

5

Visualization, Insights, and Results

After exploring machine learning, but not because the topic is less relevant than others, we are going to illustrate how to create visualizations with Python to enrich your data science project. Visualization plays an important role in helping you communicate the results and insights derived from data and the learning process.

In this chapter, you will learn how to do the following:

- Use the basic `pyplot` functions from the `matplotlib` package
- Leverage a pandas DataFrame for **Explorative Data Analysis (EDA)**
- Create beautiful and interactive charts with Seaborn
- Visualize the machine learning and optimization processes we discussed in Chapter 3, *The Data Pipeline*, and Chapter 4, *Machine Learning*
- Understand and visually communicate variables' importance and their relationship with the target outcome
- Set up a prediction server that uses HTTP to accept and provide predictions as a service

Introducing the basics of matplotlib

Visualization is a fundamental aspect of data science, allowing data scientists to better and more effectively communicate their findings to the organization they operate in, to both data experts and non-experts. Providing the nuts and bolts of the principles behind communicating information and crafting engaging beautiful visualizations is beyond the scope of our book, but we can recommend suitable resources if you want to improve your skills.

For basic visualization rules, you can visit <https://lifehacker.com/5909501/how-to-choose-the-best-chart-for-your-data>. We also recommend the books of Prof. Edward Tufte on analytic design and visualization.

We can instead provide a fast and to-the-point series of essential recipes that can get you started on visualization using Python, and that you can refer to anytime you need to create a specific graphics chart. Consider all the snippets of code as your visualization building blocks; you can arrange them with different configurations and features just by using the large choice of parameters that we are going to present to you.

`matplotlib` is a Python package for plotting graphics. Created by John Hunter, it has been developed in order to address a lack of integration between Python and external software with graphical capabilities, such as MATLAB or gnuplot. Greatly influenced by MATLAB's way of operating and functions, `matplotlib` presents a quite similar syntax. In particular, the `matplotlib.pyplot` module, perfectly compatible with MATLAB, will be the core of our essential introduction to all the indispensable graphical tools to represent your data and analysis. MATLAB is indeed a standard for visualization in the data analysis and scientific community because of its recognized capabilities when it comes to exploratory analysis, mainly due to its smooth and easy to use plotting functions.

Each `pyplot` command makes a change on an initially instantiated figure. Once you set a figure, all additional commands will operate on it. Thus, it is easy to incrementally improve and enrich your graphic representation. In order for you to take advantage of the code and be able to personalize it to your needs, all the following examples are presented together with commented building blocks so that you can later draft your basic representation, and then look through this chapter for specific parameters among the examples in order to improve your chart as you planned it.

With the `pyplot.figure()` command, you can initialize a new visualization, though it suffices to call a plotting command to automatically start it. Instead, by using `pyplot.show()`, you close the figure that you were operating on, and you can open and operate on new figures.

Before starting with a few visualization examples, let's import the necessary packages in order to run all the examples:

```
In: import numpy as np
     import matplotlib.pyplot as plt
     import matplotlib as mpl
```

In this way, we can always refer to `pyplot`, the MATLAB-like module, as `plt`, and access the complete `matplotlib` functionality set with the help of `mpl`.



If you are using a Jupyter Notebook (or Jupyter Lab), you can use this line magic: `%matplotlib inline`. After writing the command in a cell of the notebook and running it, you can have your plots drawn directly on the notebook itself, instead of having the graphics presented in a separate window (by default, the GUI backend of `matplotlib` is the `TkAgg` backend). If you prefer a different backend such as `Qt` (www.qt.io), which is often distributed with Python scientific distributions, you just have to run this line magic instead: `%matplotlib Qt`.

Trying curve plotting

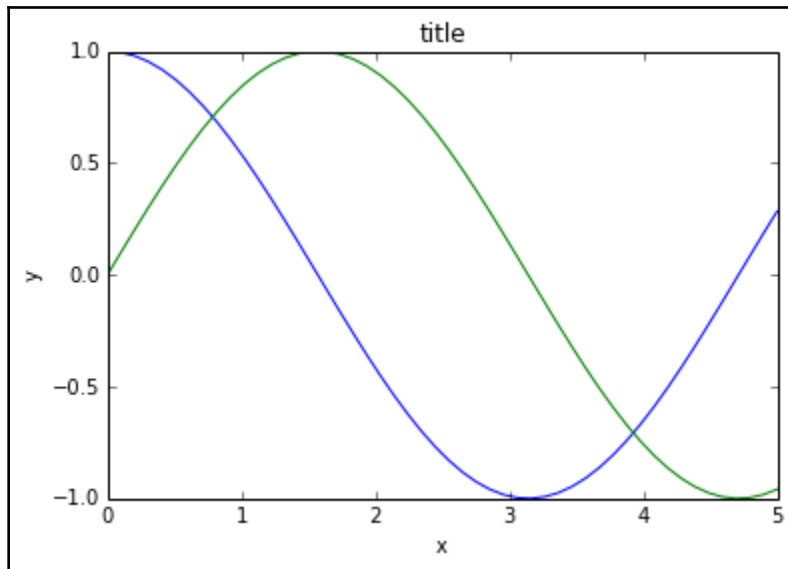
Our first problem will require you to draw a function with `pyplot`. Drawing a function is quite straightforward; you just have to get a series of x coordinates and map them to the y axis by using the function that you want to plot. Since the mapping results are stored away into two vectors, the `plot` function will deal with the curve representation. The precision of the representation will be greater if the mapped points are enough (50 points is a good sampling number):

```
In: import numpy as np
    import matplotlib.pyplot as plt
    x = np.linspace(0, 5, 50)
    y_cos = np.cos(x)
    y_sin = np.sin(x)
```

Using the NumPy `linspace()` function, we will create a series of 50 equally distanced numbers ranging from 0 to 5. We can use them to map our y to the cosine and sine functions:

```
In: plt.figure() # initialize a figure
    plt.plot(x,y_cos) # plot series of coordinates as a line
    plt.plot(x,y_sin)
    plt.xlabel('x') # adds label to x axis
    plt.ylabel('y') # adds label to y axis
    plt.title('title') # adds a title
    plt.show() # close a figure
```

Here is your first plot:



The `pyplot.plot` command can plot more curves in a sequence, with each curve taking a different color according to an internal color schema, which can be customized by explicating the favored color sequence. To do so, you have to manipulate the list containing the sequence of colors that `matplotlib` uses:

```
In: list(mpl.rcParams['axes.prop_cycle'])
```

```
Out: [{'color': '#1f77b4'},  
      {'color': '#ff7f0e'},  
      {'color': '#2ca02c'},  
      {'color': '#d62728'},  
      {'color': '#9467bd'},  
      {'color': '#8c564b'},  
      {'color': '#e377c2'},  
      {'color': '#7f7f7f'},  
      {'color': '#bcbd22'},  
      {'color': '#17becf'}]
```



TIP

`#1f77b4`, `#ff7f0e`, `#2ca02c`, and all the others are all colors expressed in hexadecimal form. In order to figure out how they look, you can use the [colorhexa](https://www.colorhexa.com/) website, providing you with useful information on each of them: <https://www.colorhexa.com/>.

The hack can be done by using the `cycler` function and feeding it with a list of string names referring to the colors you want to use in sequence:

```
In: mpl.rcParams['axes.prop_cycle'] = mpl.cycler('color',
                                                ['blue', 'red', 'green'])
```

Moreover, the `plot` command, if not given any other information, will assume that you are going to plot a line. Therefore, it will link all the provided points in a curve. If you add a new parameter such as `'.'` – that is, `plt.plot(x, y_cos, '.')` – you signal that you instead want to plot a series of separated points (the string for a line is `'-'`, but we will soon show another example).

In this way, if you've customized `rcParams['axes.prop_cycle']` as proposed previously, the next graphs will first have a blue curve, then the second will be red, and the third green. Then, the color loop will restart. We leave this decision to you. All the examples in this chapter will just follow the standard color sequence, but you are free to experiment with better color settings.

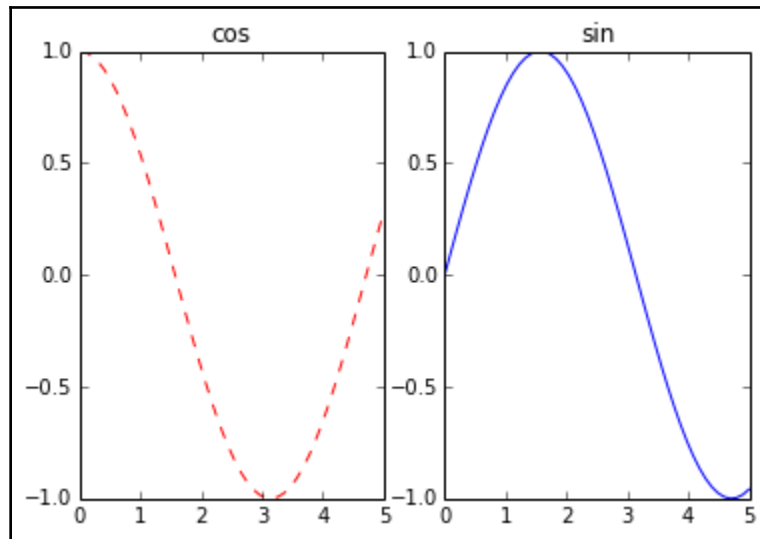
Please note that you can also set the title of the graph and label the axis by the `title`, `xlabel`, and `ylabel` from `pyplot`.

Using panels for clearer representations

Our second example will demonstrate to you how to create multiple graphics panels and plot a representation on each of them. We will also try to personalize the drawn curves by using different colors, sizes, and styles. Here is the example:

```
In: import matplotlib.pyplot as plt
    # defines 1 row 2 column panel, activates figure 1
    plt.subplot(1,2,1)
    plt.plot(x,y_cos,'r--')
    # adds a title
    plt.title('cos')
    # defines 1 row 2 column panel, activates figure 2
    plt.subplot(1,2,2)
    plt.plot(x,y_sin,'b-')
    plt.title('sin')
    plt.show()
```

The plot displays the cosine and sine curves on two distinct graphic panels:



The subplot command accepts the subplot(*nrows*, *ncols*, *plot_number*) parameter form. Therefore, when instantiated, it reserves a certain amount of space for the representation based on the *nrows* and *ncols* parameters and number of plots on the *plot_number* area (starting from area 1 on the left).

You can also accompany the plot command coordinates with another string parameter, which is useful for the definition of color and the type of the represented curve. The strings work by combining the codes that you can find on the following links:

- https://matplotlib.org/api/lines_api.html#matplotlib.lines.Line2D.set_linestyle: Will present the different line styles.
- http://matplotlib.org/api/colors_api.html: Offers a complete overview of the basic built-in colors. The page also points out that you can either use the `color` parameter together with the HTML names or hex strings for colors, or define the color you desire by using an RGB tuple, where each value of the tuple lies in the range of $[0,1]$. For instance, a valid parameter is `color = (0.1, 0.9, 0.9)`, which will create a color made of 10% red, 90% green, and 90% blue.
- http://matplotlib.org/api/markers_api.html: Lists all the possible marker styles you can adopt for your points.

Plotting scatterplots for relationships in data

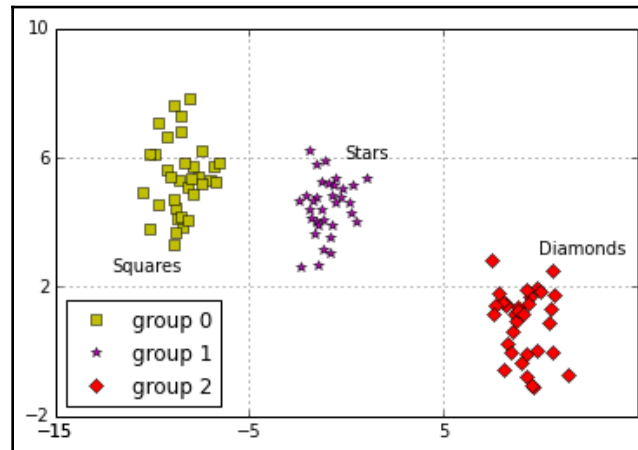
Scatterplots plot two variables as points on a plane, and they can help you figure out the relationship between the two variables. They are also quite effective if you want to represent groups and clusters. In our example, we will create three data clusters and represent them in a scatterplot with different shapes and colors:

```
In: from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
D = make_blobs(n_samples=100, n_features=2,
               centers=3, random_state=7)
groups = D[1]
coordinates = D[0]
```

Since we have to plot three different groups, we will have to use three distinct `plot` commands. Each command specifies a different color and shape (the 'ys', 'm*', 'rD' strings, where the first letter is the color and the second is the marker). Please also note that each plot instance is marked by a `label` parameter, which is used to assign a name to the group that has to be reported later in a legend:

```
In: plt.plot(coordinates[groups==0,0],
             coordinates[groups==0,1],
             'ys', label='group 0') # yellow square
plt.plot(coordinates[groups==1,0],
          coordinates[groups==1,1],
          'm*', label='group 1') # magenta stars
plt.plot(coordinates[groups==2,0],
          coordinates[groups==2,1],
          'rD', label='group 2') # red diamonds
plt.ylim(-2,10) # redefines the limits of y axis
plt.yticks([10,6,2,-2]) # redefines y axis ticks
plt.xticks([-15,-5,5,-15]) # redefines x axis ticks
plt.grid() # adds a grid
plt.annotate('Squares', (-12,2.5)) # prints text at coordinates
plt.annotate('Stars', (0,6))
plt.annotate('Diamonds', (10,3))
plt.legend(loc='lower left', numpoints=1)
# places a legend of labelled items
plt.show()
```

The resulting plot will be a scatterplot of the three groups accompanied by their respective labels:



We have also added a legend (`pyplot.legend`), fixed a limit for both the axes (`pyplot.xlim` and `pyplot.ylim`), and precisely explicated the ticks (`plt.xticks` and `plt.yticks`) that had to be put on them by specifying a list of values. Therefore, the grid (`pyplot.grid`) divides the plot exactly into nine quadrants and allows you to have a better idea of where the groups are positioned. Finally, we printed some text pointing out the group names (`pyplot.annotate`).

