신용카드 매출예측

박ㅇㅇ 황ㅇㅇ 김윤명 이ㅇㅇ 1조

01 진행과정

01 우수 코드 분석

DACON X funda 신용카드 매출예측 우승자 2명의 코드 장·단점을 비교·분석

02 우수 코드 병합

분석한 내용을 토대로 코드의 일부분을 취합하여 오류 수정 및 연결

03 내용 추가 및 보완

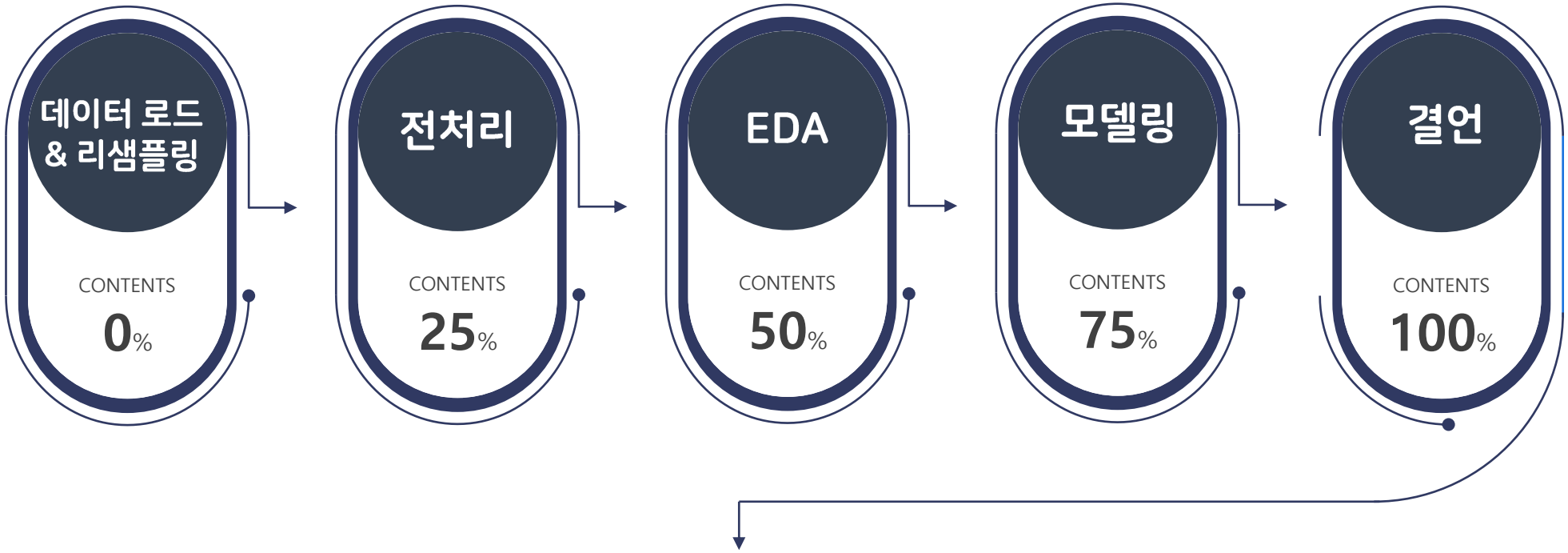
자료조사를 통해 관련 자료, 코드 등을 추가하고 보완을 통해 성능을 소폭 개선

04 자체 코드 완성

SMA, EMA(ETS), ARMA, ARIMA, Regression, LSTM 등 다양한 모델을 이용해 모델링

02

분석과정



신용카드 매출 예측

03 Issues

01

예측해야 하는 범위는 3개월 인데 데이터는 시간 단위로 나뉘어져 있음

-> month 단위로 resampling 후 예측 범위를 3개월로 지정

02

1967개의 store_id가 각각 다른 trend와 seasonality 를 가지고 있음

-> 모델을 여러 개 적용해 보면서 정확도와 성능이 가장 좋았던 모델을 적용

03

예측 날짜는 2019-03~2019-05로 동일하나, 제공 데이터의 **마지막 날짜**는 차이가 있음

-> **마지막 날짜부터 3개월만 예측**하여 제출한다.

03

ex: store_id 111의 마지막 날짜는 2018-09월로 뒤 3개월인 2018-10~2018-12만 예측하여 제출

- 예측 기간이 길어질수록 오차가 크게 발생하여 **바로 뒤 3개월만 예측**하는 것이 정확도가 높았음

03 Issues

04

amount<0인 값에 대한 해석과 처리가 애매

-> amount<0인 값을 포함 시켜도 보고 분리시켜도 보아 둘 중 성능이 좋은 쪽으로 결정 : 분리시킬 때가 더 좋았음

05

딥러닝 시 raw-data 값이 적어 학습할 data 양이 충분하지 못함

-> time-series 모델을 통해 모델링하여 예측 정확도 향상

04 Data Load & Resampling

01

*** 분석 목적**

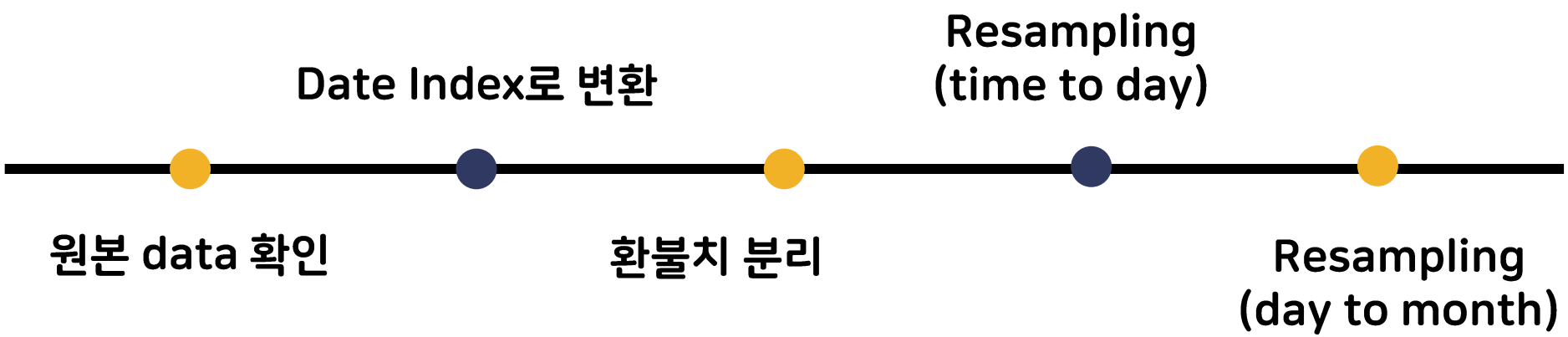
상환 기간의 매출을 예측하여
신용 점수가 낮거나 담보를
가지지 못하는 우수 상점들에
금융 기회를 제공

02

*** 분석 목표**

2019-02-28까지의
카드 거래 데이터를 이용해
시계열 매출 분석을 통해
2019-03-01~2019-05-31
각 상점별 3개월 총 매출 예측

04 *Data Load & Resampling*



04 *Data Load & Resampling*

Data List

- store_id : 상점의 고유 아이디
- card_id : 사용한 카드의 고유 아이디
- card_company : 비식별화된 카드 회사
- trasacted_date : 거래 날짜
- transacted_time : 거래 시간(시:분)
- installment_term : 할부 개월 수
- region : 상점의 지역
- type_of_business : 상점의 업종
- amount : 거래액(단위는 원이 아님)

-> 종속변수

Evaluation Metric(측정 척도)

MAE(Mean Absolute Error)

- 모든 절대 오차의 평균

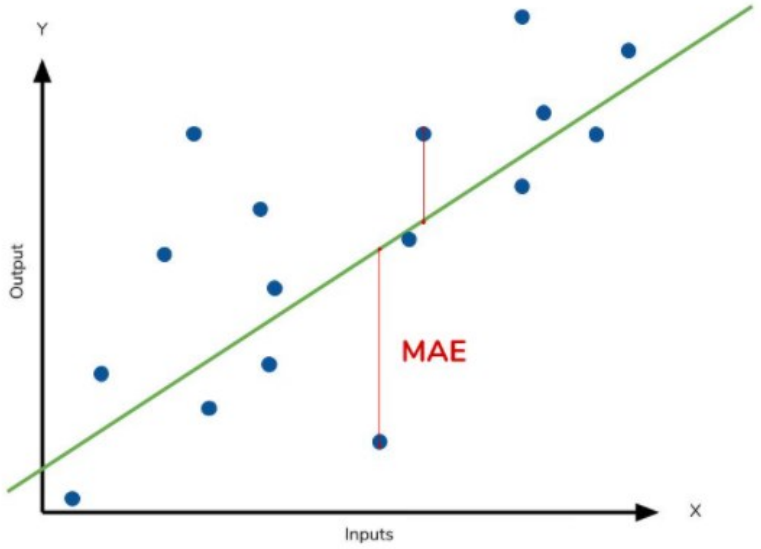
- 통계적 추정의 정확성에 대한 질적인 척도(수치가 작을수록 정확성이 높은 것)

