A Starter Pack to Exploratory Data Analysis with Python, pandas, seaborn, and scikit-learn

I'm a backpack loaded up with things and knickknacks too. Anything that you might need I've got inside for you. — Backpack from Dora the Explorer (source)

Ren Jie Tan Follow
Dec 23, 2018 · 14 min read

Exploratory Data Analysis (EDA) is the bread and butter of anyone who deals with data. With information increasing by 2.5 quintillions bytes per day (Forbes, 2018), the need for efficient EDA techniques is at its all-time high.

So where is this deluge coming from? The amount of useful information is almost certainly not increasing at such a rate. When we take a closer look, we would realize that most of this increase is **contributed by noise**. There are so many hypotheses to test, so

many datasets to mine, but a relatively constant amount of objective truth. With most data scientists, their key objective is to able to **distinguish the signal from the noise**, and EDA is the main process to do this.

Enter EDA

In this post, I shall introduce a Starter Pack to perform EDA on the Titanic dataset using popular Python packages: pandas, matplotlib, seaborn, and scikit-learn.

For code reference, you can refer to my GitHub repository here.

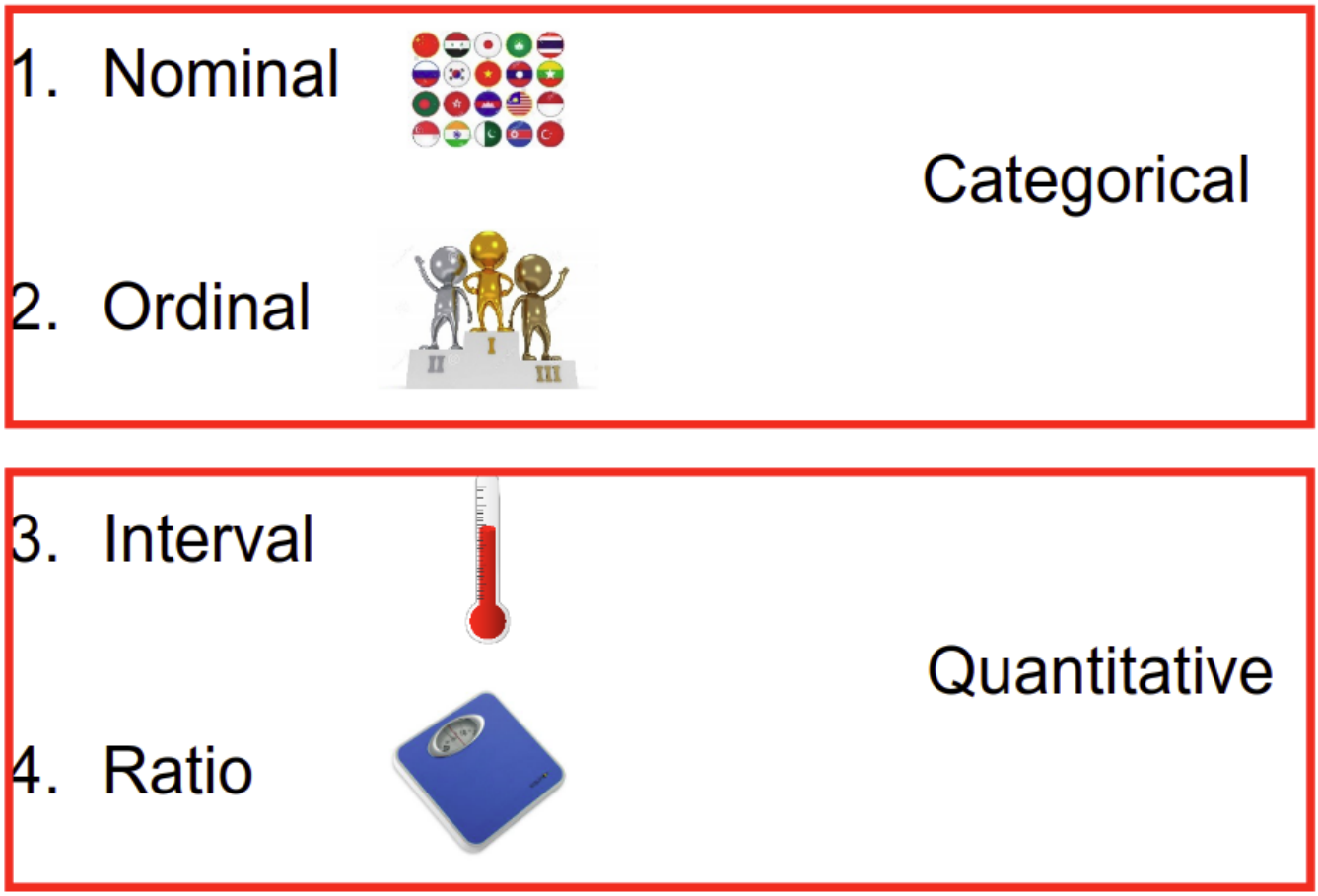
Outline:

. . .

1. What is Data

First and foremost, some theory. The word “data” was first used to mean “transmissible and storable computer information” in 1946 (source). At the highest level, the term data can be broadly categorized under two umbrellas: **structured** and **unstructured**. Structured data are pre-defined data models that normally reside in your relational database or data warehouse that has a fixed schema. Common examples include transaction information, customers’ information, and dates. On the other hand, unstructured data have no pre-defined data model and are found in NoSQL databases and data lakes. Examples include images, video files, and audio files.

In this post, we would **focus on structured data**, where I would propose a systemic approach to quickly show latent statistics from your data. Under the umbrella of Structured data, we can further categorize them to **categorical** and **quantitative**. For Categorical data, the rules of arithmetics do not apply. In the categorical family, we have **nominal** and **ordinal** data, while in the Quantitative family, we have **interval** and **ratio**. It is important that we take some time to clearly define and understand the subtle, yet significant differences each term is from the other as this would affect our analysis and preprocessing techniques later.



4 Different types of Data

Nominal data

The name “nominal” comes from the Latin word, *nomen*, which means name. Nominal data are objects which are differentiated by a **simple naming system**. An important thing to note is that nominal data may also have numbers assigned to them. This may appear ordinal (definition below), but they are not. Numbered nominal data are simply used to capture and reference. Some examples include:

- a set of countries.
- the number pinned on a marathon runner.

Ordinal data

Ordinal data are items in which their **order matters**. More formally, their relative positions on an ordinal scale provide meaning to us. This may indicate superiority or temporal positions, etc.. By default, the order of ordinal data is defined by assigning numbers to them. However, letters or other sequential symbols may also be used as appropriate. Some examples include:

- the competition ranking of a race (1st, 2nd, 3rd)
- the salary grade in an organization (Associate, AVP, VP, SVP).

Interval data

Similar to ordinal data, interval data is measured along a scale in which each object's position is **equidistant** from one another. This unique property allows arithmetic to be applied to them. An example is

- the temperature in degrees Fahrenheit where the difference between 78 degrees and 79 degrees is the same as 45 degrees and 46 degrees.

Ratio data

Like Interval data, the differences in Ratio data are meaningful. The Ratio data has an added feature which makes the ratios of the objects meaningful as well, and that is that they have a **true zero point**. Zero represents the absence of a certain property. So when we say something is of zero weight, we mean that thing has an absence of mass. Some examples include:

- the weight of a person on a weighing scale

Interval vs Ratio

The difference between interval and ratio is just one does not have a true zero point while the other does have. This is best illustrated with an example: When we say something is 0 degrees Fahrenheit, it does not mean an absence of heat in that thing.

This unique property makes statements that involve ratios such as “80 degrees Fahrenheit is twice as hot as 40 degrees Fahrenheit” not hold true.

Back to outline

. . .

Before we delve into the other sections, I would like to formalize some concepts so you would be clear on the thinking process to why we are doing things shown below.

