

빅데이터와 금융자료 분석

BAF660 Final Team Project



Team 2

김진우 20223868

류용옥 20223872

민동효 20194094

반유정 20223878

성상훈 20213844

이선우 20223888

목차

1. Introduction

2. Data Preprocessing

3. Feature Engineering

4. Forecasting Models

5. Ensemble

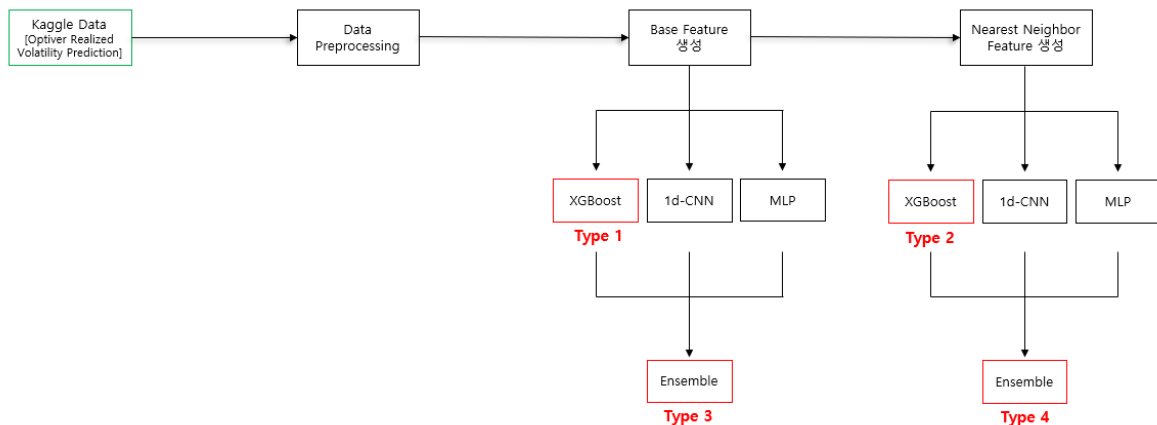
6. Conclusion

7. Reference



1. Introduction

저희 팀은 미래 변동성 예측이라는 주제로 나온 Kaggle의 competition data를 사용해 팀 프로젝트를 진행하였습니다. 아래는 저희 팀이 진행한 예측의 전체 pipeline입니다.



Kaggle Data

데이터 선정시 고려한 사항은 다음과 같습니다.

1. 수업에서 학습한 XGBoost를 적용할 수 있는 문제인가?
2. 금융과 관련된 주제인가?
3. 해당 분석결과를 토대로 향후에도 개선/발전시킬 수 있는 주제인가?
4. Real Data를 바탕으로 하는가?

이를 모두 충족하는 데이터셋을 탐색한 결과, 저희팀은 네덜란드 암스테르담에 위치한 프랍 트레이딩 전문 회사인 **Optiver**가 **Kaggle**에서 주최한 미래 변동성 예측 대회에 데이터를 선택하게 되었습니다.

해당 데이터를 통한 예측은 실습에서 계속해서 진행했던 분류문제가 아닌 회귀분석 문제인 것을 감안, 수업에서 다른 sampling 기법이나 feature를 속아내기 위한 방법(Feature Selection)등이 일부 제외되었습니다. 하지만 충분히 수업에서 학습한 내용을 바탕으로 예측력을 높이기 위한 여러 시도를 해볼 수 있는 참신한 주제라고 판단, 최종적으로 선정하게 되었습니다.

Data Preprocessing

데이터 전처리 단계에서 저희팀은 데이터 특성을 파악하여 회귀분석 문제에서 어떤 전처리가 필요한지에 대한 논의를 거쳤습니다.

1. 제공되는 기본 데이터는 모두 numerical 데이터로 이루어져 있습니다. 따라서 categorical 한 부분을 더미로 변환할 필요는 없지만, 예측을 위해 사용될 호가/체결 데이터를 활용하기 위해 Feature selection이 아닌 Feature Engineering 을 통한 새로운 Features를 생성하는 과정이 필요했습니다.
2. discrete, continuous numerical features를 생성하였으며, data의 구성 및 생성과정에 대한 내용은 이후 해당 챕터에서 자세하게 서술하였습니다.
3. 이후 Base features, Base+Nearest Neighbor features로 생성된 두개의 df를 각각 train_test_split을 통해 train, validation, test set으로 분리하여 아래에 서술된 Type별 모델의 예측에 활용하였습니다.

4 Types of Forecasting

아래 4가지 type의 모델 성능평가를 거쳐, RMSPE 값을 기준으로 최적모델과 features를 선택합니다.

Model \ Features	Base Features	Base Features + Nearest Neighbor Features
XGBoost	Type 1	Type2
Ensemble XGBoost + CNN + MLP	Type3	Type4

모델 및 특성별 유형구분

다음은 각 Type에 대한 개괄적인 설명입니다.

Type 1: Base features 를 통한 XGBoost 모델의 예측 성능 평가

수업에서 배운 내용을 토대로, 회귀분석 모델을 XGBoost로 선정하여 생성된 기본적인 features를 바탕으로 예측을 진행합니다.

Type 2: Base + Nearest Neighbor features 를 통한 XGBoost 모델의 예측 성능 평가

1. Type 1과 같은 XGBoost 모델을 사용하지만, 새로운 features를 생성하여 추가한 뒤, 예측 성능을 테스트합니다.
2. 구체적으로, KNN의 알고리즘을 이용하여 Manhattan distance를 사용해 구한 2개의 features (volatility, trade size sum) 의 최근접 이웃 인덱스를 산출합니다. 해당 인덱스를 바탕으로 10, 20, 40개의 이웃하는 인덱스들에 대한 stock_id features 들에 딕셔너리 형태로 mean, max, min, std와 같은 연산들을 적용합니다. 결과적으로 각각의 stock_id 별로 모든 time_id 에 대한 Nearest Neighbor features를 생성합니다.
3. 이렇게 생성된 Nearest Neighbor features를 기존의 Base features와 결합하여 XGBoost 모델을 적용한 예측을 수행합니다.

Type 3: Base features 를 통한 앙상블 모델의 예측 성능 평가

1. Base features 기반의 데이터에 예측모델 자체를 바꾸어 성능평가를 시도합니다. 추가로 시도되는 모델은 CNN, MLP 모델입니다.
2. 각 모델에 대한 성능평가를 위해, 각 모델별로 예측을 진행합니다.

3. 이후 CV기반의 stacking Ensemble 모델을 적용합니다. 적용을 위해 선택한 meta-learner는 Linear-Regression 모델입니다. 모델별 예측데이터를 모두 stacking 하여 최종 모델의 훈련 데이터로 사용합니다.

Type 4: Base + Nearest Neighbor features 를 통한 앙상블 모델의 예측 성능 평가

1. Base + Nearest Neighbor features 기반의 데이터에 예측모델을 바꾸어 성능평가를 시도합니다. 추가로 시도되는 모델은 CNN, MLP 모델입니다.
2. 각 모델에 대한 성능평가를 위해, 각 모델로 예측을 진행합니다.
3. 이후 CV기반의 stacking Ensemble 모델을 적용합니다. 적용을 위해 선택한 meta-learner는 Linear-Regression 모델입니다. 모델별 예측데이터를 모두 stacking 하여 최종 모델의 훈련 데이터로 사용합니다.



2. Data Preprocessing

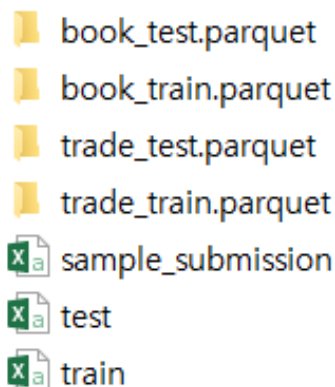
Data Set

Data set 설명

Kaggle에서 2021년 6월부터 7개월에 걸쳐 진행된 대회 데이터 활용하였습니다. 해당 대회의 주제는 '주가 변동성 예측(Optiver Realized Volatility Prediction)' 으로, 주식시장에서 주가의 단기 변동성을 머신러닝 알고리즘으로 예측하는 것입니다.

자세히 말하면, Optiver사에서 제공하는 1억개 이상의 데이터에는 각기 다른 섹터에 속한 주식의 호가, 체결 데이터가 포함되어 있습니다. 참가자는 제공된 데이터를 통해 머신러닝 모델을 학습한 뒤, 미래 10분동안의 변동성을 더 우수하게 예측(forecast)하는 것을 목표로 합니다. 예측에 있어 성능평가의 기준점은 RMSPE(Root Mean Square Percentage Error)입니다.

Kaggle에서 제공한 Data set은 다음과 같습니다.



	A	B	C
1	stock_id	time_id	target
2	0	5	0.004136
3	0	11	0.001445
4	0	16	0.002168
5	0	31	0.002195
6	0	62	0.001747
7	0	72	0.004912
8	0	97	0.009388
9	0	103	0.00412
10	0	109	0.002182
11	0	123	0.002669
12	0	128	0.003702
13	0	146	0.005397
14	0	147	0.002751
15	0	152	0.003969
16	0	157	0.005525
17	0	159	0.002077
18	0	160	0.002265

(좌) 제공되는 전체 데이터, (우) train.csv 파일 일부

1. test.csv, train.csv

stock_id, time_id 별 label(=target)의 정보입니다.

stock_id는 Optiver사에서 선택한 임의의 주식이며 총 112개가 선택되어 있습니다. 참가자는 해당 주식이 어떤 종목인지 알 수 없습니다. time_id 또한 마찬가지로, 10분 단위로 해당 주식의 book, trade 데이터를 쪼개어 임의의 숫자를 부여한 것입니다. 따라서 time_id는 정렬이 되어 있지 않습니다. 결과적으로 참가자들은 어떤 종목의 어떤 시간대에서 volatility값(target)이 산출되었는지 알 수 없습니다.

우리가 데이터를 이해하면서 느꼈던 궁금증과 이에 대한 답입니다.

Q1. stock_id별로 갖고 있는 time_id는 정확히 어떤 시간대인지 알지 못하지만, time_id가 같다면 같은 시간대에 대한 데이터를 제공하는 것이 맞는가?

- a. time_id 자체에는 sequence가 없지만, 만약 time_id=5 라면 stock_id=1,2, ..., n 모두가 같은 시간대의 정보를 담고있는 것이 맞습니다.

Q2. test.csv에는 제공된 label이 없는데, 어떻게 테스트 해야 하는가?

