

KAIST MFE, 2024 Fall

Kim Hyeonghwan

2024-09-02

Table of contents

Welcome!	9
I 머신러닝('24 가을)	10
머신러닝 Ch1	11
하이퍼파라미터	11
모델의 평가와 검증	11
일반화 오차	12
편향(Bias)	12
분산(Variance)	12
관계	12
데이터의 분할 방법	13
Hold-out 방식	13
K-fold 교차검증(Cross-validation) 방식을 이용한 검증	13
머신러닝 Ch2-3	15
규제가 있는 선형회귀	15
릿지회귀모형	15
라쏘회귀모형	16
비교	16
1 머신러닝 실습	17
1.1 Ch2. Linear regression	17
1.2 Ch3. Restricted linear regression	21
1.3 Ch4. Logistic Regression	26
인공지능 및 기계학습 과제1	29
Question 1	29
Answer	29
Question 2	31
Question 3	33
Answer : 3번	34

II 딥러닝('24 가을)	35
딥러닝 Ch1	36
머신러닝 알고리즘의 구분	36
지도학습	36
비지도학습	36
강화학습	36
지도학습 알고리즘의 절차	37
경사하강법 (Gradient Descent Method)	37
모델 평가 및 검증	37
자료의 구분	38
모델의 평가를 위한 지표	38
딥러닝 Ch2	40
퍼셉트론 (Perceptron)	40
로젠틀랜의 퍼셉트론 (Simple perceptron)	40
선형 퍼셉트론 (Linear perceptron)	40
시그모이드 퍼셉트론 (Sigmoid perceptron)	41
다층 퍼셉트론 (Multi-layer perceptron)	41
다층 퍼셉트론의 구조	41
다층 퍼셉트론의 학습	42
2 딥러닝 Ch1 실습	44
2.1 1. 텐서 데이터 만들기	44
2.2 2. 텐서 데이터 타입, 크기	45
2.3 3. 수학 연산의 적용	47
2.4 4. 텐서 데이터의 분할 및 통합	49
2.5 5. tf.data를 활용한 데이터 전처리	51
2.6 6. 선형회귀분석 (low-lever ver.)	58
2.7 7. 선형회귀분석 (tf.keras ver.)	62
2.8 Ch2 Neural Network 실습	76
III 시뮬레이션 방법론('24 가을)	83
시뮬레이션방법론 Ch1	84
블랙솔즈공식 예시	84
Volume과 적분	84
MCS 기초	85
확률기대값 및 원주율 계산 예시	85
표본표준편차 계산 : numpy는 n으로 나누고, pandas는 n-1로 나누는 것이 기본	86
표준오차 계산 및 95% 신뢰구간 계산	86

경로의존성 (Path-dependent)	86
시뮬레이션 예시	87
MCS 추정치 개선 방향	87
분산감소와 계산시간	87
Asian Option 평가 해볼 것	88
시뮬레이션방법론 Ch3-4	89
Brownian bridge	89
.	89
시뮬레이션방법론 실습	90
2.9 Ch2. 난수 생성 방법	90
2.9.1 Acceptance-rejection method	90
2.9.2 Box-muller method	95
2.9.3 Marsaglia's polar method	98
2.9.4 Correlated random	100
2.10 Ch3. 샘플 경로 생성 방법	103
2.10.1 Variance reduction	103
시뮬레이션 과제1 (베리어옵션)	106
Question	106
Answer 1	107
파라미터 및 알고리즘	107
Python 구현	107
Analytic Solution과 비교	109
Answer 2	111
In-Out parity 정의	111
증명	111
MCS에서의 활용	112
Answer 3	112
시뮬레이션 과제1 수정 (베리어옵션)	115
Question	115
Answer 1	116
파라미터 및 알고리즘	116
Python 구현	117
Analytic Solution과 비교	118
Answer 2	119
In-Out parity 정의	119
증명	120
MCS에서의 활용	120

Answer 3	121
IV 이자율파생상품('24 가을)	125
이자율파생상품 과제1	126
Question 1	126
Answer 1 : 0.631%	126
Question 2	127
Answer 2 : 최소 5.01%	127
Question 3	127
Answer 3	128
Question 4	128
Answer 4 : -0.1980	129
Question 5	129
Answer 5 : 0.2974	129
이자율파생상품 과제2	130
Question 1	130
Answer	130
Question 2-3	131
Answer	131
Question 4	131
Answer	132
Question 5	132
Answer	133
V 수치해석학('24 가을)	134
수치해석학 Ch1	135
강의 개요 : 금융수치해석의 필요성	135
파생상품 평가	135
최적화 방법론	135
컴퓨터 연산에 대한 이해	136
Rounding error 관련	136
계산오차	137
유한차분을 이용한 도함수의 근사	137
총오차 및 최적의 h 산출	138
유한차분을 이용한 도함수 근사 예시	138
수치적 불안정성과 악조건	142
행렬의 조건수	142

알고리즘의 계산 복잡도	142
알고리즘 복잡도	142
수치해석학 Ch2	143
Cholesky factorization	143
QR 분해	144
수치해석기법 실습	145
2.11 Ch3. Linear system	145
2.11.1 Equations	145
2.11.2 Equations - time	149
2.11.3 Linear system iterative	151
2.12 Ch4. Finite Difference Method	153
2.12.1 1 factor FDM	153
VI 금융시장 리스크관리('24 가을)	156
금융시장 리스크관리	157
Lecture2 : How Traders Manage Their Risks?	157
Delta hedging	157
기타 그릭스	158
Taylor Series Expansion	158
Hedging in practice	159
Lecture3 : Volatility	159
Weighting Schemes	159
최대우도법, Maximum Likelihood Method	159
Characteristics of Volatility	160
How Good is the Model?	160
Lecture6 : Value at Risk and Expected Shortfall	160
Properties of Coherent Risk Measures	160
VaR vs. ES	160
금융시장 리스크관리 과제1	161
Question 1-3	161
Answer	161
Question 4-6	162
Answer 4-5	162
Answer 6	163
Question 7-8	164
Answer 7	164
Answer 8	164

Question 9	164
Answer	165
금융시장 리스크관리 과제2	167
Question 1-4 : Basic historical approach	167
Answer	167
Question 5 : Comparison 500 vs. 1000	168
Answer	168
Question 5-10 : Back testing	169
Answer	170
Question 11 : VaR and ES	171
Answer	171
Question 12 : EVT	172
Answer	172
VII 미시경제학('24 가을)	174
미시경제학 Ch1	175
Law of demand	175
Other things?	175
Law of Supply	176
공급곡선의 이동	176
Microanalysis of financial economics Assignment1	178
Sample Question	178
Answer	178
Assignments 1	180
Answer	180
Microanalysis of financial economics Assignment2	181
Question 1	181
Answer1	181
Question 2	182
Answer2	182
Question 3	183
Answer 3	183
Question 4	183
Answer 4	184
Question 5	184
Answer 5	184
Question 6	185

Microanalysis of financial economics Assignment3	186
Question	186
Answer : (a) \$200	186
Microanalysis of financial economics Assignment4	187
Question	187
Answer	188
Microanalysis of financial economics Assignment5	189
Question 1	189
Answer	189
Question 2	190
Answer	190
VIII글로벌 지속가능회계('24 가을)	191
글로벌 지속가능회계 1주차	192

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 파라미터(α, β 등)가 아닌,

학습알고리즘의 파라미터(학습률 등)

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

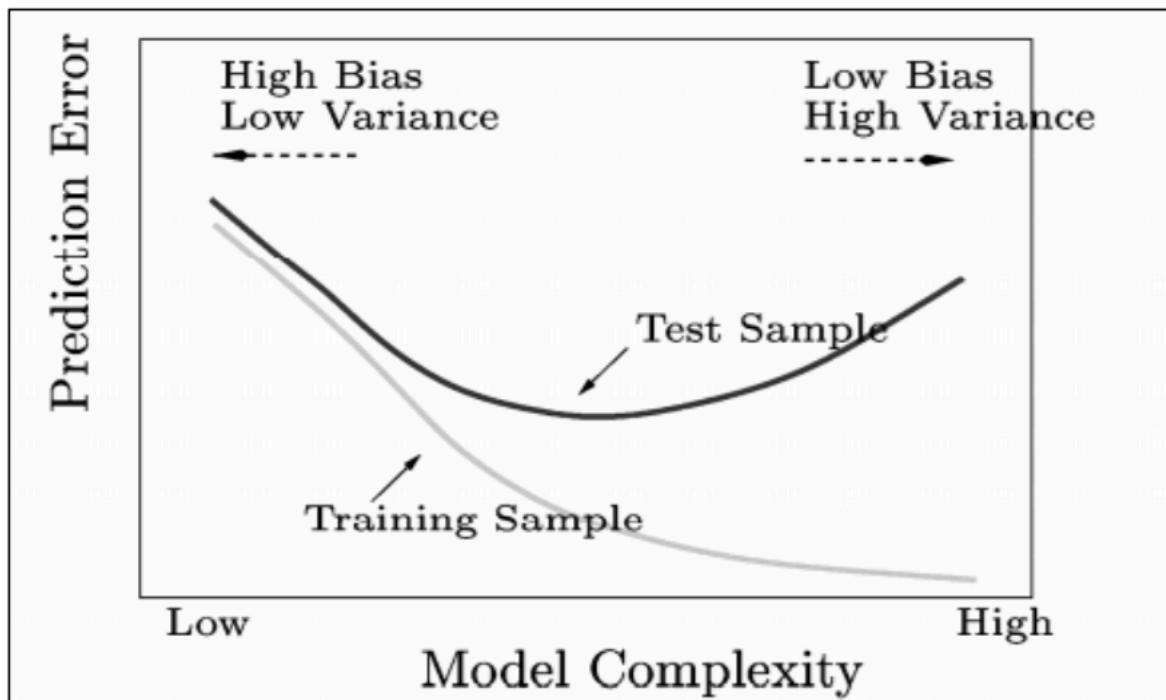
여기서, β_n 은 파라미터이며 주어진 데이터를 학습하여 파라미터를 산출하는 것임.

근데 만약에 모델 성능 향상을 위해 각 β 의 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 오차라고 함

(Generalization error, test error)

$$= \text{bias}^2 + \text{variance} + \text{noise}$$

편향(Bias)

모집단에서 크기 m 의 (x,y) 순서쌍을 샘플링할 때,

해당 샘플링을 n 번 반복해서 모델링을 한다고 하면,

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

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

분산(Variance)

한편, f_1, \dots, f_n 의 추정모델간의 편차의 제곱합이 분산이 됩니다.

관계

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

추정모델간의 오차는 작아지므로 분산이 작습니다.

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

추정모델간의 오차는 큼 것으로 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.

머신러닝 Ch2-3

선형회귀모형

$$\text{비용함수} : J(\theta) = \frac{1}{2m} \sum_{i=1}^m (\theta^T x^{(i)} - y^{(i)})^2$$

$$\theta^* = \theta - \alpha \frac{\partial}{\partial \theta} J(\theta)$$

$$\frac{\partial}{\partial \theta} J(\theta) = \frac{1}{m} X^T (X\theta - y)$$

배치 / 확률적 / 미니배치

정규방정식에서 $\hat{\theta} = (X^T X)^{-1} X^T y$ 의 closed form 존재.

역행렬 계산에 많은 시간이 소요되는 경우에는 사용이 곤란

규제가 있는 선형회귀

릿지회귀모형

L2 규제를 사용하여 파라미터의 범위를 제약

$$\text{비용함수} : J(\theta) = \frac{1}{2m} \left(\sum_{i=1}^m (\theta^T x^{(i)} - y^{(i)})^2 + \lambda \sum_{j=1}^n \theta_j^2 \right)$$

$$\text{파라미터 추정치는}, \hat{\theta}^R = \arg \min_{\theta} J(\theta)$$

$$\text{행렬식} : \hat{\theta}^R = \arg \min_{\theta} \left\{ \frac{1}{2m} (X\theta - y)^T (X\theta - y) + \lambda \theta_1^T \theta_1 \right\}$$

$$\text{Alter} : \hat{\theta}^R = \arg \min_{\theta} \left\{ \frac{1}{2m} (X\theta - y)^T (X\theta - y) \right\} \text{ subject to } \sum_{j=1}^n \theta_j^2 \leq t$$

즉, 파라미터 제곱합에 일정 상한을 정해놓는 방식으로 보면 됨.

람다가 커지면? 파라미터 제약이 커지면서 편향이 증가함

파라미터 추정치 계산

$$\text{정규방정식} : \hat{\theta}^R = (X^T X + \lambda I)^{-1} X^T y$$

일반 회귀모형에서는 $X^T X$ 가 singular이면 해가 없었으나, 럿지제약 하에서는 이를 해결할 수 있음.

또한, 다중공선성(multicollinearity)도 해결할 수 있다는 것이 알려져 있음.

$$\text{경사하강법} : \theta_{new} = \theta_{old} - \alpha J(\theta) = (1 - \alpha \frac{\lambda}{m})\theta_{old} - \alpha J^U(\theta)$$

마지막 식은, 일반선형회귀의 그레디언트를 이용해서 표현한 것이며, 직전 θ_{old} 의 영향을 상수배로 줄여주는 것으로 해석할 수 있음.

라쏘회귀모형

L1 규제를 이용. Least Absolute Shrinkage & Selection Operator

모델로 널리 이용되지는 않지만, feature가 너무 많은 경우 적절한 변수만 선택할 때 많이 이용됨.

$$\text{비용함수} : J(\theta) = \frac{1}{2m} \left(\sum_{i=1}^m (\theta^T x^{(i)} - y^{(i)})^2 + \lambda \sum_{j=1}^n |\theta_j| \right)$$

파라미터 추정치는, $\hat{\theta}^L = \arg \min_{\theta} J(\theta)$

$$\text{행렬식} : \hat{\theta}^L = \arg \min_{\theta} \left\{ \frac{1}{2m} (X\theta - y)^T (X\theta - y) + \lambda \mathbf{1}^T |\theta_1| \right\}$$

$$\text{Alter} : \hat{\theta}^L = \arg \min_{\theta} \left\{ \frac{1}{2m} (X\theta - y)^T (X\theta - y) \right\} \text{ subject to } \sum_{j=1}^n |\theta_j| \leq t$$

함수꼴에서 보듯, 미분불가능하므로 closed form solution이 존재하지 않음.

다만, θ 의 부호에 따라 절대값의 미분값은 -1 또는 1이므로 침값의 미분값이 0이라고 가정하면 다음과 같이 표현가능함.

$$\theta_{new} \leftarrow \theta_{old} - \alpha J(\theta_{old}) = \theta_{old} - \alpha (J^U(\theta_{old}) - \lambda_{(\frac{\lambda}{2m})} \text{sign}(\theta_{old}))$$

$$\text{for } \text{sign}(\theta) = \frac{|\theta|}{\theta} \text{ or } 0 \text{ where } \theta = 0$$

즉, 원점을 향해 일정부분() 계속 보정이 들어가는 형태.

비교

릿지회귀모형 vs. 라쏘회귀모형

해석력 : 라쏘

예측력 : 기본적으로, 규제가 생기면 예측력이 증가하긴 함. 어떤 모형이 예측력이 뛰어난지는 데이터에 따라 달려있음

만약 feature의 수가 샘플의 수보다 작으면, 라쏘를 쓰면 안됨

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 )
kf_id = kf.split(X_data)

kf_mse = []
for train_i, test_i in kf_id:
    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
```

[3070.539738843073,

3256.5076254441997,

2917.145487052703,

2756.124137161446,

2833.1249687769155]

```
# 잘 안쓰고 아래꺼 K-fold api를 많이 씀
kf_rmse = np.sqrt(kf_mse)
np.mean(kf_rmse)
```

54.44295671106831

```
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 ) # 지표는 값이 큰 것을 선택하도록 내부적으로 (-) 변환
rmse_scores
```

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

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

54.69179097275793

```
# 변수 표준화
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")

[153.88378151]
[-0.4 -17.7  25.2  19. -20.4  12.6   1.6   4.9  26.8  -0.5]

```

1.2 Ch3. Restricted linear regression

```

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
from sklearn.datasets import load_diabetes
diabetes = load_diabetes()
y_target = diabetes['target']
X_data = diabetes['data']

# 데이터 분리
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X_data, y_target, test_size=0.4, random_state=123 )

# 규제 선형회귀에서는 반드시 표준화를 해야함
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit( X_train )
X_train = scaler.transform( X_train )
X_test = scaler.transform( X_test )

```

전체 데이터로 표준화하고 데이터를 분리하는 방법이 있고,
 데이터를 분리하고 훈련데이터로 표준화하는 방법이 있음. (교수님 선호)
 표준화하고 데이터를 분리하면 훈련 및 평가 각각에 대해서는 정확히 표준화는 안됨.
 데이터를 분리하고 표준화하는 경우, 훈련데이터는 정확히 평균 0 표준편차 1이 되지만, 평가데이터는 이를 만족하지
 않게 됨.
 두 방식에 정답은 없으나, 훈련과 평가데이터를 각각 표준화하는 방법은 피해야 함.

```
# Hyperparameter (람다) 튜닝
# Cross validation, 5-fold
from sklearn.linear_model import Ridge
from sklearn.model_selection import cross_val_score
alphas = [0, 0.01, 0.05, 0.1, 0.5, 1, 5]
for al in alphas:
    ridge = Ridge(alpha=al)
    neg_mse_scores = cross_val_score(ridge, X_train, y_train,
                                      scoring='neg_mean_squared_error', cv=5)
    avg_rmse = np.mean(np.sqrt(-1*neg_mse_scores))
    print('alpha={} -> RMSE={}'.format(al, np.around(avg_rmse, decimals=3)))
```

```
alpha=0 -> RMSE=55.733
alpha=0.01 -> RMSE=55.733
alpha=0.05 -> RMSE=55.732
alpha=0.1 -> RMSE=55.731
alpha=0.5 -> RMSE=55.733
alpha=1 -> RMSE=55.745
alpha=5 -> RMSE=55.812
```

```
ridge = Ridge(alpha=0.1)
ridge.fit( X_train, y_train )
print(ridge.coef_)
print(ridge.intercept_)
```

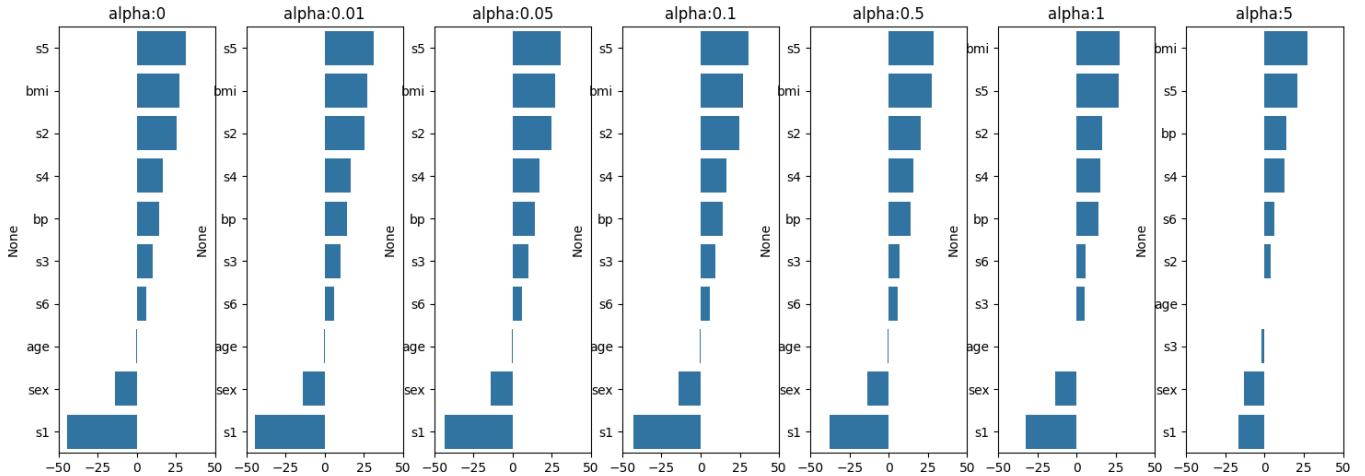
```
[ -0.49870018 -13.84726064  27.2454048   14.23003902 -42.90654351
 24.39457638    9.53842023  16.72293446   30.53624211    5.88303717]
152.9811320754717
```

```
# 일반화 성능 평가
y_preds = ridge.predict( X_test )
from sklearn.metrics import mean_squared_error
```

```
np.sqrt( mean_squared_error(y_test, y_preds) )
```

```
55.089852137313954
```

```
fig, axs = plt.subplots(figsize=(18,6), nrows=1, ncols=len(alphas))
coeff_df = pd.DataFrame()
for pos, al in enumerate(alphas):
    ridge = Ridge(alpha=al)
    ridge.fit(X_train, y_train)
    coeff = pd.Series(data=ridge.coef_, index=diabetes['feature_names'])
    coeff = coeff.sort_values(ascending=False)
    colname = 'alpha:' + str(al)
    axs[pos].set_title(colname)
    axs[pos].set_xlim(-50,50)
    sns.barplot(x=coeff.values, y=coeff.index, ax=axs[pos])
    coeff_df[colname] = coeff
```



```
coeff_df
```

	alpha:0	alpha:0.01	alpha:0.05	alpha:0.1	alpha:0.5	alpha:1	alpha:5
s5	31.138611	31.076226	30.831553	30.536242	28.526648	26.658856	20.679753
bmi	27.225471	27.227572	27.235733	27.245405	27.305084	27.347486	27.247605
s2	25.651076	25.520922	25.010508	24.394576	20.207359	16.324820	4.084076
s4	16.927558	16.906456	16.823497	16.722934	16.023963	15.344355	12.722315
bp	14.248636	14.246725	14.239197	14.230039	14.165214	14.099670	13.796797
s3	10.239273	10.166744	9.882164	9.538420	7.190375	4.990229	-2.289360
s6	5.880362	5.880633	5.881709	5.883037	5.893123	5.904725	5.981982
age	-0.508127	-0.507153	-0.503328	-0.498700	-0.466811	-0.436305	-0.321278

	alpha:0	alpha:0.01	alpha:0.05	alpha:0.1	alpha:0.5	alpha:1	alpha:5
sex	-13.873897	-13.871166	-13.860395	-13.847261	-13.753227	-13.655907	-13.164133
s1	-44.504536	-44.339080	-43.690071	-42.906544	-37.568138	-32.593582	-16.514538

```
# 하이퍼파라미터 튜닝을 자동으로 해주는 함수임
# score를 array로 제공하는것 뿐만 아니라, fold별 평균/분산 계산 및 model selection까지 완수
from sklearn.model_selection import GridSearchCV
parameters={'alpha': [0, 0.01, 0.05, 0.1, 0.5, 1, 5]}
ridge = Ridge( )
grid_ridge = GridSearchCV ( ridge, param_grid=parameters, cv=5,
scoring='neg_mean_squared_error',refit=True)
grid_ridge.fit( X_train, y_train )
scores_df = pd.DataFrame( grid_ridge.cv_results_ )
scores_df.iloc [:, 5:]
# test score는 validation data로 평가한 score(-mse)임
# std는 split별 score의 표준편차
```

params	split0_test_score	split1_test_score	split2_test_score	split3_test_score	split4_test_score
0 {'alpha': 0}	-3464.061113	-3073.496800	-2249.443435	-3714.684946	-3135.462679
1 {'alpha': 0.01}	-3463.952339	-3072.933041	-2249.812087	-3714.461827	-3135.734131
2 {'alpha': 0.05}	-3463.536751	-3070.771202	-2251.269676	-3713.589310	-3136.808917
3 {'alpha': 0.1}	-3463.057920	-3068.263953	-2253.053928	-3712.540867	-3138.126372
4 {'alpha': 0.5}	-3460.381708	-3053.879193	-2265.950063	-3705.435057	-3147.526552
5 {'alpha': 1}	-3458.580847	-3044.029474	-2279.280902	-3698.531771	-3156.575986
6 {'alpha': 5}	-3451.286784	-3025.938931	-2341.662894	-3662.319607	-3180.549641

```
# Random split 후에 튜닝하므로 위와 최적파라미터가 달라질 수 있음.
print(grid_ridge.best_params_)
print(grid_ridge.best_score_)

{'alpha': 0.5}
-3126.6345147986067

ridge_update = grid_ridge.best_estimator_
ridge_update.coef_

array([-0.46681102, -13.75322724,  27.30508411,  14.16521425,
       -37.56813822,  20.20735883,   7.19037502,  16.02396254,
       28.52664799,   5.89312283])
```

```

y_pred = ridge_update.predict( X_test )
np.sqrt( mean_squared_error(y_test, y_preds) )

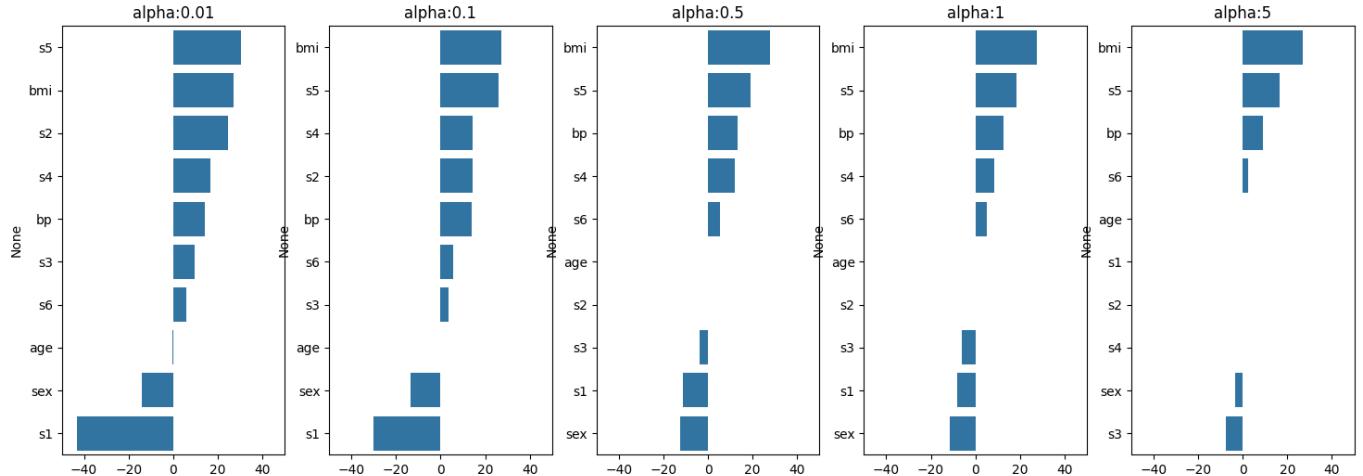
```

55.089852137313954

```

from sklearn.linear_model import Lasso
alphas = [0.01, 0.1, 0.5, 1, 5]
fig, axs = plt.subplots(figsize=(18,6), nrows=1, ncols=len(alphas))
coeff_df = pd.DataFrame()
for pos, al in enumerate(alphas):
    lasso = Lasso( alpha=al, max_iter=1000 )
    lasso.fit ( X_train, y_train)
    coeff = pd.Series(data=lasso.coef_, index=diabetes['feature_names'])
    coeff = coeff.sort_values(ascending=False)
    colname = 'alpha:'+str(al)
    axs[pos].set_title(colname)
    axs[pos].set_xlim(-50,50)
    sns.barplot( x=coeff.values, y=coeff.index, ax=axs[pos])
    coeff_df[colname]=coeff

```



coeff_df

	alpha:0.01	alpha:0.1	alpha:0.5	alpha:1	alpha:5
s5	30.609704	25.856091	18.926493	18.536744	16.632402
bmi	27.241961	27.390066	27.770736	27.646424	27.100696
s2	24.527593	14.429899	0.000000	-0.000000	-0.000000
s4	16.680089	14.454782	12.209271	8.424945	0.000000
bp	14.221648	13.978915	13.382554	12.793145	8.933273

	alpha:0.01	alpha:0.1	alpha:0.5	alpha:1	alpha:5
s3	9.557877	3.432768	-3.656929	-6.063158	-7.745877
s6	5.870014	5.776940	5.495434	5.087787	2.467916
age	-0.486420	-0.291130	-0.000000	-0.000000	0.000000
sex	-13.844396	-13.579080	-12.450801	-11.420067	-3.496842
s1	-43.043735	-29.913885	-11.357006	-8.203584	-0.000000

1.3 Ch4. Logistic Regression

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

creditcard = pd.read_csv('머신러닝ch4_data.csv' )
creditcard.head()
```

	default	student	balance	income
0	No	No	509.778439	38614.41313
1	No	No	0.000000	34836.34070
2	No	No	1197.831505	54652.30930
3	No	No	0.000000	34305.91868
4	No	No	460.234439	47305.21604

```
creditcard['default']= creditcard['default'].map( {'Yes' : 1, 'No' : 0 } )
creditcard['student']= creditcard['student'].map( {'Yes' : 1, 'No' : 0 } )
creditcard.head()
```

	default	student	balance	income
0	0	0	509.778439	38614.41313
1	0	0	0.000000	34836.34070
2	0	0	1197.831505	54652.30930
3	0	0	0.000000	34305.91868
4	0	0	460.234439	47305.21604

```
creditcard.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 833 entries, 0 to 832
Data columns (total 4 columns):
 #   Column   Non-Null Count   Dtype  
---  --  
 0   default   833 non-null    int64  
 1   student   833 non-null    int64  
 2   balance   833 non-null    float64 
 3   income    833 non-null    float64 
dtypes: float64(2), int64(2)
memory usage: 26.2 KB
```

```
creditcard['default'].value_counts()
```

```
default
0      500
1      333
Name: count, dtype: int64
```

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    creditcard[['student', 'balance', 'income']],
    creditcard['default'],
    test_size=0.3, random_state=0)
```

여기서 scaling을 하면 보다 효율적이 됨. 그러나 해석이 어려워져서 odds 해석 등을 위해 scaling은 생략

```
from sklearn.linear_model import LogisticRegression
lr_clf = LogisticRegression()
lr_clf.fit(X_train, y_train)
lr_clf.coef_
```

```
array([-1.02836979e+00,  6.04358513e-03, -7.32071417e-06])
```

```
coeff = pd.Series(data=np.round(lr_clf.coef_[0], decimals=4),
                   index=['student', 'balance', 'income'])
coeff
```

각 parameter는 변량이 1단위 증가할 때, odds가 $\exp(\text{parameter})$ 배 증가한다는 의미

student의 경우 $\exp(-1)$ 로 odds가 감소하는 효과임. 다른 변수는 모두 고정이고 student 여부에 대해서만 따지

```
student      -1.0284
balance      0.0060
income       -0.0000
dtype: float64
```

```
y_pred= lr_clf.predict(X_test)
y_pred[:10]
```

```
array([1, 1, 1, 1, 0, 1, 0, 0, 0, 0])
```

```
# cutoff=0.5를 변경하고싶다면 이 array의 2번째 칼럼 이용하면 됨
y_pred_proba = lr_clf.predict_proba(X_test)
np.round(y_pred_proba[:10],3)
```

```
array([[0.045, 0.955],
       [0.257, 0.743],
       [0.44 , 0.56 ],
       [0.08 , 0.92 ],
       [0.924, 0.076],
       [0.282, 0.718],
       [0.996, 0.004],
       [0.999, 0.001],
       [1.    , 0.    ],
       [0.882, 0.118]])
```

인공지능 및 기계학습 과제1

20249132 김형환

Question 1

1. 다음의 코드를 실행하여, 다중선형회귀모형 훈련과 평가를 위한 데이터셋 Xtrain, ytrain, Xtest, ytest를 생성하고 물음에 답하여라.