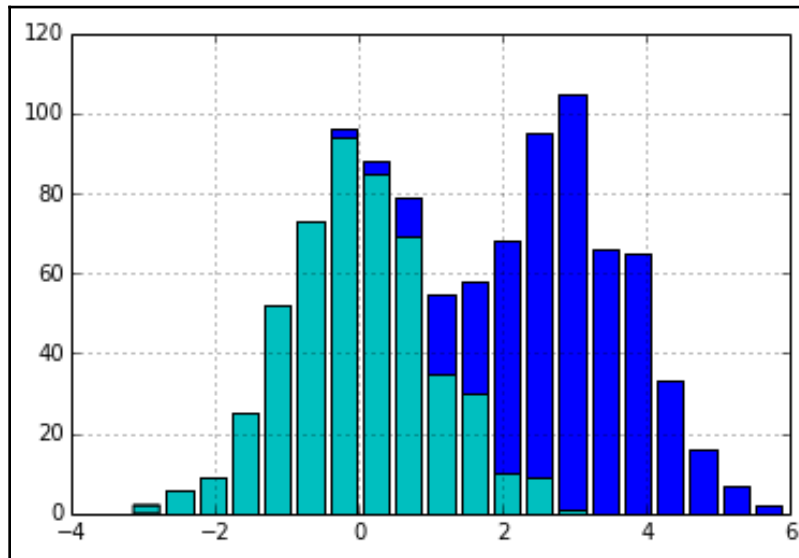
Histograms

Histograms can effectively represent the distribution of a variable. Here, we will visualize two normal distributions, both characterized by unit standard deviation, one having a mean of 0 and the other a mean of 3.0:

```
In: import numpy as np
import matplotlib.pyplot as plt
x = np.random.normal(loc=0.0, scale=1.0, size=500)
z = np.random.normal(loc=3.0, scale=1.0, size=500)
plt.hist(np.column_stack((x,z)),
         bins=20,
         histtype='bar',
         color = ['c', 'b'],
         stacked=True)

plt.grid()
plt.show()
```


The conjoint distributions can offer a different insight on the data if there is a classification problem:



There are a few ways to personalize this kind of plot and obtain further insights about the analyzed distributions. First, by changing the number of bins, you will change how the distributions are discretized (discretization is the process that transforms continuous functions or series of values into a reduced, countable set of numbers: en.wikipedia.org/wiki/Discretization). Generally, 10 to 20 bins offer a good understanding of the distribution, though it really depends on the size of the dataset as well as the distribution. For instance, the Freedman-Diaconis rule prescribes that the optimal number of bins in a histogram in order to meaningfully visualize your data depends on the bin's width, to be calculated using the **interquartile range (IQR)** and the number of observations:

$$h = 2 * IQR * n^{-\frac{1}{3}}$$

Having calculated h , which is the bin width, the number of bins is computed by dividing the difference between the maximum and the minimum value by h :

$$bins = (max - min) / h$$

We can also change the type of visualization from bars to steps by changing the parameters from `histtype='bar'` to `histtype='step'`. By changing the `stacked` Boolean parameter to `False`, the curves won't stack into a unique bar in the parts that overlap, but you will clearly see the separate bars of each one.

Bar graphs

Bar graphs are useful for comparing quantities in different categories. They can be arranged either horizontally or vertically to present the mean estimate and error bands. They can be used to present various statistics of your predictors and how they relate to the target variable.

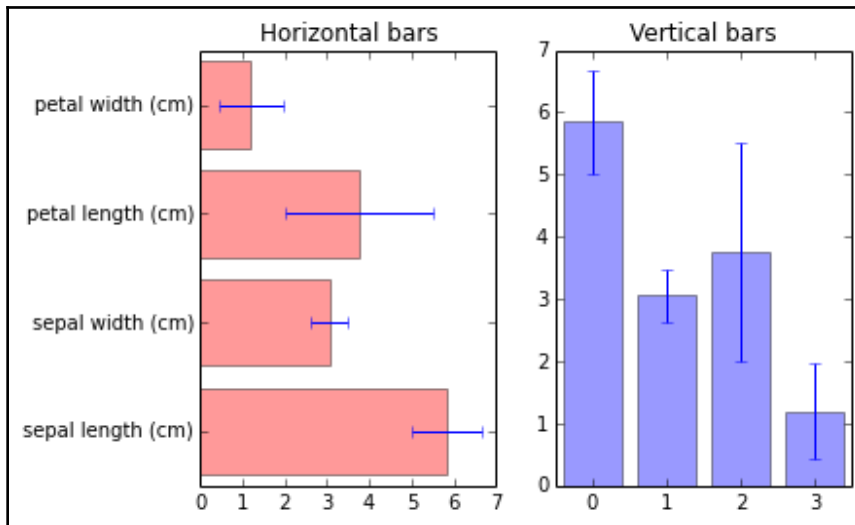
In our example, we will present the mean and standard deviation for the four variables of the Iris dataset:

```
In: from sklearn.datasets import load_iris
import numpy as np
import matplotlib.pyplot as plt
iris = load_iris()
average = np.mean(iris.data, axis=0)
std = np.std(iris.data, axis=0)
range_ = range(np.shape(iris.data)[1])
```

In our representation, we will prepare two subplots: one with horizontal bars (`plt.barh`), and the other with vertical bars (`plt.bar`). The standard error is represented by an error bar, and according to the graph orientation, we can use the `xerr` parameter for horizontal bars and `yerr` for vertical ones:

```
In: plt.subplot(1,2,1) # defines 1 row, 2 columns panel, activates figure 1
plt.title('Horizontal bars')
plt.barh(range_, average, color="r",
         xerr=std, alpha=0.4, align="center")
plt.yticks(range_, iris.feature_names)
plt.subplot(1,2,2) # defines 1 row 2 column panel, activates figure 2
plt.title('Vertical bars')
plt.bar(range_, average, color="b", yerr=std, alpha=0.4, align="center")
plt.xticks(range_, range_)
plt.show()
```

Horizontal and vertical bars are now together in the same plot:



It is important to note the use of the `plt.xticks` command (and of `plt.yticks` for the ordinate axis). The first parameter informs the command about the number of ticks that have to be placed on the axis, and the second one explicates the labels that have to be put on the ticks.

Another interesting parameter to notice is `alpha`, which has been used to set the transparency level of the bar. The `alpha` parameter is a float number ranging from 0.0, fully transparent, to 1.0, which causes the color to be shown in different levels of opaqueness.

Image visualization

The last possible visualization that we explore using `matplotlib` has to do with images. Resorting to `plt.imshow` is useful when you are working with image data. Let's take as an example the Olivetti dataset, an open source set of images of 40 people who provided 10 images of themselves at different times (and with different expressions, a fact that makes it more challenging for testing face recognition algorithms). The images from this dataset are provided as feature vectors of pixel intensities. Therefore, it is important to reshape the vectors in order to make them resemble a matrix of pixels. Setting the interpolation to `'nearest'` helps to smooth the picture:

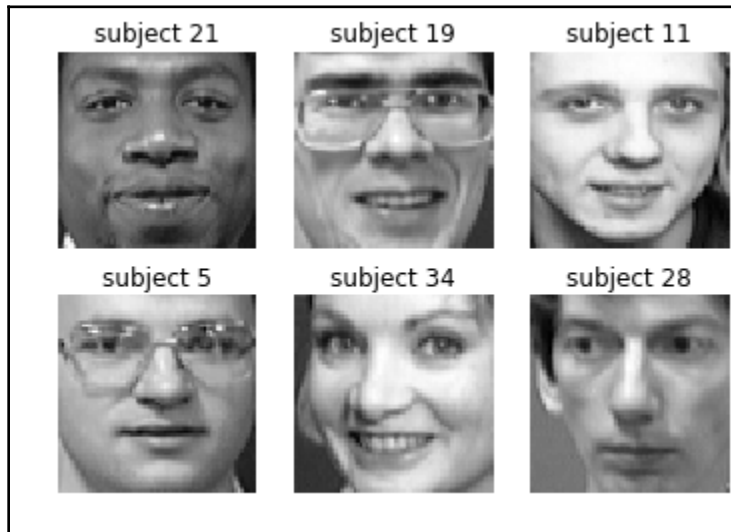
```
In: from sklearn.datasets import fetch_olivetti_faces
import numpy as np
```

```

import matplotlib.pyplot as plt
dataset = fetch_olivetti_faces(shuffle=True, random_state=5)
photo = 1
for k in range(6):
    plt.subplot(2, 3, k+1)
    plt.imshow(dataset.data[k].reshape(64, 64),
               cmap=plt.cm.gray,
               interpolation='nearest')
    plt.title('subject '+str(dataset.target[k]))
    plt.axis('off')
plt.show()

```

A complete panel of images will be plotted:



We can also visualize handwritten digits or letters. In our example, we will plot the first nine digits from the scikit-learn handwritten digit dataset and set the extent of both the axes (by using the `extent` parameter and providing a list of minimum and maximum values) to align the grid to the pixels:

```

In: from sklearn.datasets import load_digits
    digits = load_digits()
    for number in range(1,10):
        fig = plt.subplot(3, 3, number)
        fig.imshow(digits.images[number],
                   cmap='binary',
                   interpolation='none',
                   extent=[0, 8, 0, 8])

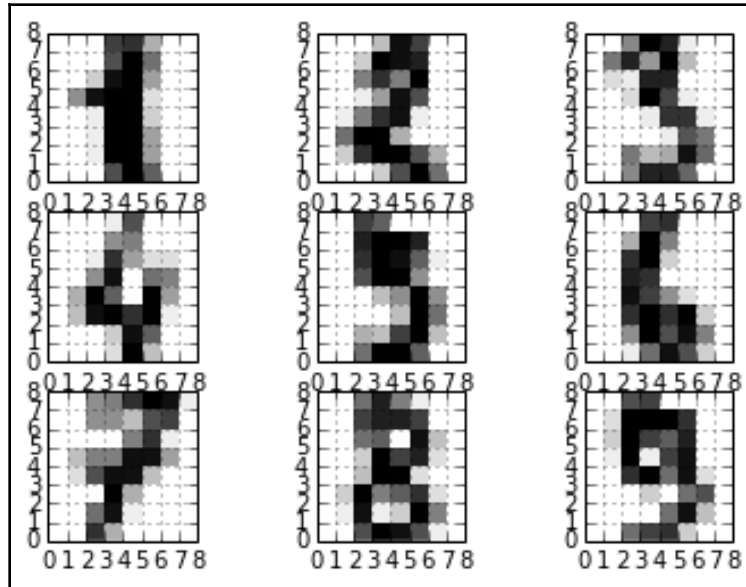
```

```

fig.set_xticks(np.arange(0, 9, 1))
fig.set_yticks(np.arange(0, 9, 1))
fig.grid()
plt.show()

```

A simple close-up on a single number can be obtained by printing only one image:

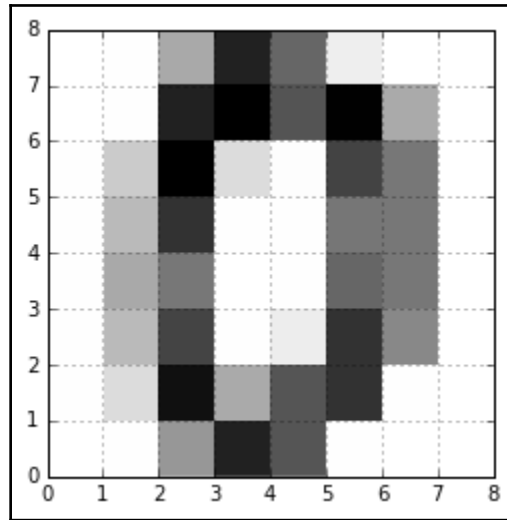


```

In: plt.imshow(digits.images[0],
               cmap='binary',
               interpolation='none',
               extent=[0,8,0,8])
# Extent defines the images max and min
# of the horizontal and vertical values
plt.grid()

```

The resulting image clearly highlights how pixels constitute the image and their gray levels:



Selected graphical examples with pandas

Using appropriately set hyper-parameters, many machine learning algorithms can optimally learn how to map your data with respect to your target outcome. Yet, their predictive performance can be improved further by fixing hidden and subtle problems in data. It is not simply a matter of detecting any missing or outlying case. Sometimes, it is a matter of whether there are any groups or unusual distributions in the data (for instance, multimodal distributions). Clearly drafted data plots can explicate the relationship between variables, and they can lead to the creation of new and better features in order to predict, with increased accuracy, your target variable.

