2021 Capstone Design - Industrial Management Engineering

머신러닝을 통한 비트코인 가격 예측 및 트레이딩 경진대회

인천대학교 산업경영공학과
시스템종합설계 인베(investing) 팀

Introduction	ANALYSIS	Modeling	Data preprocessing	Program Development
대회 및 규칙 소개	Data EDA	ARIMA / AUTO ARIMA	Data smoothing	데이터 스무딩 후 모델링
대회 내 데이터 소개	Model Research	Prophet / Neural Prophet	Data discretizing	데이터 계층화 후 모델링
일정 계획		RNN	Data log normalization	결과
		LSTM		향후 진행 방향

인공지능 비트 트레이더 경진대회 시즌 2

▶대회 및 규칙 소개

- 23시간 동안 코인 하나의 분 단위 특정 변화를 입력 받아 이후 2시간 동안의 코인의 분 단위 open가격의 움직임을 추론하는 문제
- 23시간 동안 코인 하나의 분 단위 특정 변화를 보이는 개별 샘플에 대해 현재 보유한 금액을 기준으로 매수 비율과 매도시점 결정

(단, 각 샘플마다 매수한 코인을 2시간 이내 매도해야 한다.)

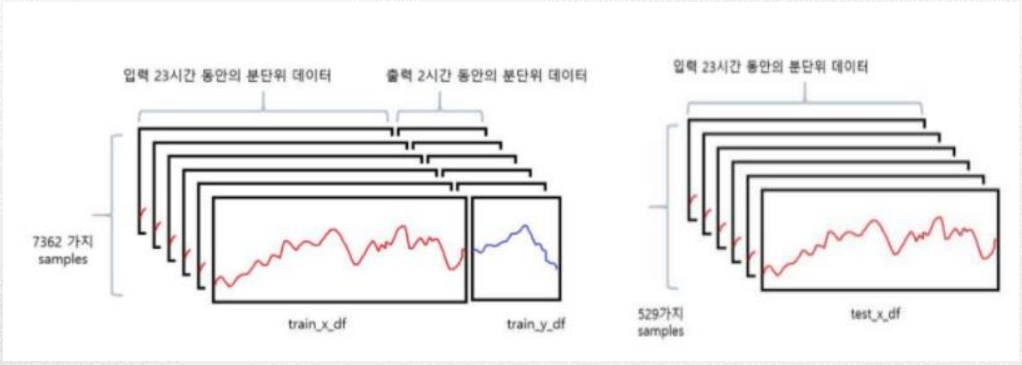
▶대회일정

04.01	05.01	05.31	06.07	06.14
대회시작	팀 병합 마감	대회종료	우승 후보 코드 제출 마감	최종 순위 발표

대회 내 데이터 소개

▶ 데이터 설명

- Train데이터 : 총 10가지 종류의 코인을 포함하는 7352가지의 sample
- Test 데이터 : train_x와 동일한 구성을 갖는 529가지의 sample
- 데이터는 (샘플 수, 시간, 특징 수) 3차원 표현



시계열 예측 문제 의미

▶ 시계열 예측 문제

- 시계열 데이터를 예측할 때, 목표는 관측값의 수열이 미래에 계속될 것인지 예측하는 것
- 본 대회는 주가(비트코인) 가격 예측으로, 일반적인 시계열 예측이 아닌, 비주기성 예측
- 전통 학문적 시계열 예측 및 주기성 확보를 위한 학문적 접근으로
  다양한 예측 모델 및 다양한 전처리 기법 등을 활용가능함

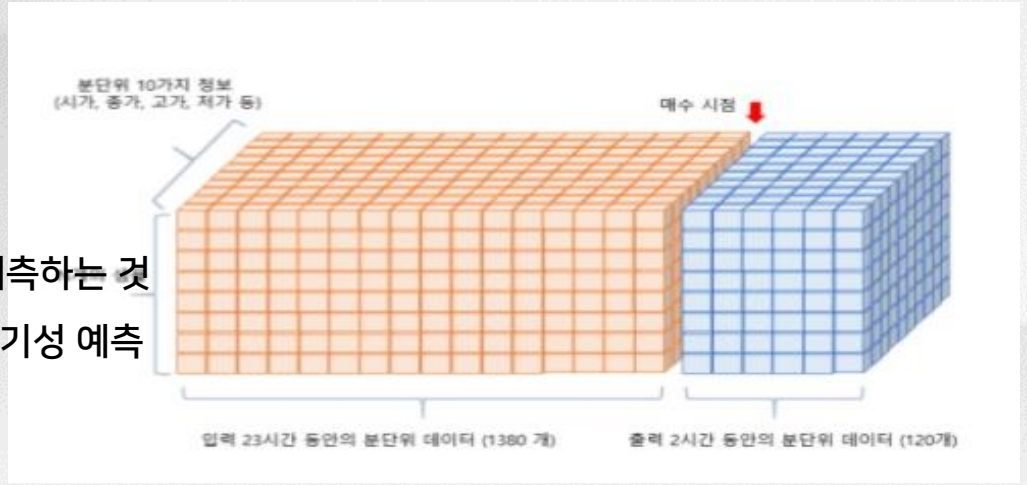


그림.1 데이터 3차원 시각화 예시

일정 계획

	4월 1주	4월 2주	4월 3주	4월 4주	5월 1주	5월 2주	5월 3주	5월 4주
Data EDA								
Model Research								
Mid term								
Modeling								
Program Development								

▶ Columns 설명

sample_id : 개별 샘플의 인덱스

time : x_df는 0 ~ 1379분, y_df는 0 ~ 119분의 값을 갖는다. 동일한 샘플 내 시간정보

coin_index : 10가지 종류의 코인에 대한 비식별화 인덱스(0~9)

open : 시가

high : 최고가

Low : 최저가

close : 종가

volume : 거래량

trades : 거래건수

*quote_av : quote asset volume, 다른 화폐 가치로 환산된 거래량

*tb_base_av : taker buy base asset volume, 체결된 거래량

*tb_quote_av : taker buy quote asset volume, 다른 화폐 단위로 환산된 구매량

Column 명	설명
Coin_index	코인의 종류
Sample_id	학습할 데이터 시퀀스 세트
Time	데이터 시간(분) 표시(1380 min = 23 hours)
Open	시가
High	최고가
Row	최저가
Close	종가
Trades	거래건수
Base asset volume(=volume)	거래량
Quote asset volume	타 화폐 단위 대비 거래량
Taker buy base asset volume	체결된 구매량
Taker buy quote asset volume	타 화폐 단위 대비 구매량

그림. 2 데이터 컬럼(피쳐) 종류

*Source : https://www.binance.kr/apidocs/#individual-symbol-mini-ticker-stream

▶ One sample data description

- X : 1380분(23시간)의 연속 데이터
- Y : 120분(2시간)의 연속 데이터
- 23시간 동안의 데이터 흐름을 보고 앞으로의 2시간 데이터를 예측
- sample_id는 7661개의 세트 구성, 각 세트는 독립적인 dataset
- coin_index는 총 10개 종류로 구성(index number is 0 ~ 9)

	sample_id	time	coin_index	open	high	low	close	volume	quote_av	trades	tb_base_av	tb_quote_av
0	0	0	9	0.983614	0.983614	0.983128	0.983246	0.001334	10.650987	0.009855	0.000848	6.771755
1	0	1	9	0.983245	0.983612	0.982453	0.982693	0.001425	11.375689	0.016137	0.000697	5.565188
2	0	2	9	0.982694	0.983612	0.982403	0.983002	0.001542	12.301942	0.014166	0.000905	7.225459
3	0	3	9	0.983009	0.984848	0.983009	0.984486	0.002520	20.134695	0.021557	0.001171	9.353000
4	0	4	9	0.984233	0.984606	0.983612	0.984164	0.002818	22.515448	0.021434	0.001799	14.372534
...	...	...	...	...	...	...	...	...	...	...	...	...
1375	0	1375	9	0.999015	0.999388	0.998400	0.998400	0.002577	20.899395	0.017492	0.001371	11.117771
1376	0	1376	9	0.998400	0.999260	0.998400	0.999016	0.001256	10.188805	0.011333	0.000556	4.510879
1377	0	1377	9	0.999018	0.999629	0.998936	0.999629	0.002902	23.543552	0.014289	0.001761	14.289263
1378	0	1378	9	0.999629	1.000116	0.999143	1.000000	0.004383	35.568905	0.020941	0.002810	22.806458
1379	0	1379	9	1.000000	1.000123	0.999388	0.999998	0.001269	10.297554	0.014782	0.000909	7.376117

