USN | | | | | | | | | |

Internal Assessment Test 2 – November 2024

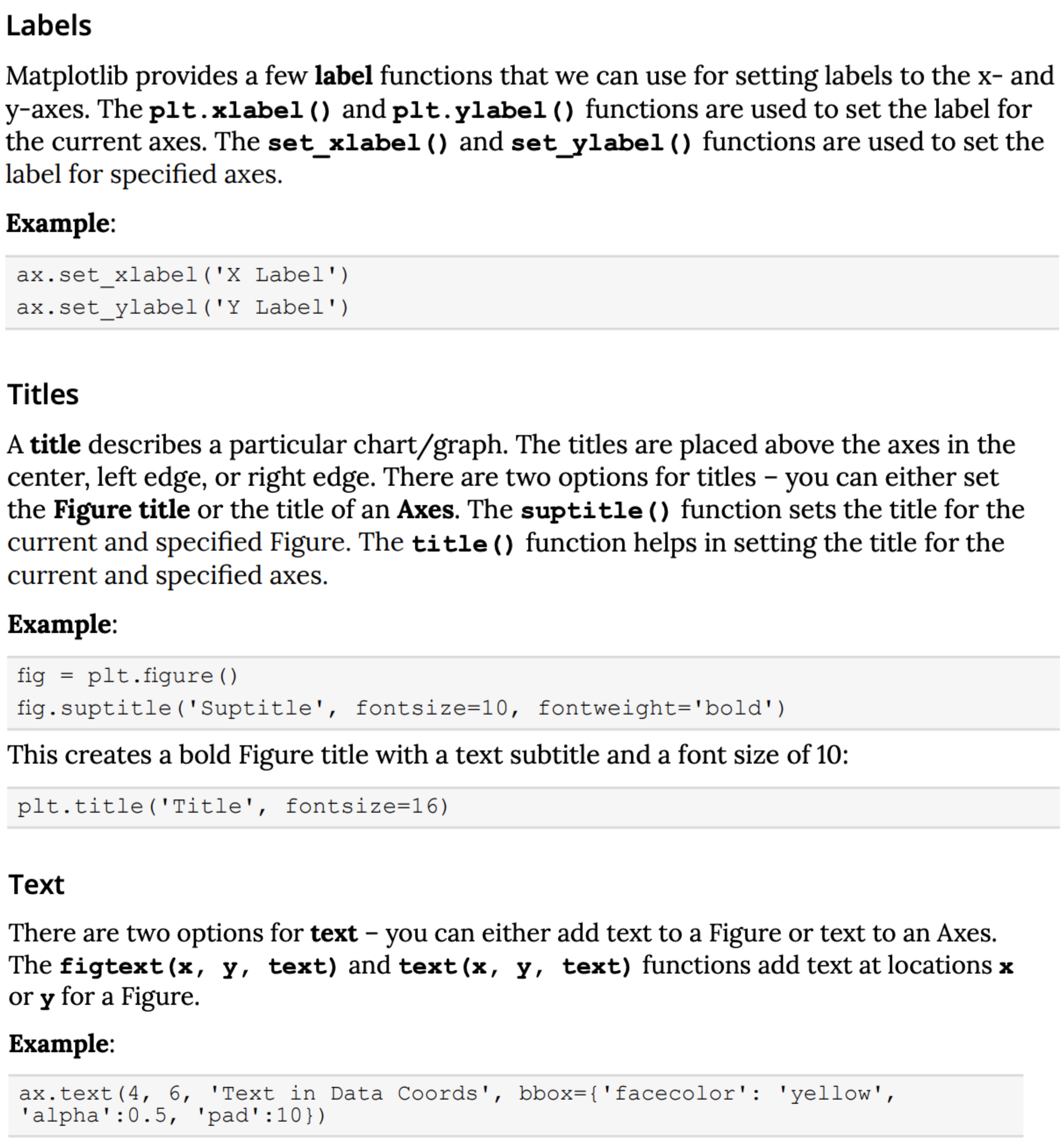
Sub:	**Data Visualization**					Sub Code:	**21AD71**	Branch:	**AInDS**	
Date:	**19/11/2024**	Duration:	**90 minutes**	Max Marks:	**50**	Sem	**VII**		**OBE**	

		Answer any FIVE Questions	**MARKS**	**CO**	**RBT**
1	a	Explain the following with respect to Matplotlib along with examples. 1. Labels 2. Titles 3. Text 4. Annotations 5. Legends	[10]	CO3	L1
2	a	Explain the different geospatial visualizations with examples?	[10]	CO4	L2
3	a	Explain GeoJSON format.	[4]	CO4	L2
	b	Why is geoplotlib preferred over matplotlib for geospatial data visualization?	[6]		
4	a	Explain how univariate and bivariate distributions are visualized in seaborn with examples?	[6]	CO3	L2
	b	Plot the following in matplotlib: 1. Bubble plot 2. Box plot	[4]		L3
5	a	Explain basic image operations of matplotlib?	[6]	CO3	L2
	b	How to create & close figures in matplotlib?	[4]		L1
6	a	Explain seaborn figure styles.	[5]	CO3	L2
	b	Explain the parameters controlling the scale of the plot elements.	[5]		

CI CCI HOD

Answers:

1.
   a.



Labels

Matplotlib provides a few **label** functions that we can use for setting labels to the x- and y-axes. The **`plt.xlabel()`** and **`plt.ylabel()`** functions are used to set the label for the current axes. The **`set_xlabel()`** and **`set_ylabel()`** functions are used to set the label for specified axes.

