Моем датасет: руководство по очистке данных в Python

[Редакция techrocks.ru](https://techrocks.ru/author/test-name/)

 02.04.2020

[0](https://techrocks.ru/2020/04/02/data-cleaning-with-python/#respond)

329

 views

Прежде чем передать набор данных ML-модели, его нужно очистить: обработать пропуски, дубликаты, выбросы. Как это сделать, не потеряв важную информацию, [описал](https://proglib.io/p/moem-dataset-rukovodstvo-po-ochistke-dannyh-v-python-2020-03-27) сайт proglib.io.



Ни одна модель машинного обучения не выдаст осмысленных результатов, если вы предоставите ей сырые данные. После формирования выборки данных их необходимо [очистить](https://techrocks.ru/2020/02/03/linux-command-line-for-data-science/).

[Очистка данных](https://en.wikipedia.org/wiki/Data_cleansing) – это процесс обнаружения и исправления (или удаления) поврежденных или неточных записей из набора записей, таблицы или базы данных. Процесс включает в себя выявление неполных, неправильных, неточных или несущественных данных, а затем замену, изменение или удаление «загрязненных» данных.

Определение очень длинное и не очень понятное 🙁

Чтобы детально во всем разобраться, мы разбили это определение на составные части и создали пошаговый гайд по очистке данных на Python. Здесь мы разберем методы поиска и исправления:

* отсутствующих данных;
* нетипичных данных – выбросов;
* неинформативных данных – дубликатов;
* несогласованных данных – одних и тех же данных, представленных в разных регистрах или форматах.

Для работы с данными мы использовали [Jupyter Notebook](https://proglib.io/p/jupyter) и библиотеку [Pandas](https://proglib.io/p/chto-novogo-v-pandas-1-0-2020-01-23).

Базой для наших экспериментов послужит [набор данных по ценам на жилье в России](https://www.kaggle.com/c/sberbank-russian-housing-market/overview/description), найденный на Kaggle. Мы не станем очищать всю базу целиком, но разберем на ее основе главные методы и операции.

Прежде чем переходить к процессу очистки, всегда нужно представлять исходный датасет. Давайте быстро взглянем на сами данные:

# импорт пакетов

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

# чтение данных

df = pd.read\_csv('sberbank.csv')

# shape and data types of the data

print(df.shape)

print(df.dtypes)

# отбор числовых колонок

df\_numeric = df.select\_dtypes(include=[np.number])

numeric\_cols = df\_numeric.columns.values

print(numeric\_cols)

# отбор нечисловых колонок

df\_non\_numeric = df.select\_dtypes(exclude=[np.number])

non\_numeric\_cols = df\_non\_numeric.columns.values

print(non\_numeric\_cols)

Этот код покажет нам, что набор данных состоит из 30471 строки и 292 столбцов. Мы увидим, являются ли эти столбцы числовыми или категориальными признаками.

Теперь мы можем пробежаться по чек-листу «грязных» типов данных и очистить их один за другим.

**1. Отсутствующие данные**

Работа с [отсутствующими](https://en.wikipedia.org/wiki/Missing_data) значениями – одна из самых сложных, но и самых распространенных проблем очистки. Большинство моделей не предполагают пропусков.

**1.1. Как обнаружить?**

Рассмотрим три метода обнаружения отсутствующих данных в наборе.

**1.1.1. Тепловая карта пропущенных значений**

Когда признаков в наборе не очень много, визуализируйте пропущенные значения с помощью тепловой карты.