	A	B	C	D
1	stock_id	time_id	row_id	
2	0	4	0-4	
3	0	32	0-32	
4	0	34	0-34	
5				
6				

- a. kaggle측에 따르면 test.csv 파일은 kaggle측에서 평가시에 활용할 데이터 셋의 형태를 보여주기 위한 목적으로 제공되었습니다. 따라서 label을 제공되지 않아, Kaggle측에서 사용하는 test set이 무엇인지 파악할 수 없습니다. 따라서, 저희 팀은 train.csv 파일을 train, validation, test 데이터로 split하여 활용하였습니다.

2. book_test.parquet, book_train.parquet, trade_test.parquet, trade_train.parquet

stock_id와 time_id 별 book & trade 데이터를 담고 있는 parquet 형식의 파일입니다. 데이터에 대한 상세 설명은 아래와 같습니다.

2.1. book_*.parquet 데이터(=호가 데이터)

book_*.parquet 에는 각 stock_id 별 모든 time_id에서의 호가 데이터를 제공합니다. seconds_in_bucket은 각 time_id가 담고있는 10분(=600초)내에서의 초 단위 시간을 의미합니다. bid_price1,2 와 ask_price1,2는 호가창에서의 매수/매도 주문가를 의미합니다.

```
load_book(stock_id=1, block='train')
```

[4] ✓ 0.2s

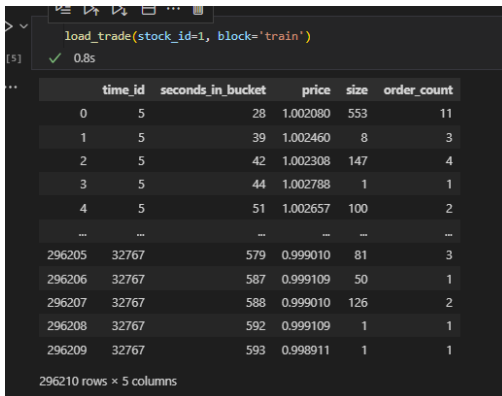
	time_id	seconds_in_bucket	bid_price1	ask_price1	bid_price2	ask_price2	bid_size1	ask_size1	bid_size2	ask_size2
0	5	0	1.000754	1.001542	1.000689	1.001607	1	25	25	100
1	5	1	1.000754	1.001673	1.000689	1.001739	26	60	25	100
2	5	2	1.000754	1.001411	1.000623	1.001476	1	25	25	125
3	5	3	1.000754	1.001542	1.000689	1.001607	125	25	126	36
4	5	4	1.000754	1.001476	1.000623	1.001542	100	100	25	25
...
1507527	32767	588	0.998911	0.999109	0.998812	0.999208	126	42	101	100
1507528	32767	589	0.998911	0.999109	0.998812	0.999208	126	126	101	200
1507529	32767	591	0.998911	0.999109	0.998812	0.999208	126	226	101	200
1507530	32767	592	0.998911	0.999109	0.998812	0.999208	226	225	101	100
1507531	32767	593	0.998911	0.999109	0.998812	0.999208	125	225	101	100

1507532 rows × 10 columns

이해를 돕기 위해 아래 표와 같은 호가창이 있다고 가정해봅시다. book_*.parquet에서의 bid_ask 1은 매수호가 중에서 가장 높은 가격인 147에 해당하며 bid_ask 2는 그 다음으로 높은 매수호가인 146에 해당합니다. 반대로 ask_price1은 매도호가 중에서 가장 낮은 가격을 의미합니다. 아래의 예시에서는 148에 해당하며, ask_price2는 그 다음으로 낮은 가격인 149가 됩니다.

bid #	price	ask #
	151	196
	150	189
	149	148
	148	221
251	147	
321	146	
300	145	
20	144	

2.1. trade_*.parquet 데이터(=체결 데이터)



```
load_trade(stock_id=1, block='train')
```

✓ 0.8s

time_id	seconds_in_bucket	price	size	order_count
0	5	28	1.002080	553
1	5	39	1.002460	8
2	5	42	1.002308	147
3	5	44	1.002788	1
4	5	51	1.002657	100
...
296205	32767	579	0.999010	81
296206	32767	587	0.999109	50
296207	32767	588	0.999010	126
296208	32767	592	0.999109	1
296209	32767	593	0.998911	1

296210 rows x 5 columns

bid #	price	ask #
	151	196
	150	189
	149	148
	148	201
251	147	
321	146	
300	145	
20	144	

book_*.parquet 데이터가 호가창의 데이터를 의미한다면, trade_*.parquet 데이터는 호가창 매물 중에 실제 체결된 거래에 대한 정보들로 구성됩니다. 각 time_id별 600초 내에 특정 시간(예: 21초)에 이루어진 거래가 어떤 price, size로 체결되었는지에 대한 정보를 제공합니다.

Evaluation Method

모델의 성능평가는 RMSE의 값을 percentage로 환산한 RMSPE를 사용합니다. 이는 Optiver에서 정한 Evaluation 방법입니다.

RMSPE = Root mean square percentage error

$$RMSPE = \sqrt{\frac{1}{N} \sum_{i=1}^N \left(\frac{y_i - \hat{y}_i}{y_i} \right)^2}$$

Loading Data

Loading raw data

제공된 book_*.parquet, trade_*.parquet 데이터는 범주형 데이터가 아닌 숫자형 데이터입니다. 또한 모든 stock_id 별 가격 정보가 표준화(scaled)되어 제공되고 있습니다.

load_trade(stock_id=1, block='train')

0.8s

time_id	seconds_in_bucket	price	size	order_count
0	5	28	1.002080	533
1	5	39	1.002460	8
2	5	42	1.002308	147
3	5	44	1.002788	1
4	5	51	1.002657	100
...
296205	32767	579	0.999010	81
296206	32767	587	0.999109	50
296207	32767	588	0.999010	126
296208	32767	592	0.999109	1
296209	32767	593	0.998911	1

296210 rows x 5 columns

load_trade(stock_id=110, block='train')

0.7s

time_id	seconds_in_bucket	price	size	order_count
0	5	41	1.001656	11
1	5	56	1.002202	2
2	5	57	1.002413	1
3	5	110	1.002977	1
4	5	188	1.001938	120
...
140067	32767	436	0.999485	203
140068	32767	441	0.999385	18
140069	32767	505	0.999763	1
140070	32767	567	0.999552	1
140071	32767	584	0.999789	4

140072 rows x 5 columns

load_book(stock_id=1, block='train')

0.2s

time_id	seconds_in_bucket	bid_price1	ask_price1	bid_price2	ask_price2	bid_size1	ask_size1	bid_size2	ask_size2
0	5	0	1.000754	1.001542	1.000689	1.001607	1	25	25
1	5	1	1.000754	1.001673	1.000689	1.001759	26	60	25
2	5	2	1.000754	1.001411	1.000623	1.001476	1	25	25
3	5	3	1.000754	1.001542	1.000689	1.001607	125	25	126
4	5	4	1.000754	1.001476	1.000623	1.001542	100	100	25
...
1507527	32767	588	0.998911	0.999109	0.998812	0.999208	126	42	101
1507528	32767	589	0.998911	0.999109	0.998812	0.999208	126	126	101
1507529	32767	591	0.998911	0.999109	0.998812	0.999208	126	226	101
1507530	32767	592	0.998911	0.999109	0.998812	0.999208	226	225	101
1507531	32767	593	0.998911	0.999109	0.998812	0.999208	125	225	101

1507532 rows x 10 columns

load_book(stock_id=110, block='train')

0.1s

time_id	seconds_in_bucket	bid_price1	ask_price1	bid_price2	ask_price2	bid_size1	ask_size1	bid_size2	ask_size2
0	5	0	0.999665	1.001620	0.999595	1.001638	12	3	11
1	5	3	0.999665	1.001620	0.999595	1.001638	40	1	11
2	5	4	0.999663	1.001620	0.999665	1.001638	38	1	12
3	5	6	0.999701	1.001638	0.999683	1.002184	38	10	12
4	5	9	0.999701	1.001638	0.999683	1.002184	40	10	10
...
1302555	32767	586	0.999420	1.000712	0.999156	1.001161	200	100	101
1302556	32767	587	0.999420	1.000422	0.999156	1.000712	200	100	101
1302557	32767	588	0.999420	1.000422	0.999156	1.000712	300	100	1
1302558	32767	590	0.999420	1.000422	0.999156	1.000712	350	100	1
1302559	32767	595	0.999420	1.000422	0.999156	1.000712	297	100	1

1302560 rows x 10 columns



3. Feature Engineering

1. **Base Feature**: 주어진 데이터에 대해 별도의 변환이나 가공없이 기본적인 연산으로 생성해낼 수 있는 특성을 의미합니다. (1) 주최측에서 제공하는 기본 특성변수와 (2) 데이터를 분류하고 합산, 평균, 로그합수 등의 연산을 통해 생성한 특성변수로 구성됩니다.
2. **Nearest Neighbors Feature**: Base Feature에 더하여, Nearest Neighbor 방법을 활용하여 생성한 특성변수입니다.

Base Features Generation

Base Book Features

기본 특성 변수

주최측에서 제공한 financial concepts에 따라, 기본적으로 생성하여 활용할 수 있는 호가데이터의 특성과 수식은 다음과 같습니다.

- Bid/ask spread

$$BidAskSpread = BestOffer / BestBid - 1$$

- Weighted Averaged Price(WAP)

- instantaneous stock valuation
- target인 10분 후 realized volatility를 계산하기 위해 필요

$$WAP = \frac{BidPrice1 * AskSize1 + AskPrice1 * BidSize1}{BidSize1 + AskSize1}$$

- Log returns

- 어제와 오늘의 주가를 어떻게 비교할 수 있을지에 대한 지표

S_t : the price of the stock S at time t

log return between t_1, t_2 :

$$r_{t_1, t_2} = \log \frac{S_{t_2}}{S_{t_1}}$$

예를 들어, 10분 전 대비 로그수익률을 구한다라고 하면

$$r_t = r_{t-10minute, t} = \log \left(\frac{S_t}{S_{t-10minute}} \right)$$

- Realized volatility

- $\sigma = \sqrt{\sum_t r_{t-1, t}^2}$

= (로그수익률의 제곱의 합에 대한 제곱근 값)

- 단, annualize 하지 않고 log return 의 mean 값은 0임.

위의 식을 바탕으로 다음과 같이 함수화하였습니다.

```
# Function to calculate first WAP
def calc_wap1(df):
    wap = (df['bid_price1'] * df['ask_size1'] + df['ask_price1'] * df['bid_size1']) / (df['bid_size1'] + df['ask_size1'])
    return wap

# Function to calculate second WAP
def calc_wap2(df):
    wap = (df['bid_price2'] * df['ask_size2'] + df['ask_price2'] * df['bid_size2']) / (df['bid_size2'] + df['ask_size2'])
    return wap

# Calculate the realized volatility
def realized_volatility(series):
    return np.sqrt(np.sum(series**2))

# Function to calculate the log of the return
# Remember that logb(x / y) = logb(x) - logb(y)
def log_return(series: np.ndarray):
    return np.log(series).diff()

def log_return_df2(series: np.ndarray):
    return np.log(series).diff(2)
```

추가 base feature

위의 base feature 함수와 데이터 내의 다른 요소들을 활용하여 다음과 같은 과정으로 추가적인 base feature를 생성합니다.

