**Подготовка данных для алгоритмов машинного обучения**

**Описание стека и некоторые вводные**

В нашей статье мы будем использовать язык программирования **python** с сопутствующими ему библиотеки (*sklearn, matplotlib, seaborn*) и в качестве среды для работы **jupyter notebook**. Цель текущего поста - показать общие подходы к подготовке данных. То есть, те манипуляции, которые необходимо совершить перед загрузкой данных в модель машинного обучения. В идеальном мире у вас будет абсолютно чистый датасет без выбросов или пропущенных значений. Однако в реальном мире такие наборы данных встречаются крайне редко.  
Далее будем рассматривать данные из Kaggle: "*Mental Health in Tech Survey*".

Отношение к психическому здоровью и частота психических расстройств

**Первый взгляд на датасет и понимание его специфики**

Трудно работать с данными, не понимая, что они из себя представляют, поэтому давайте их загрузим и выведем некоторые статистики.

import pandas as pd

import numpy as np

df = pd.read\_csv("survey.csv")

df.head()

Это даст нам первое представление о том, что есть наши данные. Далее посмотрим на размеры наших табличных данных. Выполнив построчно код ниже

df.shape # мы увидим информацию о размерности нашего датафрейма

df.info() # покажет информацию о размерности данных

# описание индекса, количество not-a-number элементов

df.describe() # показывает статистики count,mean, std, min, 25%-50%-75% percentile, max

df.nunique() # количество уникальных значений для каждого столбца

Также было бы неплохо увидеть информацию о количестве каждого уникального значения для каждого столбца в наборе данных:

feature\_names = df.columns.tolist()

for column in feature\_names:

print column

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

**value counts**

Это команда для проверки распределения значений. Например, если вы хотите проверить возможные значения и частоту для каждого отдельного значения в столбце «c», вы можете применить

df[‘c’].value\_counts()

Есть несколько полезных приемов / функций:

A. **normalize = True** : если вы хотите проверить частоту вместо подсчетов.

B. **dropna = False** : если вы хотите включить пропущенные значения в статистику.

C. **sort = False** : показать статистику, отсортированную по значениям, а не по количеству.

D. **df[‘c].value\_counts().reset\_index()**.: если вы хотите преобразовать таблицу статистики в датафрейм Pandas и управлять ими.

Большинство столбцов выглядят хорошо, но есть несколько нуждающихся в очистке. Примеры некорректных значений данных ниже.

* Столбец «a*ge*» содержит людей, которые еще не родились (отрицательные числа).
* Столбец «a*ge*» содержит детей (например, 5-летнего возраста), которые вряд ли будут проводить опрос о своем рабочем месте.
* Столбец «a*ge*» содержит возраст в 99999999999 лет
* Существует 49 различных значений для *«gender*». Для примера, «Male» и «male» обозначают одно и то же, но в рассматриваются как две разные категории.
* self\_employed и work\_interfere содержат несколько пропущенных полей.

**Разделение на обучающую выборку и целевую переменную**

Так как мы сейчас рассматриваем задачу обучения с учителем (несколько сублимированную - сами придумали, сами решаем), нам необходимо разделить на признаки для обучения и на признаки для предсказания. Целевая переменная для текущего датасета зависит от ваших целей. Для примера: вы можете, базируясь на этом наборе данных решать классификационную задачу (определять пол опрашиваемого) или же регрессионную (предсказывать возраст опрашиваемого).  Для дальнейшего рассмотрения была взята классификационная задача: будет ли опрашиваемая персона искать лечение.

features = df.drop('treatment', 1)

labels = df['treatment']

Панды предоставляют аналитикам данных способ удалять и фильтровать фрейм данных с помощью **.drop()** . С помощью этого метода строки или столбцы могут быть удалены с помощью метки индекса или имени столбца.

