A0218869J–HaojunGao

August 15, 2020

1 Individual Assignment 1

(Due on Aug 25 11:59PM)

1.1 Part I: Simulation

In this exercise, we will use simulation to illustrate the variability of statistics calculated from random samples. Suppose there is a normal population of size $N = 10000$, with mean $\mu = 100$ and standard deviation $\sigma = 15$. Now we draw a sample from the population, of size $n = 100$ **with replacement**, we can calculate sample statistics such as mean and variance. If we further repeat the sampling process many times, say 200, we will have 200 sets of similar sample statistics. Let's examine these sample statistics.

The necessary parameters are already set up as below.

[3]:
```
pop_size = 10000
pop_mean = 100
pop_sd = 15
num_of_samples = 200
sample_size = 100
```

1.1.1 Questions and Answers

1. Use random seed **1234** to conduct the simulation (i.e., simulate the population as specified, draw 200 samples (each of size 100), and calculate sample mean and variance for each sample, respectively), evaluate the mean and standard deviation of the sample statistics, and compare with their theoretical values. Draw histograms of the sample mean and sample variance respectively.

REMARK 1: Recall that, according to the Central Limit Theorem, the sample mean

$$\bar{X} \sim \mathrm{N}\left(\mu, \frac{\sigma^2}{n}\right).$$

REMARK 2: Recall that theoretically, the sample variance S^2 satisfies

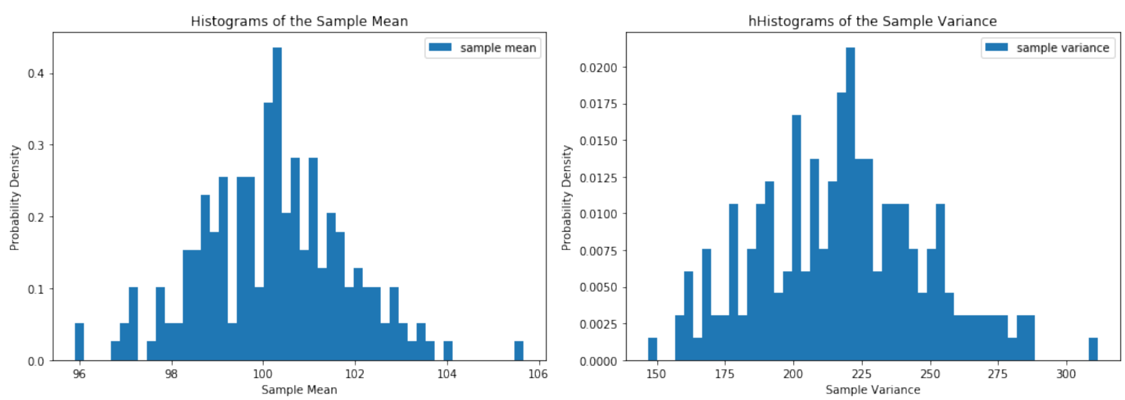
$$\mathrm{E}[S^2] = \sigma^2, \quad \mathrm{Var}[S^2] = \frac{2\sigma^4}{n-1}, \quad \text{and} \quad \frac{(n-1)S^2}{\sigma^2} \sim \chi^2_{n-1}.$$

Answer:

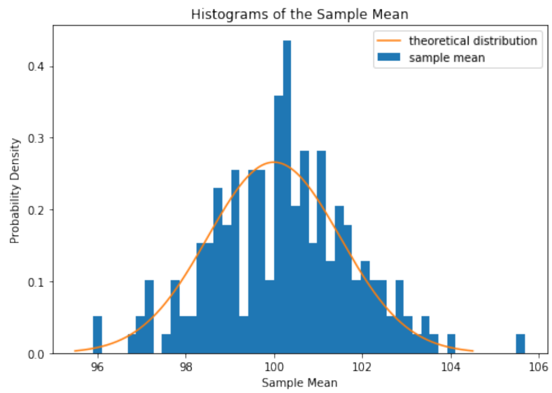
[12]:
```python
def histograms(sample_mean_lst, sample_var_lst, pop_mean, pop_sd, sample_size,␣
 ↪is_distribution=True, num_bins = 50, figsize=(14, 5)):

    fig = plt.figure(figsize=figsize)

    # plot sample mean
    ax1 = plt.subplot(121)
    ax1.hist(sample_mean_lst, num_bins, density=1, label='sample mean')
    ax1.set_xlabel('Sample Mean')
    ax1.set_ylabel('Probability Density')
    ax1.set_title('Histograms of the Sample Mean')

    # plot sample variance
    ax2 = plt.subplot(122)
    ax2.hist(sample_var_lst, num_bins, density=1, label='sample variance')
    ax2.set_xlabel('Sample Variance')
    ax2.set_ylabel('Probability Density')
    ax2.set_title('Histograms of the Sample Variance')

    if is_distribution:
        # plot theoretical distribution for figure 1
        mu, sigma = pop_mean, pop_sd/(sample_size**0.5)
        x = np.linspace(mu - 3 * sigma, mu + 3 * sigma, 1000)
        y = norm.pdf(x, loc=mu, scale=sigma)
        ax1.plot(x, y, label='theoretical distribution')

        # plot theoretical distribution for figure 2
        df = sample_size - 1
        loc = 0
        scale = (pop_sd**2)/df
        x = np.linspace(chi2.ppf(0.005, df, loc, scale),
                        chi2.ppf(0.995, df, loc, scale), 1000)
        y = chi2.pdf(x, df, loc, scale)
        plt.plot(x, y, label='theoretical distribution')

    # set plot details
    ax1.legend()
    ax2.legend()
    fig.tight_layout()
    plt.show()
```

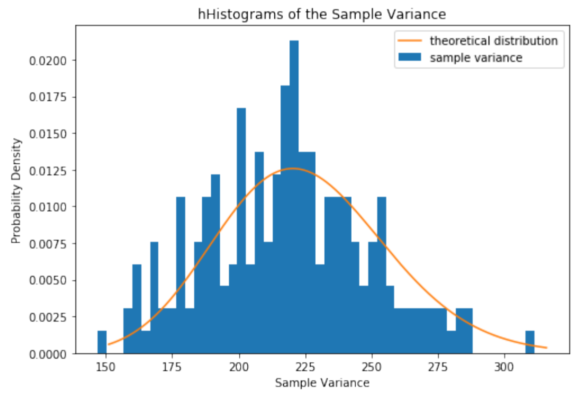
[14]:
```python
# import package
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm, chi2

# set random seed
```

```
np.random.seed(1234)

# simulate the population
random_data = np.random.normal(loc=pop_mean, scale=pop_sd, size=pop_size)

# draw 200 samples and calculate the sample mean and variance for each sample
samples_lst = [np.random.choice(
    random_data, replace=True, size=sample_size) for i in range(num_of_samples)]
sample_mean_lst = [sample.mean() for sample in samples_lst]
sample_var_lst = [sample.var() for sample in samples_lst]

# Draw histograms of the sample mean and sample variance respectively
histograms(sample_mean_lst, sample_var_lst, pop_mean, pop_sd, sample_size,␣
↪is_distribution = False)

# for comparison, you may just report the theoretical mean/sd of the sample␣
↪mean and sample variance respectively.
print('Theoretical mean of the sample mean: {}'.format(pop_mean))
print('Theoretical standard deviation of the sample mean: {}'.format(pop_sd/
↪(sample_size**0.5)))

print('Theoretical mean of the sample variance: {}'.format(pop_sd**2))
print('Theoretical standard deviation of the sample variance: {}'.
↪format(((pop_sd**0.5) * (pop_sd**2))/(sample_size - 1)))

# overlay the theoretical distribution (not required in the assignment)
histograms(sample_mean_lst, sample_var_lst, pop_mean, pop_sd, sample_size,␣
↪is_distribution = True)
```




```
Theoretical mean of the sample mean: 100
Theoretical standard deviation of the sample mean: 1.5
Theoretical mean of the sample variance: 225
Theoretical standard deviation of the sample variance: 8.80223487774413
```





1.2 Part II: K-Nearest Neighbor Algorithm

1.2.1 Introduction

In this assignment, we are going to experiment the K-Nearest Neighbor (KNN) algorithm on a higher-dimensional dataset and experience the deterioration of prediction performance as the dimensionality grows.

