Data: https://www.kaggle.com/c/sberbank-russian-housing-market/overview/description

Data Cleaning in Python: the Ultimate Guide (2020)

Techniques on what to clean and how.

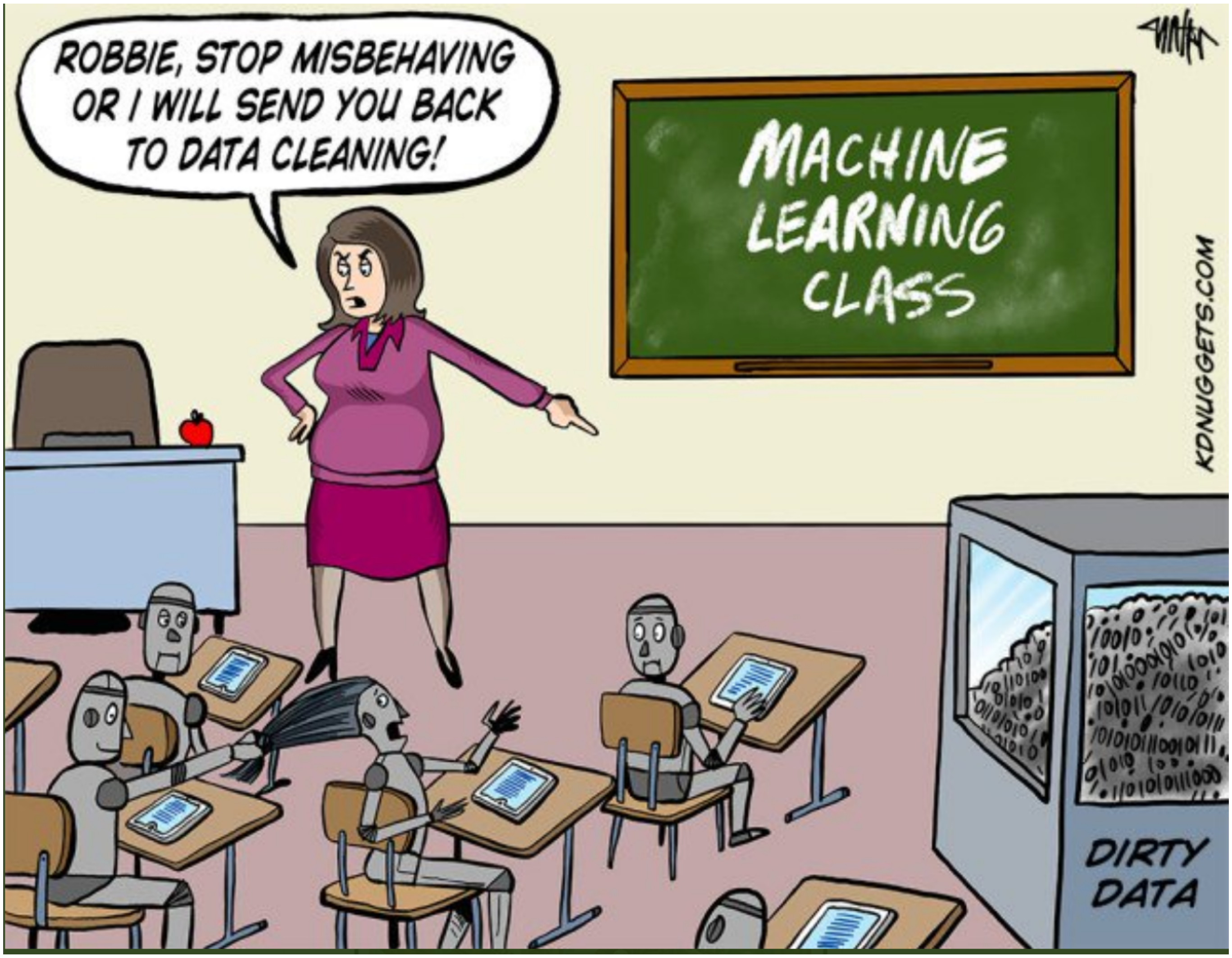
Lianne & Justin @ Just into Data Follow
Feb 3 · 11 min read ★

Source: Pixabay

Before fitting a machine learning or statistical model, we always *have to* clean the data. *No* models create meaningful results with messy data.