1. 기본적인 feature를 생성(log return, wap, etc.)하고, 해당 features를 연산하기 위한 딕셔너리를 저장합니다.
2. 이후, book, trade 데이터를 time_id별로 groupby하여 agg 메서드로 기본적인 메서드를 적용, 새로운 features를 추가적으로 생성합니다.

3. 생성된 features들을 150초 단위로 데이터를 끊어서 추가적인 features를 생성합니다.

```
def make_book_feature(stock_id, block = 'train'):
    book = load_book(stock_id, block)

    book['wap1'] = calc_wap1(book)
    book['wap2'] = calc_wap2(book)
    book['log_return1'] = book.groupby(['time_id'])['wap1'].apply(log_return)
    book['log_return2'] = book.groupby(['time_id'])['wap2'].apply(log_return)
    book['log_return_ask1'] = book.groupby(['time_id'])['ask_price1'].apply(log_return)
    book['log_return_ask2'] = book.groupby(['time_id'])['ask_price2'].apply(log_return)
    book['log_return_bid1'] = book.groupby(['time_id'])['bid_price1'].apply(log_return)
    book['log_return_bid2'] = book.groupby(['time_id'])['bid_price2'].apply(log_return)

    # Calculate wap balance
    book['wap_balance'] = abs(book['wap1'] - book['wap2'])
    # Calculate spread
    book['price_spread'] = (book['ask_price1'] - book['bid_price1']) / ((book['ask_price1'] + book['bid_price1']) / 2)
    book['bid_spread'] = book['bid_price1'] - book['bid_price2']
    book['ask_spread'] = book['ask_price1'] - book['ask_price2']
    book['total_volume'] = (book['ask_size1'] + book['ask_size2']) + (book['bid_size1'] + book['bid_size2'])
    book['volume_imbalance'] = abs((book['ask_size1'] + book['ask_size2']) - (book['bid_size1'] + book['bid_size2']))

    features = {
        'seconds_in_bucket': ['count'],
        'wap1': [np.sum, np.mean, np.std],
        'wap2': [np.sum, np.mean, np.std],
        'log_return1': [np.sum, realized_volatility, np.mean, np.std],
        'log_return2': [np.sum, realized_volatility, np.mean, np.std],
        'log_return_ask1': [np.sum, realized_volatility, np.mean, np.std],
        'log_return_ask2': [np.sum, realized_volatility, np.mean, np.std],
        'log_return_bid1': [np.sum, realized_volatility, np.mean, np.std],
        'log_return_bid2': [np.sum, realized_volatility, np.mean, np.std],
        'wap_balance': [np.sum, np.mean, np.std],
        'price_spread': [np.sum, np.mean, np.std],
        'bid_spread': [np.sum, np.mean, np.std],
        'ask_spread': [np.sum, np.mean, np.std],
        'total_volume': [np.sum, np.mean, np.std],
        'volume_imbalance': [np.sum, np.mean, np.std]
    }

    # time별로 묶어서 feature 더 생성
    for time in [450, 300, 150]:
        d = book[book['seconds_in_bucket'] >= time].groupby('time_id').agg(features).reset_index(drop=False)
        d.columns = flatten_name(f'book_{time}', d.columns)
        agg = pd.merge(agg, d, on='time_id', how='left')
    return agg

##### 동일한 로직으로 체결데이터에 대한 feature를 생성합니다. #####

def make_trade_feature(stock_id, block = 'train'):
    trade = load_trade(stock_id, block)
    trade['log_return'] = trade.groupby('time_id')['price'].apply(log_return)

    # Dict for aggregations
    features = {
        'log_return': [realized_volatility],
        'seconds_in_bucket': ['count'],
        'size': [np.sum],
        'order_count': [np.mean],
    }

    agg = trade.groupby('time_id').agg(features).reset_index()
    agg.columns = flatten_name('trade', agg.columns)
    agg['stock_id'] = stock_id

    for time in [450, 300, 150]:
        d = trade[trade['seconds_in_bucket'] >= time].groupby('time_id').agg(features).reset_index(drop=False)
        d.columns = flatten_name(f'trade_{time}', d.columns)
        agg = pd.merge(agg, d, on='time_id', how='left')
    return agg
```

Base features 데이터프레임 생성

- book, trade에 대한 features를 한번에 생성하기 위한 함수를 작성합니다.
 - log return realized volatility에 대해 생성된 features들에 대해, stock_id, time_id 별 max, min값을 적용하여 추가적인 features를 생성하여 더합니다.
 - trade와 book에 대해서 생성한 [*seconds_in_bucket.count] feature는 유용성을 나타내는 지표이므로 tau라는 새로운 feature를 추가하여 dataframe을 저장합니다.
 - 제공된 데이터는 112개의 stock에 대한 호가/체결 데이터이므로, 매우 방대한 양의 데이터를 다뤄야하는 반면 주식별로, 시간대별 데이터에 대해 실제 특성을 만들어내기 위한 작업을 반복적으로 수행해야합니다. 출력을 디스크에 저장해 세션과 세션 사이에 결과를 캐시할 수 있는 작업에 해당하므로 이러한 작업을 for문으로만 처리할 경우 처리속도가 현저하게 느려짐을 확인하였습니다. 이에 따라 저희 팀은 joblib.Parallel을 사용하여 별도의 CPU에서 동시에 작업을 실행하여 데이터 프로세싱을 진행하였습니다.
- 위 과정을 통해 train.csv에 저장된 stock_id, time_id, target 칼럼을 제외하고 **총 335개의 base features**를 생성하였습니다.
- 코드

```
def make_features(base, block):
    stock_ids = set(base['stock_id'])

    books = Parallel(n_jobs=-1)(delayed(make_book_feature)(i, block) for i in stock_ids)
    book = pd.concat(books)

    trades = Parallel(n_jobs=-1)(delayed(make_trade_feature)(i, block) for i in stock_ids)
```

```
trade = pd.concat(trades)

df = pd.merge(base, book, on=['stock_id', 'time_id'], how='left')
df = pd.merge(df, trade, on=['stock_id', 'time_id'], how='left')

# tau : 유동성 지표(보통수익률)
df['trade_tau'] = np.sqrt(1 / df['trade.seconds_in_bucket.count'])
df['trade_150_tau'] = np.sqrt(1 / df['trade_150.seconds_in_bucket.count'])
df['book_tau'] = np.sqrt(1 / df['book.seconds_in_bucket.count'])

# get realized volatility columns
vol_cols = [
    'book_log_return2.realized_volatility',
    'book_log_return_ask1.realized_volatility',
    'book_log_return_ask2.realized_volatility',
    'book_log_return1.realized_volatility',
    'book_log_return_bid1.realized_volatility',
    'book_log_return_bid2.realized_volatility',
    'book_450_log_return1.realized_volatility',
    'book_450_log_return2.realized_volatility',
    'book_450_log_return_ask1.realized_volatility',
    'book_450_log_return_ask2.realized_volatility',
    'book_450_log_return_bid1.realized_volatility',
    'book_450_log_return_bid2.realized_volatility',
    'book_300_log_return1.realized_volatility',
    'book_300_log_return2.realized_volatility',
    'book_300_log_return_ask1.realized_volatility',
    'book_300_log_return_ask2.realized_volatility',
    'book_300_log_return_bid1.realized_volatility',
    'book_300_log_return_bid2.realized_volatility',
    'book_150_log_return1.realized_volatility',
    'book_150_log_return2.realized_volatility',
    'book_150_log_return_ask1.realized_volatility',
    'book_150_log_return_ask2.realized_volatility',
    'book_150_log_return_bid1.realized_volatility',
    'book_150_log_return_bid2.realized_volatility',
    'trade_log_return.realized_volatility',
    'trade_450_log_return.realized_volatility',
    'trade_300_log_return.realized_volatility',
    'trade_150_log_return.realized_volatility']

# Groupby stock id
df_stock_id = df.groupby(['stock_id'])[vol_cols].agg(['max', 'min']).reset_index(drop=False)
df_stock_id.columns = ['.'.join(col) for col in df_stock_id.columns]

df_stock_id = df_stock_id.add_suffix('_' + 'stock')
df_stock_id.rename(columns={'stock_id._stock': 'stock_id'}, inplace=True)

# Groupby time id
df_time_id = df.groupby(['time_id'])[vol_cols].agg(['max', 'min']).reset_index(drop=False)
df_time_id.columns = ['.'.join(col) for col in df_time_id.columns]

df_time_id = df_time_id.add_suffix('_' + 'time')
df_time_id.rename(columns={'time_id._time': 'time_id'}, inplace=True)

# Merge with og df
df = df.merge(df_stock_id, on = ['stock_id'], how = 'left')
df = df.merge(df_time_id, on = ['time_id'], how = 'left')

return df
```

```
df = make_features(train, 'train')
```

- 위와 같이 코드를 입력하면, train.csv로 저장된 데이터를 기반으로 book_train.parquet과 trade_train.parquet에서 모든 stock_id, 그리고 stock_id별 모든 time_id 데이터를 불러모아 features를 계산합니다. 이렇게 계산된 dataframe은 다음과 같이 출력됩니다.

...	stock_id	time_id	target	book.seconds_in_bucket.count	book.wap1.sum	book.wap1.mean	book.wap1.std	book.wap2.sum	book.wap2.mean	book.wap2.std	...	book_150_log_retu
0	0	5	0.004136	302	303.125061	1.003725	0.000693	303.105539	1.003661	0.000781
1	0	11	0.001445	200	200.047768	1.000239	0.000262	200.041171	1.000206	0.000272
2	0	16	0.002168	188	187.913849	0.999542	0.000864	187.939824	0.999680	0.000862
3	0	31	0.002195	120	119.859781	0.998832	0.000757	119.835941	0.998633	0.000656
4	0	62	0.001747	176	175.932865	0.999619	0.000258	175.934256	0.999626	0.000317
...
428927	126	32751	0.003461	310	309.870466	0.999582	0.000486	309.871372	0.999585	0.000613
428928	126	32753	0.003113	223	223.552143	1.002476	0.001264	223.580314	1.002602	0.001303
428929	126	32758	0.004070	256	256.277050	1.001082	0.000466	256.255056	1.000996	0.000599
428930	126	32763	0.003357	399	399.721736	1.001809	0.000456	399.714332	1.001790	0.000507
428931	126	32767	0.002090	217	217.058919	1.000272	0.000384	217.079726	1.000367	0.000465

428932 rows x 338 columns

- 결측치 제거

df에 대해 추가적으로 Nearest Neighbor feature들을 생성하기 전에, 결측치를 확인하고 제거합니다. 결측치는 단순히 target열을 제외한 모든 열의 행값에서 null값이 존재한다면 해당 행을 제거하는 방식으로 진행합니다. 결측치를 제거한 후, 행은 총 427,216개가 남게 됩니다.

