Exploratory Data Analysis(Part 1)

AcadView

May 27, 2018

---

1 Overview

Exploratory Data Analysis (EDA) is the first step in your data analysis process. Here, you make sense of the data you have and then figure out what questions you want to ask and how to frame them, as well as how best to manipulate your available data sources to get the answers you need.

You do this by taking a broad look at patterns, trends, outliers, unexpected results and so on in your existing data, using visual and quantitative methods to get a sense of the story this tells. You're looking for clues that suggest your logical next steps, questions or areas of research.

In this we will discuss all of the topics using pandas and seaborn that we have already studied.

2 Why do we use EDA

We use Exploratory Data Analysis (EDA) to tackle specific tasks such as:

- Spotting mistakes and missing data;
- Mapping out the underlying structure of the data;
- Identifying the most important variables;
- Listing anomalies and outliers;
- Testing a hypotheses / checking assumptions related to a specific model;
- Establishing a parsimonious model (one that can be used to explain the data with minimal predictor variables);

- Estimating parameters and figuring out the associated confidence intervals or margins of error.

3 Univariate Data Analysis using Pandas and Seaborn

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 Seaborn and Pandas provides a wide range of opportunities for visual analysis of tabular data.

3.1 Basics of reading and manipulating of the dataset

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:

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

The output is as follows:

	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

3.1.1 Dimensions of the data

```
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')
```

3.1.2 Describing the dataset

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()
```

The output is as follows:

	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

3.1.3 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()
```

The output is as follows:

	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

3.1.4 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()
```

The output is as follows:

3.2 Summary tables

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

3.3 Handling Missing values

Something that you also might want to check when you're exploring your data is whether or not the data set has any missing values.

Examining this is important because when some of your data is missing, the data set can lose expressiveness, which can lead to weak or biased analyses. Practically, this means that when you're missing values for certain features, the chances of your classification or predictions for the data being off only increase.

We will be using the following data for understanding missing value manipulation.

	first_name	last_name	age	sex	preTestScore	postTestScore
0	Jason	Miller	42.0	m	4.0	25.0
1	NaN	NaN	NaN	NaN	NaN	NaN
2	Tina	Ali	36.0	f	NaN	NaN
3	Jake	Milner	24.0	m	2.0	62.0
4	Amy	Cooze	73.0	f	3.0	70.0

- **Drop missing observations**

```
df_no_missing = df.dropna()
df_no_missing
```

	first_name	last_name	age	sex	preTestScore	postTestScore
0	Jason	Miller	42.0	m	4.0	25.0
3	Jake	Milner	24.0	m	2.0	62.0
4	Amy	Cooze	73.0	f	3.0	70.0

- **Drop rows where all cells in that row is NaN**

```
df_cleaned = df.dropna(how='all')
df_cleaned
```

	first_name	last_name	age	sex	preTestScore	postTestScore
0	Jason	Miller	42.0	m	4.0	25.0
2	Tina	Ali	36.0	f	NaN	NaN
3	Jake	Milner	24.0	m	2.0	62.0
4	Amy	Cooze	73.0	f	3.0	70.0

- **Create a new column full of missing values**

```
df['location'] = np.nan
df
```

	first_name	last_name	age	sex	preTestScore	postTestScore	location
0	Jason	Miller	42.0	m	4.0	25.0	NaN
1	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	Tina	Ali	36.0	f	NaN	NaN	NaN
3	Jake	Milner	24.0	m	2.0	62.0	NaN
4	Amy	Cooze	73.0	f	3.0	70.0	NaN

- **Drop column if they only contain missing values**

```
df.dropna(axis=1, how='all')
```

	first_name	last_name	age	sex	preTestScore	postTestScore
0	Jason	Miller	42.0	m	4.0	25.0
1	NaN	NaN	NaN	NaN	NaN	NaN

• **Fill in missing data with zeros**

```
df.fillna(0)
```

	first_name	last_name	age	sex	preTestScore	postTestScore	location
0	Jason	Miller	42.0	m	4.0	25.0	0.0
1	0	0	0.0	0	0.0	0.0	0.0
2	Tina	Ali	36.0	f	0.0	0.0	0.0
3	Jake	Milner	24.0	m	2.0	62.0	0.0
4	Amy	Cooze	73.0	f	3.0	70.0	0.0

- **Fill in missing in preTestScore with the mean value of preTestScore** inplace=True means that the changes are saved to the df right away.

4 Univariate visualization

Univariate analysis looks at one variable at a time. When we analyze a feature independently, we are usually mostly interested in the distribution of its values and ignore the other variables in the dataset.

Below, we will consider different statistical types of variables and the corresponding tools for their individual visual analysis.

Note:We'll demonstrate the main methods in action by analyzing a dataset on the churn rate of telecom operator clients.The same one we have used before to learn how to view datasets.

4.1 Quantitative features

Quantitative features take on ordered numerical values. Those values can be discrete, like integers, or continuous, like real numbers, and usually express a count or a measurement.

