My top 4 functions to style the Pandas Dataframe

Why you need to stylish your pandas Dataframe and how to do it

Cornellius Yudha Wijaya

May 21 · 6 min read ★

Photo by Paweł Czerwiński on Unsplash

Pandas Dataframe is the most used object for Data scientists to analyze their data. While the main function is to just place your data and get on with the analysis, we could still style our data frame for many purposes; namely, for **presenting data** or **better aesthetic**.

Let's take an example with a dataset. I would use the 'planets' data available from seaborn for learning purposes.

```
#Importing the modules
import pandas as pd
import seaborn as sns

#Loading the dataset
planets = sns.load_dataset('planets')

#Showing 10 first row in the planets data
planets.head()
```

	method	number	orbital_period	mass	distance	year
0	Radial Velocity	1	269.300	7.10	77.40	2006
1	Radial Velocity	1	874.774	2.21	56.95	2008
2	Radial Velocity	1	763.000	2.60	19.84	2011
3	Radial Velocity	1	326.030	19.40	110.62	2007
4	Radial Velocity	1	516.220	10.50	119.47	2009
5	Radial Velocity	1	185.840	4.80	76.39	2008
6	Radial Velocity	1	1773.400	4.64	18.15	2002
7	Radial Velocity	1	798.500	NaN	21.41	1996
8	Radial Velocity	1	993.300	10.30	73.10	2008
9	Radial Velocity	2	452.800	1.99	74.79	2010

We could style our previous Pandas Data Frame just like below.

10 top data of Planets Dataset

method	number	orbital_period	mass	distance	year
Radial Velocity	1	269.3	7.1	77.4	2006
Radial Velocity	1	874.8	2.2	57.0	2008
Radial Velocity	1	763.0	2.6	19.8	2011
Radial Velocity	1	326.0	19.4	110.6	2007
Radial Velocity	1	516.2	10.5	119.5	2009
Radial Velocity	1	185.8	4.8	76.4	2008
Radial Velocity	1	1773.4	4.6	18.1	2002
Radial Velocity	1	798.5	nan	21.4	1996
Radial Velocity	1	993.3	10.3	73.1	2008
Radial Velocity	2	452.8	2.0	74.8	2010

It looks colorful right now because I put everything inside but I would break it down some of my favorite. Here are 4 functions to style our Pandas Data Frame object that I often use in everyday work.

> "While the main function is to just place your data and get on with the analysis, we could still style our data frame for many purposes; namely, for presenting data or better aesthetic."

. . .

1. Hiding Function

Sometimes when you do analysis and presenting the result to the other, you only want to show the most important aspect. I know when I present my Data Frame to the non-technical person, the question is often about the Index in their default number such as "what is this number?". For that reason, we could try to **hide the index** with the following code.

```
#Using hide_index() from the style function
```

```
planets.head(10).style.hide_index()
```

method	number	orbital_period	mass	distance	year
Radial Velocity	1	269.300000	7.100000	77.400000	2006
Radial Velocity	1	874.774000	2.210000	56.950000	2008
Radial Velocity	1	763.000000	2.600000	19.840000	2011
Radial Velocity	1	326.030000	19.400000	110.620000	2007
Radial Velocity	1	516.220000	10.500000	119.470000	2009
Radial Velocity	1	185.840000	4.800000	76.390000	2008
Radial Velocity	1	1773.400000	4.640000	18.150000	2002
Radial Velocity	1	798.500000	nan	21.410000	1996
Radial Velocity	1	993.300000	10.300000	73.100000	2008
Radial Velocity	2	452.800000	1.990000	74.790000	2010

Just like that, we hide our index. It is a simple thing but in the working environment I know sometimes it would become a problem. Just one extra column that becomes a question.

In addition, we could try to **hide unnecessary columns** with the chaining method. Let's say I don't want to show the 'method' and 'year' columns then we could write it with the following code.

```
#Using hide_columns to hide the unnecesary columns

planets.head(10).style.hide_index().hide_columns(['method','year'])
```

number	orbital_period	mass	distance
1	269.300000	7.100000	77.400000
1	874.774000	2.210000	56.950000
1	763.000000	2.600000	19.840000
1	326.030000	19.400000	110.620000
1	516.220000	10.500000	119.470000
1	185.840000	4.800000	76.390000

1	185.840000	4.800000	76.390000
1	1773.400000	4.640000	18.150000
1	798.500000	nan	21.410000
1	993.300000	10.300000	73.100000
2	452.800000	1.990000	74.790000

. . .

2. Highlight Function

There is a time when we want to present our data frame and only highlight the important number, for example the **highest number**. In this case, we could use the built-in method to highlight it with the following code.

```
#Highlight the maximum number for each column

planets.head(10).style.highlight_max(color = 'yellow')
```

	method	number	orbital_period	mass	distance	year
0	Radial Velocity	1	269.300000	7.100000	77.400000	2006
1	Radial Velocity	1	874.774000	2.210000	56.950000	2008
2	Radial Velocity	1	763.000000	2.600000	19.840000	2011
3	Radial Velocity	1	326.030000	19.400000	110.620000	2007
4	Radial Velocity	1	516.220000	10.500000	119.470000	2009
5	Radial Velocity	1	185.840000	4.800000	76.390000	2008
6	Radial Velocity	1	1773.400000	4.640000	18.150000	2002
7	Radial Velocity	1	798.500000	nan	21.410000	1996
8	Radial Velocity	1	993.300000	10.300000	73.100000	2008
9	Radial Velocity	2	452.800000	1.990000	74.790000	2010

In the data frame above, we highlight the maximum number in each column with the color yellow. If you want to highlight the **minimum number** instead, we could do it with the following code.

```
planets.head(10).style.highlight_min(color = 'lightblue')
```

	method	number	orbital_period	mass	distance	year
0	Radial Velocity	1	269.300000	7.100000	77.400000	2006
1	Radial Velocity	1	874.774000	2.210000	56.950000	2008
2	Radial Velocity	1	763.000000	2.600000	19.840000	2011
3	Radial Velocity	1	326.030000	19.400000	110.620000	2007
4	Radial Velocity	1	516.220000	10.500000	119.470000	2009
5	Radial Velocity	1	185.840000	4.800000	76.390000	2008
6	Radial Velocity	1	1773.400000	4.640000	18.150000	2002
7	Radial Velocity	1	798.500000	nan	21.410000	1996
8	Radial Velocity	1	993.300000	10.300000	73.100000	2008
9	Radial Velocity	2	452.800000	1.990000	74.790000	2010

and if you want to chain it, we could also do that.

```
#Highlight the minimum number with lightblue color and the maximum
number with yellow color

planets.head(10).style.highlight_max(color='yellow').highlight_min(c
olor = 'lightblue')
```

	method	number	orbital_period	mass	distance	year
0	Radial Velocity	1	269.300000	7.100000	77.400000	2006
1	Radial Velocity	1	874.774000	2.210000	56.950000	2008
2	Radial Velocity	1	763.000000	2.600000	19.840000	2011
3	Radial Velocity	1	326.030000	19.400000	110.620000	2007
4	Radial Velocity	1	516.220000	10.500000	119.470000	2009
5	Radial Velocity	1	185.840000	4.800000	76.390000	2008
6	Radial Velocity	1	1773.400000	4.640000	18.150000	2002
7	Radial Velocity	1	798.500000	nan	21.410000	1996

8	Radial Velocity	1	993.300000	10.300000	73.100000	2008
9	Radial Velocity	2	452.800000	1.990000	74.790000	2010

Instead of each column, you could actually highlight the minimum or maximum number for **each row**. I show it in the following code.

```
#Adding Axis = 1 to change the direction from column to row

planets.head(10).style.highlight_max(color = 'yellow', axis =1)
```

	method	number	orbital_period	mass	distance	year
0	Radial Velocity	1	269.300000	7.100000	77.400000	2006
1	Radial Velocity	1	874.774000	2.210000	56.950000	2008
2	Radial Velocity	1	763.000000	2.600000	19.840000	2011
3	Radial Velocity	1	326.030000	19.400000	110.620000	2007
4	Radial Velocity	1	516.220000	10.500000	119.470000	2009
5	Radial Velocity	1	185.840000	4.800000	76.390000	2008
6	Radial Velocity	1	1773.400000	4.640000	18.150000	2002
7	Radial Velocity	1	798.500000	nan	21.410000	1996
8	Radial Velocity	1	993.300000	10.300000	73.100000	2008
9	Radial Velocity	2	452.800000	1.990000	74.790000	2010

As we can see, it is useless right now to change the axis as it did not highlight any important information. It would be more useful in the case when each column is not that different from each other.

