mlcourse.ai – Open Machine Learning Course

Author: Yury Kashnitsky. Translated and edited by Christina Butsko, Yuanyuan Pao, Anastasia Manokhina, Sergey Isaev and Artem Trunov.

Topic 1. Exploratory data analysis with Pandas

Article outline

1. Demonstration of main Pandas methods

Well... There are dozens of cool tutorials on Pandas and visual data analysis. If you are already familiar with these topics, you can switch to the 3rd article in the series, where we get into machine learning.

Pandas is a Python library that provides extensive means for data analysis. Data scientists often work with data stored in table formats like `.csv`, `.tsv`, or `.xlsx`. Pandas makes it very convenient to load, process, and analyze such tabular data using SQL-like queries. In conjunction with `Matplotlib` and `Seaborn`, `Pandas` provides a wide range of opportunities for visual analysis of tabular data.

The main data structures in `Pandas` are implemented with **Series** and **DataFrame** classes. The former is a one-dimensional indexed array of some fixed data type. The latter is a two-dimensional data structure - a table - where each column contains data of the same type. You can see it as a dictionary of `Series` instances. `DataFrames` are great for representing real data: rows correspond to instances (examples, observations, etc.), and columns correspond to features of these instances.

In [1]:

```
import numpy as np
import pandas as pd
pd.set_option("display.precision", 2)
```

We'll demonstrate the main methods in action by analyzing a dataset on the churn rate of telecom operator clients. Let's read the data (using the `read_csv` method), and take a look at the first 5 lines using the `head` method:

In [2]:

```
df = pd.read_csv('../../data/telecom_churn.csv')
df.head()
```

Out[2]:

	State	Account length	Area code	International plan	Voice mail plan	Number vmail messages	Total day minutes	Total day calls	Total day charge	Total eve minutes	Total eve calls	Total eve charge	Total night minutes	Total night calls	Total night charge
0	KS	128	415	No	Yes	25	265.1	110	45.07	197.4	99	16.78	244.7	91	11.01
1	OH	107	415	No	Yes	26	161.6	123	27.47	195.5	103	16.62	254.4	103	11.45
2	NJ	137	415	No	No	0	243.4	114	41.38	121.2	110	10.30	162.6	104	7.32
3	OH	84	408	Yes	No	0	299.4	71	50.90	61.9	88	5.26	196.9	89	8.86

4	OK	75	415	Yes	No	0	166.7	113	28.34	148.3	122	12.61	186.9	121	8.41

▶ **Printing DataFrames in Jupyter notebooks**

Recall that each row corresponds to one client, an instance, and columns are features of this instance.

Let's have a look at data dimensionality, feature names, and feature types.

In [3]:

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

```
(3333, 20)
```

From the output, we can see that the table contains 3333 rows and 20 columns.

Now let's try printing out column names using `columns`:

In [4]:

```
print(df.columns)
```

```
Index(['State', 'Account length', 'Area code', 'International plan',
       'Voice mail plan', 'Number vmail messages', 'Total day minutes',
       'Total day calls', 'Total day charge', 'Total eve minutes',
       'Total eve calls', 'Total eve charge', 'Total night minutes',
       'Total night calls', 'Total night charge', 'Total intl minutes',
       'Total intl calls', 'Total intl charge', 'Customer service calls',
       'Churn'],
      dtype='object')
```

We can use the `info()` method to output some general information about the dataframe:

In [5]:

```
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3333 entries, 0 to 3332
Data columns (total 20 columns):
State                     3333 non-null object
Account length            3333 non-null int64
Area code                 3333 non-null int64
International plan        3333 non-null object
Voice mail plan           3333 non-null object
Number vmail messages     3333 non-null int64
Total day minutes         3333 non-null float64
Total day calls           3333 non-null int64
Total day charge          3333 non-null float64
Total eve minutes         3333 non-null float64
Total eve calls           3333 non-null int64
Total eve charge          3333 non-null float64
Total night minutes       3333 non-null float64
Total night calls         3333 non-null int64
Total night charge        3333 non-null float64
Total intl minutes        3333 non-null float64
Total intl calls          3333 non-null int64
Total intl charge         3333 non-null float64
Customer service calls    3333 non-null int64
Churn                     3333 non-null bool
dtypes: bool(1), float64(8), int64(8), object(3)
memory usage: 498.1+ KB
None
```

`bool`, `int64`, `float64` and `object` are the data types of our features. We see that one feature is logical (`bool`), 3 features are of type `object`, and 16 features are numeric. With this same method, we can easily see if there are any missing values. Here, there are none because each column contains 3333 observations, the same

number of rows we saw before with `shape`.

