Fundamentals of Data Analysis - Assignment

Table of Contents

Introduction to Anscombe's Quartet

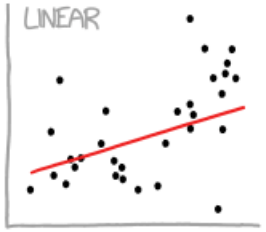
Anscombe's Quartet is a collection of four datasets constructed by the English statistician, Francis Anscombe, and published in the The American Statistician (https://www.jstor.org/stable/2682899) journal. In his words:

> Most textbooks on statistical methods, and most statistical computer programs, pay too little attention of graphs.

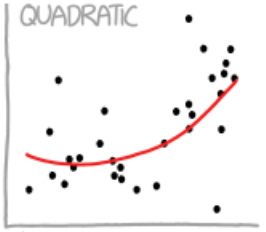
Although the paper was published in 1973, the absence of data visualization in routine data analysis is still widespread. There are often media reports where certain activies are claimed to cause diseases, such as 'Chips may cause cancer (https://www.dailymail.co.uk/news/article-125403/Chips-cause-cancer.html)' and 'Eating chicken and dairy makes breast cancer MORE deadly by 'helping it spread to other parts of the body' (https://www.thesun.co.uk/news/5524566/eating-chicken-and-dairy-makes-breast-cancer-more-deadly-by-helping-it-spread-to-other-parts-of-the-body/)'. However, the details of the studies and the sample size (number of patients studied) are usually omitted for the sake of popularity. There are more examples here (https://www.datapine.com/blog/misleading-statistics-and-data/).

In addition, there are also instances where inappropriate models are used to describe a dataset. Given a limited amount of data and no scientific background to the problem, it is possible to fit a linear, a partial quadratic and an exponential model all to the same dataset. Frequently, a chosen fit is (incorrectly) justified by using a single metric such as R^2.

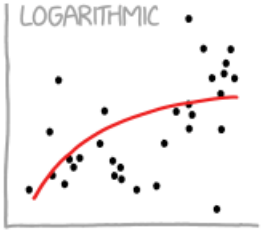
CURVE-FITTING METHODS
AND THE MESSAGES THEY SEND



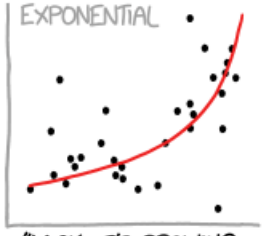
"HEY, I DID A REGRESSION."



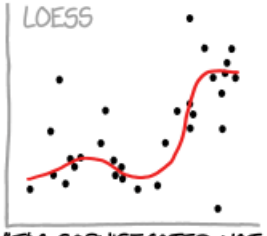
"I WANTED A CURVED LINE, SO I MADE ONE WITH MATH."



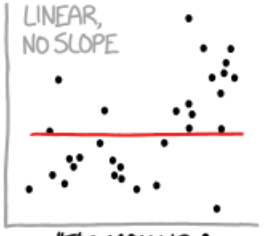
"LOOK, IT'S TAPERING OFF!"



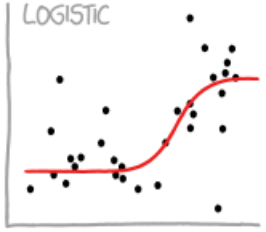
"LOOK, IT'S GROWING UNCONTROLLABLY!"



"I'M SOPHISTICATED, NOT LIKE THOSE BUMBLING POLYNOMIAL PEOPLE."

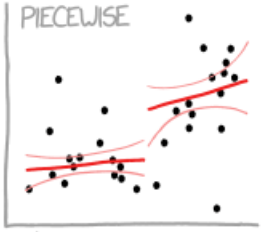


"I'M MAKING A SCATTER PLOT BUT I DON'T WANT TO."

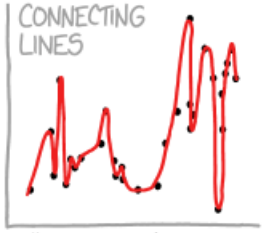


"I NEED TO CONNECT THESE TWO LINES, BUT MY FIRST IDEA DIDN'T HAVE ENOUGH MATH."

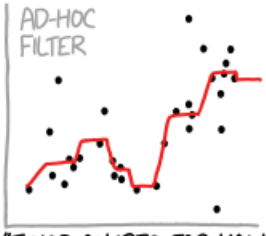
"LISTEN, SCIENCE IS HARD. BUT I'M A SERIOUS PERSON DOING MY BEST."



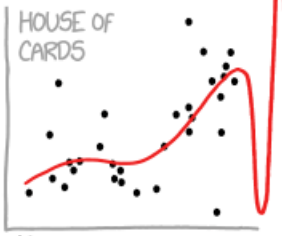
"I HAVE A THEORY, AND THIS IS THE ONLY DATA I COULD FIND.



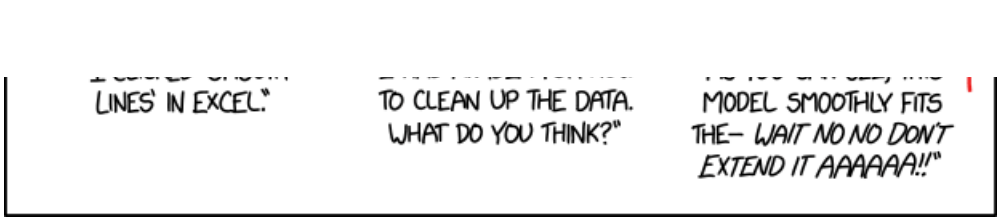
"I CLICKED 'SMOOTH



"I HAD AN IDEA FOR HOW

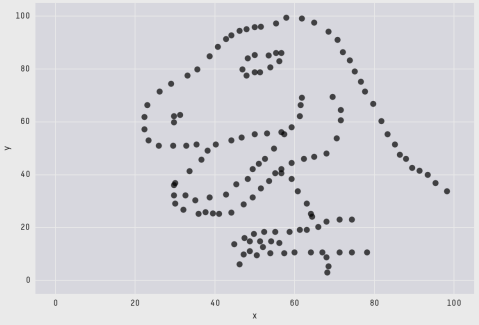


"AS YOU CAN SEE, THIS

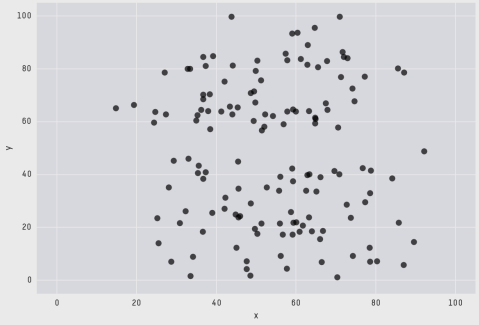
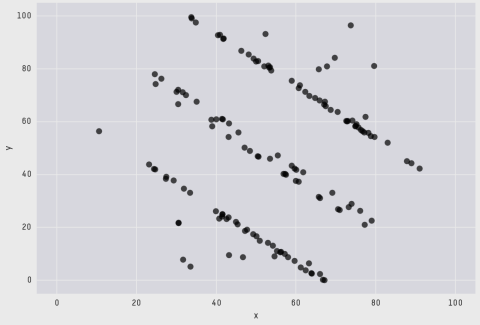
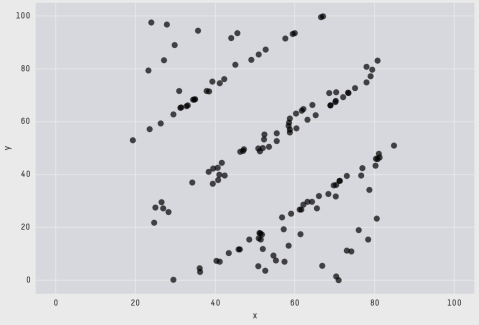
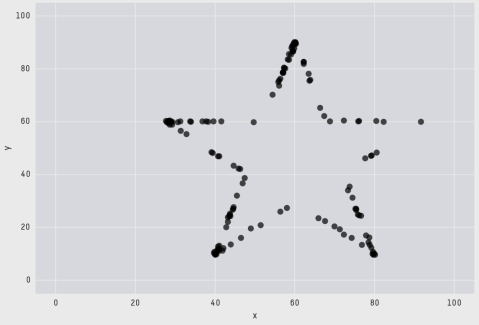
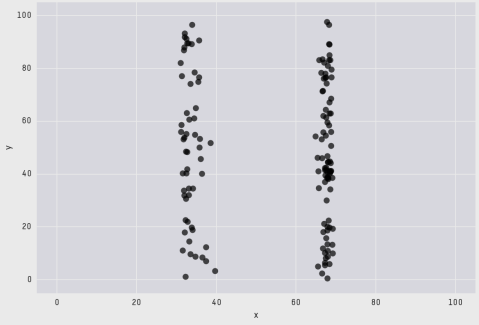
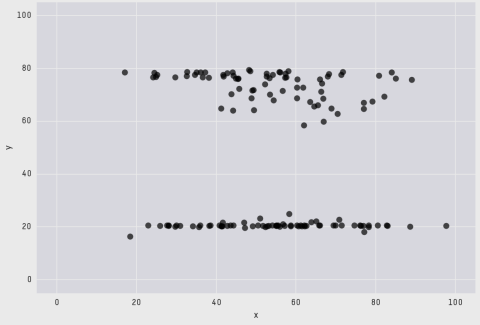
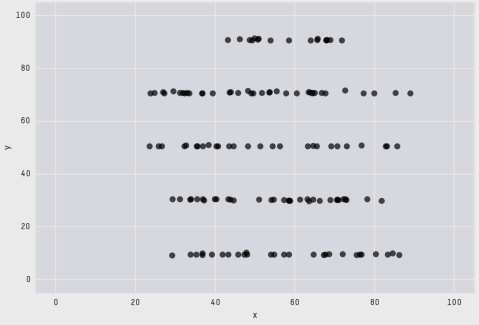
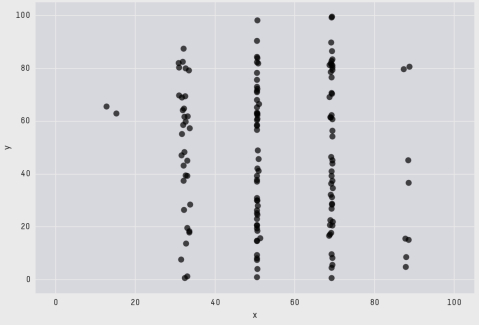
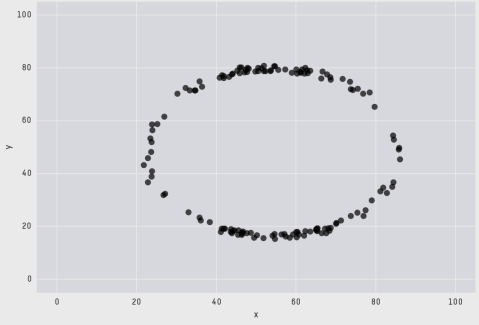
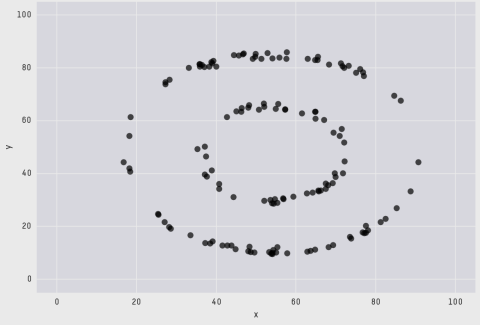
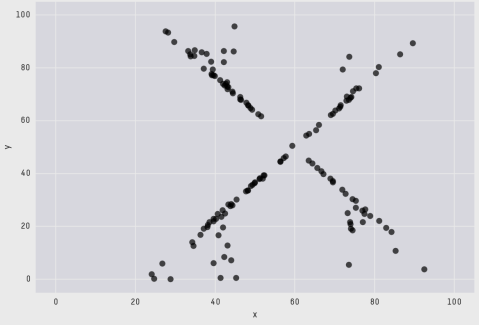
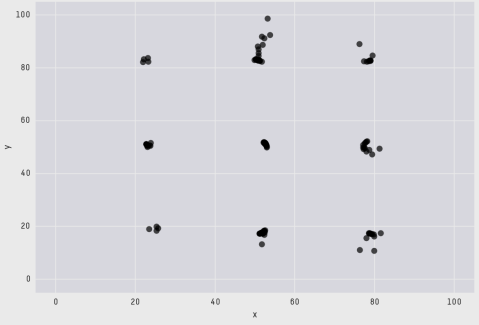