```
1 import numpy as np
2 np.random.seed(123)
3 Xtrain = 2 * np.random.rand(100, 3)
4 ytrain = 6 + Xtrain @ np.array([[3], [2], [5]]) + np.random.randn(100, 1)
5 Xtest = 2 * np.random.rand(20, 3)
6 ytest = 6 + Xtest @ np.array([[3], [2], [5]]) + np.random.randn(20, 1)
```

- (1) 훈련자료(Xtrain, ytrain)를 입력하고 반복횟수 iters를 입력하면, 배치 경사하강법(학습율 =0.01을 이용할 것)을 적용하여 다중선형회귀모형을 훈련한 뒤, 편향과 가중치 파라미터 추정치를 아래와 같은 형태로 출력해주는 함수 gradient_descent_steps를 만들어라. 이 함수를 이용하여 주어진 훈련자료를 적합한 뒤 파라미터 추정치를 아래와 같이 w_pred로 저장하여라.

```
1 w_pred = gradient_descent_steps( Xtrain, ytrain, iters=5000 )
2 print( w_pred )
```

[5.78346749]
[2.98606662]
[2.27738749]
[4.88477677]

- (2) 평가자료(Xtest, ytest)를 이용하여, 위 (1)에서 계산된 파라미터 추정치 w_pred로 정의된 다중선형회귀모델의 성능을 MSE로 평가하여라.

Answer

```
import numpy as np
from sklearn.metrics import mean_squared_error

np.random.seed(123)
xtrain = 2 * np.random.rand(100, 3)
ytrain = 6 + xtrain @ np.array([[3], [2], [5]]) + np.random.randn(100, 1)
xtest = 2 * np.random.rand(20, 3)
```

```

ytest = 6 + xtest @ np.array([[3],[2],[5]]) + np.random.randn(20,1)

# (1)
def gradient_descent_steps(xtrain, ytrain, iters):
    # m = 훈련데이터의 수, n = 변수의 개수
    m, n = xtrain.shape

    # 절편항 추가를 위해 x0=1을 각 훈련데이터에 추가
    X = np.insert(xtrain, 0, 1, axis = 1)
    Y = ytrain

    # 파라미터 theta의 초기값은 1, 절편항까지 n+1개
    theta = np.ones(n+1).reshape(n+1,1)

    for i in range(iters):
        # 비용함수 = (오차의 제곱합) / 2m 형태로 가정
        gradient = X.T @ (X @ theta - Y) / m
        theta -= 0.01 * gradient

    return theta

w_pred = gradient_descent_steps(xtrain, ytrain, iters=5000)
print("Best parameters are :")
print(w_pred)

# (2)
ypred = np.insert(xtest,0,1,axis = 1) @ w_pred
mse = mean_squared_error(ytest, ypred)
print("Mean Squared Error is :")
print(mse)

```

Best parameters are :

```

[[5.72120369]
[3.00637126]
[2.29860859]
[4.9014743 ]]

```

Mean Squared Error is :

```
0.8604989666540144
```

Question 2

2. sklearn의 dataset 모듈에서 fetch_california_housing 은 다음에 관한 데이터 정보를 담고 있다. 이 자료를 이용하여 물음에 답하여라.

타겟 데이터

1990년 캘리포니아의 각 행정 구역 내 주택 가격의 중앙값

특징 데이터

MedInc : 행정 구역 내 소득의 중앙값

HouseAge : 행정 구역 내 주택 연식의 중앙값

AveRooms : 평균 방 갯수

AveBedrms : 평균 침실 갯수

Population : 행정 구역 내 인구 수

AveOccup : 평균 자가 비율

Latitude : 해당 행정 구역의 위도

Longitude : 해당 행정 구역의 경도

관찰치 수 : 20640

(1) 주어진 데이터셋을 훈련자료 60%, 평가자료 40%로 랜덤하게 나누어라.

(2) 60%의 훈련자료에 대해 릿지회귀를 훈련하고자 한다. 5-fold 교차검증(cv)를 적용하여, 다음 주어진 규제조절 매개변수(λ) 후보 중 최적의 값을 선택하여라. 단, 검증 기준은 MSE를 이용할 것.

λ 후보값 : 0, 1, 10, 30, 50, 100

(3) 60%의 훈련자료에 대하여 (2)에서 선택된 λ 를 적용한 릿지회귀를 훈련한 뒤, 가중치 파라미터 추정치를 출력하여라. 또한 40%의 평가자료를 이용하여 훈련된 모형에 대한 R^2 를 구하여라.

```
import pandas as pd
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split

housing = fetch_california_housing()

# (1)
y = housing['target']
x = pd.DataFrame( housing['data'], columns=housing['feature_names'] )

# train_test_split 모듈 이용
x_train, x_test, y_train, y_test = train_test_split(x,y,test_size=0.4, random_state=123)
print("Split data to training 0.6 and test 0.4 ratio")
print("Data size of X train, test : ",x_train.shape[0], x_test.shape[0])
```

```
print("Data size of y train, test : ",y_train.shape[0], y_test.shape[0])
```

Split data to training 0.6 and test 0.4 ratio

Data size of X train, test : 12384 8256

Data size of y train, test : 12384 8256

```
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import StandardScaler

# (2) 표준화 후 GridSearchCV 모듈 활용. 하이퍼파라미터 튜닝 및 재훈련.
scaler = StandardScaler()
scaler.fit( x_train )
x_train = scaler.transform( x_train )
x_test = scaler.transform( x_test )

params = {'alpha': [0, 1, 10, 30, 50, 100]}
housing_grid_ridge = GridSearchCV ( Ridge(),
                                    param_grid=params,
                                    cv=5,
                                    scoring='neg_mean_squared_error',
                                    refit=True)

housing_grid_ridge.fit( x_train, y_train )
print("The best lambda(alpha) is :",housing_grid_ridge.best_params_)
```

The best lambda(alpha) is : {'alpha': 1}

```
from sklearn.metrics import r2_score

# (3) GridSearchCV 결과값 및 r2_score 함수 활용
housing_ridge = housing_grid_ridge.best_estimator_
y_pred = housing_ridge.predict( x_test )
R2 = r2_score( y_test, y_pred )

print("The coefficients are : ",housing_ridge.coef_, "\n")
print("R-square is : ", R2)
```

The coefficients are : [0.83361052 0.11707262 -0.28230537 0.33026218 0.00106348 -0.04327138
-0.89953142 -0.87471626]

```
R-square is : 0.6090749286505223
```

GridSearchCV 대신 cross_val_score를 이용해도 동일한 결과를 얻을 수 있습니다.

```
from sklearn.linear_model import Ridge
from sklearn.model_selection import cross_val_score
params = [0, 1, 10, 30, 50, 100]
mse = np.ones(6)

for al in range(6):
    ridge = Ridge(alpha=params[al])
    neg_mse_scores = cross_val_score(ridge, x_train, y_train,
                                      scoring='neg_mean_squared_error', cv=5)
    avg_rmse = np.mean(np.sqrt(-1*neg_mse_scores))
    mse[al] = avg_rmse

alpha_star = params[mse.argmin()]
ridge = Ridge(alpha=alpha_star)
ridge.fit( x_train, y_train )
print("alpha is : ", alpha_star)
print("coef. is : ", ridge.coef_)

alpha is : 1
coef. is : [ 0.83361052  0.11707262 -0.28230537  0.33026218  0.00106348 -0.04327138
 -0.89953142 -0.87471626]
```

Question 3

3. 일반적인 선형 회귀모형에 비해, 그 파라미터에 Lasso 규제를 적용된 Lasso 회귀모형은 어떠하다고 말할 수 있는지, 다음의 설명 중 가장 적절한 것을 골라라.
 - ① 유연성이 높고, 따라서 편향의 증가가 분산 감소보다 작을 경우 예측 정확도가 향상된다.
 - ② 유연성이 높고, 따라서 분산의 증가가 편향 감소보다 작을 경우 예측 정확도가 향상된다.
 - ③ 유연성이 낮고, 따라서 편향의 증가가 분산 감소보다 작을 경우 예측 정확도가 향상된다.
 - ④ 유연성이 낮고, 따라서 분산의 증가가 편향 감소보다 작을 경우 예측 정확도가 향상된다.

Answer : 3번

기본적으로 규제가 있는 회귀모형은 일반적인 회귀모형에 비하여 유연성이 떨어집니다.

비용함수를 최소화시켜나가는 과정에서 파라미터가 만족해야 할 추가적인 조건이 붙기 때문입니다.

이를 알기 쉽게 경사하강법을 예시로 들어보면, 매 파라미터를 업데이트해나갈 때,

규제가 있는 회귀모형에서는 규제로 인해 업데이트를 원하는 만큼 실시할 수 없게 됩니다.

따라서 규제가 조금이라도 존재한다면, 일반적인 회귀모형에 비해서 유연성이 떨어집니다.

유연성은 떨어지는 대신 장점이 있는데, 모델의 **overfitting** 문제를 잘 해결한다는 것 입니다.

모델이 train data로 훈련할 때 파라미터에 제약을 가하기 때문에, 모델이 과적합되지 않도록 적정히 조절하게 됩니다.

이를 종합해보면, 규제가 있는 회귀모형은 유연성이 떨어져 다소 편향이 발생하게 되지만,

과적합을 방지함으로써 분산이 감소되는 효과 있다는 의미가 됩니다.

따라서 규제의 여부와 규제의 정도를 결정할 때, 이러한 trade-off를 잘 이해하여야 합니다.

분산 감소효과가 편향 증가효과보다 클 때, 규제를 가하거나 규제의 정도를 강화하는 것이 적합할 것 입니다. (3번)

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

지도 학습 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 같은 걸 복잡한 모델에 사용할 때, 파라미터 튜닝 과정에서 손실함수가 더 이상 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 y 가 특정 범주 k 에 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

로젠틀란의 퍼셉트론 (Simple perceptron)

활성화함수 $f(\cdot)$ 를 1 또는 -1를 가지는 임계함수로 구성.

$y = 1$ 인데 $f=-1$ 인 경우, $w_i^* = w_i + x_i$

$y = -1$ 인데 $f=1$ 인 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 s 가 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

N 개의 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 $E_N = \sum E_d$

(N 으로 나누던 안나누던 파라미터 업데이트 결정에는 영향 없음)

여기서 SGD(Stochastic Gradient 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, y 가 이진분류 문제일 때 $y=1$ 일 확률값을 예측해주는 모델임.

어차피 퍼셉트론에서는 확률=0.5를 기준으로 분류하므로 선형분류면을 제공하지만,

네트워크를 구성하면 비선형 경계면을 구성할 수도 있음.

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

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

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

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

w_i 의 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_2 를 구성하고, $f = z_1 \times z_2$ 를 최종 모델로 결정하면 문제 해결 가능.

다층 퍼셉트론의 구조

여러 개의 퍼셉트론 뉴런을 여러 층으로 쌓은 구조이며, 층 내에서 뉴런간 교류는 없음.

입력 - 은닉 - 출력 다층퍼셉트론이 대표적인 구조. (I-H-O 다층퍼셉트론)

입력층, 은닉층을 거쳐 출력층의 연산을 통해 최종적으로 값을 출력

파라미터 학습은 각 층마다 가중치로 존재하므로, 실질적으로 층이 2개인 MLP라고 하기도 함.

e.g. 10-3-1 MLP는?

10개의 입력층(+1 bias항), 3개의 은닉층(+1 bias항), 1개의 출력층

총 11개의 입력층과 3개의 은닉층이 모두 연결되어 총 33개의 연결이 생기고,

4개의 은닉층과 1개의 출력층이 연결되어 4개의 연결이 생김. 전체 37개의 연결.

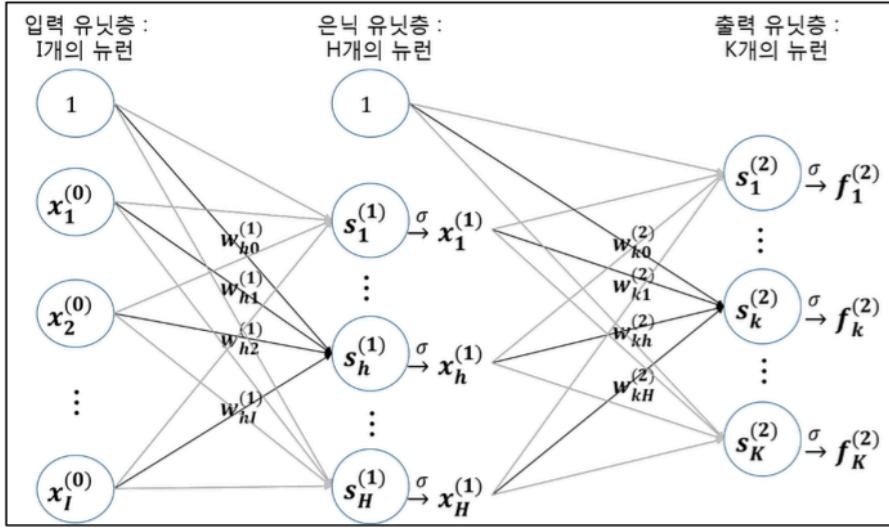
총의 개수나 뉴런의 개수를 모두 돌려봐서 최적의 모델을 찾으면 좋지 않나?

그러나, 딥러닝 모델은 하나의 학습에 많은 시간이 걸려서 그런식으로 하지 않고, 유사 사례를 참고하는 등 적정 모델을 초기부터 잘 설정해야할 필요가 있음.

다층 퍼셉트론의 학습

전향계산 Forward Calculation

각 층간에는 퍼셉트론과 동일한 연산이 적용되며, 따라서 층이 많아질수록 계산이 기하급수적으로 증가.



- 입력값 벡터 $\mathbf{x}^{(0)} = (x_1^{(0)}, \dots, x_I^{(0)})$
- 입력 뉴런 i 의 출력 $x_i^{(0)}$ 는 은닉 뉴런 h 와의 연결 가중치 $w_{hi}^{(1)}$ 를 통해서 전달됨
 - $\mathbf{w}_h^{(1)} = (w_{h0}^{(1)}, \dots, w_{hI}^{(1)}), h = 1, \dots, H$: 입력층($0, \dots, I$)과 은닉층(h) 연결 가중치 벡터
- 은닉 뉴런 h 의 출력 $x_h^{(1)}$
 - $s_h^{(1)} = \sum_{i=0}^I w_{hi}^{(1)} x_i^{(0)}$
 - $x_h^{(1)} = \sigma(s_h^{(1)}) = \frac{1}{1+exp(-s_h^{(1)})}$
- 은닉 뉴런 h 의 출력 $x_h^{(1)}$ 은 출력 뉴런 k 와의 연결 가중치 $w_{kh}^{(2)}$ 를 통해서 전달됨
 - $\mathbf{w}_k^{(2)} = (w_{k0}^{(2)}, \dots, w_{kH}^{(2)}), k = 1, \dots, K$: 은닉층($0, \dots, H$)과 출력층(k) 연결 가중치 벡터
- 출력 뉴런 k 의 출력값 $f_k^{(2)}$
 - $s_k^{(2)} = \sum_{h=0}^H w_{kh}^{(2)} x_h^{(1)}$
 - $f_k^{(2)} = \sigma(s_k^{(2)}) = \frac{1}{1+exp(-s_k^{(2)})}$
- 출력값 벡터 \mathbf{f}
 - $\mathbf{f} = (f_1^{(2)}, \dots, f_K^{(2)})$
 - $f_k^{(2)} = \sigma(\sum_{h=0}^H w_{kh}^{(2)} \cdot \sigma(\sum_{i=0}^I w_{hi}^{(1)} x_i^{(0)}))$

I-H-K MLP에서, I개의 입력층과 H개의 은닉층 사이에 $H \times (I + 1)$ 개의 가중치가 필요하며,

은닉층과 출력층 사이에 $K \times (H + 1)$ 개의 가중치가 필요.

이러한 가중치를 학습할 때 어떻게 하는지? 오류역전파 알고리즘 사용

오류역전파 알고리즘 Error back-propagation

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=float32)>

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([[0.10420418, 0.7570373 , 0.17906117, 0.12324095],
       [0.61085963, 0.85605717, 0.06688213, 0.9268582 ],
       [0.87692904, 0.67397463, 0.07511568, 0.20806742]], dtype=float32)>

t.numpy()

array([[0.10420418, 0.7570373 , 0.17906117, 0.12324095],
       [0.61085963, 0.85605717, 0.06688213, 0.9268582 ],
       [0.87692904, 0.67397463, 0.07511568, 0.20806742]], dtype=float32)

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

<tf.Tensor: shape=(3, 4), dtype=float32, numpy=
array([[ 1.7451856 ,  1.3414903 , -0.10702454, -0.36695087],
       [ 0.7358737 ,  1.091183 , -0.40278485,  0.52324367],
       [ 0.9877435 ,  1.3428662 , -0.70961404,  0.76696813]], dtype=float32)>

t_tr = tf.transpose( t )
t_tr

<tf.Tensor: shape=(4, 3), dtype=float32, numpy=
array([[0.10420418, 0.61085963, 0.87692904],
       [0.7570373 , 0.85605717, 0.67397463],
       [0.17906117, 0.06688213, 0.07511568],
       [0.12324095, 0.9268582 , 0.20806742]], dtype=float32)>

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

<tf.Tensor: shape=(6, 2), dtype=float32, numpy=
array([[0.10420418, 0.7570373 ],
       [0.17906117, 0.12324095],
       [0.61085963, 0.85605717],

```

```
[0.06688213, 0.9268582 ],  
[0.87692904, 0.67397463],  
[0.07511568, 0.20806742]], 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) # subtract, 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([[ 0.08936357,  0.05803037],  
       [-0.45387936,  0.06009912],  
       [-0.5850117 , -0.25081968],  
       [ 0.97455716,  0.2175405 ],  
       [-0.7313497 , -0.7427108 ]], dtype=float32)>
```

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

```

<tf.Tensor: shape=(5, 2), dtype=float32, numpy=
array([[-1.0126729 ,  0.6480458 ],
       [ 0.3113897 ,  0.09961595],
       [-0.5267234 , -0.5192522 ],
       [ 0.41840178,  0.21279272],
       [-0.4534227 , -0.5610016 ]], dtype=float32)>

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

array([-0.141264 , -0.1315721], dtype=float32)

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

array([ 0.07369697, -0.19689012, -0.4179157 ,  0.59604883, -0.73703027],
      dtype=float32)

    tf.multiply( M1, M2 ).numpy

<bound method _EagerTensorBase.numpy of <tf.Tensor: shape=(5, 2), dtype=float32, numpy=
array([[-0.09049607,  0.03760633],
       [-0.14133336,  0.00598683],
       [ 0.30813935,  0.13023867],
       [ 0.40775645,  0.04629103],
       [ 0.33161056,  0.41666195]], dtype=float32)>>>

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

<tf.Tensor: shape=(5, 5), dtype=float32, numpy=
array([[-0.05288974,  0.03360765, -0.07720228,  0.04973832, -0.0730746 ],
       [ 0.4985783 , -0.13534653,  0.20786226, -0.17711528,  0.1720835 ],
       [ 0.42988288, -0.20715228,  0.43837804, -0.29814255,  0.40596783],
       [-0.8459314 ,  0.32513756, -0.6262804 ,  0.45404747, -0.5639269 ],
       [ 0.25930744, -0.30172062,  0.77087325, -0.4640415 ,  0.74827254]], dtype=float32)>

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

<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[0.8156769 ,  0.9341337 ],
       [ 0.5198423 ,  0.63678485]], dtype=float32)>

```

```
tf.transpose(M1) @ M2

<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[0.8156769 , 0.9341337 ],
       [0.5198423 , 0.63678485]], dtype=float32)>
```

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

```
2.740376e-25
```

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

```
0.03380847
```

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

```
array([[ 18.835068, -27.630167],
       [-15.376097,  24.126408]], dtype=float32)
```

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

```
EigResult(eigenvalues=array([1.4287996 , 0.02366215], dtype=float32), eigenvectors=array([[ 0.83600004,  0.5487174 ],
       [ 0.5487174,  0.6467004]], dtype=float32))
```

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

```
<bound method _EagerTensorBase.numpy of <tf.Tensor: shape=(5,), dtype=float32, numpy=
array([0.10655221, 0.45784098, 0.63651335, 0.9985417 , 1.0423492 ],
      dtype=float32)>>
```

2.4 4. 텐서 데이터의 분할 및 통합

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

```
array([0.38545918, 0.40278113, 0.20513606, 0.19092834, 0.08875859,
       0.59647155], dtype=float32)
```

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

```
[array([0.38545918, 0.40278113], dtype=float32),
 array([0.20513606, 0.19092834], dtype=float32),
 array([0.08875859, 0.59647155], dtype=float32)]
```

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

```
[array([0.38545918, 0.40278113, 0.20513606, 0.19092834], dtype=float32),
 array([0.08875859, 0.59647155], dtype=float32)]
```

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

```
<tf.Tensor: shape=(6, 3), dtype=float32, numpy=
array([[0.25079012, 0.05582297, 0.95661616],
       [0.20827067, 0.40616238, 0.5345371 ],
       [0.9391911 , 0.47030878, 0.4284824 ],
       [0.6078584 , 0.02551663, 0.36507952],
       [0.33945346, 0.28689563, 0.11248183],
       [0.80560684, 0.22658253, 0.72398865]], dtype=float32)>
```

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

```
[array([[0.25079012, 0.05582297, 0.95661616],
       [0.20827067, 0.40616238, 0.5345371 ],
       [0.9391911 , 0.47030878, 0.4284824 ],
       [0.6078584 , 0.02551663, 0.36507952]], dtype=float32),
 array([[0.33945346, 0.28689563, 0.11248183],
       [0.80560684, 0.22658253, 0.72398865]], 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([[0.25079012, 0.05582297, 0.95661616, 0.38545918],
       [0.20827067, 0.40616238, 0.5345371 , 0.40278113],
       [0.9391911 , 0.47030878, 0.4284824 , 0.20513606],
       [0.6078584 , 0.02551663, 0.36507952, 0.19092834],
       [0.33945346, 0.28689563, 0.11248183, 0.08875859],
       [0.80560684, 0.22658253, 0.72398865, 0.59647155]], dtype=float32)>
```

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

```
<tf.Tensor: shape=(7, 4), dtype=float32, numpy=
array([[0.25079012, 0.05582297, 0.95661616, 0.38545918],
       [0.20827067, 0.40616238, 0.5345371 , 0.40278113],
       [0.9391911 , 0.47030878, 0.4284824 , 0.20513606],
       [0.6078584 , 0.02551663, 0.36507952, 0.19092834],
       [0.33945346, 0.28689563, 0.11248183, 0.08875859],
       [0.80560684, 0.22658253, 0.72398865, 0.59647155],
       [0.4362433 , 0.8500788 , 0.35958493, 0.52523327]], 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)
```

2024-10-08 10:58:09.829476: 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)
```

2024-10-08 10:58:10.086682: 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))
```

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

for item in ds_X: print(ds_X)

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

2024-10-08 10:58:11.241193: W tensorflow/core/framework/local_rendezvous.cc:404] Local rendezvous

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

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

2024-10-08 10:58:11.325993: W tensorflow/core/framework/local_rendezvous.cc:404] Local rendezvous

```
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
```

2024-10-08 10:58:11.410218: 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
```

2024-10-08 10:58:11.496995: 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
```

```
2024-10-08 10:58:11.598453: 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
```

```
2024-10-08 10:58:11.690558: 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
```

```
2024-10-08 10:58:11.750157: 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]
```

2024-10-08 10:58:11.839267: 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]
```

2024-10-08 10:58:11.921414: 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]
```

2024-10-08 10:58:12.000338: 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 ]]
```

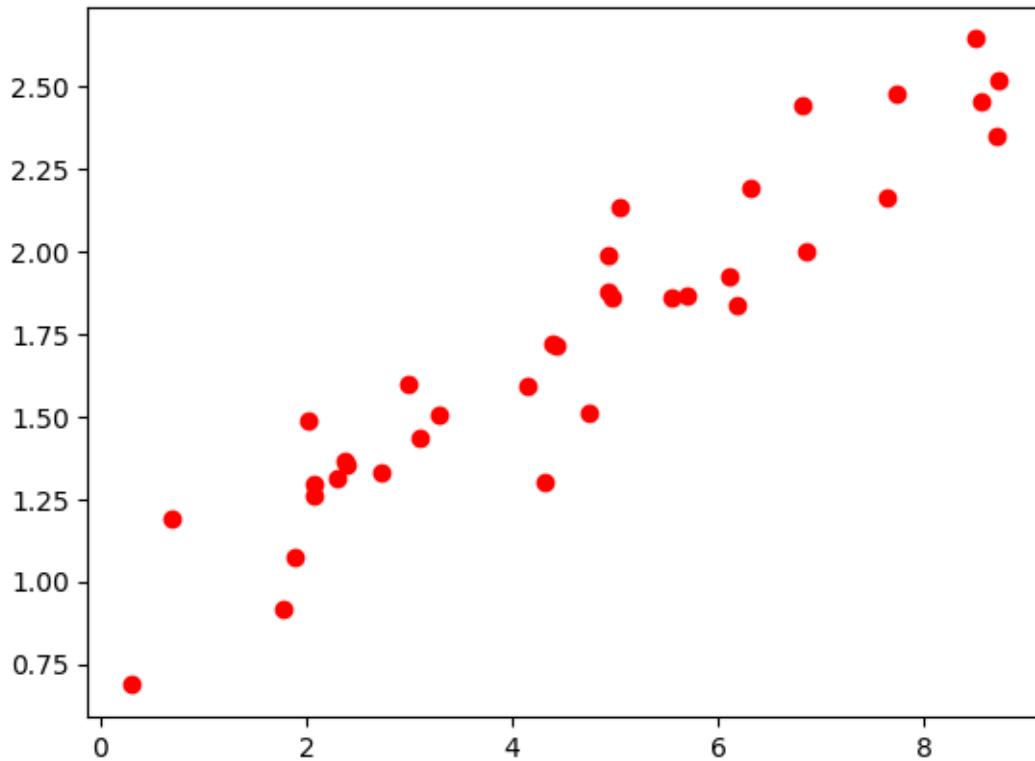
```
[0.8862232  0.88729346  0.78728163]
[0.8893349  0.6277617   0.53197503]] : [10  7  5]
[[0.9757855  0.43509948  0.6601019 ]
[0.05955195 0.0710938   0.3084147 ]
[0.60489583 0.6366315   0.6144488 ]] : [3  8  4]
[[0.25118268 0.9084705   0.47147965]] : [9]
```

2024-10-08 10:58:12.089078: 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() )

```

0.107587084 -2.0054185

```

#  $y \hat{=} wx + b$ 
#  $L = \sum(y \hat{=} -y)^{**2}$ 
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 미니배치에 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