cols = df.columns[:30] # первые 30 колонок

# определяем цвета

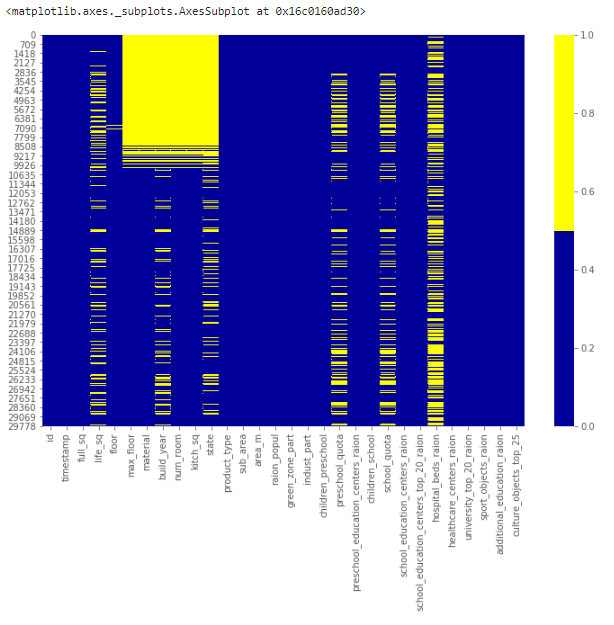
# желтый - пропущенные данные, синий - не пропущенные

colours = ['#000099', '#ffff00']

sns.heatmap(df[cols].isnull(), cmap=sns.color\_palette(colours))

Приведенная ниже карта демонстрирует паттерн пропущенных значений для первых 30 признаков набора. По горизонтальной оси расположены признаки, по вертикальной – количество записей/строк. Желтый цвет соответствует пропускам данных.

Заметно, например, что признак life\_sq имеет довольно много пустых строк, а признак floor – напротив, всего парочку – около 7000 строки.



**1.1.2. Процентный список пропущенных данных**

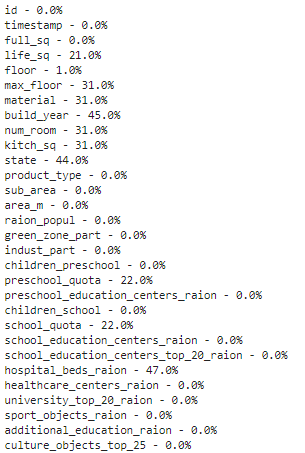
Если в наборе много признаков и визуализация занимает много времени, можно составить список долей отсутствующих записей для каждого признака.

for col in df.columns:

pct\_missing = np.mean(df[col].isnull())

print('{} - {}%'.format(col, round(pct\_missing\*100)))

Такой список для тех же 30 первых признаков выглядит следующим образом:

Список недостающих данных % – первые 30 функций

У признака life\_sq отсутствует 21% значений, а у floor – только 1%.

Этот список является полезным резюме, которое может отлично дополнить визуализацию тепловой карты.

**1.1.3. Гистограмма пропущенных данных**

Еще одна хорошая техника визуализации для наборов с большим количеством признаков – построение гистограммы для числа отсутствующих значений в записи.

# сначала создаем индикатор для признаков с пропущенными данными

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

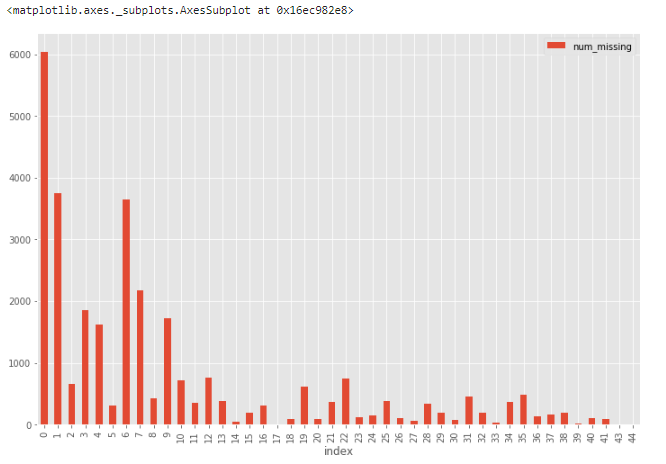
# затем на основе индикатора строим гистограмму

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='i

Отсюда понятно, что из 30 тыс. записей более 6 тыс. строк не имеют ни одного пропущенного значения, а еще около 4 тыс.– всего одно. Такие строки можно использовать в качестве «эталонных» для проверки различных гипотез по дополнению данных.