Source (https://www.explainxkcd.com/wiki/index.php/2048:_Curve-Fitting)

As such, the Anscombe's quartet is a perfect case-study on why a single metric should not be relied upon in data analysis **without** a visual inspection of the dataset. An extension of Anscombe's work is The Datasaurus Dozen (1 (https://www.autodeskresearch.com/publications/samestats) and 2 (https://doi.org/10.1145/3025453.3025912), where each dataset can be summarized similarly with respect to their mean, standard deviation, and Pearson's correlation.



X Mean: 54.26
Y Mean: 47.83
X SD : 16.76
Y SD : 26.93
Corr. : -0.06

Interestingly, the method by which Anscombe created his dataset is not known but very likely by trial and error (see Govindaraju & Haslett, 2008 (https://doi.org/10.1080/00207390701753788)). A few attempts to replicate Anscombe's work include:

1. Chatterjee & Firat (2007) (https://doi.org/10.1198/000313007X220057) - used a genetic algorithm-based approach, although the replicate dataset do not match Anscombe's dataset perfectly.
2. Govindaraju & Haslett (2008) (https://doi.org/10.1080/00207390701753788) - multiple iterations of simple linear regression approaching convergence, also not a perfect match.
3. Matejka & Fitzmaurice (https://doi.org/10.1145/3025453.3025912) - simulated annealing method agnostic to the target statistical properties.

Exploratory data analysis

Exploratory data analysis (EDA) is often used by data scientists to understand the structure of a particular dataset. No special calculations are performed, as the objective is to use as many straightforward, nonparametric descriptors as possible. In many instances, these include boxplots, scatter plots and distribution analysis.

EDA is a powerful technique that allows data scientists to focus their efforts towards a goal, *e.g.* predictive modeling, by deploying the available algorithms targeted towards their dataset. For instance, if nonlinear relationships among the variables are apparent from the EDA, the applicability of linear regression is then questionable.

In this notebook, various plots will be created using Python libraries to visually analyse the Anscombe's dataset, following by the calculation of their descriptive statistics. Before starting, however, these

In [1]:
```
# although we can read csv files with numpy, pandas is much better for data analytics
import pandas as pd
import numpy as np

# to make interactive plots with plotly
import plotly.graph_objs as go
from plotly.offline import download_plotlyjs, init_notebook_mode, iplot
init_notebook_mode(connected=True)
from plotly import tools

# for linear and quadratic regressions
from sklearn import linear_model, metrics
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline

# for calculating BIC and AIC
import pypunisher as punisher

# read in our dataset
filename = 'anscombe.csv'

df = pd.read_csv(filename)
df
```

Out[1]:

	Unnamed: 0	x1	x2	x3	x4	y1	y2	y3	y4
0	1	10	10	10	8	8.04	9.14	7.46	6.58
1	2	8	8	8	8	6.95	8.14	6.77	5.76
2	3	13	13	13	8	7.58	8.74	12.74	7.71
3	4	9	9	9	8	8.81	8.77	7.11	8.84
4	5	11	11	11	8	8.33	9.26	7.81	8.47
5	6	14	14	14	8	9.96	8.10	8.84	7.04
6	7	6	6	6	8	7.24	6.13	6.08	5.25
7	8	4	4	4	19	4.26	3.10	5.39	12.50
8	9	12	12	12	8	10.84	9.13	8.15	5.56
9	10	7	7	7	8	4.82	7.26	6.42	7.91
10	11	5	5	5	8	5.68	4.74	5.73	6.89

When we use the `read_csv()` function out-of-the-box by passing just the filename, pandas can automatically guess the header row and separator in the csv file as indicated by the documentation (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html).

Despite this table looking reasonably good, a much better presentable form can be obtained. First, the 'Unnamed' column should be used as the index, and the four datasets in the quartet presented in pairs whereby the column 'x1' is next to 'y1' and so on.

In [2]:
```
df = pd.read_csv(filename, index_col=0)
df = df[['x1','y1', 'x2','y2', 'x3', 'y3', 'x4', 'y4']]
df
```

Out[2]:

	x1	y1	x2	y2	x3	y3	x4	y4
1	10	8.04	10	9.14	10	7.46	8	6.58
2	8	6.95	8	8.14	8	6.77	8	5.76
3	13	7.58	13	8.74	13	12.74	8	7.71
4	9	8.81	9	8.77	9	7.11	8	8.84
5	11	8.33	11	9.26	11	7.81	8	8.47
6	14	9.96	14	8.10	14	8.84	8	7.04
7	6	7.24	6	6.13	6	6.08	8	5.25
8	4	4.26	4	3.10	4	5.39	19	12.50
9	12	10.84	12	9.13	12	8.15	8	5.56
10	7	4.82	7	7.26	7	6.42	8	7.91
11	5	5.68	5	4.74	5	5.73	8	6.89

The table above now is better organized than the one before, but still looks too plain. Although the Pandas library does allow styling of tables (https://pandas.pydata.org/pandas-docs/stable/style.html), it is easier to be performed using the Plotly library.

In [3]:
```
# reference: https://plot.ly/python/table/

trace = go.Table(
    header=dict(values=list(df.columns),
                fill = dict(color='blue'),
                align = ['center'],
                font = dict(color = 'white')
               ),

    cells=dict(values=[df.x1, df.y1,
                       df.x2, df.y2,
                       df.x3, df.y3,
                       df.x4, df.y4],
               fill = dict(color=['red', 'red', 'yellow', 'yellow', 'lightgreen', 'lightgreen', 'black', 'black']),
               align = ['center'],
               font = dict(color=['white', 'white', 'black', 'black', 'black', 'black', 'white', 'white'])
              )
               )

data = [trace]
layout = go.Layout(title = "<b>Table 1: Anscombe's Quartet</b>")

table = go.Figure(data = data, layout = layout)
iplot(table)
```

Table 1: Anscombe's Quartet

x1	y1	x2	y2	x3	y3	x4	y4
10	8.04	10	9.14	10	7.46	8	6.58
8	6.95	8	8.14	8	6.77	8	5.76
13	7.58	13	8.74	13	12.74	8	7.71
9	8.81	9	8.77	9	7.11	8	8.84
11	8.33	11	9.26	11	7.81	8	8.47
14	9.96	14	8.1	14	8.84	8	7.04
6	7.24	6	6.13	6	6.08	8	5.25
4	4.26	4	3.1	4	5.39	19	12.5
12	10.84	12	9.13	12	8.15	8	5.56
7	4.82	7	7.26	7	6.42	8	7.91
5	5.68	5	4.74	5	5.73	8	6.89

Export to plot.ly »

The values in the columns can also be ordered so that a sensible conclusion can be made by looking at the table itself.

