KAIST MFE, 2024 Fall

Kim Hyeonghwan

2024-09-02

Table of contents

Welcome!

안녕하세요, KAIST MFE 24년 가을학기에 이수한 과목의 과제 등을 정리해두었습니다.

Part I

머신러닝('24 가을)

머신러닝 Ch1

머신러닝 기초

하이퍼파라미터

머신러닝에 이용할 모델에 대한 파리미터(α, β 등)가 아닌,

학습알고리즘의 파라미터(학습률 등)

$\hat{y} = \beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k$

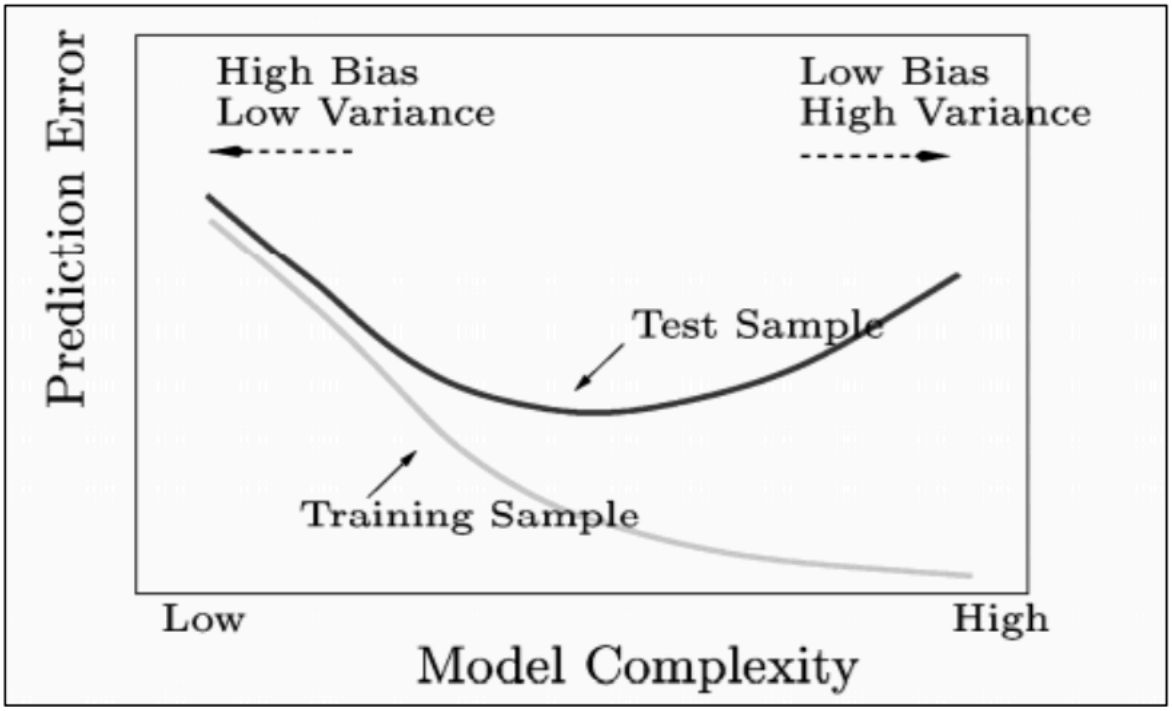
여기서, β_n은 파라미터이며 주어진 데이터를 학습하여 파라미터를 산출하는 것임.

근데 만약에 모델 성능 향상을 위해 각 β의 제약조건(constraint)를 정한다?

해당 제약조건은 하이퍼파라미터(hyper-parameter, h-para)가 되는 것임

이런 회귀분석을 릿지(Ridge regression)이라고 함.

모델의 평가와 검증



낮은 복잡도 = 선형회귀분석 or logistic 분류면 높은 복잡도 = 변수를 추가한 모델 (과대적합 케이스)

훈련데이터(training sample)은 복잡도가 높아질수록 예측오차가 줄어듬 (우하향)

평가데이터(text sample)은 복잡도가 높아지면 오차가 줄어들기는 하지만,

너무 복잡도가 높아지면 평가데이터에서는 오차가 오히려 발생함

즉, 일반화가 어렵고 과대적합(overfitting) 문제가 발생함

일반화 오차

평가데이터를 이용하였을 때 발생하는 오차를 **일반화 오차**라고 함

(Generalization error, test error)

$$= \ ^2 + \ +$$

편향(Bias)

모집단에서 크기 m의 (x,y) 순서쌍을 샘플링할 때,

해당 샘플링을 n번 반복해서 모델링을 한다고 하면,

각각의 $f_1, ..., f_n$이 있을 것이고, $\bar{f} = mean(f_m)$이면,

실제 모집단을 나타내는 모델인 f_{true}와 $\bar{f}$의 차이를 편향(bias)이라고 합니다.

분산(Variance)

한편, $f_1, ..., f_n$의 추정모델간의 편차의 제곱합이 분산이 됩니다.

관계

즉, 모델이 단순할수록 실제로는 더 복잡한 모델을 잘 반영하기 어렵기때문에 편향이 큰 대신,

추정모델간의 오차는 작아지므로 분산이 작습니다.

하지만, 모델이 복잡할수록 추정모델을 평균하면 실제 모델과 유사해질 것 이므로 편향은 작고,

추정모델간의 오차는 클 것이므로 분산이 큽니다.

데이터의 분할 방법

Hold-out 방식

주어진 자료를 목적에 따라 훈련/검증/평가 데이터로 나누어서 활용.

(훈련, 검증이 8~90% / 평가가 1~20%)

검증데이터는 h-para tuning에 주로 사용함.

1. 각 h-para별로 훈련데이터를 통해 모델 도출
2. 각 모델에 대해 검증데이터를 이용해 평가 (MSE 산출)
3. 성능이 가장 좋은 h-para를 채택
4. 해당 h-para 및 훈련+검증데이터를 통해 최종모델 도출
5. 평가데이터를 이용해 최종모델을 평가하여 성능 확인

단점 : 전체 데이터에서 평가데이터는 따로 빼놔야해서 자료가 충분치 않으면 사용하기 애매함

K-fold 교차검증(Cross-validation) 방식을 이용한 검증

데이터가 그다지 많지 않을때 유용.

모든 데이터가 훈련, 검증, 평가에 활용될 수 있음.

주어진 자료를 K개로 분할하여 반복활용

(3-fold cv 예시)

1. 주어진 자료를 3개로 분할 (1,2 훈련 + 3 검증 / 1,3 훈련 + 2 검증 / 2,3 훈련 + 1 검증)
2. 각 분할데이터로 특정 h-para에 대해 훈련 + 검증데이터로 성능 평가(MSE)
3. 3개의 분할데이터의 성능의 평균이 해당 h-para의 검증결과임
4. 모든 h-para에 대해 1~3 반복
5. h-para의 검증결과 중 가장 성능이 좋은 h-para 채택
6. 다시 주어진 자료를 3개로 분할 (훈련+평가)
7. 각 분할데이터로 훈련 및 평가를 통해 성능 평가
8. 성능의 평균값이 우리의 모델의 성능임.

방법론에 따라 한꺼번에 훈련시켜서 성능을 평가하기도 하고,

이러한 분할(folding)을 수회~수백회 반복해서 모델의 성능을 추정하기도 함.

(folding별 성능의 평균/표준편차 고려)

> 머신러닝으로 분류문제를 해결하는 경우,
> 실제 세상에서는 분류대상의 비율이 매우 적은 경우가 많음.
> 이러한 샘플을 imbalanced data라고 하며,
> Hold-out, K-fold cv 등을 할 때,
> 원 자료의 분류대상의 비율을 유지한채로 주어진 자료를 분할해야 함.

1 머신러닝 실습

1.1 Ch2. Linear regression

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.datasets import load_diabetes
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
```

```
diabetes = load_diabetes()
diabetes_DF = pd.DataFrame( diabetes['data'], columns=diabetes['feature_names'])
diabetes_DF['Y']=diabetes['target']
diabetes_DF.head(5)
```

	age	sex	bmi	bp	s1	s2	s3	s4	s5	s6
0	0.038076	0.050680	0.061696	0.021872	-0.044223	-0.034821	-0.043401	-0.002592	0.019907	-0.01
1	-0.001882	-0.044642	-0.051474	-0.026328	-0.008449	-0.019163	0.074412	-0.039493	-0.068332	-0.09
2	0.085299	0.050680	0.044451	-0.005670	-0.045599	-0.034194	-0.032356	-0.002592	0.002861	-0.02
3	-0.089063	-0.044642	-0.011595	-0.036656	0.012191	0.024991	-0.036038	0.034309	0.022688	-0.00
4	0.005383	-0.044642	-0.036385	0.021872	0.003935	0.015596	0.008142	-0.002592	-0.031988	-0.04

```
diabetes_DF.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 442 entries, 0 to 441
Data columns (total 11 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   age     442 non-null    float64
 1   sex     442 non-null    float64
```

```
 2   bmi     442 non-null    float64
 3   bp      442 non-null    float64
 4   s1      442 non-null    float64
 5   s2      442 non-null    float64
 6   s3      442 non-null    float64
 7   s4      442 non-null    float64
 8   s5      442 non-null    float64
 9   s6      442 non-null    float64
 10  Y       442 non-null    float64
dtypes: float64(11)
memory usage: 38.1 KB
```

```
y_target = diabetes_DF['Y']
X_data = diabetes_DF.drop(['Y'], axis=1, inplace=False)
X_train, X_test, y_train, y_test = train_test_split(
X_data, y_target, test_size=0.4, random_state=123 )
```

```
lr = LinearRegression()
lr.fit ( X_train, y_train )
```

```
LinearRegression()
```

```
lr.intercept_
```

```
151.71551041484278
```

```
np.round( lr.coef_, decimals=1)
```

```
array([ -11.1, -291.1,  553.8,  296.6, -915. ,  528.4,  210.2,  339.6,
        640.6,  115.7])
```

```
coeff = pd.Series( data= np.round( lr.coef_, decimals=1), index=X_data.columns )
coeff.sort_values(ascending=False)
```

```
s5     640.6
bmi    553.8
s2     528.4
s4     339.6
bp     296.6
```

```
s3     210.2
s6     115.7
age    -11.1
sex   -291.1
s1    -915.0
dtype: float64
```

```
y_preds = lr.predict( X_test )
mse = mean_squared_error( y_test, y_preds )
rmse = np.sqrt( mse )
rmse
```

```
55.09404732888505
```

```
r2 = r2_score( y_test, y_preds )
r2
```

```
0.4933408690435077
```

```
y_train_preds = lr.predict( X_train )
mse_train = mean_squared_error( y_train, y_train_preds )
rmse_train = np.sqrt( mse_train )
rmse_train
```

```
52.9486429330168
```

```
r2_train = r2_score( y_train, y_train_preds )
r2_train
```

```
0.5237974491641986
```

```
from sklearn.model_selection import KFold
kf = KFold(n_splits=5, shuffle=True )
kfid = kf.split(X_data)
```

```
kf_mse = []
for train_i, test_i in kfid:
    X_trn, X_tst = X_data.iloc[train_i], X_data.iloc[test_i]
    y_trn, y_tst = y_target.iloc[train_i], y_target.iloc[test_i]
    lr = LinearRegression()
```

```
    lr.fit ( X_trn, y_trn )
    y_preds = lr.predict( X_tst )
    mse = mean_squared_error( y_tst, y_preds )
    kf_mse.append(mse)
kf_mse
```

```
[2473.5516319387098,
 3233.267125885358,
 3553.038452271323,
 2979.5920996281343,
 2621.972092449679]
```

```
kf_rmse = np.sqrt(kf_mse)
np.mean(kf_rmse)
```

```
54.398968609104465
```

```
from sklearn.model_selection import cross_val_score
neg_mse_scores= cross_val_score(lr, X_data, y_target,
                                scoring='neg_mean_squared_error', cv=5)
rmse_scores = np.sqrt( -1 * neg_mse_scores )
rmse_scores
```

```
array([52.72497937, 55.03486476, 56.90068179, 54.85204179, 53.94638716])
```

```
np.mean( rmse_scores )
```

```
54.69179097275793
```

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
dX=diabetes['data']
dy=diabetes['target']
scaler.fit( dX )
diabetes_X_scaled = scaler.transform( dX )
np.round( diabetes_X_scaled[:3], decimals=2 )
```

```
array([[ 0.8 ,  1.07,  1.3 ,  0.46, -0.93, -0.73, -0.91, -0.05,  0.42,
        -0.37],
       [-0.04, -0.94, -1.08, -0.55, -0.18, -0.4 ,  1.56, -0.83, -1.44,
        -1.94],
       [ 1.79,  1.07,  0.93, -0.12, -0.96, -0.72, -0.68, -0.05,  0.06,
        -0.55]])
```

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor ( max_iter=50, penalty=None, eta0=0.1 )
sgd_reg.fit( diabetes_X_scaled, dy )
print(sgd_reg.intercept_, np.round( sgd_reg.coef_, decimals=1), sep="\n")
```

```
[153.41928257]
[ -0.2 -12.2  23.5  14.  -26.7  14.2   2.1   5.7  38.2   2.7]
```

Part II

딥러닝('24 가을)

딥러닝 Ch1

머신러닝 vs. 딥러닝?

암튼 머신러닝이란건 기계가 스스로 학습해서 문제를 해결한다는 의미.

우리가 컴퓨터에게 적정한 모델을 입력하면, 해당 모델을 가지고 주어진 데이터를 반복계산하여 적절한 결과값을 도출.

딥러닝은 머신러닝의 한 분류로, 적정한 모델을 인공신경망 기반의 모델을 사용한 것을 의미한다고 보면 됨.

인공신경망이 뭔지는 아직 모름. 매우 복잡하고, 적절한 결과값 도출 과정을 해석하기 어렵고, 예측력은 매우 높음.

학습데이터가 매우 많이 필요하고, 많은 계산시간, 모델 탐색시간, 하이퍼파라미터 조율시간 등이 필요함.

오류에 대한 디버깅이나 해석도 어렵다고 함.

머신러닝 알고리즘의 구분

지도학습

입력값에 대한 결과값이 주어진 경우,

모델이 입력값과 결과값을 모두 알고 학습하여 새로운 입력값에 대한 적정 결과값 추정치를 제공하게 됨.

결과값(label)이 숫자형이면 회귀(regression) 알고리즘, 범주형이면 분류(classification) 알고리즘으로 구분.

비지도학습

결과값이 없고, 주어진 입력값만으로 학습함.

의미있는 패턴을 추출하는 것이 목적.

군집화(행을 묶음) 및 차원축소(열을 묶어 열의 갯수를 감소시킴)에 주로 활용

강화학습

모델 자체가 어떠한 변화를 주도하는데,

해당 변화에 따른 보상/패널티를 주는 환경을 구성함.

모델은 이러한 변화에 따른 누적보상이 최대가 되도록하는 변화패턴을 학습함

지도학습 알고리즘의 절차

1. 전처리 및 탐색
2. 적절한 모델 선택
3. 주어진 데이터로 모델 훈련
4. 새로운 데이터를 통해 결과값을 예측하여 모델의 성능을 평가

경사하강법 (Gradient Descent Method)

다차원 선형회귀 모형에서 모델결과값과 실제결과값의 차이(MSE; Mean Square Error)를 최소화하는 방식

MSE의 미분계수(Gradient)의 일정수준(학습률)만큼 선형회귀모형의 각 파라미터가 모두 감소하도록 지속 업데이트하면서,

결과적으로 MSE의 Gradient가 0이 되면 학습이 종료되는 방식.

MSE의 미분계수가 0이면 최솟값이고, 모델결과값의 오차가 최소가 되므로 가장 적정한 파라미터를 추론한 것으로 판단.

경사하강법 종류

한번의 회귀계수 업데이트에 모든 훈련데이터를 사용하면 배치 경사하강법

임의추출로 하나의 훈련데이터만 사용하면 확률 경사하강법(SGD)

일부만 사용하면 미니 경사하강법.

많이 사용할수록 규칙적으로 MSE가 감소하고 일관적으로 움직이나, 산출시간이 오래걸리고, 지역최소값에 갇힐 가능성이 높아짐.

모델 평가 및 검증

low bias - high vol

high bias - low vol

제대로 못들어서 정리 필요

자료의 구분

일반적으로 전체 데이터를 임의로 3개로 나눔.

훈련 데이터 / 검증 데이터 / 평가 데이터

e.g. 훈련데이터로 여러 h-para에 대해 모델 돌림

검증데이터를 이용해 각 h-para별 성능 평가

제일 좋은 h-para로 모델을 구성하여 훈련+검증데이터로 다시 훈련

최종 모델을 평가데이터를 이용하여 평가

딥러닝에서는 검증데이터가 다른 의미로도 활용됨.

경사하강법 같은걸 복잡한 모델에 사용할 때, 파라미터 튜닝 과정에서 손실함수가 더이상 감소하지 않는다면?

불필요한 훈련이 될 수 있어 파라미터 자체가 학습이 잘 되고 있는지 모니터링을 해야 함.

이러한 모니터링에 검증 데이터가 활용됨.

모델의 평가를 위한 지표

회귀모형, Regression model

RMSE : $\sqrt{MSE}$

MAE(mean absolute error) : $\frac{1}{n}\sum_{i=1}^{n}|y_i-\hat{y}_i|$

R square(결정계수) : $1-\frac{\sum_{i=1}^{n}(y_i-\hat{y}_i)^2}{\sum_{i=1}^{n}(y_i-\bar{y})^2}=1-\frac{SSE}{SST}(=\frac{SSR}{SST})$

분류모형

정오분류표 : TN(true negative), TP(true positive), FN, FP

정확도, 정분류율 (Accuracy) : $\frac{TN+TP}{TN+TP+FN+FP}$

오분류율 : 1 - 정분류율

교차엔트로피 오차(Cross Entropy Error), Multiclass classification에 많이 씀

$$-\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K}y_k^{(i)}log(\hat{p}_k^{(}i))$$

y는 타깃확률, p는 예측확률, K는 범주의 개수

K=2인 경우, 위의 교차엔트로피 손실함수를 로그손실함수라고 부르며, 많이 활용함.

삼중분류문제의 경우,
모델은 훈련자료를 기반으로 훈련 후 0,1,2에 각각 속할 확률을 계산하여 반환함.($p_0 + p_1 + p_2 = 1$)
최종적으로 각 확률중 가장 큰 값을 $\hat{y}$로 산출하게 됨.
교차 엔트로피의 타깃 확률이란 1을 의미하며, 실제 y가 특정 범주 k에 속할 때 1이며, 아닐 때 0임.
따라서, 교차엔트로피의 수식을 볼 때 타깃확률이 1이고 예측확률이 1에 가까울수록 오차는 0에 수렴하며
예측확률이 1보다 작을수록 오차는 커지게 됨

딥러닝 Ch2

인공 신경망 모형 (Neural Network)

퍼셉트론 (Perceptron)

하나의 인공뉴런으로 구성된 신경망. 가장 기본적인 구조.

주어진 데이터로 분류면을 찾는 것이 목표

로젠블랜의 퍼셉트론 (Simple perceptron)

활성화함수 $f(.)$를 1 또는 -1를 가지는 임계함수로 구성.

y = 1인데 f=-1인 경우, $w_i^* = w_i + x_i$

y = -1인데 f=1인 경우, $w_i^* = w_i - x_i$

즉, 실제는 1인데 $s = \sum w_i x_i < 0$이 되어 임계가 -1로 출력되는 경우,

$\sum(w_i^* x_i + b^*_{(w_0^* x_0)}) = \sum(w_i x_i + x_i^2 + b)$이 되어 s가 증가하는 방향으로 움직이게 됨.

선형 퍼셉트론 (Linear perceptron)

활성화함수로 선형함수를 사용함 $f(s) = s$

파라미터 학습시 델타규칙(delta rule)을 사용하는데, 경사하강법(GDM)으로 보면 됨

N개의 훈련자료 $D_N = \{(x^{(d),y^{(d)}})\}_{d=1}^N$에 대해 $f^{(d)} = \sum_i w_i x_i^{(d)}$가 되며,

학습오차 $E_d = \frac{1}{2}(y^{(d)} - f(s)^{(d)})^2$ 및 $E_N = \sum E_d$

(N으로 나누던 안나누던 파라미터 업데이트 결정에는 영향 없음)

여기에 SGD(Stochastic Gredient Descent)를 적용하여 파라미터 업데이트

$w_i \leftarrow w_i + \Delta w_i = w_i - \eta \frac{\partial E_d}{\partial w_i} = w_i + \eta(y^{(d)} - f^{(d)})x_i$

> 초기 델타규칙의 파라미터 업데이트는 경사하강법의 표현법이 아니였음.
> $w_i \leftarrow w_i + \eta e x_i$, $e = y - x_i$ 방식으로 업데이트.
> 결국 목적함수를 미분하면 해당 꼴이 나타나므로 경사하강법과 동일한 논리임.

시그모이드 퍼셉트론 (Sigmoid perceptron)

머신러닝의 logistic regression과 거의 유사함. (비용함수만 조금 다름)

활성함수를 시그모이드 함수 $f = \frac{1}{1+e^{-s}} \in (0,1)$로 사용함.

즉, y가 이진분류 문제일 때 y=1일 확률값을 예측해주는 모델임.

어차피 퍼셉트론에서는 확률=0.5를 기준으로 분류하므로 선형분류면을 제공하지만,

네트워크를 구성하면 비선형 경계면을 구성할 수도 있음.

$f^{(d)} = \sigma(s^{(d)}) = \frac{1}{1+e^{-s^{(d)}}} = \frac{1}{1+e^{-\sum w_i x_i^{(d)}}}$

$E_d = \frac{1}{2}(y^{(d)} - \sigma(s^{(d)}))^2$

$\frac{\partial E_d}{\partial w_i} = (y^{(d)} - \sigma^{(d)})\sigma^{(s)}(1-\sigma^{(s)})(-x_i^{(d)})$

$\therefore w_i^* \leftarrow w_i + \eta(y^{(d)} - f^{(d)})\, f^{(s)}\, (1-f^{(s)}) x_i^{(d)}$

w_i의 업데이트는 동시에 이루어져야함에 유의

> 시그모이드 함수 미분
> $\frac{d\sigma(s)}{ds} = \frac{+e^{-s}}{(1+e^{-s})^2} = \sigma(s)(1-\sigma(s))$

다층 퍼셉트론 (Multi-layer perceptron)

XOR문제 : $(x_1, x_2, y) = (1,0,1),\ (0,1,1),\ (0,0,0),\ (1,1,0)$

기존 퍼셉트론은 이 간단한 XOR문제도 해결할 수 없음.

여러개의 퍼셉트론을 여러 층으로 쌓는다면 이를 해결 가능함.

선을 두개를 그어서 z_1, z_2를 구성하고, $f = z_1 \times z_2$를 최종 모델로 결정하면 문제 해결 가능.

2 딥러닝 Ch1 실습

2.1 1. 텐서 데이터 만들기

```
import numpy as np
import tensorflow as tf
```

```
test = tf.constant( 123 ) # 텐서 상수. numpy array 같은 데이터타입임.
test
```

```
<tf.Tensor: shape=(), dtype=int32, numpy=123>
```

```
print(test)
```

```
tf.Tensor(123, shape=(), dtype=int32)
```

```
test.numpy()
```

```
123
```

```
tf.constant([1.2, 5, np.pi], dtype = tf.float32)
```

```
<tf.Tensor: shape=(3,), dtype=float32, numpy=array([1.2      , 5.       , 3.1415927], dtype=float
```

```
ndarr = np.array([[1,2,3], [4,5,6]])
ndarr
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

```
tsarr = tf.convert_to_tensor( ndarr )
tsarr
```

```
<tf.Tensor: shape=(2, 3), dtype=int64, numpy=
array([[1, 2, 3],
       [4, 5, 6]])>
```

```
tsones = tf.ones((2,3))
tsones
```

```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[1., 1., 1.],
       [1., 1., 1.]], dtype=float32)>
```