Гистограмма пропущенных значений

**1.2. Что делать с пропущенными значениями?**

Не существует общих решений для проблемы отсутствующих данных. Для каждого конкретного набора приходится искать наиболее подходящие методы или их комбинации.

Разберем четыре самых распространенных техники. Они помогут в простых ситуациях, но, скорее всего, придется проявить творческий подход и поискать нетривиальные решения, например, промоделировать пропуски.

**1.2.1. Отбрасывание записей**

Первая техника в статистике называется методом удаления по списку и заключается в простом отбрасывании записи, содержащей пропущенные значения. Это решение подходит только в том случае, если недостающие данные не являются информативными.

Для отбрасывания можно использовать и другие критерии. Например, из гистограммы, построенной в предыдущем разделе, мы узнали, что лишь небольшое количество строк содержат более 35 пропусков. Мы можем создать новый набор данных df\_less\_missing\_rows, в котором отбросим эти строки.

# отбрасываем строки с большим количеством пропусков

ind\_missing = df[df['num\_missing'] > 35].index

df\_less\_missing\_rows = df.drop(ind\_missing, axis=0)

**1.2.2. Отбрасывание признаков**

Как и предыдущая техника, отбрасывание признаков может применяться только для неинформативных признаков.

В процентном списке, построенном ранее, мы увидели, что признак hospital\_beds\_raion имеет высокий процент недостающих значений – 47%. Мы можем полностью отказаться от этого признака:

cols\_to\_drop = ['hospital\_beds\_raion']

df\_less\_hos\_beds\_raion = df.drop(cols\_to\_drop, axis=1)

**1.2.3. Внесение недостающих значений**

Для численных признаков можно воспользоваться методом принудительного заполнения пропусков. Например, на место пропуска можно записать среднее или медианное значение, полученное из остальных записей.

Для категориальных признаков можно использовать в качестве заполнителя наиболее часто встречающееся значение.

Возьмем для примера признак life\_sq и заменим все недостающие значения медианой этого признака:

med = df['life\_sq'].median()

print(med)

df['life\_sq'] = df['life\_sq'].fillna(med)

Одну и ту же стратегию принудительного заполнения можно применить сразу для всех числовых признаков:

# impute the missing values and create the missing value indicator variables for each numeric column.

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)

К счастью, в нашем наборе не нашлось пропусков в категориальных признаках. Но это не мешает нам продемонстрировать использование той же стратегии:

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)

**1.2.4. Замена недостающих значений**

Можно использовать некоторый дефолтный плейсхолдер для пропусков, например, новую категорию \_MISSING\_ для категориальных признаков или число -999 для числовых.

Таким образом, мы сохраняем данные о пропущенных значениях, что тоже может быть ценной информацией.

# категориальные признаки

df['sub\_area'] = df['sub\_area'].fillna('\_MISSING\_')

# численные признаки

df['life\_sq'] = df['life\_sq'].fillna(-999)



**2. Нетипичные данные (выбросы)**

[Выбросы](https://en.wikipedia.org/wiki/Outlier) – это данные, которые существенно отличаются от других наблюдений. Они могут соответствовать реальным отклонениям, но могут быть и просто ошибками.

**2.1. Как обнаружить выбросы?**

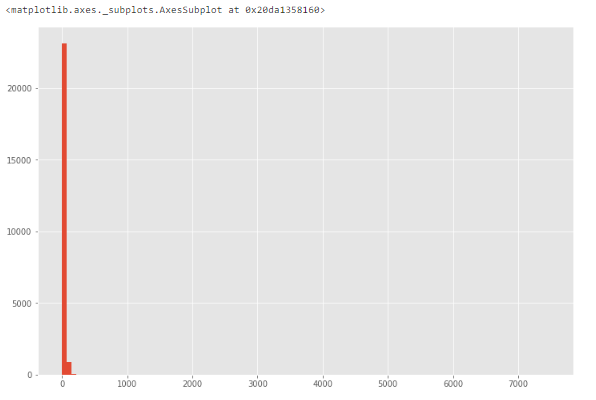
Для численных и категориальных признаков используются разные методы изучения распределения, позволяющие обнаружить выбросы.

