DSI-US-10 / 2.03-lesson-eda Private
forked from Data-science-immersive/2.05-lesson-eda

Branch: master 2.03-lesson-eda / solution-code / solution-code.ipynb Find file Copy path

Riley Dallas Fix typo and ISLR link 30b1766 17 days ago

0 contributors

1869 lines (1868 sloc) 178 KB

EDA Walkthrough

Authors: Kiefer Katovich (SF), David Yerrington (SF), Riley Dallas (AUS)

The dataset for today's lesson (`Heart.csv`) comes from the book, An Introduction to Statistical Learning (ISLR) (http://faculty.marshall.usc.edu/gareth-james/ISL/). It's comprised of diagnostic measurements for 303 patients to determine whether or not they have heart disease (the `AHD` column).

Though in many if not most cases the EDA procedure will be considerably more involved, this should give you an idea of the basic workflow a data scientist would use when working with a new dataset.

Learning Objectives

- Quickly describe a dataset, including data types, missing values and basic descriptive statistics
- Rename columns (series) in a DataFrame
- Visualize data distributions with box plots
- Calculate and visualize correlation

In [1]:
```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

%matplotlib inline
```

Load the data

Import the CSV into a pandas DataFrame.

In [2]:
```
file_path = '../datasets/Heart.csv'
```

In [3]:
```
# A:
df = pd.read_csv(file_path)
```

Describe the basic format of the data and the columns

Use the `.head()` method (and optionally pass in an integer for the number of rows you want to see) to get a glimpse of your dataset. This is a good initial step to get a feel for what is in the CSV and what problems may be present.

The `.dtypes` attribute tells you the data type for each of your columns.

In [4]:
```
# Print out the first 8 rows:
df.head(8)
```

Out[4]:

	Unnamed: 0	Age	Sex	ChestPain	RestBP	Chol	Fbs	RestECG	MaxHR	ExAng	Oldpeak	Slope	Ca	Thal	AHD
0	1	63	1	typical	145	233	1	2	150	0	2.3	3	0.0	fixed	0
1	2	67	1	asymptomatic	160	286	0	2	108	1	1.5	2	3.0	normal	1
2	3	67	1	asymptomatic	120	229	0	2	129	1	2.6	2	2.0	reversable	1
3	4	37	1	nonanginal	130	250	0	0	187	0	3.5	3	0.0	normal	0
4	5	41	0	nontypical	130	204	0	2	172	0	1.4	1	0.0	normal	0
5	6	56	1	nontypical	120	236	0	0	178	0	0.8	1	0.0	normal	0
6	7	62	0	asymptomatic	140	268	0	2	160	0	3.6	3	2.0	normal	1
7	8	57	0	asymptomatic	120	354	0	0	163	1	0.6	1	0.0	normal	0

```
In [5]: # Look at the dtypes of the columns:
        df.dtypes
```

```
Out[5]: Unnamed: 0      int64
        Age            object
        Sex             int64
        ChestPain      object
        RestBP          int64
        Chol            int64
        Fbs             int64
        RestECG         int64
        MaxHR           int64
        ExAng           int64
        Oldpeak       float64
        Slope           int64
        Ca            float64
        Thal           object
        AHD             int64
        dtype: object
```

Drop unwanted columns

It looks like `Unnamed: 0` is an index. This is redundant, since `pandas` automatically creates an index for us (the bold numbers to the left of the DataFrame).

The `.drop()` method can be used to get rid of a column like so:

```
df.drop(columns=['list', 'columns', 'to', 'drop'], inplace=True)
```

The `inplace=True` parameter makes our change permanent.

```
In [6]: # print out the index object and the first 20 items in the DataFrame's index
        # to see that we have these row numbers already:
        df['Unnamed: 0'][:20]
```

```
Out[6]: 0      1
        1      2
        2      3
        3      4
        4      5
        5      6
        6      7
        7      8
        8      9
        9     10
        10    11
        11    12
        12    13
        13    14
        14    15
        15    16
        16    17
        17    18
        18    19
        19    20
        Name: Unnamed: 0, dtype: int64
```

```
In [7]: # Remove the unneccesary column:
        df.drop(columns=['Unnamed: 0'], inplace=True)
        df.head()
```

Out[7]:

	Age	Sex	ChestPain	RestBP	Chol	Fbs	RestECG	MaxHR	ExAng	Oldpeak	Slope	Ca	Thal	AHD
0	63	1	typical	145	233	1	2	150	0	2.3	3	0.0	fixed	0
1	67	1	asymptomatic	160	286	0	2	108	1	1.5	2	3.0	normal	1
2	67	1	asymptomatic	120	229	0	2	129	1	2.6	2	2.0	reversable	1
3	37	1	nonanginal	130	250	0	0	187	0	3.5	3	0.0	normal	0
4	41	0	nontypical	130	204	0	2	172	0	1.4	1	0.0	normal	0

Clean corrupted column

From the previous step, we noticed the `Age` column was interpreted as a string, even though the values are integers.

