개별 연구

3차 코드 구현

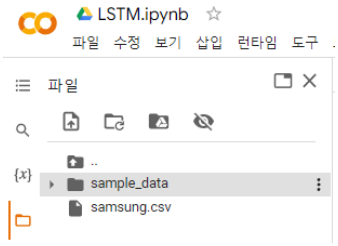
학 과	컴퓨터공학과
학 번	2017112292
이 름	김준하

1. 1차 코드 구현에 사용된 모델과 데이터셋

- 모델: LSTM (XGBoost, RNN 모델을 시도해보았지만 역량 부족으로 개념들과 코드의 실행을 이해하지 못하여 시간 내에 완성하지 못했습니다.)
- 데이터셋: 최근 5년간의 삼성전자 주가 데이터

2. 개발 환경

- Google Colab
- 파일에 주식 데이터 파일(samsung.csv)을 업로드하여 실행



3. 모델링 과정

모델링은 다음과 같은 과정을 통해 진행하였다.

I. 설정

```
seed_num = 42
np.random.seed(seed_num)
random.seed(seed_num)
tf.random.set_seed(seed_num)
```

random seed를 설정하지 않으면 실행마다 결과가 달라지므로 임의로 고정시킨 후 실행한다.

II. 데이터 전처리

- Volume이 0인 값들은 거래량이 0인 데이터로 생각된다. 이는 데이터로서 가치가 없으므로 사용하지 못하는 데이터로 바꾼다.

```
# Volume의 값이 0이면 NaN으로 바꿈

data['Volume'] = data['Volume'].replace(0, np.nan)
```

- na값은 의미 없는 데이터이므로 이에 해당하는 행들을 삭제한다.

```
# na값이 있는 행 삭제

data = data.dropna()

# 각 column의 null 개수 출력
data.isnull().sum()
```

```
Date         0
Open         0
High         0
Low          0
Close        0
Adj Close    0
Volume       0
dtype: int64
```

III. 변수 scaling

- 데이터 간의 범위 차이가 큰 경우 학습에 부정적인 영향을 미칠 수 있고, 실제 운영에서도 학습에서 사용되지 않은 큰 데이터가 들어오면 모델이 강하게 발산할 여지가 존재하므로 정규화를 통하여 모델을 안정적으로 만들어주는 과정이 필요하다.

```
# Date column을 제외한 나머지 column들을 정규화

scaler = MinMaxScaler()

scale_cols = ['Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume']
scaled = scaler.fit_transform(data[scale_cols])
scaled = pd.DataFrame(scaled, columns=scale_cols)

print(scaled)
```

```
          Open      High       Low     Close  Adj Close    Volume
0     0.252602  0.239527  0.264957  0.250047   0.187821  0.098511
1     0.253359  0.232095  0.250142  0.259384   0.195749  0.121753
2     0.248817  0.231419  0.255081  0.257890   0.194481  0.057276
3     0.261306  0.235811  0.250142  0.234734   0.174818  0.075468
4     0.238221  0.226014  0.250142  0.248179   0.186235  0.077521
...        ...       ...       ...       ...        ...       ...
1207  0.453169  0.407095  0.454891  0.439776   0.506466  0.026443
1208  0.436140  0.386824  0.435897  0.422969   0.489871  0.048691
1209  0.424787  0.388514  0.435897  0.432306   0.499091  0.030357
1210  0.434248  0.415541  0.443495  0.462185   0.528594  0.178841
1211  0.485336  0.432432  0.483381  0.469655   0.535970  0.111700

[1212 rows x 6 columns]
```

IV. shifting을 통해 window 생성

- 이전의 데이터들을 사용하여 다음의 데이터 값을 예측하기 위해 사용되는 sliding window를 생성한다.

```
# 특징 데이터와 label 데이터를 입력받아 시계열 데이터를 만든다.
# shifting을 하여 window를 생성한다.

def make_sequence(feature, label, window):

    feature_list = []
    label_list = []

    # window 크기만큼의 데이터들을 특징 리스트의 각 항목에 저장하고 마지막의 다음 값을 label 리스트에 저장한다.
    for i in range(len(feature)-window):

        feature_list.append(feature[i:i+window])
        label_list.append(label[i+window])

    return np.array(feature_list), np.array(label_list)
```