그림. 3 한 샘플 데이터 예시

▶ 코인 별 샘플 개수

- 각 코인별로 샘플 개수는 다름
- 9, 8번의 샘플 수가 가장 많음

```
9    1208.0
8    1208.0
6    1095.0
7     976.0
4     960.0
0     937.0
5     575.0
1     412.0
3     159.0
2     131.0
Name: coin_index, dtype: float64
```

그림. 4 코인 별 샘플 개수

▶ Open data outlier problem

- 샘플 내 특정 구간 이외에의 데이터들은 빈도가 너무 적음(outlier)
- Regression 학습 시 해당 부분들을 학습하기 어려움

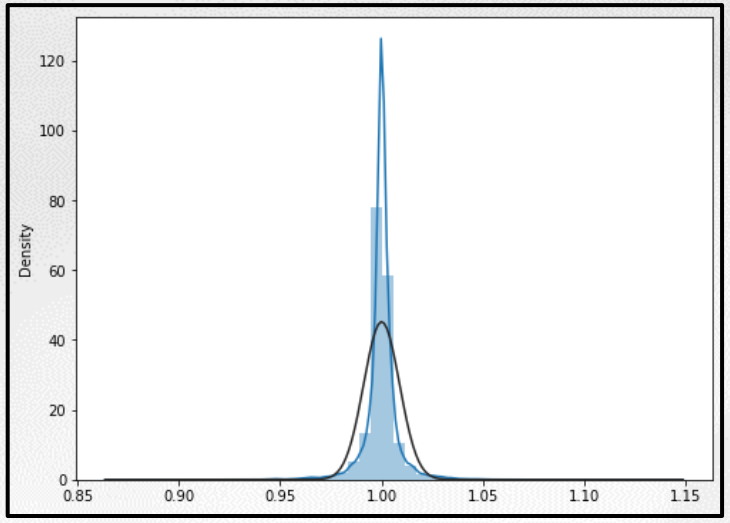


그림. 가격 데이터 분포

▶ Open data box plot

- Box-plotting 시, 데이터의 분포가 아래 그림과 같음
- Outlier 들에 대한 전처리 혹은 trimming 이 필요함

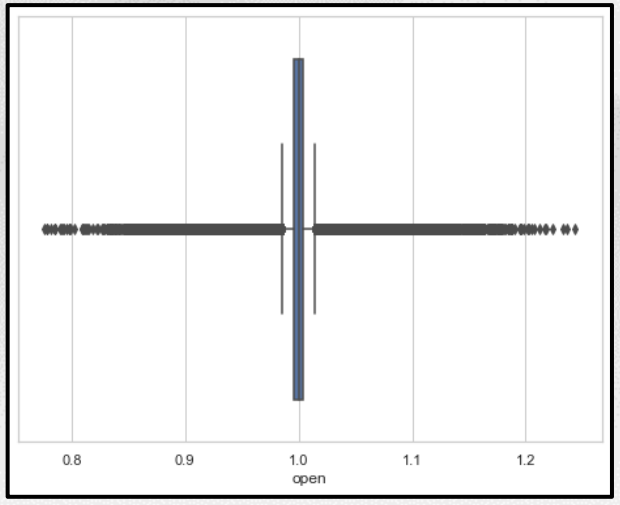


그림. 한 샘플 데이터 예시

▶ Open data range box plot

- 학습 전 outlier 데이터를 가진 샘플들을 detection하기 위해, 다음 과 같이 데이터의 max-min(range)을 plotting 하고 이를 기준으로 처리 계획

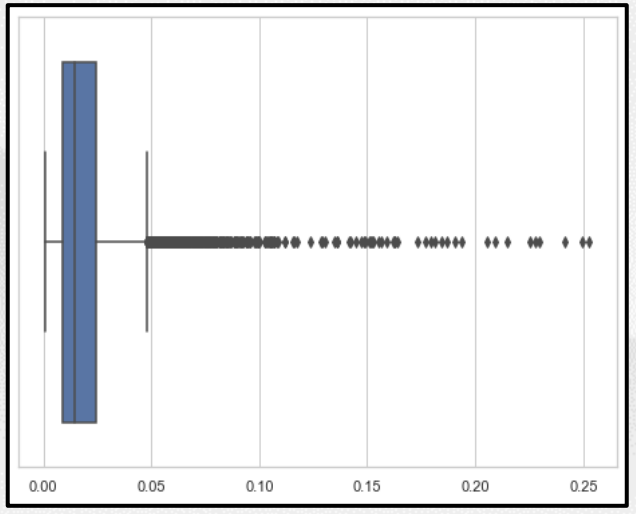


그림. 한 샘플 데이터 예시

Model Research

- ARIMA / AUTO ARIMA
- Prophet / Neural Prophet
- RNN(Encoder & Decoder)
- Deep Learning(GRU)
- Deep Learning(bidirectional GRU)
- Deep Learning(LSTM) with data compression

Baseline 시계열 예측 알고리즘 : ARIMA

1) ARMA(p, q) 모델

- AR(p)모델과 MA(q)모델을 결합하여 ARMA(p,q) 모델 도출
- 시계열의 각 값을 과거 p개 관측값과 q개 오차를 이용하여 예측

$$X(t) = (X_{t-1} * w_{11}) + (X_{t-2} * w_{12}) + (e_{t-1} * w_{21}) + (e_{t-2} * w_{22}) + b + (e_t * u)$$

그림. ARMA 수식

2) ARIMA(p, d, q) 모델

- 전통적인 시계열 모형으로, 과거 시점의 수치를 반영한 회귀 및 현시점의 오차 변동 및 차분을 고려하는 모델
- ARMA모델에 차분 과정 추가
- 시계열 데이터를 d회 차분하고 결과값은 과거 p개 관측값과 q개 오차에 의해 예측되는 모델
- 결과값은 비차분화(un_differenced)과정을 거쳐 최종 예측값으로 변환

$$\hat{y} = \mu + (w_{11} * y_{t-1} + \cdots + w_{1p} * y_{t-p}) - (w_{21} * e_{t-1} + \cdots + w_{2p} * e_{t-p})$$

그림. ARIMA 수식

Modeling(ARIMA / AUTO ARIMA)

- 적용 알고리즘 : ARIMA
- 실험 조건
  - ✓ sample id = 7657
  - ✓ coin_num = 9
  - ✓ input_data = t -1380 ~ t-1 (23시간 동안의 분당 open price)
  - ✓ output_data = t ~ t+119 (향후 2시간 동안의 분당 open price)

```
model = ARIMA(x_series, order=(2,0,2))
fit   = model.fit()
pred_by_arima = fit.predict(1381,1380+120, typ='levels')
```

그림. ARIMA 모델링 코드

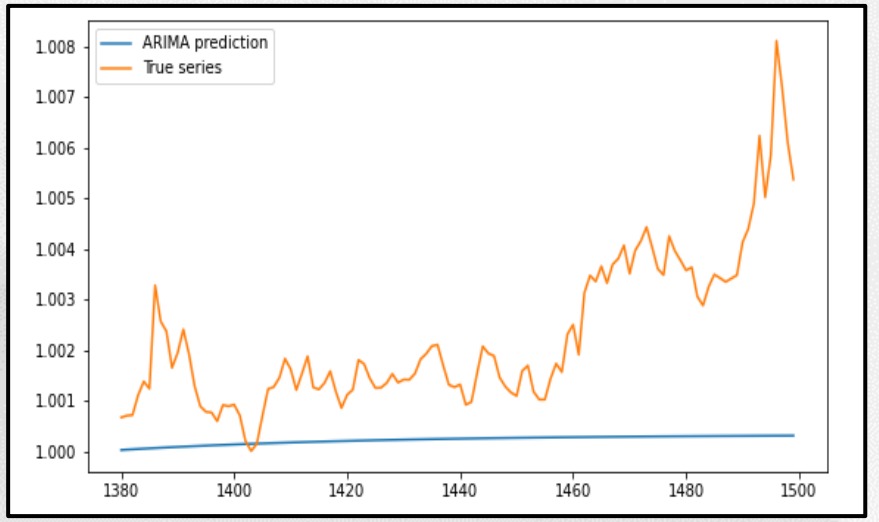


그림. ARIMA prediction

- 결과 : 임의의 파라미터 ARIMA modeling 은 예측이 불가
- 다음 단계 : ARIMA 파라미터 최적화

Modeling(ARIMA / AUTO ARIMA)

- 실험 조건 동일
- Auto ARIMA library 통한, 파라미터 최적화

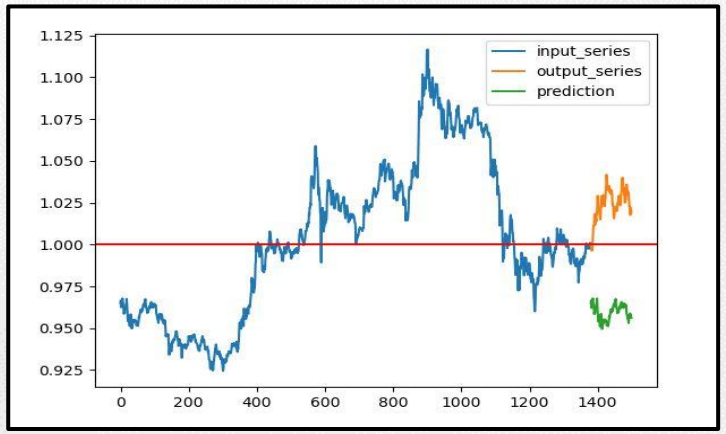


그림. Auto ARIMA prediction

- 결과 : 기존 예측 보다, 선형성이 완화되긴 하였으나, 예측력 부족
- 문제 원인 : 주가 차트는 비정상성(non-stationary)를 갖는 시계열
  → 비정상성 때문에 미래 예측 어려움

- 다음 단계 : 정상성을 갖는 시계열로 바꾸기 위해 차분 적용
  - 차분은 비정상인 시계열을 정상성을 나타내도록 만드는 방법
  - 연속적인 관측값들의 차이를 계산하는 것
  - 한계점 : 정수 차원의 차분 시계열은 정상성을 갖지만 원 시계열이 갖고 있는 메모리를 지워버린다.