Let me begin by saying that the best way to quickly show summaries of your data is through 2D plots. Despite living in a 3D spatial world, humans find it difficult to perceive the 3rd dimension, e.g. depth, needless to say, a projection of a 3D plot on a 2D screen. Hence, in the subsequent sections, you would see that we only use **bar graphs** for Categorical data, and **box plots** for Quantitative data as they succinctly express the data distributions respectively. We would only be focusing on **univariate analysis** and **bivariate analysis** with the target variable. For more details on Dashboarding Techniques, please refer to Section 6 of the article.

We would be mainly using **seaborn** and **pandas** to accomplish this. As we all know, statistics is an essential part of any data scientist’s toolkit and seaborn allows quick and easy use of matplotlib to beautifully visualize the statistics of your data. matplotlib is powerful, but it can get complicated at times. Seaborn provides a high-level abstraction of matplotlib allowing us to **plot attractive statistical plots with ease**. To make the best use of seaborn, we would also need pandas as **seaborn works best with pandas’ DataFrames**.

Also, if you want to follow along with the coding, be sure to download the data and set up your environment right. You can find the instructions in the README.MD file in my GitHub repo.

With these being said, let’s begin and get our hands dirty!

. . .

2. Categorical Analysis

We can start reading the data using `pd.read_csv()` . By doing a `.head()` on the data frame, we could have a quick peek at the top 5 rows of our data. For those who are not familiar with pandas or the concept of a data frame, I would highly recommend spending half a day going through the following resources:

1. Pandas documentation
2. What is a DataFrame?

Other useful methods are `.desribe()` and `.info()` where the former would show:

Output of .describe() method on a data frame

And the latter would show:

Output of .info() method on a data frame

We see now that,

Categorical data:

- PassengerId,
- Survived,
- Pclass,
- Name,
- Sex,
- Ticket,
- Cabin,
- and Embarked

while Qualitative data:

- Age,
- SibSp,
- Parch,
- and Fare

Now, with this knowledge and what we have learned in Section 1, let's write a custom helper function that can be used to handle most kinds of Categorical data (or at least attempt to) and give a quick summary of them. We shall do these with the help of some

pandas methods and seaborn's `.countplot()` method. The helper function is called `categorical_summarized` and is shown below.

```python
def categorical_summarized(dataframe, x=None, y=None, hue=None, palette='Set1', verbose=True):
    '''
    Helper function that gives a quick summary of a given column of categorical data

    Arguments
    =========
    dataframe: pandas dataframe
    x: str. horizontal axis to plot the labels of categorical data, y would be the count
    y: str. vertical axis to plot the labels of categorical data, x would be the count
    hue: str. if you want to compare it another variable (usually the target variable)
    palette: array-like. Colour of the plot

    Returns
    =======
    Quick Stats of the data and also the count plot
    '''
    if x == None:
        column_interested = y
    else:
        column_interested = x
    series = dataframe[column_interested]
    print(series.describe())
    print('mode: ', series.mode())
    if verbose:
        print('='*80)
        print(series.value_counts())

    sns.countplot(x=x, y=y, hue=hue, data=dataframe, palette=palette)
    plt.show()
```

categorical_summarized.py hosted with ♡ by GitHub view raw

Helper Function #1: Categorical Data Summarizer

What `categorical_summarized` does is it takes in a data frame, together with some input arguments and outputs the following:

1. The count, mean, std, min, max, and quartiles for numerical data or the count, unique, top class, and frequency of the top class for non-numerical data.

2. Class frequencies of the interested column, if `verbose` is set to `True`
3. Bar graph of the count of each class of the interested column

Let's talk a little bit about the input arguments. `x` and `y` take in a `str` type which corresponds to the interested column that we want to investigate. Setting the name of the column to `x` would create a bar graph with the x-axis showing the different classes and their respective counts on the y-axis. Setting the name of the interested column to `y` would just flip the axes of the previous plot where the different classes would be on the y-axis with x-axis showing the count. By setting the `hue` to the target variable, which is Survived in this case, the function would show the dependency of the target variable w.r.t. the interested column. Some example codes to show the usage of `categorical_summarized` are shown below:

Some examples using `categorical_summarized`

Univariate Analysis

```
# Target Variable: Survival
c_palette = ['tab:blue', 'tab:orange']
categorical_summarized(train_df, y = 'Survived', palette=c_palette)
```

categogical_analysis_survival.py hosted with ♡ by GitHub view raw

Using categorical_summarized on Survival Variable

Would give the following:

Output of categorical_summarized on Survival Variable

Bivariate Analysis

```
# Feature Variable: Gender
categorical_summarized(train_df, y = 'Sex', hue='Survived', palette=c_palette)
```

categoical_analysis_gender.py hosted with ♡ by GitHub view raw

Using categorical_summarized on Gender Variable with hue set to Survived

Would give the following:

Output of categorical_summarized on Gender Variable with hue set to Survived

For more examples, please refer to the `Titanic.ipynb` in the GitHub repo.

Back to outline

3. Quantitative Analysis

Now, we could technically use a bar graph for Quantitative data but it would generally be quite messy (you can try using `categorical_summarized` on the `Age` column, and you would see a messy plot of thin bins). A neater way would be to use a box plot, which displays the distribution based on a five number summary: minimum, Q1, median, Q3, and maximum.

The next helper function, called `quantitative_summarized` , is defined below:

```
def quantitative_summarized(dataframe, x=None, y=None, hue=None, palette='Set1', ax=None, verbos
    '''
    Helper function that gives a quick summary of quantattive data

    Arguments
    =========
    dataframe: pandas dataframe
    x: str. horizontal axis to plot the labels of categorical data (usually the target variable)
    y: str. vertical axis to plot the quantitative data
    hue: str. if you want to compare it another categorical variable (usually the target variabl
    palette: array-like. Colour of the plot
    swarm: if swarm is set to True, a swarm plot would be overlayed

    Returns
    =======
    Quick Stats of the data and also the box plot of the distribution
    '''
    series = dataframe[y]
    print(series.describe())
    print('mode: ', series.mode())
    if verbose:
        print('='*80)
        print(series.value_counts())

```

```
    sns.boxplot(x=x, y=y, hue=hue, data=dataframe, palette=palette, ax=ax)

    if swarm:
        sns.swarmplot(x=x, y=y, hue=hue, data=dataframe,
                      palette=palette, ax=ax)

    plt.show()
```

quantitative_summarized.py hosted with ♡ by GitHub view raw

Helper Function #2: Quantitative Data Summarizer

Similar to `categorical_summarized` , `quantitative_summarized` takes in a data frame and some input arguments to output the latent statistics and also a box plot and swarm plot (if `swarm` was set to `true`).

`quantitative_summarized` can take in one Quantitative Variable and up to two Categorical Variables, where the Quantitative Variable has to be assigned to `y` and the other two Categorical Variables can be assigned to `x` and `hue` respectively. Some example codes to show the usage is shown below:

Some examples using `quantitative_summarized`

Univariate Analysis

```
# univariate analysis
quantitative_summarized(dataframe= train_df, y = 'Age', palette=c_palette, verbose=False, swarm=T
```

quantitative_analysis_age.py hosted with ♡ by GitHub view raw

Using quantitative_summarized on Age Variable

Would give the following:

Output of quantitative_summarized on Age Variable

Bivariate Analysis

```
# bivariate analysis with target variable
quantitative_summarized(dataframe= train_df, y = 'Age', x = 'Survived', palette=c_palette, verbos
```

quantitative_analysis_age_survived.py hosted with ♡ by GitHub view raw

Using quantitative_summarized on Age Variable with x set to Survived

Would give the following:

Output of quantitative_summarized on Age Variable with x set to Survived

Multivariate Analysis

```
1  # multivariate analysis with Embarked variable and Pclass variable
2  quantitative_summarized(dataframe= train_df, y = 'Age', x = 'Embarked', hue = 'Pclass', palette=c
```

quantitative_analysis_age_embarked_pclass.py hosted with ♡ by GitHub view raw

Using quantitative_summarized on Age Variable with x set to Survived and hue set to Pclass