V. 훈련셋과 테스트셋 생성

- scaling이 완료된 window로 이루어진 특징 리스트와 라벨 리스트에서 8:2의 비율로 훈련셋과 테스트셋을 구성한다.

```
# 훈련 데이터와 테스트 데이터를 8:2 비율로 분리한다.

split = int(len(X)*0.8)

x_train = X[0:split]
y_train = Y[0:split]

x_test = X[split:]
y_test = Y[split:]
```

VI. LSTM 모델 생성

- 여러 층의 레이어를 선형으로 연결하여 구성하기 위하여 Sequential 모델을 생성하고, 활성화 함수로 tanh를 사용하는 LSTM셀과 활성화 함수로 선형 함수를 사용하는 Dense층을 하나씩 추가하였다.

```
# 레이어를 선형으로 연결하여 구성하기 위해 Sequential 모델을 생성한다.

model = Sequential()

#활성함수로는 ReLU 대신 tanh를 사용한다.
model.add(LSTM(128, activation='tanh', input_shape=x_train[0].shape))
model.add(Dense(1, activation='linear'))
```

- 모델은 손실함수로 평균제곱오차(mse), 옵티마이저로 adam, 측정항목함수(metrics)로 평균절대오차(mae)를 사용하여 컴파일하였다.

```
#손실함수로는 평균제곱오차를, 옵티마이저로는 adam을 사용한다.
model.compile(loss='mse', optimizer='adam', metrics=['mae'])

model.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_6 (LSTM)               (None, 128)               66560

 dense_6 (Dense)             (None, 1)                 129

=================================================================
Total params: 66,689
Trainable params: 66,689
Non-trainable params: 0
_________________________________________________________________
```

VII. 훈련셋을 사용한 모델 훈련

- Early stopping을 사용하여 validation loss가 5번 연속으로 증가하면 모델 훈련을 멈추도록 하였다.

  총 100번을 반복하여 훈련하고 훈련이 진행되는 batch의 크기는 16으로 설정하였다.

```
#특정 조건에 도달하면 종료하기 위해 EarlyStopping을 사용한다.
early_stop = EarlyStopping(monitor='val_loss', patience=5)

model.fit(x_train, y_train,
          validation_data=(x_test, y_test),
          epochs=100, batch_size=16,
          callbacks=[early_stop])
```

```
Epoch 1/100
59/59 [==============================] - 4s 35ms/step - loss: 0.0161 - mae: 0.0717 - val_loss: 0.0013 - val_mae: 0.0286
```

VIII. 테스트셋을 사용한 테스팅

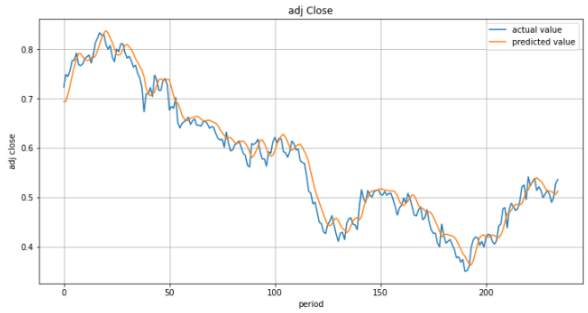
- 모델에 테스트셋을 적용하여 예측을 적용한 후 실제값과 함께 출력한 모습이다.

```
pred = model.predict(x_test)

plt.figure(figsize=(12, 6))
plt.title('adj Close')
plt.ylabel('adj close')
plt.xlabel('period')
plt.plot(y_test, label='actual value')
plt.plot(pred, label='predicted value')
plt.grid()
plt.legend(loc='best')

plt.show()
```

```
8/8 [==============================] - 0s 17ms/step
```



- 예측 결과값을 사용하여 평균 절대 백분율 오차(MAPE)를 구한 결과이다.

```
# 평균 절대 백분율 오차를 계산한다.

print( np.sum(abs(y_test-pred)/y_test) / len(x_test) )
```

```
0.025162243278213 75
```

4. 분석

I. LSTM 모델에서는 결과값을 예측하기 위해 여러 가지의 특징 데이터들이 사용될 수 있지만, 주식 데이터에는 날짜와 주식값만이 존재하므로 특징 데이터에는 주식값만을 사용하였다.

II. 위 모델에서는 활성화 함수로 tanh 함수를 사용하였다.
RNN, LSTM은 CNN과 달리 이전 step의 값을 사용하므로 활성화 함수로 ReLU를 사용하면 이전 값이 커짐에 따라 전체적인 출력이 발산하는 문제가 생길 수 있으므로 tanh를 사용한다.