✓ 1.3s Python

	stock_id	time_id	target	book.seconds_in_bucket.count	book.wap1.sum	book.wap1.mean	book.wap1.std	book.wap2.sum	book.wap2.n
	0	0	1176	0.005746	144	143.815068	0.998716	0.001774	143.849316
	1	0	8664	0.002469	147	146.899894	0.999319	0.000366	146.901871
	2	0	12758	0.002541	142	141.728688	0.998089	0.000900	141.709407
	3	0	19033	0.002515	94	93.842941	0.998329	0.000771	93.848332
	4	0	20499	0.003066	170	169.654033	0.997965	0.000716	169.650958
...
	427211	126	11589	0.003061	328	327.286943	0.997826	0.001360	327.295300
	427212	126	4927	0.005008	224	224.815252	1.003640	0.000781	224.783674
	427213	126	15155	0.018900	348	337.124379	0.968748	0.006521	336.886673
	427214	126	13316	0.010262	279	281.655248	1.009517	0.002213	281.599159
	427215	126	1464	0.005431	506	503.463445	0.994987	0.000881	503.465329

427216 rows x 338 columns

Nearest Neighbor Features

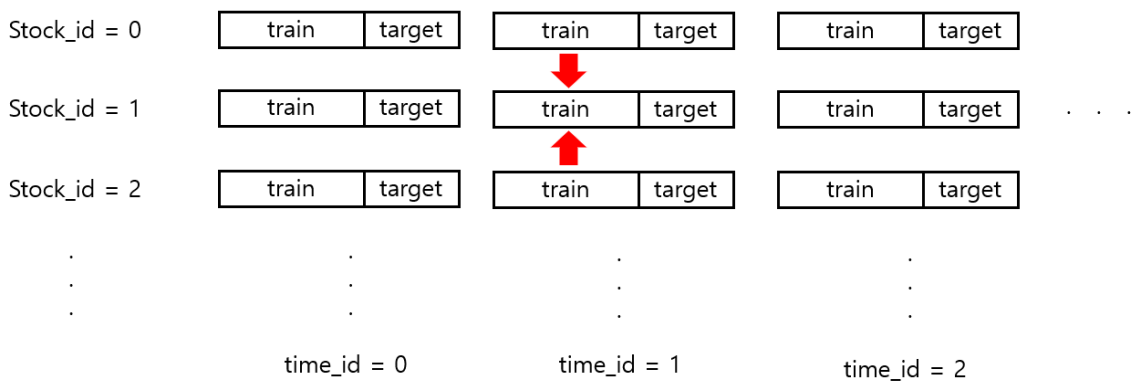
Nearest Neighbor Features 개요

체결, 호가데이터를 이용해서 만든 features에 추가하여, model의 performance를 늘릴 수 있는 새로운 features를 생성합니다. 새로운 features를 생성하는 방법론으로 Nearest Neighbor를 선택합니다.

동일한 time_id내에서 stock_id별 미래 변동성을 예측하기에 적합한 feature들로, Nearest Neighbor feature를 구성합니다. 구성은 다음과 같은 단계로 진행합니다.

- 로그수익률의 실현변동성과 거래량이 비슷한 종목을 기준으로 클러스터링합니다.
- 미래 변동성을 예측하기에 적합한 것으로 판단되는 base feature를 선별합니다.
- 각 클러스터를 구성하는 종목들의 2에서 선별된 base feature들에 대한 평균, 합산, 표준편차, 최솟값, 최댓값을 새로운 특성변수로 생성합니다.

Stock_id Nearest Neighbor Features 생성



Nearest Neighbors features 생성 방법

구체적인 생성 방법은 다음과 같습니다.

- sklearn.neighbors 모듈의 NearestNeighbors를 사용
 - 해당 알고리즘은 제공되는 데이터에 대해 설정한 N값에 대하여 N-1개의 최근접점을 저장한 indices, 그리고 해당 점들과의 거리인 distances df를 각각 제공합니다.
 - 모듈 사용시 거리 계산법은 Manhattan distance로 설정하였습니다. Minkowski metric에서 p=1로 지정 시 자동으로 Manhattan distance로 계산할 수 있으며, 이는 유클리디안 거리와 다른 L_1 -distance 입니다.

Manhattan Distance

$$= \sum_{i=1}^n |p_i - q_i| \text{ for } \vec{p} = (p_1, \dots, p_n) \text{ and } \vec{q} = (q_1, \dots, q_n)$$

$$= \|\vec{p} - \vec{q}\|_1$$

- features간의 상대적인 거리를 계산하기 위해 pivot에 대한 scaling 처리가 필요합니다. 따라서 sklearn.preprocessing의 minmax_scale을 적용합니다.

- 유사도 측정 기준 feature로 pivot 생성

N값은 계산이 오래 걸리지 않는 선에서 큰 값을 지정하고 이후 단위 별로 잘라서 feature를 생성하기 위해 80으로 부여합니다. book에서 생성한 log return realized volatility (1)와 book에서 생성한 trade volumn의 sum값 (2)을 바탕으로 pivot을 생성하여 stock별 vol (1), stock별 size (=trade size sum) (2)에 대한 Nearest Neighbor를 구하였습니다.

```
from sklearn.neighbors import NearestNeighbors
from sklearn.preprocessing import minmax_scale

# N 지정값
N_NEIGHBORS_MAX = 80

# Nearest Neighbor으로 만들 vol과 trade.size.sum에 대한 df_pv 생성
df_pv = df[['stock_id', 'time_id']].copy()
df_pv['vol'] = df['book.log_return1.realized_volatility']
df_pv['trade.size.sum'] = df['book.total_volume.sum']
```

- Nearest neighbor feature 생성

- 생성된 각각의 distance, neighbors matrix는 stock_id에 대한 nearest neighbor feature를 생성하는 함수에 적용시켜 N=10, 20, 40에 대한 feature의 column을 생성하기 위해 사용될 것입니다.

아래는 distance와 indices를 생성하는 코드입니다.

```
# vol neighbor distances, indices(.neighbors) 생성
pivot = df_pv.pivot('time_id', 'stock_id', 'vol')
pivot = pivot.fillna(pivot.mean())
pivot = pd.DataFrame(minmax_scale(pivot))

k_neighbors_stock_vol = NearestNeighbors(n_neighbors=80, metric='minkowski', p=1)
k_neighbors_stock_vol.fit(minmax_scale(pivot.transpose()))
k_neighbors_stock_vol.distances, k_neighbors_stock_vol.neighbors = k_neighbors_stock_vol.kneighbors(minmax_scale(pivot.transpose()), return_distance=True)

# trade.size neighbor, indices(.neighbors) 생성
pivot = df_pv.pivot('time_id', 'stock_id', 'trade.size.sum')
pivot = pivot.fillna(pivot.mean())
pivot = pd.DataFrame(minmax_scale(pivot))

k_neighbors_stock_size = NearestNeighbors(n_neighbors=80, metric='minkowski', p=1)
k_neighbors_stock_size.fit(minmax_scale(pivot.transpose()))
k_neighbors_stock_size.distances, k_neighbors_stock_size.neighbors = k_neighbors_stock_size.kneighbors(minmax_scale(pivot.transpose()), return_distance=True)
```

- make_neighbors_stock 함수를 통해 neighbor에 내가 원하는 size만큼의 nearest neighbor를 추가합니다. 이어서 해당 칼럼명을 생성하기 위한 make_nn_feature 함수를 생성합니다.

```
# stock-id 를 row로 하는 pivot을 바탕으로 feature pivot과 neighbor 변환
def make_neighbors_stock(df, k_neighbors, feature_col, n=5):
    feature_pivot = df.pivot('time_id', 'stock_id', feature_col)
    feature_pivot = feature_pivot.fillna(feature_pivot.mean())
    feature_pivot.head()

    neighbors = np.zeros((n, *feature_pivot.shape))

    for i in range(n):
        neighbors[i, :, :] += feature_pivot.values[:, k_neighbors[i, 1]]

    return feature_pivot, neighbors

# nn_feature 생성 함수
def make_nn_feature(df, neighbors, columns, index, n=5, agg=np.mean, postfix='', exclude_self=False, exact=False):
    start = 1 if exclude_self else 0

    if exact:
        pivot_aggs = pd.DataFrame(neighbors[n-1, :, :], columns=columns, index=index)
    else:
        pivot_aggs = pd.DataFrame(agg(neighbors[start:n, :, :], axis=0), columns=columns, index=index)
    dst = pivot_aggs.unstack().reset_index() # unstack(level)이 의미하는 것은 multi Index의 몇번째 index를 칼럼 방향으로 stacking 할것인가임.
    dst.columns = ['stock_id', 'time_id', f'{feature_col}_cluster{n}{postfix}_{agg.__name__}']
    return dst
```

- 이제 준비과정을 끝냈습니다. Nearest Neighbor Features를 생성합니다.