Example:

```
ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
```

Titles

A **title** describes a particular chart/graph. The titles are placed above the axes in the center, left edge, or right edge. There are two options for titles – you can either set the **Figure title** or the title of an **Axes**. The **`suptitle()`** function sets the title for the current and specified Figure. The **`title()`** function helps in setting the title for the current and specified axes.

Example:

```
fig = plt.figure()
fig.suptitle('Suptitle', fontsize=10, fontweight='bold')
```

This creates a bold Figure title with a text subtitle and a font size of 10:

```
plt.title('Title', fontsize=16)
```

Text

There are two options for **text** – you can either add text to a Figure or text to an Axes. The **`figtext(x, y, text)`** and **`text(x, y, text)`** functions add text at locations **`x`** or **`y`** for a Figure.

Example:

```
ax.text(4, 6, 'Text in Data Coords', bbox={'facecolor': 'yellow',
'alpha':0.5, 'pad':10})
```

Annotations

Annotations are used to annotate some features of the plot. In annotations, there are two locations to consider: the annotated location, xy, and the location of the annotation, text xytext. It is useful to specify the parameter arrow props, which results in an arrow pointing to the annotated location.

Example:

```
ax.annotate('Example of Annotate', xy=(4,2), xytext=(8,4),
arrowprops=dict(facecolor='green', shrink=0.05))
```

This creates a green arrow pointing to the data coordinates (4, 2) with the text **Example of Annotate** at data coordinates (8, 4):

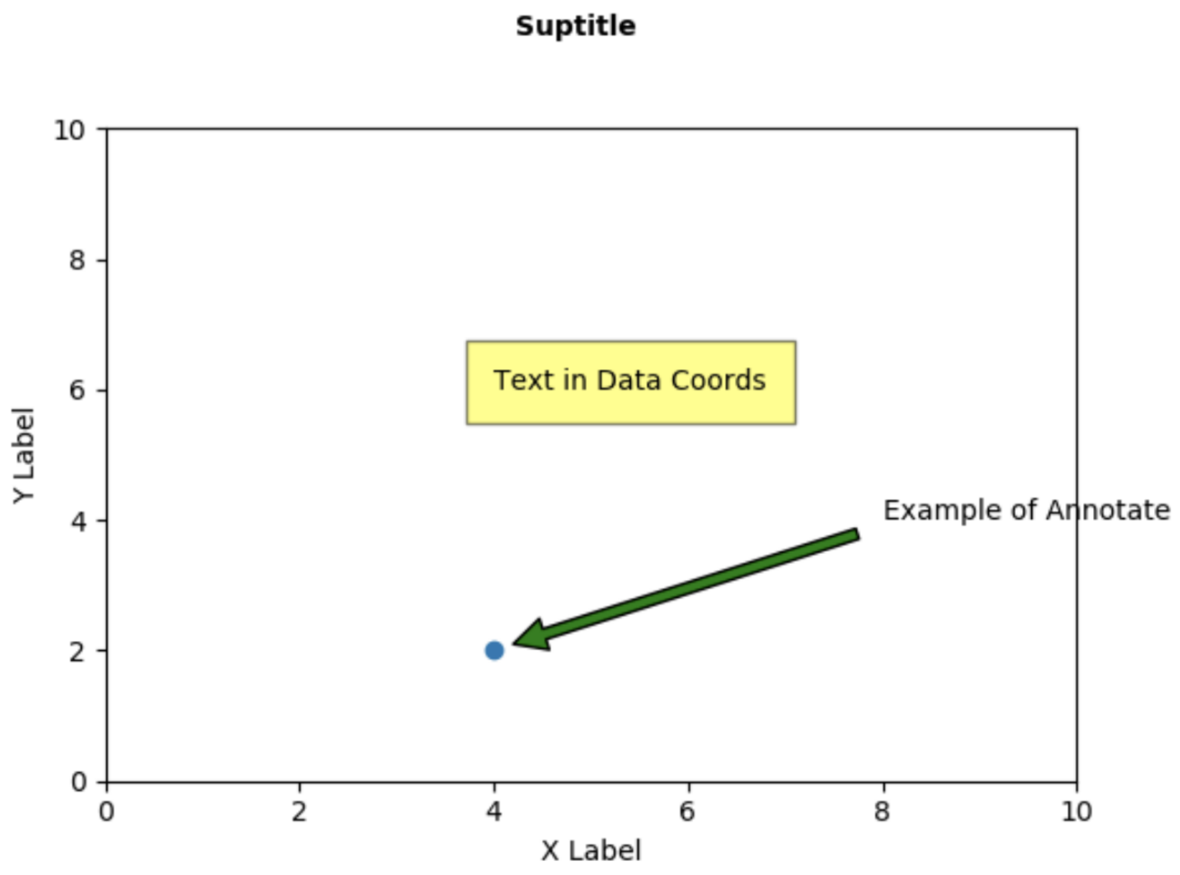


Figure 3.13: Implementation of text commands

Legends

Legend describes the content of the plot. To add a **legend** to your Axes, we have to specify the **`label`** parameter at the time of plot creation. Calling **`plt.legend()`** for the current Axes or **`Axes.legend()`** for a specific Axes will add the legend. The **`loc`** parameter specifies the location of the legend.