- 에러에 따른 손실이 선형적으로 올라가는 상황에서 쓰기 적합한 방식

- MAE를 토대로 validation

Data Load & Resampling

Evaluation Metric(측정 척도)




```python
def mae(prediction, correct):   # prediction, correct를 인자로 받음
    prediction = np.array(prediction)
    correct = np.array(correct)

    difference = correct - prediction
    abs_val = abs(difference) # 절댓값 오차

    score = abs_val.mean() # 절댓값 오차의 평균(MAE)

    return score
```

executed in 13ms, finished 16:00:41 2021-12-29

```python
mae_scorer = make_scorer(mae) # MAE를 토대로 validation
mae_scorer
make_scorer(mae)
```

executed in 13ms, finished 16:00:41 2021-12-29

```
make_scorer(mae)
```

1) 원본 data 확인

	store_id	card_id	card_company	transacted_date	transacted_time	installment_term	region	type_of_business	amount
0	0	0	b	2016-06-01	13:13	0	NaN	기타 미용업	1857.14286
1	0	1	h	2016-06-01	18:12	0	NaN	기타 미용업	857.14286
2	0	2	c	2016-06-01	18:52	0	NaN	기타 미용업	2000.00000
3	0	3	a	2016-06-01	20:22	0	NaN	기타 미용업	7857.14286
4	0	4	c	2016-06-02	11:06	0	NaN	기타 미용업	2000.00000
...	...	...	...	...	...	...	...	...	...
6556608	2136	4663855	d	2019-02-28	23:20	0	제주 제주시	기타 주점업	-4500.00000
6556609	2136	4663855	d	2019-02-28	23:24	0	제주 제주시	기타 주점업	4142.85714
6556610	2136	4663489	a	2019-02-28	23:24	0	제주 제주시	기타 주점업	4500.00000
6556611	2136	4663856	d	2019-02-28	23:27	0	제주 제주시	기타 주점업	571.42857
6556612	2136	4658616	c	2019-02-28	23:54	0	제주 제주시	기타 주점업	5857.14286

2) Data Index로 변환

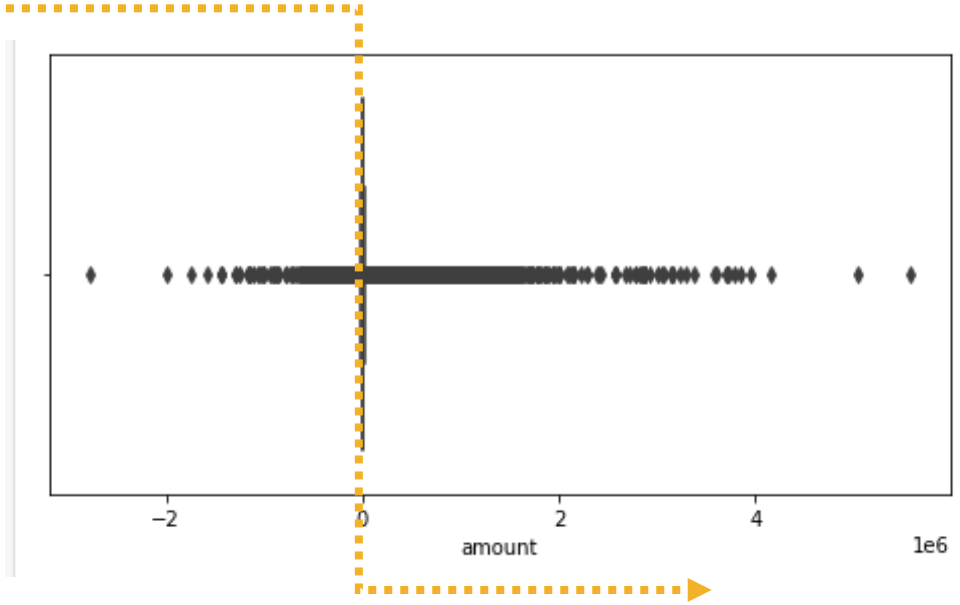
```
df_train['transacted_date'] = pd.to_datetime(df_train.transacted_date + " " +
                                             df_train.transacted_time, format='%Y-%m-%d %H:%M:%S')
```

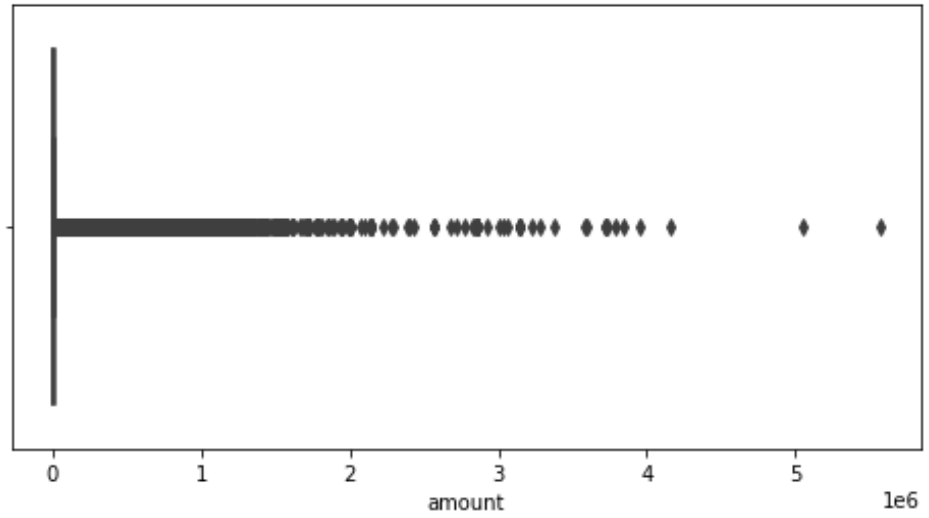
3) 환불치 분리

```
def refund_remove(df):
    refund=df[df['amount']<0]
    non_refund=df[df['amount']>0]
    remove_data=pd.DataFrame()

    for i in tqdm(df.store_id.unique()):
        divided_data=non_refund[non_refund['store_id']==i] ##non_refund 스토어 데이터를 스토어별로 나눔
        divided_data2=refund[refund['store_id']==i] ##refund 스토어 데이터를 나눔 스토어별로 나눔
```

3) 환불치 분리





> 환불치를 분리하지 않았을 때 보다 분리했을 때가 예측 정확도가 더 좋았다.

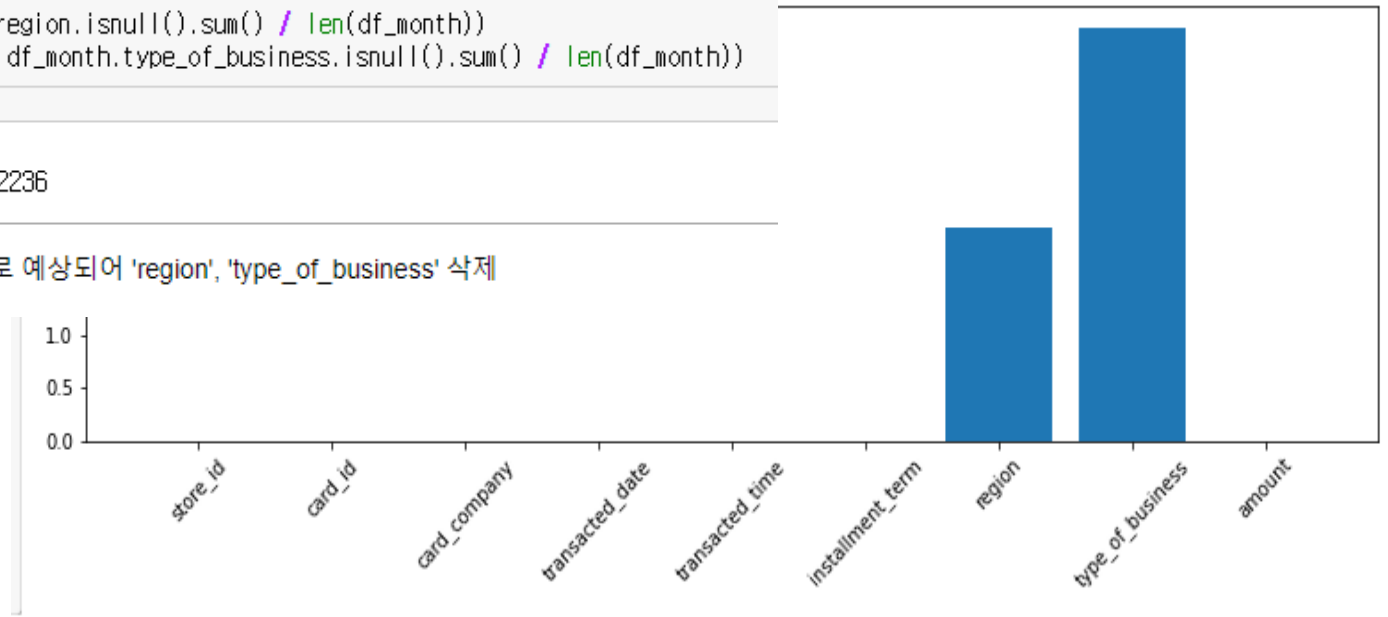
결측치 제거

```
# 결측치 비율
print("rate of 'region' :" , df_month.region.isnull().sum() / len(df_month))
print("rate of 'type_of_business' :" , df_month.type_of_business.isnull().sum() / len(df_month))
```

executed in 14ms, finished 15:30:00 2021-12-28

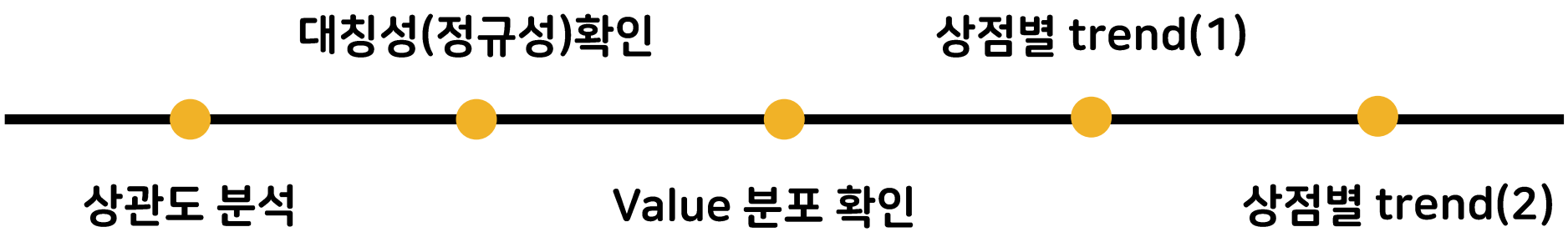
```
rate of 'region' : 0.34310104243618156
rate of 'type_of_business' : 0.5917754247722236
```

