모델링 라이브러리 소개

Contents


```
import FinanceDataReader as fdr
%matplotlib inline
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import warnings
warnings.filterwarnings('ignore')
pd.options.display.float_format = '{:,.3f}'.format
```

모델은 가설을 활용하여 타겟변수를 예측하는 알고리즘을 만드는 것이라고 생각하시면 될 것 같습니다. 파이썬에서는 모델링을 위하여 여러개의 라이브러리(패키지)를 제공하고 있습니다. 대표적인 모델개발 라이브러리는 Statsmodels, Scikit-Learn, Keras 등이 있습니다. 책에서 종목 추천으로 사용할 모델은 일반화 가법 모형(Generalized Additive Model) 입니다. 일반화가법모형은 Statsmodels 에서도 구현할 수 있으나, pyGAM 패키지를 사용하면 더 편리합니다.

Statsmodels 는 전통적인 통계모델에 특화되어 있고, Scikit-Learn 는 머신러닝모델, Keras 는 딥러닝 모델을 개발할 때 활용할 수 있습니다. 라이브러리는 모델을 만드는 패키지가 들어가 있으므로 호출해서 사용하면 됩니다. Statsmodels 은 주로 일반화 선형모형을 구현할 때 주로 사용합니다. 통계 선형모형의 장점은 해석이 가능한 모델을 만들 수 있다는 것입니다. 예를 들면 변수 X 가 1 단위 증가하면 타겟변수는 얼마나 증가하느냐? 등의 해석을 할 수 있습니다..

Scikit-Learn 은 주로 머신러닝 모델을 만들 때 활용하는 라이브러리입니다. 머신러닝 모델은 각 피쳐의 해석보다는 예측력을 최우선으로 합니다. 특히 트리(Tree) 기반 모델은 변수간의 상호작용을 고려하므로 입력 변수사이에 상호작용이 많을 때 효과가 좋습니다. 머신러닝 모델 중에는 앙상블 모델이 인기인데요. 앙상블도 Bias 를 줄이는데 집중하는 Boosting 모델(예 Ada Boost) 계열과 Variance 를 줄이는데 집중하는 Bagging 모델(예 Random Forest) 계열로 나눌 수 있습니다. Bias 랑 Variance 는 하나를 내리면 하나는 올라가는 특징이 있습니다. 두 명의 양궁선수가 있습니다. 한 명은 과녁 중앙에 골고루 퍼지게 활을 쏘는 능력이 있고, 한 명은 일단 처음 쏜 화살에 근처에 집중에서 쏘는 능력이 있다고 하면 누구를 선택하시겠습니까? 첫 번째 양궁선수는 과녁근방에 골고루 쏘는 분이므로 큰 점수는 못 얻어도 항상 기본점수 이상은 획득하는 안전함이 있습니다. 즉 Variance 가 낮음에 해당합니다. 두 번째 양궁선수는 일단 첫 화살에 중앙에 명중하면, 나머지도 10점을 얻을 수 있습니다. 하지만, 처음 화살이 빗나가면 나머지도 다 빗나갑니다. 따라서 첫 화살이 중요합니다. Bias 가 낮기 때문에 overfitting (과대적합) 을 주의해야 합니다. 운이 안 좋아 중앙에서 먼거리에 첫 화살이 명중했다면, 나머지 화살도 그 근방으로 가므로, 우리가 원하는 해답이 아닌 곳으로 모델학습이 이루어지게 되는 것입니다.

Keras 는 딥러닝을 위한 라이브러리입니다. 데이터 수가 많지 않고, 피쳐의 디멘젼이 5 개(시종고저, 거래량) 라면 데이터 복잡성도 높지 않습니다. 구현하고자 하는 예측모델은 딥러닝이 적절하지는 않아보입니다. 굳이 일봉 데이터가 요약된 피쳐로 뉴럴네트워크 모델을 구현한다면 Multi-Layer Perceptron (다층 퍼셉트론) 모델을 생각해 볼 수 있습니다. MLP 는 비선형관계를 표현하기 위해서 Activation Function (활성화함수) 를 이용하고, Activation 함수에서 나온 값을 다시 다음 층의 입력변수로 넣는 형태입니다. 이렇게 함으로써 변수간의 상호작용과 비선형관계를 동시에 표현할 수 있습니다. 사실, 활성화 함수가 Sigmoid 함수인 뉴럴네트워크 모델은 Logistic Regression.모델을 가로 세로층으로 중첩한 것과 동일한 구조가 됩니다. 즉, Logistic Regression 의 확장형으로도 생각할 수 있습니다. 뉴럴네트워크 계통의 모델은 Loss Function (손실함수) 를