2.2 2. 텐서 데이터 타입, 크기

```
tsarr.shape
```

```
TensorShape([2, 3])
```

```
tsarr.ndim
```

```
2
```

```
tsarr.dtype
```

```
tf.int64
```

```
tf.cast( tsarr, dtype=tf.float64 )
```

```
<tf.Tensor: shape=(2, 3), dtype=float64, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]])>
```

```
tsarr[0]
```

```
<tf.Tensor: shape=(3,), dtype=int64, numpy=array([1, 2, 3])>
```

```
tsarr[:1,:1]
```

```
<tf.Tensor: shape=(1, 1), dtype=int64, numpy=array([[1]])>
```

```
tsarr[0,0]
```

```
<tf.Tensor: shape=(), dtype=int64, numpy=1>
```

```
t = tf.random.uniform(shape=(3,4)) # shape 생략 가능
t
```

```
<tf.Tensor: shape=(3, 4), dtype=float32, numpy=
array([[0.36321807, 0.29708588, 0.04835212, 0.6580317 ],
       [0.52478623, 0.81622696, 0.68001425, 0.35392594],
       [0.60432065, 0.5515584 , 0.92308843, 0.2798295 ]], dtype=float32)>
```

```
t.numpy()
```

```
array([[0.36321807, 0.29708588, 0.04835212, 0.6580317 ],
       [0.52478623, 0.81622696, 0.68001425, 0.35392594],
       [0.60432065, 0.5515584 , 0.92308843, 0.2798295 ]], dtype=float32)
```

```
tnormal = tf.random.normal((3,4), mean = 0, stddev = 1)
tnormal
```

```
<tf.Tensor: shape=(3, 4), dtype=float32, numpy=
array([[ 1.2684143 ,  0.5634787 , -0.8527982 ,  0.15773794],
       [ 0.9406849 ,  0.30044   ,  0.81672424,  1.6540827 ],
       [-0.04494214, -1.9484011 , -0.03741917, -0.0658213 ]],
      dtype=float32)>
```

```
t_tr = tf.transpose( t )
t_tr
```

```
<tf.Tensor: shape=(4, 3), dtype=float32, numpy=
array([[0.36321807, 0.52478623, 0.60432065],
       [0.29708588, 0.81622696, 0.5515584 ],
       [0.04835212, 0.68001425, 0.92308843],
       [0.6580317 , 0.35392594, 0.2798295 ]], dtype=float32)>
```

```
t_sh = tf.reshape( t, shape = (6,2))
t_sh
```

```
<tf.Tensor: shape=(6, 2), dtype=float32, numpy=
array([[0.36321807, 0.29708588],
       [0.04835212, 0.6580317 ],
       [0.52478623, 0.81622696],
```

```
       [0.68001425, 0.35392594],
       [0.60432065, 0.5515584 ],
       [0.92308843, 0.2798295 ]], dtype=float32)>
```

2.3 3. 수학 연산의 적용

```
a = tf.constant(10)
b = tf.constant(20)
c = tf.constant(30)
```

```
ad = tf.add(a,b) # substract, multiply, divide 가능
ad.numpy()
```

30

```
tf.reduce_mean( [a,b,c] ).numpy()
```

20

```
tf.reduce_sum( [a,b,c] ).numpy()
```

60

```
M1 = tf.random.uniform( shape=(5,2), minval=-1.0, maxval=1.0 )
M1
```

```
<tf.Tensor: shape=(5, 2), dtype=float32, numpy=
array([[ 0.9962559 ,  0.7100415 ],
       [ 0.06644487, -0.17041707],
       [ 0.7292304 , -0.49121428],
       [-0.9295962 , -0.6074953 ],
       [ 0.43700218, -0.5817137 ]], dtype=float32)>
```

```
M2 = tf.random.normal( shape=(5,2), mean=0, stddev=1 )
M2
```

```
<tf.Tensor: shape=(5, 2), dtype=float32, numpy=
array([[ 0.3262252 , -1.6307284 ],
       [ 1.0629762 , -0.9531726 ],
       [ 1.3960881 , -0.96611947],
       [ 0.5364423 ,  0.5114645 ],
       [ 0.0176798 , -1.0373931 ]], dtype=float32)>
```

```
tf.reduce_mean( M1, axis=0 ).numpy()
```

```
array([ 0.25986743, -0.22815976], dtype=float32)
```

```
tf.reduce_mean( M1, axis=1 ).numpy()
```

```
array([ 0.8531487 , -0.0519861 ,  0.11900806, -0.76854575, -0.07235575],
      dtype=float32)
```

```
tf.multiply( M1, M2 ).numpy
```

```
<bound method _EagerTensorBase.numpy of <tf.Tensor: shape=(5, 2), dtype=float32, numpy=
array([[ 0.32500377, -1.1578848 ],
       [ 0.07062932,  0.16243689],
       [ 1.0180699 ,  0.47457168],
       [-0.4986747 , -0.31071228],
       [ 0.00772611,  0.60346574]], dtype=float32)>>
```

```
tf.matmul( M1, tf.transpose(M2) )
```

```
<tf.Tensor: shape=(5, 5), dtype=float32, numpy=
array([[-0.8328811 ,  0.38220417,  0.7048761 ,  0.8975948 , -0.7189786 ],
       [ 0.29957995,  0.2330662 ,  0.25740615, -0.05151844,  0.17796423],
       [ 1.0389304 ,  1.2433666 ,  1.4926416 ,  0.13995136,  0.52247494],
       [ 0.6874021 , -0.4090908 , -0.7108851 , -0.80938697,  0.6137764 ],
       [ 1.091178  ,  1.0189965 ,  1.1720984 , -0.06309943,  0.61119187]],
      dtype=float32)>
```

```
tf.matmul( tf.transpose(M1), M2 )
```

```
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[ 0.9227544 , -3.3212786 ],
       [-0.97146505, -0.22812282]], dtype=float32)>
```

```
tf.transpose(M1) @ M2
```

```
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[ 0.9227544 , -3.3212786 ],
       [-0.97146505, -0.22812282]], dtype=float32)>
```

```
np.linalg.det( M1 @ tf.transpose(M2) )
```

```
-1.7482287e-23
```

```
np.linalg.det( tf.transpose(M1) @ M2 )
```

```
-3.4370074
```

```
np.linalg.inv( tf.transpose(M1) @ M2 )
```

```
array([[ 0.06637251, -0.9663286 ],
       [-0.28264853, -0.26847613]], dtype=float32)
```

```
np.linalg.eig(tf.transpose(M1) @ M2 )
```

```
EigResult(eigenvalues=array([ 2.2334855, -1.5388538], dtype=float32), eigenvectors=array([[ 0.930
       [-0.36709383,  0.59544647]], dtype=float32))
```

```
tf.norm( M1, ord=2, axis=1 ).numpy
```

```
<bound method _EagerTensorBase.numpy of <tf.Tensor: shape=(5,), dtype=float32, numpy=
array([1.2233907 , 0.18291228, 0.8792431 , 1.1104952 , 0.7275725 ],
      dtype=float32)>>
```

2.4 4. 텐서 데이터의 분할 및 통합

```
t = tf.random.uniform((6,))
t.numpy()
```

```
array([0.02300322, 0.7006937 , 0.41715956, 0.25560915, 0.6631763 ,
       0.41954446], dtype=float32)
```

```
t_spl = tf.split( t, num_or_size_splits=3 )
[ item.numpy() for item in t_spl ]
```

```
[array([0.02300322, 0.7006937 ], dtype=float32),
 array([0.41715956, 0.25560915], dtype=float32),
 array([0.6631763 , 0.41954446], dtype=float32)]
```

```
t_spl2 = tf.split( t, num_or_size_splits=[4,2])
[ item.numpy() for item in t_spl2 ]
```

```
[array([0.02300322, 0.7006937 , 0.41715956, 0.25560915], dtype=float32),
 array([0.6631763 , 0.41954446], dtype=float32)]
```

```
t2 = tf.random.uniform((6,3))
t2
```

```
<tf.Tensor: shape=(6, 3), dtype=float32, numpy=
array([[0.20208406, 0.14229345, 0.24287081],
       [0.28123713, 0.9822639 , 0.5934299 ],
       [0.06746602, 0.44820297, 0.55084395],
       [0.12668848, 0.9349866 , 0.27638876],
       [0.7941842 , 0.49613452, 0.26152658],
       [0.11516786, 0.5283252 , 0.17477489]], dtype=float32)>
```

```
t_spl3 = tf.split( t2, num_or_size_splits=[4,2], axis=0)
[ item.numpy() for item in t_spl3 ]
```

```
[array([[0.20208406, 0.14229345, 0.24287081],
        [0.28123713, 0.9822639 , 0.5934299 ],
        [0.06746602, 0.44820297, 0.55084395],
        [0.12668848, 0.9349866 , 0.27638876]], dtype=float32),
 array([[0.7941842 , 0.49613452, 0.26152658],
        [0.11516786, 0.5283252 , 0.17477489]], dtype=float32)]
```

```
t_conc = tf.concat([t2, tf.reshape(t, (6,1))], axis=1)
t_conc
```

```
<tf.Tensor: shape=(6, 4), dtype=float32, numpy=
array([[0.20208406, 0.14229345, 0.24287081, 0.02300322],
       [0.28123713, 0.9822639 , 0.5934299 , 0.7006937 ],
       [0.06746602, 0.44820297, 0.55084395, 0.41715956],
       [0.12668848, 0.9349866 , 0.27638876, 0.25560915],
       [0.7941842 , 0.49613452, 0.26152658, 0.6631763 ],
       [0.11516786, 0.5283252 , 0.17477489, 0.41954446]], dtype=float32)>
```

```
tf.concat([t_conc, tf.random.uniform((1,4))], axis=0)
```

```
<tf.Tensor: shape=(7, 4), dtype=float32, numpy=
array([[0.20208406, 0.14229345, 0.24287081, 0.02300322],
       [0.28123713, 0.9822639 , 0.5934299 , 0.7006937 ],
       [0.06746602, 0.44820297, 0.55084395, 0.41715956],
       [0.12668848, 0.9349866 , 0.27638876, 0.25560915],
       [0.7941842 , 0.49613452, 0.26152658, 0.6631763 ],
       [0.11516786, 0.5283252 , 0.17477489, 0.41954446],
       [0.3500303 , 0.06093681, 0.0672853 , 0.09213388]], dtype=float32)>
```

2.5 5. tf.data를 활용한 데이터 전처리

```
arr1 = [ 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 ]
arr1
```

```
[1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7]
```

```
ds1 = tf.data.Dataset.from_tensor_slices( arr1 )
print( ds1 )
```

```
<_TensorSliceDataset element_spec=TensorSpec(shape=(), dtype=tf.float32, name=None)>
```

```
for item in ds1:
    print(item)
```

```
tf.Tensor(1.1, shape=(), dtype=float32)
tf.Tensor(2.2, shape=(), dtype=float32)
tf.Tensor(3.3, shape=(), dtype=float32)
tf.Tensor(4.4, shape=(), dtype=float32)
```

```
tf.Tensor(5.5, shape=(), dtype=float32)
tf.Tensor(6.6, shape=(), dtype=float32)
tf.Tensor(7.7, shape=(), dtype=float32)

2024-09-12 11:19:19.694220: W tensorflow/core/framework/local_rendezvous.cc:404] Local rendezvous
```

```
ds1_batch = ds1.batch(3)
for item in ds1_batch: print( item )
```

```
tf.Tensor([1.1 2.2 3.3], shape=(3,), dtype=float32)
tf.Tensor([4.4 5.5 6.6], shape=(3,), dtype=float32)
tf.Tensor([7.7], shape=(1,), dtype=float32)

2024-09-12 11:19:19.707469: W tensorflow/core/framework/local_rendezvous.cc:404] Local rendezvous
```

```
tf.random.set_seed(1)
X = tf.random.uniform( shape = (10,3), dtype = tf.float32 )
Y = tf.range(1, 11)
```

```
X.numpy()
```

```
array([[0.16513085, 0.9014813 , 0.6309742 ],
       [0.4345461 , 0.29193902, 0.64250207],
       [0.9757855 , 0.43509948, 0.6601019 ],
       [0.60489583, 0.6366315 , 0.6144488 ],
       [0.8893349 , 0.6277617 , 0.53197503],
       [0.02597821, 0.44087505, 0.25267076],
       [0.8862232 , 0.88729346, 0.78728163],
       [0.05955195, 0.0710938 , 0.3084147 ],
       [0.25118268, 0.9084705 , 0.47147965],
       [0.24238515, 0.63300395, 0.5860311 ]], dtype=float32)
```

```
Y.numpy()
```

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10], dtype=int32)
```

```
ds_X = tf.data.Dataset.from_tensor_slices( X )
ds_Y = tf.data.Dataset.from_tensor_slices( Y )
ds_joint = tf.data.Dataset.zip((ds_X, ds_Y))
```

```python
ds_joint2 = tf.data.Dataset.from_tensor_slices((X, Y))
```

```python
for item in ds_X: print(ds_X)
```

```
<_TensorSliceDataset element_spec=TensorSpec(shape=(3,), dtype=tf.float32, name=None)>
<_TensorSliceDataset element_spec=TensorSpec(shape=(3,), dtype=tf.float32, name=None)>
<_TensorSliceDataset element_spec=TensorSpec(shape=(3,), dtype=tf.float32, name=None)>
<_TensorSliceDataset element_spec=TensorSpec(shape=(3,), dtype=tf.float32, name=None)>
<_TensorSliceDataset element_spec=TensorSpec(shape=(3,), dtype=tf.float32, name=None)>
<_TensorSliceDataset element_spec=TensorSpec(shape=(3,), dtype=tf.float32, name=None)>
<_TensorSliceDataset element_spec=TensorSpec(shape=(3,), dtype=tf.float32, name=None)>
<_TensorSliceDataset element_spec=TensorSpec(shape=(3,), dtype=tf.float32, name=None)>
<_TensorSliceDataset element_spec=TensorSpec(shape=(3,), dtype=tf.float32, name=None)>
<_TensorSliceDataset element_spec=TensorSpec(shape=(3,), dtype=tf.float32, name=None)>
```

```
2024-09-12 11:19:19.907884: W tensorflow/core/framework/local_rendezvous.cc:404] Local rendezvous
```

```python
for item in ds_Y: print(ds_Y)
```

```
<_TensorSliceDataset element_spec=TensorSpec(shape=(), dtype=tf.int32, name=None)>
<_TensorSliceDataset element_spec=TensorSpec(shape=(), dtype=tf.int32, name=None)>
<_TensorSliceDataset element_spec=TensorSpec(shape=(), dtype=tf.int32, name=None)>
<_TensorSliceDataset element_spec=TensorSpec(shape=(), dtype=tf.int32, name=None)>
<_TensorSliceDataset element_spec=TensorSpec(shape=(), dtype=tf.int32, name=None)>
<_TensorSliceDataset element_spec=TensorSpec(shape=(), dtype=tf.int32, name=None)>
<_TensorSliceDataset element_spec=TensorSpec(shape=(), dtype=tf.int32, name=None)>
<_TensorSliceDataset element_spec=TensorSpec(shape=(), dtype=tf.int32, name=None)>
<_TensorSliceDataset element_spec=TensorSpec(shape=(), dtype=tf.int32, name=None)>
<_TensorSliceDataset element_spec=TensorSpec(shape=(), dtype=tf.int32, name=None)>
```

```
2024-09-12 11:19:19.923096: W tensorflow/core/framework/local_rendezvous.cc:404] Local rendezvous
```

```python
for item in ds_joint:
    print( item[0].numpy(),':',item[1].numpy() )
```

```
[0.16513085 0.9014813  0.6309742 ] : 1
[0.4345461  0.29193902 0.64250207] : 2
[0.9757855  0.43509948 0.6601019 ] : 3
[0.60489583 0.6366315  0.6144488 ] : 4
```

```
[0.8893349  0.6277617  0.53197503] : 5
[0.02597821 0.44087505 0.25267076] : 6
[0.8862232  0.88729346 0.78728163] : 7
[0.05955195 0.0710938  0.3084147 ] : 8
[0.25118268 0.9084705  0.47147965] : 9
[0.24238515 0.63300395 0.5860311 ] : 10

2024-09-12 11:19:19.939704: W tensorflow/core/framework/local_rendezvous.cc:404] Local rendezvous
```

```
# 똑같음
for item in ds_joint2:
    print( item[0].numpy(),':',item[1].numpy() )
```

```
[0.16513085 0.9014813  0.6309742 ] : 1
[0.4345461  0.29193902 0.64250207] : 2
[0.9757855  0.43509948 0.6601019 ] : 3
[0.60489583 0.6366315  0.6144488 ] : 4
[0.8893349  0.6277617  0.53197503] : 5
[0.02597821 0.44087505 0.25267076] : 6
[0.8862232  0.88729346 0.78728163] : 7
[0.05955195 0.0710938  0.3084147 ] : 8
[0.25118268 0.9084705  0.47147965] : 9
[0.24238515 0.63300395 0.5860311 ] : 10

2024-09-12 11:19:19.952051: W tensorflow/core/framework/local_rendezvous.cc:404] Local rendezvous
```

```
ds_trans = ds_joint.map( lambda x, y: (x*2-1, y/10))
for item in ds_trans:
    print( item[0].numpy(),':',item[1].numpy() )
```

```
[-0.6697383   0.80296254  0.26194835] : 0.1
[-0.13090777 -0.41612196  0.28500414] : 0.2
[ 0.951571   -0.12980103  0.32020378] : 0.3
[0.20979166 0.27326298 0.22889757] : 0.4
[0.77866983 0.25552344 0.06395006] : 0.5
[-0.9480436  -0.11824989 -0.49465847] : 0.6
[0.7724464  0.7745869  0.57456326] : 0.7
[-0.8808961 -0.8578124 -0.3831706] : 0.8
[-0.49763465  0.816941   -0.05704069] : 0.9
[-0.5152297   0.2660079   0.17206216] : 1.0
```

2024-09-12 11:19:47.889358: W tensorflow/core/framework/local_rendezvous.cc:404] Local rendezvous

```
ds_trans2 = ds_joint.map( lambda x, y: ([x[0]+1,x[1]+2,x[2]+3], y/10))
for item in ds_trans2:
    print( item[0].numpy(),':',item[1].numpy() )
```

```
[1.1651309 2.9014812 3.6309743] : 0.1
[1.4345461 2.291939  3.642502 ] : 0.2
[1.9757855 2.4350996 3.660102 ] : 0.3
[1.6048958 2.6366315 3.6144488] : 0.4
[1.8893349 2.6277618 3.531975 ] : 0.5
[1.0259782 2.440875  3.2526708] : 0.6
[1.8862232 2.8872933 3.7872815] : 0.7
[1.059552  2.0710938 3.3084147] : 0.8
[1.2511827 2.9084706 3.4714797] : 0.9
[1.2423851 2.633004  3.586031 ] : 1.0
```

2024-09-12 11:19:50.254590: W tensorflow/core/framework/local_rendezvous.cc:404] Local rendezvous

```
ds_shfl = ds_joint.shuffle( buffer_size = len( X ) )
for item in ds_shfl:
  print( item[0].numpy(),':', item[1].numpy() )
```

```
[0.60489583 0.6366315  0.6144488 ] : 4
[0.25118268 0.9084705  0.47147965] : 9
[0.16513085 0.9014813  0.6309742 ] : 1
[0.24238515 0.63300395 0.5860311 ] : 10
[0.4345461  0.29193902 0.64250207] : 2
[0.8862232  0.88729346 0.78728163] : 7
[0.05955195 0.0710938  0.3084147 ] : 8
[0.02597821 0.44087505 0.25267076] : 6
[0.9757855  0.43509948 0.6601019 ] : 3
[0.8893349  0.6277617  0.53197503] : 5
```

2024-09-12 11:19:50.712884: W tensorflow/core/framework/local_rendezvous.cc:404] Local rendezvous

```
ds_batch = ds_joint.batch(3, drop_remainder = True )
for item in ds_batch :
  print( item[0].numpy(),':', item[1].numpy() )
```

```
[[0.16513085 0.9014813  0.6309742 ]
 [0.4345461  0.29193902 0.64250207]
 [0.9757855  0.43509948 0.6601019 ]] : [1 2 3]
[[0.60489583 0.6366315  0.6144488 ]
 [0.8893349  0.6277617  0.53197503]
 [0.02597821 0.44087505 0.25267076]] : [4 5 6]
[[0.8862232  0.88729346 0.78728163]
 [0.05955195 0.0710938  0.3084147 ]
 [0.25118268 0.9084705  0.47147965]] : [7 8 9]

2024-09-12 11:19:51.035230: W tensorflow/core/framework/local_rendezvous.cc:404] Local rendezvous
```

```
ds_rpt = ds_joint.batch(3, drop_remainder = True ).repeat( count = 2 )
for item in ds_rpt :
  print( item[0].numpy(),':', item[1].numpy() )
```

```
[[0.16513085 0.9014813  0.6309742 ]
 [0.4345461  0.29193902 0.64250207]
 [0.9757855  0.43509948 0.6601019 ]] : [1 2 3]
[[0.60489583 0.6366315  0.6144488 ]
 [0.8893349  0.6277617  0.53197503]
 [0.02597821 0.44087505 0.25267076]] : [4 5 6]
[[0.8862232  0.88729346 0.78728163]
 [0.05955195 0.0710938  0.3084147 ]
 [0.25118268 0.9084705  0.47147965]] : [7 8 9]
[[0.16513085 0.9014813  0.6309742 ]
 [0.4345461  0.29193902 0.64250207]
 [0.9757855  0.43509948 0.6601019 ]] : [1 2 3]
[[0.60489583 0.6366315  0.6144488 ]
 [0.8893349  0.6277617  0.53197503]
 [0.02597821 0.44087505 0.25267076]] : [4 5 6]
[[0.8862232  0.88729346 0.78728163]
 [0.05955195 0.0710938  0.3084147 ]
 [0.25118268 0.9084705  0.47147965]] : [7 8 9]

2024-09-12 11:19:51.341547: W tensorflow/core/framework/local_rendezvous.cc:404] Local rendezvous
```

```
ds_rpt2 = ds_joint.repeat( count = 2 ).batch(3, drop_remainder = True )
for item in ds_rpt2 :
  print( item[0].numpy(),':', item[1].numpy() )
```

```
[[0.16513085 0.9014813  0.6309742 ]
 [0.4345461  0.29193902 0.64250207]
 [0.9757855  0.43509948 0.6601019 ]] : [1 2 3]
[[0.60489583 0.6366315  0.6144488 ]
 [0.8893349  0.6277617  0.53197503]
 [0.02597821 0.44087505 0.25267076]] : [4 5 6]
[[0.8862232  0.88729346 0.78728163]
 [0.05955195 0.0710938  0.3084147 ]
 [0.25118268 0.9084705  0.47147965]] : [7 8 9]
[[0.24238515 0.63300395 0.5860311 ]
 [0.16513085 0.9014813  0.6309742 ]
 [0.4345461  0.29193902 0.64250207]] : [10  1  2]
[[0.9757855  0.43509948 0.6601019 ]
 [0.60489583 0.6366315  0.6144488 ]
 [0.8893349  0.6277617  0.53197503]] : [3 4 5]
[[0.02597821 0.44087505 0.25267076]
 [0.8862232  0.88729346 0.78728163]
 [0.05955195 0.0710938  0.3084147 ]] : [6 7 8]
```

```
2024-09-12 11:19:51.662741: W tensorflow/core/framework/local_rendezvous.cc:404] Local rendezvous
```

```
# ppt 56번 제일 흔하게 씀
ds_all = ds_joint.shuffle( len(X) ).batch( 3 ).repeat( 2 )
for item in ds_all :
  print( item[0].numpy(),':', item[1].numpy() )
```

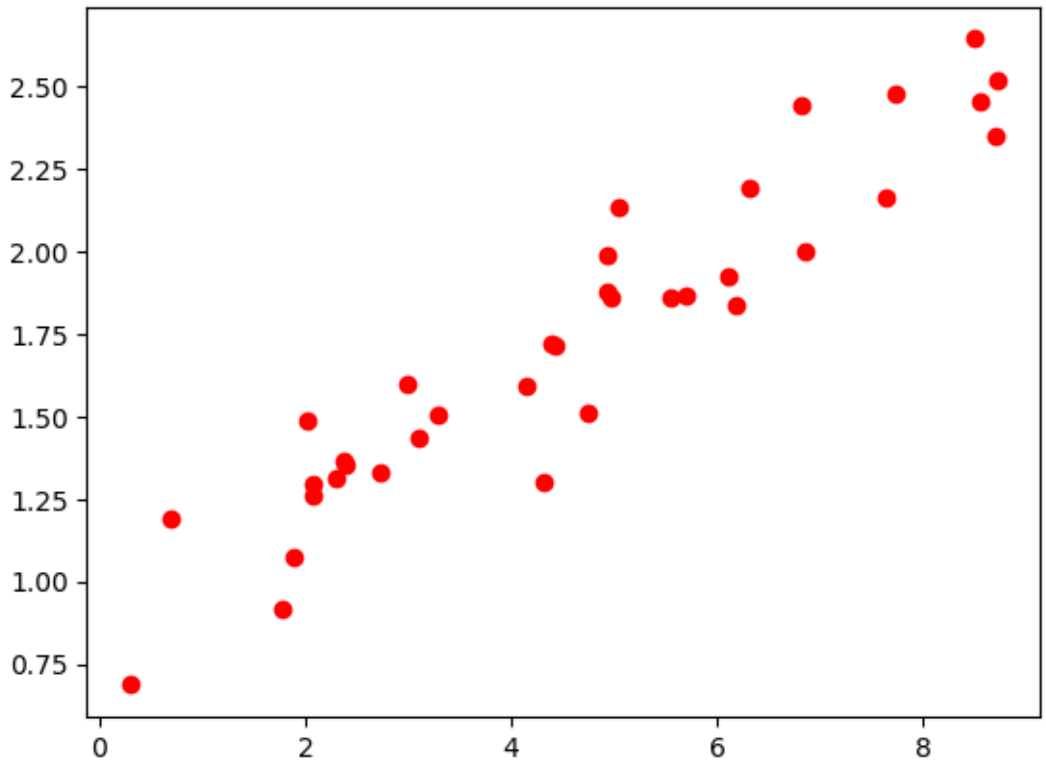
```
[[0.8893349  0.6277617  0.53197503]
 [0.16513085 0.9014813  0.6309742 ]
 [0.25118268 0.9084705  0.47147965]] : [5 1 9]
[[0.05955195 0.0710938  0.3084147 ]
 [0.9757855  0.43509948 0.6601019 ]
 [0.8862232  0.88729346 0.78728163]] : [8 3 7]
[[0.24238515 0.63300395 0.5860311 ]
 [0.4345461  0.29193902 0.64250207]
 [0.60489583 0.6366315  0.6144488 ]] : [10  2  4]
[[0.02597821 0.44087505 0.25267076]] : [6]
[[0.16513085 0.9014813  0.6309742 ]
 [0.02597821 0.44087505 0.25267076]
 [0.4345461  0.29193902 0.64250207]] : [1 6 2]
[[0.24238515 0.63300395 0.5860311 ]
```

```
 [0.8862232  0.88729346 0.78728163]
 [0.8893349  0.6277617  0.53197503]] : [10  7  5]
[[0.9757855  0.43509948 0.6601019 ]
 [0.05955195 0.0710938  0.3084147 ]
 [0.60489583 0.6366315  0.6144488 ]] : [3 8 4]
[[0.25118268 0.9084705  0.47147965]] : [9]

2024-09-12 11:19:52.244921: W tensorflow/core/framework/local_rendezvous.cc:404] Local rendezvous
```

2.6 6. 선형회귀분석 (low-lever ver.)