III. 위에서 구한 평균 절대 백분율 오차의 결과는 다음과 같다.

```
# 평균 절대 백분율 오차를 계산한다.

print( np.sum(abs(y_test-pred)/y_test) / len(x_test) )
```

```
0.025162243278213 75
```

MAPE는 각 시간 단위에 대해 관측된 값과 예측 값 사이의 백분율 오차 절대값을 취한 후 해당 값의 평균을 구하는 방식이다. 이는 시점과 이상값 간에 값이 크게 다른 경우 유용하다.
구한 값이 작을수록 모형이 정확하다는 것을 의미한다.
결과는 약 3%의 계산값을 얻게 되었는데 이는 작은 값이므로 비교적 정확한 모델이라고 볼 수 있다. 이는 예측 값이 평균적으로 약 3% 벗어난다는 의미로도 볼 수 있다.

5. 모델을 변경하면서 사용한 결과의 비교

I. LSTM셀을 한 층 추가하여 실행한 결과

```
# 평균 절대 백분율 오차를 계산한다.
print( np.sum(abs(y_test-pred)/y_test) / len(x_test) )
```

0.02516224327821375

```
# 평균 절대 백분율 오차를 계산한다.
print( np.sum(abs(y_test-pred)/y_test) / len(x_test) )
```

0.025624423965300226

왼쪽은 기존대로 하나의 LSTM셀 레이어만 사용한 결과이고, 오른쪽은 하나의 레이어를 추가로 사용한 결과이다. MAPE를 계산한 결과를 보면 두 결과의 차이가 거의 나지 않음을 알 수 있다. 하나의 레이어를 추가하는 것은 큰 차이를 보이지 않았다.

II. LSTM의 활성화 함수를 ReLU를 사용한 결과

```
# 평균 절대 백분율 오차를 계산한다.
print( np.sum(abs(y_test-pred)/y_test) / len(x_test) )
```

0.02516224327821375

```
# 평균 절대 백분율 오차를 계산한다.
print( np.sum(abs(y_test-pred)/y_test) / len(x_test) )
```

0.029576010967483628

왼쪽은 기존대로 LSTM의 활성화 함수로 tanh를 사용한 결과이고, 오른쪽은 활성화 함수로 ReLU를 사용한 결과이다. 오른쪽은 결과는 MAPE가 기존의 결과보다 높으므로 비교적 덜 정확한 모델이라고 볼 수 있다.

III. window의 크기를 다르게 사용한 결과

```
# 평균 절대 백분율 오차를 계산한다.
print( np.sum(abs(y_test-pred)/y_test) / len(x_test) )
```

0.02516224327821375

위 결과는 기존대로 window의 크기를 40으로 사용한 결과이다.

```
# 평균 절대 백분율 오차를 계산한다.
print( np.sum(abs(y_test-pred)/y_test) / len(x_test) )
```

0.06750810836273029

```
# 평균 절대 백분율 오차를 계산한다.
print( np.sum(abs(y_test-pred)/y_test) / len(x_test) )
```

0.048170421562024286

```
# 평균 절대 백분율 오차를 계산한다.
print( np.sum(abs(y_test-pred)/y_test) / len(x_test) )
```

0.04569349168550582

```
# 평균 절대 백분율 오차를 계산한다.
print( np.sum(abs(y_test-pred)/y_test) / len(x_test) )
```

0.024368936960163647

```
# 평균 절대 백분율 오차를 계산한다.
print( np.sum(abs(y_test-pred)/y_test) / len(x_test) )
```

0.030254315779466558

위 결과는 왼쪽 위에서부터 순서대로 window의 크기가 5, 10, 20, 50, 60일 때의 결과이다. window의 크기가 너무 작아도 성능이 좋지 못하고 너무 커도 좋지 못하다는 것을 알 수 있다. 위 결과들 중에서는 window의 크기가 50인 경우에 가장 적절하여 좋은 성능을 가진다는 것을 알 수 있다.