As an addition, we could highlight the **null value** with the following code.

```
#Higlight the null value

planets.head(10).style.highlight_null(null_color = 'red')
```

	method	number	orbital_period	mass	distance	year

0	Radial Velocity	1	269.300000	7.100000	77.400000	2006
1	Radial Velocity	1	874.774000	2.210000	56.950000	2008
2	Radial Velocity	1	763.000000	2.600000	19.840000	2011
3	Radial Velocity	1	326.030000	19.400000	110.620000	2007
4	Radial Velocity	1	516.220000	10.500000	119.470000	2009
5	Radial Velocity	1	185.840000	4.800000	76.390000	2008
6	Radial Velocity	1	1773.400000	4.640000	18.150000	2002
7	Radial Velocity	1	798.500000	nan	21.410000	1996
8	Radial Velocity	1	993.300000	10.300000	73.100000	2008
9	Radial Velocity	2	452.800000	1.990000	74.790000	2010

3. Gradient Function

While presenting your data, we could also use all the information as the main way to present the data. I often present the data with a background color to highlight which number is in the lower area and where is the one in the higher area. Let's use the example by the following code.

```
#Gradient background color for the numerical columns

planets.head(10).style.background_gradient(cmap = 'Blues')
```

With the background_gradient function, we could color the data frame as a gradient. The color would depend on the cmap parameter where the parameter is accepting colormaps from the matplotlib library.

We could also use a **bar chart** as our gradient background color. Let me show it in the example below.

```
#Sort the values by the year column then creating a bar chart as the background

planets.head(10).sort_values(by = 'year').style.bar(color= 'lightblue')
```

	method	number	orbital_period	mass	distance	year
7	Radial Velocity	1	798.500000	nan	21.410000	1996
6	Radial Velocity	1	1773.400000	4.640000	18.150000	2002
0	Radial Velocity	1	269.300000	7.100000	77.400000	2006
3	Radial Velocity	1	326.030000	19.400000	110.620000	2007
1	Radial Velocity	1	874.774000	2.210000	56.950000	2008
5	Radial Velocity	1	185.840000	4.800000	76.390000	2008
8	Radial Velocity	1	993.300000	10.300000	73.100000	2008
4	Radial Velocity	1	516.220000	10.500000	119.470000	2009
9	Radial Velocity	2	452.800000	1.990000	74.790000	2010
2	Radial Velocity	1	763.000000	2.600000	19.840000	2011

As we could see above, we now highlight the number from the lowest to the highest number in a different way than the background_gradient function is. We could also see the index is not in order because of the sort function; it is better to hide the index as I told you in the passage above.

4. Custom Function

If you prefer to have a more specific requirement to style your data frame, you could actually do it. We could pass our style functions into one of the following methods:

- `Styler.applymap`: element-wise
- `Styler.apply`: column-/row-/table-wise

Both of those methods take a function (and some other keyword arguments) and apply our function to the DataFrame in a certain way. Let's say that I have a threshold that any number below 20 should be colored red. We could do that by using the following code.

```
#Create the function to color the numerical value into red color

def color_below_20_red(value):
    if type(value) == type(''):
        return 'color:black'
    else:
        color = 'red' if value <= 20 else 'black'
        return 'color: {}'.format(color)

#We apply the function to all the element in the data frame by using
the applymap function

planets.head(10).style.applymap(color_below_20_red)
```

	method	number	orbital_period	mass	distance	year
0	Radial Velocity	1	269.300000	7.100000	77.400000	2006
1	Radial Velocity	1	874.774000	2.210000	56.950000	2008
2	Radial Velocity	1	763.000000	2.600000	19.840000	2011
3	Radial Velocity	1	326.030000	19.400000	110.620000	2007
4	Radial Velocity	1	516.220000	10.500000	119.470000	2009
5	Radial Velocity	1	185.840000	4.800000	76.390000	2008
6	Radial Velocity	1	1773.400000	4.640000	18.150000	2002
7	Radial Velocity	1	798.500000	nan	21.410000	1996
8	Radial Velocity	1	993.300000	10.300000	73.100000	2008
9	Radial Velocity	2	452.800000	1.990000	74.790000	2010

Just like that, every single number below or equal to 20 would be colored red and the rest is in black.

. . .

Conclusion

I have shown my top 4 functions to use when styling our data frame; hiding functions. We could style our data frame for presenting our data or just a better aesthetic. If you want to read more about what we can do to style our data frame, you could read it here.

Data Science | Python | Machine Learning | Programming | Education

9 pandas visualizations techniques for effective data analysis

Learn how to use line graphs, scatter plots, histograms, boxplots, and a few other visualization techniques using pandas library only

Magdalena Konkiewicz
May 14 · 7 min read ★

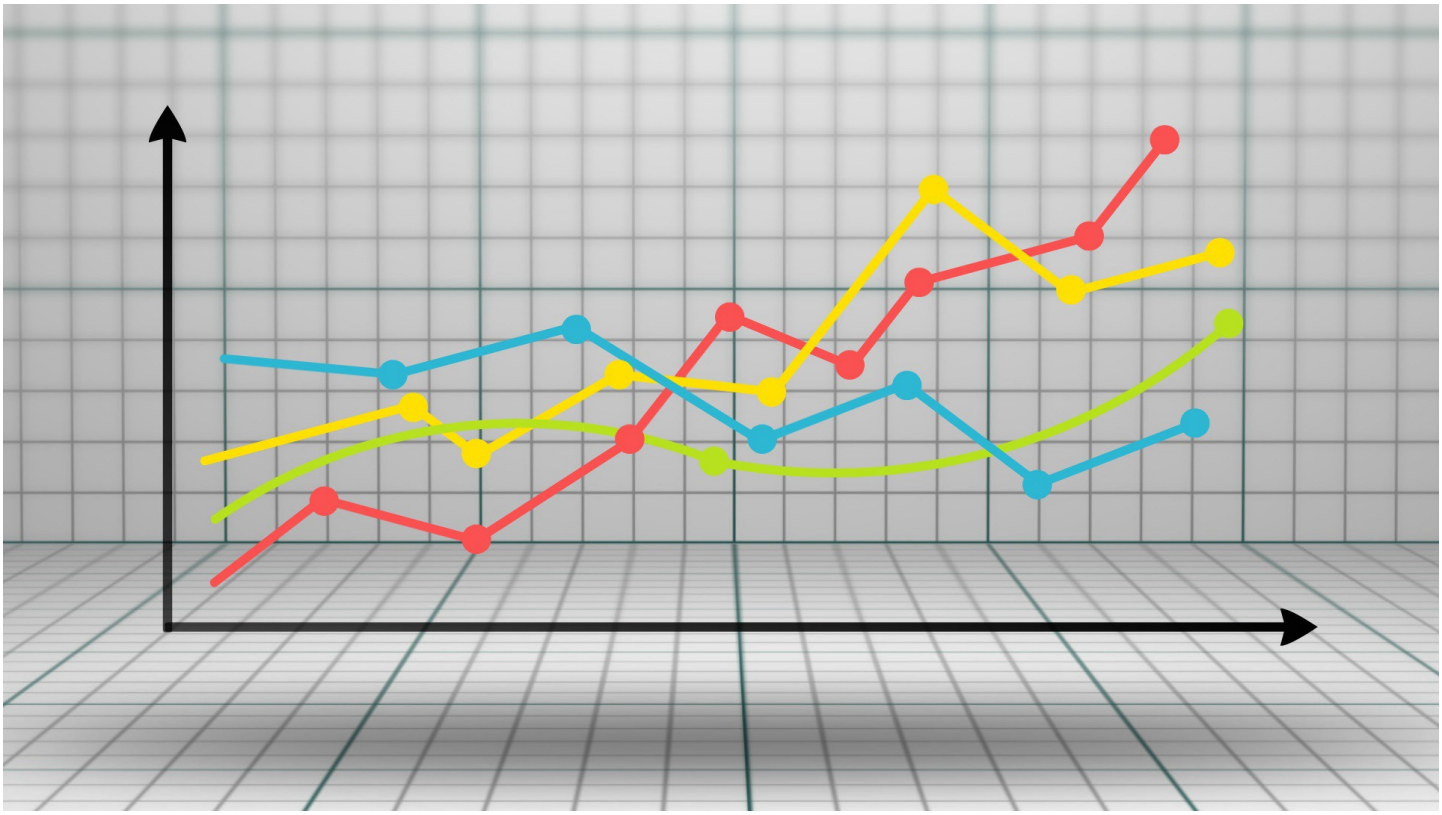


Image by Mediamodifier from Pixabay

Introduction

In this article, I would like to present you with nine different visualization techniques that will help you analyze any data set. Most of these techniques require just one line of code. We all love simplicity, don't we?