It is pretty common to have numeric columns represented as strings in your data if some of the observations are corrupted. It is important to always check the data types of your columns.

What is causing the `Age` column to be encoded as a string?

```
In [8]: df.Age.sort_values(ascending=False)
```

```
Out[8]: 198      ?
        274      ?
        26       ?
        121      ?
        207      ?
        92       ?
        249      ?
        214      ?
        62       ?
        161     77
        257     76
        233     74
        273     71
        103     71
        42      71
        170     70
        136     70
        155     70
        258     70
        189     69
        196     69
        30      69
        159     68
        299     68
        83      68
        194     68
        2       67
        152     67
        71      67
        195     67
                ..
        186     42
        182     42
        220     41
        212     41
        241     41
        147     41
        4       41
        115     41
        295     41
        57      41
        50      41
        240     41
        268     40
        41      40
        29      40
        277     39
        109     39
        82      39
        222     39
        211     38
        302     38
        210     37
        3       37
        168     35
        138     35
        117     35
        283     35
        225     34
        101     34
        132     29
        Name: Age, Length: 303, dtype: object
```

In the cell below, replace all "?" cells with `np.nan`.

```
In [9]: df.Age = df.Age.map(lambda age: np.nan if age == '?' else int(age))
        df.dtypes
```

```
Out[9]: Age          float64
        Sex            int64
        ChestPain     object
        RestBP         int64
        Chol           int64
        Fbs            int64
        RestECG        int64
        MaxHR          int64
        ExAng          int64
        Oldpeak      float64
        Slope          int64
        Ca           float64
        Thal          object
        AHD            int64
        dtype: object
```

Determine how many observations are missing

Having replaced the question marks with `np.nan` values, we know that there are some missing observations for the `Age` column.

When we start to build models with data, null values in observations are (almost) never allowed. It is important to always see how many observations are missing for each column.

We can count the null values for each column like so:

```
df.isnull().sum()
```

The `.isnull()` method will convert the columns to `True` and `False` values.

The `.sum()` method will then sum these boolean columns, and the total number of null values per column will be returned.

```
In [10]: df.isnull().sum()
```

```
Out[10]: Age          9
         Sex          0
         ChestPain    0
         RestBP       0
         Chol         0
         Fbs          0
         RestECG      0
         MaxHR        0
         ExAng        0
         Oldpeak      0
         Slope        0
         Ca           4
         Thal         2
         AHD          0
         dtype: int64
```

Drop the null values.

In this case, lets keep it simple and just drop the rows from the dataset that contain null values. If a column has a ton of null values it often makes more sense to drop the column entirely instead of the rows with null values.

The `.dropna()` function will drop any rows that have ***ANY*** null values for you. Use this carefully as you could drop many more rows than expected.

```
In [11]: df.dropna(inplace=True)
```

```
In [12]: df.shape
```

```
Out[12]: (288, 14)
```

Make the column names more descriptive

One minor annoyance is that our column names are not at all intuitive.

Let's rename them!

There are two popular methods to renaming columns.

1. Using a *dictionary substitution*, which is very useful if you only want to rename a few columns.
2. Using a *list replacement*, which is quicker than writing out a dictionary, but requires a full list of names.

We'll explore both options in the cells below.