The just-described practice is called **explorative data analysis (EDA)**, and it can bring effective results if it is done accordingly with the following:

- It should be fast, allowing you to explore and develop new ideas, and test them, and restart with a new exploration and fresh ideas
- It should be based on graphical representations in order to better describe data as a whole, no matter how high its dimensionality is

The pandas DataFrame offers many EDA tools that can help you in your explorations. However, first you have to transform your data into a DataFrame:

```
In: import pandas as pd
    print ('Your pandas version is: %s' % pd.__version__)
    from sklearn.datasets import load_iris
    iris = load_iris()
    iris_df = pd.DataFrame(iris.data, columns=iris.feature_names)
    groups = list(iris.target)
    iris_df['groups'] = pd.Series([iris.target_names[k] for k in groups])
```

```
Out: Your pandas version is: 0.23.1
```



Please check your version of pandas. We tested the code in the book under the version 0.23.1 of pandas, and it should also hold for the later releases.

We will be using the `iris_df` DataFrame for all the examples presented in the following paragraphs.

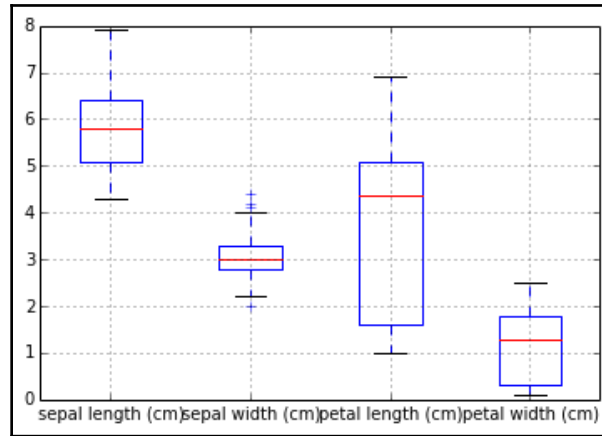
The pandas package actually relies on matplotlib functions for its visualizations. It simply provides a convenient wrapper around the otherwise complex plotting instructions. This offers advantages in terms of speed and simplicity, which are the core values of any EDA process. Instead, if your purpose is to best communicate the findings by using beautiful visualization, you may notice that it is not so easy to customize the pandas graphical outputs. Therefore, when it is paramount to create specific graphics outputs, it is better to start working directly from scratch using matplotlib instructions.

Working with boxplots and histograms

Distributions should always be the first aspect to be inspected in your data. Boxplots draft the key figures in the distribution and help you spot outliers. Just use the `boxplot` method on your DataFrame for a quick overview:

```
In: boxplots = iris_df.boxplot(return_type='axes')
```

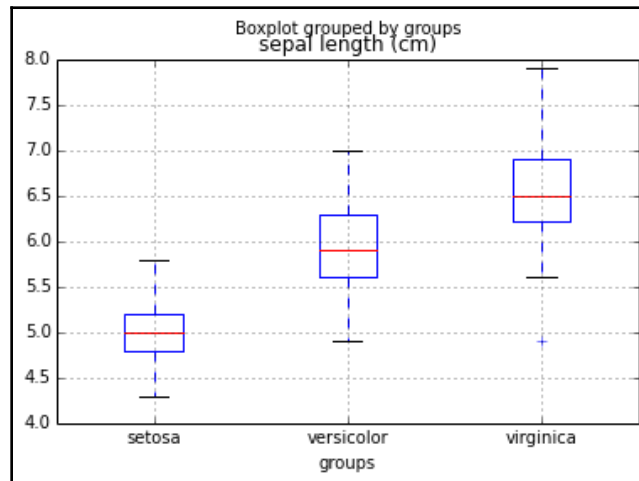
Here are the boxplots of all the numeric variables of the dataset:



If you already have groups in your data (from categorical variables, or derived from unsupervised learning), just point out the variable you need data to be represented in the boxplot and specify that you need to have it separated by the groups (use the `by` parameter followed by the string name of the grouping variable):

```
In: boxplots = iris_df.boxplot(column='sepal length (cm)',  
                               by='groups',  
                               return_type='axes')
```

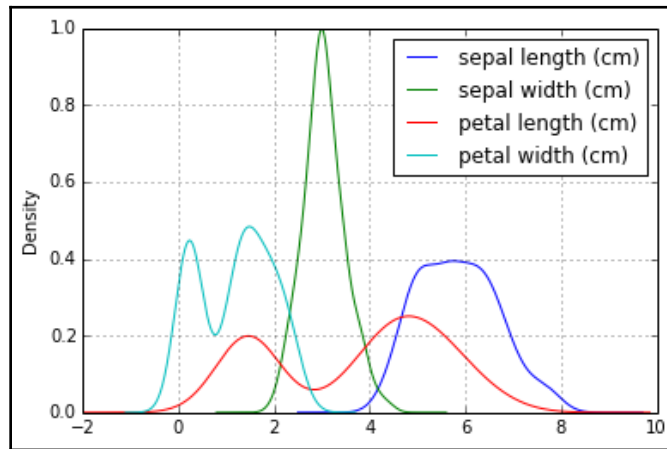
After running the code, you will get the boxplot by groups:



In this way, you can quickly know whether the variable is a good discriminator of the group differences. Anyway, boxplots cannot provide you with a complete view of distributions as histograms and density plots. For instance, by using histograms and density plots, you can figure out whether there are distribution peaks or valleys:

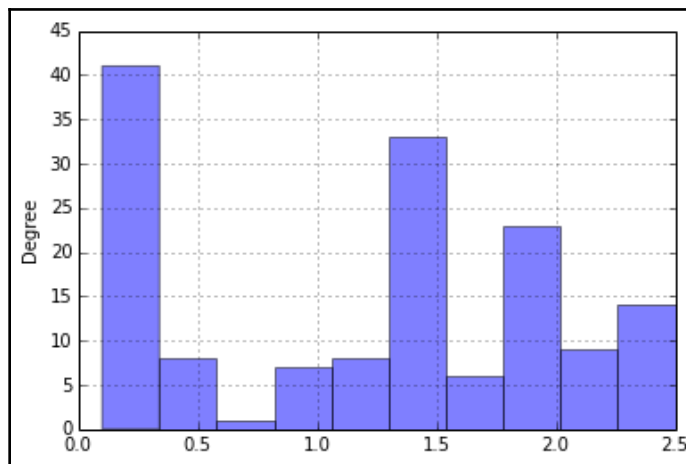
```
In: densityplot = iris_df.plot(kind='density')
```

The code prints the distributions for all the numeric variables of the dataset:



```
In: single_distribution = iris_df['petal width (cm)'].plot(kind='hist',  
alpha=0.5)
```

Here is the resulting distribution represented by a histogram:



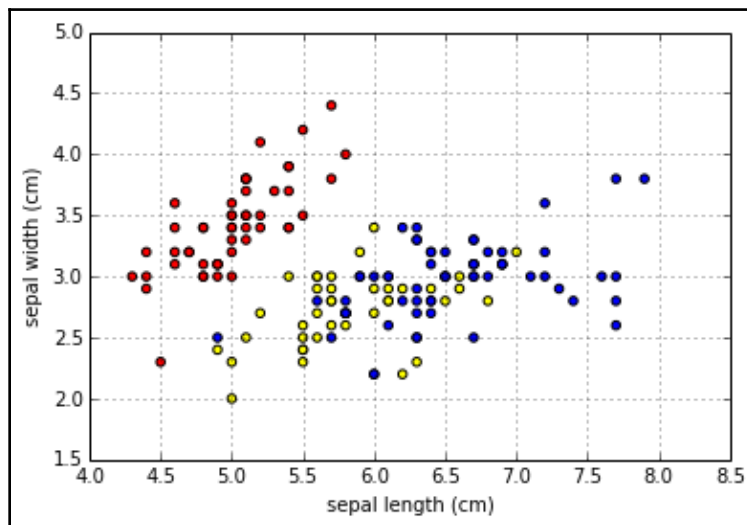
You can obtain both histograms and density plots by using the plot method. This method allows you to represent the whole dataset, specific groups of variables (you just have to provide a list of the string names and do some fancy indexing), or even single variables.

Plotting scatterplots

Scatterplots can be used to effectively understand whether the variables are in a nonlinear relationship, and you can get an idea about their best possible transformations to achieve linearization. If you are using an algorithm based on linear combinations, such as linear or logistic regression, figuring out how to render their relationship more linearly will help you achieve a better predictive power:

```
In: colors_palette = {0: 'red', 1: 'yellow', 2: 'blue'}
   colors = [colors_palette[c] for c in groups]
   simple_scatterplot = iris_df.plot(kind='scatter', x=0, y=1, c=colors)
```

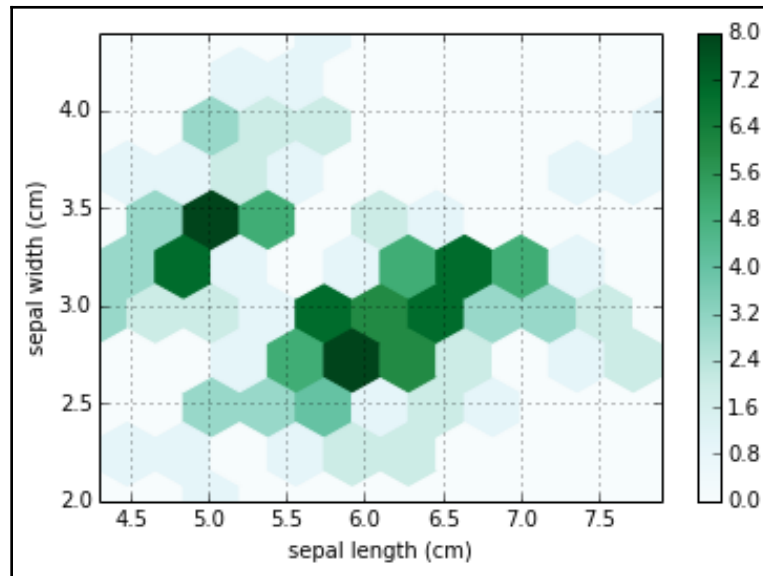
After running the code, a nicely drawn scatterplot will appear:



Scatterplots can be turned into hexagonal binning plots. In addition, they help you effectively visualize the point densities, where the points naturally aggregate together more, thus revealing clusters hidden in your data. For achieving such results, you may use some of the variables originally present in the dataset, or the dimensions obtained by a PCA or by another dimensionality reduction algorithm:

```
In: hexbin = iris_df.plot(kind='hexbin', x=0, y=1, gridsize=10)
```

Here is the resulting `hexbin` plot:

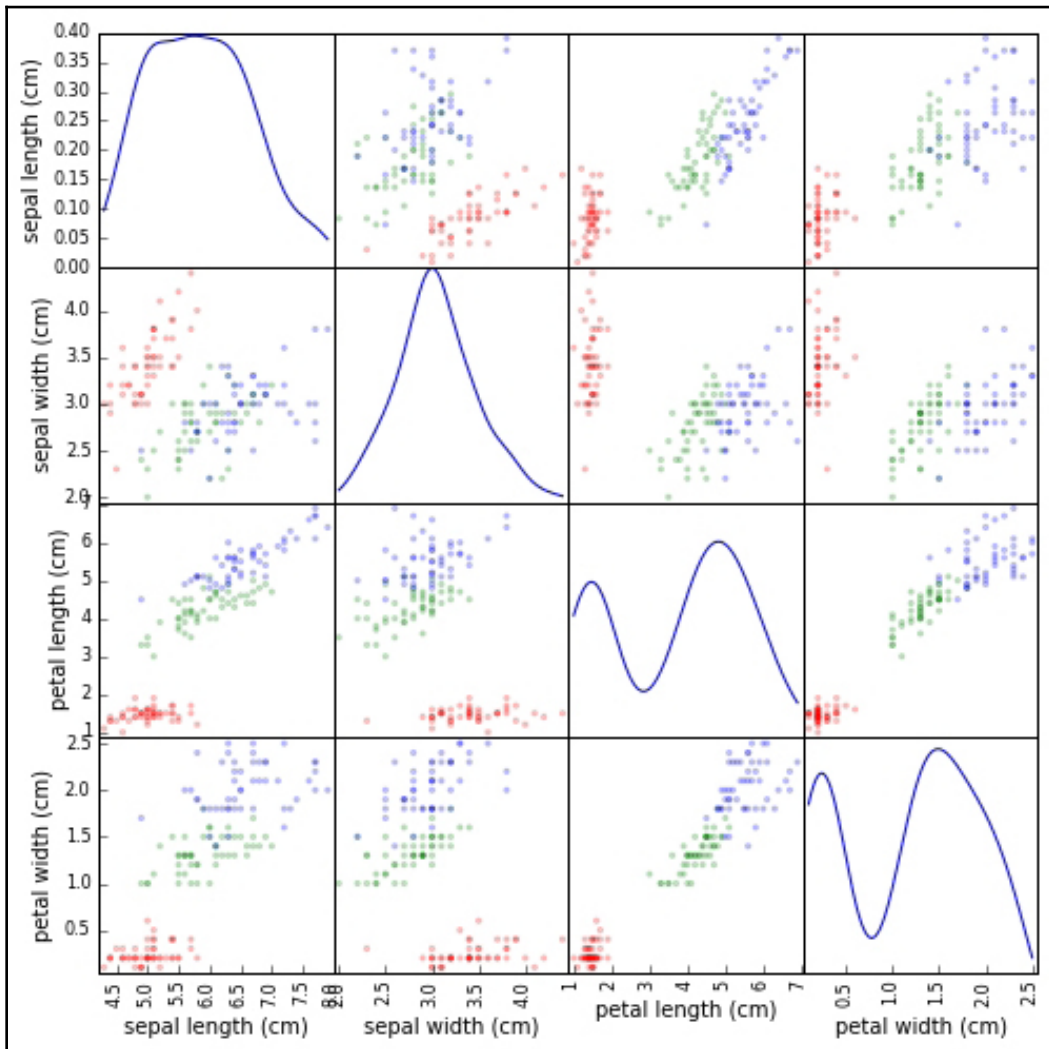


The `gridsize` parameter indicates how many data points the chart will summarize in a single grid. A larger number will create large grid cells, whereas a smaller one will create small cells.