> ***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*

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
- Irregular Data (Outliers)
- 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])
numeric_cols = df_numeric.columns.values
print(numeric_cols)

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

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

Dealing with missing data/value is one of the most tricky but common parts of data cleaning. While many models can live with other problems of the data, most models don’t accept missing data.

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

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

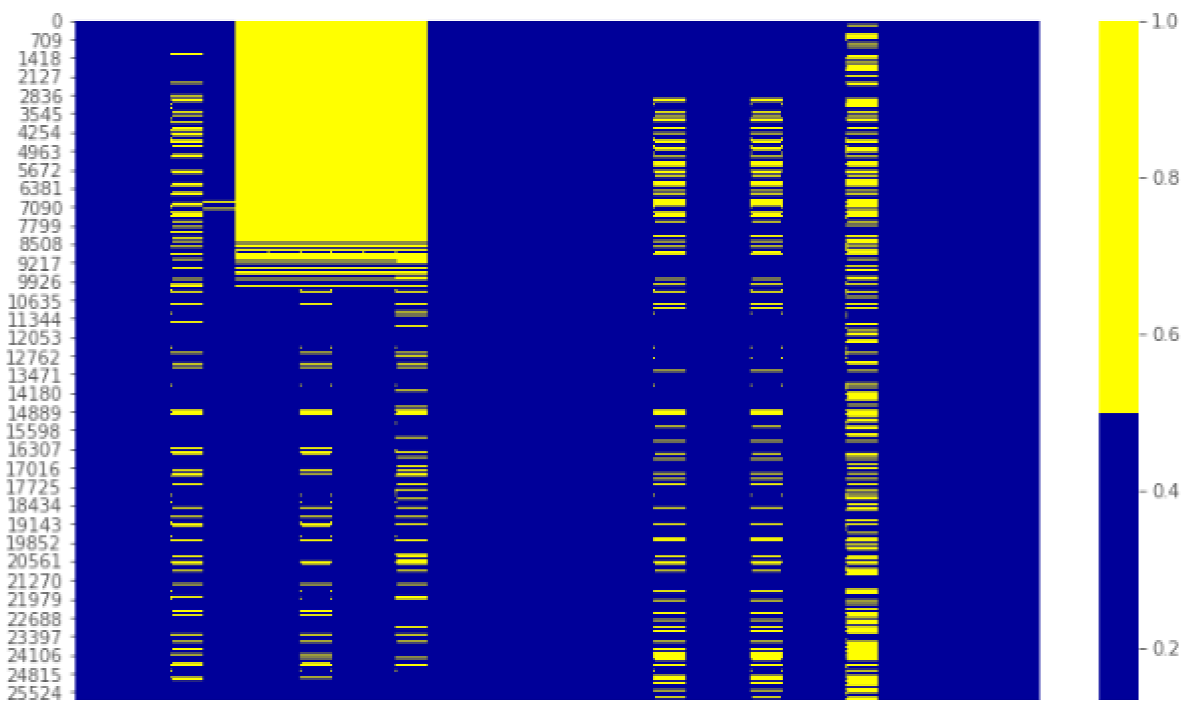
```python
cols = df.columns[:30] # first 30 columns
colours = ['#000099', '#ffff00'] # specify the colours - yellow is missing. blue is not missing.
sns.heatmap(df[cols].isnull(), cmap=sns.color_palette(colours))
```

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.