You will learn how to use:

line plot,

scatter plot,

area plot,

bar chart,

piechart,

histogram,

kernel density function,

boxplot,

and scatter matrix plot.

We will discuss all of the above visualization techniques, explore different ways of using them, and learn how to customize them to suit a data set.

Let's get started.

. . .

Load data set and quick data exploration

For simplicity purposes, we will use the Iris data set that can be loaded from a scikit-learn library using the following code:

```
from sklearn.datasets import load_iris
import pandas as pd
```

```
data = load_iris()
df = pd.DataFrame(data['data'], columns=data['feature_names'])
df['species'] = data['target']
df.head()
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

As you can see we have a data set with five columns only. Let's call info() function on the data frame for quick analysis:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
sepal length (cm)    150 non-null float64
sepal width (cm)     150 non-null float64
petal length (cm)    150 non-null float64
petal width (cm)     150 non-null float64
species              150 non-null int64
dtypes: float64(4), int64(1)
memory usage: 6.0 KB
```

As you can see there are only 150 entries, there are no missing values in any of the columns.

Additionally, we have learnt that the first four columns have float values whereas the last column allows integers only. In fact from the data set description we know that *species* column will take only three values each one representing one type of flower.

To confirm this you can call the unique() function on that column:

```
df.species.unique()
array([0, 1, 2])
```

Indeed *species* column take only three values: 0, 1, and 2.

Knowing this basic information about our data set we can proceed to visualizations. Note that if there were some missing values in the columns you should either drop them or fill them in. This is because some of the techniques we will discuss later on will not allow for missing values.

. . .

Line plot

We are going to start our visualizations with a simple line plot. Let's just call it on the whole data frame.

```
df.plot()
```


As you can see here it has plotted all the column values in different colours against the index value (x-axis). This is the default behaviour when we do not supply the x-axis parameter for the function.

As you can see this plot is not very useful. The line plot would be a good choice if the x-axis was a time series. Then we could probably see some trends in time within data.

In our case, we can only see that the data is ordered by *species* column (purple steps in the graph) and that some other columns have a moving average that follows that

pattern (petal length especially that is marked in red).

. . .

Scatter plot

The next type of visualisation we are going to discover is a scatter plot. This a perfect type of plot to visualise correlations between two continuous variables. Let's demonstrate it by plotting sepal length against sepal width.

```
df.plot.scatter(x='sepal length (cm)', y='sepal width (cm)')
```


As you can see in order to produce this graph you need to specify x and y-axis for the plot by supplying its column names. This graph reveals that there is no strong correlation between the two variables. Let's examine a different pair, *sepal length* and *petal length*:

```
df.plot.scatter(x='sepal length (cm)', y='petal length (cm)')
```

In this case, we can see that when sepal length increases petal length increases as well (it is stronger for values of sepal length larger than 6 cm).

. . .

Area plot

Let's create an area plot for the data frame. I will include all dimensions with centimetres in my plot but remove the *species* column as it will not make sense to include it in our case.

```
columns = ['sepal length (cm)', 'petal length (cm)', 'petal width
(cm)', 'sepal width (cm)']
df[columns].plot.area()
```

The measurements on this graph are stuck one on top of each other. Looking at this chart you can visually examine the ratio between each measurement that is included in the graph. You can see that all sizes have a growing trend towards the later entries.

. . .

Bar chart

This is the good type of a graph to include when showing averages or counts of entries. Let's use it to compute averages for each dimension for each species in our data set.

In order to do it, you will need to use a groupby() and mean() function. I am not going to explain how they work in detail here but you can check this article that explains these concepts.

```
df.groupby('species').mean().plot.bar()
```

As you can see this is very straight forward to read. I can see that there are differences in average measurements for different species and different columns.

. . .

Piechart

You can use a pie chart in order to visualize the class count for your target variable. We will do it here for the Iris data set we are working on. Again we will need some helper functions. This time it is groupby() and count():

```
df.groupby('species').count().plot.pie(y='sepal length (cm)')
```

As you can see we have perfect proportions for our classes as our data set consist of 50 entries for each class.

Note that we had to use *y* parameter here and set it to some column name. We have used *sepal length* column here but it could be any column as the counts are the same for all of them.

. . .

Histogram

This is a perfect visualization for any continuous variable. Let's start with simple hist() function.

```
import matplotlib.pyplot as plt
df.hist()
plt.tight_layout()
```

As you can see this produces a histogram for each numeric variable in the data set.

I had to add some extra lines of code in order to customize the graph. This is the first import line and the last line where I call tight_layout() function. If this is not added the labels and subgraph names may overlap and not be visible.

. . .

Kernel density function

In a similar way as a histogram you can use kernel density function:

```
df.plot.kde(subplots=True, figsize=(5,9))
```

You can see that it gives similar results to the histogram.

We had to specify a figure size here as without it the graphs were squashed vertically too much. Also, we had set subplots parameter to *True* as by default all columns would be displayed on the same graph.

. . .

Boxplot

Another visualisation that should be used for numerical variables. Let's create boxplots for all measurement columns (we are excluding *species* column as the box plot does not make sense for this categorical variable):

```
columns = ['sepal length (cm)', 'petal length (cm)', 'petal width
(cm)', 'sepal width (cm)']
df[columns].plot.box()
plt.xticks(rotation='vertical')
```

As you can see all boxplots are drawn on the same plot. This is fine in our case as we do not have too many variables to visualise.

Note that we had to rotate the x labels as without it the names of the labels were overlapping with each other.

. . .

Scatter matrix plot

This is one of my favourites visualisation technique from pandas as it allows you to do a quick analysis of all numerical values in the dataset and their correlations.

By default, it will produce scatterplots for all numeric pairs of variables and histograms for all numeric variables in the data frame:

```
from pandas.plotting import scatter_matrix
scatter_matrix(df, figsize=(10, 10))
```

As you can the results is this beautiful set of plots that indeed can tell you a lot about the data set using only one line of code. I can spot some correlations between variables in this data set just by glancing at it.

The only additional parameter I had to set was figure size and this is because the plots were very small with a default size of the chart.

. . .

Summary

I hope you have enjoyed this short tutorial about different pandas visualisations techniques and that you will be able to apply this knowledge to a data set of your choice.

Happy graphing!

. . .

PS: I am writing articles that explain basic Data Science concepts in a simple and comprehensible manner. If you liked this article there are some other ones you may enjoy:

Sorting data frames in pandas

How to sort data frames quickly and efficiently

towardsdatascience.com

7 practical pandas tips when you start working with the library

Explaining some things that are not so obvious at first glance...

towardsdatascience.com

Jupyter notebook autocompletion

The best productivity tool for Data Scientist you should be using if you are not doing it yet...

towardsdatascience.com

Artificial Intelligence | Data Science | Programming | Data Visualization | Machine Learning