**Syntax:**  
DataFrame.drop(labels=None, axis=0, index=None, columns=None, level=None, inplace=False, errors=’raise’)

**Parameters:**

**labels:**String or list of strings referring row or column name.  
**axis:**int or string value, 0 ‘index’ for Rows and 1 ‘columns’ for Columns.  
**index or columns:**Single label or list. index or columns are an alternative to axis and cannot be used together.  
**level:**Used to specify level in case data frame is having multiple level index.  
**inplace:**Makes changes in original Data Frame if True.  
**errors:**Ignores error if any value from the list doesn’t exists and drops rest of the values when errors = ‘ignore’

**Return type:** Dataframe with dropped values

df.drop (<name>, 0 ) –удаляет строку,

df.drop (<name>, 1 ) –удаляет столбец,

Теперь в features будет таблица без столбца «treatment», а в labels только столбец «treatment».

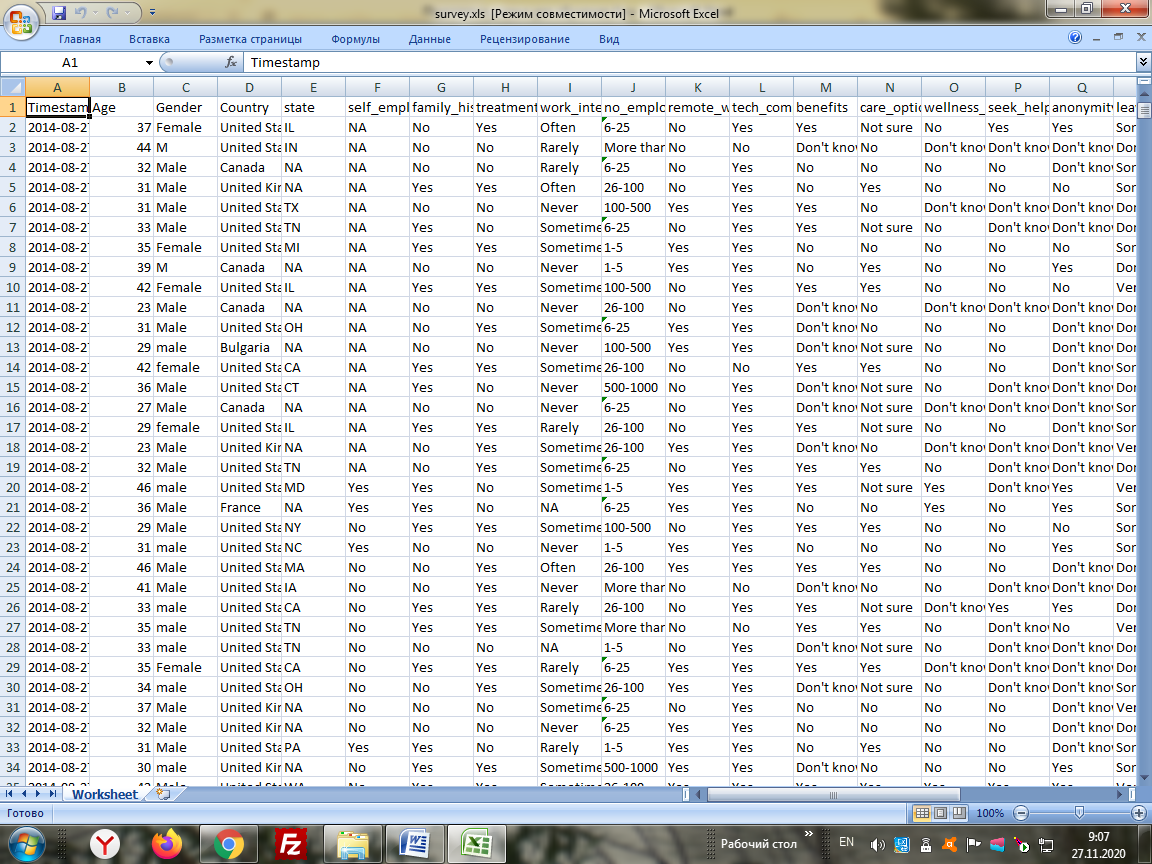
**Обработка пропусков в данных**

Зачастую не существует шаблонных подходов к этой задаче, поскольку подходы во многом зависит от контекста и характера данных. Например, являются ли данные случайными пропусками или же есть скрытая связь между пропусками и некоторым другим записями в обучающем примере?