Example:

```
plt.plot([1, 2, 3], label='Label 1')
plt.plot([2, 4, 3], label='Label 2')
plt.legend()
```

This example is illustrated in the following diagram:

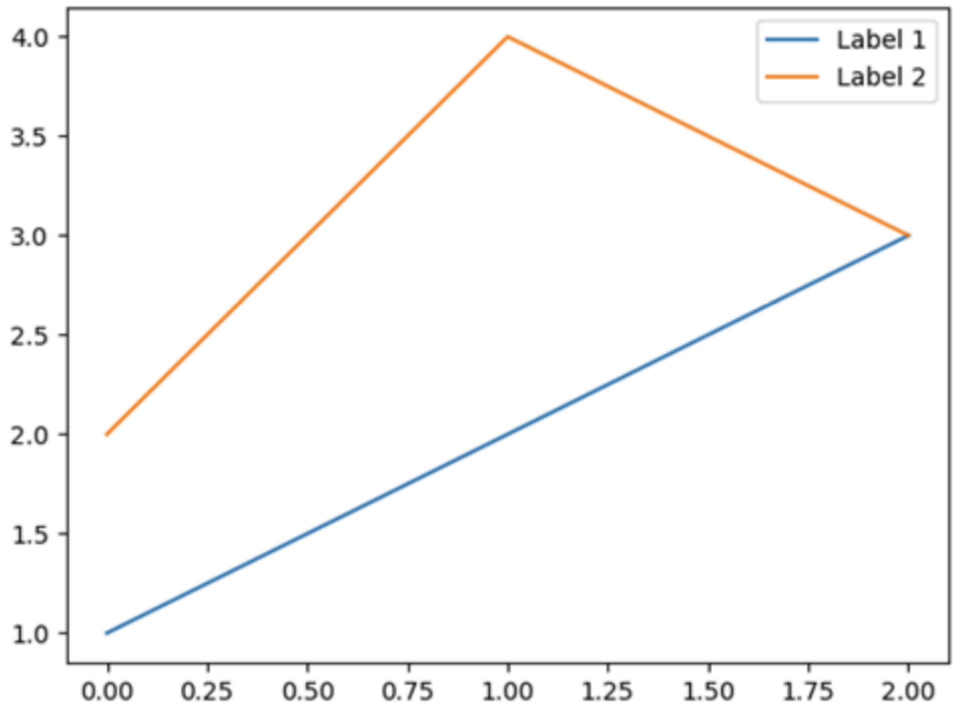


Figure 3.14: Legend example

2. Voronoi tessellation, Delaunay triangulation, and choropleth plots are a few of the geospatial visualizations.

Voronoi Tessellation

In a **Voronoi tessellation**, each pair of data points is separated by a line that is the same distance from both data points. The separation creates cells that, for every given point, marks which data point is closer. The closer the data points, the smaller the cells.

The following example shows how you can simply use the **`voronoi`** method to create this visualization:

```
# plotting our dataset as voronoi plot
geoplotlib.voronoi(dataset_filtered, line_color='b')
geoplotlib.set_smoothing(True)

geoplotlib.show()
```

Delaunay Triangulation

A **Delaunay triangulation** is related to Voronoi tessellation. When connecting each data point to every other data point that shares an edge, we end up with a plot that is triangulated. The closer the data points are to each other, the smaller the triangles will be. This gives us a visual clue about the density of points in specific areas. When combined with color gradients, we get insights about points of interest, which can be compared with a heatmap:

```
# plotting our dataset as a delaunay
geoplotlib.delaunay(dataset_filtered, cmap='hot_r')
geoplotlib.set_smoothing(True)

geoplotlib.show()
```

Choropleth Plot

This kind of geographical plot displays areas such as the states of a country in a shaded or colored manner. The shade or color of the plot is determined by a single data point or a set of data points. It gives an abstract view of a geographical area to visualize the relationships and differences between the different areas. In the following code and

3.

a. **The GeoJSON Format**

The GeoJSON format is used to encode a variety of data structures, such as points, lines, and polygons with a focus on geographical visualization. The format has a defined structure that each valid file has to follow:

```
{
  "type": "Feature",
  "properties": {
    "name": "Dinagat Islands"
  },
  "geometry": {
    "type": "Point",
    "coordinates": [125.6, 10.1]
  }
}
```

Each object with additional properties, for example, an ID or name attribute, is a **`Feature`**. The **`properties`** attribute simply allows additional information to be added to the feature. The **`geometry`** attribute holds information about the type of feature we are working with, for example, a **`Point`**, and its specific coordinates. The coordinates define the positions for the "waypoints" of the given type. Those coordinates define the shape of the element to be displayed by the plotting library.

b.

i. Matplotlib is the commonly used visualization library in python.
ii. However, Matplotlib is not designed for this task because its interfaces are complicated and inconvenient to use.
iii. Matplotlib also restricts how geographical data can be displayed.