In [4]:

```
# reorder the table above
# first, split the df because we need to do some separate operations, plotting, regression etc.
# lastly, we need to reset the indices to make it easier for remerging etc.

df_Q1 = df[['x1', 'y1']]
df_Q1 = df_Q1.sort_values('x1')
df_Q1 = df_Q1.reset_index(drop=True)

df_Q2 = df[['x2', 'y2']]
df_Q2 = df_Q2.sort_values('x2')
df_Q2 = df_Q2.reset_index(drop=True)

df_Q3 = df[['x3', 'y3']]
df_Q3 = df_Q3.sort_values('x3')
df_Q3 = df_Q3.reset_index(drop=True)

df_Q4 = df[['x4', 'y4']]
df_Q4 = df_Q4.sort_values('x4')
df_Q4 = df_Q4.reset_index(drop=True)

# because the plotly table header can only take 1 argument, we need to join the dataframes back again
# ref: https://pandas.pydata.org/pandas-docs/stable/merging.html
df = pd.concat([df_Q1, df_Q2, df_Q3, df_Q4], axis=1)
```

In [5]:

```
trace = go.Table(
    header=dict(values=list(df.columns),
                fill = dict(color='blue'),
                align = ['center'],
                font = dict(color = 'white')
               ),

    cells=dict(values=[df.x1, df.y1,
                       df.x2, df.y2,
                       df.x3, df.y3,
                       df.x4, df.y4],
               fill = dict(color=['red', 'red', 'yellow', 'yellow', 'lightgreen', 'lightgreen', 'black', 'black']),
               align = ['center'],
               font = dict(color=['white', 'white', 'black', 'black', 'black', 'black', 'white', 'white'])
              )
               )


data = [trace]

layout = go.Layout(title = "<b>Table 2: Anscombe's Quartet sorted by increasing <i>x</i> values</b>")

table = go.Figure(data = data, layout = layout)
iplot(table)
```

Table 2: Anscombe's Quartet sorted by increasing *x* values

x1	y1	x2	y2	x3	y3	x4	y4
4	4.26	4	3.1	4	5.39	8	6.58
5	5.68	5	4.74	5	5.73	8	5.76
6	7.24	6	6.13	6	6.08	8	7.71
7	4.82	7	7.26	7	6.42	8	8.84
8	6.95	8	8.14	8	6.77	8	8.47
9	8.81	9	8.77	9	7.11	8	7.04
10	8.04	10	9.14	10	7.46	8	5.25
11	8.33	11	9.26	11	7.81	8	5.56
12	10.84	12	9.13	12	8.15	8	7.91
13	7.58	13	8.74	13	12.74	8	6.89
14	9.96	14	8.1	14	8.84	19	12.5

Export to plot.ly »

The table above easily shows to us that:

1. Dataset I - there is an almost linear increase in y as x increases but with dips at a minimum of 2 points (roughly)
2. Dataset II - y increases as x increases and then decreases again, suggesting a quadratic relationship
3. Dataset III - y increases as x increases but there is an outlier in y (12.74)
4. Dataset IV is a bit problematic because the same value of x is associated with multiple y values, and there seems to be an outlier x (19)

Of course, because we as humans are better with visuals, we will create interactive plots* of these datasets using Plotly.

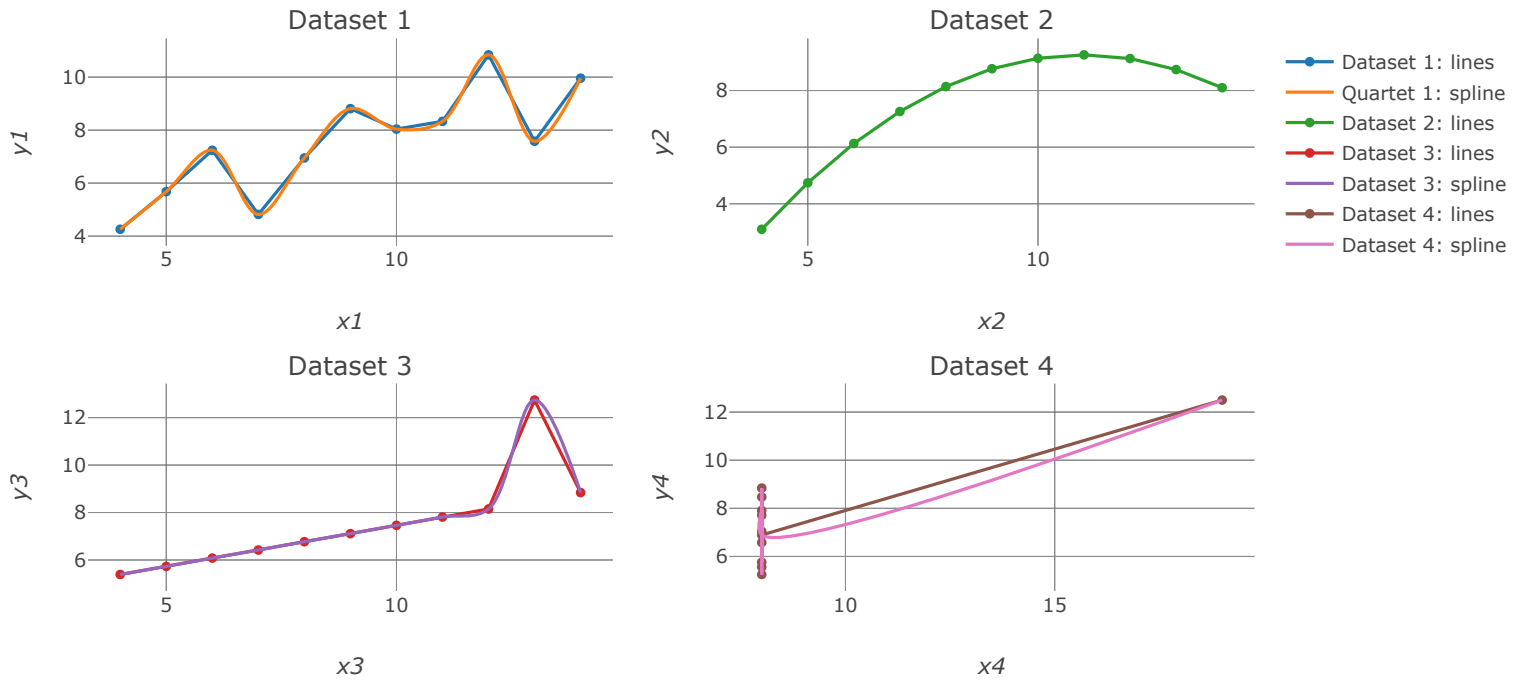
**Interactivity means the end-user can zoom in/out, change scales and even edit the plots with Plotly Chart Studio (a free tool).*

A classical method in EDA for simple two-variable datasets is a scatter plot. While true scatter plots only contain the data points themselves, it is also often the case that lines, either straight or smoothed (splines), are added to obtain a cursory trend information. The resulting plot is usually termed line plots.

In [6]:

```
# https://plot.ly/python/subplots/#customizing-subplot-axes

trace1 = go.Scatter(x=df_Q1['x1'], y=df_Q1['y1'], mode='lines+markers', name = 'Dataset 1: lines')
trace1_spline = go.Scatter(
                    x=df_Q1['x1'],
                    y=df_Q1['y1'],
                    mode='lines',
                    line=dict(shape='spline'),
                    name='Quartet 1: spline'
                    )

# no spline for Q2, as it will just be like the lines anyway
trace2 = go.Scatter(x=df_Q2['x2'], y=df_Q2['y2'], mode='lines+markers', name = 'Dataset 2: lines')

trace3 = go.Scatter(x=df_Q3['x3'], y=df_Q3['y3'], mode='lines+markers', name = 'Dataset 3: lines')
trace3_spline = go.Scatter(
                    x=df_Q3['x3'],
                    y=df_Q3['y3'],
                    mode='lines',
                    line=dict(shape='spline'),
                    name='Dataset 3: spline'
                    )

trace4 = go.Scatter(x=df_Q4['x4'], y=df_Q4['y4'], mode='lines+markers', name = 'Dataset 4: lines')
trace4_spline = go.Scatter(
                    x=df_Q4['x4'],
                    y=df_Q4['y4'],
                    mode='lines',
                    line=dict(shape='spline'),
                    name='Dataset 4: spline'
                    )

# we use print_grid = False to remove the grid output
# ref: https://stackoverflow.com/questions/43106170/remove-the-this-is-the-format-of-your-plot-grid-in-a-plotly-subplot-in-a-jupy
fig = tools.make_subplots(rows=2, cols=2,
                          subplot_titles=('Dataset 1', 'Dataset 2', 'Dataset 3', 'Dataset 4'),
                         print_grid = False)

# subplot for upper left
fig.append_trace(trace1, 1, 1)
fig.append_trace(trace1_spline, 1, 1)

# subplot for upper right
fig.append_trace(trace2, 1, 2)

# subplot for lower left
fig.append_trace(trace3, 2, 1)
fig.append_trace(trace3_spline, 2, 1)

# subplot for lower right
fig.append_trace(trace4, 2, 2)
fig.append_trace(trace4_spline, 2, 2)
```

```
fig['layout']['xaxis1'].update(title='<i>x1</i>')
fig['layout']['xaxis2'].update(title='<i>x2</i>')
fig['layout']['xaxis3'].update(title='<i>x3</i>')
fig['layout']['xaxis4'].update(title='<i>x4</i>')

fig['layout']['yaxis1'].update(title='<i>y1</i>')
fig['layout']['yaxis2'].update(title='<i>y2</i>')
fig['layout']['yaxis3'].update(title='<i>y3</i>')
fig['layout']['yaxis4'].update(title='<i>y4</i>')

fig['layout'].update(title="<b>Figure 1: Lines plots of the Anscombe's Quartet</b>")

iplot(fig)
```



These line plots visually confirms our table assessment (see Table 2 above). Let's see what happens if we fit a linear regression line.

