**Department of Computer Science and Engineering**

|  |  |
| --- | --- |
| **Course Code:CSE422** | **Credits: 1.5** |
| **Course Name: Artificial Intelligence** | **Prerequisite:** CSE111, CSE221 |

**Lab 04  
Data Visualization and Statistical Analysis**

1. **Lab Overview:**

Learn Python programming for Data Visualization and Statistical Analysis with seaborn library.

1. **Why Python for AI course:**

AI (artificial intelligence) opens up a world of possibilities. By taking advantage of machine learning or deep learning, you could produce many fascinating applications.But, which programming language should you use? You want a language havingwide range of well documented libraries and a large community of programmers. Hence, whatever you want to do can be found in web as a reference. Python has all these advantages.

1. **Lesson Fit:**

There is pre-requisite to this lab: CSE111, CSE221. You should have intensive Programming Knowledge and capability of understand algorithms.

1. **Acceptance and Evaluation**

Performed lab tasks will be evaluated by the Lab Instructor (LI)

* 1. Short viva will be conducted in each Lab or occasionally to examine your work.
  2. You may work in groups but be aware that you will be evaluated individually; hence active participation during the Lab work demonstration is recommended.
  3. There will be Lab handout after your work you have to handover it to LI

1. **Learning Outcome:**

After this Lab, the students will be able to:

* 1. Understand basic python codes and solve basic programming problems in python

1. **Activity Detail**
   1. **Hour: 1  
      Getting Started:**
      1. Have a glance at Books “Python code for Artificial Intelligence: Foundations of Computational Agents,” by David L. Poole and Alan K. Mackworth, May 28, 2018
      2. “Artificial Intelligence with Python written by Prateek Joshi, January 2017
      3. Check \\TSR to see e-book copy and codes, tutorials and useful links
      4. http://seaborn.pydata.org/
      5. [**https://jakevdp.github.io/PythonDataScienceHandbook/04.14-visualization-with-seaborn.html**](https://jakevdp.github.io/PythonDataScienceHandbook/04.14-visualization-with-seaborn.html)
      6. http://seaborn.pydata.org/tutorial.html

**Installing Library**

To install library use the following command in Terminal/windows command prompt “pip install seaborn” or conda install seaborn

1. **Visualization with Seaborn**

Matplotlib has proven to be an incredibly useful and popular visualization tool, but Seaborn provides simple high-level functions for common statistical plot types, and integrates with the functionality provided by Pandas DataFrames.

**import matplotlib.pyplot as plt**

plt.style.use('classic')

%**matplotlib** inline

**import numpy as np**

**import pandas as pd**

*# Create some data*

rng = np.random.RandomState(0)

x = np.linspace(0, 10, 500)

y = np.cumsum(rng.randn(500, 6), 0)

*# Plot the data with Matplotlib defaults*

plt.plot(x, y)

plt.legend('ABCDEF', ncol=2, loc='upper left');

The main idea of Seaborn is that it provides high-level commands to create a variety of plot types useful for statistical data exploration, and even some statistical model fitting.

**import seaborn as sns**

sns.set()

**Histograms, KDE, and densities**

Often in statistical data visualization, all you want is to plot histograms and joint distributions of variables.

data = np.random.multivariate\_normal([0, 0], [[5, 2], [2, 2]], size=2000)

data = pd.DataFrame(data, columns=['x', 'y'])

**for** col **in** 'xy':

plt.hist(data[col], normed=**True**, alpha=0.5)

**kernel density estimation plot**

Rather than a histogram, we can get a smooth estimate of the distribution using a kernel density estimation, which Seaborn does with sns.kdeplot:

**for** col **in** 'xy':

sns.kdeplot(data[col], shade=**True**)

Histograms and KDE can be combined using distplot:

sns.distplot(data['x'])

sns.distplot(data['y']);

**Jointplot joint and marginal distributions**

We can see the joint distribution and the marginal distributions together using sns.jointplot. For this plot, we'll set the style to a white background:

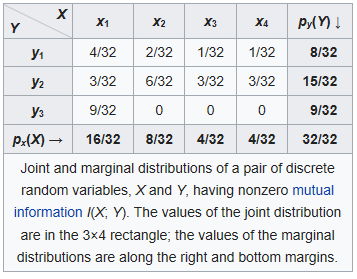
**with**sns.axes\_style('white'):

sns.jointplot("x", "y", data, kind='kde');