Один из способов простых способов решения этой проблемы - просто игнорировать или удалять строки, в которых отсутствуют данные, выбрасывая их из нашего анализа. Однако этот метод может быть плох из-за потери информации.

Еще один способ — это заполнение пропусков, где мы заменяем отсутствующее значение каким-либо образом. Базовые реализации просто заменят все отсутствующие значения средним, медианным, либо же константой.

Для начала выясним, что делать с пропущенными значениями, найденными в *self\_employed* и *work\_interfere (название полей в нашем примере, в которых случается NA )*. В обоих случаях столбец содержит категориальные данные.



Возьмем следующий пример, который представляет собой небольшой набор данных, содержащий три признака (погодное условие, температуру и влажность), чтобы предсказать, могу ли я играть в теннис или нет.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| id | weather | temperature | humidity | play tennis? |
| 1 | cloudy | 60 | NaN | yes |
| 2 | rainy | 75 | 80% | NaN |
| 3 | cloudy | NaN | 50% | no |
| 4 | sunny | 65 | 40% | yes |

Если бы мы удалили все строки с отсутствующими значениями, осталась бы только одна строка, и наш предиктор всегда бы предполагал, что я должен играть в теннис, так как других вариантов, по которым он будет учиться, просто не будет. Предположим, мы вместо этого решили заменить нулевое значение температуры в строке 3 средним. В этом случае температура строки 3 искусственно сообщалась бы равной 65.  И это уже позволит при каких-то входных параметрах получать от алгоритма отрицательный результат.  
  
**Scikit-learn** предоставляет реализацию для обработки пропусков

from sklearn.preprocessing import Imputer

imputer = Imputer(missing\_values = 'NaN', strategy = 'mean', axis = 0)

imputer.fit(features)

features = imputer.transform(features)

В приведённом коде мы создадим[Scikit-Learn](http://scikit-learn.org/stable/)-объект Imputer с медианной (mean) стратегией. Затем обучим его на обучающих данных (с помощью imputer.fit), и применим для заполнения отсутствующих значений в обучающем и тестовом наборах (с помощью imputer.transform). То есть записи, которых не хватает в *тестовых данных*, будут заполняться соответствующим медианным значением из *обучающих данных*. **axis = 0 – сортировка по строкам, =1 – по столбцам.**

**Поиск неявных дубликатов**

Как упоминалось ранее, для «*gender*»(поле в таблице) есть 49 различных значений, и было подозрение, что некоторые из этих значений не следует рассматривать как разные категории. В конечном итоге для простоты мы разделим данные на 3 категории: мужчина, женщина и другие (сюда вошли те категории, которые можно однозначно исключить из предыдущих двух, для примера - трансгендер).

Если бы требовалось создать механизм предварительной обработки, который мог бы очищать входящие данные, требовалось бы воспользоваться более умным подходом. Но так как наша задача — это работа с уже имеющемся датасетом, то мы просто используем этот подход с заменой определенных типов.

male\_terms = ["male", "m", "mal", "msle", "malr", "mail", "make", "cis male", "man", "maile", "male (cis)", "cis man"]

female\_terms = ["female", "f", "woman", "femake", "femaile", "femake", "cis female", "cis-female/femme", "female (cis)", "femail", "cis woman"]

def clean\_gender(response):

if response.lower().rstrip() in male\_terms:

return "Male"

elif response.lower().rstrip() in female\_terms:

return "Female"

else:

return "Other"

df['Gender'] = df["Gender"].apply(lambda x: clean\_gender(x))

**Обнаружение выбросов**

Как уже упоминалось ранее, оказалось, что для *Age* существуют значения, которые кажутся ошибочными. Такие как отрицательный возраст или чрезвычайно большие целые числа, могут негативно повлиять на результат работы алгоритма машинного обучения, и нам нужно будет их устранить.

