LAB 2 - Data Analysis using Hypothesis Testing in Python using PANDA

Hypothesis testing is a statistical method used to make inferences about a population based on a sample of data. In the context of big data analysis, hypothesis testing can be used to test whether a certain feature or characteristic of the data is statistically significant or not. In this lab execises, you will see how hypothesis testing can be performed using Python and Pandas.

Pandas is not included with the standard Python installation, so you will need to install it separately. One way to install Pandas is by using a package manager such as pip or conda.

For example, to install Pandas using conda in Anaconda PowerShell Prompt, you can run the following command in your terminal or command prompt:

```
conda install pandas
```

Once installed, you can import the Pandas library in your Python code and use it to manipulate and analyze data.

On the other hand, Scipy is a scientific computing library in Python that provides many useful functions for statistical analysis, including the **scipy.stats** module which contains a wide variety of probability distributions and statistical functions.

If you want to use the statistical functions provided by **scipy.stats**, you will need to have Scipy installed on your system. Scipy can be installed using pip or conda, similar to Pandas.

To install Scipy using `conda`, you can run the following command in your Anaconda prompt:

```
conda install scipy
```

Exercise 2-1

Let's say you have a dataset containing information about customers of an e-commerce company, including their age and purchase frequency. You want to test the hypothesis that there is a significant relationship between customer age and purchase frequency.

First, you would import the necessary Python libraries, such as pandas for data manipulation and scipy.stats for statistical analysis:

```
import pandas as pd
from scipy.stats import pearsonr
```

Next, you would load the dataset into a pandas dataframe:

```
df = pd.read_csv('customer_data.csv')
```

You would then clean and preprocess the data as necessary. For example, you might remove any missing values or outliers:

```
df.dropna(inplace=True)
```

```
df = df[df['age'] > 18]
```

Next, you would run a Pearson correlation test to determine if there is a significant correlation between customer age and purchase frequency:

```
corr, p_value = pearsonr(df['age'],
df['purchase_frequency'])
print("Correlation:", corr)
print("P-value:", p_value)
```

If the p-value is less than the significance level (usually 0.05), you can reject the null hypothesis and accept the alternative hypothesis, indicating that there is a significant relationship between customer age and purchase frequency.

Finally, you would interpret the results and communicate your findings to stakeholders in the company. For example, you might create a scatterplot to visualize the relationship between age and purchase frequency:

```
import matplotlib.pyplot as plt
plt.scatter(df['age'], df['purchase_frequency'])
plt.xlabel('Age')
plt.ylabel('Purchase Frequency')
plt.show()
```

This could help inform marketing strategies or customer segmentation based on age.

Exercise 2-2: Data statistics

NumPy is a powerful library for numerical computing in Python and is commonly used for data statistics. NumPy provides many functions for performing various statistical operations on arrays of data, such as mean, median, variance, standard deviation, correlation, and regression.

Here are some examples of how NumPy can be used for statistical analysis:

```
import numpy as np

# create a sample array of data
data = np.array([2, 4, 6, 8, 10])

# compute the mean of the data
mean = np.mean(data)

# compute the median of the data
median = np.median(data)

# compute the variance of the data
variance = np.var(data)

# compute the standard deviation of the data
stddev = np.std(data)

# compute the correlation between two arrays of data
```

```
x = np.array([1, 2, 3, 4, 5])
y = np.array([2, 4, 6, 8, 10])
correlation = np.corrcoef(x, y)[0, 1]

# perform a linear regression on two arrays of data
slope, intercept = np.polyfit(x, y, 1)
print("Mean:", mean)
print("Median:", median)
print("Variance:", variance)
print("Standard deviation:", stddev)
print("Correlation:", correlation)
print("Slope:", slope)
print("Intercept:", intercept)
```

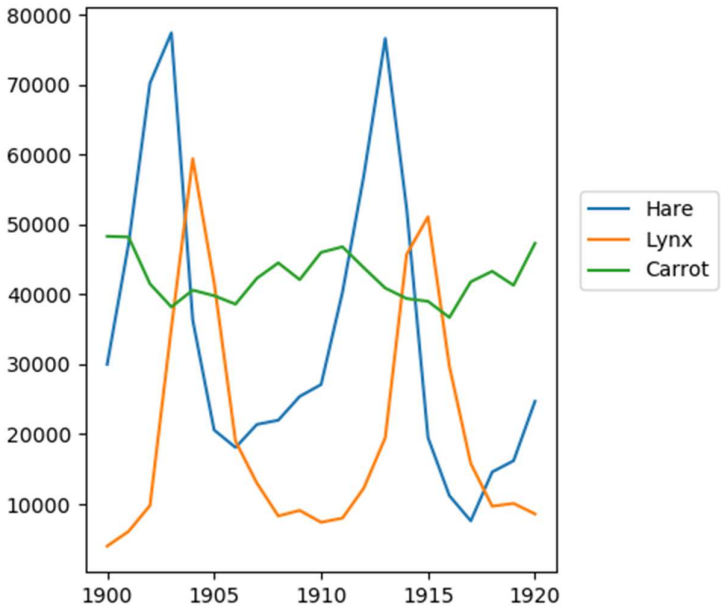
In the above example, we create a sample array of data and use various NumPy functions to compute its mean, median, variance, standard deviation, correlation, and perform a linear regression on two arrays of data. These are just a few examples of what NumPy can do for statistical analysis; there are many other functions available for different types of analyses.