- null값 채우기 어렵고 채워도 오차가 클 것으로 예상되어 'region', 'type_of_business' 삭제



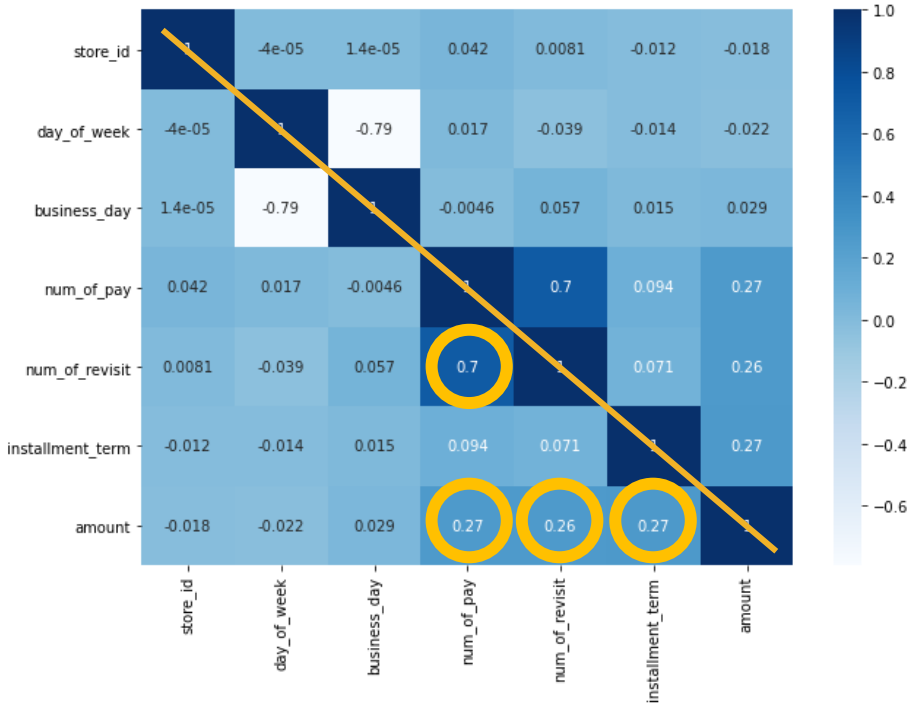
06

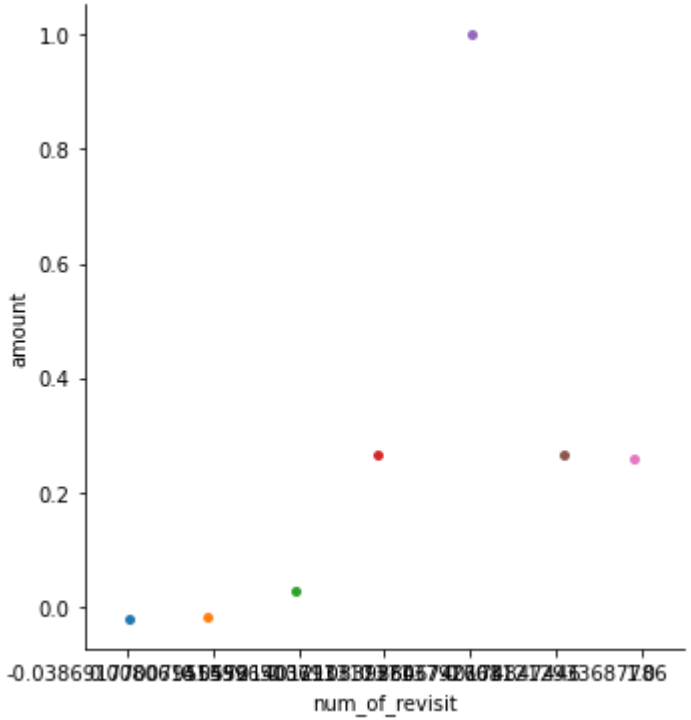
EDA(Exploratory Data Analysis)



EDA(Exploratory Data Analysis)