Для этого возьмем нашу эвристическую оценку, в каком возрасте могут работать люди: от 14 до 100 лет. И все величины, не попадающие в этот диапазон, преобразуем в формат *Not-a-Number*.

df.Age.loc [(df.Age <14) | (df.Age> 100)] = np.nan

Эти нулевые значения затем могут быть обработаны с использованием описанного выше **sklearn Imputer**.

После определения диапазона для работающего человека, визуализируем распределение возраста, присутствующего в этом наборе данных.

%matplotlib inline

import seaborn as sns

sns.set(color\_codes=True)

plot = sns.distplot(df.Age.dropna())

plot.figure.set\_size\_inches(6,6)

**Кодирование данных**

Многие алгоритмы машинного обучения ожидают числовые входные данные, поэтому нам нужно выяснить способ представления наших категориальных данных численным образом.  
  
Одним из решений этого было бы произвольное присвоение числового значения для каждой категории и отображение набора данных из исходных категорий в каждое соответствующее число. Например, давайте посмотрим на столбец «*leave*» (как легко вам взять отпуск по болезни для состояния психического здоровья?) В нашем наборе данных

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

Т.е. включить пропущенные значения в выборку.

Который возвращает следующие значения

Don't know 563

Somewhat easy 266

Very easy 206

Somewhat difficult 126

Very difficult 98

Name: leave, dtype: int64

Для кодирования этих данных, сопоставим каждое значение с числом.

df['leave'] = df['leave'].map({'Very difficult': 0,

'Somewhat difficult': 1,

'Don\'t know': 2,

'Somewhat easy': 3,

'Very easy': 4})

pandas.map () используется для отображения значений из двух серий, имеющих один и тот же столбец. Для отображения двух рядов последний столбец первого ряда должен совпадать с индексным столбцом второго ряда, также значения должны быть уникальными.

Этот процесс известен как **Label Encoding** и **sklearn** может сделать это за нас.

from sklearn import preprocessing

label\_encoder = preprocessing.LabelEncoder()

label\_encoder.fit(df['leave'])

label\_encoder.transform(df['leave'])

В некоторых случаях полезны LabelEncoder или DictVectorizor, но, на мой взгляд, они весьма ограничены из-за порядочности.

LabelEncoder может превратить [собака, кошка, собака, мышь, кошка] в [1,2,1,3,2], но тогда навязанная ординальность означает, что среднее значение для собаки и мыши - это кошка. Тем не менее, существуют алгоритмы, такие как деревья решений и случайные леса, которые могут отлично работать с категориальными переменными, и LabelEncoder можно использовать для хранения значений, используя меньше дискового пространства.

Преимущество One-Hot-Encoding заключается в том, что результат является двоичным, а не порядковым, и что все находится в ортогональном векторном пространстве. Недостаток в том, что для высокой мощности пространство функций может быстро взорваться, и вы начнете бороться с проклятием размерности. В этих случаях я обычно использую горячее кодирование с последующим PCA для уменьшения размерности. Я считаю, что разумная комбинация «горячий» плюс «PCA» редко может быть побеждена другими схемами кодирования. PCA обнаруживает линейное перекрытие, поэтому, естественно, будет стремиться сгруппировать аналогичные функции в одну и ту же функцию.

**Examples**

LabelEncoder can be used to normalize labels.

>>>

**>>> from** **sklearn** **import** preprocessing

**>>>** le = preprocessing.LabelEncoder()

**>>>** le.fit([1, 2, 2, 6])

LabelEncoder()

**>>>** le.

array([1, 2, 6])

**>>>** le.transform([1, 1, 2, 6])

array([0, 0, 1, 2]...)