In [13]:
```
# Dictionary Method
new_columns_dict = {
    'Age': 'age',
    'Sex': 'sex_male',
    'ChestPain': 'chest_pain',
    'RestBP': 'resting_blood_pressure',
    'Chol': 'cholesterol',
    'Fbs': 'fasting_blood_sugar',
    'RestECG': 'resting_ecg',
    'MaxHR': 'max_heart_rate',
    'ExAng': 'exercise_induced_angina',
    'Oldpeak': 'old_peak',
    'Slope': 'slope',
    'Ca': 'ca',
    'Thal': 'thallium_stress_test',
    'AHD': 'has_heart_disease',
}

df.rename(columns=new_columns_dict, inplace=True)
```

In [14]:
```
df.head()
```

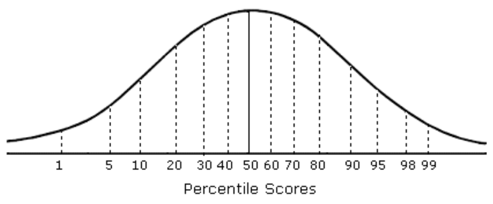
Out[14]:

	age	sex	chest_pain	resting_blood_pressure	cholesterol	fasting_blood_sugar	resting_ecg	max_heart_rate	exercis
0	63.0	1	typical	145	233	1	2	150	0
1	67.0	1	asymptomatic	160	286	0	2	108	1
2	67.0	1	asymptomatic	120	229	0	2	129	1
3	37.0	1	nonanginal	130	250	0	0	187	0
4	41.0	0	nontypical	130	204	0	2	172	0

In [ ]:
```
# List Replacement Method
new_columns_list = [
    'age',
    'sex_male',
    'chest_pain',
    'resting_blood_pressure',
    'cholesterol',
    'fasting_blood_sugar',
    'resting_ecg',
    'max_heart_rate',
    'exercise_induced_angina',
    'old_peak',
    'slope',
    'ca',
    'thallium_stress_test',
    'has_heart_disease',
]

# df.columns = new_columns_list
```

Describe the summary statistics for the columns



The `.describe()` function gives summary statistics for each of your columns. What are some, if any, oddities you notice about the columns based on this output?

based on this output?

In [15]:
```
df.describe()
```

Out[15]:

	age	sex	resting_blood_pressure	cholesterol	fasting_blood_sugar	resting_ecg	max_heart_rate	exe
count	288.000000	288.000000	288.000000	288.000000	288.000000	288.00000	288.000000	288
mean	54.461806	0.677083	131.760417	247.163194	0.145833	1.00000	149.576389	0.3
std	9.138039	0.468405	17.924393	51.425510	0.353553	0.99476	23.152601	0.4
min	29.000000	0.000000	94.000000	126.000000	0.000000	0.00000	71.000000	0.0
25%	47.000000	0.000000	120.000000	211.750000	0.000000	0.00000	133.000000	0.0
50%	56.000000	1.000000	130.000000	243.000000	0.000000	1.00000	152.500000	0.0
75%	61.000000	1.000000	140.000000	276.250000	0.000000	2.00000	166.000000	1.0
max	77.000000	1.000000	200.000000	564.000000	1.000000	2.00000	202.000000	1.0

You can also use `.groupby() + .describe()` for cohort analysis

In [17]:
```
df.groupby('has_heart_disease').mean().T
```

Out[17]:

has_heart_disease	0	1
age	52.506410	56.772727
sex	0.557692	0.818182
resting_blood_pressure	129.294872	134.674242
cholesterol	243.179487	251.871212
fasting_blood_sugar	0.141026	0.151515
resting_ecg	0.852564	1.174242
max_heart_rate	158.673077	138.825758
exercise_induced_angina	0.147436	0.545455
old_peak	0.595513	1.589394
slope	1.416667	1.833333
ca	0.262821	1.121212

Plot variables with potential outliers using boxplots.

Here we will use the seaborn package to plot boxplots of the variables we have identified as potentially having outliers.

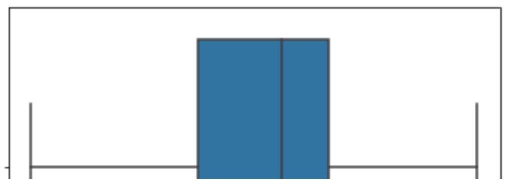
Some notes on seaborn's boxplot keyword argument options:

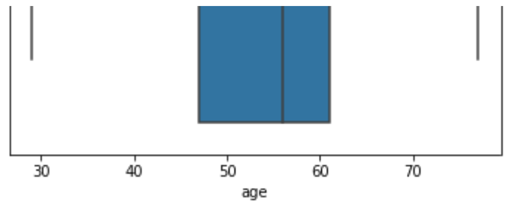
```
orient: can be 'v' or 'h' for vertical and horizontal, respectively
fliersize: the size of the outlier points (pixels I think)
linewidth: the width of line outlining the boxplot
notch: show the confidence interval for the median (calculated by seaborn/plt.boxplot)
saturation: saturate the colors to an extent
```

There are more keyword arguments available but those are most relevant for now.