There are other parameters that can be passed to jointplot—for example, we can use a hexagonally based histogram instead:

**with**sns.axes\_style('white'):

sns.jointplot("x", "y", data, kind='hex')



**Faceted histograms**

Sometimes the best way to view data is via histograms of subsets. Seaborn's FacetGrid makes this extremely simple. We'll take a look at some data that shows the amount that restaurant staff receive in tips based on various indicator data:

In [14]:

tips = sns.load\_dataset('tips')

tips.head()

Out[14]:

|  | **total\_bill** | **tip** | **sex** | **smoker** | **day** | **time** | **size** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 16.99 | 1.01 | Female | No | Sun | Dinner | 2 |
| **1** | 10.34 | 1.66 | Male | No | Sun | Dinner | 3 |
| **2** | 21.01 | 3.50 | Male | No | Sun | Dinner | 3 |
| **3** | 23.68 | 3.31 | Male | No | Sun | Dinner | 2 |
| **4** | 24.59 | 3.61 | Female | No | Sun | Dinner | 4 |

tips['tip\_pct'] = 100 \* tips['tip'] / tips['total\_bill']

grid = sns.FacetGrid(tips, row="sex", col="time", margin\_titles=**True**)

grid.map(plt.hist, "tip\_pct", bins=np.linspace(0, 40, 15));

**Factor plots**

Factor plots can be useful for this kind of visualization as well. This allows you to view the distribution of a parameter within bins defined by any other parameter:

**with**sns.axes\_style(style='ticks'):

g = sns.factorplot("day", "total\_bill", "sex", data=tips, kind="box")

g.set\_axis\_labels("Day", "Total Bill");

**Joint distributions**

Similar to the pairplot we saw earlier, we can use sns.jointplot to show the joint distribution between different datasets, along with the associated marginal distributions:

**with**sns.axes\_style('white'):

sns.jointplot("total\_bill", "tip", data=tips, kind='hex')

**Regression and Decision**

The joint plot can even do some automatic kernel density estimation and regression:

sns.jointplot("total\_bill", "tip", data=tips, kind='reg');

**Bar plots**

Time series can be plotted using sns.factorplot. In the following example, we'll use the Planets data that we first saw in [Aggregation and Grouping](https://jakevdp.github.io/PythonDataScienceHandbook/03.08-aggregation-and-grouping.html):

planets = sns.load\_dataset('planets')

planets.head()

Out[19]:

|  | **method** | **number** | **orbital\_period** | **mass** | **distance** | **year** |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | Radial Velocity | 1 | 269.300 | 7.10 | 77.40 | 2006 |
| **1** | Radial Velocity | 1 | 874.774 | 2.21 | 56.95 | 2008 |
| **2** | Radial Velocity | 1 | 763.000 | 2.60 | 19.84 | 2011 |
| **3** | Radial Velocity | 1 | 326.030 | 19.40 | 110.62 | 2007 |
| **4** | Radial Velocity | 1 | 516.220 | 10.50 | 119.47 | 2009 |

withsns.axes\_style('white'):

g = sns.factorplot("year", data=planets, kind="count")

g.set\_xticklabels(step=5)

We can learn more by looking at the *method* of discovery of each of these planets:

In [21]:

withsns.axes\_style('white'):

g = sns.factorplot("year", data=planets, aspect=2.0, kind='count',

hue='method')

g.set\_ylabels('Number of Planets Discovered')

g.set\_xticklabels(step=2)

**Activity List**

**Task:Pair plots**

When you generalize joint plots to datasets of larger dimensions, you end up with *pair plots*. This is very useful for exploring correlations between multidimensional data, when you'd like to plot all pairs of values against each other.

We'll demo this with the well-known Iris dataset, which lists measurements of petals and sepals of three iris species:

fromsklearn.datasets import load\_iris

iris= load\_iris()

Convert the dataset to a pandasdataframe and make it similar like below

Hints: you can call specific array using their ‘key’ names, for an example: iris.data, irish.target

|  | **sepal\_length** | **sepal\_width** | **petal\_length** | **petal\_width** | **species** |
| --- | --- | --- | --- | --- | --- |
| **0** | 5.1 | 3.5 | 1.4 | 0.2 | setosa |
| **1** | 4.9 | 3.0 | 1.4 | 0.2 | setosa |
| **2** | 4.7 | 3.2 | 1.3 | 0.2 | setosa |
| **3** | 4.6 | 3.1 | 1.5 | 0.2 | setosa |
| **4** | 5.0 | 3.6 | 1.4 | 0.2 | setosa |