In [7]:

```
# ref: https://plot.ly/scikit-learn/plot-ols/

# Dataset 1 - the scikit fit() function expects numpy ndarray, not pandas dataframe so we convert first
Q1_X = df_Q1['x1'].iloc[:].values
Q1_X = Q1_X.reshape(-1,1) # we have to reshape because the fit() function expects a 2D array
Q1_Y = df_Q1['y1'].iloc[:].values

# Dataset 2
Q2_X = df_Q2['x2'].iloc[:].values
Q2_X = Q2_X.reshape(-1,1)
Q2_Y = df_Q2['y2'].iloc[:].values

# Dataset 3
Q3_X = df_Q3['x3'].iloc[:].values
Q3_X = Q3_X.reshape(-1,1)
Q3_Y = df_Q3['y3'].iloc[:].values

# Dataset 4
Q4_X = df_Q4['x4'].iloc[:].values
Q4_X = Q4_X.reshape(-1,1)
Q4_Y = df_Q4['y4'].iloc[:].values

# variable to instantiate and hold the LinearRegression object
regr = linear_model.LinearRegression()

# let's fit linear lines to the X/Y in each quartet separately
# this is actually similar to numpy polyfit function
# ref: https://stackoverflow.com/questions/32660231/how-to-fit-a-polynomial-curve-to-data-using-scikit-learn
regr_Q1 = regr.fit(Q1_X, Q1_Y)
regr_Q2 = regr.fit(Q2_X, Q2_Y)
regr_Q3 = regr.fit(Q3_X, Q3_Y)
regr_Q4 = regr.fit(Q4_X, Q4_Y)

# predicted Y values
Q1_Y_pred = regr_Q1.predict(Q1_X)
Q2_Y_pred = regr_Q2.predict(Q2_X)
Q3_Y_pred = regr_Q3.predict(Q3_X)
Q4_Y_pred = regr_Q4.predict(Q4_X)
```

Linear regression models take the form of $y = ax + b$, where a is the slope of the line and b is the intercept (the y value when $x = 0$. To compare the values of a and b, the `coef_` and `intercept_` properties of the `LinearRegression()` object can be called. We can also determine the errors and R^2 (coefficient of determination) values for each fit using the `metrics` functions.

According to the documentation (https://scikit-learn.org/stable/modules/model_evaluation.html), these metrics are calculated according to the equations below:

Mean Absolute Error

$$\text{MAE}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} |y_i - \hat{y}_i|$$

Mean Squared Error

$$\text{MSE}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} (y_i - \hat{y}_i)^2$$

Root Mean Squared Error

This is simply the square root of MSE.

R^2

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=0}^{n_{\text{samples}}-1} (y_i - \hat{y}_i)^2}{\sum_{i=0}^{n_{\text{samples}}-1} (y_i - \bar{y})^2}$$

For convenience, all the values in Table 3 below have been rounded up/down to 5 decimal places.

In [8]:
```
trace = go.Table(
    header=dict(values=['<b>Property</b>', '<b>Dataset 1</b>', '<b>Dataset 2</b>', '<b>Dataset 3</b>', '<b>Dataset 4</b>'],
                fill = dict(color='blue'),
                align = ['center'],
                font = dict(color = 'white')
               ),

    cells=dict(values=[
        ['<b>Slope</b>', '<b>Intercept</b>', '<b>Mean Absolute Error</b>', '<b>Mean Squared Error</b>',
         '<b>Root Mean Squared Error</b>', '<b><i>R<sup>2</sup></i></b>'],
        [np.around(regr_Q1.coef_, 5), np.around(regr_Q1.intercept_, 5),
             np.around(metrics.mean_absolute_error(Q1_Y, Q1_Y_pred), 5),
             np.around(metrics.mean_squared_error(Q1_Y, Q1_Y_pred), 5),
             np.around(np.sqrt(metrics.mean_squared_error(Q1_Y, Q1_Y_pred)), 5),
             np.around(metrics.r2_score(Q1_Y, Q1_Y_pred), 5)
        ],
        [np.around(regr_Q2.coef_, 5), np.around(regr_Q2.intercept_, 5),
             np.around(metrics.mean_absolute_error(Q2_Y, Q2_Y_pred), 5),
             np.around(metrics.mean_squared_error(Q2_Y, Q2_Y_pred), 5),
             np.around(np.sqrt(metrics.mean_squared_error(Q2_Y, Q2_Y_pred)), 5),
             np.around(metrics.r2_score(Q2_Y, Q2_Y_pred), 5)
        ],
        [np.around(regr_Q3.coef_, 5), np.around(regr_Q3.intercept_, 5),
             np.around(metrics.mean_absolute_error(Q3_Y, Q3_Y_pred), 5),
             np.around(metrics.mean_squared_error(Q3_Y, Q3_Y_pred), 5),
             np.around(np.sqrt(metrics.mean_squared_error(Q3_Y, Q3_Y_pred)), 5),
             np.around(metrics.r2_score(Q3_Y, Q3_Y_pred), 5)
        ],
        [np.around(regr_Q4.coef_, 5), np.around(regr_Q4.intercept_, 5),
             np.around(metrics.mean_absolute_error(Q4_Y, Q4_Y_pred), 5),
             np.around(metrics.mean_squared_error(Q4_Y, Q4_Y_pred), 5),
             np.around(np.sqrt(metrics.mean_squared_error(Q4_Y, Q4_Y_pred)), 5),
             np.around(metrics.r2_score(Q4_Y, Q4_Y_pred), 5)
        ]
    ],
              fill = dict(color=['red', 'rgb(123, 252, 199)', 'rgb(123, 252, 199)', 'rgb(123, 252, 199)',
                                 'rgb(123, 252, 199)']),
              align = ['center'],
              font = dict(color=['white', 'black', 'black', 'black', 'black'])
             ),
    columnwidth = [12,6,6,6,6]
              )

data = [trace]

layout = go.Layout(title = "<b>Table 3: Summary of linear fits for the Anscombe's Quartet</b>")

table = go.Figure(data = data, layout = layout)
iplot(table)
```

Table 3: Summary of linear fits for the Anscombe's Quartet

Property	Dataset 1	Dataset 2	Dataset 3	Dataset 4
Slope	0.49991	0.49991	0.49991	0.49991
Intercept	3.00173	3.00173	3.00173	3.00173
Mean Absolute Error	0.83731	0.96793	0.71648	0.90273
Mean Squared Error	1.25115	1.25239	1.25056	1.24932
Root Mean Squared Error	1.11855	1.1191	1.11829	1.11773
R^2	0.66654	0.66624	0.66632	0.66671

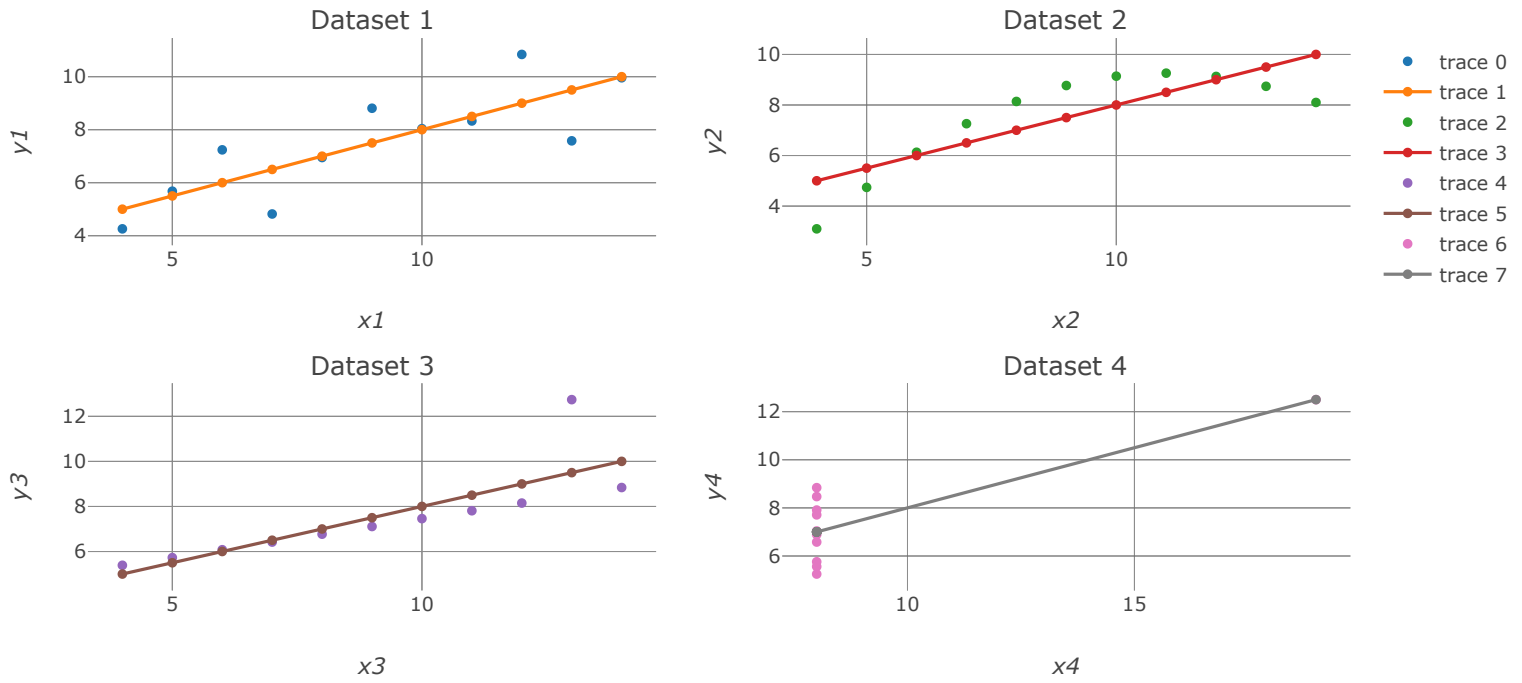
Export to plot.ly »

Looking at the various metrics, there are only very small differences among them such that it is not possible to make any conclusion with respect to the quality of the linear fits. For example, a linear model is seemingly capable of explaining about 70 % of the data points in Dataset 2, which is obviously not entirely accurate given their curvature (Figure 1). Therefore, we will need to plot these fits and their residuals to obtain a better view of the corresponding goodness of fit.