<matplotlib.axes._subplots.AxesSubplot at 0x16c0160ad30>



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.

```python
# 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%
material - 31.0%
build_year - 45.0%
num_room - 31.0%
kitch_sq - 31.0%
state - 44.0%
product_type - 0.0%
sub_area - 0.0%
```

```
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.

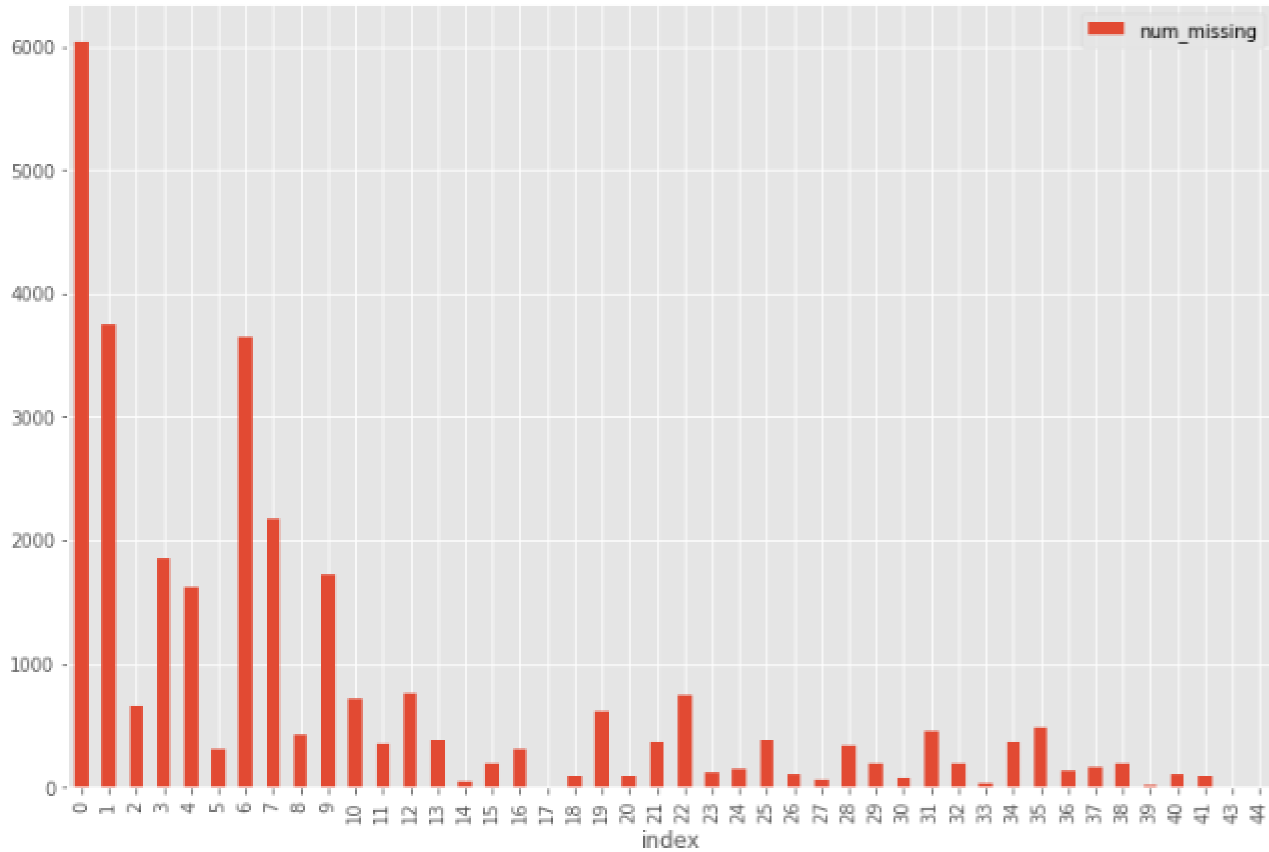
To learn more about the missing value patterns among observations, we can visualize it by a histogram.

```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

This histogram helps to identify the missing values situations among the 30,471 observations.

```
<matplotlib.axes._subplots.AxesSubplot at 0x16ec982e8>
```



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**

In statistics, this method is called the listwise deletion technique. In this solution, we drop the entire observation as long as it contains a missing value.

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.

```python
# 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.

```python
# 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.

When the feature is a categorical variable, we may impute the missing data by the mode (the most frequent value).

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

```
print(med)
df['life_sq'] = df['life_sq'].fillna(med)
```


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)
```



```
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_frame
imputing missing values for: build_count_brick
imputing missing values for: build_count_monolith
imputing missing values for: build_count_panel
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: 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.

```python
# 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)
```