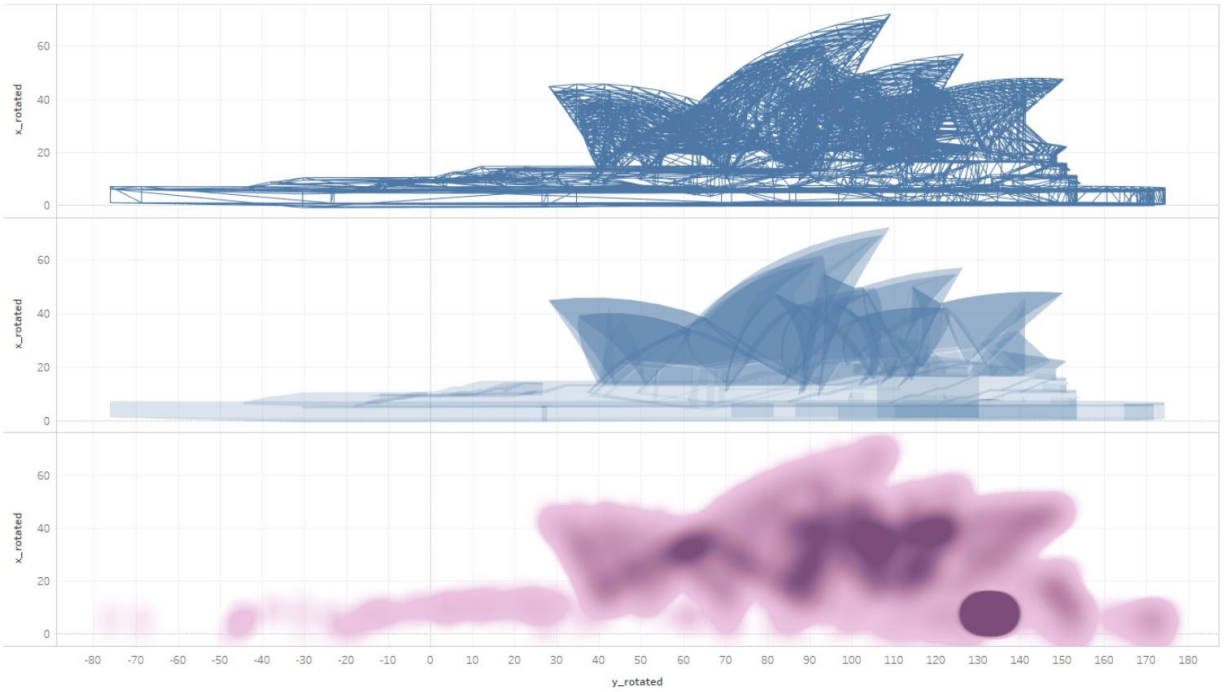
Wrestling with Tableau's Polygon Mapping

Darcy Vance Oct 18, 2019 · 8 min read ★

I could not, for the life of me, find a simple tutorial regarding this incredibly powerful feature of Tableau.

So I thought I'd try my hand at writing so that the next person wanting to learn this (or heaven forbid, the next time-constrained data viz consultant needing this information urgently), doesn't have to trawl through message boards and answer forums, and can get building sooner.

Tableau's Polygon chart type is an incredibly powerful tool for any data viz professional. Most of the insane dashboards for Makeover Monday and crazy, "Tableau-breaking" feature pieces (like this one by Frances Jones), use Polygon Mapping in some way.



3D “Wireframe” Path Model, 3D Polygon Model and Density Map of the Sydney Opera House — Credit: Frances Jones

But beyond fancy shapes and 3D monuments, Polygon mapping can also make the difference in professional visualisation development. Let’s have a look at a demo, and then we can look at some professional use cases.

The Demo

A bit of context:

I’ve always been a fan of professional wrestling. If Game of Thrones was a soap opera, told entirely through interpretive dance, and played out weekly over the span of decades, it would be closing in on WWE’s market share of the sport.

"The Heartbreak Kid" Shawn Michaels performing a high-risk manoeuvre on "Mankind" Mick Foley.

And I'm not using the word "sport" lightly. The performers that involve themselves in this amalgamation of sport and entertainment consistently place themselves in dangerous situations for the benefit of the live, and television, audience. The injuries that can result from this are as real as in any other sport.

I wanted to try and visualise the injuries sustained by some of these performers, as both a training exercise in Tableau, but also due to my own curiosity. This is how I did it:

Step 0: Understanding the data

First, we must understand what is required for polygons to work in Tableau.

Tableau's Polygon mapping requires “an ordered, enclosed sets of X, Y coordinates”. This is a fancy way of saying that, in order to draw a shape on a plane, we require:

- A number of X, Y coordinates
- An order that these X, Y coordinates occur
- And that the set needs to be completed (i.e. the last X, Y value should be the same as the first)

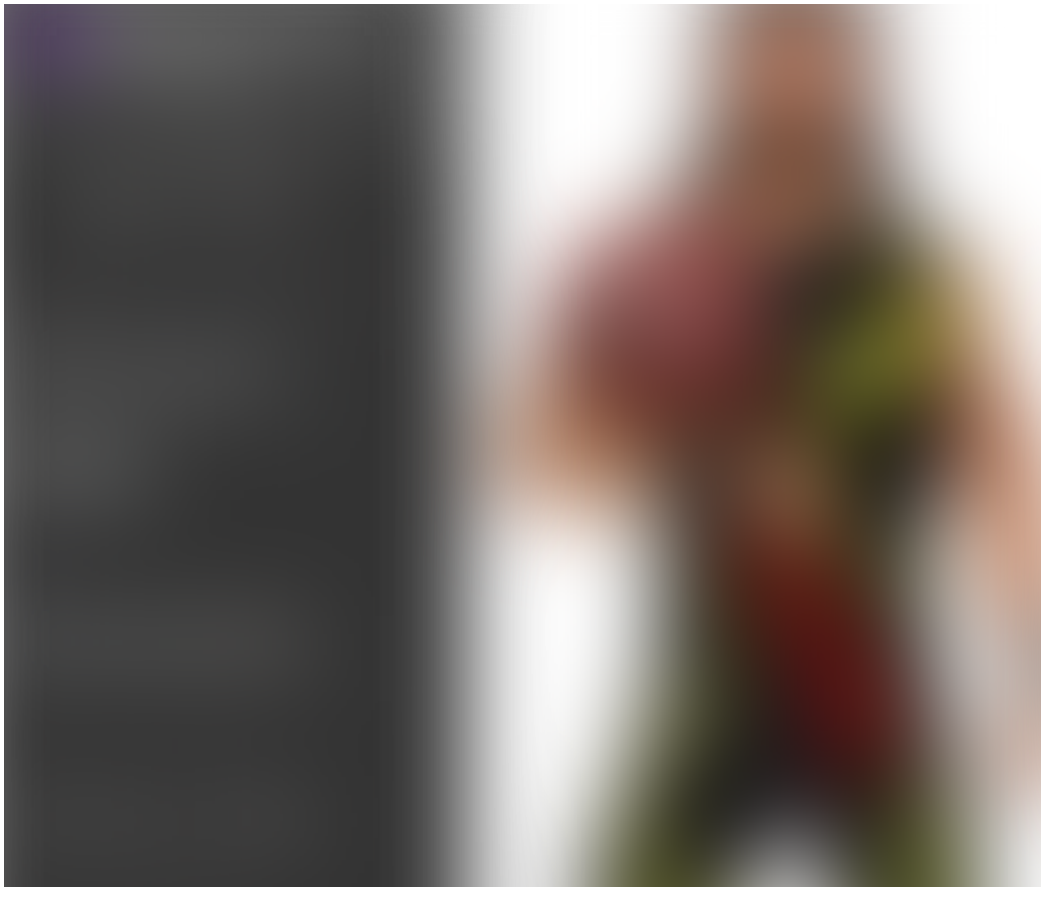
Example: Required data to create the above quadrilateral. Notice there are five values, as we have “completed” the shape by returning to the first X, Y position.

Now, this can be done in any way you like, and it’s relatively easy when creating basic shapes. But as soon as we want to replicate complex shapes, we require some help. This is where the *InterWorks Drawing Utility for Tableau* comes in.

This is a free, browser-based piece of software that allows you to plot points, paths and polygons on top of any image you like. For our example, to better understand the patterns of injuries for professional wrestlers, we are going to represent our data in the form of the human body.

Step 1: Creating our polygons

1. Load your selected image into the InterWorks Drawing Utility for Tableau. (I am using current Universal Champion Seth Rollins as my polygon model, but you can use any image you like.)
2. Under Drawing Options, select Polygons and check “Snap to Existing Points” and “Show Guide Line”. (If mapping mainly straight, rigid objects, “Align with Previous Points” will allow perfect 90° angles.)



3. Start drawing polygons. This is as simple as clicking around the areas of the image that you wish to map. One of the body parts listed in my injury statistics was “Elbow and Forearm”. As you begin your path, you will see the data being generated in the table (or as text, if you’ve selected “CSV”) below Point Data.

Once you finish a polygon, you will notice that the tool fills in the space to make it clear what has been mapped so far. If you have selected “Snap to Existing Points”, you can also use the existing points for new polygons to ensure all edges match correctly.

Once a polygon is completed, you can also click and drag points to change their location. This massively reduces time in fixing up intricate details later on.

After we've mapped all of our polygons, we can "Copy" the table data and paste this into Excel or some other tool. (I used Excel over a text editor to consolidate all of my data — more on this to come).

Step 2: Displaying our polygons

Now that we have our Shape ID, Point ID, X and Y coordinate data. Constructing the polygons in Tableau is easy!