Visualizing the multidimensional relationships among the samples is as easy as calling sns.pairplot:

sns.pairplot(iris, hue='species', size=2.5);

Hints

[**https://jakevdp.github.io/PythonDataScienceHandbook/04.14-visualization-with-seaborn.html**](https://jakevdp.github.io/PythonDataScienceHandbook/04.14-visualization-with-seaborn.html)

**Task 01:** Mark 10 **Time:** 1.5 Hour

Exploring Marathon Finishing Times with seaborn

Here we'll look at using Seaborn to help visualize and understand results from a marathon dataset by loading it into Pandas:

data = pd.read\_csv('marathon-data.csv')

data.head()

|  | **age** | **gender** | **split** | **final** |
| --- | --- | --- | --- | --- |
| **0** | 33 | M | 01:05:38 | 02:08:51 |
| **1** | 32 | M | 01:06:26 | 02:09:28 |
| **2** | 31 | M | 01:06:49 | 02:10:42 |
| **3** | 38 | M | 01:06:16 | 02:13:45 |
| **4** | 31 | M | 01:06:32 | 02:13:59 |

Let's use converter for the times:

data['split'] = pd.to\_timedelta(data['split'])

data['final'] = pd.to\_timedelta(data['final'])

data.head()

|  | **age** | **gender** | **split** | **final** |
| --- | --- | --- | --- | --- |
| **0** | 33 | M | 01:05:38 | 02:08:51 |
| **1** | 32 | M | 01:06:26 | 02:09:28 |
| **2** | 31 | M | 01:06:49 | 02:10:42 |
| **3** | 38 | M | 01:06:16 | 02:13:45 |
| **4** | 31 | M | 01:06:32 | 02:13:59 |

data.dtypes

Out[26]:

age int64

gender object

split timedelta64[ns]

final timedelta64[ns]

dtype: object

That looks much better. For the purpose of our Seaborn plotting utilities, let's next add columns that give the times in seconds:

In [27]:

data['split\_sec'] = data['split'].astype(int) / 1E9

data['final\_sec'] = data['final'].astype(int) / 1E9

data.head()

Out[27]:

|  | **age** | **gender** | **split** | **final** | **split\_sec** | **final\_sec** |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | 33 | M | 01:05:38 | 02:08:51 | 3938.0 | 7731.0 |
| **1** | 32 | M | 01:06:26 | 02:09:28 | 3986.0 | 7768.0 |
| **2** | 31 | M | 01:06:49 | 02:10:42 | 4009.0 | 7842.0 |
| **3** | 38 | M | 01:06:16 | 02:13:45 | 3976.0 | 8025.0 |
| **4** | 31 | M | 01:06:32 | 02:13:59 | 3992.0 | 8039.0 |

To get an idea of what the data looks like, we can plot a jointplot over the data:

In [28]:

**with**sns.axes\_style('white'):

g = sns.jointplot("split\_sec", "final\_sec", data, kind='hex')

g.ax\_joint.plot(np.linspace(4000, 16000),

np.linspace(8000, 32000), ':k')

The dotted line shows where someone's time would lie if they ran the marathon at a perfectly steady pace. The fact that the distribution lies above this indicates (as you might expect) that most people slow down over the course of the marathon. If you have run competitively, you'll know that those who do the opposite—run faster during the second half of the race—are said to have "negative-split" the race.

Let's create another column in the data, the split fraction, which measures the degree to which each runner negative-splits or positive-splits the race:

In [29]:

data['split\_frac'] = 1 - 2 \* data['split\_sec'] / data['final\_sec']

data.head()

Out[29]:

|  | **age** | **gender** | **split** | **final** | **split\_sec** | **final\_sec** | **split\_frac** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 33 | M | 01:05:38 | 02:08:51 | 3938.0 | 7731.0 | -0.018756 |
| **1** | 32 | M | 01:06:26 | 02:09:28 | 3986.0 | 7768.0 | -0.026262 |
| **2** | 31 | M | 01:06:49 | 02:10:42 | 4009.0 | 7842.0 | -0.022443 |
| **3** | 38 | M | 01:06:16 | 02:13:45 | 3976.0 | 8025.0 | 0.009097 |
| **4** | 31 | M | 01:06:32 | 02:13:59 | 3992.0 | 8039.0 | 0.006842 |

Where this split difference is less than zero, the person negative-split the race by that fraction. Let's do a distribution plot of this split fraction:

In [30]:

sns.distplot(data['split\_frac'], kde=**False**);

plt.axvline(0, color="k", linestyle="--");

In [31]:

sum(data.split\_frac< 0)

Out[31]:

251

Out of nearly 40,000 participants, there were only 250 people who negative-split their marathon.