Would give the following:

Output of quantitative_summarized on Age Variable with x set to Survived and hue set to Pclass

Correlation Analysis

Another popular quantitative analysis we could use is to find the correlation between our variables. Correlation is a way to understand the relationship between the variables in our dataset. For any pairs of variables we have:

- **Positive Correlation**: both variables change in the same direction
- **Neutral Correlation**: No relationship in the change of the variables
- **Negative Correlation**: variables change in opposite directions

The Pearson's correlation coefficient is commonly used to summarize the strength of the linear relationship between two sample variables. It is the normalization of the covariance between the two variables which is given as,

Pearson's Correlation Coefficient Formula

A good way to visualize this is with a heatmap. We first drop the categorical variables and fill up the missing values of the remaining quantitative variables with their mode. Pandas DataFrames allow for an easy calculation of Pearson's CC with a `.corr` method. The code can be found below

```
plt.figure(figsize=(14,12))
plt.title('Pearson Correlation of Features', size = 15)
colormap = sns.diverging_palette(10, 220, as_cmap = True)
sns.heatmap(corr_df.corr(),
            cmap = colormap,
            square = True,
            annot = True,
            linewidths=0.1,vmax=1.0, linecolor='white',
            annot_kws={'fontsize':12 })
plt.show()
```

pearson_cc_heatmap.py hosted with ♡ by GitHub view raw

Heatmap of Pearson's CC for Quantitative Variables

One thing useful about having this visualization is that we can not only see the relationships between variables, for example, there is a negative Pearson's CC between Fare and Pclass, we could avoid the phenomenon of **multicollinearity** from happening.

Multicollinearity can adversely affect generalized linear models, such as logistic regression. It occurs when there are high correlations between variables, resulting in unstable estimates of regression coefficients. However, if you are planning to use a Decision Tree for your predictor, then you need not need to worry about this.

An interesting read about collinearity can be found here.

Correlation can help with data imputation, where correlated variables can be used to fill in the miss values of each other. Correlation can also indicate the presence of a causal

relationship. With that being said, it is important to note that correlation does not imply causation.

For more examples, please refer to the `Titanic.ipynb` in the GitHub repo.

Back to outline

4. Clustering

k-Means Clustering

k-means clustering belongs to the family of **partitioning clustering**. In Partition clustering, we have to specify the number of clusters, *k*, that we want. This can be done by picking out the "elbow" point of an inertia plot shown below. **A good choice of *k*, is one that is not too large and has a low inertia value**. The inertia is a metric to measure the cluster quality by giving us an indicator of how tightly packed a cluster is.

Inertia plot using K Means Clustering to select a suitable k value

As K Means calculates the distance between the features to decide if the following observation belongs to a certain centroid, we have to preprocess our data by encoding our Categorical Variables and filling up missing values. A simple function to preprocess is shown below.

```
def simple_preprocessing(dataframe, train=True):
    le = LabelEncoder()
    X = dataframe.drop(['PassengerId', 'Cabin', 'Name', 'Ticket'], axis=1)
```

```
    X['Age'] = X['Age'].fillna(value=X['Age'].mode()[0])
    X['Embarked'] = le.fit_transform(X['Embarked'].fillna(value=X['Embarked'].mode()[0]))
    X['Sex'] = np.where(X['Sex'] == 'male', 1, 0)

    if train:
        X = X.drop(['Survived'], axis=1)
        y = np.where(dataframe['Survived'] == 1, 'Alive', 'Dead')
        y = pd.get_dummies(y, columns=['Survived'])
        return X, y
    else:
        return X
```

simple_preprocessing_titanic.py hosted with ♡ by GitHub view raw

Function to preprocess the data to prepare for Clustering

Now that we have treated our data, we have to perform **feature scaling** so that the distance calculated across the features can be compared. This is done easily via `sklearn.preprocessing` library. For code implementation, please refer to the `Titanic.ipynb` . After running the k-means algorithm, where we set $k = 2$, we can plot the variables as shown below.