Simply change the Graph Type to "Polygon" and you will see the new Mark Type "Path". The path to which this is referring is looking for the order of the points provided, to be able to plot them accurately on the X, Y plane.

Drag X and Y to Rows and Columns, Point ID to Path and Shape ID to Detail.

And there you have it! Polygon mapping in Tableau!

But if all we wanted to do was draw and image on an X, Y plane, we wouldn't need Tableau. And we aren't exactly generating any insights from this recreation of an image. We need data!

Step 3: Constructing our data

I retrieved some data on injuries in professional wrestling and it looked like this.

We have a two-value hierarchy of Body Group (Upper Extremity, Lower Extremity, Trunk and Head & Neck), and then this group split into Body Part, followed by a raw count of injuries over the study's life cycle.

Seeing as we already have Interworks data with a value representing each polygon (ShapeID), I saw creating an intermediate table to map Body Part to ShapeID as the most simple solution.

So once we jump into Tableau, we map "Interworks.ShapeID" to "PolygonMapping.ShapeID" and "PolygonMapping.BodyPart" to "ResearchData"."BodyPart".

We now have a connection between our polygons and the data we want to display!

> *Step 3: Adding context through background images*

We do still have one problem though. Looking back at what we have in Tableau so far. We can see that Seth's title belt is being treated as a polygon (technically another body part).

For people without context, they may not understand what this shape is or they may mistake it for a part of the body. Additionally, simply removing the polygon through filters doesn't particularly help.

Now we have a big gash through the centre of our figure, and while this would be one hell of an injury, it's not accurate to what we are trying to visualise.

Luckily, we can add a Background Image to the sheet to give context.

Simply go to Maps > Background Images > *Your Data Source*, and select the image you want to underlay. For our example, I have simply painted out most of Seth, from the image, leaving the Universal Championship belt.

Once we upload our image, give the X and Y margins to denote scale, and change the opacity, the output we are left with provides context without sacrificing any of the data points we wish to display.

Step 4: Visualising the data

Now that we have generated and displayed our polygons, constructed and connected our data sources, and added context to the image, the only part left is to visualise our results.

Drag your measure to the Colour mark type and viola!

Not only do we have a cool viz, but we can immediately make connections and gain insight.

While before we were focusing on “Body Group” in our data, by visualising it in a natural way, we can see that injuries are not localised in that way. Rather, they are most serious over the performers’ joints (knee, wrist, shoulder, elbow/forearm and ankles). This is something that we may not have immediately understood by looking at raw data or conventional visualisations, like bar and column graphs.

Professional Use Cases

I’ll admit, I highly doubt I, or anyone else, will ever make a living by visualising professional wrestling (if you do, let me know), but this feature has a wide variety of use cases.

The ability to visualise data in it’s natural, physical form, takes advantage of neural pathways and connections that are already available in humans. Mapping the human body is an example of this. We are able to immediately and accurately identify patterns.

correlations and potential causes to issues involving the human body.

Some other use cases include:

Custom Floorplan Visualisations

By using a customer's own floorplan as a template, future climate control systems could be customised to an individual dwelling (with the above example displaying temperature data captured by IoT-enabled devices inside a residential dwelling).

Additionally, similar implementations could use floorplans of shopping centres or airports to display spending trends and foot-traffic patterns.

Water Quality Visualisation Using Spatial Polygons

Mockup of water quality across the Lane Cove river and inner harbour region of Sydney Harbor.

As Tableau allows for the addition of custom spatial regions, the Interworks “Maps” option is very powerful. This makes the creation of custom, intricate polygons easy.

In the above example, part of the Lane Cove River and inner Sydney Habour have been given custom regions that local councils or organisations may want to keep an eye on.

By visualising the data in a physical representation, rather than by bar charts and dot plots, we would more easily point out sources of poor water quality and identify the flow patterns in these systems.

And there you have it. If you have any cool use cases to share, let me know and I can add them to the post.

If you have any further questions, don’t hesitate to reach out.

Darcy Vance - Graduate - Deloitte Australia | LinkedIn

View Darcy Vance's profile on LinkedIn, the world's largest professional community. Darcy has 1 job listed on their...

www.linkedin.com

Polygon Mapping Injury Tableau Data

Twitter Sentiment Analysis using fastText

Sanket Doshi Mar 6, 2019 · 9 min read

In this blog, we'll be analyzing the sentiments of various tweets using a fastText library which is easy to use and fast to train.

Twitter sentiment analysis

What is fastText?

FastText is an NLP library developed by the Facebook AI. It is an open-source, free, lightweight library that allows users to learn text representations and text classifiers. It works on standard, generic hardware. Models can later be reduced in size to even fit on mobile devices.

Why fastText?

The main disadvantage of deep neural network models is that they took a large amount of time to train and test. Here, fastText have an advantage as it takes very less amount of time to train and can be trained on our home computers at high speed.

As per the Facebook AI blog on fastText, the accuracy of this library is on par of deep neural networks and requires very less amount of time to train.

	Yahoo		Amazon full		Amazon polarity	
	Accuracy	Time	Accuracy	Time	Accuracy	Time
char-CNN	71.2	1 day	59.5	5 days	94.5	5 days
VDCNN	73.4	2h	63	7h	95.7	7h
fastText	72.3	5s	60.2	9s	94.6	10s

comparison between fastText and other deep learning based models

Now, we know about fastText and why we're using it we'll see how to use this library for sentiment analysis.

Get Dataset

We'll be using the dataset available on betsentiment.com. Tweets have four labels with values positive, negative, neutral and mixed. We'll ignore all the tweets with the mixed label.

We'll use teams tweet dataset as a training set while player dataset as a validation set.

Cleaning dataset

As we know, before training any model we need to clean data and it’s true here also.

We’ll clean tweets based on these rules:

1. Remove all the hashtags as hashtags do not affect sentiments.
2. Remove mentions as they also do not weigh in sentiment analyzing.
3. Replace any emojis with the text they represent as emojis or emoticons plays an important role in representing a sentiment.
4. Replace contractions with their full forms.
5. Remove any URLs present in tweets as they are not significant in sentiment analysis.
6. Remove punctuations.
7. Fix misspelled words (very basic as this is a very time-consuming step).
8. Convert everything to lowercase.
9. Remove HTML tags if present.

Rules to clean tweets:

We’ll clean this tweet

```
tweet = '<html> bayer leverkusen goalkeeeeper bernd leno will
not be #going to napoli. his agent uli ferber to bild: "I can
confirm that there were negotiations with napoli, which we
have broken off. napoli is not an option." Atletico madrid
and Arsenal are the other strong rumours. #b04 #afc </html>'
```

Remove HTML tags

Sometimes twitter response contains HTML tags and we need to remove this.

We'll be using `Beautifulsoup` package for this purpose.

If there are not HTML tags present than it will return the same text.

```
tweet = BeautifulSoup(tweet).get_text()

#output
'bayer leverkusen goalkeeeeper bernd leno will not be #going
to napoli. his agent uli ferber to bild: "I can confirm that
there were negotiations with napoli, which we have broken
off. napoli is not an option." Atletico madrid and Arsenal
are the other strong rumours. #b04 #afc'
```

We'll be using regex to match expressions to removed or to be replaced. For this, `re` package will be used.

Remove hashtags

Regex `@[A-Za-z0-9]+` represents mentions and `#[A-Za-z0-9]+` represents hashtags. We'll we replacing every word matching this regex with spaces.

```
tweet = ' '.join(re.sub("(@[A-Za-z0-9]+)|(#[A-Za-z0-9]+)", "
", tweet).split())

#output
'bayer leverkusen goalkeeeeper bernd leno will not be to
napoli. his agent uli ferber to bild: "I can confirm that
there were negotiations with napoli, which we have broken
```

```
off. napoli is not an option." Atletico madrid and Arsenal
are the other strong rumours.'
```

Remove URLs

Regex `\w+:\/\/\S+` matches all the URLs starting with http:// or https:// and replacing it with space.

```
tweet = ' '.join(re.sub("(\w+:\/\/\S+)", " ", tweet).split())

#output
'bayer leverkusen goalkeeeeper bernd leno will not be to
napoli. his agent uli ferber to bild: "I can confirm that
there were negotiations with napoli, which we have broken
off. napoli is not an option." Atletico madrid and Arsenal
are the other strong rumours.'
```