The experiment is built on top of the 3rd-order polynomial model:

$$y = \beta_0 + \beta_1 * x + \beta_2 * x^2 + \beta_3 * x^3 + \epsilon, \quad \epsilon \sim \mathrm{N}(0, \sigma^2)$$

and we are going to introduce an extra 20-dimensional predictor z, which does NOT actually play a role in generating y. Yet, when in estimation, we do not know the fact and will use both x and z as predictors in the KNN algorithm.

1.2.2 Generation of the high-dimensional dataset

We first simulate the 3rd-order polynomial datasets

[16]:
```
import numpy as np
## population parameters
beta0 = 1
beta1 = -2
beta2 = 6
beta3 = -1
sigma = 2

np.random.seed(7890)

## training data
x = np.arange(0, 5, 0.05)
f_x = beta0 + beta1 * x + beta2 * x**2 + beta3 * x**3
epsilon = np.random.normal(loc=0, scale=sigma, size=100)
y = f_x + epsilon
```

```
## test data
x_test = np.arange(0, 5.1, 0.1)
f_x_test = beta0 + beta1 * x_test + beta2 * x_test**2 + beta3 * x_test**3
epsilon_test = np.random.normal(loc=0, scale=sigma, size=len(x_test))
y_test = f_x_test + epsilon_test
```

The resulted training and test dataset have **100** and **51** data points, respectively.

Next, we need to generate z, the 20-dimensional predictors, of the same sizes. Each z is a 20-dimensional multivariate normal random variable, with mean being $(0, 0, \ldots, 0)$ and identity covariance matrix (so that the 20 elements are independent standard normal random variables). The resulted z is a 100*20 matrix, with each row being a data point with 20 dimensions.

[17]:
```
np.random.seed(7891)
z = np.random.multivariate_normal(size = 100, mean = [0]*20, cov = np.
 ↪identity(20))
z_test = np.random.multivariate_normal(size = 51, mean = [0]*20, cov = np.
 ↪identity(20))
```

Later, we will use (x, z) to predict y. Let's first combine x and z into matrices

[59]:
```
train_x = np.concatenate((np.expand_dims(x, axis = 1),z),axis = 1)
test_x =  np.concatenate((np.expand_dims(x_test, axis = 1),z_test),axis = 1)
```

1.2.3 Questions and Answers

1. For a fixed $k = 15$, fit a KNN model to predict y with (x, z), and measure the training and test MSE. Answer:

[60]:
```
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error as mse

# set fixed parameter
fix_k = 15

# training model
knn = KNeighborsRegressor(n_neighbors=fix_k)
knn.fit(train_x, y)

# predict and measure the performance
y_predict = knn.predict(test_x)
mse_predict = mse(y_test, y_predict)
print(f'Model Performance (MSE): {mse_predict}')
```

```
Model Performance (MSE): 32.94639319047726
```

2. With the same data, plot the training and test MSE of the KNN model against k, and find the optimal k and the corresponding test MSE. Answer:

[94]:
```
def train_model(train_x, test_x, y, y_test, is_print = True):
    k = 0
    # Initialize
    optimal_mse, currenct_mse = 0,0
    while optimal_mse == currenct_mse:
        # set k for this training round
        k += 1
        # train & predict & measure
        knn = KNeighborsRegressor(n_neighbors=k)
        knn.fit(train_x, y)
        y_predict = knn.predict(test_x)
        current_mse = mse(y_test, y_predict)
        optimal_mse = current_mse if current_mse <= optimal_mse or k == 1␣
↪else optimal_mse
        if is_print: print(f'Parameter k = {k}; Model Performance (MSE):␣
↪{current_mse}')
    return k-1, optimal_mse
```

[62]:
```
# stop looking for the optimal k when test MSE no longer decreases as k goes up.
k, optimal_mse = train_model(train_x, test_x, y, y_test)
print(f'Optimal k: {k}\nTest MSE: {optimal_mse}')
```

```
Parameter k = 1; Model Performance (MSE): 76.76871868226493
Parameter k = 2; Model Performance (MSE): 50.63365314897203
Parameter k = 3; Model Performance (MSE): 38.01708840946738
Parameter k = 4; Model Performance (MSE): 34.54505042921676
Parameter k = 5; Model Performance (MSE): 29.643836178903474
Parameter k = 6; Model Performance (MSE): 32.15296586511268
Optimal k: 5
Test MSE: 29.643836178903474
```

3. Based on the analysis above, compare the above model with (x, z) being the predictors and the previous model with x only. Briefly explain why. Answer:

[63]:
```
# transfer the x and x_test to 2D array
train_x_only = np.expand_dims(x, axis = 1)
test_x_only =  np.expand_dims(x_test, axis = 1)
```

[64]:
```
k, optimal_mse = train_model(train_x_only, test_x_only, y, y_test)
print(f'Optimal k: {k}\nTest MSE: {optimal_mse}')
```

```
Parameter k = 1; Model Performance (MSE): 8.00997691055793
Parameter k = 2; Model Performance (MSE): 5.44808287318148
Parameter k = 3; Model Performance (MSE): 5.426316500001306
Parameter k = 4; Model Performance (MSE): 4.843594490271816
Parameter k = 5; Model Performance (MSE): 4.367417625187504
Parameter k = 6; Model Performance (MSE): 4.93632176506213
```

```
Optimal k: 5
Test MSE: 4.367417625187504
```

Explaination: Curse of Dimensionality

- The distance between neighbors is dominated by a large number of unrelated attributes.
- In this case, instances whose values of two related attributes are consistent may be far apart in the 20-dimensional space.
- As the result, relying on the similarity measure of these 20 attributes can mislead the k-nearest neighbor algorithm.

4. We have seen that the test MSE is significantly worse than what we had without using predictor z. To better understand the impact of including irrelevant predictors in the KNN algorithm, let's try to include the 20 dimensions of z one by one. So in each round j, we construct the predictors by combining x and the first j columns of z, then repeat the analysis in Question 2 and find the optimal k and test MSE. At the end, plot the optimal MSE agaist j, and interpret the result. Answer:

[103]:
```
for j in range(len(z[0]) + 1):
    train_x_incremental = np.concatenate(
        (np.expand_dims(x, axis=1), z[:, :j]), axis=1)
    test_x_incremental = np.concatenate(
        (np.expand_dims(x_test, axis=1), z_test[:, :j]), axis=1)
    k, optimal_mse = train_model(
        train_x_incremental, test_x_incremental, y, y_test, is_print=False)
    print(
        f'Add first {j} columns \t Optimal k: {k} \t Test MSE: {optimal_mse}')
```

```
Add first 0 columns     Optimal k: 5    Test MSE: 4.367417625187504
Add first 1 columns     Optimal k: 5    Test MSE: 7.2142500345041976
Add first 2 columns     Optimal k: 4    Test MSE: 13.359619107018597
Add first 3 columns     Optimal k: 2    Test MSE: 15.263100962227195
Add first 4 columns     Optimal k: 5    Test MSE: 16.674825717732034
Add first 5 columns     Optimal k: 2    Test MSE: 17.879872756941463
Add first 6 columns     Optimal k: 2    Test MSE: 24.436585793561473
Add first 7 columns     Optimal k: 5    Test MSE: 22.42699951576807
Add first 8 columns     Optimal k: 2    Test MSE: 26.47237779960154
Add first 9 columns     Optimal k: 8    Test MSE: 24.642352439593413
Add first 10 columns    Optimal k: 7    Test MSE: 23.370264787676305
Add first 11 columns    Optimal k: 6    Test MSE: 21.906956923650522
Add first 12 columns    Optimal k: 4    Test MSE: 25.543163075510957
Add first 13 columns    Optimal k: 6    Test MSE: 27.054482241553522
Add first 14 columns    Optimal k: 7    Test MSE: 24.069530259757233
Add first 15 columns    Optimal k: 7    Test MSE: 23.60896179380117
Add first 16 columns    Optimal k: 4    Test MSE: 27.17057594236693
Add first 17 columns    Optimal k: 5    Test MSE: 26.178734436435576
Add first 18 columns    Optimal k: 6    Test MSE: 32.814828831731226
Add first 19 columns    Optimal k: 3    Test MSE: 31.48016670629021
```

```
Add first 20 columns     Optimal k: 5    Test MSE: 29.643836178903474
```

Explaination: Curse of Dimensionality

- Result basically follows: the more unrelated dimensions added, the worse the model's performance.
- The KNN's rationale is that using sample's neighbours(using distance to measure) to represent itself in that similar samples are more likely to be the same category.
- The above result shows that unrelated dimensions undermine the process of defining the neighbor, in other words, the similarity between samples is more difficult to describe with distance.