In [9]:

```
# add linear fit lines to scatter plots
# https://plot.ly/python/subplots/#customizing-subplot-axes

trace1 = go.Scatter(x=df_Q1['x1'], y=df_Q1['y1'], mode='markers')
trace1_linReg = go.Scatter(x=df_Q1['x1'], y = Q1_Y_pred)

trace2 = go.Scatter(x=df_Q2['x2'], y=df_Q2['y2'], mode='markers')
trace2_linReg = go.Scatter(x=df_Q2['x2'], y = Q2_Y_pred)

trace3 = go.Scatter(x=df_Q3['x3'], y=df_Q3['y3'], mode='markers')
trace3_linReg = go.Scatter(x=df_Q3['x3'], y = Q3_Y_pred)

trace4 = go.Scatter(x=df_Q4['x4'], y=df_Q4['y4'], mode='markers')
trace4_linReg = go.Scatter(x=df_Q4['x4'], y = Q4_Y_pred)

fig = tools.make_subplots(rows=2, cols=2, subplot_titles=('Dataset 1', 'Dataset 2',
                                                          'Dataset 3', 'Dataset 4'), print_grid = False)
fig.append_trace(trace1, 1, 1)
fig.append_trace(trace1_linReg, 1, 1)

fig.append_trace(trace2, 1, 2)
fig.append_trace(trace2_linReg, 1, 2)

fig.append_trace(trace3, 2, 1)
fig.append_trace(trace3_linReg, 2, 1)

fig.append_trace(trace4, 2, 2)
fig.append_trace(trace4_linReg, 2, 2)

fig['layout']['xaxis1'].update(title='<i>x1</i>')
fig['layout']['xaxis2'].update(title='<i>x2</i>')
fig['layout']['xaxis3'].update(title='<i>x3</i>')
fig['layout']['xaxis4'].update(title='<i>x4</i>')

fig['layout']['yaxis1'].update(title='<i>y1</i>')
fig['layout']['yaxis2'].update(title='<i>y2</i>')
fig['layout']['yaxis3'].update(title='<i>y3</i>')
fig['layout']['yaxis4'].update(title='<i>y4</i>')

fig['layout'].update(title="<b>Figure 2: Scatter plots of Anscombe's Quartet with linear regression lines</b>")

iplot(fig)
```

Figure 2: Scatter plots of Anscombe's Quartet with linear regression lines



The presence of outliers and data curvature have distorted the fits such that a linear fit looks deceivingly acceptable quantitatively for all these cases but not visually. In reality, a linear fit is only suitable for Dataset 1 and (to a lesser extent) for Dataset 3. Dataset 2 is best modeled by a quadratic fit, while the outlier for Dataset 4 ($x = 19, y = 12.5$) should be removed prior to any fitting to yield a vertical line (parallel to the y axis of $x = 8$).

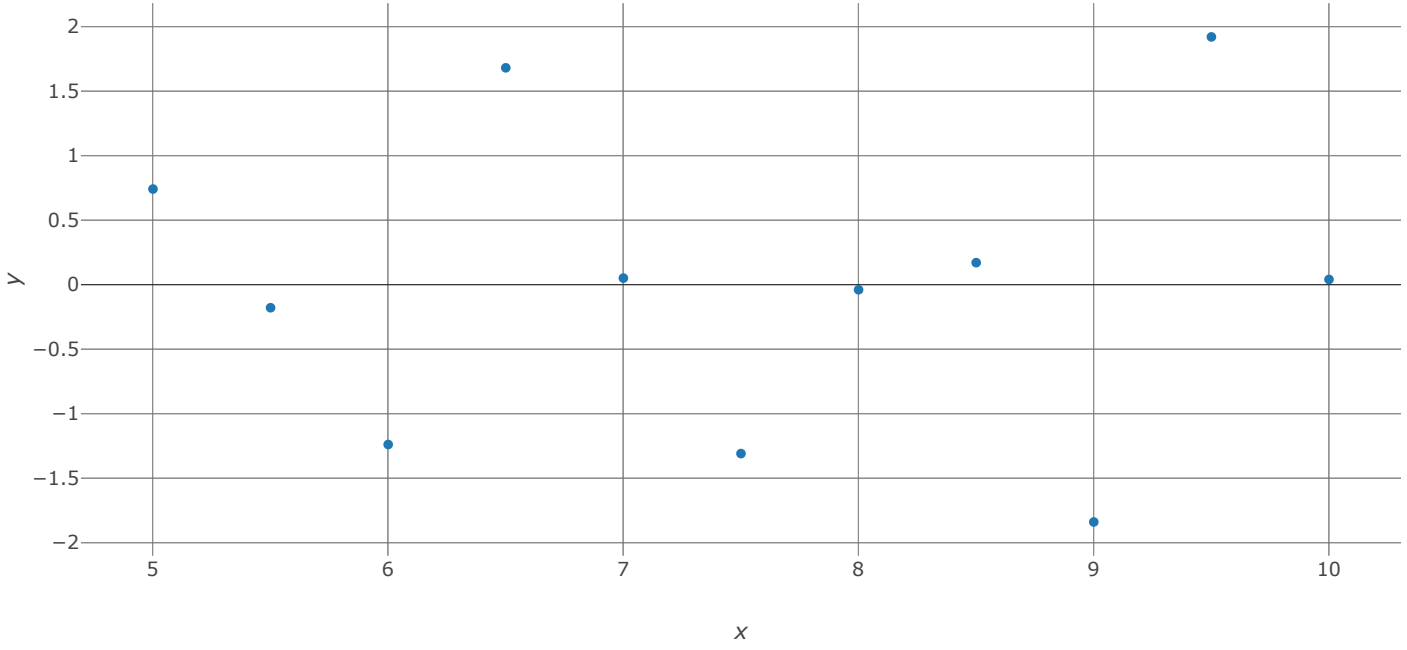
To confirm our line of thought, we will create the following additional plots and fits:

1. check the residual plot for Dataset 1 for goodness of linear fit
2. confirm that a quadratic fit for Dataset 2 is much better than a linear fit and then check the residual plot to identify the better fit
3. confirm the presence of outliner(s) for Datasets 3 and 4

Assessment of Linear Fit for Dataset 1

In [10]:
```
# calculate the residuals
Q1_residual = Q1_Y_pred - Q1_Y

data = [go.Scatter(x=Q1_Y_pred, y = Q1_residual, mode='markers')]

layout= go.Layout(
    title= '<b>Figure 3: Residual plot for linear fit of Dataset 1',
    xaxis = dict(title = '<i>x</i>'),
    yaxis = dict(title = '<i>y</i>')
)

fig = go.Figure(data=data, layout=layout)
iplot(fig)
```

Figure 3: Residual plot for linear fit of Dataset 1



Export to plot.ly »

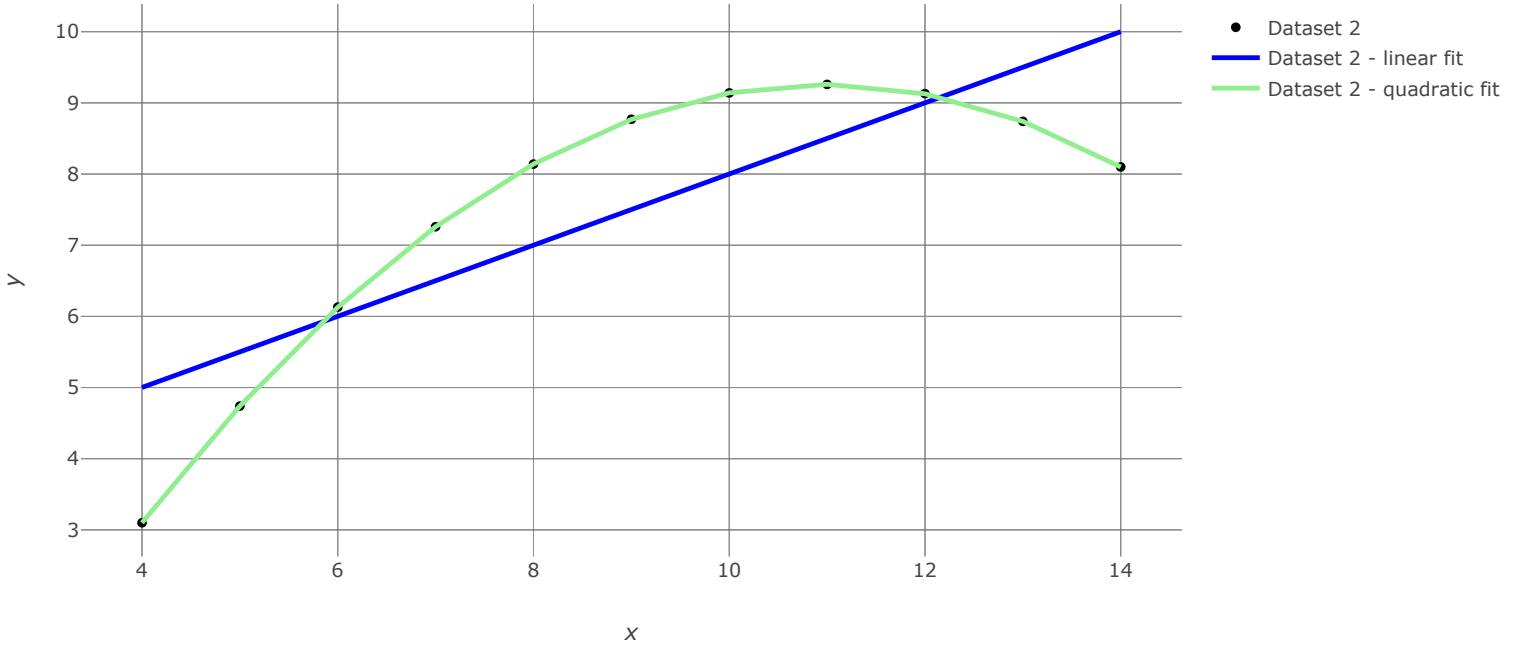
This residual plot shows no apparent structure and the values are scattered above and below 0 randomly, which means the linear fit is statically acceptable for Dataset 1.

