types of duplicate data.

- **Duplicates type #1: All Features based**

How to find out?

This duplicate happens when all the features' values within the observations are the same. It is easy to find.

We first remove the unique identifier *id* in the dataset. Then we create a dataset called *df_dedupped* by dropping the duplicates. We compare the shapes of the two datasets (df and df_dedupped) to find out the number of duplicated rows.

10 rows are being complete duplicate observations.

```
(30471, 344)
(30461, 343)
```

What to do?

We should remove these duplicates, which we already did.

- **Duplicates type #2: Key Features based**

How to find out?

Sometimes it is better to remove duplicate data based on a set of unique identifiers.

We can set up a group of critical features as unique identifiers for transactions. We include *timestamp, full_sq, life_sq, floor, build_year, num_room, price_doc*. We check if there are duplicates based on them.

There are 16 duplicates based on this set of key features.

What to do?

We can drop these duplicates based on the key features.

We dropped the 16 duplicates within the new dataset named *df_dedupped2*.

Inconsistent data

It is also crucial to have the dataset follow specific standards to fit a model. We need to explore the data in different ways to find out the inconsistent data. Much of the time, it

Below we cover four inconsistent data types:

Inconsistent type #1: Capitalization

Inconsistent usage of upper and lower cases in categorical values is a common mistake. It could cause issues since analyses in Python is case sensitive.

How to find out?

Let's look at the *sub_area* feature.

It stores the name of different areas and looks very standardized.

But sometimes there is inconsistent capitalization usage within the same feature. The "Poselenie Sosenskoe" and "pOseleNie sosenskeo" could refer to the same area.

What to do?

To avoid this, we can put all letters to lower cases (or upper cases).

Inconsistent type #2: Formats

How to find out?

The feature *timestamp* is in string format while it represents dates.

What to do?

We can convert it and extract the date or time values by using the code below. After this, it's easier to analyze the transaction volume group by either year or month.

Related article: How To Manipulate Date And Time In Python Like A Boss

Inconsistent type #3: Categorical Values

Inconsistent categorical values are the last inconsistent type we cover. A categorical feature has a limited number of values. Sometimes there may be other values due to

How to find out?

We need to observe the feature to find out this inconsistency. Let's show this with an example.

We create a new dataset below since we don't have such a problem in the real estate dataset. For instance, the value of *city* was typed by mistakes as "torontoo" and "tronto". But they both refer to the correct value "toronto".

A simple way to identify them is fuzzy logic (or edit distance). It measures how many letters (distance) we need to change the spelling of one value to match with another value.

We know that the categories should only have four values of "toronto", "vancouver", "montreal", and "calgary". We calculate the distance between all the values and the word "toronto" (and "vancouver"). We can see that the ones likely to be typos have a smaller distance with the correct word. Since they only differ by a couple of letters.

What to do?

We can set criteria to convert these typos to the correct values. For example, the below code sets all the values within 2 letters distance from "toronto" to be "toronto".

Inconsistent type #4: Addresses

The address feature could be a headache for many of us. Because people entering the data into the database often *don't* follow a standard format.

How to find out?

We can find messy address data by looking at it. Even though sometimes we can't spot any issues, we can still run code to standardize them.

There is no address column in our dataset for privacy reasons. So we create a new dataset *df_add_ex* with feature *address*.

As we can see, the address feature is quite messy.

What to do?

We run the below code to lowercase the letters, remove white space, delete periods and standardize wordings.

It looks much better now.

We *did* it! What a long journey we have come along.

Clear all the “dirty” data that’s blocking your way to fit the model.

Be the *boss* of *cleaning*!

Source: GIPHY

Thank you for reading.

I hope you found this data cleaning guide helpful. Please leave any comments to let us know your thoughts.

Before you leave, don’t forget to *sign up for the Just into Data newsletter*! Or connect with

How to Learn Data Science Online: ALL You Need to Know - Just into Data

This is a complete roadmap/curriculum of getting into data science with online resources. Whether you want to learn for...

www.justintodata.com

3 Steps to Forecast Time Series: LSTM with TensorFlow Keras - Just into Data

This is a complete roadmap/curriculum of getting into data science with online resources. Whether you want to learn for...

www.justintodata.com

How to Manipulate Date And Time in Python Like a Boss - Just into Data

This is a complete roadmap/curriculum of getting into data science with online resources. Whether you want to learn for...

www.justintodata.com

Sign up for The Variable

By Towards Data Science

Every Thursday, the Variable delivers the very best of Towards Data Science: from hands-on tutorials and cutting-edge research to original features you don't want to miss. Take a look.

Your email

Get this newsletter

By signing up, you will create a Medium account if you don't already have one. Review our Privacy Policy for more information about our privacy practices.

Get started Open in app

Data Science Python Machine Learning Programming Modeling

About Help Legal

Get the Medium app

Download on the App Store GET IT ON Google Play