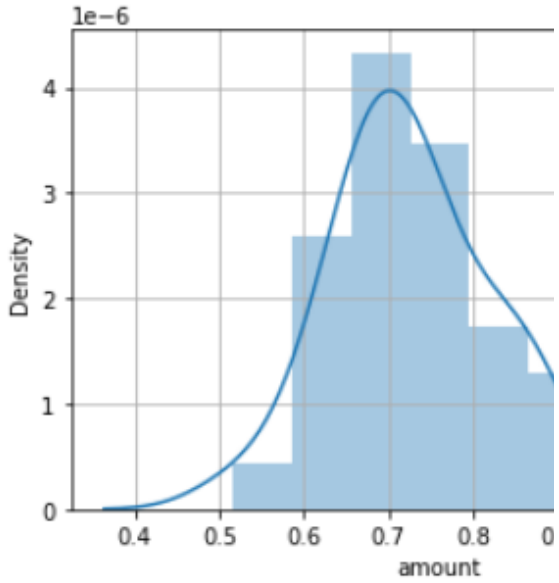
1) 상관도 분석



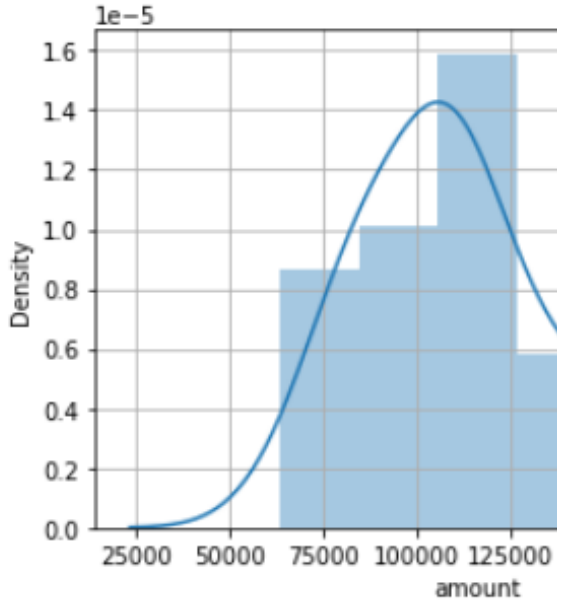


2) 대칭성(정규성)확인

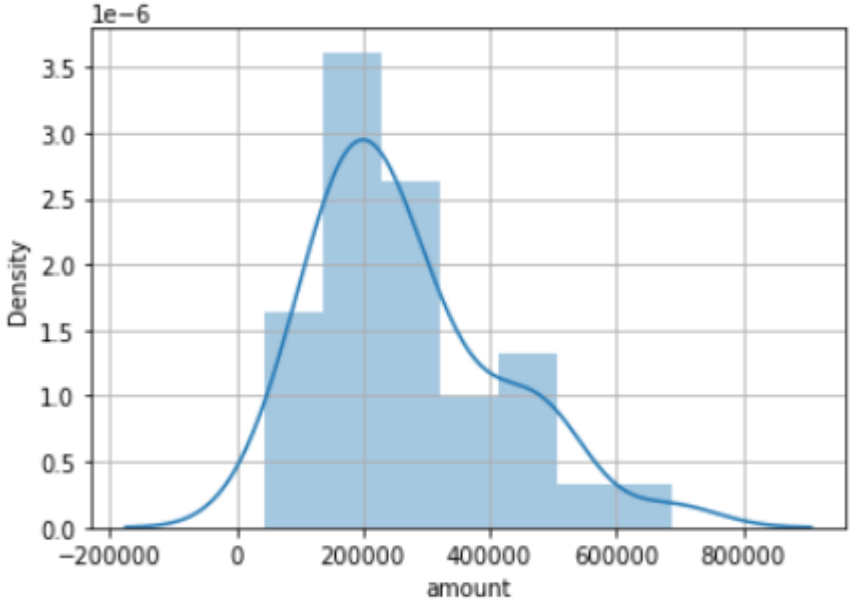
Skewness : 0.48245510908401146



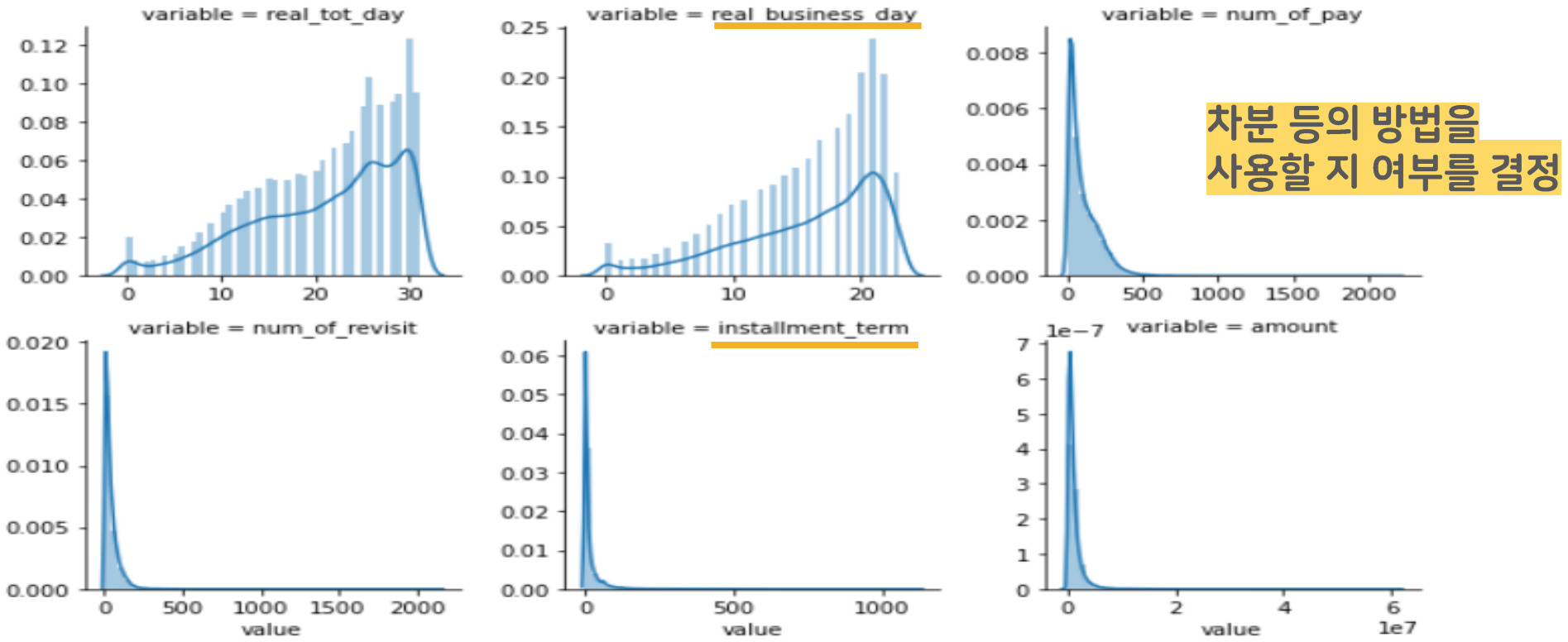
Skewness : 0.5102521050694766



Skewness : 0.9982599271676978

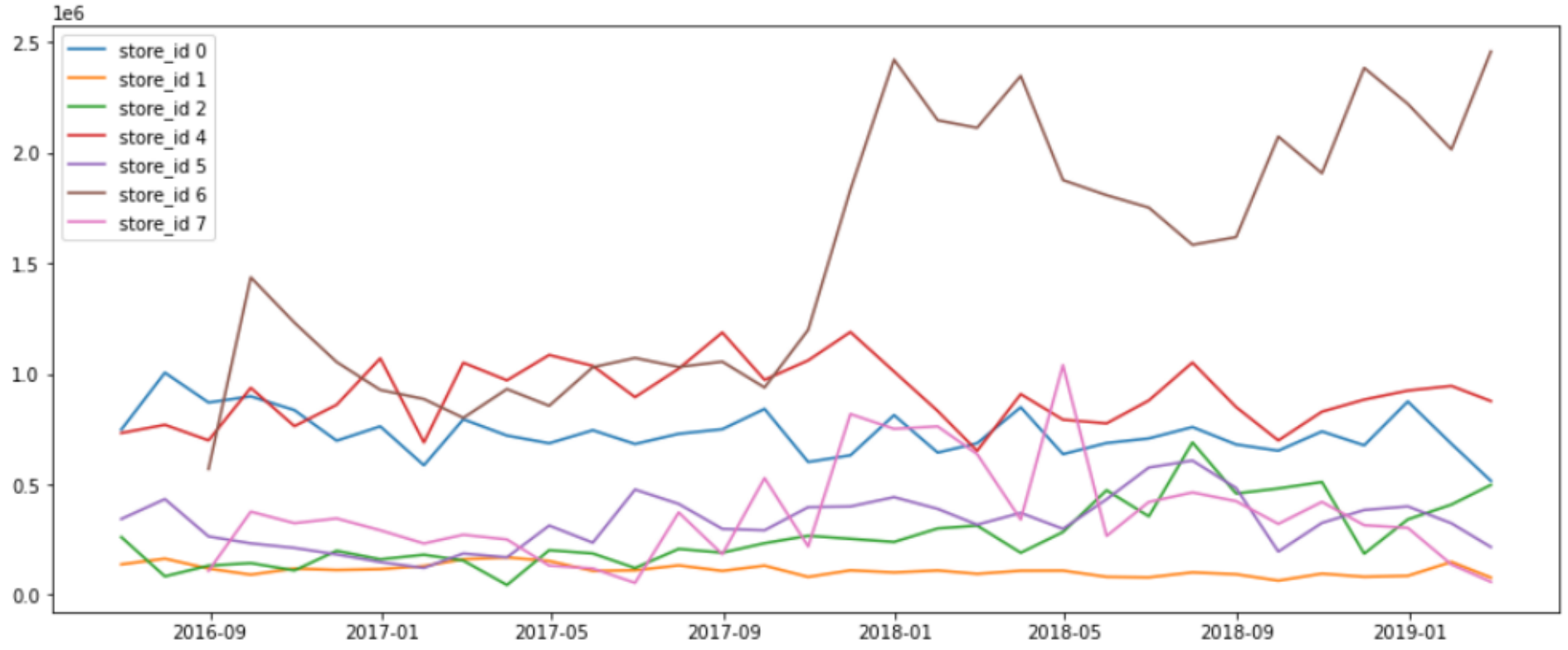


3) 각 변수의 value 분포

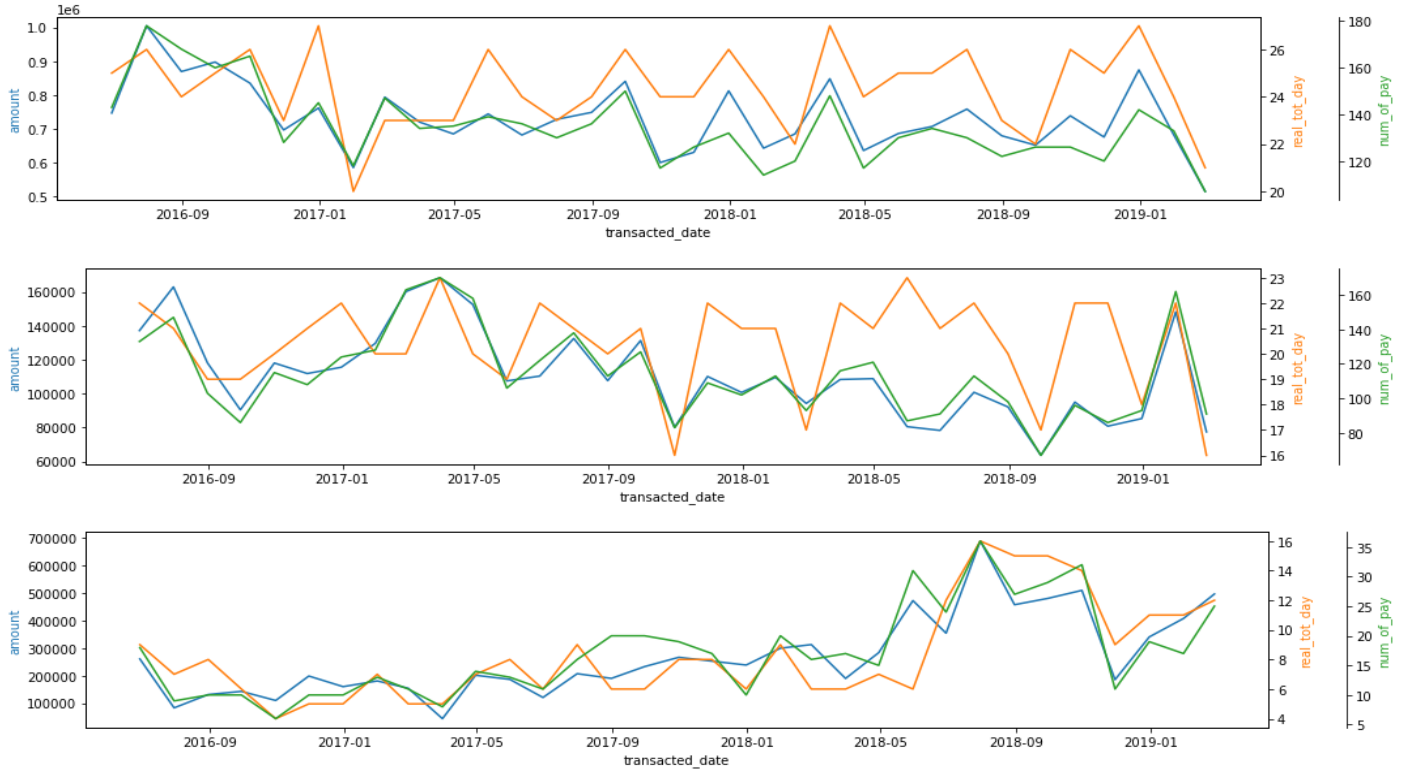


차분 등의 방법을
사용할 지 여부를 결정

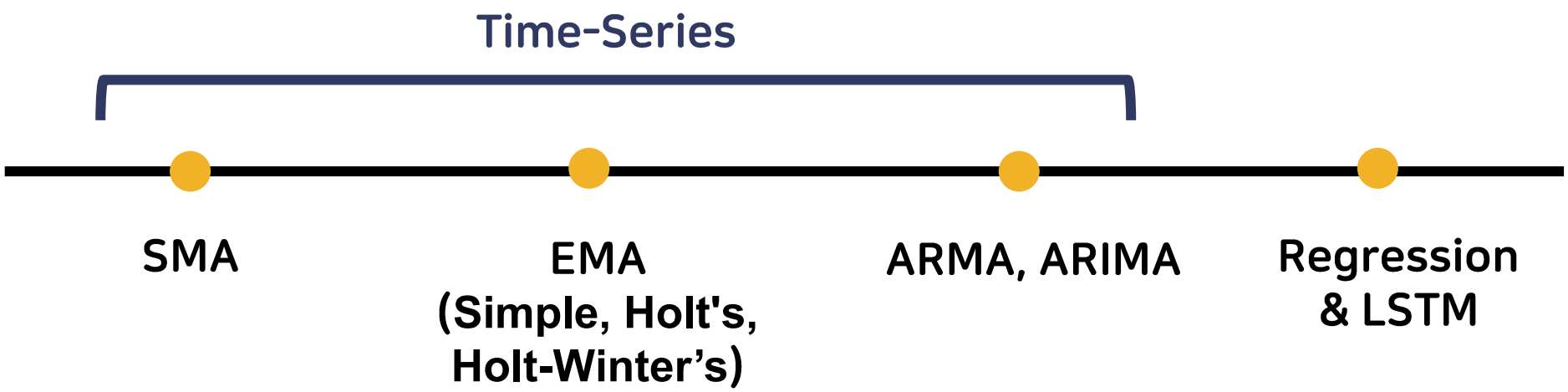
4) Store 별 amount trend



5) Store 0,1,2의 amount, 전체 일수, 매출 횟수 별 trend



-> 다른 변수가 amount와 같은 **패턴**을 갖는지 확인


Time-Series
SMA
EMA
(Simple, Holt's,
Holt-Winter's)
ARMA, ARIMA
Regression
& LSTM