**2.1.1. Гистограмма/коробчатая диаграмма**

Если признак численный, можно построить [гистограмму](https://en.wikipedia.org/wiki/Histogram) или [коробчатую диаграмму](https://www.khanacademy.org/math/statistics-probability/summarizing-quantitative-data/box-whisker-plots/a/box-plot-review) ([ящик с усами](https://ru.wikipedia.org/wiki/%D0%AF%D1%89%D0%B8%D0%BA_%D1%81_%D1%83%D1%81%D0%B0%D0%BC%D0%B8)). Посмотрим на примере уже знакомого нам признака life\_sq.

df['life\_sq'].hist(bins=100)

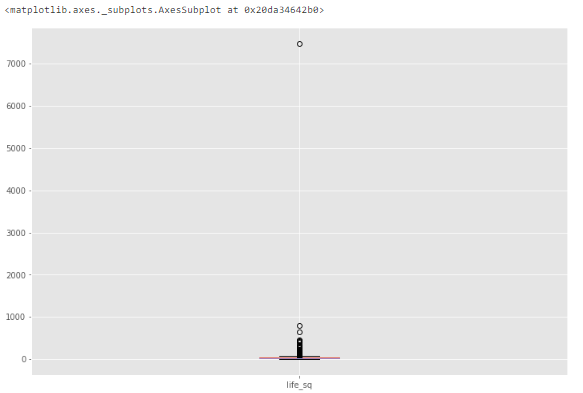
Из-за возможных выбросов данные выглядят сильно искаженными.

Построение гистограммы для обнаружения выбросов

Чтобы изучить особенность поближе, построим коробчатую диаграмму.

df.boxplot(column=['life\_sq'])

Видим, что есть выброс со значением более 7000.

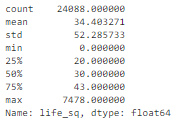
Построение коробчатой диаграммы для обнаружения выбросов

**2.1.2. Описательная статистика**

Отклонения численных признаков могут быть слишком четкими, чтобы не визуализироваться коробчатой диаграммой. Вместо этого можно проанализировать их описательную статистику.

Например, для признака life\_sq видно, что максимальное значение равно 7478, в то время как 75% квартиль равен только 43. Значение 7478 – выброс.

df['life\_sq'].describe()

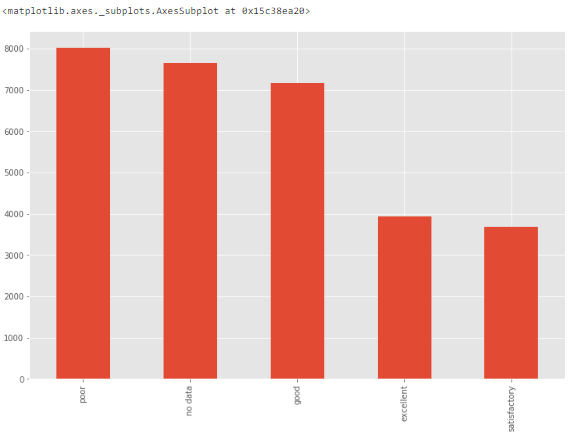


**2.1.3. Столбчатая диаграмма**

Для категориальных признаков можно построить столбчатую диаграмму – для визуализации данных о категориях и их распределении.

Например, распределение признака ecology вполне равномерно и допустимо. Но если существует категория только с одним значением "другое", то это будет выброс.

df['ecology'].value\_counts().plot.bar()

Построение столбчатой диаграммы для обнаружения выбросов

**2.1.4. Другие методы**

Для обнаружения выбросов можно использовать другие методы, например, построение точечной диаграммы, [z-оценку](https://ru.wikipedia.org/wiki/Z-%D0%BE%D1%86%D0%B5%D0%BD%D0%BA%D0%B0) или [кластеризацию](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%82%D0%B5%D1%80%D0%BD%D1%8B%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7). В этом руководстве они не рассматриваются.