  → 실수 차원의 차분 시계열은 메모리 보존이 가능하면서 정상성을 갖는 시계열 생성 가능 (Advance in Financial Machine Learning 참고)

실수 차분 방법(Fractional Differencing method)

$$(1-B)^d y_t.$$

$$7)\ (1-B)^d = 1 - dB + \frac{d(d-1)}{2!}B^2 - \frac{d(d-1)(d-2)}{3!}B^3 + \cdots$$

그림. 이항급수 적용 차분 연산식

$$8)\ (1-B)^{0.2} = 1 - 0.2B + \frac{0.2(0.2-1)}{2!}B^2 - \frac{0.2(0.2-1)(0.2-2)}{3!}B^3 + \cdots$$

$$9)\ (1-B)^{0.2} = 1 - 0.2B - 0.08B^2 - 0.048B^3 - 0.0336B^4 \cdots$$

$$10)\ (1-B)^{0.2}x_t = x_t - 0.2x_{t-1} - 0.08x_{t-2} - 0.048x_{t-3} - 0.0336x_{t-4} \cdots$$

$$11)\ (1-B)^{0.4}x_t = x_t - 0.4x_{t-1} - 0.12x_{t-2} - 0.064x_{t-3} - 0.0416x_{t-4} \cdots$$

$$12)\ (1-B)^{0.6}x_t = x_t - 0.6x_{t-1} - 0.12x_{t-2} - 0.056x_{t-3} - 0.0336x_{t-4} \cdots$$

$$13)\ (1-B)^{0.8}x_t = x_t - 0.8x_{t-1} - 0.08x_{t-2} - 0.032x_{t-3} - 0.0176x_{t-4} \cdots$$

$$14)\ (1-B)^{1.0}x_t = x_t - 1.0x_{t-1} - 0x_{t-2} - 0x_{t-3} - 0x_{t-4} \cdots$$

x_{t-1}의 계수가 1에 가까워 지고 있다.

d가 작을수록 먼 과거의 데이터가 반영되고 있다.

그림. 실수 차원의 차분 연산식

- 즉, d가 1보다 작아지면 x(t-2) 이후의 모든 데이터들이 차분 시계열 반영
  → 과거 기억 보존
- d가 0에 가까워질수록 보존되는 메모리 양은 증가(non-stationary 특성 커짐)
- d가 1에 가까워질수록 보존되는 양은 감소, (stationary 특성 커짐)
- d는 메모리와 정상성의 정도를 조절하는 변수

실수 차분 전처리 전후 차이

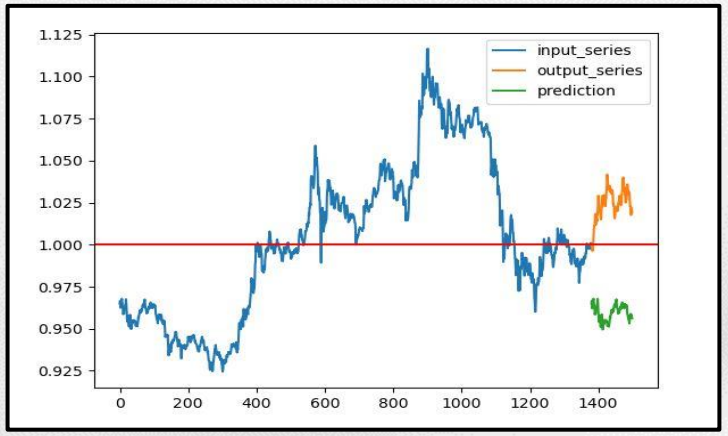


그림. Auto ARIMA prediction

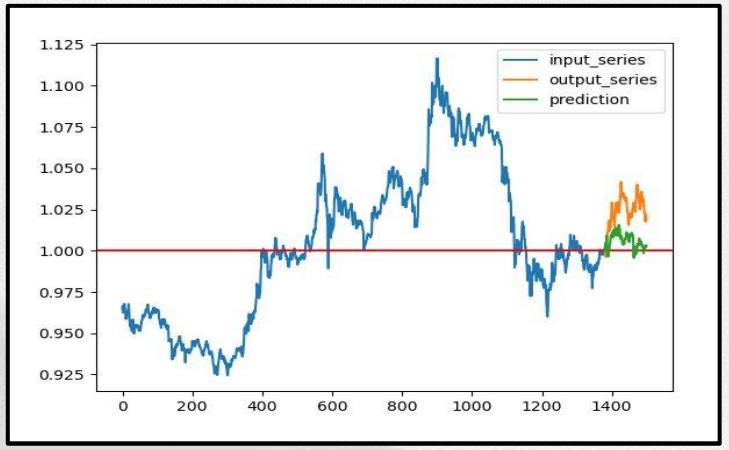


그림. 실수 차분 후 prediction

- 결과 : 부분적으로 예측력이 향상됨

Modeling(Prophet, Neural Prophet)

Facebook's Prophet & Neural Prophet

- AR-Net과 같은 딥 러닝 모델을 사용하는 Prophet의 확장판
- 장점은 Prophet의 간단한 API를 제공 보다 정교한 딥러닝 모델에 액세스 가능
- Model parameter 소개
    1. G(t) : Linear Growth(+Change Point), 선형성
    2. S(t) : Seasonality, 푸리에 패턴 주기
    3. H(t) : Holidays, 이벤트 효과

$$y(t) = g(t) + s(t) + h(t) + \epsilon_i$$

그림. Prophet 알고리즘 수식

```python
# pprophet 모델 학습
prophet = Prophet(seasonality_mode='multiplicative',
                  yearly_seasonality=False,
                  weekly_seasonality=False, daily_seasonality=True,
                  changepoint_prior_scale=0.06)
prophet.fit(x_df)

future_data = prophet.make_future_dataframe(periods=120, freq='min')
forecast_data = prophet.predict(future_data)
```

그림. Prophet python code

Modeling(Prophet, Neural Prophet)

- 적용 알고리즘 : Prophet
- 실험 조건
  - ✓ 9번 코인만을 가지고 진행
  - ✓ input_data = t -1380 ~ t-1 (23시간 동안의 분당 open price)
  - ✓ output_data = t ~ t+119 (향후 2시간 동안의 분당 open price)
  - ✓ Seasonality mode = Multiplicative

```
prophet= Prophet(seasonality_mode='multiplicative',
                 yearly_seasonality='auto',
                 weekly_seasonality='auto', daily_seasonality='auto',
                 changepoint_range=0.9,
                 changepoint_prior_scale=0.1  # 오버피팅, 언더피팅을 피하기 위해 조정
                )

prophet.add_seasonality(name='first_seasonality', period=1/12, fourier_order=7) # seasonality 추가
prophet.add_seasonality(name='second_seasonality', period=1/8, fourier_order=15) # seasonality 추가

prophet.fit(x_df)

future_data = prophet.make_future_dataframe(periods=120, freq='min')
forecast_data = prophet.predict(future_data)
```