We can change the column type with the `astype` method. Let's apply this method to the `Churn` feature to convert it into `int64`:

In [6]:

```
df['Churn'] = df['Churn'].astype('int64')
```

The `describe` method shows basic statistical characteristics of each numerical feature (`int64` and `float64` types): number of non-missing values, mean, standard deviation, range, median, 0.25 and 0.75 quartiles.

In [7]:

```
df.describe()
```

Out[7]:

	Account length	Area code	Number vmail messages	Total day minutes	Total day calls	Total day charge	Total eve minutes	Total eve calls	Total eve charge	Total night minutes	Total night calls	Total night charge	Total intl minutes
count	3333.00	3333.00	3333.00	3333.00	3333.00	3333.00	3333.00	3333.00	3333.00	3333.00	3333.00	3333.00	3333.00
mean	101.06	437.18	8.10	179.78	100.44	30.56	200.98	100.11	17.08	200.87	100.11	9.04	10.24
std	39.82	42.37	13.69	54.47	20.07	9.26	50.71	19.92	4.31	50.57	19.57	2.28	2.79
min	1.00	408.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	23.20	33.00	1.04	0.00
25%	74.00	408.00	0.00	143.70	87.00	24.43	166.60	87.00	14.16	167.00	87.00	7.52	8.50
50%	101.00	415.00	0.00	179.40	101.00	30.50	201.40	100.00	17.12	201.20	100.00	9.05	10.30
75%	127.00	510.00	20.00	216.40	114.00	36.79	235.30	114.00	20.00	235.30	113.00	10.59	12.10
max	243.00	510.00	51.00	350.80	165.00	59.64	363.70	170.00	30.91	395.00	175.00	17.77	20.00

In order to see statistics on non-numerical features, one has to explicitly indicate data types of interest in the `include` parameter.

In [8]:

```
df.describe(include=['object', 'bool'])
```

Out[8]:

	State	International plan	Voice mail plan
count	3333	3333	3333
unique	51	2	2
top	WV	No	No
freq	106	3010	2411

For categorical (type `object`) and boolean (type `bool`) features we can use the `value_counts` method. Let's take a look at the distribution of `Churn`:

In [9]:

```
df['Churn'].value_counts()
```

Out[9]:

```
0    2850
1     483
Name: Churn, dtype: int64
```

2850 users out of 3333 are *loyal*; their `Churn` value is 0. To calculate fractions, pass `normalize=True` to the

`value_counts` function.

In [10]:

```
df['Churn'].value_counts(normalize=True)
```

Out[10]:

```
0    0.86
1    0.14
Name: Churn, dtype: float64
```

Sorting

A `DataFrame` can be sorted by the value of one of the variables (i.e columns). For example, we can sort by *Total day charge* (use `ascending=False` to sort in descending order):

In [11]:

```
df.sort_values(by='Total day charge', ascending=False).head()
```

Out[11]:

	State	Account length	Area code	International plan	Voice mail plan	Number vmail messages	Total day minutes	Total day calls	Total day charge	Total eve minutes	Total eve calls	Total eve charge	Total night minutes	Total night calls	To ni cha
365	CO	154	415	No	No	0	350.8	75	59.64	216.5	94	18.40	253.9	100	11
985	NY	64	415	Yes	No	0	346.8	55	58.96	249.5	79	21.21	275.4	102	12
2594	OH	115	510	Yes	No	0	345.3	81	58.70	203.4	106	17.29	217.5	107	9
156	OH	83	415	No	No	0	337.4	120	57.36	227.4	116	19.33	153.9	114	6
605	MO	112	415	No	No	0	335.5	77	57.04	212.5	109	18.06	265.0	132	11

We can also sort by multiple columns:

In [12]:

```
df.sort_values(by=['Churn', 'Total day charge'], ascending=[True, False]).head()
```

Out[12]:

	State	Account length	Area code	International plan	Voice mail plan	Number vmail messages	Total day minutes	Total day calls	Total day charge	Total eve minutes	Total eve calls	Total eve charge	Total night minutes	Total night calls	To ni cha
688	MN	13	510	No	Yes	21	315.6	105	53.65	208.9	71	17.76	260.1	123	11
2259	NC	210	415	No	Yes	31	313.8	87	53.35	147.7	103	12.55	192.7	97	8
534	LA	67	510	No	No	0	310.4	97	52.77	66.5	123	5.65	246.5	99	11
575	SD	114	415	No	Yes	36	309.9	90	52.68	200.3	89	17.03	183.5	105	8
2858	AL	141	510	No	Yes	28	308.0	123	52.36	247.8	128	21.06	152.9	103	6

Indexing and retrieving data

A `DataFrame` can be indexed in a few different ways.

To get a single column, you can use a `DataFrame['Name']` construction. Let's use this to answer a question about that column alone: **what is the proportion of churned users in our dataframe?**

In [13]:

```
df['Churn'].mean()
```

Out[13]:

```
0.14491449144914492
```

14.5% is actually quite bad for a company; such a churn rate can make the company go bankrupt.

Boolean indexing with one column is also very convenient. The syntax is `df[P(df['Name'])]`, where `P` is some logical condition that is checked for each element of the `Name` column. The result of such indexing is the `DataFrame` consisting only of rows that satisfy the `P` condition on the `Name` column.

Let's use it to answer the question:

What are average values of numerical features for churned users?

In [14]:

```
df[df['Churn'] == 1].mean()
```

Out[14]:

```
Account length            102.66
Area code                 437.82
Number vmail messages       5.12
Total day minutes         206.91
Total day calls           101.34
Total day charge           35.18
Total eve minutes         212.41
Total eve calls           100.56
Total eve charge           18.05
Total night minutes       205.23
Total night calls         100.40
Total night charge          9.24
Total intl minutes         10.70
Total intl calls            4.16
Total intl charge           2.89
Customer service calls      2.23
Churn                       1.00
dtype: float64
```

How much time (on average) do churned users spend on the phone during daytime?

In [15]:

```
df[df['Churn'] == 1]['Total day minutes'].mean()
```

Out[15]:

```
206.91407867494814
```

What is the maximum length of international calls among loyal users (`Churn == 0`) who do not have an international plan?

In [16]:

```
df[(df['Churn'] == 0) & (df['International plan'] == 'No')]['Total intl minutes'].max()
```

Out[16]:

```
18.9
```

DataFrames can be indexed by column name (label) or row name (index) or by the serial number of a row. The `loc` method is used for indexing by name, while `iloc()` is used for indexing by number.

In the first case below, we say *"give us the values of the rows with index from 0 to 5 (inclusive) and columns labeled from State to Area code (inclusive)"*. In the second case, we say *"give us the values of the first five rows in the first three columns"* (as in a typical Python slice: the maximal value is not included).

In [17]:

In [17]:

```
df.loc[0:5, 'State':'Area code']
```

Out[17]:

	State	Account length	Area code
0	KS	128	415
1	OH	107	415
2	NJ	137	415
3	OH	84	408
4	OK	75	415
5	AL	118	510

In [18]:

```
df.iloc[0:5, 0:3]
```

Out[18]:

	State	Account length	Area code
0	KS	128	415
1	OH	107	415
2	NJ	137	415
3	OH	84	408
4	OK	75	415

If we need the first or the last line of the data frame, we can use the `df[:1]` or `df[-1:]` construction:

In [19]:

```
df[-1:]
```

Out[19]:

	State	Account length	Area code	International plan	Voice mail plan	Number vmail messages	Total day minutes	Total day calls	Total day charge	Total eve minutes	Total eve calls	Total eve charge	Total night minutes	Total night calls	To ni cha
3332	TN	74	415	No	Yes	25	234.4	113	39.85	265.9	82	22.6	241.4	77	10