```

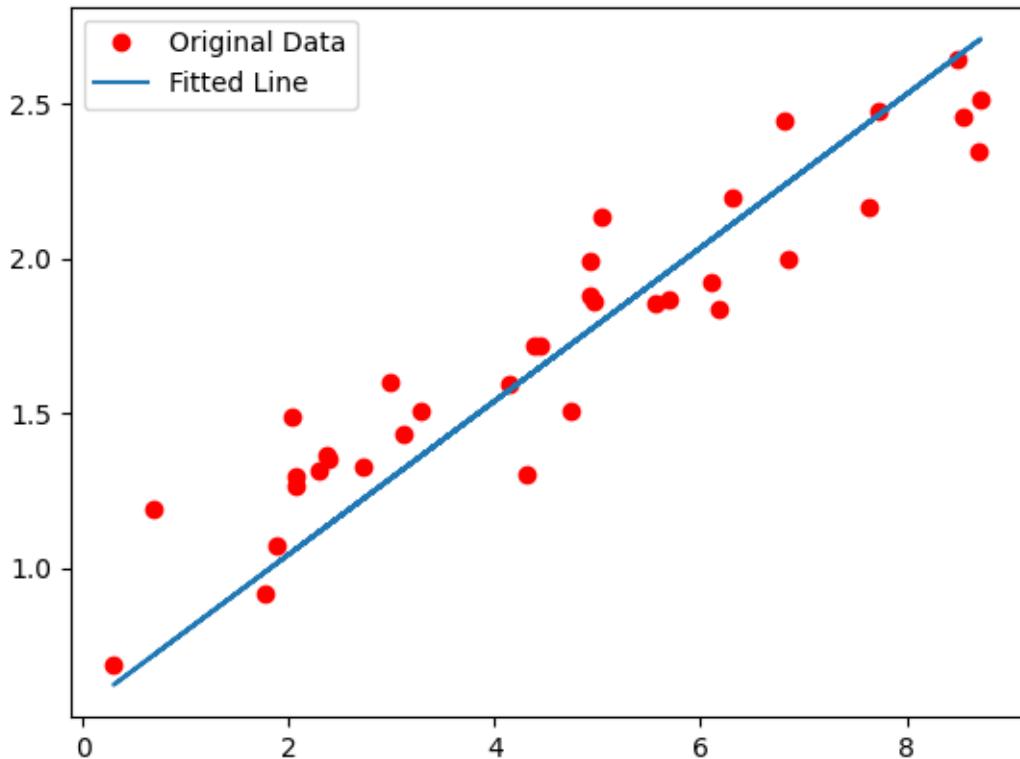
```
optimizer.apply_gradients( zip( gradients, [W, b] ) )

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

```
0 11.433556 0.5089697 -1.9378632
50 1.7550793 0.58110666 -1.4415802
100 1.6205308 0.48617768 -1.0559856
150 0.38359195 0.48032612 -0.72484714
200 0.6002413 0.39505363 -0.46445534
250 0.16318747 0.36287442 -0.2406381
300 0.12085445 0.3561582 -0.05668269
350 0.12631986 0.3037317 0.09371764
400 0.04648442 0.30448112 0.22234914
450 0.040807903 0.26320156 0.32556358
500 0.10858695 0.26078078 0.41337729
550 0.08867306 0.26260927 0.4896761
```

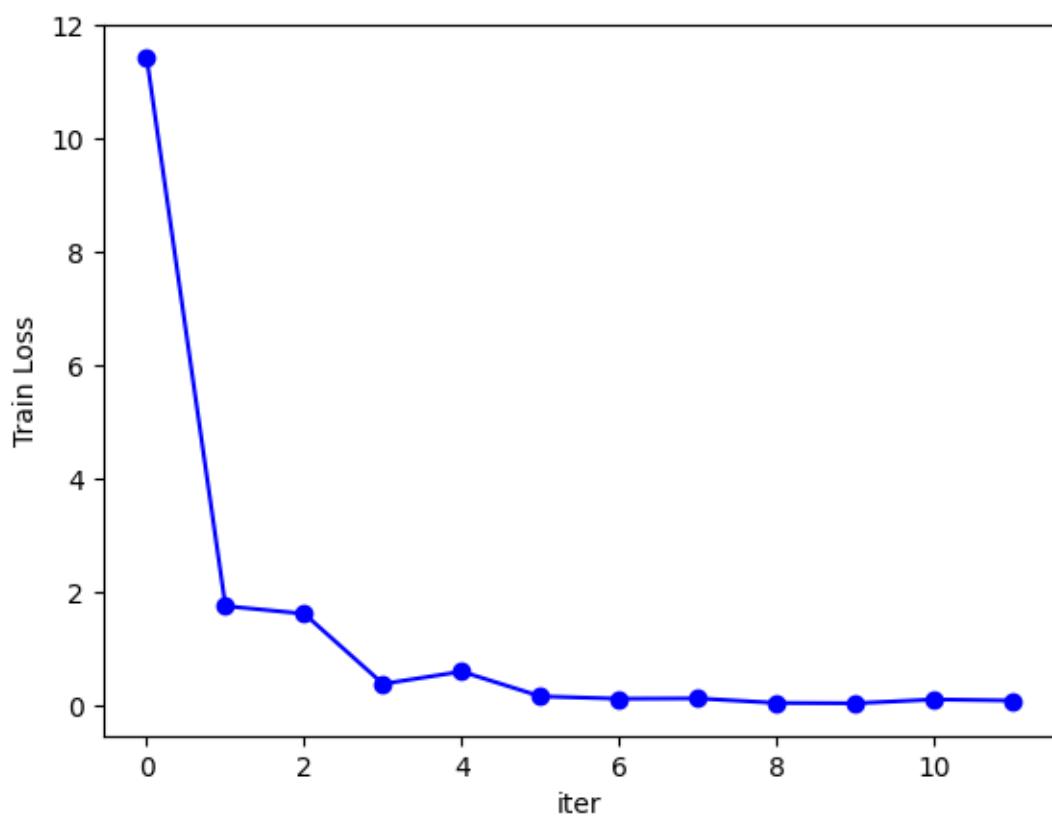
```
2024-10-08 10:58:20.180918: W tensorflow/core/framework/local_rendezvous.cc:404] Local rendezvous
```

```
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()
```

0.0767333

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()
```

Model: "sequential_20"

Layer (type)	Output Shape	Param #
dense_21 (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 944us/step - loss: 0.5380

Epoch 2/100

6/6 ————— 0s 616us/step - loss: 0.1370

Epoch 3/100

6/6 ————— 0s 651us/step - loss: 0.1581

Epoch 4/100
6/6 ————— 0s 722us/step - loss: 0.1099
Epoch 5/100
6/6 ————— 0s 650us/step - loss: 0.1450
Epoch 6/100
6/6 ————— 0s 521us/step - loss: 0.1239
Epoch 7/100
6/6 ————— 0s 684us/step - loss: 0.1102
Epoch 8/100
6/6 ————— 0s 707us/step - loss: 0.1223
Epoch 9/100
6/6 ————— 0s 538us/step - loss: 0.1342
Epoch 10/100
6/6 ————— 0s 495us/step - loss: 0.1137
Epoch 11/100
6/6 ————— 0s 622us/step - loss: 0.1366
Epoch 12/100
6/6 ————— 0s 634us/step - loss: 0.0900
Epoch 13/100
6/6 ————— 0s 545us/step - loss: 0.1060
Epoch 14/100
6/6 ————— 0s 545us/step - loss: 0.0904
Epoch 15/100
6/6 ————— 0s 668us/step - loss: 0.0762
Epoch 16/100
6/6 ————— 0s 548us/step - loss: 0.0984
Epoch 17/100
6/6 ————— 0s 10ms/step - loss: 0.1052
Epoch 18/100
6/6 ————— 0s 1ms/step - loss: 0.0751
Epoch 19/100
6/6 ————— 0s 870us/step - loss: 0.0757
Epoch 20/100
6/6 ————— 0s 759us/step - loss: 0.0702
Epoch 21/100
6/6 ————— 0s 787us/step - loss: 0.0645
Epoch 22/100
6/6 ————— 0s 837us/step - loss: 0.0650
Epoch 23/100

6/6 ————— 0s 1ms/step - loss: 0.0655
Epoch 24/100
6/6 ————— 0s 1ms/step - loss: 0.0627
Epoch 25/100
6/6 ————— 0s 991us/step - loss: 0.0785
Epoch 26/100
6/6 ————— 0s 981us/step - loss: 0.0651
Epoch 27/100
6/6 ————— 0s 1ms/step - loss: 0.0562
Epoch 28/100
6/6 ————— 0s 1ms/step - loss: 0.0879
Epoch 29/100
6/6 ————— 0s 1ms/step - loss: 0.0558
Epoch 30/100
6/6 ————— 0s 817us/step - loss: 0.0628
Epoch 31/100
6/6 ————— 0s 949us/step - loss: 0.0686
Epoch 32/100
6/6 ————— 0s 1ms/step - loss: 0.0473
Epoch 33/100
6/6 ————— 0s 1ms/step - loss: 0.0631
Epoch 34/100
6/6 ————— 0s 1ms/step - loss: 0.0406
Epoch 35/100
6/6 ————— 0s 1ms/step - loss: 0.0491
Epoch 36/100
6/6 ————— 0s 981us/step - loss: 0.0688
Epoch 37/100
6/6 ————— 0s 3ms/step - loss: 0.0460
Epoch 38/100
6/6 ————— 0s 950us/step - loss: 0.0514
Epoch 39/100
6/6 ————— 0s 2ms/step - loss: 0.0470
Epoch 40/100
6/6 ————— 0s 1ms/step - loss: 0.0424
Epoch 41/100
6/6 ————— 0s 1ms/step - loss: 0.0449
Epoch 42/100
6/6 ————— 0s 2ms/step - loss: 0.0346

Epoch 43/100
6/6 ————— 0s 948us/step - loss: 0.0575

Epoch 44/100
6/6 ————— 0s 1ms/step - loss: 0.0459

Epoch 45/100
6/6 ————— 0s 878us/step - loss: 0.0383

Epoch 46/100
6/6 ————— 0s 1ms/step - loss: 0.0413

Epoch 47/100
6/6 ————— 0s 1ms/step - loss: 0.0366

Epoch 48/100
6/6 ————— 0s 1ms/step - loss: 0.0535

Epoch 49/100
6/6 ————— 0s 1ms/step - loss: 0.0439

Epoch 50/100
6/6 ————— 0s 937us/step - loss: 0.0447

Epoch 51/100
6/6 ————— 0s 12ms/step - loss: 0.0350

Epoch 52/100
6/6 ————— 0s 2ms/step - loss: 0.0373

Epoch 53/100
6/6 ————— 0s 875us/step - loss: 0.0289

Epoch 54/100
6/6 ————— 0s 855us/step - loss: 0.0340

Epoch 55/100
6/6 ————— 0s 885us/step - loss: 0.0429

Epoch 56/100
6/6 ————— 0s 1ms/step - loss: 0.0410

Epoch 57/100
6/6 ————— 0s 956us/step - loss: 0.0321

Epoch 58/100
6/6 ————— 0s 995us/step - loss: 0.0426

Epoch 59/100
6/6 ————— 0s 1ms/step - loss: 0.0345

Epoch 60/100
6/6 ————— 0s 1ms/step - loss: 0.0313

Epoch 61/100
6/6 ————— 0s 810us/step - loss: 0.0262

Epoch 62/100

6/6 ————— 0s 824us/step - loss: 0.0268
Epoch 63/100
6/6 ————— 0s 892us/step - loss: 0.0418
Epoch 64/100
6/6 ————— 0s 974us/step - loss: 0.0266
Epoch 65/100
6/6 ————— 0s 1ms/step - loss: 0.0245
Epoch 66/100
6/6 ————— 0s 818us/step - loss: 0.0260
Epoch 67/100
6/6 ————— 0s 1ms/step - loss: 0.0365
Epoch 68/100
6/6 ————— 0s 883us/step - loss: 0.0261
Epoch 69/100
6/6 ————— 0s 856us/step - loss: 0.0233
Epoch 70/100
6/6 ————— 0s 706us/step - loss: 0.0240
Epoch 71/100
6/6 ————— 0s 3ms/step - loss: 0.0204
Epoch 72/100
6/6 ————— 0s 982us/step - loss: 0.0260
Epoch 73/100
6/6 ————— 0s 882us/step - loss: 0.0268
Epoch 74/100
6/6 ————— 0s 1ms/step - loss: 0.0236
Epoch 75/100
6/6 ————— 0s 966us/step - loss: 0.0426
Epoch 76/100
6/6 ————— 0s 1ms/step - loss: 0.0300
Epoch 77/100
6/6 ————— 0s 920us/step - loss: 0.0253
Epoch 78/100
6/6 ————— 0s 752us/step - loss: 0.0207
Epoch 79/100
6/6 ————— 0s 915us/step - loss: 0.0204
Epoch 80/100
6/6 ————— 0s 1ms/step - loss: 0.0265
Epoch 81/100
6/6 ————— 0s 1ms/step - loss: 0.0210

Epoch 82/100
6/6 ————— 0s 1ms/step - loss: 0.0302

Epoch 83/100
6/6 ————— 0s 880us/step - loss: 0.0210

Epoch 84/100
6/6 ————— 0s 823us/step - loss: 0.0197

Epoch 85/100
6/6 ————— 0s 588us/step - loss: 0.0237

Epoch 86/100
6/6 ————— 0s 734us/step - loss: 0.0295

Epoch 87/100
6/6 ————— 0s 1ms/step - loss: 0.0198

Epoch 88/100
6/6 ————— 0s 1ms/step - loss: 0.0321

Epoch 89/100
6/6 ————— 0s 871us/step - loss: 0.0212

Epoch 90/100
6/6 ————— 0s 701us/step - loss: 0.0246

Epoch 91/100
6/6 ————— 0s 708us/step - loss: 0.0241

Epoch 92/100
6/6 ————— 0s 953us/step - loss: 0.0259

Epoch 93/100
6/6 ————— 0s 746us/step - loss: 0.0185

Epoch 94/100
6/6 ————— 0s 856us/step - loss: 0.0296

Epoch 95/100
6/6 ————— 0s 711us/step - loss: 0.0333

Epoch 96/100
6/6 ————— 0s 832us/step - loss: 0.0236

Epoch 97/100
6/6 ————— 0s 913us/step - loss: 0.0272

Epoch 98/100
6/6 ————— 0s 824us/step - loss: 0.0246

Epoch 99/100
6/6 ————— 0s 786us/step - loss: 0.0282

Epoch 100/100
6/6 ————— 0s 698us/step - loss: 0.0232

```

model.weights

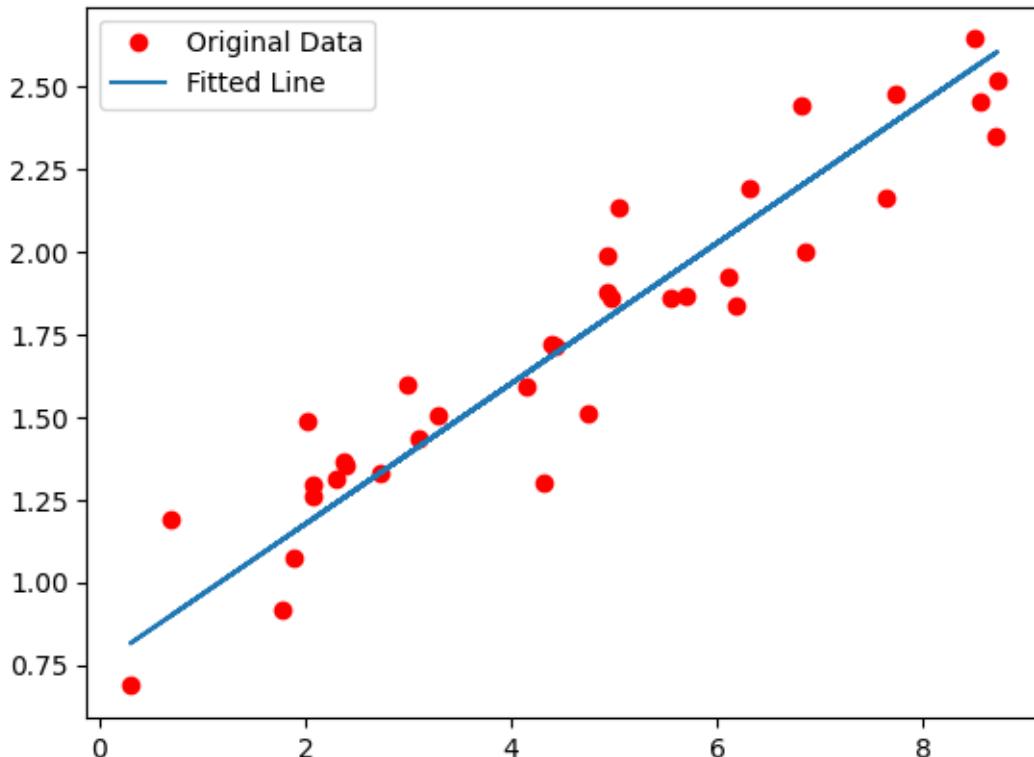
[<KerasVariable shape=(1, 1), dtype=float32, path=sequential_9/dense_10/kernel>,
<KerasVariable shape=(1,), dtype=float32, path=sequential_9/dense_10/bias>]

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

tf.Tensor(0.21239755, shape=(), dtype=float32) tf.Tensor(0.7511314, 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()

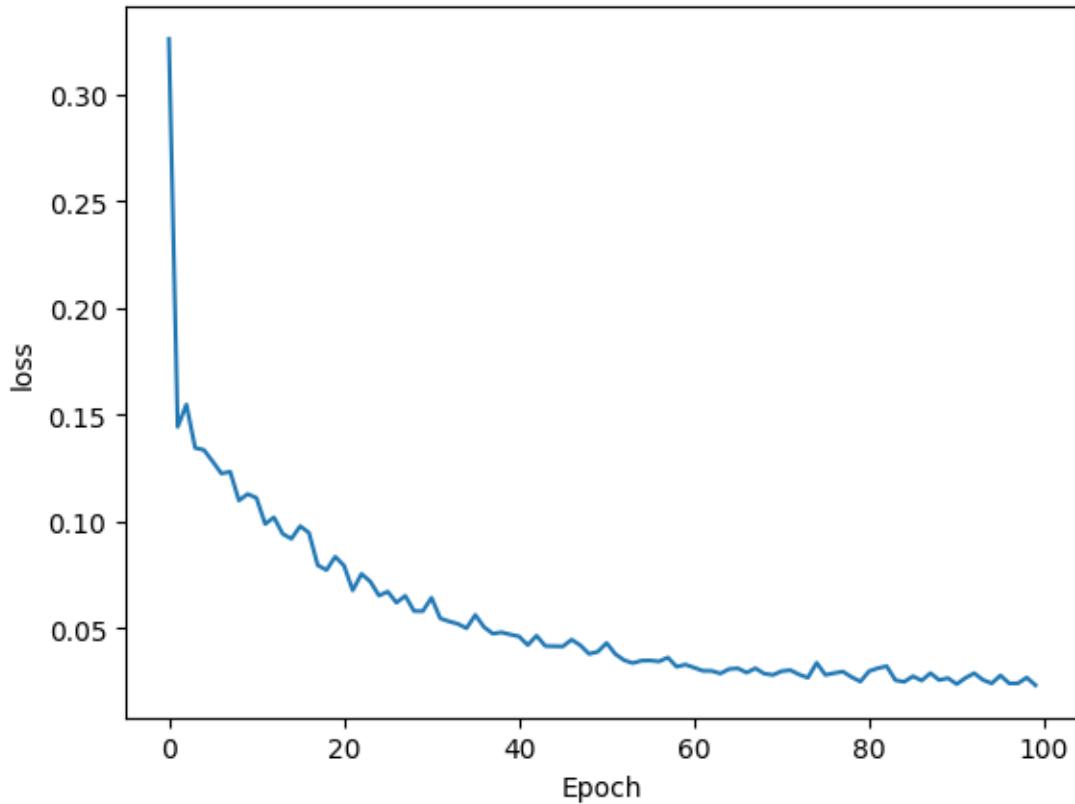
```

1/1 ————— 0s 23ms/step

0.6987257

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

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



```
# validation_split 기능을 .fit에 적용할 수 있으나, tensor 데이터에는 불가함
# tensor는 사전에 validation 데이터를 쪼개서 만들어둬야함
```

```
ds_train3 = tf.data.Dataset.from_tensor_slices((trainX, trainY))
```

```
ds_val = ds_train3.take(6)
ds_val = ds_val.batch(6)
```

```
ds_train_f = ds_train3.skip(6)
ds_train_f = ds_train_f.shuffle(24).batch(6)
```

```
model2 = tf.keras.models.Sequential()
model2.add(tf.keras.layers.Dense(1, input_dim = 1, activation='linear'))
model2.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_23"

Layer (type)	Output Shape	Param #
dense_24 (Dense)	(None, 1)	2

Total params: 2 (8.00 B)

Trainable params: 2 (8.00 B)

Non-trainable params: 0 (0.00 B)

```
model2.compile( loss='mse', optimizer = tf.keras.optimizers.SGD( learning_rate=0.01 ) )
history2 = model2.fit( ds_train_f, epochs=100, verbose=1, validation_data=ds_val )
```

Epoch 1/100

4/4 ━━━━━━━━━━ 0s 14ms/step - loss: 35.8137 - val_loss: 0.3099

Epoch 2/100

4/4 ━━━━━━━━━━ 0s 1ms/step - loss: 0.1149 - val_loss: 0.0790

Epoch 3/100

4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0795 - val_loss: 0.0780

Epoch 4/100

4/4 ━━━━━━━━━━ 0s 1ms/step - loss: 0.0698 - val_loss: 0.0848

Epoch 5/100

4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0557 - val_loss: 0.0752

Epoch 6/100

4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0541 - val_loss: 0.0775

Epoch 7/100

4/4 ━━━━━━━━━━ 0s 1ms/step - loss: 0.0581 - val_loss: 0.0727

Epoch 8/100

4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0677 - val_loss: 0.0713

Epoch 9/100

4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0500 - val_loss: 0.0693

Epoch 10/100

4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0469 - val_loss: 0.0728

Epoch 11/100

```
4/4 ━━━━━━━━━━ 0s 1ms/step - loss: 0.0710 - val_loss: 0.0695
Epoch 12/100
4/4 ━━━━━━━━━━ 0s 1ms/step - loss: 0.0452 - val_loss: 0.0662
Epoch 13/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0717 - val_loss: 0.0666
Epoch 14/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0439 - val_loss: 0.0716
Epoch 15/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0464 - val_loss: 0.0630
Epoch 16/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0457 - val_loss: 0.0770
Epoch 17/100
4/4 ━━━━━━━━━━ 0s 3ms/step - loss: 0.0621 - val_loss: 0.0611
Epoch 18/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0512 - val_loss: 0.0603
Epoch 19/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0455 - val_loss: 0.0698
Epoch 20/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0505 - val_loss: 0.0589
Epoch 21/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0543 - val_loss: 0.0612
Epoch 22/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0452 - val_loss: 0.0571
Epoch 23/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0464 - val_loss: 0.0719
Epoch 24/100
4/4 ━━━━━━━━━━ 0s 3ms/step - loss: 0.0429 - val_loss: 0.0568
Epoch 25/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0466 - val_loss: 0.0531
Epoch 26/100
4/4 ━━━━━━━━━━ 0s 3ms/step - loss: 0.0453 - val_loss: 0.0565
Epoch 27/100
4/4 ━━━━━━━━━━ 0s 3ms/step - loss: 0.0473 - val_loss: 0.0530
Epoch 28/100
4/4 ━━━━━━━━━━ 0s 13ms/step - loss: 0.0424 - val_loss: 0.0508
Epoch 29/100
4/4 ━━━━━━━━━━ 0s 3ms/step - loss: 0.0373 - val_loss: 0.0502
Epoch 30/100
4/4 ━━━━━━━━━━ 0s 3ms/step - loss: 0.0422 - val_loss: 0.0721
```

Epoch 31/100
4/4 ————— 0s 7ms/step - loss: 0.0386 - val_loss: 0.0517

Epoch 32/100
4/4 ————— 0s 3ms/step - loss: 0.0406 - val_loss: 0.0526

Epoch 33/100
4/4 ————— 0s 3ms/step - loss: 0.0462 - val_loss: 0.0625

Epoch 34/100
4/4 ————— 0s 3ms/step - loss: 0.0393 - val_loss: 0.0508

Epoch 35/100
4/4 ————— 0s 4ms/step - loss: 0.0356 - val_loss: 0.0527

Epoch 36/100
4/4 ————— 0s 2ms/step - loss: 0.0342 - val_loss: 0.0508

Epoch 37/100
4/4 ————— 0s 4ms/step - loss: 0.0349 - val_loss: 0.0529

Epoch 38/100
4/4 ————— 0s 4ms/step - loss: 0.0368 - val_loss: 0.0455

Epoch 39/100
4/4 ————— 0s 4ms/step - loss: 0.0371 - val_loss: 0.0465

Epoch 40/100
4/4 ————— 0s 2ms/step - loss: 0.0346 - val_loss: 0.0520

Epoch 41/100
4/4 ————— 0s 3ms/step - loss: 0.0334 - val_loss: 0.0437

Epoch 42/100
4/4 ————— 0s 3ms/step - loss: 0.0377 - val_loss: 0.0463

Epoch 43/100
4/4 ————— 0s 2ms/step - loss: 0.0304 - val_loss: 0.0482

Epoch 44/100
4/4 ————— 0s 3ms/step - loss: 0.0328 - val_loss: 0.0541

Epoch 45/100
4/4 ————— 0s 2ms/step - loss: 0.0303 - val_loss: 0.0499

Epoch 46/100
4/4 ————— 0s 2ms/step - loss: 0.0298 - val_loss: 0.0467

Epoch 47/100
4/4 ————— 0s 2ms/step - loss: 0.0311 - val_loss: 0.0418

Epoch 48/100
4/4 ————— 0s 2ms/step - loss: 0.0264 - val_loss: 0.0474

Epoch 49/100
4/4 ————— 0s 2ms/step - loss: 0.0378 - val_loss: 0.0463

Epoch 50/100

```
4/4 ━━━━━━━━━━ 0s 3ms/step - loss: 0.0301 - val_loss: 0.0561
Epoch 51/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0333 - val_loss: 0.0436
Epoch 52/100
4/4 ━━━━━━━━━━ 0s 3ms/step - loss: 0.0275 - val_loss: 0.0472
Epoch 53/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0387 - val_loss: 0.0442
Epoch 54/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0313 - val_loss: 0.0447
Epoch 55/100
4/4 ━━━━━━━━━━ 0s 14ms/step - loss: 0.0260 - val_loss: 0.0484
Epoch 56/100
4/4 ━━━━━━━━━━ 0s 4ms/step - loss: 0.0281 - val_loss: 0.0379
Epoch 57/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0310 - val_loss: 0.0425
Epoch 58/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0294 - val_loss: 0.0399
Epoch 59/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0254 - val_loss: 0.0414
Epoch 60/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0283 - val_loss: 0.0378
Epoch 61/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0361 - val_loss: 0.0392
Epoch 62/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0234 - val_loss: 0.0412
Epoch 63/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0318 - val_loss: 0.0359
Epoch 64/100
4/4 ━━━━━━━━━━ 0s 4ms/step - loss: 0.0258 - val_loss: 0.0436
Epoch 65/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0320 - val_loss: 0.0457
Epoch 66/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0234 - val_loss: 0.0499
Epoch 67/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0309 - val_loss: 0.0408
Epoch 68/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0226 - val_loss: 0.0386
Epoch 69/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0306 - val_loss: 0.0384
```

Epoch 70/100
4/4 ————— 0s 1ms/step - loss: 0.0297 - val_loss: 0.0500

Epoch 71/100
4/4 ————— 0s 1ms/step - loss: 0.0367 - val_loss: 0.0370

Epoch 72/100
4/4 ————— 0s 1ms/step - loss: 0.0212 - val_loss: 0.0493

Epoch 73/100
4/4 ————— 0s 2ms/step - loss: 0.0249 - val_loss: 0.0389

Epoch 74/100
4/4 ————— 0s 2ms/step - loss: 0.0263 - val_loss: 0.0379

Epoch 75/100
4/4 ————— 0s 1ms/step - loss: 0.0278 - val_loss: 0.0485

Epoch 76/100
4/4 ————— 0s 1ms/step - loss: 0.0322 - val_loss: 0.0356

Epoch 77/100
4/4 ————— 0s 1ms/step - loss: 0.0274 - val_loss: 0.0465

Epoch 78/100
4/4 ————— 0s 1ms/step - loss: 0.0306 - val_loss: 0.0539

Epoch 79/100
4/4 ————— 0s 1ms/step - loss: 0.0316 - val_loss: 0.0491

Epoch 80/100
4/4 ————— 0s 1ms/step - loss: 0.0255 - val_loss: 0.0389

Epoch 81/100
4/4 ————— 0s 2ms/step - loss: 0.0304 - val_loss: 0.0441

Epoch 82/100
4/4 ————— 0s 1ms/step - loss: 0.0220 - val_loss: 0.0452

Epoch 83/100
4/4 ————— 0s 1ms/step - loss: 0.0188 - val_loss: 0.0315

Epoch 84/100
4/4 ————— 0s 1ms/step - loss: 0.0165 - val_loss: 0.0477

Epoch 85/100
4/4 ————— 0s 1ms/step - loss: 0.0191 - val_loss: 0.0417

Epoch 86/100
4/4 ————— 0s 1ms/step - loss: 0.0263 - val_loss: 0.0429

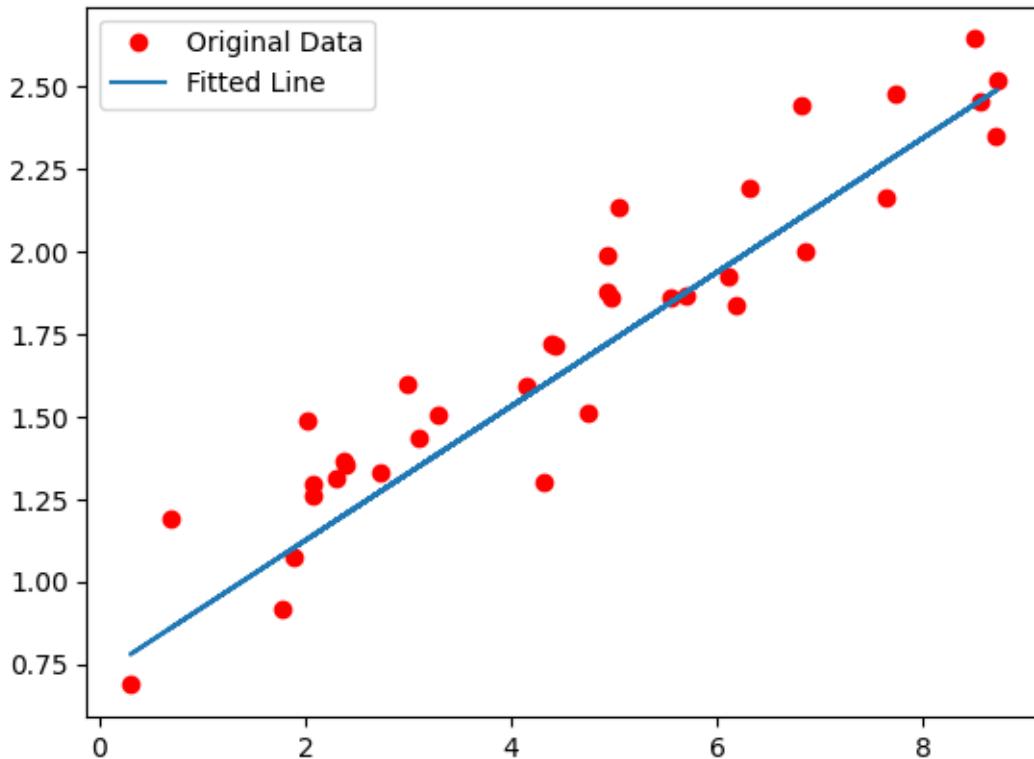
Epoch 87/100
4/4 ————— 0s 1ms/step - loss: 0.0248 - val_loss: 0.0414

Epoch 88/100
4/4 ————— 0s 3ms/step - loss: 0.0214 - val_loss: 0.0521

Epoch 89/100