missing_data_imputation3.py hosted with ♡ by GitHub view raw

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

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**

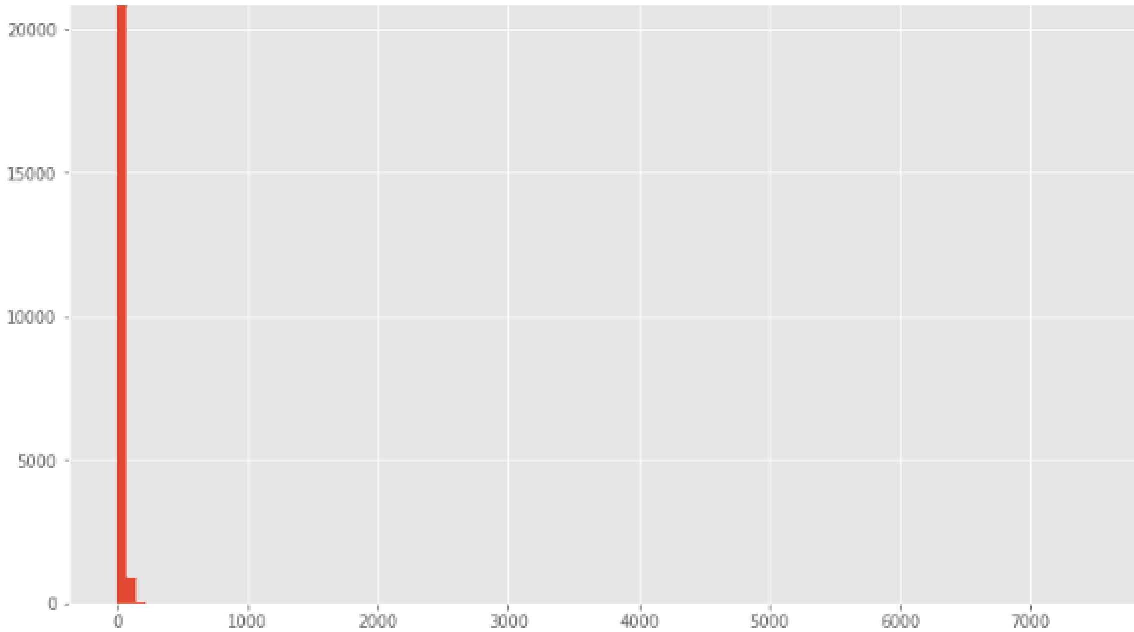
When the feature is numeric, we can use a histogram and box plot to detect outliers.

Below is the histogram of feature *life_sq*.

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

outlier_histogram.py hosted with ♡ by GitHub view raw

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



Histogram

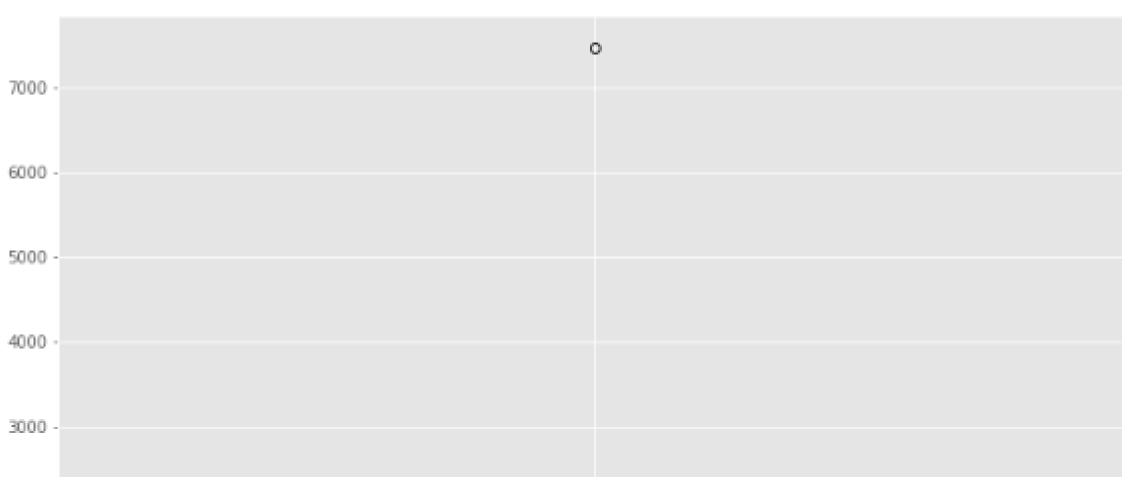
To study the feature closer, let’s make a box plot.

```python
# box plot.
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.

<matplotlib.axes._subplots.AxesSubplot at 0x20da34642b0>



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.