```
# alpha=0.8, beta=0.2, error term 일반적인 선형회귀식
X = tf.random.uniform( minval=0, maxval=10, shape=(36,))
Y = 0.2 * X + 0.8 + tf.random.normal(mean=0, stddev=0.15, shape=(36, ))
```

```
import matplotlib.pyplot as plt
plt.plot(X, Y, 'ro', label='Original Data')
```




```
# 훈련데이터와 평가데이터로 구분
trainX, testX = tf.split( X, num_or_size_splits= [30, 6] )
trainY, testY = tf.split( Y, num_or_size_splits= [30, 6] )
```

```
ds_train = tf.data.Dataset.from_tensor_slices( ( trainX, trainY ) )
```

```
W = tf.Variable( np.random.randn() )
b = tf.Variable( np.random.randn() )
```

```
print( W.numpy(), b.numpy() )
```

```
0.8828039 -0.736979
```

```
# y^=wx=b
# L = sum(y^-y)**2
def linear_regression( x ):
  return tf.add( tf.multiply( W, x ), b )

def mean_square( ypred, y ):
  return tf.reduce_mean( tf.square( y-ypred ) )

optimizer = tf.optimizers.SGD( learning_rate= 0.01 )
```

```
num_epochs = 100
log_steps = 50
batch_size = 5
steps_per_epoch = int( np.ceil( len(ds_train)/ batch_size ) )
L=[]
ds_train = ds_train.shuffle( buffer_size = len( ds_train ) ).batch( batch_size ).repeat( count
len( ds_train )
```

```
600
```

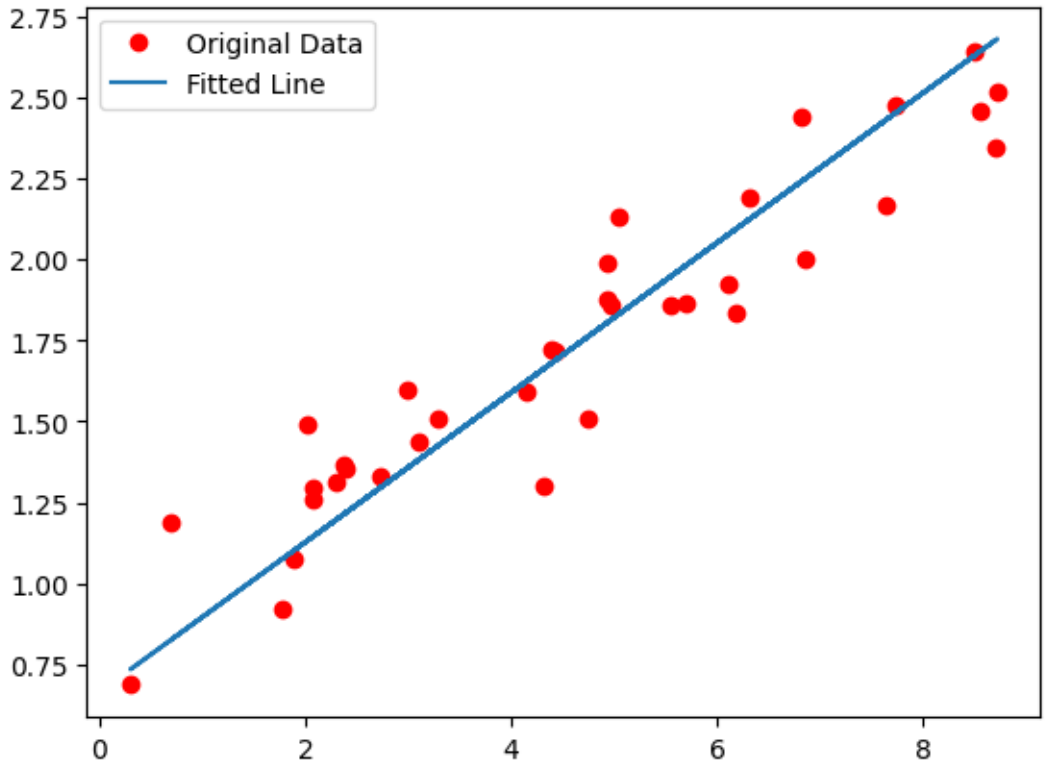
```
# enumerate : index와 item을 동시에 반환함
for i, batch in enumerate( ds_train ):
  bX, bY = batch
  # pred, loss의 미니배치별 반복연산을 tape에 저장 (tf.GT함수)
  with tf.GradientTape() as tape:
    pred = linear_regression( bX )
    loss = mean_square( pred, bY ) # 각 미치배치에 대한 MSE

  # 미분값 자동 계산하여 gradient 산출 (손실함수, [변수]) > output은 gredient vector 형태
  gradients = tape.gradient( loss, [W, b] )
  # 위 gredient를 이용해서 W, b의 값 자체를 업데이트시킴
```

```
    optimizer.apply_gradients( zip( gradients, [W, b] ) )

    if i % log_steps == 0 :
      print( i, loss.numpy(), W.numpy(), b.numpy() )
      L.append( loss.numpy() )
```

```
0 7.52958 0.5514875 -0.78580034
50 0.6072274 0.4242987 -0.52097505
100 0.618061 0.36404002 -0.29201853
150 0.15292463 0.36243728 -0.09439203
200 0.21440308 0.30857712 0.060085576
250 0.08130302 0.29104465 0.19385691
300 0.07720743 0.28864947 0.30360907
350 0.050805908 0.25714234 0.39301616
400 0.02672484 0.25849962 0.47066927
450 0.014631959 0.2344376 0.53165174
500 0.07283725 0.23046146 0.5840433
550 0.06650426 0.23649402 0.63017404
```

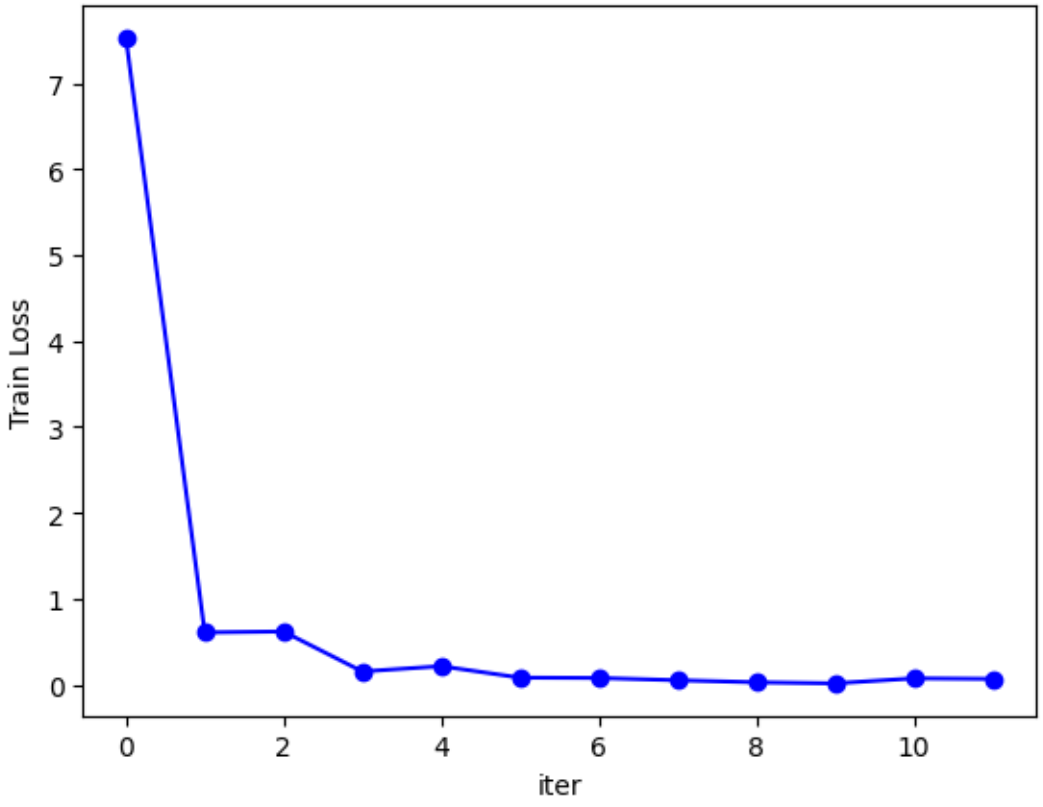
```
2024-09-12 11:24:24.415082: W tensorflow/core/framework/local_rendezvous.cc:404] Local rendezvous
```

```
import matplotlib.pyplot as plt
plt.plot(X, Y, 'ro', label='Original Data')
plt.plot(X, np.array( W * X + b ), label='Fitted Line' )
plt.legend()
```




```
plt.plot(L, 'bo-')
plt.ylabel('Train Loss')
plt.xlabel('iter')
```

```
Text(0.5, 0, 'iter')
```



```
tpred = linear_regression( testX )
test_mse = mean_square( tpred, testY )
test_mse.numpy()
```

```
0.054581102
```

2.7 7. 선형회귀분석 (tf.keras ver.)

```
ds_train2 = tf.data.Dataset.from_tensor_slices((trainX, trainY))
ds_train2 = ds_train2.shuffle(30).batch(5)
```

```
model = tf.keras.models.Sequential()
model.add( tf.keras.layers.Dense(1, input_dim = 1, activation='linear'))
model.summary()
```

```
/Users/hwan/.pyenv/versions/3.10.14/envs/hwan/lib/python3.10/site-packages/keras/src/layers/core/
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
Model: "sequential"
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                         ┃ Output Shape                ┃         Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━┩
│ dense (Dense)                        │ (None, 1)                   │               2 │
└──────────────────────────────────────┴─────────────────────────────┴─────────────────┘
```

```
 Total params: 2 (8.00 B)

 Trainable params: 2 (8.00 B)

 Non-trainable params: 0 (0.00 B)
```

```
model.compile( loss='mse', optimizer = tf.keras.optimizers.SGD( learning_rate=0.01 ) )
history = model.fit( ds_train2, epochs=100, verbose=1 )
```

```
Epoch 1/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 9.2056
Epoch 2/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 754us/step - loss: 0.1788
Epoch 3/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 830us/step - loss: 0.2041
Epoch 4/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 629us/step - loss: 0.1412
Epoch 5/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 722us/step - loss: 0.1873
Epoch 6/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 712us/step - loss: 0.1623
Epoch 7/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 911us/step - loss: 0.1415
Epoch 8/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 647us/step - loss: 0.1558
Epoch 9/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.1734
Epoch 10/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 560us/step - loss: 0.1475
Epoch 11/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 613us/step - loss: 0.1752
Epoch 12/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 729us/step - loss: 0.1125
Epoch 13/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 529us/step - loss: 0.1356
Epoch 14/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 672us/step - loss: 0.1163
Epoch 15/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 645us/step - loss: 0.0936
Epoch 16/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 949us/step - loss: 0.1252
Epoch 17/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 934us/step - loss: 0.1352
Epoch 18/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 769us/step - loss: 0.0969
Epoch 19/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 883us/step - loss: 0.0930
Epoch 20/100
```

```
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.0873
Epoch 21/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 849us/step - loss: 0.0812
Epoch 22/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 632us/step - loss: 0.0802
Epoch 23/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 567us/step - loss: 0.0840
Epoch 24/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 650us/step - loss: 0.0755
Epoch 25/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.0970
Epoch 26/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 664us/step - loss: 0.0781
Epoch 27/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.0698
Epoch 28/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 676us/step - loss: 0.1052
Epoch 29/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.0698
Epoch 30/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 959us/step - loss: 0.0751
Epoch 31/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 620us/step - loss: 0.0814
Epoch 32/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 703us/step - loss: 0.0579
Epoch 33/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 769us/step - loss: 0.0752
Epoch 34/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 701us/step - loss: 0.0487
Epoch 35/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 574us/step - loss: 0.0602
Epoch 36/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 720us/step - loss: 0.0826
Epoch 37/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 551us/step - loss: 0.0546
Epoch 38/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 595us/step - loss: 0.0613
Epoch 39/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 712us/step - loss: 0.0552
```

```
Epoch 40/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 551us/step - loss: 0.0515
Epoch 41/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 843us/step - loss: 0.0542
Epoch 42/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 617us/step - loss: 0.0401
Epoch 43/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 978us/step - loss: 0.0669
Epoch 44/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 563us/step - loss: 0.0532
Epoch 45/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 728us/step - loss: 0.0451
Epoch 46/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 568us/step - loss: 0.0468
Epoch 47/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 767us/step - loss: 0.0416
Epoch 48/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 626us/step - loss: 0.0615
Epoch 49/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 698us/step - loss: 0.0498
Epoch 50/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 606us/step - loss: 0.0507
Epoch 51/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 823us/step - loss: 0.0406
Epoch 52/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 543us/step - loss: 0.0414
Epoch 53/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 702us/step - loss: 0.0328
Epoch 54/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 528us/step - loss: 0.0382
Epoch 55/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 867us/step - loss: 0.0480
Epoch 56/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 681us/step - loss: 0.0454
Epoch 57/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 592us/step - loss: 0.0360
Epoch 58/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 549us/step - loss: 0.0473
Epoch 59/100
```

```
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 846us/step - loss: 0.0390
Epoch 60/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 560us/step - loss: 0.0338
Epoch 61/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 552us/step - loss: 0.0294
Epoch 62/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 622us/step - loss: 0.0294
Epoch 63/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 756us/step - loss: 0.0449
Epoch 64/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 626us/step - loss: 0.0288
Epoch 65/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 554us/step - loss: 0.0272
Epoch 66/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 645us/step - loss: 0.0283
Epoch 67/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - loss: 0.0394
Epoch 68/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.0272
Epoch 69/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 651us/step - loss: 0.0258
Epoch 70/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 570us/step - loss: 0.0260
Epoch 71/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 554us/step - loss: 0.0219
Epoch 72/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 708us/step - loss: 0.0274
Epoch 73/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 563us/step - loss: 0.0280
Epoch 74/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 590us/step - loss: 0.0249
Epoch 75/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.0451
Epoch 76/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 843us/step - loss: 0.0315
Epoch 77/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 665us/step - loss: 0.0271
Epoch 78/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 604us/step - loss: 0.0217
```

```
Epoch 79/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.0219
Epoch 80/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 601us/step - loss: 0.0285
Epoch 81/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 525us/step - loss: 0.0224
Epoch 82/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 685us/step - loss: 0.0315
Epoch 83/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 588us/step - loss: 0.0224
Epoch 84/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 860us/step - loss: 0.0208
Epoch 85/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 494us/step - loss: 0.0243
Epoch 86/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 617us/step - loss: 0.0307
Epoch 87/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 623us/step - loss: 0.0206
Epoch 88/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 668us/step - loss: 0.0332
Epoch 89/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 514us/step - loss: 0.0222
Epoch 90/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 525us/step - loss: 0.0257
Epoch 91/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 754us/step - loss: 0.0249
Epoch 92/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 746us/step - loss: 0.0270
Epoch 93/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 511us/step - loss: 0.0190
Epoch 94/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 664us/step - loss: 0.0302
Epoch 95/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 674us/step - loss: 0.0335
Epoch 96/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 649us/step - loss: 0.0241
Epoch 97/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 499us/step - loss: 0.0279
Epoch 98/100
```

```
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 953us/step - loss: 0.0254
Epoch 99/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 777us/step - loss: 0.0289
Epoch 100/100
6/6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 635us/step - loss: 0.0236
```

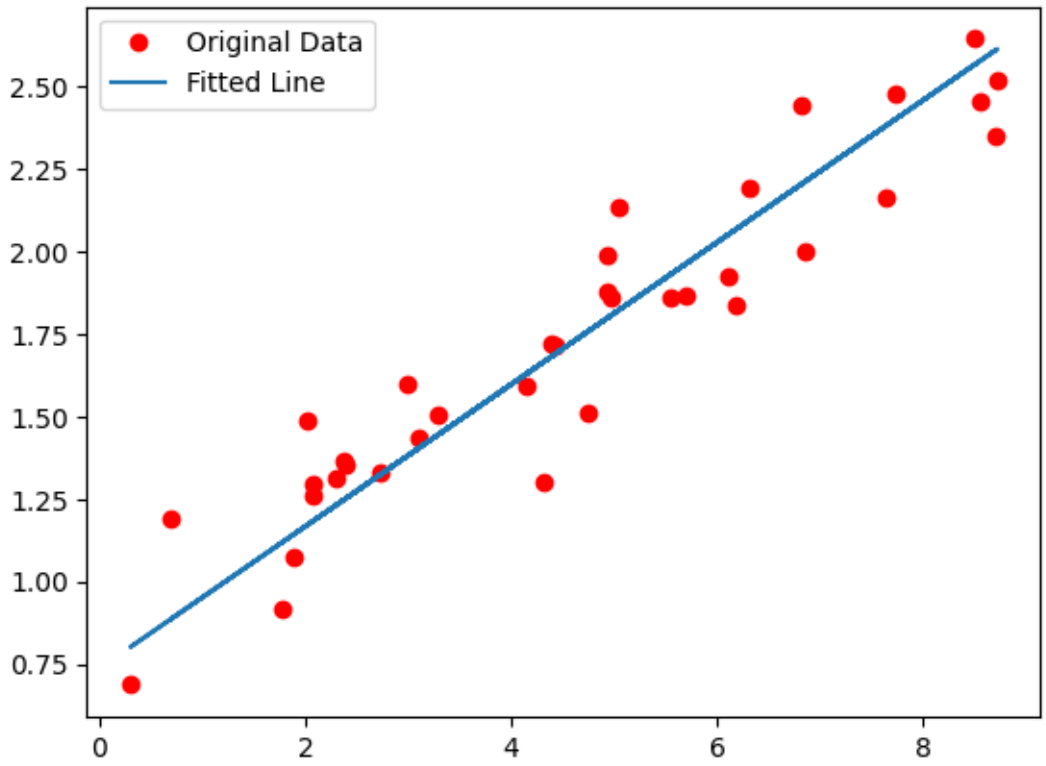
```
model.weights
```

```
[<KerasVariable shape=(1, 1), dtype=float32, path=sequential/dense/kernel>,
 <KerasVariable shape=(1,), dtype=float32, path=sequential/dense/bias>]
```

```
W2 = model.weights[0][0][0]
b2 = model.weights[1][0]
print( W2, b2 )
```

```
tf.Tensor(0.21501322, shape=(), dtype=float32) tf.Tensor(0.7366244, shape=(), dtype=float32)
```

```
plt.plot(X, Y, 'ro', label='Original Data')
plt.plot(X, np.array( W2 * X + b2 ), label='Fitted Line' )
plt.legend()
```



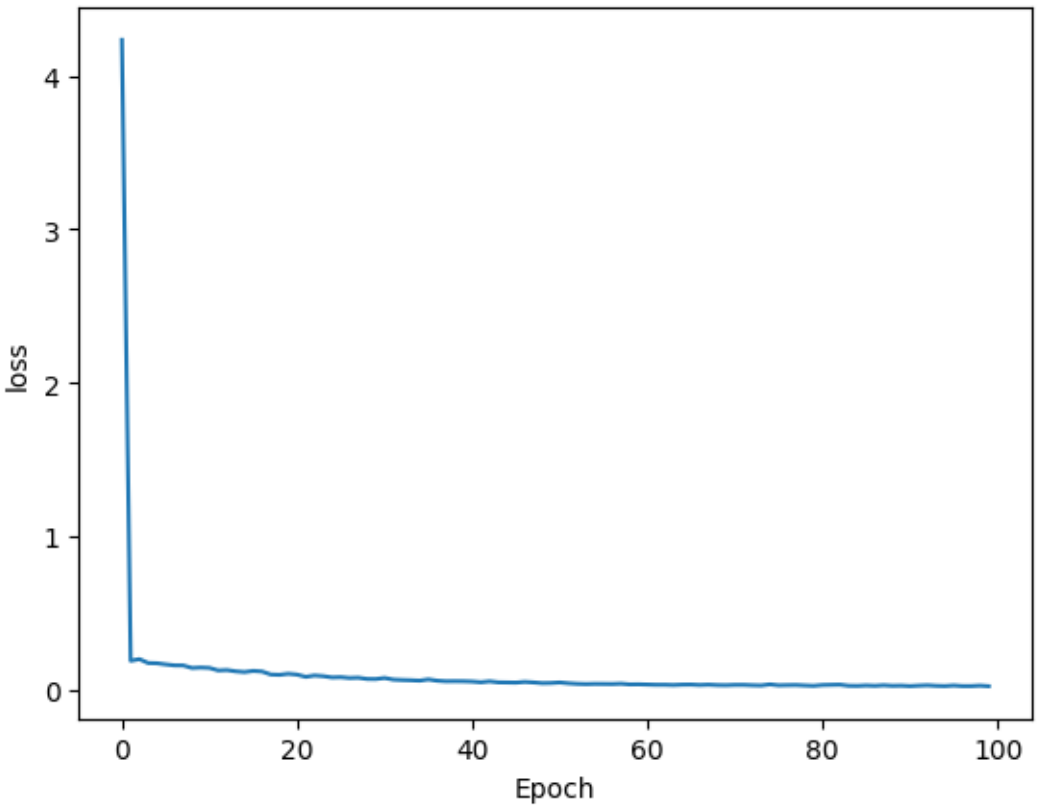
```
tpred2 = model.predict( testX )
test_mse2 = mean_square( tpred2, testY )
test_mse2.numpy()
```