Scatterplots are bivariate. Consequently, you'll require a single plot for every variable combination. If your variables are not so many in number (otherwise, the visualization will be cluttered), a quick solution is to use the `pandas` command to draw a matrix of scatterplots automatically (using the kernel density estimation, `'kde'`, in order to plot the distribution of each feature on the diagonal of the chart):

```
In: from pandas.plotting import scatter_matrix
    colors_palette = {0: "red", 1: "green", 2: "blue"}
    colors = [colors_palette[c] for c in groups]
    matrix_of_scatterplots = scatter_matrix(iris_df,
                                           alpha=0.2,
                                           figsize=(6, 6),
                                           color=colors,
                                           diagonal='kde')
```

After running the previous code, you will get a complete matrix of plots (densities on the diagonal):



A few parameters can control various aspects of the scatterplot matrix. The `alpha` parameter controls the amount of transparency, and `figsize` provides the width and height of the matrix in inches. Finally, `color` accepts a list indicating the color of each point in the plot, thus allowing the depicting of different groups in data. In addition, by selecting 'kde' or 'hist' on your `diagonal` parameter, you can opt to represent density curves or histograms of each variable on the diagonal of the scatter matrix.

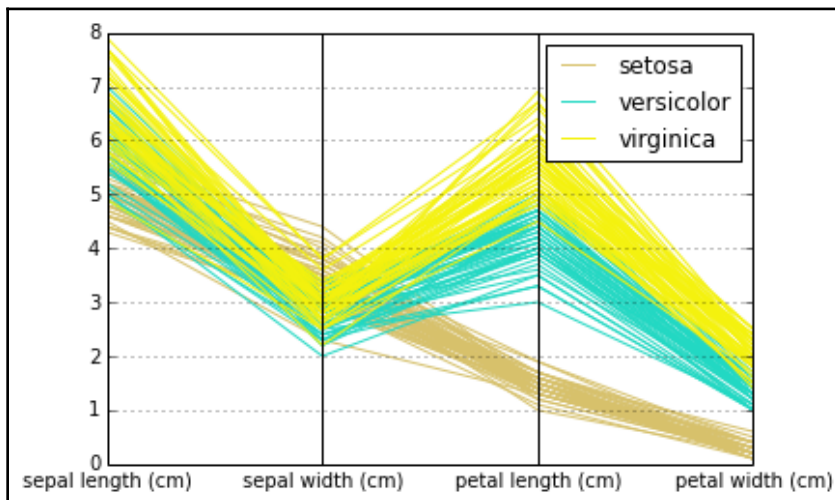
Discovering patterns by parallel coordinates

The scatterplot matrix can inform you about the conjoint distributions of your features. It helps you locate groups in data and verify whether they are distinguishable. Parallel coordinates are another kind of plot that is helpful in providing you with a hint about the most group-discriminating variables present in your data.

By plotting all the observations as parallel lines with respect to all the possible variables (arbitrarily aligned on the abscissa), parallel coordinates will help you spot whether there are streams of observations grouped as your classes, and understand the variables that best separate the streams (the most useful predictor variables). Naturally, in order for the chart to be meaningful, the features in the plot should have the same scale (otherwise, normalize them) as in the Iris dataset:

```
In: from pandas.tools.plotting import parallel_coordinates
    pll = parallel_coordinates(iris_df, 'groups')
```

The previous code will output the parallel coordinates:



`parallel_coordinates` is a pandas function that, in order to work properly, just needs as parameters the data DataFrame and the string name of the variable containing the groups whose separability you want to test. For this reason, you should have the group variable available in your dataset. However, don't forget to remove it after you finish exploring by using the `DataFrame.drop('variable name', axis=1, inplace=True)` method.

Wrapping up matplotlib's commands

As we have seen in the previous paragraph, pandas can speed up exploring data visually since it wraps up into single commands what would have required an entire code snippet using matplotlib. The idea behind this is that unless you need to tailor and configure a special visualization, using a wrapper can allow you to create standard graphics faster.

Apart from pandas, other packages assemble low-level instructions from matplotlib into more user-friendly commands for specific representations and usage:

- Seaborn is a package that extends your visualization capabilities by providing you with a set of statistical plots useful for finding out trends and discriminating groups
- `ggplot` is a port of a popular R library, `ggplot2` (ggplot2.tidyverse.org), based on the visualization grammar proposed in Leland Wilkinson's book, *Grammar of Graphics*. The R library is continuously developed and it offers much functionality; the Python porting (ggplot.yhathq.com) features the basics (ggplot.yhathq.com/docs/index.html) and its complete development is still underway (github.com/yhat/ggplot).
- `MPLD3` (mpld3.github.io) leverages the JavaScript library for graphics manipulation, `D3.js`, in order to easily transform any matplotlib output into HTML code, which can be rendered using a browser and a tool such as a Jupyter Notebook; or within an internet website.
- `Bokeh` (bokeh.pydata.org/en/latest/) is an interactive visualization package that leverages JavaScript and browser-rendered outputs. It is a great replacement for `D3.js` since you just need Python in order to leverage the capabilities of JavaScript to quickly represent your data in an interactive way.

In the following pages, we will introduce Seaborn, providing some building blocks for leveraging their visualizations in your data science projects.

Introducing Seaborn

Created by Michael Waskom and hosted on the PyData website (<http://seaborn.pydata.org/>), Seaborn is a library that wraps up the low-level matplotlib with the entire pyData stack, allowing integrating charts with data structures from NumPy and pandas, and with statistical routines from SciPy and StatModels. All that is achieved with a particular care to aesthetics, thanks to built-in themes, and to color palettes especially devised to reveal patterns in data.

If you don't have Seaborn presently installed on your system (the Anaconda distributions provide it by default, for instance), you can easily get it both by `pip` and `conda` (reminding you that the `conda` version may lag behind the `pip` version taken directly from PyPI, the Python Package Index).

```
$> pip install seaborn
$> conda install seaborn
```

In these examples, we have used version 0.9 of the Seaborn package.

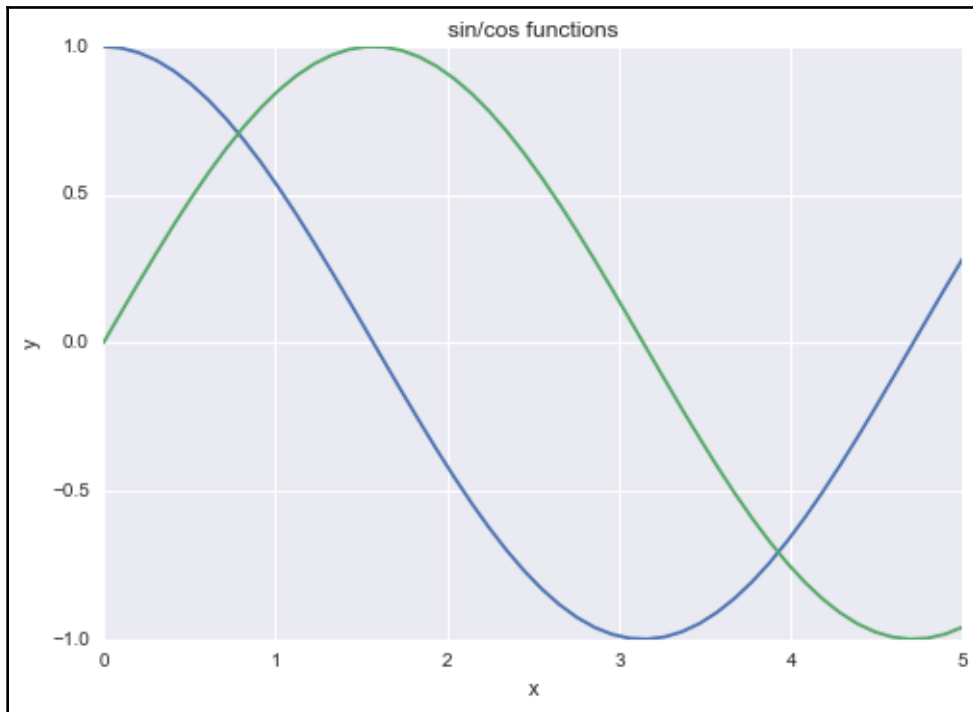
You can upload the package and set the Seaborn style as the default matplotlib style by the following:

```
In: import seaborn as sns
     sns.set()
```

This is enough to turn all your matplotlib-based representations into more visually appealing charts:

```
In: x = np.linspace(0, 5, 50)
     y_cos = np.cos(x)
     y_sin = np.sin(x)
     plt.figure()
     plt.plot(x, y_cos)
     plt.plot(x, y_sin)
     plt.xlabel('x')
     plt.ylabel('y')
     plt.title('sin/cos functions')
     plt.show()
```

Here is the result:



You can obtain interesting results from any of the previously seen charts, even the ones generated using graphical methods in pandas (after all, pandas also relies on matplotlib for creating its explorative plots).

There are five preset themes in Seaborn:

- darkgrid
- whitegrid
- dark
- white
- ticks

`darkgrid` is the default one. You can easily try each one by using the `set_style` command and the name of your preferred theme, and then running your plot commands:

```
In: sns.set_style('whitegrid')
```

All you have to do is just decide which theme helps you better convey the information on your chart. You can limit a style to a single representation enclosing it:

```
In: with sns.axes_style('whitegrid'):  
     # Your plot commands here  
     pass
```

Other stylish changes may involve the spines, which are the borders of the chart. Using the `despine` command, you can easily remove the top and right borders:

```
In: sns.despine()
```

Moreover, you can remove the left border using the `left=True` parameter, offset the axis using the `offset` parameter, and trim it (using `trim=True`). All these operations were otherwise not so accessible because of matplotlib commands alone.

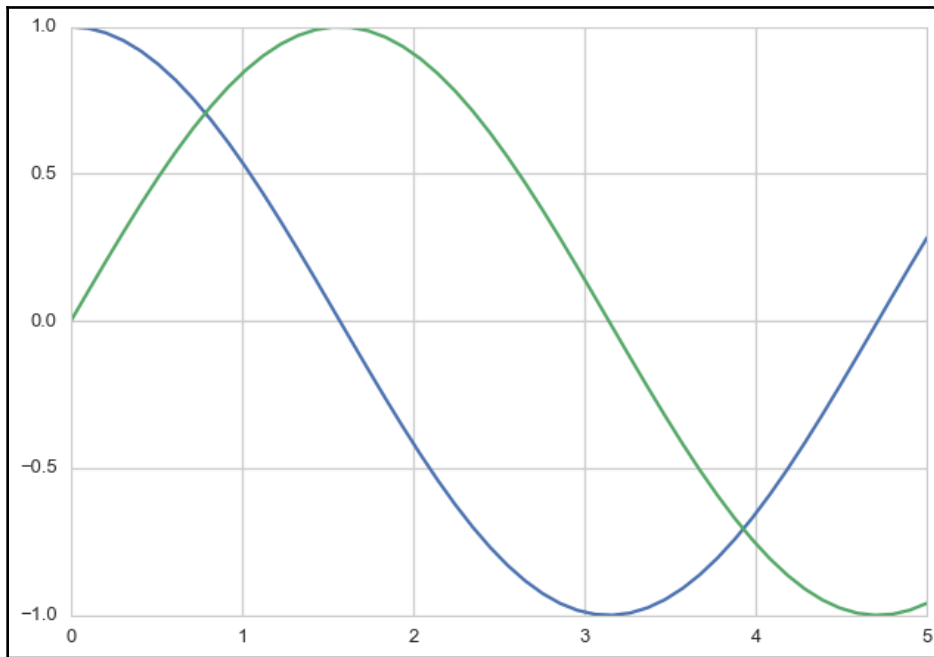
Another useful control that Seaborn permits you regards the scale of the chart. A certain chart scale (involving different thickness of lines, size of fonts, and so on) is called a context, and the available contexts are `self-explicative-paper`, `notebook`, `talk`, and `poster` as possible options. For instance, if your chart has to be displayed on an MS PowerPoint presentation, just run the following command before creating the graphics:

```
In: sns.set_context("talk")
```

Let's see an example of some of such stylish effects on our initial sin/cos chart:

```
In: sns.set_context("talk")  
     with sns.axes_style('whitegrid'):  
         plt.figure()  
         plt.plot(x, y_cos)  
         plt.plot(x, y_sin)  
         plt.show()  
     sns.set()
```

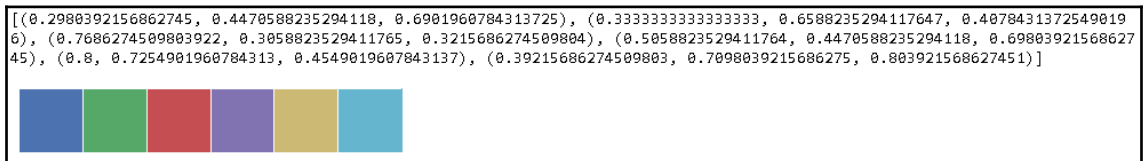
The code will plot the following chart:



Also, choosing the right color cycle or set may help your graphical representation shine. For this, Seaborn offers the `color_palette()` command, which won't just tell the current palette's RGB values (if run with no parameters); it will also accept the name of any palette offered by Seaborn or any matplotlib colormap. It even accepts custom lists of colors provided by you in any matplotlib format (RGB tuples, hex color codes, or HTML color names) in order to create your own palette:

```
In: current_palette = sns.color_palette()
    print (current_palette)
    sns.palplot (current_palette)
```

After running the code, you will visualize the current palette both in values and colors:



There are a few palettes available, as mentioned. First, all Seaborn palettes are the following:

- deep
- muted
- bright
- pastel
- dark
- colorblind

You also have to add `hls`, `husl`, and all the `matplotlib` colormaps, which can be reversed by appending `_r` to their name, or made darker by appending `_d`.