1) SMA(Simple Moving Average)

* MA(Moving Average;이동평균) : 구하고자 하는 전체 데이터의 일부분(subset)

  에 대해 순차적(series)으로 평균을 구하는 것

- Stationary 시계열을 대상으로 분석

- MA에는 SMA(Simple Moving Average, 단순이동평균),

  WMA(Weighted Moving Average,가중이동평균) 등이 있음

- SMA는 가격의 이동평균을 계산할 때 가장 일반적인 방법

1) SMA(Simple Moving Average)

- SMA는 **특정 기간** 동안의 대표적인 data를 단순 평균하여 계산하며, 그 안에는 그 동안의 data의 움직임이 포함되어 있음
- 이 때 수학적으로 n/2 시간 만큼의 **지연(lag)**이 발생
- 단순이동평균은 **모든 데이터의 중요도**를 **동일**하다고 간주

$$SMAt=Dt-(n-1)+Dt-(n-2)+\cdots+Dt-1+Dtn$$

1) SMA(Simple Moving Average)

```
def make_sma_arr(window_num):
    ma_arr = np.array([])
    for i in df_month.store_id.unique():
        df_set = df_month[df_month.store_id == i]
        ma_arr = np.concatenate((ma_arr, df_set.amount.rolling(window=window_num).mean().values)) # 이동평균 계산

    return ma_arr
```

executed in 13ms, finished 08:59:38 2021-12-30

```
sma_month = df_month.copy()

sma_month.insert(7, 'amount_2ma', make_sma_arr(2)) # 2개월치 평균치
sma_month.insert(8, 'amount_3ma', make_sma_arr(3))
sma_month.insert(9, 'amount_6ma', make_sma_arr(6))
```

executed in 4.62s, finished 08:59:42 2021-12-30

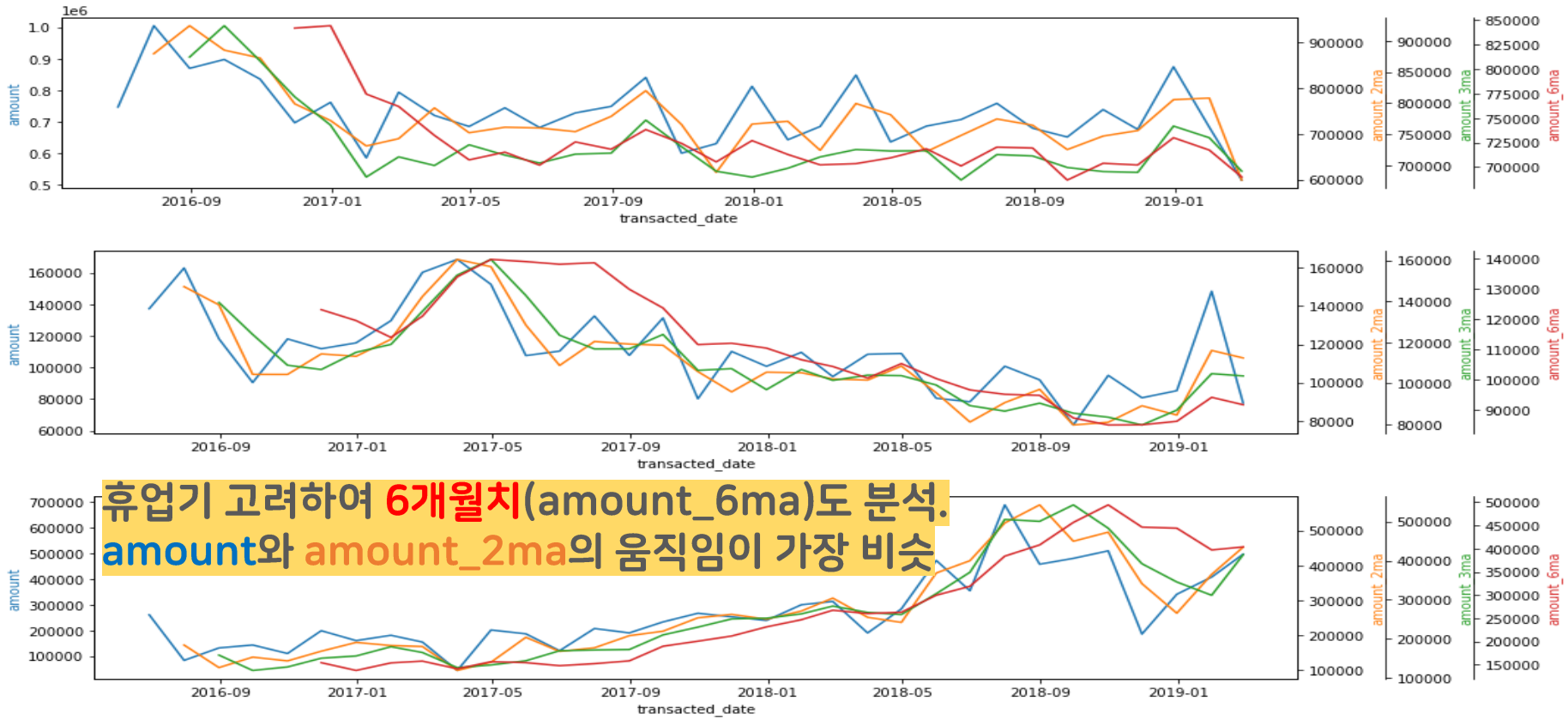
```
sma_month.head(7)
# 2달의 평균으로 876000 만들고, 3달의 평균으로 938285 만들고...
```

executed in 28ms, finished 08:59:43 2021-12-30

1) SMA(Simple Moving Average)

transacted_date	store_id	real_tot_day	real_business_day	num_of_pay	num_of_revisit	installment_term	amount	amount_2ma	amount_3ma	amount_6ma
2016-06-30	0	25	17.00000	145.00000	77.00000	13.00000	747000.00000	NaN	NaN	NaN
2016-07-31	0	26	16.00000	178.00000	105.00000	24.00000	1005000.00000	876000.00000	NaN	NaN
2016-08-31	0	24	16.00000	171.00000	97.00000	69.00000	871571.42857	938285.71429	874523.80952	NaN
2016-09-30	0	25	19.00000	160.00000	103.00000	15.00000	897857.14286	884714.28571	924809.52381	NaN
2016-10-31	0	26	16.00000	167.00000	115.00000	9.00000	835428.57143	866642.85714	868285.71429	NaN
2016-11-30	0	23	15.00000	132.00000	93.00000	21.00000	697000.00000	766214.28571	810095.23810	842309.52381
2016-12-31	0	27	18.00000	145.00000	103.00000	11.00000	761857.14286	729428.57143	764761.90476	844785.71429

1) SMA(Simple Moving Average)



1) SMA(Simple Moving Average)

```
def make_minus_rolling(data_frame, rolling_num):
    def minus_shift_rolling(df_num, num):
        a = np.average(df_num.values[-num:])
        b = np.average(np.append(df_set.values[-(num-1):], a))
        if num > 2:
            c = np.average(np.append(np.append(df_set.values[-(num-2):], a), b))
        else:
            c = np.average((a, b))
        return np.sum((a, b, c))

    minus_rolling_arr = np.array([])
    for i in data_frame.store_id.unique():
        df_set = pd.DataFrame(data_frame[data_frame.store_id == i].amount)
        minus_rolling_arr = np.concatenate((minus_rolling_arr, np.array([minus_shift_rolling(df_set, rolling_num)])))

    df_rolling = pd.DataFrame({'store_id' : df_sub.store_id, 'amount' : minus_rolling_arr})

    return df_rolling
```

executed in 13ms, finished 08:59:44 2021-12-30

독자적인 code, 기준 달을 구하기 위해서 과거의 두 달의 매출을 기준 삼아 누적시키는 방향으로 이동평균 구함

1) SMA(Simple Moving Average)

- 2 window SMA MAE Score : 180687.25906004856
- 3 window SMA MAE Score : 251607.4552831229
- 6 window SMA MAE Score : 387465.0993765708

- SMA 2 rolling Score : 836184.506520
- SMA 3 rolling Score : 831158.397180
- SMA 4 rolling Score : 854300.339380

2) EMA(Exponential Moving Average)

- 지수이동평균, ETS(Exponential Smoothing; 지수평활법)라고도 함
- 가중이동평균 중의 하나로 과거보다 최근의 데이터에 높은 가중치를 부여하는 방법 (Exponentially 하게 Weight가 감소)
- 데이터 평균값이 시간에 따라 변화 경향 (주식의 5,10,20일 이동평균선)
- Smoothing : 평균값이 서서히 변화하는 data에 적용 가능(결과 bad)