Comparing Linear and Quadratic Fits for Dataset 2

In [11]:

```
# https://stackoverflow.com/questions/33710829/linear-regression-with-quadratic-terms
# https://scikit-learn.org/stable/auto_examples/linear_model/plot_polynomial_interpolation.html

# variable to initialize and hold the LinearRegression() object
quadratic_model = make_pipeline(PolynomialFeatures(2), linear_model.LinearRegression())
regr_Q2_quad = quadratic_model.fit(Q2_X, Q2_Y)
Q2_Y_predQuadratic = regr_Q2_quad.predict(Q2_X)

# variable to hold model coefficients, which we will extract later on
# https://stackoverflow.com/questions/33876900/how-to-extract-equation-from-a-polynomial-fit/33876965
regr_Q2_quad_coeffs = regr_Q2_quad.named_steps['linearregression']

trace1 = go.Scatter(x=df_Q2['x2'], y=df_Q2['y2'],
                    mode='markers', name = 'Dataset 2', marker = dict(color = 'black'))

trace1_linReg = go.Scatter(x=df_Q2['x2'], y = Q2_Y_pred,
                           name = 'Dataset 2 - linear fit',
                           mode = 'lines',
                           line = dict(color = 'blue', width= 3)
                          )
trace1_quadReg = go.Scatter(x=df_Q2['x2'], y = Q2_Y_predQuadratic,
                            name = 'Dataset 2 - quadratic fit',
                            mode = 'lines',
                            line = dict(color = 'lightgreen', width = 3)
                           )

data = [trace1, trace1_linReg, trace1_quadReg]

layout= go.Layout(
    title= '<b>Figure 4: Linear and Quadratic Fits for Dataset 2</b>',
    xaxis = dict(title = '<i>x</i>'),
    yaxis = dict(title = '<i>y</i>')
)

fig = go.Figure(data=data, layout=layout)

iplot(fig)
```

Figure 4: Linear and Quadratic Fits for Dataset 2



Export to plot.ly »

While it is clear that the quadratic fit is significantly better than the linear fit, we can use the `metrics` functions to quantify their corresponding goodness of fit.

When comparing fits, it is important to penalize the more complicated models. Higher-order polynomials, for example, will better describe both the data and noise, leading to overfitting (see Figure 1). At the same time, linear models are incapable of handling true curvature in the data. To quantitatively decide the best model, there are two similar albeit different metrics that can be used - Akaike Information Criterion (AIC) and Bayesian information criterion (BIC), which can be calculated as (https://ubc-mds.github.io/PyPunisher/code.html):

$$AIC = -2ln(L) + 2k$$
$$BIC = -2ln(L) + ln(n)k$$

where L is the maximized value of the likelihood function and k is the number of parameters

The difference between these metrics stems from the penalty term of $2k$ for AIC and $ln(n)k$ for BIC, leading to suggestions that one is more appropriate than the other under different situations [Burnham & Anderson (2004) (http://doi.org/10.1177/0049124104268644) and Vrieze (2012) (https://doi.org/10.1037/a0027127)].

When the fit of a model to the data is relatively good, AIC and BIC tend to have similar values. A better model is represented by a lower AIC/BIC values.

In [12]:
```
trace = go.Table(
    header=dict(values=['<b>Property</b>', '<b>Linear</b>', '<b>Quadratic</b>'],
                fill = dict(color='blue'),
                align = ['center'],
                font = dict(color = 'white')
               ),

    cells=dict(values=[
        ['<b>Mean Absolute Error</b>', '<b>Mean Squared Error</b>',
         '<b>Root Mean Squared Error</b>', '<b><i>R<sup>2</sup></i></b>', '<b>AIC</b>', '<b>BIC</b>', '<b>Equation</b>'],
        [np.around(metrics.mean_absolute_error(Q2_Y, Q2_Y_pred), 5),
             np.around(metrics.mean_squared_error(Q2_Y, Q2_Y_pred), 5),
             np.around(np.sqrt(metrics.mean_squared_error(Q2_Y, Q2_Y_pred)), 5),
             np.around(metrics.r2_score(Q2_Y, Q2_Y_pred), 5),
             np.around(punisher.metrics.criterion.aic(regr_Q2, Q2_X, Q2_Y), 5),
             np.around(punisher.metrics.criterion.bic(regr_Q2, Q2_X, Q2_Y), 5),
             # we need to convert np.around object to string to enable string concatenation
             ['y2 = ' + np.array2string(np.around(regr_Q2.coef_[0], 5)) + 'x + ' +
              np.array2string(np.around(regr_Q2.intercept_, 5))]

        ],
        [np.around(metrics.mean_absolute_error(Q2_Y, Q2_Y_predQuadratic), 5),
             np.around(metrics.mean_squared_error(Q2_Y, Q2_Y_predQuadratic), 5),
             np.around(np.sqrt(metrics.mean_squared_error(Q2_Y, Q2_Y_predQuadratic)), 5),
             np.around(metrics.r2_score(Q2_Y, Q2_Y_predQuadratic), 5),
             np.around(punisher.metrics.criterion.aic(regr_Q2_quad, Q2_X, Q2_Y), 5),
             np.around(punisher.metrics.criterion.bic(regr_Q2_quad, Q2_X, Q2_Y), 5),
             ['y2 = ' + np.array2string(np.around(regr_Q2_quad_coeffs.coef_[2], 5)) + 'x^2 + ' +
              np.array2string(np.around(regr_Q2_quad_coeffs.coef_[1], 5)) + 'x + (' +
              np.array2string(np.around(regr_Q2_quad_coeffs.intercept_, 5)) + ')']
        ]
    ],
                fill = dict(color=['red', 'rgb(123, 252, 199)', 'rgb(123, 252, 199)', 'rgb(123, 252, 199)',
                                   'rgb(123, 252, 199)']),
                align = ['center'],
                font = dict(color=['white', 'black', 'black', 'black', 'black'])
               ),
    # columnwidth = [12,6,6,6,6]
                )

data = [trace]

layout = go.Layout(title = "<b>Table 4: Summary of linear and quadratic fits for Dataset 2</b>")

table = go.Figure(data = data, layout = layout)
iplot(table)

# add the equations at the bottom
```

Table 4: Summary of linear and quadratic fits for Dataset 2

Property	Linear	Quadratic
Mean Absolute Error	0.96793	0.00122
Mean Squared Error	1.25239	0
Root Mean Squared Error	1.1191	0.00143
R^2	0.66624	1
AIC	36.13668	-110.4977
BIC	36.09014	-110.54425
Equation	y2 = 0.49991x + 3.00173	y2 = -0.12671x^2 + 2.78084x + (-5.99573)

Export to plot.ly »

The negative BIC and AIC clearly show that the quadratic fit is the best one.

Notes

We can perform polynomial fitting by using `numpy.polyfit()` (https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.polyfit.html), `numpy.polynomial.polynomial.polyfit()` (https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.polynomial.polynomial.polyfit.html) and scikit-learn (as used here). The numpy functions, with slight differences in their respective returned objects, are useful for straightforward fitting and to obtain the coefficients of the equation/model. However, they are slightly less easier to be used in error calculations and making predictions. In contrast, the scikit-learn is easier in making predictions but slightly more difficult to obtain the model coefficients, as scikit-learn is designed for machine learning and not statistical analysis. Another alternative is the statsmodels library (https://www.statsmodels.org/stable/index.html).

For scikit-learn, the coefficients are given by both `coef_` and `intercept_` objects, related to the predictor variables and the regression constant, respectively. In contrast, `np.polyfit()` objects can be accessed as follows:

In [13]:
```
# full = True returns also the errors
regr_Q2_quad_polyfit = np.polyfit(df_Q2['x2'], df_Q2['y2'], 2, full=False)

# polyfit: Polynomial coefficients, highest power first
# polynomial.polynomial.polyfit: Polynomial coefficients ordered from low to high
regr_Q2_quad_polyfit
```