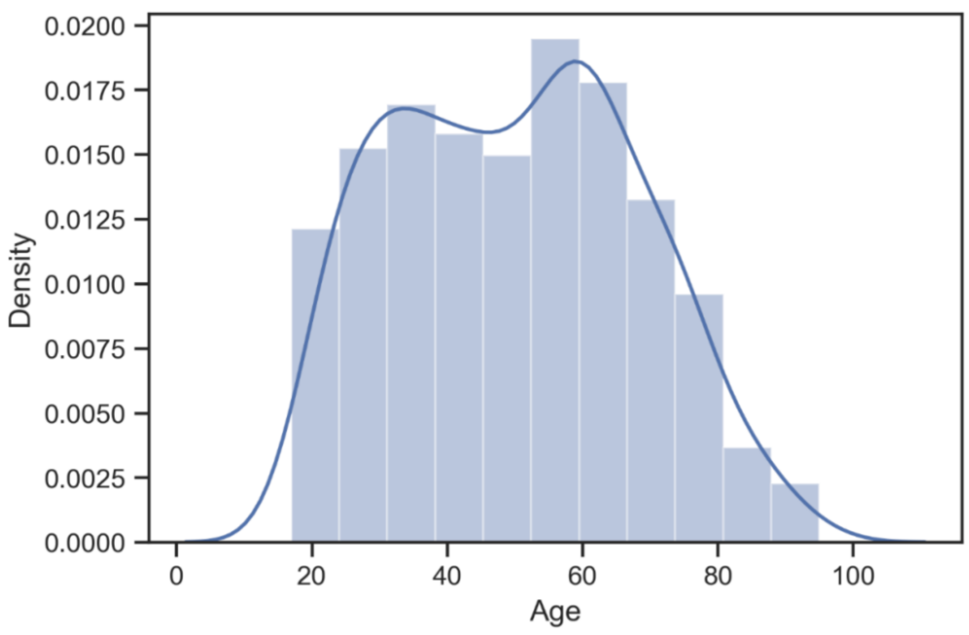
iv. Cartopy is a Python package designed to make drawing maps for data analysis and visualization easy.
v. Basemap is a great tool for creating maps using python in a simple way. It's a matplotlib extension, so it has got all its features to create data visualizations
vi. The Basemap and Cartopy libraries allow you to plot on a world map, but these packages do not support drawing on map tiles.
vii. Maptiles are image file that represents a specific geographic area at a particular zoom level.
viii. Geoplotlib, on the other hand, was designed precisely for this purpose; it not only provides map tiles but also allows for interactivity and simple animations.
ix. It provides a simple interface that allows access to compelling geospatial visualizations such as histograms, point-based plots, tessellations such as Voronoi or Delaunay, and choropleth plots.

4.

a. Seaborn offers handy functions to examine univariate and bivariate distributions. One possible way to look at a univariate distribution in Seaborn is by using the distplot() function. This will draw a histogram and fit a kernel density estimate (KDE), as illustrated in the following example:

```
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
data = pd.read_csv('../../Datasets/age_salary_hours.csv')
sns.distplot(data.loc[:, 'Age'])
plt.xlabel('Age')
plt.ylabel('Density')
```

The result is shown in the following diagram:



To just visualize the KDE, Seaborn provides the **`kdeplot()`** function:

```
sns.kdeplot(data.loc[:, 'Age'], shade=True)
plt.xlabel('Age')
plt.ylabel('Density')
```

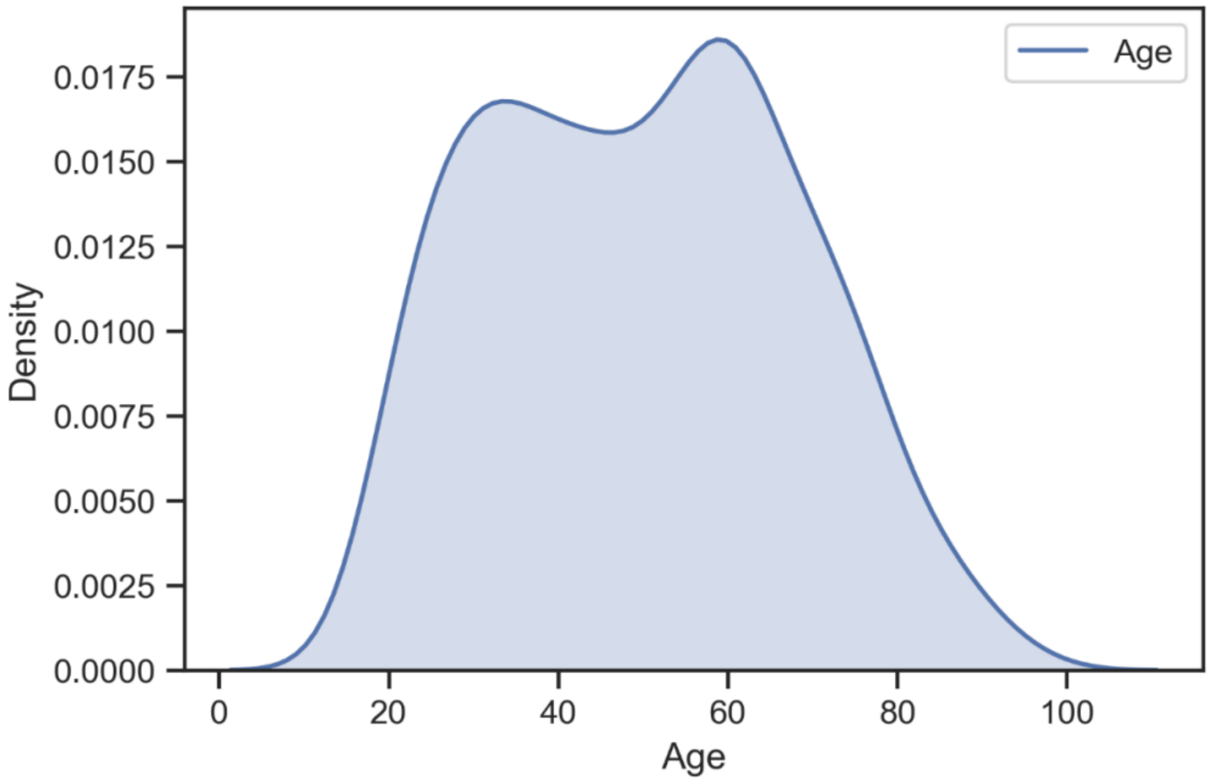


Figure 4.31: KDE for a univariate distribution

Plotting Bivariate Distributions

For visualizing **bivariate distributions**, we will introduce three different plots. The first two plots use the **`jointplot()`** function, which creates a multi-panel figure that shows both the joint relationship between both variables and the corresponding marginal distributions.