```
1/1 ——————————————————————————— 0s 25ms/step
```

```
0.70796406
```

```
plt.plot(history.history['loss'])
plt.ylabel('loss')
plt.xlabel('Epoch')
```

```
Text(0.5, 0, 'Epoch')
```



Part III

시뮬레이션 방법론('24 가을)

시뮬레이션방법론 Ch1

몬테카를로 시뮬레이션 기초

블랙숄즈공식 예시

1. $f_t + \frac{1}{2}\sigma^2 S^2 f_{ss} + rSf_s - rf = 0$

-> 수치해석적인 방법으로 풀게 됨, FDM(Finite Difference Method)

2. $P(0) = e^{-rT}E^Q[P(T)]$

-> 마팅게일, 몬테카를로 시뮬레이션(Montecarlo simulation, MCS)을 주로 사용함

Volume과 적분

$x \sim uniform[0,1]$, $\alpha = E[f(x)] = \int_0^1 f(x)dx$

그러나, MCS를 이용하는 경우 임의변수 $x_1, x_2, ..., x_n$을 샘플링하여 $\hat{\alpha} = \frac{1}{N}\sum_i^N f(x_i)$로 산출함

두 값이 정확히 일치하지는 않지만, 표본이 커질수록 그 오차는 0으로 수렴함($\alpha \approx \hat{\alpha}$)

이는 **대수의 법칙과 중심극한정리**에 따라 수학적으로 정의할 수 있음

> 중심극한정리
>
> 표본평균($\hat{\alpha}$)은 정규분포를 따르므로, $\hat{\alpha} - \alpha \sim N(0, \frac{\sigma^2}{N})$
> 즉, 표본의 크기가 커질수록 두 차이는 0으로 수렴함(probibility convergence)
> 오차의 표준편차는 $\frac{\sigma}{\sqrt{N}}$이므로, 표본의 크기가 100배 증가하면 오차의 표준편차는 10배 감소함

이외에도 간단힌 사다리꼴(trapezoidal) 방식을 이용해볼 수 있음.

3. $\alpha \approx \frac{f(0)+f(1)}{2n} + \frac{1}{n}\sum_{i=1}^{n-1} f(\frac{i}{n})$ ( )

이는 매우 간단하고 효율적인 방법이지만, 변수가 늘어날 수록 효율이 급감함.

MCS 기초

몬테카를로 시뮬레이션을 개념적으로 설명함

예를 들어, 1X1 사각형에 내접한 원에 대하여, 사각형 안에 임의의 점을 찍을 때 원에 포함될 확률?

직관적으로 면적을 통해 $\Pi/4$임을 알 수 있음.

이를 다변수, 다차원, 복잡한 함수꼴로 확장한다면 면적을 구하는 적분을 통해 구할 수 있음을 의미함.

근데 그런 복잡한 계산 대신에 랜덤변수를 생성해서 시행횟수를 수없이 시행하고,

원(면적) 안에 속할 확률을 구한다면? 이게 몬테카를로 시뮬레이션의 기초임.

수없이 많은 (x, y)를 생성하고, 좌표평면의 1X1 사각형에 대해 원안에 속할 확률은 $x^2 + y^2 < 1/4$임.

이러한 확률을 구하는 것은 기대값으로 표현할 수 있게 되고, 결국 이 확률은 $\Pi/4$로 수렴

$$Pr(x \in B) = E(\int_A 1_B) = \Pi/4$$

확률기대값 및 원주율 계산 예시

```
import numpy as np
n = 10000
x = np.random.rand(n) # uniform random number in (0,1)
x -= 0.5
y = np.random.rand(n)
y -= 0.5

d = np.sqrt(x**2+y**2)
i = d<0.5
prob = i.sum() / n
pi = 4 * i.sum() / n

print(prob,pi,sep="\n")
```

```
0.7841
3.1364
```

표본표준편차 계산 : numpy는 n으로 나누고, pandas는 n-1로 나누는 것이 기본

```
import pandas as pd
np_s = i.std()
pd_s = pd.Series(i).std()
np_s_df1 = i.std(ddof=1)
print(np_s, pd_s, np_s_df1, sep = "\n")
```

```
0.41144524544585515
0.4114658192511757
0.4114658192511757
```

표준오차 계산 및 95% 신뢰구간 계산

```
se = pd_s / np.sqrt(n)
prob_95lb = prob - 2*se
prob_95ub = prob + 2*se
pi_95lb = prob_95lb*4
pi_95ub = prob_95ub*4
print(se, pi_95lb, pi_95ub, sep="\n")
```

```
0.004114658192511757
3.103482734459906
3.1693172655400943
```

경로의존성 (Path-dependent)

일반적인(Plain vanilla) 옵션은 pay-off가 기초자산의 만기시점의 가격 $S(T)$에 의해서만 결정되므로,

그 사이의 기초자산의 가격을 생성할 필요는 없음(0~T)

그러나, 아시안옵션 등은 $S(T)$ 뿐만 아니라 그 과정에 의해서 pay-off가 결정되므로 그 경로를 알아야 함.

또한, 블랙숄즈의 가정이 성립하지 않는 경우 모델링을 하기 위해서도 그 경로를 알아야 할 필요가 있음.

이를 **경로의존성**이라고 함.

시뮬레이션 예시

일반적인 주가에 대한 확률과정이 GBM을 따른다면,

$dS(t) = rS(t)dt + \sigma S(t)dW(t)$

그러나, 변동성이 주가에 따라 변하면 주가의 흐름에 따라 변동성이 바뀌므로 경로의존성이 발생

즉, $dS(t) = rS(t)dt + \sigma(S(t))S(t)dW(t)$를 따르게 되므로

우리가 앞서 사용한 $S(T) = S(0)e^{(r-\frac{1}{2}\sigma^2)T+\sigma\sqrt{T}Z}$를 사용할 수 없음.

따라서, Analytic solution이 없으므로 근사치를 구할 수 밖에 없으며 그 예시로 **이산오일러근사**가 있음

(0~T) 구간을 m개로 나누고, 각 구간의 길이 $\frac{T}{m} = \Delta t$라고 하면 기초자산의 경로 $S(t)$는,

$$S(t + \Delta t) = S(t) + rS(t)\Delta t + \sigma(S(t))S(t)\sqrt{\Delta t}Z$$

다만, 이러한 경우에는 그 경로의 길이를 얼마나 짧게 구성하는지에 따라 시뮬레이션 정밀도에 영향을 미침.

즉, 시뮬레이션 횟수 n과 경로의 길이 m이 모두 정확도를 결정하는 파라미터가 됨.

MCS 추정치 개선 방향

MCS의 효율성은 아래 3개의 기준에 따라 평가할 수 있습니다.

1. 계산시간 (Computing time)
2. 편의 (Bias)
3. 분산 (Variance)

여기서, 시뮬레이션의 $Prediction\ error\ =\ Variance\ +\ Bias^2$

> $Var[\epsilon] = E[\epsilon^2] - (E[\epsilon])^2$
> $MSE = E[\epsilon^2) = Var[\epsilon] + (E[\epsilon])^2 = Variance + Bias^2$

분산감소와 계산시간

시행횟수가 증가하면 분산은 감소함. $(n \to \infty, Var[\epsilon] \to 0)$

한번의 시뮬레이션에 정확한방법을 사용할 수록 편의는 감소함$(m \to \infty, Bias \to 0)$

(정확한방법을 사용할 수록 분산은 증가할 수 있음 (머신러닝 overfitting 같은 문제?))

(정확한방법을 쓸수록 계산비용이 증가하여 시뮬레이션 횟수가 감소함, 분산이 그래서 증가함)

시뮬레이션의 횟수

계산 예산에 s이고, 한번의 시뮬레이션의 계산량이 τ일 때, 가능한 시뮬레이션 횟수는 s/τ임

이 때, 추정치의 분포 $\sqrt{\frac{s}{\tau}}[\widehat{C} - C] \to N(0, \sigma_c^2)$

$\Rightarrow [\widehat{C} - C] \to N(0, \sigma_c^2(\frac{\tau}{c}))$ 이므로,

계산오차는 분산이 $\sigma_c^2(\frac{tau}{c})$인 정규분포에 수렴함을 의미

편의

경로의존성이 있는 시뮬레이션 중, 과거 연속적인 수치에 따라 pay-off가 정해진다면,

이산오일러근사를 사용할 때 편의가 발생함.

e.g. 룩백옵션의 경우 시뮬레이션이 항상 실제 pay-off를 과소평가 = (-) bias 존재

이 때, 이산구간의 간격 m을 작게할 수록 편의는 감소함.

또는, 기초자산이 비선형구조인 경우 등에도 편의가 발생할 수 있음.

e.g. Compound 옵션의 경우 기초자산인 옵션 가격이 비선형이므로,

Compound 옵션을 Analytic solution을 적용하여 푸는 경우 항상 실제 옵션보다 가격이 높음 = (+) bias 존재

이 때, $T_1 \sim T_2$의 n_2개의 경로를 추가로 생성하여 경로를 이중으로 구성한다면 bias 제거가 가능함.

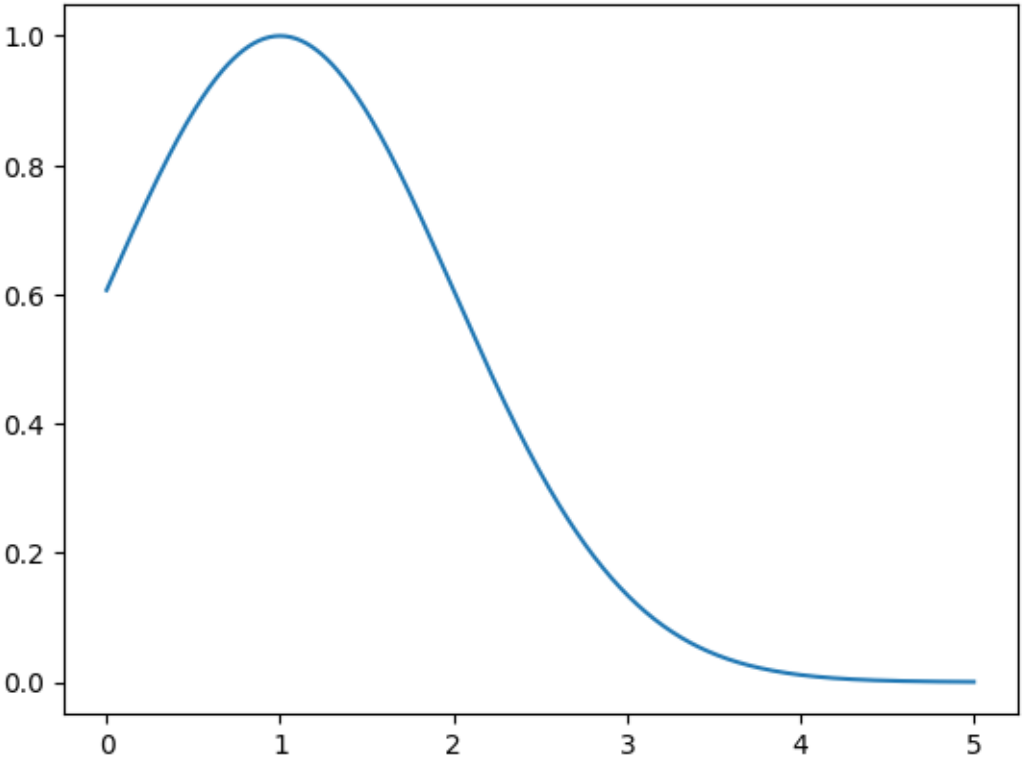
Asian Option 평가 해볼 것

3 시뮬레이션방법론 실습

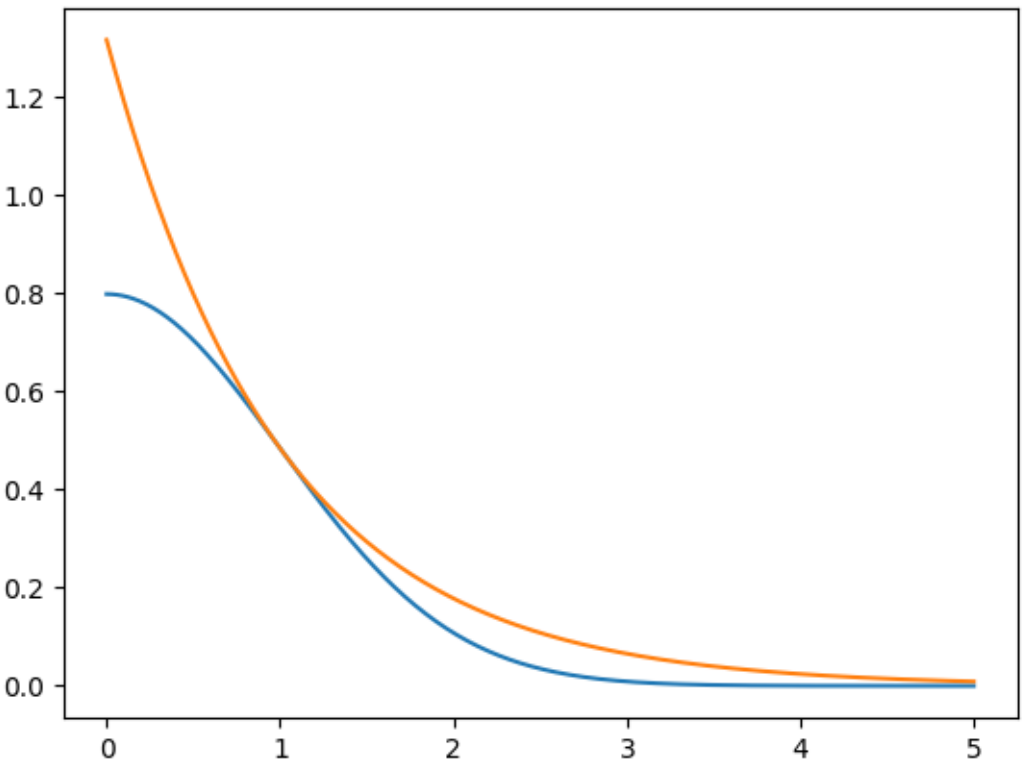
3.1 Ch2. 난수 생성 방법

3.1.1 Acceptance-rejection method

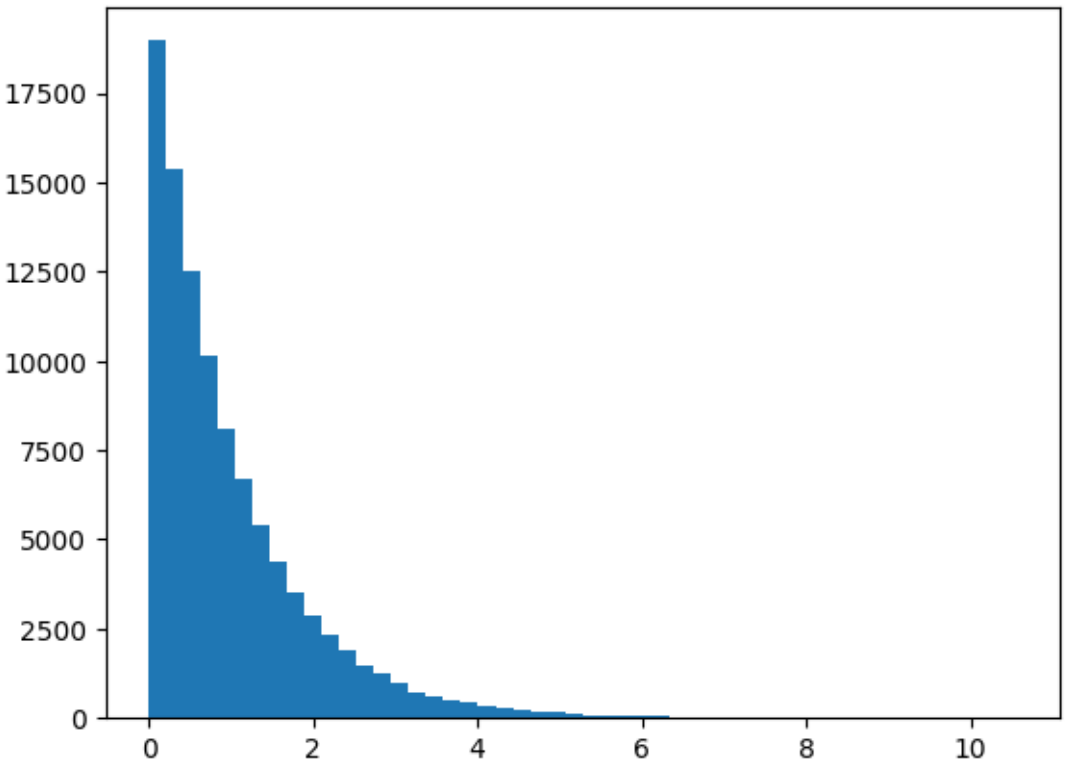
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import scipy.stats as stats

f = lambda x: 2/np.sqrt(2*np.pi)*np.exp(-x**2/2)
g = lambda x: np.exp(-x)
Ginv = lambda x: -np.log(1-x)
x = np.linspace(0,5,501)
c = np.sqrt(2/np.pi)*np.exp(0.5)

#c = 1
plt.plot(x,f(x)/(c*g(x)))
plt.show()
```



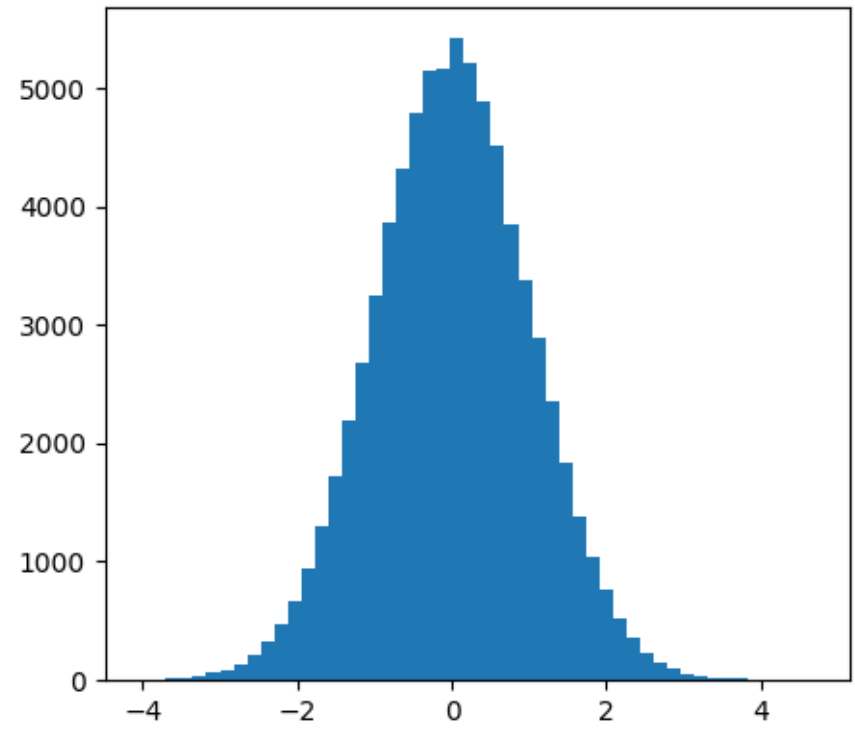

```
plt.plot(x,f(x))
plt.plot(x,c*g(x))
plt.show()
```

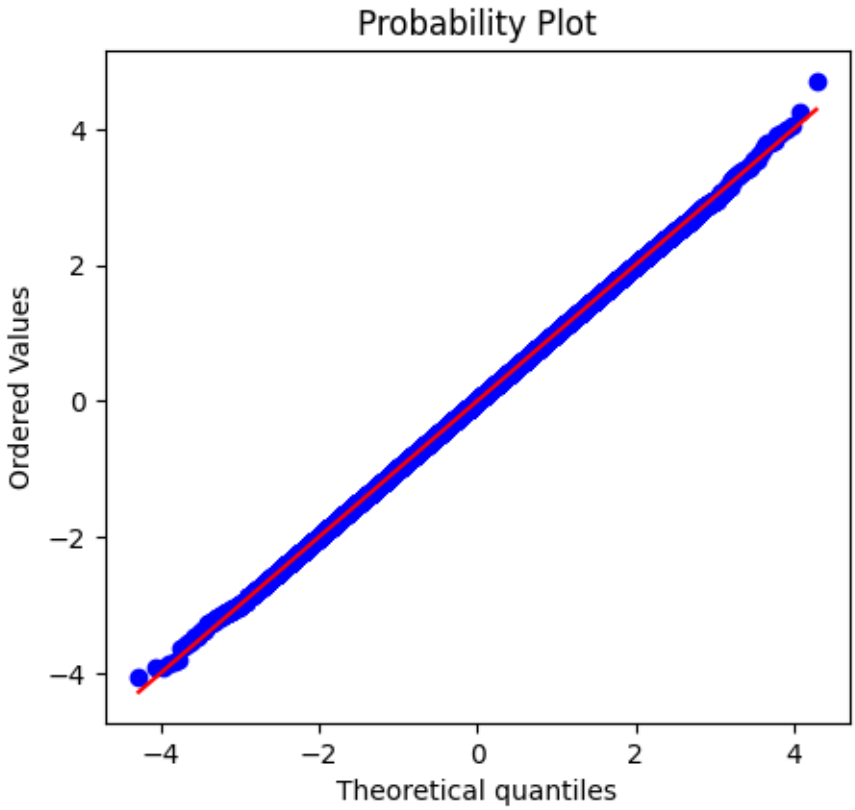


```
#random sampling from Exponential dist.
n = 100000
e = np.random.rand(n)
x = Ginv(e)
plt.hist(x, bins=50)
plt.show()
```




```
#acceptance-rejection
u = np.random.rand(n)
idx = u < (f(x) / (c*g(x)))
y = x[idx]

#signx
s = np.random.rand(len(y))
sign = (+1)*(s>0.5) + (-1)*(s<=0.5)
z  = y * sign

fig, ax = plt.subplots(2,1,figsize=(5,10))
ax[0].hist(z, bins=50)
stats.probplot(z, dist="norm", plot=ax[1])
plt.show()
```






```
# accept된 갯수, 통계량
z = pd.Series(z)
print("Size = ", len(z))
print("Mean = ", z.mean())
print("Std = ", z.std())
print("Skewness = ", z.skew())
```

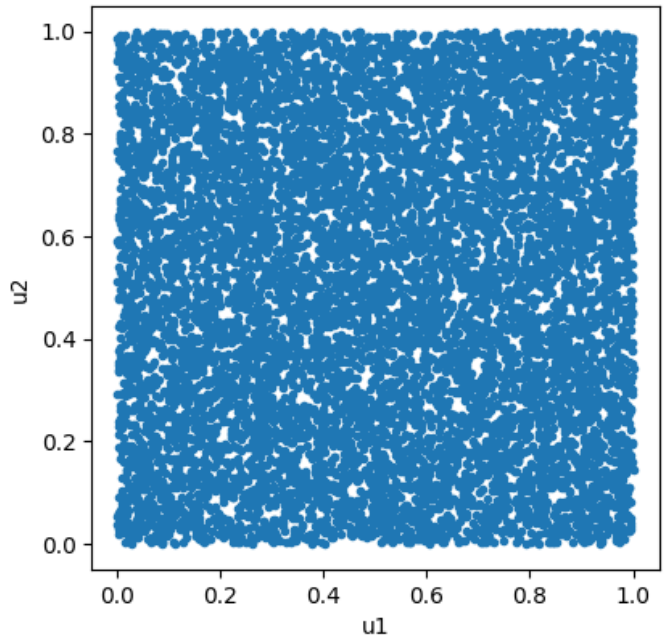
```
print("Kurtosis = ", z.kurt())
```

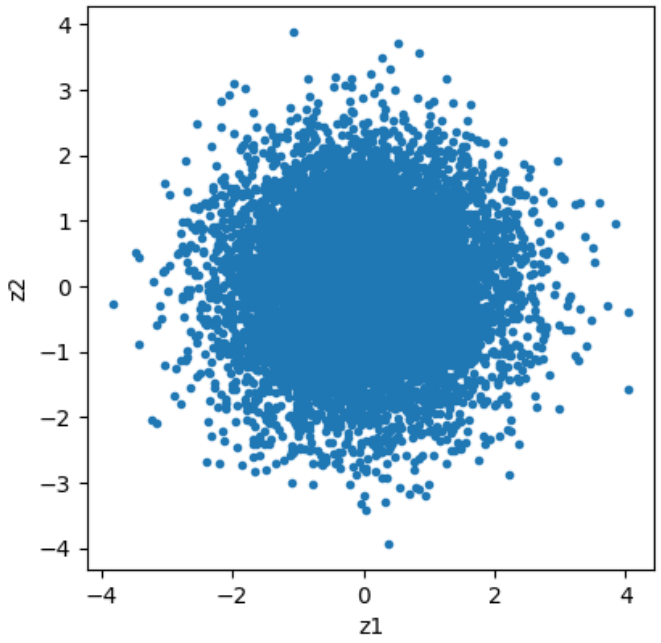
```
Size =  76242
Mean =  0.004399928352169691
Std =  0.9994359934475461
Skewness =  -0.0035700194262359955
Kurtosis =  -0.020022846863712473
```