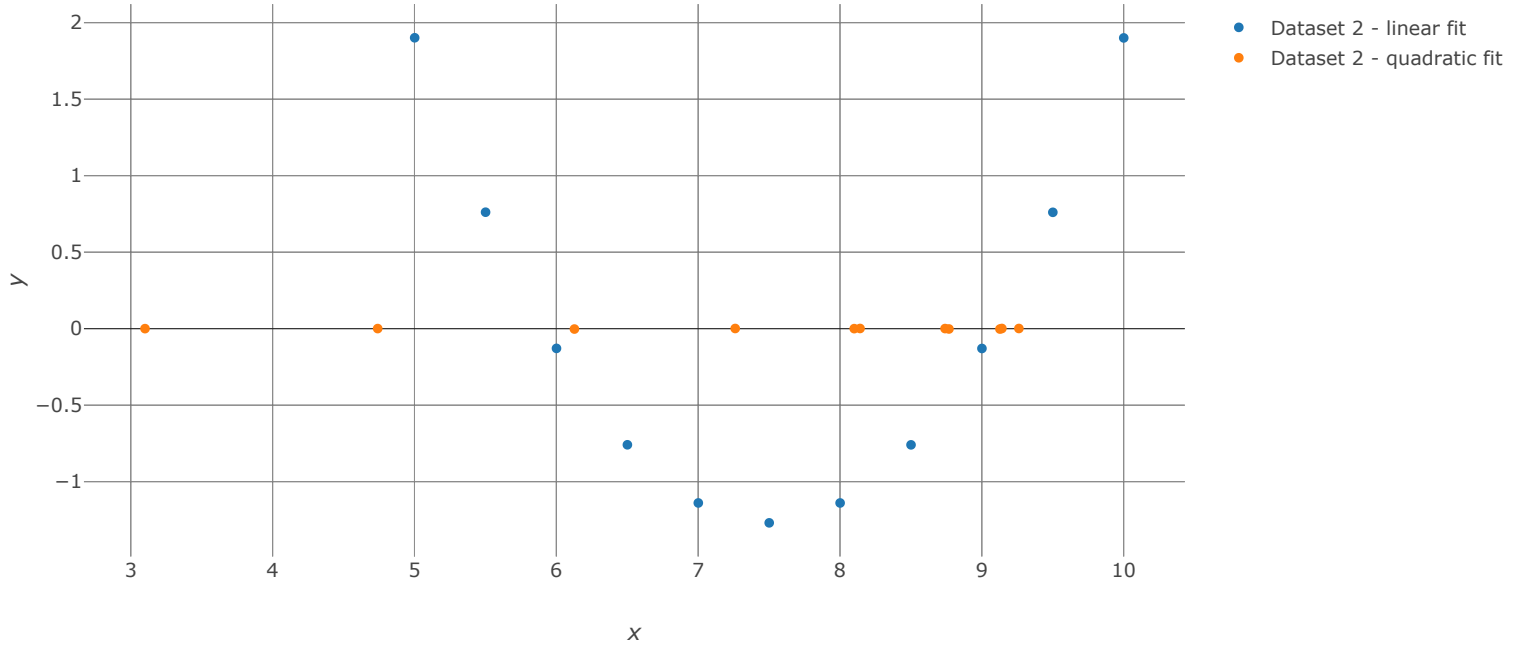
Out[13]:
```
array([-0.12671329,  2.78083916, -5.99573427])
```

Therefore, from the `polyfit` array above, the fit is $y2 = -0.12671x^2 + 2.78084x - 5.99573$, which is equivalent to the equation in Table 4.

Another way to check the goodness of fit is through the residual plot.

In [14]:
```
# calculate the residuals
Q2_residual = Q2_Y_pred - Q2_Y
Q2_residualQuad = Q2_Y_predQuadratic - Q2_Y

trace1 = go.Scatter(x=Q2_Y_pred, y = Q2_residual, mode = 'markers', name = 'Dataset 2 - linear fit')
trace2 = go.Scatter(x=Q2_Y_predQuadratic, y = Q2_residualQuad, mode = 'markers', name = 'Dataset 2 - quadratic fit')

data = [trace1, trace2]

layout= go.Layout(
    title= '<b>Figure 5: Residual plots for Dataset 2 linear and quadratic fits</b>',
    xaxis = dict(title = '<i>x</i>'),
    yaxis = dict(title = '<i>y</i>')
)

fig = go.Figure(data=data, layout=layout)
iplot(fig)
```

Figure 5: Residual plots for Dataset 2 linear and quadratic fits



Export to plot.ly »

Unlike the data points for the linear fit that displays a structure of parabolic curvature, the data points for quadratic fit residuals lie almost perfectly on the $y = 0$ line. Subsequently, we can conclude that indeed, the quadratic fit is better than linear for Dataset 2.

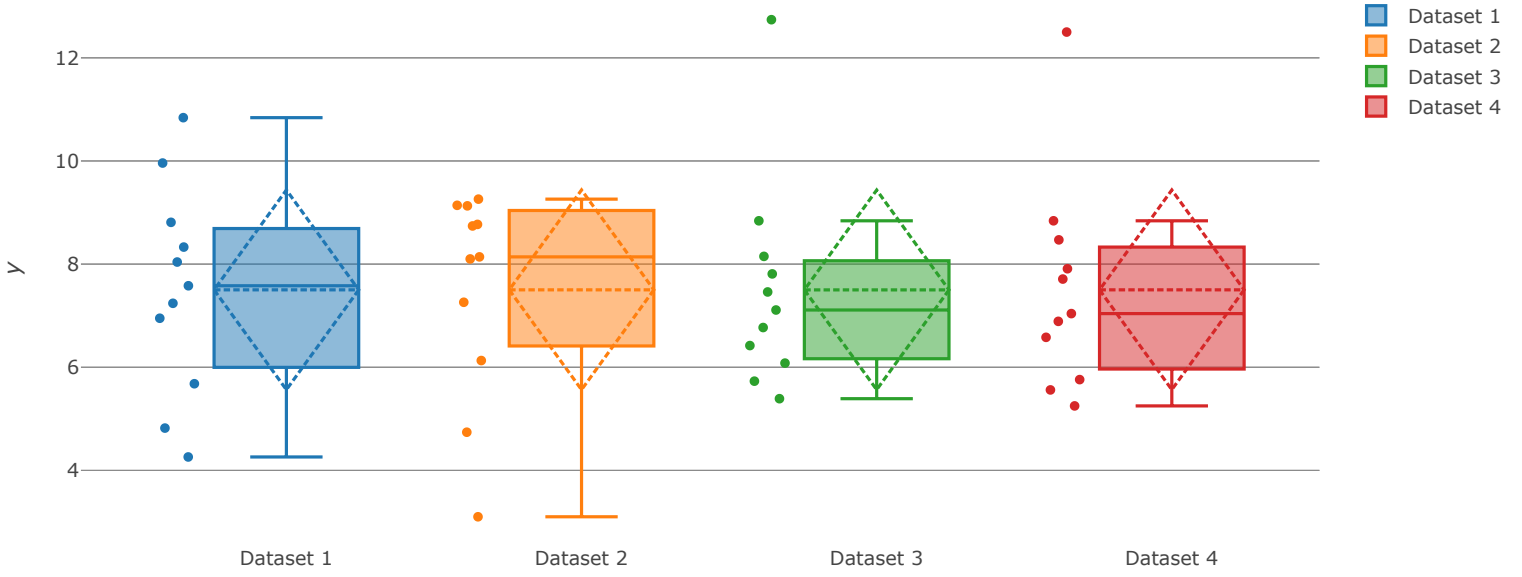
Datasets 3 and 4 - Outlier Detection

The scatter plot in Figure 1 suggests the presence of outliers in Datasets 3 and 4. We can also use box and violin plots as complementary tools to identify potential outliers. For comparative purposes, we will plot the y values of all the datasets side-by-side, together with their respective standard deviation (sd).

In [15]:

```
#https://plot.ly/python/box-plots/

boxQ1 = go.Box(
    y=df_Q1['y1'],
    name = 'Dataset 1',
    boxpoints = 'all',
    boxmean = 'sd'
)

boxQ2 = go.Box(
    y=df_Q2['y2'],
    name = 'Dataset 2',
    boxpoints = 'all',
    boxmean = 'sd'
)

boxQ3 = go.Box(
    y=df_Q3['y3'],
    name = 'Dataset 3',
    boxpoints = 'all',
    boxmean = 'sd'
)

boxQ4= go.Box(
    y=df_Q4['y4'],
    name = 'Dataset 4',
    boxpoints = 'all',
    boxmean = 'sd'
)

data = [boxQ1, boxQ2, boxQ3, boxQ4]

layout= go.Layout(
    title= "<b>Figure 6: Box plots of Anscombe's quartet",
    #xaxis = dict(title = '<i>x</i>'),
    yaxis = dict(title = '<i>y</i>')
)

fig = go.Figure(data=data, layout=layout)
iplot(fig)
```

Figure 6: Box plots of Anscombe's quartet

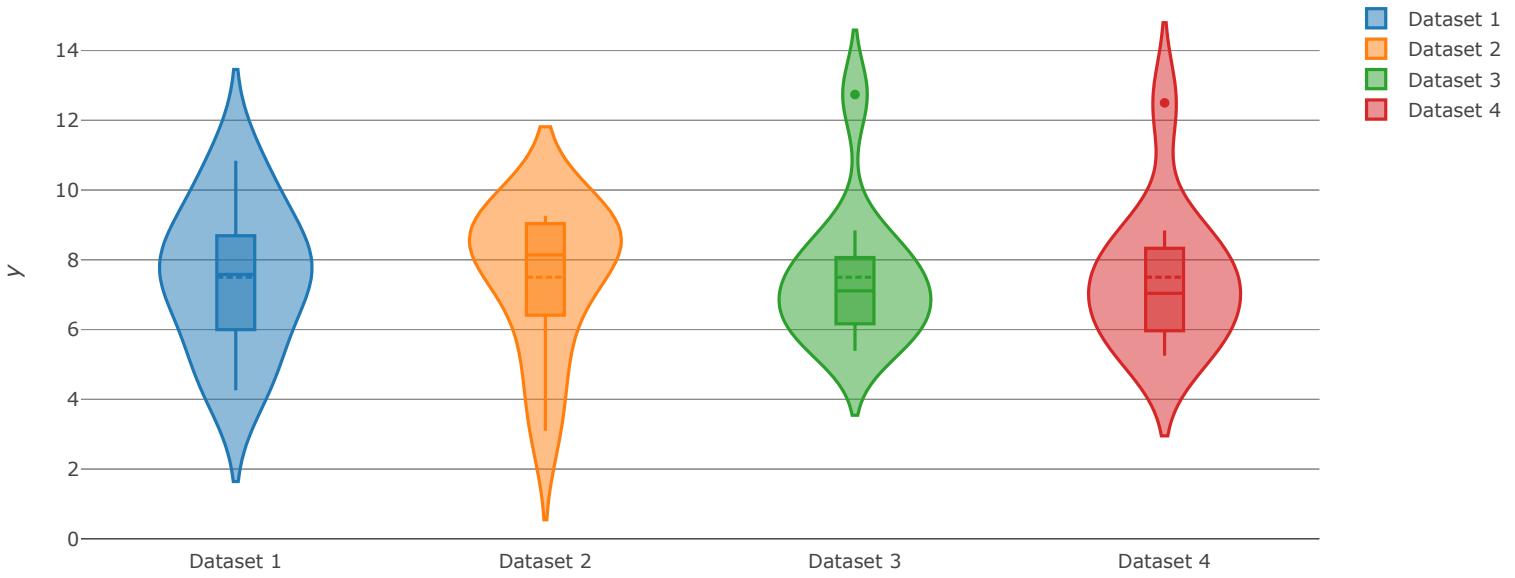


Export to plot.ly »