A scatter plot shows each observation as points on the **x** and **y** axes. Additionally, a histogram for each variable is shown:

```
import pandas as pd
import seaborn as sns
data = pd.read_csv('../../Datasets/age_salary_hours.csv')
sns.set(style="white")
sns.jointplot(x="Annual Salary", y="Age", data=data))
```

The scatter plot with marginal histograms is shown in the following diagram:

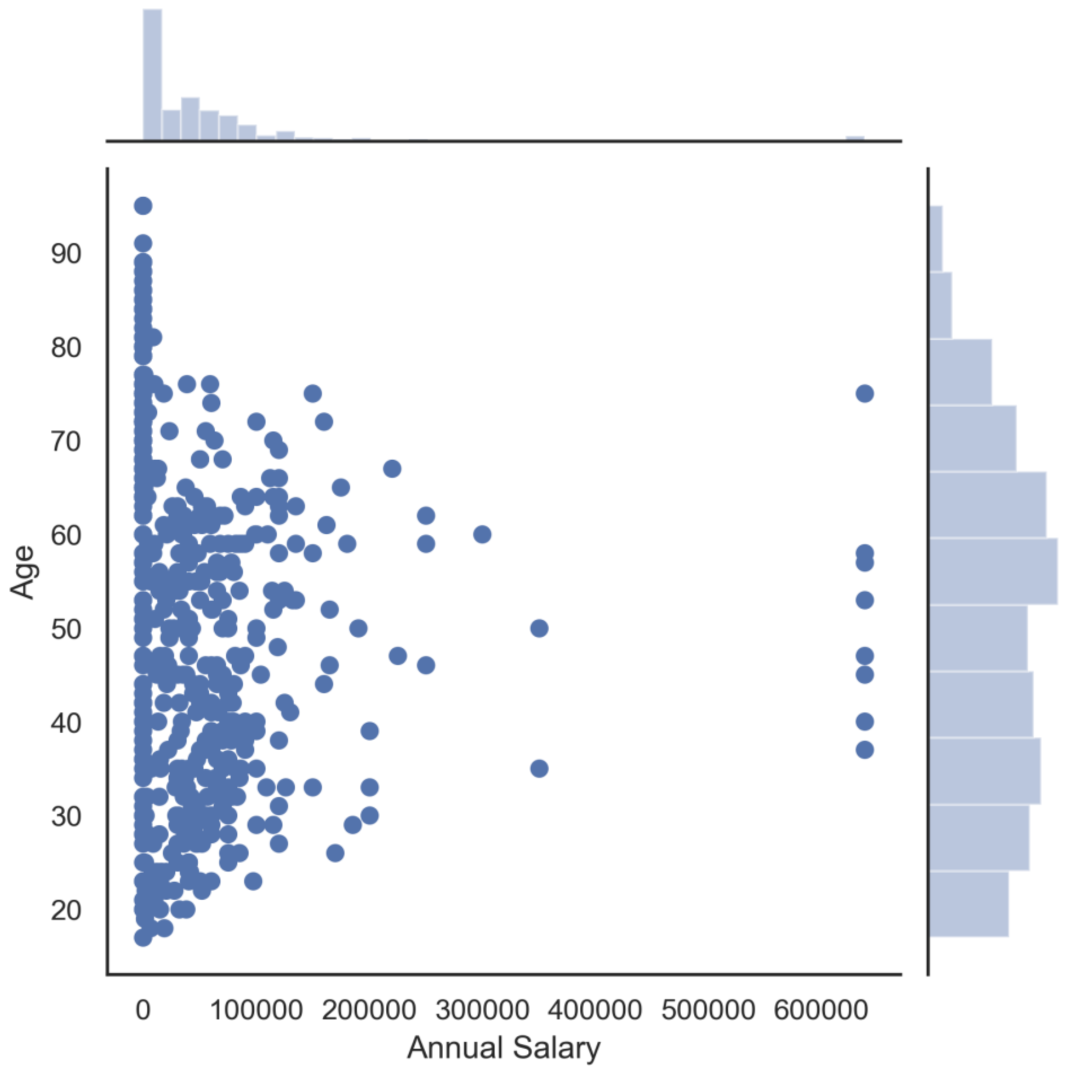


Figure 4.32: Scatter plot with marginal histograms

b. **Bubble Plot**

The **`plt.scatter`** function is used to create a bubble plot. To visualize a third or fourth variable, the parameters **`s`** (scale) and **`c`** (color) can be used.