**>>>** le.inverse\_transform([0, 0, 1, 2])

array([1, 1, 2, 6])

It can also be used to transform non-numerical labels (as long as they are hashable and comparable) to numerical labels.

>>>

**>>>** le = preprocessing.LabelEncoder()

**>>>** le.fit(["paris", "paris", "tokyo", "amsterdam"])

LabelEncoder()

**>>>** list(le.classes\_)

['amsterdam', 'paris', 'tokyo']

**>>>** le.transform(["tokyo", "tokyo", "paris"])

array([2, 2, 1]...)

**>>>** list(le.inverse\_transform([2, 2, 1]))

['tokyo', 'tokyo', 'paris']

Проблема с этим подходом заключается в том, что вы вводите порядок, который может отсутствовать в исходных данных. В нашем случае можно утверждать, что данные являются ранжированными («*Very difficult*» меньше «*Somewhat difficult*», который меньше «*Very easy*», который меньше «*Somewhat easy*»), но в большинстве своем категориальные данные не имеют порядка. Например, если у вас есть признак, обозначающий вид животного, зачастую высказывание кошка больше собаки не имеет смысла. Опасность кодирования меток заключается в том, что ваш алгоритм может научиться отдавать предпочтение собакам или кошкам из-за искусственных порядковых значений, введенных вами во время кодирования.

Общим решением для кодирования номинальных данных является *one-hot-encoding*.  
Вместо того, чтобы заменять категориальное значение на числовое значение (кодирование меток), как показано ниже

|  |  |  |
| --- | --- | --- |
| id | type | numerical |
| 1 | cat | 1 |
| 2 | dog | 2 |
| 3 | snake | 3 |
| 4 | cat | 1 |
| 5 | dog | 2 |
| 6 | turtle | 4 |
| 7 | dog | 2 |

Вместо этого мы создаем столбец для каждого значения и используем 1 и 0 для обозначения выражения каждого значения. Эти новые столбцы часто называются фиктивными переменными.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| id | type | is\_cat | is\_dog | is\_snake | is\_turtle |
| 1 | cat | 1 | 0 | 0 | 0 |
| 2 | dog | 0 | 1 | 0 | 0 |
| 3 | snake | 0 | 0 | 1 | 0 |
| 4 | cat | 1 | 0 | 0 | 0 |
| 5 | dog | 0 | 1 | 0 | 0 |
| 6 | turle | 0 | 0 | 0 | 1 |
| 7 | dog | 0 | 1 | 0 | 0 |

Вы можете выполнить *one-hot-encoding* непосредственно в **Pandas** или использовать **sklearn**, хотя **sklearn** немного более прозрачен, поскольку *one-hot-encoding* из него работает только для целых значений. В нашем примере (где входные данные представляют собой строки) нам нужно сначала выполнить кодировку меток, а затем *one-hot-encoding*.

# Using Pandas

import pandas as pd

pd.get\_dummies(features['leave'])

# Using sklearn

from sklearn.preprocessing import LabelEncoder, OneHotEncoder

label\_encoder = LabelEncoder()

ohe = OneHotEncoder(categorical\_features = ['leave'])

label\_encoded\_data = label\_encoder.fit\_transform(features['leave'])

ohe.fit\_transform(label\_encoded\_data.reshape(-1,1))

# pandas.get\_dummies (преобразование категориальных данных в фиктивные)

**pandas.get\_dummies(*data*, *prefix=None*, *prefix\_sep='\_'*, *dummy\_na=False*, *columns=None*, *sparse=False*, *drop\_first=False*, *dtype=None*)**[**[source]**](https://github.com/pandas-dev/pandas/blob/v1.1.3/pandas/core/reshape/reshape.py#L725-L910)

Convert categorical variable into dummy/indicator variables.

**Parameters**

***array-like, Series, or DataFrame***

Data of which to get dummy indicators.

**prefix*str, list of str, or dict of str, default None***

String to append DataFrame column names. Pass a list with length equal to the number of columns when calling get\_dummies on a DataFrame. Alternatively, prefix can be a dictionary mapping column names to prefixes.

