Open Machine Learning Course. Topic 1. Exploratory Data Analysis with Pandas

Yury Kashnitskiy Follow
Feb 5, 2018 · 14 min read

With this article, we, OpenDataScience, launch an open Machine Learning course. This is not aimed at developing another *comprehensive* introductory course on machine learning or data analysis (so this is not a substitute for fundamental education or online/offline courses/specializations and books). The purpose of this series of articles is to quickly refresh your knowledge and help you find topics for further advancement. Our approach is similar to that of the authors of Deep Learning book, which starts off with a review of mathematics and basics of machine learning—short, concise, and with many references to other resources.

UPD: YouTube playlist with videolectures

The course is designed to perfectly balance theory and practice; therefore, each topic is followed by an **assignment** with a deadline in a week. You can also take part in several Kaggle Inclass **competitions** held during the course.

All materials are available as a **Kaggle Dataset** and in a GitHub repo.

The course is going to be actively discussed in the OpenDataScience Slack team. Please fill in this form to be invited. The next session of the course will start on **October 1**, 2018. Invitations will be sent in September.

Article outline

1. About the course
2. Assignments
3. Demonstration of main Pandas methods
4. First attempt on predicting telecom churn
5. Assignment #1
6. Useful resources

1. About the course

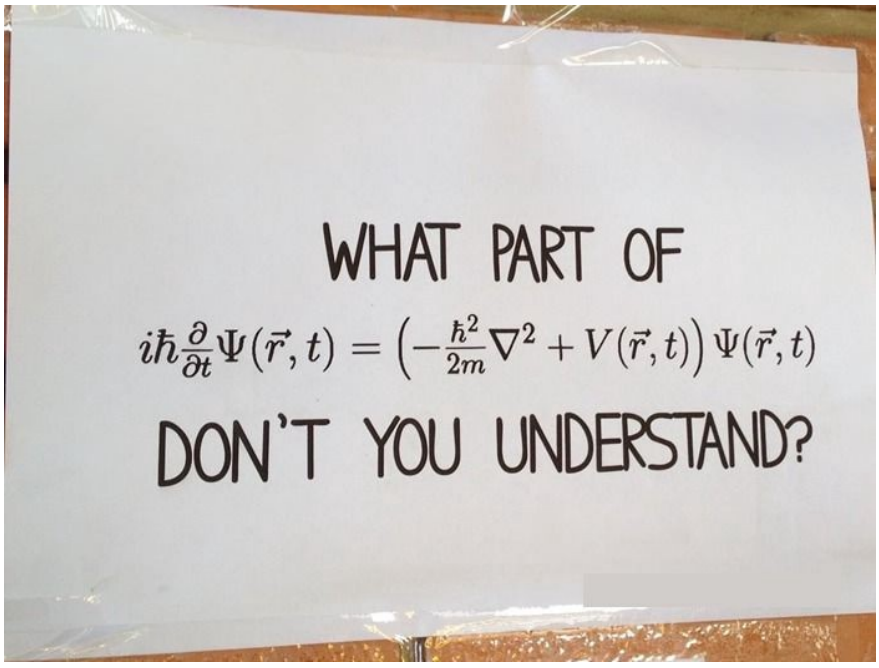
Syllabus

1. Exploratory Data Analysis with Pandas
2. Visual Data Analysis with Python

3. Classification, Decision Trees and k Nearest Neighbors
4. Linear Classification and Regression
5. Bagging and Random Forest
6. Feature Engineering and Feature Selection
7. Unsupervised Learning: Principal Component Analysis and Clustering
8. Vowpal Wabbit: Fast Learning with Gigabytes of Data
9. Time Series Analysis with Python, Predicting the future with Facebook Prophet
10. Gradient Boosting

Community

One of the most vivid advantages of our course is active community. If you join the OpenDataScience Slack team, you'll find the authors of articles and assignments right there in the same channel (#eng_mlcourse_open) eager to help you. This can help very much when you make your first steps in any discipline. Fill in this form to be invited. The form will ask you several questions about your background and skills, including a few easy math questions.