Example:

```
plt.scatter(x, y, s=z*500, c=c, alpha=0.5)
plt.colorbar()
```

The **`colorbar`** function adds a colorbar to the plot, which indicates the value of the color. The result is shown in the following diagram:

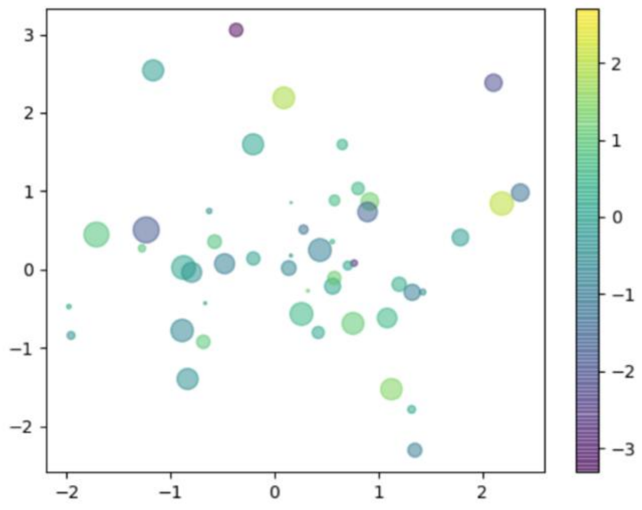


Figure 3.33: Bubble plot with color bar

Box Plot

The box plot shows multiple statistical measurements. The box extends from the lower to the upper quartile values of the data, thereby allowing us to visualize the interquartile range. For more details regarding the plot, refer to the previous chapter. The **`plt.boxplot(x)`** function creates a box plot.

Important parameters:

- **`x`**: Specifies the input data. It specifies either a 1D array for a single box, or a sequence of arrays for multiple boxes.
- **`notch`**: (optional) If true, notches will be added to the plot to indicate the confidence interval around the median.

- **`labels`**: (optional) Specifies the labels as a sequence.
- **`showfliers`**: (optional) By default, it is true, and outliers are plotted beyond the caps.
- **`showmeans`**: (optional) If true, arithmetic means are shown.

Example:

```
plt.boxplot([x1, x2], labels=['A', 'B'])
```

The result of the preceding code is shown in the following diagram:

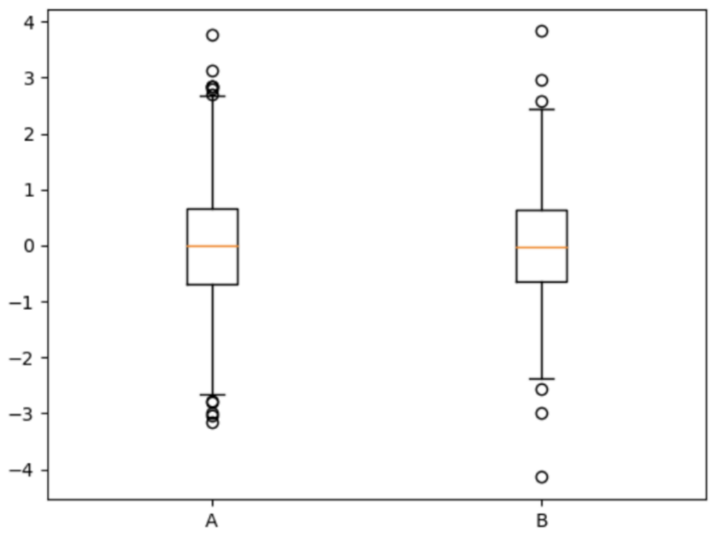


Figure 3.27: Box plot

5.

a. **Basic Image Operations**

The following are the basic operations for designing an image.

Loading Images

If you encounter image formats that are not supported by Matplotlib, we recommend using the **Pillow** library to load the image. In Matplotlib, loading images is part of the **image** submodule. We use the alias **mpimg** for the submodule, as follows:

```
import matplotlib.image as mpimg
```

The **mpimg.imread(fname)** reads an image and returns it as a **numpy.array** object. For grayscale images, the returned array has a shape (height, width), for RGB images (height, width, 3), and for RGBA images (height, width, 4). The array values range from 0 to 255.

We can also load the image in the following manner:

```
img_filenames = os.listdir('../../Datasets/images')
imgs = [mpimg.imread(os.path.join('../../Datasets/images', img_filename))
for img_filename in img_filenames]
```

The **os.listdir()** method in Python is used to get the list of all files and directories in the specified directory and then the **os.path.join()** function is used to join one or more path components intelligently.

Saving Images

The **mpimg.imsave(fname, array)** saves a **numpy.array** object as an image file. If the **format** parameter is not given, the format is deduced from the filename extension. With the optional parameters **vmin** and **vmax**, the color limits can be set manually. For a grayscale image, the default for the optional parameter, **cmap**, is **'viridis'**; you might want to change it to **'gray'**.

Plotting a Single Image

The plt.imshow(img) displays an image and returns an AxesImage object. For grayscale images with shape (height, width), the image array is visualized using a colormap. The default colormap is 'viridis'. To actually visualize a grayscale image, the colormap has to be set to 'gray' (that is, plt.imshow(img, cmap='gray'). Values for grayscale, RGB, and RGBA images can be either float or uint8, and range from [0…1] or [0…255], respectively. To manually define the value range, the parameters vmin and vmax must be specified.