그림. 하이퍼파라미터 수정 후 Prophet code

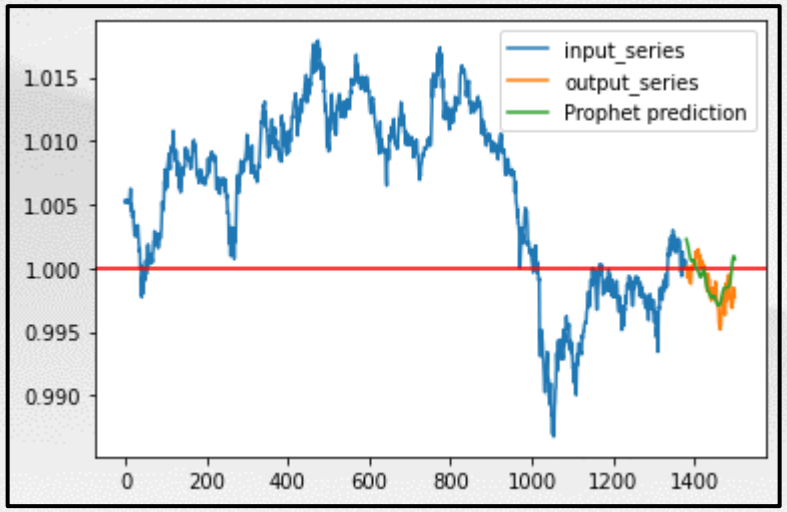


그림. Prophet prediction

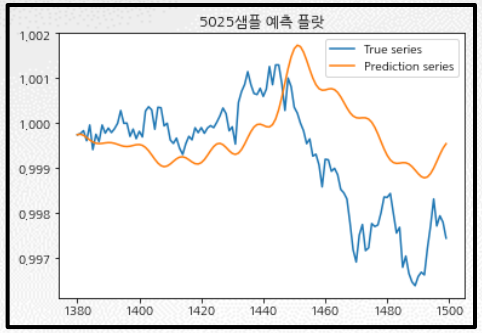
range = Change Point 설정 가능 범위 (이상치탐지)

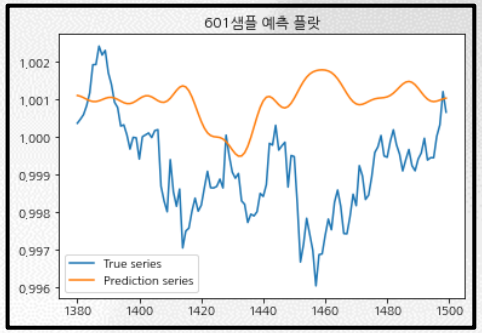
scale = trend 유연성 조절(trend 변화 반영 정도) → Prophet은 선형적이지 않고 일반적으로 유추 가능성

Modeling(Prophet, Neural Prophet)

- 적용 알고리즘 : Neural Prophet
- 실험 조건
  - ✓ 9번 코인만을 가지고 진행
  - ✓ Input_data = t -1380 ~ t-1 (23시간 동안의 분당 open price)
  - ✓ Output_data = t ~ t+119 (향후 2시간 동안의 분당 open price)
  - ✓ Seasonality mode = Multiplicative

- HyperparameterGird Research
  1. Seasonality add research
  2. Model capacity research





- 결과 : Best parameter를 적용해보았으나, 크게 개선되지 못함
- 다음 단계 : 데이터 전처리 및 다른 모델 시도

```
params_grid = {
'seasonality_fourier_order' : [[1, 5], [5, 10], [10,15], [15, 20]],
'seasonality_periods' : [[1/12, 1/8], [1/12, 1/6], [1/12, 1/4], [1/24, 1/16], [1/24, 1/12], [1/24, 1/8]]
}

grid = ParameterGrid(params_grid)
cnt = 0
for parameters in grid:
    cnt = cnt+1

print('Total Possible Models',cnt)
```

```
Total Possible Models 24
```

	MSE*10E5	PARAMETERS
0	183.0543	{'seasonality_fourier_order': [5, 10], 'seasonality_periods': [0.041666666666666664, 0.0833
1	186.5197	{'seasonality_fourier_order': [5, 10], 'seasonality_periods': [0.08333333333333333, 0.125]}
2	189.1772	{'seasonality_fourier_order': [1, 5], 'seasonality_periods': [0.041666666666666664, 0.125]}

그림. 주기성 파라미터 그리드 리서치

```
params_grid = {'n_changepoints':[5, 10, 15, 20], #default 5
               'changepoints_range' : [0.9, 0.95, 1.0], #defalut 0.8
               'num_hidden_layers' : [0, 1, 2],
               'd_hidden' : [10, 30, 50],

}

grid = ParameterGrid(params_grid)
cnt = 0
for parameters in grid:
    cnt = cnt+1

print('Total Possible Models',cnt)
```

```
Total Possible Models 108
```

	MSE*10E5	PARAMETERS
0	90.08979	{'changepoints_range': 0.95, 'd_hidden': 30, 'n_changepoints': 20, 'num_hidden_layers': 1}
1	90.95572	{'changepoints_range': 0.95, 'd_hidden': 30, 'n_changepoints': 20, 'num_hidden_layers': 2}
2	91.00896	{'changepoints_range': 0.95, 'd_hidden': 10, 'n_changepoints': 20, 'num_hidden_layers': 1}
3	91.2683	{'changepoints_range': 0.95, 'd_hidden': 10, 'n_changepoints': 20, 'num_hidden_layers': 0}

그림. 모델 Capacity 그리드 리서치

Modeling(Pytorch RNN)

- 적용 알고리즘 : Recurrent Neural Network
- 실험 조건
  - ✓ 9번 코인으로만 진행
  - ✓ Input_data = t -255 ~ t-1 (255분 간의 all data)
  - ✓ Output_data = t ~ t+119 (향후 120분의 open price)

- 결과 : 자기회귀 시 오차 누적 문제
- 다음 단계 : time-series 압축

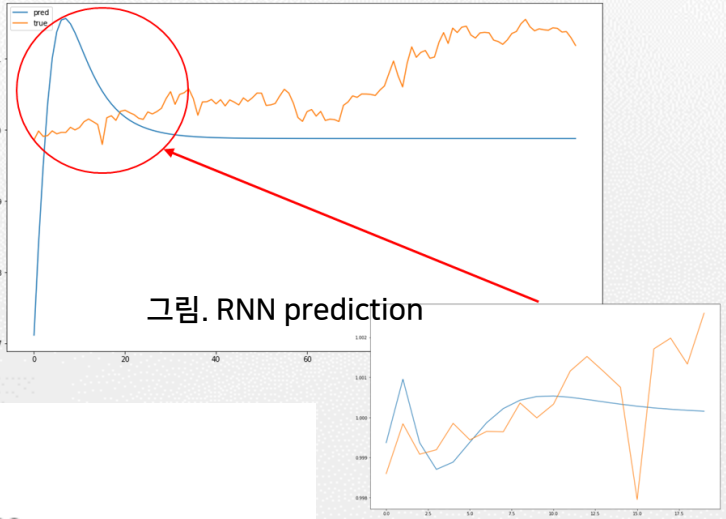
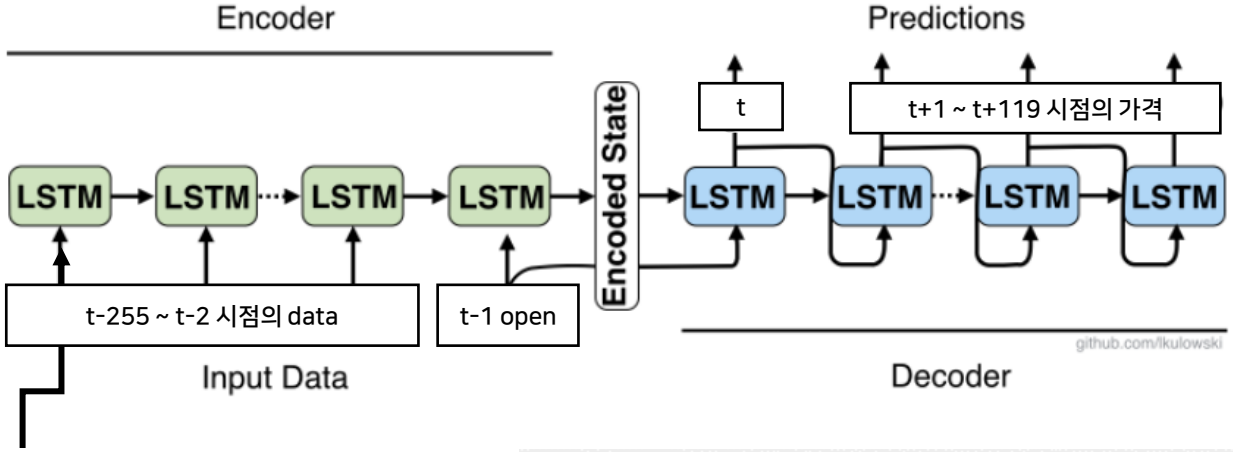


그림. RNN prediction

Encoder&Decoder 구조



open	high	low	close	volume	quote_av	trades	tb_base_av	tb_quote_av
0.983614	0.983614	0.983128	0.983246	0.003158	0.003609	0.005328	0.003121	0.003071