만들고 Loss Function 를 최소화하는 네트워크의 가중치를 찾도록 훈련합니다. 많이 쓰는 훈련방식은 오류 역전파(BackPropagation) 입니다. 이런 식의 접근 법은 과대적합이 항상 문제가 됩니다. 따라서 과대적합을 피하기 위해 다양한 기법이 개발 되고 있습니다.

이번절에서는 Statsmodel 과 Scikit-Learn 라이브러리가 모델 개발에 어떻게 활용되는지 경험해 보는 시간입니다.

모델링을 위해 준비한 데이터를 읽습니다. 그리고 모델의 오버피팅을 최소화하기 위하여 타겟변수를 0 과 1 로 치환합니다. 예를 들어, 5% 익절의 데이터 표현은 - 'max_close' 가 5% 이상일 때 1, 아니면 0 이 됩니다. 'max_close' 가 1 인 비율을 보니, 약 24% 입니다. 10000 개 샘플을 뽑아 예측모델을 만들고 나머지로 데이터로 테스트(혹은 백테스팅)를 하겠습니다.

```
feature_all = pd.read_pickle('feature_all.pkl')
feature_all['target'] = np.where(feature_all['max_close']>= 1.05, 1, 0)
target = feature_all['target'].mean()
print(f'% of target:{target: 5.1%}')

mdl_all = feature_all.set_index([feature_all.index,'code'])

train = mdl_all.sample(10000, random_state=124)
test = mdl_all.loc[~mdl_all.index.isin(train.index)]
```

```
% of target: 24.3%
```

아래 코드는 Statsmodels 라이브러리에 대한 이해가 목적입니다. Statsmodels 는 전통적인 통계모델을 구현하는데 주로 활용하는데요. 통계모델의 장점은 변수의 해석이 가능하다는 것입니다. 아래 코드는 랜덤해게 뽑은 5천개의 샘플로 모델을 만들고, 나머지 데이터로 모델 성능을 테스트하는 과정입니다. 모델 개발은 여기서부터 시작입니다. 결과를 보면 P Value(P>|z|) 가 0.01(유의수준) 보다 큰 변수가 많은데요. P Value(P>|z|) 가 유의수준보다 크다는 이야기는 coefficient 가 0 일 가능성이 높다는 말이고, Coefficient 가 0 이라는 말은 예측에 도움을 안 준다는 말입니다. 이런 변수들은 적절한 변형을 통하여 유의미하게 만들거나 제거해야 합니다. 가장 대표적인 방법이 Binning 입니다. 이 절은 라이브러리를 소개하는 것이 목적이라, 모델 완성을 위한 나머지 과정은 생략하도록 하겠습니다. 제가 통계모델의 장점으로 해석을 언급했는데요. 아직 모델이 완성되지 않았지만, 변수 'volume_z' 를 해석해 보도록 하겠습니다. 'volume_z' 는 과거 20일대비 당일 거래량이 얼마나 많은 지를 의미하는 변수입니다. 'volume_z' 가 1 증가하면 log(odds) 는 그 변수의 계수 0.1765 만큼 증가하게 됩니다. 즉, odds 는 exp(0.1765) 증가하게 됩니다. 풀어서 이야기하면, 전일 20일 대비 당일 거래량 표준화 값 z 가 1 증가할 때마다, 5% 로 익절할 odds(=p/1-p)는 exp(0.1765) 증가한다고 말할 수 있습니다.

모델을 완성하기까지 필요한 나머지 절차는 아래와 같습니다.