Let's see whether there is any correlation between this split fraction and other variables. We'll do this using a pairgrid, which draws plots of all these correlations:

In [32]:

g = sns.PairGrid(data, vars=['age', 'split\_sec', 'final\_sec', 'split\_frac'],

hue='gender', palette='RdBu\_r')

g.map(plt.scatter, alpha=0.8)

g.add\_legend();

It looks like the split fraction does not correlate particularly with age, but does correlate with the final time: faster runners tend to have closer to even splits on their marathon time. (We see here that Seaborn is no panacea for Matplotlib's ills when it comes to plot styles: in particular, the x-axis labels overlap. Because the output is a simple Matplotlib plot, however, the methods in [Customizing Ticks](https://jakevdp.github.io/PythonDataScienceHandbook/04.10-customizing-ticks.html) can be used to adjust such things if desired.)

The difference between men and women here is interesting. Let's look at the histogram of split fractions for these two groups:

In [33]:

sns.kdeplot(data.split\_frac[data.gender=='M'], label='men', shade=**True**)

sns.kdeplot(data.split\_frac[data.gender=='W'], label='women', shade=**True**)

plt.xlabel('split\_frac');

The interesting thing here is that there are many more men than women who are running close to an even split! This almost looks like some kind of bimodal distribution among the men and women. Let's see if we can suss-out what's going on by looking at the distributions as a function of age.

A nice way to compare distributions is to use a *violin plot*

In [34]:

sns.violinplot("gender", "split\_frac", data=data,

palette=["lightblue", "lightpink"]);

This is yet another way to compare the distributions between men and women.

Let's look a little deeper, and compare these violin plots as a function of age. We'll start by creating a new column in the array that specifies the decade of age that each person is in:

In [35]:

data['age\_dec'] = data.age.map(**lambda** age: 10 \* (age // 10))

data.head()

Out[35]:

|  | **age** | **gender** | **split** | **final** | **split\_sec** | **final\_sec** | **split\_frac** | **age\_dec** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 33 | M | 01:05:38 | 02:08:51 | 3938.0 | 7731.0 | -0.018756 | 30 |
| **1** | 32 | M | 01:06:26 | 02:09:28 | 3986.0 | 7768.0 | -0.026262 | 30 |
| **2** | 31 | M | 01:06:49 | 02:10:42 | 4009.0 | 7842.0 | -0.022443 | 30 |
| **3** | 38 | M | 01:06:16 | 02:13:45 | 3976.0 | 8025.0 | 0.009097 | 30 |
| **4** | 31 | M | 01:06:32 | 02:13:59 | 3992.0 | 8039.0 | 0.006842 | 30 |

In [36]:

men = (data.gender == 'M')

women = (data.gender == 'W')

**with**sns.axes\_style(style=**None**):

sns.violinplot("age\_dec", "split\_frac", hue="gender", data=data,

split=**True**, inner="quartile",

palette=["lightblue", "lightpink"]);

Looking at this, we can see where the distributions of men and women differ: the split distributions of men in their 20s to 50s show a pronounced over-density toward lower splits when compared to women of the same age (or of any age, for that matter).

Also surprisingly, the 80-year-old women seem to outperform *everyone* in terms of their split time. This is probably due to the fact that we're estimating the distribution from small numbers, as there are only a handful of runners in that range:

In [38]:

(data.age> 80).sum()

Out[38]:

7

Back to the men with negative splits: who are these runners? Does this split fraction correlate with finishing quickly? We can plot this very easily. We'll use regplot, which will automatically fit a linear regression to the data:

In [37]:

g = sns.lmplot('final\_sec', 'split\_frac', col='gender', data=data,

markers=".", scatter\_kws=dict(color='c'))

g.map(plt.axhline, y=0.1, color="k", ls=":");

Apparently the people with fast splits are the elite runners who are finishing within ~15,000 seconds, or about 4 hours. People slower than that are much less likely to have a fast second split.