

medium.com

Exploratory Data Analysis — What is it and why is it so important? (Part 1/2)

Terence Shin

7–9 minutes

This is week 3 of my “52 Weeks of Data Science” series.



What is Exploratory Data Analysis?

Exploratory Data Analysis (EDA), also known as Data Exploration, is a step in the Data Analysis Process, where a number of techniques are used to better understand the dataset being used.

‘Understanding the dataset’ can refer to a number of things including but not limited to...

- Extracting important variables and leaving behind useless variables
- Identifying outliers, missing values, or human error
- Understanding the relationship(s), or lack of, between variables
- Ultimately, maximizing your insights of a dataset and minimizing potential error later in the process

Here’s why this is important.

Have you heard of the phrase, “garbage in, garbage out”?

With EDA, it’s more like, “garbage in, perform EDA, **possibly** garbage out.”

By conducting EDA, you can turn an **almost** useable dataset into a completely useable dataset. I’m not saying that EDA can magically make any dataset clean — that is not true. However, many EDA techniques can remedy some common problems that are present in every dataset.

Exploratory Data Analysis does two main things:

1. It helps clean up a dataset.
2. It gives you a better understanding of the variables and the

relationships between them.

Components of EDA

To me, there are main components of exploring data:

1. Understanding your variables
2. Cleaning your dataset
3. Analyzing relationships between variables

In this article, we'll take a look at the first two components.

1. Understanding Your Variables

You don't know what you don't know. And if you don't know what you don't know, then how are you supposed to know whether your insights make sense or not? You won't.

To give an example, I was exploring data provided by the NFL (data [here](#)) to see if I could discover any insights regarding variables that increase the likelihood of injury. One insight that I got was that Linebackers accumulated more than eight times as many injuries as Tight Ends. However, I had no idea what the difference between a Linebacker and a Tight End was, and because of this, I didn't know if my insights made sense or not. Sure, I can Google what the differences between the two are, but I won't always be able to rely on Google! Now you can see why understanding your data is so important. Let's see how we can do this in practice.

As an example, I used the same dataset that I used to create my first Random Forest model, the Used Car Dataset [here](#). First, I imported all of the libraries that I knew I'd need for my analysis and conducted some preliminary analyses.

```
#Import Libraries
```

```
import numpy as np
```

```
import pandas as pd
```

```
import matplotlib.pyplot as plt
```

```
import seaborn as sns#Understanding my variables
```

```
df.shape
```

```
df.head()
```

```
df.columns
```

.shape returns the number of rows by the number of columns for my dataset. My output was (525839, 22), meaning the dataset has 525839 rows and 22 columns.

.head() returns the first 5 rows of my dataset. This is useful if you want to see some example values for each variable.

.columns returns the name of all of your columns in the dataset.

```
Index(['url', 'city', 'city_url', 'price', 'year', 'manufacturer', 'make',  
      'condition', 'cylinders', 'fuel', 'odometer', 'title_status',  
      'transmission', 'VIN', 'drive', 'size', 'type', 'paint_color',  
      'image_url', 'desc', 'lat', 'long'],  
      dtype='object')
```

df.columns output

Once I knew all of the variables in the dataset, I wanted to get a better understanding of the different values for each variable.

```
df.nunique(axis=0)
```

```
df.describe().apply(lambda s: s.apply(lambda x: format(x, 'f')))
```

.nunique(axis=0) returns the number of unique values for each variable.

.describe() summarizes the count, mean, standard deviation, min, and max for numeric variables. The code that follows this simply formats each row to the regular format and suppresses scientific notation ([see here](#)).

```
url          525839  
city         112
```

```

city          413
city_url      413
price         18937
year          110
manufacturer   41
make          36969
condition      6
cylinders      8
fuel           5
odometer      121686
title_status   6
transmission   3
VIN           183526
drive          3
size           4
type           13
paint_color    12
image_url     369183
desc          444072
lat           69386
long          69537
dtype: int64

```

`df.nunique(axis=0)` output

	price	year	odometer	lat	long
count	525839.000000	524399.000000	427248.000000	513618.000000	513618.000000
mean	61966.045503	2009.375184	101150.248338	38.470853	-94.259216
std	9949703.964406	8.975889	105525.184435	5.895637	17.742010
min	0.000000	1900.000000	0.000000	-83.668334	-176.748047
25%	3900.000000	2006.000000	49009.750000	34.669400	-108.386684
50%	8999.000000	2011.000000	94240.000000	39.213214	-88.544050
75%	17900.000000	2015.000000	138000.000000	42.445565	-80.990754
max	3048344231.000000	2020.000000	10000000.000000	78.928357	132.078349

`df.describe().apply(lambda s: s.apply(lambda x: format(x, 'f')))`

output

Immediately, I noticed an issue with price, year, and odometer. For example, the minimum and maximum price are \$0.00 and \$3,048,344,231.00 respectively. You'll see how I dealt with this in the next section. I still wanted to get a better understanding of my discrete variables.