**prefix\_sep*str, default ‘\_’***

If appending prefix, separator/delimiter to use. Or pass a list or dictionary as with prefix.

**dummy\_na*bool, default False***

Add a column to indicate NaNs, if False NaNs are ignored.

**columns*list-like, default None***

Column names in the DataFrame to be encoded. If columns is None then all the columns with object or category dtype will be converted.

**sparse*bool, default False***

Whether the dummy-encoded columns should be backed by a **SparseArray** (True) or a regular NumPy array (False).

**drop\_first*bool, default False***

Whether to get k-1 dummies out of k categorical levels by removing the first level.

**dtype*dtype, default np.uint8***

Data type for new columns. Only a single dtype is allowed.

*New in version 0.23.0.*

**Returns**

**DataFrame**

Dummy-coded data.

**See also**

[**Series.str.get\_dummies**](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.str.get_dummies.html#pandas.Series.str.get_dummies)

Convert Series to dummy codes.

**Examples**

**>>>** s = pd.Series(list('abca'))

**>>>** pd.get\_dummies(s)

a b c

0 1 0 0

1 0 1 0

2 0 0 1

3 1 0 0

**>>>** s1 = ['a', 'b', np.nan]

**>>>** pd.get\_dummies(s1)

a b

0 1 0

1 0 1

2 0 0

**>>>** pd.get\_dummies(s1, dummy\_na=**True**)

a b NaN

0 1 0 0

1 0 1 0

2 0 0 1

**>>>** df = pd.DataFrame({'A': ['a', 'b', 'a'], 'B': ['b', 'a', 'c'],

**...**  'C': [1, 2, 3]})

**>>>** pd.get\_dummies(df, prefix=['col1', 'col2'])

C col1\_a col1\_b col2\_a col2\_b col2\_c

0 1 1 0 0 1 0

1 2 0 1 1 0 0

2 3 1 0 0 0 1

**>>>** pd.get\_dummies(pd.Series(list('abcaa')))

a b c

0 1 0 0

1 0 1 0

2 0 0 1

3 1 0 0

4 1 0 0

**>>>** pd.get\_dummies(pd.Series(list('abcaa')), drop\_first=**True**)

b c

0 0 0

1 1 0

2 0 1

3 0 0

4 0 0

**>>>** pd.get\_dummies(pd.Series(list('abc')), dtype=float)

a b c

0 1.0 0.0 0.0

1 0.0 1.0 0.0

2 0.0 0.0 1.0

**Нормализация тренировочных данных**

На этом этапе мы успешно очистили наши данные и превратили их в форму, которая подходит для алгоритмов машинного обучения. Однако на данном этапе мы должны рассмотреть вопрос о том, полезен ли какой-либо метод нормализации данных для нашего алгоритма. Это зависит от данных и алгоритма, который мы планируем реализовать.  
  
ML алгоритмы, которые требуют нормализации данных:

* Логистическая регрессия
* Метод опорных векторов
* Нейронная сеть
* PCA

ML алгоритмы, которые не требуют нормализации данных:

* Деревья принятия решений (и случайные леса)
* Градиентный бустинг
* Наивный Байес

Примечание: приведенные выше списки ни в коем случае не являются исчерпывающими, а просто служат примером.

Предположим, у вас есть набор данных с различными единицами: температура в Кельвине, относительная влажность и день года. Мы можем увидеть следующие диапазоны для каждой функции.