Both the names and examples of `matplotlib` colormaps can be found at this web page: http://matplotlib.org/examples/color/colormaps_reference.html.

The `hls` color space is an automatic transformation in the RGB scale of values, which may or may not work for your representations since colors have different intensities (for instance, yellow and green colors are perceived as brighter whereas blue is perceived as darker).

As an alternative to `hsl`, you can use the `husl` palette, which is friendlier for the human eye, as explained by <http://www.hsluv.org/>.

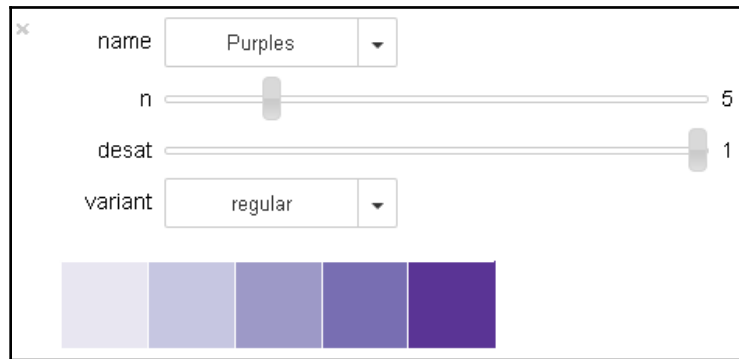
Finally, you can just create a personalized palette using the Color Brewer tool, which can be both found online (http://www.personal.psu.edu/cab38/ColorBrewer/ColorBrewer_intro.html) or required in an app from your Jupyter Notebook. In a notebook cell, using the `choose_colorbrewer_palette` command will make an interactive tool appear. For everything to work, it is essential that you specify as a parameter the `data_type`, a string explicating the nature of your palette related to the data you intend to represent:

- **Sequential** if you want to represent continuity
- **Diverging** for representing contrasts
- **Qualitative** when you just want to discriminate between different classes

Let's see how to create a custom sequential palette, and use it:

```
In: your_palette = sns.choose_colorbrewer_palette('sequential')
```

A complete dashboard will appear:



After setting the colors, `your_palette` will turn into a list of the RGB values:

```
In: print(your_palette)
```

```
Out: [(0.91109573770971852, 0.90574395025477683, 0.94832756940056306),
      (0.7764706015586853, 0.77908498048782349, 0.88235294818878174),
      (0.61776242186041452, 0.60213766261643054, 0.78345253116944269),
      (0.47320263584454858, 0.43267974257469177, 0.69934642314910889),
      (0.35681661753093497, 0.20525952297098493, 0.58569783322951374)]
```

When you are done with your choice, you can just call `sns.set_palette(your_palette)` and have the colors used when drawing all your charts.

If you need just to operate on a chart with some specific colors, using a `with` statement and nesting the chart snippet under it will suffice, as we have seen for the themes before. Instead, if you definitely need to set a certain palette, use `set_palette`.

The color palette is made up of six colors, helping you distinguish at least six trends or classes. If you need to distinguish more, you simply can operate with the `hls` palette and point out the number of colors you need to cycle:

```
In: new_palette=sns.color_palette('hls', 10)
     sns.palplot(new_palette)
```

Here is the resulting palette:



Finally, closing our section about themes and colors, since Seaborn is another, smarter way to use functions offered by matplotlib, we remind you that the resulting charts can be modified further using any basic command coming from matplotlib itself. Or, they can be further transformed by packages such as MPLD3 or Bokeh into JavaScript.

Enhancing your EDA capabilities

Seaborn doesn't just make your charts more beautiful and easily controlled in their aspect; it also provides you with new tools for EDA that helps you discover distributions and relationships between variables.

Before proceeding, let's reload the package and have both the Iris and Boston datasets ready in pandas DataFrame format:

```
In: import seaborn as sns
     sns.set()

     from sklearn.datasets import load_iris
     iris = load_iris()
     X_iris, y_iris = iris.data, iris.target
     features_iris = [a[:-5].replace(' ', '_') for a in iris.feature_names]
     target_labels = {j: flower \
                     for j, flower in enumerate(iris.target_names)}
     df_iris = pd.DataFrame(X_iris, columns=features_iris)
     df_iris['target'] = [target_labels[y] for y in y_iris]

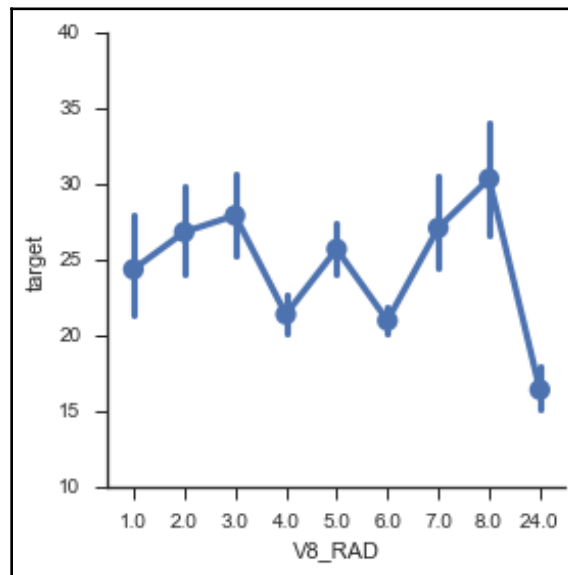
     from sklearn.datasets import load_boston
     boston = load_boston()
     X_boston, y_boston = boston.data, boston.target
     features_boston = np.array(['V'+'_'.join([str(b), a])
                                for a,b in zip(boston.feature_names,
                                                range(len(boston.feature_names))])])
     df_boston = pd.DataFrame(X_boston, columns=features_boston)
     df_boston['target'] = y_boston
     df_boston['target_level'] = pd.qcut(y_boston, 3)
```

As for as the Iris dataset, the target variable has been converted into a descriptive text of the Iris species. For the Boston dataset, the continuous target variable, the median value of owner-occupied homes, has been divided into three equal parts, representing lower, median, and high prices (using the pandas function, `qcut`).

Seaborn can first help your data exploration with figuring out how discretely valued or categorical variables are related to numeric ones. This is achieved using the `catplot` function:

```
In: with sns.axes_style('ticks'):
     sns.catplot(data=df_boston, x='V8_RAD', y='target', kind='point')
```

You will find it insightful exploring similar plots, since they explicit the target level and its variance:

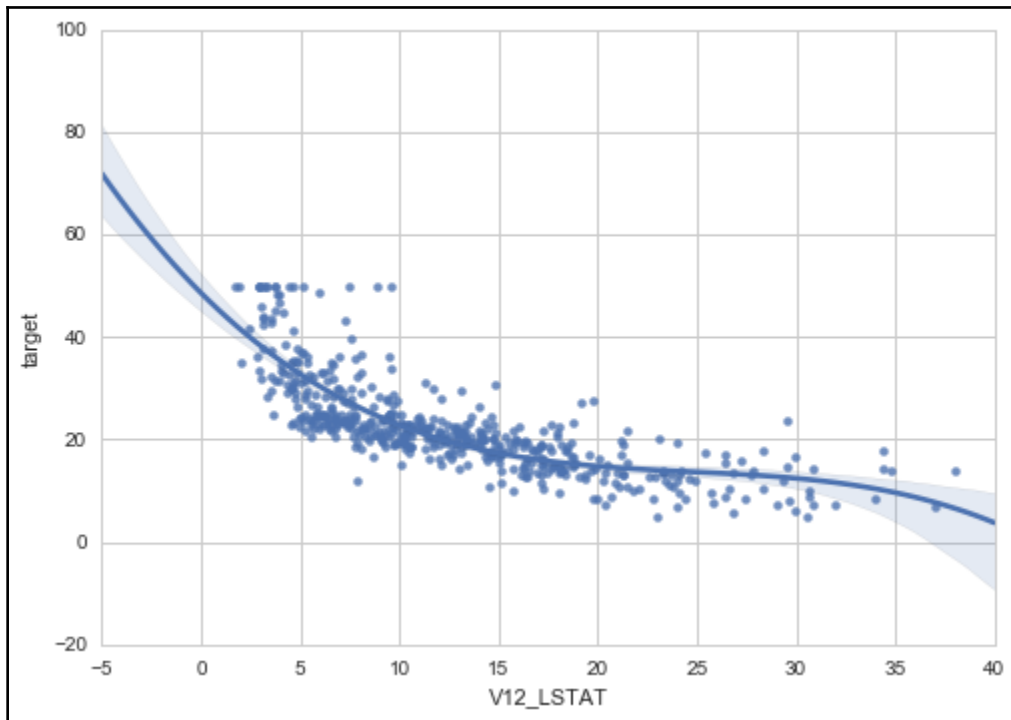


In our example, in the Boston dataset, the index of accessibility to radial highways, which is discretely valued, is compared with the target in order to check both the functional form of its relationships and the associated variance at each level.

In the case, instead, the comparison is between numeric variables; Seaborn offers an enhanced scatterplot with a regression fitted curve trend incorporated, which can clue you in to possible data transformations when the relationship is not linear:

```
In: with sns.axes_style("whitegrid"):
     sns.regplot(data=df_boston, x='V12_LSTAT', y="target", order=3)
```

The fitting line is promptly displayed:

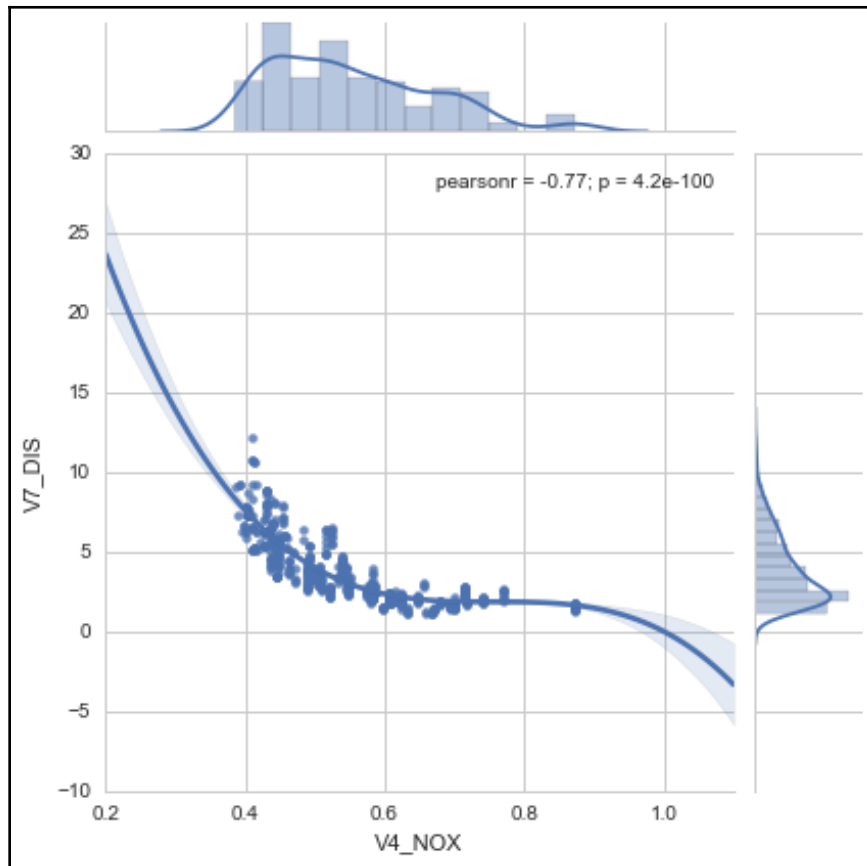


`regplot` in Seaborn can visualize regression plots of any order (we displayed a second-degree polynomial fit). Among the available regression plots, you can use a standard linear regression, a robust regression or even a logistic regression if one of the inspected features is binary.