```
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0242 - val_loss: 0.0363
Epoch 90/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0242 - val_loss: 0.0306
Epoch 91/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0215 - val_loss: 0.0376
Epoch 92/100
4/4 ━━━━━━━━━━ 0s 1ms/step - loss: 0.0233 - val_loss: 0.0376
Epoch 93/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0227 - val_loss: 0.0330
Epoch 94/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0228 - val_loss: 0.0462
Epoch 95/100
4/4 ━━━━━━━━━━ 0s 1ms/step - loss: 0.0240 - val_loss: 0.0339
Epoch 96/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0227 - val_loss: 0.0298
Epoch 97/100
4/4 ━━━━━━━━━━ 0s 1ms/step - loss: 0.0264 - val_loss: 0.0332
Epoch 98/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0267 - val_loss: 0.0367
Epoch 99/100
4/4 ━━━━━━━━━━ 0s 2ms/step - loss: 0.0296 - val_loss: 0.0418
Epoch 100/100
4/4 ━━━━━━━━━━ 0s 1ms/step - loss: 0.0214 - val_loss: 0.0404
```

```
W2 = model2.weights[0][0][0]
b2 = model2.weights[1][0]
plt.plot(X, Y, 'ro', label='Original Data')
plt.plot(X, np.array( W2 * X + b2 ), label='Fitted Line' )
plt.legend()
```



2.8 Ch2 Neural Network 실습

```
import pandas as pd
resign_df = pd.read_csv('실습/resign.csv')
resign_df.head()
```

	satisfaction	evaluation	project	workhour	years	accident	resign	promotion	good
0	0.38	0.53	2	157	3	0	1	0	0
1	0.80	0.86	5	262	6	0	1	0	1
2	0.11	0.88	7	272	4	0	1	0	1
3	0.72	0.87	5	223	5	0	1	0	1
4	0.37	0.52	2	159	3	0	1	0	0

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
resign_df_X = scaler.fit_transform(resign_df.iloc[:, [0, 1, 2, 3, 4, 5, 7]])
resign_df_Y = resign_df['resign']
```

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(
    resign_df_X, resign_df_Y, test_size=0.2, random_state=0)
```

```

import tensorflow as tf
model = tf.keras.Sequential()
model.add( tf.keras.layers.Dense ( units=3, input_dim=7, activation='relu') )
model.add( tf.keras.layers.Dense ( units=1, activation='sigmoid') )

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

model.summary()

```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 3)	24
dense_2 (Dense)	(None, 1)	4

Total params: 28 (112.00 B)

Trainable params: 28 (112.00 B)

Non-trainable params: 0 (0.00 B)

```

model.compile( optimizer='sgd', loss='binary_crossentropy', metrics=['accuracy'])

result = model.fit( x_train, y_train, validation_split=0.2, epochs=40, verbose=1)

```

Epoch 1/40

300/300 ━━━━━━━━━━━━ 1s 789us/step - accuracy: 0.6652 - loss: 0.6689 - val_

Epoch 2/40

300/300 ━━━━━━━━━━━━ 0s 765us/step - accuracy: 0.7722 - loss: 0.4778 - val_

Epoch 3/40

300/300 ━━━━━━━━━━━━ 0s 509us/step - accuracy: 0.7917 - loss: 0.4320 - val_

Epoch 4/40

300/300 ━━━━━━━━━━━━ 0s 460us/step - accuracy: 0.8007 - loss: 0.4001 - val_

Epoch 5/40

300/300 ————— 0s 499us/step - accuracy: 0.8181 - loss: 0.3704 - val_

Epoch 6/40

300/300 ————— 0s 587us/step - accuracy: 0.8767 - loss: 0.3425 - val_

Epoch 7/40

300/300 ————— 0s 681us/step - accuracy: 0.8997 - loss: 0.3177 - val_

Epoch 8/40

300/300 ————— 0s 542us/step - accuracy: 0.9072 - loss: 0.2968 - val_

Epoch 9/40

300/300 ————— 0s 1ms/step - accuracy: 0.9124 - loss: 0.2788 - val_ac

Epoch 10/40

300/300 ————— 0s 570us/step - accuracy: 0.9167 - loss: 0.2637 - val_

Epoch 11/40

300/300 ————— 0s 458us/step - accuracy: 0.9233 - loss: 0.2514 - val_

Epoch 12/40

300/300 ————— 0s 455us/step - accuracy: 0.9255 - loss: 0.2416 - val_

Epoch 13/40

300/300 ————— 0s 455us/step - accuracy: 0.9274 - loss: 0.2338 - val_

Epoch 14/40

300/300 ————— 0s 444us/step - accuracy: 0.9308 - loss: 0.2273 - val_

Epoch 15/40

300/300 ————— 0s 447us/step - accuracy: 0.9317 - loss: 0.2219 - val_

Epoch 16/40

300/300 ————— 0s 509us/step - accuracy: 0.9327 - loss: 0.2173 - val_

Epoch 17/40

300/300 ————— 0s 512us/step - accuracy: 0.9350 - loss: 0.2134 - val_

Epoch 18/40

300/300 ————— 0s 486us/step - accuracy: 0.9359 - loss: 0.2101 - val_

Epoch 19/40

300/300 ————— 0s 521us/step - accuracy: 0.9374 - loss: 0.2074 - val_

Epoch 20/40

300/300 ————— 0s 744us/step - accuracy: 0.9383 - loss: 0.2050 - val_

Epoch 21/40

300/300 ————— 0s 464us/step - accuracy: 0.9393 - loss: 0.2030 - val_

Epoch 22/40

300/300 ————— 0s 459us/step - accuracy: 0.9399 - loss: 0.2012 - val_

Epoch 23/40

300/300 ————— 0s 456us/step - accuracy: 0.9406 - loss: 0.1994 - val_

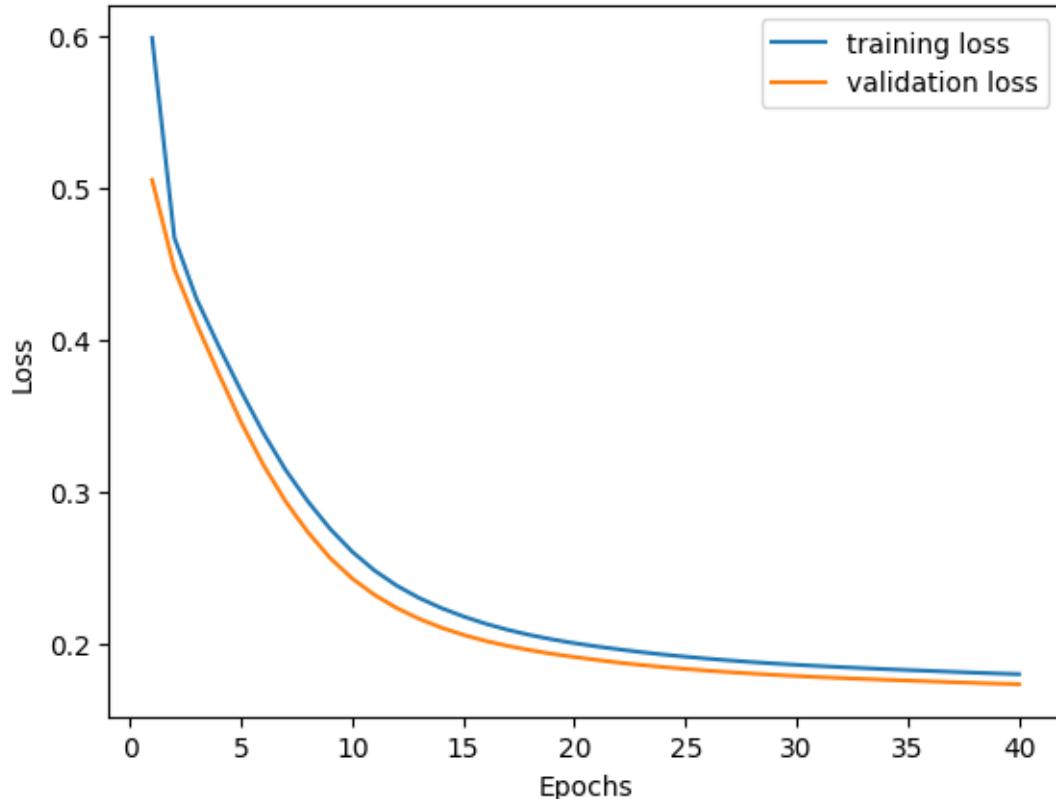
Epoch 24/40

300/300 ————— 0s 445us/step - accuracy: 0.9410 - loss: 0.1979 - val_

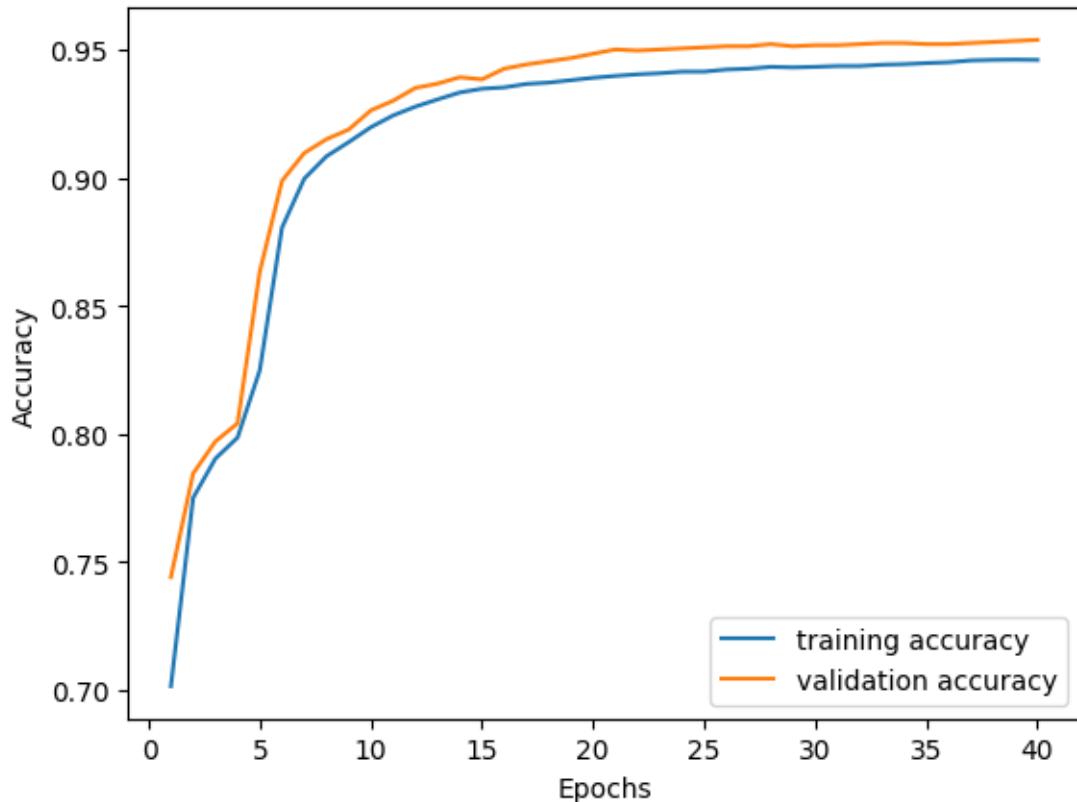
```
Epoch 25/40
300/300 ----- 0s 446us/step - accuracy: 0.9409 - loss: 0.1965 - val_
Epoch 26/40
300/300 ----- 0s 444us/step - accuracy: 0.9417 - loss: 0.1952 - val_
Epoch 27/40
300/300 ----- 0s 484us/step - accuracy: 0.9424 - loss: 0.1941 - val_
Epoch 28/40
300/300 ----- 0s 573us/step - accuracy: 0.9432 - loss: 0.1931 - val_
Epoch 29/40
300/300 ----- 0s 560us/step - accuracy: 0.9431 - loss: 0.1922 - val_
Epoch 30/40
300/300 ----- 0s 476us/step - accuracy: 0.9433 - loss: 0.1914 - val_
Epoch 31/40
300/300 ----- 0s 565us/step - accuracy: 0.9439 - loss: 0.1907 - val_
Epoch 32/40
300/300 ----- 0s 580us/step - accuracy: 0.9438 - loss: 0.1900 - val_
Epoch 33/40
300/300 ----- 0s 520us/step - accuracy: 0.9444 - loss: 0.1894 - val_
Epoch 34/40
300/300 ----- 0s 481us/step - accuracy: 0.9449 - loss: 0.1888 - val_
Epoch 35/40
300/300 ----- 0s 454us/step - accuracy: 0.9452 - loss: 0.1882 - val_
Epoch 36/40
300/300 ----- 0s 454us/step - accuracy: 0.9456 - loss: 0.1877 - val_
Epoch 37/40
300/300 ----- 0s 455us/step - accuracy: 0.9462 - loss: 0.1872 - val_
Epoch 38/40
300/300 ----- 0s 460us/step - accuracy: 0.9467 - loss: 0.1867 - val_
Epoch 39/40
300/300 ----- 0s 466us/step - accuracy: 0.9470 - loss: 0.1862 - val_
Epoch 40/40
300/300 ----- 0s 464us/step - accuracy: 0.9470 - loss: 0.1857 - val_
```

```
import matplotlib.pyplot as plt
import numpy as np
epochs=np.arange(1, 40+1)
plt.plot(epochs, result.history['loss'], label='training loss')
plt.plot(epochs, result.history['val_loss'], label='validation loss')
plt.xlabel('Epochs')
```

```
plt.ylabel('Loss')
plt.legend()
plt.show()
# 검증데이터의 loss가 더 낮은게 이상해보일 수 있는데, epoch를 키울수록 훈련데이터가 낮아짐
```



```
epochs=np.arange(1, 40+1)
plt.plot(epochs, result.history['accuracy'], label='training accuracy')
plt.plot(epochs, result.history['val_accuracy'], label='validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



```
model.evaluate(x_test, y_test) # 평가데이터를 이용하여 모델 일반화 성능 평가
```

```
94/94 ━━━━━━━━━━ 0s 996us/step - accuracy: 0.9522 - loss: 0.1687
```

```
[0.17468173801898956, 0.9523333311080933]
```

```
x_test[0, ]
```

```
array([-0.69516483, -0.85358047, -1.46286291, -1.20241514, -0.34123516,
       -0.41116529, -0.14741182])
```

```
x_new = np.array([-0.7, -0.9, -1.3, -0.9, 0.3, 0.4, -0.02], dtype=np.float32)
y_pred = model.predict(x_new.reshape(1,7)) # 차원변환 필요
y_pred
```

```
1/1 ━━━━━━━━━━ 0s 25ms/step
```

```
array([[0.46218672]], dtype=float32)
```

```
x_new2 = np.array([[-0.7, -0.9, -1.3, -0.9, 0.3, 0.4, -0.02], [0, 0, 0, 0, 0, 0, 0]])
y_pred2 = model.predict(x_new2)
y_pred2
```

1/1 ━━━━━━━━ 0s 25ms/step

```
array([[0.46218672],  
       [0.01171311]], dtype=float32)
```

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, \dots, 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 중심극한정리에 따라 수학적으로 정의할 수 있음

i 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(probability convergence)

오차의 표준편차는 $\frac{\sigma}{\sqrt{N}}$ 이므로, 표본의 크기가 100배 증가하면 오차의 표준편차는 10배 감소함

이외에도 간단한 사다리꼴(trapezoidal) 방식을 이용해볼 수 있음.

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

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

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

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 경로의존성이라고 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 이산오일러근사가 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 $Prediction error = Variance + Bias^2$

$$\begin{aligned} Var[\epsilon] &= E[\epsilon^2] - (E[\epsilon])^2 \\ MSE &= E[\epsilon^2] = Var[\epsilon] + (E[\epsilon])^2 = Variance + Bias^2 \end{aligned}$$

분산감소와 계산시간

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

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

(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 s 이고, 한번의 시뮬레이션의 계산량이 τ 일 때, 가능한 시뮬레이션 횟수는 s/τ 임

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

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

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

편의

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

이산오일리근사를 사용할 때 편의가 발생함.

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

이 때, 이산구간의 간격 m 을 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_2 개의 경로를 추가로 생성하여 경로를 이중으로 구성한다면 bias 제거가 가능함.

Asian Option 평가 해볼 것

시뮬레이션방법론 Ch3-4

샘플 경로 생성법, Variance Reduction Techniques

Brownian bridge

GBM에 따라 결정된 t 시점의 값이 있을 때, 0과 t 사이의 시점 s 에 대한 분포를 어떻게 구하지?

Brownian bridge를 이용해 베리어옵션 평가

먼저, T 시점의 값을 생성하고, ITM일 때만 bridge를 생성해서 베리어옵션 평가!

시뮬레이션방법론 실습

```
import os
if os.getcwd() != '/Users/hwan/Desktop/Homepage/study_24fall/실습':
    os.chdir('실습')
os.getcwd()
'/Users/hwan/Desktop/Homepage/study_24fall/실습'
```

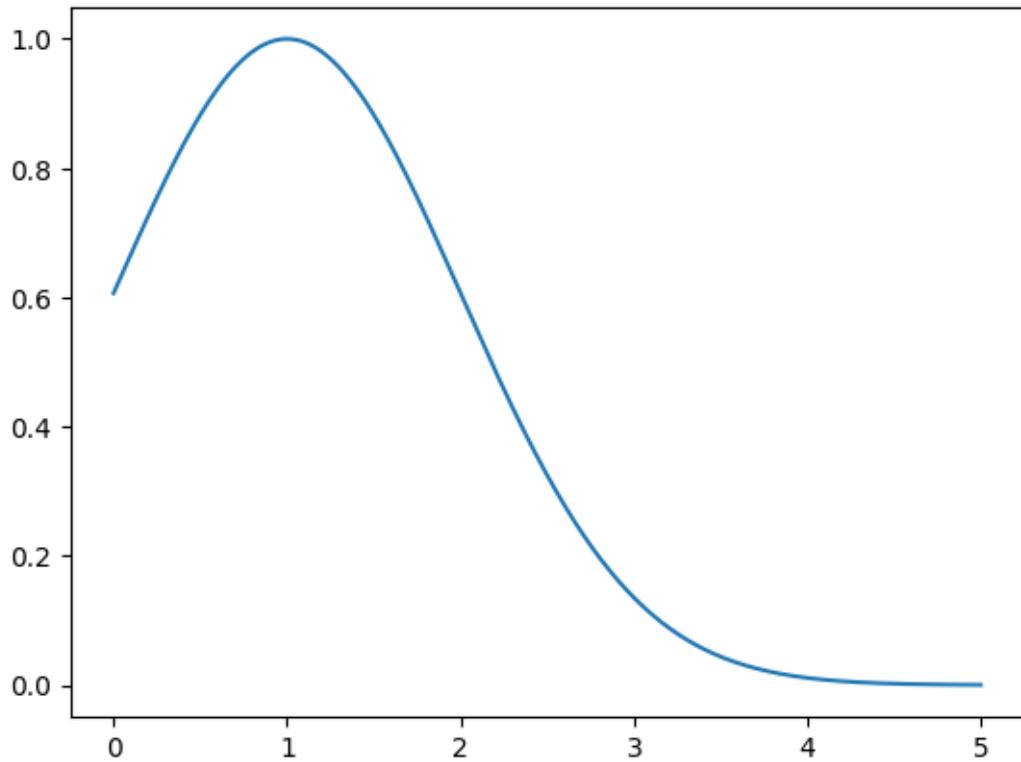
2.9 Ch2. 난수 생성 방법

2.9.1 Acceptance-rejection method

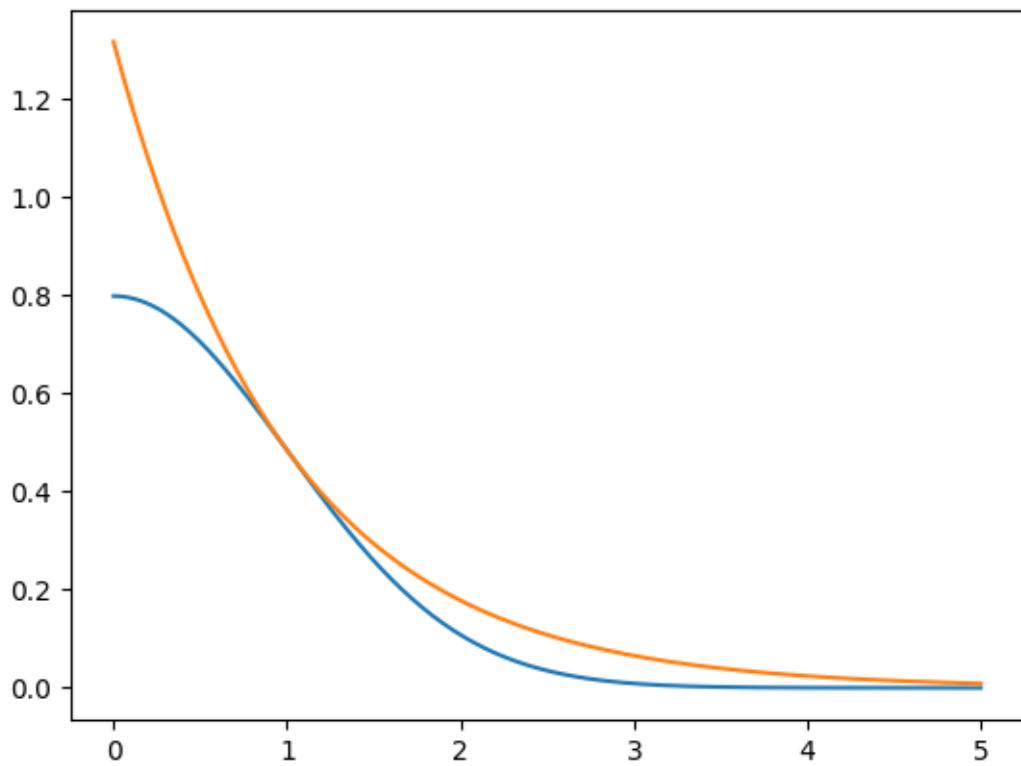
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import scipy.stats as stats

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)

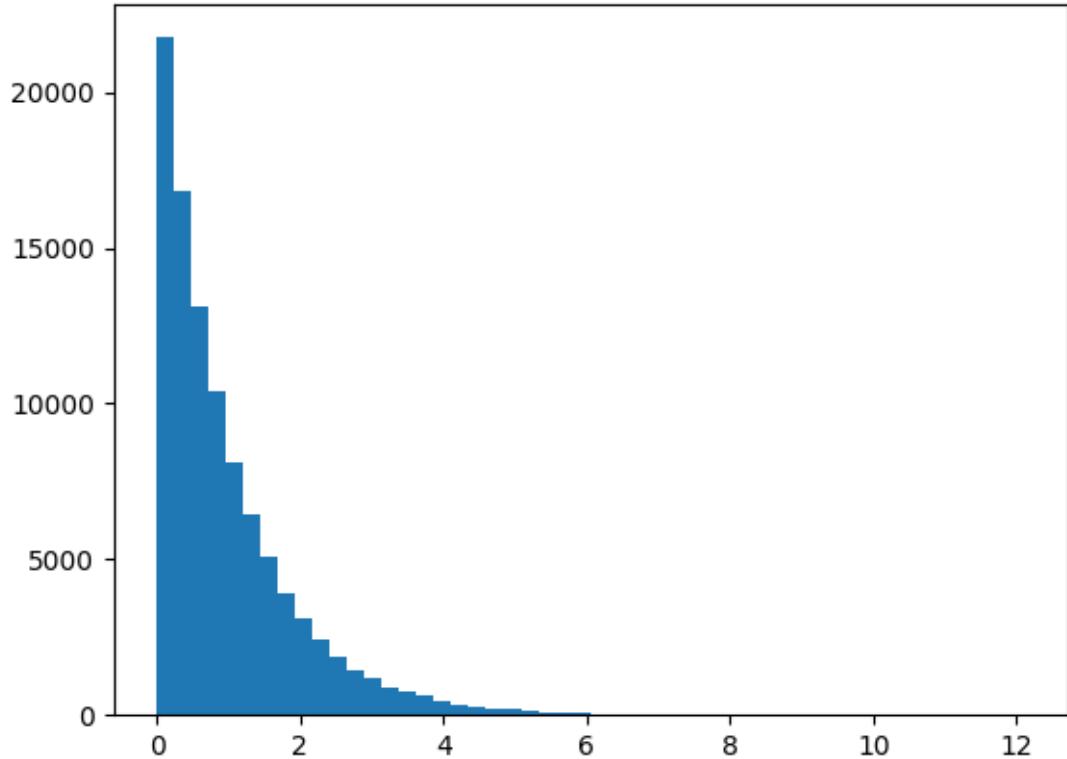
#c = 1
plt.plot(x,f(x)/(c*g(x)))
plt.show()
```



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



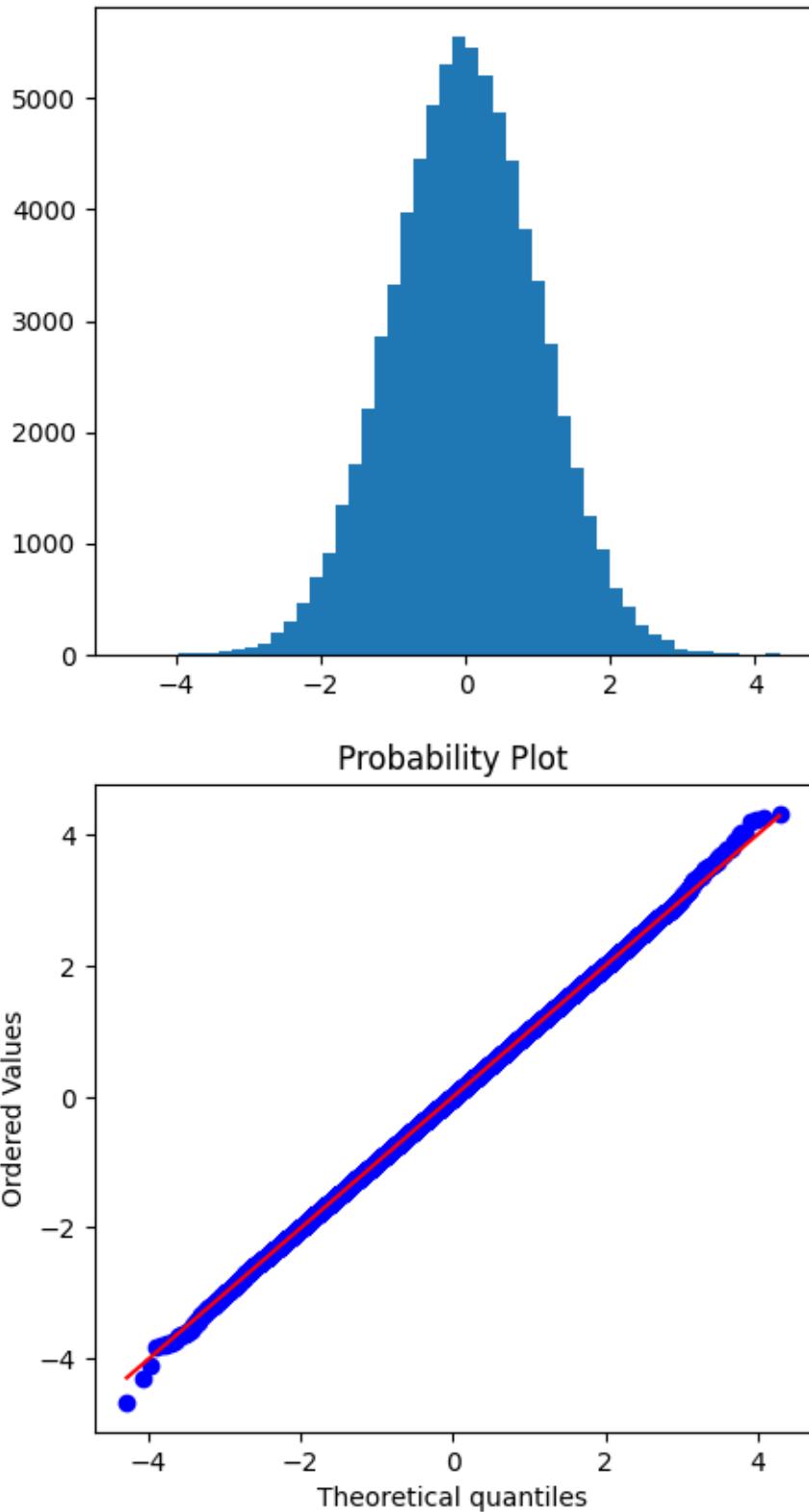
```
#random sampling from Exponential dist.
n = 100000
e = np.random.rand(n)
x = Ginv(e)
plt.hist(x, bins=50)
plt.show()
```