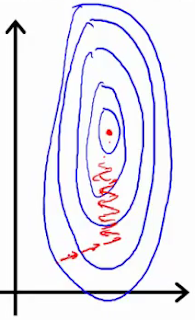
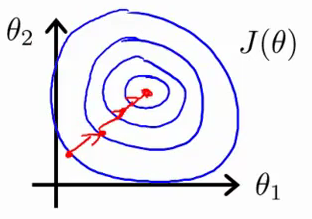
* Температура: от 270 K до 305 K
* Влажность: от 0 до 1 (т. е. Влажность 30%, равная 0,3)
* День года: от 0 до 365

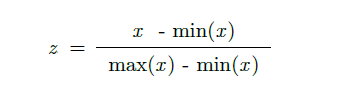
Когда вы смотрите на эти значения, вы интуитивно нормализуете значения. Например, вы знаете, что увеличение на 0,5 (=50%) для влажности намного более значимо, чем увеличение на 0,5 для температуры. И если мы не будем нормализовать эти данные, наш алгоритм может научиться использовать температуру в качестве основного предиктора просто потому, что масштаб является наибольшим (и, следовательно, изменения в значениях температуры наиболее значительны для алгоритма). Нормализация данных позволяет всем признакам вносить одинаковый вклад (или, что более точно, позволяет добавлять признаки в зависимости от их важности, а не их масштаба).

**Алгоритм нормализации**

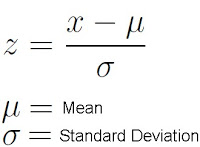
Если вы используете такой инструмент, как градиентный спуск, для оптимизации алгоритма, нормализация данных позволяет проводить последовательное обновление весов во всех измерениях.

Первое изображение представляет две функции с разными шкалами, в то время как последняя представляет собой нормализованное пространство признаков. Оптимизация градиентным спуском в первом случае может занять большее количество времени и в конечном итоге не прийти к минимуму.

[](http://3.bp.blogspot.com/-x5EGJe3Ogts/Ws3FkdtGMdI/AAAAAAAACTM/2TN0BVnlXsU6JBAzD9R9_DxzvJtvUpamwCK4BGAYYCw/s1600/GeAcX.png)[](http://3.bp.blogspot.com/-IEt8tPrAMk8/Ws3FjjV1KaI/AAAAAAAACTE/TSiaBrXzBj8SYoeQGqCeZizNDN8dE_SpwCK4BGAYYCw/s1600/20b0e3520bfa7d69d9bbbcfdb9d07f56.png)  
  
Существует несколько различных методов нормализации данных, самые популярные из них:  
  
Нормализация Min-max устанавливает наименьшее наблюдаемое значение равным 0, а наибольшее наблюдаемое значение — 1.

[](http://4.bp.blogspot.com/-TzItNvr-WL8/Ws3FGogmDtI/AAAAAAAACSg/bI9kwL3fbgQWyoans-tyLBMJhzU4IjF7ACK4BGAYYCw/s1600/main-qimg-0d692d88876aeb26b1f1a578d1c5a94e.png)

Нормализация на стандартное отклонение.

[](http://2.bp.blogspot.com/-eQgNjvTRcQM/Ws3E6SQxUpI/AAAAAAAACSU/juIH7Wi0esUDtErVXf48kyFXZyKSOusbACK4BGAYYCw/s1600/QiV53.jpg)

Для выполнения нормализации мы можем использовать функции в **sklearn**.

# Feature scaling with StandardScaler

from sklearn.preprocessing import StandardScaler

scale\_features\_std = StandardScaler()

features\_train = scale\_features\_std.fit\_transform(features\_train)

features\_test = scale\_features\_std.transform(features\_test)

# Feature scaling with MinMaxScaler

from sklearn.preprocessing import MinMaxScaler

scale\_features\_mm = MinMaxScaler()

features\_train = scale\_features\_mm.fit\_transform(features\_train)

features\_test = scale\_features\_mm.transform(features\_test)

Несколько замечаний по этой реализации:  
На практике вы можете выбрать только определенные столбцы. Например, вам не нужно нормализовать фиктивные переменные из *one-hot-encoding*.