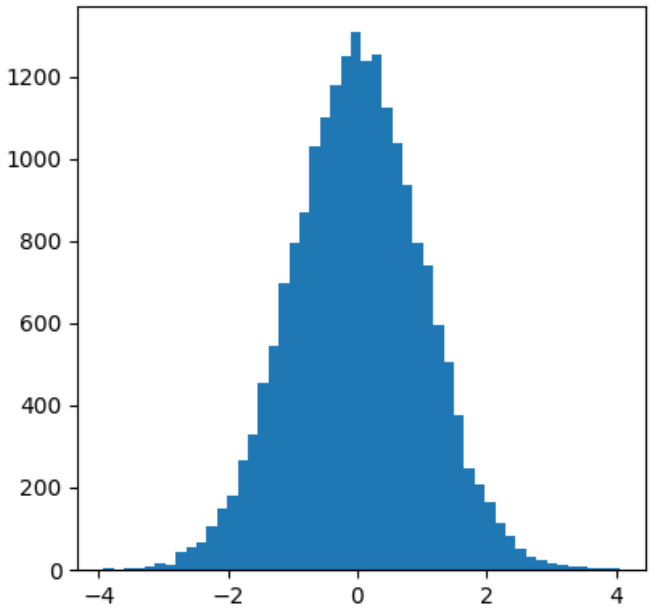
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import scipy.stats as stats


u1 = np.random.rand(10000)
u2 = np.random.rand(10000)

z1 = np.sqrt(-2*np.log(u1))*np.cos(2*np.pi*u2)
z2 = np.sqrt(-2*np.log(u1))*np.sin(2*np.pi*u2)

fig, ax = plt.subplots(2,2,figsize=(10,10))
ax[0,0].plot(u1,u2,'.')
ax[0,0].set_xlabel("u1")
ax[0,0].set_ylabel("u2")
ax[0,1].plot(z1,z2,'.')
ax[0,1].set_xlabel("z1")
ax[0,1].set_ylabel("z2")

z = np.concatenate([z1,z2])
ax[1,0].hist(z, bins=50)
stats.probplot(z, dist="norm", plot=ax[1,1])

z = pd.Series(z)
print("Mean = ", z.mean())
print("Std = ", z.std())
print("Skewness = ", z.skew())
print("Kurtosis = ", z.kurt())
```

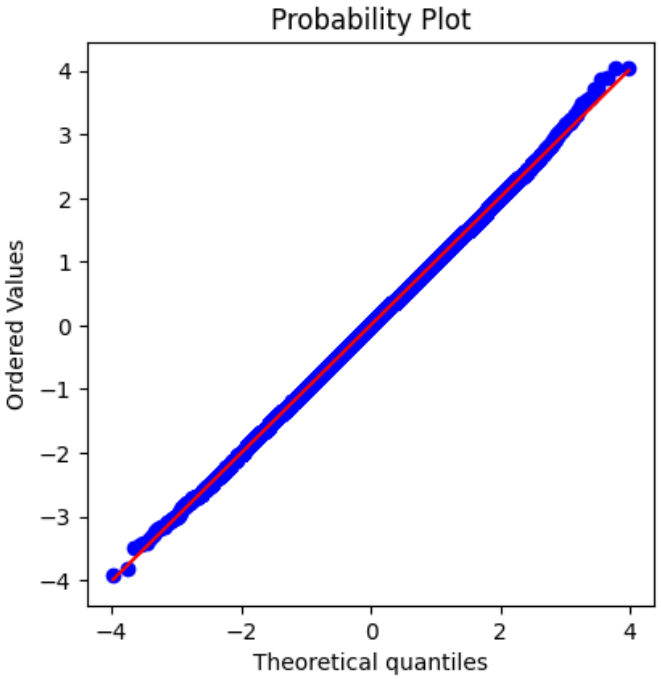
```
Mean =  0.005319050470821333
```

```
Std =  1.0041333833693569
Skewness =  0.007870653875667408
Kurtosis =  0.026864384771177363
```









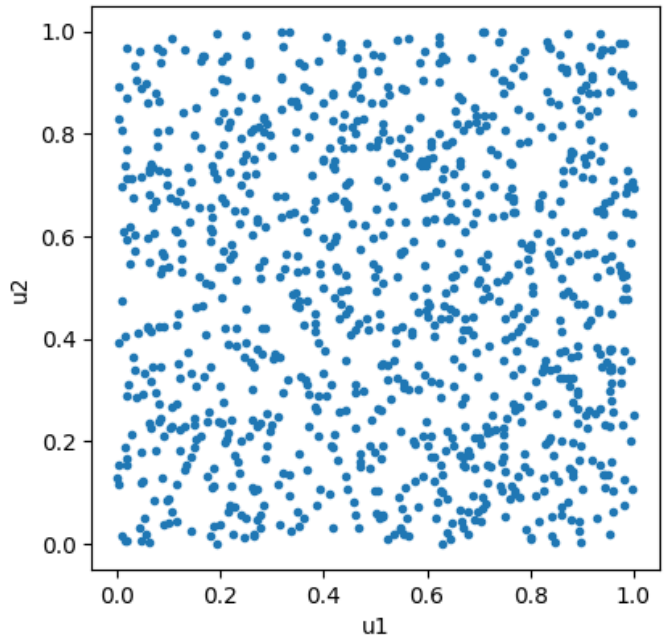
3.1.2 Box-muller method

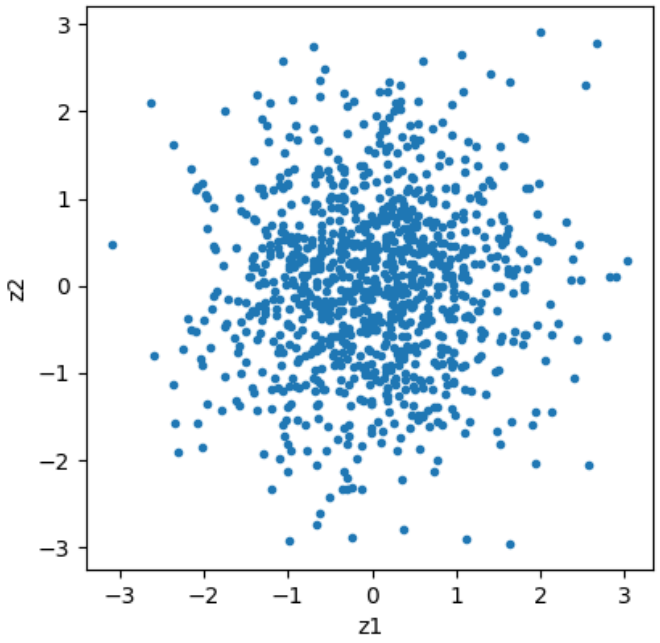
3.1.2.1 u1과 u2를 극좌표 변환(길이,각도)하여 표준정규난수를 생성

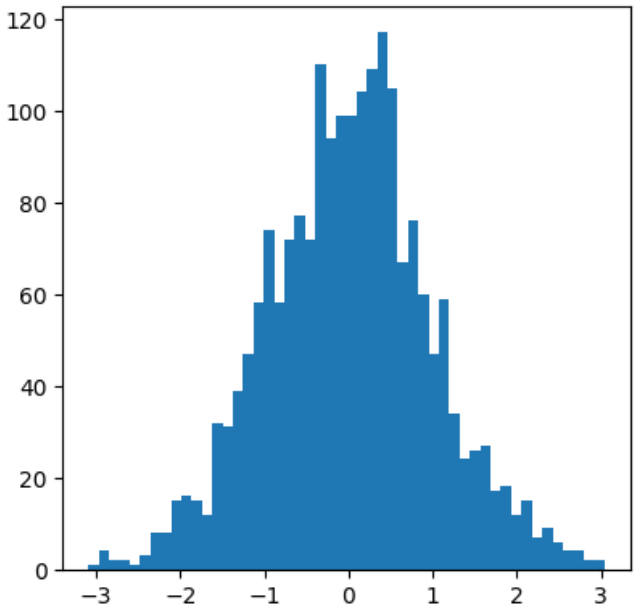
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import scipy.stats as stats
```

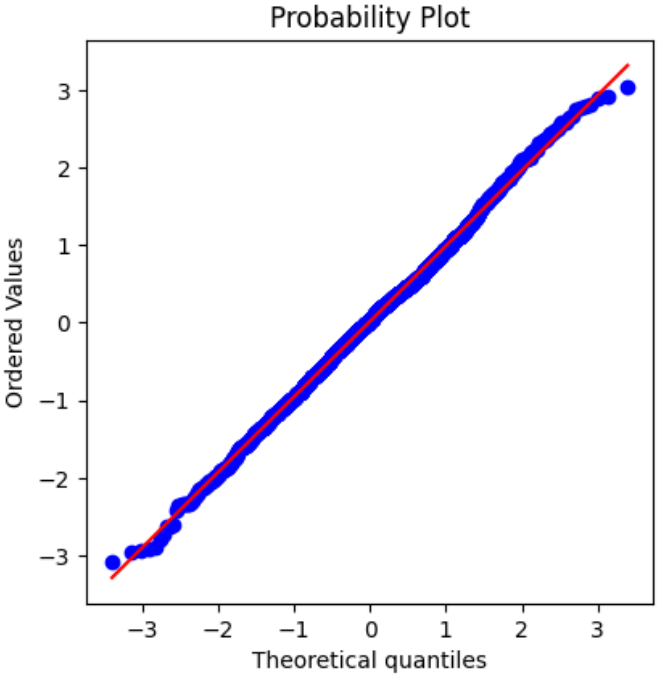
```
u1 = np.random.rand(1000)
# u1 = np.array(np.repeat(0.2,1000))

u2 = np.random.rand(1000)
# u2 = np.repeat(0.3,1000)

z1 = np.sqrt(-2*np.log(u1))*np.cos(2*np.pi*u2)
z2 = np.sqrt(-2*np.log(u1))*np.sin(2*np.pi*u2)

fig, ax = plt.subplots(2,2,figsize=(10,10))
ax[0,0].plot(u1,u2,'.')
ax[0,0].set_xlabel("u1")
ax[0,0].set_ylabel("u2")
ax[0,1].plot(z1,z2,'.')
ax[0,1].set_xlabel("z1")
ax[0,1].set_ylabel("z2")

z = np.concatenate([z1,z2])
ax[1,0].hist(z, bins=50)
stats.probplot(z, dist="norm", plot=ax[1,1])
```

```
((array([-3.39232293, -3.14126578, -3.00201262, ...,  3.00201262,
         3.14126578,  3.39232293]),
  array([-3.08879653, -2.96225972, -2.93767351, ...,  2.89101449,
         2.92098679,  3.03513649])),
 (0.9735473709334279, 0.01518582487043528, 0.9993181942933621))
```









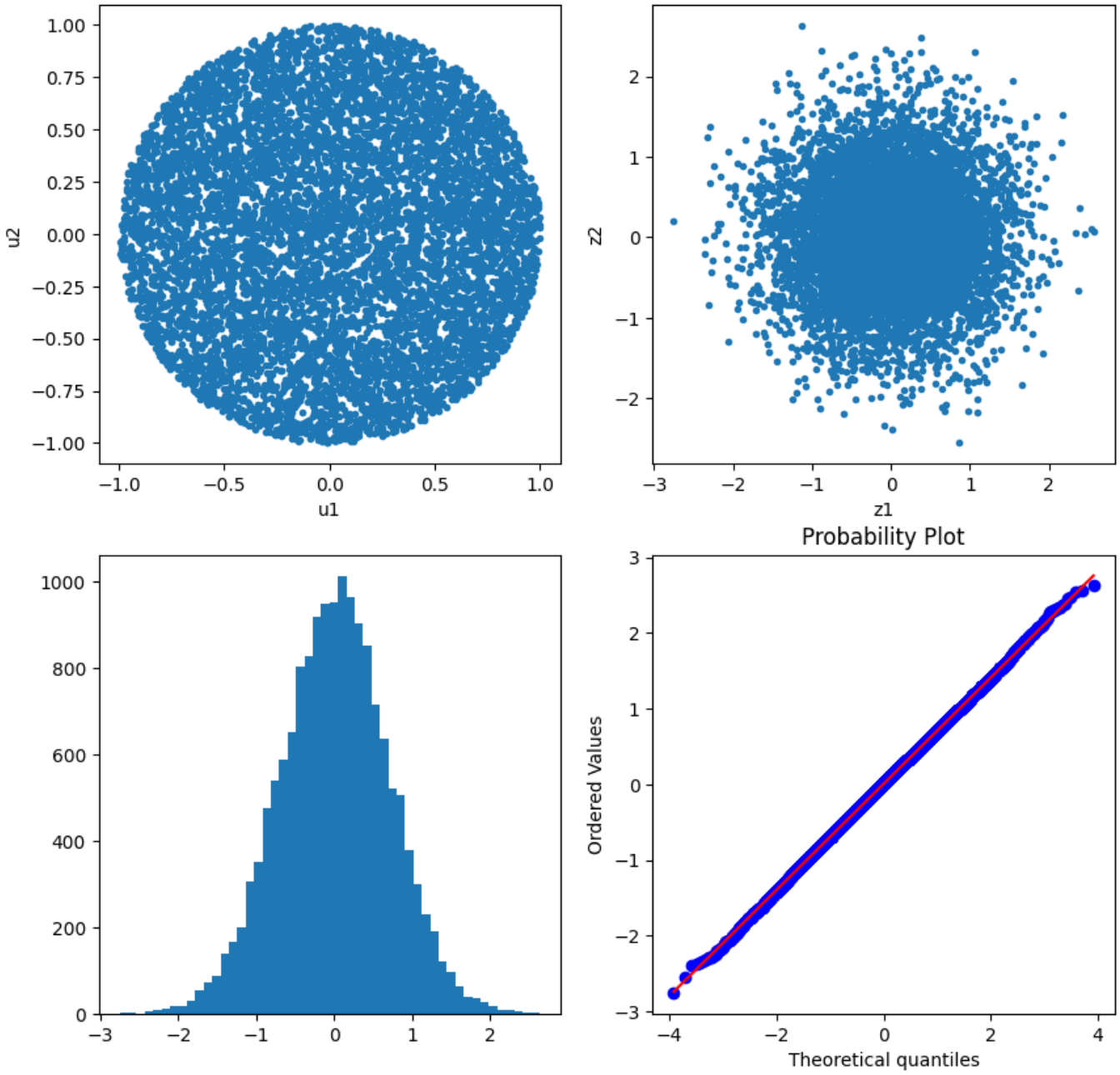

```
z = pd.Series(z)
print("Mean = ", z.mean())
print("Std = ", z.std())
print("Skewness = ", z.skew())
print("Kurtosis = ", z.kurt())
```

```
Mean =  0.015185824870435093
Std =  0.9729848649387326
Skewness =  0.026318252305075517
Kurtosis =  0.10017595589855377
```

3.1.3 Marsaglia's polar method

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import scipy.stats as stats

u1 = 2*np.random.rand(10000) - 1
u2 = 2*np.random.rand(10000) - 1
idx = u1**2+u2**2<1
u1 = u1[idx]
u2 = u2[idx]
r = np.sqrt(u1**2 + u2**2)
z1 = u1*np.sqrt(-2*np.log(r)/(r**2))
z2 = u2*np.sqrt(-2*np.log(r)/(r**2))

fig, ax = plt.subplots(2,2,figsize=(10,10))
ax[0,0].plot(u1,u2,'.')
ax[0,0].set_xlabel("u1")
ax[0,0].set_ylabel("u2")
ax[0,1].plot(z1,z2,'.')
ax[0,1].set_xlabel("z1")
ax[0,1].set_ylabel("z2")

z = np.concatenate([z1,z2])
ax[1,0].hist(z, bins=50)
stats.probplot(z, dist="norm", plot=ax[1,1])
```

```
((array([-3.92200263, -3.70288844, -3.58286194, ...,  3.58286194,
         3.70288844,  3.92200263]),
  array([-2.75062726, -2.54667074, -2.3831233 , ...,  2.54073674,
         2.5642353 ,  2.63545236])),
 (0.7004291238850505, 0.009082295652274254, 0.9999096754512135))
```




```
# 표준편차 이상함
z = pd.Series(z)
print("Mean = ", z.mean())
print("Std = ", z.std())
print("Skewness = ", z.skew())
print("Kurtosis = ", z.kurt())
```

```
Mean =  0.005214667686108389
Std =  0.7034156408834322
Skewness =  -0.010790249902950888
Kurtosis =  -0.022110372173252735
```

3.1.4 Correlated random

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

corr = np.array([[1,0.3,0.5],[0.3,1,0.6],[0.5,0.6,1]])
pos_def = np.all(np.linalg.eigvals(corr) > 0)
print(corr)
print(pos_def)

#%%
#Cholesky Decomposition
c = np.linalg.cholesky(corr)
x = np.random.randn(10000,3)
y = x @ c.T

y = pd.DataFrame(y, columns=['z1','z2','z3'])
print("Mean")
print(y.apply(['mean','std']))
print()

print("Correlation")
print(y.corr())
```

```
[[1.  0.3 0.5]
 [0.3 1.  0.6]
 [0.5 0.6 1. ]]
True
Mean
            z1        z2        z3
mean  0.005413  0.014955  0.002848
std   0.994726  0.999791  0.999323

Correlation
          z1        z2        z3
z1  1.000000  0.293904  0.482619
z2  0.293904  1.000000  0.601780
z3  0.482619  0.601780  1.000000
```

```
#Positive Definite 하지 않은 상관계수 행렬 생성
pos_def = True
while pos_def:
    x = np.random.randn(1000, 2)
    x = np.concatenate([x[:,0:1], x[:,0:1]+x[:,1:2], x[:,0:1]-2*x[:,1:2]], axis=1)
    corr = pd.DataFrame(x).corr()
    pos_def = np.all(np.linalg.eigvals(corr) > 0)

print(corr)
print(pos_def)
```

```
          0         1         2
0  1.000000  0.716544  0.460920
1  0.716544  1.000000 -0.288758
2  0.460920 -0.288758  1.000000
False
```

```
#cholesky: error
#c = np.linalg.cholesky(corr)

#Eigenvalue Decomposition
values, vectors = np.linalg.eig(corr)
values = np.maximum(0, values)
B = vectors @ np.diag(np.sqrt(values))
print(B)
print()
print(B @ B.T)
print()
```

```
[[ 0.          0.98180222  0.18990632]
 [ 0.          0.83597193 -0.54877221]
 [ 0.          0.28400185  0.95882373]]

[[ 1.          0.71654378  0.46092033]
 [ 0.71654378  1.         -0.28875824]
 [ 0.46092033 -0.28875824  1.        ]]
```

```
z = np.random.randn(10000,3)
y = z @ B.T
```

```
y = pd.DataFrame(y, columns=['z1','z2','z3'])
print("Mean")
print(y.apply(['mean','std']))
print()
print("Correlation")
print(y.corr())
```

```
Mean
            z1        z2        z3
mean  0.011488  0.003370  0.011480
std   1.001032  1.010722  0.994734

Correlation
          z1        z2        z3
z1  1.000000  0.722161  0.447696
z2  0.722161  1.000000 -0.295222
z3  0.447696 -0.295222  1.000000
```

```
#Singular value decomposition
print("=== original data ===")
print(pd.DataFrame(x).apply(['mean','std']))
print(pd.DataFrame(x).corr())
print()

U, S, Vh = np.linalg.svd(x)
np.allclose(U[:,:3] @ np.diag(S) @ Vh, x)

B = Vh.T @ np.diag(S) / np.sqrt(len(x))
z = np.random.randn(10000,3)
y = z @ B.T

print("=== simulation data ===")
y = pd.DataFrame(y, columns=['z1','z2','z3'])
print("Mean")
print(y.apply(['mean','std']))
print()
print("Correlation")
print(y.corr())
```

```
=== original data ===
             0         1         2
mean -0.031273 -0.024253 -0.045312
std   1.025723  1.426155  2.241957
          0         1         2
0  1.000000  0.716544  0.460920
1  0.716544  1.000000 -0.288758
2  0.460920 -0.288758  1.000000

=== simulation data ===
Mean
            z1        z2        z3
mean -0.006377 -0.013568  0.008004
std   1.027312  1.432960  2.227500

Correlation
          z1        z2        z3
z1  1.000000  0.721764  0.454958
z2  0.721764  1.000000 -0.287987
z3  0.454958 -0.287987  1.000000
```

시뮬레이션 방법론 과제1 (베리어옵션)

20249132 김형환

Question

시뮬레이션방법론 과제 1

아래 1번에 대해서는 파이썬 코드를 작성하여 제출하고, 2~3번에 대해서는 간단한 보고서(3장 이내)를 제출하세요.

1. 몬테카를로 시뮬레이션으로 베리어옵션(Barrier Option)의 가격을 계산하는 함수를 작성하시오.
   A. 베리어옵션은 Up-and-Out / Up-and-In / Down-and-Out / Down-and-In 의 네 가지 종류가 있으며, 각각 call 옵션과 put 옵션의 payoff 구조를 가질 수 있다.
   B. 기초자산은 1개로 옵션 평가일의 기초자산의 가격은 S이고, GBM 프로세스를 따른다.
   C. 옵션의 만기가 T (years)이고, 만기까지 베리어 knock 여부를 확인하는 관측시점은 같은 간격으로 m번 관측한다. ($\Delta t = T/m$)
   D. 옵션의 베리어는 B, 행사가격은 K이고, 무위험금리(연속복리) r과 변동성 σ는 상수라고 가정한다. (배당 = 0 으로 가정)
   E. 시뮬레이션의 replication 회수는 n번이다.

2. 베리어옵션의 In-Out parity에 대해서 조사하고, 몬테카를로 옵션 평가에서 활용할 수 있는 방법에 대해서 설명하시오.

3. m과 n을 변경할 때, bias와 variance의 변화를 시뮬레이션 결과로 설명하시오.

Answer 1

파라미터 및 알고리즘

먼저, MCS를 이용한 베리어옵션의 가격 계산에 필요한 파라미터는 아래와 같습니다.

```
s : 기초자산의 가격
k : 옵션의 행사가격
t : 옵션의 만기(연)
b : 옵션의 베리어
r : 무위험 금리
std : 기초자산의 변동성(표준편차)
UpDown : "U"이면 기초자산이 베리어보다 크면 knock, "D"이면 작으면 knock
InOut : "I"이면 Knock-in, "O"이면 Knock-out
CallPut : "C"이면 콜옵션, "P"이면 풋옵션
n : 시뮬레이션의 반복 횟수
m : 기초자산의 가격 관측 횟수
seed(=0) : 난수 생성의 최초 시드값
```

위 파라미터를 이용해 베리어옵션 가격 산출 함수를 구성할 계획이며, 알고리즘은 아래와 같습니다.

1. GBM을 따르는 기초자산의 가격 경로를 이산오일러근사를 이용하여 구성
2. 이를 통해, 관측된 m개의 기초자산의 가격과 초기값 S_0까지 m+1개의 기초자산 기준값 생성
3. 기초자산 기준값과 베리어를 비교하여 옵션 pay-off 발생 여부 판단
    - Up and Out : **모든** 기준값이 베리어보다 **작은** 경우, pay-off 발생
    - Down and Out : **모든** 기준값이 베리어보다 **큰** 경우, pay-off 발생
    - Up and In : 어느 기준값 중 **하나**라도 베리어보다 **큰** 경우, pay-off 발생
    - Down and In : 어느 기준값 중 **하나**라도 베리어보다 **작은** 경우, pay-off 발생
4. pay-off가 없으면 옵션가치는 0, 있으면 Call/Put 종류에 따라 pay-off를 계산하고, 그 현재가치가 이번 시뮬레이션의 옵션의 가치
5. 1~5를 n번 반복후 모든 옵션가치를 평균하여 최종적으로 베리어옵션의 가격 산출

이에 따른 Python 코드는 아래와 같습니다.

Python 구현

```python
import numpy as np


def GBM_path(s, r, std, t, m):
    dt = t/m
```

```
    z = np.random.standard_normal( m )
    ratio_path = np.exp((r-0.5*(std**2))*dt+std*np.sqrt(dt)*z)
    price_path = s*ratio_path.cumprod()
    return price_path

def BarrierOptionsPrice(s, k, t, b, r, std, UpDown, InOut, CallPut, n=10000,m=250):
    '''
    s : underlying price at t=0
    k : strike price
    t : maturity (year)
    b : barrier price
    r : risk-free rate (annualization, 1%=0.01)
    std : standard deviation of underlying return (annualization, 1%=0.01)
    UpDown : Up is "U", Down is "D" (should be capital)
    InOut : In is "I", Out is "O" (should be capital)
    CallPut : Call is "C", Put is "P" (should be capital)
    n : number of simulation
    m : number of euler-discrete partition
    '''
    barrier_simulation = np.zeros(n)
    for i in range(n):
        underlying_path = GBM_path(s,r,std,t,m)

        if UpDown=="U" and InOut=="O" :
            payoff_logic = 1 if np.sum(underlying_path>=b)==0 else 0
        elif UpDown=="U" and InOut=="I" :
            payoff_logic = 0 if np.sum(underlying_path>=b)==0 else 1
        elif UpDown=="D" and InOut=="O" :
            payoff_logic = 1 if np.sum(underlying_path<b)==0 else 0
        elif UpDown=="D" and InOut=="O" :
            payoff_logic = 0 if np.sum(underlying_path<b)==0 else 1

        if CallPut=="C" :
            plain_price = np.maximum(underlying_path[-1]-k,0)*np.exp(-r*t)
        elif CallPut=="P" :
            plain_price = np.maximum(k-underlying_path[-1],0)*np.exp(-r*t)

        barrier_simulation[i] = plain_price*payoff_logic
```

```
    barrier_price = barrier_simulation.mean()

    return barrier_price, barrier_simulation
```

저는 한번의 시뮬레이션에 이용되는 이산-오일러 구간마다의 기초자산가격 m개를,
먼저 m개의 z분포 변수를 생성하고, log-normal dist.에 따른 구간별 수익률로 변환 후,
해당 수익률을 누적곱하여 path를 생성하였습니다.
즉, 구간별 가격을 하나씩 생성하여 총 n*m번의 루프문을 작성하는 대신 n번의 루프문을 작성한 것 입니다.
또한 이러한 방식 외에도, 한번에 nm개의 z분포 변수를 생성하고, mn matrix로 변환하여
루프문 없이 한번의 행렬연산으로 MCS를 진행하는 방법도 생각해볼 수 있으나, (속도는 빠를 것으로 예상)
시뮬레이션의 횟수 n이 100만번 혹은 그 이상 커질 경우 메모리부족 등이 우려되어 고려하지 않았습니다.