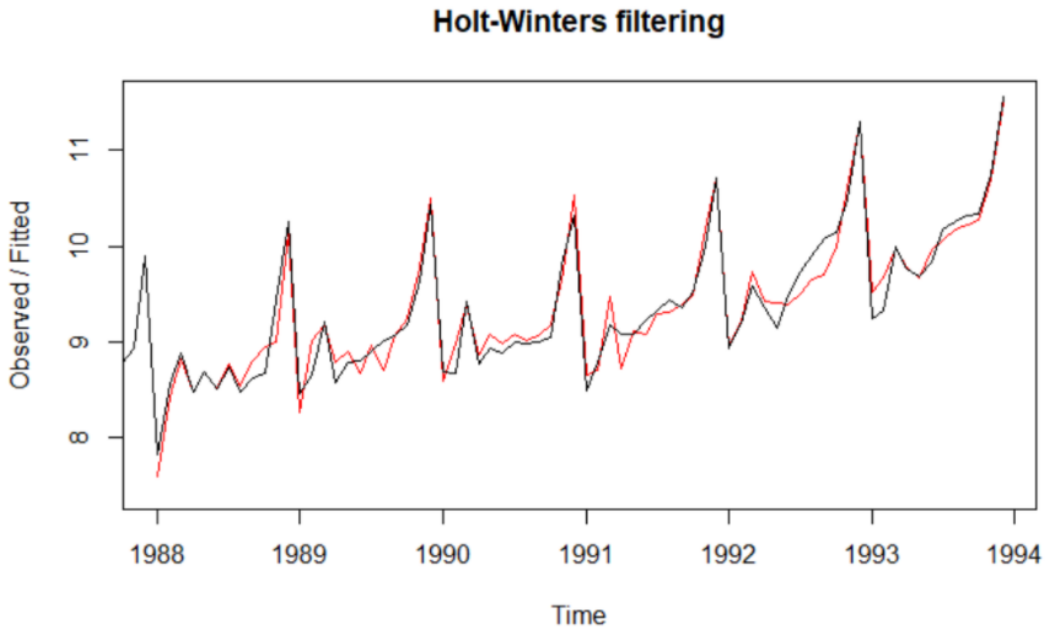
$$EMAt=Dt\times\frac{2}{N+1}+EMAt-1\times(1-\frac{2}{N+1})$$

2) EMA(Exponential Moving Average)

- Simple exponential smoothing : Trend, Seasonal 패턴이 없을 때 씀
- Double(Holt's) exponential smoothing : linear한 trend o, seasonal 패턴 x 일때 씀.
- Triple(Holt-Winters) exponential smoothing: 수준(Level), 추세(Trend), 계절성 (Seasonality) 3가지 정보를 고려. 추세와 계절성이 뚜렷한 데이터에 대해 좋은 예측값을 제공함(프로젝트 data:x) → Time series는 보통 trend, seasonal pattern을 동반하기 때문에 Time series를 고려한 Exponential Smoothing을 이야기할 때 보통 Holt-Winters 를 말함

2) EMA(Exponential Moving Average)

Holt-Winters model 적용 예시: data를 log scale로 변환 후 사용



2) EMA(Exponential Moving Average)

```
1  def make_ewm_arr(data_frame, span_num):
2      arr_ewm = np.array([])
3      for i in data_frame.store_id.unique():
4          df_set = data_frame[data_frame.store_id == i]
5          # 여기에서 지정하는 span값은 위 수식에서 N에 해당한다.
6          arr_ewm = np.concatenate((arr_ewm, df_set.amount.ewm(span=span_num).mean().values))
7
8      return arr_ewm
```

executed in 12ms, finished 08:59:50 2021-12-30

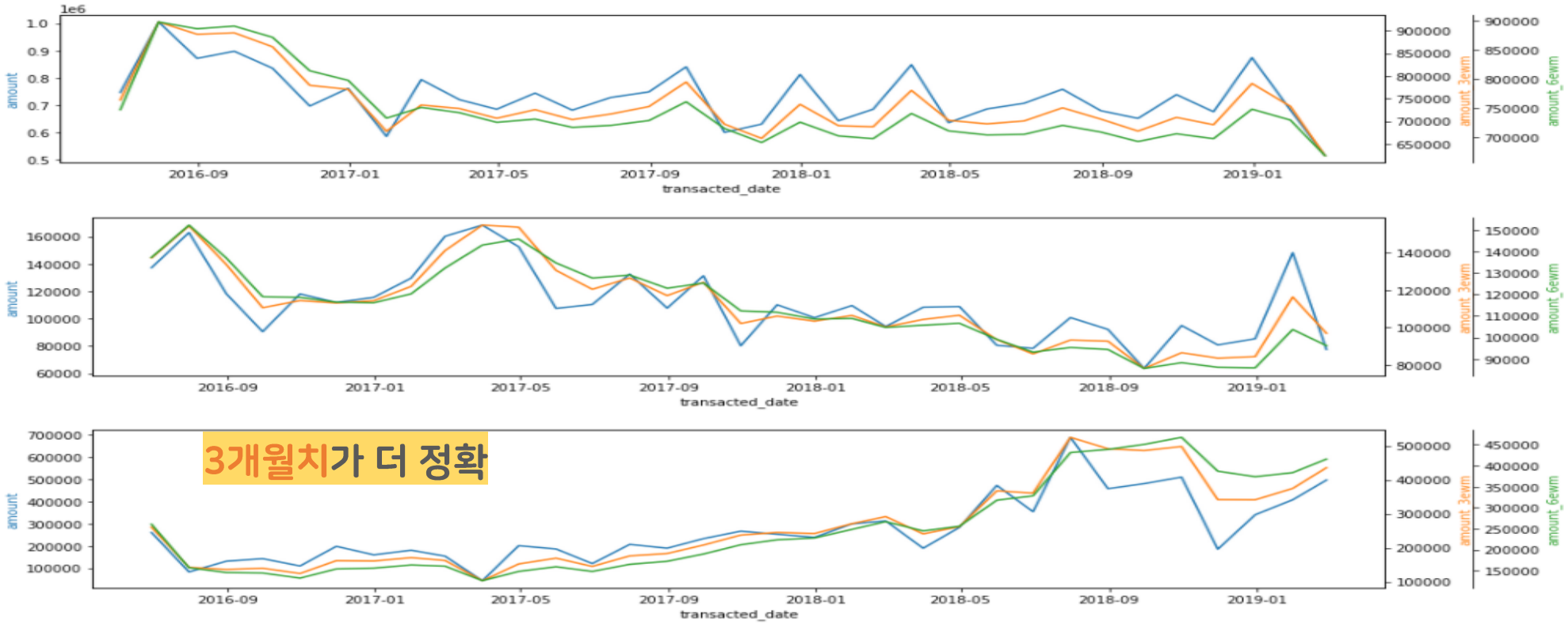
https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.ewm.html

2) EMA(Exponential Moving Average)

transacted_date	store_id	real_tot_day	real_business_day	num_of_pay	num_of_revisit	installment_term	amount	amount_3ewm	amount_6ewm
2016-06-30	0	25	17.00000	143.00000	74.00000	9.00000	747000.00000	747000.00000	747000.00000
2016-07-31	0	26	16.00000	178.00000	105.00000	24.00000	1005000.00000	919000.00000	897500.00000
2016-08-31	0	24	16.00000	168.00000	94.00000	69.00000	869714.28571	890836.73469	885009.17431
2016-09-30	0	25	19.00000	160.00000	103.00000	15.00000	897857.14286	894580.95238	889971.84685
2016-10-31	0	26	16.00000	165.00000	115.00000	9.00000	835428.57143	864050.69124	870828.68002
2016-11-30	0	23	15.00000	128.00000	89.00000	21.00000	697000.00000	779199.54649	813557.08461

가중치를 현재의 데이터에 더 두고 있음

2) EMA(Exponential Moving Average)



2) EMA(Exponential Moving Average)

```python
def make_wma_sub(data_frame, span_num):
    concat_3mon = pd.DataFrame(index=pd.to_datetime(['2019-03-31', '2019-04-30', '2019-05-31'])) # 예측할 데이터
    wma_sub = np.array([])

    for i in df_month.store_id.unique():
        df_set = pd.DataFrame(data_frame[data_frame.store_id == i].amount)
        wma_train = pd.concat([df_set, concat_3mon], axis=0)

        num_sub = np.array([wma_train.amount.ewm(span=span_num).mean()['2019-03':].sum()])

        wma_sub = np.concatenate((wma_sub, num_sub))

    df_wma_sub = pd.DataFrame({'store_id' : df_sub.store_id, 'amount' : wma_sub})

    return df_wma_sub
```

executed in 14ms, finished 08:59:54 2021-12-30

독자적인 code, 지수평활 시 구간을
여러 개로 나눠보면서 가장 성능 높은 것 찾기

```python
for i in range(2, 7):   # 한달치씩 추가해서 2개월치, 3개월치, ..., 6개월치 EMA 구하기 > 가장 성능 높은 것 찾기!
    wma_sub = make_wma_sub(df_month, i)
    wma_sub.to_csv('funda_{}wma_sub.csv'.format(i), index=False)

# 낮을수록 좋은 것: 5를 span으로 했을 때 가장 좋은 값

for i in range(len(df_month.amount)):
    result = wma_sub.amount.mean() * df_month.real_tot_day.iloc[-i+2:-i].mean()
    print(result)
```

2) EMA(Exponential Moving Average)

- 3 N EWM MAE Score : 134855.11946915495
- 6 N EWM MAE Score : 193447.63093078104
- 2 span 제출 Score : 820102.106670
- 3 span 제출 Score : 785488.281930
- 4 span 제출 Score : 770667.895320
- 5 span 제출 Score : 767498.551420 <- 5개월로 구간을 잡았을 때
- 6 span 제출 Score : 770414.027040

2) EMA(Exponential Moving Average)

Library 사용하여 Smoothing

```
4 train = df_set[:size]
5 test = df_set[size:]
6
7 ses_model = SimpleExpSmoothing(train.amount)
8 ses_result = ses_model.fit()
9 ses_pred = ses_result.forecast(len(test))
```

executed in 43ms, finished 17:29:32 2021-12-30

```
C:₩Users₩ym₩anaconda3₩envs₩store_amount_prediction₩lib₩site-packages₩st
  % freq, ValueWarning)
C:₩Users₩ym₩anaconda3₩envs₩store_amount_prediction₩lib₩site-packages₩st
  ConvergenceWarning)
```

```
1 print("store_id 0 mean value : ", df_set.amount.mean())
2 print("MAE Score of test :", mae(test.amount, ses_pred))
```

executed in 12ms, finished 17:29:32 2021-12-30

```
store_id 0 mean value :  732503.463203463
MAE Score of test : 78181.66728282764
```

2) EMA(Exponential Moving Average)