Now, let us conduct data analysis using numpy with data in a file. In this exercise, we use the data in **populations.txt** that describes the populations of hares and lynxes (and carrots) in northern Canada during 20 years**. In the python console**, do the following to conduct simple analysis.

```
>>>
```

```
>>> data = np.loadtxt('populations.txt')
>>> year, hares, lynxes, carrots = data.T  # trick: columns to variables

>>> import matplotlib.pyplot as plt
>>> plt.axes([0.2, 0.1, 0.5, 0.8])
<matplotlib.axes...Axes object at ...>
>>> plt.plot(year, hares, year, lynxes, year, carrots)
[<matplotlib.lines.Line2D object at ...>, ...]
>>> plt.legend(('Hare', 'Lynx', 'Carrot'), loc=(1.05, 0.5))
<matplotlib.legend.Legend object at ...>
>>> plt.show()
```



Computes and print, based on the data in populations.txt…

1. The mean and std of the populations of each species for the years in the period.
2. Which year each species had the largest population. (Hint: argmax)
3. Which species has the largest population for each year. (Hint: argsort & fancy indexing of np.array(['H', 'L', 'C']))
4. Which years any of the populations is above 50000. (Hint: comparisons and np.any)
5. The top 2 years for each species when they had the lowest populations. (Hint: argsort, fancy indexing)
6. Compare (plot) the change in hare population (see help(np.gradient)) and the number of lynxes. Check correlation (see help(np.corrcoef)).

… all without for-loops.

Exercise 2-3 Polynomials

NumPy also contains polynomials in different bases:

For example, $3x^2 + 2x - 1$.

Using the console, try the following commands.

```
>>>
```

```
>>> p = np.poly1d([3, 2, -1])
>>> p(0)
-1
>>> p.roots
array([-1.        ,  0.33333333])
>>> p.order
2
```

Then, try the following codes

```
import numpy as np
import matplotlib.pyplot as plt

# set the random seed for reproducibility
np.random.seed(12)

# generate some noisy data
x = np.linspace(0, 1, 20)
y = np.cos(x) + 0.3*np.random.rand(20)

# fit a polynomial of degree 3 to the data
p = np.poly1d(np.polyfit(x, y, 3))

# generate a set of evenly-spaced points for the plot
t = np.linspace(0, 1, 200)

# plot the data and the fitted polynomial
plt.plot(x, y, 'o', t, p(t), '-')
plt.show()
```

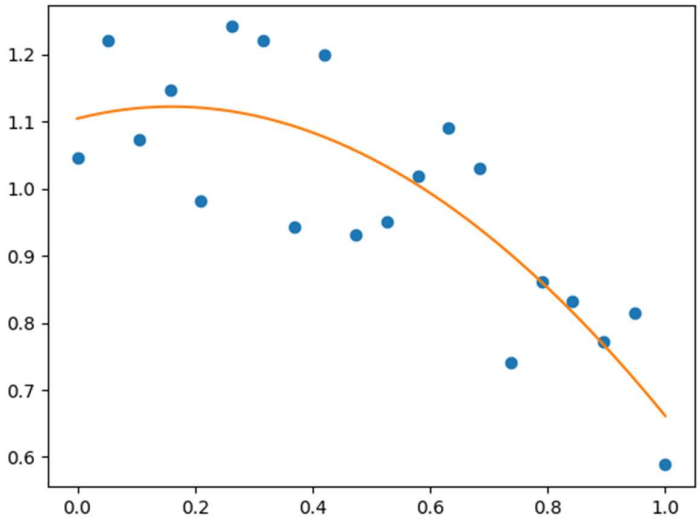
First, we set the random seed using **np.random.seed(12)** to ensure that the same random numbers are generated every time we run the code.

Then, we generate some noisy data using the **np.linspace** function to create an array of 20 evenly-spaced values between 0 and 1 for **x**, and the **np.cos** function to create corresponding values of **y**. We add some random noise to **y** using **0.3*np.random.rand(20)**.

Next, we use NumPy's **np.polyfit** function to fit a polynomial of degree 3 to the data points **(x, y)**. The **np.poly1d** function creates a polynomial function from the fitted coefficients.

We then generate a set of evenly-spaced points **t** between 0 and 1 using **np.linspace**, and use the **p(t)** function to evaluate the fitted polynomial at each of these points.

Finally, we plot the data points as circles (**'o'**), and the fitted polynomial as a solid line (**'-'**) using Matplotlib's **plt.plot** function. The resulting plot should show the noisy data points and the smooth curve of the fitted polynomial.


1.2
1.1
1.0
0.9
0.8
0.7
0.6
0.0
0.2
0.4
0.6
0.8
1.0