Analytic Solution과 비교

해당 코드를 이용하여 베리어옵션 가격을 추정할 수 있으며, 이를 예재(QuantLib)의 결과값과 비교해보겠습니다.

시뮬레이션 파라미터는 n=10000, m=250으로 설정하였습니다.

```
import QuantLib as ql

S = 100; r = 0.03; vol = 0.2; T = 1; K = 100; B = 120; rebate = 0
barrierType = ql.Barrier.UpOut; optionType = ql.Option.Call

#Barrier Option
today = ql.Date().todaysDate(); maturity = today + ql.Period(T, ql.Years)

payoff = ql.PlainVanillaPayoff(optionType, K)
euExercise = ql.EuropeanExercise(maturity)
barrierOption = ql.BarrierOption(barrierType, B, rebate, payoff, euExercise)

#Market
spotHandle = ql.QuoteHandle(ql.SimpleQuote(S))
flatRateTs = ql.YieldTermStructureHandle(ql.FlatForward(today, r, ql.Actual365Fixed()))
flatVolTs = ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today, ql.NullCalendar(), vol,
bsm = ql.BlackScholesProcess(spotHandle, flatRateTs, flatVolTs)
analyticBarrierEngine = ql.AnalyticBarrierEngine(bsm)

#Pricing
barrierOption.setPricingEngine(analyticBarrierEngine)
```

```
QL_UOCprice = barrierOption.NPV()

# Hyeonghwan Pricing
HH_UOCprice, HH_UOCmatrix = BarrierOptionsPrice(S, K, T, B, r, vol, "U", "O", "C")

print("Up & Out Call with S=100, K=100, B=120, T=1, Vol=0.2, r= 0.03","\n",
    "QuantLib price :", QL_UOCprice,"\n",
    "Hyeonghwan price :", HH_UOCprice,"\n",
    "Difference is", QL_UOCprice - HH_UOCprice)
```

```
Up & Out Call with S=100, K=100, B=120, T=1, Vol=0.2, r= 0.03
 QuantLib price : 1.155369999815115
 Hyeonghwan price : 1.3261202813666222
 Difference is -0.17075028155150718
```

다음은 동일한 파라미터를 이용하여 Up and In Call Barrier Option price를 비교하였습니다.

```
Up & In Call with S=100, K=100, B=120, T=1, Vol=0.2, r= 0.03
 QuantLib price : 8.258033384037908
 Hyeonghwan price : 8.026630416874214
 Difference is 0.23140296716369413
```

비교결과, 대체로 유사하였으나 오차가 상당수준 발생하였습니다.

Up&Out에서는 MCS의 결과값이 크고 Up&In에서는 Analytic form의 결과값이 큰 경향이 있는데,

이는 이산-오일러 근사을 통해 Continuous 구간을 m개(discrete)로 나누면서 발생한 것으로 추정됩니다.

(모형 이산화 오류(Model Discretization Error)로 인해 편의(Bias) 발생)

즉, 실제 베리어 Knock 여부는 기초자산의 연속적인 가격흐름을 모두 관측하여 판단해야하지만,

이산화 과정에서 m번만 관측(m=250은 1일에 1번꼴)하게 되면서 그 사이의 가격을 관측할 수 없게 됩니다.

이로 인해 Knock-out 방식의 옵션은 고평가되고, Knock-in 방식의 옵션은 저평가되는 결과가 나타납니다.

이러한 편의는 m이 커질수록 작아져서 0으로 수렴하게 되며, 이에 대해서는 Answer3에서 다루겠습니다.

Answer 2

In-Out parity 정의

베리어옵션의 In-Out parity란, 특정 상황에서 베리어옵션과 plain vanilla option의 가격 사이에 성립하는 등식을 말합니다.

구체적으로 plain vanilla call option이 $c_{plain} = f(S, K, T, r, \sigma, d)$로 주어져있고,

베리어 B를 Knock할 때, 위 옵션과 동일한 pay-off를 제공하는 베리어옵션을 c_{In}, c_{Out}라고 한다면,

이들 옵션 사이에는 아래와 같은 등식이 성립하게 됩니다.

$$c_{In} + c_{Out} = c_{plain}$$

이는 풋옵션에서도 동일하게 성립되며, 일반적인 유로피안 옵션은 Knock-In + Knock-Out 배리어옵션으로 분해할 수 있다는 의미가 됩니다.

증명

예시를 통해 In-Out parity가 성립함을 쉽게 알 수 있습니다.

배리어가 B로 동일한 Knock-In & Out 옵션을 각각 I와 O라고 하겠습니다.

I는 lookback period동안 기초자산의 가격이 B를 한번이라도 Knock하는 경우 payoff가 발생합니다. (Up & Down 포괄)

O는 lookback period동안 기초자산의 가격이 B를 한번이라도 Knock하지 않는 경우 payoff가 발생합니다.

따라서, 기간동안 Knock가 발생하면 I는 payoff가 발생하고 O는 payoff가 0이 되며,

Knock가 발생하지 않으면 I는 payoff가 0이 되고 O는 payoff가 발생합니다.

즉, I+O로 구성된 배리어옵션 포트폴리오를 생각하면 모든 기초자산의 가격범위에 대하여 payoff가 한번 발생하고

해당 payoff는 plain vanilla 옵션의 payoff와 동일하므로 In-Out parity가 성립하게 됩니다.

이를 수식으로 표현하면 아래와 같습니다.

$$\begin{aligned} c_{In} + c_{Out} &= E^Q[e^{-rT}(S_T - K)^+ \mathbb{I}_{(\exists S_t \geq B)}] + E^Q[e^{-rT}(S_T - K)^+ \mathbb{I}_{(\forall S_t < B)}] \\ &= E^Q[e^{-rT}(S_T - K)^+](\mathbb{I}_{(\exists S_t \geq B)} + \mathbb{I}_{(\forall S_t < B)}) = E^Q[e^{-rT}(S_T - K)^+] \\ &= c_{plain} \; where \; \mathbb{I}_A = 1 \; if \; A \; is \; true \; else \; 0 \end{aligned}$$

이는 MCS방식으로 베리어옵션을 가치평가를 할 때에도 쉽게 알 수 있는데,

위 python코드에서 베리어옵션의 종류에 따라 payoff 발생여부를 판별할 때 사용한 if문에서

In, Out의 차이는 동전던지기의 앞뒷면처럼 상호배타적(mutually exclusive)임을 알 수 있습니다.

MCS에서의 활용

한종류의 베리어옵션과 plain 옵션의 가격을 알고 있다면 다른 한 종류의 베리어옵션의 가격이 결정되므로,

MCS를 이용하여 베리어옵션의 가격을 계산할 때 두번의 시뮬레이션을 한번으로 축소할 수 있을 것으로 생각해볼 수 있습니다.

그러나, 이는 현재 위 코드가 사전에 In, Out을 지정하고 한 경우에 대해서 return값을 반환하기 때문인데

이를 수정하여 Input으로 In, Out을 지정하지 않고 함수 내에서 In, Out 결과값을 각각 반환하게 한다면

시뮬레이션의 축소효과는 사라지게 됩니다.

더 나아가, 한번의 GBM경로를 생성하는 것에서 plain vanilla call&put, 배리어 In&Out, Up&Down옵션의 가격을

모두 산출할 수 있으므로 parity를 이용하여 시뮬레이션 시간을 극적으로 단축하기는 어려울 것 같습니다.

이외에도 산출된 결과값들끼리 parity를 이용해 적정성 여부를 검증하는 용도로는 활용성이 있을 것 같습니다.

이때에도, parity가 성립하려면 bsm fomula를 통한 plain vanilla옵션이 아닌,

베리어옵션과 동일한 이산오일러근사를 사용한 plain vanilla옵션의 가격을 사용해야 합니다.

Answer 3

베리어옵션의 실제 가격을 y_{real},

각 옵션가치의 평균인 MCS가격을 $\bar{y}$라고 하면,

$bias^2 = (y_{real} - \bar{y})^2$, $variance = S.E^2$라고 할 수 있습니다.

y_{real}을 QuantLib의 결과값이라고 가정하고, Up&Out call옵션을 통해 이들의 관계를 살펴보겠습니다.

> 실제 $S.E^2 = \frac{\sigma^2}{N} = \frac{variance}{N}$으로, 둘은 다릅니다.
> CLT에 따라 $\bar{y} \sim N(y_{real}, \frac{\sigma^2}{N})$이 되는데,
> 여기에서 $\bar{y}$ 표준편차의 불편추정량이 $S.E$이므로 이 둘을 구분해야합니다.
> 따라서, 실제로는 N 및 M을 설정하고, 이 MCS을 K(>30)회 반복하여 얻은 $\bar{y}_i$에 대해
> bias와 variace($var(\bar{y}_i)$)를 계산하는 것이 정확한 방법일 것으로 보입니다.
> 그러나 위 방법은 계산시간이 오래걸리고, 한번의 MCS로 얻은 본문의 결과값과 차이가 거의 없어
> 실제로 코드 구현 및 계산은 제외하였습니다.

```
import time
y_real = QL_UOCprice

N = 10000; M = 250

start = time.time()
```

```
y_mean, y = BarrierOptionsPrice(S, K, T, B, r, vol, "U", "O", "C", n=N, m=M)
end = time.time()

bias = (y_real - y_mean)**2
variance = y.std(ddof = 1) / np.sqrt(N)
cal_time = end-start

print("When N :",N," and M :",M,"\n",
"Bias^2 :",bias,"\n",
"Variance :", variance, "\n",
"MSE :", bias+variance, "\n",
"Calculation time :", cal_time, "second", "\n")
```

```
When N : 10000  and M : 250
 Bias^2 : 0.02674018607421331
 Variance : 0.03422571750506748
 MSE : 0.06096590357928079
 Calculation time : 0.19444012641906738 second
```

N을 증가시킬수록 시뮬레이션의 분산은 감소하나 편의는 감소하지 않는 경향이 있습니다.

```
When N : 50000  and M : 250
 Bias^2 : 0.026400732946555667
 Variance : 0.015379053616909322
 MSE : 0.04177978656346499
 Calculation time : 0.9767191410064697 second

When N : 100000  and M : 250
 Bias^2 : 0.024510323483535962
 Variance : 0.010767836858171948
 MSE : 0.03527816034170791
 Calculation time : 1.9618959426879883 second

When N : 500000  and M : 250
 Bias^2 : 0.020889259813925875
 Variance : 0.0048001322125535645
 MSE : 0.02568939202647944
 Calculation time : 9.752629041671753 second
```

M을 증가시킬수록 편의는 감소하나 분산은 유사한 경향이 있습니다.

```
When N : 10000  and M : 250
 Bias^2 : 0.023435288927162203
 Variance : 0.034158754044405604
 MSE : 0.05759404297156781
 Calculation time : 0.19353890419006348 second

When N : 10000  and M : 1000
 Bias^2 : 0.004344840618769495
 Variance : 0.03247146446652574
 MSE : 0.03681630508529524
 Calculation time : 0.42801475524902344 second

When N : 10000  and M : 5000
 Bias^2 : 0.0055788945107353395
 Variance : 0.03275229387502476
 MSE : 0.0383311883857601
 Calculation time : 1.6544427871704102 second

When N : 10000  and M : 10000
 Bias^2 : 0.00032630388897924364
 Variance : 0.031249596207288236
 MSE : 0.03157590009626748
 Calculation time : 3.19057297706604 second
```

N과 M을 증가시킬수록 계산시간도 증가하므로, 한정된 계산시간 하에 MSE를 최소화하도록 N과 M을 정해야할 필요가 있습니다.

Part IV

이자율파생상품('24 가을)

이자율파생상품 1주차

Part V

수치해석학('24 가을)

수치해석학 Ch1

금융 수치해석의 소개

강의 개요 : 금융수치해석의 필요성

주로 파생상품 평가와 최적화 방법론에 대해서 다룰 예정

파생상품 평가

$ds = rSdt + \sigma SdW^Q$

기하학적 브라운운동을 따르는 기초자산에 대한 파생상품의 가격 $f(t, S)$는 아래의 PDE로 표현됨

$f_t + \frac{1}{2}\sigma^2 S^2 f_{ss} + rSf_s - rf = 0$

이 블랙숄즈 미분방정식을 컴퓨터로 풀어내는 것이 주요 내용임

여기에는 반드시 연속적인 수식을 이산화하는 과정이 필요하며, 다양한 수치해석적인 기법이 활용됨

대표적으로 유한차분법(Finite Difference Method, FDM)이 존재

최적화 방법론

이외의 다양한 최적화방법론은 시간이 여유롭다면 이것저것 다룰 예정

- Minimum Variance Portfolio : Single-period에 대해 Sharpe ratio 극대화 등
- Stochastic programming : Multi-period에 대해 Minimum var 문제 해결 등
- Non-convex optimization : 미분을 통해 극값을 산출할 수 없는 경우의 최적화
- Parameter estimation 또는 Model calibration : $min_{\theta,\sigma,k} \sum (model\ price - market\ price)^2$ 와 같은 문제 등

컴퓨터 연산에 대한 이해

수치해석기법을 사용할 때 필연적으로 오차(error) 발생

1. Truncation error : 연속적인 수학적인 모델을 이산화하면서 발생하는 오차(e.g. 미분계수)
2. Rounding error : 컴퓨터 시스템상 실수(real number)를 정확히 표현할 수 없는 데에서 기인(2진법 vs. 10진법)

```
import numpy as np

a = 0.1

print(a+a+a==0.3,a+a+a+a==0.4)
```

```
False True
```

Rounding error 관련

컴퓨터가 실수를 나타내는 방법은 일반적으로 $x = \pm n \times b^e$ 로 나타냄.

여기서 n은 가수, e는 지수이며, 일반적으로 밑인 b는 2를 사용함.

컴퓨터에서 많이 사용하는 float타입 실수는 32bit를 사용하여 실수를 표현하며,

이는 $2^32$2가지로 모든 실수를 표현하게됨을 의미함. (정수는 int타입으로 모두 표현가능)

따라서 소수점에 따라 정확한 값을 나타내지 못하는 문제는 항상 존재.

Precision of floating point arithmetic

실수표현의 정밀도는 $float(1 + \epsilon_{math}) > 1$이 되는 가장 작은 ϵ_{math}를 의미

```
e = 1
while 1 + e > 1:
    e = e/2
e_math = 2 * e
print(e_math)
```

```
2.220446049250313e-16
```

내장함수 활용 가능. 파이썬에서는 기본적으로 64bit double타입을 사용함

```
import numpy as np
print(np.finfo(np.double).eps,
```

```
        np.finfo(float).eps)
```

```
2.220446049250313e-16 2.220446049250313e-16
```

```
print(1+e, 1+e+e, 1+2*e, 1+1.0000001*e)
```

```
1.0 1.0 1.0000000000000002 1.0000000000000002
```

많이 쓰이는 double타입의 경우 64bit로 실수를 표현하는데,

$x = \pm n \times 2^e$ 에서 부호(±) 1자리, 가수(n) 52자리, 지수 11자리(e)를 의미

계산오차

절대오차 : $|\hat{x} - x|$

상대오차 : $\frac{|\hat{x}-x|}{|x|}$

결합오차 : $e_{comb} = \frac{|\hat{x}-x|}{|x|+1}$

유한차분을 이용한 도함수의 근사

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

컴퓨터로는 $h \to 0$을 정확히 표현할 수 없음.

따라서, 적당히 작은 값으로 이를 대체하여 $f'(x)$를 근사해야함.

1. Truncation error 최소화를 위해서는 h는 작을 수록 좋음
2. 그러나, 너무 작은 값을 선택하면 rounding error가 발생하여 $x = x + h$ 될 가능성

Taylor expansion

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k + \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)^{n+1}$$

이를 도함수에 적용하면,

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{3!}f'''(x) + \cdots + \frac{h^n}{n!}f^{(n)}(x) + R_n(x+h)$$

$n = 1$을 적용하면,

$\Rightarrow \ f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(\xi) \ for \ \xi \in [x, x+h]$

$\Rightarrow\ f'(x) = \frac{f(x+h)-f(x)}{h} - \frac{h}{2}f''(\xi)\ (Forward\ Approximation)$

$n = 2$를 적용하고 forward - backward를 정리하면,

$f'(x) = \frac{f(x+h)-f(x-h)}{h} - \frac{h^2}{3}f'''(\xi)\ (Central\ Difference\ Approximation)$

::: {.callout, title="Central Difference Approximation"} $for\ n = 2,$

$(Forward)\ f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{3!}f'''(\xi_+),\ \xi \in [x, x+h]$

$(Backward)\ f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{3!}f'''(\xi_-),\ \xi \in [x-h, x]$

$f(x+h) - f(x-h) = 2hf'(x) + \frac{h^2}{6}\{f'''(\xi_+) + f'''(\xi_-)\}$

$\Rightarrow\ f'(x) = \frac{f(x+h)-f(x-h)}{h} - \frac{h^2}{3}f'''(\xi),\ \xi \in [x-h, x+h]$ :::

위의 식에서 볼 수 있는 것처럼, Central 방식에서는 truncation error의 order가 h^2이므로,

다른 방식에 비해서 오차가 훨씬 줄어들게 됨

유사한 방식으로 이계도함수와 편도함수를 유도하면,

$f''(x) = \frac{f(x+h)+f(x-h)-2f(x)}{h^2} - \frac{h^2}{24}f^{(4)}(\xi)$

$f_x(x,y) = \frac{f(x+h_x,y)-f(x-h_x,y)}{2h_x} + trunc.\ error$

총오차 및 최적의 h 산출

Forward difference approximation을 사용하고, $|f''(x)| <= M$이라고 하면,

$|f'_h(x) - f'(x)| = \frac{h}{2}|f''(x)| <= \frac{h}{2}M\ (trunc.\ error)$

유인물 참조

총오차 최소화를 위한 h^* 산출이 목표

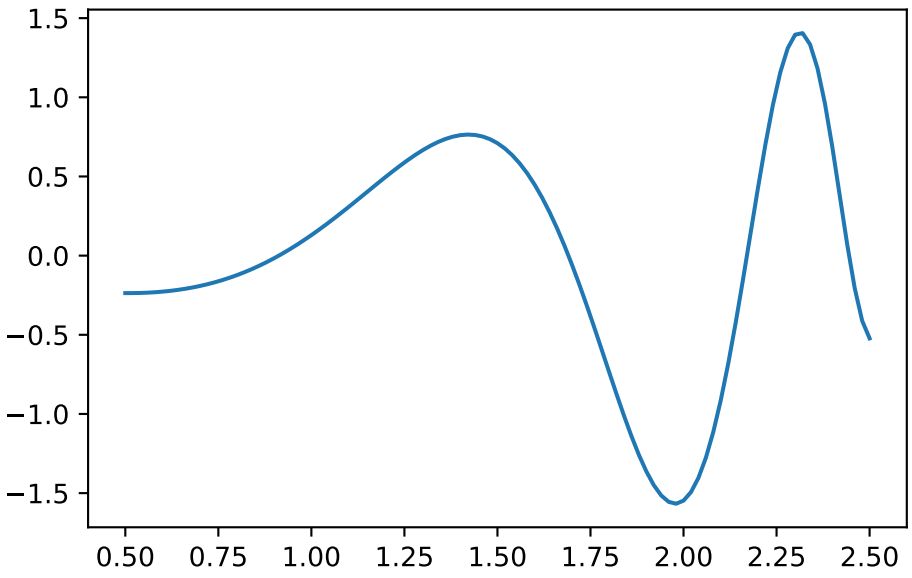
유한차분을 이용한 도함수 근사 예시

$f(x) = cos(x^x) - sin(e^x)$

함수 및 도함수(*analytic form*) 정의 및 도식화

```
import numpy as np
import matplotlib.pyplot as plt
def fun(x):
    return np.cos(x**x) - np.sin(np.exp(x))

def fprime(x):
    return -np.sin(x**x)*(x**x)*(np.log(x)+1)  - np.cos(np.exp(x))*np.exp(x)
```

```
x = np.linspace(0.5,2.5,101)
y = fun(x)
plt.plot(x,y,'-')
```



미분계수 산출

```
x = 1.5
d = fprime(x)
print("derivative = ", d)
```

```
derivative =  -1.466199173237208
```

***forward* 및 *central difference approx.* 산출 및 비교, 총오차를 *log scale*로 표현**

trunc. error는 h가 작아질수록 감소하지만 특정구간 이후에는 rounding error가 발생하므로

총오차는 항상 감소하지 않게 됨.

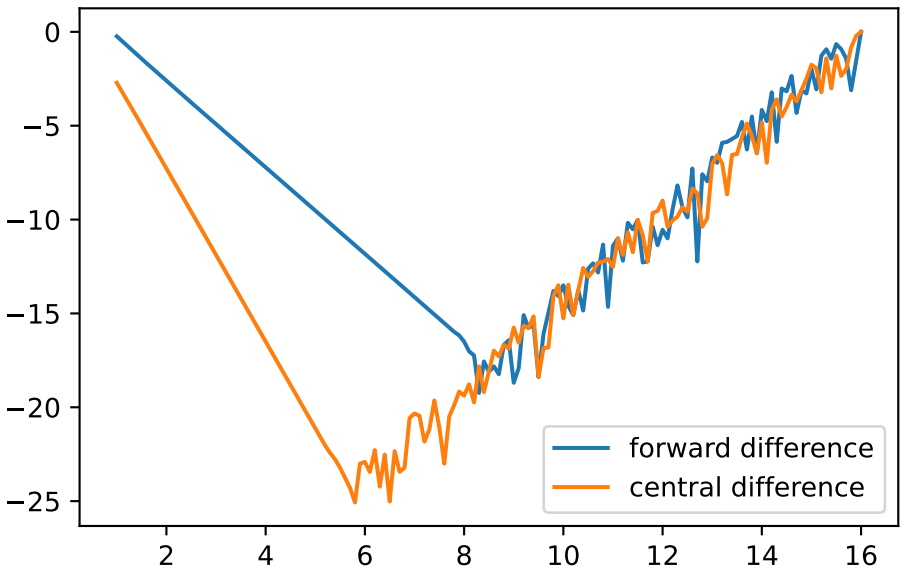
최적 h^*를 찾는 것이 매우 중요함

```
p = np.linspace(1,16,151)
h = 10**(-p)

def forward_difference(x,h):
    return (fun(x+h)-fun(x)) / h

def central_difference(x,h):
    return (fun(x+h)-fun(x-h)) / (2*h)
```

```
fd = forward_difference(x, h)
cd = central_difference(x, h)
print("forward = ", fd)
print("central = ", cd)

fd_error = np.log(np.abs(fd-d)/np.abs(d))
cd_error = np.log(np.abs(cd-d)/np.abs(d))
plt.plot(p,fd_error, p, cd_error)
plt.legend(['forward difference', 'central difference'])
```

```
forward =  [-2.62212289 -2.37366424 -2.17930621 -2.02733993 -1.90838758 -1.81511559
 -1.74184228 -1.68417626 -1.63872005 -1.60283855 -1.57448171 -1.55204964
 -1.53429022 -1.52022092 -1.50906909 -1.50022597 -1.49321118 -1.48764519
 -1.48322779 -1.47972134 -1.47693759 -1.47472735 -1.4729723  -1.4715786
 -1.47047179 -1.46959277 -1.46889464 -1.46834015 -1.46789975 -1.46754995
 -1.4672721  -1.46705142 -1.46687612 -1.46673689 -1.46662629 -1.46653844
 -1.46646866 -1.46641323 -1.46636921 -1.46633424 -1.46630646 -1.46628439
 -1.46626686 -1.46625294 -1.46624188 -1.4662331  -1.46622612 -1.46622058
 -1.46621618 -1.46621268 -1.4662099  -1.4662077  -1.46620594 -1.46620455
 -1.46620344 -1.46620257 -1.46620187 -1.46620131 -1.46620087 -1.46620053
 -1.46620025 -1.46620002 -1.46619985 -1.46619971 -1.46619959 -1.46619951
 -1.46619944 -1.46619939 -1.46619934 -1.46619931 -1.46619927 -1.46619923
 -1.46619922 -1.46619918 -1.46619921 -1.46619915 -1.46619915 -1.46619919
 -1.46619909 -1.46619907 -1.46619916 -1.46619915 -1.46619876 -1.46619938
 -1.46619893 -1.46619919 -1.46619932 -1.46619869 -1.46619769 -1.4662003
 -1.46619716 -1.46619985 -1.46619876 -1.46620071 -1.46619865 -1.46619427
 -1.46619269 -1.46620314 -1.46618158 -1.46619854 -1.46618273 -1.46617469
 -1.46619173 -1.46614311 -1.46615961 -1.46626449 -1.46620595 -1.46619201
 -1.46615356 -1.46621617 -1.46616053 -1.46617469 -1.46608615 -1.46578868
 -1.46632693 -1.46612406 -1.46518938 -1.46619201 -1.46545305 -1.46568705
 -1.46438417 -1.46757238 -1.46221506 -1.46202287 -1.46130718 -1.46050672
 -1.45413969 -1.46897416 -1.45004198 -1.46392328 -1.44328993 -1.4535955
 -1.40766793 -1.46202287 -1.39437708 -1.40433339 -1.32596324 -1.44671698
 -1.40100674 -1.41101039 -1.66533454 -1.39768798 -1.05575095 -0.88607446
 -1.11550166 -0.70216669 -0.88397549 -1.11285921 -1.40100674 -1.76376299
  0.         ]
central =  [-1.5635526  -1.52856423 -1.50592274 -1.49141188 -1.48216656 -1.4762975
 -1.47258018 -1.47022905 -1.46874334 -1.46780503 -1.46721263 -1.46683872
 -1.46660274 -1.46645382 -1.46635985 -1.46630056 -1.46626314 -1.46623954
```

```
-1.46622464 -1.46621524 -1.46620931 -1.46620557 -1.46620321 -1.46620172
-1.46620078 -1.46620019 -1.46619981 -1.46619958 -1.46619943 -1.46619933
-1.46619927 -1.46619924 -1.46619921 -1.4661992  -1.46619919 -1.46619918
-1.46619918 -1.46619918 -1.46619918 -1.46619917 -1.46619917 -1.46619917
-1.46619917 -1.46619917 -1.46619917 -1.46619917 -1.46619917 -1.46619917
-1.46619917 -1.46619917 -1.46619917 -1.46619917 -1.46619917 -1.46619917
-1.46619917 -1.46619917 -1.46619917 -1.46619917 -1.46619917 -1.46619917
-1.46619918 -1.46619917 -1.46619917 -1.46619917 -1.46619917 -1.46619917
-1.46619917 -1.46619917 -1.46619918 -1.46619918 -1.46619917 -1.46619916
-1.46619918 -1.46619915 -1.46619918 -1.46619915 -1.46619923 -1.46619922
-1.46619909 -1.46619924 -1.46619938 -1.46619908 -1.46619894 -1.46619938
-1.46619879 -1.46619919 -1.4661991  -1.46619925 -1.46619804 -1.46620118
-1.46619883 -1.46620125 -1.46619876 -1.46620071 -1.46620423 -1.46619603
-1.4661949  -1.46620592 -1.46619208 -1.46620736 -1.46619383 -1.46617469
-1.46620932 -1.46616527 -1.4661875  -1.46626449 -1.46622805 -1.46619201
-1.46629366 -1.46630436 -1.46638257 -1.46624457 -1.46626211 -1.46612096
-1.46632693 -1.4662996  -1.46585236 -1.46647023 -1.46615356 -1.46612799
-1.46493928 -1.46827122 -1.46485444 -1.46645324 -1.46409593 -1.46401756
-1.4607695  -1.47732061 -1.46054953 -1.46392328 -1.45439216 -1.46757238
-1.44285963 -1.50632659 -1.45015216 -1.43944172 -1.41436079 -1.50235994
-1.40100674 -1.58738669 -1.72084569 -1.67722557 -1.40766793 -1.10759308
-1.39437708 -1.05325004 -1.32596324 -1.66928882 -2.10151011 -2.64564449
 0.        ]
```