```
df["preTestScore"].fillna(df["preTestScore"].mean(), inplace=True)
df
```

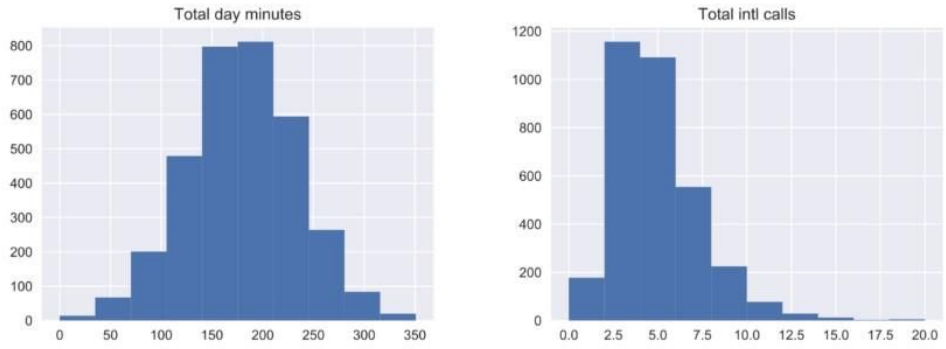
	first_name	last_name	age	sex	preTestScore	postTestScore	location
0	Jason	Miller	42.0	m	4.0	25.0	NaN

4.1.1 Histograms and density plots

The easiest way to take a look at the distribution of a numerical variable is to plot its histogram using the DataFrame's method hist().

```
features = ['Total day minutes', 'Total intl calls']
df[features].hist(figsize=(12, 4));
```

The output is as follows:



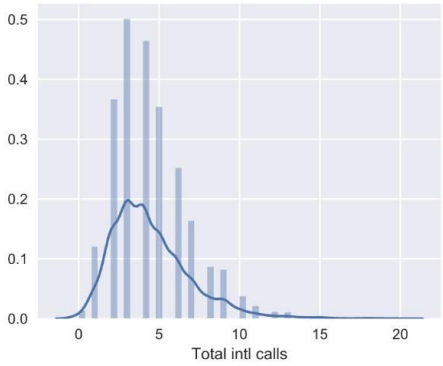
A histogram groups values into bins of equal value range. The shape of the histogram may contain clues about the underlying distribution type: Gaussian, exponential etc. You can also spot any skewness in its shape when the distribution is nearly regular but has some anomalies. Knowing the distribution of the feature values becomes important when you use Machine Learning methods that assume a particular type of it, most often Gaussian.

Note:-In the above plot, we see that the variable Total day minutes is normally distributed, while Total intl calls is prominently skewed right (its tail is longer on the right).[Recall this from the statistics and numpy class]

It is also possible to plot a distribution of observations with seaborn's distplot(). For example, let's look at the distribution of Total day minutes. By default, the plot displays both the histogram with the kernel density estimate (KDE) on top.

```
sns.distplot(df['Total intl calls']);
```

The output is as follows:



The height of the histogram bars here is normed and shows the density rather than the number of examples in each bin.

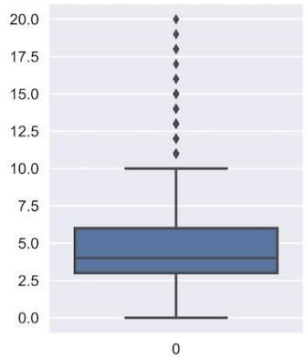
4.1.2 Boxplots

Another useful type of visualization is a box plot. seaborn does a great job here:

```
_, ax = plt.subplots(figsize=(3, 4))