**Разделение данных для обучения и тестирования**

**Разделение данных на две подвыборки**

Одной из последних вещей, которые нам нужно будет сделать, чтобы подготовить данные для обучения, является разделение данных на обучающую и тестовую выборку. Выделение тестовой выборки необходимо для понимания того, что мы обучили алгоритм в достаточной степени (не произошло переобучение или недообучение)

from sklearn.model\_selection import train\_test\_split

features\_train, features\_test, labels\_train, labels\_test = train\_test\_split(features, labels, test\_size=0.2, random\_state = 0)

sklearn.model\_selection.**train\_test\_split**(*\*массивы*, *\* \* опции*)[[источник]](https://github.com/scikit-learn/scikit-learn/blob/0fb307bf3/sklearn/model_selection/_split.py#L2029)

Разбейте массивы или матрицы на случайные последовательности и тестовые подмножества

Быстрая утилита, которая обертывает проверку ввода и next(ShuffleSplit().split(X, y)) и применение к входным данным в один вызов для разбиения (и, возможно, субдискретизации) данных в один лайнер.

Подробнее читайте в руководстве [пользователя](https://scikit-learn.org/stable/modules/cross_validation.html#cross-validation).

**Параметры**

**\* массивы*последовательность индексируемых объектов с одинаковой длиной / формой[0]***

Допустимыми входными данными являются списки, массивы numpy, scipy-sparse матрицы или фреймы данных панд.

**test\_size*float или int, по умолчанию=None***

Если поплавок, то должен находиться между 0.0 и 1.0 и представлять собой пропорцию набора данных для включения в тестовое разделение. Если int, то представляет собой абсолютное число тестовых образцов. Если нет, то значение устанавливается равным дополнение размера поезда. Если train\_size также нет, то это будет установите значение 0.25.

**train\_size*float или int, по умолчанию=None***

Если поплавок, то должен находиться между 0.0 и 1.0 и представлять собой доля набора данных, включаемого в раздел поезда. Если int, представляет собой абсолютное число выборок поездов. Если Нет, это значение автоматически устанавливается в дополнение к размеру теста.

**random\_state*int или экземпляр RandomState, default=None***

Управляет перетасовкой, применяемой к данным перед применением разделения. Передайте int для воспроизводимого вывода через несколько вызовов функций. См[. Глоссарий](https://scikit-learn.org/stable/glossary.html#term-random-state).

**shuffle*bool, default=True***

Следует ли перетасовывать данные перед разделением. If shuffle=False тогда стратификации не должно быть никакой.

**stratify*array-like, default=None***

Если нет, то данные разбиваются стратифицированным образом, используя это как метки классов.

**ВОЗВРАТ**

***список разбиения, длина=2 \* len (массивы)***

Список, содержащий поезд-тестовое разделение входных данных.

*Новое в версии 0.16:*если входные данные разрежены, то выходные данные будут scipy.sparse.csr\_matrix. В противном случае тип вывода будет таким же, как и у тип ввода.

**Примеры**

>>>

**>** Если мне нужно случайное разделение поезда / теста, я использую вспомогательную функцию sklearn:

In [1]: from sklearn.model\_selection import train\_test\_split

...: train\_test\_split([1,2,3,4,5,6])

...:

Out[1]: [[1, 6, 4, 2], [5, 3]]

Каков самый краткий способ получить не перетасованный поезд / тестовый сплит, т. е.

[[1,2,3,4], [5,6]]

**Редактировать** в настоящее время я использую

train, test = data[:int(len(data) \* 0.75)], data[int(len(data) \* 0.75):]

**Разделение данных на три подвыборки**

Можно пойти дальше и разделять данные на три подмножества: обучение, валидация и отложенная выборка. Данные обучения используются для «обучения» модели, данные валидации используются для поиска лучшей архитектуры модели, а отложенная выборка зарезервирована для финальной оценки нашей модели. При построении модели нам часто дают выбор в отношении общего дизайна модели; данные валидации позволяют нам оценивать несколько проектов в поисках лучшего дизайна, но при этом мы «подгоняем» дизайн нашей модели по этому подмножеству. Таким образом, тестовые данные по-прежнему полезны при определении того, насколько хорошо наша модель будет обобщать то, чему она научилась для новых данных.