```
#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 = 76149
Mean = -0.0007738332678312738
Std = 1.0026630200413653
Skewness = 0.0014052484620287072
Kurtosis = 0.02832094979234201

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())

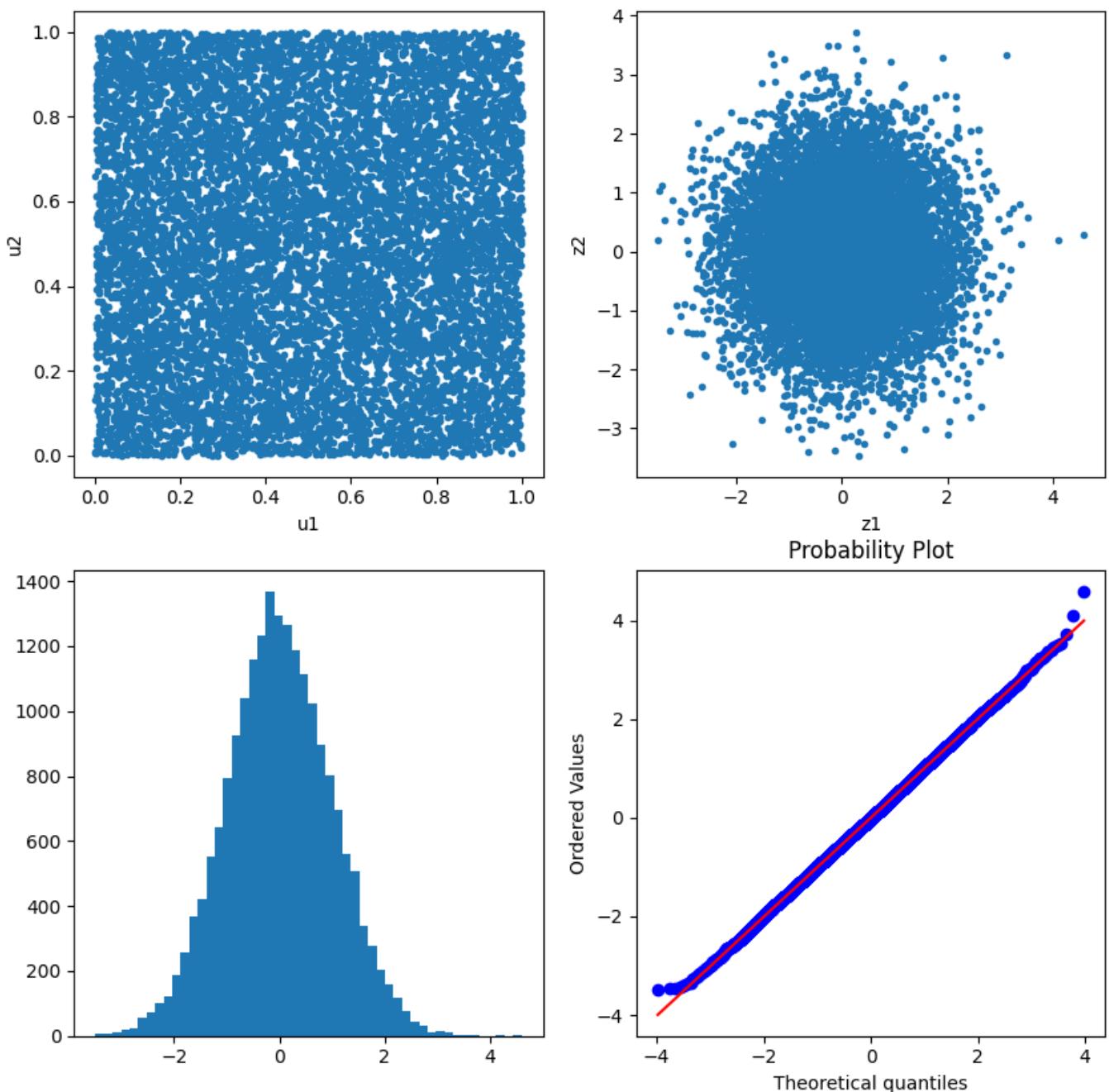

Mean = -0.0005109536692502728

```

```

Std = 1.0033155307284243
Skewness = 0.02486394723725498
Kurtosis = 0.004147895282052172

```



2.9.2 Box-muller method

2.9.2.1 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)

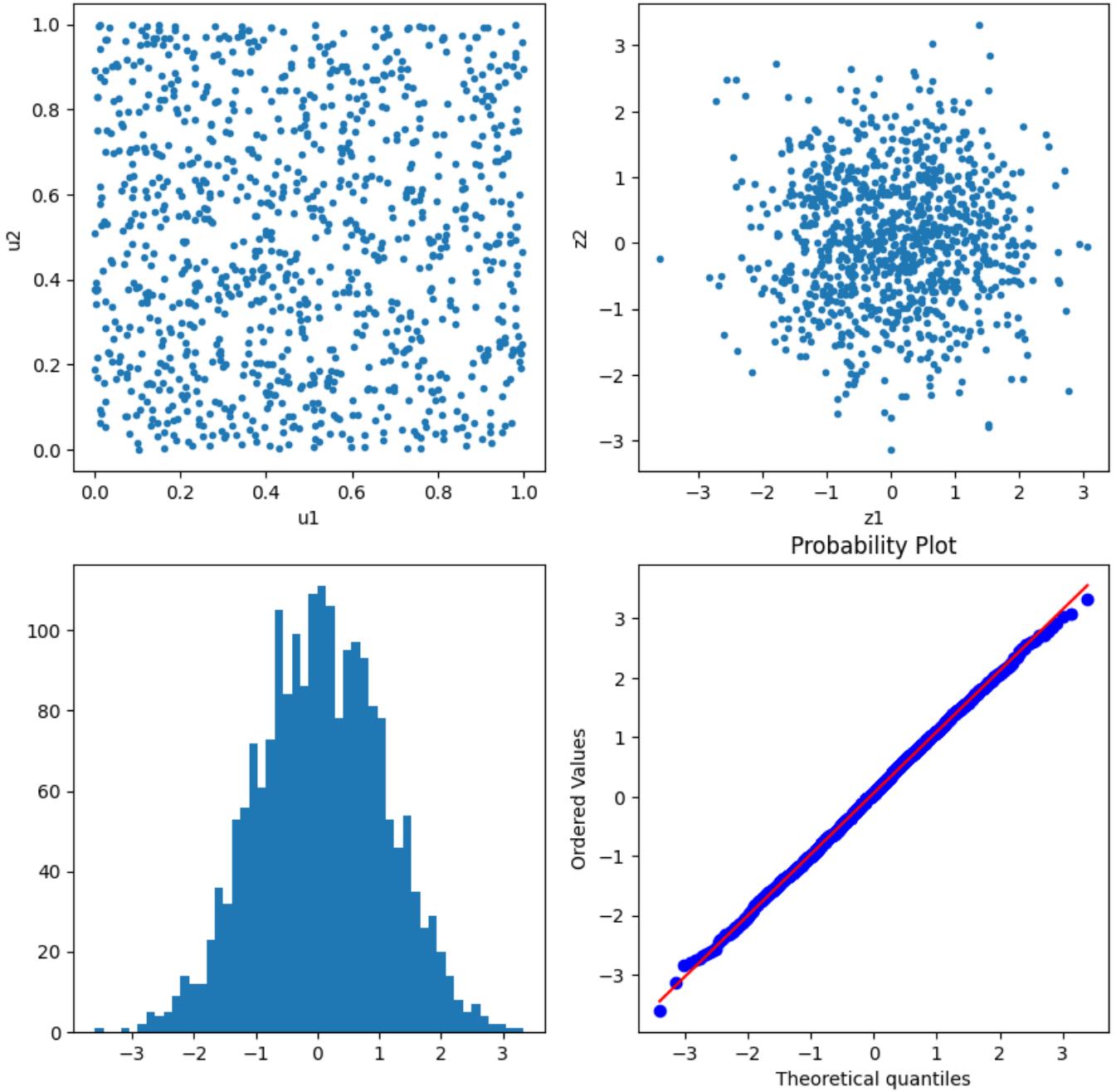
z1 = np.sqrt(-2*np.log(u1))*np.cos(2*np.pi*u2)
z2 = np.sqrt(-2*np.log(u1))*np.sin(2*np.pi*u2)

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([-3.60306231, -3.13548782, -2.83263488, ..., 3.03258201,
       3.06619721, 3.32173166])),
(1.0327708990587872, 0.05466952756732349, 0.9996989636334398))

```



```

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

```

```

Mean =  0.05466952756732329
Std =  1.0317810356699553
Skewness = -0.018633416552330598
Kurtosis = -0.17031449124549036

```

2.9.3 Marsaglia's polar method

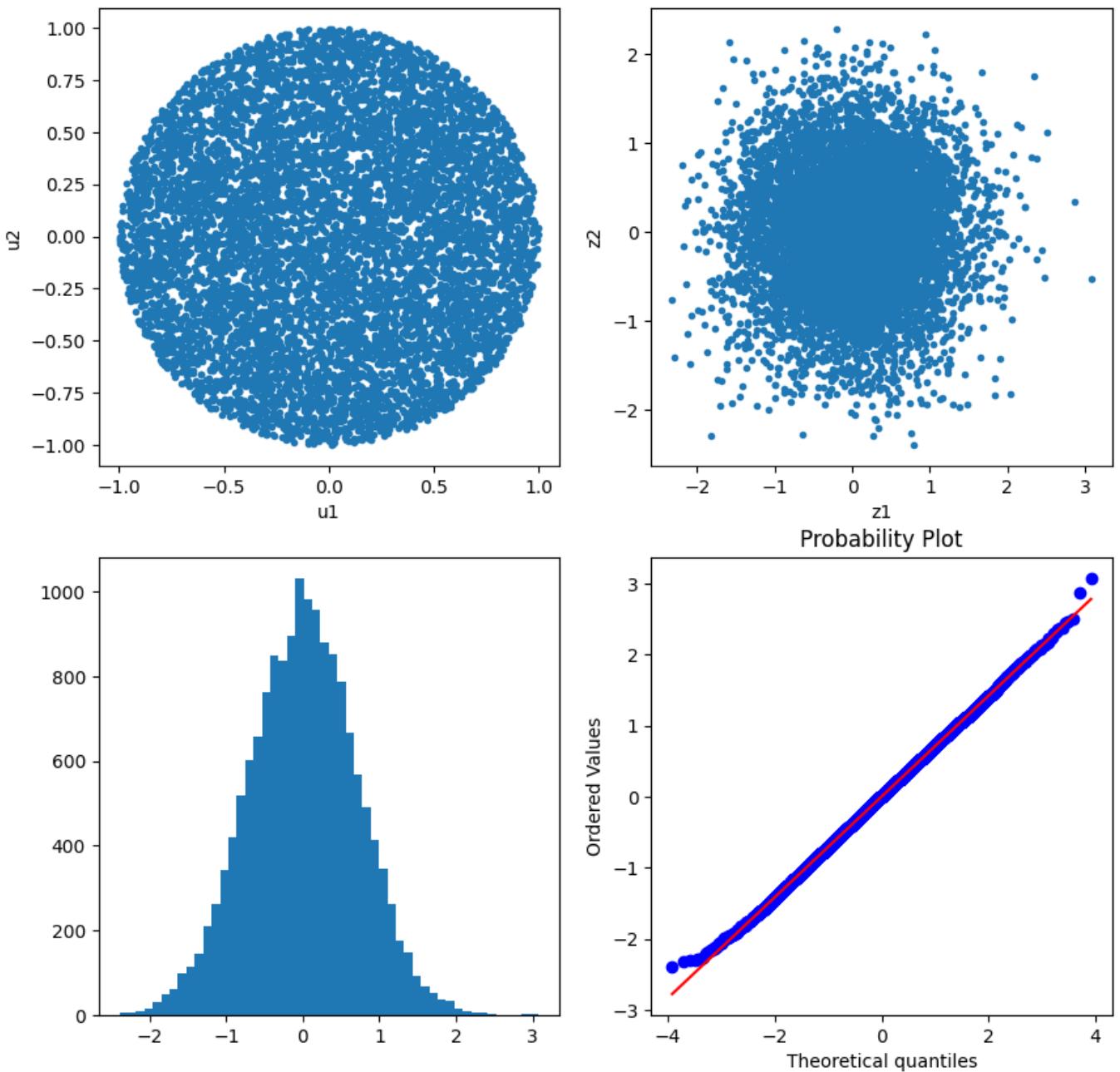
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import scipy.stats as stats

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([-3.92157497, -3.70243821, -3.58239836, ..., 3.58239836,
         3.70243821, 3.92157497]),
array([-2.39692813, -2.31770256, -2.29498998, ..., 2.50125154,
         2.86382852, 3.07531846])),
(0.7077405533780114, 0.0010379840057725947, 0.9999066551363593))
```



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

```
Mean =  0.001037984005772708
Std =  0.7076612420548687
Skewness = -0.007878945852490927
Kurtosis = -0.0333895630592389
```

2.9.4 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)
```

```
[[1.  0.3  0.5]
 [0.3  1.   0.6]
 [0.5  0.6  1. ]]
True
```

```
#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())
```

Mean

```
          z1         z2         z3
mean  0.015803 -0.003859 -0.004027
std   1.000498  1.011745  1.008933
```

Correlation

```
          z1         z2         z3
z1   1.000000  0.308514  0.509939
z2   0.308514  1.000000  0.602482
z3   0.509939  0.602482  1.000000
```

```

#Positive Definite 하지 않은 상관계수 행렬 생성 (3번째 변수를 선형결합으로 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  0.726141  0.460392
1  0.726141  1.000000 -0.276035
2  0.460392 -0.276035  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.          0.98367741  0.17994096]
 [ 0.          0.83800603 -0.54566097]
 [ 0.          0.29314128  0.95606913]]

```

```

[[ 1.          0.72614084  0.46039245]
 [ 0.72614084  1.          -0.27603545]
 [ 0.46039245 -0.27603545  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	-0.009456	-0.017486	0.009358
std	1.004328	1.003282	0.998185

Correlation

	z1	z2	z3
z1	1.000000	0.729317	0.460086
z2	0.729317	1.000000	-0.271913
z3	0.460086	-0.271913	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  0.044162  0.067192 -0.001899
std   1.039992  1.440805  2.231840
      0          1          2
0  1.000000  0.726141  0.460392
1  0.726141  1.000000 -0.276035
2  0.460392 -0.276035  1.000000

```

```

==== simulation data ====

```

Mean

	z1	z2	z3
mean	0.008099	0.014018	-0.003738
std	1.041920	1.451097	2.234800

Correlation

	z1	z2	z3
z1	1.000000	0.727481	0.453941
z2	0.727481	1.000000	-0.281128
z3	0.453941	-0.281128	1.000000

2.10 Ch3. 샘플 경로 생성 방법

2.10.1 Variance reduction

```

import numpy as np
from blackscholes import bsprice

def mcprice_controlvariates(s,k,r,q,t,sigma,nsim,flag):
    z = np.random.randn(nsim)
    st = s*np.exp((r-q-0.5*sigma**2)*t + sigma*np.sqrt(t)*z)
    callOrPut = 1 if flag.lower()=='call' else -1
    payoff = np.maximum(callOrPut*(st-k), 0)
    disc_payoff = np.exp(-r*t)*payoff
    price = disc_payoff.mean()
    se = disc_payoff.std(ddof=1) / np.sqrt(nsim)

    c = np.cov((disc_payoff, st), ddof=1)
    cv_disc_payoff = disc_payoff - c[1,0]/c[1,1]*(st-s*np.exp((r-q)*t))

```

```

cv_price = cv_disc_payoff.mean()
cv_se = cv_disc_payoff.std(ddof=1) / np.sqrt(nsim)

return price, se, cv_price, cv_se


def mcprice_antithetic(s,k,r,q,t,sigma,nsim,flag):
    z = np.random.randn(nsim)
    st = s*np.exp((r-q-0.5*sigma**2)*t + sigma*np.sqrt(t)*z)
    callOrPut = 1 if flag.lower()=='call' else -1
    payoff = np.maximum(callOrPut*(st-k), 0)
    disc_payoff = np.exp(-r*t)*payoff
    price = disc_payoff.mean()
    se = disc_payoff.std(ddof=1) / np.sqrt(nsim)

    z[nsim/2:] = -z[:nsim]
    st = s*np.exp((r-q-0.5*sigma**2)*t + sigma*np.sqrt(t)*z)
    payoff = np.maximum(callOrPut*(st-k), 0)
    disc_payoff = np.exp(-r*t)*payoff
    price2 = disc_payoff.mean()
    se2 = disc_payoff.std(ddof=1) / np.sqrt(nsim)
    return price, se, price2, se2

s, k, r, q, t, sigma = 100, 100, 0.03, 0.01, 0.25, 0.2
flag = 'put'

#Analytic Formula
price = bsprice(s,k,r,q,t,sigma,flag)
print(f"  Price = {price:0.6f}")
print("-"*50)

#Control-Variates Simulation
nsim = 100000
mc_price, se, cv_price, cv_se= mcprice_controlvariates(s,k,r,q,t,sigma,nsim,flag)
print(f"MC Price = {mc_price:0.6f} / se = {se:0.6f}")
print(f"CV Price = {cv_price:0.6f} / se = {cv_se:0.6f}")
print("-"*50)

#Antithetic
mc_price, se, price2, se2= mcprice_antithetic(s,k,r,q,t,sigma,nsim,flag)
print(f"MC Price = {mc_price:0.6f} / se = {se:0.6f}")

```

```
print(f"Antithetic Price = {price2:0.6f} / se = {se2:0.6f}")
print("-"*50)
```

Price = 3.724086

MC Price = 3.742272 / se = 0.016858

CV Price = 3.744590 / se = 0.009456

MC Price = 3.728164 / se = 0.016863

Antithetic Price = 3.728332 / se = 0.009425

시뮬레이션 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 S 이고, GBM 프로세스를 따른다.
 - C. 옵션의 만기가 T (years)이고, 만기까지 베리어 knock 여부를 확인하는 관측시점은 같은 간격으로 m 번 관측한다. ($\Delta t = T/m$)
 - D. 옵션의 베리어는 B , 행사가격은 K 이고, 무위험금리(연속복리) r 과 변동성 σ 는 상수라고 가정 한다. (배당 = 0 으로 가정)
 - E. 시뮬레이션의 replication 회수는 n 번이다.
2. 베리어옵션의 In-Out parity에 대해서 조사하고, 몬테카를로 옵션 평가에서 활용할 수 있는 방법에 대해서 설명하시오.
3. m 과 n 을 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_0 까지 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 하나라도 베리어보다 큰 경우, pay-off 발생
 - Down and In : 어느 기준값 중 하나라도 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

```
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=="I" :
            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 $n*m$ 번의 루프문을 작성하는 대신 n번의 루프문을 작성한 것 입니다. 또한 이러한 방식 외에도, 한번에 nm 개의 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 $n=10000$, $m=250$ 으로 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 : 1.2995675937856517
Difference is -0.14419759397053666

```

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 : 8.272186560888736
Difference is -0.01415317685082762

```

비교결과, 대체로 유사하였으나 오차가 상당수준 발생하였습니다.

Up&Out에서는 MCS의 결과값이 크고 Up&In에서는 Analytic form의 결과값이 큰 경향이 있는데,
이는 이산-오일러 근사를 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 option의 $c_{plain} = f(S, K, T, r, \sigma, d)$ 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 베리어옵션으로 분해할 수 있다는 의미가 됩니다.

증명

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

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

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

O는 lookback period 동안 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 베리어옵션 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.

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

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, 베리어 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 formula를 통한 plain vanilla옵션이 아닌,

베리어옵션과 동일한 이산오일러근사를 사용한 plain vanilla옵션의 가격을 사용해야 합니다.

Answer 3

N, M에 따른 bias와 variance의 변화를 살펴보겠습니다.

이를 살펴보기 위해 시뮬레이션 결과값인 베리어옵션가격. 즉, 표본평균 \bar{y} 의 분포를 이용하겠습니다.

CLT에 따라 베리어옵션가격의 분포는 $\bar{y} \sim N(y_{real}, \frac{\sigma^2}{N})$ 를 따르게 됩니다.

$Bias^2 = (E[\bar{y}] - y_{real})^2$, $Variance = E[(\bar{y} - E[\bar{y}])^2]$ 이므로,

주어진 N, M의 값에 대하여 K(30)번 시뮬레이션을 반복하여 \bar{y}_k 를 얻은 후 이를 계산해보겠습니다.

y_{real} 은 QuantLib의 결과값이라고 가정하고, Up&Out call옵션을 예시로 살펴보았습니다.

N값이 충분히 크다면 \bar{y} 의 분산이 충분히 작아지므로, 한번의 시뮬레이션으로 이 근사값을 계산할 수 있습니다.

$$Bias^2 \approx (\bar{y} - y_{real})^2$$

$$Variance = Var[\bar{y}] = \frac{\sigma^2}{N} \approx \frac{s^2}{N} = S.E^2$$

```
import time
y_real = QL_UOCprice

N = 1000; M = 250
y_k, se_k = np.zeros(30), np.zeros(30)

start = time.time()
```

```

for i in range(30):
    y_mean, y = BarrierOptionsPrice(S, K, T, B, r, vol, "U", "O", "C", n=N, m=M)
    se = y.std(ddof = 1) / np.sqrt(N)
    y_k[i], se_k[i] = y_mean, se
end = time.time()

bias = (y_k.mean()-y_real)**2
variance = y_k.var(ddof = 1)
stan_error = se_k.mean()**2
cal_time = (end-start)/30

print("When N :",N," and M :",M,"\n",
      "Bias^2 :",bias,"\n",
      "Variance :", variance, "\n",
      "S.E^2 :", stan_error, "\n",
      "MSE :", bias+variance, "\n",
      "Time for 1 simulation:", cal_time, "second", "\n")

```

When N : 1000 and M : 250
 Bias² : 0.024272389654162986
 Variance : 0.012170768897249329
 S.E² : 0.011545553409607707
 MSE : 0.036443158551412315
 Time for 1 simulation: 0.019399333000183105 second

N을 증가시킬수록 시뮬레이션의 분산은 감소하나 편의는 감소하지 않는 경향이 있습니다.

When N : 3000 and M : 250
 Bias² : 0.0184339108482227
 Variance : 0.003476728649605832
 S.E² : 0.0038095509624510564
 MSE : 0.021910639497828534
 Time for 1 simulation: 0.06197342872619629 second

When N : 5000 and M : 250
 Bias² : 0.013752890210891254
 Variance : 0.0031118524015823104
 S.E² : 0.0022555294874679666
 MSE : 0.016864742612473563
 Time for 1 simulation: 0.0992981990178426 second

```
When N : 10000 and M : 250
Bias^2 : 0.020888138546843835
Variance : 0.001117408653347828
S.E^2 : 0.0011493753656563702
MSE : 0.022005547200191662
Time for 1 simulation: 0.19343246618906657 second
```

M을 증가시킬수록 편의는 감소하나 분산은 유사한 경향이 있습니다.

```
When N : 3000 and M : 100
Bias^2 : 0.0515679736427854
Variance : 0.00582905751382059
S.E^2 : 0.004108939081535306
MSE : 0.057397031156605986
Time for 1 simulation: 0.044307708740234375 second
```

```
When N : 3000 and M : 250
Bias^2 : 0.018928611152255404
Variance : 0.0032641069747157917
S.E^2 : 0.0038083902742757176
MSE : 0.022192718126971198
Time for 1 simulation: 0.05864783128102621 second
```

```
When N : 3000 and M : 500
Bias^2 : 0.013075189332203788
Variance : 0.003022934661511448
S.E^2 : 0.0037254695467228006
MSE : 0.016098123993715237
Time for 1 simulation: 0.08189740180969238 second
```

```
When N : 3000 and M : 1000
Bias^2 : 0.002370847294571083
Variance : 0.0030987301130841506
S.E^2 : 0.003496549580377565
MSE : 0.005469577407655234
Time for 1 simulation: 0.12750016848246257 second
```

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

시뮬레이션 과제1 수정 (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 S 이고, GBM 프로세스를 따른다.
 - C. 옵션의 만기가 T (years)이고, 만기까지 베리어 knock 여부를 확인하는 관측시점은 같은 간격으로 m 번 관측한다. ($\Delta t = T/m$)
 - D. 옵션의 베리어는 B , 행사가격은 K 이고, 무위험금리(연속복리) r 과 변동성 σ 는 상수라고 가정 한다. (배당 = 0 으로 가정)
 - E. 시뮬레이션의 replication 회수는 n 번이다.
2. 베리어옵션의 In-Out parity에 대해서 조사하고, 몬테카를로 옵션 평가에서 활용할 수 있는 방법에 대해서 설명하시오.
3. m 과 n 을 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 경로 n개를 이산오일러 근사를 통해 구성
2. 하나의 경로는 m개의 관측기준점에 따라 나누어지며, 초기값 S_0 를 제외한 m개로 이루어짐. (전체 $n*m$ matrix)
3. 각각의 경로에 대하여, 기초자산 기준값과 베리어를 비교하여 옵션 pay-off 발생 여부 판단
 - Up and Out : 기준값의 최대값이 베리어보다 작거나 같은 경우, pay-off 발생
 - Up and In : 기준값의 최대값이 베리어보다 큰 경우, pay-off 발생
 - Down and Out : 기준값의 최소값이 베리어보다 크거나 같은 경우, pay-off 발생
 - Down and In : 기준값의 최소값이 베리어보다 작은 경우, pay-off 발생
4. pay-off가 없으면 옵션 가치는 0, 있으면 Call/Put 종류에 따라 pay-off를 계산하고, 그 현재가치가 하나의 경로의 옵션의 가치
5. n개의 경로에 대해 옵션의 가치를 모두 산출하고, 산술평균하여 최종적으로 베리어옵션의 가격(및 Standard error) 산출

이에 따른 Python 코드는 아래와 같습니다.

Python 구현

```
import numpy as np

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
    """
    dt = t/m
    z = np.random.standard_normal( m*n ).reshape(n,m)
    underlying_path = s*np.exp((r-0.5*(std**2))*dt+std*np.sqrt(dt)*z).cumprod(axis=1)
    if UpDown=="U" and InOut=="O" :
        payoff_logic = underlying_path.max(axis=1)<=b
    elif UpDown=="U" and InOut=="I" :
        payoff_logic = underlying_path.max(axis=1)>b
    elif UpDown=="D" and InOut=="O" :
        payoff_logic = underlying_path.min(axis=1)>=b
    elif UpDown=="D" and InOut=="I" :
        payoff_logic = underlying_path.min(axis=1)<b

    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 = payoff_logic * plain_price
    barrier_price = barrier_simulation.mean()
    barrier_se = barrier_simulation.std(ddof = 1) / np.sqrt(n)
    return barrier_price, barrier_se
```

Analytic Solution과 비교

해당 코드를 이용하여 베리어옵션 가격을 추정할 수 있으며, 이를 예제(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_UOCse = 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 : 1.2965456033951188

Difference is -0.1411756035800038

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 : 8.192630511804628  
Difference is 0.06540287223328001
```

비교결과, 대체로 유사하였으나 오차가 상당수준 발생하였습니다.

Up&Out에서는 MCS의 결과값이 크고 Up&In에서는 Analytic form의 결과값이 큰 경향이 있는데,
이는 이산-오일러 근사를 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($m=250$ 은 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 Knock 할 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 베리어옵션으로 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 period 동안 기초자산의 가격이 B를 한번이라도 Knock하는 경우 payoff가 발생합니다. (Up & Down 포괄)

O는 lookback period 동안 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.

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

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 formula를 통한 plain vanilla옵션이 아닌,

베리어옵션과 동일한 이산오일러근사를 사용한 plain vanilla옵션의 가격을 사용해야 합니다.

Answer 3

N, M에 따른 bias와 variance의 변화를 살펴보겠습니다.

이를 살펴보기 위해 시뮬레이션 결과값인 베리어옵션가격. 즉, 표본평균 \bar{y} 의 분포를 이용하면 됩니다.