그림. Input 데이터 예시

Modeling(Pytorch RNN)

- 기존 model의 문제점 : n = 예측에 사용되는 sequence의 길이가 너무 길 → 자기 회귀 구간 압축

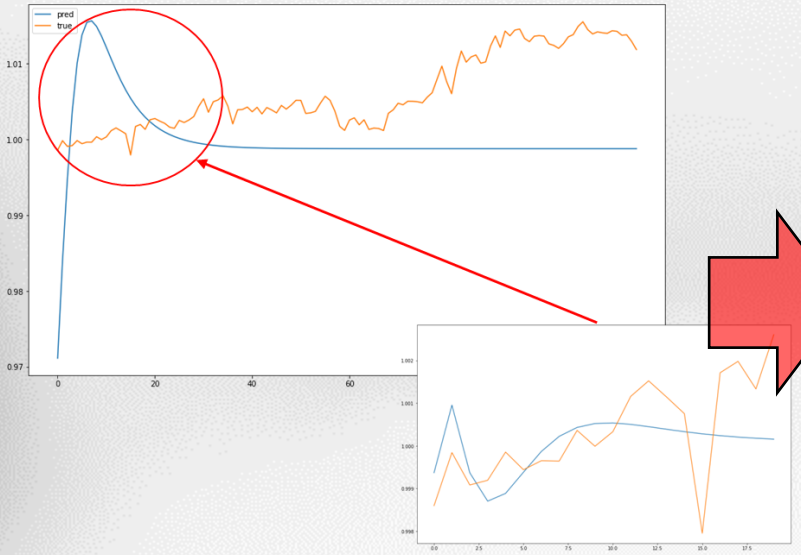
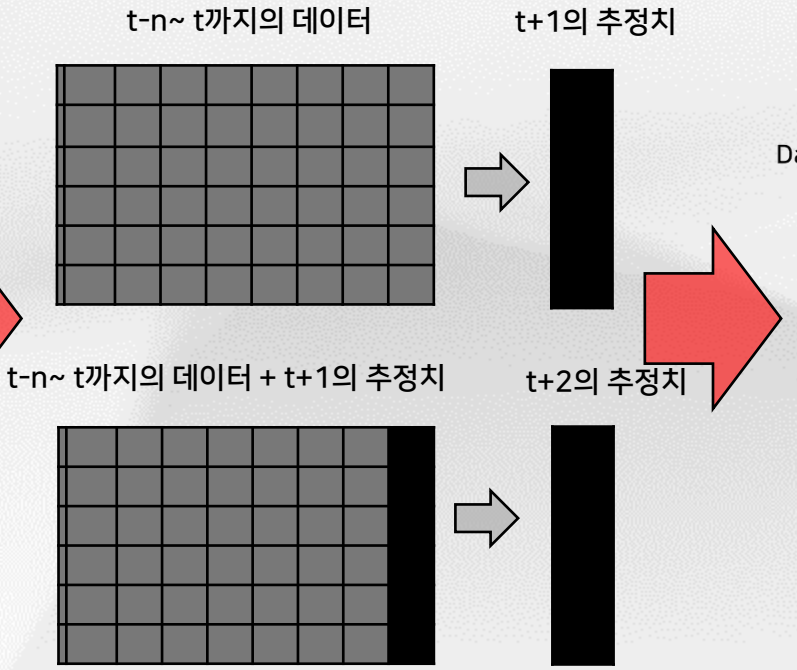


그림. RNN prediction



Data 압축 방법

- 1분 데이터를 t분 데이터로 압축 시 (i는 0 ≤ t*(i+1) ≤ 1380인 0 이상의 정수)
- open : t*I 시점의 open 값
- high : t*(i+1)-1 시점 동안 가장 높은 high 값
- low : t*(i+1)-1 시점 동안 가장 높은 low 값
- close : t*(i+1)-1 시점의 close값
- 기타 데이터(volume, quote_av, trades, tb_base_av, tb_quote_av)
  t*i : t*(i+1)-1 시점 동안의 합

오차가누적되어 긴 sequence(=120)을 예측하는동안결과가수렴해버린다.

Modeling(keras LSTM)

- 적용 알고리즘 :RNN(LSTM)
- 실험 조건 동일
  - ✓ 9번 코인만을 가지고 진행
  - ✓ Input_data = t -1380 ~ t-1 (23시간 동안의 분당 open price)
  - ✓ Output_data = t ~ t+119 (향후 2시간 동안의 분당 open price)
  - ✓ 가격 데이터를 6분 마다의 대푯값으로 압축

  (2시간을 6분으로 압축하면, 120개의 yhat → 20개의 yhat)
  - ✓ Target length에 맞춰서 반복 예측

```python
#모델
model = Sequential()
model.add(LSTM(128, return_sequences=True, input_shape= (x_train.shape[1],x_train.shape[2] )))
model.add(LSTM(64, return_sequences=False))
model.add(Dense(25))
model.add(Dense(1))
```

1. Yhat 개수정의

2. 반복 예측

```python
look_ahead = 20
yhat=[]
x = signaldata[-look_back:,variable]
x=x.reshape(1,look_back,1)
for i in range(look_ahead):
  fc = model.predict(x)
  yhat.append(fc)
  x=np.append(x,fc)
  x=np.delete(x,0)
  x=x.reshape(1,6,1)
```

그림. RNN 모델 및 자기 회귀 함수식

```python
# n분봉으로 나누는 함수
# idxsize= _array[0], time_size = _array[1], time_split = 몇분봉으로 나눌건지, arrayy = array명
def time_split(input_array, split_size = 6):

    # origin size define
    index_size = input_array.shape[0]
    origin_time_size = input_array.shape[1]
    variable_size = input_array.shape[2]

    # new array size define
    new_time_size = int(origin_time_size/split_size) # 1380 / 6
    new_array = np.zeros((index_size, new_time_size, variable_size))

    for idx in range(index_size):
        for time_idx in range(new_time_size):


            first_time_idx = time_idx * split_size
            last_time_idx = ((time_idx+1) * split_size) -1

            new_array[idx, time_idx, 0] = input_array[idx, first_time_idx, 0] #coin_num
            new_array[idx, time_idx, 1] = input_array[idx, first_time_idx, 1] #open
```

그림. 데이터 6분 압축 python code

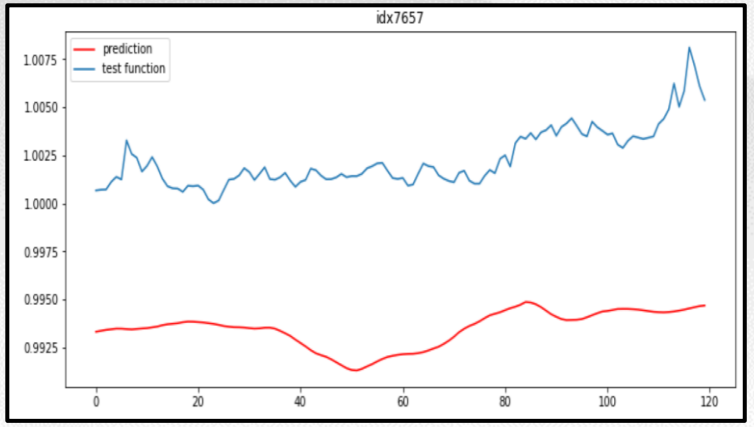


그림. 전처리 후 RNN prediction

Data smoothing

Smoothing 적용 이유 : 주기성이 부족해, regression이 되지 않음
→ 기존 데이터는 너무 진폭이 심해서 모델이 regression을 하기 어렵다고 판단

- smoothing method 1 : simple exponential smoothing
- smoothing method 2 : moving average

```python
def simple_exponetial_smoothing(arr, alpha=0.3):

    y_series = list()

    for temp_arr in arr:
        target_series = temp_arr[:, 1].reshape(-1) # open col is 1 index

        smoother = SimpleExpSmoothing(target_series, initialization_method="heuristic"
        smoothing_series = smoother.fittedvalues

        y_series.append(smoothing_series)

    return np.array(y_series)


def moving_average(arr, window_size = 10):

    #length = ma 몇 할지
    length = window_size
    ma = np.zeros((arr.shape[0], arr.shape[1] - length, arr.shape[2]))

    for idx in range(arr.shape[0]):
        for i in range(length, arr.shape[1]):
            for col in range(arr.shape[2]):
                ma[idx, i-length, col] = arr[idx,i-length:i, col].mean() #open

    return ma[:, :, 1] # open col is 1
```