Where it is necessary to consider distributions too, `jointplot` will provide additional plots on the side of the scatterplot:

```
In: with sns.axes_style("whitegrid"):  
     sns.jointplot("V4_NOX", "V7_DIS",  
                  data=df_boston, kind='reg',  
                  order=3)
```

`jointplot` produces the following chart:

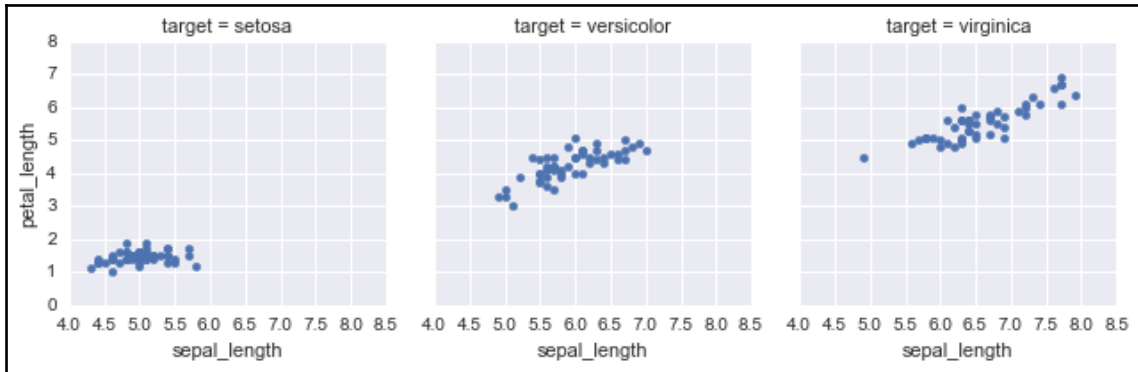


Ideal for representing bivariate relationships by acting on the `kind` parameter, `jointplot` can also represent simple scatterplots or densities (`kind=scatter` or `kind=kde`).

When the purpose is to discover what discriminates classes, `FacetGrid` can arrange different plots in a comparable way and help you understand where there are differences. For instance, we can inspect the scatterplot of Iris species in order to figure out whether they occupy different parts of the feature state:

```
In: with sns.axes_style("darkgrid"):  
    chart = sns.FacetGrid(df_iris, col="target_level")  
    chart.map(plt.scatter, "sepal_length", "petal_length")
```

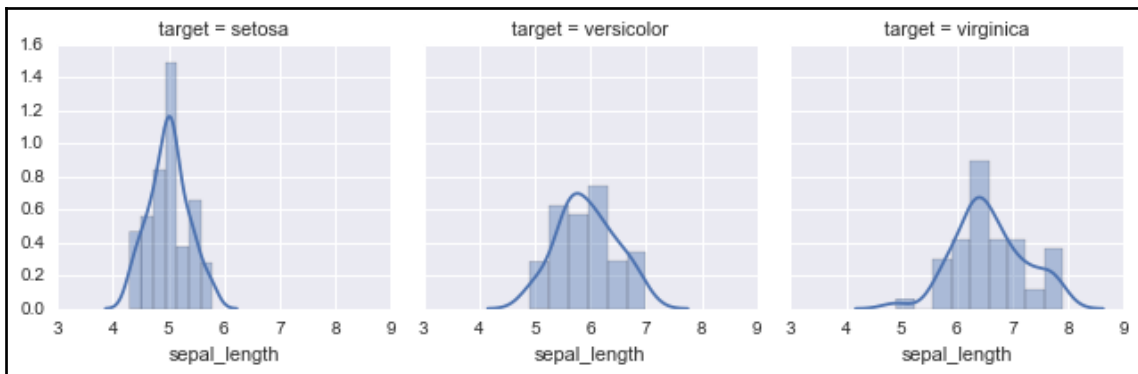

The code will nicely print a panel representing the comparisons based on groups:



Similar comparisons can be made using distributions (`sns.distplot`) or regression slopes (`sns.regplot`):

```
In: with sns.axes_style("darkgrid"):  
     chart = sns.FacetGrid(df_iris, col="target")  
     chart.map(sns.distplot, "sepal_length")
```

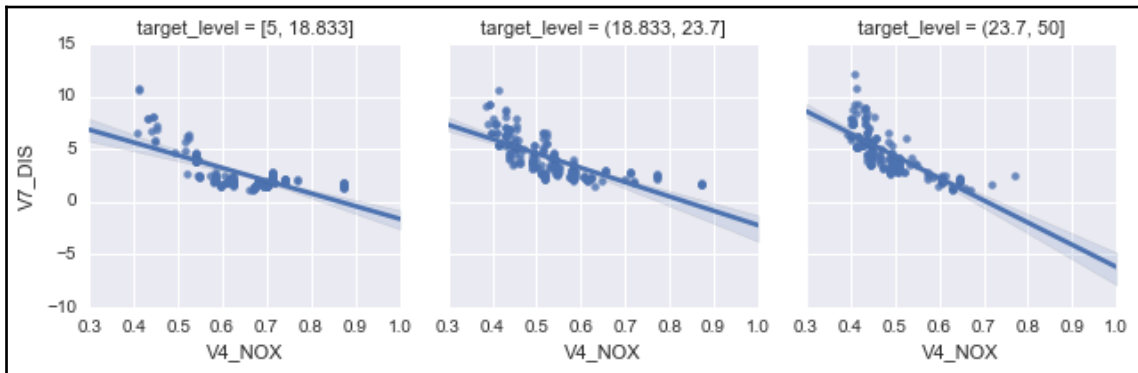
The first comparison is based on distributions:



The subsequent comparison is based on fitting a linear regression line:

```
In: with sns.axes_style("darkgrid"):  
    chart = sns.FacetGrid(df_boston, col="target_level")  
    chart.map(sns.regplot, "V4_NOX", "V7_DIS")
```

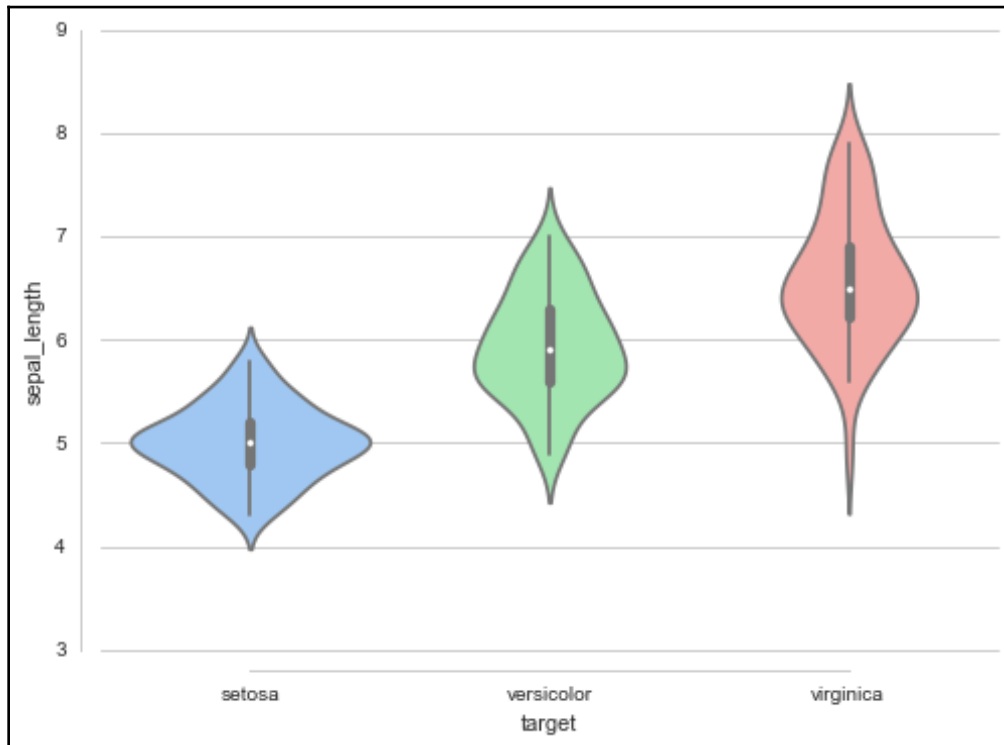
Here is the regression-based comparison:



As for evaluating data distributions across classes, Seaborn offers an alternative tool, which is the violin plot (<https://medium.com/@bioturing/5-reasons-you-should-use-a-violin-graph-31a9cdf2d0c6>). A violin plot is simply a boxplot whose box is shaped based on density estimation, thus visually conveying information that is more intuitive:

```
In: with sns.axes_style("whitegrid"):  
    ax = sns.violinplot(x="target", y="sepal_length",  
                        data=df_iris, palette="pastel")  
    sns.despine(offset=10, trim=True)
```

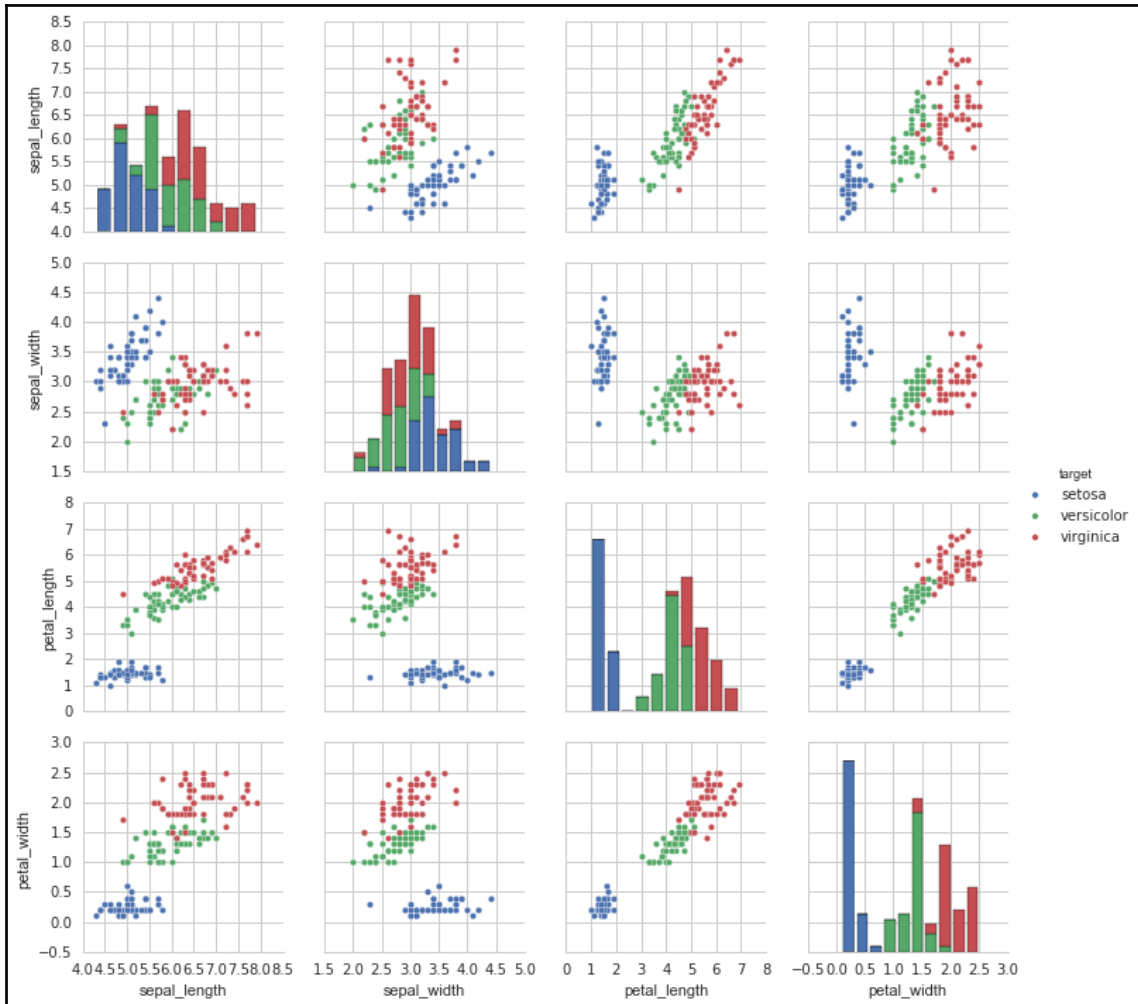
The violin plot produced by the previous code can provide interesting insights into the dataset:



Finally, Seaborn offers a much better way of creating a matrix of scatterplots by using the `pairplot` command and allowing you to define group colors (parameter `hue`) and how to populate the diagonal row. This is by using the `diag_kind` parameter, which can be a histogram (`'hist'`) or kernel density estimation (`'kde'`):

```
In: with sns.axes_style("whitegrid"):
     chart = sns.pairplot(data=df_iris, hue="target", diag_kind="hist")
```

The previous code will output a complete matrix of scatterplots for the dataset:



Advanced data learning representation

Some useful representations can be derived from the data science process. That is, the representation is not done directly from the data, but is achieved by using machine learning procedures, which inform us about how the algorithms operate and offer us a more precise overview of the role of each predictor in the predictions obtained. In particular, learning curves can provide a quick diagnosis to improve your models. This helps you figure out whether you need more observations, or need to enrich your variables.