**2.2. Что делать?**

Выбросы довольно просто обнаружить, но выбор способа их устранения слишком существенно зависит от специфики набора данных и целей проекта. Их обработка во многом похожа на обработку пропущенных данных, которую мы разбирали в предыдущем разделе. Можно удалить записи или признаки с выбросами, либо скорректировать их, либо оставить без изменений.

Переходим к более простой части очистки данных – удалению мусора.

Вся информация, поступающая в модель, должна служить целям проекта. Если она не добавляет никакой ценности, от нее следует избавиться.

Три основных типа «ненужных» данных:

* неинформативные признаки с большим количеством одинаковых значений,
* нерелевантные признаки,
* дубликаты записей.

Рассмотрим работу с каждым типом отдельно.

**3. Неинформативные признаки**

Если признак имеет слишком много строк с одинаковыми значениями, он не несет полезной информации для проекта.

**3.1. Как обнаружить?**

Составим список признаков, у которых более 95% строк содержат одно и то же значение.

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:

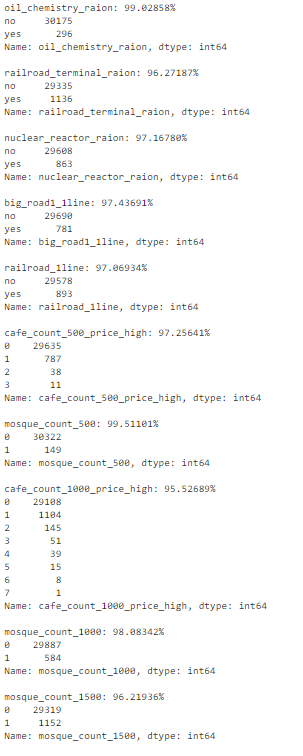
low\_information\_cols.append(col)

print('{0}: {1:.5f}%'.format(col, top\_pct\*100))

print(cnts)

print()

Теперь можно последовательно перебрать их и определить, несут ли они полезную информацию.

Список признаков с высоким процентом одинаковых значений

**3.2. Что делать?**

Если после анализа причин получения повторяющихся значений вы пришли к выводу, что признак не несет полезной информации, используйте drop().

**4. Нерелевантные признаки**

Нерелевантные признаки обнаруживаются ручным отбором и оценкой значимости. Например, признак, регистрирующий температуру воздуха в Торонто точно не имеет никакого отношения к прогнозированию цен на российское жилье. Если признак не имеет значения для проекта, его нужно исключить.

**5. Дубликаты записей**

Если значения признаков (всех или большинства) в двух разных записях совпадают, эти записи называются дубликатами.

**5.1. Как обнаружить повторяющиеся записи?**

Способ обнаружения дубликатов зависит от того, что именно мы считаем дубликатами. Например, в наборе данных есть уникальный идентификатор id. Если две записи имеют одинаковый id, мы считаем, что это одна и та же запись. Удалим все неуникальные записи:

# отбрасываем неуникальные строки

df\_dedupped = df.drop('id', axis=1).drop\_duplicates()

# сравниваем формы старого и нового наборов

print(df.shape)

print(df\_dedupped.shape)

Получаем в результате 10 отброшенных дубликатов:

Обнаружение неуникальных записей по идентификатору

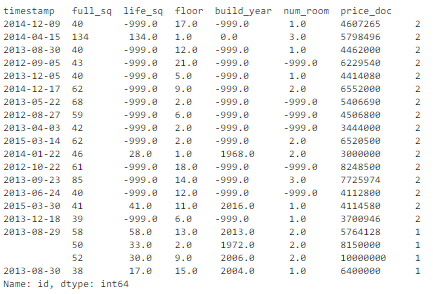
Другой распространенный способ вычисления дубликатов: по набору ключевых признаков. Например, неуникальными можно считать записи с одной и той же площадью жилья, ценой и годом постройки.

Найдем в нашем наборе дубликаты по группе критических признаков – full\_sq,life\_sq, floor, build\_year, num\_room, price\_doc:

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)

Получаем в результате 16 дублирующихся записей:

Обнаружение дубликатов по набору ключевых признаков

**5.2. Что делать с дубликатами?**

Очевидно, что повторяющиеся записи нам не нужны, значит, их нужно исключить из набора.