수치적 불안정성과 악조건

수치적 불안정성 : 알고리즘이 rounding error를 증폭시켜 결과값이 크게 달라짐

악조건 : input data의 작은 변동이 output solution에 큰 변화를 일으킴

행렬의 조건수

문제 f(x)의 해가 x(input)에 얼마나 영향을 받는지 나타내는 값

탄력성의 절대값 : $cond(f(x)) \approx \frac{|xf'(x)|}{|f(x)|}$

탄력성의 절대값이 크면 악조건임

Linear system에서 행렬의 조건수 $k(A) = ||A^{-1}||\ ||A||$

$> 1/\sqrt{eps} \approx 6.7 \times 10^7$ 이면 약조건 우려

알고리즘의 계산 복잡도

실행시간을 많이 다룰거임.

알고리즘 복잡도

order가 중요함

big-O를 표현식으로 쓰는데, 계산효율성이나 오차크기를 나타낼때 씀

$O(n^2)$: 데이터를 10배 늘리면 계산이 100배 늘어남

$O(n^{-2})$: 데이터를 10배 늘리면 오차가 100배 감소함

수치해석학 Ch2

선형방정식 및 최소자승법

PLU분해 예시

```
import numpy as np
from numpy.linalg import cholesky
from scipy.linalg import lu

A = np.arange(1,10)
A = A.reshape(3,3)
print(A)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

```
P,L,U = lu(A)