CLT에 따라 베리어옵션가격의 분포는 $\bar{y} \sim N(y_{real}, \frac{\sigma^2}{N})$ 를 따르게 되므로,

$Bias^2 = (E[\bar{y}] - y_{real})^2$, $Variance = E[(\bar{y} - E[\bar{y}])^2]$ 가 됩니다.

이제, 주어진 N, M의 값에 대하여 K번 시뮬레이션을 반복하여 이를 계산하면 되며,

계산의 효율화를 위해 편의와 분산은 아래와 같이 근사값으로 계산하여도 무방합니다.

$Bias^2 \approx (\bar{y} - y_{real})^2$

$Variance = Var[\bar{y}] = \frac{\sigma^2}{N} \approx \frac{s^2}{N} = S.E^2$

y_{real} 은 QuantLib의 결과값이라고 가정하고, Up&Out call옵션을 예시로 위의 결과를 살펴보겠습니다.

```
import time
import pandas as pd
y_real = QL_UOCprice
M = 50

Ns, bias, var, cal = np.zeros(10), np.zeros(10), np.zeros(10), np.zeros(10)

for i in range(10):
    start = time.time()
    N = (i+1)*1000
    y, y_se = BarrierOptionsPrice(S, K, T, B, r, vol, "U", "O", "C", n=N, m=M)
    Ns[i] = N
    bias[i] = (y-y_real)**2
    var[i] = y_se**2
    end = time.time()
    cal[i] = end-start

result = pd.DataFrame({"N":Ns,"M":M,'Bias^2':bias,'Variance':var,'MSE':bias+var,"time":cal})
result
```

	N	M	Bias ²	Variance	MSE	time
0	1000.0	50	0.233815	0.014036	0.247851	0.001814
1	2000.0	50	0.064735	0.006439	0.071175	0.003320
2	3000.0	50	0.042222	0.004161	0.046383	0.005007
3	4000.0	50	0.128648	0.003471	0.132119	0.006655
4	5000.0	50	0.152907	0.002813	0.155720	0.008116
5	6000.0	50	0.118306	0.002321	0.120627	0.010099
6	7000.0	50	0.139482	0.001986	0.141467	0.011514
7	8000.0	50	0.090082	0.001676	0.091758	0.013284
8	9000.0	50	0.093373	0.001467	0.094841	0.014840
9	10000.0	50	0.131647	0.001382	0.133030	0.016739

N을 증가시킬수록 시뮬레이션의 분산은 감소하며, 편의의 증감추세는 관측되지 않았습니다.

N = 1000

```
Ms, bias, var, cal = np.zeros(10), np.zeros(10), np.zeros(10), np.zeros(10)

for i in range(10):
    start = time.time()
    M = (i+1)*50
    y, y_se = BarrierOptionsPrice(S, K, T, B, r, vol, "U", "0", "C", n=N, m=M)
    Ms[i] = M
    bias[i] = (y-y_real)**2
    var[i] = y_se**2
    end = time.time()
    cal[i] = end-start

result = pd.DataFrame({"N":N,"M":Ms,'Bias^2':bias,'Variance':var,'MSE':bias+var,"time":cal})
```

	N	M	Bias ²	Variance	MSE	time
0	1000	50.0	0.028666	0.011598	0.040265	0.001948
1	1000	100.0	0.049862	0.012735	0.062597	0.003508
2	1000	150.0	0.004199	0.009074	0.013273	0.005504
3	1000	200.0	0.148563	0.014316	0.162879	0.006588
4	1000	250.0	0.000185	0.010087	0.010272	0.008092

	N	M	Bias^2	Variance	MSE	time
5	1000	300.0	0.019473	0.011303	0.030776	0.009590
6	1000	350.0	0.000539	0.010266	0.010805	0.011320
7	1000	400.0	0.021450	0.012609	0.034060	0.014259
8	1000	450.0	0.017826	0.011640	0.029466	0.015960
9	1000	500.0	0.079944	0.012243	0.092186	0.017616

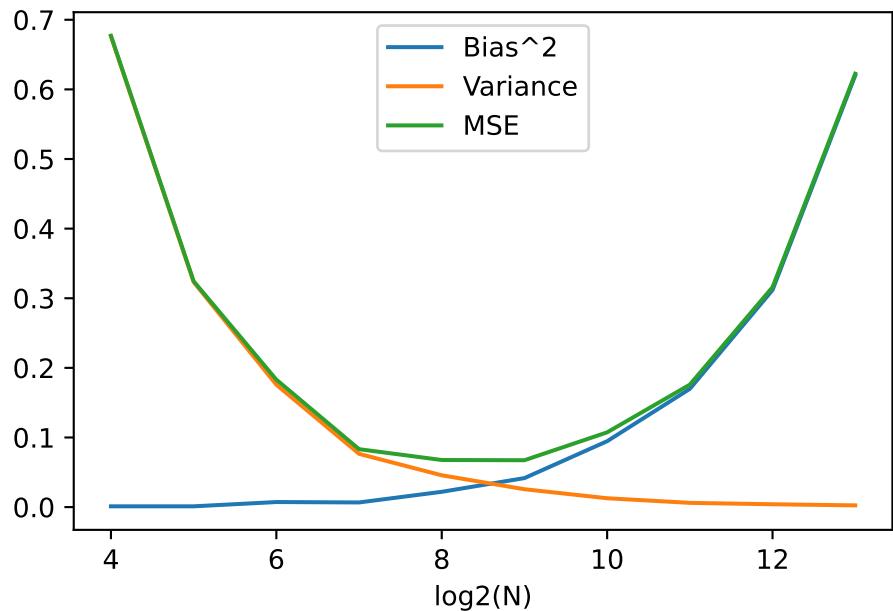
M을 증가시킬수록 시뮬레이션의 편의는 감소하며, 분산은 일정하게 유지되는 추세를 보입니다.

```
# 산출 신뢰도 향상을 위해 각 N,M 별로 시뮬레이션 L번 반복 예정
L = 200
Ns, Ms, bias, var = np.zeros(10), np.zeros(10), np.zeros(10), np.zeros(10)
# 예산제약, N*M = tau, 계산시간이 N, M과 정비례한다고 가정
tau = 2**16

for i in range(10):
    N = 2**(i+4)
    M = int(np.round(tau / N, 0))
    y = []
    for j in range(L):
        tmp1, tmp2 = BarrierOptionsPrice(S, K, T, B, r, vol, "U", "0", "C", n=N, m=M)
        y.append(tmp1)
    Ns[i], Ms[i] = N, M
    bias[i] = (np.mean(y) - y_real)**2
    var[i] = np.var(y, ddof = 1)

result = pd.DataFrame({"N":Ns,
                      "log2(N)":np.log2(Ns),
                      "M":Ms,
                      'Bias^2':bias,
                      'Variance':var,
                      'MSE':bias+var})

result.plot(x='log2(N)',y=['Bias^2','Variance','MSE'])
```



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

Part IV

이자율파생상품('24 가을)

0|자율파생상품 과제1

20249132 김형환

Question 1

- (ㄱ) 은행은 아래와 같은 3년 만기 채권을 발행하여 자금을 조달하려고 하고 동시에 이자율스왑거래를 통해 이를 변동금리로 변화시키고자 한다. 그리고 시장엔 USD에 대한 3년 이자율스왑금리가 3.341%로 주어져 있다. 스왑은 6개월마다 (ㄱ)은행에 고정금리로 지급하고 은행은 스왑딜러에게 변동금리를 지급하는 거래이다. 은행이 스왑딜러와 거래할 때 액면 200 million USD를 이용한다면 매년 이자비용을 맞추기 위하여 은행은 변동금리 + x 를 지급하고, 채권에 대한 이자비용을 받는다. 이 때 x 는 얼마인가? (여기에서 스왑금리를 annual로 바꾸어 계산하시오. 그리고 은행, 딜러의 신용위험은 고려하지 않는다.)

액면	200 million USD
만기	3년
금리	4% p.a. (annual)
발행가	99.659
수수료	20bp
기타비용	USD 75,000

Answer 1 : 0.631%

(ㄱ)은행은 문제의 채권발행으로 인해 향후 3년간 연 4%의 고정금리 이자를 지급해야하는 상황이며,

이 고정금리 이자지급을 변동금리로 바꾸는 것이 목표입니다.

스왑딜러와 IRS를 체결함으로써 이를 해결할 수 있고, (ㄱ)은행은 4% fixed receiver가 됩니다.

4%만큼 fixed rate를 지급하고, (변동금리+ x)만큼 Float rate를 지급하게 되는데, 계약체결시점에서 (ㄱ)은행의 NPV가 0이 되도록 하는 스프레드(x)를 산출하면 됩니다.

한편, swap rate를 이용하여 IRS를 체결하면 체결시점의 NPV는 0입니다. (swap rate vs. float rate)

따라서, 4%-annual swap rate를 통해 간단히 스프레드 x 를 계산할 수 있습니다.

6개월 스왑금리를 annual로 바꾸면 $(1 + \frac{0.03341}{2})^2 = 1 + r$, $r = 0.03369$ 이므로, $x = 0.03369 - 0.04 = -0.00631$ 입니다.

즉, (ㄱ)은행과 스왑딜러가 (4% vs. float rate + 0.631%)의 IRS를 체결하면 됩니다..

Question 2

2. (ㄴ) 은행은 스왑딜러로서 통화스왑과 금리스왑을 거래한다. 특히 통화관련해서 베이시스 통화스왑을 거래하는데, 현재 EUR/USD 호가는 -4/+1로 주어져 있다. 이는 USD SOFR를 수취하면 EUR ESTR -4bp를 지급하고, USD SOFR를 지급하면 EUR ESTR +1bp를 지급한다는 의미이다. EUR 금리스왑은 ESTR를 수취하면 EUR 5.05% 고정금리를 지급하도록 결정되어 있다. 어떤 고객이 USD SOFR를 수취하고 EUR 고정금리를 이 은행에 지불하는 거래를 하고자 한다. 이 때 고정금리는 얼마가 되어야 하는가? (이 거래를 해지하기 위하여 A 사와는 베이시스 통화스왑을, B 사와는 EUR 금리스왑을 진행한다고 가정하시오.)

Answer 2 : 최소 5.01%

고객은 은행으로부터 USD SOFR를 받고, EUR fixed를 지급하므로,

은행은 EUR fixed를 받고, USD SOFR를 지급해야합니다.

따라서, 은행은 고객과의 계약해지를 위해 USD SOFR를 받고, EUR fixed를 지급하는 cashflow를 만들어야 합니다.

이를 위해 두가지 계약을 사용합니다.

1. EUR 금리스왑, fixed payer
2. 베이시스 통화스왑 EUR ESTR payer / USD SOFR receiver

1번 계약에서 은행은 5.05% 고정금리를 지급하고 EUR ESTR을 수취합니다.

2번 계약에서 은행은 EUR ESTR - 4bp를 지급하고, USD SOFR를 수취합니다.

두 계약을 결합하면, 은행은 5.01% 고정금리를 지급하고 USD SOFR를 수취하게 됩니다.

따라서, 고객과의 계약에서 은행은 최소 5.01%의 EUR fixed를 받아야합니다.

Question 3

3. 현재는 2024년 6월이다. 한 트레이더가 2024년 9월 3개월 SOFR 선물을 가지고 있다고 하자. 이 선물에 대한 SOFR 해당 기간은 9-12월이다. 이 선물가에 영향을 주는 것은 해당 기간 시작일 9.16일의 SOFR 금리와 11.6-7일 FOMC 미팅에 따른 target rate 변화라고만 가정하자. 즉 initial level 과 중간의 jump size 에 의하여 선물가가 영향을 받는다. 이 트레이더는 1개월 10월 SOFR 선물과 1개월 11월 SOFR 선물을 이용해서 해지를 하고자 한다. 어떤 방법이 있을지 자유롭게 써보시오. (빈 칸만 아니면 full score 부여함)

Answer 3

트레이더는 현재 두가지 위험에 직면해있습니다.

1. 현재(6월)부터 9.16일까지의 금리하락으로 initial level SOFR가 하락할 위험
2. 11.6-7 FOMC 결과에 따라 target rate SOFR가 하락할 위험

이 두가지 위험을 주어진 hedging instrument를 통해 관리하려면, 1개월 10월 SOFR 선물을 매도하면 됩니다.

1개월 10월 SOFR 선물은 10.16일 initial level과 11월 FOMC target rate에 영향을 받는 상품입니다.

문제의 가정처럼 오직 두가지의 요소에 의해 선물가격이 영향을 받는다면, 9.16~11.6까지 및 11.7~12월 만기까지의 SOFR의 변화가 없다는 의미입니다.

따라서 현재시점에 1개월 10월 SOFR 선물을 매도한다면, 9.16일 initial level의 변동위험은 10.16일 initial level 변동 위험과 정확하게 상쇄되며 11월 FOMC target rate 리스크도 정확히 상쇄될 것 입니다.

즉, 9.16부터 10.16까지 SOFR의 변동이 없고, 11월 선물만기일 이후 12월 선물만기일까지 SOFR의 변동이 없다면 1개월 10월 선물은 완전헷지가 가능한 hedge instrument가 됩니다.

한편, 1개월 11월 SOFR 선물의 경우 11.16일 initial level에 영향을 받으며 이후 FOMC가 없으므로 오직 initial level에 따라서 가격이 결정되는 상품이 됩니다.

하지만 11월 initial level은 현재시점부터 9월까지의 initial level 변동과 그 이후의 금리변동, 11월 FOMC target rate 변동이 모두 반영되어있는 금리입니다.

따라서 간접적으로 트레이더의 위험을 모두 내포하고 있는 상품이며, 이를 현재시점에 매도한다면 헷지가 가능하게 됩니다.

다만, 1개월 11월 선물을 이용하는 경우, 기존 포지션의 모든 가격변동분이 11.16일에 일시반영되므로 그 사이에 FOMC 급락으로 일일정산 손실이 누적되는 등 유동성 위험에 직면할 수 있습니다.

또한, 실제 시장에서는 SOFR금리가 일일단위로 변동하며 3개월 선물과 1개월 선물의 compounding 기간이 달라 이러한 완전헷지는 불가능할 것으로 보입니다.

Question 4

4. 다음과 같은 이항트리를 생각하자. 이 트리는 물리적 측도, 그리고 연속복리법에 의하여 구성되었다. 상승, 하락 확률은 50% 으로 동일하다. 그리고 1년 만기 액면가 100의 무이표 국채의 시장가는 $P_0(2) = 97.4845$ 로 주어져 있다. market price of risk λ 를 구하시오.

period	$i = 0$	$i = 1$
time (in years)	$t = 0$	$t = 0.5$
	$r_0 = .02$	$r_{1,0} = 0.04$
		$r_{1,1} = .01$

Answer 4 : -0.1980

tree

period	i=0	i=1
time	t=0	t=.5
		4.00%
	2.00%	
		1.00%

bond prices

period	i=0	i=1	i=2
time	t=0	t=.5	t=1
		100	
		98.0199	
market	97.7779	99.5012	100
diff	97.485	-1.4814	100
lambda	0.2934	-1.4814	100
	-0.1980		

Market price of risk $\lambda = \frac{e^{-r_0 t} E[P_1(2)] - P_0(2)}{P_{1,0} - P_{1,1}}$ 입니다.

위 그림처럼 이를 각각 계산하면, $\lambda = -0.1980$ 입니다.

Question 5

5. 위의 이항트리를 이용하여 6개월 후 만기가 도래하는 채권 콜옵션의 가격을 구하시오. 이 때 기초자산은 1년 만기 액면가 100의 무이표 국채이며 행사가는 98.5이다.

Answer 5 : 0.2974

bond option

maturity	1
strike	98.5
period	i=0
time	t=0
	i=1
	t=.5
period	i=0
time	t=0
	i=1
	t=.5
	0.0000
0.2974	0.0000
replication	1.0012
	0.2974
risk-neutral pricing	1.0012
a	0.675888
b	-0.6625
p	0.700031

포트폴리오 복제 및 위험중립가치평가를 통해 옵션가격을 계산할 수 있습니다.

위험중립가치평가를 이용할 때, 위험중립확률은 $p^* = \frac{e^{r_0/2} P_0(2) - P_{1,1}}{P_{1,0} - P_{1,1}} = 0.7000$ 으로,

$$c = e^{-r_0 t} (p^*(P_{1,0} - K)^+ + (1 - p^*)(P_{1,1} - K)^+) = 0.2974$$

이자율파생상품 과제2

20249132 김형환

Question 1

- 엑셀파일 swap_rate.xlsx 에는 만기 0.5년부터 만기 5년까지 매분기별로 지급하는 이자율스왑의 swap rate 이 주어져 있다. 이를 이용하여 분기별 discount factor 들을 계산하고 Ho-Lee 모형 기반 이항트리(연속복리법)를 완성하시오. 이 때 $i = 0$ 에서 short rate 은 2.967% 라 가정하고 $\sigma = 1.68\%$ 이라 놓으시오. 또한 θ_i 들의 초기값을 모두 0.001 로 놓은 후 calibration 을 진행하시오.

Answer

1. 주어진 swap rate를 이용하여 discount factor를 계산(초록색)
 2. short rate, vol, theta=0.001인 상황에서 Ho-lee 모형의 tree를 전개

3. state price를 통해 Ho-lee 모형의 zero coupon bond price를 산출
4. discount factor와 이를 비교하여 SSE를 산출하고, 이를 최소화시키는 theta의 조합을 추정

Question 2-3

2. 위에서 구성한 Ho-Lee 이항트리에 대하여 2년 만기인 cap의 가격을 구하시오. 이 때 행사가는 해당 만기의 swap rate이고, caplet은 분기별로 존재한다. 그리고 notional =100 이다.
3. 위에서 구성한 Ho-Lee 이항트리에 대하여, 2-by-3 payer swaption (European)의 가격을 구하려고 한다. 계약의 액면은 100이고 기초자산은 앞으로 5년 후에 만기가 도래하는 fixed payer swap이다. 고정금리는 연 3.5%으로 주어져 있고 분기별 지급한다.

Answer

2-Year cap value tree										swap									
T	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
P _{t,k}	0.8932	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
notional	100	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
R	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
7	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
8	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
9	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
13	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
14	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
15	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
16	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
17	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
18	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
19	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
5-Year Swap																			
T	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
P _{t,k}	0.8932	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
notional	100	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
R	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
7	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
8	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
9	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
13	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
14	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
15	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
16	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
17	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
18	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
19	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2-by-3 Payer Swaption																			
T	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
P _{t,k}	0.8932	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
notional	100	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
R	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
7	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
8	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
9	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
13	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
14	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
15	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
16	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
17	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
18	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
19	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Cap의 가격은 1.2115이며, 2X3 swaption의 가격은 4.4513.

Question 4

4. 위에서 주어진 트리 대신 simple BDT 모형에 적용하시오. 특히 2년 cap 가격이 0.8932 이라 할 때 이를 재생산하는 σ 를 구하시오. 초기 θ_i 값들은 0.01로 놓고 $\sigma = 0.0168$ 로 놓고 calibration

Answer

문제1과 유사한 방식으로 calibration을 진행하였으며,

최적해 추정시 2년 cap가격까지 이용하여 전체 SSE(discount factors + 2-y cap)를 구한 다음

theta와 sigma를 대상으로 최적화를 1회 실시하였음.

Question 5

5. 위에서 calibrate 한 BDT 모형에 대하여 수의상환채의 가격을 구하고자 한다. 이 채권은 5년 만기이고 4% 표면금리를 가지고 있다. 이자는 분기별로 지급한다. 1년 지난 후부터 채권발행사가 액면가 100 으로 상환가능하다고 가정하시오.

Answer

tree for bond prices (non-callable)																																						
time	0	0.25	0.5	0.75	1	1.25	1.5	1.75	2	2.25	2.5	2.75	3	3.25	3.5	3.75	4	4.25	4.5	4.75	5																	
1 / 1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20																	
0	-	100.00	99.93	99.45	98.48	91.99	79.88	69.16	60.02	59.09	58.18	44.12	36.85	30.87	29.59	28.37	24.79	24.00	23.47	21.29	19.99																	
1	-	100.00	99.96	99.52	98.44	91.95	79.84	69.12	60.08	59.15	58.23	44.15	36.88	30.90	29.62	28.34	24.75	24.07	23.50	21.32	19.90																	
2	-	-	100.00	99.99	99.56	98.47	97.11	83.85	74.82	69.23	68.32	59.25	44.21	36.93	31.04	29.75	28.47	24.88	24.10	22.85	20.50																	
3	-	-	-	100.00	99.99	99.56	99.34	98.55	97.15	93.15	92.03	85.29	79.91	76.36	74.45	71.56	71.16	72.44	73.93	64.18	59.78																	
4	-	-	-	-	100.00	99.99	99.56	99.34	98.55	97.15	93.15	92.03	85.29	79.91	76.36	74.45	71.56	71.16	72.44	73.93	64.18	59.78																
5	-	-	-	-	-	100.00	99.99	99.56	99.34	98.55	97.15	93.15	92.03	85.29	79.91	76.36	74.45	71.56	71.16	72.44	73.93	64.18	59.78															
6	-	-	-	-	-	-	100.00	99.99	99.56	99.34	98.55	97.15	93.15	92.03	85.29	79.91	76.36	74.45	71.56	71.16	72.44	73.93	64.18	59.78														
7	-	-	-	-	-	-	-	100.00	99.99	99.56	99.34	98.55	97.15	93.15	92.03	85.29	79.91	76.36	74.45	71.56	71.16	72.44	73.93	64.18	59.78													
8	-	-	-	-	-	-	-	-	100.00	99.99	99.56	99.34	98.55	97.15	93.15	92.03	85.29	79.91	76.36	74.45	71.56	71.16	72.44	73.93	64.18	59.78												
9	-	-	-	-	-	-	-	-	-	100.00	99.99	99.56	99.34	98.55	97.15	93.15	92.03	85.29	79.91	76.36	74.45	71.56	71.16	72.44	73.93	64.18	59.78											
10	-	-	-	-	-	-	-	-	-	-	100.00	99.99	99.56	99.34	98.55	97.15	93.15	92.03	85.29	79.91	76.36	74.45	71.56	71.16	72.44	73.93	64.18	59.78										
11	-	-	-	-	-	-	-	-	-	-	-	100.00	99.99	99.56	99.34	98.55	97.15	93.15	92.03	85.29	79.91	76.36	74.45	71.56	71.16	72.44	73.93	64.18	59.78									
12	-	-	-	-	-	-	-	-	-	-	-	-	100.00	99.99	99.56	99.34	98.55	97.15	93.15	92.03	85.29	79.91	76.36	74.45	71.56	71.16	72.44	73.93	64.18	59.78								
13	-	-	-	-	-	-	-	-	-	-	-	-	-	100.00	99.99	99.56	99.34	98.55	97.15	93.15	92.03	85.29	79.91	76.36	74.45	71.56	71.16	72.44	73.93	64.18	59.78							
14	-	-	-	-	-	-	-	-	-	-	-	-	-	-	100.00	99.99	99.56	99.34	98.55	97.15	93.15	92.03	85.29	79.91	76.36	74.45	71.56	71.16	72.44	73.93	64.18	59.78						
15	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	100.00	99.99	99.56	99.34	98.55	97.15	93.15	92.03	85.29	79.91	76.36	74.45	71.56	71.16	72.44	73.93	64.18	59.78					
16	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	100.00	99.99	99.56	99.34	98.55	97.15	93.15	92.03	85.29	79.91	76.36	74.45	71.56	71.16	72.44	73.93	64.18	59.78				
17	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	100.00	99.99	99.56	99.34	98.55	97.15	93.15	92.03	85.29	79.91	76.36	74.45	71.56	71.16	72.44	73.93	64.18	59.78			
18	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	100.00	99.99	99.56	99.34	98.55	97.15	93.15	92.03	85.29	79.91	76.36	74.45	71.56	71.16	72.44	73.93	64.18	59.78		
19	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	100.00	99.99	99.56	99.34	98.55	97.15	93.15	92.03	85.29	79.91	76.36	74.45	71.56	71.16	72.44	73.93	64.18	59.78	
20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	100.00	99.99	99.56	99.34	98.55	97.15	93.15	92.03	85.29	79.91	76.36	74.45	71.56	71.16	72.44	73.93	64.18	59.78
call option prices		exercise value vs. indicated induction value																																				
time	0	0.25	0.5	0.75	1	1.25	1.5	1.75	2	2.25	2.5	2.75	3	3.25	3.5	3.75	4	4.25	4.5	4.75	5																	
1 / 1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20																	
0	-	1.00	1.13	1.30	1.50	1.72	1.96	2.20	2.46	2.74	3.04	3.36	3.69	4.03	4.38	4.74	5.11	5.49	5.88	6.28	6.69																	
1	-	4.18	2.82	1.68	0.91	0.46	0.23	0.12	0.06	0.03	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00																	
2	-	-	6.56	4.88	3.48	2.28	1.32	0.68	0.34	0.17	0.08	0.04	0.02	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00																	
3	-	-	-	7.18	5.50	3.82	2.08	1.11	0.55	0.27	0.13	0.06	0.03	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00																	
4	-	-	-	-	6.97	5.72	3.91	2.11	1.16	0.58	0.31	0.15	0.07	0.03	0.01	0.00	0.00	0.00	0.00	0.00	0.00																	
5	-	-	-	-	-	19.20	8.41	4.21	2.11	1.05	0.52	0.26	0.12	0.05	0.02	0.01	0.00	0.00	0.00	0.00	0.00																	
6	-	-	-	-	-	-	26.46	8.89	4.17	2.11	1.05	0.52	0.26	0.12	0.05	0.02	0.01	0.00	0.00	0.00																		
7	-	-	-	-	-	-	-	19.09	8.95	4.28	2.11	1.05	0.52	0.26	0.12	0.05	0.02	0.01	0.00	0.00																		
8	-	-	-	-	-	-	-	-	36.00	8.86	7.48	5.18	3.86	2.56	1.26	2.46	1.36	0.51	0.11	0.00																		
9	-	-	-	-	-	-	-	-	-	8.02	8.49	7.12	5.78	4.43	3.07	1.71	2.19	1.19	0.57	0.19	0.00																	
10	-	-	-	-	-	-	-	-	-	8.62	7.86	6.78	5.78	4.68	3.61	2.37	2.58	1.69	0.81	0.23	0.00																	
11	-	-	-	-	-	-	-	-	-	-	8.31	7.24	6.78	5.78	4.68	3.61	2.37	2.58	1.69	0.81	0.23	0.00																
12	-	-	-	-	-	-	-	-	-	-	-	7.52	6.49	5.49	4.44	3.44	2.49	1.58	1.74	0.94	0.34	0.00																
13	-	-	-	-	-	-	-	-	-	-	-	-	6.88	5.88	4.88	3.88	2.88	1.98	1.28	1.44	0.84	0.34	0.00															
14	-	-	-	-	-	-	-	-	-	-	-	-	-	6.16	5.16	4.16	3.16	2.16	1.26	0.76	0.92	0.56	0.34	0.00														
15	-	-	-	-	-	-	-	-	-	-	-	-	-	-	5.45	4.45	3.45	2.45	1.45	0.75	0.45	0.56	0.36	0.24	0.00													
16	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	4.74	3.74	2.74	1.74	0.74	0.44	0.24	0.36	0.24	0.16	0.00												
17	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	4.03	3.03	2.03	1.03	0.43	0.23	0.13	0.24	0.16	0.08	0.00											
18	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	3.33	2.33	1.33	0.33	0.13	0.08	0.05	0.16	0.11	0.06	0.00										
19	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	2.63	1.63	0.63	0.13	0.06	0.03	0.02	0.08	0.05	0.03	0.00									
20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1.93	0.93	0.13	0.06	0.03	0.02	0.01	0.05	0.03	0.02	0.00								
callable bond prices																																						
time	0	0.25	0.5	0.75	1	1.25	1.5	1.75	2	2.25	2.5	2.75	3	3.25	3.5	3.75	4	4.25	4.5	4.75	5																	
1 / 1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20																	
0	-	100.00	99.93	99.45	98.48	91.99	79.88	69.16	60.02	59.09	58.18	44.12	36.85	30.87	29.59	28.37	24.79	24.00	23.47	21.29	19.99																	
1	-	100.00	99.96	99.52	98.44	91.95	79.84	69.12	60.08	59.06	58.15	44.15	36.88	30.89	29.62	28.47	24.88	24.00	23.47	21.29	19.99																	
2	-	-	100.00	99.99	99.56	98.47	91.93	81.33	71.33	69.09	68.17	44.21	36.87	30.87	29.62	28.47	24.88	24.00	23.47	21.29	19.99																	
3	-	-	-	100.00	99.99	99.56	98.47	91.93	81.33	71.33	69.09	68.17	44.21	36.87	30.87	29.62	28.47	24.88	24.00	23.47	21.29																	
4	-	-	-	-	100.00	99.99	99.56	98.47	91.93	81.33	71.33	69.09	68.17	44.21	36.87	30.87	29.62	28.47	24.88	24.00	23.47																	
5	-	-	-	-	-	100.00	99.99	99.56	98.47	91.93	81.33	71.33	69.09	68.17	44.																							

4% 쿠폰 채권의 가격은 100.91, 수의상환부 콜옵션의 가치는 3.05입니다.

따라서, 수의상환채의 가격은 약 97.86 입니다.

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 : $\min_{\theta, \sigma, k} \sum (\text{model price} - \text{market price})^2$ 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 n 은 가수, e 는 지수이며, 일반적으로 밑인 b 는 2를 사용함.