Remove punctuations

Replacing all the punctuations such as `.,!?:;-=` with space.

```
tweet = ' '.join(re.sub("[\.\,\!\?\:\;\-\=]", " ",
tweet).split())

#output
'bayer leverkusen goalkeeeeper bernd leno will not be napoli
his agent uli ferber to bild "I can confirm that there were
negotiations with napoli which we have broken off napoli is
not an option " Atletico madrid and Arsenal are the other
strong rumours'
```

Lower case

To avoid case sensitive issue

```
tweet = tweet.lower()


#output
'bayer leverkusen goalkeeeeper bernd leno will not be napoli
his agent uli ferber to bild "i can confirm that there were
negotiations with napoli which we have broken off napoli is
not an option " atletico madrid and arsenal are the other
strong rumours'
```

Replace contractions

Remove contractions and translate into appropriate slang. There is no universal list to replace contractions so we have made it for our purpose.

```
CONTRACTIONS = {"mayn't":"may not", "may've":"may
have",......}


tweet = tweet.replace("’","'")
words = tweet.split()
reformed = [CONTRACTIONS[word] if word in CONTRACTIONS else
word for word in words]
tweet = " ".join(reformed)


#input
'I mayn’t like you.'


#output
'I may not like you.'
```

Fix misspelled words

Here we are not actually building any complex function to correct the misspelled words but just checking that each character should occur not more than 2 times in every word. It’s a very basic misspelling check.

```
tweet = ''.join(''.join(s)[:2] for _, s in
itertools.groupby(tweet))


#output
'bayer leverkusen goalkeeper bernd leno will not be napoli
his agent uli ferber to bild "i can confirm that there were
negotiations with napoli which we have broken off napoli is
not an option " atletico madrid and arsenal are the other
strong rumours'
```

Replace emojis or emoticons

As emojis and emoticons play a significant role in expressing the sentiments we need to replace them with the expression they represent in plain English.

For emojis, we'll be using `emoji` package and for emoticons, we'll be building our own dictionary.

```
SMILEYS = {":-(":"sad", ":-)":"smiley", ....}

words = tweet.split()
reformed = [SMILEY[word] if word in SMILEY else word for word
in words]
tweet = " ".join(reformed)


#input
'I am :-('


#output
'I am sad'
```

For emojis

Emoji package return values for given emoji as `:flushed_face:` so we need to remove `:` from a given output.

```
tweet = emoji.demojize(tweet)
tweet = tweet.replace(":"," ")
tweet = ' '.join(tweet.split())


#input
'He is 😳'


#output
'He is flushed_face'
```

So, we've cleaned our data.

Why not use NLTK stop words?

Removing stop words is an efficient way while cleaning data. It removes all the insignificant words and usually is the most common words used in each sentence. To get all the stop words present in the NLTK library

```
from nltk.corpus import stopwords
stop_words = stopwords.words('english')
print(stop_words)
```

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you've", "you'll", "you'd", 'your',
'yours', 'yourself', 'yourselves', 'he', 'him', 'his', 'himself', 'she', "she's", 'her', 'hers', 'herself', 'it', "i
t's", 'its', 'itself', 'they', 'them', 'their', 'theirs', 'themselves', 'what', 'which', 'who', 'whom', 'this', 'tha
t', "that'll", 'these', 'those', 'am', 'is', 'are', 'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'havi
ng', 'do', 'does', 'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until', 'while', 'o
f', 'at', 'by', 'for', 'with', 'about', 'against', 'between', 'into', 'through', 'during', 'before', 'after', 'abov
e', 'below', 'to', 'from', 'up', 'down', 'in', 'out', 'on', 'off', 'over', 'under', 'again', 'further', 'then', 'onc
e', 'here', 'there', 'when', 'where', 'why', 'how', 'all', 'any', 'both', 'each', 'few', 'more', 'most', 'other', 'so
me', 'such', 'no', 'nor', 'not', 'only', 'own', 'same', 'so', 'than', 'too', 'very', 's', 't', 'can', 'will', 'just',
'don', "don't", 'should', "should've", 'now', 'd', 'll', 'm', 'o', 're', 've', 'y', 'ain', 'aren', "aren't", 'could
n', "couldn't", 'didn', "didn't", 'doesn', "doesn't", 'hadn', "hadn't", 'hasn', "hasn't", 'haven', "haven't", 'isn',
"isn't", 'ma', 'mightn', "mightn't", 'mustn', "mustn't", 'needn', "needn't", 'shan', "shan't", 'shouldn', "should
n't", 'wasn', "wasn't", 'weren', "weren't", 'won', "won't", 'wouldn', "wouldn't"]
```

NLTK stop words

We can see that if NLTK stopwords are used than all the negative contractions will be removed which plays a significant role in sentiment analysis.

Formatting the Dataset

Need to format the data in which fastText requires for supervised learning.

FastText assumes the labels are words that are prefixed by the string `__label__`.

The input to the fastText model should look like

```
__label__NEUTRAL _d i 'm just fine i have your fanbase angry
over
__label__POSITIVE what a weekend of football results & hearts
```

We can format our data using

```
def transform_instance(row):
    cur_row = []
    #Prefix the index-ed label with __label__
    label = "__label__" + row[4]
    cur_row.append(label)

cur_row.extend(nltk.word_tokenize(tweet_cleaning_for_sentimen
t_analysis(row[2].lower())))
    return cur_row


def preprocess(input_file, output_file):
    i=0
    with open(output_file, 'w') as csvoutfile:
        csv_writer = csv.writer(csvoutfile, delimiter=' ',
lineterminator='\n')
```

```
        with open(input_file, 'r', newline='',
encoding='latin1') as csvinfile: # encoding='latin1'
            csv_reader = csv.reader(csvinfile, delimiter=',',
quotechar='"')
            for row in csv_reader:
                if row[4]!="MIXED" and row[4].upper() in
['POSITIVE','NEGATIVE','NEUTRAL'] and row[2]!='':
                    row_output = transform_instance(row)
                    csv_writer.writerow(row_output )
                    # print(row_output)
                i=i+1
                if i%10000 ==0:
                    print(i)
```

Here, we are ignoring tweets with labels other than `Positive, Negative and neutral .`

`nltk.word_tokenize()` converts string into independent words.

```
nltk.word_tokenize('hello world!')

#output
['hello', 'world', '!']
```

Upsampling the dataset

In our dataset data is not equally divided into different labels. It contains around 72% of data in the neutral label. So, we can see that our model will tend to be overwhelmed by the large class and ignore the small ones.

```
import pandas as pd
import seaborn as sns
```

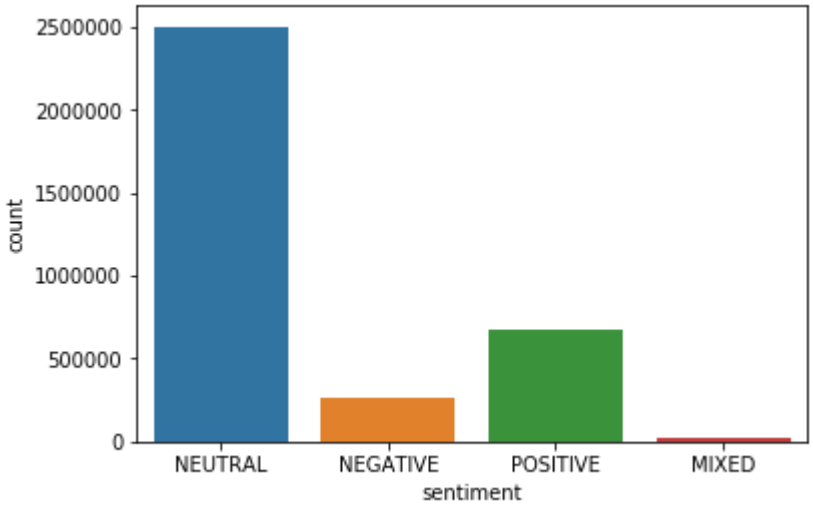
```
df = pd.read_csv('betsentiment-EN-tweets-sentiment-
teams.csv',encoding='latin1')
```

```
df['sentiment'].value_counts(normalize=True)*100
```

```
NEUTRAL     72.284338
POSITIVE    19.306613
NEGATIVE     7.624154
MIXED        0.784895
Name: sentiment, dtype: float64
```

percentage of tweets for each labels

```
sns.countplot(x="sentiment", data=df)
```



countplot for sentiment labels

As the NEUTRAL class consists of a large portion of the dataset, the model will always try to predict NEUTRAL label as it'll guarantee 72% of accuracy. To prevent this we need to have an equal number of tweets for each label. We can achieve this by adding new tweets to the minor class. This process of adding new tweets to the minority labels is known as upsampling.

We'll achieve upsampling by repeating the tweets present in the given label again and again until the number of tweets is equal in each label.