Applying Functions to Cells, Columns and Rows

To apply functions to each column, use `apply()`:

In [20]:

```
df.apply(np.max)
```

Out[20]:

```
State                            WY
Account length                  243
Area code                       510
International plan              Yes
Voice mail plan                 Yes
Number vmail messages            51
Total day minutes           3.5e+02
Total day calls                 165
Total day charge                 60
Total eve minutes           3.6e+02
Total eve calls                 170
```

```
Total eve charge              31
Total night minutes        4e+02
Total night calls            175
Total night charge            18
Total intl minutes            20
Total intl calls              20
Total intl charge            5.4
Customer service calls         9
Churn                          1
dtype: object
```

The `apply` method can also be used to apply a function to each row. To do this, specify `axis=1`. Lambda functions are very convenient in such scenarios. For example, if we need to select all states starting with 'W', we can do it like this:

In [21]:

```
df[df['State'].apply(lambda state: state[0] == 'W')].head()
```

Out[21]:

	State	Account length	Area code	International plan	Voice mail plan	Number vmail messages	Total day minutes	Total day calls	Total day charge	Total eve minutes	Total eve calls	Total eve charge	Total night minutes	Total night calls	Tota nigh charg
9	WV	141	415	Yes	Yes	37	258.6	84	43.96	222.0	111	18.87	326.4	97	14.6
26	WY	57	408	No	Yes	39	213.0	115	36.21	191.1	112	16.24	182.7	115	8.2
44	WI	64	510	No	No	0	154.0	67	26.18	225.8	118	19.19	265.3	86	11.9
49	WY	97	415	No	Yes	24	133.2	135	22.64	217.2	58	18.46	70.6	79	3.1
54	WY	87	415	No	No	0	151.0	83	25.67	219.7	116	18.67	203.9	127	9.1

The `map` method can be used to replace values in a column by passing a dictionary of the form `{old_value: new_value}` as its argument:

In [22]:

```
d = {'No' : False, 'Yes' : True}
df['International plan'] = df['International plan'].map(d)
df.head()
```

Out[22]:

	State	Account length	Area code	International plan	Voice mail plan	Number vmail messages	Total day minutes	Total day calls	Total day charge	Total eve minutes	Total eve calls	Total eve charge	Total night minutes	Total night calls	Total night charge
0	KS	128	415	False	Yes	25	265.1	110	45.07	197.4	99	16.78	244.7	91	11.01
1	OH	107	415	False	Yes	26	161.6	123	27.47	195.5	103	16.62	254.4	103	11.45
2	NJ	137	415	False	No	0	243.4	114	41.38	121.2	110	10.30	162.6	104	7.32
3	OH	84	408	True	No	0	299.4	71	50.90	61.9	88	5.26	196.9	89	8.86
4	OK	75	415	True	No	0	166.7	113	28.34	148.3	122	12.61	186.9	121	8.41

The same thing can be done with the `replace` method:

In [23]:

```
df = df.replace({'Voice mail plan': d})
df.head()
```

Out[23]:

Account	Area	International	Voice	Number	Total	Total	Total	Total	Total	Total	Total	Total	Total

	State	Account length	Area code	International plan	Voice mail plan	Number vmail messages	Total day minutes	Total day calls	Total day charge	Total eve minutes	Total eve calls	Total eve charge	Total night minutes	Total night calls	Total night charge
0	KS	128	415	False	True	25	265.1	110	45.07	197.4	99	16.78	244.7	91	11.01
1	OH	107	415	False	True	26	161.6	123	27.47	195.5	103	16.62	254.4	103	11.45
2	NJ	137	415	False	False	0	243.4	114	41.38	121.2	110	10.30	162.6	104	7.32
3	OH	84	408	True	False	0	299.4	71	50.90	61.9	88	5.26	196.9	89	8.86
4	OK	75	415	True	False	0	166.7	113	28.34	148.3	122	12.61	186.9	121	8.41

Grouping

In general, grouping data in Pandas works as follows:

```
df.groupby(by=grouping_columns)[columns_to_show].function()
```