Learning curves

A learning curve is a useful diagnostic graphic that depicts the behavior of your machine learning algorithm (your hypothesis) with respect to the available quantity of observations. The idea is to compare how the training performance (the error or accuracy of the in-sample cases) behaves with respect to the cross-validation (usually tenfold) using different in-sample sizes.

As far as the training error is concerned, you should expect it to be high at the start and then decrease. However, depending on the bias and variance level of the hypothesis, you will notice different behaviors:

- A high-bias hypothesis tends to start with average error performances, decreases rapidly on being exposed to more complex data, and then remains at the same level of performance no matter how many cases you further add.
- A low-bias learners tend to generalize better in the presence of many cases, but they are limited in their capability to approximate complex data structures, hence their limited performance.
- A high-variance hypothesis tends to start high in error performance and then slowly decreases as you add more cases. It tends to decrease slowly because it has a high capacity of recording the in-sample characteristics.

As for cross-validation, we can notice two behaviors:

- High-bias hypothesis tends to start with low performance, but it grows very rapidly until it reaches almost the same performance as that of the training. Then, it stops growing.
- High-variance hypothesis tends to start with very low performance. Then, steadily but slowly, it improves as more cases help generalize. It hardly reads the in-sample performances, and there is always a gap between them.

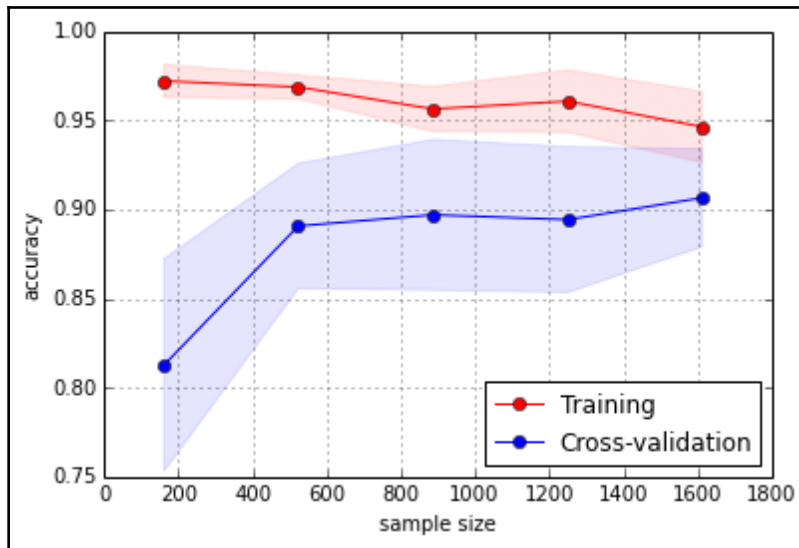
Being able to estimate whether your machine learning solution is behaving as a high-bias or high-variance hypothesis immediately helps you in deciding how to improve your data science project. Scikit-learn makes it simpler to calculate all the statistics that are necessary for the drawing of the visualization thanks to the `learning_curve` class, although visualizing them properly requires a few further calculations and commands:

```
In: import numpy as np
    from sklearn.learning_curve import learning_curve, validation_curve
    from sklearn.datasets import load_digits
    from sklearn.linear_model import SGDClassifier

    digits = load_digits()
    X, y = digits.data, digits.target
    hypothesis = SGDClassifier(loss='log', shuffle=True,
                              n_iter=5, penalty='l2',
                              alpha=0.0001, random_state=3)
    train_size, train_scores, test_scores = learning_curve(hypothesis, X,
                                                         y, train_sizes=np.linspace(0.1,1.0,5), cv=10,
                                                         scoring='accuracy',
                                                         exploit_incremental_learning=False,
                                                         n_jobs=-1)
    mean_train = np.mean(train_scores,axis=1)
    upper_train = np.clip(mean_train + np.std(train_scores,axis=1),0,1)
    lower_train = np.clip(mean_train - np.std(train_scores,axis=1),0,1)
    mean_test = np.mean(test_scores,axis=1)
    upper_test = np.clip(mean_test + np.std(test_scores,axis=1),0,1)
    lower_test = np.clip(mean_test - np.std(test_scores,axis=1),0,1)
    plt.plot(train_size,mean_train,'ro-', label='Training')
    plt.fill_between(train_size, upper_train,
                    lower_train, alpha=0.1, color='r')
    plt.plot(train_size,mean_test,'bo-', label='Cross-validation')
    plt.fill_between(train_size, upper_test, lower_test,
                    alpha=0.1, color='b')

    plt.grid()
    plt.xlabel('sample size') # adds label to x axis
    plt.ylabel('accuracy') # adds label to y axis
    plt.legend(loc='lower right', numpoints= 1)
    plt.show()
```

Based on different sample sizes, you soon get a learning curve plot:



The `learning_curve` class requires the following as an input:

- A series of training sizes stored in a list
- An indication of the number of folds to use, and the error measure
- Your machine learning algorithm to test (parameter estimator)
- The predictors (parameter x) and the target outcome (parameter y)

As a result, the class will produce three arrays; the first one containing the effective training sizes, the second presenting the training scores obtained at each cross-validation iteration, and the last one carrying the cross-validation scores.

By applying the mean and the standard deviation for both training and cross-validation, it is possible to display in the graph both the curve trends and their variation. You can also provide information about the stability of the recorded performances.

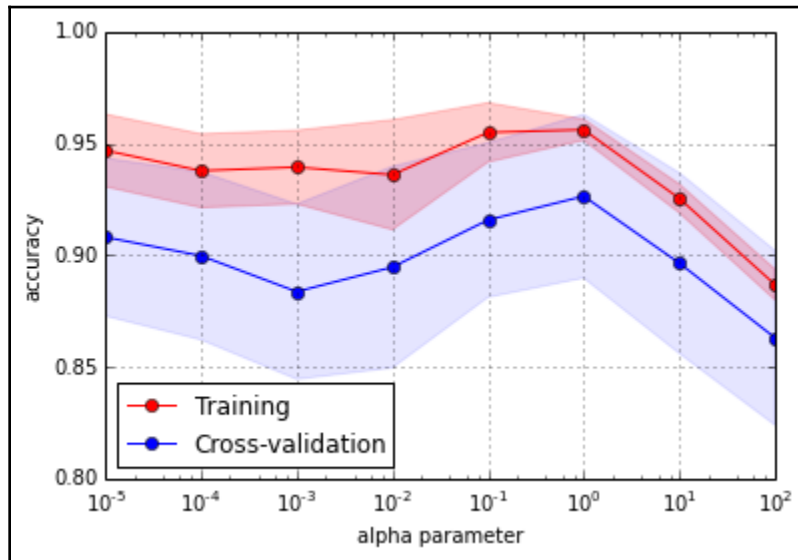
Validation curves

As learning curves operate on different sample sizes, validation curves estimate the training and cross-validation performance with respect to the values that a hyper-parameter can take. As in learning curves, similar considerations can be applied, though this particular visualization will grant you further insight about the optimization behavior of your parameter, visually suggesting to you the part of the hyper-parameter space that you should concentrate your search on:

```
In: from sklearn.learning_curve import validation_curve
testing_range = np.logspace(-5,2,8)
hypothesis = SGDClassifier(loss='log', shuffle=True,
                           n_iter=5, penalty='l2',
                           alpha=0.0001, random_state=3)
train_scores, test_scores = validation_curve(hypothesis, X, y,
                                             param_name='alpha',
                                             param_range=testing_range,
                                             cv=10, scoring='accuracy', n_jobs=-1)
mean_train = np.mean(train_scores,axis=1)
upper_train = np.clip(mean_train + np.std(train_scores,axis=1),0,1)
lower_train = np.clip(mean_train - np.std(train_scores,axis=1),0,1)
mean_test = np.mean(test_scores,axis=1)
upper_test = np.clip(mean_test + np.std(test_scores,axis=1),0,1)
lower_test = np.clip(mean_test - np.std(test_scores,axis=1),0,1)
plt.semilogx(testing_range,mean_train,'ro-', label='Training')
plt.fill_between(testing_range, upper_train, lower_train,
                 alpha=0.1, color='r')
plt.fill_between(testing_range, upper_train, lower_train,
                 alpha=0.1, color='r')
plt.semilogx(testing_range,mean_test,'bo-', label='Cross-validation')
plt.fill_between(testing_range, upper_test, lower_test,
                 alpha=0.1, color='b')

plt.grid()
plt.xlabel('alpha parameter') # adds label to x axis
plt.ylabel('accuracy') # adds label to y axis
plt.ylim(0.8,1.0)
plt.legend(loc='lower left', numpoints= 1)
plt.show()
```


After some computations, you will get a representation of the validation curve for the parameter:



The syntax of the `validation_curve` class is similar to that of the previously seen `learning_curve` but for the `param_name` and `param_range` parameters, which should be provided respectively with the hyper-parameter and the range that has to be tested. As for the results, the training and test results are returned in arrays.

Feature importance for RandomForests

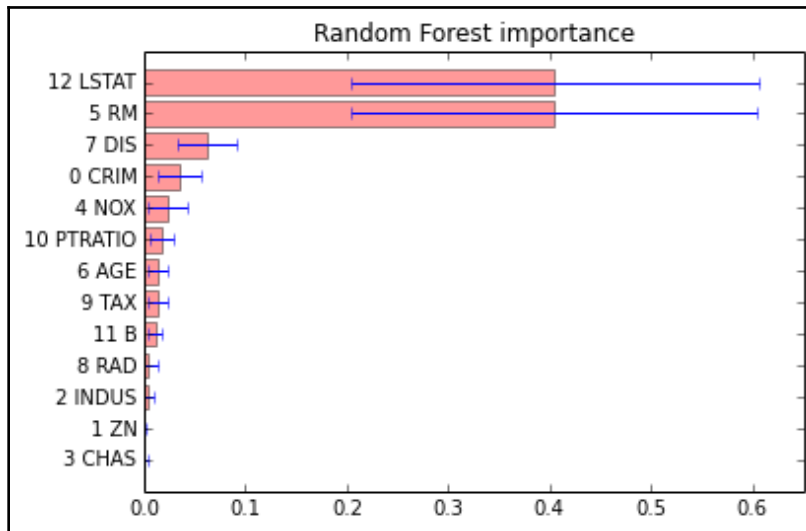
As discussed in the conclusion of Chapter 3, *The Data Pipeline*, selecting the right variables can improve your learning process by reducing noise, the variance of estimates, and the burden of too many computations. Ensemble methods, such as Random Forest in particular, can provide you with a different view of the role played by a variable when working together with other ones in your dataset.

Here, we show you how to extract the importance of RandomForest and Extra-Tree models. Importance is calculated in the fashion originally described in the book *Classification and Regression Trees* by Breiman, Friedman et al. in 1984. It was a true classic that laid solid foundations for classification trees. In the book, importance is described in terms of *gini importance* or *mean decrease impurity*, which is the total decrement in node impurity due to a specific variable averaged over all trees of the ensemble. In other words, mean decrease impurity is the total error reduction of nodes split on that variable multiplied by the number of samples that were routed to each of the nodes. Noticeably, accordingly to this importance calculation method, not only does error reduction depend on the error measure-Gini or Entropy for classification, and MSE for regression, but also splits at the head of the tree are deemed more important because they involve dealing with more examples.

In a few steps, we'll learn how to obtain such information and project it onto a clear visualization:

```
In: from sklearn.datasets import load_boston
    boston = load_boston()
    X, y = boston.data, boston.target
    feature_names = np.array([' '.join([str(b), a]) for a,b in
                               zip(boston.feature_names,range(
                                   len(boston.feature_names)))]])
    from sklearn.ensemble import RandomForestRegressor
    RF = RandomForestRegressor(n_estimators=100,
                              random_state=101).fit(X, y)
    importance = np.mean([tree.feature_importances_ for tree in
                          RF.estimators_],axis=0)
    std = np.std([tree.feature_importances_ for tree in
                  RF.estimators_],axis=0)
    indices = np.argsort(importance)
    range_ = range(len(importance))
    plt.figure()
    plt.title("Random Forest importance")
    plt.barh(range_,importance[indices],
             color="r", xerr=std[indices], alpha=0.4, align="center")
    plt.yticks(range(len(importance)), feature_names[indices])
    plt.ylim([-1, len(importance)])
    plt.xlim([0.0, 0.65])
    plt.show()
```

The code will produce the following chart highlighting important features of the model:



For each of the estimators (in our case, we have 100 models), the algorithm estimated a score to rank each variable's importance. The RandomForest model is made up of decision trees that can be made up of many branches, since the algorithm tries to obtain very small terminal leaves. One of its variables is deemed important if, after casually permuting its original values, the resulting predictions of the permuted model are very different in terms of accuracy as compared to the predictions of the original model.

The importance vectors are averaged over the number of estimators, and the standard deviation of the estimations is computed by a list comprehension (the assignment of variables importance and `std`). Now, sorted according to the importance score (the vector indices), the results are projected onto a bar graph with an error bar provided by the standard deviation.