```
def upsampling(input_file, output_file, ratio_upsampling=1):
    # Create a file with equal number of tweets for each
label
    #    input_file: path to file
    #    output_file: path to the output file
    #    ratio_upsampling: ratio of each minority classes vs
majority one. 1 mean there will be as much of each class than
there is for the majority class

    i=0
    counts = {}
    dict_data_by_label = {}


# GET LABEL LIST AND GET DATA PER LABEL
    with open(input_file, 'r', newline='') as csvinfile:
        csv_reader = csv.reader(csvinfile, delimiter=',',
quotechar='"')
        for row in csv_reader:
            counts[row[0].split()[0]] =
counts.get(row[0].split()[0], 0) + 1
            if not row[0].split()[0] in dict_data_by_label:
                dict_data_by_label[row[0].split()[0]]=
[row[0]]
            else:
                dict_data_by_label[row[0].split()
[0]].append(row[0])
            i=i+1
            if i%10000 ==0:
                print("read" + str(i))


# FIND MAJORITY CLASS
    majority_class=""
    count_majority_class=0
    for item in dict_data_by_label:
        if
len(dict_data_by_label[item])>count_majority_class:
            majority_class= item

count_majority_class=len(dict_data_by_label[item])

    # UPSAMPLE MINORITY CLASS
    data_upsampled=[]
    for item in dict_data_by_label:
        data_upsampled.extend(dict_data_by_label[item])
```

```
        if item != majority_class:
            items_added=0
            items_to_add = count_majority_class -
len(dict_data_by_label[item])
            while items_added<items_to_add:

data_upsampled.extend(dict_data_by_label[item]
[:max(0,min(items_to_add-
items_added,len(dict_data_by_label[item])))])
                items_added = items_added +
max(0,min(items_to_add-
items_added,len(dict_data_by_label[item])))


# WRITE ALL
    i=0


with open(output_file, 'w') as txtoutfile:
        for row in data_upsampled:
            txtoutfile.write(row+ '\n' )
            i=i+1
            if i%10000 ==0:
                print("writer" + str(i))
```

As of repeating tweets, again and again, may cause our model to overfit our dataset but due to the large size of our dataset, this is not a problem.

Training

Try to install fastText with git clone rather than using pip.

We'll be using supervised training method.

```
hyper_params = {"lr": 0.01,
                "epoch": 20,
                "wordNgrams": 2,
                "dim": 20}

        print(str(datetime.datetime.now()) + ' START=>' +
str(hyper_params) )
```

```
# Train the model.
        model =
fastText.train_supervised(input=training_data_path,
**hyper_params)
        print("Model trained with the hyperparameter \n
{}".format(hyper_params))
```

`lr` represents `learning rate` , `epoch` represents `number of epoch` , `wordNgrams` represents `max length of word Ngram` , `dim` represents `size of word vectors` .

`train_supervised` is a function used to train the model using supervised learning.

Evaluate

We need to evaluate the model to find it's accuracy.

```
model_acc_training_set = model.test(training_data_path)
model_acc_validation_set = model.test(validation_data_path)

# DISPLAY ACCURACY OF TRAINED MODEL
text_line = str(hyper_params) + ",accuracy:" +
str(model_acc_training_set[1])  + ",validation:" +
str(model_acc_validation_set[1]) + '\n'


print(text_line)
```

We'll evaluate our model on both training as well as validation dataset.

`test` returns precision and recall of model rather than accuracy. But in our case both the values are almost the same so, we'll be using

precision only.

Overall the model gives an accuracy of 97.5% on the training data, and 79.7% on the validation data.

Predict

We'll predict the sentiment of text passed to our trained model.

```
model.predict(['why not'],k=3)
model.predict(['this player is so bad'],k=1)
```

`predict` lets us predict the sentiment of the passed string and `k` represents the number of labels to return with a confidence score.

Quantize the model

Quantizing helps us to reduce the size of the model.

```
model.quantize(input=training_data_path, qnorm=True,
retrain=True, cutoff=100000)
```

Save model

We can save our trained model and then can use anytime on the go rather than training it every time.

```
model.save_model(os.path.join(model_path,model_name +
".ftz"))
```

Conclusion

We learn how to clean data, and pass it to train model to predict the sentiment of tweets. We also learn to implement sentiment analysis model using fastText.

```
import fastText
import sys
import os
import nltk
nltk.download('punkt')
import csv
import datetime
from bs4 import BeautifulSoup
import re
import itertools
import emoji


##############################################################################
#
# DATA CLEANING
#
##############################################################################

# emoticons
def load_dict_smileys():

    return {
        ":-)":"smiley",
        ":-]":"smiley",
        ":-3":"smiley",
        ":->":"smiley",
        "8-)":"smiley",
        ":-}":"smiley",
        ":)":"smiley",
```

```
        ":]":"smiley",
        ":3":"smiley",
        ":>":"smiley",
        "8)":"smiley",
        ":}":"smiley",
        ":o)":"smiley",
        ":c)":"smiley",
        ":^)":"smiley",
        "=]":"smiley",
        "=)":"smiley",
        ":-))":"smiley",
        ":‿D":"smiley",
        "8‿D":"smiley",
        "x‿D":"smiley",
        "X‿D":"smiley",
        ":D":"smiley",
        "8D":"smiley",
        "xD":"smiley",
        "XD":"smiley",
        ":‿(":"sad",
        ":‿c":"sad",
        ":‿<":"sad",
        ":‿[":"sad",
        ":(":"sad",
        ":c":"sad",
        ":<":"sad",
        ":[":"sad",
        ":-||":"sad",
        ">:[":"sad",
        ":{":"sad",
        ":@":"sad",
        ">:(":"sad",
        ":'‿(":"sad",
        ":'(":"sad",
        ":‿P":"playful",
        "X‿P":"playful",
        "x‿p":"playful",
        ":‿p":"playful",
        ":‿Þ":"playful",
        ":‿þ":"playful",
        ":‿b":"playful",
        ":P":"playful",
        "XP":"playful",
        "xp":"playful",
        ":p":"playful",
```

```
          ":p":"playful",
          ":Þ":"playful",
          ":þ":"playful",
          ":b":"playful",
          "<3":"love"
          }

# self defined contractions
def load_dict_contractions():

    return {
          "ain't":"is not",
          "amn't":"am not",
          "aren't":"are not",
          "can't":"cannot",
          "'cause":"because",
          "couldn't":"could not",
          "couldn't've":"could not have",
          "could've":"could have",
          "daren't":"dare not",
          "daresn't":"dare not",
          "dasn't":"dare not",
          "didn't":"did not",
          "doesn't":"does not",
          "don't":"do not",
          "e'er":"ever",
          "em":"them",
          "everyone's":"everyone is",
          "finna":"fixing to",
          "gimme":"give me",
          "gonna":"going to",
          "gon't":"go not",
          "gotta":"got to",
          "hadn't":"had not",
          "hasn't":"has not",
          "haven't":"have not",
          "he'd":"he would",
          "he'll":"he will",
          "he's":"he is",
          "he've":"he have",
          "how'd":"how would",
          "how'll":"how will",
          "how're":"how are",
          "how's":"how is",
          "I'd":"I would",
```

```
        "I'll":"I will",
        "I'm":"I am",
        "I'm'a":"I am about to",
        "I'm'o":"I am going to",
        "isn't":"is not",
        "it'd":"it would",
        "it'll":"it will",
        "it's":"it is",
        "I've":"I have",
        "kinda":"kind of",
        "let's":"let us",
        "mayn't":"may not",
        "may've":"may have",
        "mightn't":"might not",
        "might've":"might have",
        "mustn't":"must not",
        "mustn't've":"must not have",
        "must've":"must have",
        "needn't":"need not",
        "ne'er":"never",
        "o'":"of",
        "o'er":"over",
        "ol'":"old",
        "oughtn't":"ought not",
        "shalln't":"shall not",
        "shan't":"shall not",
        "she'd":"she would",
        "she'll":"she will",
        "she's":"she is",
        "shouldn't":"should not",
        "shouldn't've":"should not have",
        "should've":"should have",
        "somebody's":"somebody is",
        "someone's":"someone is",
        "something's":"something is",
        "that'd":"that would",
        "that'll":"that will",
        "that're":"that are",
        "that's":"that is",
        "there'd":"there would",
        "there'll":"there will",
        "there're":"there are",
        "there's":"there is",
        "these're":"these are",
        "they'd":"they would"
```