If you want to check out more, place your cursor in the `boxplot` *argument bracket and press* `shift+tab` *(Press four times repeatedly to bring up detailed documentation).*

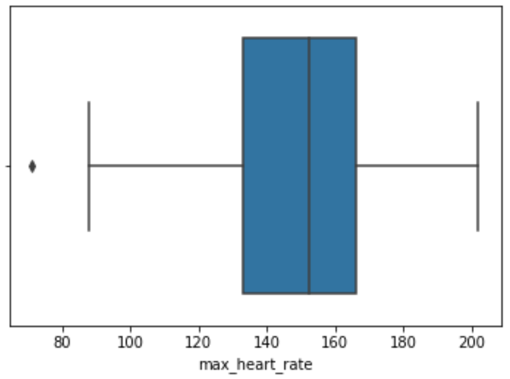
In [18]:
```
# age
sns.boxplot(df.age);
```





```
In [21]: # max heart rate
         sns.boxplot(df.max_heart_rate)
```

```
Out[21]: <matplotlib.axes._subplots.AxesSubplot at 0x1a1ce88b38>
```



Correlation matrices

A great way to easily get a feel for linear relationships between your variables is with a correlation matrix.

Below is the formula for the correlation between two variables X and Y:

Correlation

$$\text{pearson correlation } r = cor(X, Y) = \frac{cov(X, Y)}{std(X)std(Y)}$$

The correlation matrix

We can see the correlation between all the numeric variables in our dataset by using the `.corr()` method.

It's useful to get a feel for which columns are correlated. The `.corr() method` can help you decide what is worth investigating further (though with a lot of variables, the matrix can be a bit overwhelming...)

```
In [22]: # A:
         df.corr()
```

Out[22]:

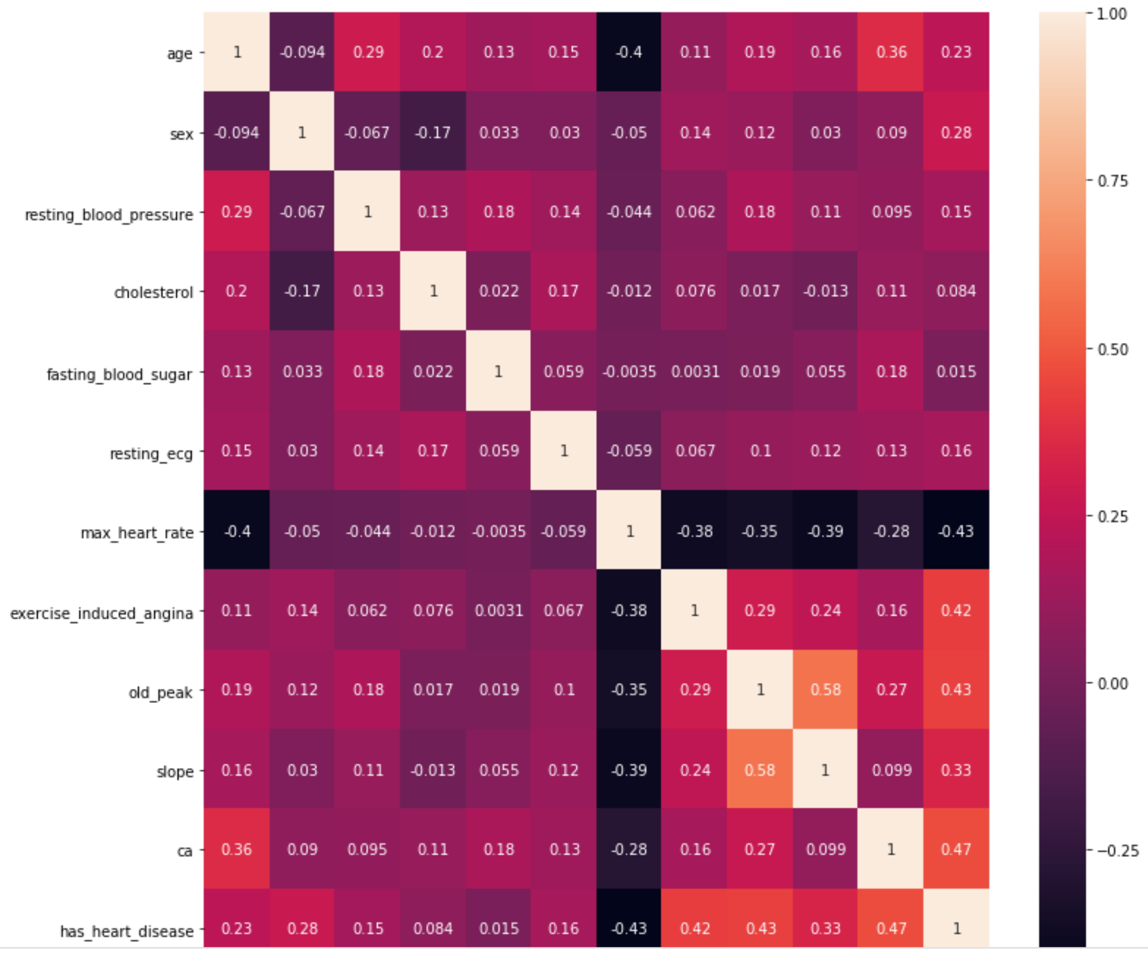
	age	sex	resting_blood_pressure	cholesterol	fasting_blood_sugar	resting_ecg	ma
age	1.000000	-0.094471	0.291517	0.200144	0.130069	0.150640	-0.3
sex	-0.094471	1.000000	-0.067347	-0.173120	0.032875	0.029912	-0.0
resting_blood_pressure	0.291517	-0.067347	1.000000	0.129044	0.184773	0.139721	-0.0
cholesterol	0.200144	-0.173120	0.129044	1.000000	0.022258	0.170143	-0.0
fasting_blood_sugar	0.130069	0.032875	0.184773	0.022258	1.000000	0.059442	-0.0
resting_ecg	0.150640	0.029912	0.139721	0.170143	0.059442	1.000000	-0.0
max_heart_rate	-0.399860	-0.049927	-0.043997	-0.012242	-0.003494	-0.058548	1.0
exercise_induced_angina	0.105350	0.137046	0.061811	0.075597	0.003052	0.066933	-0.3
old_peak	0.193581	0.119853	0.182143	0.016893	0.019111	0.104160	-0.3
slope	0.161608	0.030073	0.111736	-0.013046	0.055218	0.124097	-0.3
ca	0.358584	0.089553	0.094889	0.109374	0.176791	0.133791	-0.2