컴퓨터에서 많이 사용하는 float 타입 실수는 32bit를 사용하여 실수를 표현하며,

이는 2^{32} 가지로 모든 실수를 표현하게 됨을 의미함. (정수는 int 타입으로 모두 표현 가능)

따라서 소수점에 따라 정확한 값을 나타내지 못하는 문제는 항상 존재.

Precision of floating point arithmetic

실수표현의 정밀도는 $\text{float}(1 + \epsilon_{\text{math}}) > 1$ 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 double 타입을 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 double 타입의 경우 64bit로 실수를 표현하는데,

$x = \pm n \times 2^e$ 에서 부호(\pm) 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

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

컴퓨터로는 $h \rightarrow 0$ 을 정확히 표현할 수 없음.

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

1. Truncation error 최소화를 위해서는 h 는 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,

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

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

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

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

::: {.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^2 으로,

다른 방식에 비해서 오차가 훨씬 줄어들게 됨

유사한 방식으로 이계도함수와 편도함수를 유도하면,

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

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

총오차 및 최적의 h 산출

Forward difference approximation을 사용하고, $|f''(x)| \leq M$ 이라고 하면,

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

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

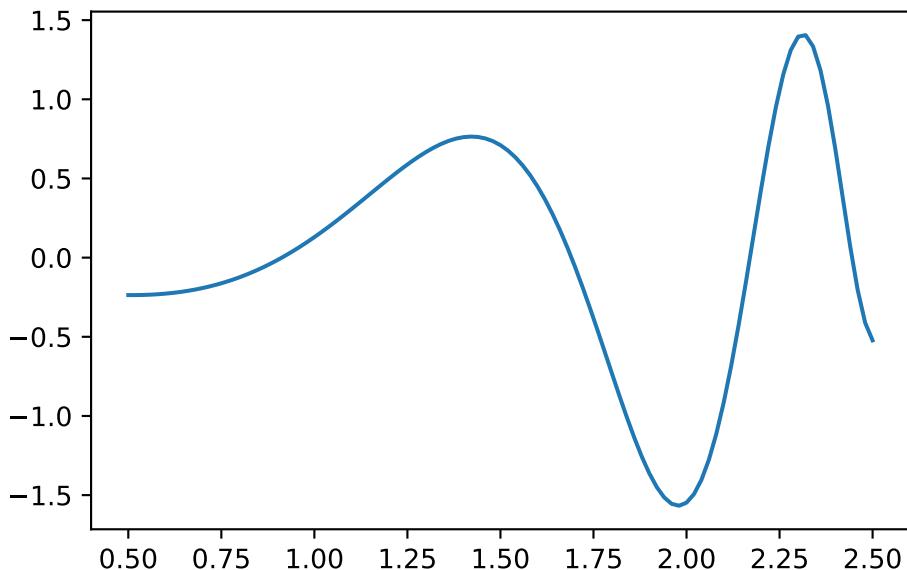
```
import numpy as np
import matplotlib.pyplot as plt
def fun(x):
    return np.cos(x**x) - np.sin(np.exp(x))

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 h 가 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'])

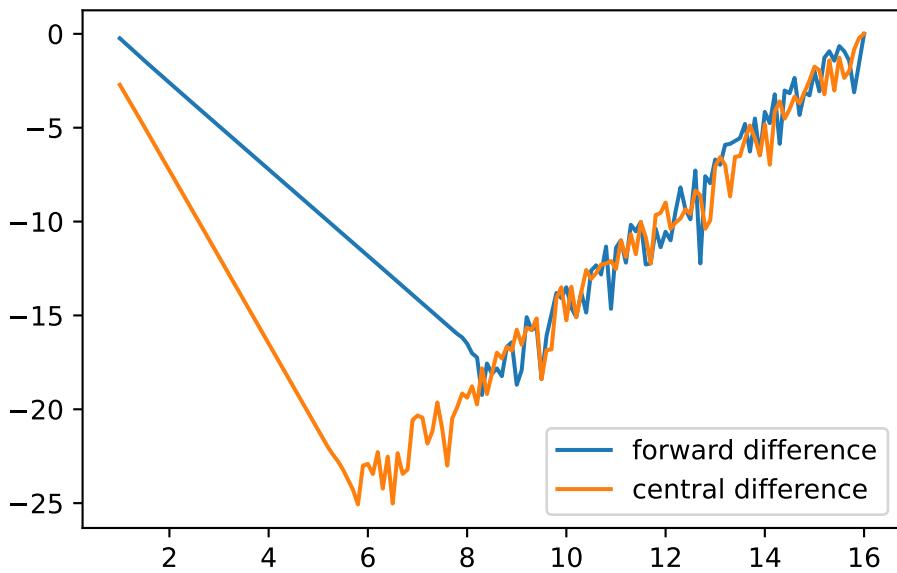
```

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 악조건 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

콜레스키 분해

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

PLU보다 연산량이 적음.

다면량 정규분포 난수를 생성할 때, 공분산행렬에 콜레스키 분해를 적용하면 쉽게 생성 가능.

QR 분해

수치해석기법 실습

```
import os
if os.getcwd() != '/Users/hwan/Desktop/Homepage/study_24fall/실습':
    os.chdir('실습')
os.getcwd()

'/Users/hwan/Desktop/Homepage/study_24fall/실습'
```

2.11 Ch3. Linear system

2.11.1 Equations

```
import numpy as np
from scipy.linalg import lu
from numpy.linalg import eig, cholesky, qr, svd
import time

# 예제 행렬 A와 벡터 b 정의
A = np.array([[1,2,3],[3,2,1],[5,1,1]], dtype=float)
b = np.array([12,10,8], dtype=float)

#
n = 5
A = np.random.randn(n, n)
b = np.random.randn(n)

#####
# Forward / Backward Substitution
#####

def forward(L, b):
    # 전방 대입 (Ly = Pb)
    y = np.zeros_like(b)
```

```

y[0] = b[0]/L[0,0]
for i in range(1,len(y)):
    y[i] = (b[i] - np.dot(L[i, :i], y[:i])) / L[i,i]
return y

def backward(U, y):
    # 후방 대입 (Ux = y)
    x = np.zeros_like(y)
    x[-1] = y[-1] / U[-1,-1]
    for i in range(len(x)-2, -1, -1):
        x[i] = (y[i] - np.dot(U[i, i+1:], x[i+1:])) / U[i, i]
    return x

#####
# LU Decomposition
#####
# LU 분해 수행: A = P @ L @ U
t0 = time.time()
P, L, U = lu(A)
Pb = np.dot(P.T, b)
y = forward(L, Pb)
x = backward(U, y)
t1 = time.time()

# 결과 출력
print("Solution x from PLU decomposition:")
print(x)

print("\nSolution x from A inverse")
print(np.linalg.inv(A).dot(b))

```

Solution x from PLU decomposition:
[12.76083942 2.28155216 -26.85469595 -23.45033461 28.17326524]

Solution x from A inverse
[12.76083942 2.28155216 -26.85469595 -23.45033461 28.17326524]

```

#####
#cholesky decomposition
#####

```

```

#check whether A is positive definite
eigen_values, eigen_vectors = eig(A)
print(eigen_values)
#c = cholesky(A) #error

[ 0.67444857+1.16816178j  0.67444857-1.16816178j  0.08484185+0.j
 -1.10418616+0.68605571j -1.10418616-0.68605571j]

```

```

B = A.T @ A
print("B = \n", B, '\n')

B =
[[ 6.75980329  2.42615302 -1.43962748  3.22213589 -1.98014089]
 [ 2.42615302  3.44204392 -0.39668661  1.96145718 -0.18451717]
 [-1.43962748 -0.39668661  2.02620681  0.20972367  2.79204537]
 [ 3.22213589  1.96145718  0.20972367  2.87802662  0.9190702 ]
 [-1.98014089 -0.18451717  2.79204537  0.9190702   4.31655632]]

```

```

eigen_values, eigen_vectors = eig(B)
print("Eigen values = ", eigen_values)

```

```

Eigen values =  [1.07007907e+01 6.31151224e+00 2.00747918e+00 1.24979393e-03
 4.01605076e-01]

```

```

c = cholesky(B) #A = c@c'
print(np.allclose(B, c@c.T), "\n")

```

True

```

y = forward(c, b)
x = backward(c.T, y)
print("Solution x from Cholesky decomposition:")
print(x)

print("\nSolution x from A inverse")
print(np.linalg.inv(B).dot(b))

```

```
Solution x from Cholesky decomposition:
```

```
[ 255.85772214  52.2097322 -577.02114312 -469.10314857  592.88299097]
```

```
Solution x from A inverse
```

```
[ 255.85772214  52.2097322 -577.02114312 -469.10314857  592.88299097]
```

```
#####
#QR decomposition
#####
Q, R = qr(A)
print(np.allclose(A, Q@R))
```

```
True
```

```
x = backward(R, Q.T @ b)
print("Solution x from QR decomposition:")
print(x)

print("\nSolution x from A inverse")
print(np.linalg.inv(A).dot(b))
```

```
Solution x from QR decomposition:
```

```
[ 12.76083942  2.28155216 -26.85469595 -23.45033461  28.17326524]
```

```
Solution x from A inverse
```

```
[ 12.76083942  2.28155216 -26.85469595 -23.45033461  28.17326524]
```

```
#####
#SVD decomposition
#####
U, S, Vh = svd(A)
print(np.allclose(A, U @ np.diag(S) @ Vh))
```

```
True
```

```
x = Vh.T @ np.diag(1/S) @ U.T @ b
print("Solution x from SVD decomposition:")
print(x)

print("\nSolution x from A inverse")
```

```

print(np.linalg.inv(A).dot(b))

Solution x from SVD decomposition:
[ 12.76083942   2.28155216 -26.85469595 -23.45033461  28.17326524]

```

```

Solution x from A inverse
[ 12.76083942   2.28155216 -26.85469595 -23.45033461  28.17326524]

```

2.11.2 Equations - time

```

n = 1000
A = np.random.randn(n,n)
b = np.random.randn(n)

#####
# LU Decomposition
#####
# LU 분해 수행: A = P ⊗ L ⊗ U
t0 = time.time()
P, L, U = lu(A)
Pb = np.dot(P.T, b)
y = forward(L, Pb)
x = backward(U, y)
t1 = time.time()

# 결과 출력
print("\nSolution x from PLU decomposition:")
print("Time = ", t1-t0)

#####
#cholesky decomposition
#####
B = A.T ⊗ A

t0 = time.time()
c = cholesky(B) #A = c@c'
y = forward(c, b)
x = backward(c.T, y)
t1 = time.time()

```

```

print("\nSolution x from Cholesky decomposition:")
print("Time = ", t1-t0)

#####
#QR decomposition
#####
t0 = time.time()
Q, R = qr(A)
x = backward(R, Q.T @ b)
t1 = time.time()

print("\nSolution x from QR decomposition:")
print("Time = ", t1-t0)

#####
#SVD decomposition
#####
t0 = time.time()
U, S, Vh = svd(A)
x = Vh.T @ np.diag(1/S) @ U.T @ b
t1 = time.time()
print("\nSolution x from SVD decomposition:")
print("Time = ", t1-t0)

#####
#A inverse
#####
print("\nSolution x from A inverse")
t0 = time.time()
Ainv = np.linalg.inv(A)
x = Ainv.dot(b)
t1 = time.time()
print("Time = ", t1-t0)

#####
# Numpy solution
#####
print("\nSolution x from Numpy solution(np.linalg.solve)")
```

```

t0 = time.time()
x = np.linalg.solve(A,b)
t1 = time.time()
print("Time = ", t1-t0)

```

Solution x from PLU decomposition:

Time = 0.025469064712524414

Solution x from Cholesky decomposition:

Time = 0.13914179801940918

Solution x from QR decomposition:

Time = 1.0981662273406982

Solution x from SVD decomposition:

Time = 0.5532040596008301

Solution x from A inverse

Time = 0.02708721160888672

Solution x from Numpy solution(np.linalg.solve)

Time = 0.00940704345703125

2.11.3 Linear system iterative

```

import numpy as np
import matplotlib.pyplot as plt

# 예제 행렬 A와 벡터 b 정의
A = np.array([[1,2,3],[3,2,1],[5,1,1]], dtype=float)
A = A.T@A # 이거 안하면 발산
b = np.array([12,10,8], dtype=float)

print("Solution x from A inverse")
sol = np.linalg.inv(A).dot(b)
print(sol)

```

Solution x from A inverse

[-0.3125 2.875 -1.3125]

```

#Gauss-Seidel

n = len(b)

x = np.ones(n) #initial value
idx = np.arange(n)
iter = 50

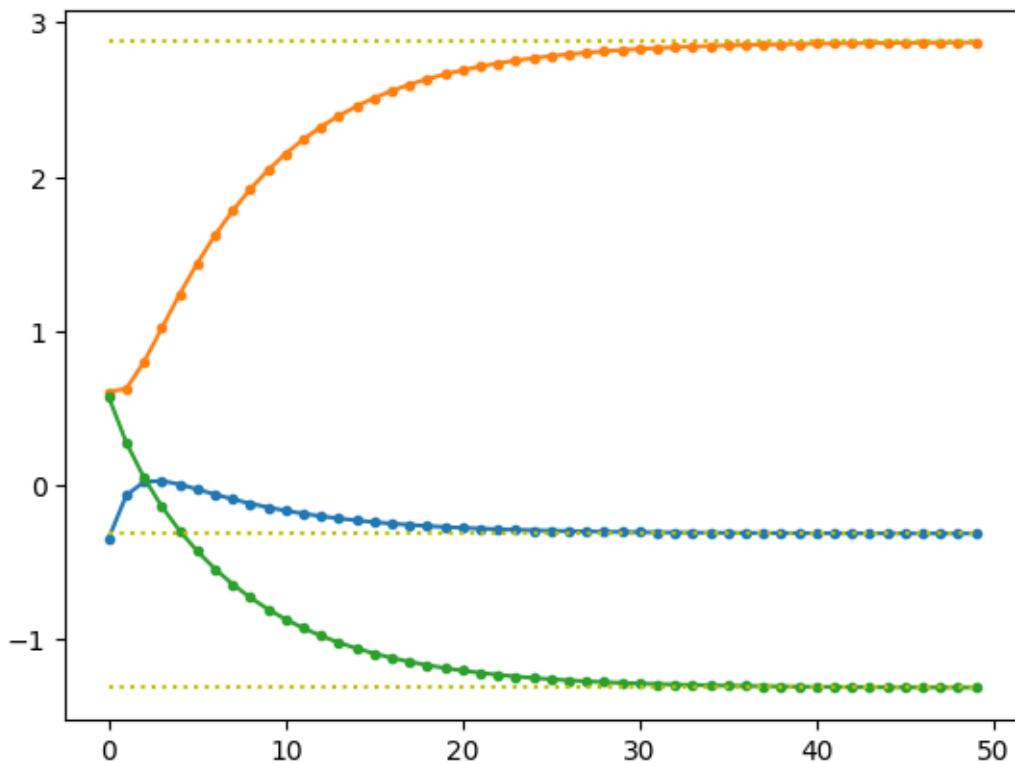
xs = np.empty((iter,n))

for j in range(iter):
    for i in range(n):
        mask = idx!=i
        x[i] = (b[i] - (A[i,mask] * x[mask]).sum()) / A[i,i]
    xs[j,:] = x

plt.plot(xs,'.-')

for i in range(n):
    plt.plot(np.arange(iter), np.ones(iter)*sol[i], ":y")

```



```

#SOR

omega = 0.5

n = len(b)

x = np.ones(n) #initial value
iter = 50

xs = np.empty((iter,n))

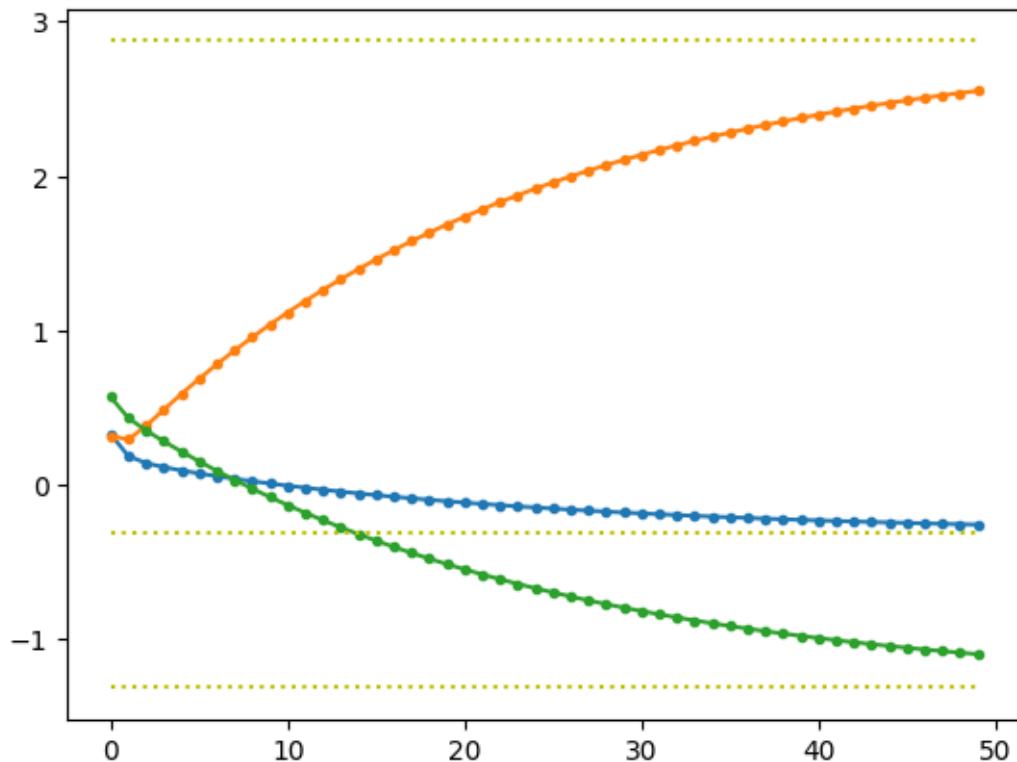
```

```

for j in range(iter):
    for i in range(n):
        x[i] = x[i] + omega * (b[i] - (A[i,:] * x).sum()) / A[i,i]
    xs[j,:] = x

plt.plot(xs,'.-')
for i in range(n):
    plt.plot(np.arange(iter), np.ones(iter)*sol[i], ":y")

```



2.12 Ch4. Finite Difference Method

2.12.1 1 factor FDM

```

from FDM_blackscholes import bsprice
from FDM_fdm import fdm_vanilla_option, exfdm_vanilla_option
import numpy as np
import time

s = 100
k = 100
r = 0.03
q = 0.01

```

```

t = 0.25
sigma = 0.2
optionType = 'put'

#Analytic Formula
t0 = time.time()
price = bsprice(s,k,r,q,t,sigma,optionType)
print(f"Analytic Price = {price:0.6f}")
print("computation time = ", time.time()-t0, "\n")

maxS, n, m = s*2, 1000, 10000
t0 = time.time()
v, ex_price = exfdm_vanilla_option(s, k, r, q, t, sigma, optionType,
                                     maxS, n, m)
print(f"EX-FDM Price = {ex_price:0.6f}")
print("computation time = ", time.time()-t0, "\n")

t0 = time.time()
v, ex_price = fdm_vanilla_option(s, k, r, q, t, sigma, optionType,
                                   maxS, n, m, 0)
print(f"EX-FDM Price = {ex_price:0.6f}")
print("computation time = ", time.time()-t0, "\n")

t0 = time.time()
v, im_price = fdm_vanilla_option(s, k, r, q, t, sigma, optionType,
                                   maxS, n, m)
print(f"IM-FDM Price = {im_price:0.6f}")
print("computation time = ", time.time()-t0, "\n")

t0 = time.time()
v, cn_price = fdm_vanilla_option(s, k, r, q, t, sigma, optionType,
                                   maxS, n, m, 0.5)
print(f"CN-FDM Price = {cn_price:0.6f}")
print("computation time = ", time.time()-t0, "\n")

```

Analytic Price = 3.724086
computation time = 0.000476837158203125

EX-FDM Price = 3.723938

```
computation time = 0.09713983535766602
```

```
EX-FDM Price = 3.723938
```

```
computation time = 0.6065318584442139
```

```
IM-FDM Price = 3.723837
```

```
computation time = 0.5502662658691406
```

```
CN-FDM Price = 3.723888
```

```
computation time = 0.542182207107544
```

```
...
```

Explicit FDM이 빠르고 좋아보이지만, 수치적 불안정성 문제가 있을 수 있어 안쓰는게 좋음 특히, 델타s<델타t인 경우, 발산할 가능성이 큼. 변동성이 커도 발산할 가능성이 큼.

정밀도는 s에 달려있어서 수렴성을 고려하면 잘 안써야하는게 맞음

```
...
```

```
maxS, n, m = s*2, 1000, 9500
```

```
v, ex_price = exfdm_vanilla_option(s, k, r, q, t, sigma, optionType,  
                                     maxS, n, m)
```

```
print(f"EX-FDM Price = {ex_price:0.6f}")
```

```
v, ex_price = fdm_vanilla_option(s, k, r, q, t, sigma, optionType,  
                                     maxS, n, m, 0)
```

```
print(f"EX-FDM Price = {ex_price:0.6f}")
```

```
v, im_price = fdm_vanilla_option(s, k, r, q, t, sigma, optionType,  
                                     maxS, n, m)
```

```
print(f"IM-FDM Price = {im_price:0.6f}")
```

```
v, cn_price = fdm_vanilla_option(s, k, r, q, t, sigma, optionType,  
                                     maxS, n, m, 0.5)
```

```
print(f"CN-FDM Price = {cn_price:0.6f}")
```

```
EX-FDM Price = -0.300684
```

```
EX-FDM Price = -0.300684
```

```
IM-FDM Price = 3.723835
```

```
CN-FDM Price = 3.723888
```

Part VI

금융시장 리스크관리('24 가을)

금융시장 리스크관리

Lecture2 : How Traders Manage Their Risks?

Greeks letters & Scenario analysis

Delta hedging

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

가장 기본적인 헛징방법으로, 파생상품과 같은 금융상품으로 구성된 포트폴리오에 대해

기초자산의 가격변동에 대한 민감도인 델타를 계산하여 이를 0으로 만듦으로써

기초자산의 가치변화로 인한 포트폴리오의 가치변화를 0으로 만드는 방법.

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

이를 Hedge and forget이라고 함.

그러나 비선형이라면, 기초자산의 가격변동에 따라 델타도 변하게 됨.

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

베가로 그런거는 대충넘어갔음

Taylor Series Expansion

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

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

금융시장에서 이를 적용한다면? $f(x)$ 는 포트폴리오의 가격함수이며, x 는 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 P 를 기초자산의 가격 및 시간에 따른 함수 $P(S, t)$ 라고 한다면, (변동성은 상수로 가정)

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

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

텔타헷징은 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

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

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

Weighting Schemes

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

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 어떤 분포를 따르는지 추정하기 위함.

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

주가수익률의 관측치 u_i 가 평균이 0인 정규분포를 따른다고 가정한다면?

변동성 σ 를 추정하기 위해 최대우도법을 사용할 수 있음.

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

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

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

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

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

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_n 은 서로 독립인지를 봄으로써 검증할 수 있음.

이건 Ljung-Box Test로 널리 알려져 있음.

z_n 을 통해 autocorrelation=0(H0)임을 검증하는 테스트임.

Lecture6 : Value at Risk and Expected Shortfall

VaR : 임계값

Expected Shortfall : $E[loss | loss > VaR]$

Properties of Coherent Risk Measures

1. Monotonicity : if $X \leq Y$ then $\eta(X) \geq \eta(Y)$
2. Translation invariance : For $K > 0$, $\eta(X + K) = \eta(X) - K$
3. Positive homogeneity : For $I > 0$, $\eta(I X) = I \times \eta(X)$
4. Subadditivity : $\eta(X + Y) \leq \eta(X) + \eta(Y)$

VaR vs. ES

VaR은 Subadditivity 만족하지 않음.

ES는 다 만족함.

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($LH = -\ln \sigma_i^2 - \frac{r_i^2}{\sigma_i^2}$)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

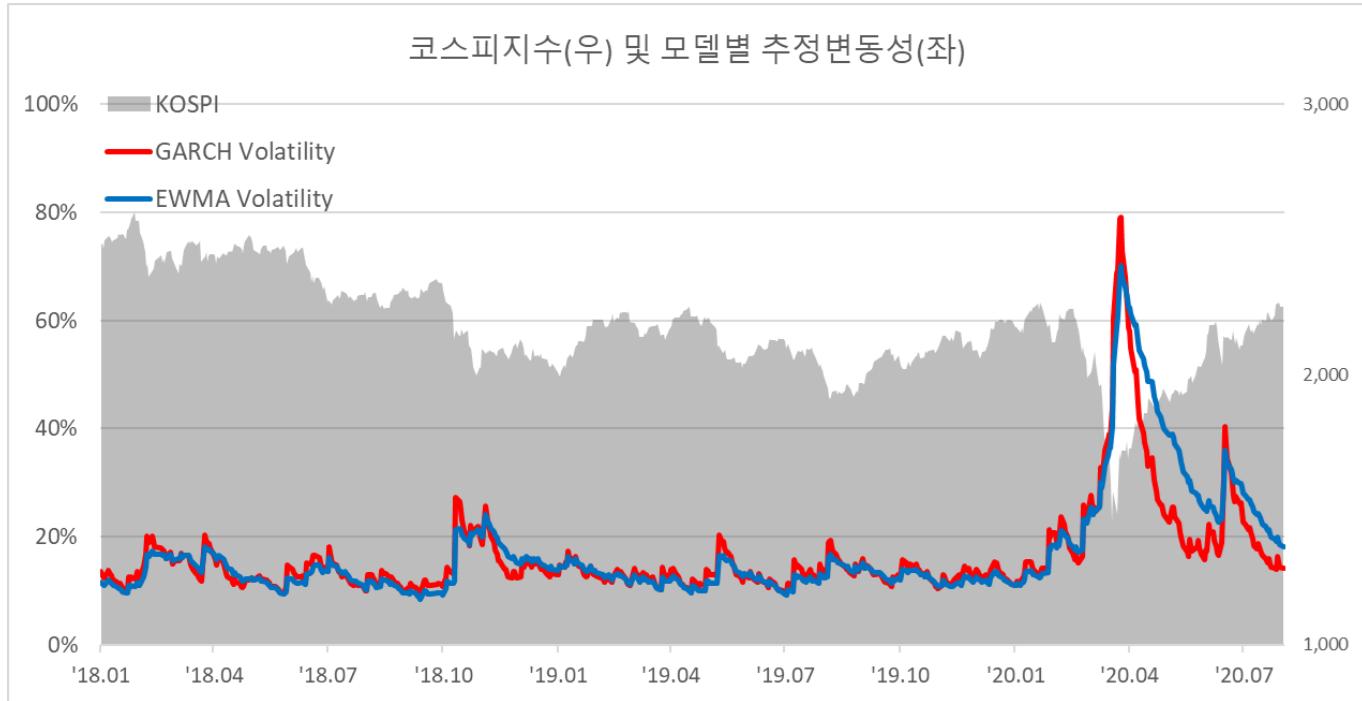
주어진 기간의 코스피지수에 대한 EWMA 모델의 파라미터 $\lambda = 0.934$ 입니다.

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($LH = -\ln \sigma_i^2 - \frac{r_i^2}{\sigma_i^2}$)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 경향이 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 $11.5\%(\alpha)$, 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 급등하게 되는 것입니다.

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 급등하는 패턴을 관측할 수 있습니다.(적색>청색)

다음으로, 변동성이 급등하고나서 시간이 지남에 따라 반영비율이 회복되면서 변동성이 평균 수준으로 회귀하게 되는데, 이때에도 당일 주가수익률의 반영비율이 큰, 직전 추정변동성의 반영비율이 상대적으로 낮은 **GARCH(1,1)** 모형의 회귀 속도가 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 $\omega = 0.00000135$, $\alpha = 0.0833$, and $\beta = 0.9101$. The current daily volatility is estimated to be 1%. Estimate the daily volatility in 30 days.

Answer 7

EWMA의 λ 가 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 추가흐름이 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 손실액을 찾는 것입니다. 이는 포트폴리오의 확률분포를 통해 알 수 있으며, CDF에서 하위 1% 임계값을 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_2 는 각각 평균이 0인 정규분포를 따르므로, 각각의 표준편차를 σ_1, σ_2 라고 하겠습니다.

이에 따라 포트폴리오의 가격변동 $\Delta P = 0.05PC_1 - 3.88PC_2$ 는 두 정규분포의 선형결합이므로 joint normal distribution이 되고, 각 PC 는 평균이 0, 독립임을 이용하여 ΔP 의 평균과 표준편차 σ 는 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, Y 에 대해 $E[XY] = E[X]E[Y]$ 가 성립합니다.

$$\begin{aligned} \text{따라서, } E[\Delta P^2] &= E[0.05^2PC_1^2 - 2 \times 0.05 \times 3.88PC_1PC_2 + 3.88^2PC_2^2] \\ &= 0.05^2E[PC_1^2] + 3.88^2E[PC_2^2] = 0.05^2\sigma_1^2 + 3.88^2\sigma_2^2 \end{aligned}$$

즉, $\Delta P \sim N(0, 0.05^2\sigma_1^2 + 3.88^2\sigma_2^2)$ 이므로 각 PC 의 표준편차를 알면 1% 임계값을 알 수 있습니다. Table2에 따라 $\sigma_1 = 17.55, \sigma_2 = 4.77$ 으로 이를 이용하면,

$$\sigma^2 = 0.05^2\sigma_1^2 + 3.88^2\sigma_2^2 = 343.30, \therefore \sigma = 18.53$$

$z_{0.01} = -2.33$ 임이 잘 알려져 있으므로, 1% 임계값은 $-2.33\sigma \approx -43.17$ 입니다.