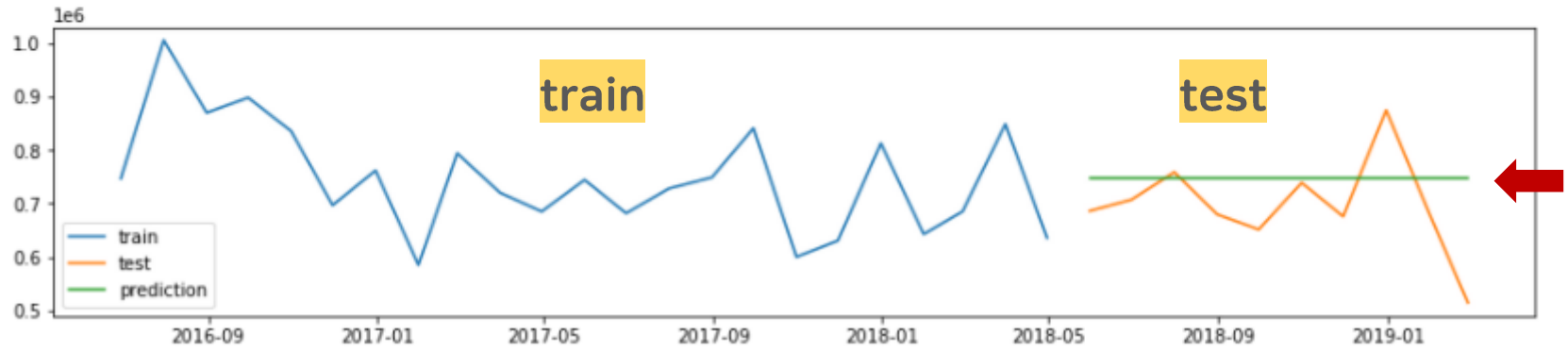
```
1  def plot_train_test_pred_graph(trainset, testset, pred):
2      plt.figure(figsize=(15,3))
3      plt.plot(trainset.amount, label='train')
4      plt.plot(testset.amount, label='test')
5      plt.plot(testset.index, pred, label='prediction')
6      plt.legend()
7      plt.show()
```

executed in 14ms, finished 17:29:32 2021-12-30

Simple Exponential Smoothing

```
1  plot_train_test_pred_graph(train, test, ses_pred)
```

executed in 215ms, finished 17:29:32 2021-12-30



train

test

2) EMA(Exponential Moving Average)

Holt's Exponential Smoothing

```
df_set = df_month[df_month.store_id == 0]

size = int(len(df_set) * 0.7)
train = df_set[:size]
test = df_set[size:]

holt_model = Holt(np.array(train.amount))
holt_result = holt_model.fit()
holt_pred = holt_result.forecast(len(test))
```

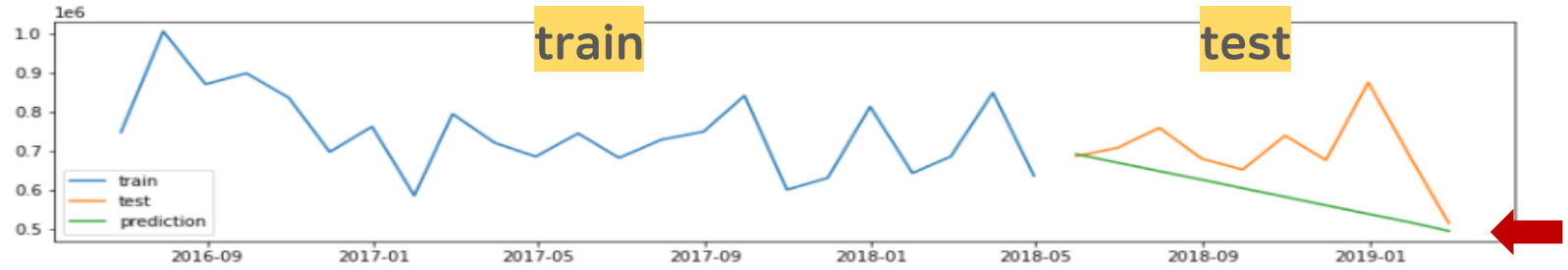
executed in 29ms, finished 17:29:58 2021-12-30

```
print("Mean value of store_id 0 : ", df_set.amount.mean())
print("MAE Score of test :", mae(test.amount, holt_pred))

plot_train_test_pred_graph(train, test, holt_pred)
```

executed in 198ms, finished 17:29:58 2021-12-30

```
Mean value of store_id 0 :  732503.463203463
MAE Score of test : 104762.88757316317
```



train

test

2) EMA(Exponential Moving Average)

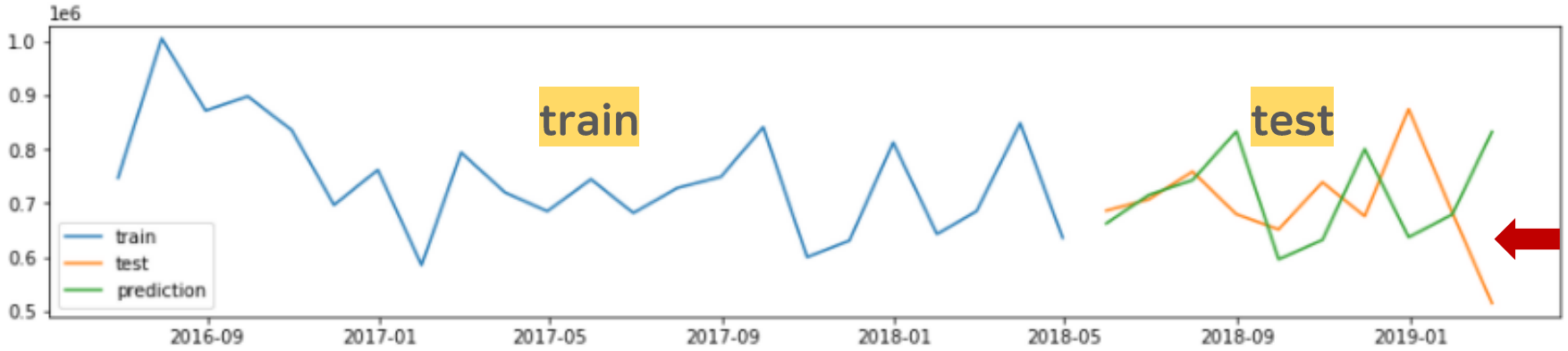
```
print("Mean value of store_id 0 :", df_set.amount.mean())
print("MAE Score of test :", es_score)
print("Best seasonal period :", best_period)

plot_train_test_pred_graph(train, test, es_pred)
```

executed in 261ms, finished 09:02:00 2021-12-30

Holt Winter's Exponential Smoothing

```
Mean value of store_id 0 : 732559.7402597402
MAE Score of test : 63279.99370008395
Best seasonal period : 4
```



train

test

2) EMA(Exponential Moving Average)

- Simple Exponential Smoothing Score : 818205.82245

- Holt's Exponential Smoothing Score : 922772.7023052298

- Holt-Winter's Exponential Smoothing Score : 962259.599880

3) ARMA & ARIMA (Auto Regressive Integrated Moving Average)

ARMA : stationary 시계열을 대상으로 분석. AR(p) 모형과 MA(q) 모형의 특징을 모두 가지는 모형으로, 자기 자신의 p개의 과거값과 q개의 과거 백색 잡음의 선형 조합으로 현재의 값이 정해지는 모형

- 이 data에서 많은 store_id들이 $|\varphi|<1$ 의 AR 정상상태(stationary) 조건에 맞지 않아 1 이상의 p 값을 적용할 수 없다. → 따라서 차분(difference)을 이용해 비정상상태(non-stationary)의 설명이 가능한 ARIMA를 추가로 진행한다.

3) ARMA & ARIMA (Auto Regressive Integrated Moving Average)

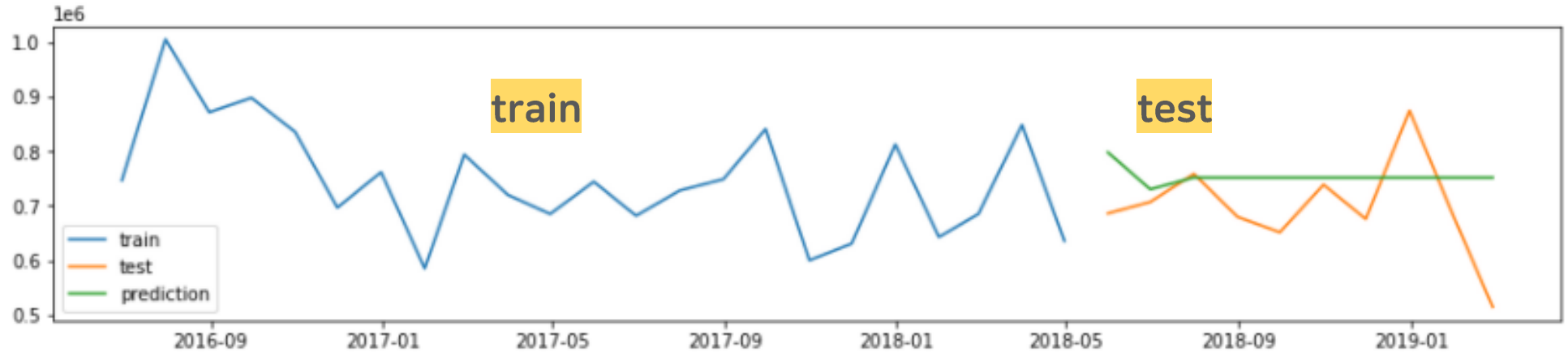
ARIMA : non-stationary 시계열을 대상으로 분석

- d차 차분한 데이터에 AR(p)모형과 MA(q)모형을 합친 모형
- AR(자기상관) + I(d차 차분) + MA(이동평균)
- 자기상관 : 자기 자신 이전의 값이 이후의 값에 영향을 미치는 상황
- non-stationary 시계열을 차분해서 stationary 시계열로 변환한 후, stationary 모형 (ARMA)으로 분석한 다음 차분 시계열을 다시 누적해서 원 시계열로 복원

3) ARMA & ARIMA (Auto Regressive Integrated Moving Average)