Sometimes, it might be helpful to get an insight into the color values. We can simply add a color bar to the image plot. It is recommended to use a colormap with high contrast—for example, **jet**:

```
plt.imshow(img, cmap='jet')
plt.colorbar()
```

Another way to get insight into the image values is to plot a histogram, as shown in the following diagram. To plot the histogram for an image array, the array has to be flattened using **`numpy.ravel`**:

```
plt.hist(img.ravel(), bins=256, range=(0, 1))
```

Plotting Multiple Images in a Grid

To plot multiple images in a grid, we can simply use **`plt.subplots`** and plot an image per **`Axes`**:

```
fig, axes = plt.subplots(1, 2)
for i in range(2):
    axes[i].imshow(imgs[i])
```

In some situations, it would be neat to remove the ticks and add labels. **`axes.set_xticks([])`** and **`axes.set_yticks([])`** remove x-ticks and y-ticks, respectively. **`axes.set_xlabel('label')`** adds a label:

```
fig, axes = plt.subplots(1, 2)
labels = ['coast', 'beach']
for i in range(2):
    axes[i].imshow(imgs[i])
    axes[i].set_xticks([])
    axes[i].set_yticks([])
    axes[i].set_xlabel(labels[i])
```

b.

Creating Figures

You can use **`plt.figure()`** to create a new **Figure**. This function returns a Figure instance, but it is also passed to the backend. Every Figure-related command that follows is applied to the current Figure and does not need to know the Figure instance.

By default, the Figure has a width of 6.4 inches and a height of 4.8 inches with a **`dpi`** (dots per inch) of 100. To change the default values of the Figure, we can use the parameters **`figsize`** and **`dpi`**.

The following code snippet shows how we can manipulate a Figure:

```
plt.figure(figsize=(10, 5)) #To change the width and the height
plt.figure(dpi=300) #To change the dpi
```

Closing Figures

Figures that are no longer used should be closed by explicitly calling **`plt.close()`**, which also cleans up memory efficiently.

If nothing is specified, the **`plt.close()`** command will close the current Figure. To close a specific Figure, you can either provide a reference to a Figure instance or provide the Figure number. To find the **number** of a Figure object, we can make use of the **`number`** attribute, as follows:

```
plt.gcf().number
```

The **`plt.close('all')`** command is used to close all active Figures. The following example shows how a Figure can be created and closed:

```
plt.figure(num=10) #Create Figure with Figure number 10
plt.close(10) #Close Figure with Figure number 10
```

6.

a. **Seaborn Figure Styles**

To control the plot style, Seaborn provides two methods: **`set_style(style, [rc])`** and **`axes_style(style, [rc])`**.

`seaborn.set_style(style, [rc])` sets the aesthetic style of the plots.

Parameters:

- **`style`**: A dictionary of parameters or the name of one of the following preconfigured sets: **`darkgrid`**, **`whitegrid`**, **`dark`**, **`white`**, or **`ticks`**
- **`rc`** (optional): Parameter mappings to override the values in the preset Seaborn-style dictionaries

Here is an example:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("whitegrid")
plt.figure()
x1 = [10, 20, 5, 40, 8]
x2 = [30, 43, 9, 7, 20]
plt.plot(x1, label='Group A')
plt.plot(x2, label='Group B')
plt.legend()
plt.show()
```

This results in the following plot:

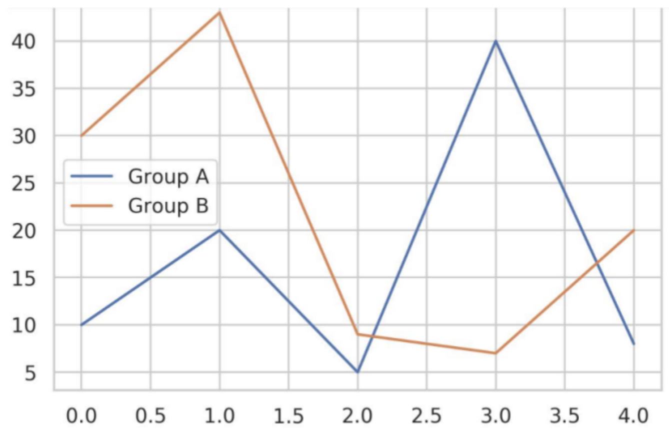


Figure 4.4: Seaborn line plot with whitegrid style

`seaborn.axes_style(style, [rc])` returns a parameter dictionary for the aesthetic style of the plots. The function can be used in a **`with`** statement to temporarily change the style parameters.

Here are the parameters:

- **`style`**: A dictionary of parameters or the name of one of the following pre-configured sets: **`darkgrid`**, **`whitegrid`**, **`dark`**, **`white`**, or **`ticks`**
- **`rc`** (optional): Parameter mappings to override the values in the preset Seaborn-style dictionaries

Here is an example:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
plt.figure()
x1 = [10, 20, 5, 40, 8]
x2 = [30, 43, 9, 7, 20]
with sns.axes_style('dark'):
    plt.plot(x1, label='Group A')
    plt.plot(x2, label='Group B')
plt.legend()
plt.show()
```

The aesthetics are only changed temporarily. The result is shown in the following diagram:

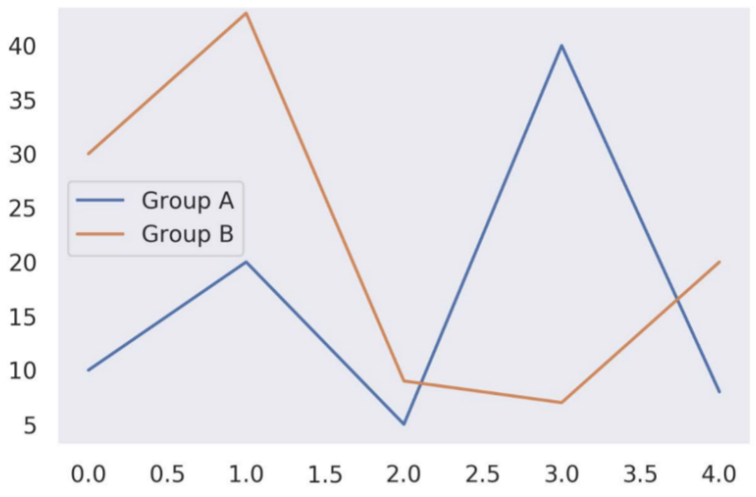


Figure 4.5: Seaborn line plot with dark axes style

Removing Axes Spines

Sometimes, it might be desirable to remove the top and right axes spines. The **`despine()`** function is used to remove the top and right axes spines from the plot:

```
seaborn.despine(fig=None, ax=None, top=True, right=True, left=False,
bottom=False, offset=None, trim=False)
```

The following code helps to remove the axes spines:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("white")
plt.figure()
x1 = [10, 20, 5, 40, 8]
x2 = [30, 43, 9, 7, 20]
plt.plot(x1, label='Group A')
plt.plot(x2, label='Group B')
sns.despine()
plt.legend()
plt.show()
```

This results in the following plot:

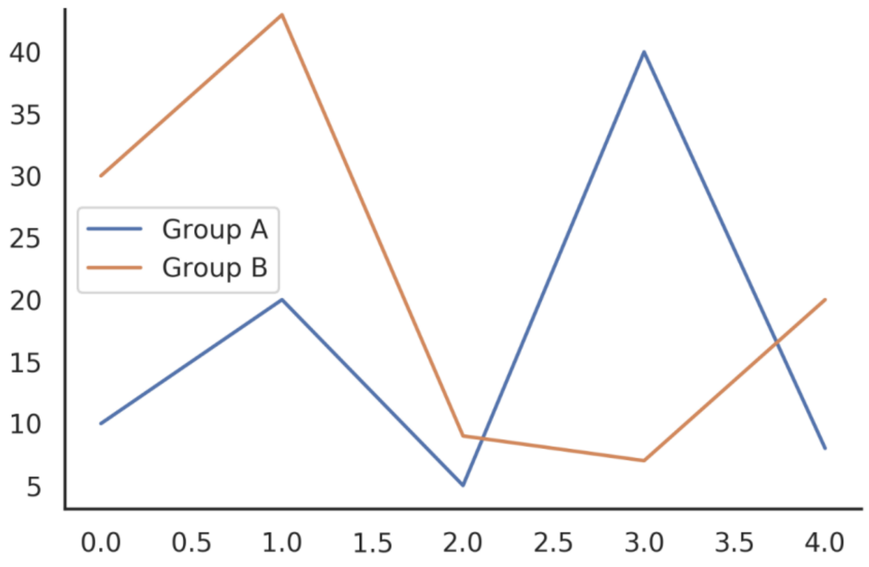


Figure 4.6: Despined Seaborn line plot

b.

Controlling the Scale of Plot Elements

A separate set of parameters controls the scale of plot elements. This is a handy way to use the same code to create plots that are suited for use in contexts where larger or smaller plots are necessary. To control the context, two functions can be used.

`seaborn.set_context(context, [font_scale], [rc])` sets the plotting context parameters. This does not change the overall style of the plot but affects things such as the size of the labels and lines. The base context is a **`notebook`**, and the other contexts are **`paper`**, **`talk`**, and **`poster`**—versions of the **`notebook`** parameters scaled by 0.8, 1.3, and 1.6, respectively.

Here are the parameters:

- **`context`**: A dictionary of parameters or the name of one of the following preconfigured sets: **`paper`**, **`notebook`**, **`talk`**, or **`poster`**
- **`font_scale`** (optional): A scaling factor to independently scale the size of font elements
- **`rc`** (optional): Parameter mappings to override the values in the preset Seaborn context dictionaries

The following code helps set the context:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_context("poster")
plt.figure()
x1 = [10, 20, 5, 40, 8]
x2 = [30, 43, 9, 7, 20]
plt.plot(x1, label='Group A')
plt.plot(x2, label='Group B')
plt.legend()
plt.show()
```

The preceding code generates the following output:

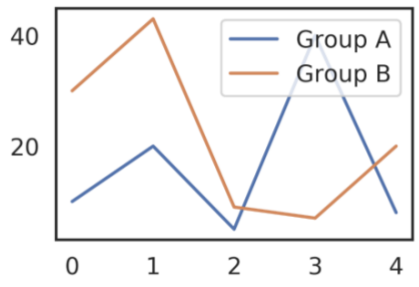


Figure 4.7: Seaborn line plot with poster context

`seaborn.plotting_context(context, [font_scale], [rc])` returns a parameter dictionary to scale elements of the Figure. This function can be used with a statement to temporarily change the context parameters.

Here are the parameters:

- **`context`**: A dictionary of parameters or the name of one of the following pre-configured sets: **`paper`**, **`notebook`**, **`talk`**, or **`poster`**
- **`font_scale`** (optional): A scaling factor to independently scale the size of font elements
- **`rc`** (optional): Parameter mappings to override the values in the preset Seaborn context dictionaries

Contexts are an easy way to use preconfigured scales of plot elements for different use cases. We will apply them in the following exercise, which uses a box plot to compare the IQ scores of different test groups.