그림. smoothing python code

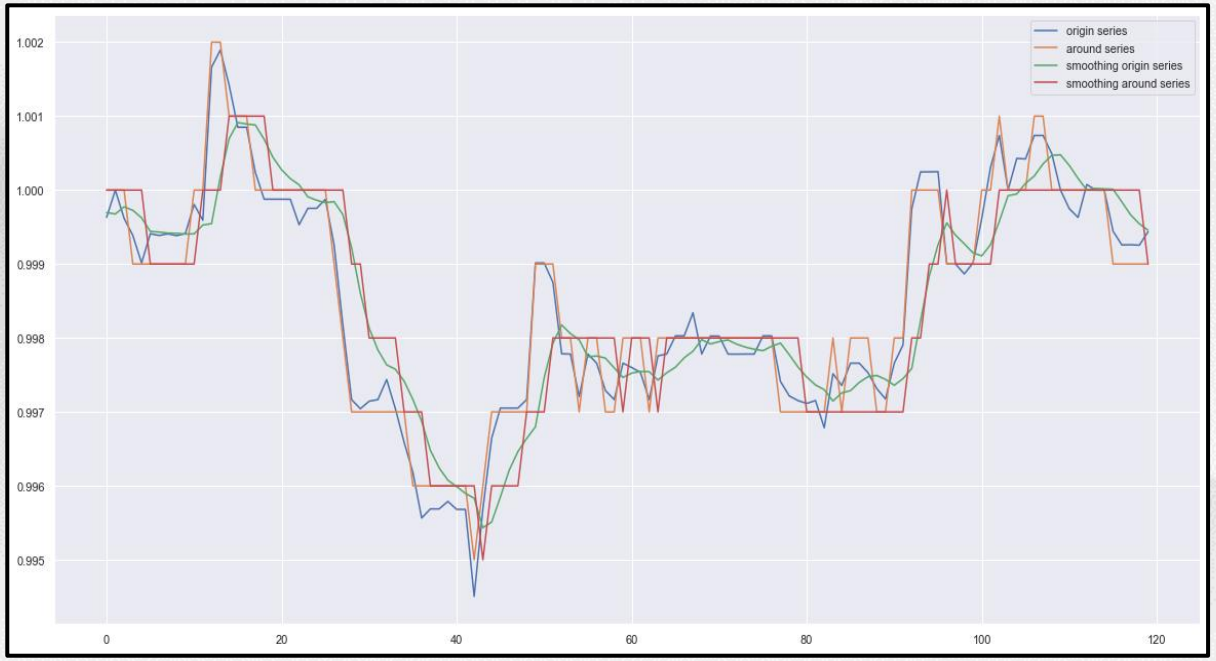


그림. price data smoothing plot

Data discretizing

Discretizing 적용 이유 : open data range가 각기 다름 또한, outlier cases 존재
→ true y을 prediction 하는 것보다 y 값의 패턴 양상만을 학습하는 방법으로 바꿔 driving

- discretize method : KBinsdiscretizer library(in scikit-learn)

```
from sklearn.preprocessing import KBinsDiscretizer
kb = KBinsDiscretizer(n_bins=10, strategy='uniform', encode='ordinal')
kb.fit(open_y_series)
#  이때 `bin_edges_` 메소드를 이용하여 저장되어진 경계값을 확인할 수 있다.
print("bin edges :\n", kb.bin_edges_ )
```

그림. kbinsdiscretizer python code

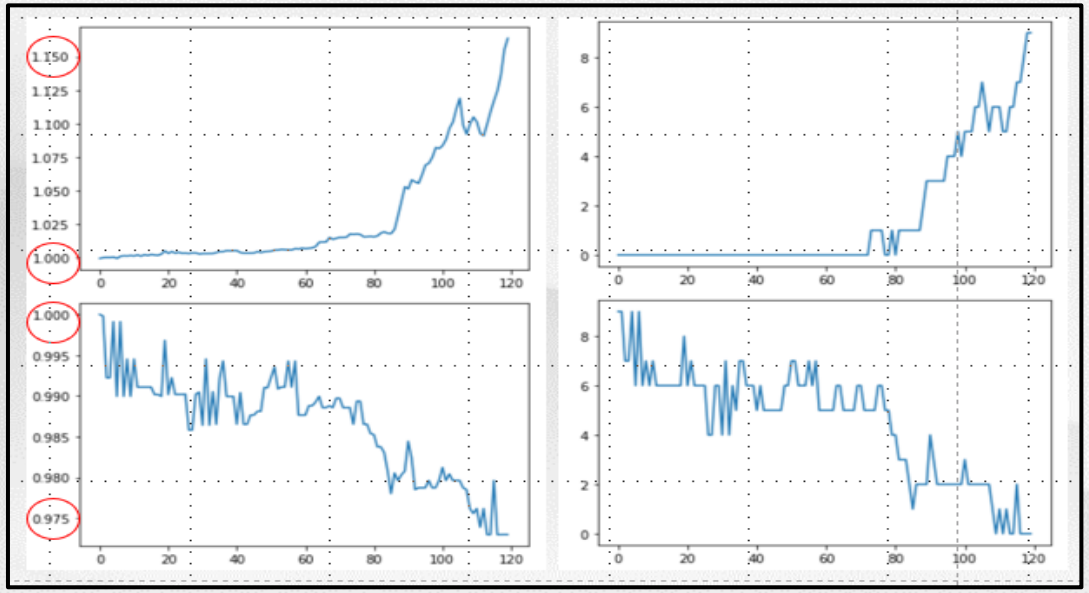


그림. kbinsdiscretizer before & after plot

Data log normalization

- 데이터 인풋 시 open data 이외에 다른 feature을 같이 활용하기 위해, 다음과 같은 방법으로 normalization을 적용
- 일반적인 scikit-learn normalizer은 바로 사용하기에는 대회 내에서 1380분일 때의 open price를 1로 수정하면서 전반적인 전처리가 이미 한번 된 상태이기 때문에 해당 방법을 사용

```
data = data.apply(lambda x: np.log(x+1) - np.log(x[self.x_frames-1]+1))
```

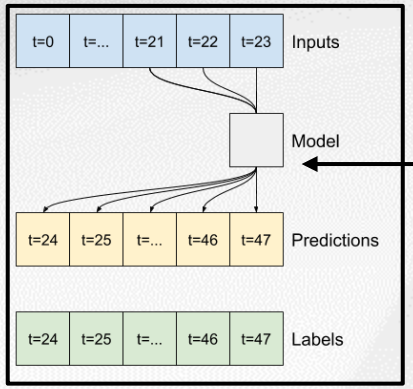
그림. Log normalization python code

데이터 스무딩 후 모델링

- 전처리 이후에도 크게 변화가 없어, 기존 데이터 시퀀스(1380)이 길어 모델이 학습하기 어렵다고 판단

  → Conv1d로 특정 구간에 대해 특징을 추출하고 이를 LSTM에 반영하여 driving

- 실험 조건
  1. time-size = 1380
  2. input features : open, high, low, close, volume …
  3. label feature : open




```python
def forward(self, X):

    # input은 (Batch, Feature dimension, Time_step)순
    output = F.relu(self.conv1(X))
    output = self.pooling1(output)
    output = F.relu(self.conv2(output))
    output = self.pooling2(output)
    # output = self.flatten(output)

    # [Batch_size, Seq_len, Hidden_size]
    # x_input.reshape(1, -1, self.output_dim
    # torch.Size([16, 32, 135])
    # torch.Size([16, 135, 32])

    output, self.hidden = self.lstm(output.reshape(args.batch_size, -1, 32))
    y_pred = self.linear(output[:, -1, :])

    return y_pred
```

그림. 모델 구성 도식화

데이터 스무딩 후 모델링 결과

- normalization이나 smoothing의 문제가 아닌, 애초에 데이터가 주기성이 없어서 샘플 별로 데이터를 regression 하는 방법의 방향이 틀림

- LSTM과 같은 RNN 계열의 모델들은 패턴을 학습하는 것으로, onestep이 아닌 multistep에서는 너무 동일한 결과를 출력하게 됨.