아래 df2 변수는 Nearest Neighbor feature가 추가되기 전/후를 비교하기 위해 생성되었습니다. add_ndf 함수를 통해 순차적으로 stock vol과 stock size에 대한 Nearest Neighbor features를 df2에 결합합니다.

```
import gc #순환참조를 탐지하고 해결하기 위해 사용하는 모듈
gc.collect()

df2 = df.copy()
print(df2.shape)
```

```

## neighbor stock id 에 대한 feature
feature_cols_stock = {
    'book.log_return1.realized_volatility': [np.mean, np.min, np.max, np.std],
    'trade.seconds_in_bucket.count': [np.mean],
    'trade.tau': [np.mean],
    'trade_150.tau': [np.mean],
    'book.tau': [np.mean],
    'trade.size.sum': [np.mean],
    'book.seconds_in_bucket.count': [np.mean]
}

# ndf가 아무것도 없으면 dst 그대로 반환
# ndf가 있으면 dst의 마지막 열을 ndf에 추가해서 ndf 반환
ndf = None
cols = []
def _add_ndf(ndf, dst):
    if ndf is None:
        return dst
    else:
        ndf[dst.columns[-1]] = dst[dst.columns[-1]].astype(np.float32)
        return ndf

# stock_id에 대한 neighbor 추가
stock_id_neighbor_sizes = [10, 20, 40]

for feature_col in feature_cols_stock.keys():
    feature_pivot, neighbors_stock_vol = make_neighbors_stock(df2, k_neighbors_stock_vol.neighbors, feature_col, n=N_NEIGHBORS_MAX)
    _, neighbors_stock_size = make_neighbors_stock(df2, k_neighbors_stock_size.neighbors, feature_col, n=N_NEIGHBORS_MAX)

    columns = feature_pivot.columns
    index = feature_pivot.index

    for agg in feature_cols_stock[feature_col]:
        for n in stock_id_neighbor_sizes:
            exclude_self = True
            exact = False

            dst = make_nn_feature(df2, neighbors_stock_vol, columns, index, n=n, agg=agg, postfix='_sv', exclude_self=exclude_self, exact=exact)
            ndf = _add_ndf(ndf, dst)

            dst = make_nn_feature(df2, neighbors_stock_size, columns, index, n=n, agg=agg, postfix='_ssize', exclude_self=exclude_self)
            ndf = _add_ndf(ndf, dst)

        del feature_pivot, neighbors_stock_vol

df2 = pd.merge(df2, ndf, on=['time_id', 'stock_id'], how='left')
ndf = None

print(df2.shape)
df2.reset_index(drop=True)

del ndf

```

- 마지막으로 base + Nearest Neighbor features dataframe의 최종 형태를 확인하면,

df2.drop(columns=['stock_id', 'time_id', 'target'], inplace=False)
1.6s
Python

	book.seconds_in_bucket.count	book.wap1.sum	book.wap1.mean	book.wap1.std	book.wap2.sum	book.wap2.mean	book.wap2.std	book.lo
0	144	143.815068	0.998716	0.001774	143.849316	0.998954	0.001861	
1	147	146.899894	0.999319	0.000366	146.901871	0.999332	0.000398	
2	142	141.728688	0.998089	0.000900	141.709407	0.997954	0.000960	
3	94	93.842941	0.998329	0.000771	93.848332	0.998387	0.000798	
4	170	169.654033	0.997965	0.000716	169.650958	0.997947	0.000761	
...
427211	328	327.286943	0.997826	0.001360	327.295300	0.997852	0.001418	
427212	224	224.815252	1.003640	0.000781	224.783674	1.003499	0.000937	
427213	348	337.124379	0.968748	0.006521	336.886673	0.968065	0.006172	
427214	279	281.655248	1.009517	0.002213	281.599159	1.009316	0.002231	
427215	506	503.463445	0.994987	0.000881	503.465329	0.994991	0.000936	

427216 rows × 397 columns

397개의 features에 대해 427,216개의 행이 존재함을 확인할 수 있습니다.

이제 예측모델의 성능을 테스트 하기 위해, 해당 dataframe을 이용하여 train,val,test split을 진행하여 최적 조합을 찾기 전 모든 준비가 완료되었습니다.



4. Forecasting Models

XGBoost

첫번째 변동성 예측모델로 XGBoost 모델을 사용하였습니다.

XGB는 Extreme Gradient Boost 의 준말로 Tianqi Chen , Carlos Guestrin이 개발한 알고리즘입니다. XGBoost는 Gradient Boosting 알고리즘에 대한 근사로, parallelize 를 가능하게 하여 분산환경에서도 실행될 수 있고 하드웨어적으로 다양한 최적화 기법을 적용했기 때문에 예측 성능과 자원 효율이 우수할 뿐만 아니라 확장성도 뛰어나 여러 종류의 대용량 데이터에 적용이 가능합니다.

- sklearn wrapper class의 XGBoost를 사용합니다. 변동성을 예측하여 손실함수인 RMSPE를 최소화하는 문제이므로, xgboost.XGBRegressor(회귀) 클래스를 사용합니다.

아래는 Base로만 features를 형성한 후, hyperparameter tuning 할때의 과정에 대한 설명입니다. Nearest Neighbor features가 추가 되었을때, 더 최적인 조합을 찾기 위해 halvingserachCV를 적용하여 다른 조합으로 튜닝하였으며 이는 Type 2 부분에서 설명합니다.

- XGBRegressor 파라미터
 - booster
 - default인 gbtree를 사용합니다.
 - n_jobs
 - xgboost를 실행하는 데 사용되는 병렬 thread의 수를 지정하는 파라미터입니다. default로 사용합니다.
- Booster 파라미터
 - n_estimators
 - Tree의 개수를 지정하는 파라미터입니다. Boosting round를 의미하기도 합니다. 400, 500 중 하나로 튜닝하여 사용합니다.
 - learning_rate
 - 학습률을 지정하는 파라미터입니다. 0.04, 0.05 중 하나로 튜닝하여 사용합니다.
 - colsample_bytree
 - 각 iteration에서 학습에 사용되는 feature의 비율을 지정하는 파라미터입니다. 0.8로 지정하여 사용합니다.
 - subsample
 - 각 iteration에서 학습에 사용되는 data의 비율을 지정하는 파라미터입니다. 0.7로 지정하여 사용합니다.
 - max_depth
 - Tree depth의 최대값을 지정하는 파라미터입니다. default로 사용합니다.
 - min_child_weight
 - 학습 시 직전 예측에 대한 가중치를 지정하는 파라미터입니다. regression model을 사용하므로 default인 1을 사용합니다.

Type 1 (Base → XGBoost), Type 2 (Base+ Nearest Neighbor features → XGBoost) 단계에서 파라미터를 튜닝합니다.

먼저 base features를 바탕으로 hyperparameter에 대한 튜닝 전/후 의 성능을 평가합니다. 튜닝된 값이 더 개선이 되었다면, 해당 hyperparameter 조합으로 Type2에서 튜닝없이 성능을 평가하고, 새로 추가된 데이터 전부에 대한 hyperparameter 튜닝을 다시 진행

합니다. 이후 성능을 확인하고, 이후 Type3, 4에서 XGBoost와 다른 딥러닝 모델과의 앙상블에서 가장 최적인 조합을 선택하여 진행할 것입니다.

세부적인 진행단계는 다음과 같습니다.

Type 1 (Base features → XGBoost)

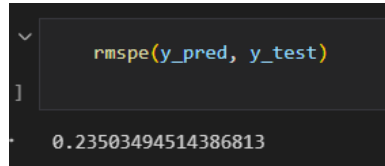
- 먼저 Type 1에서 hyperparameter에 대한 튜닝 없이 임의로 값을 부여하여 성능을 확인합니다.

```
params = {'n_estimators': 100,
          'learning_rate': 0.1,
          'colsample_bytree': 0.8,
          'subsample': 0.8
        }
```

- 해당 hyperparameter로 튜닝한 RMSPE 값은 1.151입니다.
- 그 다음, RandomSearchCV를 통해 hyperparameter를 튜닝하여 최적 조합을 찾아 성능을 확인합니다.

```
params = {'xgb__n_estimators': [400, 500],
          'xgb__learning_rate': [0.04, 0.05],
          'xgb__colsample_bytree': [0.8],
          'xgb__subsample': [0.7]
        }
```

- 찾은 최적의 조합은 'xgb__subsample': 0.7, 'xgb__n_estimators': 400, 'xgb__learning_rate': 0.04, 'xgb__colsample_bytree': 0.8 입니다. 이 조합으로 XGBoost 모델을 사용하여 예측합니다.

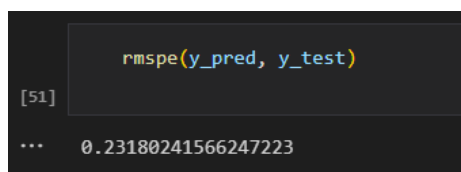


```
rmse(y_pred, y_test)
0.23503494514386813
```

- 예측 성능이 상당히 개선되었습니다. 하지만 해당 RMSPE 점수는 kaggle competition의 순위권 점수에 못미치는 점수이므로, 아직 성능개선이 필요함을 알 수 있습니다. 따라서, Type 2, 3, 4에서 추가적으로 생성하는 feature와 다른 model들과의 조합을 통해 성능을 개선합니다.

Type 2 (Base + Nearest Neighbor features → XGBoost)

- Type1에서 선택된 hyperparameter의 최적 조합으로, Nearest Neighbor features 가 추가된 데이터를 사용해 XGBoost 모델을 적용한 예측을 실시합니다. Type2에서는 해당 Nearest Neighbor features가 추가되었으므로 한번 더 튜닝을 진행하여, Nearest Neighbor features가 추가된 상태에서의 최적 조합을 찾을 것입니다.
- 여기서 찾아낸 최적 조합은 이후 다른 딥러닝 모델과의 앙상블에서 그대로 hyperparameter로 사용됩니다.



```
rmse(y_pred, y_test)
[51]
... 0.23180241566247223
```

- base feature보다 약간 개선된 결과를 보여줍니다.

- 다음은 hyperparameter tuning을 통한 성능 평가입니다. 여기서 tuning은 RandomSearchCV가 아닌 **HalvingRandomSearchCV**를 사용합니다. factor와 n_candidates를 지정하는 값에 따라 몇번의 라운드를 돌릴 수 있을지 정할 수 있습니다.
 - HalvingRandomSearchCV는 sklearn의 hyper parameter tuning model입니다. RandomSearchCV와는 다르게, factor라는 파라미터를 통해 n_estimator의 값을 순차적으로 높여가면서 n_candidate에 할당된 값만큼의 조합을 적용하여 scoring값을 산출합니다.
 - 저희 모델은 min_resources = 65로 지정하였으므로, n_estimator는 65부터 순차적으로 시작하여 1625까지 적용합니다 (65 → 325 → 1625).
 - n_candidate 지정 값은 25이므로, 1개까지 적용합니다 (25 → 5 → 1).
 - 총 3번의 라운드를 거쳐 정해진 Best Params 조합은, 'xgb__subsample': 0.8, 'xgb__max_depth': 17, 'xgb__learning_rate': 0.22399999999999995, 'xgb__gamma': 0.4, 'xgb__colsample_bytree': 0.6, 'n_estimators': 1625로 선정되었습니다.
- 최적조합을 통해 예측한 결과, 성능이 일부 개선되었음을 알 수 있습니다(RMSPE = 0.2208)

```
In [5]: print('Best parameters: ', grid_xgb.best_params_)
        print('Best score: ', grid_xgb.best_score_)

        params = grid_xgb.best_params_

        final_model_with_tuning = XGBRegressor(**params)
        final_model_with_tuning.fit(X_train, y_train, eval_metric = 'rmse', eval_set = [(X_train, y_train), (X_val, y_val)], early_stopping_rounds=10)

        # Get predictions
        y_pred = final_model_with_tuning.predict(X_test)

        # eval by RMSPE
        def rmspe(predictions, targets):
            return np.sqrt((((predictions - targets) / targets) ** 2).mean())

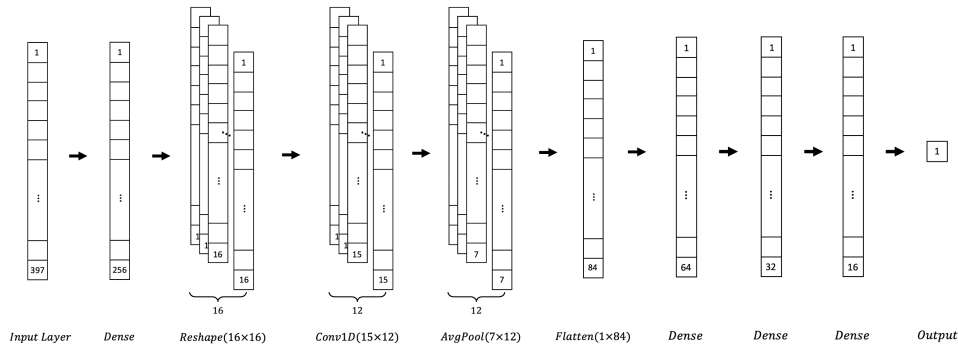
        rmspe(y_pred, y_test)
```