We chat informally, like humor and emoji. Not every MOOC can boast to have such an alive community.

Prerequisites

The prerequisites are the following: basic concepts from calculus, linear algebra, probability theory and statistics, and Python programming skills. If you need to catch up, a good resource will be Part I from the "Deep Learning" book and various math and Python online courses (for Python, CodeAcademy will do). More info is available on the corresponding Wiki page.

What software you'll need

As for now, you'll only need Anaconda (built with Python 3.6) to reproduce the code in the course. Later in the course you'll have to install other libraries like Xgboost and Vowpal Wabbit.

You can also resort to the Docker container with all necessary software already installed. More info is available on the corresponding Wiki page.

2. Assignments

- Each article comes with an assignment in the form of a Jupyter notebook. The task will be to fill in the missing code snippets and to answer questions in a Google Quiz form;
- Each assignment is due in a week with a hard deadline;
- Please discuss the course content (articles and assignments) in the #eng_mlcourse_open channel of the OpenDataScience Slack team or here in the comments to articles on Medium;
- The solutions to assignments will be sent to those who have submitted the corresponding Google form.

3. Demonstration of main Pandas methods

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

The following material is better viewed as a Jupyter notebook and can be reproduced locally with Jupyter if you clone the course repository.

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 (objects, observations, etc.), and columns correspond to features for each of the instances.

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 `read_csv`), and take a look at the first 5 lines using the `head` method:

```
import pandas as pd
import numpy as np
df = pd.read_csv('../../data/telecom_churn.csv')
df.head()
```

	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 intl minutes	Total intl calls	Total intl charge	Customer service calls	Churn
0	KS	128	415	No	Yes	25	265.1	110	45.07	197.4	99	16.78	10.0	3	2.70	1	False
1	OH	107	415	No	Yes	26	161.6	123	27.47	195.5	103	16.62	13.7	3	3.70	1	False
2	NJ	137	415	No	No	0	243.4	114	41.38	121.2	110	10.30	12.2	5	3.29	0	False
3	OH	84	408	Yes	No	0	299.4	71	50.90	61.9	88	5.26	6.6	7	1.78	2	False
4	OK	75	415	Yes	No	0	166.7	113	28.34	148.3	122	12.61	10.1	3	2.73	3	False

Recall that each row corresponds to one client, the **object** of our research, and columns are **features** of the object.

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

```
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 the column names using `columns`:

```
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:

```
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`:

```
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.

```
df.describe()
```

	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
count	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000
mean	101.064806	437.182418	8.099010	179.775098	100.435644	30.562307	200.980348	100.114311	17.083540	200.872037	100.107711
std	39.822106	42.371290	13.688365	54.467389	20.069084	9.259435	50.713844	19.922625	4.310668	50.573847	19.568609
min	1.000000	408.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	23.200000	33.000000
25%	74.000000	408.000000	0.000000	143.700000	87.000000	24.430000	166.600000	87.000000	14.160000	167.000000	87.000000
50%	101.000000	415.000000	0.000000	179.400000	101.000000	30.500000	201.400000	100.000000	17.120000	201.200000	100.000000
75%	127.000000	510.000000	20.000000	216.400000	114.000000	36.790000	235.300000	114.000000	20.000000	235.300000	113.000000
max	243.000000	510.000000	51.000000	350.800000	165.000000	59.640000	363.700000	170.000000	30.910000	395.000000	175.000000

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

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

	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 have a look at the distribution of `Churn`:

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

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

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

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

```
0    0.855086
1    0.144914
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):

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

	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 intl minutes	Total intl calls	Total intl charge	Customer service calls	Churn
365	CO	154	415	No	No	0	350.8	75	59.64	216.5	94	18.40	10.1	9	2.73	1	1
985	NY	64	415	Yes	No	0	346.8	55	58.96	249.5	79	21.21	13.3	9	3.59	1	1
2594	OH	115	510	Yes	No	0	345.3	81	58.70	203.4	106	17.29	11.8	8	3.19	1	1
156	OH	83	415	No	No	0	337.4	120	57.36	227.4	116	19.33	15.8	7	4.27	0	1
605	MO	112	415	No	No	0	335.5	77	57.04	212.5	109	18.06	12.7	8	3.43	2	1

Alternatively, we can also sort by multiple columns:

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

	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 intl minutes	Total intl calls	Total intl charge	Customer service calls	Churn
688	MN	13	510	No	Yes	21	315.6	105	53.65	208.9	71	17.76	12.1	3	3.27	3	0
2259	NC	210	415	No	Yes	31	313.8	87	53.35	147.7	103	12.55	10.1	7	2.73	3	0
534	LA	67	510	No	No	0	310.4	97	52.77	66.5	123	5.65	9.2	10	2.48	4	0
575	SD	114	415	No	Yes	36	309.9	90	52.68	200.3	89	17.03	14.2	2	3.83	1	0
2858	AL	141	510	No	Yes	28	308.0	123	52.36	247.8	128	21.06	7.4	3	2.00	1	0

Indexing and retrieving data

DataFrame can be indexed in 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?**

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

```
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 variables for churned users?

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

```
Account length            102.664596
Area code                 437.817805
Number vmail messages       5.115942
Total day minutes         206.914079
Total day calls           101.335404
Total day charge           35.175921
Total eve minutes         212.410145
Total eve calls           100.561077
Total eve charge           18.054969
Total night minutes       205.231677
Total night calls         100.399586
Total night charge          9.235528
Total intl minutes         10.700000
Total intl calls            4.163561
Total intl charge           2.889545
Customer service calls      2.229814
Churn                       1.000000
dtype: float64
```

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

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

```
206.91407867494814
```

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

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

```
18.899999999999999
```

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

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

	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

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

	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 last line of the data frame, we use the `df[:1]` or `df[-1:]` syntax.

Applying Functions to Cells, Columns and Rows

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

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

```
State                        WY
Account length              243
Area code                   510
International plan          Yes
Voice mail plan             Yes
Number vmail messages        51
Total day minutes         350.8
Total day calls             165
Total day charge          59.64
Total eve minutes         363.7
Total eve calls             170
Total eve charge          30.91
Total night minutes         395
Total night calls           175
Total night charge        17.77
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 line. 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:

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

	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 intl minutes	Total intl calls	Total intl charge	Customer service calls	Churn
9	WV	141	415	Yes	Yes	37	258.6	84	43.96	222.0	111	18.87	11.2	5	3.02	0	0
26	WY	57	408	No	Yes	39	213.0	115	36.21	191.1	112	16.24	9.5	3	2.57	0	0
44	WI	64	510	No	No	0	154.0	67	26.18	225.8	118	19.19	3.5	3	0.95	1	0
49	WY	97	415	No	Yes	24	133.2	135	22.64	217.2	58	18.46	11.0	3	2.97	1	0
54	WY	87	415	No	No	0	151.0	83	25.67	219.7	116	18.67	9.7	3	2.62	5	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:

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

	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 intl minutes	Total intl calls	Total intl charge	Customer service calls	Churn
0	KS	128	415	False	Yes	25	265.1	110	45.07	197.4	99	16.78	10.0	3	2.70	1	0
1	OH	107	415	False	Yes	26	161.6	123	27.47	195.5	103	16.62	13.7	3	3.70	1	0
2	NJ	137	415	False	No	0	243.4	114	41.38	121.2	110	10.30	12.2	5	3.29	0	0
3	OH	84	408	True	No	0	299.4	71	50.90	61.9	88	5.26	6.6	7	1.78	2	0
4	OK	75	415	True	No	0	166.7	113	28.34	148.3	122	12.61	10.1	3	2.73	3	0

Same thing can be done with the `replace` method:

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

	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 intl minutes	Total intl calls	Total intl charge	Customer service calls	Churn
0	KS	128	415	False	True	25	265.1	110	45.07	197.4	99	16.78	10.0	3	2.70	1	0
1	OH	107	415	False	True	26	161.6	123	27.47	195.5	103	16.62	13.7	3	3.70	1	0
2	NJ	137	415	False	False	0	243.4	114	41.38	121.2	110	10.30	12.2	5	3.29	0	0
3	OH	84	408	True	False	0	299.4	71	50.90	61.9	88	5.26	6.6	7	1.78	2	0
4	OK	75	415	True	False	0	166.7	113	28.34	148.3	122	12.61	10.1	3	2.73	3	0

Grouping

In general, grouping data in Pandas goes 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:

```
columns_to_show = ['Total day minutes', 'Total eve minutes',
                   'Total night minutes']
df.groupby(['Churn'])[columns_to_show].describe(percentiles=
```

	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%	max
Churn																		
0	2850.0	175.175754	50.181655	0.0	177.2	315.6	2850.0	199.043298	50.292175	0.0	199.6	361.8	2850.0	200.133193	51.105032	23.2	200.25	395.0
1	483.0	206.914079	68.997792	0.0	217.6	350.8	483.0	212.410145	51.728910	70.9	211.3	363.7	483.0	205.231677	47.132825	47.4	204.80	354.9

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

```
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])
```

	Total day minutes				Total eve minutes				Total night minutes			
	mean	std	amin	amax	mean	std	amin	amax	mean	std	amin	amax
Churn												
0	175.175754	50.181655	0.0	315.6	199.043298	50.292175	0.0	361.8	200.133193	51.105032	23.2	395.0
1	206.914079	68.997792	0.0	350.8	212.410145	51.728910	70.9	363.7	205.231677	47.132825	47.4	354.9

Summary tables

Suppose we want to see how the observations in our sample 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:

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

International plan	False	True
Churn		
0	2664	186
1	346	137

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

Voice mail plan	False	True
Churn		
0	0.602460	0.252625
1	0.120912	0.024002

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 - e.g sum, mean, maximum, minimum or something else.

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

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

Area code	Total day calls	Total eve calls	Total night calls
408	100.496420	99.788783	99.039379
415	100.576435	100.503927	100.398187
510	100.097619	99.671429	100.601190

DataFrame transformations

Like many other things in Pandas, adding columns to a DataFrame is doable in several 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:

```
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=t
df.head()
```

	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 intl minutes	Total intl calls	Total intl charge	Customer service calls	Churn	Total calls
0	KS	128	415	No	Yes	25	265.1	110	45.07	197.4	99	16.78	10.0	3	2.70	1	0	303
1	OH	107	415	No	Yes	26	161.6	123	27.47	195.5	103	16.62	13.7	3	3.70	1	0	332
2	NJ	137	415	No	No	0	243.4	114	41.38	121.2	110	10.30	12.2	5	3.29	0	0	333
3	OH	84	408	Yes	No	0	299.4	71	50.90	61.9	88	5.26	6.6	7	1.78	2	0	255
4	OK	75	415	Yes	No	0	166.7	113	28.34	148.3	122	12.61	10.1	3	2.73	3	0	359

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

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

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 intl minutes	Total intl calls	Total intl charge	Customer service calls	Churn	Total calls	Total charge
KS	128	415	No	Yes	25	265.1	110	45.07	197.4	99	16.78	10.0	3	2.70	1	0	303	75.56
OH	107	415	No	Yes	26	161.6	123	27.47	195.5	103	16.62	13.7	3	3.70	1	0	332	59.24
NJ	137	415	No	No	0	243.4	114	41.38	121.2	110	10.30	12.2	5	3.29	0	0	333	62.29
OH	84	408	Yes	No	0	299.4	71	50.90	61.9	88	5.26	6.6	7	1.78	2	0	255	66.80
OK	75	415	Yes	No	0	166.7	113	28.34	148.3	122	12.61	10.1	3	2.73	3	0	359	52.09

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.

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

	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 intl minutes	Total intl calls	Total intl charge	Customer service calls	Churn
0	KS	128	415	No	Yes	25	265.1	110	45.07	197.4	99	16.78	10.0	3	2.70	1	0
3	OH	84	408	Yes	No	0	299.4	71	50.90	61.9	88	5.26	6.6	7	1.78	2	0
4	OK	75	415	Yes	No	0	166.7	113	28.34	148.3	122	12.61	10.1	3	2.73	3	0
5	AL	118	510	Yes	No	0	223.4	98	37.98	220.6	101	18.75	6.3	6	1.70	0	0
6	MA	121	510	No	Yes	24	218.2	88	37.09	348.5	108	29.62	7.5	7	2.03	3	0

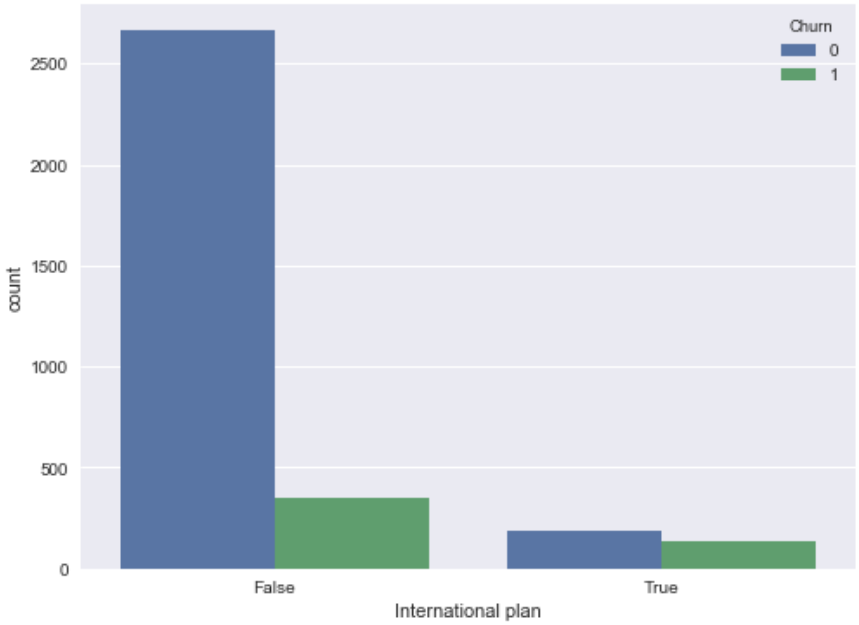
4. First attempt on predicting telecom churn

Let's see how churn rate is related to the *International plan* variable. 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).

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

International plan	No	Yes	All
Churn			
0	2664	186	2850
1	346	137	483
All	3010	323	3333