Вот так выглядит удаление дубликатов, основанное на наборе ключевых признаков:

key = ['timestamp', 'full\_sq', 'life\_sq', 'floor', 'build\_year', 'num\_room', 'price\_doc']

df\_dedupped2 = df.drop\_duplicates(subset=key)

print(df.shape)

print(df\_dedupped2.shape)

В результате новый набор df\_dedupped2 стал короче на 16 записей.

Результат отбрасывания дубликатов записей

Большая проблема очистки данных – разные форматы записей. Для корректной работы модели важно, чтобы набор данных соответствовал определенным стандартам – необходимо тщательное исследование с учетом специфики самих данных. Мы рассмотрим четыре самых распространенных несогласованности:

* Разные регистры символов.
* Разные форматы данных (например, даты).
* Опечатки в значениях категориальных признаков.
* Адреса.

**6. Разные регистры символов**

Непоследовательное использование разных регистров в категориальных значениях является очень распространенной ошибкой, которая может существенно повлиять на анализ данных.

**6.1. Как обнаружить?**

Давайте посмотрим на признак sub\_area:

df['sub\_area'].value\_counts(dropna=False)

В нем содержатся названия населенных пунктов. Все выглядит вполне стандартизированным:

Записи с разным регистром символов

Но если в какой-то записи вместо Poselenie Sosenskoe окажется poselenie sosenskoe, они будут расценены как два разных значения.

**6.2. Что делать?**

Эта проблема легко решается принудительным изменением регистра:

# пусть все будет в нижнем регистре

df['sub\_area\_lower'] = df['sub\_area'].str.lower()

df['sub\_area\_lower'].value\_counts(dropna=False)

Приведение всех символов к нижнему регистру



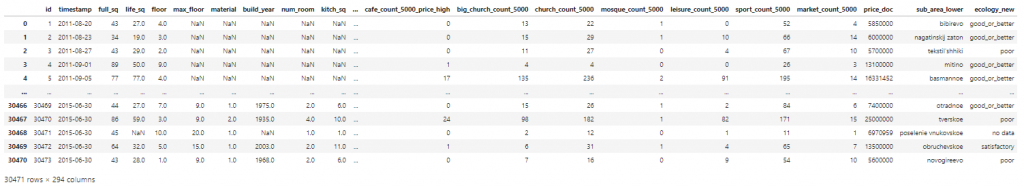
**7. Разные форматы данных**

Ряд данных в наборе находится не в том формате, с которым нам было бы удобно работать. Например, даты, записанные в виде строки, следует преобразовать в формат DateTime.

**7.1. Как обнаружить?**

Признак timestamp представляет собой строку, хотя является датой:

df



**7.2. Что же делать?**

Чтобы было проще анализировать транзакции по годам и месяцам, значения признака timestamp следует преобразовать в удобный формат:

df['timestamp\_dt'] = pd.to\_datetime(df['timestamp'], format='%Y-%m-%d')

df['year'] = df['timestamp\_dt'].dt.year

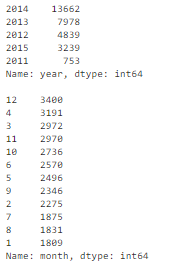
df['month'] = df['timestamp\_dt'].dt.month

df['weekday'] = df['timestamp\_dt'].dt.weekday

print(df['year'].value\_counts(dropna=False))

print()

print(df['month'].value\_counts(dropna=False))

Преобразование строк в даты

Взгляните также на публикацию [How To Manipulate Date And Time In Python Like A Boss](https://towardsdatascience.com/how-to-manipulate-date-and-time-in-python-like-a-boss-ddea677c6a4d).

**8. Опечатки**

Опечатки в значениях категориальных признаков приводят к таким же проблемам, как и разные регистры символов.

**8.1. Как обнаружить?**

Для обнаружения опечаток требуется особый подход. В нашем наборе данных о недвижимости опечаток нет, поэтому для примера создадим новый набор. В нем будет признак city, а его значениями будут torontoo и tronto. В обоих случаях это опечатки, а правильное значение – toronto.