`df.condition.unique()`

Using `.unique()`, I took a look at my discrete variables, including 'condition'.

```

array(['good', nan, 'excellent', 'like new', 'fair', 'salvage', 'new'],
      dtype=object)

```

`df.condition.unique()`

You can see that there are many synonyms of each other, like 'excellent' and 'like new'. While this isn't the greatest example, there will be some instances where it's ideal to clump together different words. For example, if you were analyzing weather patterns, you may want to reclassify 'cloudy', 'grey', 'cloudy with a chance of rain', and 'mostly cloudy' simply as 'cloudy'.

Later you'll see that I end up omitting this column due to having too many null values, but if you wanted to re-classify the condition values, you could use the code below:

```
# Reclassify condition column
```

```
def clean_condition(row):  
    good = ['good', 'fair']  
    excellent = ['excellent', 'like new']  
  
    if row.condition in good:  
        return 'good'  
    if row.condition in excellent:  
        return 'excellent'  
    return row.condition
```

```
# Clean dataframe
```

```
def clean_df(playlist):  
    df_cleaned = df.copy()  
    df_cleaned['condition'] = df_cleaned.apply(lambda row:  
clean_condition(row), axis=1)  
    return df_cleaned# Get df with reclassified 'condition' column  
df_cleaned = clean_df(df)print(df_cleaned.condition.unique())
```

And you can see that the values have been re-classified below.

```
['good' nan 'excellent' 'salvage' 'new']
```

```
print(df_cleaned.condition.unique()) output
```

2. Cleaning your dataset

You now know how to reclassify discrete data if needed, but there are a number of things that still need to be looked at.

a. Removing Redundant variables

First I got rid of variables that I thought were redundant. This includes url, image_url, and city_url.

```
df_cleaned = df_cleaned.copy().drop(['url','image_url','city_url'],
axis=1)
```

b. Variable Selection

Next, I wanted to get rid of any columns that had too many null values. Thanks to my friend, Richie, I used the following code to remove any columns that had 40% or more of its data as null values. Depending on the situation, I may want to increase or decrease the threshold. The remaining columns are shown below.

```
NA_val = df_cleaned.isna().sum()def na_filter(na, threshold = .4):
#only select variables that passees the threshold
    col_pass = []
    for i in na.keys():
        if na[i]/df_cleaned.shape[0]<threshold:
            col_pass.append(i)
    return col_passdf_cleaned = df_cleaned[na_filter(NA_val)]
df_cleaned.columns
```

```
Index(['url', 'city', 'city_url', 'price', 'year', 'manufacturer', 'make',
      'fuel', 'odometer', 'title_status', 'transmission', 'drive', 'type',
      'paint_color', 'image_url', 'desc', 'lat', 'long'],
      dtype='object')
```

c. Removing Outliers

Revisiting the issue previously addressed, I set parameters for price, year, and odometer to remove any values outside of the set

boundaries. In this case, I used my intuition to determine parameters — I'm sure there are methods to determine the optimal boundaries, but I haven't looked into it yet!

```
df_cleaned = df_cleaned[df_cleaned['price'].between(999.99,
99999.00)]
df_cleaned = df_cleaned[df_cleaned['year'] > 1990]
df_cleaned = df_cleaned[df_cleaned['odometer'] <
899999.00]df_cleaned.describe().apply(lambda s: s.apply(lambda
x: format(x, 'f')))
```

You can see that the minimum and maximum values have changed in the results below.

	price	year	odometer	lat	long
count	371982.000000	371982.000000	371982.000000	365343.000000	365343.000000
mean	14689.685463	2010.738684	102015.760220	38.749623	-94.387297
std	11110.147204	5.409731	63704.691733	5.807578	17.836298
min	1000.000000	1991.000000	0.000000	-83.081983	-168.242626
25%	6100.000000	2007.000000	53000.000000	35.025913	-108.570506
50%	11995.000000	2012.000000	97690.000000	39.565110	-88.603600
75%	19995.000000	2015.000000	140276.000000	42.541378	-81.047065
max	99999.000000	2020.000000	899332.000000	71.525358	94.124800

d. Removing Rows with Null Values

Lastly, I used **.dropna(axis=0)** to remove any rows with null values. After the code below, I went from 371982 to 208765 rows.

```
df_cleaned = df_cleaned.dropna(axis=0)
df_cleaned.shape
```

And that's it for now! In the second part, we'll cover exploring the relationship between variables through visualizations. (Click [here](#) for part 2.)

You can see my Kaggle Notebook [here](#).