```
# some imports and "magic" commands to set up plotting
%matplotlib inline
import matplotlib.pyplot as plt
# pip install seaborn
import seaborn as sns
```



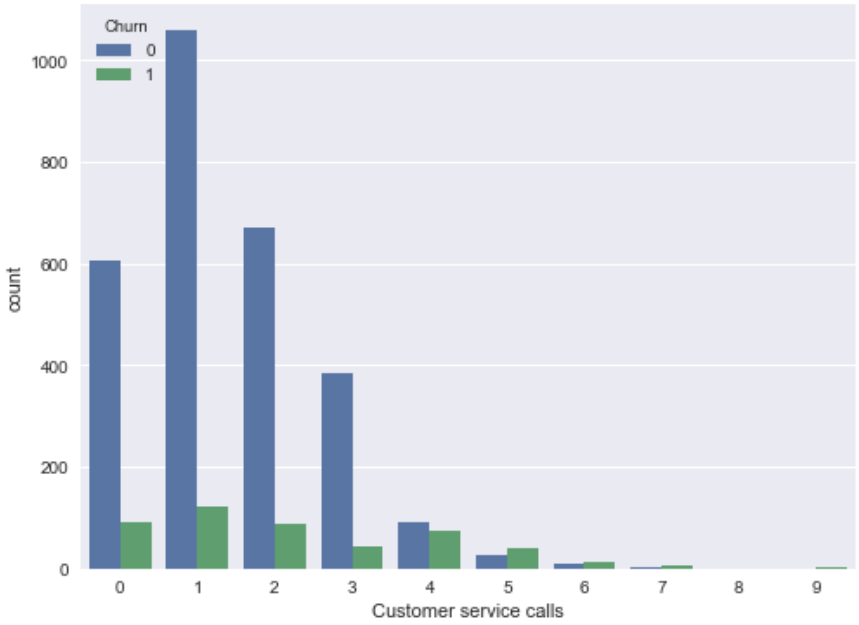
We see that, with *International Plan*, the churn rate is much higher, which is an interesting observation! Perhaps large and poorly controlled expenses with international calls are very conflict-prone and lead to dissatisfaction among the telecom operator's customers.

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

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

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
All	697	1181	759	429	166	66	22	9	2	2	3333

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

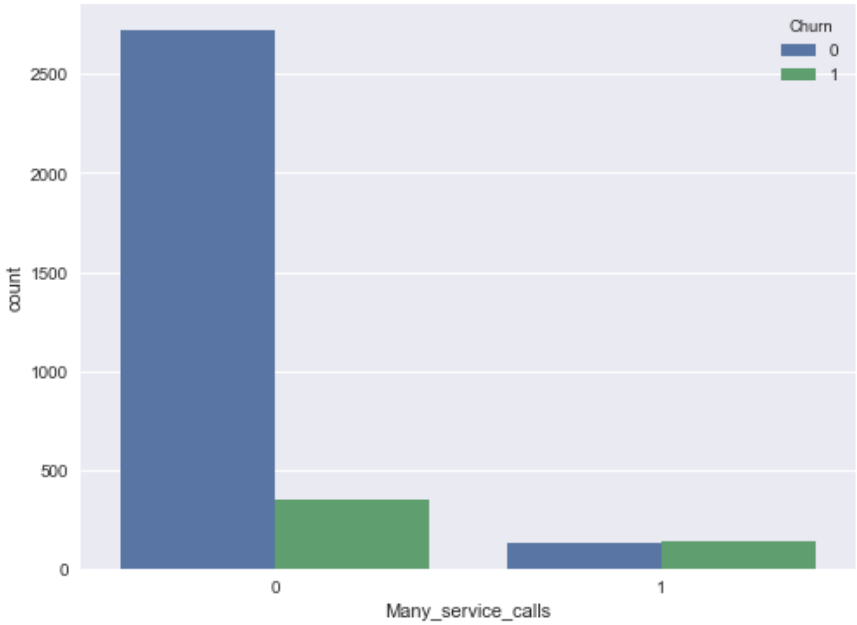


Perhaps, it is not so obvious from the summary table, but the picture clearly states that the churn rate strongly increases starting from 4 calls to the service center.

Let's now add a binary attribute to our DataFrame— `Customer service calls > 3` . And once again, let's see how it relates to the churn.

```
df['Many_service_calls'] = (df['Customer service calls'] > 3
pd.crosstab(df['Many_service_calls'], df['Churn'], margins=T
sns.countplot(x='Many_service_calls', hue='Churn', data=df);
```

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



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

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

Churn	0	1
row_0		
False	2841	464
True	9	19

Therefore, predicting that a customer will churn (*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 with very simple reasoning serves as a good starting point (*baseline*) for the further machine learning models that we will build.

As we move on in 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 sample 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 forecast that can be expressed by the following formula: "(Customer Service calls > 3) & (International plan = True) => 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 efforts, we increase the share of correct answers by 0.5% per se, then perhaps we are doing something wrong, and it suffices to confine ourselves to a simple model with two conditions;
- Before training complex models, it is recommended to manipulate 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.

5. Assignment #1

Full versions of assignments are announced each week in a new run of the course (October 1, 2018). Meanwhile, you can practice with a demo version: Kaggle Kernel, nbviewer.

6. Useful resources

- First of all, of course, official Pandas documentation
- 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
- This course's GitHub repository, same materials as a Kaggle Dataset

Authors: Yury Kashnitskiy, and Katya Demidova. Translated and edited by Yuanyuan Pao, Christina Butsko, Anastasia Manokhina, Egor Polusmak, Sergey Isaev, and Artem Trunov.