1. 각 설명변수와 타겟변수와 관계를 분석합니다 (변수간에 상호작용 강한 지 체크)
2. 선형적인 관계가 없는 변수는 binning 등을 통해 문제를 해결합니다. 혹은 제곱근, 제곱, 로그 등의 변형으로 선형적으로 만들 수 도 있습니다.
3. 다중 공선성이 의심되는 변수는 제거하거나 새로운 변수로 대체합니다. (다중 공선성이 높은 모델은 변수의 해석이 부정확함)
4. 테스트 데이터셋과 예측성능을 비교합니다 (오버피팅 여부 확인).
5. 변수를 해석하고 예측값을 만듭니다.

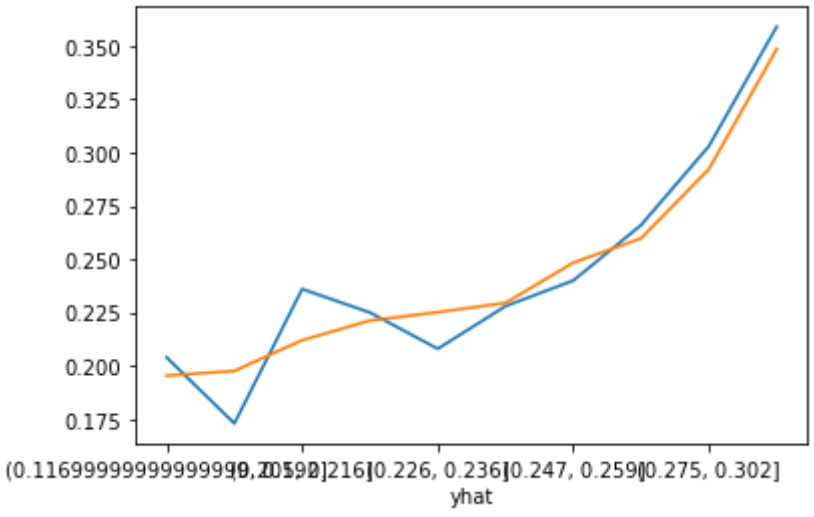
```
import statsmodels.api as sm

feature_list = ['price_z', 'volume_z', 'num_high/close', 'num_long', 'num_z>1.96',
'num_win_market', 'pct_win_market', 'return over sector']

X = train[feature_list]
y = train['target']

X = sm.add_constant(X)
model = sm.Logit(y, X)
results = model.fit()
print(results.summary())
yhat = results.predict(X)
yhat = pd.Series(yhat, name='yhat')

X_test = test[feature_list]
y_test = test['target']
X_test = sm.add_constant(X_test)
yhat_test = results.predict(X_test)
yhat_test = pd.Series(yhat_test, name='yhat')
```

```
Optimization terminated successfully.
         Current function value: 0.549822
         Iterations 6
                           Logit Regression Results
==============================================================================
Dep. Variable:                 target   No. Observations:                10000
Model:                          Logit   Df Residuals:                     9991
Method:                           MLE   Df Model:                            8
Date:                Sun, 26 Jun 2022   Pseudo R-squ.:                 0.01089
Time:                        05:30:08   Log-Likelihood:                -5498.2
converged:                       True   LL-Null:                       -5558.7
Covariance Type:            nonrobust   LLR p-value:                 2.047e-22
==============================================================================
=
                         coef    std err          z      P>|z|      [0.025
0.975]
------------------------------------------------------------------------------
-
const                -24.3868      6.812     -3.580      0.000     -37.739
-11.035
price_z               -0.1266      0.021     -6.156      0.000      -0.167
-0.086
volume_z               0.1491      0.024      6.255      0.000       0.102
0.196
num_high/close         0.1396      0.058      2.411      0.016       0.026
0.253
num_long              -0.1293      0.172     -0.752      0.452      -0.466
0.208
num_z>1.96             0.0257      0.013      1.982      0.047       0.000
0.051
num_win_market         0.0251      0.009      2.722      0.006       0.007
0.043
pct_win_market        23.8495      6.797      3.509      0.000      10.527
37.172
return over sector    -0.8839      0.835     -1.058      0.290      -2.521
0.753
==============================================================================
=
```