[284]	validation_0-rmse:0.00066	validation_1-rmse:0.00105
[285]	validation_0-rmse:0.00066	validation_1-rmse:0.00105
[286]	validation_0-rmse:0.00066	validation_1-rmse:0.00105
[287]	validation_0-rmse:0.00066	validation_1-rmse:0.00105
[288]	validation_0-rmse:0.00065	validation_1-rmse:0.00105
[289]	validation_0-rmse:0.00065	validation_1-rmse:0.00105
[290]	validation_0-rmse:0.00065	validation_1-rmse:0.00105
[291]	validation_0-rmse:0.00065	validation_1-rmse:0.00105
[292]	validation_0-rmse:0.00065	validation_1-rmse:0.00105
[293]	validation_0-rmse:0.00065	validation_1-rmse:0.00105
[294]	validation_0-rmse:0.00065	validation_1-rmse:0.00105
[295]	validation_0-rmse:0.00065	validation_1-rmse:0.00105
[296]	validation_0-rmse:0.00065	validation_1-rmse:0.00105
[297]	validation_0-rmse:0.00065	validation_1-rmse:0.00105
[298]	validation_0-rmse:0.00065	validation_1-rmse:0.00105
[299]	validation_0-rmse:0.00065	validation_1-rmse:0.00105
[300]	validation_0-rmse:0.00065	validation_1-rmse:0.00105
[301]	validation_0-rmse:0.00065	validation_1-rmse:0.00105

```
Out[5]: 0.22076532853794661
```

CNN

두번째 변동성 예측모델로 Convolutional Neural Network(이하 CNN) 모델을 사용하였습니다. CNN은 인공신경망 모델의 앞부분에 컨볼루션 기법을 추가한 모델로, 입력 데이터를 컨볼루션 레이어에 통과시켜 전처리한 후, 이후 신경망(neural network)을 이용하여 회귀분석을 하게됩니다. CNN 모델의 구성은 다음과 같습니다.



- 입력층: 입력층의 input_dim은 학습데이터의 특성변수 개수만큼 설정하였으며 배치 크기는 1,024개로 설정하였습니다.
- 컨볼루션 레이어: 입력된 데이터로부터 특징을 추출하는 역할을 합니다.
 - 256개의 unit을 가진 층을 하나 형성한 후 가중치 초기화를 위해 he_normal 초기화를 사용하였습니다. 활성화함수는 ELU로 지정하였습니다.
 - 컨볼루션 작업을 수행하기 위해 위의 층을 (16, 16)의 이차원 행렬로 전환합니다.
 - 특징을 추출하기 위한 필터는 필터의 개수 12개, 필터의 크기는 2로 설정하였습니다.
 - 처리해야할 데이터의 양을 줄이기 위해 풀링(pooling) 과정을 적용하였습니다. 풀링 방법은 평균 풀링(Average Pooling)을 사용하여 (16,16) 데이터의 대표값으로 평균을 선택하였습니다. 이 때 풀링 윈도우 크기는 2로 설정하여 전체 크기를 절반으로 줄였습니다.
 - Flatten() 함수를 통해 위에서 작업한 2차원 배열을 1차원 배열인 (84,)로 바꿔주었습니다.
- 은닉층:
 - for문을 통해 총 3개의 은닉층을 형성하였습니다. 이 때 unit개수는 64개 → 32개 → 16개로 설정하였습니다. 초기화는 he_normal 초기화, 활성화함수는 ELU로 지정하였습니다.
 - 과적합을 완화하기 위해 가우시안 노이즈 작업을 수행하였습니다. 평균의 0인 가우시안 랜덤 데이터를 추가하여 의도적으로 기존 데이터에 변형을 가하는 작업입니다. 이 때 노이즈 분포의 표준편차는 0.01로 설정하였습니다.
 - dropout 기능을 활용하여 특정 노드에 학습이 지나치게 몰리는 것을 방지하여 과적합을 완화하였습니다. 드롭할 입력데이터의 비율을 0.2로 설정하였습니다.
- 출력층: 회귀분석 모델이므로, 1개의 출력값이 나오도록 하였습니다.
- 모델 컴파일:
 - loss: evaluation 지표이자 loss function인 rmspe로 지정하였습니다.
 - optimizer: Adam을 사용하였으며 learning rate은 초기값인 6e-3으로 설정하였습니다.
- Callback:
 - learning rate과 epochs 초기값은 각각 6e-3, 1000으로 설정하고 다음과 같이 callback 기능을 활용하여 업데이트 하였습니다.
 - ReduceLROnPlateau: 검증데이터의 손실함수가 5회의 연속적인 에포크 동안 향상되지 않을 때마다 현재의 학습률에 0.5가 곱해지도록 설정하였습니다. 개선된 것으로 간주하기 위한 최소한의 변화량인 min_delta는 학습 시 손실함수 감소 추이를 보아 1e-5로 부여하였습니다.
 - EarlyStopping: 검증데이터의 손실함수가 31회 연속적인 에포크 동안 향상되지 않을 때 학습을 멈추는 기능을 학습시 사용하였습니다.
- CNN 알고리즘은 다음과 같이 작성하였습니다.

```
def CNN(X_train, y_train, X_val, y_val, num_columns, num_labels, learning_rate, epochs):
    inp = tf.keras.layers.Input(shape=(num_columns,))
    x = tf.keras.layers.BatchNormalization()(inp)
    x = tf.keras.layers.Dense(256, kernel_initializer='he_normal', activation='ELU')(x)
    x = tf.keras.layers.Reshape((16,16))(x)
```

```

x = tf.keras.layers.Conv1D(filters=12, kernel_size=2, kernel_initializer='he_normal', activation='ELU')(x)
x = tf.keras.layers.AveragePooling1D(pool_size=2)(x)
x = tf.keras.layers.Flatten()(x)

for i in range(3):
    x = tf.keras.layers.Dense(64//(2**i), kernel_initializer='he_normal', activation='ELU')(x)
    x = tf.keras.layers.BatchNormalization()(x)
    x = tf.keras.layers.GaussianNoise(0.01)(x)
    x = tf.keras.layers.Dropout(0.20)(x)

x = tf.keras.layers.Dense(num_labels)(x)

model = tf.keras.models.Model(inputs=inp, outputs=x)
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate), loss=rmspe)

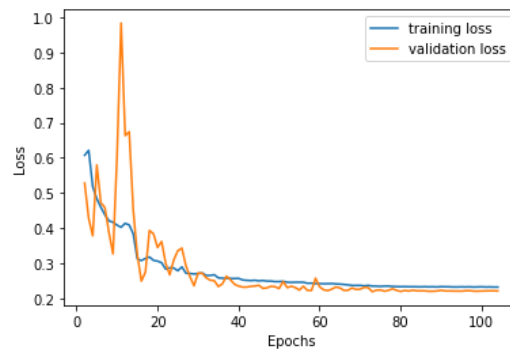
rlr = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=5, min_delta=1e-5, verbose=2)
es = EarlyStopping(monitor='val_loss', min_delta=1e-5, patience=31, restore_best_weights=True, verbose=2)

history = model.fit(X_train, y_train, epochs=epochs, validation_data=(X_val, y_val),
                    validation_batch_size=len(y_val), batch_size=batch_size, verbose=1, callbacks=[rlr, es])

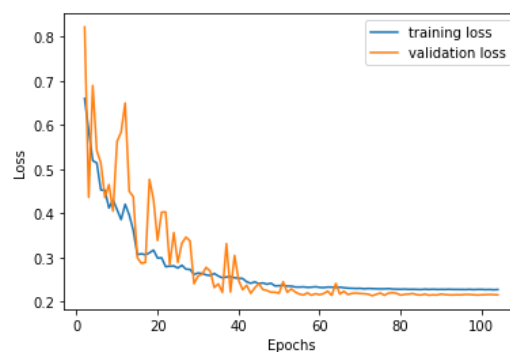
return model, history

```

- Base feature만으로 CNN 알고리즘에 적용한 결과는 **rmspe는 0.2173**이며 각 에포크마다 손실함수의 감소 추이는 아래 그래프와 같습니다. 에포크 초기에는 validation data에 대한 손실함수 값이 매우 컸으나 에포크가 증가할 수록 감소하면서 일정 범위 내의 값으로 안정적으로 산출되고 있음을 확인할 수 있습니다.