1. **First, the `groupby` method divides the `grouping_columns` by their values. They become a new index in the resulting dataframe.**
2. **Then, columns of interest are selected (`columns_to_show`). If `columns_to_show` is not included, all non groupby clauses will be included.**
3. **Finally, one or several functions are applied to the obtained groups per selected columns.**

Here is an example where we group the data according to the values of the `Churn` variable and display statistics of three columns in each group:

In [24]:

```
columns_to_show = ['Total day minutes',
                   'Total eve minutes',
                   'Total night minutes']

df.groupby(['Churn'])[columns_to_show].describe(percentiles=[])
```

Out[24]:

	Total day minutes						Total eve minutes						Total night minutes					
	count	mean	std	min	50%	max	count	mean	std	min	50%	max	count	mean	std	min	50%	ma
Churn																		
0	2850.0	175.18	50.18	0.0	177.2	315.6	2850.0	199.04	50.29	0.0	199.6	361.8	2850.0	200.13	51.11	23.2	200.25	39
1	483.0	206.91	69.00	0.0	217.6	350.8	483.0	212.41	51.73	70.9	211.3	363.7	483.0	205.23	47.13	47.4	204.80	35

Let's do the same thing, but slightly differently by passing a list of functions to `agg()`:

In [25]:

```
columns_to_show = ['Total day minutes',
                   'Total eve minutes',
                   'Total night minutes']

df.groupby(['Churn'])[columns_to_show].agg([np.mean, np.std, np.min, np.max])
```

Out[25]:

	Total day minutes				Total eve minutes				Total night minutes			
	mean	std	amin	amax	mean	std	amin	amax	mean	std	amin	amax
Churn												
0	175.18	50.18	0.0	315.6	199.04	50.29	0.0	361.8	200.13	51.11	23.2	395.0

1	206.91	69.00	0.0	350.8	212.41	51.73	70.9	363.7	205.23	47.13	47.4	354.9

Total day minutes Total eve minutes Total night minutes

Summary tables

Suppose we want to see how the observations in our dataset are distributed in the context of two variables - `Churn` and `International plan`. To do so, we can build a **contingency table** using the `crosstab` method:

In [26]:

```
pd.crosstab(df['Churn'], df['International plan'])
```

Out[26]:

International plan	False	True
Churn		
0	2664	186
1	346	137

In [27]:

```
pd.crosstab(df['Churn'], df['Voice mail plan'], normalize=True)
```

Out[27]:

Voice mail plan	False	True
Churn		
0	0.60	0.25
1	0.12	0.02

We can see that most of the users are loyal and do not use additional services (International Plan/Voice mail).

This will resemble **pivot tables** to those familiar with Excel. And, of course, pivot tables are implemented in Pandas: the `pivot_table` method takes the following parameters:

- `values` – a list of variables to calculate statistics for,
- `index` – a list of variables to group data by,
- `aggfunc` – what statistics we need to calculate for groups, ex. sum, mean, maximum, minimum or something else.

Let's take a look at the average number of day, evening, and night calls by area code:

In [28]:

```
df.pivot_table(['Total day calls', 'Total eve calls', 'Total night calls'],
               ['Area code'], aggfunc='mean')
```

Out[28]:

	Total day calls	Total eve calls	Total night calls
Area code			
408	100.50	99.79	99.04
415	100.58	100.50	100.40
510	100.10	99.67	100.60

DataFrame transformations

Like many other things in Pandas, adding columns to a DataFrame is doable in many ways.

For example, if we want to calculate the total number of calls for all users, let's create the `total_calls` Series and paste it into the DataFrame:

In [29]:

```
total_calls = df['Total day calls'] + df['Total eve calls'] + \
              df['Total night calls'] + df['Total intl calls']
df.insert(loc=len(df.columns), column='Total calls', value=total_calls)
# loc parameter is the number of columns after which to insert the Series object
# we set it to len(df.columns) to paste it at the very end of the dataframe
df.head()
```

Out[29]:

	State	Account length	Area code	International plan	Voice mail plan	Number vmail messages	Total day minutes	Total day calls	Total day charge	Total eve minutes	...	Total eve charge	Total night minutes	Total night calls	Total night charge	m
0	KS	128	415	False	True	25	265.1	110	45.07	197.4	...	16.78	244.7	91	11.01	
1	OH	107	415	False	True	26	161.6	123	27.47	195.5	...	16.62	254.4	103	11.45	
2	NJ	137	415	False	False	0	243.4	114	41.38	121.2	...	10.30	162.6	104	7.32	
3	OH	84	408	True	False	0	299.4	71	50.90	61.9	...	5.26	196.9	89	8.86	
4	OK	75	415	True	False	0	166.7	113	28.34	148.3	...	12.61	186.9	121	8.41	

5 rows × 21 columns

It is possible to add a column more easily without creating an intermediate Series instance:

In [30]:

```
df['Total charge'] = df['Total day charge'] + df['Total eve charge'] + \
                     df['Total night charge'] + df['Total intl charge']
df.head()
```

Out[30]:

	State	Account length	Area code	International plan	Voice mail plan	Number vmail messages	Total day minutes	Total day calls	Total day charge	Total eve minutes	...	Total night minutes	Total night calls	Total night charge	Total intl minutes
0	KS	128	415	False	True	25	265.1	110	45.07	197.4	...	244.7	91	11.01	10.0
1	OH	107	415	False	True	26	161.6	123	27.47	195.5	...	254.4	103	11.45	13.7
2	NJ	137	415	False	False	0	243.4	114	41.38	121.2	...	162.6	104	7.32	12.2
3	OH	84	408	True	False	0	299.4	71	50.90	61.9	...	196.9	89	8.86	6.6
4	OK	75	415	True	False	0	166.7	113	28.34	148.3	...	186.9	121	8.41	10.1

5 rows × 22 columns

To delete columns or rows, use the `drop` method, passing the required indexes and the `axis` parameter (`1` if you delete columns, and nothing or `0` if you delete rows). The `inplace` argument tells whether to change the original DataFrame. With `inplace=False`, the `drop` method doesn't change the existing DataFrame and returns a new one with dropped rows or columns. With `inplace=True`, it alters the DataFrame.

In [31]:

```
# get rid of just created columns
df.drop(['Total charge', 'Total calls'], axis=1, inplace=True)
# and here’s how you can delete rows
df.drop([1, 2]).head()
```

Out[31]:

	State	Account	Area	International	Voice	Number	Total	Total	Total	Total	Total	Total	Total	Total	Total

	State	Account length	Area code	International plan	Voice mail plan	Number vmail messages	Total day minutes	Total day calls	Total day charge	Total eve minutes	Total eve calls	Total eve charge	Total night minutes	Total night calls	Total night charge
0	KS	128	415	False	True	25	265.1	110	45.07	197.4	99	16.78	244.7	91	11.01
3	OH	84	408	True	False	0	299.4	71	50.90	61.9	88	5.26	196.9	89	8.86
4	OK	75	415	True	False	0	166.7	113	28.34	148.3	122	12.61	186.9	121	8.41
5	AL	118	510	True	False	0	223.4	98	37.98	220.6	101	18.75	203.9	118	9.18
6	MA	121	510	False	True	24	218.2	88	37.09	348.5	108	29.62	212.6	118	9.57

2. First attempt at predicting telecom churn

Let's see how churn rate is related to the *International plan* feature. We'll do this using a `crosstab` contingency table and also through visual analysis with `Seaborn` (however, visual analysis will be covered more thoroughly in the next article).

In [32]:

```
pd.crosstab(df['Churn'], df['International plan'], margins=True)
```

Out[32]:

International plan	False	True	All
Churn			
0	2664	186	2850
1	346	137	483
All	3010	323	3333

In [33]:

```
# some imports to set up plotting
import matplotlib.pyplot as plt
# !pip install seaborn
import seaborn as sns
# import some nice vis settings
sns.set()
# Graphics in SVG format are more sharp and legible
%config InlineBackend.figure_format = 'svg'
```

In [34]:

```
sns.countplot(x='International plan', hue='Churn', data=df);
```

We see that, with *International Plan*, the churn rate is much higher, which is an interesting observation! Perhaps large and poorly controlled expenses with international calls are very conflict-prone and lead to dissatisfaction among the telecom operator's customers.

Next, let's look at another important feature – *Customer service calls*. Let's also make a summary table and a picture.