Простой способ идентификации подобных элементов – [нечеткая логика](https://ru.wikipedia.org/wiki/%D0%9D%D0%B5%D1%87%D1%91%D1%82%D0%BA%D0%B0%D1%8F_%D0%BB%D0%BE%D0%B3%D0%B8%D0%BA%D0%B0) или [редактирование расстояния](https://en.wikipedia.org/wiki/Edit_distance). Суть этого метода заключается в измерении количества букв (расстояния), которые нам нужно изменить, чтобы из одного слова получить другое.

Предположим, нам известно, что в признаке city должно находиться одно из четырех значений: toronto, vancouver, montreal или calgary. Мы вычисляем расстояние между всеми значениями и словом toronto (и vancouver).

Те слова, в которых содержатся опечатки, имеют меньшее расстояние с правильным словом, так как отличаются всего на пару букв.

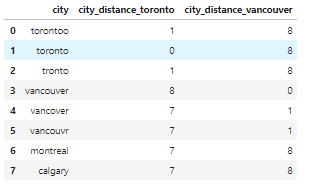
from nltk.metrics import edit\_distance

df\_city\_ex = pd.DataFrame(data={'city': ['torontoo', 'toronto', 'tronto', 'vancouver', 'vancover', 'vancouvr', 'montreal', 'calgary']})

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, 'vancouver'))

df\_city\_ex

Вычисление расстояния между словами для обнаружения опечаток

**8.2. Что делать?**

Мы можем установить критерии для преобразования этих опечаток в правильные значения.

Например, если расстояние некоторого значения от слова toronto не превышает 2 буквы, мы преобразуем это значение в правильное – toronto.

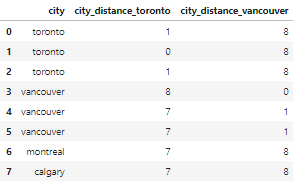
msk = df\_city\_ex['city\_distance\_toronto'] <= 2

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

Исправление найденных опечаток

**9. Адреса**

Адреса – ужасная головная боль для всех аналитиков данных. Ведь мало кто следует стандартному формату, вводя свой адрес в базу данных.

**9.1. Как обнаружить?**

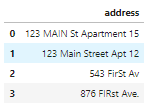
Проще предположить, что проблема разных форматов адреса точно существует. Даже если визуально вы не обнаружили беспорядка в этом признаке, все равно стоит стандартизировать их для надежности.

В нашем наборе данных по соображениям конфиденциальности отсутствует признак адреса, поэтому создадим новый набор df\_add\_ex:

df\_add\_ex = pd.DataFrame(['123 MAIN St Apartment 15', '123 Main Street Apt 12 ', '543 FirSt Av', ' 876 FIRst Ave.'], columns=['address'])

df\_add\_ex

Признак адреса здесь загрязнен:

Значения адреса

**9.2. Что делать?**

Минимальное форматирование включает следующие операции:

* приведение всех символов к нижнему регистру;
* удаление пробелов в начале и конце строки;
* удаление точек;
* стандартизация формулировок: замена street на st, apartment на apt и т. д.

df\_add\_ex['address\_std'] = df\_add\_ex['address'].str.lower()

df\_add\_ex['address\_std'] = df\_add\_ex['address\_std'].str.strip()

df\_add\_ex['address\_std'] = df\_add\_ex['address\_std'].str.replace('\.', '')

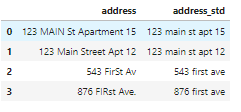
df\_add\_ex['address\_std'] = df\_add\_ex['address\_std'].str.replace('\bstreet\b', 'st')

df\_add\_ex['address\_std'] = df\_add\_ex['address\_std'].str.replace('\bapartment\b', 'apt')

df\_add\_ex['address\_std'] = df\_add\_ex['address\_std'].str.replace('\bav\b', 'ave')

df\_add\_ex

Теперь признак стал намного чище:

Стандартизация адресов

Мы сделали это! Это был долгий и трудный путь, но теперь все «грязные» данные очищены и готовы к анализу, а вы стали спецом по чистке данных!