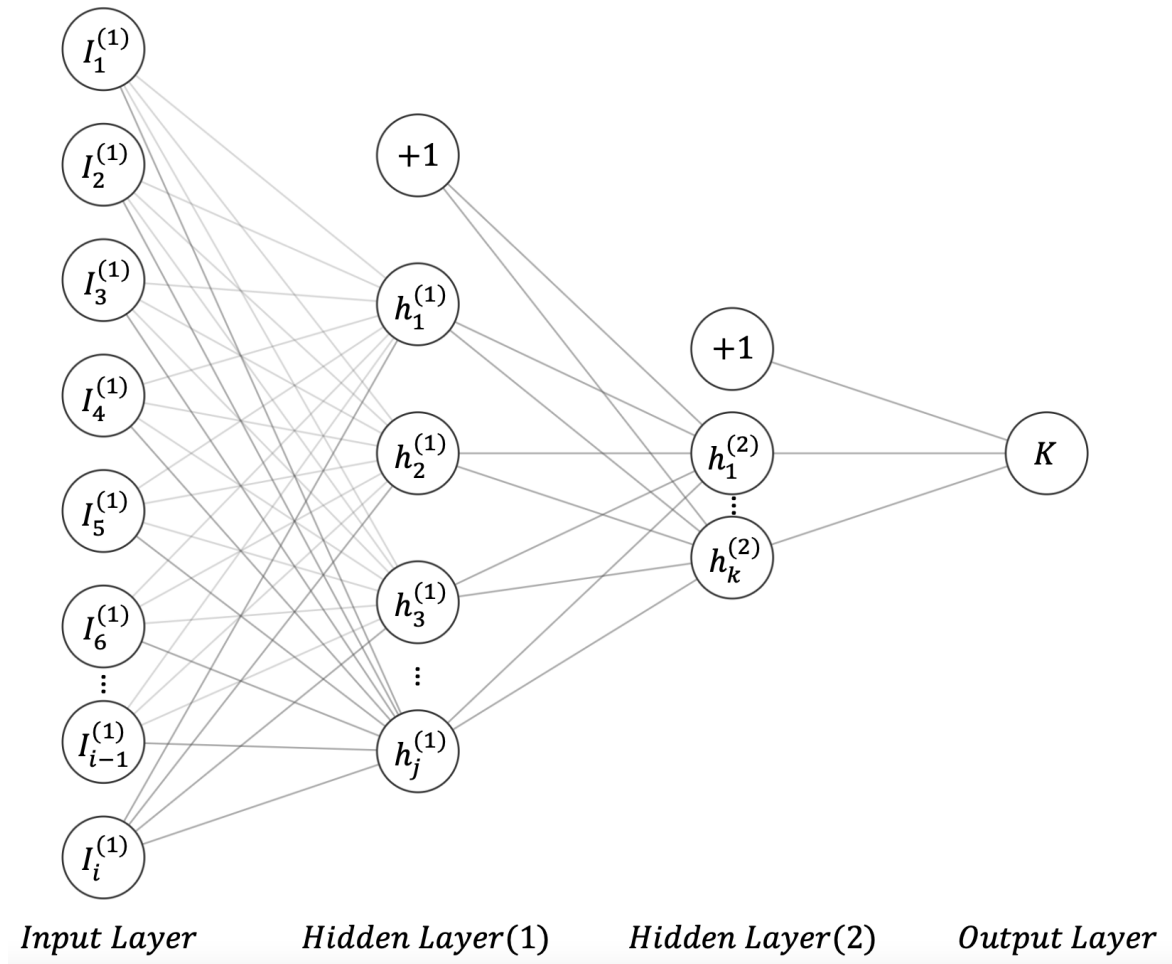
- Base feature와 Nearest Neighbor feature를 CNN 알고리즘에 적용한 결과 **rmspe는 0.2133**으로 Base feature만으로 학습시킨 모델의 성능보다 일부 개선되었습니다. 각 에포크별 손실함수의 감소 추이는 아래 그래프와 같습니다. 초반 에포크에서 변동폭이 큰 경향을 보이기는 하지만 에포크가 증가할 수록 감소하면서 일정 범위 내의 값으로 안정적으로 산출되고 있음을 확인할 수 있습니다.



MLP

세번째 변동성 예측모델로 Multi Layer Perceptron (이하 MLP) 모델을 사용하였습니다. MLP는 Train Set Data를 통해 출력값을 구하는 형태로 Model이 구축됩니다. 위 과정에서 Target 값과, 실제 모델이 만들어낸 Output의 차이를 오차로 잡아 그 오차를 뒤로 전파하며 각 노드의 변수를 갱신하는 방법을 통해 예측성능을 높이는 방법을 사용합니다.

Numerical Target Data의 특성을 반영해 구상한 MLP Regression 모델은 아래와 같습니다.



Input Layer:

데이터를 입력받는 층 (기존 Feature Data들)

Base Features의 경우, $i = 335$

Nearest Neighbor Features의 경우, $i = 397$

```
inputs= tf.keras.Input(shape=(X_train.shape[1],))
```

Hidden Layer (1), (2):

은닉층으로, 각각의 은닉층의 뉴런이 Feature 갯수의 1/2와 1/4가 되도록 설정합니다. 은닉층에 필요한 초기값, 활성화 함수는 아래와 같은 방식을 이용합니다.

1. 'he_uniform' : he_uniform 으로 초기화를 진행합니다.
2. 'LeakyReLU' : Gradient Vanishing 현상을 피하고 dying Relu 현상을 피하기 위해 LeakyReLU 활성화 함수를 이용해 은닉층을 구성합니다.

```
hidden1=tf.keras.layers.Dense(
    units=int(np.round(X_train.shape[1]/2, 0)),
    kernel_initializer='he_uniform',
    activation='LeakyReLU'
)(inputs)
hidden2=tf.keras.layers.Dense(
    units=int(np.round(X_train.shape[1]/4, 0)),
    kernel_initializer='he_uniform',
    activation='LeakyReLU'
)(hidden1)
```

Output Layer:

출력층으로, 선형회귀분석의 값을 구하는 문제이므로, 하나의 값을 출력하도록 설정해줍니다.

출력값이 제한되어 있지 않으므로 (부호, 범위 등), 활성화 함수를 설정하지 않았습니다.

```
outputs=tf.keras.layers.Dense(
    units=1,
)(hidden2)
```

위 MLP 모델을 요약하면 아래와 같습니다.

Model: "model"

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 397)]	0
dense (Dense)	(None, 198)	78804
dense_1 (Dense)	(None, 99)	19701
dense_2 (Dense)	(None, 1)	100

=====
Total params: 98,605
Trainable params: 98,605
Non-trainable params: 0

MLP모델을 아래와 같은 손실함수와, 옵티마이저를 이용해 컴파일 합니다.

손실함수 : $RMSPE = \sqrt{\frac{1}{N} \sum_{i=1}^N \left(\frac{y_i - \hat{y}_i}{y_i} \right)^2}$ 를 이용합니다. (Optiver 평가지표)

옵티마이저: Adam(Adaptive Moments)를 활용하여 과거 기울기의 변화량을 유지하며 최신 정보를 반영하려고 했습니다.

```
model.compile(optimizer=tf.keras.optimizers.Adam(0.001),
              loss=rmspe)
```

모델의 학습에 있어, 헛곡에 빠지는 문제와, 불필요한 계산을 줄이며 Fitting을 진행합니다.

Early_Stopping: 1e-5의 기준으로, 개선이 없다면, 11번의 에포크를 진행해보고 종료하고, model의 weight를 monitor값이 가장 좋았던 값으로 반환하게 설정합니다.

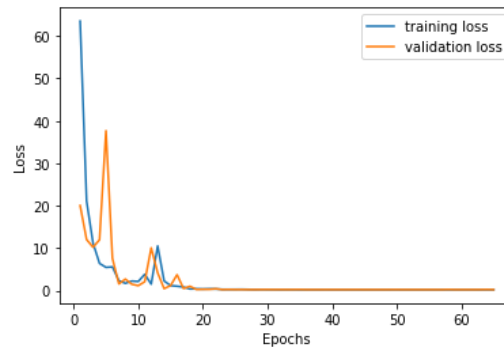
ReduceLROnPlateau: val_loss가 감소하지 않을 경우, epoch를 세번 진행 후, 학습률을 0.5만큼 감소시켜서 fitting을 진행합니다. 이때, 변할수 있는 학습률의 하한을 1e-5 로 설정해 줍니다.

```
rlr = tf.keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=3, min_delta=1e-5, eps=1e-5, verbose=1)
es = tf.keras.callbacks.EarlyStopping(monitor='val_loss', min_delta=1e-5, patience=11, restore_best_weights=True, verbose=1)
```

Batch_size는 500, Epoch는 1000으로 설정 후, 분할된 Validation 데이터를 이용해 학습을 진행하여 결과를 확인합니다.

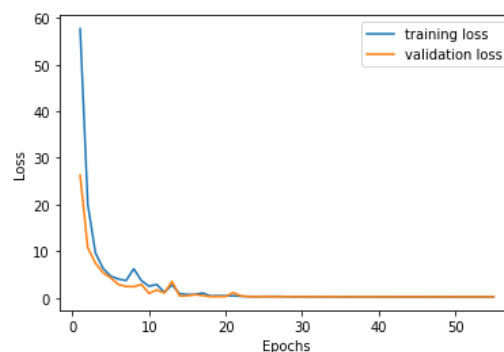
```
history = model.fit(X_train, y_train,
                    batch_size=500, epochs=1000, verbose=1,
                    validation_data=(X_val, y_val), callbacks=callback_list
                    )
```

Nearest Neighbor Feature를 활용해 fitting 후, Test데이터를 이용해 예측을 진행합니다.



RMSPE : 0.2181

Base Feature를 활용해 fitting 후, Test데이터를 이용해 예측을 진행합니다.



RMSPE : 0.2268

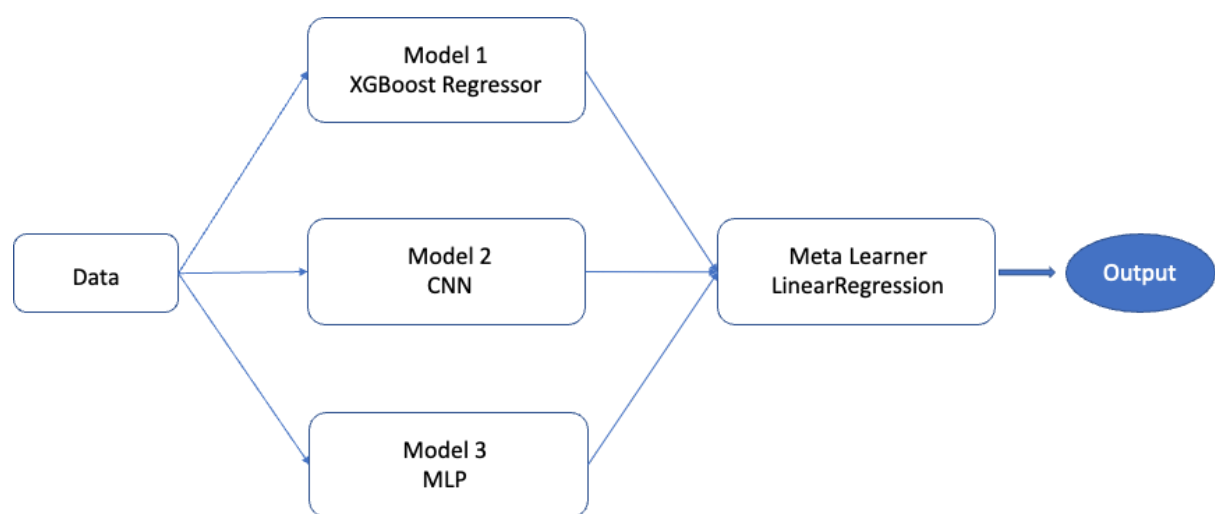
Features가 다른 두 가지 예측 모델들의 결과에 큰 차이는 없으나, **Nearest Neighbor features를 활용한 모델이 좋은 성능을 가지고 있는 것을 확인할 수 있습니다.**