In [35]:

```
pd.crosstab(df['Churn'], df['Customer service calls'], margins=True)
```

Out[35]:

Customer service calls	0	1	2	3	4	5	6	7	8	9	All
Churn											
0	605	1059	672	385	90	26	8	4	1	0	2850
1	92	122	87	44	76	40	14	5	1	2	483

Customer service calls	0	1	2	3	4	5	6	7	8	9	All
All	697	1181	759	429	166	66	22	9	2	2	3333

Churn

In [36]:

```
sns.countplot(x='Customer service calls', hue='Churn', data=df);
```

Although it's not so obvious from the summary table, it's easy to see from the above plot that the churn rate increases sharply from 4 customer service calls and above.

Now let's add a binary feature to our DataFrame – `Customer service calls > 3`**. And once again, let's see how it relates to churn.**

In [37]:

```
df['Many_service_calls'] = (df['Customer service calls'] > 3).astype('int')

pd.crosstab(df['Many_service_calls'], df['Churn'], margins=True)
```

Out[37]:

Churn	0	1	All
Many_service_calls			
0	2721	345	3066
1	129	138	267
All	2850	483	3333

In [38]:

```
sns.countplot(x='Many_service_calls', hue='Churn', data=df);
```

Let’s construct another contingency table that relates *Churn* with both *International plan* and freshly created *Many_service_calls*.

In [39]:

```
pd.crosstab(df['Many_service_calls'] & df['International plan'] , df['Churn'])
```

Out[39]:

Churn	0	1
row_0		
False	2841	464
True	9	19

Therefore, predicting that a customer is not loyal (*Churn*=1) in the case when the number of calls to the service center is greater than 3 and the *International Plan* is added (and predicting *Churn*=0 otherwise), we might expect an accuracy of 85.8% (we are mistaken only 464 + 9 times). This number, 85.8%, that we got through this very simple reasoning serves as a good starting point (*baseline*) for the further machine learning models that we will build.

As we move on through this course, recall that, before the advent of machine learning, the data analysis process looked something like this. Let's recap what we've covered:

- **The share of loyal clients in the dataset is 85.5%. The most naive model that always predicts a "loyal customer" on such data will guess right in about 85.5% of all cases. That is, the proportion of correct answers (*accuracy*) of subsequent models should be no less than this number, and will hopefully be significantly higher;**
- **With the help of a simple prediction that can be expressed by the following formula:** `International plan =`

`True & Customer Service calls > 3 => Churn = 1, else Churn = 0`, we can expect a guessing rate of 85.8%, which is just above 85.5%. Subsequently, we'll talk about decision trees and figure out how to find such rules **automatically** based only on the input data;
- We got these two baselines without applying machine learning, and they'll serve as the starting point for our subsequent models. If it turns out that with enormous effort, we increase accuracy by only 0.5%, persay, then possibly we are doing something wrong, and it suffices to confine ourselves to a simple "if-else" model with two conditions;
- Before training complex models, it is recommended to wrangle the data a bit, make some plots, and check simple assumptions. Moreover, in business applications of machine learning, they usually start with simple solutions and then experiment with more complex ones.

3. Demo assignment

To practice with Pandas and EDA, you can complete [this assignment] where you'll be analyzing socio-demographic data. The assignment is just for you to practice, and goes with [solution].

4. Useful resources

- The same notebook as an interactive web-based [Kaggle Kernel]
- ["Merging DataFrames with pandas"] - a tutorial by Max Plako within mlcourse.ai (full list of tutorials is [here])
- ["Handle different dataset with dask and trying a little dask ML"] - a tutorial by Irina Knyazeva within mlcourse.ai
- Main course [site], [course repo], and YouTube [channel]
- Official Pandas [documentation]
- Course materials as a [Kaggle Dataset]
- Medium ["story"] based on this notebook
- If you read Russian: an [article] on Habr.com with ~ the same material. And a [lecture] on YouTube
- [10 minutes to pandas]
- [Pandas cheatsheet PDF]
- GitHub repos: [Pandas exercises] and ["Effective Pandas"]
- [scipy-lectures.org] — tutorials on pandas, numpy, matplotlib and scikit-learn