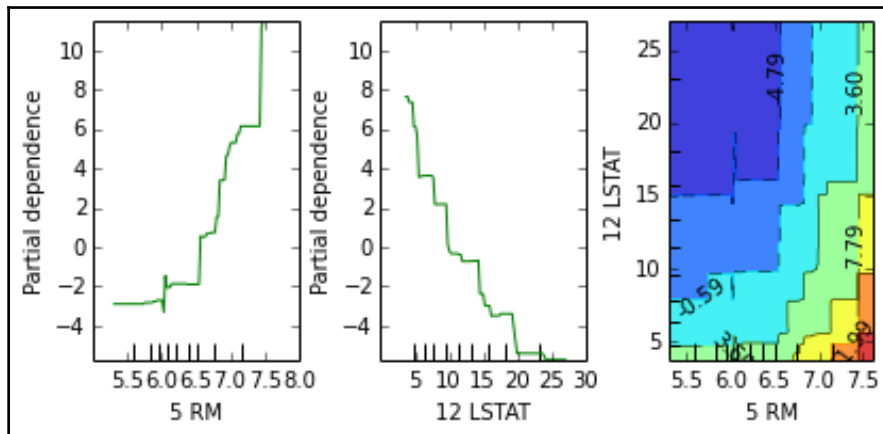
In our LSTAT analysis, the percentage of the lower status population in the area and RM, which is the average number of rooms per dwelling, are pointed out as the most decisive variables in our RandomForest model.

Gradient Boosting Trees partial dependence plotting

The estimate of the importance of a feature is a piece of information that can help you operate on the best choices to determine the features to be used. Sometimes, you may need to understand better why a variable is important in predicting a certain outcome. Gradient Boosting Trees, by controlling the effect of all the other variables involved in the analysis, provide you with a clear point of view of the relationship of a variable with respect to the predicted results. Such information can provide you with more insights into causation dynamics than what you may have obtained by using a very effective EDA:

```
In: from sklearn.ensemble.partial_dependence import
    plot_partial_dependence
    from sklearn.ensemble import GradientBoostingRegressor
    GBM = GradientBoostingRegressor(n_estimators=100,
                                    random_state=101).fit(X, y)
    features = [5,12, (5,12)]
    fig, axis = plot_partial_dependence(GBM, X, features,
                                       feature_names=feature_names)
```

As an output, you get three plots, which constitute the partial plots of RM and LSTAT features:



The `plot_partial_dependence` class will automatically provide you with the visualization after you provide an analysis plan on your part. You need to present a list of indexes of the features to be plotted singularly, and the tuples of the indexes of those that you would like to plot on a heat map (the features are the axis, and the heat value corresponds to the outcome).

In the preceding example, both the average number of rooms and the percentage of the lower status population have been represented, thus displaying an expected behavior. Interestingly, the heat map, which explains how they together contribute to the value of the outcome, reveals that they do not interact in any particular way (it is single hill-climbing). However, it is also revealed that LSTAT is a strong delimiter of the resulting housing values when it is above 5.

Creating a prediction server with machine-learning-as-a-service

Many times, during your working career as a data scientist, you'll find yourself having need of a predictor decoupled from the code you're currently working on; for example, as follows:

- You're developing an app for your phone, and you want to save on memory
- You're coding in a non-Python programming language (Java, Scala, C, C++, and so on) and you need to call the predictor you've developed in Python
- You're operating on big data, and the model is trained in the same remote location where the data is stored

In all these cases, it would be nice to have a service over HTTP that does predictions-as-a-service, or generically, any **machine-learning-as-a-service (ML-AAS)**.

Bottle, a Python web framework, is the starting point for micro apps over HTTP. It is a very simple library for Python, providing the essential objects and functions to create a web app. Also, it can be paired with all the other libraries available in Python. Before going into the prediction-as-a-service, let's see how a basic `Hello World` program is built with Bottle. Please note that the following listings are meant for Python REPL, as a script, and not for a Jupyter Notebook:

```
# File: bottle1.py

from bottle import route, run, template

port = 9099
```

```
@route('/personal/<name>')
def homepage(name):
    return template('Hi <b>{{name}}</b>!', name=name)

print("Try going to http://localhost:{}".format(port))
print("Try going to http://localhost:{}".format(port))

run(host='localhost', port=port)
```

Let's analyze the code line by line before executing it:

1. We started importing the functions and the classes that we need from the Bottle module.
2. Then, we specified the port that the HTTP server would listen to.
3. In the example, we selected port 9099; feel free to change it to another one, but first check whether any other service is using it (remember that HTTP is on top of TCP).
4. The next step is the definition of the API endpoint. The *route* decorator applies the function defined after it when an HTTP call to the path specified as an argument is performed. Note that in the path, it says *name*, and that is the argument of the coming function. That means *name* is a parameter of the call; you can select whatsoever string you like in the HTTP call, and your selection will be passed to the function as the parameter *name*.
5. Then, inside the function *home page*, a template with an HTML code was returned. In a simpler way, think of it as the *template* function creating the page you'll see from your browser.



Template, in this example, is just a plain HTML page, but it can be more complex (it can actually be a template page with some blanks to fill in). A complete description of templates is out of the scope of this section since we will be using the framework just for a simple, plain output. If you need additional information, surf the Bottle help pages.

6. Finally, after the *print* functions, there's the core *run* function. It's a blocking function and will set up the web server on the host and port provided as arguments. When you run the code in the listing, once that function is executed, you can open your browser and point it to `http://localhost:9099/personal/Carl`, and you'll find the following text:
Hi Carl!

Of course, changing the name in the HTTP call from *Carl* to *Tom* or any other name will result in a different page, containing the name specified in the call.



Please note that in this dummy example, we just defined the `/personal/<name>` route. Any other call will result in `Error 404`, unless defined in the code.

To turn it off, we need to press `Ctrl + C` in the command line (remember that the `run` function is blocking).

Let's now create a service that is more data science-oriented; we will create an HTML page with a form asking for the sepal length and width, and the petal length and width, to classify the iris sample. For this example, we will use the iris dataset to train our scikit-learn classifier. Then, for each prediction, we simply call the `predict` function on the classifier, sending back the prediction:

```
# File: bottle2.py

from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
from bottle import run, request, get, post
import numpy as np

port = 9099

@get('/predict')
def predict():
    return '''
        <form action="/prediction" method="post">
            Sepal length [cm]: <input name="sl" type="text" /><br/>
            Sepal width [cm]: <input name="sw" type="text" /><br/>
            Petal length [cm]: <input name="pl" type="text" /><br/>
            Petal width [cm]: <input name="pw" type="text" /><br/>
            <input value="Predict" type="submit" />
        </form>
    '''

@post('/prediction')
def do_prediction():
    try:
        sample = [float(request.POST.get('sl')),
                  float(request.POST.get('sw')),
                  float(request.POST.get('pl')),
                  float(request.POST.get('pw'))]

        pred = classifier.predict(np.matrix(sample))[0]
        return "<p>The predictor says it's a <b>{}</b></p>"\
            .format(iris['target_names'][pred])
```

```
except:
    return "<p>Error, values should be all numbers</p>"

iris = load_iris()
classifier = LogisticRegression()
classifier.fit(iris.data, iris.target)

print("Try going to http://localhost:{}".format(port))
run(host='localhost', port=port)

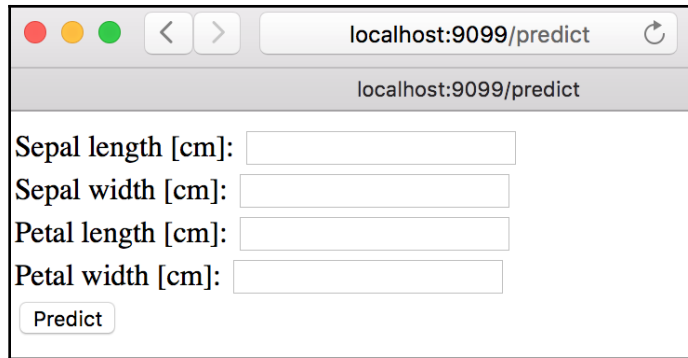
# Try insert the following values:
# [ 5.1, 3.5, 1.4, 0.2] -> setosa
# [ 7.0 3.2, 4.7, 1.4] -> versicolor
# [ 6.3, 3.3, 6.0, 2.5] -> virginica
```

After some imports, here we use the `get` decorator, specifying a route valid only for HTTP GET calls. The decorator, as well as the function following, has no parameters since all the features should be inserted into the HTML form, defined in the `predict` function. The form, when submitted, is passed to the `/prediction` page using an HTTP POST.

Now, we need to create a route for this call, and that's what we do in the `do_prediction` function. Its decorator is `post` (that is, opposite to `get`; it defines only POST routes) on the `/prediction` page. Data is parsed and transformed into a double (default parameters are strings), and then the feature vector is fed into the `classifier` global variable to obtain a prediction. This is returned using a simple template. The object request contains all the parameters passed to the service, including the entire variable we *POST-ed* to the route. Finally, it seems we just need to define the global variable `classifier` – that is, a classifier trained on the iris dataset – and lastly, we can call the `run` function.

For this dummy example, we've used a logistic regressor as a classifier and trained on the full Iris dataset, leaving all the parameters as default. In a real case, here you would tune your classifier as best as possible.

When this code is run, if everything works well, you can point your browser to `http://localhost:9099/predict` and you'll see the form:



The screenshot shows a web browser window with the address bar displaying 'localhost:9099/predict'. The page content includes four text input fields labeled 'Sepal length [cm]', 'Sepal width [cm]', 'Petal length [cm]', and 'Petal width [cm]'. Below these fields is a button labeled 'Predict'.

Inserting the values (5.1, 3.5, 1.4, 0.2) after clicking on the **Predict** button, you should be redirected to <http://localhost:9099/prediction>, where the The predictor says it's a setosa string should be displayed. Also, note that if you insert invalid entries in the form (for example, leaving it empty or inserting a string instead of a number), you'll get an HTML page that says that there's an error.

We're halfway through this section, and we've already seen how easy and quick it is to create an HTTP endpoint with Bottle. Now, let's try to create a prediction-as-a-service that can be called in any program. We will submit the feature vector as a `get` call, and the returned prediction will be in JSON format. Here's the code for this solution:

```
# File: bottle3.py

from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
from bottle import run, request, get, response
import numpy as np
import json

port = 9099

@get('/prediction')
def do_prediction():

    pred = {}

    try:
        sample = [float(request.GET.get('sl')),
                  float(request.GET.get('sw')),
                  float(request.GET.get('pl')),
                  float(request.GET.get('pw'))]

        pred['predicted_label'] =
```

```

        iris['target_names']
    [classifier.predict(np.matrix(sample)) [0]]
        pred['status'] = "OK"
    except:
        pred['status'] = "ERROR"

    response.content_type = 'application/json'
    return json.dumps(pred)

iris = load_iris()
classifier = LogisticRegression()
classifier.fit(iris.data, iris.target)

print("Try going to http://localhost:{}/prediction\
      s1=5.1&sw=3.5&pl=1.4&pw=0.2".format(port))
print("Try going to http://localhost:{}/prediction\
      s1=A&sw=B&pl=C&pw=D".format(port))
run(host='localhost', port=port)

```

The solution is pretty straightforward and easy; still, let's analyze it step by step. The entry point of the feature is defined by the `get` decorator on the `/prediction` path. In there, we will access the `GET` values to extract the predictions (note that if your classifier needs many features, it may be better to use a `POST` call here). Exactly as in the previous example, the prediction is generated; finally, the value is inserted in a Python dictionary, altogether with the `OK` value for the `status` key. If an exception is raised in this function, there will be no prediction, but an `ERROR` string in the `status` key. Then, we set the output application format to `JSON`, and we serialize the Python dictionary to a `JSON` string.

When it runs, we can access the URL, `localhost:9099/prediction`, followed by the feature values, and we will get back the prediction as `JSON`. Note that we don't need a browser to interpret the returned `HTTP` response since it's a `JSON`. Therefore, we can call the endpoint from different applications (`wget`, browser, or `curl`) or any programming language (including Python itself). To see it working, start it and point your browser to (or request the URL in any way)

```
http://localhost:9099/prediction?s1=5.1&sw=3.5&pl=1.4&pw=0.2. You'll get
back the valid JSON: {"predicted_label": "setosa", "status": "OK"}. Also, if
something goes wrong in the parsing of the parameters, you'll get this JSON: {"status":
"ERROR"}. And that's your first ML-AAS!
```

Although simple and quick, `Bottle` has many other functions to be explored. It's not as complete as other frameworks, however. If your application needs some extraordinary functionality, check out `Flask` or `Django` modules.

Summary

This chapter provided an overview of essential data science by providing examples of both basic and advanced graphical representations of data, machine learning processes, and results. We explored the `pylab` module from `matplotlib`, which gives the easiest and fastest access to the graphical capabilities of the package. We used `pandas` for EDA, and tested the graphical utilities provided by `scikit-learn`. All examples were like building blocks, and they are all easily customizable in order to provide you with a fast template for visualization.

In the next chapter, you'll be introduced to **graphs**, which are an interesting deviation from the predictors/target flat matrices. They are quite a hot topic in data science now. Expect to delve into very complex and intricate networks.