따라서, 약 1% 확률로 발생할 수 있는 포트폴리오의 예상손실액은 최소 **43.17 million \$**입니다.

금융시장 리스크관리 과제2

Group 1 (김형환, 염지아, 유문선, 이희예, 홍지호)

Question 1-4 : Basic historical approach

Basic Historical Simulation Approach with 500 Days of Data

Estimate the VaR and ES of your portfolio for August 4, 2020 using 500 days of data.

- 1) Using the data you have collected in Assignment 1, generate 500 scenarios for one day loss of your portfolio. Note that you should use the daily observations over the most recent 500 days in your simulation.

(Hint: The sample period is from July 23, 2018 to Aug 3, 2020. The first observation on July 23, 2018 is scenario 0 and the last observation on Aug 3, 2020 is scenario 500.)

- 2) What is the one day 99% VaR and ES that are calculated using the basic historical simulation approach over the most recent 500 days?

Basic Historical Simulation Approach with 1000 Days of Data

Estimate the VaR and ES of your portfolio for August 4, 2020 using 1000 days of data.

- 3) Using the data you have collected in Assignment 1, generate 1000 scenarios for one day loss of your portfolio. Note that you should use the daily observations over the most recent 1000 days in your simulation.

(Hint: The sample period is from July 7, 2016 to Aug 3, 2020. The first observation on July 7, 2016 is scenario 0 and the last observation on Aug 3, 2020 is scenario 1000.)

- 4) What is the one day 99% VaR and ES that are calculated using the basic historical simulation approach over the most recent 1000 days?

Answer

(1) 엑셀 참조 / (2) : 500-days 99% VaR = 48.46, ES = 61.42 (만원)

(2) 엑셀 참조 / (4) : 1000-days 99% VaR = 37.46, ES = 50.91 (만원)

Question 5 : Comparison 500 vs. 1000

Comparison between the Two Measures

- 5) You have two sets of VaR and ES measures, one based on 500 observations and the other based on 1000 observations. Discuss how and why the measures differ from each other.

Answer

먼저, Historical VaR와 ES는 기간 중 포트폴리오 수익률을 고려하여 산출되므로, 산출기간 중 손실을 많이 기록한 거래일이 얼마나 있는지에 따라 결정됩니다.

참조기간(lookback period)가 짧더라도 이례적인 손실을 기록한 거래일이 포함되어있다면 VaR과 ES가 모두 클 확률이 높고, 길더라도 보합장이 지속되었다면 VaR과 ES 모두 낮을 가능성이 높습니다. 즉, 참조기간의 길이는 크게 중요하지 않습니다.

이제, 결과값을 살펴보면 참조기간 500일이 1000일보다 VaR과 ES 모두 높습니다.

500일의 기간은 '18.7.23 ~ '20.8.3이며 1000일은 '16.7.7 ~ '20.8.3인데, 1% level에서 500일은 상위 5거래일, 1000일은 상위 10거래일의 손실에 영향을 받게 됩니다.

두 기간은 공통적으로 '20년 코로나19 팬데믹 기간을 포함하는데요, 이 기간에는 증시가 단기간에 급락하면서 포트폴리오가 이례적인 손실을 기록한 거래일이 많았습니다.

특히, 두 참조기간 모두 손실 상위 5개 거래일은 '20년 상반기입니다.

따라서 참조기간 500일은 해당 5개 손실의 임계값과 평균값을 통해 VaR과 ES를 계산하고, 1000일은 해당 5개 손실과 그보다 낮은 5개의 손실을 이용하여 VaR과 ES를 계산합니다.

동일한 방법론에서 상위 5개의 손실값만들 사용한 500일 VaR과 ES가 높을 수 밖에 없습니다.

만약 중복되지 않는 기간('16.7.7 ~ '18.7.22)에 그 이상으로 이례적인 손실이 발생하였다면, 참조기간 1000일의 VaR과 ES가 더 높았을 것 입니다.

Question 5-10 : Back testing

Back-Testing

You are curious about how well the VaR measures generated from the basic simulation approach would have worked in the past. You back-test the first VaR measure from 500 observations using the full data you collected in Assignment 1.

- 6) Simulate scenarios for one day loss of your portfolio using the full sample from January 3, 2005 to August 3, 2020. You will end up with 3855 scenarios in total.
- 7) Calculate the portfolio's one-day 99% VaR from 500 observations for each day between January 9, 2007 and August 3, 2020. You will have 3355 number of VaR estimates in total.

(Hint: For example, the one-day 99% VaR for January 19, 2009 can be calculated as the 5th largest simulated loss over the previous 500 days from January 9, 2007 to January 16, 2009. The VaR for the next day then can be computed by rolling the training window forward by one day. That is, the one-day 99% VaR for January 20, 2009 can be calculated as the 5th largest simulated loss over the 500 days from January 10, 2007 to January 19, 2009. Repeat this process until you reach August 3, 2020.)

(Hint: Use Excel's "LARGE" function to get the 5th largest simulated loss.)

- 8) Compare the daily VaR with the actual portfolio loss on the corresponding day. On how many days does the actual loss exceed the VaR between January 9, 2007 and August 3, 2020? These observations are referred to as exceptions.
- 9) Plot the VaR and realized portfolio loss from January 9, 2007 to August 3, 2020. When do the exceptions tend to happen?
- 10) Let m the number of exceptions you get. Define $p = 1 - \text{confidence level}$. Recall that if the VaR model is accurate, that is, if the probability of an exception on any given day is p , then the probability of the VaR level being exceeded on m or more days out of a total of n days is

$$\text{probability of } m \text{ or more exceptions} = \sum_{k=m}^n \frac{n!}{k!(n-k)!} p^k (1-p)^{n-k}$$

In Excel, you can calculate this probability using the BINOM.DIST function. In particular, for our VaR measure,

$$\text{probability of } m \text{ or more exceptions} = 1 - \text{BINOM.DIST}(m - 1, 3355, 0.01, \text{TRUE}).$$

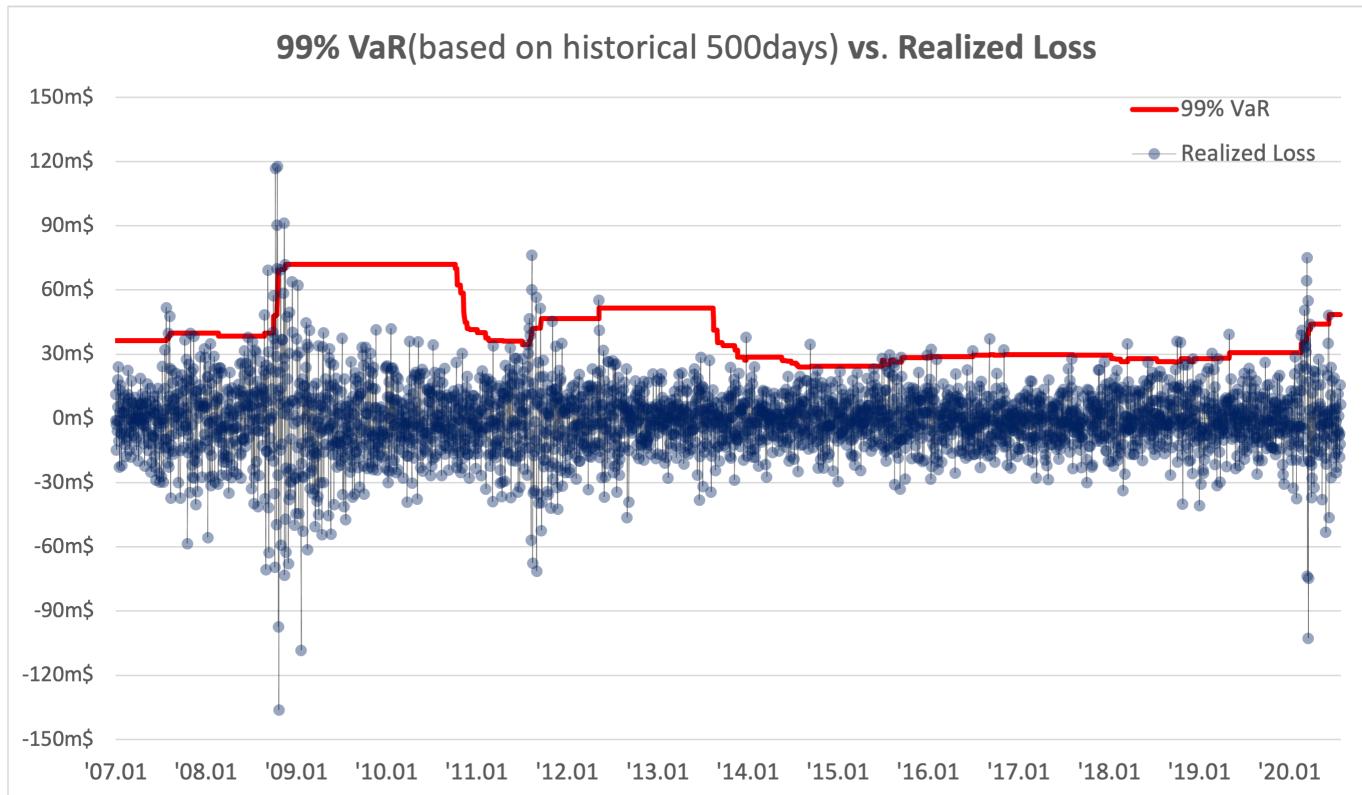
Should we reject or accept the VaR model at a 5% significance level?

Answer

(6) - (7) : 엑셀 시트 참조

(8) : 49 days

(9)



실제 포트폴리오 손실이 99% VaR을 초과한 날을 기록한 그래프입니다. 주로 '08년말, '11년말, '20년초 3개의 구간에 해당 거래일이 집중되어 있음을 알 수 있습니다.

각 구간은 미국발 서브프라임 금융위기, 유럽발 재정위기, 코로나19 팬데믹 기간에 해당하는데, 모두 증시가 단기간에 급락한 기간이라는 공통점이 있습니다.

이는 VaR 그래프(적색)의 추세에 나타나는데, VaR 산출은 과거 500일의 포트폴리오 수익률에 기반하므로 주가가 급락하여 포트폴리오가 연속적인 손실을 기록하면, 익일 VaR 산출부터 해당 손실이 반영되어 급증하게 됩니다.

각 구간의 시작지점에 이러한 특징이 나타나며, 모두 증시 급락으로 VaR이 계단식으로 상승하였음을 알 수 있습니다.

(10) : Reject. VaR model is NOT significant.

우리의 VaR 모델이 유의하다면, 포트폴리오 손실이 1% VaR을 초과하는 확률은 1%로 수렴할 것 입니다.

데이터의 3355개의 VaR와 포트폴리오 실현 손실, 초과 횟수가 49회임을 이용하여 가설검정을 수행해보겠습니다.

1. H0 : VaR is significant. $\frac{m}{N} < 0.01$
2. H1 : VaR is not significant. $\frac{m}{N} \geq 0.01$

실현손실이 VaR을 초과하면 1 아닌 경우 0인 확률변수를 X 라 하면, 우리는 이를 $p = 0.01$ 이고 3355번 시행하는 베르누이 분포로 모델링할 수 있습니다.

문제의 식에 따라 산출한 p 값은 0.007인데, 이는 3355번 동안 49번 이상 손실이 초과할 확률이 0.007이라는 뜻입니다.

다시 말해, 가설검정에서 p -value가 0.007이므로 5% 수준에서 귀무가설을 기각하고 대립가설을 채택하게 됩니다.

Question 11 : VaR and ES

Value at Risk and Expected Shortfall

- 11) Suppose that the change in the value of a portfolio over a one-day time period is normal with a mean of zero and a standard deviation of \$2 million.

(Hint: Use the formula on slide 23 of lecture 6.)

(Hint: Use NORM.S.INV(probability) function to calculate the inverse of the standard normal cumulative distribution for a given probability level.)

(Hint: You can enter PI() in a cell to get the value of pi=3.14159265 in Excel.)

- a) What is the one-day 99% VaR?
- b) What is the five-day 99% VaR? Assume that the changes in the value of the portfolio on successive days are independent.
- c) What difference does it make to you answer to (b) if there is first-order daily autocorrelation with a correlation parameter equal to 0.16?
- d) What is the one-day 97.5% ES? Note that the one-day 99% VaR and one-day 97.5% ES are almost the same.

Answer

$$T - day \quad VaR_{\alpha} = \left(\mu + \sigma \times N^{-1}(\alpha) \right) \sqrt{T}$$

$$T - day \quad ES_{\alpha} = \left(\mu + \sigma \frac{e^{-\frac{N^{-1}(\alpha)}{2}}}{\sqrt{2\pi(1-\alpha)}} \right) \sqrt{T}$$

(a) \$4.65m

(b) \$10.40m

(c) \$11.82m

autocorrelation o] 있는 경우, \sqrt{T} 대신 $\sqrt{T + 2(T-1)\rho + 2(T-2)\rho^2 \dots}$ 를 사용

(d) \$4.68m

Question 12 : EVT

Extreme Value Theory

12) Recall the application of Extreme Value Theory in lecture 7, slide 36 through 40.

(Hint: Recall that $n = 500$, $n_u = 22$, $u = 160$, $\beta = 32.532$, and $\xi = 0.436$.)

- a) What is the probability that the loss will exceed \$400,000?
- b) What is the one-day VaR with a confidence level of 97%?

Answer

문제풀이에 앞서, 극단값 이론(*Extreme Value Theory*)의 주요 내용을 간단히 정리하겠습니다.

우리가 널리 사용하는 중심극한정리(CLT)는 표본들의 평균의 정규성에 대한 정리입니다.

모집단의 분포를 모르더라도 표본이 충분히 크다면 표본평균이 정규분포를 따른다는 의미입니다.

반면, EVT는 표본들의 극단값에 대한 정리입니다.

마찬가지로 모집단의 분포를 모르더라도, 표본이 충분히 크다면 특정 임계값 이상의 극단값(extreme value)들은 파레토 분포를 따르게 됩니다.

Tip

확률변수 X 의 오른쪽(right tail) 임계값 u (대략 95% 수준)에 대해, $u < y \in X$ 인 y 를 극단값으로 정의한다면, y 에 대한 확률분포는 기존 X 의 확률분포를 이용한 조건부확률로 표현할 수 있습니다.

$P(x > y) = P(x > u) \times P(x > y | x > u)$ 이를 y 에 대하여 정리하면 파레토분포가 됩니다. (Gnedenko)

$$\text{CDF} : G_{\xi,\beta}(y) = 1 - \left(1 + \frac{\xi}{\beta}y\right)^{-\frac{1}{\xi}}$$

$$\text{PDF} : g_{\xi,\beta}(y) = \frac{dG}{dy} = \frac{1}{\beta} \left(1 + \frac{\xi}{\beta}y\right)^{-\frac{1}{\xi}-1}$$

여기서, 파레토분포의 형태는 scale β 와 shape ξ 두개의 파라미터에 따라 결정됩니다.

X 가 정규분포라면, ξ 는 0이 됩니다.

일반적으로 일정수준의 임계값 u 를 지정한 후, 주어진 데이터에 최대우도법(MLE)을 적용하여 scale, shape를 추정하게 됩니다.

이후 과거 데이터를 통해 도출된 극단치의 분포를 활용하여 VaR, ES 등을 산출할 수 있습니다.

i EVT를 활용한 VaR 등의 산출

임계값 u 보다 큰 VaR_α 에 대하여, $v \in X$ 가 VaR_α 보다 클 확률은 두가지로 표현할 수 있습니다.

1. $Prob(v > VaR_\alpha) = 1 - \alpha$

2. EVT의 파레토분포 활용,

$$Prob(v > VaR_\alpha) = P(v > u) \times P(v > VaR_\alpha | v > u) = (1 - F(u)) \times (1 - G_{\xi, \beta}(VaR_\alpha - u))$$

$$\Rightarrow Prob(v > VaR_\alpha) = \frac{n_u}{n} \left(1 + \frac{\xi(VaR_\alpha - u)}{\beta} \right)^{-\frac{1}{\xi}}$$

이) 두식을 연립하여 정리하면,

$$VaR_\alpha = u + \frac{\beta}{\xi} \left\{ \left(\frac{n}{n_u} (1 - \alpha) \right)^{-\xi} - 1 \right\}$$

유사한 방식으로 ES(Expected Shortfall)을 유도하면,

$$ES_\alpha = E[v | v > VaR_\alpha] = \frac{VaR_\alpha + \beta - \xi u}{1 - \xi}$$

이제, 위의 내용을 바탕으로 문제를 풀어보겠습니다.

(a) : 0.16%

위의 $Prob(v > VaR_\alpha) = \frac{n_u}{n} \left(1 + \frac{\xi(VaR_\alpha - u)}{\beta} \right)^{-\frac{1}{\xi}}$ 를 이용하면,

손실이 400,000USD를 초과할 확률 $P(v > 400) = \frac{n_u}{n} \left(1 + \frac{\xi(400 - u)}{\beta} \right)^{-\frac{1}{\xi}}$ 입니다.

주어진 파라미터를 이용하여 이를 계산하면, $P(v > 400) \approx 0.16\%$

(b) : 173.56 (\$173,560m)

$VaR_\alpha = u + \frac{\beta}{\xi} \left\{ \left(\frac{n}{n_u} (1 - \alpha) \right)^{-\xi} - 1 \right\}$ 을 이용하면,

97% 수준의 1-day VaR는 $VaR_{0.97} = u + \frac{\beta}{\xi} \left\{ \left(\frac{n}{n_u} (0.03) \right)^{-\xi} - 1 \right\} \approx 173.56$

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, 베거킹)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.

$a+b$ $a+b+c+f+g$ $c+f+g$

$c+d$ $c+d+b+e$ $b+e$

subsidy expenditure $b+c+e+f+g+h$

$b+c+e+f+g - b-c-e-f-g-h$: dead weight loss

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

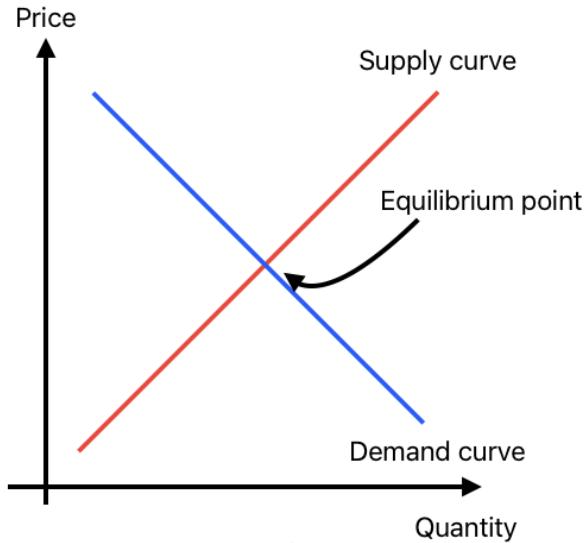
- > Decrease number of buyers for steel production
- > Demand curve shifts to the left.

Both curves will shift to the left side,

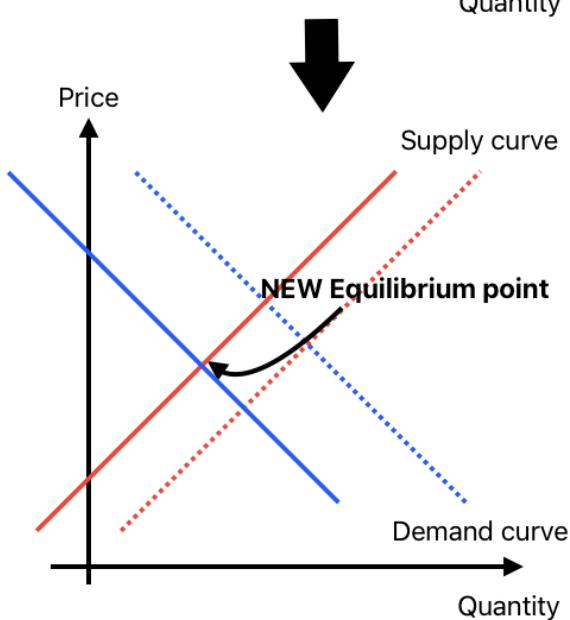
so NEW equilibrium point also shift to the left side.

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 expensive 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 decreased**
But **price of equilibrium can be decrease or increase**
It depends on how much each curve shifts.

Figure 2.1: Sample question

Assignments 1

Assignments: Due 9/10 (Tuesday)

■ Examine the behavior of supply and demand for Wheat

$$\text{Demand: } Q_D = 3550 - 266P \quad \text{Supply: } Q_S = 1800 + 240P$$

the market-clearing price of wheat:

$$Q_S = Q_D$$

$$1800 + 240P = 3550 - 66P$$

$$506P = 1750$$

$$P = \$3.46 \text{ per bushel}$$

Substituting into the supply curve equation, we get

$$Q = 1800 + (240)(3.46) = 2630 \text{ million bushels}$$

•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

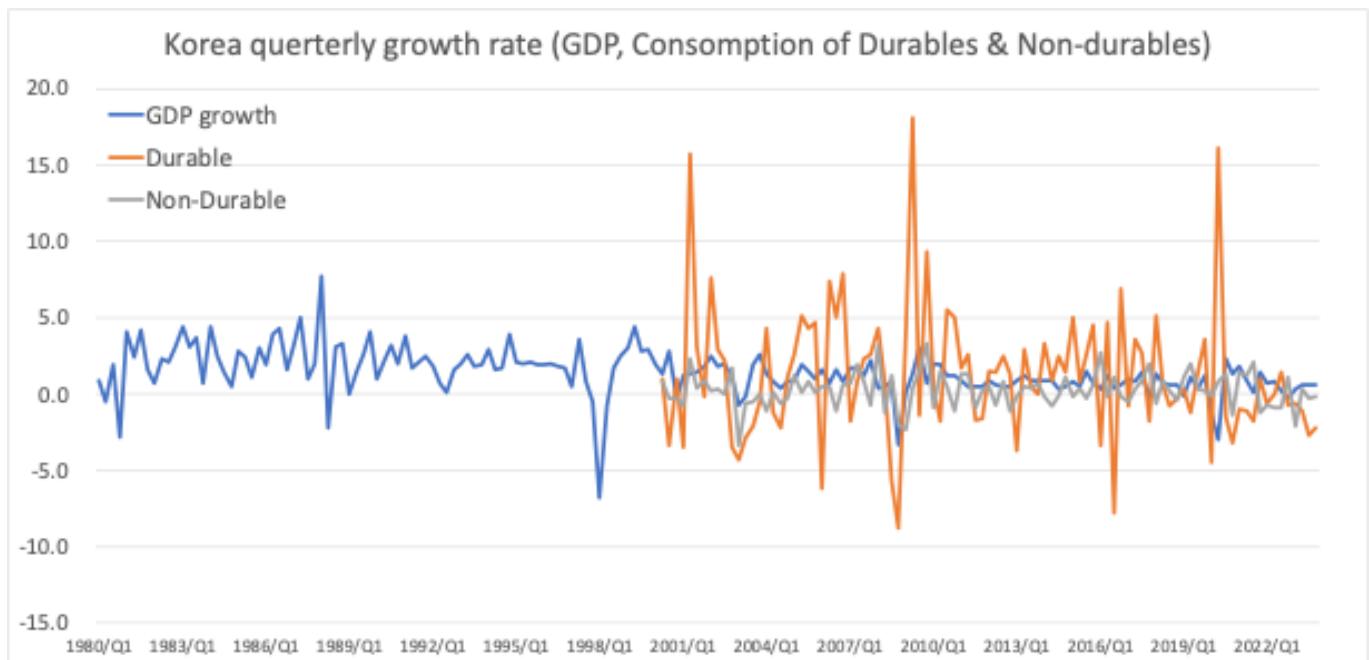
Home work:

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

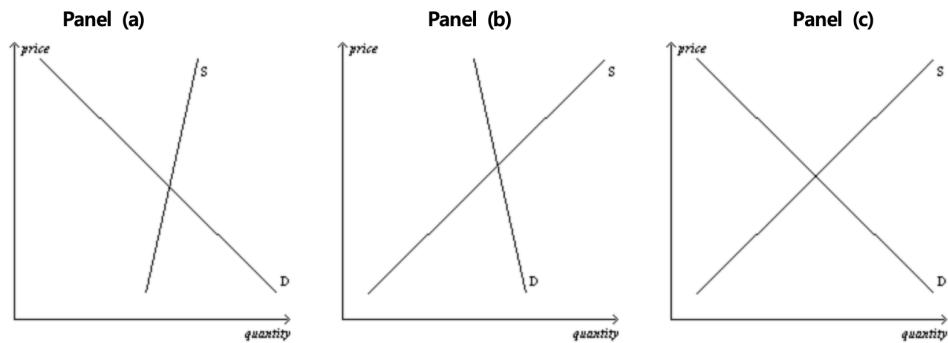
So, demand curve of donuts shift to the left, price and quantity will decrease.

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, does 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, does 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.

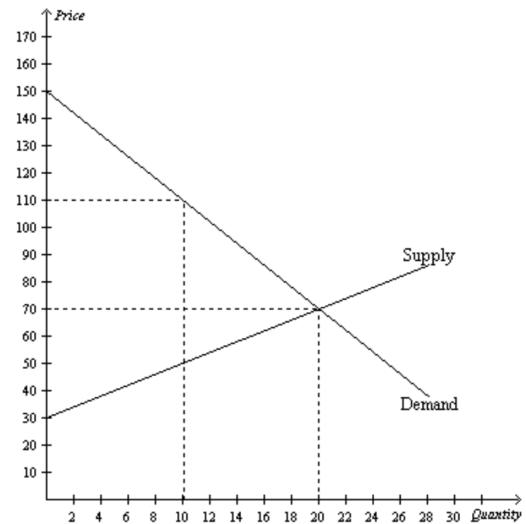
Microanalysis of financial economics Assignment3

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.

Microanalysis of financial economics Assignment4

20249132 Kim Hyeonghwan (김형환)

Question

Home Work

- ① Please show the portion of Indirect tax as of 2023 in Korea
(or your home country).
- ② Please show the portion of income tax, corporate tax, value-added tax as of 2023 in Korea (or your home country)

— Due 9/26 5pm

Answer

구 분	세수(조원)	비율
소득세	115.8	35.1%
법인세	80.4	24.4%
상속세	8.5	2.6%
증여세	6.1	1.8%
간접세	93.0	28.2%
부가가치세	73.8	22.4%
환경세	10.8	3.3%
방위세	0.0	0.0%
교육세	5.2	1.6%
농어촌특별세	5.5	1.7%
종합부동산세	4.6	1.4%
전 체	330.0	100.0%

Indirect tax : 28.2%

Income tax : 35.1%

Corporate tax : 24.4%

Value-added tax : 22.4%

Microanalysis of financial economics Assignment5

20249132 Kim Hyeonghwan (김형환)

Question 1

Home Work

Suppose the market is described by the following supply and demand equations:

$$Q_s = 2P$$

$$Q_D = 300 - P$$

- a. Solve for the equilibrium quantity.
- b. Suppose that a tax of T is placed on buyers, so the new demand curve is
$$Q_D = 300 - (P + T)$$
Solve for the new equilibrium.
- c. Use your answer from part (b) to solve for tax revenue as a function of T .
- d. Solve for the DWL as a function of T
- e. The government now levies a tax of \$200 per unit on this good. Is this a good policy? Why? Why not? Can you propose a better policy?

Answer

- a. $Q_S = Q_D \Rightarrow 2P = 300 - P \Rightarrow P^* = 100, Q^* = 200$
- b. $300 - P - T = 2P \Rightarrow P^T = 100 - T/3, Q^T = 200 - 2T/3$
- c. $T \times Q^T = 200T - 2T^2/3$
- d. $T \times (Q^* - Q^T) \div 2 = T^2/3$
- e. if $T = 200$, $P^T = Q^T = 200/3$ and DWL occurs the amount of $40000/3$. Since the supply curve is elastic and the demand curve's slope is 1, any policy about price control should be bad for market. Because quantity change very fast, it causes many DWL.

Question 2

Sample questions: #3 in Mankiw

When China's clothing industry expands, the increase in World supply lowers the world price of clothing.

- a. Draw an appropriate diagram to analyze how this change in price affects CS, PS and total surplus in a nation that imports clothing, such as the U.S.
- b. Now draw an appropriate diagram to show how this change in price affects CS, PS, TS in a nation that exports clothing, such as Dominican Republic.
- c. Compare your answer to parts (a) and (b). What are the similarities and what are the differences? Which country should be concerned about the expansion of the Chinese textile industry? Which country should be applauding it?

Answer

- a. In U.S., China's industry expands makes U.S. cloth importing supply curve shift to the right. Price goes down and quantity goes up. CS and TS should increase, but PS depends on demand curve's elasticity. If elastic, PS will increase.
- b. In export nations, demand curve shifts to the left. Both price and quantity go down. CS, PS, TS should decrease.
- c. Importing sides, it is good for total surplus. But exporting sides, it should be bad.

Part VIII

글로벌 지속가능회계('24 가을)

글로벌 지속가능회계 1주차