In [16]:

```
# https://plot.ly/python/violin/#reference

violinQ1 = go.Violin(y = df_Q1['y1'],
                     name = 'Dataset 1',
                     box = dict(visible=True),
                     meanline = dict(visible=True)
                      )

violinQ2 = go.Violin(y = df_Q2['y2'],
                     name = 'Dataset 2',
                     box = dict(visible=True),
                     meanline = dict(visible=True)
                     )

violinQ3 = go.Violin(y = df_Q3['y3'],
                     name = 'Dataset 3',
                     box = dict(visible=True),
                     meanline = dict(visible=True)
                     )

violinQ4 = go.Violin(y = df_Q4['y4'],
                     name = 'Dataset 4',
                     box = dict(visible=True),
                     meanline = dict(visible=True)
                     )

data = [violinQ1, violinQ2, violinQ3, violinQ4]

layout = go.Layout(
    title = "<b>Figure 7: Violin plots of Anscombe's quartet</b>",
    yaxis = dict(title = '<i>y</i>'))

fig = go.Figure(data = data, layout=layout)
iplot(fig)
```

Figure 7: Violin plots of Anscombe's quartet



Export to plot.ly »

Box and violin plots are similar - box plots are more common while violin plots are more informative as they show the kernel density. Looking at these plots, we can make the following interpretations:

1. Dataset 1 has a median of about 8, with the data points equally above and below the median resembling a normal distribution.
2. the y values in Dataset 2 cluster mostly in the range of about 6 to 10, with an extreme point of 3.1 that skews the distribution.
3. in Datasets 3 and 4, we can detect an outlier in each that is significantly beyond the distribution of most points. Typically, outliers are classified as data points beyond 1.5 * interquartile range (IQR).
   A. For Dataset 3, the IQR is 8.065 (third quartile) minus 6.165 (first quartile), which is 1.9. Therefore, most points should lie between $(6.165 - 1.5 * 1.9 = 3.315)$ and $(8.065 + 1.5 * 1.9 = 10.915)$. Therefore, $y3 = 12.74$ is an outlier.
   B. Similarly for Dataset 4, the IQR is 2.365. As most points should lie between 2.4175 and 11.8775, $y4 = 12.5$ is an outlier.

More information on interpreting violin plots can be found at 1 (https://blog.modeanalytics.com/violin-plot-examples/) and 2 (https://en.wikipedia.org/wiki/Violin_plot).

Descriptive Statistics

The descriptive statistics for a pandas dataframe can be easily calculated using the `describe()` function.

In [17]:
```
df.describe()
```

Out[17]:

	x1	y1	x2	y2	x3	y3	x4	y4
count	11.000000	11.000000	11.000000	11.000000	11.000000	11.000000	11.000000	11.000000
mean	9.000000	7.500909	9.000000	7.500909	9.000000	7.500000	9.000000	7.500909
std	3.316625	2.031568	3.316625	2.031657	3.316625	2.030424	3.316625	2.030579
min	4.000000	4.260000	4.000000	3.100000	4.000000	5.390000	8.000000	5.250000
25%	6.500000	6.315000	6.500000	6.695000	6.500000	6.250000	8.000000	6.170000
50%	9.000000	7.580000	9.000000	8.140000	9.000000	7.110000	8.000000	7.040000
75%	11.500000	8.570000	11.500000	8.950000	11.500000	7.980000	8.000000	8.190000
max	14.000000	10.840000	14.000000	9.260000	14.000000	12.740000	19.000000	12.500000

Given the rather 'boring' look of the `describe()` output above, a more presentable table should be created with Plotly.

In [18]:

```
descriptiveStats = df.describe()

trace = go.Table(
    header = dict(values = ['<b>Property</b>', '<b>' + df.columns[0] + '</b>', '<b>' + df.columns[1] + '</b>',
                                               '<b>' + df.columns[2] + '</b>', '<b>' + df.columns[3] + '</b>',
                                               '<b>' + df.columns[4] + '</b>', '<b>' + df.columns[5] + '</b>',
                                               '<b>' + df.columns[6] + '</b>', '<b>' + df.columns[7] + '</b>'],
                  fill = dict(color='darkorchid'),
                  align = ['center'],
                  font = dict(color = 'white')
                  ),

    cells = dict(
        values = [
                    ['<b>Count</b>', '<b>Mean</b>', '<b>Standard deviation</b>', '<b>Minimum</b>',
                     '<b>1st Quartile (25 %)</b>', '<b>Median (50 %)</b>',
                     '<b>3rd Quartile (75 %)</b>', '<b>Maximum</b>'],

                    [df[['x1']].count(), df[['x1']].mean(), df[['x1']].std().round(2), df[['x1']].min(),
                     df[['x1']].quantile(q=0.25), df[['x1']].quantile(q=0.5), df[['x1']].quantile(q=0.75),
                     df[['x1']].max()],

                    [df[['y1']].count(), df[['y1']].mean().round(2), df[['y1']].std().round(2), df[['y1']].min(),
                     df[['y1']].quantile(q=0.25).round(2), df[['y1']].quantile(q=0.5), df[['y1']].quantile(q=0.75),
                     df[['y1']].max()],

                    [df[['x2']].count(), df[['x2']].mean().round(2), df[['x2']].std().round(2), df[['x2']].min(),
                     df[['x2']].quantile(q=0.25).round(2), df[['x2']].quantile(q=0.5), df[['x2']].quantile(q=0.75),
                     df[['x2']].max()],

                    [df[['y2']].count(), df[['y2']].mean().round(2), df[['y2']].std().round(2), df[['y2']].min(),
                     df[['y2']].quantile(q=0.25).round(2), df[['y2']].quantile(q=0.5), df[['y2']].quantile(q=0.75),
                     df[['y2']].max()],

                    [df[['x3']].count(), df[['x3']].mean().round(2), df[['x3']].std().round(2), df[['x3']].min(),
                     df[['x3']].quantile(q=0.25).round(2), df[['x3']].quantile(q=0.5), df[['x3']].quantile(q=0.75),
                     df[['x3']].max()],

                    [df[['y3']].count(), df[['y3']].mean().round(2), df[['y3']].std().round(2), df[['y3']].min(),
                     df[['y3']].quantile(q=0.25).round(2), df[['y3']].quantile(q=0.5), df[['y3']].quantile(q=0.75),
                     df[['y3']].max()],

                    [df[['x4']].count(), df[['x4']].mean().round(2), df[['x4']].std().round(2), df[['x4']].min(),
                     df[['x4']].quantile(q=0.25).round(2), df[['x4']].quantile(q=0.5), df[['x4']].quantile(q=0.75),
                     df[['x4']].max()],

                    [df[['y4']].count(), df[['y4']].mean().round(2), df[['y4']].std().round(2), df[['y4']].min(),
                     df[['y4']].quantile(q=0.25).round(2), df[['y4']].quantile(q=0.5), df[['y4']].quantile(q=0.75).round(2),
                     df[['y4']].max()]
                ],
        fill = dict(color=['crimson', 'cornsilk', 'cornsilk', 'cornsilk', 'cornsilk', 'cornsilk', 'cornsilk', 'cornsilk']),
        align = ['center'],
```

```
        font = dict(color=['white', 'black', 'black', 'black', 'black', 'black', 'black', 'black'])
        ),
    columnwidth = [12,6,6,6,6]
    )


data = [trace]

layout = go.Layout(title = "<b>Table 5: Descriptive Statistics of Anscombe's Quartet</b>")

table = go.Figure(data = data, layout = layout)
iplot(table)
```

Table 5: Descriptive Statistics of Anscombe's Quartet

Property	x1	y1	x2	y2	x3	y3	x4	y4
Count	11	11	11	11	11	11	11	11
Mean	9	7.5	9	7.5	9	7.5	9	7.5
Standard deviation	3.32	2.03	3.32	2.03	3.32	2.03	3.32	2.03
Minimum	4	4.26	4	3.1	4	5.39	8	5.25
1st Quartile (25 %)	6.5	6.32	6.5	6.7	6.5	6.25	8	6.17
Median (50 %)	9	7.58	9	8.14	9	7.11	8	7.04
3rd Quartile (75 %)	11.5	8.57	11.5	8.95	11.5	7.98	8	8.19
Maximum	14	10.84	14	9.26	14	12.74	19	12.5

Export to plot.ly »

Discussion

Anscombe initially created the dataset to demonstrate that no type of numerical value can substitute a graphical representation of a dataset. Table 3 highlights how a simple linear regression fit can be skewed by the presence of outliers and curvature in the data, leading to similar goodness-of-fit values. In practical terms, while a linear fit describes the Dataset 1 adequately, a quadratic fit is required for Dataset 2. For the Datasets 3 and 4, the outlier removal is necessary prior to any fitting or predictive modeling, unless there are strong technical reasons (such as multiple confirmed measurements) to keep them. While it might have been difficult in Anscombe's time to generate multiple complementary plots, there is no reason in modern times to not plot a given dataset first.

Conclusion

Much like a doctor who will use a combination of his stethoscope, charts and scan images rather than relying on any one tool, a good data scientist will articulate his ideas and interpretations cohesively using different plots and regression techniques. This is especially crucial prior to time-consuming predictive modeling, as the algorithms are only as good as the input data - in other words, garbage in equals garbage out.