```python
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
```

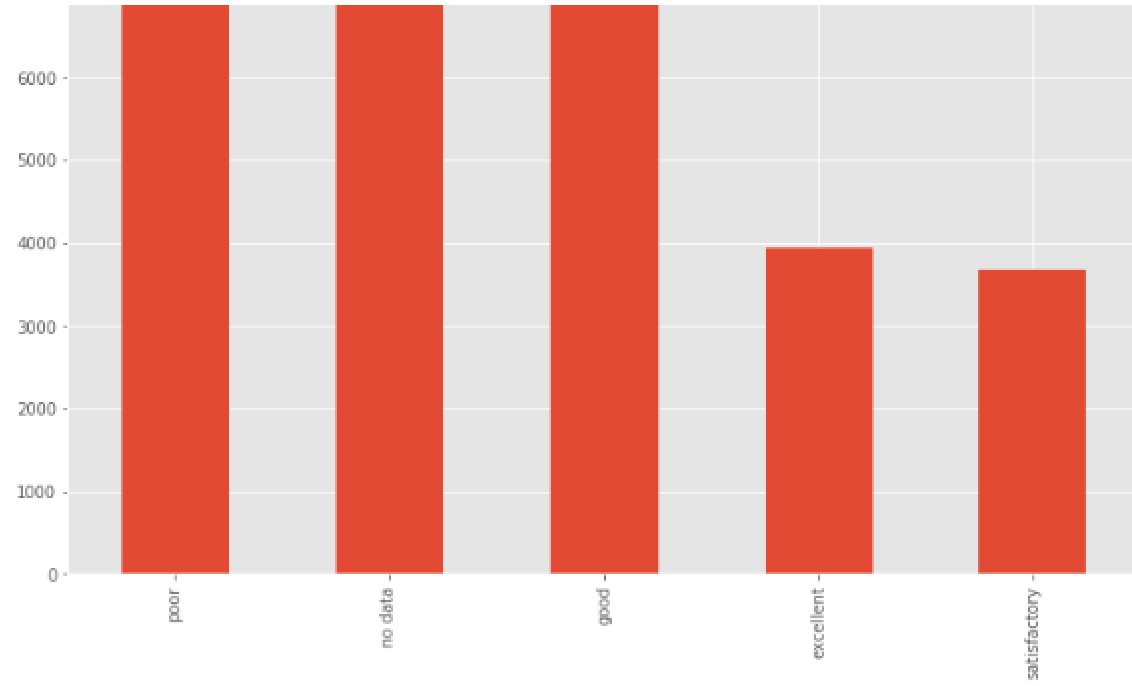
- **Technique #3: Bar Chart**

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.

```python
# bar chart -  distribution of a categorical variable
df['ecology'].value_counts().plot.bar()
```

outlier_barchart.py hosted with ♡ by GitHub view raw



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.

. . .

Unnecessary data

After all the hard work done for missing data and outliers, let's look at unnecessary data, which is more straightforward.

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.

```python
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

railroad_terminal_raion: 96.27187%
no      29335
yes      1136
Name: railroad_terminal_raion, dtype: int64
```

```
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?

We need to skim through the features to identify irrelevant ones.

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.

```python
# we know that column 'id' is unique, but what if we drop it?
df_dedupped = df.drop('id', axis=1).drop_duplicates()

# there were duplicate rows
print(df.shape)
print(df_dedupped.shape)
```

duplicate_data_rows.py hosted with ♡ by GitHub view raw

10 rows are being complete duplicate observations.

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

What to do?

How to find out?

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

For example, the chances of two transactions happening at the same time, with the same square footage, the same price, and the same build year are close to zero.

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.

```python
key = ['timestamp', 'full_sq', 'life_sq', 'floor', 'build_year', 'num_room', 'price_doc']

df.fillna(-999).groupby(key)['id'].count().sort_values(ascending=False).head(20)
```

duplicate_data_key_check.py hosted with ♡ by GitHub　　view raw

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

```
timestamp   full_sq  life_sq  floor  build_year  num_room  price_doc
2014-12-09  40       -999.0   17.0   -999.0       1.0      4607265      2
2014-04-15  134       134.0   1.0     0.0         3.0      5798496      2
2013-08-30  40       -999.0   12.0   -999.0       1.0      4462000      2
2012-09-05  43       -999.0   21.0   -999.0      -999.0    6229540      2
2013-12-05  40       -999.0   5.0    -999.0       1.0      4414080      2
2014-12-17  62       -999.0   9.0    -999.0       2.0      6552000      2
2013-05-22  68       -999.0   2.0    -999.0      -999.0    5406690      2
2012-08-27  59       -999.0   6.0    -999.0      -999.0    4506800      2
2013-04-03  42       -999.0   2.0    -999.0      -999.0    3444000      2
2015-03-14  62       -999.0   2.0    -999.0       2.0      6520500      2
2014-01-22  46        28.0    1.0     1968.0      2.0      3000000      2
2012-10-22  61       -999.0   18.0   -999.0      -999.0    8248500      2
2013-09-23  85       -999.0   14.0   -999.0       3.0      7725974      2
2013-06-24  40       -999.0   12.0   -999.0      -999.0    4112800      2
2015-03-30  41        41.0    11.0    2016.0      1.0      4114580      2
2013-12-18  39       -999.0   6.0    -999.0       1.0      3700946      2
2013-08-29  58        58.0    13.0    2013.0      2.0      5764128      1
            50        33.0    2.0     1972.0      2.0      8150000      1
            52        30.0    9.0     2006.0      2.0      10000000     1
2013-08-30  38        17.0    15.0    2004.0      1.0      6400000      1
Name: id, dtype: int64
```