ARMA

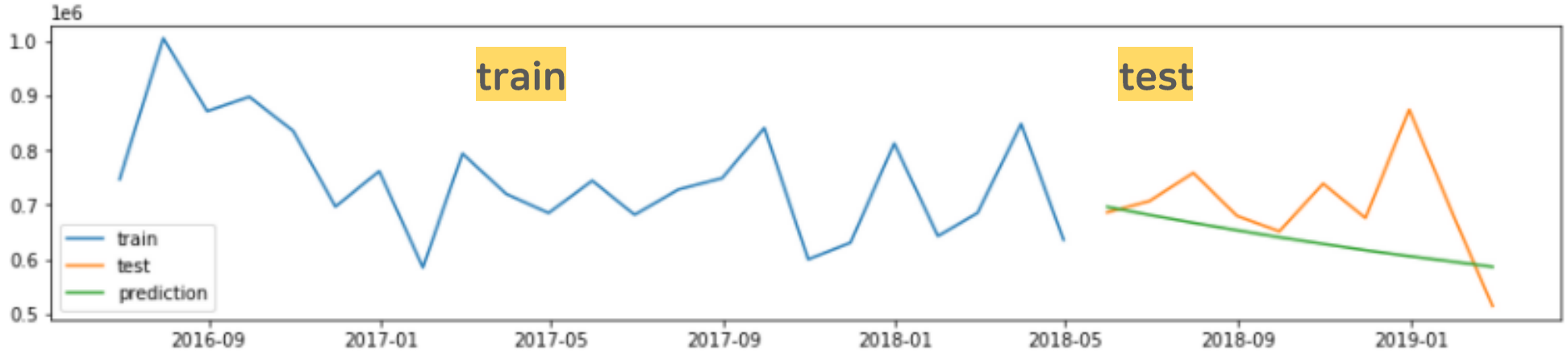
```
Mean value of store_id 0 : 732559.7402597402
MAE Score of test : 83125.43549192042
```



3) ARMA & ARIMA (Auto Regressive Integrated Moving Average)

ARIMA

```
Mean value of store_id 0 : 732559.7402597402
AIC Score of test : 556.1804194299168
Best parameter of (p, d, q):  (0, 2, 2)
```



3) ARMA & ARIMA (Auto Regressive Integrated Moving Average)

-ARMA Score : 984368.752690

-ARIMA MAE Score : 1080182.482790

4) Facebook Prophet

- 페이스북에서 개발한 시계열 예측 패키지
- ARIMA와 같이 확률론적이고 이론적인 모형이 아닌 몇가지 경험적 규칙 (heuristic rule)을 사용하는 단순 회귀 모형이지만, 단기적 예측에서는 큰 문제 없이 사용할 수 있음
- 회귀분석 모형 만드는 순서: 시간 데이터의 각종 특징을 임베딩> 계절성 추정 > 나머지 데이터는 구간별 선형회귀분석(piecewise linear regression)

4) Facebook Prophet

$$y(t)=g(t)+s(t)+h(t)+error$$

- g(t) : Growth, 'linear'와 'logistic'으로 구분되어 있다.

- s(t) : Seasonality

- h(t) : Holidays, 계절성을 가지진 않지만 전체 추이에 영향을 주는 이벤트를 의미하며 이벤트의 효과는 독립적이라 가정

- 결과가 그닥 좋지 않았음.

5) Time-Series Model 최종 Score

* Simple Moving Average(3 rolling) : 831,158.397180
* Exponential Moving Average(5 span) : 767,498.551420
* Simple Exponential Smoothing : 818,205.822450
* Holt's Exponential Smoothing : 926,470.756080
* Holt-Winter's Exponential Smoothing : 962,259.599880
* ARMA model : 984,368.752690
* ARIMA model : 1,080,182.482790
* Facebook prophet : 1,221,173.032530

1) Models

- Linear Regression, Ridge, Lasso, ElasticNet,
- Gradient Boosting Regression, Support Vector Regression,
- XGB Regressor

```
%%time
reg_models = [LinearRegression(), Ridge(), Lasso(), ElasticNet(), GradientBoostingRegressor(), SVR(), XGBRegressor()]
reg_model_names = ["LinearRegression", "Ridge", "Lasso", "ElasticNet", "GradientBoositng", " SupportVector", "XGBoost"]
for i in range(len(reg_models)):
    full_cols = -cross_val_score(reg_models[i], month_reg_x, month_reg_y, scoring="neg_mean_absolute_error", cv=cv)
    certain_cols = -cross_val_score(reg_models[i], month_reg_x[['store_id', 'year', 'month']], month_reg_y, scoring="neg_mean_absolute_error", cv=cv)

    print("{} : {} / {}".format(reg_model_names[i], full_cols.mean(), certain_cols.mean()))
```

executed in 0ms, finished 09:48:49 2021-12-30

2) Results

- 적은 data로 인해 점수가 굉장히 좋지 않게 나왔다.

- 예측에 사용할 수 있는 변수들의 수가 적거나 정확한 값을 채울 수 없어서였던걸로 추정

```
%%time
reg_models = [LinearRegression(), Ridge(), Lasso(), ElasticNet(), GradientBoostingRegressor(), SVR(), XGBRegressor()]
reg_model_names = ["LinearRegression", "Ridge", "Lasso", "ElasticNet", "GradientBoositng", " SupportVector", "XGBoost"]
for i in range(len(reg_models)):
    full_cols = -cross_val_score(reg_models[i], month_reg_x, month_reg_y, scoring="neg_mean_absolute_error", cv=cv)
    certain_cols = -cross_val_score(reg_models[i], month_reg_x[['store_id', 'year', 'month']], month_reg_y, scoring="neg_mean_absolute_error", cv=cv)

    print("{} : {} / {}".format(reg_model_names[i], full_cols.mean(), certain_cols.mean()))
```

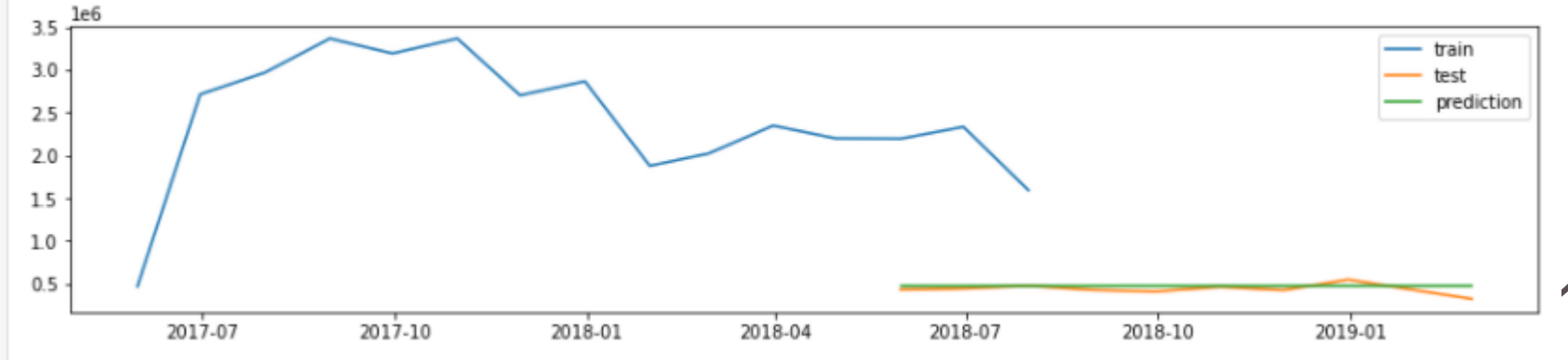
executed in 0ms, finished 09:48:49 2021-12-30

Issues & Results

- Test 값 부분이 거의 직선(학습 data가 너무 적어 예측을 제대로 못함)
- 1967개의 상점을 학습했지만 월 단위로 resampling 하고 나니 데이터가 33개월치 정도로 줄었고 test_size=0.3을 적용시키니 대부분의 값이 0으로 나옴

  > 딥러닝을 하기에 데이터가 부족

```
100%|██████████| 1967/1967 [5:30:40<00:00, 10.09s/it]
[[0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]]
```



Modeling-LSTM

Issues & Results

```
Value of store_id 0 prediction : [0. 0. 0.]
Value of store_id 1 prediction : [0. 0. 0.]
Value of store_id 2 prediction : [0. 0. 0.]
Value of store_id 4 prediction : [0.22865641 0.22865641 0.22865641]
Value of store_id 5 prediction : [0. 0. 0.]
Value of store_id 6 prediction : [1.6137322 1.6137322 1.6137322]
Value of store_id 7 prediction : [2.5254931 2.5254931 2.5254931]
Value of store_id 8 prediction : [3.88921 3.88921 3.88921]
Value of store_id 9 prediction : [3.628832 3.628832 3.628832]
Value of store_id 10 prediction : [0. 0. 0.]
Value of store_id 11 prediction : [10.266693 10.266693 10.266693]
Value of store_id 12 prediction : [0.734782 0.734782 0.734782]
Value of store_id 13 prediction : [0. 0. 0.]
Value of store_id 14 prediction : [0. 0. 0.]
Value of store_id 15 prediction : [3.4008975 3.4008975 3.4008975]
Value of store_id 16 prediction : [0. 0. 0.]
Value of store_id 17 prediction : [0. 0. 0.]
Value of store_id 18 prediction : [8.488293 8.488293 8.488293]
Value of store_id 19 prediction : [0.23646662 0.23646662 0.23646662]
Value of store_id 20 prediction : [0.44612283 0.44612283 0.44612283]
CPU times: user 3min 13s, sys: 6.28 s, total: 3min 20s
Wall time: 3min 7s
```

08 Conclusion

1

상대적으로 단순한 모형들의 점수가 높았고,

그 중 EMA의 결과가 가장 좋았다.

Check

60%

2

뚜렷한 trend나 seasonality가 없고

data의 수가 적어서 파악하기 힘든 관계로

신경망 학습 자체는 가능하나 올바른 예측이 불가능했다.

40%


Q&A

감사합니다