```
            "they'd":"they would",
            "they'll":"they will",
            "they're":"they are",
            "they've":"they have",
            "this's":"this is",
            "those're":"those are",
            "'tis":"it is",
            "'twas":"it was",
            "wanna":"want to",
            "wasn't":"was not",
            "we'd":"we would",
            "we'd've":"we would have",
            "we'll":"we will",
            "we're":"we are",
            "weren't":"were not",
            "we've":"we have",
            "what'd":"what did",
            "what'll":"what will",
            "what're":"what are",
            "what's":"what is",
            "what've":"what have",
            "when's":"when is",
            "where'd":"where did",
            "where're":"where are",
            "where's":"where is",
            "where've":"where have",
            "which's":"which is",
            "who'd":"who would",
            "who'd've":"who would have",
            "who'll":"who will",
            "who're":"who are",
            "who's":"who is",
            "who've":"who have",
            "why'd":"why did",
            "why're":"why are",
            "why's":"why is",
            "won't":"will not",
            "wouldn't":"would not",
            "would've":"would have",
            "y'all":"you all",
            "you'd":"you would",
            "you'll":"you will",
            "you're":"you are",
            "you've":"you have",
            "Whatcha":"What are you",
```

```
209         "luv":"love",
210         "sux":"sucks"
211         }
212 
213 
214 def tweet_cleaning_for_sentiment_analysis(tweet):
215 
216     #Escaping HTML characters
217     tweet = BeautifulSoup(tweet).get_text()
218 
219     #Special case not handled previously.
220     tweet = tweet.replace('\x92',"'")
221 
222     #Removal of hastags/account
223     tweet = ' '.join(re.sub("(@[A-Za-z0-9]+)|(#[A-Za-z0-9]+)", " ", tweet).spl
224 
225     #Removal of address
226     tweet = ' '.join(re.sub("(\w+:\/\/\S+)", " ", tweet).split())
227 
228     #Removal of Punctuation
229     tweet = ' '.join(re.sub("[\.\,\!\?\:\;\-\=]", " ", tweet).split())
230 
231     #Lower case
232     tweet = tweet.lower()
233 
234     #CONTRACTIONS source: https://en.wikipedia.org/wiki/Contraction_%28grammar
235     CONTRACTIONS = load_dict_contractions()
236     tweet = tweet.replace("’","'")
237     words = tweet.split()
238     reformed = [CONTRACTIONS[word] if word in CONTRACTIONS else word for word
239     tweet = " ".join(reformed)
240 
241     # Standardizing words
242     tweet = ''.join(''.join(s)[:2] for _, s in itertools.groupby(tweet))
243 
244     #Deal with emoticons source: https://en.wikipedia.org/wiki/List_of_emotico
245     SMILEY = load_dict_smileys()
246     words = tweet.split()
247     reformed = [SMILEY[word] if word in SMILEY else word for word in words]
248     tweet = " ".join(reformed)
249 
250     #Deal with emojis
251     tweet = emoji.demojize(tweet)
252 
253     tweet = tweet.replace(":"," ")
```

```
    tweet = ' '.join(tweet.split())

    return tweet


#############################################################################
#
# DATA PROCESSING
#
#############################################################################

def transform_instance(row):
    cur_row = []
    #Prefix the index-ed label with __label__
    label = "__label__" + row[4]
    cur_row.append(label)
    cur_row.extend(nltk.word_tokenize(tweet_cleaning_for_sentiment_analysis(r
    return cur_row


def preprocess(input_file, output_file, keep=1):
    i=0
    with open(output_file, 'w') as csvoutfile:
        csv_writer = csv.writer(csvoutfile, delimiter=' ', lineterminator='\n'
        with open(input_file, 'r', newline='', encoding='latin1') as csvinfile
            csv_reader = csv.reader(csvinfile, delimiter=',', quotechar='"')
            for row in csv_reader:
                if row[4]!="MIXED" and row[4].upper() in ['POSITIVE','NEGATIVE
                    row_output = transform_instance(row)
                    csv_writer.writerow(row_output )
                    # print(row_output)
                i=i+1
                if i%10000 ==0:
                    print(i)

# Preparing the training dataset
preprocess('betsentiment-EN-tweets-sentiment-teams.csv', 'tweets.train')

# Preparing the validation dataset
preprocess('betsentiment-EN-tweets-sentiment-players.csv', 'tweets.validation'


#############################################################################
```

```
298 #
299 # UPSAMPLING
300 #
301 ################################################################################
302
303 def upsampling(input_file, output_file, ratio_upsampling=1):
304     # Create a file with equal number of tweets for each label
305     #    input_file: path to file
306     #    output_file: path to the output file
307     #    ratio_upsampling: ratio of each minority classes vs majority one. 1 m
308
309     i=0
310     counts = {}
311     dict_data_by_label = {}
312
313     # GET LABEL LIST AND GET DATA PER LABEL
314     with open(input_file, 'r', newline='') as csvinfile:
315         csv_reader = csv.reader(csvinfile, delimiter=',', quotechar='"')
316         for row in csv_reader:
317             counts[row[0].split()[0]] = counts.get(row[0].split()[0], 0) + 1
318             if not row[0].split()[0] in dict_data_by_label:
319                 dict_data_by_label[row[0].split()[0]]=[row[0]]
320             else:
321                 dict_data_by_label[row[0].split()[0]].append(row[0])
322             i=i+1
323             if i%10000 ==0:
324                 print("read" + str(i))
325
326     # FIND MAJORITY CLASS
327     majority_class=""
328     count_majority_class=0
329     for item in dict_data_by_label:
330         if len(dict_data_by_label[item])>count_majority_class:
331             majority_class= item
332             count_majority_class=len(dict_data_by_label[item])
333
334     # UPSAMPLE MINORITY CLASS
335     data_upsampled=[]
336     for item in dict_data_by_label:
337         data_upsampled.extend(dict_data_by_label[item])
338         if item != majority_class:
339             items_added=0
340             items_to_add = count_majority_class - len(dict_data_by_label[item]
341             while items_added<items_to_add:
342                 data_upsampled.extend(dict_data_by_label[item][:max(0,min(item
```

```
343                 items_added = items_added + max(0,min(items_to_add-items_added
344
345     # WRITE ALL
346     i=0
347
348     with open(output_file, 'w') as txtoutfile:
349         for row in data_upsampled:
350             txtoutfile.write(row+ '\n' )
351             i=i+1
352             if i%10000 ==0:
353                 print("writer" + str(i))
354
355
356 upsampling( 'tweets.train','uptweets.train')
357 # No need to upsample for the validation set. As it does not matter what valid
358
359
360 ##############################################################################
361 #
362 # TRAINING
363 #
364 ##############################################################################
365
366 # Full path to training data.
367 training_data_path ='uptweets.train'
368 validation_data_path ='tweets.validation'
369 model_path =''
370 model_name="model-en"
371
372 def train():
373     print('Training start')
374     try:
375         hyper_params = {"lr": 0.01,
376                         "epoch": 20,
377                         "wordNgrams": 2,
378                         "dim": 20}
379
380         print(str(datetime.datetime.now()) + ' START=>' + str(hyper_params) )
381
382         # Train the model.
383         model = fastText.train_supervised(input=training_data_path, **hyper_pa
384         print("Model trained with the hyperparameter \n {}".format(hyper_param
385
386         # CHECK PERFORMANCE
```

```
387           print(str(datetime.datetime.now()) + 'Training complete.' + str(hyper_
388
389           model_acc_training_set = model.test(training_data_path)
390           model_acc_validation_set = model.test(validation_data_path)
391
392           # DISPLAY ACCURACY OF TRAINED MODEL
393           text_line = str(hyper_params) + ",accuracy:" + str(model_acc_training_
394           print(text_line)
395
396           #quantize a model to reduce the memory usage
397           model.quantize(input=training_data_path, qnorm=True, retrain=True, cut
398
399           print("Model is quantized!!")
400           model.save_model(os.path.join(model_path,model_name + ".ftz"))
401
402           ################################################################################
403           #
404           #  TESTING PART
405           #
406           ################################################################################
407           model.predict(['why not'],k=3)
408           model.predict(['this player is so bad'],k=1)
409
410       except Exception as e:
411           print('Exception during training: ' + str(e) )
412
413
414   # Train your model.
415   train()
```

Machine Learning Sentiment Analysis Twitter Fasttext Supervised Learning