What to do?

We can drop these duplicates based on the key features.

```
df_dedupped2 = df.drop_duplicates(subset=key)

print(df.shape)
print(df_dedupped2.shape)
```

duplicate_data_drop.py hosted with ♡ by GitHub view raw

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

```
(30471, 292)
(30455, 292)
```

. . .

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 depends on observations and experience. There is no set code to run and fix them all.

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.

```
df['sub_area'].value_counts(dropna=False)
```

string_lower_case1.py hosted with ♡ by GitHub view raw

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

```
                                    ...
Molzhaninovskoe                       3
Poselenie Kievskij                    2
Poselenie Shhapovskoe                 2
Poselenie Mihajlovo-Jarcevskoe        1
Poselenie Klenovskoe                  1
Name: sub_area, Length: 146, dtype: int64
```

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).

```python
# make everything lower case.
df['sub_area_lower'] = df['sub_area'].str.lower()
df['sub_area_lower'].value_counts(dropna=False)
```

string_lower_case2.py hosted with ♡ by GitHub view raw

```
poselenie sosenskoe               1776
nekrasovka                        1611
poselenie vnukovskoe              1372
poselenie moskovskij               925
poselenie voskresenskoe            713
                                  ...
molzhaninovskoe                      3
poselenie shhapovskoe                2
poselenie kievskij                   2
poselenie klenovskoe                 1
poselenie mihajlovo-jarcevskoe       1
Name: sub_area_lower, Length: 146, dtype: int64
```

Inconsistent type #2: Formats

Another standardization we need to perform is the data formats. One example is to convert the feature from string to DateTime format.

How to find out?

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

```python
df
```

	id	timestamp	full_sq	life_sq	floor	max_floor	material	build_year	num_room	kitch_sq	...
0	1	2011-08-20	43	27.0	4.0	NaN	NaN	NaN	NaN	NaN	...
1	2	2011-08-23	34	19.0	3.0	NaN	NaN	NaN	NaN	NaN	...
2	3	2011-08-27	43	29.0	2.0	NaN	NaN	NaN	NaN	NaN	...
3	4	2011-09-01	89	50.0	9.0	NaN	NaN	NaN	NaN	NaN	...
4	5	2011-09-05	77	77.0	4.0	NaN	NaN	NaN	NaN	NaN	...
...	...	...	...	...	...	...	...	...	...	...	...
30466	30469	2015-06-30	44	27.0	7.0	9.0	1.0	1975.0	2.0	6.0	...
30467	30470	2015-06-30	86	59.0	3.0	9.0	2.0	1935.0	4.0	10.0	...
30468	30471	2015-06-30	45	NaN	10.0	20.0	1.0	NaN	1.0	1.0	...
30469	30472	2015-06-30	64	32.0	5.0	15.0	1.0	2003.0	2.0	11.0	...
30470	30473	2015-06-30	43	28.0	1.0	9.0	1.0	1968.0	2.0	6.0	...

	cafe_count_5000_price_high	big_church_count_5000	church_count_5000	mosque_count_5000	leisure_count_5000	sport_count_5000	market_count_5000	price_doc	sub_area_lower	ecology_new
0	0	13	22	1	0	52	4	5850000	bibirevo	good_or_better
1	0	15	29	1	10	66	14	6000000	nagatinskij zaton	good_or_better
2	0	11	27	0	4	67	10	5700000	tekstil'shhiki	poor
3	1	4	4	0	0	26	3	13100000	mitino	good_or_better
4	17	135	236	2	91	195	14	16331452	basmannoe	good_or_better
...	...	...	...	...	...	...	...	...	...	...
30466	0	15	26	1	2	84	6	7400000	otradnoe	good_or_better
30467	24	98	182	1	82	171	15	25000000	tverskoe	poor
30468	0	2	12	0	1	11	1	6970959	poselenie vnukovskoe	no data
30469	1	6	31	1	4	65	7	13500000	obruchevskoe	satisfactory
30470	0	7	16	0	9	54	10	5600000	novogireevo	poor