Cluster red and blue plotted on Age and Survived Axes

Hierarchical Agglomerative Clustering

For this subsection, I would present another quick way to perform EDA via clustering. Agglomerative clustering uses a bottom-up approach where individual observations are joined together iteratively based on their distance. We would be using the `scipy.cluster.hierarchy` package to perform the linkages and show our results using a dendrogram. The distance between two clusters is calculated via the **nearest neighbor method**.

```
from scipy.cluster.hierarchy import linkage
from scipy.cluster.hierarchy import dendrogram
sample_train,sample_val, gt_train, gt_val = train_test_split(train_df,
                                                             train_df['Survived'],
                                                             test_size=0.05,
                                                             random_state=99)

sample_val_processed = simple_preprocessing(sample_val, train = False)
sample_val_processed = scaler.fit_transform(sample_val_processed)
mergings = linkage(sample_val_processed, method='complete')
fig = plt.figure(figsize = (16,10))

dendrogram(mergings,
           labels=np.array(sample_val['Name']),
           leaf_rotation=90,
           leaf_font_size=10)
plt.show()
```

hierarchical_clustering_titanic.py hosted with ♡ by GitHub view raw

Code to implement Hierarchical Agglomerative Clustering

Dendrogram of the Hierarchical Clustering of the Titanic Dataset

Back to outline

5. Feature Importance by Tree-based estimators

Another quick way to perform EDA is via tree-based estimators. A decision tree learns how to "best" split the dataset into smaller subsets before finally outputting the predicted leaf node. The split is commonly defined by an **impurity criterion** like **Gini** or **Information Gain Entropy**. Since this is a post on EDA and not decision trees, I shall not explain the maths behind them in detail, but I shall show you how you could use them to understand your features better. You may refer to this well-written article for more details.

Based on the impurity criterion, a tree can be built by greedily picking the features that contribute to the most information gain. To illustrate this, I shall use the `scikit-learn` library.

Build a Random Forest Classifier

We first build a random forest classifier. By default, the impurity criterion is set to Gini. Using the following code, we could see the corresponding feature importance of our Titanic dataset.

```
from sklearn.ensemble import RandomForestClassifier
rf_clf = RandomForestClassifier(n_estimators = 500, max_depth=12)
rf_clf.fit(X_train, y_train)
rf_y_pred = rf_clf.predict(X_val)

pd.Series(rf_clf.feature_importances_, index = X_train.columns).nlargest(12).plot(kind = 'barh',
                                                                                  figsize = (10, 10)
                                                                                 title = 'Feature im
```

rf_clf.py hosted with ♡ by GitHub view raw

Code to generate a Random Forest Classifier, train, and also plot the Feature Importance

Let's try XGBoost

Another way to create an ensemble of decision trees is via XGBoost, which is part of the family of gradient boosted framework. Using the following code, we could see which corresponding feature is important to our XGBoost. Similarly, by default, the impurity criterion is set to Gini.

```
from xgboost import XGBClassifier

xgb_clf = XGBClassifier(max_depth=12, learning_rate=1e-4,n_estimators=500)
xgb_clf.fit(X_train, np.argmax(np.array(y_train), axis = 1))
xgb_y_pred = xgb_clf.predict(X_val)

pd.Series(xgb_clf.feature_importances_, index = X_train.columns).nlargest(12).plot(kind = 'barh',
                                                                                   figsize = (10, 10)
                                                                                  title = 'Feature im
```

xgb_clf.py hosted with ♡ by GitHub view raw

Code to generate an XGBoost Classifier, train, and also plot the Feature Importance

Back to outline

6. Dashboarding Techniques

Unlike infographics, dashboards are created to objectively display important information in a clean, concise manner on a single screen with the purpose to inform and not misguide its readers. Very often, dashboards are a representation of the data that exploit our visual perception abilities in order to amplify cognition. The information they display is of high graphical excellence and is understood by all, requiring no supplementary information for interpretation.