개발 데이터가 아니라, 테스트데이터에서도 좋은 성능을 보이는지 확인해봅니다. 쉽게 확인하는 방법은 Decile 분석입니다. 예측값의 변별력을 알기 위해서 정렬된 예측값을 10 개 구간으로 나누고, 각 구간에서 'target'의 평균값을 찍어봅니다. 파란색이 개발데이터, 주황색이 테스트 데이터입니다. 모델이 예측력이 좋다면, 예측값의 십분위 수가 증가하면 5%로 익절할 확률도 같이 증가하는 형태를 보이게 됩니다. 아래 결과에서 완성되지 않은 모델이지만 단조증가하는 좋은 흐름을 보여주고 있습니다. 테스트 결과에서 제 1 십분위수(첫 번째 구간) 에서 종목을 선택한다면 19.7%로 익절할 확률이 있지만, 제 10 분위수(마지막 구간)에서 종목을 선택한다면 34.9% 로 익절할 확률이 생깁니다.

```
def perf(y, yhat): # Decile 분석 함수
    combined = pd.concat([y, yhat], axis=1)
    ranks = pd.qcut(combined['yhat'], q=10)
    print(combined.groupby(ranks)['target'].agg(['count','mean']))
    combined.groupby(ranks)['target'].mean().plot()

perf(y, yhat)
perf(y_test, yhat_test)
```

```
                count  mean
yhat
(0.138, 0.192]   1000 0.204
(0.192, 0.206]   1000 0.173
(0.206, 0.218]   1000 0.236
(0.218, 0.227]   1000 0.225
(0.227, 0.237]   1000 0.208
(0.237, 0.248]   1000 0.228
(0.248, 0.259]   1000 0.240
(0.259, 0.275]   1000 0.266
(0.275, 0.302]   1000 0.303
(0.302, 0.637]   1000 0.359
                              count  mean
yhat
(0.11699999999999999, 0.192]  31931 0.195
(0.192, 0.205]                31931 0.198
(0.205, 0.216]                31930 0.212
(0.216, 0.226]                31931 0.221
(0.226, 0.236]                31931 0.225
(0.236, 0.247]                31930 0.230
(0.247, 0.259]                31931 0.248
(0.259, 0.275]                31930 0.260
(0.275, 0.302]                31931 0.292
(0.302, 0.728]                31931 0.349
```



Scikit-Learn 에서도 Logistic Regression 을 지원합니다. 하지만 계수를 추정하는 방식이 Statsmodels 과는 다른데요. Scikit-Learn Logistic Regression 은 loss 함수를 만들고, 과대적합을 해결하기 위해 penalty term (L1/L2) 도 추가합니다. 이런 방식으로 Penalty Term 이 있는 Loss 함수를 최소화하는 방식을 계수를 찾을 때는 입력 피처의 스케일이 동일해야 의미가 있습니다. 아래 코드에서 입력 피쳐를 Scaling 하는 부분이 반드시 들어가야 합니다. 아래 Test 데이터의 결과가 Train 데이터보다 모델성능의 차이가 크지는 않습니다. 즉 overfitting(과대적합)이 심하지는 않아 보입니다.

SK-Learn 으로 만드는 모델은 절차가 아래와 같습니다.