- 해당 문제를 특정 패턴을 학습하게 하기 위해서는, discretize시켜서 classification 문제로 접근하는 것도 하나의 방법(향후 시즌3에서 검토)

- regression 문제로 풀기 위해서는 일반적인 time-series forcasting 모델(ARIMA or Prophet)처럼 한 샘플 내 open data seriees를 가지고 onestep씩 학습 후 이를 target length(120min)만큼 loop하여 시도해야 함(향후 시즌3에서 검토)

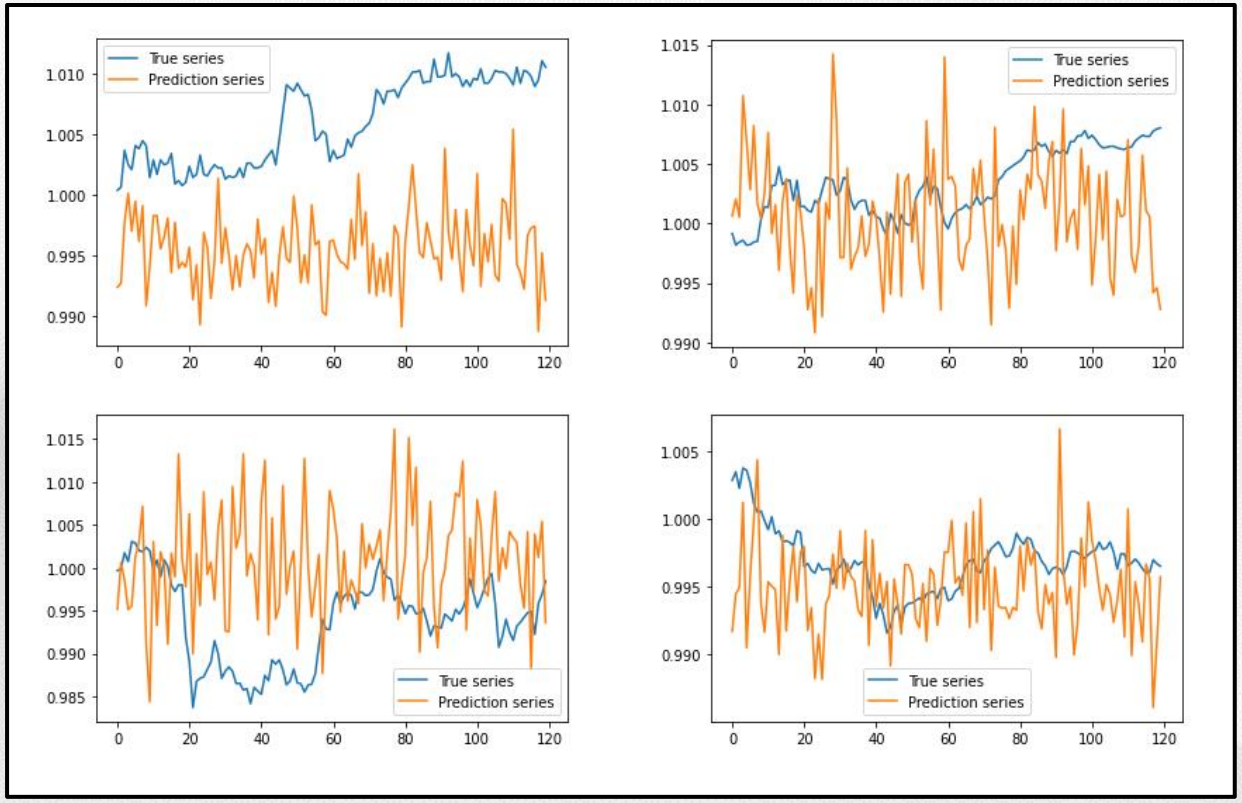


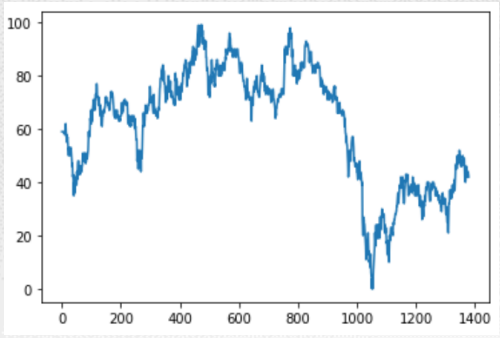
그림. Conv1d-LSTM prediction

데이터 계층화 후 모델링

1) 최고가의 index 분류 → classification problem

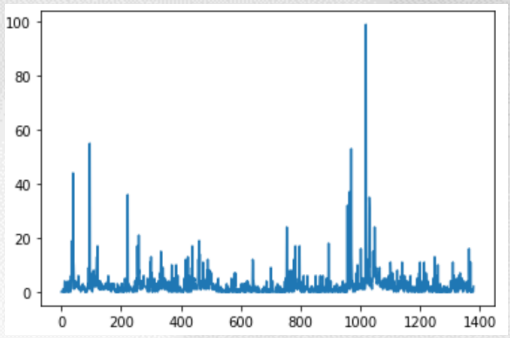
2) 계층화된 가격 데이터 Auto-regression → regression problem

<input data>



일정 시간 동안의 가격 데이터

(345~1380)



일정 시간 동안의 거래량

(Volume)

<output data>

0 = [1,0,0,0,0, ⋯ ,0]

1 = [0,1,0,0,0, ⋯ ,0]

2 = [0,0,1,0,0, ⋯ ,0]

:

가장 높은 가격의 time index를

벡터화한 확률 값

데이터 계층화 전처리 진행과정

```python
def kbindiscreter(input_array):
  kb = KBinsDiscretizer(n_bins=10, strategy='uniform', encode='ordinal')
  processed_data = np.zeros((input_array.shape[0], input_array.shape[1], 1))
  for i in range(input_array.shape[0]):
    processing_array = input_array[i,:,1]
    kb.fit(processing_array.reshape(input_array.shape[1],1))
    processed_fit = kb.transform(processing_array.reshape(input_array.shape[1],1))
    processed_data[i,:,:] = processed_fit
  return processed_data
```

그림. 데이터 계층화 python code

```python
model = Sequential()
model.add(Conv1D(64,10, activation='relu', input_shape=(X_traina.shape[1],X_traina.shape[2] )))
model.add(Conv1D(32,5, activation='relu'))
model.add(Conv1D(32,4, activation='relu'))
model.add(Conv1D(16,4, activation='relu'))
model.add(MaxPooling1D(4))
model.add(Flatten())
model.add(Dense(256,activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(units=120, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
history = model.fit(X_traina,y_traina,epochs=100,validation_split=0.1,shuffle=True)
```

그림. Keras Conv1d model code

데이터 계층화 전처리 진행과정

```
k=7000
n_past=1380
ddd = model.predict(X_traina[k:k+100,:,:].reshape(100,n_past,2))


kk = np.zeros((100,120))
for x in range(100):
  for y in range(120):
    if np.argmax(ddd[x]) ==y:
      kk[x,y] =1


for i in range(100):
  print(i+1)
  plt.plot(train_y_array[i+k,:,1].reshape(-1),label = "true")
  plt.axvline(np.argmax(kk[i]), c ='red')
  plt.show()
```

그림. 최고가 시점 분류 모델 시각화 코드

```
#train set auto regression

aa =x_traina[800,:,:]
yhat=[]
x= aa.reshape(1,24,100)


for i in range(100):
  x=x.reshape(1,24,100)
  fc = model.predict(x)
  yhat.append(np.argmax(fc))
  fd = np.zeros((1,100))
  for j in range(100):
    if j==np.argmax(fc):
      fd[0,j] = 1

  x=x.reshape(24,100)
  x=np.vstack([x,fd])
  x=x[1:,:]


bb = []
for x in range(100):
  bb.append(np.argmax(y_traina[800+x]))


# train set auto regression plotting
plt.plot(yhat,label="prediction")
plt.show()
plt.plot(bb, label="real")
plt.show()
plt.plot(yhat,label="prediction")
plt.plot(bb, label="real")
```

```
#val set auto regression

aa =x_traina[61395,:,:]
yhat=[]
x= aa.reshape(1,24,100)


for i in range(100):
  x=x.reshape(1,24,100)
  fc = model.predict(x)
  yhat.append(np.argmax(fc))
  fd = np.zeros((1,100))
  for j in range(100):
    if j==np.argmax(fc):
      fd[0,j] = 1

  x=x.reshape(24,100)
  x=np.vstack([x,fd])
  x=x[1:,:]


bb = []
for x in range(100):
  bb.append(np.argmax(y_traina[61395+x]))


plt.plot(yhat,label="prediction")
plt.show()
plt.plot(bb, label="real")
plt.show()
plt.plot(bb, label="real")
plt.plot(yhat,label="prediction")
```