sns.boxplot(data=df['Total intl calls'], ax=ax);
```

The output is as follows:



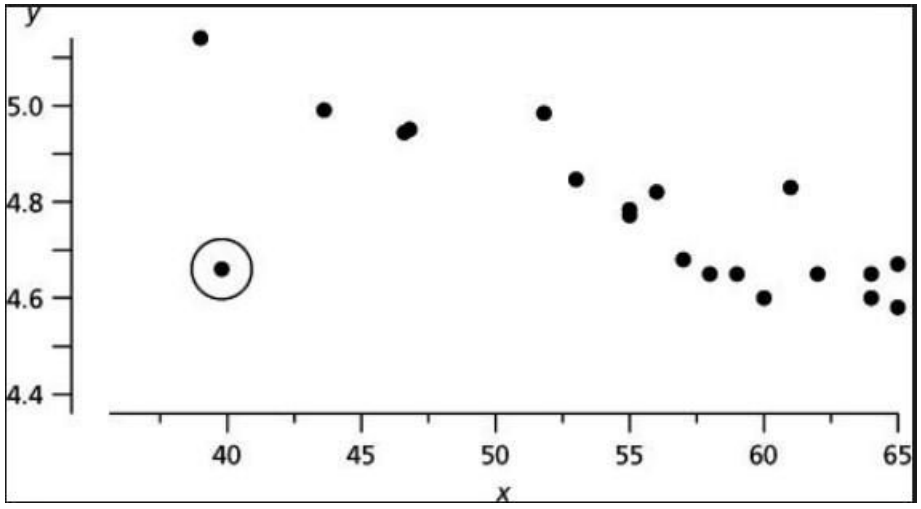
Let's see how to interpret a box plot. Its components are a box (obviously, this is why it is called a box plot), the so-called whiskers, and a number of individual points (outliers). The box by itself illustrates the interquartile spread of the distribution; its length is determined by the 25th (Q1) and 75th (Q3) percentiles. The vertical line inside the box marks the median (50%) of the distribution.

The whiskers are the lines extending from the box. They represent the entire scatter of data points, specifically the points that fall within the interval (Q11.5IQR, Q3+1.5IQR), where IQR=Q3Q1 is the interquartile range.

Outliers that fall out of the range bounded by the whiskers are plotted individually as black points along the central axis.
We can see that a large number of international calls is quite rare in our data.

Outliers
When modeling, it is important to clean the data sample to ensure that the observations best represent the problem.

Sometimes a dataset can contain extreme values that are outside the range of what is expected and unlike the other data. These are called outliers and often machine learning modeling and model skill in general can be improved by understanding and even removing these outlier values.



The circled point is the outlier in the data as it lies quite away from the distributiion of the data.

4.2 Categorical and binary features

Categorical features take on a fixed number of values. Each of these values assigns an observation to a corresponding group, known as a category, which reflects some qualitative property of this example.

4.2.1 Bar plot

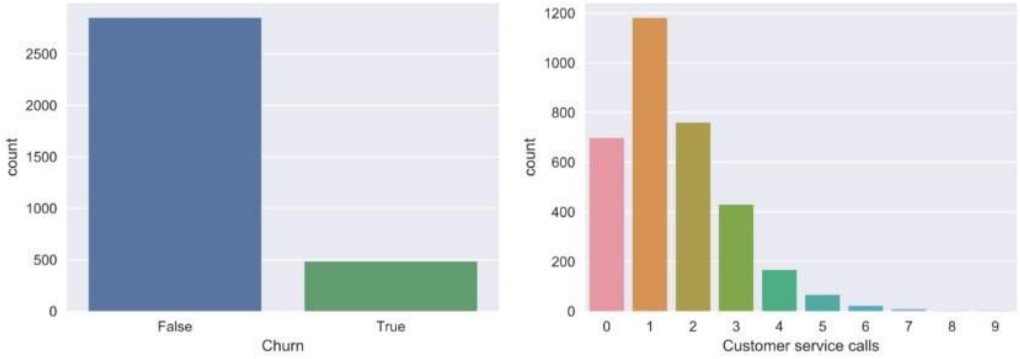
The bar plot is a graphical representation of the frequency . The easiest way to create it is to use the seaborn's function countplot(). There is another function in seaborn that is somewhat confusingly called barplot() and is mostly used for representation of some basic statistics of a numerical variable grouped by a categorical feature.

Let's plot the distributions for two categorical variables:

```
_, axes = plt.subplots(nrows=1, ncols=2, figsize=(12, 4))

sns.countplot(x='Churn', data=df, ax=axes[0]);
sns.countplot(x='Customer service calls', data=df, ax=axes[1]);
```

The output is as follows:



While the histograms, discussed above, and bar plots may look similar, there are several differences between them:

- Histograms are best suited for looking at the distribution of numerical variables while bar plots are used for categorical features.
- The values on the X-axis in the histogram are numerical; a bar plot can have any type of values on the X-axis: numbers, strings, booleans.
- The histogram's X-axis is a Cartesian coordinate axis along which values cannot be changed; the ordering of the bars is not predefined. Still, it is useful to note that the bars are often sorted by height, that is, the frequency of the values. Also, when we consider ordinal variables (like Customer service calls in our data), the bars are usually ordered by variable value.

The left chart above vividly illustrates the imbalance in our target variable. The bar plot for Customer service calls on the right gives a hint that the majority of customers

resolve their problems in maximum 23 calls. But, as we want to be able to predict the minority class, we may be more interested in how the fewer dissatisfied customers behave. It may well be that the tail of that bar plot contains most of our churn. These are just hypotheses for now, so let's move on to some more interesting and powerful visual techniques.