P@L@U
```

```
array([[1., 2., 3.],
       [4., 5., 6.],
       [7., 8., 9.]])
```

Cholesky factorization

촐레스키 분해

$Ax = b$에서, A가 대칭이고 positive-definite인 경우 적용 가능.

PLU보다 연산량이 적음.

다변량 정규분포 난수를 생성할 때, 공분산행렬에 촐레스키 분해를 적용하면 쉽게 생성 가능.

QR 분해

4 수치해석기법 실습

4.1 Linear system of equation

Part VI

금융시장 리스크관리('24 가을)

금융시장 리스크관리 1~2주차

Lecture2 : How Traders Manage Their Risks?

Greeks letters & Scenario analysis

Delta hedging

$\Delta = \frac{\partial P}{\partial S}$

가장 기본적인 헷징방법으로, 파생상품과 같은 금융상품으로 구성된 포트폴리오에 대해

기초자산의 가격변동에 대한 민감도인 델타를 계산하여 이를 0으로 만듦으로써

기초자산의 가격변화로 인한 포트폴리오의 가치변화를 0으로 만드는 방법.

포트폴리오의 payoff가 선형이라면, 한번의 헷징만으로 완전헷지(perfect hedge)가 가능

이를 Hedge and forget이라고 함.

그러나 비선형이라면, 기초자산의 가격변동에 따라 델타도 변하게 됨.

> 델타헷지 예시
>
> 은행이 특정 주식 10만주에 대한 콜옵션을 30만불에 매도할 수 있음.
> 블랙숄즈공식에 따른 이 옵션의 가치는 24만불 ($S_0 = 49, K = 50, r = 0.05, \sigma = 0.2, T = 20w$)
> 어떻게 6만불의 차익거래를 실현시킬지?
>
> 1. 풋콜페리티 또는 시장에서 동일한 옵션을 24만불에 매수하여 실현
> 2. 그러나, 옵션매수가 불가능한 경우 기초자산 주식을 이용한 델타헷징을 반복
>
> 즉, 옵션 매도포지션의 델타만큼 주식을 매수하고 매주 리밸런싱
> 20주 후 주식 매도수를 반복하여 구축한 델타헷징은 약 26만불의 비용이 발생하였음
> -> 약 4만불의 차익거래를 실현함
> 2만불은 어디로 증발함? : 델타헷징에 드는 비용 (거래비용 등)
> 헷지를 자주할수록, 거래비용이 적을수록, 기초자산의 가격변동이 작을수록 차익은 6만불로 수렴

기초자산을 이용한 델타헷징은 비용이 발생할수밖에 없음.

콜옵션을 기준으로 할 때, 기초자산의 가격이 상승하면 콜옵션의 머니니스가 증가하면서 델타가 증가함.

콜옵션 매도를 델타헷징하다보면, 주가 상승 -> 델타 상승 -> 주식 매수

반대로, 주가 하락 -> 델타 감소 -> 주식 매도

즉, 주식이 오르면 팔고 내리면 팔아야함 (Sell low, Buy high Strategy)

기타 그릭스

Gamma ($\Gamma = \frac{\partial \Delta}{\partial S} = \frac{\partial^2 P}{\partial S^2}$)

베가 로 그런거는 대충넘어갔음

Taylor Series Expansion

테일러 전개는 다항전개식의 일종으로, 복잡한 함수를 다항함수를 이용하여 간단히 전개할 수 있어 근사식에 많이 활용

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2}f''(x_0)(x - x_0)^2 + \cdots$$

금융시장에서 이를 적용한다면? $f(x)$는 포트폴리오의 가격함수이며, x는 기초자산가격으로 대입 가능

$\Rightarrow f(x) - f(x_0) = f'(x_0)(x - x_0) + \frac{1}{2}f''(x_0)(x - x_0)^2$

$\Rightarrow \Delta f(x) = f'(x_0)\Delta x + \frac{1}{2}f''(x_0)\Delta x^2$

기초자산의 변화(Δx)에 따른 포트폴리오 가치변화(Δf)는 델타(듀레이션) 및 감마(컨벡시티)로 근사 가능

포트폴리오 P를 기초자산의 가격 및 시간에 따른 함수 $P(S, t)$라고 한다면, (변동성은 상수로 가정)

$$\Delta P = \frac{\partial P}{\partial S}\Delta S + \frac{\partial P}{\partial t}\Delta t + \frac{1}{2}\frac{\partial^2 P}{\partial S^2}\Delta S^2 + \frac{1}{2}\frac{\partial^2 P}{\partial t^2}\Delta t^2 + \frac{\partial^2 P}{\partial S \partial t}\Delta S \Delta t + \cdots$$

일반적으로 $\Delta t^2 = 0, \Delta S \Delta t = 0$으로 가정하므로,

$$\Rightarrow \Delta P \approx \frac{\partial P}{\partial S}\Delta S + \frac{\partial P}{\partial t}\Delta t + \frac{1}{2}\frac{\partial^2 P}{\partial S^2}\Delta S^2$$

즉, 포트폴리오의 가치변화는 델타, 세타, 감마로 표현되며 델타중립 포트폴리오를 구성했다면,

$$\Delta P = \Theta \Delta t + \frac{1}{2}\Gamma \Delta S^2$$

> **Note**
>
> 아래로 볼록한 형태인 **옵션 매수는 positive gamma**,
> 위로 볼록한 형태인 **옵션 매도는 negative gamma** (관리 어려움)

만약 변동성이 변수라면?

$\Delta P = \delta \Delta S + Vega \Delta \sigma + \Theta \Delta t + \frac{1}{2}\Gamma \Delta S^2$

Hedging in practice

델타헷징은 보통 매일하고, 감마나 베가는 영향이 매우 크지는 않아서 모니터링하다가,

일정 임계치를 넘어가면 헷지 시작(헷지도 어렵고 비용도 보다 많이 듬)

특히, 만기가 임박한 ATM옵션은 감마와 베가가 매우 크므로, 주로 관리하게됨

Lecture3 : Volatility

Standard approach to estimating Volatility

$\sigma_n^2 = \frac{1}{m-1}\sum_{i=1}^{m}(u_{n-i} - \bar{n})^2 \; for \; u_i = \ln(\frac{S_i}{S_{i-1}})$

Simplify, $\sigma_n^2 = \frac{1}{m}\sum_{i=1}^{m} u_{n-i}^2 \; for \; u_i = \frac{S_i - S_{i-1}}{S_{i-1}}, \bar{u} = 0$

Weighting Schemes

$\$\sigma_n^2 = \sum_{i=1}^{m} \alpha_i u_{n-i}^2 \; for \; \sum_i \alpha_i = 1$

EWMA(Exponentially Weighted Moving Average) : $\alpha_{i+1} = \lambda\alpha_i \; where \; 0 < \lambda < 1$

ARCH, GARCH 등등 많음

최대우도법, Maximum Likelihood Method

최대우도법이란, 우리에게 주어진 데이터가 있고, 이 데이터가 어떠한 분포를 따르는지 추정하기 위함.

1. 주어진 데이터가 있고
2. 어떤 분포를 따르는지 사전에 설정함
3. 분포에 따라 추정이 필요한 파라미터 θ_n이 생길 때,
4. 주어진 데이터에 대한 확률밀도함수의 곱(독립된 결합밀도함수)을 최대화시키는 θ를 찾는 것이 목표
5. 즉, 확률을 최대화시키는 파라미터를 추정하여 추정분포를 결정함

주가수익률의 관측치 u_i가 평균이 0인 정규분포를 따른다고 가정한다면?

변동성 σ를 추정하기 위해 최대우도법을 사용할 수 있음.

Maximize : $ML = \Pi_{i=1}^{n}[\frac{1}{\sqrt{2\pi\sigma^2}}e^{\frac{-u_i^2}{2\sigma}}]$

$y = x$와 $y = \ln x$는 일대일대응관계가 성립하므로, log transform을 통해

Same to maximize : $\ln ML = \sum_{i=1}^{n}[\ln(\frac{1}{\sqrt{2\pi\sigma^2}}e^{\frac{-u_i^2}{2\sigma^2}})] = n\ln(\frac{1}{\sqrt{2\pi\sigma^2}}) - \frac{1}{2\sigma^2}\sum_{i=1}^{n} u_i^2$

위 식을 σ에 대해 다시 정리하면, $n\ln(\frac{1}{\sqrt{2\pi}}) - \frac{n}{2}\ln(\sigma^2) - \sum u_i^2 \frac{1}{2\sigma^2}$

σ^2에 대해 미분을 통해, $\ln ML_{\sigma^2} = -\frac{n}{2\sigma^2} + \frac{\sum u_i}{2(\sigma^2)^2}$

미분계수가 0인 점이 ML 함수를 극대화 시키는 점이므로, $-\frac{n}{2\sigma^2} + \frac{\sum u_i}{2(\sigma^2)^2} = 0$

$\Rightarrow\ n\sigma^2 = \sum u_i,\ \therefore \sigma^2 = \frac{\sum u_i}{n}$

Characteristics of Volatility

상수는 아님

근데 경향성이 있음 (persistence), 따라서 모아놓으면 군집화 경향이 있음 (Clustering)

평균회귀 성향이 있음 (mean reverting)

주가수익률과 음의 상관관계가 있음. (경기침체에 변동성 증가)

근데 EWMA, GARCH는 이런 음의 상관관계를 반영하지는 않음

How Good is the Model?

변동성 모델을 평가할 때, 일반적으로 $u_n \sim N(0, \sigma_n^2)$을 따르므로

$\frac{u_n}{\sigma_n} \sim Z$를 통해서 검증함.

이 의미는, 매일매일 자산수익률과 모델변동성을 통해 독립된 Z분포를 따르는 $z_n = \frac{u_n}{\sigma_n}$을 생성할 수 있고

이 z_n은 서로 독립인지를 봄으로써 검증할 수 있음.

이건 Ljung-Box Test로 널리 알려져 있음.

z_n을 통해 autocorrelation=0(H0)임을 검증하는 테스트임.

금융시장 리스크관리 과제1

Group 1 (김형환, 염지아, 유문선, 이희예, 홍지호)

Question 1-3

GARCH (1,1)

2) Estimate parameters for the GARCH (1,1) model on the KOSPI index data **over the most recent 1000 days** using the maximum likelihood method. Use the Solver tool in Excel.[2] To start the GARCH calculation, set the **variance forecast at the end of the first day** equal to the square of the return on that day.

(Hint: The total sample period is from July 7, 2016 to Aug 3, 2020. The return observations are available from July 8, 2016 to Aug 3, 2020, and the variance and likelihood estimates for MLE are available from July 11, 2016 to Aug 3, 2020.)

3) What is the annualized volatility estimate at the end of August 3, 2020 based on the GARCH (1,1) model?

(Hint: Note that this is the volatility estimate for August 4, 2020.)

Answer

주어진 기간의 코스피지수에 대한 GARCH(1,1) 모델의 파라미터는 아래와 같습니다.

$\omega = 0.00000363,\ \alpha = 0.115,\ \beta = 0.844$

이 모델을 이용하여 추정한 **2020년 8월 3일 장 종료 후 코스피지수의 연환산 변동성은 약 13.4%**입니다.

위 내용의 산출과정은 아래와 같습니다.

1. 주어진 코스피 지수의 일별 값(Sheet1)을 C열에 채운다. (vlookup 사용)
2. 코스피 지수의 일별 산술수익률을 D열에 채운다. ($r_i = \frac{p_i - p_{i-1}}{p_{i-1}}$)
3. GARCH(1,1) 모델의 파라미터 초기값을 이용하여 당일 장 종료 시점의 추정 분산을 산출하고, E열에 채운다. ($\sigma_i^2 = \omega + \alpha r_i^2 + \beta \sigma_{i-1}^2$)
4. 일별 추정분산을 이용하여 일별 로그우도값($LH = -\ln \sigma_i^2 - \frac{r_i^2}{\sigma^2}$)을 계산하여 F열에 채우고, 이를 모두 더하여 전체 우도값을 산출한다.
5. 전체 우도값을 최대화시키는 파라미터를 solver 기능을 이용하여 추정한다.
6. 적정 파라미터를 추정하였으면, 8/3일의 분산값을 통해 연환산 변동성을 산출한다. ($\sigma_{annual} = \sqrt{252}\sigma_{8.3}$)

Question 4-6

EWMA

4) Estimate parameters for the EWMA model on the KOSPI index data **over the most recent 1000 days** using the maximum likelihood method. Use the Solver tool in Excel. To start the EWMA calculation, set the **variance forecast at the end of the first day** equal to the square of the return on that day.

(Hint: The total sample period is from July 7, 2016 to Aug 3, 2020. The return observations are available from July 8, 2016 to Aug 3, 2020, and the variance and likelihood estimates for MLE are available from July 11, 2016 to Aug 3, 2020.)

5) What is the annualized volatility estimate at the end of August 3, 2020 based on the EWMA model?

(Hint: Note that this is the volatility estimate for August 4, 2020.)

Comparison Between GARCH (1,1) and EWMA

6) Plot a line graph that shows the GARCH (1,1) and EWMA volatility estimates (primary axis) along with the KOSPI index level (secondary axis) between **January 2, 2018 and August 4, 2020**. Explain your findings.

Answer 4-5

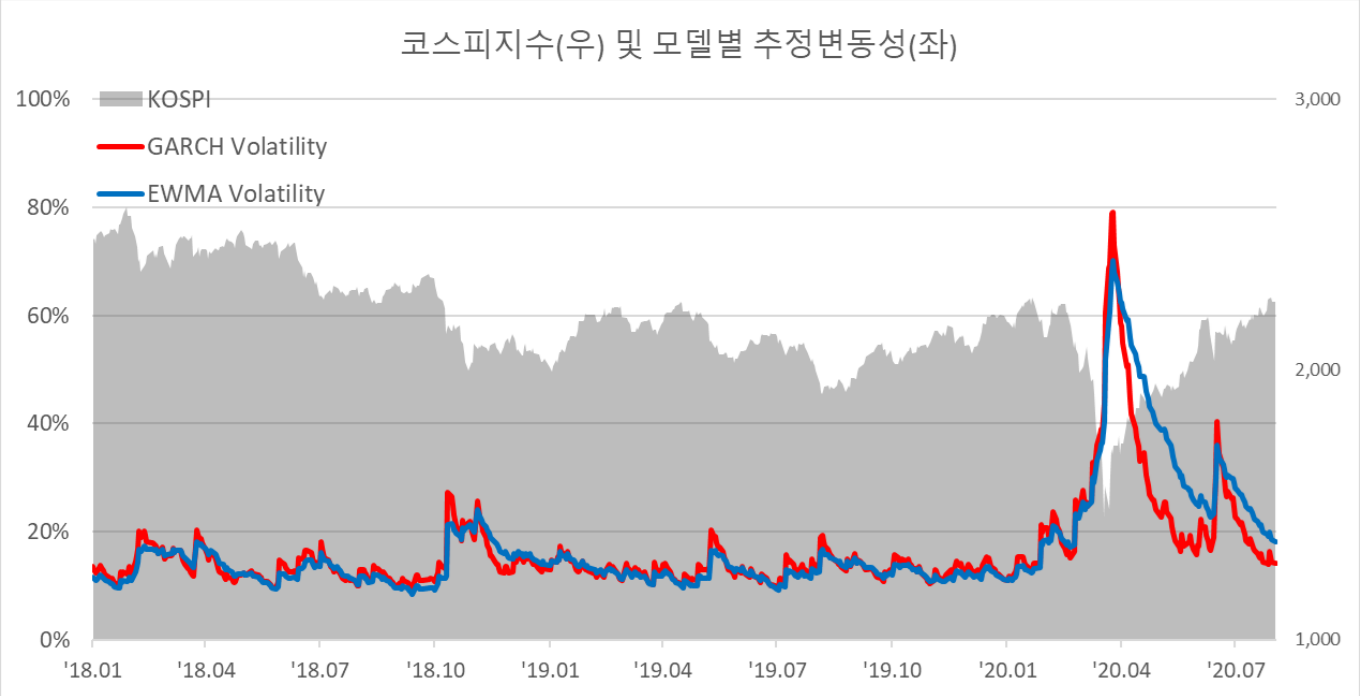
주어진 기간의 코스피지수에 대한 EWMA 모델의 파라미터 $\lambda = 0.934$입니다.

이 모델을 이용하여 추정한 **2020년 8월 3일 장 종료 후 코스피지수의 연환산 변동성은 약 17.6%**입니다.

위 내용의 산출 과정은 아래와 같습니다.

1. 주어진 코스피 지수의 일별 값(Sheet1)을 C열에 채운다. (vlookup 사용)
2. 코스피 지수의 일별 산술수익률을 D열에 채운다. ($r_i = \frac{p_i - p_{i-1}}{p_{i-1}}$)
3. EWMA 모델의 람다 초기값을 이용하여 당일 장 종료 시점의 추정 분산을 산출하고, E열에 채운다. ($\sigma_i^2 = \lambda\sigma_{i-1}^2 + (1-\lambda)r_i^2$)
4. 일별 추정분산을 이용하여 일별 로그우도값($LH = -\ln\sigma_i^2 - \frac{r_i^2}{\sigma^2}$)을 계산하여 F열에 채우고, 이를 모두 더하여 전체 우도값을 산출한다.
5. 전체 우도값을 최대화시키는 람다를 solver 기능을 이용하여 추정한다.
6. 적정 파라미터를 추정하였으면, 8/3일의 분산값을 통해 연환산 변동성을 산출한다. ($\sigma_{annual} = \sqrt{252}\sigma_{8.3}$)

Answer 6



2018년 ~ 2020년 8월 4일까지 코스피 지수(회식 면) 및 GARCH(적색), EWMA(청색)을 도식화하였습니다.

먼저, **전체적인 추세를 볼 때 GARCH(1,1) 모형과 EWMA 모형이 추정한 일 변동성은 크게 다르지 않습니다.** 근본적으로 GARCH(1,1) 모형에서 장기변동성을 제외한 모형이 EWMA이며, GARCH(1,1)의 세 파라미터 중 장기변동성의 가중치가 가장 낮기 때문입니다.

두 번째로는, **코스피 지수(회색 면)의 하락폭이 클 때 모델의 추정변동성이 급등하는 경향**이 있습니다. 이러한 경향은 2020년 3월경 코로나19 펜데믹으로 인해 주가가 매우 큰 폭으로 급락하였을 때 잘 나타납니다. 주가는 상승할 때는 완만히 상승하다가 하락할 때는 급락하는 경향이 있는데, 두 모델이 이러한 특성을 잘 반영하여 하락시 변동성이 급등하는 현상을 잘 표현하는 것으로 보입니다.

모델의 식을 생각해보면, 우리가 사용한 모델에서 GARCH(1,1)은 약 11.5%(α), EWMA는 약 6.6%($1-\lambda$)만큼 당일 수익률의 제곱을 추정변동성에 반영하고 있습니다. 따라서, **주가가 오늘 급등락하였다면 해당 비율만큼 추정변동성에 영향을 주게되고, 그 급등락이 클수록 추정변동성이 급등**하게 되는 것 입니다.

한편, **두 모델의 차이는 이러한 변동성 급등 및 평균회귀(mean reverting) 과정에서 잘 나타납니다.** 먼저, 급등시에는 **당일 주가수익률의 반영비율이 큰 GARCH(1,1) 모형의 추정변동성이 더 급등**하는 패턴을 관측할 수 있습니다.(적색〉청색)

다음으로, 변동성이 급등하고나서 시간이 지남에 따라 반영비율이 희석되면서 변동성이 평균 수준으로 회귀하게 되는데, 이때에도 당일 주가수익률의 반영비율이 큰, **직전 추정변동성의 반영비율이 상대적으로 낮은 GARCH(1,1) 모형의 회귀속도가 빠르게 됩니다.** 이러한 패턴은 20년 3월 변동성 급등 이후 20년 6월경까지 변동성이 하락할 때 잘 관측됩니다.

추가적으로, GARCH(1,1) 모형은 그 반영비율이 낮기는 하지만 장기변동성을 포함하여 변동성을 추정하기 때문에, 역시 EWMA보다 평균회귀가 빠른 이점을 가지게 됩니다. 따라서, **일시적인 주가 급등락으로 변동성이 급등하는 경우에는 GARCH(1,1) 모형이 정상수준으로 잘 회귀한다는 점에서 EWMA보다 적합한 모형인 것으로 보입니다.**

Question 7-8

Volatility

7) A company uses an EWMA model for forecasting volatility. It decides to change the parameter λ from 0.85 to 0.9. Explain the likely impact on the forecasts.

8) Suppose that GARCH(1,1) parameters have been estimated as ω = 0.00000135, α = 0.0833, and β = 0.9101. The current daily volatility is estimated to be 1%. Estimate the daily volatility in 30 days.

Answer 7

EWMA의 λ가 증가한다는 의미는, 변동성을 추정할 때 최신 데이터의 반영비율을 늘린다는 의미입니다.

EWMA는 $\sigma_i^2 = \sum_k \lambda r_{i-k}^2$의 방식으로 변동성을 추정하는데, 여기에서 람다값이 증가하면 가장 최근에 형성된 수익률이 보다 많이 반영되게 됩니다.

따라서, 최근 주가흐름이 추정변동성에 미치는 영향이 커지게 되므로, 급등락장이 이어졌다면 추정 변동성이 보다 빠르게 급등할 것이고, 보합장이 이어졌다면 추정변동성이 빠르게 감소할 것으로 보입니다.

Answer 8

주어진 파라미터와 현재 일 변동성이 1%임을 활용하여 30일 변동성을 산출하겠습니다.

이 때 이용하는 수식은 $E[\sigma_{n+30}^2|I_{n-1}] = V_L + (\alpha+\beta)^{30}(\sigma_n^2 - V_L)$ 입니다.

먼저, 장기변동성 $V_L = \frac{\omega}{1-\alpha-\beta} = 0.0002045$입니다.

이에 따라 **현재까지의 정보를 이용하여 30일 뒤의 일변동성을 추정하면 약 1.09%가** 됩니다.

$$E[\sigma_{n+30}^2|I_{n-1}] = 0.0002045 + 0.9934^{30}(0.01^2 - 0.0002045) = 0.0001188$$

$$E[\sigma_{n+30}|I_{n-1}] = \sqrt{E[\sigma_{n+30}^2|I_{n-1}]} = \sqrt{0.0001188} = 1.09\%$$

Question 9

Principal Component Analysis

9) Recall the PCA result of swap rates (Table 1 and 2) and the portfolio's exposures to interest rate moves (Table 3) in example 4 of week 4 lecture note.

Table 1: Factor Loadings for Swap Data (bps)

	PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC8
1-year	0.216	−0.501	0.627	−0.487	0.122	0.237	0.011	−0.034
2-year	0.331	−0.429	0.129	0.354	−0.212	−0.674	−0.100	0.236
3-year	0.372	−0.267	−0.157	0.414	−0.096	0.311	0.413	−0.564
4-year	0.392	−0.110	−0.256	0.174	−0.019	0.551	−0.416	0.512
5-year	0.404	0.019	−0.355	−0.269	0.595	−0.278	−0.316	−0.327
7-year	0.394	0.194	−0.195	−0.336	0.007	−0.100	0.685	0.422
10-year	0.376	0.371	0.068	−0.305	−0.684	−0.039	−0.278	−0.279
30-year	0.305	0.554	0.575	0.398	0.331	0.022	0.007	0.032

Table 2: Standard Deviation of Factor Scores (bps)

PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC8
17.55	4.77	2.08	1.29	0.91	0.73	0.56	0.53

Table 3: Change in portfolio value for a 1 bps rate move ($M)

3-Year Rate	4-Year Rate	5-Year Rate	7-Year Rate	10-Year Rate
+10	+4	−8	−7	+2

Since the first two factors together account for 97.7% of the variance in the data, let's use the first two factors to model the rate moves.

Using the data in Table 1 and Table 2, the portfolio's delta exposure to the first PC is $10 \times 0.372 + 4 \times 0.392 - 8 \times 0.404 - 7 \times 0.394 + 2 \times 0.376 = 0.05$ million dollars per unit of the factor. Likewise, the portfolio's delta exposure to the second PC is $10 \times (-0.267) + 4 \times (-0.110) - 8 \times 0.019 - 7 \times 0.194 + 2 \times 0.371 = -3.88$ million dollars per unit of the factor. In other words,

$$\Delta P = 0.05 \times PC_1 - 3.88 \times PC_2$$

Suppose we are interested in a "worst case" outcome for tomorrow where the loss has a probability of only 1% of being exceeded. What is the loss? Assume that the two PCs are independent from each other and follow a normal distribution with mean 0.

(Hint: PC1's standard deviation is the square root of the first eigenvalue and PC2's standard deviation is the square root of the second eigenvalue. See Table 2.)
(Hint: The sum of two normally distributed random variables is also normal.)

Answer

먼저, 목표는 **1% 이하의 확률로 발생할 수 있는 포트폴리오의 손실액**을 찾는 것입니다. 이는 포트폴리오의 확률분포를 통해 알 수 있으며, **CDF에서 하위 1% 임계값**을 통해 계산할 수 있습니다. 이 의미는, 향후 시장상황에 따라 약 1%의 확률로 해당 임계값보다 큰 손실이 발생할 수 있다는 뜻이며, 이를 **VaR(Value at Risk)**라고 부릅니다.

이제 문제에서 주어진 정보를 이용하여 이 임계값을 계산해보겠습니다.

문제에서 PC_1, PC_2는 각각 평균이 0인 정규분포를 따르므로, 각각의 표준편차를 σ_1, σ_2라고 하겠습니다.

이에 따라 포트폴리오의 가격변동 $\Delta P = 0.05PC_1 - 3.88PC_2$는 두 정규분포의 선형결합이므로 joint normal distribution이 되고, 각 PC는 평균이 0, 독립임을 이용하여 ΔP의 평균과 표준편차 σ는 아래와 같이 계산할 수 있습니다.

1. $E[\Delta P] = 0.05E[PC_1] - 3.88E[PC_2] = 0$

2. $\sigma^2 = E[\Delta P^2] - (E[\Delta P])^2 = 0.05^2\sigma_1^2 + 3.88^2\sigma_2^2$

> 정규분포이고 독립인 두 확률변수 X, Y에 대해 $E[XY] = E[X]E[Y]$가 성립합니다.
> 따라서, $E[\Delta P^2] = E[0.05^2 PC_1^2 - 2 \times 0.05 \times 3.88 PC_1 PC_2 + 3.88^2 PC_2^2]$
> $= 0.05^2 E[PC_1^2] + 3.88^2 E[PC_2^2] = 0.05^2\sigma_1^2 + 3.88^2\sigma_2^2$

즉, $\Delta P \sim N(0,\ 0.05^2\sigma_1^2 + 3.88^2\sigma_2^2)$이므로 각 PC의 표준편차를 알면 1% 임계값을 알 수 있습니다. *Table2*를 이용해 이를 계산하면,

$\sigma^2 = 0.05^2\sigma_1^2 + 3.88^2\sigma_2^2 = 144.39, \ \ \therefore \sigma = 12.02$

$z_{0.01} = -2.33$임이 잘 알려져 있으므로, 1% 임계값은 $-2.33\sigma \approx -28$입니다.

따라서, **약 1% 확률로 발생할 수 있는 포트폴리오의 예상손실액은 최소 28 million $ 입니다.**

Part VII

미시경제학('24 가을)

미시경제학 Ch1

Demand, Supply, Elasticity

Law of demand

가격이 상승하면 수요는 하락하고, 가격이 하락하면 수요는 증가한다

when Other things equal(ceteris paribus)

따라서, x축이 수요량이고 y축이 가격일 때 수요곡선은 우하향함

Other things?

이는 수요곡선 자체를 변화시키는 모든 변수 일체를 의미함.

1. Income : normal goods vs. inferior goods
2. Number of buyers
3. Substitutes vs. Complements
4. Tastes

1. Income

일반적인 재화는 소득이 증가하면 수요가 증가함.

즉, 수요곡선을 오른쪽으로 평행이동시킴 ***Normal Goods***

그러나, 대중교통이나 감자같은 재화는 소득이 증가하면 오히려 수요가 감소할 수 있음.

이 경우는 수요곡선을 왼쪽으로 평행이동시킴 ***Inferior Goods***

이는 재화에 따라 고정되어있지 않으며,

때로는 소득수준이 증가함에 따라 수요가 증가하는 Normal goods였다가 소득수준이 더 크게 증가하면 수요가 감소하는 Inferior goods가 되기도 함. (e.g. Hamburger)

2. Substitutes vs. Complements

대체제(e.g. 맥도날드, 버거킹)는 대체관계에 있는 재화들로, 대체제의 수요가 감소하면 재화의 수요가 증가함.

즉, 대체제의 가격상승은 재화의 수요곡선을 우측 평행이동시킴

보완재(자동차, 기름)은 상호보완관계에 있는 재화로, 보완재의 수요가 증가하면 재화의 수요가 감소함. 보완재 가격상승은 수요곡선 좌측 평행이동

이외의 수요곡선을 이동시키는건 매우 많을 수 있음

중요한건, 특정재화의 수요에 영향을 미치는 방법은

1. 다른 요소를 건드려서 수요곡선 자체를 이동시키거나
2. 재화의 가격을 건드려서 수요곡선 내에서 이동시키는거임

Law of Supply

공급의 법칙

똑같음. 가격이 올라가면 공급이 늘어나고 감소하면 공급도 감소함

따라서 일반적으로 우상향하는 공급곡선이 나타남.

언제? 다른 모든 것들이 동일할 때

공급곡선의 이동

1. 생산가격 Input price
2. Technology
3. Number of sellers

1. Input price

생산단가가 감소하면 공급은 증가함. 공급곡선 우측 이동

생산단가 증가 - 공급량 감소 - 곡선 좌측 이동

2. Technology

기술수준이 상승하면 생산단가가 감소함. 공급곡선 우측이동.

Microanalysis of financial economics Assignment1

20249132 Kim Hyeonghwan (김형환)

Sample Question

sample question:

- Suppose that due to more stringent environmental regulation it becomes more expensive for steel production firms to operate. Also, recent technological advances in plastics has reduced the demand for steel products.

Q1) Use Supply and Demand analysis to predict how these shocks will affect equilibrium price and quantity of steel.
Q2) Can we say with certainty that the market price for steel will fall? Why?

Answer

Becomes more expensive for steel production

-> Increase input prices for steel production

-> Supply curve shifts to the left.

Reduced the demand for steel products

-> Decrease number of buyers for steel production

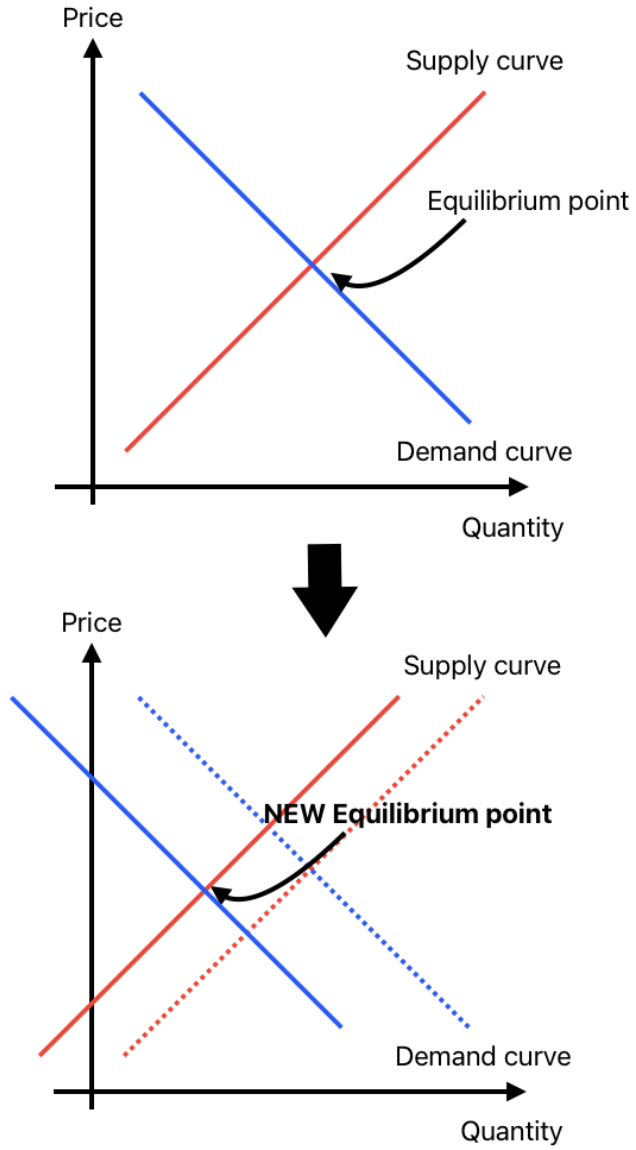
-> Demand curve shifts to the left.

Both curves will shift to the left side,

so NEW equilibrium point also shift to the left side.

It is clear that NEW equilibrium quantity will increase, but price is not.

Whether equilibrium price increase or not, it depends on how each curves shifts and their elasticity.



1. More expansive for steel production
-> Left shift of the supply curve

2. Reduced the demand for steel production
-> Left shift of the demand curve

Therefore, both curves shift to the left side,

Also equilibrium point shifts to the left.

It is obvious that **quantities of equilibrium is decr**

But **price of equilibrium can be decrease or incre**

It depends on how much each curve shifts.

Figure 4.1: Sample question

Assignments 1

Assignments: Due 9/10 (Tuesday)

- **Examine the behavior of supply and demand for Wheat**

Demand: $Q_D = 3550 - 266P$ *Supply*: $Q_S = 1800 + 240P$

the market-clearing price of wheat:

$Q_S = Q_D$

$1800 + 240P = 3550 - 66P$

$506P = 1750$

P = \$3.46 per bushel

Substituting into the supply curve equation, we get

$Q = 1800 + (240)(3.46) = 2630$ million bushels

==========

•What is the price E of Demand?

•What is the price E of Supply?

Answer

We can say that price elasticity, $E_p = \frac{\frac{\Delta Q}{Q}}{\frac{\Delta P}{P}} = \frac{P}{Q}\frac{\Delta Q}{\Delta P}$

Since $P_0 = 3.46,\ Q_0^D = Q_0^S = 2630$ and $\frac{\Delta Q^D}{\Delta P^D} = -266,\ \frac{\Delta Q^S}{\Delta P^S} = 240,$

Price elasticity of Demand is $\frac{3.46}{2630} \times -266 \approx -0.35.$

Price elasticity of Supply is $\frac{3.46}{2630} \times 240 \approx 0.32$

Microanalysis of financial economics Assignment2

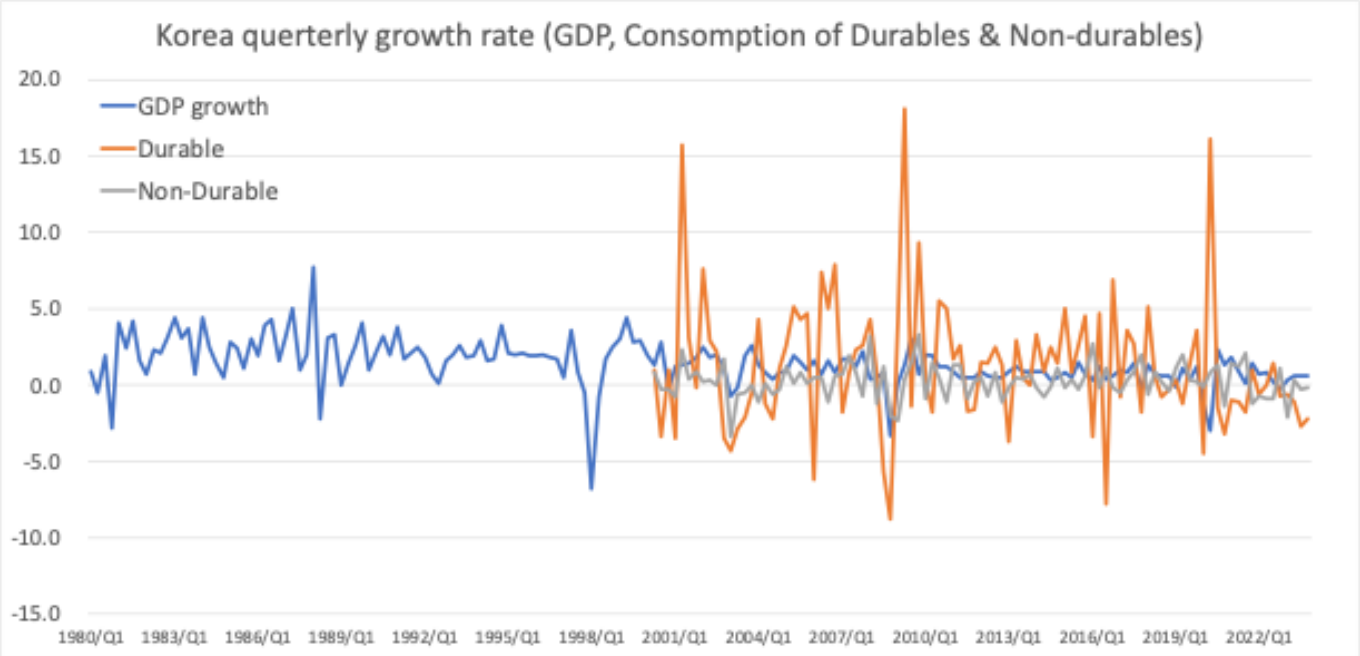
20249132 Kim Hyeonghwan (김형환)

Question 1

Home work:
Please draw the same graph as shown above (GDP/ Durables/ Non-Durables) in the case of KOREA (or your Home country). Time series would be during 1980~2023 using quarterly data.

Due date:
You need to send the graph through KLMS to the Instructor.

Answer1



Question 2

sample question

Cups of coffee and donuts are complements. Both have inelastic demand. A hurricane destroys half the coffee bean crop. Use appropriately labeled diagrams to answer the following questions:

1) What happens to the price of coffee beans?
2) What happens to the price of cups of coffee? What happens to the total expenditures (revenues) on cups of coffee?
3) What happens to the price of donuts? What happens to the total expenditures (revenues) on donuts?

Answer2

1. Increase

Because of hurricane, input price of coffee bean will increase, supply curve of coffee bean shift to the left.

So, price of coffee beans will increase.

2. Both increase

Since price of coffee beans increases, input price of cups of coffee will increase, supply curve of coffee shift to the left.

so, price of cups of coffee will increase and quantity will decrease.

By the question, cups of coffee have inelastic demand.

so, $P \times Q < (P + \Delta P)(Q - \Delta Q)$. Total revenue increases.

3. Both decrease

Since cups of coffe and donuts are complements, increase of price of cups of coffee causes decrease of demand of donuts.

So, demand curve of donuts shift to the left, price and quantity will decrease.

Therefore, Total revenue of donuts decrease.

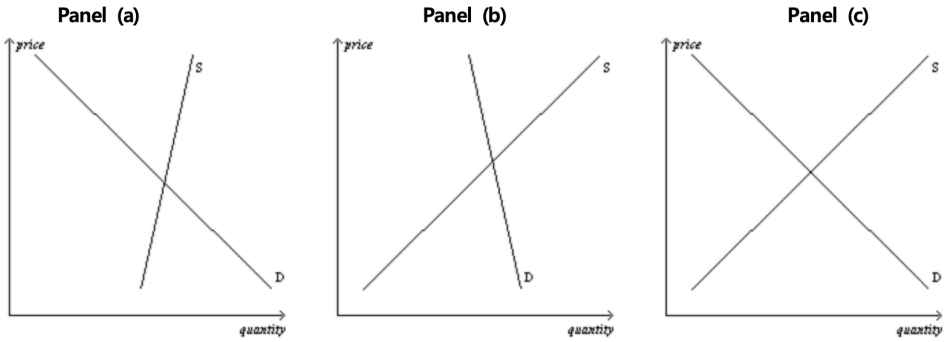
Question 3

EX) In which market will the majority of the tax burden fall on buyers?

1 . the market shown in panel a)

2. the market shown in panel b)

3.. the market shown in panel c)



Answer 3

1. sellers bear more burden of the tax than buyers.
2. buyers bear more burden of the tax than sellers.
3. both participants bear same burden of the tax.

Question 4

Ex) Market is described by the following D and S curves:

$$Q^s = 2P$$
$$Q^D = 300 - P$$

1) Solve for the equilibrium price and quantity
2) If the government imposes a price ceiling of $90, dose a shortage or surplus develop? what are the price, quantity supplied, quantity demanded, and size of the shortage or surplus?
3) If the government imposes a price floor of $90, dose a shortage or surplus develop? what are the price, quantity supplied, quantity demanded, and size of the shortage or surplus?
4) Instead of a price control, the government levies a tax on producers of $30. As a result, the new supply curve is: $Q^s = 2(P - 30)$. Does a shortage or surplus (or neither) develop? What are the price, quantity supplied, quantity demanded, and the size of the shortages or surplus?

Answer 4

1. P=100, Q=200

$Q^S = Q^D \;\Rightarrow\; 2P = 300 - P \;\therefore P = 100, Q = 200$

2. Yes. $P = 90\ Q^S = 180,\ Q^D = 210,\ Shortage = 30$

3. No. $P = 100,\ Q^S = Q^D = 200$

4. No. $P^* = 120,\ Q^{S^*} = Q^{D^*} = 180$

Question 5

(sample question)

- An Assemblyman wants to raise tax revenue and make workers better off. A straff member proposes raising the payroll tax paid by firms and using part of the extra revenue to reduce the payroll tax paid by workers. Would this accomplish the Assemblyman's goal? Explain.

Answer 5

It depends on elasticity of labor market and ratio of using extra revenues.

If all of extra revenues can be used for workers, it will make workers better. (when demand of labor is not perfect elatic.)

But if just part of extra can be used for workers and demand is more elastic than supply in labor market, it will be different.

Workers bear more burden than firm when tax paid by firms raises,

so it can make workers poor.

Question 6

(sample questions): The 2011 payroll tax cut

Prior to 2011, the Social Security payroll tax was 6.2% taken from workers' pay and 6.2% paid by employers (total 12.4%). The Tax Relief Act (2010) reduced the worker's portion from 6.2% to 4.2% in 2011, but left the employer's portion at 6.2%.

Q1) Should this change have increased the typical worker's **take-home pay** by exactly 2%, more than 2%, or less than 2%? Do any elasticities affect your answer? Explain.

Q2) Who gets the bigger share of this tax cut, workers or employers? How do elasticities determine the answer?

1. Less than 2%

2% tax-cut of workers makes supply curve shifting to the right in labor market.

So, price of labor(wage) will decrease and its magnitude depends on elasticity of demand curve.

It will be less than 2% when demand curve is not zero elasticity.

If demand is zero elastic, it is exactly 2%.

Since tax-cut makes workers take-home pay rises 2%,

Total change of workers' take-home pay will be less than 2%.

2. More elastic take less benefit.

By explain (1), it depends on elasticity in labor market.

More elastic bear less burden of tax.

It means more elastic take less benefit of tax-cut.

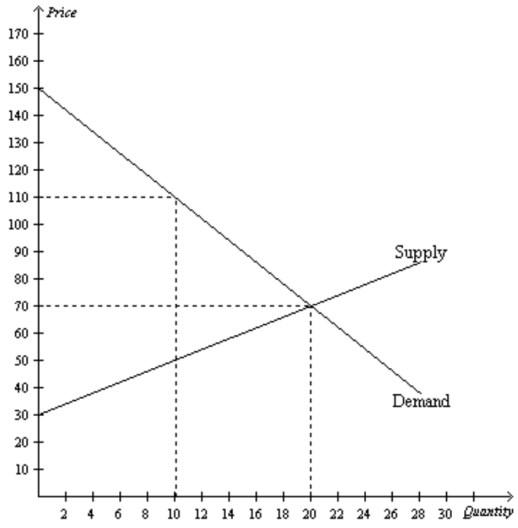
Microanalysis of financial economics Assignment3

20249132 Kim Hyeonghwan (김형환)

Question

Q) If the government imposes a price floor of $110 in this market, then consumer surplus will decrease by

a. $200
b. $400
c. $600
d. $800



Answer : (a) $200

At equilibrium(p=70), consumer surplus is 800($20 \times 80 \times 0.5$) and supplier surplus is 400.($20 \times 40 \times 0.5$) so total surplus is 1200.

If there exists a price floor of $110, quantity decrease to 10 and price is $110. so consumer surplus will be 200 ($10 \times 40 \times 0.5$) and suplier surplus will be 700($10 \times 60 + 10 \times 20 \times 0.5$).

Now, total surplus decrease to 900 and deadweight loss occur 300.

Part VIII

글로벌 지속가능회계('24 가을)

글로벌 지속가능회계 1주차