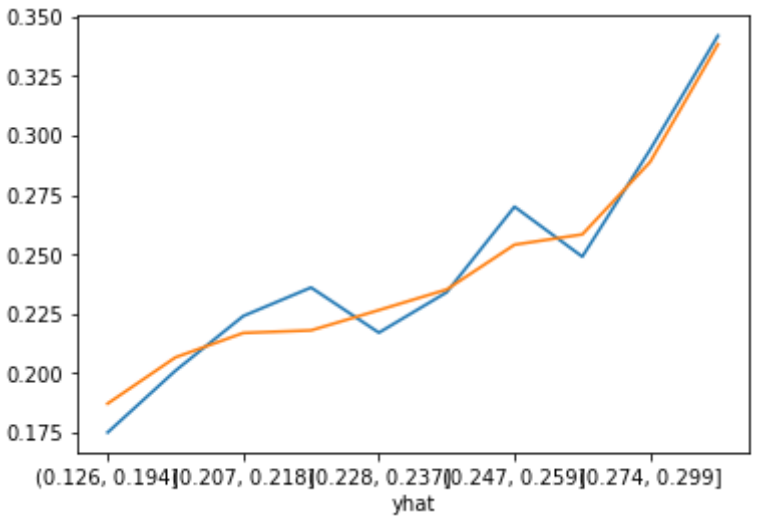
1. 입력 피처 스케일링 (Loss 함수 + Penalty Term 로 훈련하는 모델만 해당)
2. 하이퍼파라미터 튜닝 (성능을 최고로 만드는 하이퍼파라미터을 찾기)
3. 테스트 데이터셋과 예측성능을 비교합니다 (오버피팅 여부 확인)
4. 변수의 중요도 파악과 이해
5. 예측값 만들기

```python
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler

feature_list = ['price_z', 'volume_z', 'num_high/close', 'num_long', 'num_z>1.96',
'num_win_market', 'pct_win_market', 'return over sector']

X = train[feature_list]
y = train['target']

X_test = test[feature_list]
y_test = test['target']

# 입력 피처 표준화
scaler = StandardScaler() # 평균이 0 이고 편차가 1 가 되도록 피처 표준화
scaler.fit_transform(X)
scaler.transform(X_test)

lr = LogisticRegression(fit_intercept=True, C=1) # 정해진 하이퍼파라미터를 가진 객체를
생성, C 값은 다중 공선성을 제거하기 위한 페널티의 가중치이며 디폴트는 L2(Ridge) 페널티
lr.fit(X, y)
print(lr.coef_)
yhat = lr.predict_proba(X)[:,1]
yhat_test = lr.predict_proba(X_test)[:,1]

yhat = pd.Series(yhat, name='yhat', index=y.index)
yhat_test = pd.Series(yhat_test, name='yhat', index=y_test.index)
```

```
[[-0.10997133  0.14023527  0.15290205 -0.03634754  0.04334864  0.03748485
   0.51234537 -0.51150234]]
```

```python
perf(y, yhat)
perf(y_test, yhat_test)
plt.show()
```

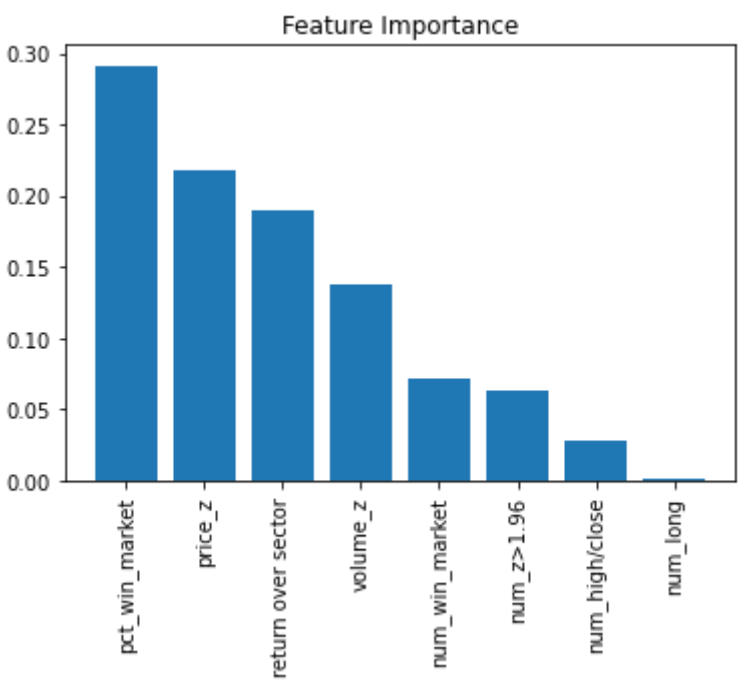
```
               count  mean
yhat
(0.142, 0.194]   1000 0.175
(0.194, 0.208]   1000 0.201
(0.208, 0.219]   1000 0.224
(0.219, 0.228]   1000 0.236
(0.228, 0.238]   1000 0.217
(0.238, 0.248]   1000 0.234
(0.248, 0.26]    1000 0.270
(0.26, 0.275]    1000 0.249
(0.275, 0.3]     1000 0.294
(0.3, 0.632]     1000 0.342
               count  mean
yhat
(0.126, 0.194]  31931 0.187
(0.194, 0.207]  31931 0.207
(0.207, 0.218]  31930 0.217
(0.218, 0.228]  31931 0.218
(0.228, 0.237]  31931 0.226
(0.237, 0.247]  31930 0.235
(0.247, 0.259]  31931 0.254
(0.259, 0.274]  31930 0.258
(0.274, 0.299]  31931 0.289
(0.299, 0.652]  31931 0.338
```