30471 rows × 294 columns

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.

```python
df['timestamp_dt'] = pd.to_datetime(df['timestamp'], format='%Y-%m-%d')
df['year'] = df['timestamp_dt'].dt.year
df['month'] = df['timestamp_dt'].dt.month
df['weekday'] = df['timestamp_dt'].dt.weekday

print(df['year'].value_counts(dropna=False))
print()
print(df['month'].value_counts(dropna=False))
```

string_to_datetime2.py hosted with ♡ by GitHub view raw

```
2014    13662
2013     7978
2012     4839
2015     3239
2011      753
Name: year, dtype: int64

12    3400
4     3191
3     2972
11    2970
10    2736
6     2570
5     2496
9     2346
2     2275
7     1875
8     1831
1     1809
Name: month, dtype: int64
```

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

×

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 reasons such as typos.

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.

```python
from nltk.metrics import edit_distance

df_city_ex = pd.DataFrame(data={'city': ['torontoo', 'toronto', 'tronto', 'vancouver', 'vancover'


df_city_ex['city_distance_toronto'] = df_city_ex['city'].map(lambda x: edit_distance(x, 'toronto'
df_city_ex['city_distance_vancouver'] = df_city_ex['city'].map(lambda x: edit_distance(x, 'vancou
df_city_ex
```

fuzzy_logic_distance.py hosted with ♡ by GitHub view raw

	city	city_distance_toronto	city_distance_vancouver

5	vancouvr	7	1
6	montreal	7	8
7	calgary	7	8

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”.

```
msk = df_city_ex['city_distance_toronto'] <= 2
df_city_ex.loc[msk, 'city'] = 'toronto'

msk = df_city_ex['city_distance_vancouver'] <= 2
df_city_ex.loc[msk, 'city'] = 'vancouver'

df_city_ex
```

fuzzy_logic_replace.py hosted with ♡ by GitHub view raw

	city	city_distance_toronto	city_distance_vancouver
0	toronto	1	8
1	toronto	0	8
2	toronto	1	8
3	vancouver	8	0
4	vancouver	7	1
5	vancouver	7	1
6	montreal	7	8
7	calgary	7	8

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.

```
# no address column in the housing dataset. So create one to show the code.
df_add_ex = pd.DataFrame(['123 MAIN St Apartment 15', '123 Main Street Apt 12   ', '543 FirSt Av'
df_add_ex
```

address_cleaning1.py hosted with ♡ by GitHub view raw

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

	address
0	123 MAIN St Apartment 15
1	123 Main Street Apt 12
2	543 FirSt Av
3	876 FIRst Ave.

What to do?

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

```
df_add_ex['address_std'] = df_add_ex['address'].str.lower()
df_add_ex['address_std'] = df_add_ex['address_std'].str.strip() # remove leading and trailing whi
df_add_ex['address_std'] = df_add_ex['address_std'].str.replace('\\.', '') # remove period.
df_add_ex['address_std'] = df_add_ex['address_std'].str.replace('\\bstreet\\b', 'st') # replace s
df_add_ex['address_std'] = df_add_ex['address_std'].str.replace('\\bapartment\\b', 'apt') # repla
df_add_ex['address_std'] = df_add_ex['address_std'].str.replace('\\bav\\b', 'ave') # replace apar

df_add_ex
```

address_cleaning2.py hosted with ♡ by GitHub view raw

It looks much better now.

	address	address_std
0	123 MAIN St Apartment 15	123 main st apt 15
1	123 Main Street Apt 12	123 main st apt 12
2	543 FirSt Av	543 first ave

• • •

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.

For more data science articles from Lianne & Justin:

How To Manipulate Date And Time In Python Like A Boss

Commonly used datetime functions with examples

towardsdatascience.com

How to Improve Sports Betting Odds — Step by Step Guide in Python

The data science strategy I used to make $20,000 betting on sports.

towardsdatascience.com

How To Visualize A Decision Tree In 5 Steps

Tailored to corporate Windows environments

towardsdatascience.com

Data Science Python Machine Learning Programming Modeling