5. Ensemble

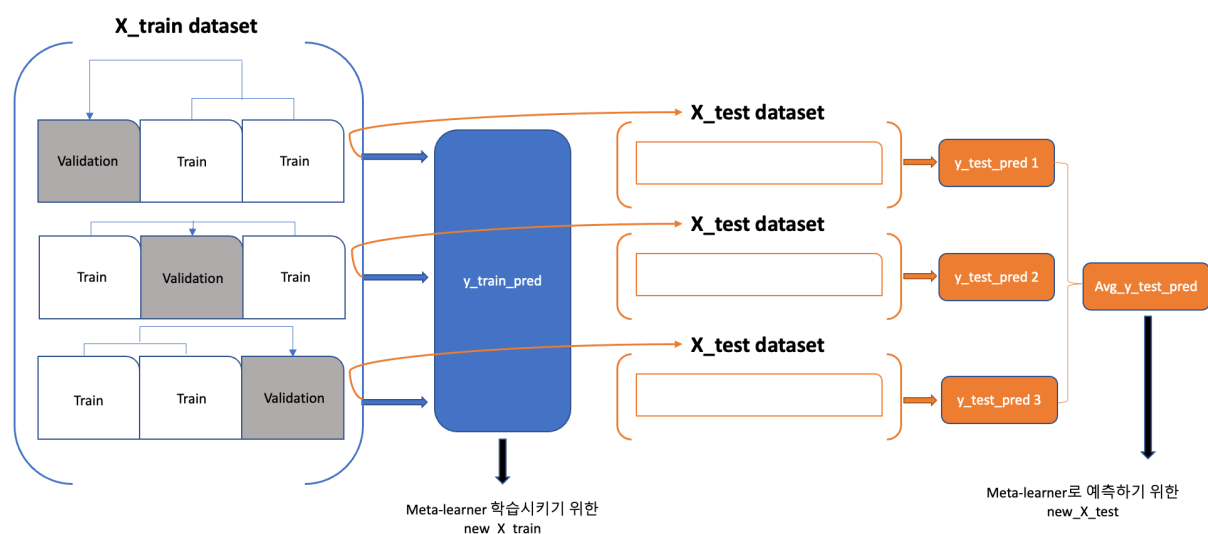
Ensemble

머신러닝에서 앙상블 학습(Ensemble Learning)이란 여러 개의 개별 모델을 조합하여 최적 모델로 일반화 하는 방법입니다. 앙상블을 통해 개별 학습 알고리즘에 대한 편향을 줄이고 최종 예측 결과를 일관성있게 안정적으로 낼 수 있다는 장점이 있습니다. 저희 팀은 XGBoost, CNN, MLP 3개의 학습 모델에 대해 LinearRegression을 Meta-Learner로 하는 앙상블 기법을 적용하여 최종 예측값을 도출하였습니다.



Ensemble Method

앙상블 기법으로 개별 모델을 최종 모델에 다시 학습시키는 것은 과적합을 불러올 수 있습니다. 이를 해결하기 위해 CV기반의 스택킹 앙상블(stacking ensemble) 모델을 구성하였습니다. 구축 과정은 다음과 같습니다.



개별 학습모델 하나에 대한 CV Stacking ensemble 모델 구성 과정

1. X_train 데이터의 fold를 3개로 나눕니다.
2. 모델 하나에 대해 fold로 나누어진 데이터를 기반으로 훈련을 진행합니다. 이 때 사용하는 훈련데이터는 (X_train, y_train)이고, 모델의 하이퍼파라미터는 앞서 튜닝한 결과를 반영하여 적용합니다.
 - a. 각 Fold마다 뽑힌 훈련데이터로 모델을 피팅시키고 검증 데이터를 활용해 예측 후 값을 저장합니다.
 - b. 동시에 피팅된 학습모델에 X_test 데이터를 입력시켜 y_test_pred값을 산출합니다.(fold가 총 3개이므로 각 모델 별 3개의 y_test_pred가 산출되고 이를 평균하여 결국 모델 하나별로 y_test_pred 한 개가 생성됨)
3. 위의 과정을 모든 개별 학습 모델에 대해 동일하게 진행하고, 모델별 예측데이터를 모두 stacking 하여 최종 모델의 훈련 데이터로 사용합니다. 이 때 훈련의 target은 y_train값으로 진행합니다.
4. 과정 2.b에서 생성된 각 모델별 y_test_pred를 피팅된 meta-learner에 넣고 최종 예측값을 산출합니다.

저희 팀의 개별 학습 모델 중 XGBoost 모델에 대한 CV stacking ensemble 코드 예시는 다음과 같습니다.

```
n_folds = 3
kfold = KFold(n_splits=n_folds, shuffle=True, random_state=0)
train_fold_predict = np.zeros((X_train_base.shape[0], 1))
test_predict = np.zeros((X_test_base.shape[0], n_folds))

xgboost_reg = XGBRegressor(n_estimators = 400,
                           learning_rate = 0.04,
                           colsample_bytree = 0.8,
                           subsample = 0.7 )

for cv_num, (train_index, val_index) in tqdm(enumerate(kfold.split(X_train_base))):
    X_train_base_ = X_train_base.iloc[train_index,:]
    y_train_base_ = y_train_base.iloc[train_index]
    X_val_base_ = X_train_base.iloc[val_index,:]

    xgboost_reg.fit(X_train_base_, y_train_base_)

    train_fold_predict[val_index, :] = xgboost_reg.predict(X_val_base_).reshape(-1,1)
    test_predict[:, cv_num] = xgboost_reg.predict(X_test_base)

xgb_test_predict_mean = np.mean(test_predict, axis=1).reshape(-1,1)
xgb_train_predict = train_fold_predict
```

위의 코드를 각 학습모델에 대해서도 동일하게 진행한 뒤, 다음과 같이 예측 결과들을 stacking 하고 최종 meta-learner인 LinearRegression에 학습시킨 후 평가를 진행합니다.

```
new_X_train = np.concatenate((xgb_train_predict, cnn_train_predict, mlp_train_predict), axis=1)
new_X_test = np.concatenate((xgb_test_predict_mean, cnn_test_predict_mean, mlp_test_predict_mean), axis=1)

final_model = LinearRegression()
final_model.fit(new_X_train, y_train)

y_pred_final = final_model.predict(new_X_test)
```

끝으로, base feature로만 앙상블 모델에 적용하여 산출한 최종예측값과 Nearest neighbors feature를 추가하여 산출한 최종예측값의 성능평가를 진행하였으며, 성능평가 결과는 각각 rmspe 0.2433, 0.2370으로 Nearest neighbors feature를 추가하여 학습한 모델이 조금 더 나은 성능을 보여주었습니다.



6. Conclusion

Conclusion

저희는 Kaggle의 Competition 의도에 따라 호가/체결 데이터를 통해 10분 후 변동성을 예측하는 모델을 설계하였습니다. 아래의 그림에서 알 수 있듯이, 어떤 모델을 사용하더라도 저희가 생성한 특성변수를 바탕으로 예측했을 때 75% 이상의 설명력을 보이는 것으로 나타났습니다.

Model \ Features	Base Features	Base Features + Nearest Neighbor Features
XGBoost	Type 1 rmspe 0.2350	Type2 rmspe 0.2208
Ensemble XGBoost + CNN + MLP	Type3 rmspe 0.2433	Type4 rmspe 0.2370

특성변수의 경우, 트레이더들이 매매시 주로 참고하는 지표들로 구성된 base feature보다 KNN 알고리즘을 바탕으로 유사한 realized volatility와 trade size를 가진 종목들의 특성들로 생성한 특성변수를 추가하는 것이 효과적임을 알 수 있었습니다.

또한 단일 XGBoost로 예측한 성능이 앙상블보다 더 좋음을 확인하였습니다. 앙상블을 구성하는 모델을 학습시키고 하이퍼파라미터를 조정하는 과정에서 과적합 문제가 일부 발생하였을 수도 있고, CV 스택킹 작업 시 학습 데이터를 쪼개서 학습을 시키기에 따라 예측 성능이 저하된 것으로 파악됩니다.

그러나 개별 모델의 경우 하이퍼파라미터에 따라 예측결과와 민감도가 높은 반면, 앙상블 모델의 예측값은 meta-learner의 종류를 변경하여도 안정적으로 예측결과를 산출하는 것을 확인할 수 있었습니다.



7. References

Reference

다음은 저희가 참고한 사이트의 링크입니다. Kaggle data 특성상, 대회 개최기간 동안 참가자들이 특성변수를 생성하거나 모델을 선정하는 과정에서 홈페이지의 Discussion 게시판을 통해 다양한 토론이 오갑니다. 저희는 종료된 대회의 데이터를 이용했기 때문에, 해당 대회에서 1등 수상자의 source code를 비롯하여 다양한 reference를 바탕으로 예측을 시도할 수 있었습니다. 다만 예측력을 높이려고 시도하는 동시에 수업에서 학습한 XGBoost의 회귀문제에서의 성능을 테스트 하는 것이 저희 팀프로젝트의 주안점이었기 때문에, 순위권 참가자들보다 설명력이 떨어질 수 밖에 없었던 부분이 존재합니다.

1st Place Solution

Github/Discussion source code

<https://github.com/nyanp/optiver-realized-volatility-prediction/tree/master/notebook>

<https://www.kaggle.com/code/nyanpn/1st-place-public-2nd-place-solution?scriptVersionId=85907908>

Discussion - idea of Nearest Neighbours

<https://www.kaggle.com/competitions/optiver-realized-volatility-prediction/discussion/274970>

Feature Engineering

Feature engineering ideas

<https://www.kaggle.com/competitions/optiver-realized-volatility-prediction/discussion/256080>

Introduction to financial concepts and data

<https://www.kaggle.com/code/jiashenliu/introduction-to-financial-concepts-and-data/notebook>

XGBoost Hyperparameter Tuning

<https://www.kaggle.com/code/lifesailor/xgboost>

CNN

<https://www.kaggle.com/code/tomforbes/keras-cnn-baseline-with-stock-embedding/notebook>

<https://www.kaggle.com/code/slawekbiel/deep-learning-approach-with-a-cnn-inference/notebook>

Ensemble

<https://developer.ibm.com/articles/stack-machine-learning-models-get-better-results/>