Random Forest 는 Scikit-Learn 에서 인기있는 모델인데요. Decision-Tree(의사결정나무)의 문제점을 보완하고자 나온 개념입니다. 모델을 훈련시키기 위한 loss 함수와 Penalty term 이 없기 때문에 피쳐 스케일링이 불필요해서 쉽게 모델을 만들어 볼 수 있습니다. 보통 모델의 최소성능을 파악하기 위해 먼저 만들어보는 모델입니다. sklearn 의 ensemble(앙상블) 모델군에서

RandomForestClassifier 을 불러옵니다. 그 다음 정해진 하이퍼파라미터(hyperparameter)를 가진 객체를 하나 생성합니다. 여기서 어떤 하이퍼파라미터로 객체를 생성하는가에 따라 모델의 성능이 결정되므로, 하이퍼파라미터 튜닝이라 절차가 필요합니다. 보통 최적의 하이퍼파라미터는 Grid Search 로 찾습니다. 그리고 fit 를 이용해서 데이터를 적용하면 됩니다. 예측값 생성은 predict 함수나 predict_proba 함수로 할 수 있는데요. predict 함수는 0/1 값을 리턴하고, predict_proba 함수는 '0 일 확률'/'1 일 확률'을 리턴합니다. 각 피처의 중요도를 그래프로 파악해 보겠습니다. 이전 모델들에 비해 예측성능이 좋습니다.

```
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier(max_depth=4, min_samples_leaf=30) # 정해진 하이퍼파라미터를 가진 객체를 생성

feature_list = ['price_z', 'volume_z', 'num_high/close', 'num_long', 'num_z>1.96',
'num_win_market', 'pct_win_market', 'return over sector']

X = train[feature_list]
y = train['target']
rf.fit(X, y)
yhat = rf.predict_proba(X)[:,1] # 첫번째 열은 0일 확률, 두번째 열은 1 일 확률 -> 1 일 확률을 저장
yhat = pd.Series(yhat, name='yhat', index=y.index)

X_test = test[feature_list]
y_test = test['target']
yhat_test = rf.predict_proba(X_test)[:,1] # 첫번째 열은 0일 확률, 두번째 열은 1 일 확률 -> 1 일 확률을 저장
yhat_test = pd.Series(yhat_test, name='yhat', index=y_test.index)

importances = rf.feature_importances_
sorted_indices = np.argsort(importances)[::-1]

import matplotlib.pyplot as plt

plt.title('Feature Importance')
plt.bar(range(X.shape[1]), importances[sorted_indices], align='center')
plt.xticks(range(X.shape[1]), X.columns[sorted_indices], rotation=90)
plt.show()
```




```
perf(y, yhat)
perf(y_test, yhat_test)
plt.show()
```

```
               count  mean
yhat
(0.167, 0.203]   1000 0.131
(0.203, 0.212]   1000 0.152
(0.212, 0.218]   1000 0.182
(0.218, 0.226]   1000 0.209
(0.226, 0.233]   1000 0.221
(0.233, 0.242]   1000 0.241
(0.242, 0.255]   1000 0.263
(0.255, 0.274]   1000 0.295
(0.274, 0.304]   1000 0.313
(0.304, 0.469]   1000 0.435
               count  mean
yhat
(0.164, 0.203]  31932 0.155
(0.203, 0.212]  31930 0.180
(0.212, 0.218]  31930 0.187
(0.218, 0.225]  31931 0.204
(0.225, 0.233]  31931 0.225
(0.233, 0.241]  31931 0.238
(0.241, 0.254]  31930 0.268
(0.254, 0.274]  31930 0.287
(0.274, 0.304]  31931 0.308
(0.304, 0.487]  31931 0.378
```