그림. 가격 계층 자기 회귀 모델 시각화 코드

데이터 계층화 후 모델링 결과

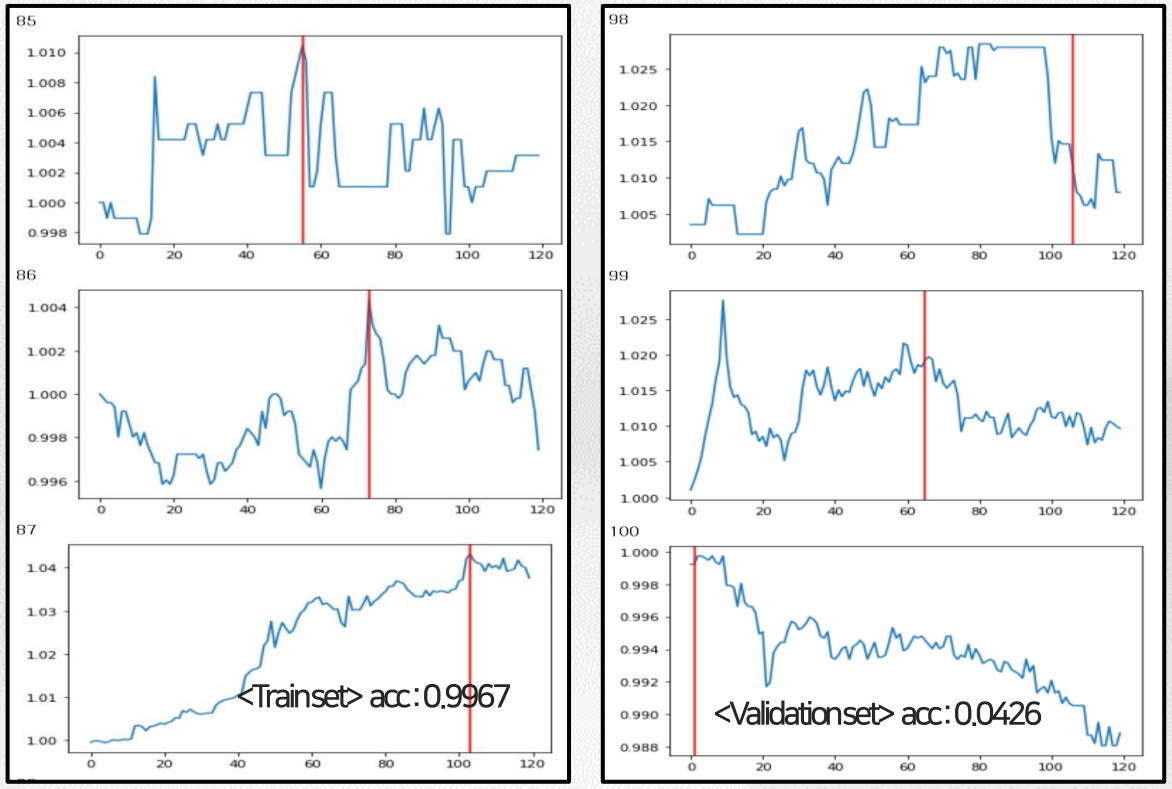


그림. 최고가 시점 분류 모델

→ Train set에서 학습은 효과적이나, validation set에서는 특정 패턴에서만 제대로 된 학습이 이루어짐

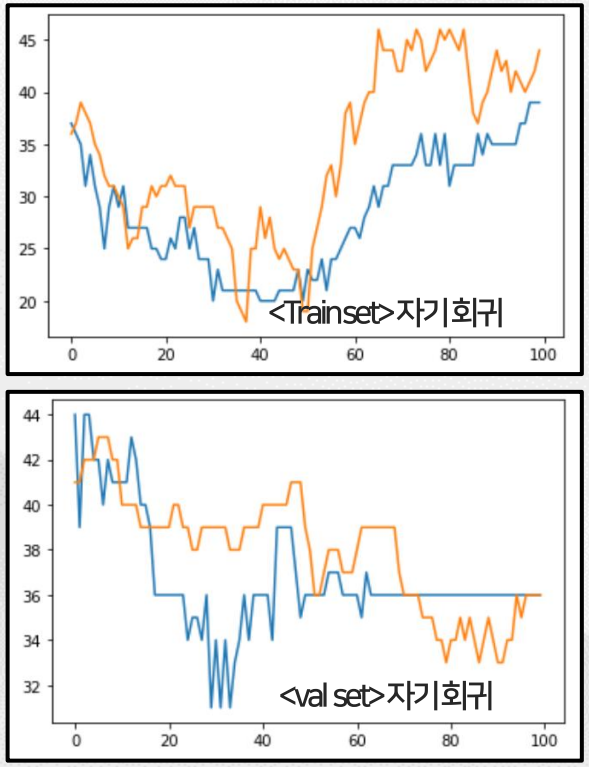


그림. 가격 계층 자기회귀 모델

→ 다른 모델과 달리 긴 시간의 자기회귀에도 어느정도 비슷하게 따라감.
Validation set에서는 추세는 따라가는 것처럼 보이나 중간에 수렴되어 더 많은 학습량이 필요해 보임.

투자 시뮬레이션 프로그래밍

- Arrary_to_submission function : Input = Prediction array / output = 매수 시점 index
- COIN function : predicted 된 매수 시점을 기준으로 투자 시뮬레이션

```python
def array_to_submission(pred_array, start_idx=0, increase_rate = 1.04):
    submission = np.zeros((pred_array.shape[0],3))

    for x in range(int(pred_array.shape[0])):
        #시작 인덱스 설정
        idx = int(start_idx + x)
        submission[x,0] = idx
        #예측값의 최고가에 따른 buy_quantity 결정
        high_price = np.max(pred_array[x,:])/pred_array[x,0]
        if high_price >=increase_rate:
            submission[x, 1] = 1
        #예측값의 최고가의 time_number
        sell_time = int(np.argmax(pred_array[x,:]))
        submission[x, 2] = sell_time
    submission = pd.DataFrame(submission)
    submission.columns = ['sample_id','buy_quantity', 'sell_time']
    return submission
```

그림. Array_to_submission python code

```python
def COIN(y_array, submission):
    # 2차원 데이터프레임에서 open 시점 데이터만 추출하여 array로 복원
    # sample_id정보를 index에 저장
    # y_array= df2d_to_answer(y_df)

    # 초기 투자 비용은 10000 달러
    total_money = 10000
    total_money_list = []


    for row_idx in range(submission.shape[0]):
        sell_time = int(submission.loc[row_idx, 'sell_time'])
        buy_price = y_array[row_idx, 0]
        sell_price = y_array[row_idx, sell_time]
        buy_quantity = submission.loc[row_idx, 'buy_quantity'] * total_money
        residual = total_money - buy_quantity
        ratio = sell_price / buy_price
        total_money = buy_quantity * ratio * 0.9995 * 0.9995 + residual
        total_money_list.append(total_money)

    return total_money, total_money_list
```

그림. COIN python code

결과

- 1166팀 중 28위로 대회 종료
- 사용 모델 : Prophet
- 매수 기준 : 115% 이상 상승


```python
m = Prophet(
            yearly_seasonality=False,
            weekly_seasonality=True,
            daily_seasonality=True,
            changepoint_prior_scale=0.01,
            changepoint_range= 1,
            interval_width=0.9,
            seasonality_mode='multiplicative'

)

m.add_seasonality(name='first_seasonality', period=1/24, fourier_order=5)
m.add_seasonality(name='second_seasonality', period=1/12, fourier_order=10)

m.fit(x_df)

future = m.make_future_dataframe(periods=120, freq='min')
forecast= m.predict(future)
```

그림. Prophet model python code

```python
def array_to_submission(x_array, pred_array):
    # 입력 x_arrry와 출력 pred_arry를 통해서
    # buy_quantitiy와 sell_time을 결정
    submission = pd.DataFrame(np.zeros([pred_array.shape[0],2], np.int64),
                              columns = ['buy_quantity', 'sell_time'])
    submission = submission.reset_index()
    submission.loc[:, 'buy_quantity'] = 0.1

    buy_price = []
    for idx, sell_time in enumerate(np.argmax(pred_array, axis = 1)):
        buy_price.append(pred_array[idx, sell_time])
    buy_price = np.array(buy_price)
    # 115% 이상 상승한하고 예측한 sample에 대해서만 100% 매수
    submission.loc[:, 'buy_quantity'] = (buy_price > 1.15) * 1
    # 모델이 예측값 중 최대 값에 해당하는 시간에 매도
    submission['sell_time'] = np.argmax(pred_array, axis = 1)
    submission.columns = ['sample_id','buy_quantity', 'sell_time']
    return submission
```

그림. Buy criterion python code

향후 진행 방향

- Modeling – Data discretize & classification driving
  - ✓ 해당 문제를 최고점 패턴 분류 모델로 변형하여 학습
  - ✓ y값 구간 내 open price 최고점을 labling
  - ✓ 모델은 1380분 간의 input 패턴에 따라, 최고점인 lable을 분류
  - ✓ Pytorch Conv1d + bidirectional LSTM

- Modeling – Open data series regression by one sample Seasonality add research
  - ✓ 일반적인 time-series forcasting 모델(ARIMA or Prophet)처럼 한 샘플 내 open data 만으로 one-step씩 학습 후 Auto-Regressioning
  - ✓ smoothing 및 fractional differecing, log normalization 적용
  - ✓ moving average 재적용
  - ✓ 특정 분류 불가할 것 같은 outlier data sample remove
  - ✓ Pytorch Conv1d + bidirectional LSTM