has_heart_disease	0.233030	0.277575	0.149796	0.084361	0.014809	0.161405	-0.4

It can be difficult to spot any outliers simply by staring at our correlation matrix. To help get around this issue, let's use Seaborn's `.heatmap()` to give our correlation matrix some color.

```
In [23]: # A:
         plt.figure(figsize=(12,12))
         sns.heatmap(df.corr(), annot=True)
```

```
Out[23]: <matplotlib.axes._subplots.AxesSubplot at 0x1a1d1c3320>
```



age	1	-0.094	0.29	0.2	0.13	0.15	-0.4	0.11	0.19	0.16	0.36	0.23
sex	-0.094	1	-0.067	-0.17	0.033	0.03	-0.05	0.14	0.12	0.03	0.09	0.28
resting_blood_pressure	0.29	-0.067	1	0.13	0.18	0.14	-0.044	0.062	0.18	0.11	0.095	0.15
cholesterol	0.2	-0.17	0.13	1	0.022	0.17	-0.012	0.076	0.017	-0.013	0.11	0.084
fasting_blood_sugar	0.13	0.033	0.18	0.022	1	0.059	-0.0035	0.0031	0.019	0.055	0.18	0.015
resting_ecg	0.15	0.03	0.14	0.17	0.059	1	-0.059	0.067	0.1	0.12	0.13	0.16
max_heart_rate	-0.4	-0.05	-0.044	-0.012	-0.0035	-0.059	1	-0.38	-0.35	-0.39	-0.28	-0.43
exercise_induced_angina	0.11	0.14	0.062	0.076	0.0031	0.067	-0.38	1	0.29	0.24	0.16	0.42
old_peak	0.19	0.12	0.18	0.017	0.019	0.1	-0.35	0.29	1	0.58	0.27	0.43
slope	0.16	0.03	0.11	-0.013	0.055	0.12	-0.39	0.24	0.58	1	0.099	0.33
ca	0.36	0.09	0.095	0.11	0.18	0.13	-0.28	0.16	0.27	0.099	1	0.47
has_heart_disease	0.23	0.28	0.15	0.084	0.015	0.16	-0.43	0.42	0.43	0.33	0.47	1