To achieve Graphical Excellence, the following two key aspects have to be obeyed:

1. Maximize Data: Ink and Minimize Chartjunk
2. Have High Graphical Integrity

Data: Ink & Chartjunk

Data: Ink is defined as the ink used to represent data, while chartjunk is defined as the superfluous, decorative, or diverting ink. Examples of chartjunk are moiré vibration, grids, and the duck. To achieve this, we make use of our **pre-attentive attributes** and **Gestalt's Principles** to bring out patterns in visualizations.

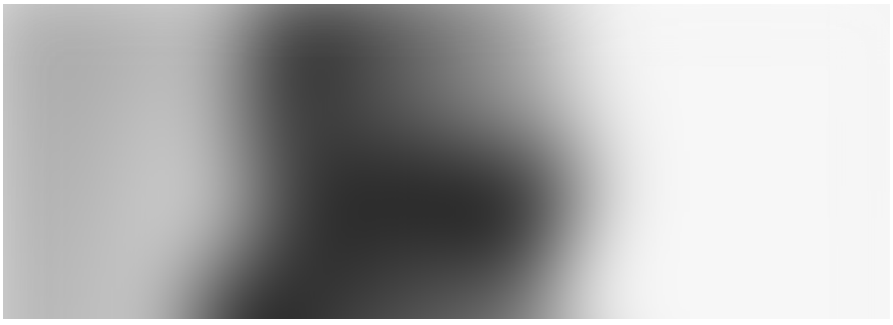


Image showing the side of a man's face, at the same time, the front of his face. An example to illustrate Figure & Ground

Achieving Graphical Integrity

Adapted from: Tufte, Edward. (2001). The Visual Display of Quantitative Information, 2nd Editions, Graphics, there are six principles to ensure Graphical Integrity:

1. Make the representation of numbers proportional to quantities
2. Use clear, detailed, and thorough labeling
3. Show data variation, not design variation
4. Use standardized units, not nominal values
5. Depict 'n' data dimensions with less than or equal to 'n' variable dimensions
6. Quote data in full context

As such, we avoid things like pie charts or putzing around with 3D graphs or with area dimensions. Bar graphs and box plots are such good examples to achieve graphical integrity as they are simple (everyone could understand without ambiguity) and powerful. It is also important to not miss out on context, like having the axis displaying quantitative data referenced to zero.

Examples of Misleading Graphics with Poor Graphical Integrity

Till date, Tableau is probably the industry leader when it comes to dashboarding. They take in best dashboarding practices and also do the heavy lifting of plotting the graphs for you just by dragging and dropping. I would highly recommend anyone who has an interest in this field to take a look at Tableau. Another up and coming open-source project, which does similar things to Tableau, is called Apache Superset.

Back to outline

. . .

Conclusion

All in all, there are numerous ways to do EDA. EDA is part of the entire data science process, which is highly iterative. What I have shared in this article are things which I have picked up and stayed with me after joining a number of data science hackathons, my time as a data scientist in AI Singapore, as well as my projects in my MTech course. I firmly believe in first establishing a firm foundation and then we optimize for speed and performance and hence, my motivation for writing this post. With this set of helper functions, my team and I were able to understand the latent statistics of our given dataset, and then develop a sound strategy to build a classifier within half a day of hacking.

This post is not meant to be a complete guide to EDA, but a Starter Pack to get aspiring Data Scientist kickoff their first data science projects using open-source tools like Python, pandas, seaborn, and scikit-learn. If you have any cool custom functions that are built upon Python packages and have aided you in your EDA process, please feel free to leave a comment below with your code. You could also submit a pull request and contribute to the `helper.py` file. Thanks for reading 😁. I hope you took away a thing or two.

. . .

Special mention to Raimi, Derek, and Chin Yang for proofreading and giving me feedback on improving this article

. . .

Feel free to connect with me via Twitter, LinkedIn!

. . .

If you are interested in the other project that I have worked on, feel free to visit my GitHub!

. . .

Want to learn how to build a Human Action Classifier quickly? Do check out my previous post over here.

)

Thanks to Raimi Karim.

Data Science | Exploratory Data Analysis | Pandas | Python | Scikit Learn